ECE 385

Spring 2023

Experiment 3

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

Kevin Huang (kuanwei2), Steven Chang (sychang5)
TY 10:15 A.M.
Tianhao Yu

Introduction

This lab intends to be an introduction to the RTL design on FPGA board using SystemVerilog. This lab aims to get familiar with the operation of Quartus Prime and to construct three types of adders: carry-ripple adder, carry-select adder, and carry-lookahead adder. These three types of adders are used to calculate the addition result of 16-bit inputs A and B and a carry-in input cin. Then the sum will be stored in output S while the carry-out is stored in output logic cout. Throughout the process, the simulation and synthesis tools will be utilized to observe the performance of each type of adder.

Operation of the 16-Bit Adders

Full-Adder: The full adder is used as part of the design for each 16-bit adder. To compute the sum and carry-out of the inputs x, y, and z, the full adder utilizes a 3-input XOR gate to produce the sum of the three inputs and uses AND and OR gates to produce the carry-out with the expression (x&y) | (y&z) | (x&z) as shown in Figure 1.

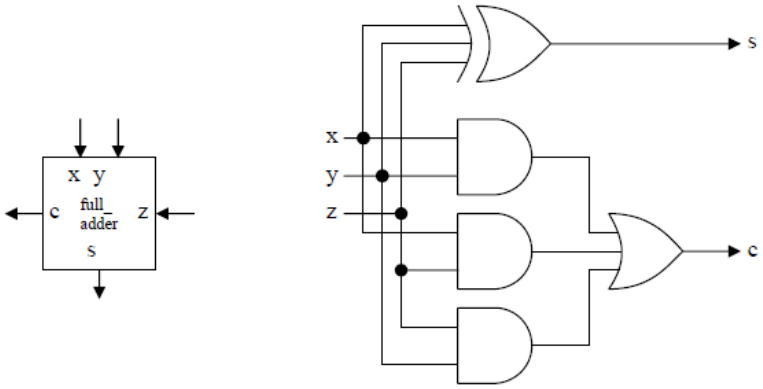

Figure 1. Full-Adder Block Diagram

Carry-Ripple Adder: The carry-ripple adder is constructed with a series of 16 full adders, in which each bit of the inputs A and B correspond to the inputs to one of the 16 full adders. The carry-in input will then be connected to the cin of the full adder of the rightmost bit. Then, with A, B, and cin, the full adder will produce a bit of the output S and a carry-out bit. Afterward, the carry-out will be connected to the carry-in of the next full adder, and so on. Thus, the carry-ripple adder will produce a 16-bit S with a carry-out as shown in Figure 2 below.

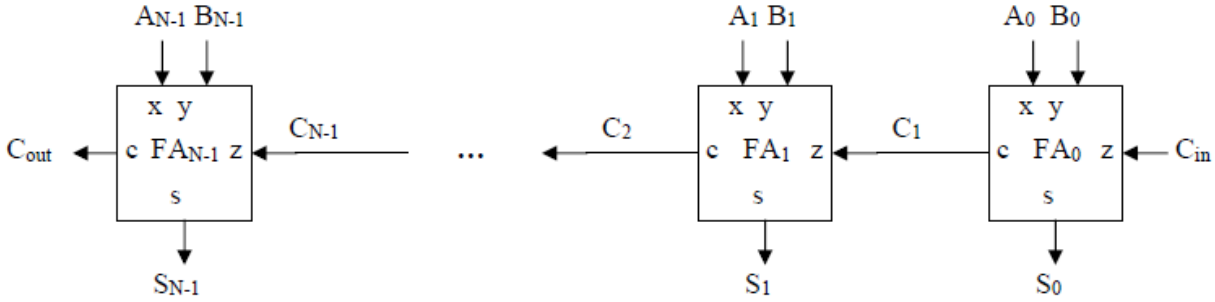

Figure 2. N-bit Carry-Ripple Adder Block Diagram

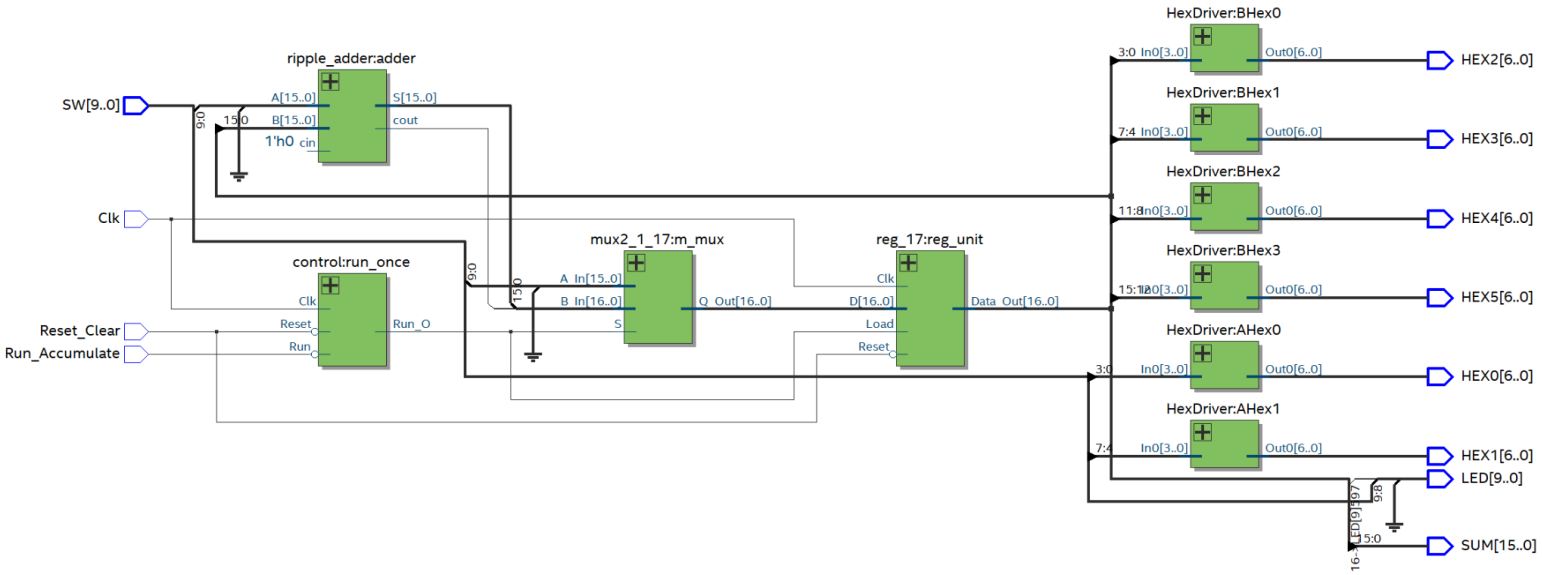

Figure 3. 16-bit Carry-Ripple Adder High-Level RTL Diagram

Carry-Select Adder: Different from the carry-ripple adder, the carry-select adder computes both addition results of A and B with a carry-in value of 1 and 0 using 16 full adders for each result. Afterward, the two different results from each bit and the final carry-out value will be connected to a 2-to-1 MUX with a select input connecting to the actual carry-in value to determine the result as shown in Figure 3. In this way, the carry-select adder can neglect the waiting time for the carry-in bit for each full adder. Instead, it only needs to wait for one clock cycle for each bit to determine whether to choose the result with cin of 0 or 1.

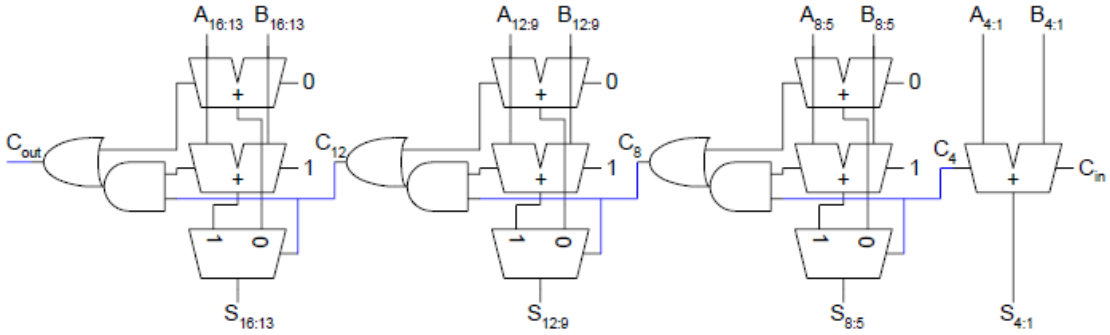

Figure 4. 16-Bit Carry-Select Adder Block Diagram

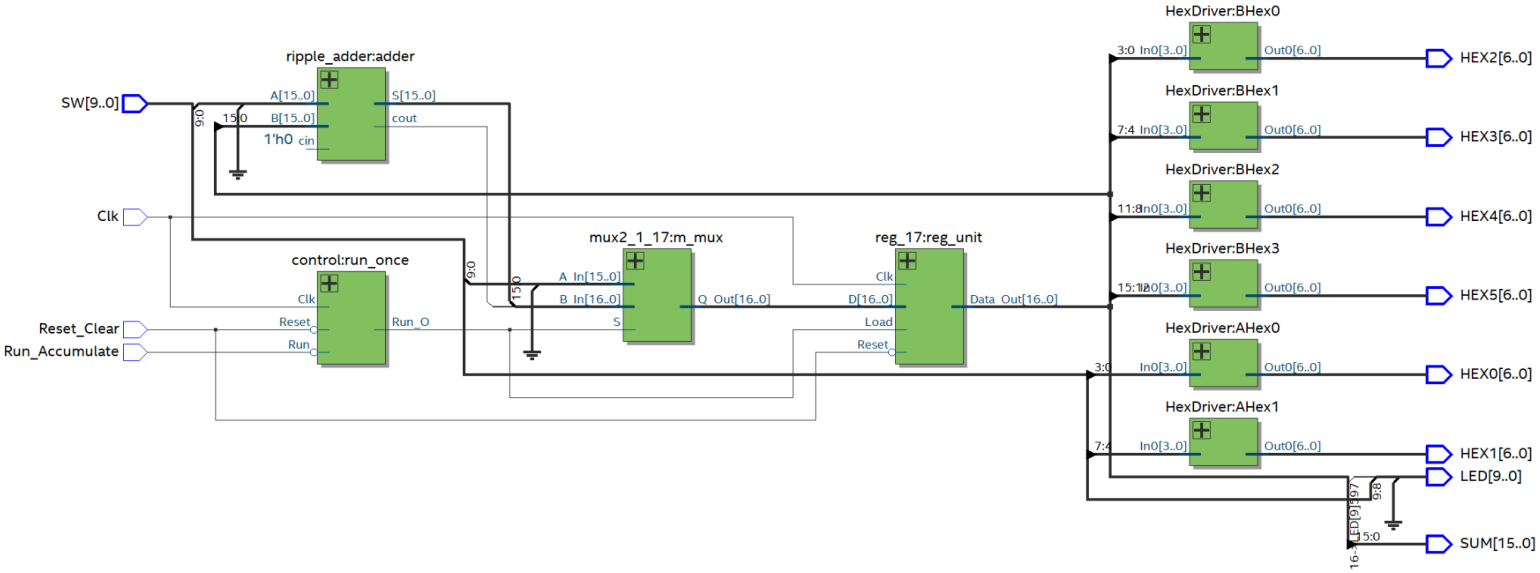

Figure 5. 16-bit Carry-Select Adder High-Level RTL Diagram

Carry-Lookahead Adder: Last but not least, the carry-lookahead adder is implemented with 4x4-bit design which computes propagated signal (P) with an expression of A^B and Generated signal (G) with an expression of AB for each bit. In each 4-bit design, there will be 4 full adders with each bit of A and B as the inputs shown in Figure 4. Then, the full adder will produce the sum (S) of the inputs like normal. For each carry-in input, the value will be determined by the expression

$C0 = Cin,\ C1 = Cin \cdot P0 + G0,\ C2 = Cin \cdot P0 \cdot P1 + G0 \cdot P1 + G1,\ \ so\ on...$

Then, with all propagated and generated signals in the 4-bit design, the group propagate signal (PG) and group generate signal (GG) can be computed with the logics

$PG = P0 \cdot P1 \cdot P2 \cdot P3,\ GG = G3 + G2 \cdot P3 + G1 \cdot P3 \cdot P2 + G0 \cdot P3 \cdot P2 \cdot P1$

Thus, the carry-in bit of the next 4-bit design can be generated with the expression

$C4 = GG0 + C0 \cdot PG0,\ C8 = GG4 + GG0 \cdot PG4 + C0 \cdot PG0 \cdot PG4$ and so on with C16 the carry-out bit of the entire adder shown in Figure 5.

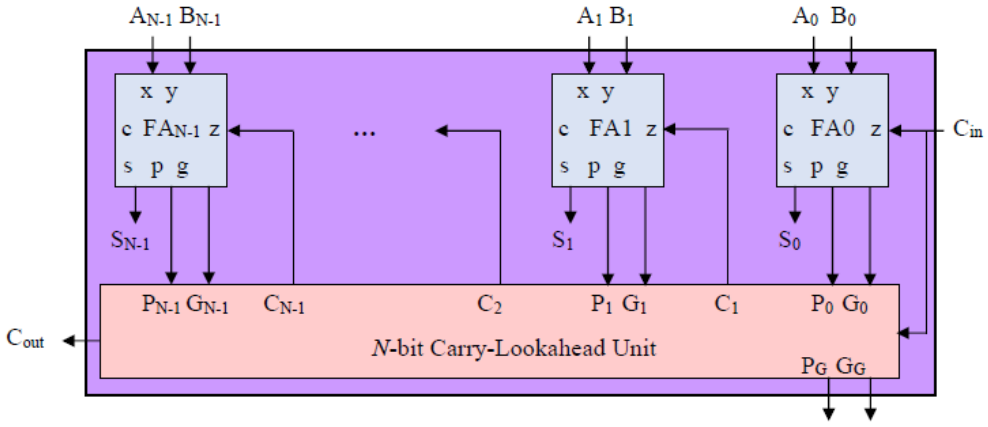

Figure 6. N-Bit Carry-Lookahead Adder Block Diagram

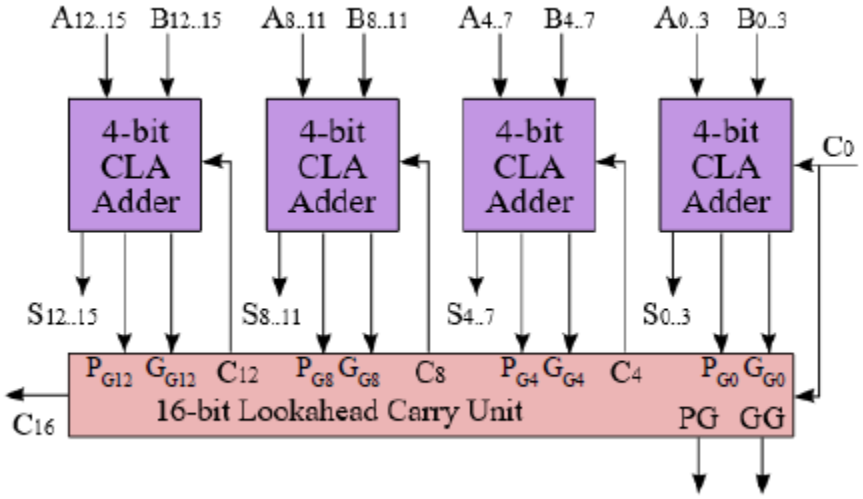

Figure 7. 4x4-Bit Hierarchical Carry-Lookahead Adder Block Diagram

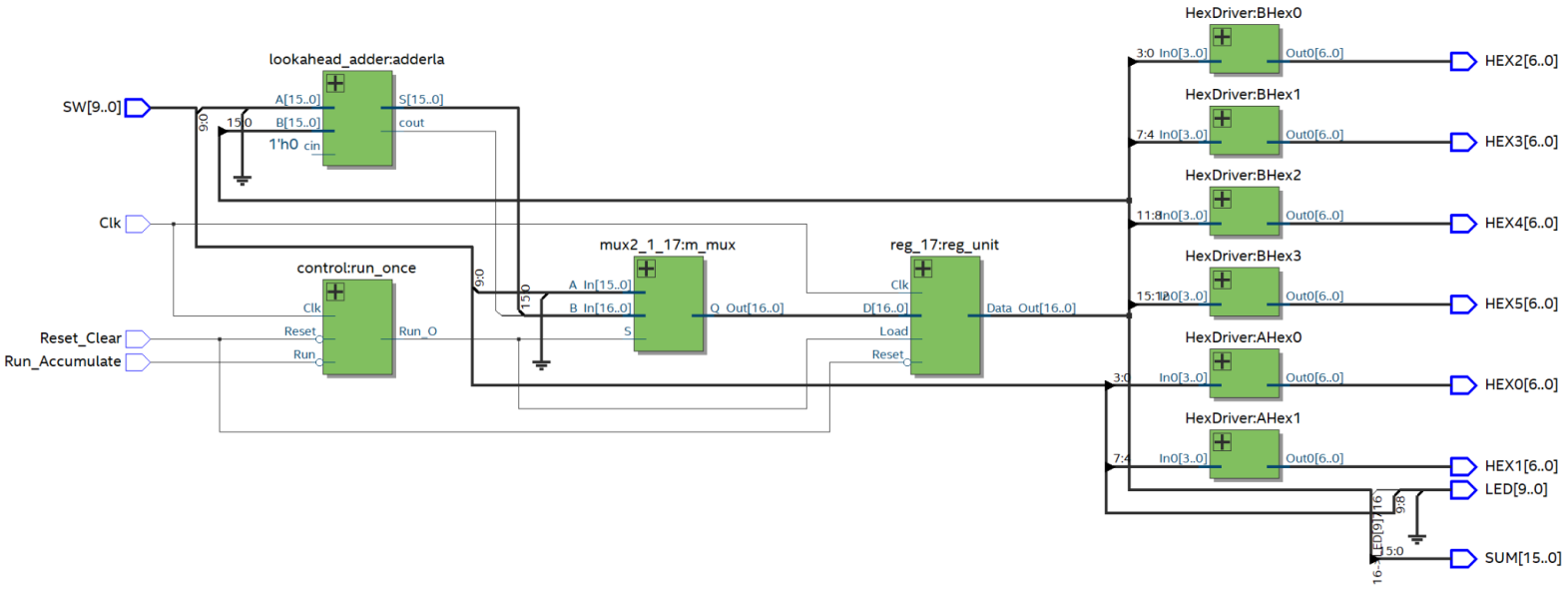

Figure 8. 16-bit Carry-Lookahead Adder High-Level RTL Diagram

SV Modules Summary

Module: reg17.sv
Inputs: [16:0] D, Clk, Load, Reset
Outputs: [16:0] Data_out
Description: This is a positive-edge triggered 17-bit register with synchronous reset and synchronous load. When the Load is high, data is loaded from Din into the register on the positive edge of Clk. Otherwise, if the Reset is high, the register is set with a value of 0.
Purpose: This module is used to create the registers that store operands A and B.

Module: control.sv
Inputs: Clk, Reset, Run
Outputs: Run_O
Description: This is the positive-edge triggered flip-flop that has 3 states, which include a start state, a loading state, and a halt state. When the Reset signal is high, the flip-flop will return to the start state in which the output signal is low. Otherwise, if the Run signal is high during the start state, the flip-flop will then proceed to the loading state which has an output signal of high and then automatically proceeds to the halt state. In the halt state, the output signal is low, and it will only return to the start state when the Run signal is returned back to low.
Purpose: This module is used as a simple state machine to make sure the Run switch input will only run the adders once until the Run switch is back to low.

Module: full_adder.sv
Inputs: A, B, cin
Outputs: s, cout
Description: This is a full adder module that takes in the inputs A, B, and cin and then produce the output s, which is A^B^cin, and output cout, which is (A&B)|(A&cin)|(B&cin).
Purpose: This module is used as the routing unit to determine which input values have to be stored in the shift in bits(A_In and B_In) of A and B.
Module: adder4.sv
Inputs: [3:0] A, [3:0] B, cin
Outputs: [3:0] s, cout
Description: This is a 4-bit full adder module that takes in 3-bit inputs A, B, and a 1-bit input cin and has a series of 4 full adders with cin connecting to the carry-in bit of the first full adder and the carry-out bit of the first full adder connecting to the carry-in of the second full adder and so on to produce a 3-bit output s and a 1-bit output cout.
Purpose: This module is used to combine 4 full adders to create a 4-bit full adder.

Module: mux2_1_1.sv
Inputs: couttop, coutbot, select
Outputs: cout
Description: This is the 2-to-1 MUX that takes in couttop and coutbot, which are the carry-out bit calculated when assuming the carry-in is either one or zero. Then, the select input will choose which of them will be the carry-out bit.

Purpose: This module is used in the carry-select adder to select the final carry-out bit depending on the actual carry-in bit, which is used as the select input.

Module: mux2_1_4.sv
Inputs: [3:0] top, [3:0] bot, cin
Outputs: [3:0] s
Description: This is the 2-to-1 MUX that takes in 4-bit inputs top and bot, which are the sum calculated when assuming the carry-in is either one or zero. Then, the select input cin will choose which of them will be the 4-bit output s.
Purpose: This module is used in the carry-select adder to select the 4-bit sum depending on the actual carry-in bit, which is used as the select input.

Module: mux2_1_17.sv
Inputs: [15:0] A_In, [16:0] B_In, S
Outputs: [16:0] Q_Out
Description: This is the 2-to-1 MUX that takes in 16-bit input A and 17-bit input B. Then, the select input S will choose which of them will be the 17-bit output Q_Out. If S is 0, A will be extended to 17-bit with the most significant bit of 0 and be set as Q_Out. Otherwise, Q_Out will be B.
Purpose: This module is used to determine whether to store the sum of A and B or B in the register based on the control unit signal.

Module: ripple_adder.sv
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] S, cout
Description: This is a ripple adder module that takes in 16-bit inputs A and B and a carry-in input cin and computes the addition result using a series of 16 full adders by connecting cin to the carry-in input of the first full adder and the carry-out bit of the first full adder to the carry-in bit of the second full adder and so on. The sum of each full adder will produce the 16-bit output S while the carry-out of the last full adder will be the output cout.
Purpose: This module is designed as a carry-ripple adder that computes the addition result of A, B, and Cin using a series of full adders.

Module: select_adder4.sv
Inputs: [3:0] A, [3:0] B, cin
Outputs: [3:0] S, cout
Description: This is a 4-bit carry-select adder that takes in 4-bit inputs A and B and produces the addition result using two 4-bit full adders with one of them having a carry-in of 0 and another having a carry-in bit of 1. Then, the input cin will be used as the select input for 2-to-1 MUX to choose whether to output the sum and carry-out that has carry-in as 0 or 1.
Purpose: This module is designed as a 4-bit carry-select adder.

Module: select_adder.sv
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] S, cout

Description: This is a 16-bit carry-select adder that takes in 16-bit inputs A and B and produces the addition result by connecting four 4-bit carry-select adders in series with the carry-out bit of the first carry-select adder connecting to the carry-in bit of the second carry-select adder and so on.
Purpose: This module is designed as a 16-bit carry-select adder using four 4-bit carry-select adders.

Module: addercla4.sv
Inputs: [3:0] A, [3:0] B, cin
Outputs: [3:0] s, pg, gg
Description: This is a 4-bit carry-lookahead adder that takes in 4-bit inputs A and B and carry-in input cin to produce the addition result by computing the propagate signal P for each of the 4 bits with the expression of A^B and the generate signal G with the expression A&B. Then, along with cin, c1, c2, and c3 can be calculated with c1 = (cin&p0)|g0 and c2= ((cin&p0)|g0)&p1|g1 and so on. Afterward, with each carry-in input calculated, the sum of each bit can be calculated by XORing each bit of A, B, and c. Last, the group propagate signal pg can be computed by ANDing all the ps, and the group generate signal gg can be computed with the expression gg = g3|(g2&p3)|(g1&p3&p2)|(g0&p3&p2&p1).
Purpose: This module is designed as a 4-bit carry-lookahead adder.

Module: lookahead_adder.sv
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] s, cout
Description: This is a 16-bit carry-lookahead adder that takes in 16-bit inputs A and B and carry-in input cin to produce the addition result by using four 4-bit carry-lookahead adders. The first adder will produce pg0 and gg0. Then, the expression gg0|(pg0&cin) will produce the carry-in bit of the second adder, and so on. Last, the carry-out will be computed by using all the gg and pg produced.
Purpose: This module is designed as a 16-bit carry-lookahead adder that uses four 4-bit carry-lookahead adders.

Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This is the Hex driver that takes in 4-bit value In0 from the register in order to determine which segment of the 7-segment LED has to be light up and store the value into 7-bit output Out0 with 0 meaning turning light on and 1 meaning turning light off.
Purpose: This module is used to show the value in the adder in the LED on the FPGA board by having 7-bit output indicating which segments of the LED have to be turned on.

Module: adder_toplevel.sv
Inputs: clk, Reset_Clear, Run_Accumulate, [9:0] SW
Outputs: [9:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5

Description: This is the entire processor where each input values are used to initiate all the modules that are created and described above and organize all the modules into a single 16-bit adder with the 10-bit input SW being extended to 16-bit with 0 in front of all 10-bit.
Purpose: This module is used to combine all the modules into a single adder.

Adder Area, Complexity, and Performance Tradeoffs

To discuss the area, complexity, and performance tradeoffs of the three types of adders discussed in this lab, we first need to review the fundamental design approach of the adders. The carry ripple adder cascades the carry bit from each full adder to the next. In other words, the summation of each bit depends on the carry out bit that was generated by the previous full adder; hence the ripple effect. The benefit of implementing a carry ripple adder is the reduced area and complexity; a N-bit carry ripple adder can simply be constructed using N full adders, with each linked to the next. However, the tradeoff is that a carry ripple adder is inefficient and has poor performance due to the propagation delay of the cascading carry bit.

A carry lookahead adder, on the other hand, utilizes propagation (P) and generate (G) signals to predict the carry-out using available input bits. The generation of the carry outs from available readily available inputs eliminates the slow rippling of the carry bits that significantly decreases the performance of the carry ripple adder. However, the additional gates needed to produce propagation and generate signals as well as the carry bits leads to a significant increase in complexity and power consumption of the adder, which is the primary tradeoff for a performance increase.

Lastly, a carry select adder approaches the design with the idea of computing the sum and carry-out beforehand with the assumption that the carry-in is 0, as well as with the assumption that the carry-in is 1, then utilizing multiplexers to select the appropriate sum and carry-out utilizing the actual carry-in signal as the select signal of the multiplexer. In other words, we would have the two possible outcomes pre-computed (carry-in of 0 or carry-in of 1), then utilize the actual carry-in value to determine the final result. This effectively reduces the slow rippling of the carry bits in the carry ripple adder, leading to a performance increase over the carry ripple adder. The drawback of the carry select adder is the increase in area and complexity from having to use around twice the number of full adders as a carry ripple adder in order to pre-compute the two possible results. In conclusion, the carry ripple adder is the least efficient but least area complex, while carry lookahead and carry select adders have higher performance at the cost of increased area and complexity, specifically additional gates for the lookahead adder and additional full adders for the select adder.

Adder Design Analysis

	Carry-Ripple	Carry-Select	Carry-Lookahead
Memory (BRAM)	0/1,677,312	0/1,677,312	0/1,677,312
Normalized Memory (BRAM)	1.000	1.000	1.000
LUTs	79	85	88
Normalized LUT	1.000	1.076	1.114
Max Frequency	67.95 MHz	67.16 MHz	67.69 MHz
Normalized Max Frequency	1.000	0.988	0.996
Total Power	107.20 mW	107.51 mW	107.44 mW
Normalized Total Power	1.000	1.003	1.002

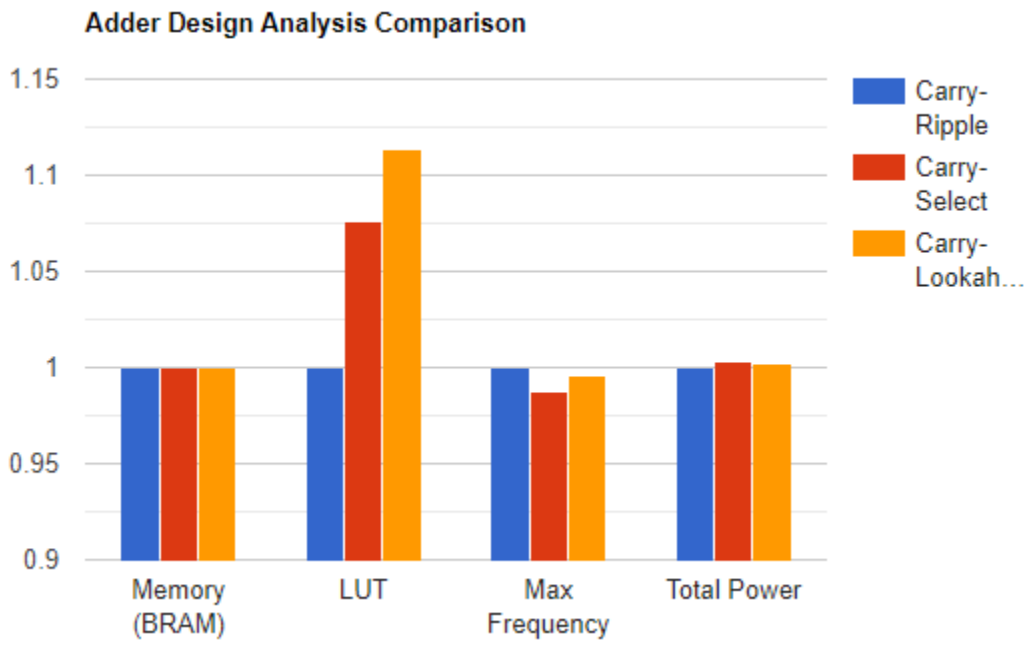

Figure 9. Adder Design Analysis Data Comparison

Annotated Simulation Trace

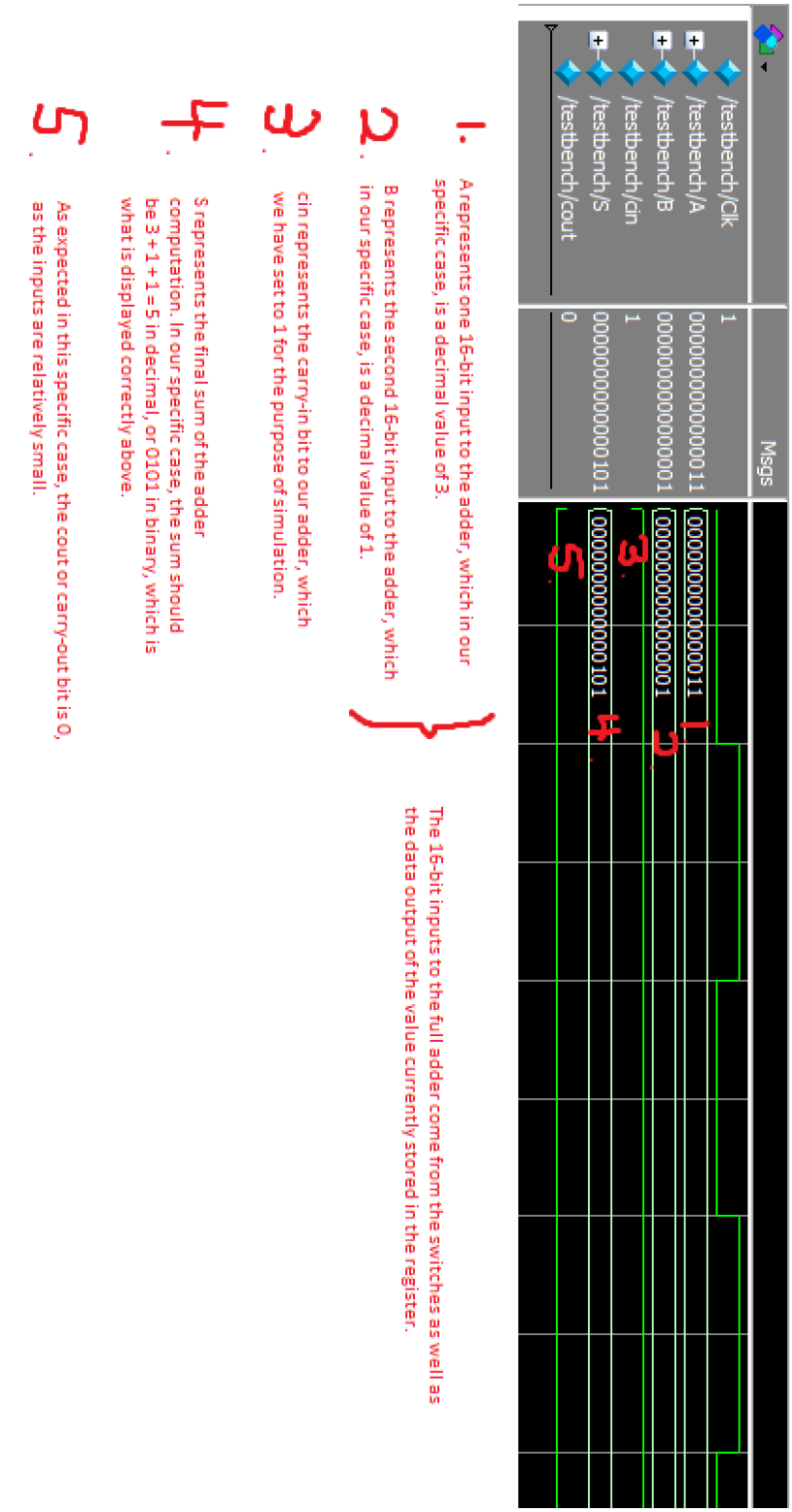

Figure 10. Annotated Simulation Trace of an Adder Computation

Critical Path Analysis (Extra Credit)

Carry-Ripple Adder Critical Path Analysis:

For the carry-ripple adder, from the time analyzer shown in Figure 11, the measured slack value is about 8.691ns, meaning that the adder needs this amount of time for it to exceed the clock period. Compared with all other adders, the carry-ripple adder tends to have the greatest slack value, which means it completes the task faster than the other 2 adders; however, this doesn't match the expected result from our manual analysis. According to the critical path shown in Figure 12, the carry-ripple adder tends to have the longest path with a series of full adders, which means it will have to wait for a longer time to receive the result as each full adder has to wait for a clock cycle to receive the carry-in bit. Nonetheless, the carry-ripple adder is the fastest.

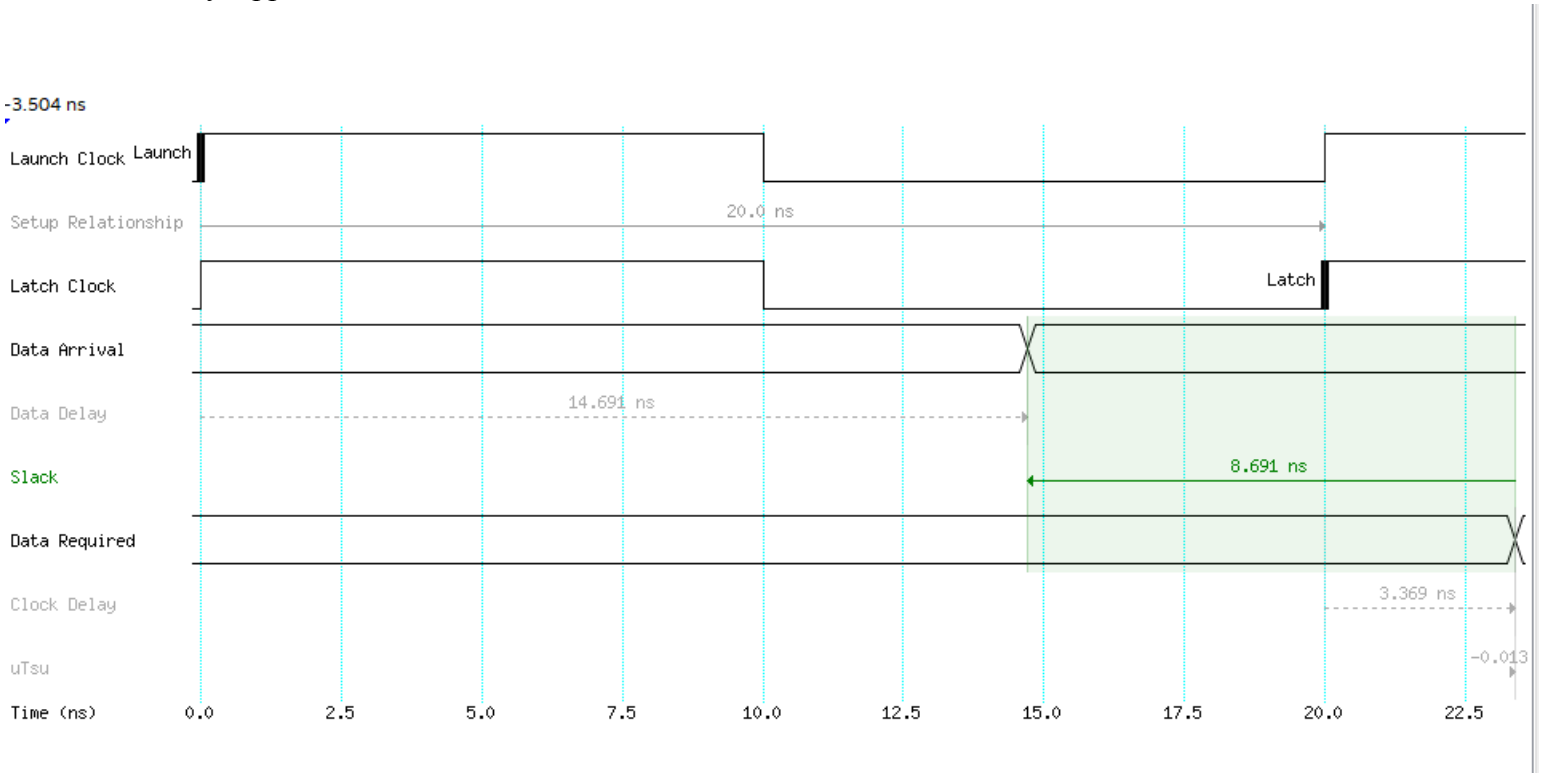

Figure 11. Time Analyzer for Carry-Ripple Adder

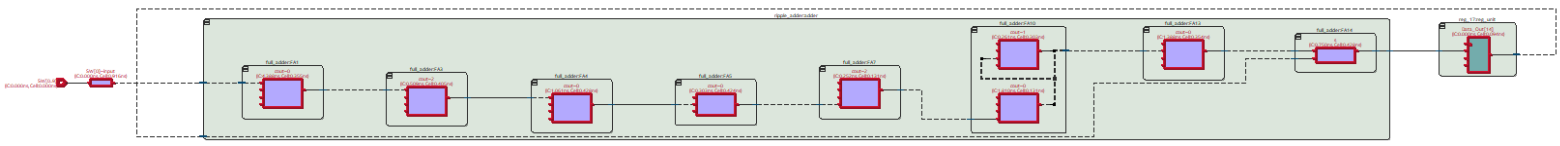

Figure 12. Critical Path for Carry-Ripple Adder

Carry-Lookahead Adder Critical Path Analysis:

Furthermore, for the carry-lookahead adder, its time analyzer from Figure 13 shows a slack of 7.472ns, meaning that it has less waiting time than that of the carry-ripple adder and thus, is slower. This also contradicts the critical path of the carry-lookahead adder. According to Figure 14, the critical path of the carry-lookahead adder has fewer components that are triggered by the clock cycle. Thus, theoretically, it should have a faster time than the carry-ripple adder.

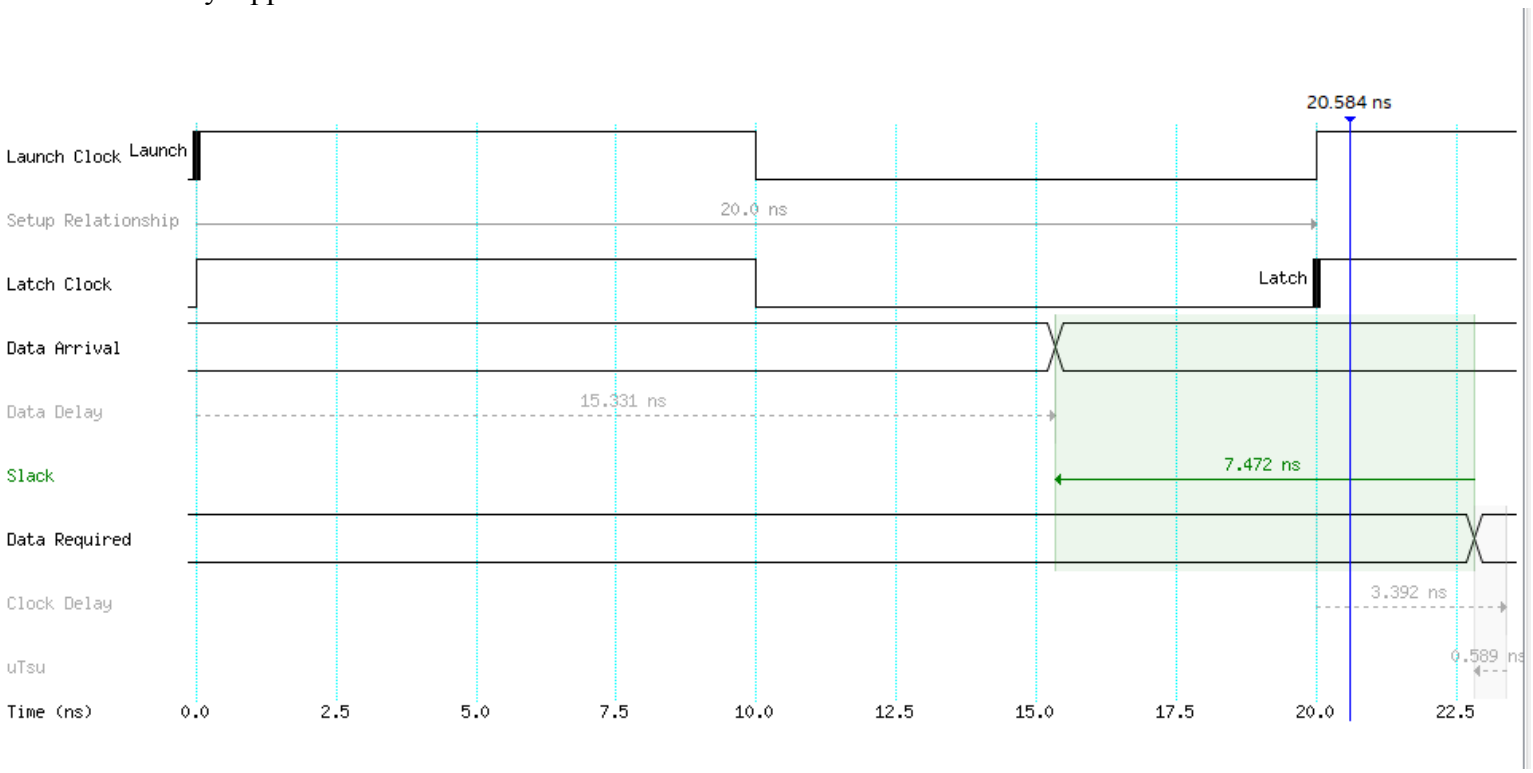

Figure 13. Time Analyzer for Carry-Lookahead Adder

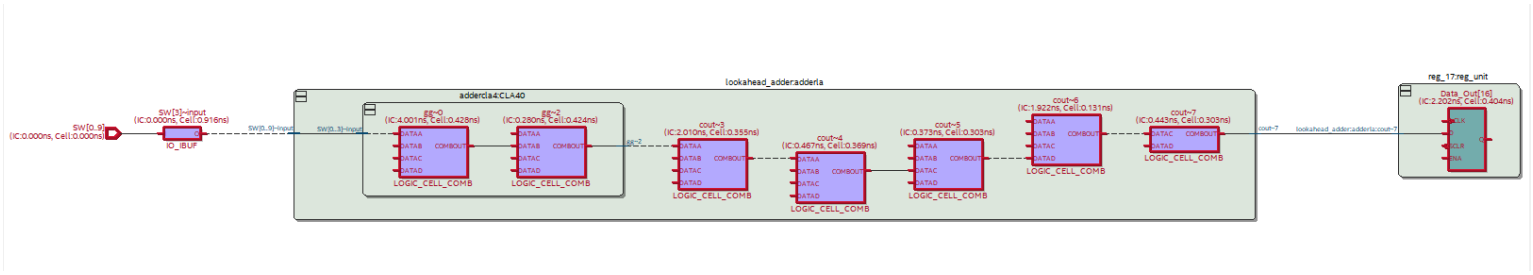

Figure 14. Critical Path for Carry-Lookahead Adder

Carry-Select Adder Critical Path Analysis:

Last but not least, the carry-select adder has a slack value of 8.474ns, which is greater than the carry-lookahead adder but slower than the carry-ripple adder. Ideally, as the carry-select adder only has to wait for one clock cycle for the carry-in bit to propagate to each 2-to-1

MUX, it should spend the least amount of time doing the task; however, it is only the second fastest.

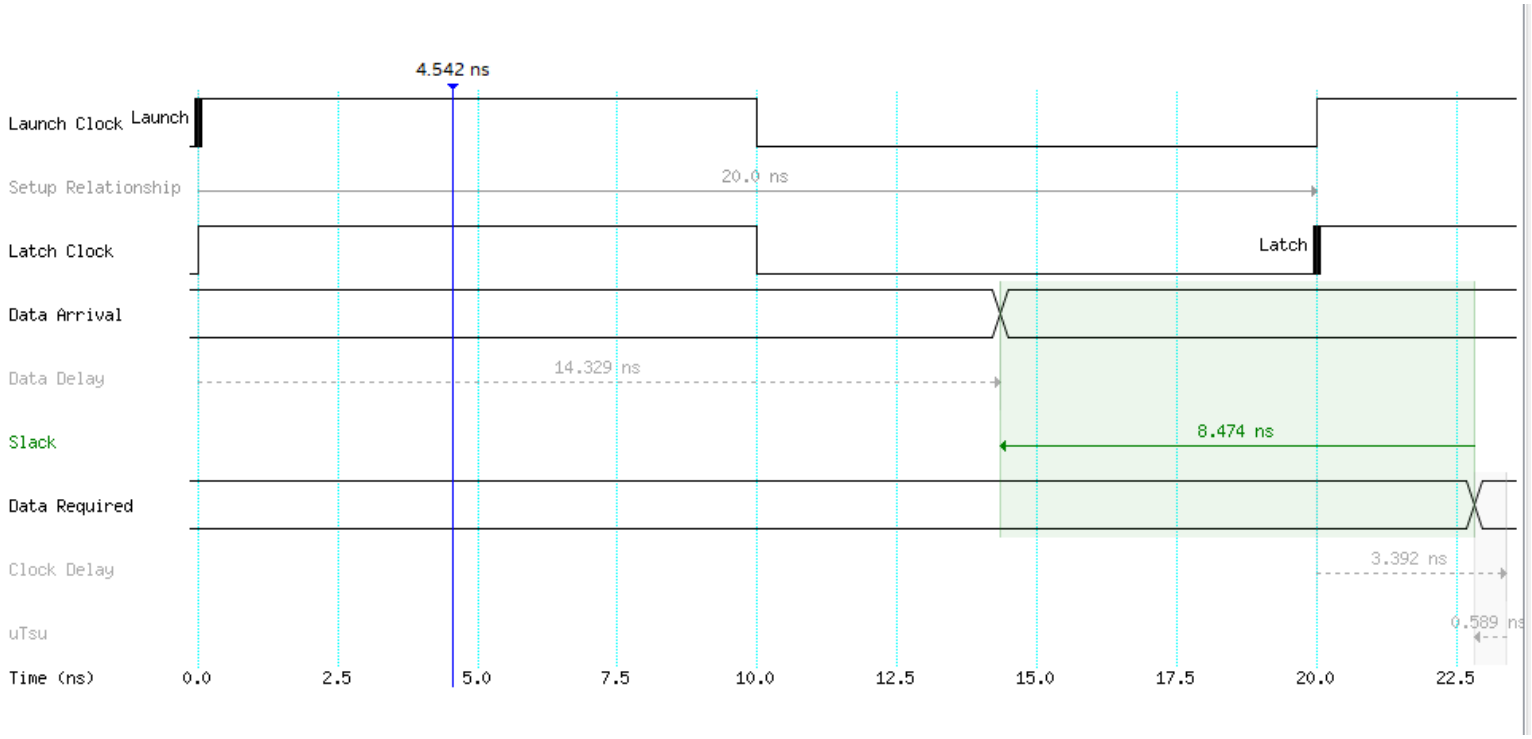

Figure 15. Time Analyzer for Carry-Select Adder

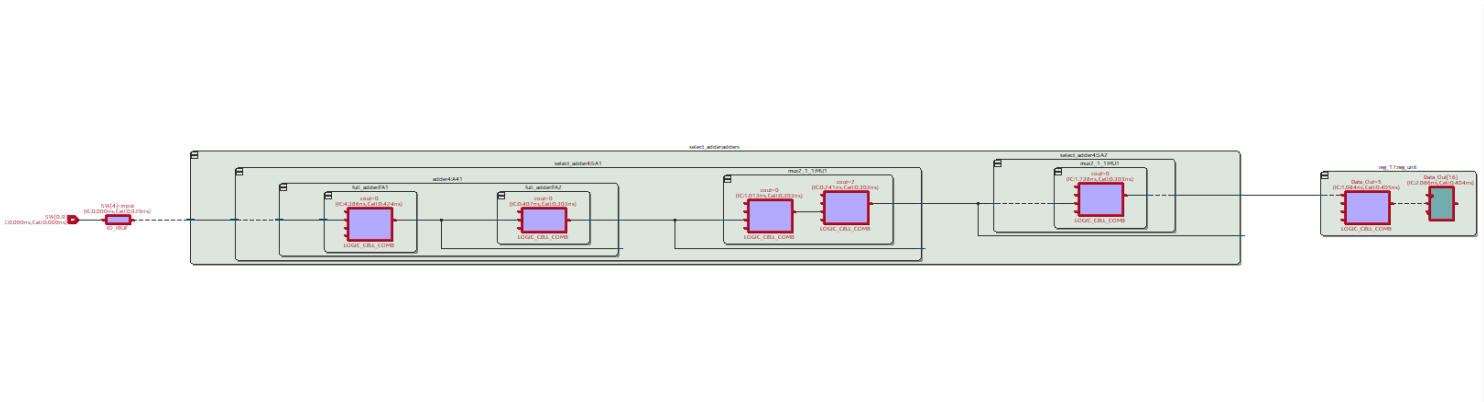

Figure 16. Critical Path for Carry-Select Adder

From the results obtained from the critical analysis simulation, we see that the slack value of the carry-ripple adder is the greatest, followed by the carry-select, with the carry-lookahead having a relatively smaller slack value compared to the other two adders. This is in contrast with our manual analysis, which predicts a greater slack value for the carry-select adder, followed by the carry-lookahead then the carry-ripple adder. We reached this conclusion from our manual analysis based on the fact that the carry-select adder requires only 4 clock cycles to complete the 16-bit adder operation (although more gates are involved, the logic is only 2-level), in contrast to the 16 clock cycles required to complete the same operation in a carry-ripple adder (due to the slow ripple of the carry bit from one adder

to the next). The carry-lookahead adder, like the carry-select adder, should also require less time to complete the operation compared to the carry-ripple adder, as the carry bits are pre-generated utilizing available inputs. Our manual analysis conflicts the result from the critical analysis simulation, which may be explained by routing delay. The lookahead adder involves more signals in predicting the carry bits, which may have a significant impact on the slack value if we take into consideration the potential routing delay between logic inputs and outputs. Likewise, the select adder would also be impacted due to its reliance on multiplexers to determine the sum and carry bits.

Post-Lab Questions

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 hierarchy on the FPGA (what information would you need, what experiments would you do to figure out?).

The 4x4 hierarchy implementation of the carry-select adder is not necessarily ideal, although it would be hard to specify whether the implementation is ideal without stating the constraints and factors that limit our design. If we consider an ideal implementation simply to be the fastest implementation, then technically a flat, single-level approach of implementing a 1x16 carry-select adder could yield a faster result. This is because we would be able to bypass the delay of the muxes in selecting and propagating the correct carry bit and sum in between the 4-bit carry-select adders. However, if we make considerations from a complexity and power usage standpoint, the 1x16 approach would be less ideal and not as easily scalable as the 4x4 hierarchical design. Thus, in order to determine the ideal approach for designing the hierarchy of the FPGA, we would ideally want information about the technical limitations of the FPGA, such as its available flip-flops and logic elements. Without specific constraints, experimenting with different hierarchical implementations and then analyzing the results, including power consumption and speed, would be a suitable way of finding the best design approach.

Complete the design statistics table for each adder.

	Carry-Ripple	Carry-Select	Carry-Lookahead
LUTs	79	85	88
Normalized LUT	1.000	1.076	1.114
DSP	0	0	0
Normalized DSP	1.000	1.000	1.000
Memory (BRAM)	0/1,677,312	0/1,677,312	0/1,677,312
Normalized Memory (BRAM)	1.000	1.000	1.000
Flip-Flop	20	20	20
Normalized Flip-Flop	1.000	1.000	1.000
Max Frequency	67.95 MHz	67.16 MHz	67.69 MHz
Normalized Max Frequency	1.000	0.988	0.996
Static Power	89.98 mW	89.98 mW	89.98 mW

Normalized Static Power	1.000	1.000	1.000
Dynamic Power	1.39 mW	1.74 mW	1.64 mW
Normalized Dynamic Power	1.000	1.252	1.180
Total Power	107.20 mW	107.51 mW	107.44 mW
Normalized Total Power	1.000	1.003	1.002

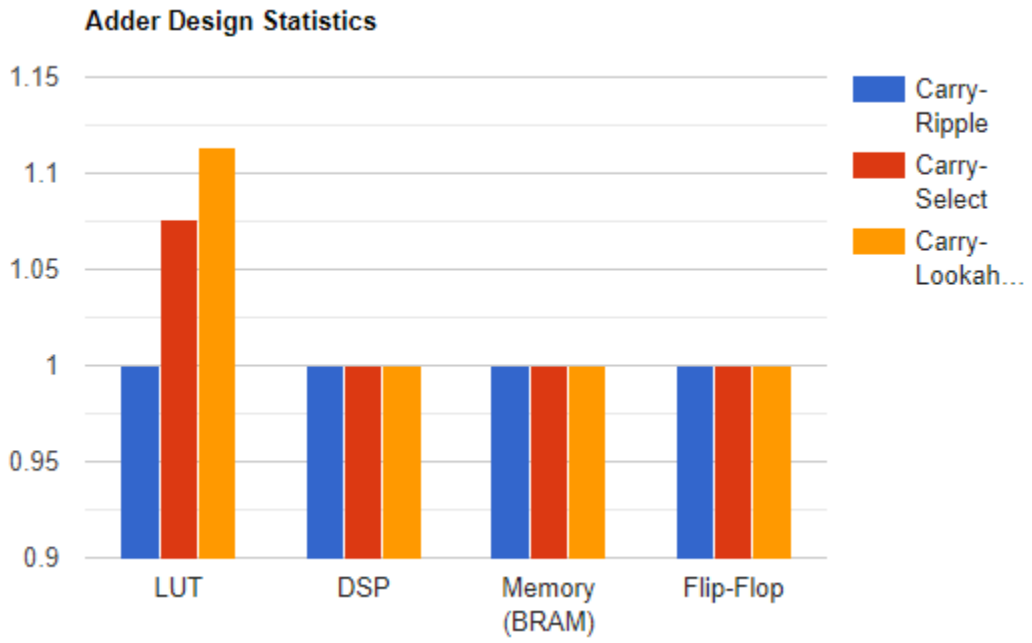

Figure 17. Adder Design Statics Graph Part I.

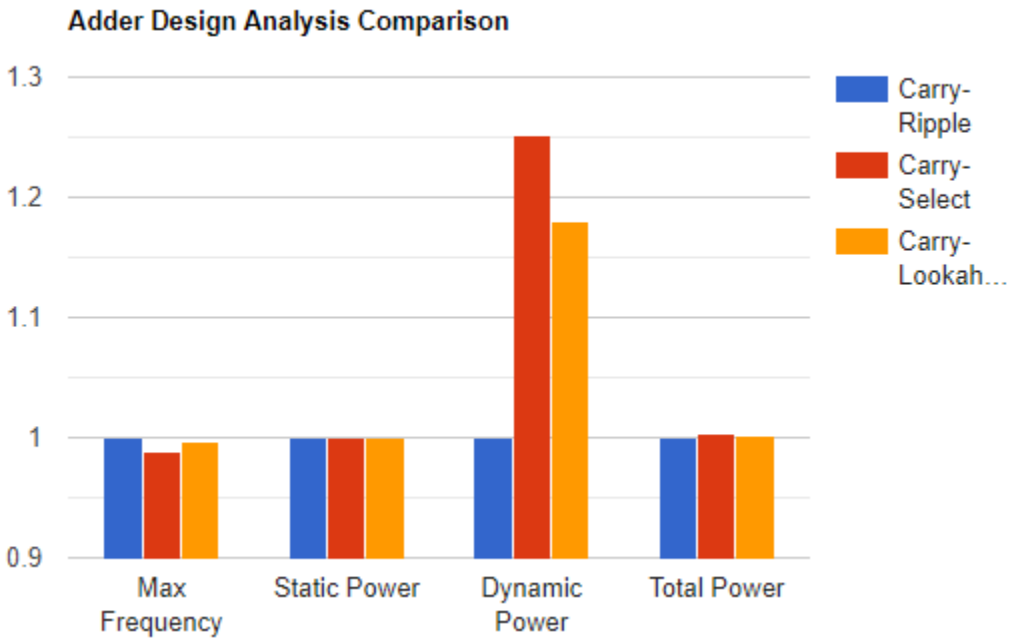

Figure 18. Adder Design Statistics Graph Part II.

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?

If we observe the data plot above, we can see that the carry-lookahead adder utilized the most LUTs, with the carry-select adder not far behind. This makes sense if we consider the additional logic elements and components that are required to construct the lookahead and select adders, particularly the multiplexers and additional full adders in the select adder as well as additional logic gates to predict the carry bits in the lookahead adder. In addition, the carry-select and carry-lookahead adders also consumed relatively more power than the carry-ripple adder, which is as expected, due to the added complexity of the two adders in comparison to the ripple adder.

On the other hand, we can also see that the DSP, memory (BRAM), and number of flip-flops is the same amongst all three adder designs. The reason for this due to our adders not utilizing any memory to implement, as well as the fact that our adders are all processing the same number of bits, 16, which means that the number of flip-flops should not be different amongst the adders.

Based on the results, the carry-ripple adder had the highest maximum operating frequency, with the carry-lookahead coming second, followed by the carry-select adder. This result does not comply with the theoretical design expectation of the maximum operating frequency of the carry-lookahead adder being higher than that of the carry-ripple adder. However, a possible explanation could be due to the relatively small number of bits being processed by our adders. I believe if we increase the number bits, the maximum frequency disparity would increase in favor of the carry-select and carry-lookahead adders.

Conclusions

Describe any bugs and countermeasures taken during this lab. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? What have you learned? What worked? What didn't? What can you possibility do to enhance/simplify/fix your circuit to make it better?

In this lab, we experimented with and learned about 3 different 16-bit adder implementations, namely the carry-ripple adder, 4x4 carry-select adder, and 4x4 carry-lookahead adder. Through analysis of the design approaches and outcomes, we learned about the tradeoffs between the different design approaches, such as how the carry-select and carry-lookahead adders increased performance speed at the cost of increased complexity and power consumption. During our design implementation, we encountered some confusion regarding the method with which to implement the 4x4 carry-lookahead adder. We initially approached the design by substituting in sequentially generated carry bits to generate the next carry bits. However, we quickly realized that this would result in the delay of the slow ripple that we wanted to eliminate, thus highlighting the importance of reducing the time required to generate and propagate the carry bits in seeking the ideal adder design implementation.