ECE 385

Fall 2024
Experiment 3

Lab 3 Report

Yousef Khojah (yousefk2)
Mohammed Alnasser (ma138)

Introduction

In this lab, we constructed the design of three types of 16-bit adders: the ripple carry adder, the lookahead adder, and the carry select adder. This lab acts as a transition introduction to using SystemVerilog on Xilinx Vivado to implement on the FPGA board. It required some basic understanding of SystemVerilog's syntax and Vivado's analysis tools. The ripple adder is the simplest one of the three; it consists of N full adders, which is the 1-bit version of the binary adder. Each carryout from the full adder goes as a carry-in into the next full adder. The carry lookahead adder uses the carry-in with the inputs of every bit to make predictions about the carry-out. A carry-out is generated when both inputs are 1, or a carry-out could be propagated if either input is 1. The carry select adder uses two full adders and a multiplexer for each bit. One adder computes the carry-out assuming that the carry-in is one, and the other assumes it's zero. Then, the real carry-in selects the output of the desired full adders in all bits simultaneously. We will discuss each adder in this report regarding area, power, and maximum operating frequencies.

1. Adders:

Ripple Carry Adder

We built the ripple carry adder in a hierarchical manner, meaning that we implemented it using different layers of design. The first layer consists of the full-adder circuit. The second layer is a group of four full adders connected together, meaning that the carry-out (c) from the first full adder would be connected to the carry-in (z) of the second full adder and so on until the fourth full adder. The third layer is four groups of four full adders, as shown in Figure 2, where the carry-out from the first four group of full adders is connected to the carry-in of the second group of full adders and so on, giving a total of 16 full adders, allowing for 16-bit addition. The negative aspect of this design is that each full-adder cell needs to wait for the carry-out of the previous full-adder cell, which costs a lot of time as we want to add larger numbers. On the other hand, the positive aspect is that it's a simple design that uses minimum resources to implement.

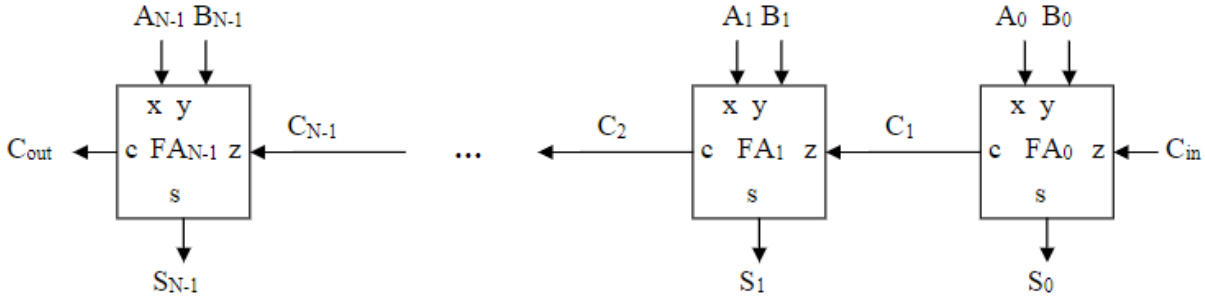

Figure 1: Ripple Carry Adder block diagram

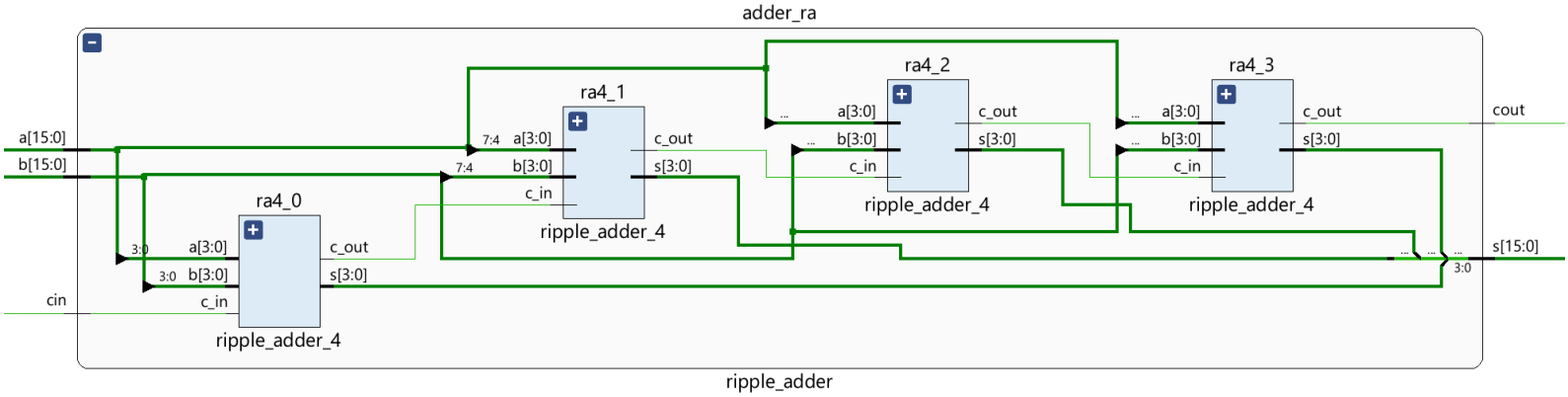

Figure 2: Ripple Carry Adder block diagram from Vivado where each ra4 cell contains 4 full adders.

Carry Lookahead Adder

The carry lookahead adder, as seen in Figure.3, uses the two signals Generated (G) and Propagated (P) to calculate the carry-out of each full adder depending on the available inputs A and B. A carry-out is generated whenever both A and B are 1, no matter what the carry-in is as follows: **G(A,B)=AB**. Similarly, the previously calculated carry-out is propagated if either A or B is 1, **P(A,B)=A^B**. Therefore, the carry-out is calculated by $C_{out\ i+1} = G_i + (P_i\ C_i)$. This way, the carry-ins of each bit don't depend on the carry-out of the previous sum, which makes this design faster than the ripple adder. However, building an N-bit lookahead adder will make the carry-in expressions very complex, using many logic gates. So, in order to make this design work, it should be constructed in a hierarchical fashion. We are using a 4x4 hierarchical design for our 16-bit inputs, as seen in Figure.4. The expressions for the four full adder carry-outs are as follows:

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

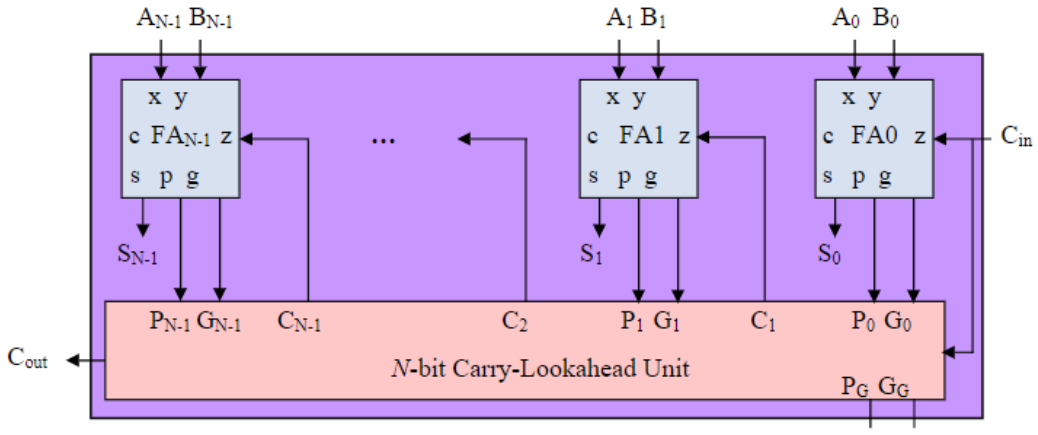

Figure.3: Block diagram of a 4-bit group of the carry-lookahead adder

After that, each 4-bit group will output two signals that will be used to calculate the carry-in for the next 4-bit group propagate (P_G) and group generate (G_G), and their logic is:

$$\mathbf{P_G = P_0\ P_1\ P_2\ P_3}$$

$$\mathbf{G_G = G_3 + G_2\ P_3 + G_1\ P_2\ P_3 + G_0\ P_1\ P_2\ P_3}$$

Denoting the P_G and the G_G from every 4 bits as $P_{G0}\ P_{G4}\ P_{G8}\ P_{G12}$ and $G_{G0}\ G_{G4}\ G_{G8}\ G_{G12}$ to use them in calculating the carry-ins for each 4-bit group, instead of just connecting the fourth bit's carry-out to the next first carry-in, which defeats the purpose of speeding up the process. The expressions for the carry-ins of the 4-bit group are as follows:

$$\mathbf{C_4 = C_0\ P_{G0} + G_{G0}}$$

$$\mathbf{C_8 = C_0\ P_{G0}\ \ P_{G4} + G_{G0}\ P_{G4} + G_{G4}}$$

$$\mathbf{C_{12} = C_0\ P_{G0}\ \ P_{G4}\ \ P_{G8} + G_{G0}\ P_{G4}\ P_{G8} + G_{G4}\ P_{G8} + G_{G8}}$$

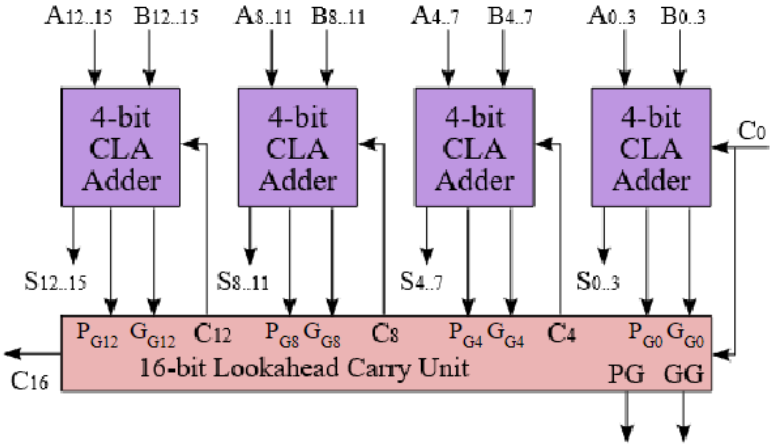

Figure.4: High-level block diagram of hierarchical design of the carry-lookahead adder

Carry Select Adder

Just like the previous adders, we implemented the Carry Select Adder in a hierarchical manner. The first two layers are very similar to the ripple carry adder, with the first layer consisting of a full-adder circuit and the second layer consisting of a group of four full-adders tied together, let's call one group of four-full adders a Ripple Adder cell (RA).

The third layer is where things get different from the ripple adder. In this third layer, we have 7 RA cells and a few muxes. The first RA cell would have the first four bits of the numbers we want to add and the carry-in connected to it. Furthermore, three RA cells would compute the addition of the remaining 12 bits of the two numbers with the assumption that the carry-in for each RA cell is zero, while the other three RA cells would compute the exact same addition. However, the assumption here is that there is one carry-in for each RA cell. Then, the output of each pair of RA is connected to a 2:1 mux. Moreover, the carry-out of each pair of RA cells is connected to a 2:1 mux to determine the select signal for the 2:1 mux that has the output of each RA cell connected to it, as shown in Figure.5.

The reason this approach is way faster than the Ripple Carry Adder, for example, is that once the first RA cell gets the carry-in, the whole 16-bit addition is available after a fixed amount of delays. This is because we compute the addition of the two 16-bit numbers A and B in a parallel manner, meaning that each RA cell does not need to wait for the previous RA cell's carry-out because it is assuming that the carry-in is either zero or one, corresponding to the top and bottom RA groups respectively as shown in Figure 5. The fixed delay mentioned earlier comes from the RA cell itself; since the RA cell consists of 4 full adders connected to each other, each full adder would need to wait for the carry-out of the previous full adder in order to compute the addition, except the first full adder cell.

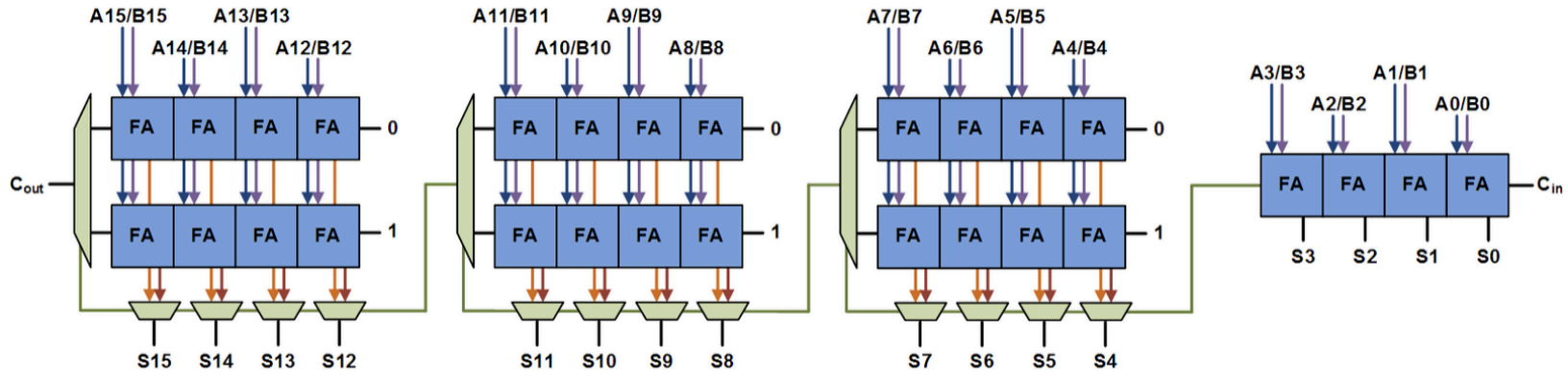

Figure 5: Carry Select Adder block diagram[1]

[1] https://en.wikipedia.org/wiki/Carry-select_adder

2. .SV Modules:

- **adder_toplevel.SV**
  Input: clk, reset, run_i, [15:0] sw_i
  Output: sign_led, [7:0] hex_seg_a, [7:0] hex_seg_b, [3:0] hex_grid_a, [3:0] hex_grid_b
  Purpose: This is the top-level module where all instantiations and internal wiring between modules happen. To choose which adder to test, we commented the other adders insantiation.

- **ripple_adder.SV**
  Input: [15:0] a, [15:0] b, cin
  Output: [15:0] s, cout
  Purpose: This module is the ripple adder. It takes the inputs and calculates their sum in a rippling manner, taking each carry-in from the previous carry-out.

- **lookahead_adder.SV**
  Input: [15:0] a, [15:0] b, cin, [3:0] PG, [3:0] GG
  Output: [15:0] s, [3:0] C, PGF, GGF, out
  Purpose: This module implements the lookahead adder discussed in the lab document. The adder is designed in a 4x4 hierarchy, which is implemented using the signals P, G, PG, and GG.
- **selecet_adder.SV**
  Input: [15:0] a, [15:0] b, cin
  Output: [15:0] s, [15:0] s0, [15:0] s1, cout
  Purpose: This module implements the Carry Select Adder described earlier in this lab. It uses two ripple carry adders and 2:1 muxes.
- **fa.SV**
  Input: a, b, c_in
  Output: s, c_out
  Purpose: This module implements the Full-Adder circuit. It takes inputs a, b, and c_in and outputs s and c_out. It computes the sum of the inputs and outputs it in s and c_out.

- **negedge.SV**
  Input: clk, in
  Output: out
  Purpose: This module takes input ‘in’ and outputs ‘out’ on the falling edge of the clock.

- **sync_debounce.SV**
  Purpose: This file had multiple sync_debounce modules for each button used on

the FPGA. The purpose of this is to debounce switches since mechanical switches cause glitches during the switch due to contact bounce.

- **hex_driver.SV**
  Input: clk, reset, [3:0] in[4]
  Output: [7:0] hex_seg, [3:0] hex_grid
  Purpose: This file is responsible for formatting and converting an input to be displayed on the LED so that each LED corresponds to one hexadecimal. No changes were made to it.

- **load_reg.SV**
  Input: clk, reset, load, [DATA_WIDTH-1:0] data_i
  Output: [DATA_WIDTH-1:0] data_q
  Purpose: This module implements a 17-bit register with parallel load and synchronous reset.

Area and Tradeoffs:

- **Carry Ripple Adder:**

The ripple adder is the simplest and most straightforward adder to implement. It doesn't require much logic compared to other implementations. However, this adder has a high delay because to implement N bits, you would have to wait for N full adder delays since each full adder waits for the previous one to finish computation to take the carry-in.

- **Carry Lookahead Adder:**

The lookahead adder has the fastest implementation compared to the other three adders because it predicts the carry-ins without waiting for previous full adders. However, this fast implementation comes with a relatively more complex design with much more logic, which impacts the debugging and building costs.

- **Carry Select Adder:**

The carry select adder is the middle ground between the previous two adders. It has a fast implementation compared to the ripple adder because it computes the sum with the carry-in as 1 and 0 and then chooses the actual value. Moreover, this method has a simpler implementation compared to the lookahead but not as fast.

Post-lab Questions:

Adders Performance

The table below shows an analysis of the three implemented adders by performance. As shown in Figure.6, the three adders are compared in Look Up Tables (LUT) used, maximum Frequency, and the total power used. The maximum frequency is calculated using the equation:

$$Fmax = \frac{1}{(T + WNS)}$$

Where T is the period of the clock cycle, which is 10ns, and WNS is the Worst Negative Slack.

	WNS (ns)	Fmax(MHz)	LUT	Total Power (W)
CRA	**2.857**	**77.78**	**88**	**0.08**
CLA	**5.363**	**65.09**	**111**	**0.079**
CSA	**3.976**	**71.55**	**95**	**0.079**

Table.1 : Performance data for each adder

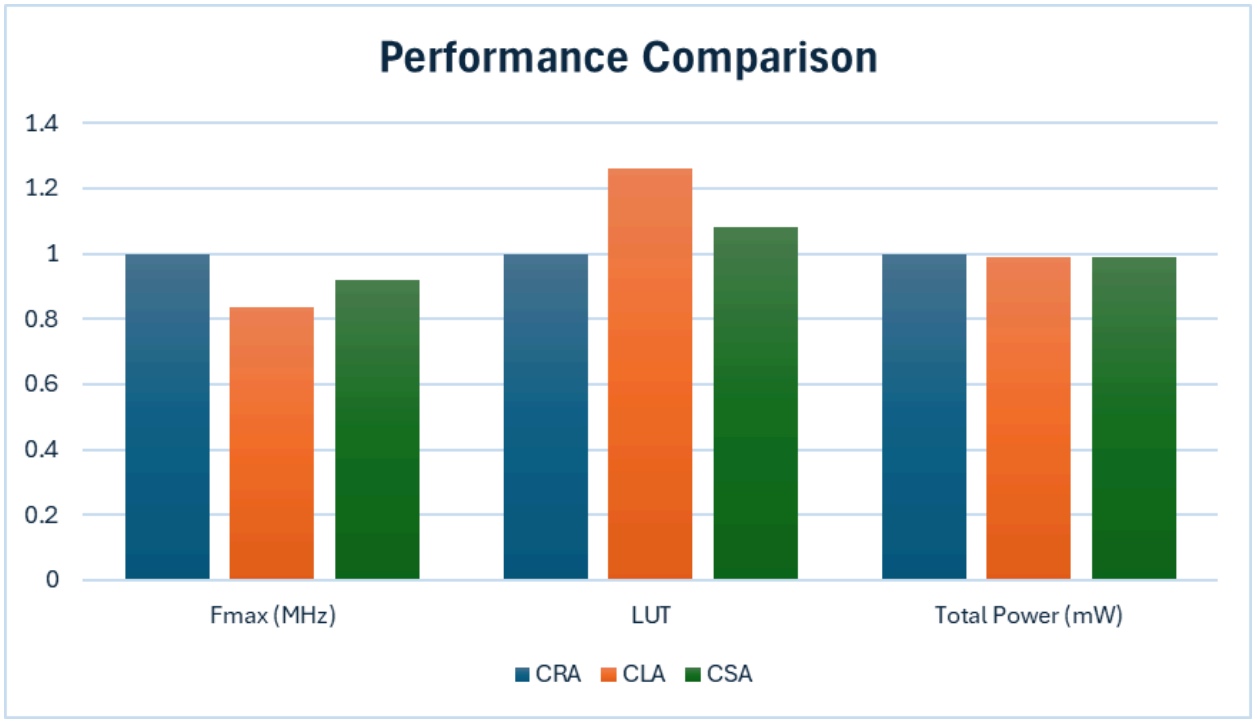

Figure.6: Performance comparison bar graph between the three adders

The above outcomes all align with what our experiment and design expected to output. Because the CRA has simpler, fewer used chips than the CLA and CSA, it uses fewer LUTs overall. However, because it operates at a higher frequency than the other two, the CRA is slower and less efficient than the other two. The CRA also uses more power than the other adders, which is predicted given the poorer performance of the CRA.

Performance Metrics

	Carry Ripple	Carry Select	Carry Lookahead
LUT	**88**	**95**	**111**
DSP	**0**	**0**	**0**
Memory (BRAM)	**0**	**0**	**0**
Flip-Flop	**90**	**90**	**90**
Freq. Static (MHz)	**77.78**	**71.55**	**65.09**
Static Power (mW)	**71**	**71**	**71**
Dynamic Power (mW)	**9**	**8**	**8**
Total Power (mW)	**80**	**79**	**79**

Table.2: Performance metrics of the three adders

Annotated Simulation Trace

The simulation time trace below displays a sample simulation conducted using a basic test bench to verify the adders' operation. The operation we tested was 96 + 9, as indicated by the graph's top two lines. On the fourth line of the graph, output S indicates the operation's outcome, which is that the addition yielded a result of 105.

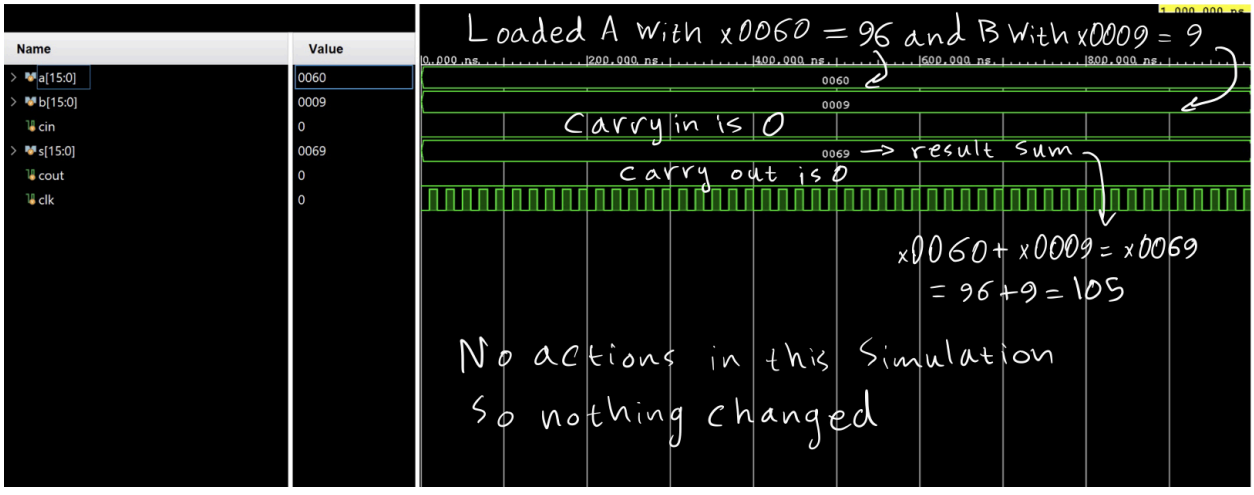

Figure.7: Annotated Simulation Trace displaying the test result of x0060 + x0009

Critical Path Analysis

Carry Ripple Adder

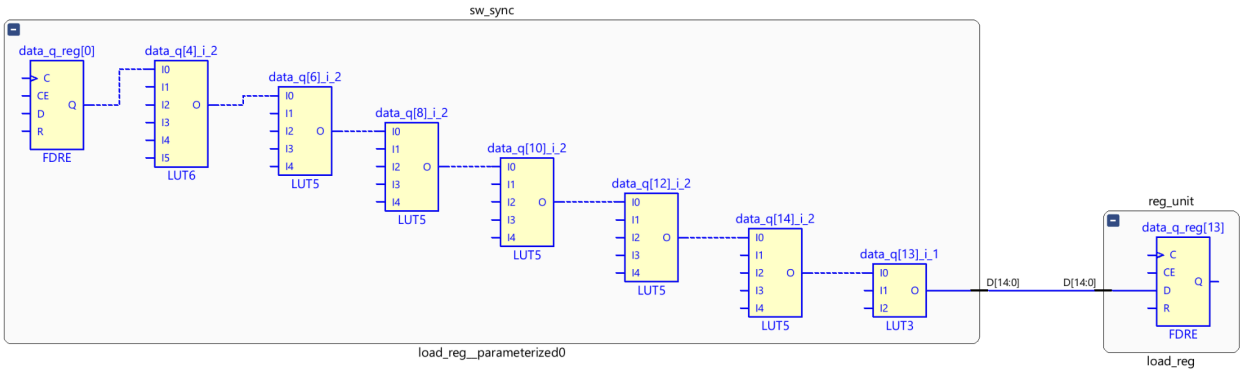

Figure.8: Critical Path for the Ripple Carry Adder, with a Slack of 2.857

The critical path of the Ripple Carry Adder (CRA) reveals the key limitation of this design: the carry-out of each full adder must propagate sequentially to the next full adder. This creates a long chain of dependencies from the least significant bit (LSB) to the most significant bit (MSB), leading to increased propagation delay. The Slack value of 2.857 ns indicates that there is a moderate amount of time before the circuit fails to meet the clock timing requirements, but the delay remains significant due to the cumulative carry propagation.

Carry Lookahead Adder

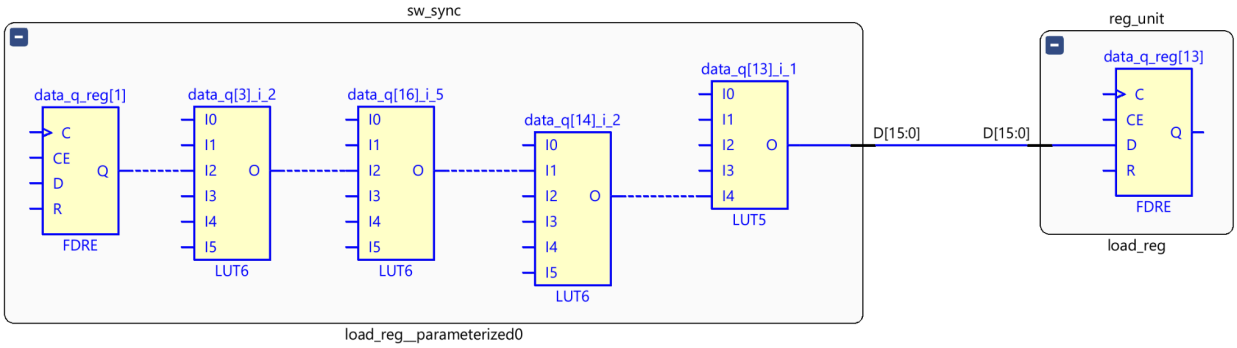

Figure.9: Critical Path for the Carry Look-ahead Adder, with a Slack of 5.363

The critical path of the Carry Lookahead Adder (CLA) shows a significant reduction in delay compared to the CRA. This is achieved by parallelizing the carry computation using Generate (G) and Propagate (P) logic, allowing carry-in values to be predicted rather than waiting for the previous bit's carry-out. Despite the additional complexity in the logic, the critical path is shorter, and the Slack value of 5.363 ns reflects the higher speed potential of this design due to the faster carry computation.

Carry Select Adder

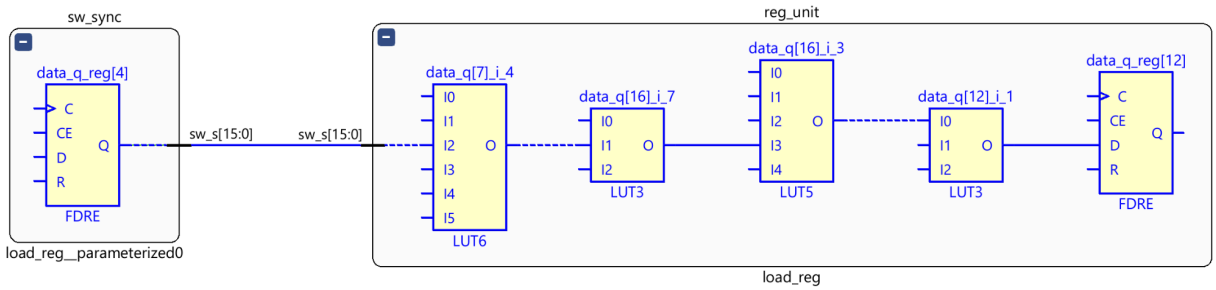

Figure.10: Critical Path for the Select Carry Adder, with a Slack of 5.363

The critical path of the Carry Select Adder (CSA) highlights its balance between speed and complexity. In this design, the carry-in values for each group of adders are precomputed for both carry-in states (0 and 1) and selected by a multiplexer. This reduces the carry propagation delay compared to the CRA, but it does not achieve the same speed as the CLA. The Slack value of 3.976 ns indicates a moderate improvement in speed, thanks to the parallel carry calculation across groups of bits, but with less complexity than the CLA.

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?)

No, even though the 4x4 block size provides a consistent approach, carry propagation time, a crucial component of adder performance, may not be fully optimized for this design.

The propagation delay resulting from the carry operation would be taken into account in the ideal hierarchy for a CSA on an FPGA. If all the blocks have the same size, there may be a performance bottleneck since each block's computation is dependent on the carry-out value of the previous block. Changing the block sizes within the CSA hierarchy could potentially address this issue by resulting in a non-uniform block structure that adjusts to the carry propagation delay.

For example, we can optimize the computation time by starting with a single bit and expanding to larger blocks (two, three bits, etc.) as we traverse deeper into the adder chain. Because they require fewer operations to compute their carry-out values, smaller blocks at the beginning enable the carry signal to spread to following blocks more quickly. On the other hand, because they have more intricate calculations to complete, larger blocks that come later in the sequence may afford to wait a little bit longer for the carry-in value.

By synchronizing each block's computation time with the expected carry propagation time, this gradual hierarchy seeks to decrease latency and increase the adder's overall speed, creating the ideal configuration.

Conclusion

In this lab, we implemented and analyzed three 16-bit adder designs: Carry Ripple Adder (CRA), Carry Lookahead Adder (CLA), and Carry Select Adder (CSA). Each design had distinct trade-offs: the CRA was simple and resource-efficient but slow due to sequential carry propagation, the CLA was the fastest by predicting carry-ins in parallel but required more logic complexity and resources, and the CSA provided a balance between speed and complexity by precomputing sums for both carry-in values. Performance analysis confirmed that the CLA achieved the highest speed, while the CRA consumed more power due to slower operation. Thankfully, we did not encounter any bugs. As for the lab manual, we both think that the provided material combined with the lecture was just enough for us to start the lab. The only suggestion we would make is maybe to provide a better diagram for the CSA. Overall, the lab was nice.