ECE 385

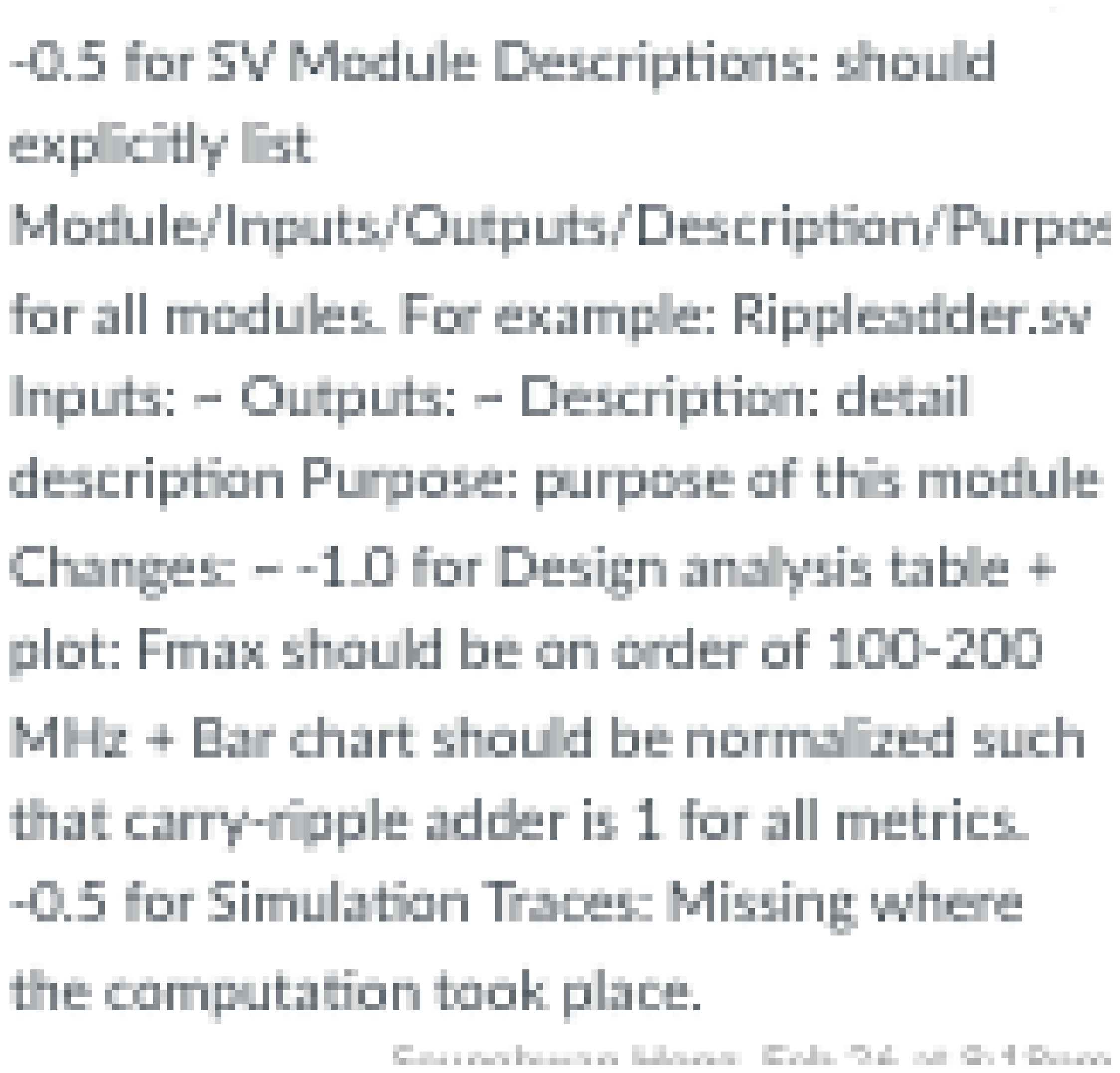
Spring 2024

Experiment 3

Lab 3 Report

Ziad AlDohaim (ziada2)
Mohammed Alnemer (mta8)

Table of Contents



-0.5 for SV Module Descriptions: should explicitly list Module/Inputs/Outputs/Description/Purpos for all modules. For example: Rippleadder.sv Inputs: -- Outputs: -- Description: detail description Purpose: purpose of this module Changes: -- -1.0 for Design analysis table + plot: Fmax should be on order of 100-200 MHz + Bar chart should be normalized such that carry-ripple adder is 1 for all metrics. -0.5 for Simulation Traces: Missing where the computation took place.

[illegible]

Introduction

The Lab 3 experiment explores three types of adders: the Ripple Carry Adder, the Carry Lookahead Adder, and the Carry Select Adder, each utilizing the full adder as a fundamental component. A full adder is a fundamental component in digital circuits designed to perform the addition of three binary digits, typically referred to as A, B, and Cin (carry-in).The full adder outputs two values: the sum (S) and the carry-out (Cout). The sum is the result of the XOR operation applied to A, B, and Cin, representing the bitwise addition, while the carry-out indicates whether there was an overflow out of the bit being added, which becomes the carry-in for the next bit in a sequence of additions. This carry-out is determined through a combination of

AND and OR operations on A, B, and Cin. Through this lab, we aim to understand how the three adders work, their efficiency, and application.

Ripple Carry Adder

The Ripple Carry Adder (CRA) is a binary adder design that performs addition. It consists of a series of full adders, each handling a single bit of the two numbers being added, along with a carry from the previous bit.

The full adder takes three inputs: two binary digits from the numbers being added (A and B) and a carry-in (Cin), and it produces two outputs: a sum (S) and a carry-out (Cout). The carry-out from each full adder is connected to the carry-in of the next higher-order full adder in the series, forming a 'ripple' of carry operations, hence the name, from the least significant bit to the most significant bit.

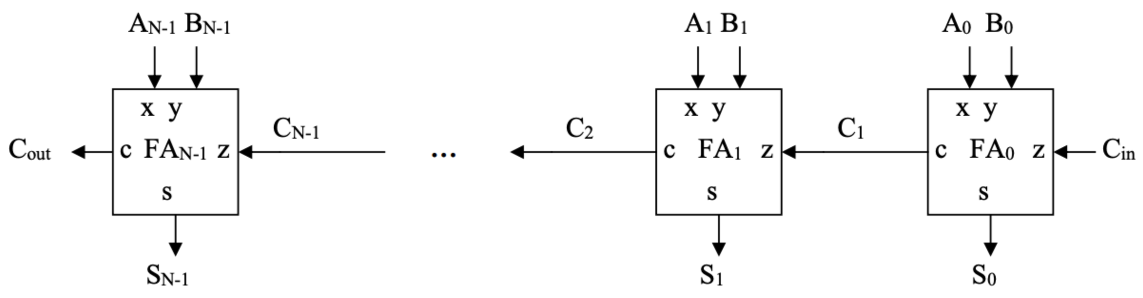


Figure 1: Ripple Carry Adder Diagram showing the connection between them

This linear connection of full adders means that the Ripple Carry Adder operates in a sequence, with each full adder waiting for the carry-out from the previous one before it can complete its operation. This sequential dependency creates a critical path through all the full adders, resulting in a propagation delay that increases linearly with the number of bits. Consequently, the main drawback of the Ripple Carry Adder is its long computation time, especially as the number of bits increases, making it less suitable for high-speed applications where quick arithmetic operations are necessary, making its runtime O(n).

Despite its simplicity and ease of implementation, the Ripple Carry Adder's performance is limited by the delay in carry propagation.

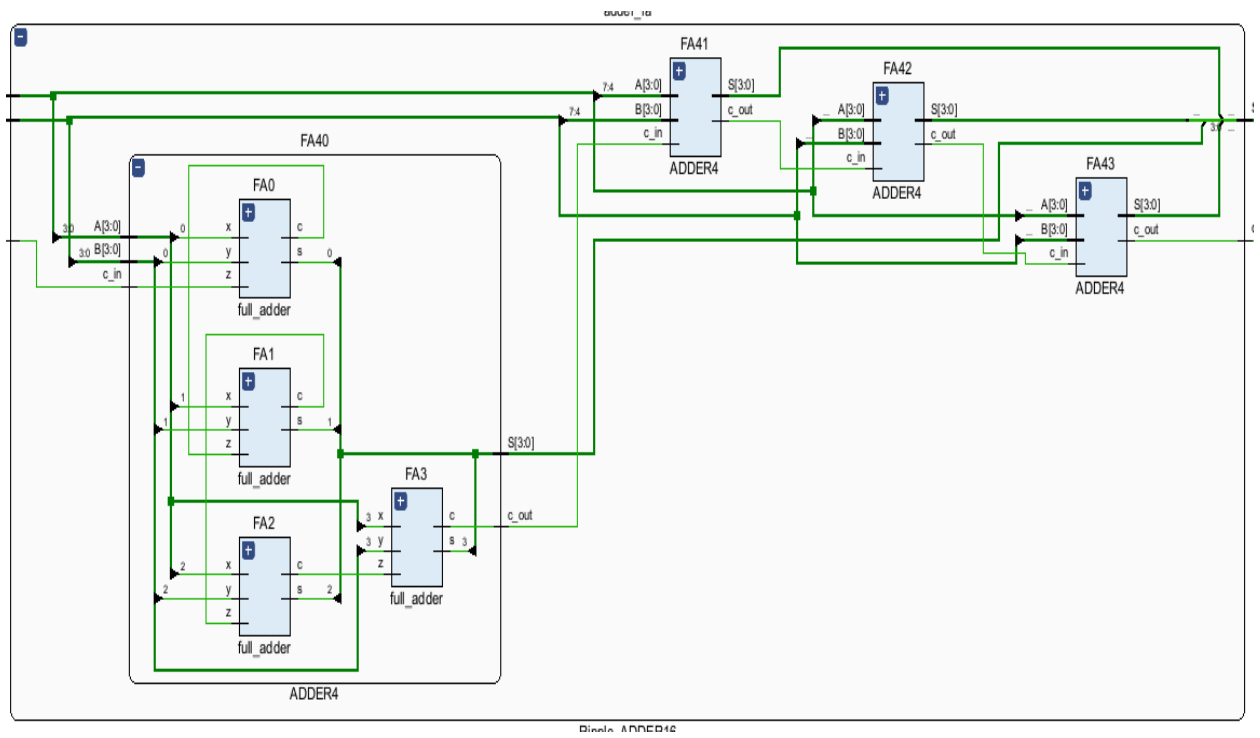


Figure 2: CRA RTL Design from Vivado

Carry Ripple adder four ADDER4 blocks each containing four individual full adder as shown by the expanded first ADDER4 module.

Carry Lookahead Adder

The Carry-Lookahead Adder (CLA) improves the speed of binary addition by reducing the time it takes to determine the carry bits. The CLA calculates carry bits in advance using generate and propagate, to put it simply, it ‘predicts’ whether an adder will need a carry in or not. Through these two key concepts:

Propagate (P): This indicates that a carry input will be passed through to the carry output if it is present. Mathematically, for a single bit, this is determined by P = A XOR B, A propagate condition occurs if either A or B is 1, implying that the carry input will influence the carry output.
Generate (G): This indicates that the operation will generate a carry output irrespective of the carry input. Mathematically, G=A AND B. A generate condition occurs if both A and B are 1, guaranteeing a carry output.

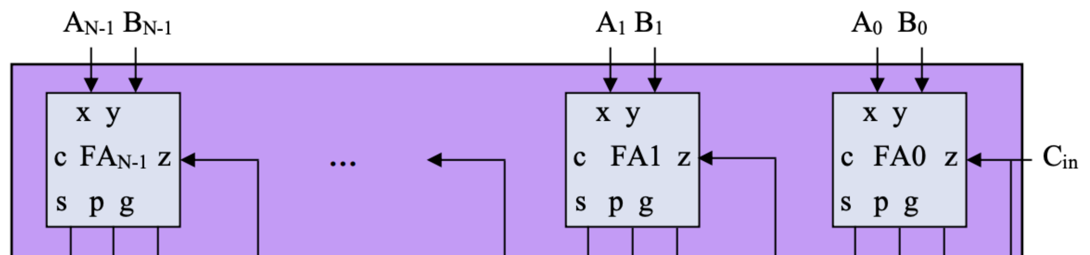


Figure 3: Block diagram inside a single CLA

To manage hardware limitations and avoid the slowdown associated with flat design across many bits, the CLA is structured hierarchically. The 4-bit CLA is the base unit (Figure 2), which can be expanded to 16 bits and beyond using a hierarchical approach. Which is done using the same concept of propagate and generate, but grouped together.

PG (Propagate Group) indicates if a carry input to the group will propagate through to the carry output. It is 1 if all individual propagate signals within the group are true using the equation
PG = (P3 AND P2 AND P1 AND P2)

GG (Generate Group) indicates if any bit within the group generates a carry that is propagated to the output. It is calculated based on the individual generate and propagate signals within the group GG =

$$(G_{n-1} + P_{n-1}G_{n-2} + P_{n-1}P_{n-2}G_{n-3} + P_{n-1}P_{n-2}P_{n-3}G_{n-4})$$

The hierarchical design allows the CLA to achieve a time complexity of $Ologn$ for carry computation. By organizing the adder into 4-bit CLA blocks and using PG and GG signals for larger groups, the CLA avoids the linear delay associated with the carry propagation in Ripple Carry Adders.

The hierarchical CLA does not rely on the traditional carry-out signals (Cout) of each bit directly. Instead, it computes the carry inputs for each block based on the PG and GG signals from previous blocks, leading to a faster overall operation by preemptively determining carry bits across multiple levels of the hierarchy.

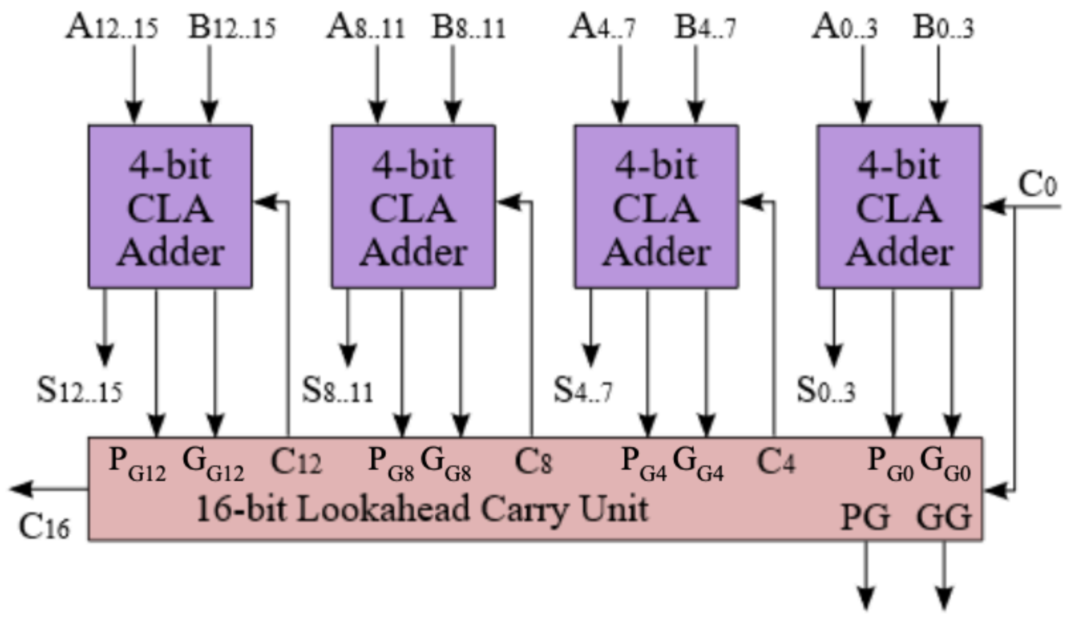


Figure 4: 4 CLA chained together.

The Cin for every 4 blocks, are calculated using the following logic:

$$\begin{aligned}
&\mathrm{C}_0 = Cin\\
&\quad \mathrm{C}_4 = GG_0 + C_0 \cdot P_{G0}\\
&\quad \mathrm{C}_8 = GG_4 + GG_0 \cdot P_{G4} + C_0 \cdot P_{G0} \cdot P_{G4}\\
&\quad \mathrm{C}_{12} = GG_8 + GG_4 \cdot P_{G8} + GG_0 \cdot P_{G8} \cdot P_{G4} + C_0 \cdot P_{G8} \cdot P_{G4} \cdot P_{G8}\\
&\quad \mathrm{C}_{out} = GG_{12} + (GG_8 \cdot PG_{12}) + (GG_4 \cdot PG_{12} \cdot PG_8) + (GG_0 \cdot PG_{12} \cdot PG_8 \cdot\\
&PG_4) + (C_0 \cdot PG_{12} \cdot PG_8 \cdot PG_4 \cdot PG_0);
\end{aligned}$$

Which can be seen in the RTL design below Figure 5:

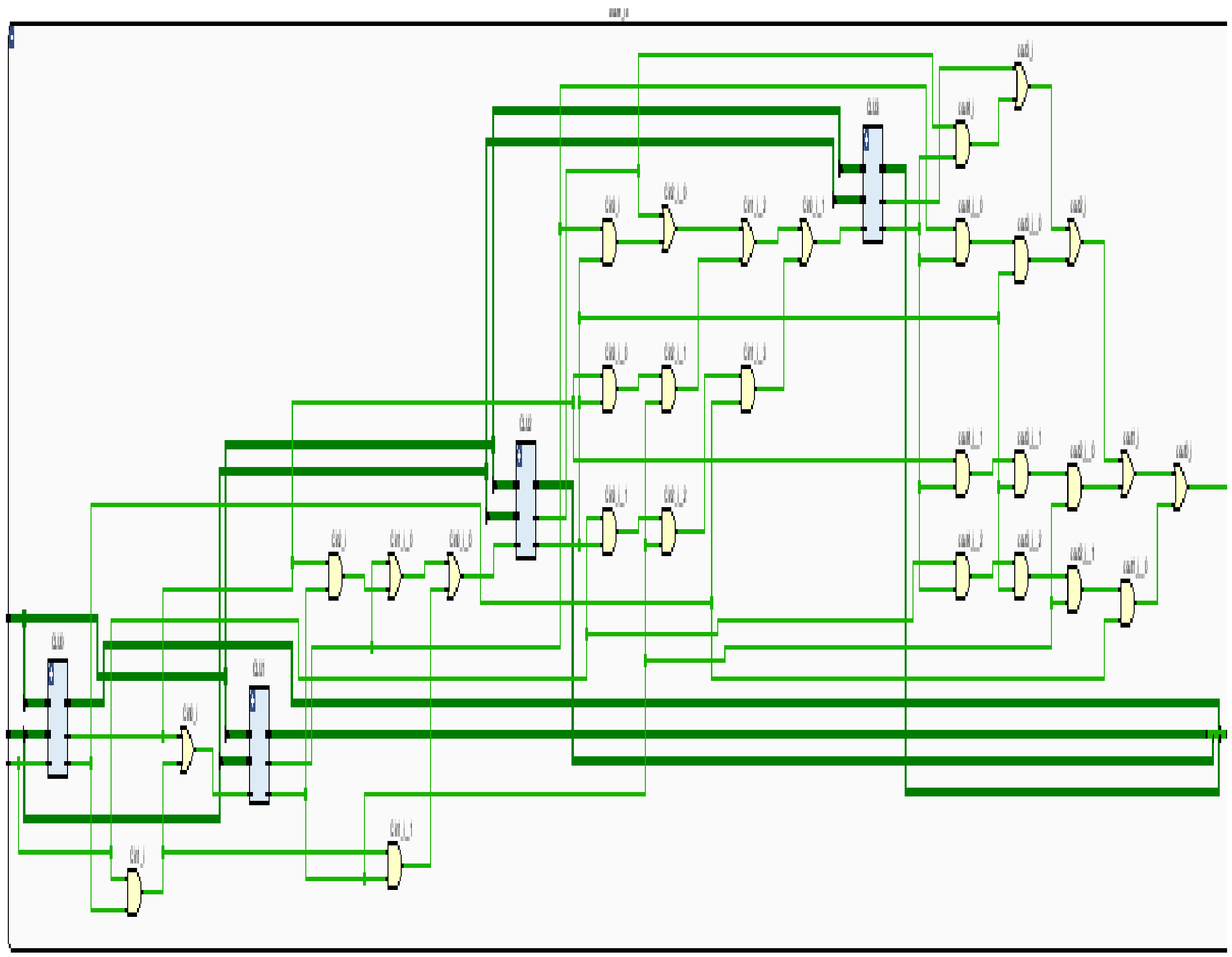


Figure 5: RTL Synthesis for CLA on Vivado

Carry Select Adder

The Carry Select Adder (CSA) accelerates the addition process by calculating two potential results for each block of bits at the same time, selecting the correct outcome based on the actual carry-in value from the block before it. This approach utilizes parallel computation to reduce the ripple delay characteristic of Carry Ripple Carry Adders (CRAs).

For each 4-bit segment of the inputs A and B, the CSA uses two sets of Carry Ripple Adders. One set assumes that the carry-in (Cin) is 0, and the other assumes Cin is 1. This dual approach precomputes the sums and carry-outs for both possible carry-in values.
Once the actual carry-in value for a block is determined (from the carry-out of the previous block), a multiplexer selects between the two precomputed outcomes. If the carry-in is 0, the result from the set with Cin = 0 is chosen; if it is 1, the result from the set with Cin = 1 is selected.

Although you have to consider that for the first 4-bit group, since there's no previous block to provide a carry-in, the CSA can always use the result assuming a Cin of 0, further optimizing the computation start. The design is illustrated in Figure 4:

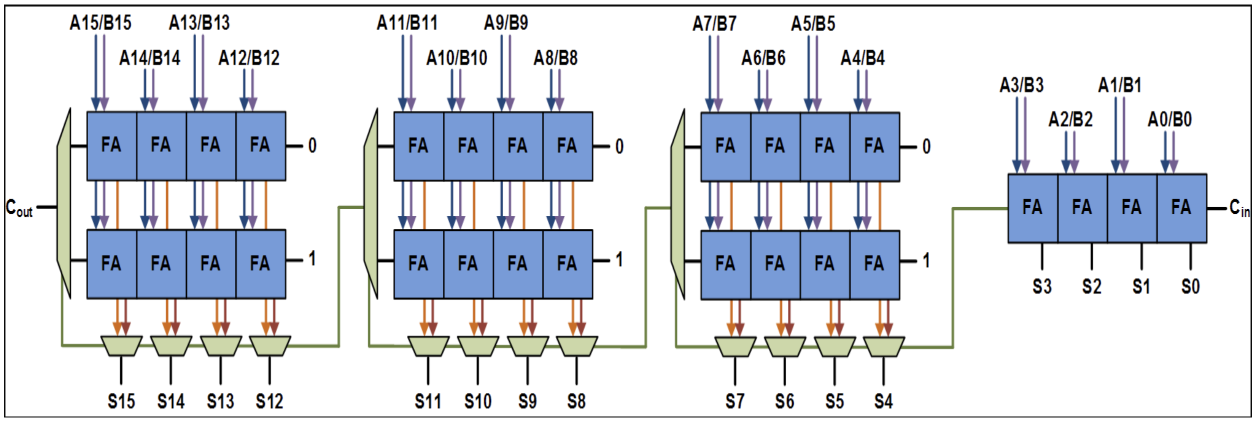


Figure 6: Block Diagram of the whole CSA circuit[1]

By simultaneously calculating the outcomes for both potential carry-in values, the CSA avoids the delay caused by carry propagation through the adder chain. Calculating both possibilities all at once allows for a faster computation time despite the additional logic (multiplexers) required for selection. Making its runtime O(logn).

[1] https://upload.wikimedia.org/wikipedia/en/3/36/Carry-select-adder-fixed-size.png

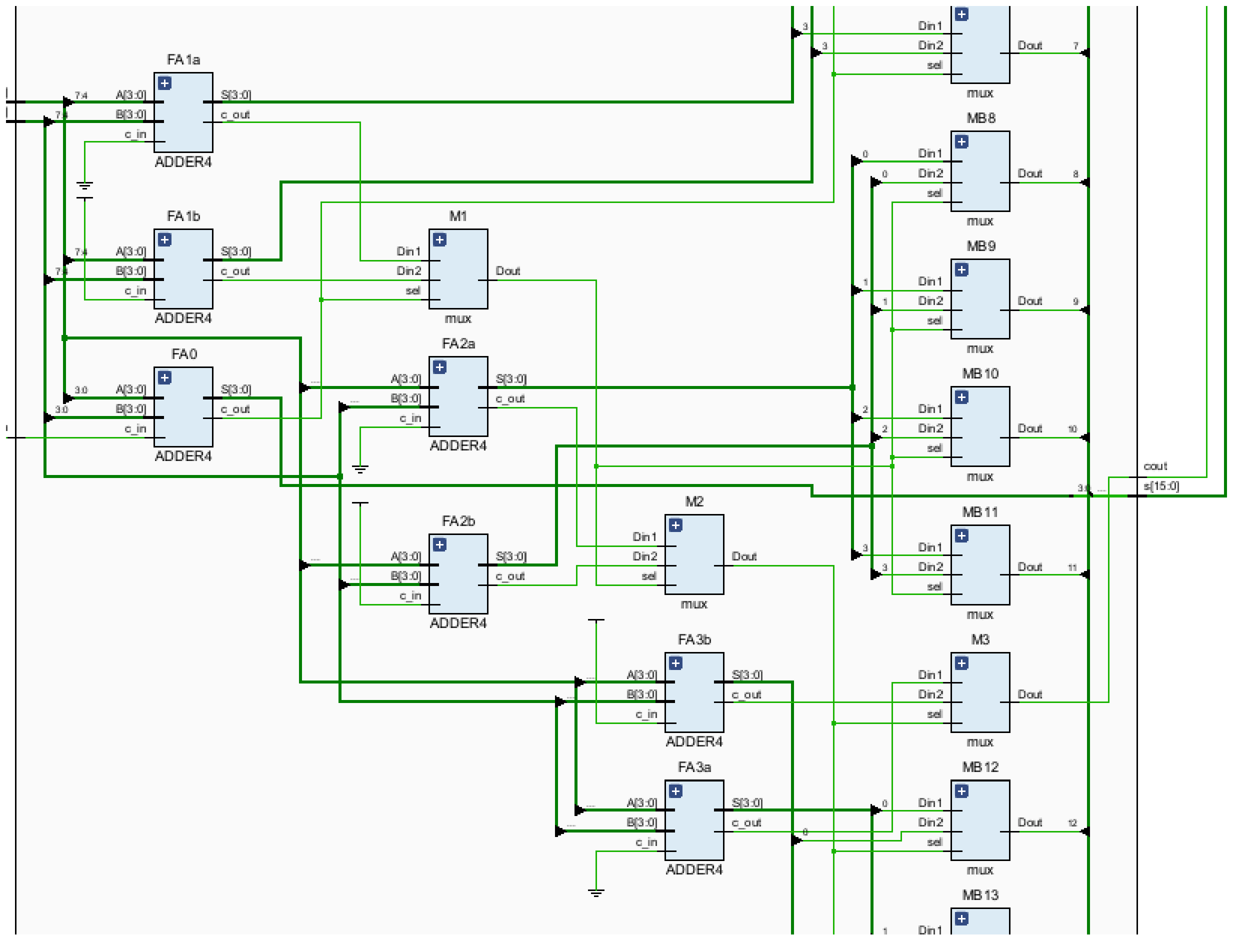


Figure 7: RTL for the CSA module from Vivado

The hierarchical or block-based approach is much better than a flat design for both CSA and CLA. For CSA, using 4-bit blocks is more efficient than individual 1-bit adders. This method allows for parallel computation of larger numbers by managing them in 4-bit chunks, significantly reducing the overall computation time compared to a flat design of single-bit adders connected sequentially.

Although the CSA improves the speed of addition by having parallel computations of each 4-bit block, the overall runtime complexity remains linear $O(\sqrt{n})$, as the selection for each block still depends on the carry-out of the previous block. However, by reducing the delay associated with carry propagation, the CSA achieves a practical speedup over the Ripple Carry Adder.

The CSA cleverly balances the need for speed with the limitations of hardware by calculating multiple segments of the addition in parallel. While it doesn't eliminate the ripple entirely, it reduces the waiting time for carry propagation, making it a preferred choice for applications where speed is crucial but the simplicity of the design also matters.

.SV Modules

Adder_toplevel.sv
Inputs:
Clk, Reset, Execute, Sw_in [15:0]

Outputs: sign_led
hex_gridA [3:0], hex _segA [7:0], hex_gridB [3:0], hex_segB [7:0]

The top level controls everything, it's where all the pieces are connected.

Changes:
We simply choose when to use which adder by commenting out the adders that we didn't want to test at a given time.

Control.sv
Inputs:
Clk, Reset, Run
Outputs**:**
Run_O

This module contains the control unit. The control unit implements the finite state machine of the adders.

Changes:
We did not have to change anything.

Adder_sa.sv
Inputs:
A[15:0],B[15:0],Cin
Outputs:
S [15:0], Cout

This module implements the Carry Select Adder (CSA) which can be found in page 4 of this lab document. It is implemented through 2 Ripple Carry Adders (which we did remake in this file) and 2:1 MUXes.

Changes:
None, we created this .sv file.

Rippleadder.sv
Inputs:
A[15:0],B[15:0],Cin
Outputs:
S [15:0], Cout

This module implements the Carry Ripple Adder (CRA) which can be found in page 1 of this lab. It utilizes the 16 full adders all chained together through the carry in and carry outs

Changes:
Created the 16-bit CRA implementation.

Lookahead_adder.sv
Inputs**:**
A[15:0], B[15:0], Cin
Outputs:
S [15:0], Cout, PG, GG

This module implements the Carry Look-ahead Adder (CLA) which can be found in page 3 It is made using a 4x4 hierarchical design which is made using 5 CLA modules defined in the same file. In the CLA module the necessary logic for signals such as PG, GG, and C16 are assigned (these can be found under Carry Lookahead Adder in this document)

Changes:
None, we created this .sv file.

Mux.sv
Inputs:
S, A_In[1:0], B_In[1:0]
Outputs:
Q_Out[1:0]

This module implements a 2:1 MUX.

Changes:
We created this .sv file.

Reg_17.sv:

Inputs:
Clk, Reset, Load, D [16:0]
Outputs:
Data_Out [16:0]

This module implements a 17 bit register with parallel load and reset features.

Changes:
We did not have to change anything.

hex_diver.sv:
Inputs:
in[3:0], ln(), and reset.
Outputs:
hex_grid[3:0] and hex_seg[7:0]

This module takes in an 4 or 8 bit sequence and formats it to be displayed on the hexadecimal LEDs to show the values in hexadecimal.

Changes:
We did not have to change anything.

Post Lab Questions

Adders Tradeoffs

Carry-Ripple Adder (CRA):
The CRA is simple in design and straightforward to implement, but its main drawback is the long computation time due to the sequential computation of carry-out bits. The propagation delay increases with the size of the adder, making it less suitable for operations requiring high speed.

Carry-Lookahead Adder (CLA):
The CLA improves upon the CRA's speed by making the computation of carry-out bits parallel with the main computation using the concept of generating and propagating logic. This significantly reduces computation time, resulting in a higher operating frequency. However, the downside of the CLA is the increased complexity and area due to additional logic gates, which also leads to higher power consumption.

Carry-Select Adder (CSA):
The CSA offers another method to speed up carry computation by using two CRAs and a multiplexer to pre-compute both possible outcomes of the sum and carry-out based on different carry-in assumptions. Once the actual carry-in is known, the correct result is selected. This approach nearly doubles the number of adders, which increases the area and potentially the power consumption, but it provides a speedup in performance by reducing the overall computation time for the carry bits.

Adders Performance

The Design Analysis below for the three adders shows the comparison of performance by showing the amount Lookup tables (LUT) used, the frequency and the total power use by all three adders. We calculated the frequency by using the period T of the clock cycle (10ns through all adders) and the Worst Negative Slack (WNS) for each Adder. To calculate the frequency we used the following equation: $Frequency = \frac{1}{T - WNS}$.

	LUT	**Frequency (mH)**	**Total Power (W)**
Carry-Ripple Adder	88	77.78	0.084
Carry-lookahead	111	64.6	0.083
Carry Select	95	71.77	0.083

Figure 8: Design Analysis for the three adders displaying the LUT, frequency, and power of each Adder.

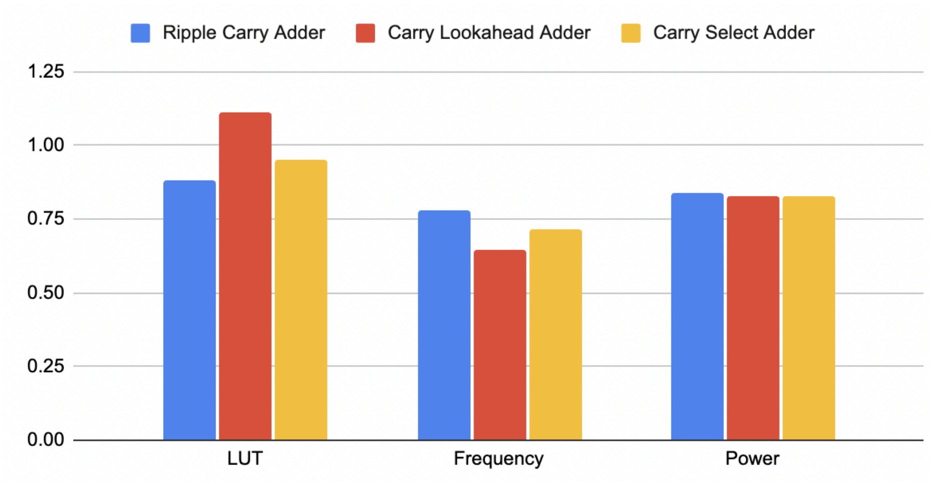


Figure 9: Bar graph for visualizing the performance comparison of the three adders showing the scaled value of the LUT, frequency and power of each adder.

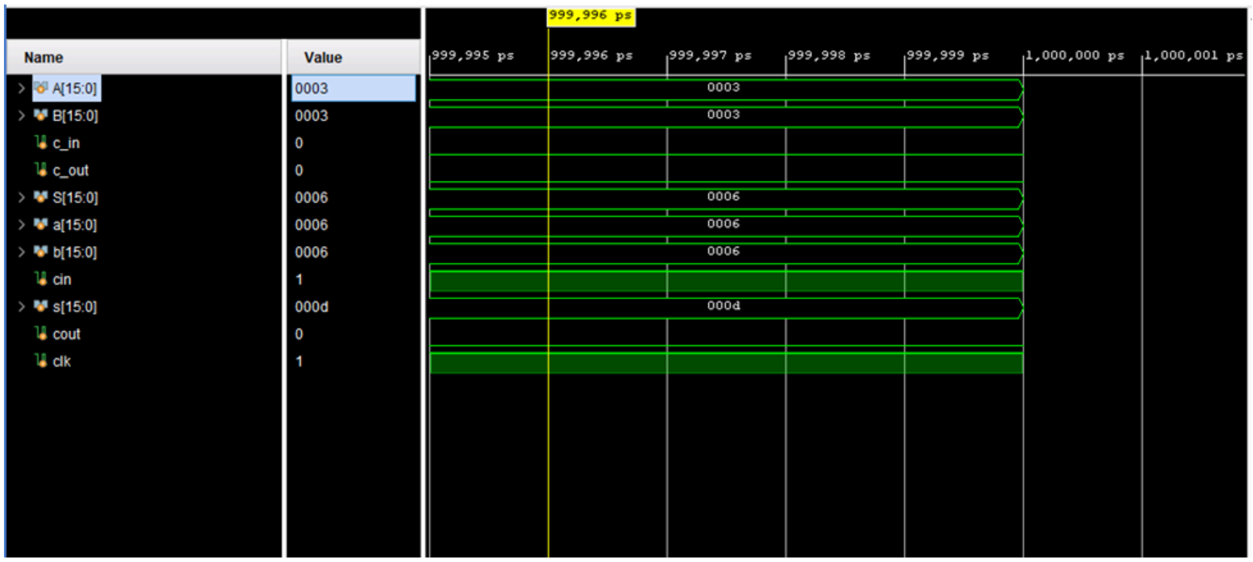
The above results all match with the expected results of our experiment and design. The amount LUT's used in the CRA is lower than both CLA and CSA as it is a more straightforward design using fewer chips while the operating frequency in CRA is much higher than the rest as it is less efficient and much slower. The CRA also uses more power that the other adders which is expected given the lower performance of the CRA.

Performance Metrics:

	Carry Ripple	Carry Lookahead	Carry Select
LUT	88	111	95
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flop	90	90	90
Frequency (gH)	0.07778	0.0.0646	0.07155
Static Power (W)	0.074	0.074	0.074
Dynamic Power (W)	0.009	0.008 W	0.009
Total Power (W)	0.084	0.083 W	0.083

Annotated Simulation Trace

The simulation time trace below shows a sample simulation done to test the functionality of the adders using a simple test bench. The first operation we tested was 3 + 3 as shown in the first two lines on the graph with A being 3 and B being 3 as well and cin being 0. The result of the operation is shown using the output S on the fifth line of the graph which shows that the results of the addition is 6. Another test is done testing the operation 6+6 in which the inputs and output are shown the next five lines with A and B being to 6 and the S output being d which is the accurate result.



Name	Value
A[15:0]	0003
B[15:0]	0003
c_in	0
c_out	0
S[15:0]	0006
a[15:0]	0006
b[15:0]	0006
cin	1
s[15:0]	000d
cout	0
clk	1

Figure 10: Simulation Trace displaying a testbench testing the result of 3+3 and 6+6 with A and B being the input and A being the output.

Critical Path Analysis

Carry Ripple Adder

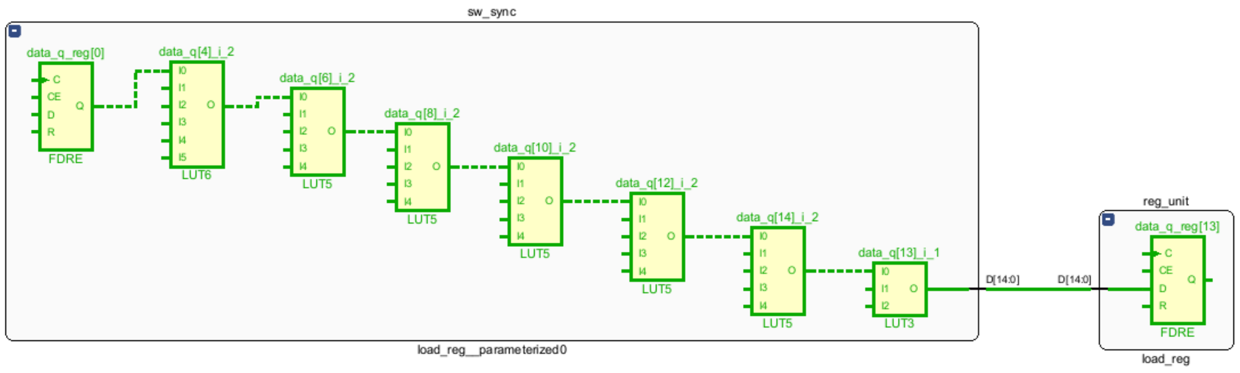


Figure 11 : Critical Path for the Carry Ripple Adder (CRA) : 2.857 Slack

The critical path of a CRA, as depicted in the diagram, illustrates the delay characteristic of this type of adder. The path runs sequentially from the initial carry-in (Cin) to the final carry-out (Cout) across the full adders, involving look-up tables (LUTs) that process the carry bits. This sequential processing is consistent with our expected theoretical behavior of a CRA, where each bit's addition is contingent on receiving the carry-out from the preceding bit.

Carry Lookahead Adder:

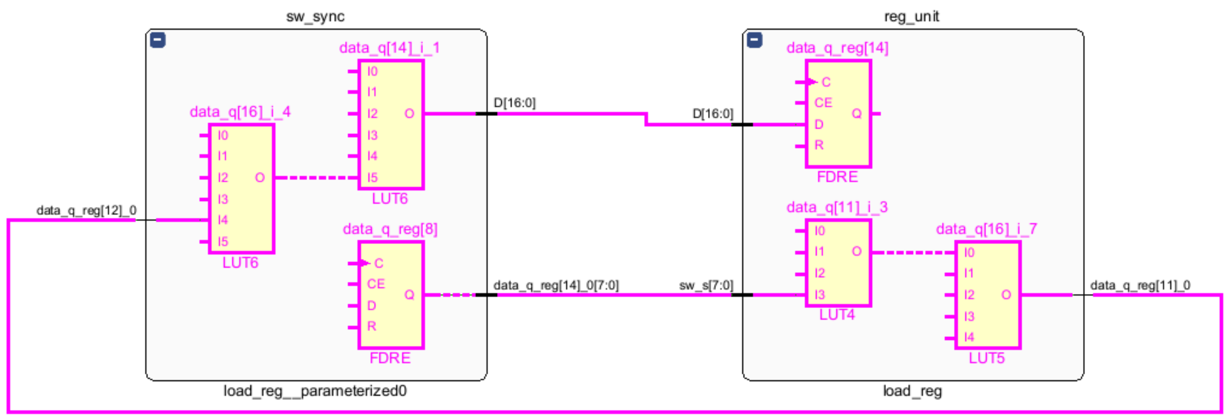


Figure 12 : Critical path for the carry Lookahead adder (CLA) : slack 4.573

The CLA shown in Figure12 is designed to improve the carry operation's speed over a traditional CRA. The critical path in a CLA is shorter because it generates carry signals in advance, which reduces the dependency chain seen in a CRA. This is reflected in the critical path slack time as the slack time for the CRA was 4.573 ns while it was 2.857 for the CLA indicating that the Lookahead Adder is able to compute its results faster. This aligns with theoretical expectations since Lookahead Adders are designed to address the carry propagation delay problem by generating carries quickly and in parallel.

Carry Select Adder:

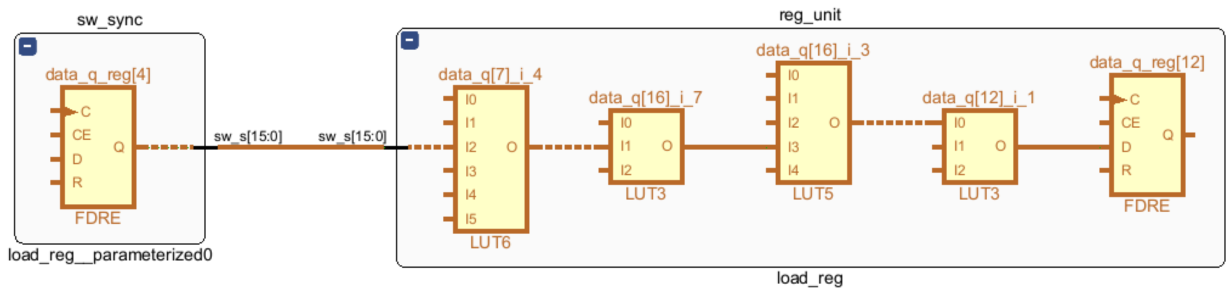


Figure 13: Critical path for carry select adder (CSA): slack 3.976

The critical path depicted for the Carry Select Adder (CSA) shows a structure designed to improve upon the delays encountered in a Carry Ripple Adder (CRA). The CSA achieves this by computing two potential sum outputs for each bit block in parallel, selecting the correct one based on the carry-in value. The critical path therefore includes the time for parallel computation in the look-up tables (LUTs) and the selection mechanism, which is typically a multiplexer.

In this CSA diagram, the slack of 3.976 indicates that the CSA is faster than the CRA, whose critical path analysis showed a slack of 2.857. The CSA's improved slack suggests that it handles carry operations more efficiently, which is in line with the theoretical understanding that CSAs are designed to reduce the carry propagation delay by precomputing the carry values. This is a practical implementation that should, according to theory, result in a faster addition process compared to CRAs.

In the CSA for this lab, we asked you to create a 4x4 hierarchy. Is this ideal?

No. While the 4x4 block size offers a uniform approach, it may not be the most efficient in terms of carry propagation time, which is a critical factor in adder performance.

The ideal hierarchy for a CSA on an FPGA would take into consideration the propagation delay inherent in the carry operation. Since each block's computation depends on the carry-out value of the previous block, there could be a performance bottleneck if all blocks are of the same size. A potential solution to this would be to vary the block sizes within the CSA hierarchy, creating a non-uniform block structure that adapts to the carry propagation delay.

For instance, beginning with a single bit and then increasing to larger blocks (e.g., three, six, nine bits, etc.) as we move deeper into the adder chain can optimize the computation time. Smaller blocks at the start can compute their carry-out values more rapidly due to fewer operations, allowing the carry signal to propagate faster to subsequent blocks. Conversely, larger blocks later in the sequence can afford to wait slightly longer for the carry-in value because they have more complex computations to perform.

This graded hierarchy aims to synchronize the computation time of each block with the anticipated carry propagation time, effectively reducing the delay and improving the overall speed of the adder. Designing the ideal hierarchy would involve a process of experimentation and iteration: testing various configurations of block sizes, analyzing their performance in terms of speed and resource utilization, and refining the design to achieve the optimal balance for the specific FPGA and application requirements.

Conclusion

In this lab, we examined three distinct 16-bit adders, each implemented using full adders. Firstly, the Carry Ripple Adder (CRA),which has a simple and straightforward design. The CRA's performance is the slowest compared to the other adders we implemented due to its sequential carry processing.

Next the Carry Select Adder (CSA), we implemented a design that balances complexity with performance. By precomputing results for multiple potential carry inputs and selecting the appropriate outcome, the CSA enhances computation speed. The CSA demands more hardware, which increases the footprint and the expenses of the design.

Lastly, the Carry Lookahead Adder (CLA), which prioritizes speed by eliminating the sequential dependency found in the other adders. Through use of Propagate and Generate groups, the CLA reduces computation time, establishing itself as the fastest adder in our experiment.
When it comes to operation speed of the three designs this the order

CSA > CLA > CRA

Despite initial hurdles and bugs, particularly with the CLA's logic, the invaluable support during office hours led to successful implementations. We also had some trouble getting used to the System Verilog Syntax, and mixed up the inputs with the outputs, but after reading the manual provided we got the hang of it.

In conclusion, this lab gave us an insight into the trade-offs between design complexity, operational speed, and resource requirements in digital adder architectures. By navigating through the implementation of the CRA, CSA, and CLA.