ECE385

Lab 3

Jiaming Xu
Rui Gong

Introduction

In this lab, we designed a binary adder using three methods, which are Carry-Ripple Adder (CRA), Carry-Lookahead Adder (CLA), and Carry-Select Adder (CSA).For CRA, there will be n full-adders connected in series, and a full-adder is a single-bit version of the binary adder. It takes in three binary bits (A, B, and Cin) as input and produces two outputs, single-bit sum (S) and a single-bit carry-out (Cout). CRA is simple and straightforward to design but the drawback of this adder is it will take long to do the computation. Every full adder needs to wait for their lower-bits neighbor's Cout to act as its Cin. Instead of waiting for a long time for the actual carry-in value, we can build a Carry-Lookahead Adder (CLA). It uses the concept of generating (G) and propagating (G) logic. The equation of G is G(A, B) = A*B, and the equation of P is P(A, B) = A XOR B. By have G and P we can get the function of Ci+1 = Gi + (Pi*Ci). Using this way, we can get Ci+1 in terms of Pis and Gis. In this CLA, we design a 4*4-bit hierarchical CLA. There will be two additional signals PG and GG. We can use the same calculation before in terms of PG and GG to get the carry-out bit of 4 bits. By using CLA, we will save much computation time. For Carry-Select Adder (CSA), it consists of two full adders. One adder computes the S and Cout based on the Cin is 0, and the other assumes the Cin is 1. S from both of the adders will come to a multiplexer. When the actual Cin arrives, it will select the corresponding sum. Also, there is a function in terms of Cin, Cout from one of the full adders, and Cout from the other adder to determine Cin for the next state.

Adders

a. Carry Ripple Adder

Carry Ripple Adder is the most straightforward adder compared with other adders. A Carry Ripple Adder is constructed to N 1-bit full adders connected in series though the carry-out bit and the carry-in bit. A 1-bit full adder will take three inputs: A, B, and Cin(Carry-in) and produce two outputs: Cout(Carry-out) and S(Sum).

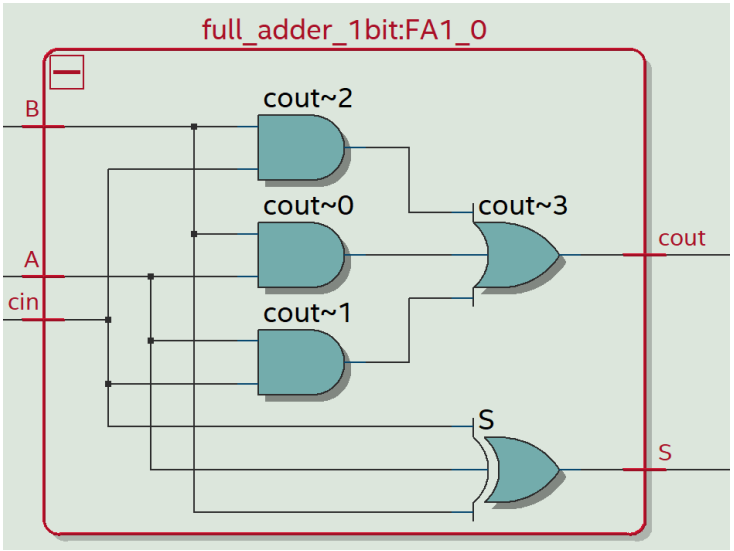
input			output	
A	B	cin	S	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

bitwise expression:

S = A^B^cin.

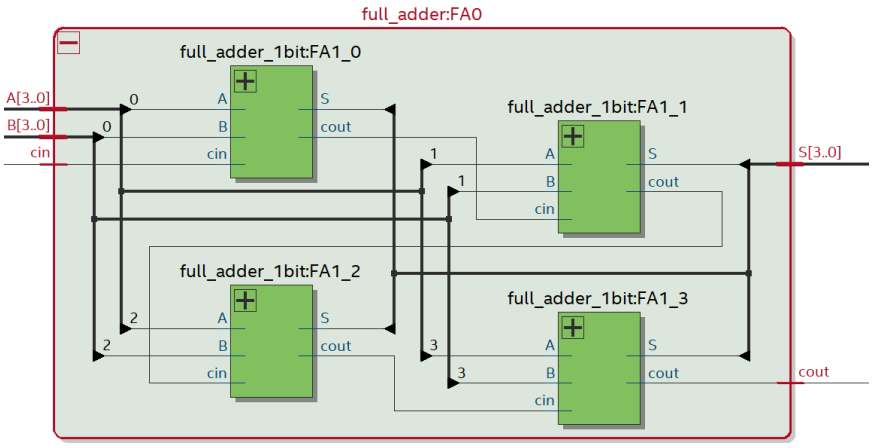
Cout = (A&B)|(A&cin)|(B&cin).

Truth table and bitwise expression for S and Cout.

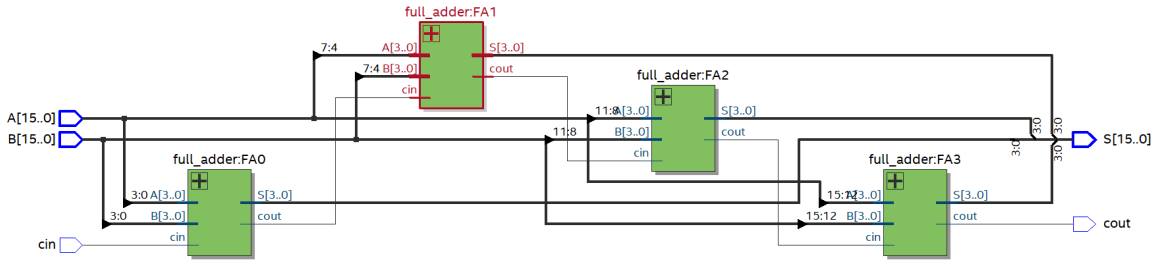


Block Diagram for 1-bit Full Adder.

The carry-out bit of the first full adder will feed into the carry-in of the second full adder, and so on till the N-th full adder. The inputs of A and B will be parallel loaded into each full adder. 4 1-bit full adder will constitute a 4-bit full adder, and 4 4-bit full adder will constitute a 16-bit Carry Ripple Adder.



Block Diagram for 4-bit full adder constituted of 4 1-bit full adder.



Block Diagram for 16-bit Carry Ripple Adder

b. Carry Lookahead Adder

The Carry Lookahead Adder utilizes two logic concepts: P(Propagate) and G(Generate) so that it could predict the carry-out bit. If both inputs A and B are 1, the carry-out bit must be 1 no matter what value the carry-in holds, so we define the G(Generate) to be A&B(A AND B).

If exactly one input of A and B is 1, the carry-in would be propagated to the carry-out, so we define P(Propagated) to be A^B(A XOR B). With P and Q, we could generate the boolean expression for Cout:

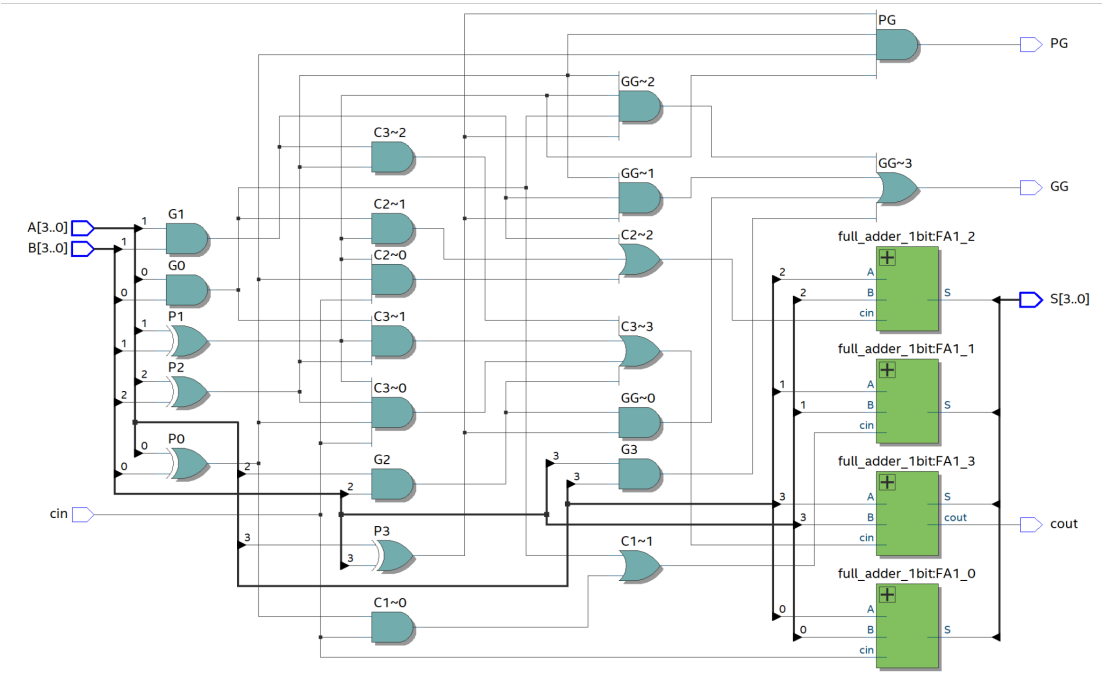
Cout = G + (P · Cin). With this boolean expression, we could generate the carry-in bits for 4 full adders, as shown in the following.

$C0 = Cin$

$C1 = Cin \cdot P0 + G0$

$C2 = Cin \cdot P0 \cdot P1 + G0 \cdot P1 + G1$

$C3 = Cin \cdot P0 \cdot P1 \cdot P2 + G0 \cdot P1 \cdot P2 + G1 \cdot P2 + G2$



Block Diagram for 4-bit Carry Lookahead Adder

Compared with Ripple Carry Adder, Carry Lookahead Adder does not have to wait for the carry-in bit from the previous full adder, which saves computation time. While using additional logic gates could also lead to increase in power consumption and areas. So in this case, to construct a 16-bit adder and avoid extremely long expressions for the carry-in bit, we first construct a 4-bit Carry Lookahead Adder and then construct a 4x4-bit Carry Lookahead

Adder with 4 4-bit Carry Lookahead Adder. Each 4-bit Carry Lookahead Adder would generate 2 logic outputs: Group Generate(GG) and Group Propagate(PG).

$PG = P0 \cdot P1 \cdot P2 \cdot P3$

$GG = G3 + G2 \cdot P3 + G1 \cdot P3 \cdot P2 + G0 \cdot P3 \cdot P2 \cdot P1$

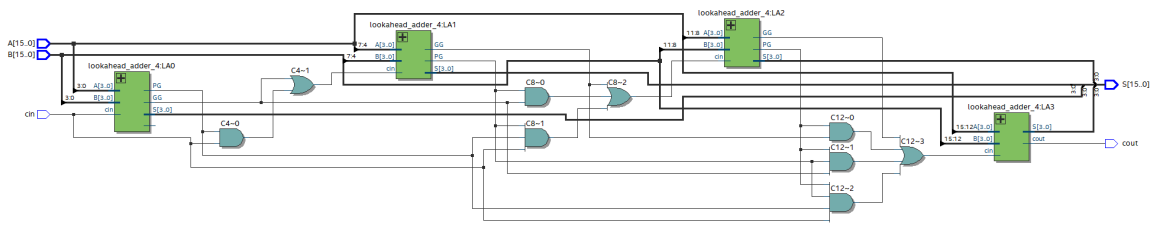
With PG and GG from the first 4-bit Carry Lookahead Adder, we could predict the carry-in bit for the next 4-bit Carry Lookahead Adder without waiting for the actual carry-out bit to come out.

$C0 = Cin$

$C4 = GG0 + C0 \cdot PG0$

$C8 = GG4 + GG0 \cdot PG4 + C0 \cdot PG0 \cdot PG4$

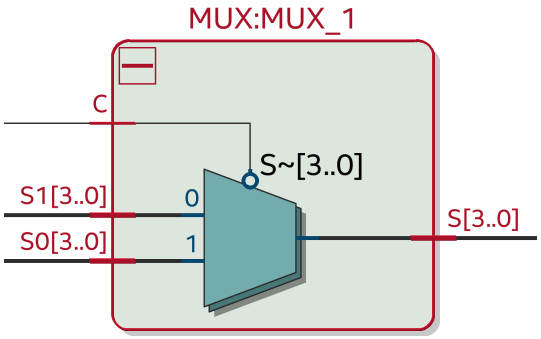
$C12 = GG8 + GG4 \cdot PG8 + GG0 \cdot PG8 \cdot PG4 + C0 \cdot PG8 \cdot PG4 \cdot PG0$



Block Diagram for 16-bit Carry Lookahead Adder

c. Carry Select Adder

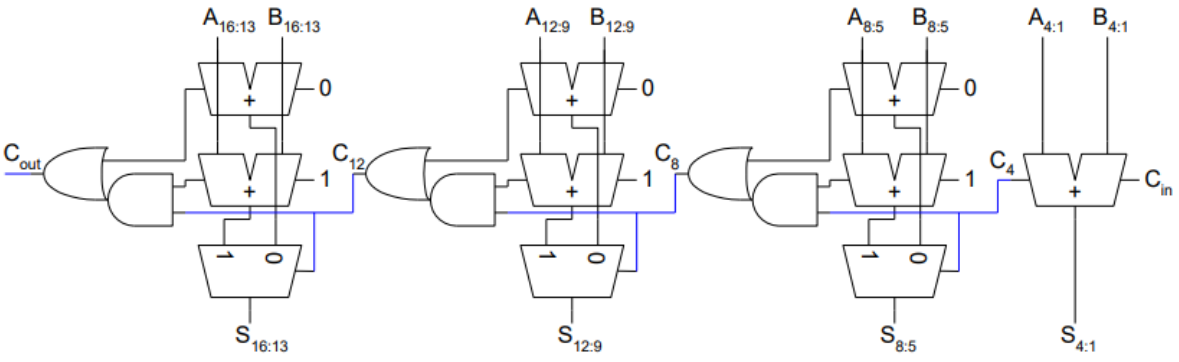
For the Carry Select Adder (CSA), there will be a 4-bit full adder at first, which will take in the first 4 bits input of A and B and Cin. For the rest of the circuit there will be two 4-bit full adders. One of them has Cin=0, and the other one has Cin=1. They both have the same 4 bits input of A and B. The sums of the two full adders will be the input of a 2 to 1 multiplexer. Cout from the previous full adder will be the selecting input for the multiplexer.

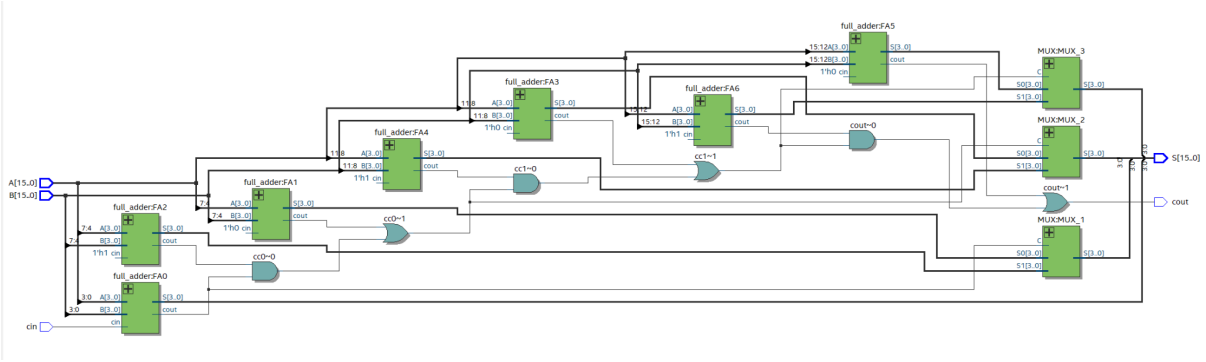


Block Diagram for MUX

If the selecting input is 0, the answer will be the sum of the full adder which Cin is 0. Otherwise, the answer will be the sum of the full adder which Cin is 1. For the Cout of the current state, there will be a Boolean function in terms of Cout from the previous state, Cout from the full adder in the current state whose Cin=0, and Cout from the full adder in the current state whose Cin=1.

Cout(current state) = (Cout(previous state) AND Cout(from the adder with Cin=1)) OR Cout(from the adder with Cin=0).





Block Diagram for Carry Select Adder

d. Description for all modules

Module: mux.sv

Inputs: [3:0] S0, [3:0] S1, C

Outputs: [3:0] S

Description: This is a function of multiplier, where the select bit C will select one of two inputs S0 and S1 and output to S. When C is low, the output will be S0, while when C is high, the output will be S1.

Purpose: This module is used to select the correct outputs based on the input carry-in bit in the Carry Select Adder.

Module: full_adder_1bit.sv

Inputs: A, B, cin

Outputs: S, cout

Description: This module implements a 1-bit full adder, It takes A, B, and Cin as inputs and produces the Sum S and Carry-out bit cout.

Purpose: This module is used as a 1-bit full adder in all three Adders.

Module: lookahead_adder_4.sv

Inputs: [3:0] A, [3:0] B, cin

Outputs: [3:0] S, cout, PG, GG

Description: This module is the 4-bit Carry Lookahead Adder. It predicts the carry-in bit with Generate G and Propagate P based on the input A and B, and produces the output sum S and carry-out bit cout by using 4 1-bit full adders. It also outputs PG and GG, which are the Group Generate and Group Propagate.

Purpose: This 4-bit Carry Lookahead Adder module is a part of the 4x4 Carry Lookahead Adder.

Module: select_adder.sv

Inputs: [15:0] A, [15:0] B, cin

Outputs: [15:0] S, cout

Description: The module will implement the 16-bit Carry Select Adder by first implementing 4-bit full adder based on two possible carry-in values, and then select the correct outputs with actual carry-in bit.

Purpose: This module is used as the Carry Select Adder.

Module: router.sv

Inputs: R, [15:0] A_In, [16:0] B_In

Outputs: [16:0] Q_Out

Description: This module is a 17-bit parallel multiplexer. It will select the inputs A_In and B_In based on select R and output to Q_Out.

Purpose: This module will put either A or B value into the designated register.

Module: ripple_adder.sv

Inputs: [15:0] A, [15:0] B, cin

Outputs: [15:0] S, cout

Description: This module is a 16-bit Carry Ripple Adder, which is composed of 4 4-bit full adders, while each 4-bit full adder is composed of 4 1-bit full adders. This module will take two 16-bit inputs and a carry-in and outputs a 16-bit sum and a carry-out bit.

Purpose: This module is the 16-bit Carry Ripple Adder.

Module: reg_17.sv

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

Outputs: [16:0] Data_Out

Description: This module is a 17-bit positive-edge triggered register with Load and Reset controls. It will load data D into Data_Out when Load is high, while reset Data_Out as 0 when Reset is high.

Purpose: This module will serve as the Register Unit and hold the value of one operator.

Module: lookahead_adder.sv

Inputs: [15:0] A, [15:0] B, cin

Outputs: [15:0] S, cout

Description: This module will implement a 4x4-bit Carry Lookahead Adder, which consists of 4 4-bit Carry Lookahead Adder. It predicts the carry-in bit with Group Generate GG and Group Propagate PG based on the input A and B, and produces the output sum S and carry-out bit cout.

Purpose: This is the Carry Lookahead Adder.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module will implement a HexDriver, which could convert a 4-bit input number in Binary into Hexadecimal.

Purpose: This module could convert a 4-bit binary number into a hexadecimal.

Module: Control.sv

Inputs: Clk, Reset, Run

Outputs: Run_O

Description: This module will implement the state machine of this program, which has three self-looping states A, B and C. Only during State B, the output Run_O will be set to high.

Purpose: This control module will generate a Load control signal, which will feed into Register Unit and Router Unit and determine when to load value.

Module: full_adder.sv

Inputs: [3:0] A, [3:0] B, cin

Outputs: [3:0] S, cout

Description: This module is a 4-bit full adder, which is composed of 4 1-bit full adders connected in series by feeding the carry-out bit of the first full adder into the second full adder.

Purpose: This 4-bit full adder module will serve as a part of the 16-bit Carry Ripple Adder and Carry Select Adder.

Module: testbench_lab3.sv

Inputs: None

Outputs: None

Description: This module is the testbench for Lab3. It will test one of three 16-bit Adders that we create at a time and compare with the correct solution.

Purpose: This module will test three Adders that we create.

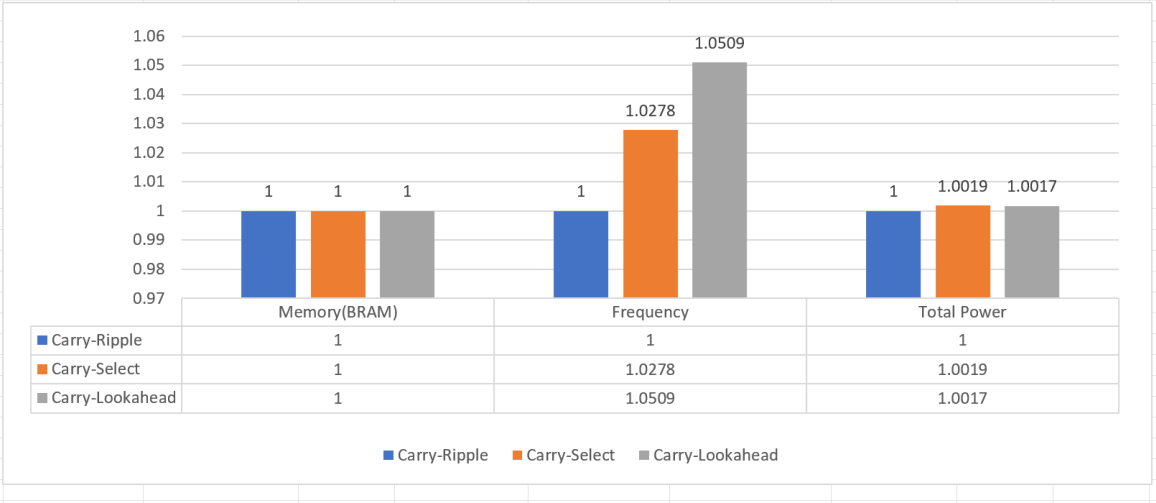
e. High level area, complexity, and performance tradeoffs between adders.

The Carry Ripple Adder is definitely the most straightforward adder with the most simple design among all three adders, since to implement a 16-bit Carry Ripple Adder, we just connect 16 1-bit full adders in series. However, each 1-bit full adder has to wait till the carry-in bit, the carry-out bit of previous 1-bit full adder, to become available, which increases computation time. To implement a 4x4-bit Carry Lookahead Adder, we need to calculate the carry-in bit for each 1-bit full adder inside the 4-bit full adder, which leads to an increase in the areas and power consumption as we have to use additional gates to implement the carry-in bit. However, Carry Lookahead Adder could save some computation time compared with Carry Ripple Adder since each 1-bit full adder does not have to wait for the outcome from previous 1-bit full adder. The Carry Select Adder has the most unique design as it will produce two versions of outcomes based on two possible carry-in values and use a multiplexer to select the correct outcome with the actual carry-in bit. With these three 16-bit adders, the computation time might not have a distinct difference, while when we increase the size of inputs to a large number, we should expect the Carry Lookahead Adder and the Carry Select Adder to be much quicker than the Carry Ripple Adder.

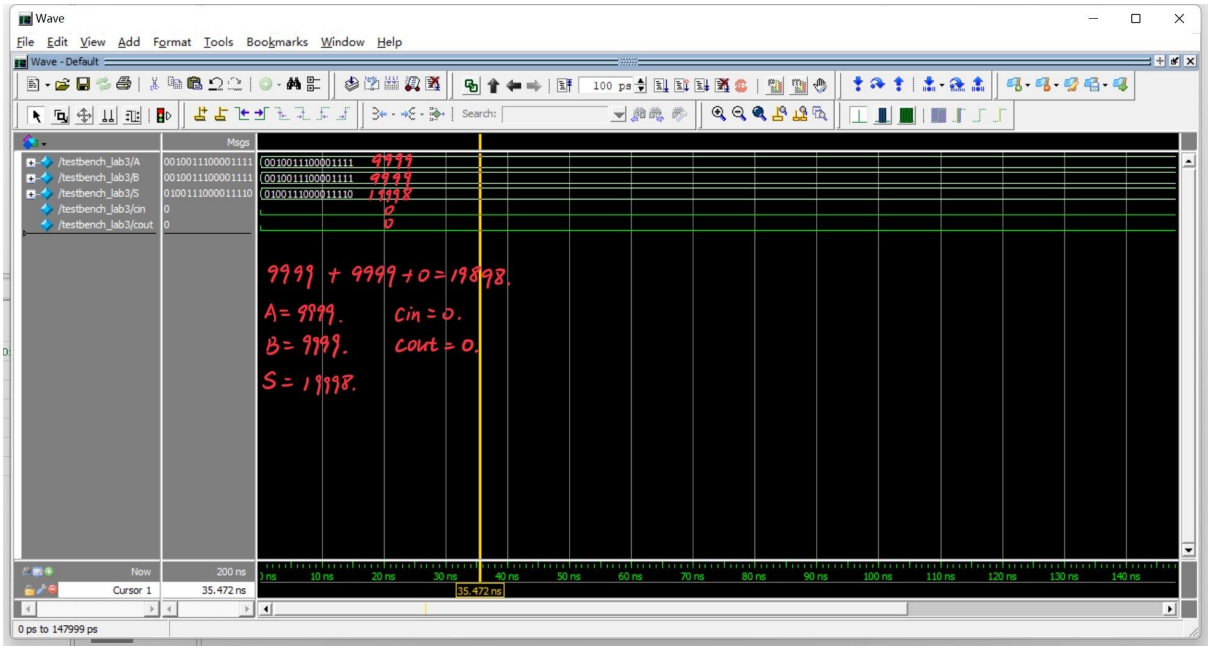
f. Graph and chart of performances of each adder

	Carry-Ripple	Carry-Select	Carry-Lookahead
Memory(BRAM)	1677312	1677312	1677312
Frequency	64.47 MHz	66.26 MHz	67.75 MHz
Total Power	105.11 mW	105.31 mW	105.29 mW

	Memory(BRAM)	Frequency	Total Power
Carry-Ripple	1	1	1
Carry-Select	1	1.0278	1.0019
Carry-Lookahead	1	1.0509	1.0017



g. Annotated simulation trace.



h. Extra Credit

Since our design for three adders are quite small, the limited IO speed of the FPGA pins would limit the design. To better illustrate this, we calculated 50 slowest paths for each adder and found the lowest path for each adder not involving starting at SW[x] and ending at HEX[x], as illustrated below.

Slowest path in Carry Ripple Adder, not involving SW or HEX:

From Node: reg_17:reg_unit|Data_Out[3]

To Node: reg_17:reg_unit|Data_Out[14]

Delay: 9.631

Slowest path in Carry Lookahead Adder, not involving SW or HEX:

From Node: reg_17:reg_unit|Data_Out[2]

To Node: reg_17:reg_unit|Data_Out[11]

Delay: 8.388

Slowest path in Carry Select Adder, not involving SW or HEX:

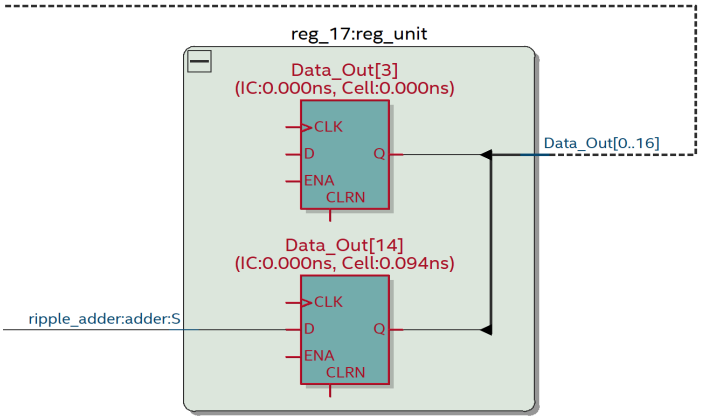
From Node: reg_17:reg_unit|Data_Out[4]

To Node: reg_17:reg_unit|Data_Out[11]

Delay: 7.229

The Carry Select Adder's lowest path has a delay time of 7.229 time unit and Carry Lookahead Adder's lowest path has a delay of 8.388 time unit, while the Carry Ripple Adder's lowest path has a delay of 9.631 time unit. This data makes sense since the Carry Ripple Adder is constructed of 16 1-bit full adders, and each 1-bit full adder has to wait for

the previous 1-bit full adder to generate the carry-out bit, and use it as the carry-in bit for itself. While for Carry Select Adder, the results are calculated based on two possible carry-in values and uses a MUX to select the correct one with the actual carry-in bit, which could largely reduce the delay time. As for the Carry Lookahead Adder, it uses P and G to predict the carry-in bit so that each unit does not have to wait for the previous unit to generate the carry-out bit, which could also reduce delay time.



Technology map viewer for Carry Ripple Adder's lowest path

Post-lab Question

1) 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. The gate delay of Cout and S is not the same. We will get the value of S first and need to wait for the value of Cout. This will be kind of a waste of time. We can redesign the hierarchy by using the combination of different bit adders, such as two three-bit adders

and two five-bit adders, instead of making four same four-bit adders.By doing this way, we can adjust the gate delay to generate S and Cout at the same time to avoid waste of time.

2) For the adders, refer to the Design Resources and Statistics in IQT.16-18 and complete the following design statistics table for each adder. This is more comprehensive than the above design analysis and is required for every SystemVerilog circuit.Observe the data plot and provide explanation to the data, i.e., does each resource breakdown comparison from the plot makes sense? 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? Which design consumes more power than the other as you expected, why?

ripple- adder

LUT	78
DSP	no DSP
Memory (BRAM)	1677312
Flip-Flop	20
Frequency	64.47
Static Power	89.97mW
Dynamic Power	1.39 mw
Total Power	105. 11 mw

lookahead adder

LUT	91
DSP	no DSP
Memory (BRAM)	1677312
Flip-Flop	20
Frequency	67.75 MHz
Static Power	89. 97 mw
Dynamic Power	1.51 mw
Total Power	105.29 mw

Select adder

LUT	82
DSP	no DSP
Memory (BRAM)	1677312
Flip-Flop	20
Frequency	66.26 MHz
Static Power	89.97 mw
Dynamic Power	1.59 mW
Total Power	105.31 mW

Look-up table (LUT) is the numbers of combinational with no register logic element and combinational with register logic element. In this experiment, we used 78 LUT for ripple adder, 91 LUT for lookahead adder, and 82 LUT for select adder. DSP is Altera's built-in digital signal processing. All of the adders we built do not utilize DSP. BRAM is block memory. It represents total block memory implementation bits. For all adders, they use 1677312 bits. Flip-flop is the total number of registers. All of them use 20 registers. Frequency is the maximum possible operating frequency of the design. Ripple-adder has a frequency of 64.47MHz. Lookahead adder has a frequency of 67.75MHz.Select adder has a frequency of 66.26MHz. By seeing the data above, each resource breakdown comparison makes sense. Both select adder and lookahead adder compute faster than ripple adder, so the frequency of the two adders is bigger than that of ripple adder. Also, both of them have more LUT than ripple adder, which also makes sense because they need to do more computation. Select adder consumes more power because it uses almost twice the numbers of adders than the ripple adder. It will cost more power to do calculations.

Conclusion

In this lab, we built a binary adder in three different ways by using system Verilog. During the process of doing the lab, we met some problems and bugs. Most of the bugs are related syntax errors. Because we were not very familiar with the system Verilog at first, we made

many syntax errors and it took us some time to fix them. There was not any task that made us feel really difficult or ambiguous. I think writing the adder in three ways can make us know different ways of the full adder and know the advantages and drawbacks of each adder. Also, we can be more familiar with system Verilog by doing the tasks.