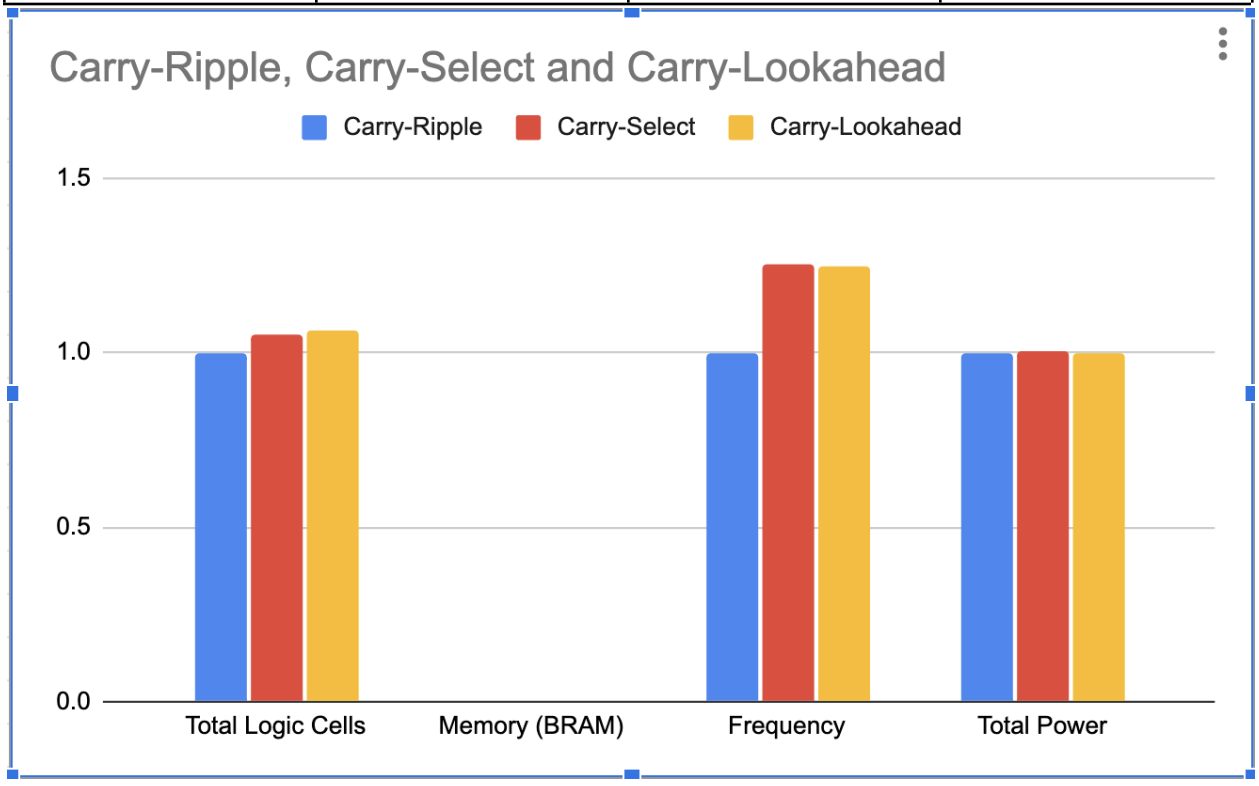
Lab 3 Report

Noah Conner & Adam Naboulsi
Section ABP

Prelab:

Document design analysis for the three adders in the table below. Plot out the data from the table for comparison studies. Normalize the data across the three adders with the carry-ripple adder. When normalizing, choose data from one the carry-ripple adder as the baseline, and then divide the other two with the baseline number. Say, you got 20 from carry-ripple, 21 from carry-select, and 23 from carry-lookahead, the numbers after normalization becomes 20/20=1.0, 21/20=1.05, 23/20=1.15, respectively. The resulting plot should resemble the one below (the plot below does not use real data).

	Carry-Ripple	Carry-Select	Carry-Lookahead
Total Logic Cells	1.000 (78)	1.051 (82)	1.064 (83)
Memory (BRAM)	0.000 (0)	0.000 (0)	0.000 (0)
Frequency	1.000 (78.44Mhz)	1.254 (98.37Mhz)	1.249 (97.98Mhz)
Total Power	1.000 (105.11mW)	1.002 (105.31mW)	1.000 (105.10mW)



Report:

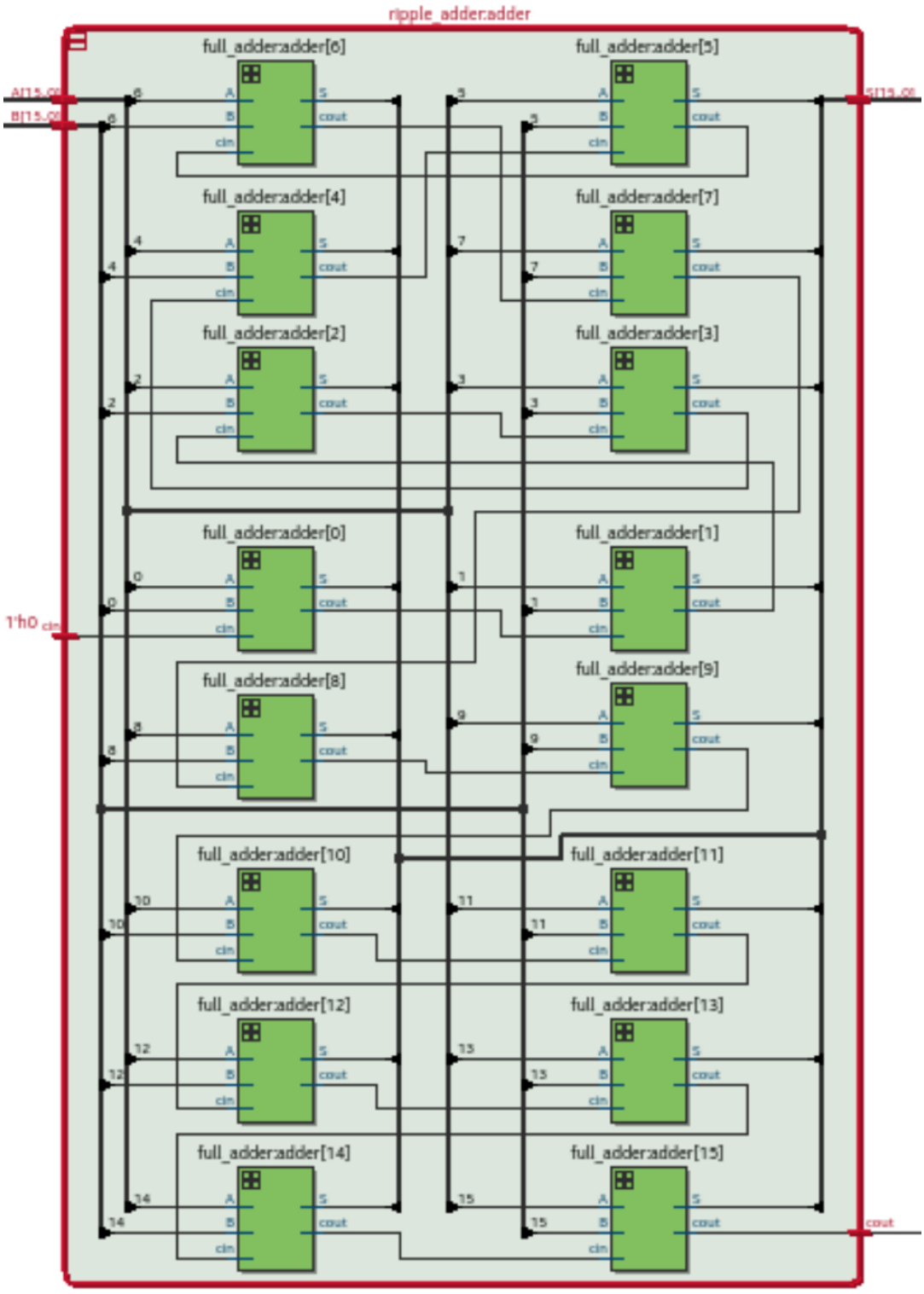
Introduction

Our serial logic processor computes the sum of two 16-bit numbers. There are three separate implementations of the adders, all of which have their advantages and disadvantages. While they all perform very similarly in this processor, the differences could be very prominent in larger designs. The first adder is a ripple carry adder which computes the sum of a 16-bit number by daisy chaining sixteen full adders together. The next adder is a carry lookahead adder which computes the sum in a similar fashion except there is a unit which computes the carry-out bits in constant time. The final adder is a carry select adder which utilizes almost twice the number of full adders to precompute the sum of both possible carry-in values; the correct sum is then immediately available once the true carry-in bit is available. We have analyzed data from all three adders and have outlined their differences and similarities in this lab report.

Adders

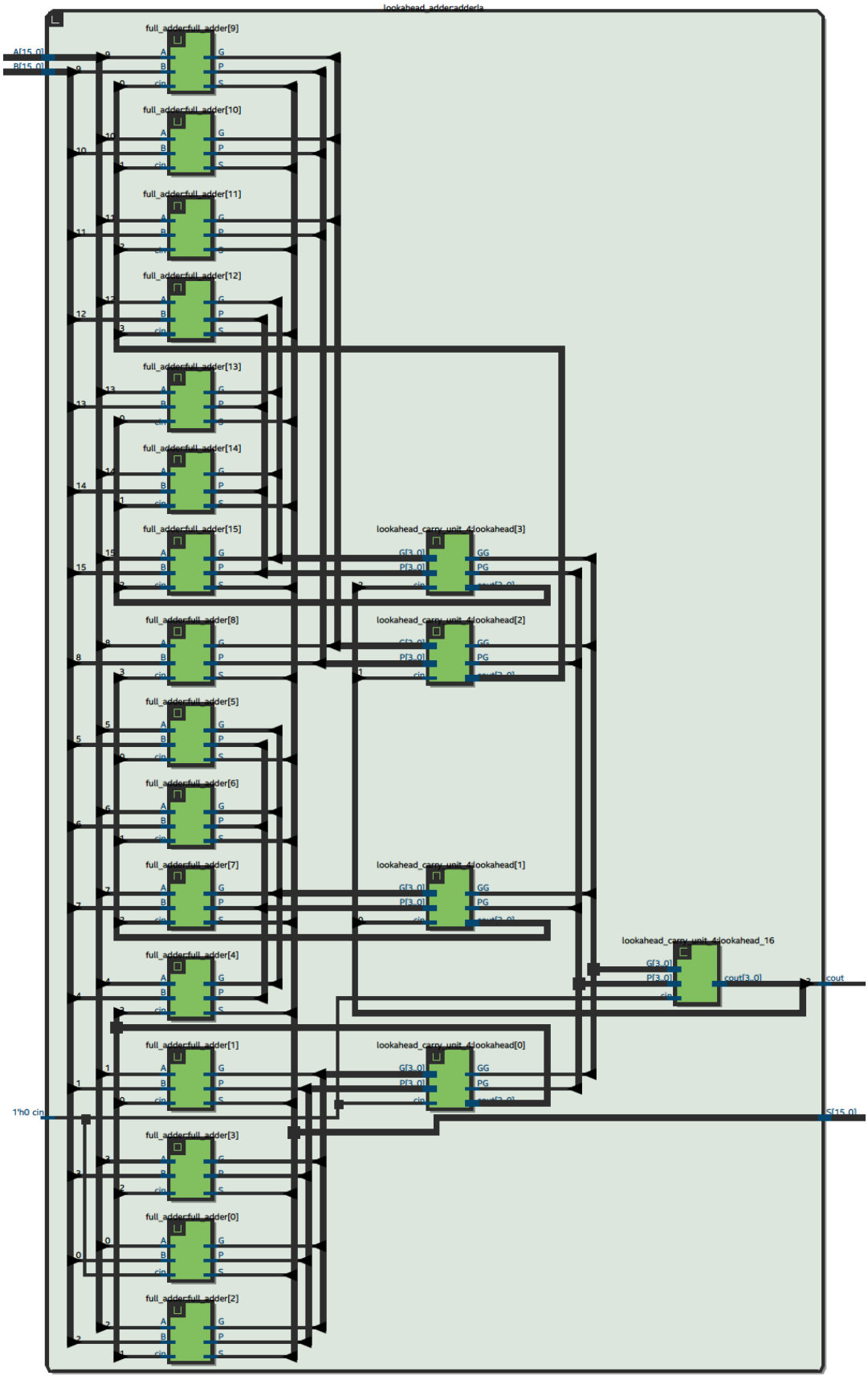
Ripple Carry Adder

The ripple carry adder is the simplest yet highest latency implementation of the three adders. This adder utilizes 16 full adders to create a 16-bit adder. The full adders are arranged in series with the carry-out bit of the previous full adder going into the carry-in bit of the next full adder. The first full adder has a carry-in bit equal to zero and the final full adder's carry-out bit is an output to the ripple carry adder. This simple architecture requires very little resources, which can be useful for projects with little resources available.


ripple_adder:adder
full_adder:adder[6]
full_adder:adder[5]
full_adder:adder[4]
full_adder:adder[7]
full_adder:adder[2]
full_adder:adder[3]
full_adder:adder[0]
full_adder:adder[1]
full_adder:adder[8]
full_adder:adder[9]
full_adder:adder[10]
full_adder:adder[11]
full_adder:adder[12]
full_adder:adder[13]
full_adder:adder[14]
full_adder:adder[15]
A[15..0]
B[15..0]
S[15..0]
1'h0
cin
cout
A
B
S
cout
cin

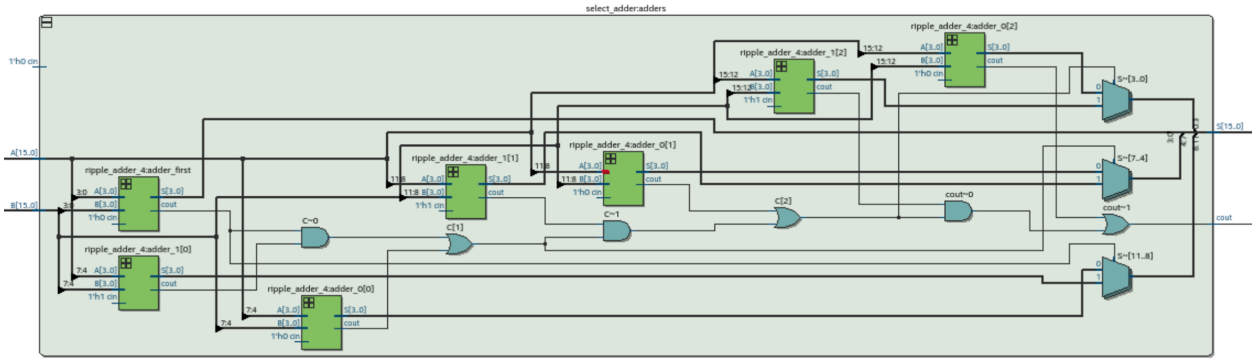
Carry Lookahead Adder

The carry lookahead adder is the most expensive implementation of the three adders. In addition to 16 full adders, the CLA also needs 5 lookahead units. In order to minimize the amount of resources, we implemented a 4x4 hierarchical design. The 16-bit CLA was created using four 4-bit CLAs and a lookahead unit. This implementation retains the low latency of a typical CLA while reducing the hardware drastically. In order for this design to work, we first had to create the 4-bit CLAs. They are made up of four full adders that have additional outputs bits P and G, propagation and generation, and one lookahead unit that takes these P and G bits and produces a carry-out value for that full adder without the latency of the full adder itself. This continues for the remaining full adders. The P bit tells the lookahead unit if there is a chance of having a carry-out bit and is the XOR of inputs A and B. The G bit works with the P bit to determine if there will be a carry-out bit and is the OR of inputs A and B. Since these bits are independent of the carry-in bit, so they can be calculated in constant time. In addition to the P and G bits, the lookahead unit must also know the initial carry-in bit which is immediately available. Therefore, the lookahead unit only relies on P, G and the initial carry-in and can calculate the carry-out bits in constant time. This design achieves constant time complexity but requires far more resources because the carry-out logic gets more complex as the number of bits increases. To counteract this, a hierarchical design was implemented. Each of the four CLAs can be seen as a full adder in the 4-bit CLA design. These CLAs have outputs PG, propagation group, and GG, generation group, that act as the P and G bits of the full adders. These bits are then fed into another lookahead unit in the same fashion as before. This hierarchical design allows for a much more scalable architecture.


lookahead_adder:adder1a
full_adder:full_adder[9]
full_adder:full_adder[10]
full_adder:full_adder[11]
full_adder:full_adder[12]
full_adder:full_adder[13]
full_adder:full_adder[14]
full_adder:full_adder[15]
full_adder:full_adder[8]
full_adder:full_adder[5]
full_adder:full_adder[6]
full_adder:full_adder[7]
full_adder:full_adder[4]
full_adder:full_adder[1]
full_adder:full_adder[3]
full_adder:full_adder[0]
full_adder:full_adder[2]
lookahead_carry_unit_4:lookahead[3]
lookahead_carry_unit_4:lookahead[2]
lookahead_carry_unit_4:lookahead[1]
lookahead_carry_unit_4:lookahead[0]
lookahead_carry_unit_4:lookahead_16
A[15..0]
B[15..0]
1'h0 cin
cout
S[15..0]

Carry Select Adder

The carry select adder uses another method to speed up the carry computation. It is made up of seven 4-bit carry ripple adders and 3 MUXes. We implemented it with a 4x4-bit hierarchical structure in which we designate two CRAs for each group of 4-bit inputs; one calculates the sum output with the carry-in bit assumed to be 0, and the other with the carry-in bit assumed to be 1. Each group computes the sums in parallel because they are independent of the carry-out bit of the previous group. Once the carry-out bits of all groups are available (happens in parallel), the correct sum and carry-out bit can be selected via MUXes and the previous carry-out. We do not need a MUX for the least significant group of bits because the carry-in bit is immediately available, thus there is no need for precomputation and selection. This design allows both possible outcomes to be precomputed, such that once the actual carry-in bit is available, the correct sum and carry-out can be selected with no delay from groups.



Module Descriptions:

Module: Full_Adder.sv

Inputs: A, B, Cin

Outputs: S, Cout, P, G

Description: Single-bit adder with carry-out and sum; uses combinational logic to compute cout, and also computes the Propagate (P = A^B) and Generate (G = A&B) signals as outputs.

Purpose: Used as the basic building block for the Ripple Adder, the Lookahead Adder, and the Select Adder. Chaining multiple full adders allows for computation with larger numbers (more bits).

Module: Ripple_Adder_4.sv

Inputs: [3:0] A, [3:0] B, Cin
Outputs: [3:0] S, Cout

Description: A 4-bit adder made up of 4 instantiated full adders, and some internal logic – wiring between carry-out of each individual full adder's cout, and the cin of the following consecutive full adder. The full adder corresponding to the most significant bit outputs the carry-out for the ripple adder as a unit.

Purpose: Used to add two 4-bit inputs, A and B, outputting the sum to S and the carryout bit, Cout; also as the building block for the larger 16-bit ripple adder.

Module: Ripple_Adder_16.sv

Inputs: [16:0] A, [16:0] B, Cin
Outputs: [16:0] S, Cout

Description: A 16-bit adder made up of 4 instantiated (4-bit) ripple adders, and some internal logic – wiring between carry-out of each individual 4-bit ripple adder, and the cin of the following consecutive 4-bit ripple adder. The ripple adder corresponding to the most significant bit outputs the carry-out for the overall 16-bit ripple adder.

Purpose: Used to add two 16-bit inputs, A and B, outputting the sum to S and the carryout bit, Cout.

Module: LookAhead_Carry_Unit_4.sv

Inputs: [3:0] P, [3:0] G, Cin
Outputs: [3:0] Cout, PG, GG

Description: This module takes in the P and G outputs from 4 ripple adders and calculates and outputs the carry outs that are to be inputted into the following consecutive full adders as carry-ins. It also computes and outputs the overall/group Propagate (PG) and Generate (GG) signals of the P and G inputs from the adder that it is passing in those values.

Purpose: Used in each individual 4-bit Carry-Lookahead Adder (with the P and G inputs coming from 4 full-adders), and once for the overall 16 bit Carry-LookAhead Adder, but instead of the inputs coming in from full-adders, the inputs are group Propagates and Generates (PGs and GGs). In both cases, this unit is used to compute the carry ins for the chained/consecutive Full-Adders/CLAs using the almost immediately produced P and G

(or PG and GG) signals, essentially "looking ahead" rather than waiting for each individual carry in based on the initial to be computed.

Module: LookAhead_Adder.sv

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

Description: This 16-bit LookAhead_Adder instantiates 16 full_adders where the 4-bit full-adder segments each have their carry-ins computed through the 4 instantiated lookahead carry units (with the P and G signals), and the 4 (higher level view) CLA adders have their carry-ins computed through an additional instantiated lookahead carry unit (with the PG and GG signals).

Purpose: The purpose of this LookAhead_Adder is to assemble the full 16-bit carry-lookahead adder unit through connecting the full-adder and lookahead modules. This variation of a 16-bit adder reduces the computation time through parallelizing the computation of the cout bits through the P, G, PG, and GG signals.

Module: Select_Adder.sv

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

Description: This module creates a 16-bit Select Adder through instantiating 3 4-bit Ripple Adders with a given carry-in value (0), 3 4-bit Ripple Adders with a given carry-in value (1), and a single 4-bit Ripple Adder with carry-in wired to input Cin. The Single Ripple Adder corresponds to the least 4 significant bits of the 16-bit output; and the 3 following 4-bit segments of the output have one Ripple adder from each group of ripple adders with carry-in 1 and 0. When carry-in is fed into the single Ripple Adder, the always_comb selects the correct 4-bit output for each group of two ripple adders (with 0 and 1 carry ins) corresponding to the carry-in. It also computes the overall carry-out (Cout) of the Select Adder through combinational logic based on the outputs of the individual 4-bit Ripple Adders.

Purpose: The purpose of this module is to assemble a 16-bit Select Adder through the use of Ripple Adders and MUXs (represented through always_comb if statements in our code). This takes on another method to speed up the computation of adding two 16-bit signals.

Module: Reg_17.sv

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

Description: This module represents a 17-bit register which sets the output of the register to zeroes when Reset is pressed and loads D into the register when the load button is pressed.

Purpose: This module is used to hold the value of one operator in the logic processor.

Module: HexDriver.sv

Inputs: [3:0] In0
Outputs: [6:0] Out0

Description: Based on the inputs, a switch statement is used to map the outputs to the corresponding LED segments (7 total) on the HEX display of the FPGA board.

Purpose: Used to output bits of registers in the register unit to the HEX displays on the FPGA board.

Module: Router.sv

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

Description: This module acts as a 17-bit parallel MUX through the use of case statements in which the output bits Q_Out are dependent on the value of input R.

Purpose: Used as a mux that puts either sum of A and B or B into the register router route in the control unit of the logic processor.

Module: Control.sv

Inputs: Clk, Reset, Run
Outputs: Run_O

Description: The control module consists of three states where transitions to the next state are synchronous with the clock and outputs (Run_O) are assigned based on the current state. It also takes in two signals: Run which tells the processor to continue transitioning from state to state, and Reset which tells the processor to return to the initial state (A).

Purpose: The control module controls and acts as the state machine of the processor, assigning outputs based on the current state and handling signaling transitions between states.

Module: Adder2.sv

Inputs: Clk, Reset_Clear, Run_Accumulate, [9:0] SW
Outputs: [9:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5

Description: This module connected the pieces of our logic processor; it instantiated the major modules within our processor's design including the control unit, the router, the register unit, the Hex Drivers, and the various addition units – with the necessary internal logic needed/used by the other modules. Lastly, it assigns the LED output bits to the desired input values.

Purpose: This module served as the top level of our design, connecting the various modules to form our serial logic processor.

Area, Complexity, Performance Trade-offs of the Adders:

With the ripple carry adder, the major advantage is the minimal materials/resources required to build a fully functional adder. Due to its serial design, the computation time of the ripple adder is quite long in comparison to the other methods implemented in this lab that involve parallelizing the process. The high latency becomes more prominent with increasing number of bits.

With the lookahead adder, we parallelize the propagation process with the carry-ins and carry-outs by implementing P and G bits from each full adder that are independent of the carry-in data and can be computed from solely the inputs. These signals are then fed into lookahead units which output the carry-ins in a parallel fashion. Using a 4x4 hierarchical design, we are able to complete the design for the 16-bit carry lookahead adder. This parallelization by introducing the two new signals and implementing lookahead units requires much more resources than needed with the carry ripple adder. The major advantage of this lookahead design is the speed with which the outputs are computed due to the parallel process.

With the carry select adder, we again parallelize the process but in a different fashion; we split up the design into 4-bit (serially computed) segments made up of two ripple adders each with a 2-1 MUX excluding the lowest 4-bit segment which is only made up of one ripple adder since it's the segment that receives the carry in directly. For the segments with two ripple adders, one is given an assumed carry-in value of 1 and the other an assumed carry-in value of 0, such that when the correct carry-in is available, the corresponding output is selected through the MUX and outputted. This design also requires more resources than the ripple adder due to the extra

4-bit ripple units needed for precomputation purposes. This, again, allows for a lower latency computation in comparison to the ripple carry adder. In addition, the carry select adder is also slightly faster than the carry lookahead adder due to a slightly shorter critical path.

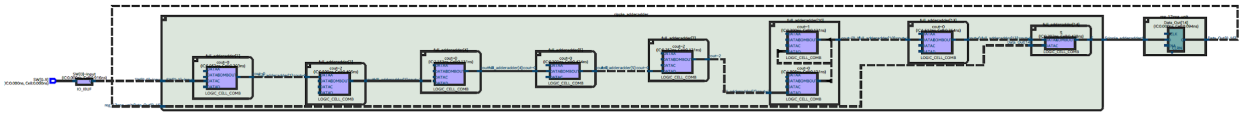
Extra Credit:

As shown in the figures below, the critical path analysis of each adder uncovers which adder is fastest. The ripple carry adder has a data arrival time of 16.130 nanoseconds. Compared to the other adders, this is very high. In fact, the path through the ripple carry adder was the critical path of the whole design while the critical paths of the carry lookahead and carry select adder designs were not the adders but instead IO pins. This data agrees with our theoretical understanding of each adder. Theoretically, the carry select adder should be the fastest, followed by the carry lookahead and ripple carry adder. This is because the carry select adder has a lower number of logic gates to propagate through to get to the output.

Ripple Carry Adder

Path Summary

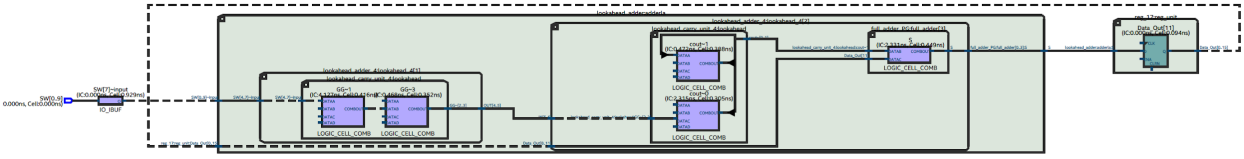
	Property	Value
1	From Node	SW[0]
2	To Node	reg_17:reg_unit|Data_Out[14]
3	Launch Clock	Clk
4	Latch Clock	Clk
5	Data Arrival Time	16.130
6	Data Required Time	23.382
7	Slack	7.252



Carry Lookahead Adder

Path Summary

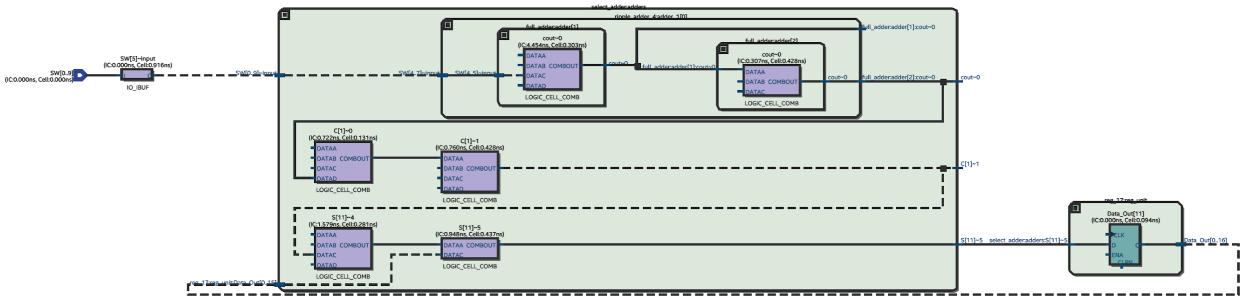
	Property	Value
1	From Node	SW[7]
2	To Node	reg_17:reg_unit|Data_Out[11]
3	Launch Clock	Clk
4	Latch Clock	Clk
5	Data Arrival Time	12.646
6	Data Required Time	23.392
7	Slack	10.746



Carry Select Adder

Path Summary

	Property	Value
1	From Node	SW[5]
2	To Node	reg_17:reg_unit|Data_Out[16]
3	Launch Clock	Clk
4	Latch Clock	Clk
5	Data Arrival Time	11.669
6	Data Required Time	22.803
7	Slack	11.134



Post-Lab:

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?)

In order for the CSA to have an advantage over the conventional ripple carry adder, it must have a hierarchical design of some sort. A flat CSA would be just as fast as a ripple carry adder with more than twice the amount of resources. This is because the CSA relies on speculative parallel computing of sections of bits. In a flat design, there is only one branch and is thus purely a serial computation. With a hierarchical design, the CSA can compute the sum of multiple bits in parallel. As a result, the total computation time only depends on the size of one block. As far as finding which hierarchical design is best, there is a tradeoff between resources and speed. As you divide the bits into more blocks there will be an increase in speed as well as in resources. To find the ideal hierarchy, you would need to know how many resources are available and the increase

in speed per division of bits. With this information, one could implement the ideal CSA for their project.

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.

Carry Ripple Adder:

LUT	78
DSP	0
Memory(BRAM)	0
Flip-Flop	20
Frequency	78.44Mhz
Static Power	89.97mW
Dynamic Power	1.48mW
Total Power	105.11mW

Carry Lookahead Adder:

LUT	83
DSP	0
Memory(BRAM)	0
Flip-Flop	19
Frequency	97.98Mhz
Static Power	89.97mW
Dynamic Power	1.53mW
Total Power	105.36mW

Carry Select Adder:

LUT	82
DSP	0
Memory(BRAM)	0
Flip-Flop	20
Frequency	98.37Mhz
Static Power	89.97mW
Dynamic Power	1.66mW
Total Power	105.31mW

Observe the data plot and provide explanation to the data, i.e., does each resource breakdown comparison from the plot make 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?

These plots prove that the theoretical design expectations of the three adders hold true when actually implemented. The number of resources for each adder makes sense with the theoretical design. It was expected that the ripple carry adder and the carry lookahead adder use the least and most resources, respectively. The maximum operating frequencies agree with the theoretical design expectations. The carry lookahead and carry select adder operate at much higher frequencies than the ripple carry adder because of its high latency. Furthermore, while the carry lookahead and carry select adder are close in speed, the carry select adder has a slightly higher operating frequency. Since these adders are so fast, their critical path is actually not the adder entity but instead IO pins. Due to this, the frequency does not depend on the adders themselves. However, when comparing the path length of the adder entities, it becomes clear that the carry select adder is slightly faster than the carry lookahead adder. In terms of power, it was expected that the carry lookahead adder consumed the most amount of power, because it requires more resources. The ripple carry adder was also expected to be much lower than the other adders because of the few number of resources required.

Conclusion:

_____ *Overview:*

We really enjoyed working through this lab and getting some more experience with SystemVerilog. Implementing the carry ripple adder, the carry lookahead adder, and the carry select adder and discussing the advantages and disadvantages of the three implementations demonstrated the versatility that comes with the design of logic processors. It also showed us

how to think through designing to minimize computation time through parallelization which was great.

Bugs:

One problem that we had was neglecting to connect the carry-in bits of the 4 carry lookahead adder's full adders to the central carry lookahead unit. Instead, we connected the carry-out bits of the full adders of the previous carry lookahead adder's carry-in bits. While this did not create any functional issues, this design defeated the purpose of the central carry lookahead unit. Since this design was essentially a serial chain of carry lookahead adders, it performed much slower than anticipated. After fixing the problem, there was a relatively large increase in performance. As countermeasures, we will modularize sub systems like this so a problem like this cannot occur.

Lab Manual Feedback:

Overall, I felt that the lab manual was very useful, clear, and well-organized. The description of the different implementations for the three adders built for this lab along with supporting diagrams really simplified the coding process. In terms of improvements for next semester, maybe adding a link to an excel sheet with the PIN assignments would be helpful, and also providing photos displaying the location of the design analysis data within the compilation report.