# Lab Report #3

Aryan Shah (aryans5), Dev Patel (devdp2)

University of Illinois at Urbana Champaign

ECE 385 - Digital Systems Laboratory

Professor Zuofu Cheng, T.A. Gene Lee

September 25, 2023

## Introduction

***Summarize the high-level function performed by the three adders***

The purpose of this lab was to implement a binary adder utilizing an RTL design on our FPGA. We designed and compared the performance metrics of three different adders: carry-ripple, carry-lookahead, and carry-select.

Our carry-ripple adder works by taking in 2 binary numbers as inputs, adding them one by one (least significant to most significant bit) using full adders. Each following sum also takes into account a carry-out from the previous stage then being fed in as a carry-in for the next stage, up until the most significant bit. As binary values get larger, the performance of this adder worsens as there is more propagation delay due to waiting for the carry value to ripple across stages.

The carry-lookahead adder works by calculating a generate (G) and propagate (P) term. The carry out is then calculated by the carry-in of the previous bit in addition to the G and P terms. This allows for an improvement in performance as opposed to the carry-ripple adder due to the fact that it avoids serial carry propagation by precomputing the P and G terms, and thus carries can be calculated in parallel.

Finally, our carry-select adder involved dividing up our binary inputs into blocks, which operate independently of each other. For each block, separate sum and carry-out values are calculated beforehand. Using the carry-in from the previous block, a select operation is then performed. As the carry-out from the previous block is used as the next carry-in, we can also say that this is a significant performance increase as compared to the previous adders, using a cascading design as well as eliminating propagation delay.

## Adders

### Ripple Carry Adder

#### *Written description of the architecture*

The carry-ripple adder was composed of an N number of full adders. Each full adder feeds in 2 binary numbers and a carry-in as inputs. Each iteration produces a single-bit sum and a carry-out value. Since the full-adders are connected by these carry values, the previous carry-out value then becomes the carry-in value for the next sum. The process of rippling carries and computing sums go from the least significant bit up until the most significant bit, which produces the final sum and carry-out value. This was the easiest adder to implement, but it was very inefficient especially at higher input values due to propagation delay. Every full-adder had to wait for the next adder to compute the correct carry value, thus severely worsening the computation time.

#### *Block diagram*

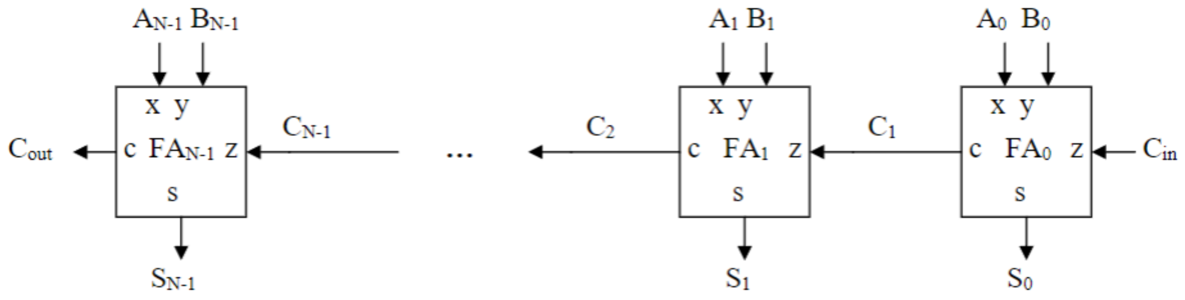


*Figure 3 - n-bit Carry Ripple Adder Chain*

## Carry Lookahead Adder

### ***Written description of the architecture***

The carry-lookahead adder was required to make use of a 4x4 hierarchical design. In this adder, a generate term is produced by taking the AND value of the 2 input bits at a specific position (this indicates whether a carry will be generated if both inputs are 1). Next, a propagate term is calculated by taking the OR value of the two inputs at that specific position as well (this indicates whether a carry will be generated from the previous position). The carry-out can thus be calculated by this formula: Cout = G + (P * Cin). Since this is a pre-calculated value, we do not have to wait for carry values to "ripple" through any adders, therefore eliminating propagation delay in this design. Finally, the sum for each bit position is calculated with the following formula: S = A XOR B XOR Cin. However, the iterations of carries are still calculated in a parallel manner, therefore making them not the most efficient adder of the list.

### ***How the P and G logic are used***

Our P and G values were utilized to pre-calculate the values for our next carry-out. P (propagate) was calculated by taking a logical OR of the two input bits at a specified position, determining whether a carry will then propagate from the previous position. G (generate) was calculated by taking a logical AND of the two input bits at the specified position, determining whether a carry will be generated in the case that both inputs are binary '1's. The formula to calculate the carry-out is as follows: Cout = G + (P * Cin).

***How the hierarchical 4x4 adder was used***

In order to implement the 4x4 hierarchical design, we constructed 4-bit carry-lookahead adders and integrated them into our larger, final adder. The 16 bits from our inputs A and B were divided into groups of 4 bits. Each group of bits is assigned to its own 4-bit adder, consisting of 4 1-bit adders. Each 4-bit carry-lookahead adder generates its own propagate and generate values, and thus are able to calculate their own carry values rather than waiting for it to be rippled through from the previous adder.

***Block Diagram inside a single CLA (4-bits)***

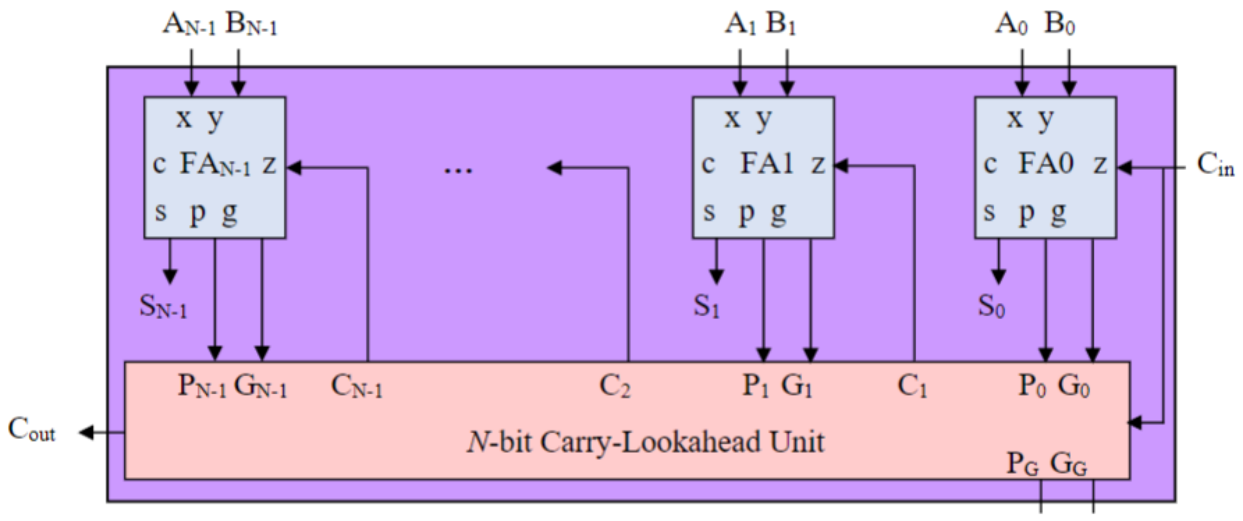
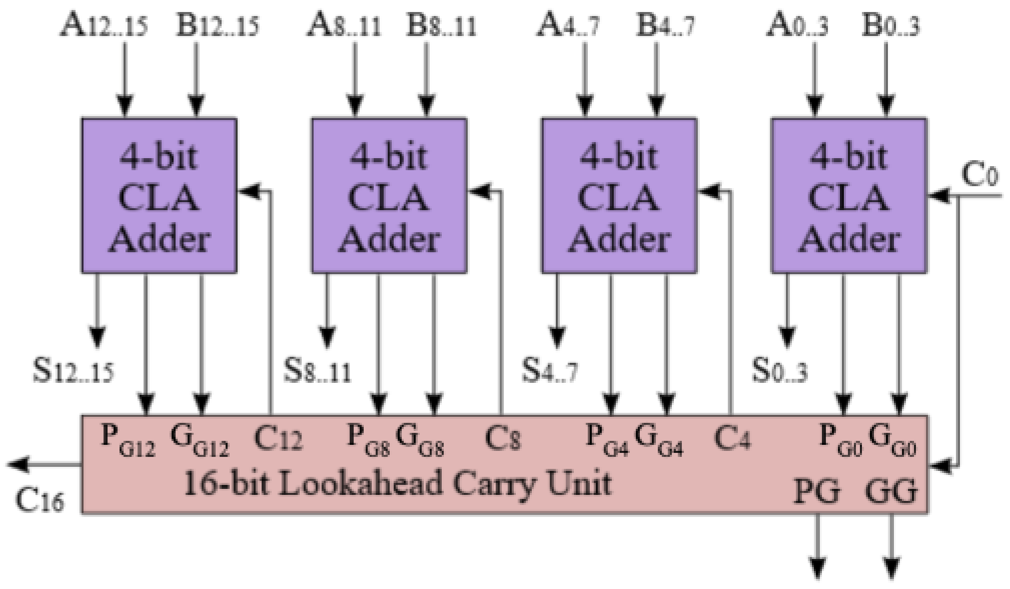
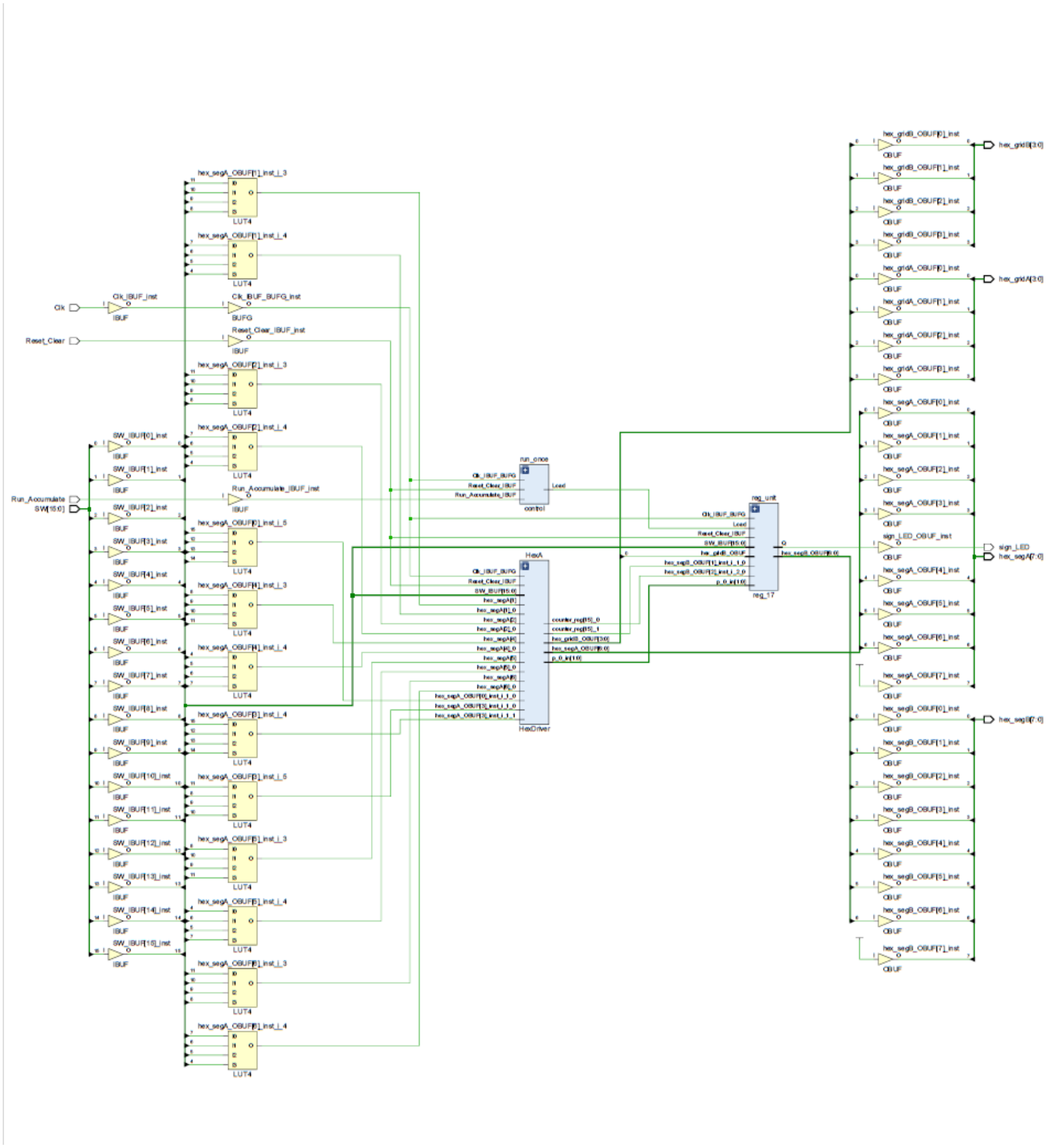


*Figure 4 - n-bit Carry Lookahead Adder*

***Block diagram of how each CLA was chained together***



***RTL DIAGRAM:***



## Carry Select Adder

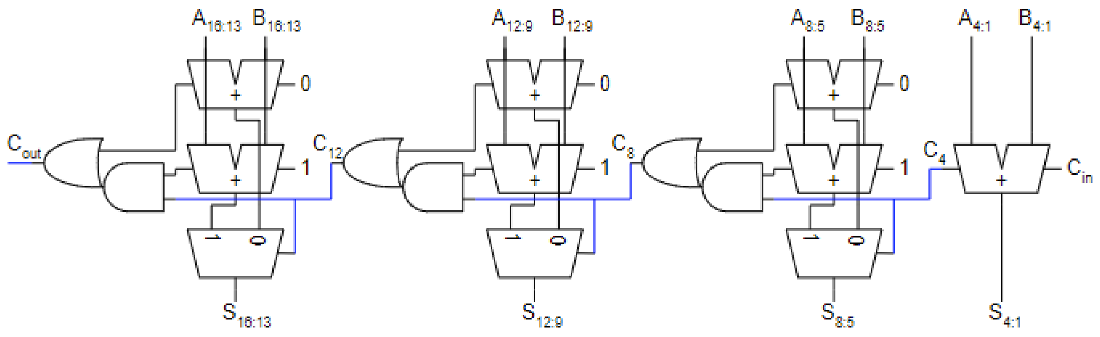
### *Written description of the architecture*

The carry-select adder also took in two binary numbers as inputs, but divided them into different blocks. In our case, we also utilized a 4x4 hierarchical design for this adder and thus they were divided into 4 bit blocks. Two separate sums for each block are pre-computed, a sum assuming no carry in and a sum including a carry in value. A carry out is also pre-computed accordingly for each case. Then, a select operation is performed based upon whether a carry was actually generated or not from the previous block. Finally, the individual sums are connected to one another and a final carry out from the most significant bit's addition is also outputted as the final carry out. Since all these blocks operate independently of each other, it can be categorized as a cascading design. It can be realized that due to the fact of having a cascading design and eliminating propagation delay, the carry-select adder is the least computationally intensive of all the adders.

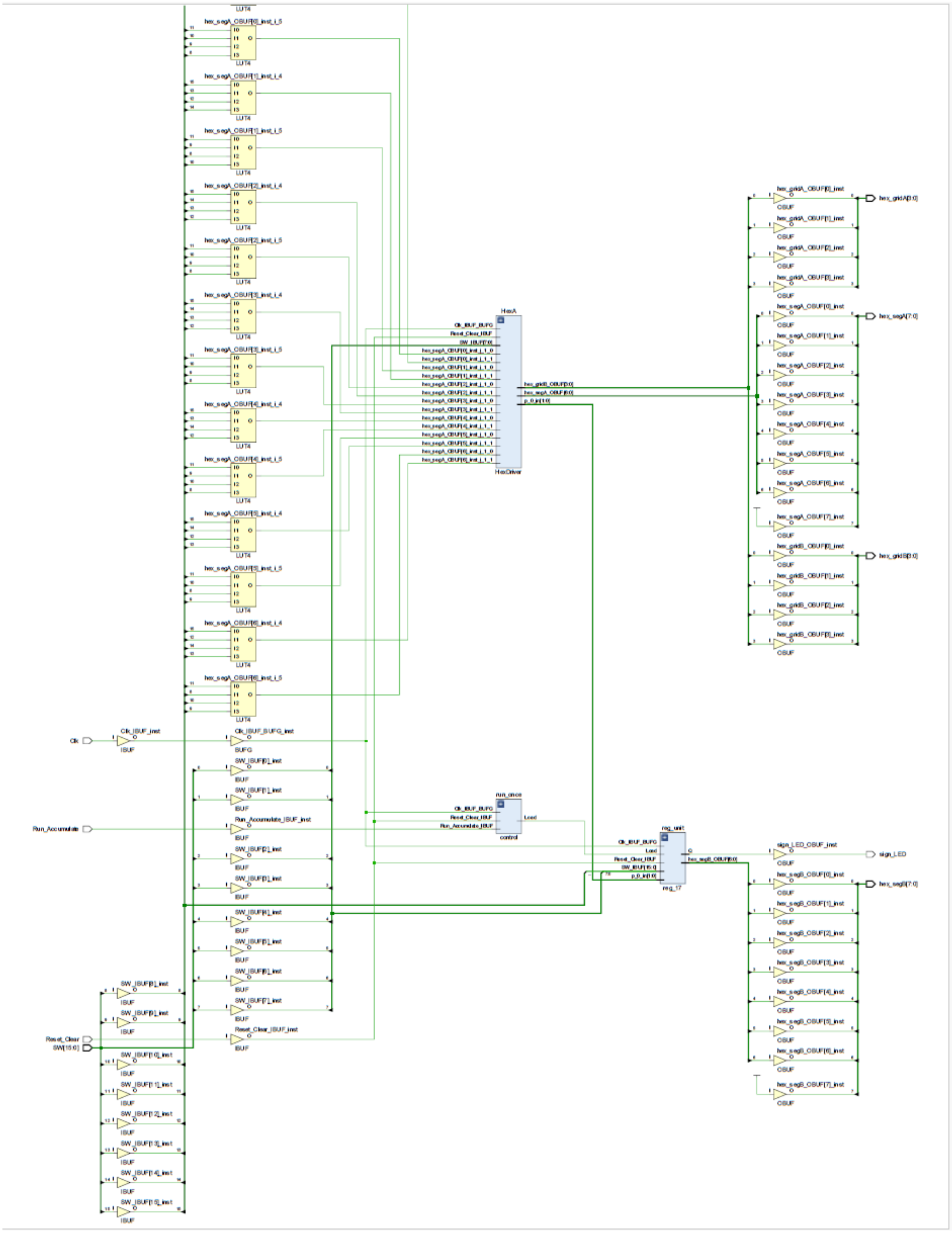
***How the CSA speculatively computes multiple sums in parallel and rapidly chooses the correct one later***

The way the sums are computed involves firstly pre-calculating "guessed" sums. This is done by calculating a sum for each of the two possibilities: assuming there is no carry from the previous block or assuming that there is one. Along with a sum, the carry-out is also pre-calculated based upon the idea that there both exists and does not exist a carry-in. When the carry-in does actually feed into the next block, all the adder has to do is select the right values that were already pre-computed. The block then sends the appropriate carry-out to the next block, and the process repeats until we reach the most significant bit position and the final sum along with the final carry-out are both calculated.

***Block Diagram of the whole CSA circuit containing adders, multiplexers, and glue logic***



***RTL DIAGRAM OF CSA:***



**Written description of all .SV modules**

Adder_toplevel.sv - This file holds the top level logic of our design. It connects the rest of the modules together and defines the input/output logic and variables. It also connects the output signals of our hex display to the LEDs.

Control.sv - This module handles the logic that creates a clock cycle to handle the timing of the switches

Mux_2_1_17.sv - This module incorporates the design of a 17-bit multiplexer.

Reg_17.sv - This module resets and loads values accordingly based upon the user's input to the switches when pressing either the load button or reset button.

Lookahead_adder.sv - This module defines our carry-lookahead adder. It also defines our standard logic for a 1-bit adder but then divides itself into a 4x4 hierarchical design. We instantiate our logic for the propagate and generate bits, therefore each adder does not depend upon the carry-out of the previous one.

Hex.sv - This module defines the input/output logic so that the FPGA's hex drivers are passed in accurate values based upon switch inputs

Ripple_adder.sv - This file instantiates another full adder utilizing our established design. The eight bit ripple adder combines each 1-bit adder from the full adder to create an 8-bit adder. Finally, the ripple adder module (finalized adder) connects 2 8-bit adders to complete the 16-bit ripple adder design. It connects them based on the carry-out of the previous adders as the carry-in to the next.

Select_adder.sv - This file handled the logic for our select adder. It involved a multiplexer unit that instantiates the input/output scheme to the 3 different MUXes in our adder. A full 1 bit adder utilizing our standard design was again defined. A four bit ripple adder to divide the adder in our

4x4 design as also defined, computing its own sum based on “guessing and checking” each case scenario of carries. Finally, it defines the carry-select adder which computes the final sum and carry out value for the entire adder.

**High level area, complexity, and performance tradeoffs amongst adders**

Beginning with the ripple adder, it is the least area-efficient among all the adders. Due to its structure of rippling carries across stages involving propagation, it takes up more area than our other designs. In terms of complexity, however, it was the easiest to implement out of the three. When it comes to performance, the ripple adder lacks heaviest in this area. Although it was easy to design, at larger input numbers it must wait for the carry to propagate across each bit position and thus is the slowest in terms of speed. The carry-select adder is a good medium between our adders. It is more area-efficient as compared to the carry-ripple due to the fact that it precomputes carry values using the propagate and generate bits. However, it is still not the most efficient in our design. Its design required a moderate level of complexity, being harder to design than the carry-ripple but easier than the carry-lookahead. As a result of this higher complexity, though, we see a faster performance than the carry ripple due to eliminating propagation delay with its pre-computed values using P and G. Finally, our carry-lookahead adder was the most area-efficient among the three. Having pre-computed sum values and carries and then choosing the correct one to carry based on the previous value made it the most complex to design. However, it is the fastest performing adder especially at higher valued input numbers.

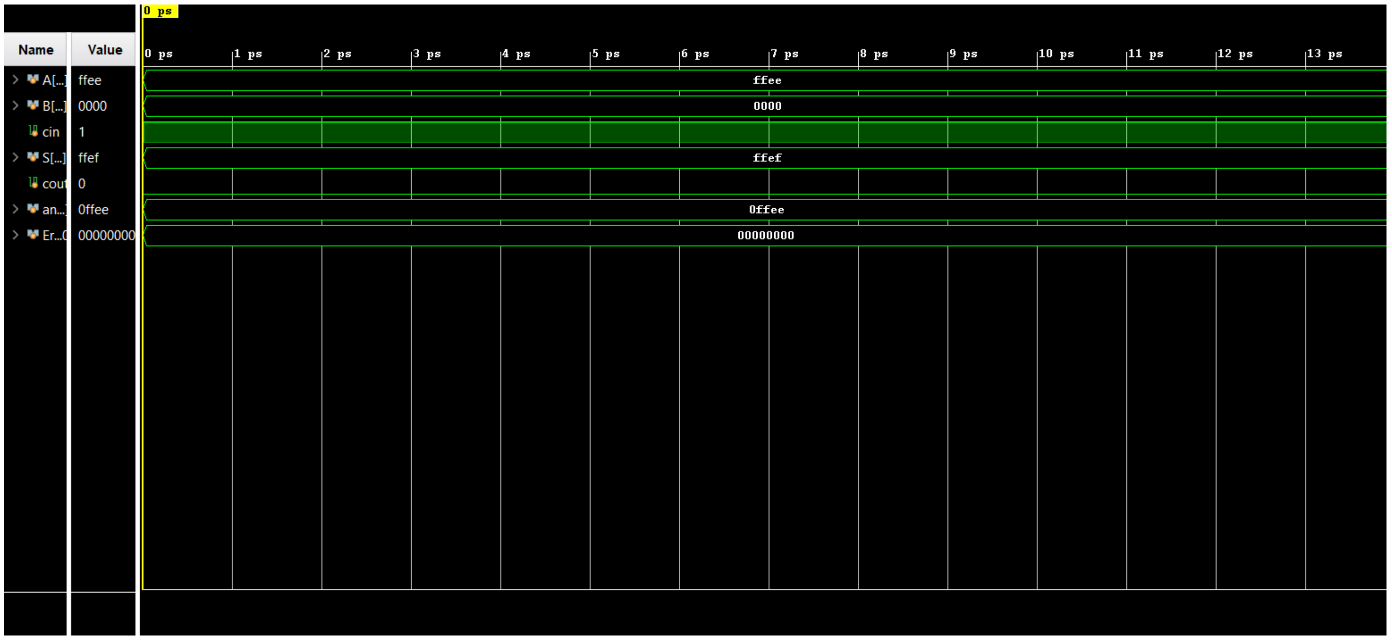
**Performance of each adder (Post-Lab Part 1 graph)**

| | Carry-ripple | Carry-select | Carry-lookahead |
|---|---|---|---|
| LUT | 84 | 91 | 91 |
| Frequency | 149 MHZ | 236 Mhz | 178 Mhz |
| Total Power | 0.088W | 0.088W | 0.088W |

**Table for remaining performance metrics (Post-Lab Part 3)**

| | Ripple | Look Ahead | Select |
|---|---|---|---|
| LUT | 84 | 91 | 91 |
| DSP | 0 | 0 | 0 |
| Memory (BRAM) | 0 | 0 | 0 |
| Flip-Flop | 53 | 53 | 53 |
| Frequency | 149 MHZ | 178 MHZ | 236 MHZ |
| Static Power | 0.074W | 0.074W | 0.074W |
| Dynamic Power | 0.014W | 0.014W | 0.014W |
| Total Power | 0.088W | 0.088W | 0.088W |

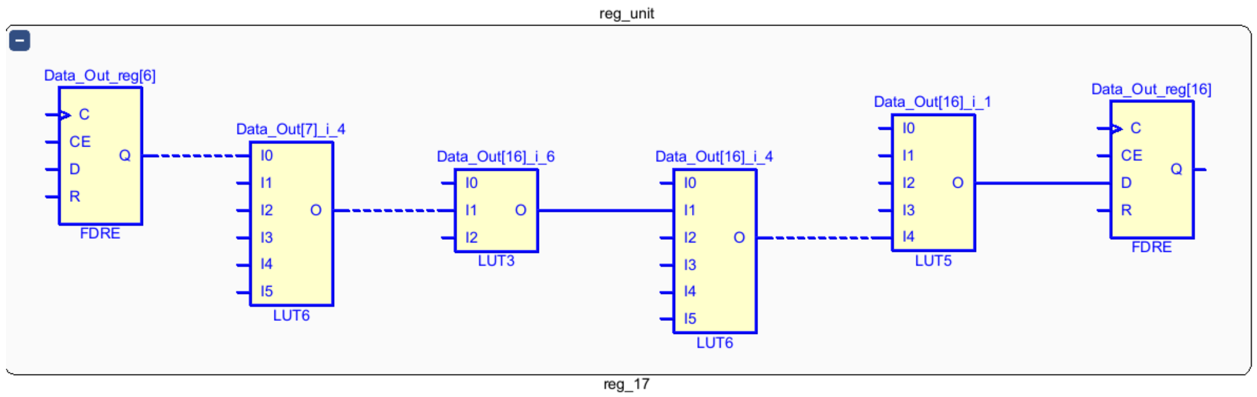
**Annotated simulation trace**



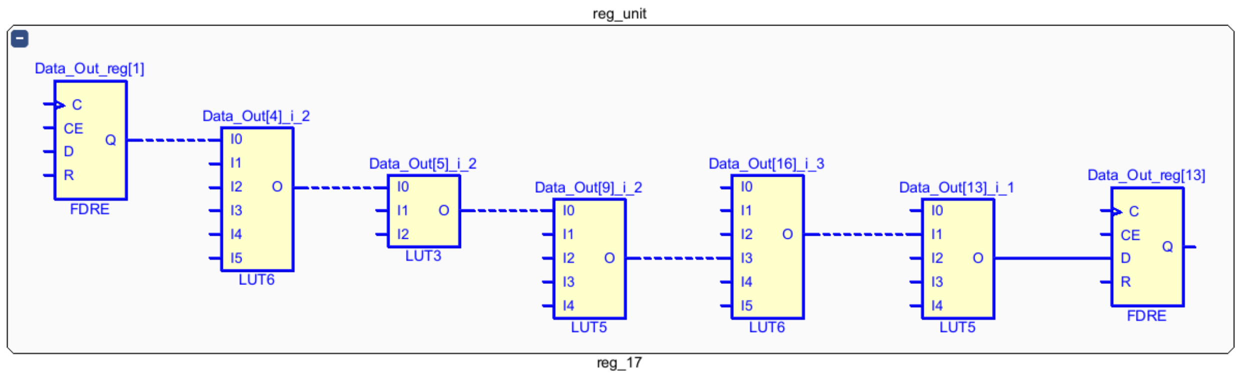
1) Values of A, B, and carry in are loaded in to FFEE, 0000, and 1 respectively
2) We then execute the sum operation
3) We see that our output is FFEF since even though B is 0000, we have a carry in of one
4) This operation is demonstrated on the waveform as per the testbench

**Critical path analysis (screenshot of each) and theoretical understanding of each adder (Post-Lab Part 4)**

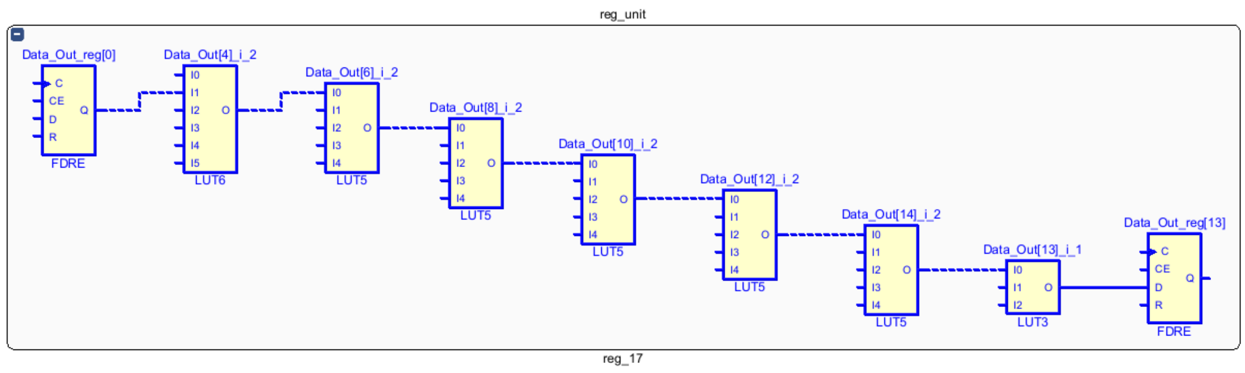
Select Carry Critical Path:



Look Ahead Critical Path:



Ripple Adder Critical Path:



Looking at the critical paths, we can see that the ripple carry adder uses more components than the other two adder critical paths. This confirms our understanding that the ripple carry adder is the least efficient adder when chained together for the larger lab application.

## Answers to Post-Lab Questions

**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. Does each resource breakdown comparison from the plot make sense? Are they complying with the theoretical expectations? Which design consumes more power than the other as you expected, why?**

We first calculate the clock period by taking WNS and putting it into this formula: 1000 / (10-WNS) MHz. Resource breakdown comparison is the analysis of the behavior of FPGA resources. We measured these utilizing Vivado tools and reported them in the tables above. The resources do make sense. Earlier in this report we discussed the complexity, performance, and resource use and tradeoffs amongst each adder. We were able to validate this with the data we received from the resource breakdown. In general, the ripple adder was the least complex to design, with the lookahead being next and finally the select adder. On the other hand, our performance metrics reported that the select adder did the best, then the lookahead, and finally the ripple adder. As we mentioned the theory behind these values above, we were able to validate them with actual metrics. The design that consumed the most power was our ripple carry adder. Although it was a less complex design, its performance lacked due to the carries having had to be calculated across each bit position. This makes the adder consume the most power out of any of the other adders, due to the high amount of computation it has to do especially at high input values.

**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?**

The ideal hierarchy would depend on the constraints that we are given. Since we utilized a 4x4 design, we were able to increase the size of our select adder by 4-bit blocks without having to make major implementations to our logic. This allowed for us to scale our design as new blocks were needed. Additionally, there are performance and power constraints to also consider. If we were to prioritize performance, we could go for a less-complex design such as a 2x8 hierarchy. However, this would come at the expense of power, where the 4x4 hierarchy would be better. To sum it up, the ideal hierarchy would depend upon the constraints we are given and the metrics we wish to prioritize for our final design.

**Did the critical path for the Vivado carry-ripple adder consist of the carry chain through each full adder module? Why or why not?**

The critical path for the carry-ripple adder did consist of the carry chain through each full adder. As noted in our design objectives for the ripple adder, the carry value propagates across each adder to be used as the carry-in. Therefore, it makes sense that we see a serial carry chain across each full adder in our Vivado critical path. In the ripple adder, the carry-out of the previous bit position is used as the carry-in for the next. We see this in our critical path as the carry signal traverses through each full adder, showing the delay we get from carry propagation.

## Conclusion

### Bugs/Countermeasures taken during the lab

There were no significant bugs that we encountered throughout the lab. The only major instance of us pivoting our design came after we designed our carry-ripple-adder. Our ripple adder made use of an 8x2 design rather than a 4x4, and thus we had to be mindful for the remaining adders to implement a 4x4 hierarchical design as they were required to be such.

### Ambiguity, errors, or difficulty in lab materials

N/A

### Additional summary

The process of completing this lab was our first instance of differentiating performance and optimization metrics of different designs that achieved the same purpose. By seeing the difference in the adders' area, power, and operating frequencies we were able to notice a large discrepancy in performance especially at higher input values. This taught us that it is important to consider multiple designs that have different constraints when working through a project.