ECE 385

Fall 2023

Lab 3: 16 bit adders

Alex Yu, Corey Yu
GL/ 13:45-14:00
Gene Lee

Introduction

In this lab we created 3 types of 16 bit adders, ripple adder, carry look ahead adder, and carry select adder. These three adders have the same functionality of adding two 16-bit numbers, and they share the same load A, load B, and LED display, but use a different method to add. The ripple adder is a naive design with the longest propagation delay (and thus the slowest), as such two alternate methods to increase speed via parallelization were implemented: the carry look ahead method and the carry select method.

Adders

- **Ripple Carry Adder**

The n bit ripple adder is constructed by cascading together n one bit full adders. The one bit full adder takes in the one bit data from A, B and carry-in, generates the one bit result and one bit carry-in bit. This implementation, while the simplest, is sequential and thus the slowest since all the adders have to wait for the previous carry in to calculate the output.

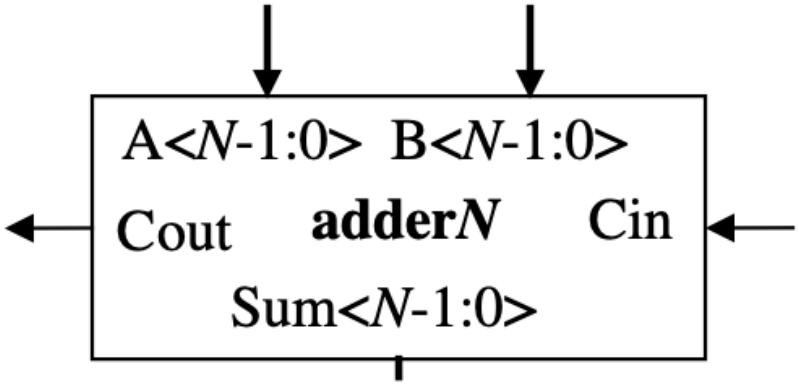
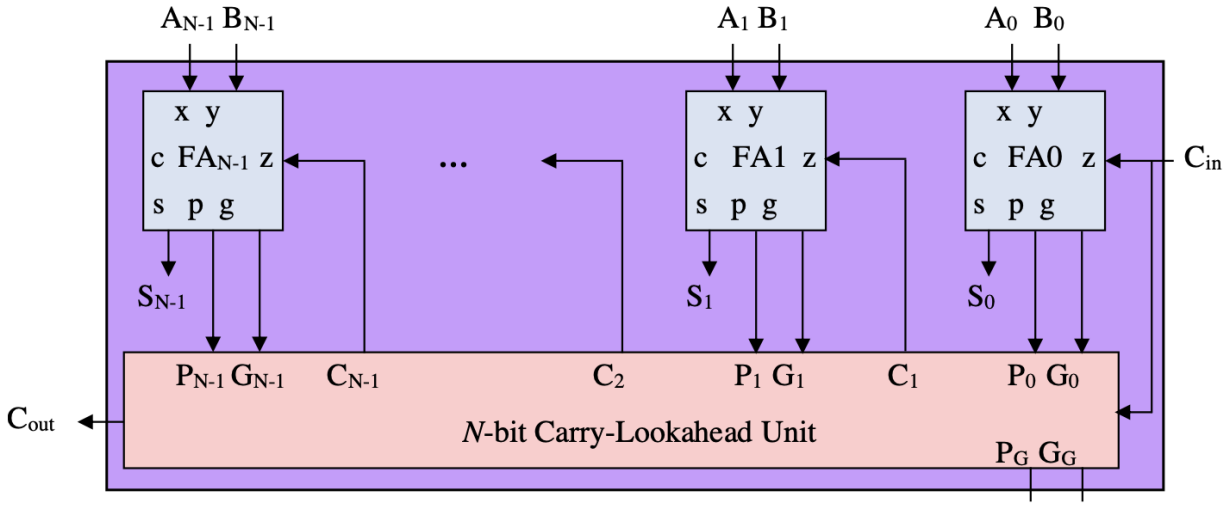


Figure 1 - N-bit Binary Adder Block Diagram

- **Carry Lookahead Adder**

The carry lookahead adder seeks to increase the speed of the naive ripple carry adder design. It generates two signals propagate(P) and generate(G) to to generate the carry-in from.



For a given one bit adder, a carry-out will only be generated when A and B are both 1. A carry-out will only be propagated from the previous one bit adder through the next one bit adder if either A or B is one. Thus P = A xor B and G = A & B. The carry-in bit for each adder can thus be derived from P and G and the one bit adders are implemented in parallel (with additional logic, a “look ahead block”) to generate a sum faster than the ripple adder.

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

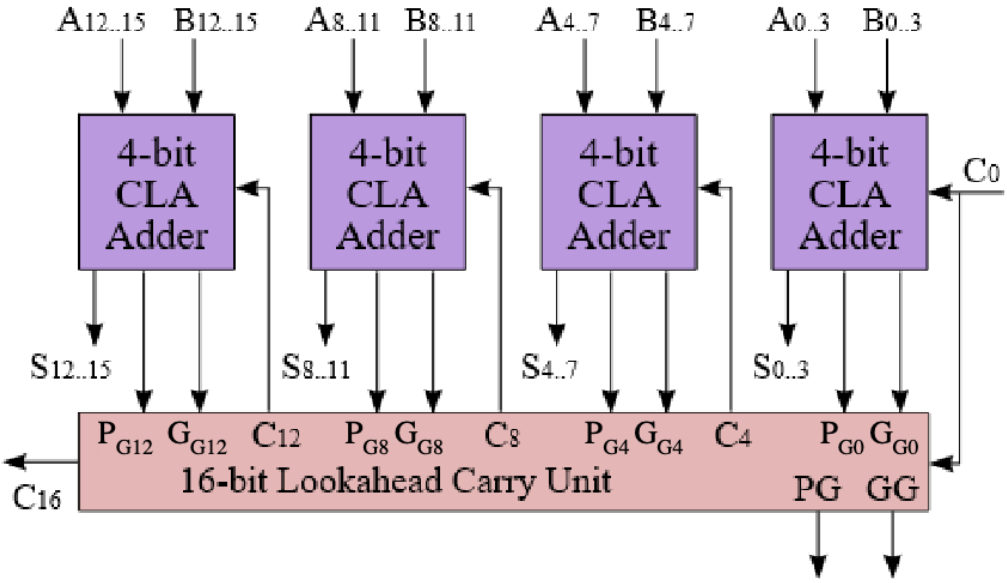
$$C_8 = G_{G4} + G_{G0} \cdot P_{G4} + C_0 \cdot P_{G0} \cdot P_{G4}$$

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

...

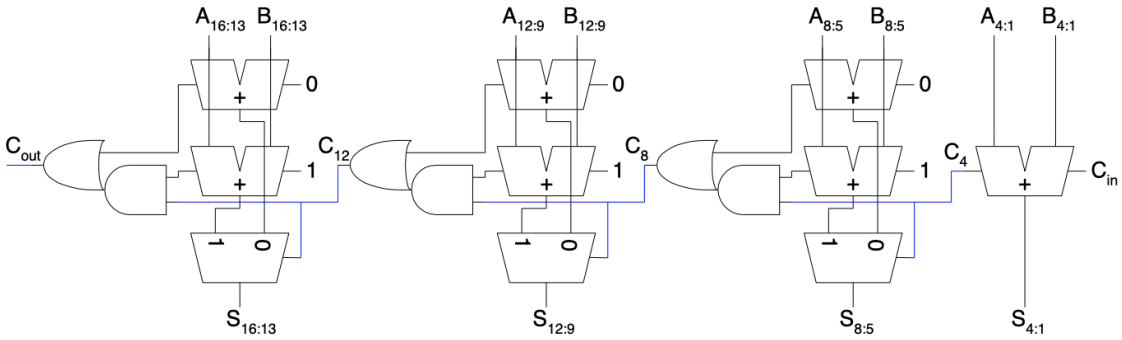
To construct a 16 bit ripple adder, rather than stringing 16 one-bit full adders together in parallel, a hierarchical 4x4 bit design was implemented. The number of logic gates required grows unsustainably with the size of n. It is common practice to create smaller CLA’s and feed them into another lookahead unit to create a larger adde in a

hierarchical fashion; 4 CLA's were combined to create a 16 bit adder in this fashion.



- **Carry Select Adder**

The carry select adder is another way to increase the speed of an adder. It precomputes both possible outcomes of sums with cin=0 and cin=1, and when the actual cin from the previous block of adders is generated, this is used to select the actual output of the sum via a MUX.



Each one-bit adder takes one logic level to generate the sum, and 2 logic levels to generate the carry-out. Thus a 16 bit adder takes 16*2 logic levels to generate the outputs. By contrast, the 16 bit CSA takes 4*2 + 3*2 logic levels to generate the outputs. By precomputing possible sums, the cascading delay of the ripple adder is avoided and replaced with an increase of two logic levels.

- **All sv files**

Module: adder_toplevel.sv

input Clk, Reset_Clear, Run_Accumulate,
input [15:0] SW,
output logic sign_LED,
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: This is the top level module that takes the input data from the switch, calls the adder modules with the data, and outputs the results from the adder to the hex display.

Purpose: This module organizes data between all the sub modules and wraps everything. We need this module to control which adder we want to use and how we want to call the module.

Module: ripple_adder.sv(Module: ADDER4, Module: full_adder)

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

Description: the ripple_adder module contains two submodules full_adder and ADDER4. The full_adder is a one bit full adder that takes in one bit input and does the calculation only on a single bit. In the ADDER4 module, we call the full_adder 4 times, to calculate a total of 4 bits. Finally, in the actual ripple adder, we call the 4 bit adder (ADDER4 module) 4 times to get a 16 bit result in total.

Purpose: We need this module to implement the ripple adder, and the reason why we wrote two submodules instead of just calling one bit adder four times is that it will make the debugging faster and the code more efficient.

Module: lookahead_adder.sv (Module: LA_ADDER, Module: pg_adder)

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

Description: The lookahead adder also consists of two sub modules that deal with a smaller number of bits. We have the pg_adder that takes in the a, b, and cin to output the result for the bit and not the cout bit but the Propogate(P), and Generate(G) signal for that bit. The P and G are calculated through the combinational logic. In the module LA adder, we call the pg_adder 4 times, and set the cin for the next bit using the P and G signal instead of the Cout. Finally, we call the LA_ADDER 4 times in the lookahead_adder to get the final 16 bit adder.
Purpose: We need this module to implement the look ahead adder, and the use of signal P and G will speed up the adding process without having to wait for the Cout bit.

Module: select_adder.sv (Module: MUX)
input [15:0] A, B,
input cin,
output [15:0] S,
output cout
Description: In the select adder module, we implement it using the mux module and the adder module from the ripple adder with the different logic gates. We feed in the A and B data into the adder for both possible Cin signals, and we use the mux module to select which is the correct one to output.
Purpose: We need this module to implement the select adder, this module is also faster than the ripple adder since it pre-calculated the result for different cin bits, so the signal won't have to go through as many gate delays to reach the final output.

Module: reg_17.sv
input Clk, Reset, Load,
input [16:0] D,
output logic [16:0] Data_Out
Description: This is the register that stores data in register A and register B. It can clear the data when it receives the reset signal, and load the data when the load button is pressed.
Purpose: We need the register module to store data, load data, and clear data.

Module: mux_2_1_17.sv
input S,
input [15:0] A_In,
input [16:0] B_In,
output logic [16:0] Q_Out
Description: this is the mux that either put the result of A+B into B or the original B data into register.

Purpose: We need this module to select where we want to put the data.

Module: Control.sv
input logic Clk, Reset, Run,
output logic Run_O
Description: This is a three state control unit that uses a counter to make sure the button signal lasts a one clock cycle long.
Purpose: We need the control unit to make sure the button is pressed for at least one time cycle, so the fpga can actually receive the signal and execute the proper behavior after the button is pressed.

Module: HexDriver.sv
Inputs: Clk, reset, [3:0] in[4], [3:0] nibble
Outputs: [7:0] hex_seg, [3:0] hex_grid, [7:0] hex
Description: The hex driver takes in the data in the binary format and outputs the number to hex that can be shown on the LED segment display.
Purpose: The LED segment display doesn't take in numbers in binary so we need this module to translate it to display.

The area, complexity, and performance tradeoffs:

Ripple Adder is the simplest design of the three. It implements a 16*1 flat design, in which each full bit adder is connected through the carry in and the carry out. The ripple adder uses up the least area of the three with 86 LUTs; however, this implementation is also the slowest. It has to wait for the carry out and carry in signal to go through all the gates to get the final result. From Vivado, we can see that the ripple adder with WNS = 3.278. This means that the carry ripple design tradeoff the performance for the least area.

The Carry Lookahead Adder is more complex than the Ripple Adder. It uses the two signals Propagate and Generate to replace the cin and cout bit. This allows us to get the carry in bit for all the adders without having to wait for the signal to go through all delays. The P, Q, and the Cin signal for the first adder will go through a combinational logic to generate the carry in bit. This implementation makes the adder perform better, with WNS = 5.655, the fastest of the three, but the area of the CLA is also the biggest using 103 LUTs. This means that the carry lookahead adder design trades off the area for the best performance.

The Carry Select Adder is the most complex of the three, using MUX and combinational logic to generate the output. The carry select adder is also faster than the ripple adder

since for most of the case, the adder calculates the result for both cases before the carry in bit arrives, so when the carry in bit gets to the adder, it doesn't have to go through the gate delay of the adder but just 1 mux to select the correct output. The performance is therefore better with WNS = 5.598. However, because the select adder uses extra muxes and logic gates, the area is also bigger than the ripple adder with 92 LUTs.

Result:
Area: Carry Lookahead Adder > Carry Select Adder > Ripple Adder
Complexity: Carry Select Adder > Carry Lookahead Adder > Ripple Adder
Performance: Carry Lookahead Adder > Carry Select Adder > Ripple Adder

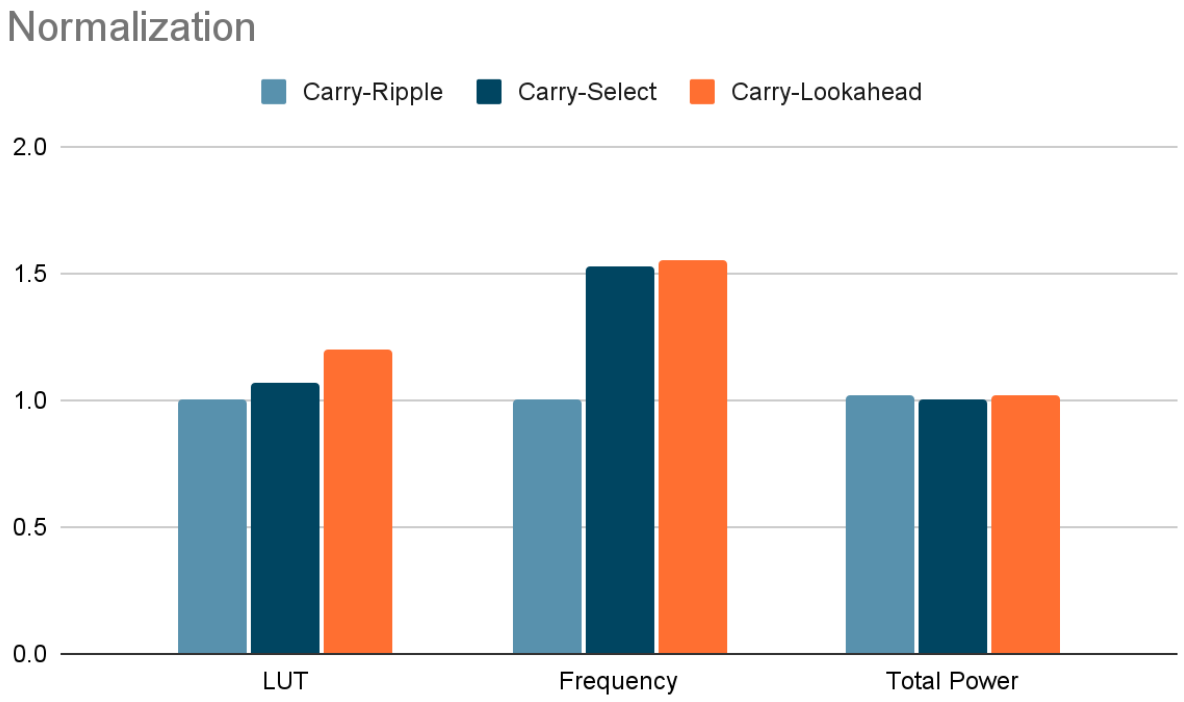
PostLab

Original Data	**Carry-Ripple**	**Carry-Select**	**Carry-Lookahead**
LUT	**86**	**92**	**103**
Frequency	**0.1487**	**0.2271**	**0.23014**
Total Power	**0.095W**	**0.093W**	**0.095W**

WNS = one clock period - time of critical path. In this case, it is 10ns - time of critical path. To get the frequency, we need to inverse the result of 10 - WNS.

Frequency = 1/(10-WNS), Max freq is largest freq s.t. 0 = 1/(freq) - crit path time

Normalization	**Carry-Ripple**	**Carry-Select**	**Carry-Lookahead**
LUT	**1**	**1.07**	**1.20**
Frequency	**1**	**1.53**	**1.55**
Total Power	**1.02**	**1**	**1.02**



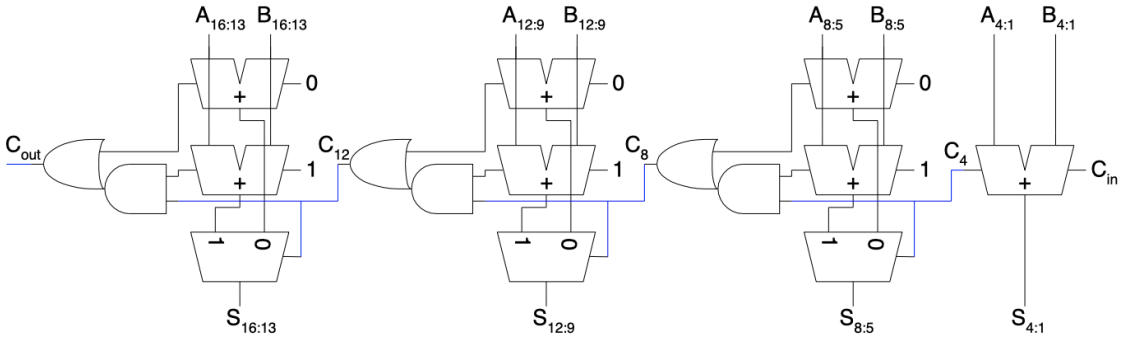
Yes, the LUT and Frequency results are exactly as we expected. Carry-Lookahead Adder uses the most LUT because of the combinational logic of P&Q and it is also the one with the best performance. The maximum frequency of the carry lookahead adder is higher than the ripple adder. For the total power of all three adders, it is all around the same value. This is not what we expected, but we think this might be because they uses so little of the FPGA that there isn't really a difference between them.

	Carry-Ripple	**Carry-Select**	**Carry-Lookahead**
LUT	**86**	**92**	**103**
DSP	**0**	**0**	**0**
Memory (BRAM)	**0**	**0**	**0**
Flip Flop	**53**	**53**	**53**
Frequency	**0.1487**	**0.2271**	**0.23014**
Static Power	**0.0722**	**0.0716**	**0.0722**
Dynamic Power	**0.0228**	**0.02139**	**0.0228**

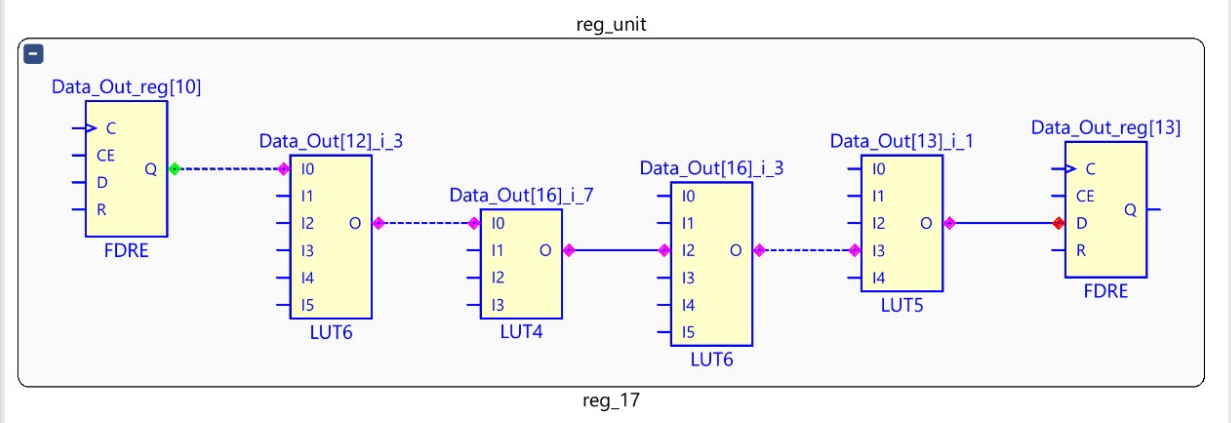
Total Power	**0.095W**	**0.093W**	**0.095W**

Is 4x4 the ideal design for CSA?

As this lab was to design faster adders, ideal is presumably taken to mean fastest. The ideal design occurs when the previous c-out takes the same logic levels to generate as the current adder block. E.g. if the number of gate delays to generate C8 was the same as the number of gate delays to generate the sum of A(12:9) + B(12:9). That way both signals are ready at the same time and there is no wasted time. This occurs with non heterogeneous blocks, 5/4/3/2/2 bit full adders for each MUX block respectively, with the rightmost 2 bit adder as the least two significant bits.



Carry Lookahead Adder

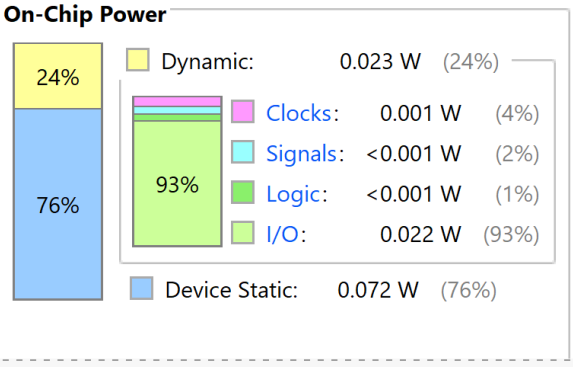


Summary

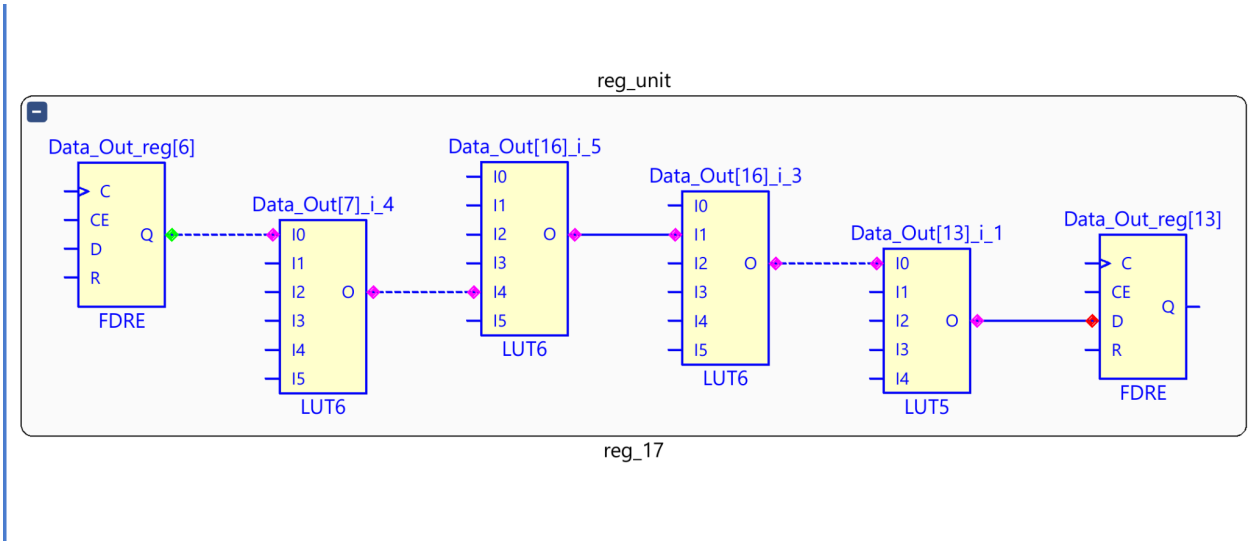
N/A

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.095 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.5°C**
Thermal Margin:	59.5°C (12.0 W)
Effective ϑJA:	4.9°C/W



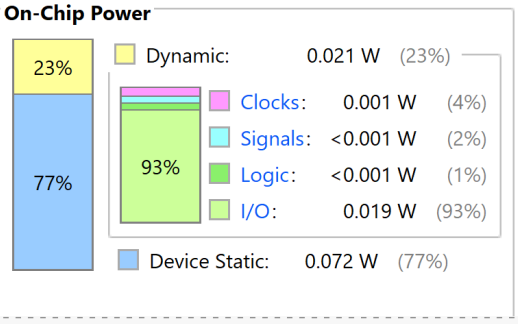
Carry Select Adder



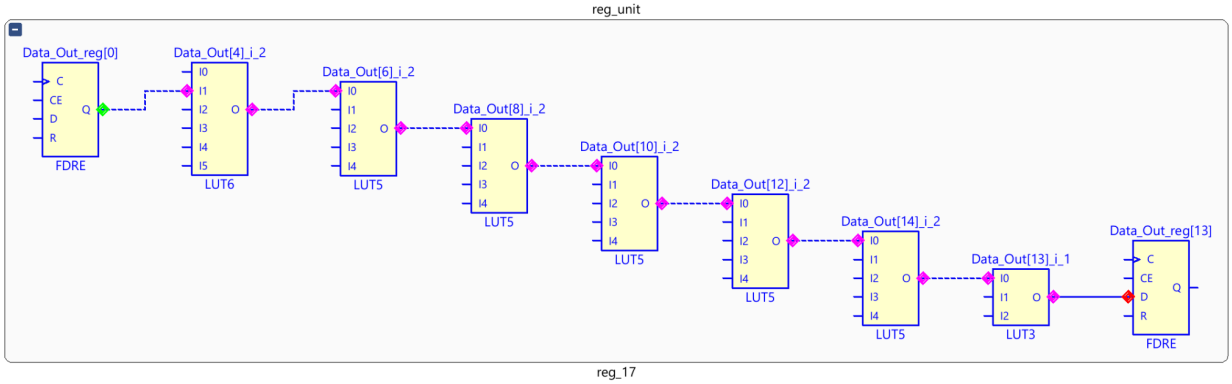
A

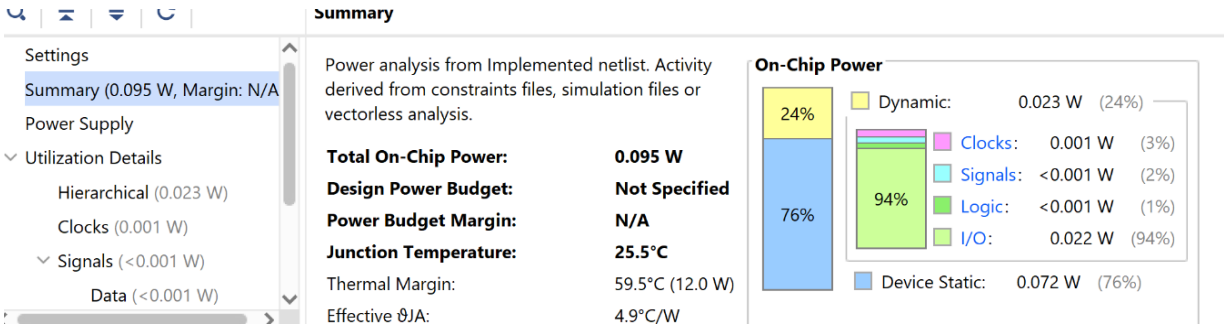
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.093 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.5°C**
Thermal Margin:	59.5°C (12.0 W)
Effective ϑJA:	4.9°C/W



Carry Ripple Adder





The critical path given by Vivado is very close to the theoretical critical path we expected. For the carry ripple adder, the Vivado generates a longer critical path with more look up tables. And for the other two adders, they have a relatively shorter path and therefore it will take a shorter amount of time to go through the whole thing. This is the same as what we predict will happen in the previous part.

Also we can see that for the ripple adder, the critical path shows the carry chain through all the adder. This is because the carry ripple adder chains every adder with cin and cout signal between them. On the other hand, the carry lookahead doesn't chain together, since each individual adder doesn't rely on the previous adder's cout but the P and G.

Conclusion

Overall, we didn’t really face a lot of bugs in this lab. It is fairly simple and the content is easy to understand. We encounter some bugs while we are writing the code for carry lookahead adder because there is a lot of trivial logic, and it is easy to mistype something, but we were able to figure it out soon. In this lab, we learned about how the different implementations of the adders affect the performance, area, power, frequency, etc. The only thing that can be improved might be the design of the Urbana board since the reset button can’t really reset the register properly.