ECE385

Fall 2023

Experiment #3

Lab 3

Haoyu Huang Eitan Tse
Lab section: ZY 16/Day & Time: 9.25
Your TA's Name: Yang Zhou

1. Introduction

In this lab, we designed three different types of adders using SystemVerilog.
The first adder that we created was a Ripple Carry Adder. The adders consist of multiple full-adders which have inputs and outputs. They have carry-in, carry-out, input A, and B. Full-adders are connected to adjacent full-adders. The carry-out of previous adders is connected to the carry-in of another full-adder.
The Carry Lookahead Adder uses pre-computed logic expressions for propagating and generating bits (P, G, respectively. Predicting the carry-in in advance will save a lot of time waiting for the rippling of full-adders. Based on the previous carry-in input, propagating bits, and generating bits, the CLA predicts what the carry-out is.
The Carry Select Adder uses two sets of RCA adders for computing two different results with different carry-ins. Then we use a multiplexor to select between a carry-in value of either zero or on.

Ripple Carry Adder

i. Written description

The ripple adder consists of 16 full adders. We design 4-bit adders and connect four of them together. They have carry-in connected to carry-out. The former adder's cin will wait for the rippled bit, meaning carry-in will ripple through full-adders. There are three inputs for 16-bit full-adders, A[15:0], B[15:0], and cin. There are two outputs S[15:0], and cout. For 4-bit full—adders, inputs, and outputs are similar, but A, B, and S are four bits.
The logical expression for full-adder are S = A^B^Cin, and c_out = (A&B) | (A&c_in) |(B&c_in). The ripple adder calls the full adder 16 times with the previous c_out connected to the next c_in.

ii Block diagram

The n-bit ripple adder consists of n full-adders. Each full adder takes in three one-bit inputs and two one-bit outputs: A, B, and C-in are the inputs, and S and C-out are the outputs. The logic expression for S and Cout is shown in Figure 2.

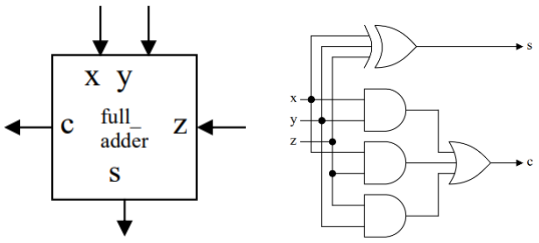


Figure 1 Full Adder

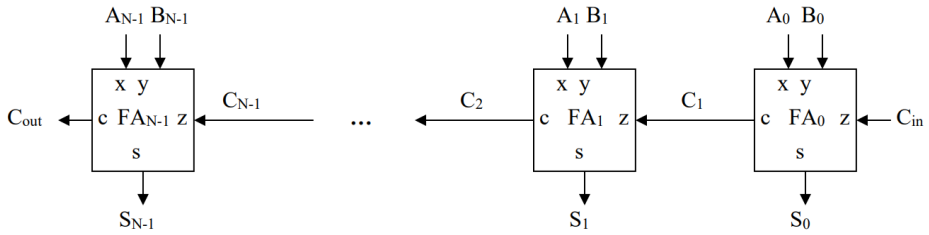
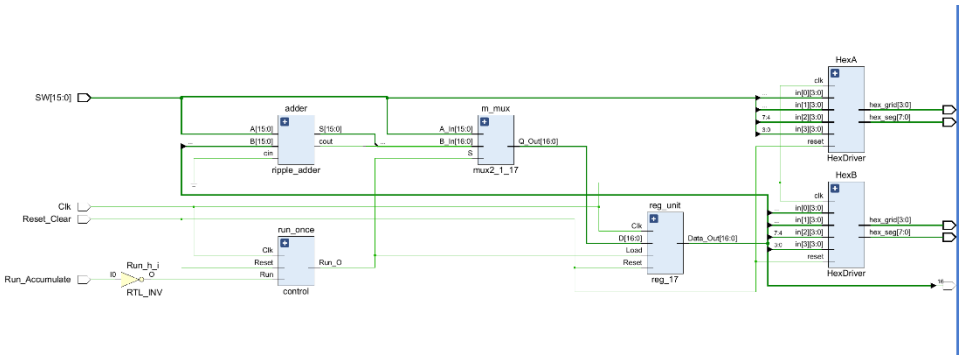


Figure 2 N-bit Ripple Adder Chain



Carry Lookahead Adder

i. Written description of the architecture of the adder

The Carry-Lookahead Adder (CLA) is implemented in 4x4-bit instead of 16-bit. CLA uses the concept of generating (G) and propagating (P) logic. The concept is that every bit of the CLA makes predictions using its immediate available inputs (A and B) and predicts what its carry-out would be for any value of its carry-in The architecture is composed of 4 4-bit CLA that each have a c_in, and a PG and a GG output. The inputs are A[15:0], B[15:0], and c_in. The outputs are S[15:0] and c_out. Each 4-bit CLA is similar to 4 1-bit full adders. We used same logic expression to derive S. The c_in inputs are calculated using combinational logic. It will depend on A, B and the original c_in. The computation will be parallel and save a lot of time. The adder is like a 4X4 full adder in how it calculates its sums, but extra logic gate to predict c_in and c_out.

ii. Describe how the P and G logic are used.

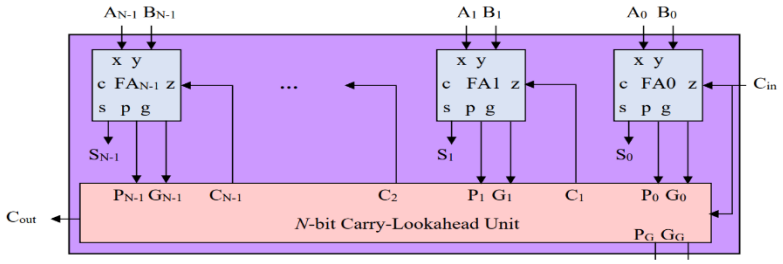
P and G signals describe whether a carry-in will be propagated or if a carry will be generated with each sum that is calculated. These logical operations for propagate and generate signals can be combined with one another. Since we have created a hierarchical 4x4 adder, the propagate and carry signals will be batched together in groups of 4 in a

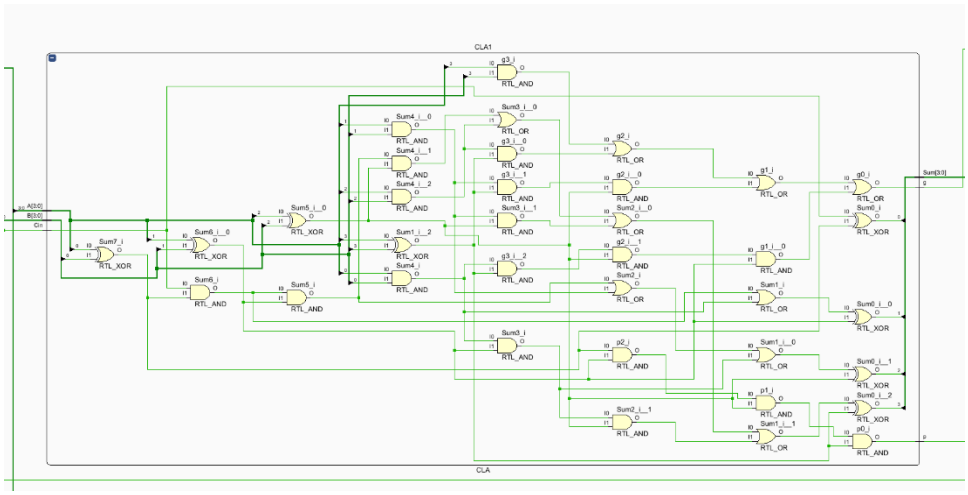
module that calculate 4 carry-ins and an overall propagate and generate signal for all 4 P and G signals that are inputted, in a 4-bit carry-lookahead unit.

iii. Describe how you created the hierarchical 4x4 adder.
As described above, the propagates and carries are done in groups of 4 bits. First, the P, G, and sum signals are calculated from the input bits. Then, they are blocked together in groups of 4 to the 4-bit carry-lookahead unit that generates a P and G signal from a carry-in and 4 P and G signals. There are four such units, one for each group of 4 bits. The resultant 4 P and G signals are fed one more time into a fifth module that generates the final P and G signal. The carry-ins to each unit and sum calculation are connected to the appropriate outputs from each module, though the lowest-level module is connected to the input carry-in of the entire adder. Finally, simple logic is conducted to generate the carry-out from the final propagate and carry signals.

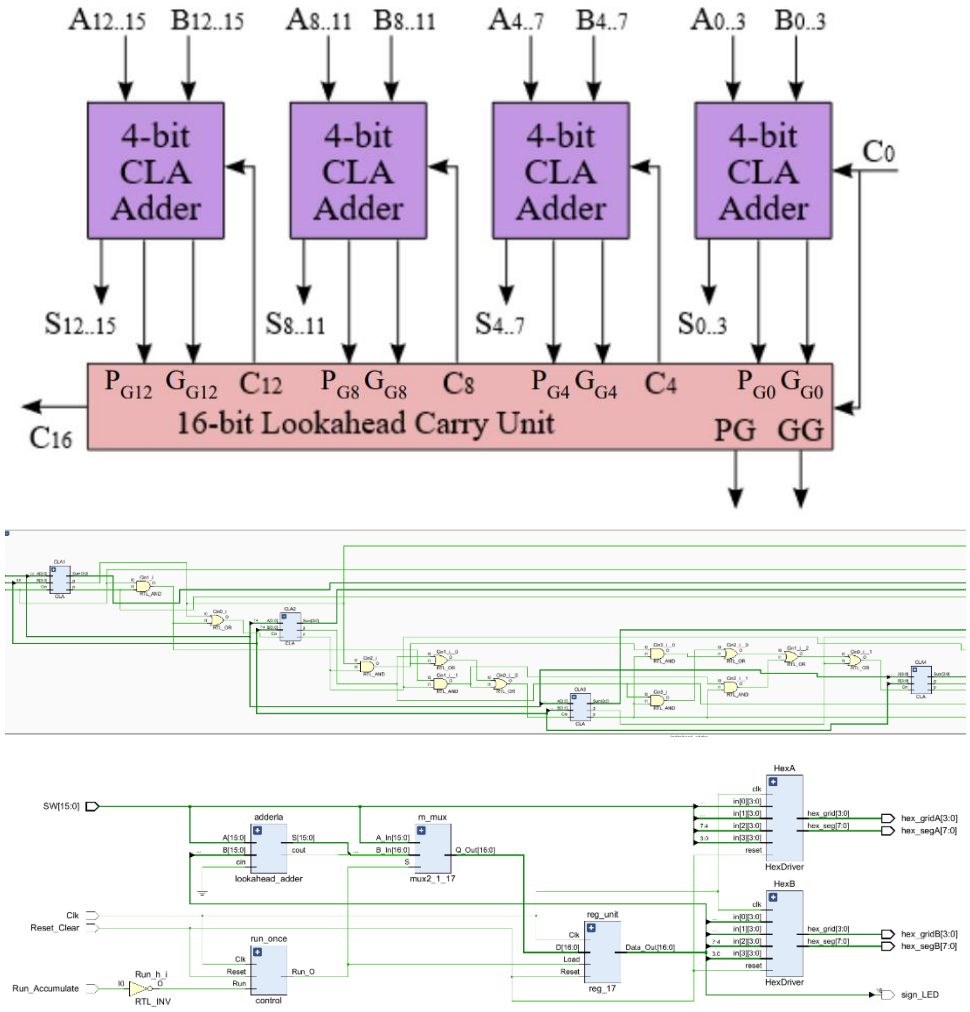
iv. Block diagram (4-bits and chain)

CLA unit:





CLA chain:



Carry Select Adder

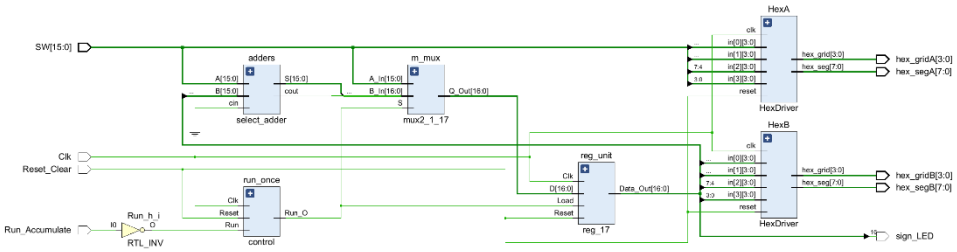
i. Written description of the architecture of the adder

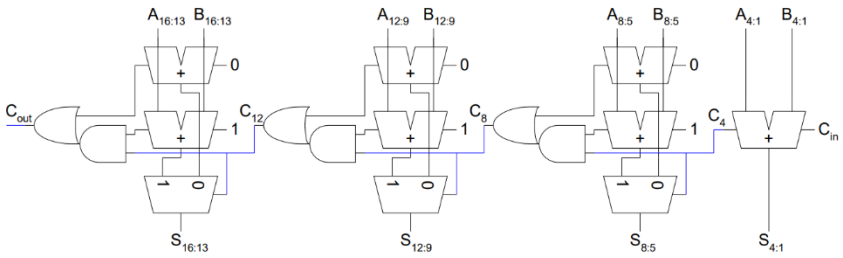
The Carry-Select Adder (CSA) is implemented in 4x4-bit instead of 16-bit. The adder uses 7 CLA units to compute the result. There are three inputs for 16-bit full-adders, A[15:0], B[15:0], and cin. There are two outputs S[15:0], and cout. CSA uses two series of computations. One assumes c_in is “1”, the other assumes c_in is “0”. Two different results will be computed in parallel and save a lot of time. Then we use MUX to select different results according to the previous result of CLA unit. If the previous unit indicates that cout is “1” then we select the result that assume cout is “1”. Besides, we simplify the first CLA unit by using only one 4-bit CLA adder. Because we already know the first c_in and we don’t need to select.

ii. Describe at a high level how the CSA speculatively computes multiple sums in parallel and rapidly chooses the correct one later. Make sure you understand this!

The CSA calculates each possible sum for each 4-bit block, and it will feed the result to a MUX. When the first adder finishes the calculation, the other will finish. But MUX logic has a delay. We need to wait for MUX to get S and C_out. The result will be fed to the next MUX. Once it decides the next c_out, it will be fed to the next MUX rapidly. Since the MUX delay is negligible compared to ripple adder, it can calculate rapidly.

iii. Block Diagram





2. Written Description of Modules

Module: ripple_adder (ripple_adder.sv)
Inputs: [15:0] A, B, cin
Output: [15:0] S, cout
Description: This module is the 16-bit ripple adder that takes in the inputs A, B, and cin The module will instantiate four 4-bit ripple adders. It will generate outpus S and cout.
Purpose: The module will instantiate 4-bit ripple adders and add them together to implement 16-bit adders.

Module: FA_4 (full_adder.sv)
Inputs: [3:0] A, [3:0] B, Cin
Outputs: Cout, [3:0] sum
Description: This module is the 4-bit ripple adder that takes in the inputs A, B, and cin The module will instantiate four 1-bit ripple adders. It will generate output sum and Cout.
Purpose: This module is used in 4x4 hierarchical Carry Select Adder and 4x4 hierarchical Carry Ripple Adder. We divide the 16-bit adder into several four bits adders in CRA and CSA.

Module: full_adder (full_adder.sv)
Inputs: A, B, Cin
Outputs: sum, Cout
Description: This module is the 1-bit ripple adder that takes in the inputs A, B, and Cin and adds them. Then it will generate the outputs sum and Cout. This is done through the logical expression $s = A \oplus B \oplus Cin$, and Cout = (a&b) | (a&c_in) | (b&c_in). Below is a truth table that displays how these functions are just adding these inputs.
Purpose: The purpose of this module is to build a one-bit adder which will be implemented in CSA and CRA.

Module: Hexdriver(Hexdriver,sv)
Inputs: Clk, Reset, [3:0] in [4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: The module takes output bits of registers in the register unit. It will change 3-bits into Hex number and display on the FPGA. The input is array type (each 4-bits) which can take in 16 bits in total. It can convert upper 4bits and lower 4bits of register to each hex display respectively. Output is 8-bits hex_seg and 4-bits hex_grid. Hex_seg is used to display numbers in LED. Every LED has 7 lights. Number can be expressed in 7 bits in LED. For example, 0 can be expressed as 11000000. Hex_seg [7] is constant 1 in this case. Since FPGA has multiple HEX displays, hex_grid will instruct the board to use specific displays.
Purpose: The module is used to display Hex numbers on FPGA boards, allowing the user to see the contents of registers A and B.

Module: mux2_1_17 (mux2_1_17.sv)
Inputs: S, [15:0] A_In, [16:0] B_In,
Outputs: [16:0] Q_Out
Description: The module is 2-1 mux. It will select values between B and sum of A and B. When S (Load signal in adder_toplevel) is high, it will select B value and load to register. When S is low, it will select sum of A and B.
Purpose: The module is instantiated in adder_toplevel. It will help register unit load value between Bval and sum of A and B

Module: control (control.sv)
Inputs: Clk, Reset, Run

Outputs: Run_O
Description: The module is a finite state machine with a counter. When Run is pressed, the state machine changes state after waiting for 2^{12} clock cycles, raises Run_O to a logic 1 for a single clock cycle, then returns to 0 and waits for the push button to be released and another 2^{12} clock cycles to pass.
Purpose: The module converts the pushbutton input to a one clock-cycle-long-event, generating a signal to load the register for only one clock cycle instead of the entire duration that the button is pushed.

Module: adder_toplevel (adder_toplevel.sv)
Inputs: Clk, Reset_Clear, Run_Accumulate, SW
Outputs: sign_LED, hex_segA, hex_segB, hex_gridA, hex_gridB
Description: It connects the FPGA's physical I/O and clock to the circuit. It also instantiates all the internal modules, and links them both to each other and the external I/O.
Purpose: It instantiates all inputs and outputs of the overall circuit, internal control modules, and links all of them internally.

Module: lookahead_adder(lookahead_adder.sv)
Input [15:0] A, [15:0] B, cin
Output [15:0] S, cout
Description: It takes the two 16-bit input numbers and generates a propagate and generate signal for each bit being summed together. These propagates and carries are grouped together in groups of 4. In these groups of 4, the 4-bit unit generates carry-ins for layers above and an overall propagate and generate from a single carry-in signal, and can generate a carry-out with additional logic. There are two layers of these 4-bit units. The first layer, taking in the propagates and generates of the 2 16 bit inputs, consists of 4 of these units. The second layer consists of just 1, taking in the propagates and generates first layer's units and generating a carry-out. The carry-ins generated are fed back to the first layer. This arrangement is a 4x4 hierarchical adder, creating a 16-bit carry-lookahead adder.
Purpose: It adds two 16-bit numbers together using a 4x4 hierarchical adder, and generates a sum and carry-out.

Module: CLA (lookahead_adder.sv)
Input [3:0] A, [3:0] B, cin
Output [3:0] S, cout, p, g
Description: It takes two 4-bit inputs and generate a propagate and generate signal for each bit. The circuit uses the same propagate and generate logic as lookahead_adder.
Purpose: The module is the sub-module for sixteen-bit lookahead adder. We use the 4x4 hierarchical design. This module is used four times to create the hierarchical design for the sixteen-bit lookahead adder.

Module: reg_17 (reg_17.sv)
Input Clk, Reset, Load, [16:0] D
Output [16:0] Data_Out
Description: The module implements the register part in block diagram. The module is a positive edge clock-triggered seventeen-bit register. When the reset signal is high, the register is initialized with zeros. When the load signal is high, D [16:0] is loaded into the register.
Purpose: The 17-bit register store A and B value. When the computation is completed, it will store the sum of A and B.
Module: select_adder (select_adder.sv)
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] S, cout
Description: The module takes two 16-bit inputs, A and B. Then add them together to S. This module implements a 4x4 hierarchical design by instantiating the 4-bit ripple adder for cin = 1 and cin = 0 for each 4-bit block. The module consists of four if-statement blocks. Each block will use previous c_out as condition. The sum will be selected depending on the condition. The cout is then used as the condition for the next 4-bit block.
Purpose: This module is used to implement CSA adders. It can compute results with different c_in and do parallel computation which saves a lot of time.

3.Tradeoffs Between Adders

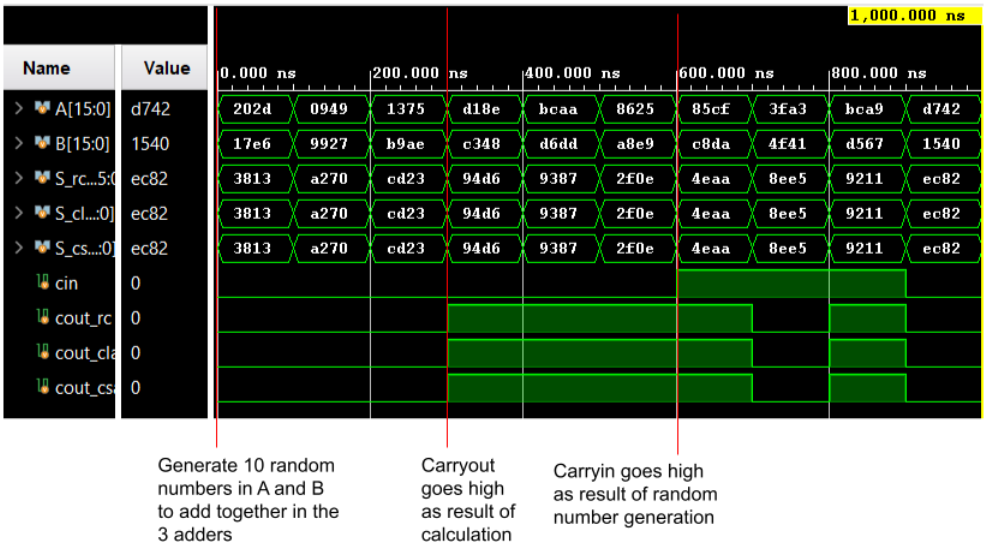
Ripple Carry Adder simply consists of a chain of full-adders. We don't need extra logic gates to connect the full adders. It uses the least number of gates and therefore it is easy to implement the adder because of the same full-adder module being used without extra logic gate to connect them. Besides, it has the least amount of area to implement the adder. However, the Ripple Carry Adder has to wait for the result of the previous adder which has the greatest computation time. The propagation time increases largely with an increase in the number of full-adders in a chain. The actual performance fits with our prediction.

Carry Lookahead Adder uses the greatest number of gates compared to the RCA and CSA. It is complicated to construct CLA adders compared to other adders. Though we have implemented 4*4 hierarchy. A great number of logic gates are required for the propagating and generating logic. Besides, it has the largest area to implement the adder. However, the problem that each adder must wait for the previous adders to propagate a c_out for carry-in is solved.

Carry Select adder uses more gate level logic due to the mux and double the number of full adders. Because it needs to compute result twice. Because of that, the area of the adder is also large. But it is easy to implement the adder since the logic expression is easy and we use 4*4 hierarchy to simplify the design. We predict that it will save a lot of time

because two pre-computing are parallel. However, its performance is not very good as CLA, which is only slightly better than CRA adder. We think the double computation may affect its performance.

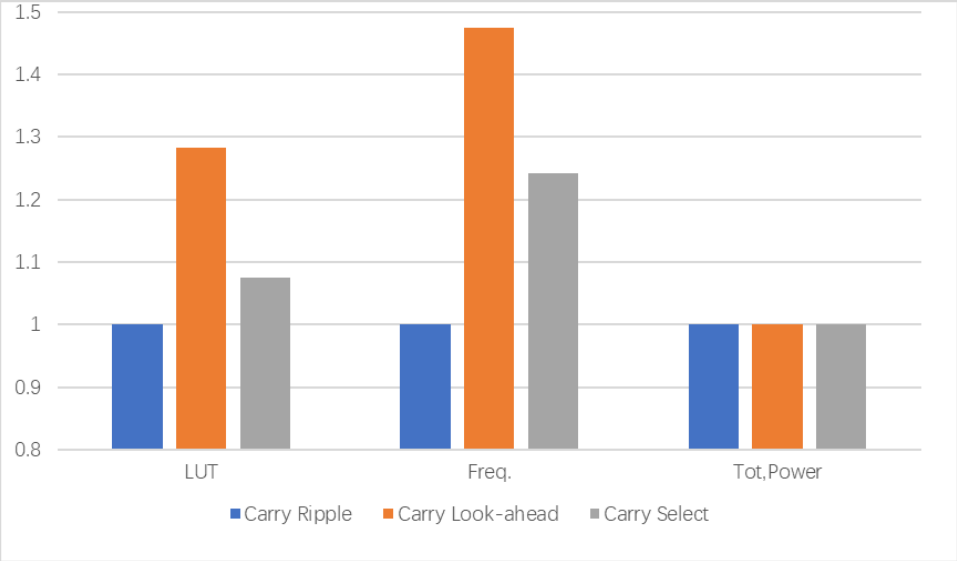
4.Annotated Simulation Waveform



5.Performance Analysis:

(After normalization)

	CRA	CLA	CSA
LUT	1	1.2826	1.076
Freq.	1	1.4742	1.2414
Tot. Power	1	1.0115	1



Annotation:

LUT makes sense. CLA has the largest LUT. CRA has the least LUT. This is because the lookahead has the largest number of logic gates.

The frequency makes sense. CLA has the highest frequency, CSA is slightly better than CRA. Extra logic gates may influence the performance of CSA.

The Power is kind of strange, they all consume the same static power, CSA computes the sum twice which takes more power than one time. The lookahead adder takes more power because it will use extra energy to get signal P and G. Extra logic gate will consume some energy. However, the result is not the same as expected. I think this is because the logic uses so little power that even though there might be big difference in between the logic power of each adder, the overall usage is so small. It doesn't make a difference to the overall.

The maximum frequency a design can run on Hardware in a given implementation = 1/(T-WNS), with WNS positive or negative. Urbana board operates at 100 MHz by default, the clock period is 10 ns. Freq= 1/(10-WNS)

	CRA	CLA	CSA
LUT	92	118	99

DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flop	53	53	53
Frequency (MHz)	148.76	219.30	184.67
Static Power(W)	0.074	0.074	0.074
Dynamic Power(W)	0.012	0.012	0.013
Total Power(W)	0.087	0.087 (E: 0.088)	0.087

6.Answers to Post-Lab Questions

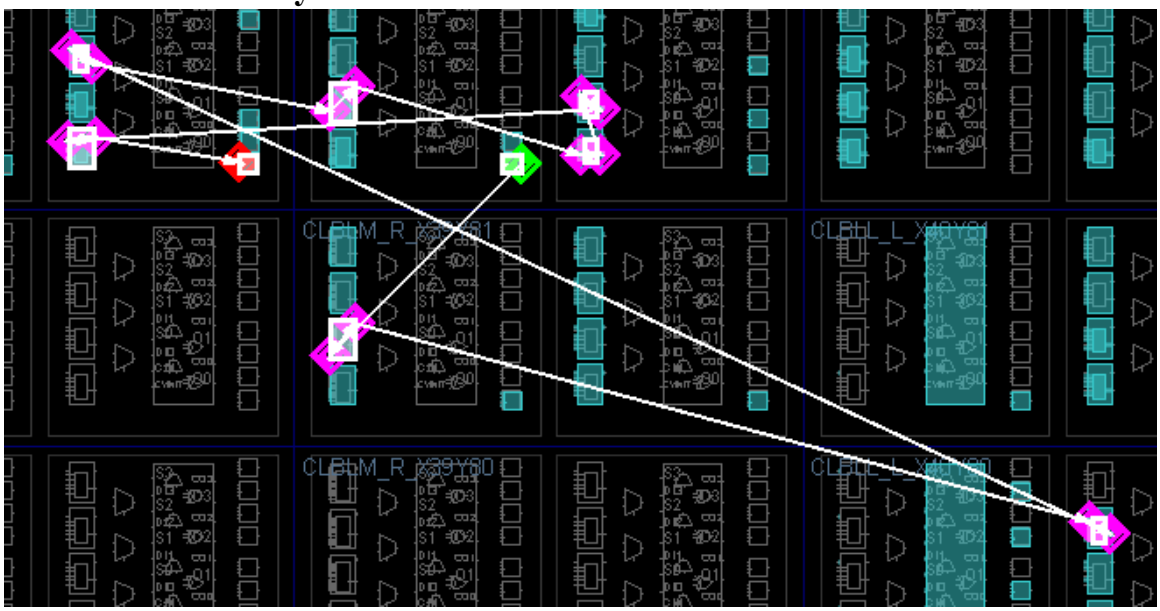
In the CSA for this lab, we asked you to create a 4x4 hierarchy. Is this ideal? If not, how would you go about designing the ideal hierarchy on the FPGA (what information would you need, what experiments would you do to figure out?)

First, the CSA is not strictly 4*4 hierarchy. We simplify the first CRA unit by using only one 4-bit Carry-Ripple adder because we already know the first C_in and we don't need to use a mux to select different values. This can reduce the total power and simplify the design.

Second, we use CRA to calculate the sum, which can be replaced by 4-bit Carry-Lookahead Adders which are faster. However, we would use more logic gates which will consume more energy and thus will produce more heat, which may influence the performance of the hardware. We need to check total power and frequency to decide whether it is practical.

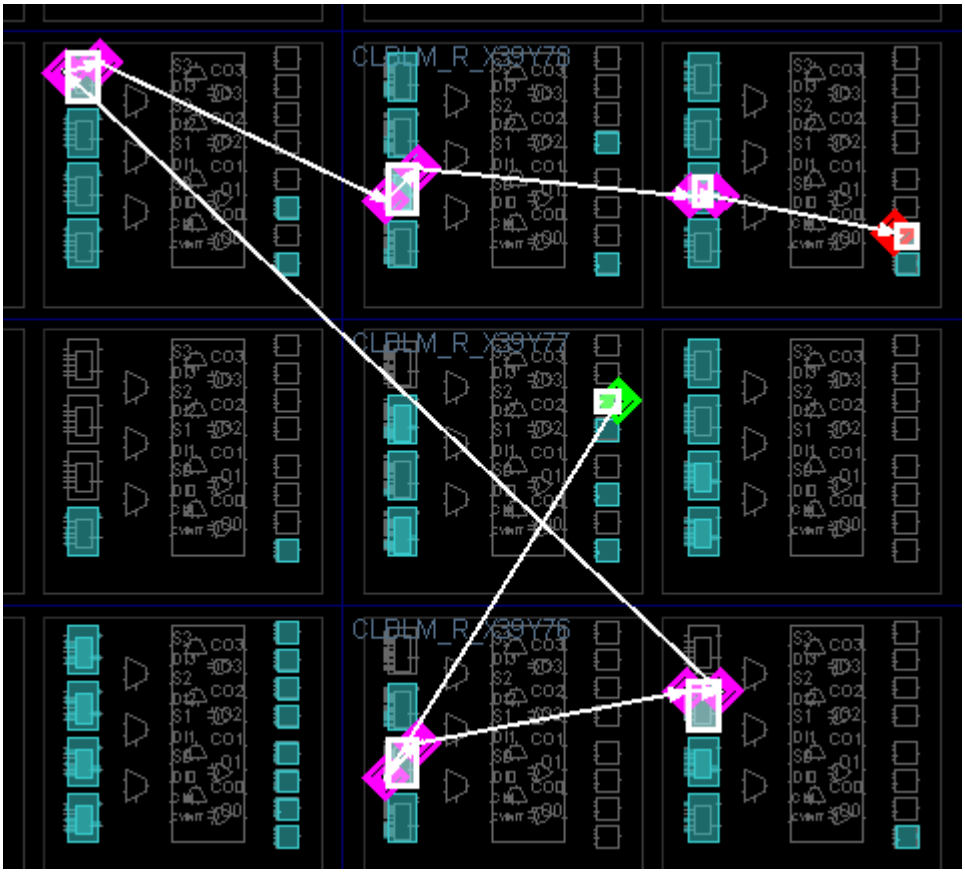
Thirdly, we should try to implement different designs like 2*8 hierarchy. We should do an experiment to test its performance.

Critical Path Analysis



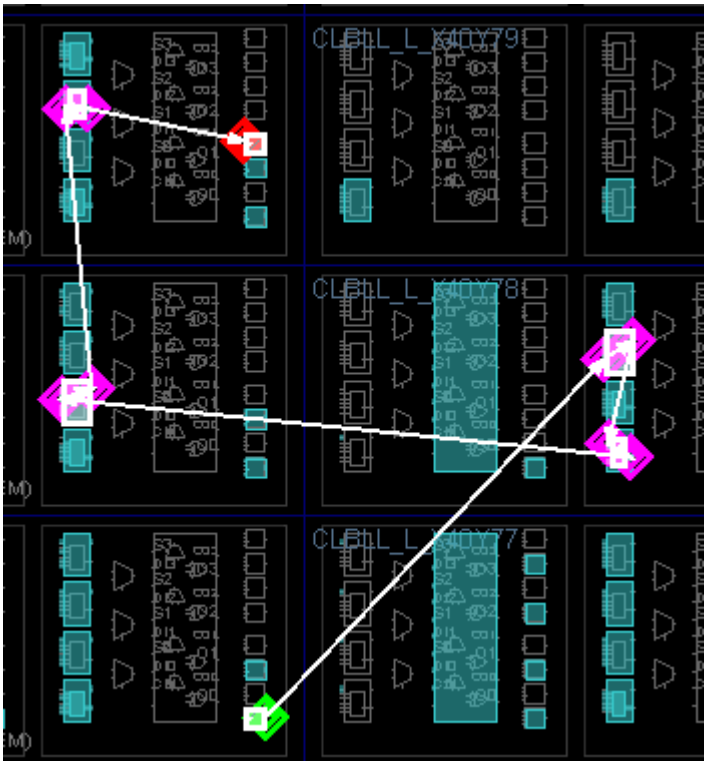
CRA

The critical path lies between the output of the Data_Out_Reg[0] and the input of Data_Out_Reg[13]. This is somewhat peculiar, as I would have anticipated the critical path to be the rippling carry, which would have stretched from Data_Out_Reg[0] to Data_Out_Reg[16]. I think this is attributable to the pathing chosen by the FPGA- there are two very long paths stretching across a large distance as is clearly visible, to that LUT. That distance could be responsible for the delay that causes it to be slower than the theoretical critical path.



CLA

The critical path of the carry-lookahead adder is between Data_Out_Reg[7] and Data_Out_Reg[15]. The theoretical critical path is the same as the ripple-carry adder- from Data_Out_Reg[0] to Data_Out_Reg[15]. Though the carry calculations are massively accelerated due to the use of propagate and generate signals, there is still a delay for it to carry from the first bit to the last bit. This is somewhat more in line with the theoretical critical path. I suspect the delay is due in part to the longer distances covered and needing the carry to propagate/generate from one propagate/generate unit to the other.



CSA

The critical path lies between Data_Out_Reg[4] and Data_Out_Reg[13]. Theoretically, the critical path would sit between Data_Out_Reg[0] and the final sum block, since the carry has to make its way through multiplexors and logic to select the requisite outputs and continue onwards through each block. The result we got lines up fairly well with this, being between the final bit of the first ripple-carry adder and the second bit of the final ripple-carry adder. The distance is quite far, which adds to the delay, but the logic has to ripple through first.

8. Conclusions

Haoyu encounters a bug when implementing CSA. At first, I didn't pay attention to the order of assignments. The behavior of simulation results is kind of strange. I fixed the bug by adjusting the order of assigning value. The problem makes me better understand <always_comb> and <assign>.

We think control.sv should be improved better. At first, we try to simulate top-level and due to the debounce counter we have to wait for a long time to see the results. We think control.sv should be explained in the lab manual. But the design of the debounce button is great. The problem that adder

sometimes adds twice is solved. But if the professor can explain a little bit about debounce design in System Verilog, that will help us.

Overall, the lab was a great introduction for students to create three different types of adders. We know the benefit of creating 4*4 hierarchy design. It's also a very important practice to be able to understand the advantages and disadvantages of using different implementations of the circuits. We get in touch with timing analysis tool and analyze the performance of different designs. Besides, we learn to do critical path analysis.