ECE 385
Fall 2023
Experiment #3

Carry Ripple, Carry Lookahead and Carry Select Adders

Cher Rui Tan / Adithya Balaji
11.30 am / Gene Lee

Introduction

This lab involved the designing and implementing of three types of binary adders: The Carry Ripple Adder (CRA), the Carry Lookahead Adder (CLA) and the Carry Select Adder (CSA). Each adder had its own implementation as well as logic which resulted in different pros and cons of each adder, often sacrificing timing or simplicity of logic. In this report, we aim to compare the differences between the different adders and how to decide between using each adder and for what purpose.

Ripple Carry Adder

The Ripple Carry Adder was the most simple to implement. It is constructed using 16 full-adders. A full-adder is a single-bit version of the binary adder, where three binary bits (A, B and Cin) are inputted through a set of logic gates to produce a single-bit sum (S) and a single-bit carry-out (Cout), as shown in Figure 2. The 16 full-adders are then linked together in series through the carry bits, forming an 16-bit binary adder. When the binary inputs are provided, the full-adder of the least significant bit (LSB) will produce a sum (S0) and a carry-out (C1). The carry-out is fed to the carry-in of the second full-adder, which then produces a second sum (S1) and a second carry-out (C2). The process ripples through all 16 bits of the adder as shown in Figure 3, and settles when the full-adder of the most significant bit (MSB) outputs its sum (SN-1) and carry-out (Cout), which are the results that we want.

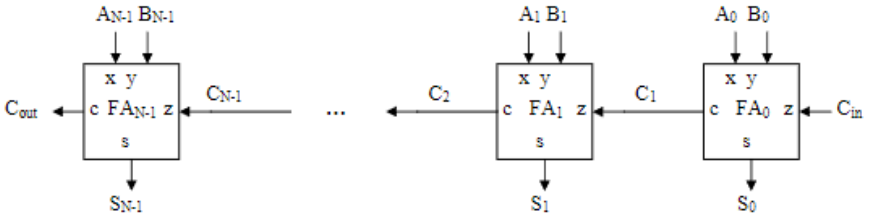

Figure 1 : Carry Ripple Adder

Carry Lookahead Adder

The 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 immediately available inputs (A and B), and predicts what its carry-out would be for any value of its carry-in.

The CLA functions using Propagate (P) and Generate (G) logic. A carry-out is generated (G) if and only if both available inputs (A and B) are 1, regardless of the carry-in, and this is represented by the equation $G(A, B) = A \cdot B$. A carry-out has the possibility of being propagated (P) if either A or B is 1, which is written as $P(A, B) = A \oplus B$.

Thus, we can compute each of the carry out bits using the logic shown in figure 1:

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

Figure 2: equations to calculate the carry outs

We see that the CLA is able to calculate without waiting for the carryout bit of the previous adder as in the CRA. 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

However, observe that the logic equations above would be very difficult to implement as the number of bits get larger and larger, and this flat approach does not become very useful.

Thus, for the task of designing a 16 bit CLA, we break it down into a 4x4 hierarchical CLA instead. 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. The 4-bit CLA generates two additional output signals, the group propagate (PG) and the group generate (GG), with their logics being shown in the figure below.

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

Figure 3: Logical equations for P and G

We would then have 4 individual groups of CLA's combining to form a larger CLA, hence we have the hierarchical CLA. The block diagram for the hierarchical CLA as well as each individual CLA are shown in the figures below.

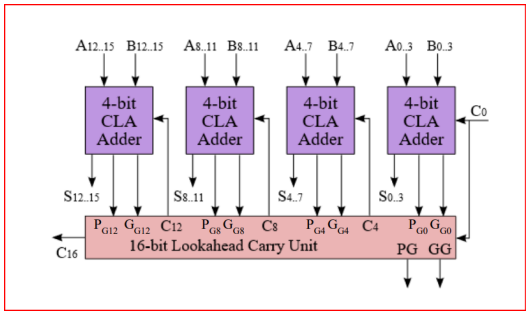

Figure 4: Top level block diagram

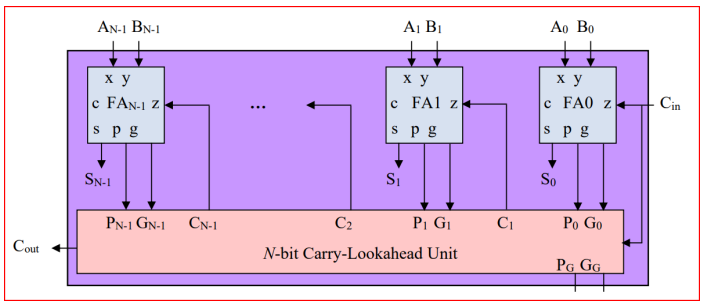

Figure 5: Each CLA block out of 4

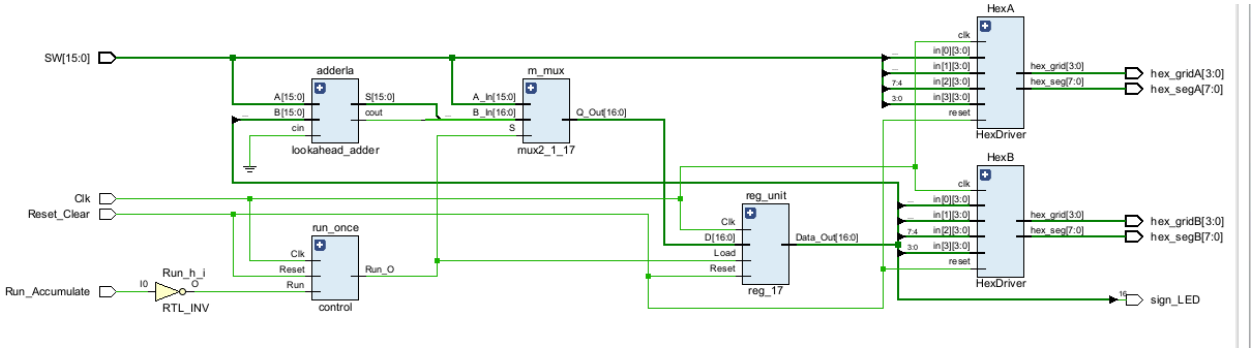

Figure 6: RTL block diagram for CLA

Carry Select Adder

Carry-Select Adder (CSA) is another way to speed up the computation. It consists of two full adders (or CRAs if multiple bits are grouped) and a multiplexor (MUX). 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 even before the carry in bit is fed into the system.

Once the actual carry in is fed in from the previous adder, the MUX will use the value of this carry in (0 or 1) to select the correct pre-computed result to deliver to the next stage.

A block diagram representing this is shown below.

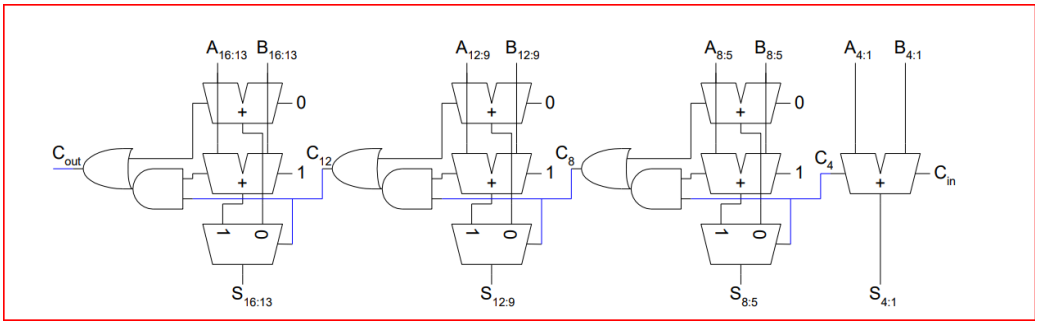

Figure 7: CSA block diagram

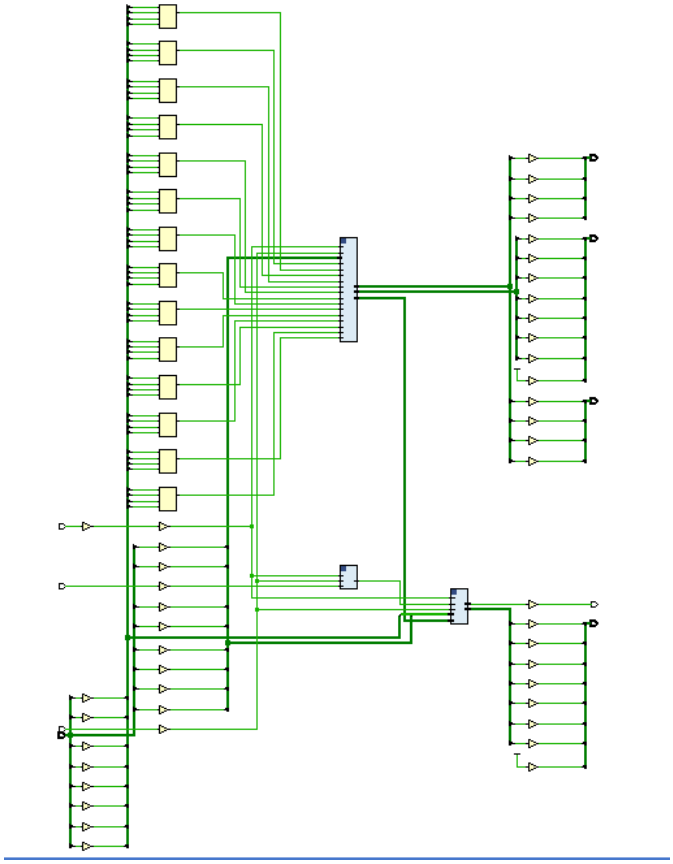

Figure 8: RTL Block Diagram for CSA

This approach speeds up the process because instead of the delay being related to each of the adders' delay in trying to calculate the carryout (and them being serially connected), all the carry out possibilities are computed at once, and the new overall delay of the circuit is based on the delay of the MUX select rather than the delay of the adders' competition time.

While we do take the approach of ripple adding each 4 bit block in this CSA implementation, this would introduce some delay into the overall processing speed, but it is still faster than having a 16 bit CRA or CLA.

Written Description of Systemverilog modules

Module: ripper_adder (ripple_adder.sv)

Inputs: Input [15:0] A, B,
input cin,

Outputs: output [15:0] S,
output cout
output logic A_out, B_out,

output logic [7:0] A,
output logic [7:0] B);

Description:
Top level unit for the CRA which takes the two 16 bit inputs A and B, splits both of them into groups of 4 bits each and calls 4 individual 4-bit ripple adders to compute S and Cout of the overall 16 bit CRA.

Purpose: To instantiate multiple 4 bit CRAs to be able to implement a 16 bit CRA that computes the sum of the and Cout.

Module: ripper_adder (.sv)

Inputs: Input [15:0] A, B,
input cin,

Outputs: output [15:0] S,
output cout
output logic A_out, B_out,
output logic [7:0] A,
output logic [7:0] B);

Description:
Top level unit for the CRA which takes the two 16 bit inputs A and B, splits both of them into groups of 4 bits each and calls 4 individual 4-bit ripple adders to compute S and Cout of the overall 16 bit CRA.

Purpose: To instantiate multiple 4 bit CRAs to be able to implement a 16 bit CRA that computes the sum of the inputs and Cout.

Module:fourbit_full_adder (.sv)

Inputs: input logic [3:0] A, B,
input logic c_in

Outputs: output logic [3:0] S,
output logic c_out

Description:
Top level unit for each four bit adder, which calls on 4 individual full adders to compute

the necessary results of S and c_out for one group of 4 bits of A added to one group of 4 bits of B.

Purpose: To instantiate 4 adders to be able to implement a 4 bit adder to compute the result for 4 bits of A and 4 bits of B.

Module: full_adder (.sv)

Inputs: input logic x, y, z,
Outputs: output logic s, c

Description:
One full adder that performs an add operation on 1 bit of A with 1 bit of B, and determines the carryout from the values in X, Y and Z, which are the two bits and carryin.

Purpose: To act as a single-bit full adder to be used and called back in the higher level modules which implement the 4 bit and 16 bit CRAs respectively.

Module: Select_Adder (.sv)

Inputs: input logic [3:0] A, B,
input logic c_in

Outputs: output logic [15:0] S,
output logic cout

Description:
Top level unit for the carry select adder which takes the two 16 bit inputs A and B, splits both of them into groups of 4 bits each and calls two four bit ripple adders for each four bits it calculates. One of these four bit adders would have a carry in of one while the other would have a carry in of 0. This allows for both possibilities to be taken into account, and the real carryin will simply make the decision to output one of the two answers.

Purpose: To instantiate 8 total 4 bit carry ripple adders with multiple muxes to allow for timing of addition to be shortened as compared to a normal ripple carry adder.

Module: adder_toplevel (.sv)

Inputs: input Clk, Reset_Clear, Run_Accumulate,
input [15:0] SW,
Outputs: output logic sign_LED,
output logic [16:0] Out
output logic [7:0] hex_segA,
output logic [3:0] hex_gridA,

output logic [7:0] hex_segB,
output logic [3:0] hex_gridB

Description: Top level module which declares local logical variables. Instantiate all the modules. Call Hexdriver to view the upper nibble of registers by converting digital logic outputs to integer outputs. Input synchronisers for asynchronous inputs (from the switches).

Purpose: Module encompasses the full logic of the rest of the module

Module: control (.sv)

Inputs: input logic Clk, Reset, Run,
Outputs: output logic Run_O

Description: Simple state machine to convert a pushbutton input to one clock cycle long event. Has 3 states encompassing the logic for this.

Purpose: To ensure the appropriate clock cycle is used for the adders

Module: mux2_1_17 (.sv)

Inputs: input S,
input [15:0] A_In,
input [16:0] B_In,
Outputs: output logic [16:0] Q_Out

Description: A general 2 bit input MUX that is able to output one of the two signals based

on the select bits given to it, which are programmed as unique cases in an always_comb bloc.
Purpose: To be called by the CSA whenever necessary to be able to select one of the two computed C outs

Module: reg_17 (.sv)

Inputs: input Clk, Reset, Load,
Input [16:0] D,
Outputs: output logic [16:0] Data_Out

Description: Similar to the MUX, a general module representing a register to store a 16 bit input and output it on the positive edge of the clock. We also have cases in the always_comb block where the value out is forced to 0 when Reset is pressed and the value out is made to the data input D when Load is pressed.
Purpose: To be used to load/reset the values and store them during the computation made by the other modules.

Module: lookahead_adder (.sv)

Inputs: input [15:0] A, B,
input cin,
Outputs: output [15:0] S,
output logic cout

Description: Top level module for the CLA that calls 4 individual CLAs and processes the logic for the carryouts based on P and G.
Purpose: To compute the sum and carryout based on the CLA implementation

Module: fourbit_cla (.sv)

Inputs: input logic [3:0] A, B,
input logic c_in,
Outputs: output logic [3:0] S,
output logic c_out
Description: Performs CLA computation for a group of 4 bits of A and B, using P, G and C

logic for each bit being added together. Instantiates 4 individual full adders to perform bitwise addition. Instantiated in the higher level module to be used 4 times for 16 bits.
Purpose: To compute the sum and carryout for a group of 4 bits out of 16.

Module: hex (.sv)

Inputs: input logic clk,
input logic reset,

input logic [3:0] in[4],

Outputs: output logic [7:0] hex_seg,
output logic [3:0] hex_grid

Description: Implements a 4-digit hexadecimal display driver by using a counter to cycle through 4 digits, and selecting the appropriate pattern based on the input nibbles
Purpose: To display hexadecimal numbers on the 4-digit 7-segment display

Module: nibble_to_hex (.sv)

Inputs: input logic [3:0] nibble,
Outputs: output logic [7:0] hex

Description: Takes asynchronous signals and holds them in a flip flop allowing for them to be implemented on a clock signal. This results in the signals being converted from asynchronous to synchronous signals.
Purpose: To eliminate asynchronous components such that the entire circuit remains synchronous to reduce the chance of static hazards arising from timing issues and ensure proper signal handling in FPGA designs.

Area, complexity and performance tradeoffs

It was theorized that the timings would result in the performance in increasing order from the CRA to the CLA and CSA, with CSA having the fastest computation time due to the pre calculated outputs, this with the trade off of having the largest implementation. In terms of area, notice that the CLA used 103 LUTs, followed by CSA using 92 LUTs and carry_ripple using 83 LUTs. This however was different to our actual implementation where the CLA had the best timing results as compared to the CSA and the CRA.

However, notice that for the CLA, a flat implementation would require too many gates due to subsequent equations over 4 bits of an adder being too complex logic wise to compute. Thus to simplify this, we trade off the timing complexity, breaking it into hierarchical CLAs.

For the CSA, while pre computing carryouts and then selecting the correct one may reduce the computing time, the delay is now based on the delay of the MUX select, and having multiple MUXes for larger CSAs would inevitably increase the delays as well, which would present the same problem as the CRA. Furthermore, the computation of both particular solutions results in the area of the circuit being the greatest, along with the large amount of muxes being used in between the different adders.

While the CRA is the simplest to implement, it simply does not make sense for a larger number of bits as the delay is serially added from each full adder which would introduce a large delay when dealing with 16 bits and when expanded to a larger design. This resulted in the delay being the longest.

Thus, each of the three implementations have their own benefits and tradeoffs which need to be considered.

Simulation Trace

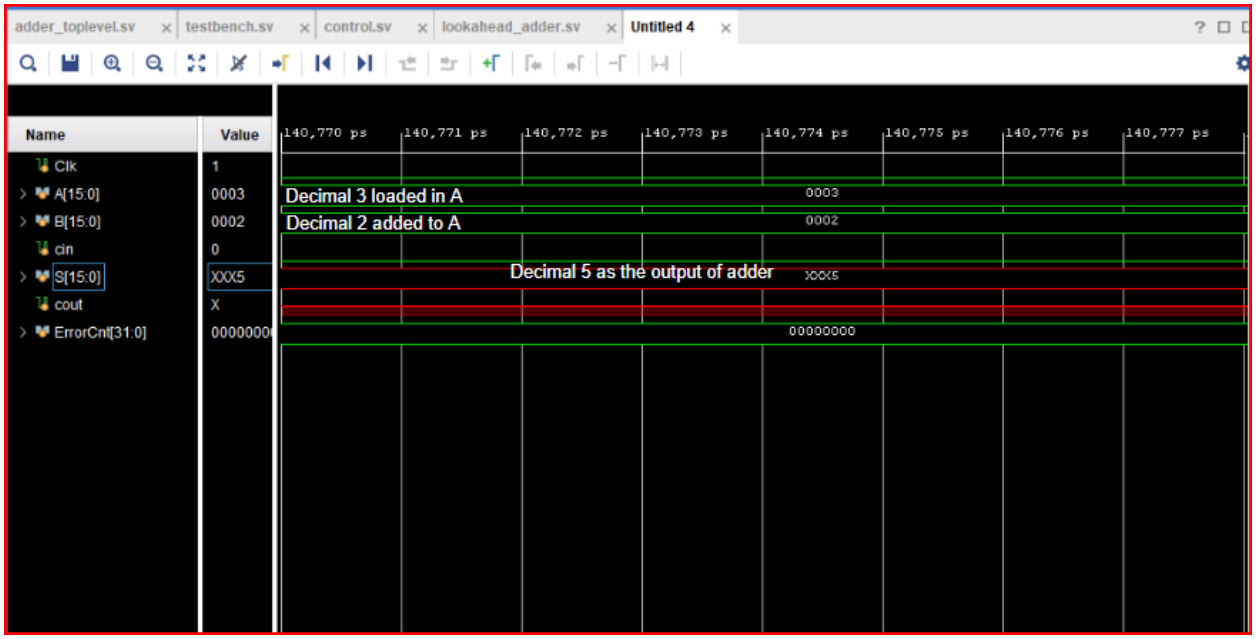

Figure 9: Annotated simulation trace

Answers to the post-lab questions.

1) Document design analysis for the three adders in the table below.
The frequency was calculated by using f = 1/(10 - wns). All other values were obtained on Vivado.

	Carry_Ripple	Carry-select	Carry-Lookahead
LUT	86	92	103
Frequency	0.149 Hz	0.309Hz	0.222 Hz
Total Power	0.095 W	0.093 W	0.092 W

After normalizing by dividing all values by the CRA values,, we obtain the following:

	Carry_Ripple	Carry-select	Carry-Lookahead
LUT	1	1.07	1.2
Frequency	1	2.07	1.48
Total Power	1	0.98	0.97

We plot a bar chart from these values to analyze the data.

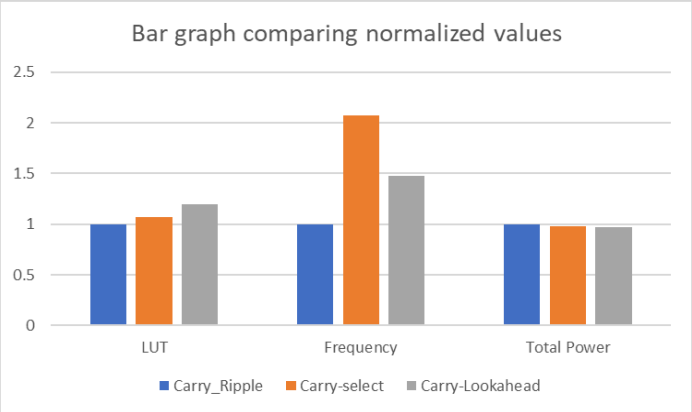

Figure 10: Bar chart plot for comparison

From the values that we obtained, the first key variable we observe is that the CLA uses the most LUTs, followed by CSA and then CRA. This makes sense as the CLA requires the most logical gates to implement, and thus has a higher area. For frequency, we see that the CSA and CLA are both significantly faster than the CRA, which holds true against the theoretical expectation of the speeds of the 3 adders. As for total power, all 3 of them consume similar amounts of power. This is likely because we use a very small fraction of the FPGA itself which does not allow us to see observable differences in power between them.

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?)

For the 4x4 hierarchy, the delay of the adder is the sum of 4 full adder delays, as well as 3 MUX select delays (since the first group of 4 bits does not require a MUX). While this might be faster than a ripple adder, it is still not the most ideal due to the combined delay of the MUX and adders.

To determine the ideal hierarchy, we would need to know the delay of a full adder and MUX specifically, to know whether it is a faster implementation to have more full adders or more MUXs. We would run experiments with different numbers of MUXs and full adders in each block under the hierarchy to determine the delay.

The most ideal hierarchy would be a variable-sized adder, which consists of block sizes of 2, 2, 3, 4 and 5. Generally, this is the best approach assuming the full-adder delay is equal to the MUX delay. The total delay would be 2 full adder and 4 MUX delays, which would be better than the 4x4 implementation.

3) Remaining variables

The frequency was calculated by using f = 1/(10 - wns). All other values were obtained on Vivado.

	CRA	CLA	CSA
LUT	86	103	91
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-flop	53	53	53
Frequency	0.149 Hz	0.309Hz	0.222 Hz
Static power	0.072 W	0.072 W	0.072 W
Dynamic power	0.023 W	0.021 W	0.020 W
Total power	0.095 W	0.093 W	0.092 W

4) Do a critical path analysis using the Vivado tools. Specifically, you should show the critical path for each adder

CRA

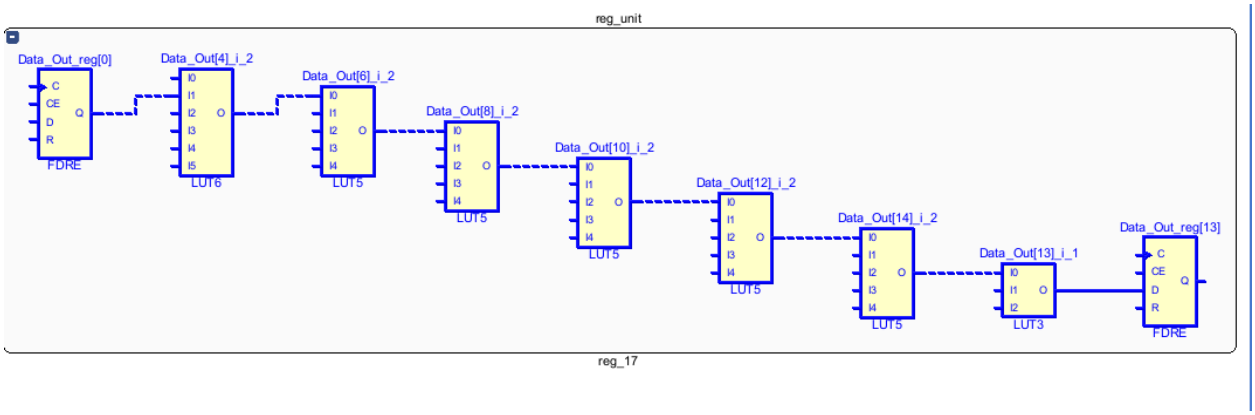

Figure 11: Critical path for CRA

CLA

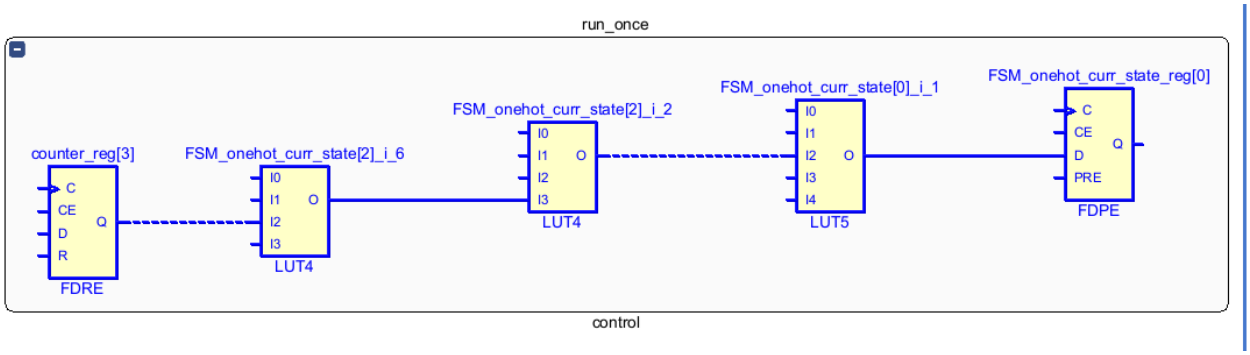

Figure 12: Critical path for CLA

CSA

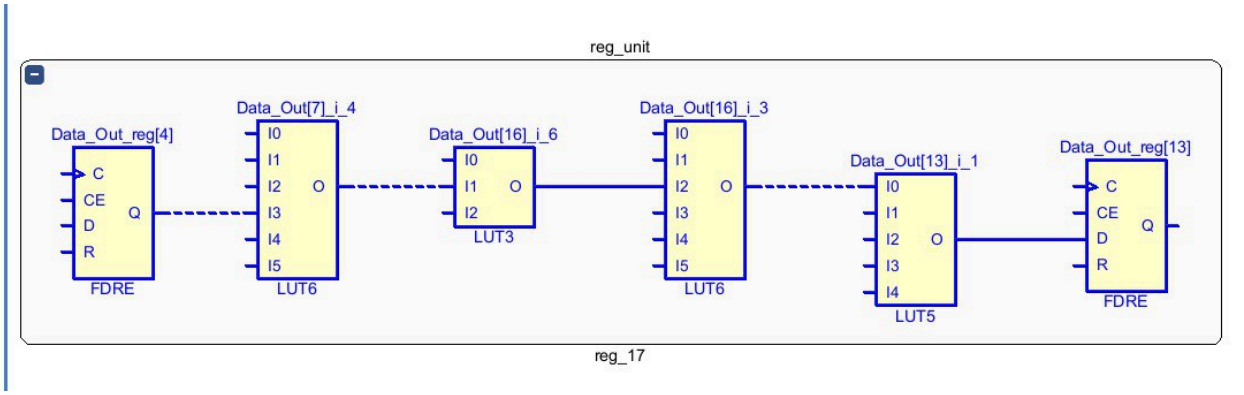

Figure 13: Critical path for CSA

The critical paths above are the longest path an output can take, and represents the worst case scenario of delay which we would want to use as an indicator of the delays. We see from the critical paths above that the CRA has the longest critical path trace, and the CLA and CSA, which matches our theoretical understanding that the CRA would have the longest delay.

The actual critical path through the CRA was as expected as in the theoretical critical path, having to traverse through each adder before finally arriving at cout, this involving 9 block modules. We can see that the critical path for the CLA had 5 modules to transverse as compared to the 6 modules in the CSA, this matching the timing differences in the CLA and CSA. However due to the differences in the design, we conclude that the CSA could be optimized to a much greater extent and this would not be the best implementation of the CSA as we describe above.

Conclusion

We experienced a bug where the implementation on the board was not working even though the synthesis and the implementation has no issues. We realized that this was due to the clock implementation being ambiguous where we did not create a clock constraint with 100 Mhz.

In conclusion, we implemented a 16 bit adder in 3 different ways, using CRA, CLA and CSA. We observed that the CLA and CSA are both significantly faster than the CRA, but also use up more gates and are more complex to implement. For a potential future experiment, we might want to figure out a better way to implement the CSA than the 4x4 design that we used in this lab, which clearly introduces more delays than we need with alternative designs.