Zhihao Wang (zhihaow6), Laurenz Nava (lfnava2)

Introduction

In this lab, several variants of 16-bit adders are designed in hierarchical 4x4 style, such as the carry ripple adder, the carry lookahead adder, and the carry select adder, in order to compare each of the methods in terms of performance, size, and power consumption. The carry ripple adder adds individual bits together, potentially generating a carry bit, and inputs the carry bit into the next carry ripple adder. The carry lookahead adds all the bits in parallel and creates bits that signify that a carry bit will be generated or propagated for a given pair of bits. The carry select adder computes two sums in parallel, one assuming a carry in bit and the other assuming there is not a carry in, and then the actual carry in activates a MUX to select the correct adder's output to output as a whole.

Adders

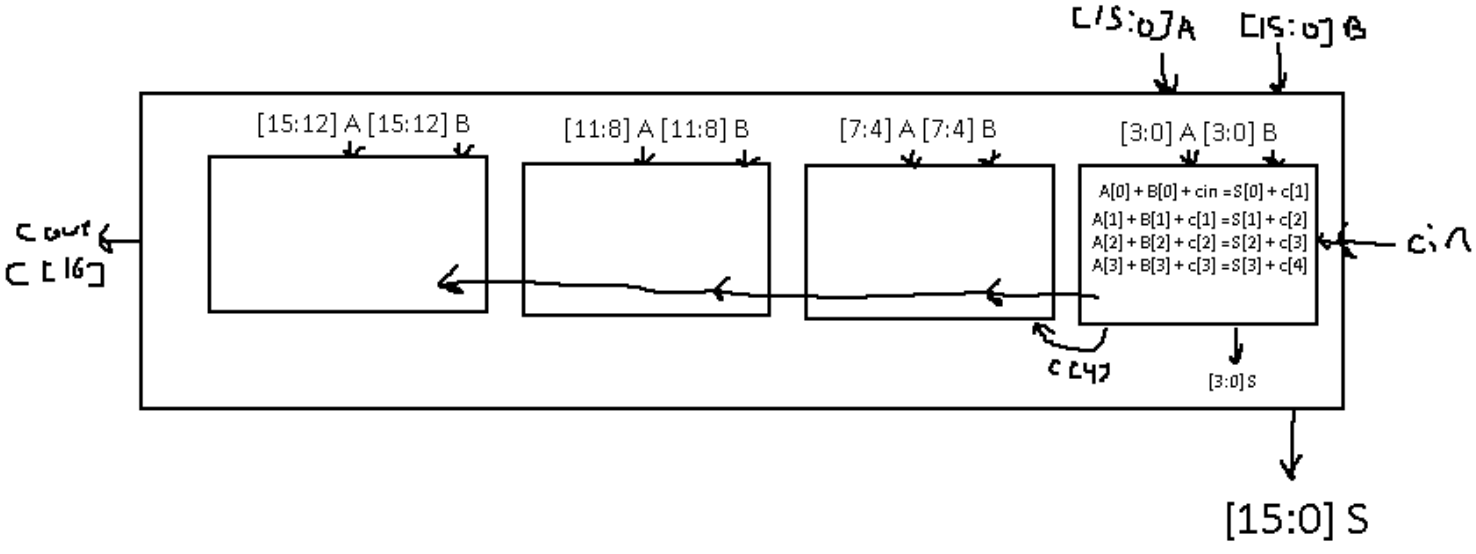
Ripple Carry

The ripple carry adder is composed of four 4-bit carry ripple adders, which themselves are composed of four 1-bit carry ripple adders.

The 1-bit carry ripple adders work by taking in 1 bit inputs A, B, and Ci to produce 1 bit outputs S and Co. A and B are the numbers that are to be summed up and Ci is a possible carry in from a previous sum. S is the sum of the inputs and is the result of A ^ B ^ Ci, and Co is the carry out is the result of A & B | B & Ci | A & Ci.

The 4-bit carry ripple adders take 4-bit inputs A and B, and 1-bit input Ci, to produce 4-bit output S and 1-bit output Co. The inputs and outputs have the same meaning. The 4-bit carry ripple adder breaks up the inputs into individual bits, and use the least significant bits of A and B in addition to Ci as inputs for a 1-bit carry ripple adder, it then takes the outputted S for the least significant S and the Co for the Ci of the next bit with the next bits of A and B. This continues until all 4-bits are completed.

The full ripple carry adder works similarly. It takes 16-bit inputs A and B, and a 1-bit input cin to produce a 16-bit output S and a 1-bit output cout. The inputs and outputs have the same meaning, with cin meaning the same as Ci, and cout the same as Co. It breaks up A and B into four parts of 4-bits, and inputs the least significant 4-bits with cin into a 4-bit carry ripple adder, store the outputted S as the least significant 4-bits of S and use the Co for the next 4-bit carry ripple adder's Ci. This repeats until all 16-bits are completed.



Carry Lookahead

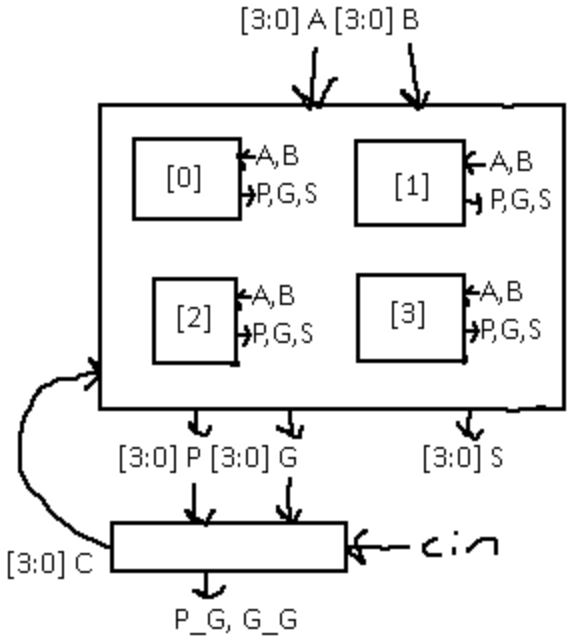
The carry lookahead adder is designed as a hierarchical 4x4-bit carry lookahead adder. The carry lookahead adder starts from a 1-bit carry lookahead adder. It takes in 1-bit inputs A, B, and C, and produces 1-bit outputs S, P, G. A and B are the numbers that are to be summed up and C is a possible carry in from a previous sum. S is the sum of the inputs, P is a propagating carry, meaning if there is a carry in, there will be a carry out, and G is a generated carry, meaning there will be a carry out, no matter the carry in. S is the result of A ^ B ^ C, P is the result of A ^ B, and G is the result of A & B.
Four 1-bit carry lookahead adders are combined for a 4-bit carry lookahead adder. It takes in 4-bit inputs A and B, and 1-bit input Cin, and produces 4-bit output S and 1-bit output P_G and G_G. The inputs and outputs have the same meaning, with Cin the same as C, P_G as the group propagate, and G_G as the group generate. It breaks the 4-bit inputs A and B into individual bits, starts from the least significant bit and Cin, and uses a 1-bit carry lookahead adder. In addition to the four 1-bit carry lookahead adders, the 4-bit adder uses P and G from the 1-bit adders and Cin to produce the intermediate C for the 1-bit adders and P_G and G_G. This is the basis for the hierarchical design, as the next level down can use this same logic in its implementation. The boolean logic is shown below.

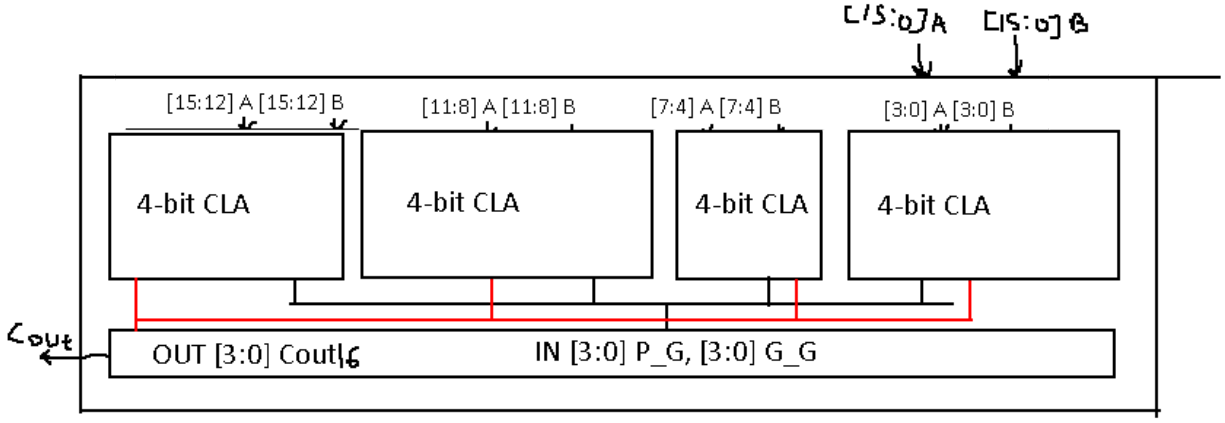
C[0] = Cin
C[1] = (Cin & P[0]) | (G[0])
C[2] = (Cin & P[0] & P[1]) | (G[0] & P[1]) | (G[1])
C[3] = (Cin & P[0] & P[1] & P[2]) | (G[0] & P[1] & P[2]) | (G[1] & P[2]) | (G[2])

P_G = P[0] & P[1] & P[2] & P[3]
G_G = (G[3]) | (G[2] & P[3]) | (G[1] & P[3] & P[2]) | (G[0] & P[3] & P[2] & P[1])

The full carry lookahead adder is implemented similarly. It takes 16-bit inputs A and B, and a 1-bit input cin to produce a 16-bit output S and a 1-bit output cout. The inputs and outputs have the same meaning, with cout meaning a carry out number of the sum. It breaks up A and B into four parts of 4-bits, starts with inputting the least significant group into a 4-bit carry lookahead adder, and repeats until all 16 bits are done. Similar to the 4-bit lookahead adder, the full lookahead adder uses the P_G and G_G with the cin to produce the intermediate Cin of the 4-bit adder following the same logic as above. The boolean logic is shown below, with P16 and G16 being the propagate and generate bits of the full 16-bit carry lookahead adder.

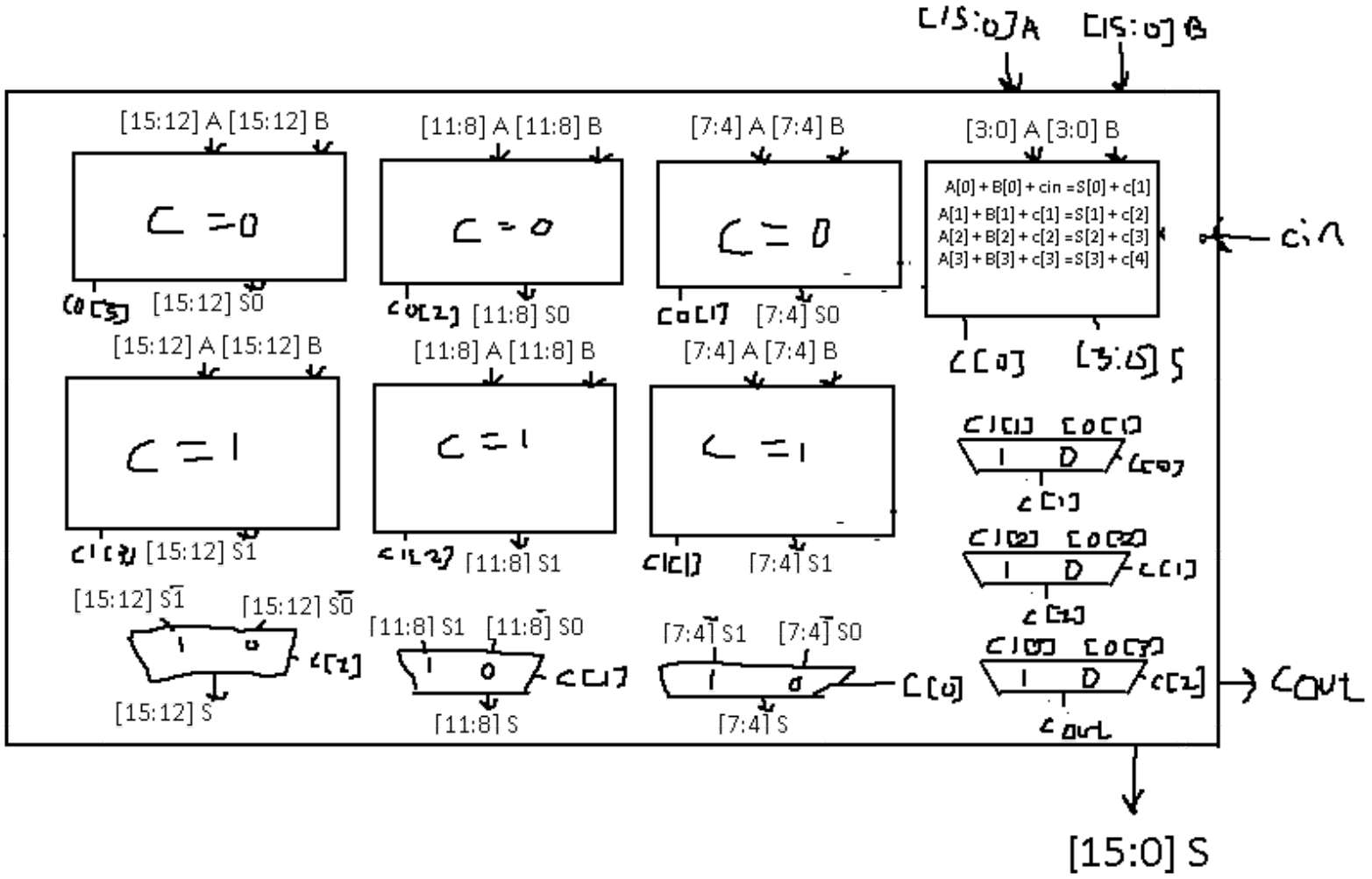
cout = (cin & P16[0] & P16[1] & P16[2] & P16[3]) | (G16[0] & P16[1] & P16[2] & P16[3]) | (G16[1] & P16[2] & P16[3]) | (G16[2] & P16[3]) | (G16[3])


[3:0] A [3:0] B
[0]
A,B
P,G,S
[1]
A,B
P,G,S
[2]
A,B
P,G,S
[3]
A,B
P,G,S
[3:0] P [3:0] G
[3:0] S
[3:0] C
cin
P_G, G_G


[15:0] A
[15:0] B
[15:12] A [15:12] B
[11:8] A [11:8] B
[7:4] A [7:4] B
[3:0] A [3:0] B
4-bit CLA
4-bit CLA
4-bit CLA
4-bit CLA
Cout
OUT [3:0] Cout
IN [3:0] P_G, [3:0] G_G

Carry Select

The carry select adder works very similarly to the carry ripple adder described above, except for the fact that after the first 4-bit carry ripple adder, there are twice as many 4-bit carry ripple adders in parallel. The reason for having two adders in parallel is to calculate the sum of the bits with one adder assuming the carry in is 1 and the other assuming the carry in is 0. It then selects the adder to output the sum and carry out by MUXs in the actual implementation controlled by the carry out of the previous 4-bit ripple adder. This repeats until all 16 bits are added together.



Written description of all .SV modules

Module: control.sv
Inputs: Clk, Reset, Run
Outputs: Run_O
Description: This is a positive edge Clk triggered to set the next state and counter, and an asynchronous reset to set state to A and counter to x00.
Purpose: Manage states for timing to not update registers during computation cycle

Module: mux2_1_17.sv
Inputs: S, [15:0] A_In, [16:0] B_In
Outputs: [16:0] Q_out
Description: If S = 0, Q_out = {0 , A_In}, If S = 1, Q_out = B_In
Purpose: Takes in value from switches (S = 0) or value in registers of previous sum (S = 1)

Module: reg_17.sv
Inputs: Clk, Reset, Load, [16:0] D
Outputs: [16:0] Data_Out
Description: Synchronous register, if Reset is high, Data_Out = 0x00, or if Load is high, Data_Out = D.
Purpose: Register to hold cumulative sum, reset registers upon button press

Module: ripple_adder.sv
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] S, cout
Description: See above
Purpose: Compute sum of 16-bit numbers

Module: single_cra, ripple_adder.sv
Inputs: A, B, Ci
Outputs: Co, S
Description: See above
Purpose: Compute sum of 1-bit numbers

Module: four_cra, ripple_adder.sv
Inputs: [3:0] A, [3:0] B, Ci
Outputs: Co, [3:0] S
Description: See above
Purpose: Compute sum of 4-bit numbers

Module: lookahead_adder.sv
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] S, cout
Description: See above
Purpose: Compute sum of 16-bit numbers

Module: group, lookahead_adder.sv
Inputs: Cin, [3:0] P, [3:0] G
Outputs: [3:0] C, P_G, G_G
Description: See above
Purpose: To produce P_G. G_G. and intermediate C values

Module: single_cla, lookahead_adder.sv
Inputs: A, B, C
Outputs: S, P, G
Description: See above
Purpose: Compute sum of 1-bit numbers

Module: four_cla, lookahead_adder.sv
Inputs: [3:0] A, [3:0] B, Cin
Outputs: P_G, G_G, [3:0] S
Description: See above
Purpose: Compute sum of 4-bit numbers

Module: select_adder.sv
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] S, cout
Description: See above
Purpose: Compute sum of 16-bit numbers

Module: two_four_sel, select_adder.sv
Inputs: [3:0] A, [3:0] B, C_sel_in
Outputs: C_sel_out, [3:0] S
Description: See above
Purpose: Compute sum of 4-bit numbers

Module: Hex Driver, hex.sv
Inputs: clk, reset, [3:0] in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: Convert input data into displayable data, then display on 7 segment display
Purpose: Display input data on 7 segment display

Module: nibble to hex, hex.sv
Inputs: [3:0] nibble
Outputs: [7:0] hex
Description: Turns the 4-bit nibble input into an 8-bit output readable by the hex segments to display the hexadecimal value of the input
Purpose: Convert binary input data into displayable on 7 segment display output data

Area, complexity, performance tradeoffs between all adders

Due to its simplicity, the ripple adder has the smallest area, since it uses the least gates, the least complex, with the simplest design that the rest are based off of, and the worst performance, since there are no optimizations.

The lookahead adder would have the largest area, since it has the most combinational logic to implement, the most complexity, again most combinational logic, and better performance compared to the ripple adder by adding propagation and generate bits to simply some of the carry ripple

The select adder has a larger area compared to the ripple adder, using twice the amount of adders after the first 4-bit block, but not as much as lookahead since it doesn't use as much other logic, in the middle in terms of complexity for the same reason as before, and better performance than the ripple adder as it does not need to wait for a bit to carry in to calculate the sum.

Performance by graph

Frequency = 1/(10 ns - WNS)

	Carry Ripple	Carry Lookahead	Carry Select
LUT	84	109	91
Frequency	149.254 MHz (3.300 ns WNS)	193.648 MHz (4.836 ns WNS)	220.556 MHz (5.466 ns WNS)
Total Power	0.095 W	0.092 W	0.093 W

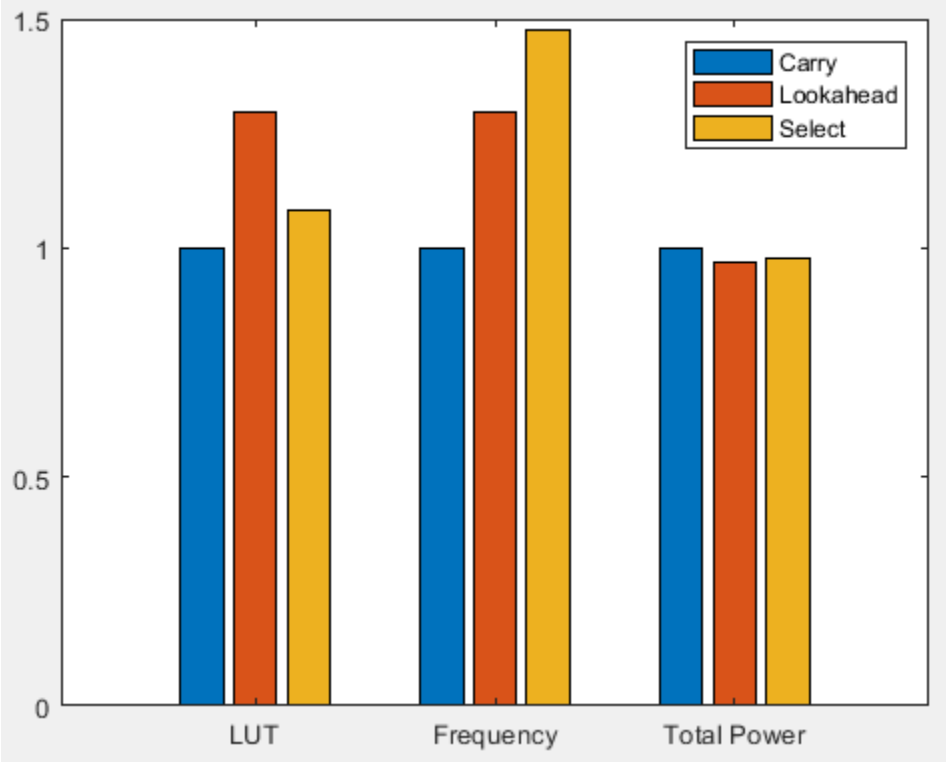
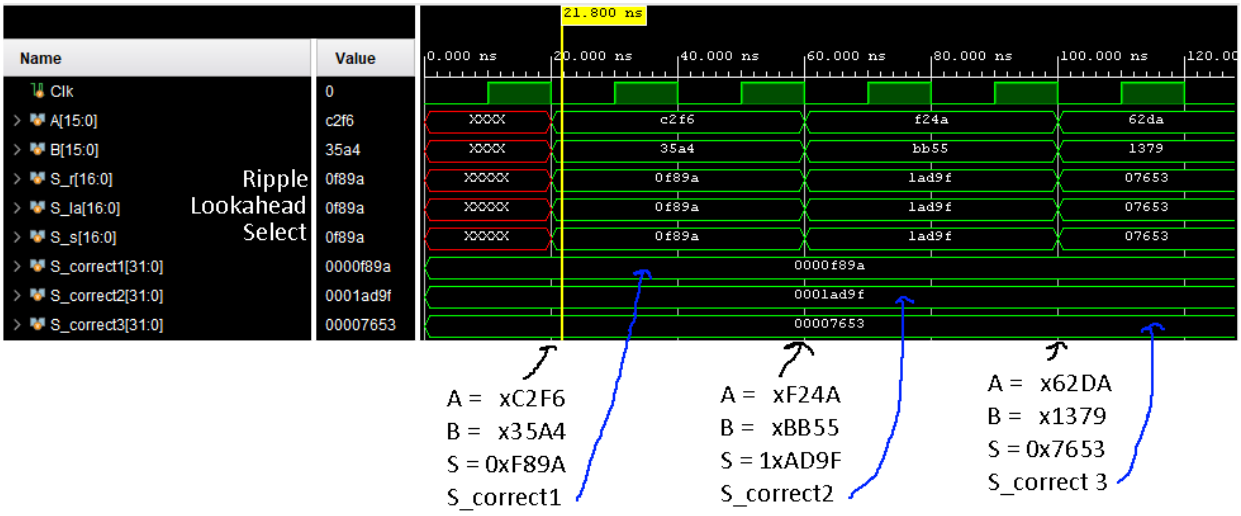


Table all remaining properties

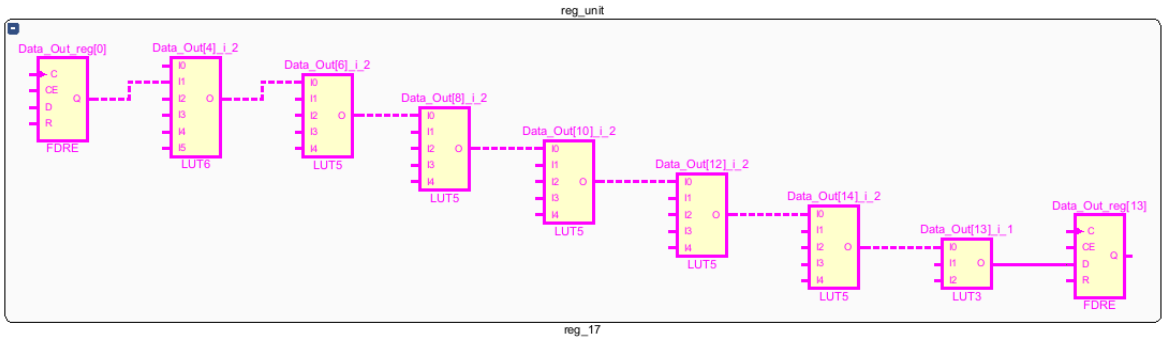
	Carry Ripple	Carry Lookahead	Carry Select
LUT	84	109	91
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flop	53	53	53
Frequency	149.254 MHz (3.300 ns WNS)	193.648 MHz (4.836 ns WNS)	220.556 MHz (5.466 ns WNS)
Static Power	0.072 W	0.072 W	0.072 W
Dynamic Power	0.023 W	0.020 W	0.021 W
Total Power	0.095 W	0.092 W	0.093 W

Annotated simulation trace



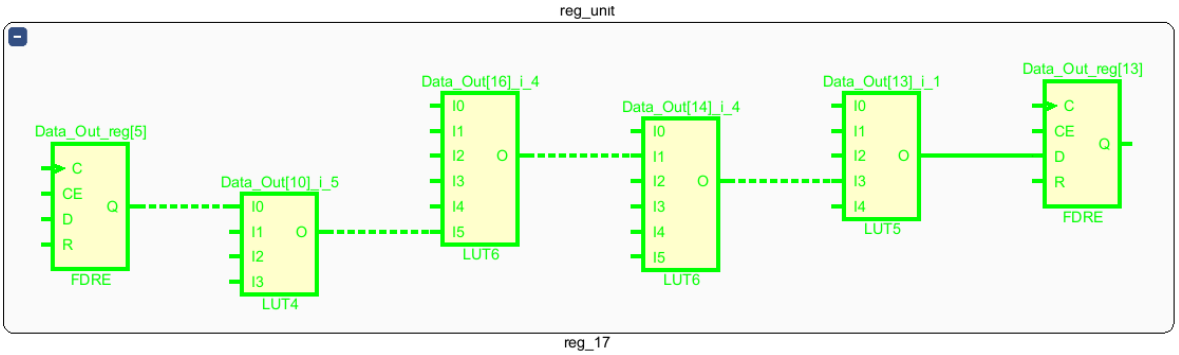
Critical path analysis

Carry Ripple Adder



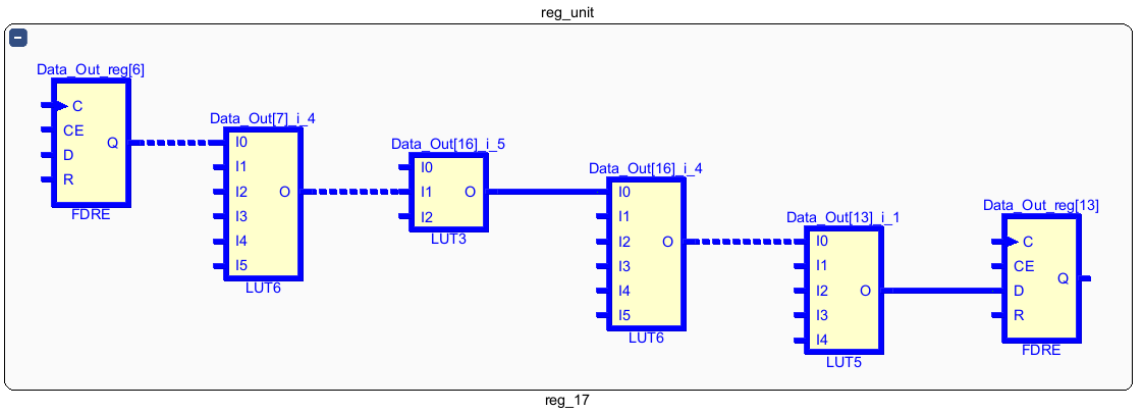
The critical path does not appear to follow our theoretical critical path, but knowing the FPGA does not actually use traditional logic gates, and that an individual LUT is able to consolidate multiple adders together, it does seem to fit our general idea as it has a long chain until it reaches its latch register.

Carry Lookahead Adder



The critical path matches up with intuition as the lookahead adder, which is faster than the ripple adder, has less LUTs in between its launch and latch register.

Carry Select Adder



The critical path matches up with intuition as the select adder is faster than the ripple adder, demonstrated by having less LUTs, and faster than the lookahead adder, represented by the fact the total LUTs are smaller for the carry select adder.

Post lab questions

Does each resource breakdown comparison from the plot make sense?

It does make sense that both the lookahead and select adders use more LUTs and are faster since they both use more logic than the ripple adder which in order to simplify the carry process. This also demonstrates why the lookahead uses more LUTs and is slower than the select since all the new combinational logic for the P and G is more complicated than just some MUXs and an extra set of adders, as well as the needing more time to ripple P and G through the group. I did not not expect power to go down, but thinking about it does make sense. Since the register is used by all the adders, the difference in power comes from the adders themselves, which are composed entirely of logic gates. Logic gates do not consume much power, as seen by the minimal decrease, but since the lookahead and select adders are able to perform faster than the ripple adder, it uses less power in the shorter time.

Are they complying with the theoretical design expectations, e.g., the maximum operating frequency of the carry-lookahead adder is higher than the carry-ripple adder?

Yes, it is as expected that the lookahead and select adders will be faster than the ripple adder.

Which design consumes more power than the other as you expected, why?

The design that consumes the most power was the ripple adder. As mentioned above, it takes the longest, and it will keep the logic gates active the longest, so it will consume the most power.

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?)

This is not ideal since, in a 4-bit carry ripple adder, it would take time for the carry bit of the internal 1-bit adder to ripple through the other 4 bits, taking longer than necessary. The ideal hierarchical select adder would have the same delay as the final carry out but to the MUX and the MUX selecting an output. To figure this out, the gate delay of input to carry out and the gate delay of an input to output for a MUX must be known

Conclusions

The only bugs encountered in this lab were some minor syntax errors in the programming that were easily resolved. The lab manual was sufficient instruction to understand the working and design process of the lab.