ECE385

Spring 2020

Experiment #4

Introduction to SystemVerilog, FPGA, CAD, and 16-bit Adders

Yuantao Lu, Yizhen Lu

Section ABD

TA: Nicholas Cebry, Wenjie Pan

Introduction

In this lab, we extended a 4-bit logical processor similar to the one in lab3 into 8-bit using SystemVerilog. The processor is used for 8-bit logic operations of AND, OR, XOR, 0, and all the inverse operations. We also made adders of three types that perform 16-bit arithmetic binary adding operation also using SystemVerilog. The three types of adders include a ripple adder, a carry lookahead adder, and a carry select adder. The three adders have the same operations but a relatively different design and behavior.

Part I - Serial Logic Processor

Block Diagram

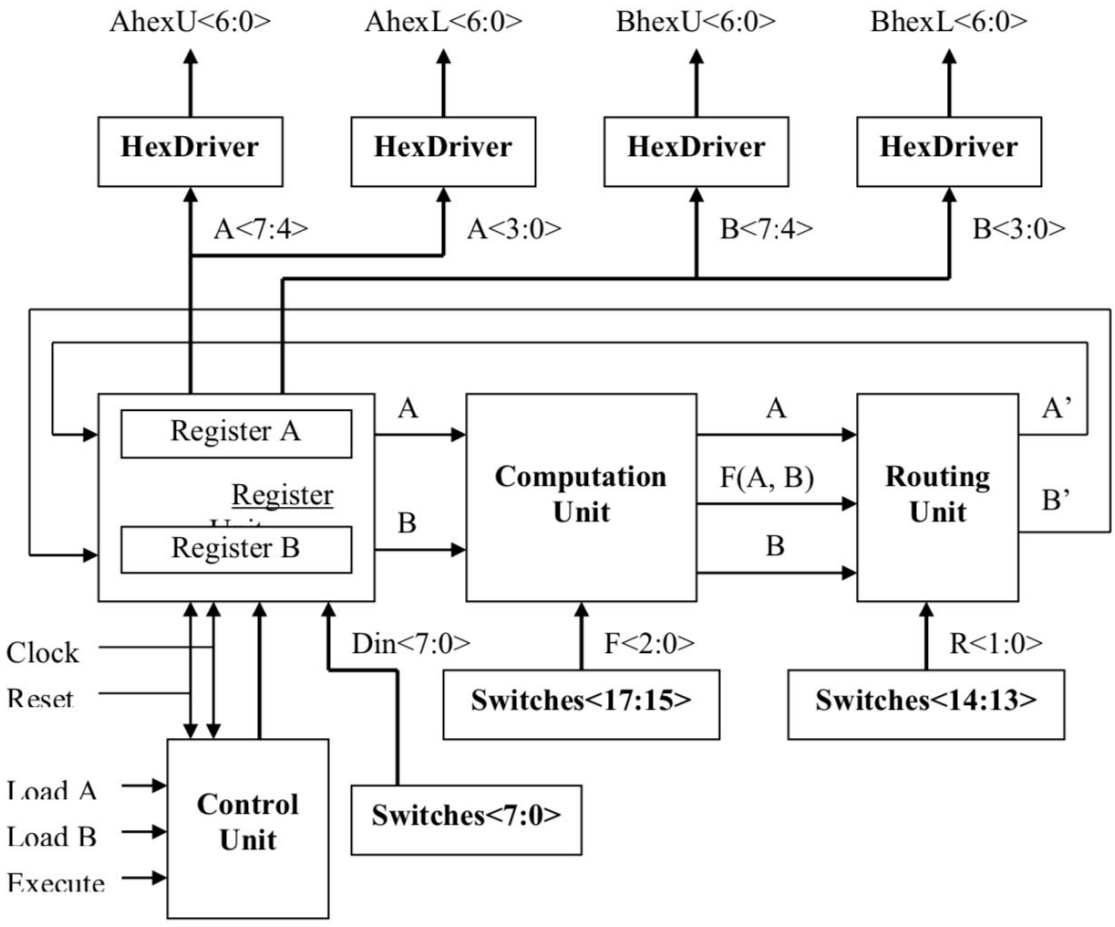


Figure 1. Top- Level Schematic of 8-bit Serial Processor by RTL Viewer

Modifications on Provided Code

To extend the 4-bit logic serial processor into 8-bit, there are several changes we need to make. First of all, since *Din* has become 8 bits, we need to change every corresponding 4-bit part corresponding to *Din* into 8 bits.

First of all, in the top level processor, we need to change signals representing *A* and *B* and *Output* into 8 bits. Also, when we extend to 8 bits, we will need more Hex Drivers to view the upper nipples of registers. In the diagram above, the *AhexU* and *BhexU* were assigned by the upper four bits of *A* and *B* respectively.

Secondly, the register Module needs to be changed. The original registers were both 4-bit to hold Parameter *A* and *B* and the *Output*. Now, we want All the signals mentioned above 8-bit. However, there wasn't much to change for this part. We just need to change the 4-bit register Module into an 8-bit register Module by extending all the variables into 8 bits. The shift logic would not change so the lowest bit will still be the bit that's shifted out.

Most importantly, we need to change the control part of the design because the logic has changed for control bits of the shift registers. Instead of making the shift register stop shifting after four operations, we now want this register to stop shifting after eight operations. However, we figured that the control state machine was composed of many states instead of using the Mealy State Machine we used in Lab3. Each state corresponds to the number of bits shifted and there is an ending state and reset state. By stopping the shift registers after eight operations, we

simply need to add four inner states and adjust the correct stopping state and the reset state.

There's no need to make changes to the Compute Module and Router Module because we still perform a one bit operation each time.

Simulation Result

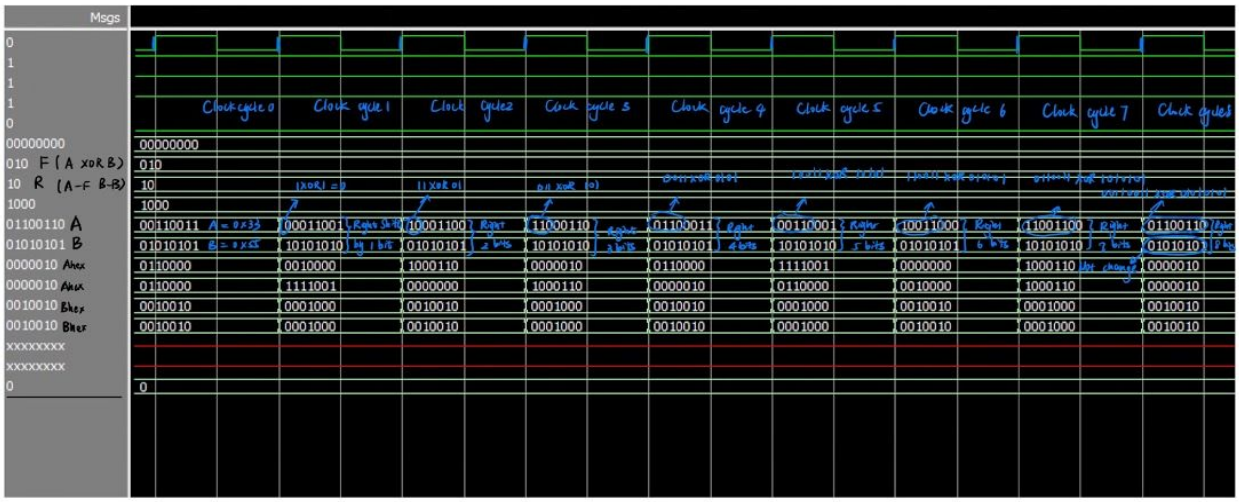


Figure 2. Simulation Result for A XOR B →A of 1 operation cycle(F = 010, R = 10)

Part II - Adders

Ripple Carry Adder

Written Description

This adder is the simplest design that is made of sixteen binary full adders. The initial carry-in bit was zero and each carry-in bit of a full adder is connected to the carry-out bit of the previous full adder. The carry-out bit of the last full adder will be the carry-out bit of the operation. The output result of each full adder makes up the final result where the *n*th bit of the result corresponds to the *n*th full adder. This adder works because of the definition of the arithmetic adding operation.

Block Diagram

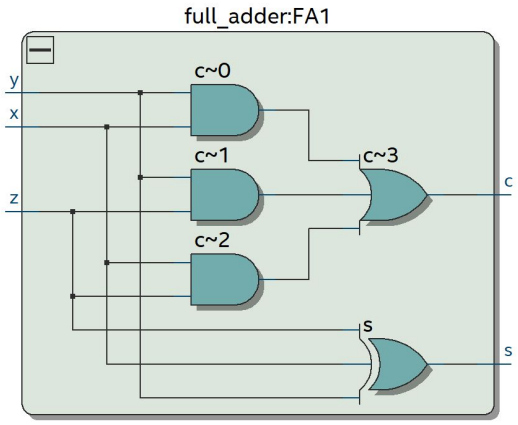


Figure 3. Gate level diagram for full adder

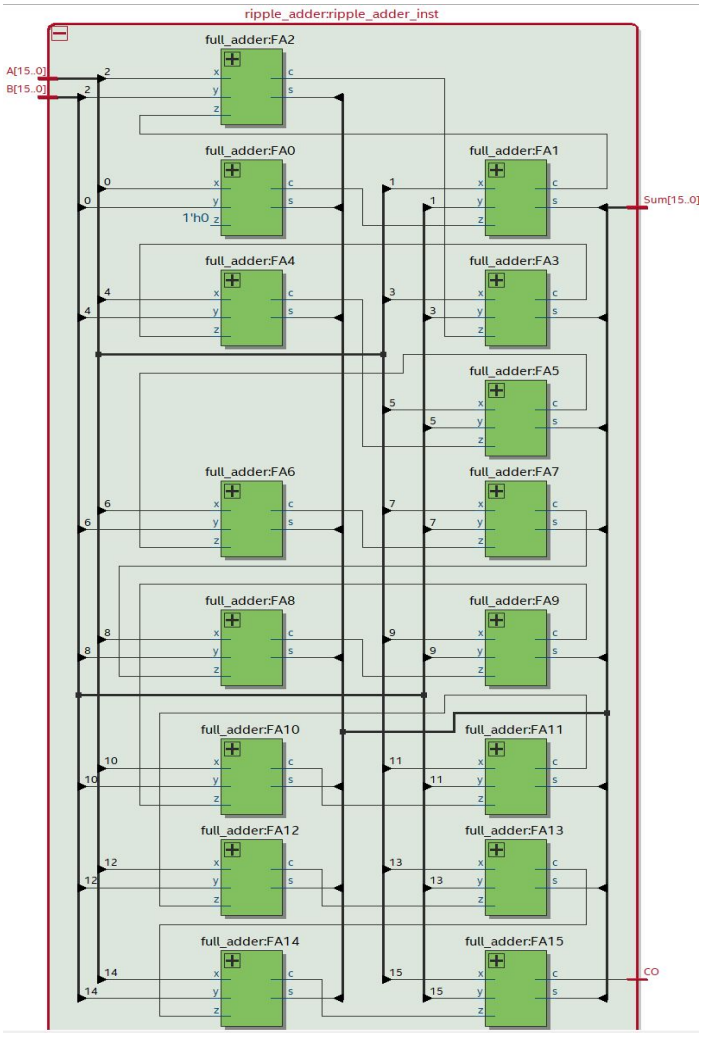


Figure 4. Block Diagram for 16-bit ripple adder

Carry Lookahead Adder

Written Description

This Carry Lookahead Adder used the *P* and *G* logic to find out the carry-out bit of a full adder. Unlike the ripple adder, before getting the carry-out bit from the previous full adder, the carry lookahead adder will predict the carry-out bit using the input *A* and *B* for each full adder and generate the logic *P* and *G* standing for *propagated* and *generated.* The carry-out bit of the *n*th full adder can be generated directly from the 0th carry-in bit and each of the previous *P* and *G* directly instead of waiting for the carry-out bit from the previous full adder. The previous *P* and *G* signals can also be generated from the corresponding input bits directly. Compared to the ripple adder, this adder will be much faster and have a higher operating frequency. The downside of this adder is that it will have much more logic gates for the carry-out, *P* and *G* logic, which increased both area and the consumption of power.

4x4-bit hierarchical CLA design

However, when the bit level gets high, the logic of computing the carry-out bit will be much complexed and the cost of making such logic will be high. So we followed the lab instruction and implemented the 4x4-bit hierarchical CLA design. The 16-bit input was divided into groups of 4-bit, each feeding into a 4-bit CLA. Each of the 4-bit CLA was considered as a single unit. These four units were then fed into a 16-bit carry look-ahead unit to construct a 16-bit CLA. This 16-bit carry look-ahead unit is similar to a 4-bit look ahead unit, but instead of taking the input(*P, G*) coming from the full adders, it will take the output from the 4-bit CLA and look ahead for the carry-out bit of each unit. This design is also hierarchical. We can combine the four 16-bit designs in the same way to design a 64-bit CLA.

P & G Logic

P and *G* stand for *propagated* and *generated*. When both inputs *A* and *B* are, a carry-out bit was generated. When either *A* or *B* is one, the carry-out bit would be propagated into the next level. So $G(A, B) = A \cdot B$ and $P(A, B) = A \oplus B$. When *P* and *G* are both defined, the carry-out bit $C_{i+1} = C_i + (P_i \cdot C_i)$. However, to avoid slow rippling like the ripple adder, we used C_{in} to compute the C_i directly.

$$C_i = C_{in} \cdot P_{i-1} \cdot P_{i-2} \ldots P_0 + G_0 \cdot P_{i-1} \cdot P_{i-2} \ldots P_1 + G_1 \cdot P_{i-1} \cdot P_{i-2} \ldots P_2 + \ldots$$

Block Diagram

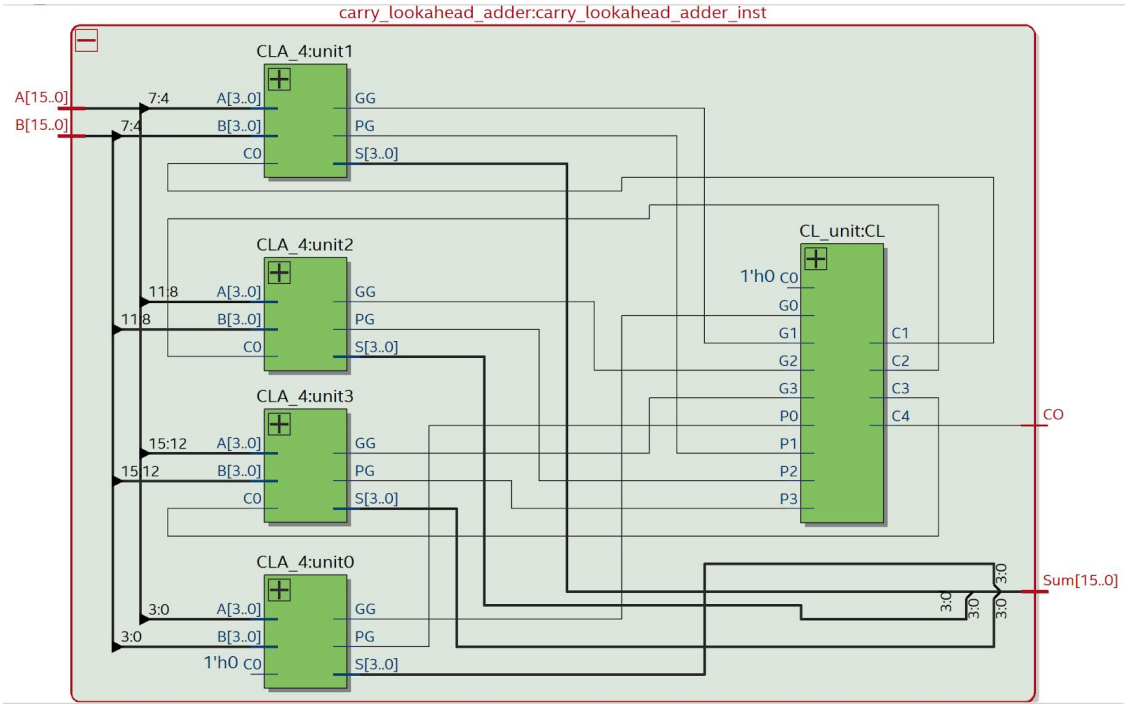


Figure 5. High Level Block Diagram for 16-bit CLA

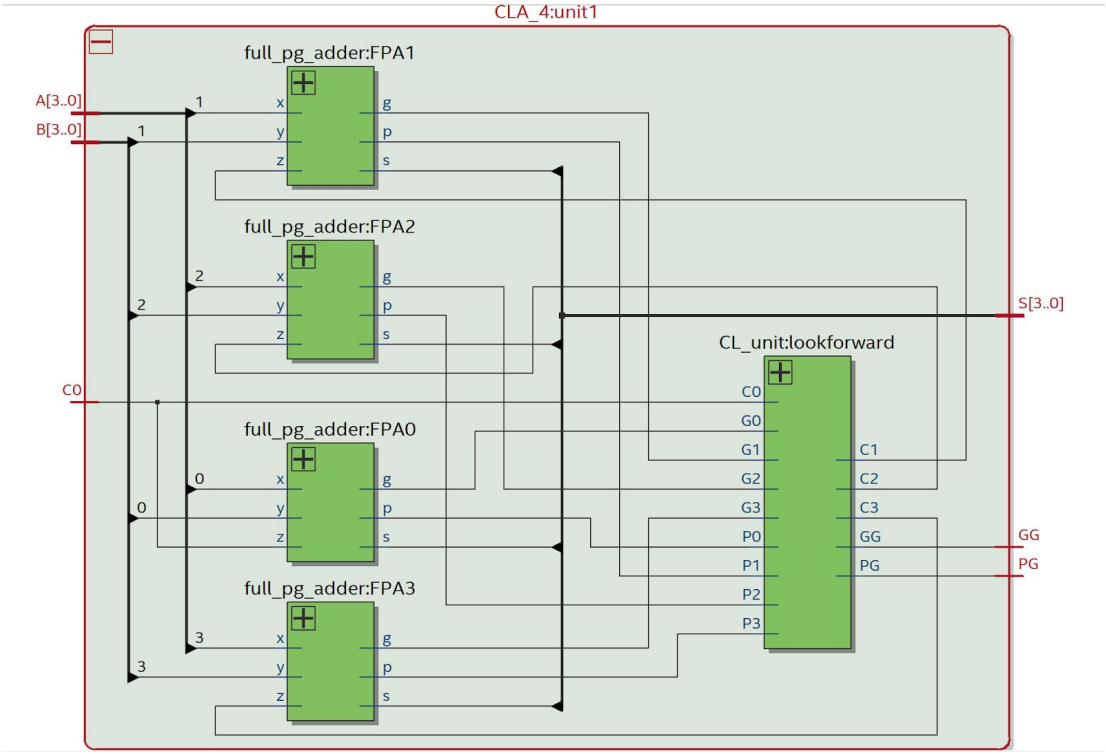


Figure 6. Block Diagram for 4-bit CLA unit

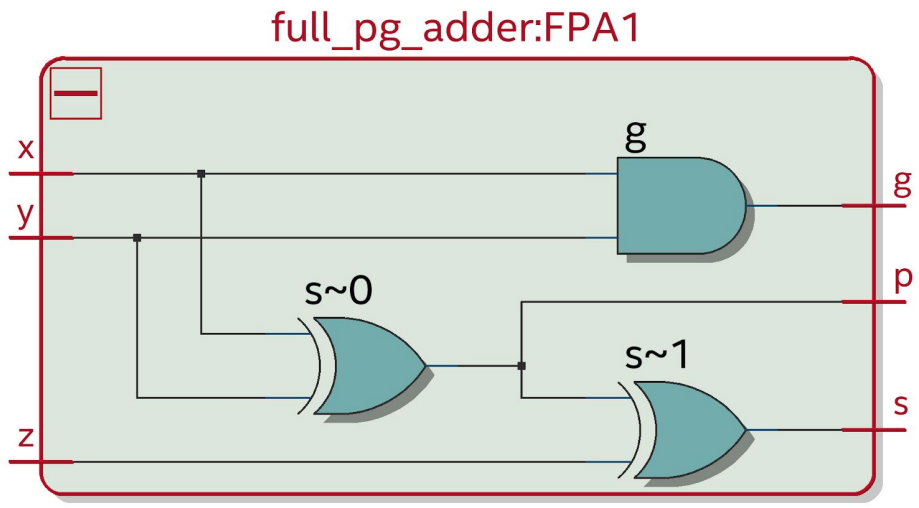


Figure 7. Gate-level diagram for pg-adder

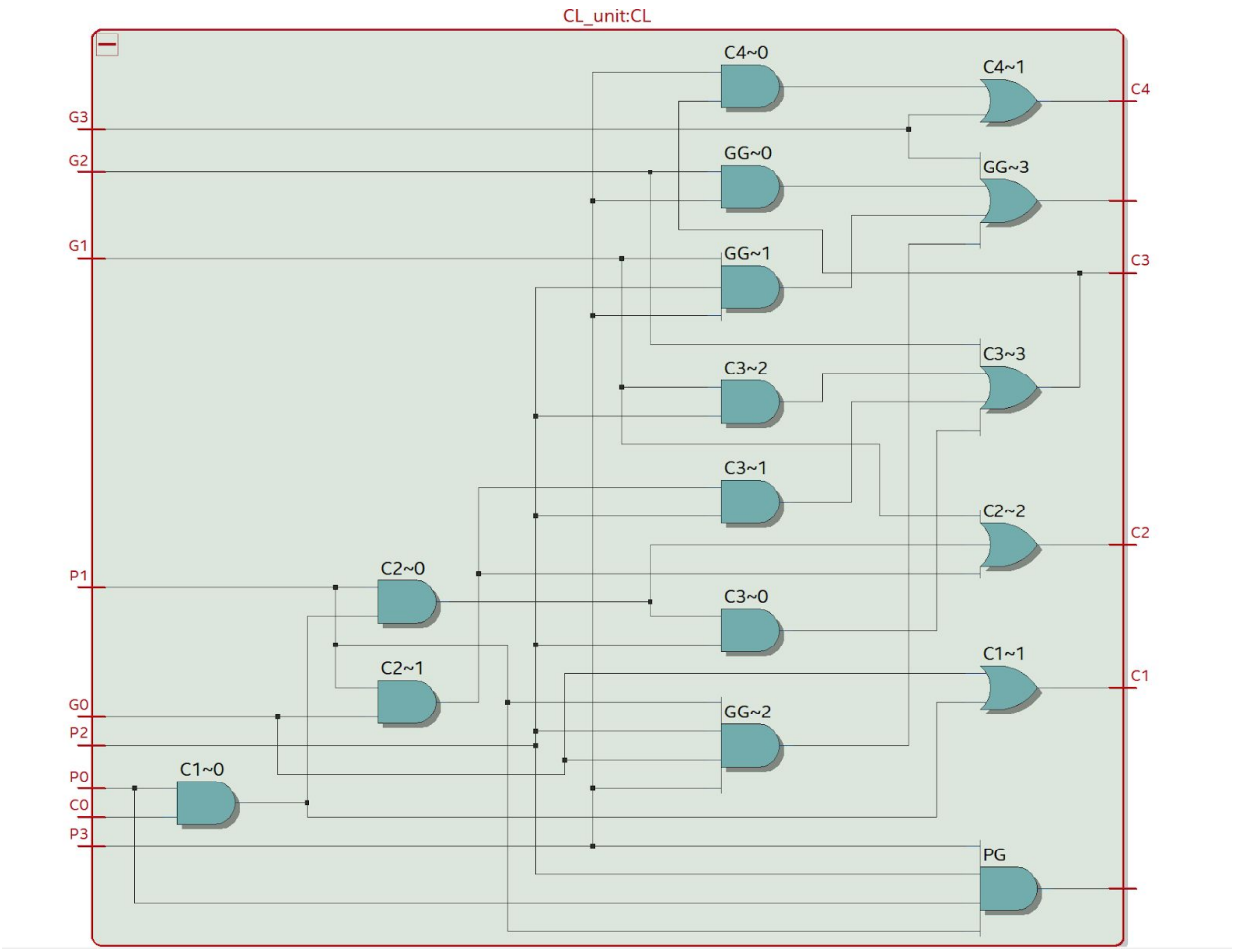


Figure 8. Block diagram for 4-bit carry look-ahead unit

Carry Select Adder

Written Description

In this lab, we also used a 4x4-bit hierarchical design for the Carry Select Adder. We implemented the 4-bit ripple adder and viewed it as one unit. Then a 4-bit CSA unit was implemented using two 4-bit ripple adders and one 2-to-1 MUX for selecting the actual four output bits using a carry-in bit. The carry-out bit was also selected depending on the carry-in bit but a 2-to-1 MUX was not used. Instead, we used a simpler logic based on mathematical facts. Since for the lowest four bits, we have only one carry-in bit, so only a four-bit ripple adder is enough to implement the 4-bit CSA.

Carry Select Adder was built upon the fact that there are actually only two choices *0* and *1* for the carry-in bit of a full adder. This select adder was built by using two full adders of the same size and a multiplexer. Both possible results were precomputed before the decision for the carry-in bit was made. The carry-in bit will then be used for selecting the real output of the precomputed two possible outputs. The carry-in bit will also be used for deciding which carry-out bit to choose from the two possible carry-out bits precomputed.

Carry Select Adder is fast because it trades the design complexity and almost twice the operations used for speed. Since our design is composed of four units. Those four units compute the possible outputs for the input already known instead of waiting for the previously computed result. When the unit representing the lowest four bits finished computing and passes its carry-out bit, all of the other units would finish computing as well. What is left to do is simply pass the carry-out bit through several glue parts(work like the MUX but simpler) and select the correct output and carry-bit for each unit. As a result, the total time needed for our Carry Select Adder is just the time for computing one unit(4 bits) plus the time needed for passing three glue units.

However, to select the carry-out bit, a 2-to-1 MUX is not needed because we could just use an AND and an OR gate. This selecting logic is the glue logic between two units of CSAs. See the glue logic diagrams for details. The K-Maps below analyzed in detail why a 2-1-MUX was not needed.

2-1-MUX	*Input A, B*

S(selecting)		00	01	11	10
	0	0	1	1	0
	1	0	0	1	1

Table 1. K-Map for 2-1-MUX

Logic S'B + SA

Glue logic	*Input A, B(two possible Carry-outs)*				
C(Carry-in)		00	01	11	10
	0	0	1/x	1	0
	1	0	1/x	1	1

Table 2. K-Map for glue logic of CSA

Logic BC + A

The K-Map is actually used as a truth table comparing the difference between two designs. Signal *A* is used as the carry-out bit for the full adder with carry-in bit *1* and signal *B* is used as the carry-out bit for the full adder with carry-in bit *0*. As we can see on the table, the only difference that the two designs had is when *A* is *0* and *B* is *1*(labeled red). However, in practice, this case would not appear according to mathematical reasoning. So the indexes for the glue logic design using AND and OR gates with *AB = 01* are actually *don't care.* As a result, this design using *Carry-out = BC + A* actually saved one logic gate in the design and thus improved the performance of the CSA.

Block Diagram

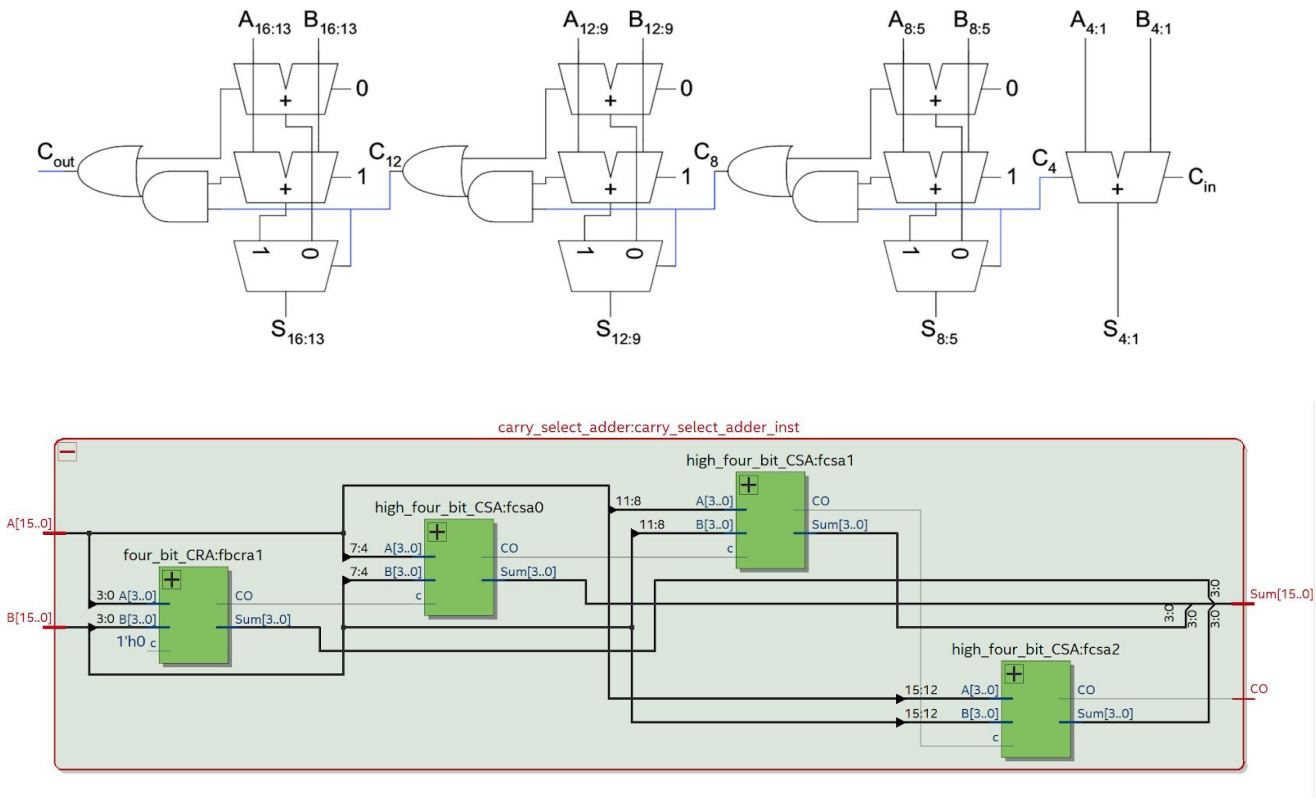


Figure 9. High Level Block Diagrams for 16-bit CSA

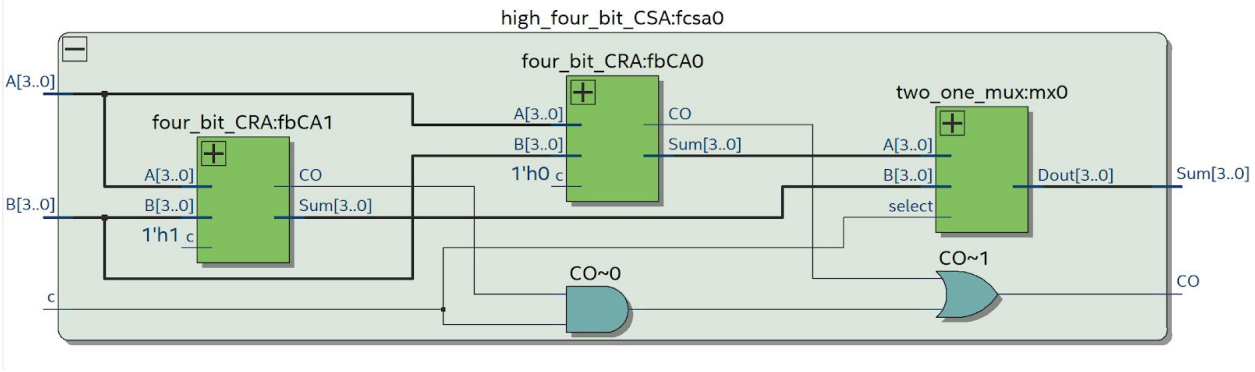


Figure 10. Block Diagram for 4-bit CSA unit with glue logic

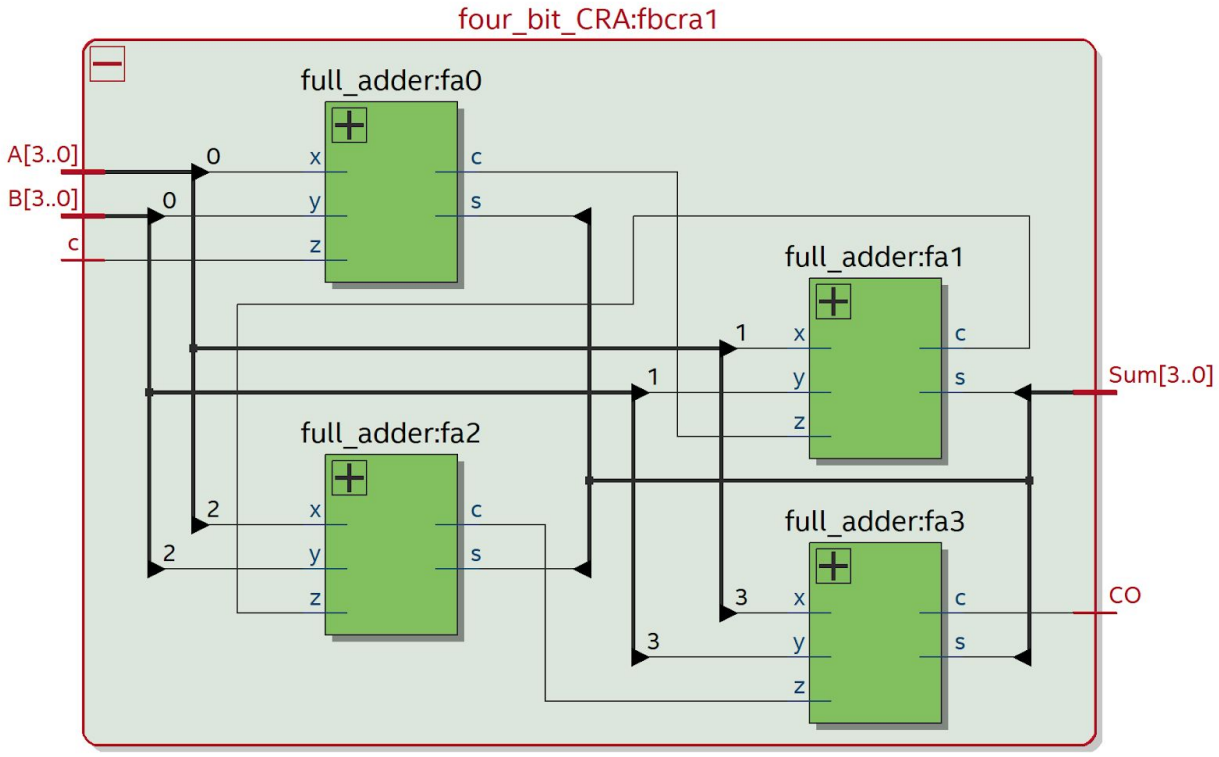


Figure 11. Block Diagram for 4-bit CRA unit(used for lowest 4-bit CSA)

Area, complexity, and performance tradeoffs

We analyze the complexity, the area and the performance of the Carry Ripple Adder, the Carry Lookahead Adder and the Carry Select Adder in this part.

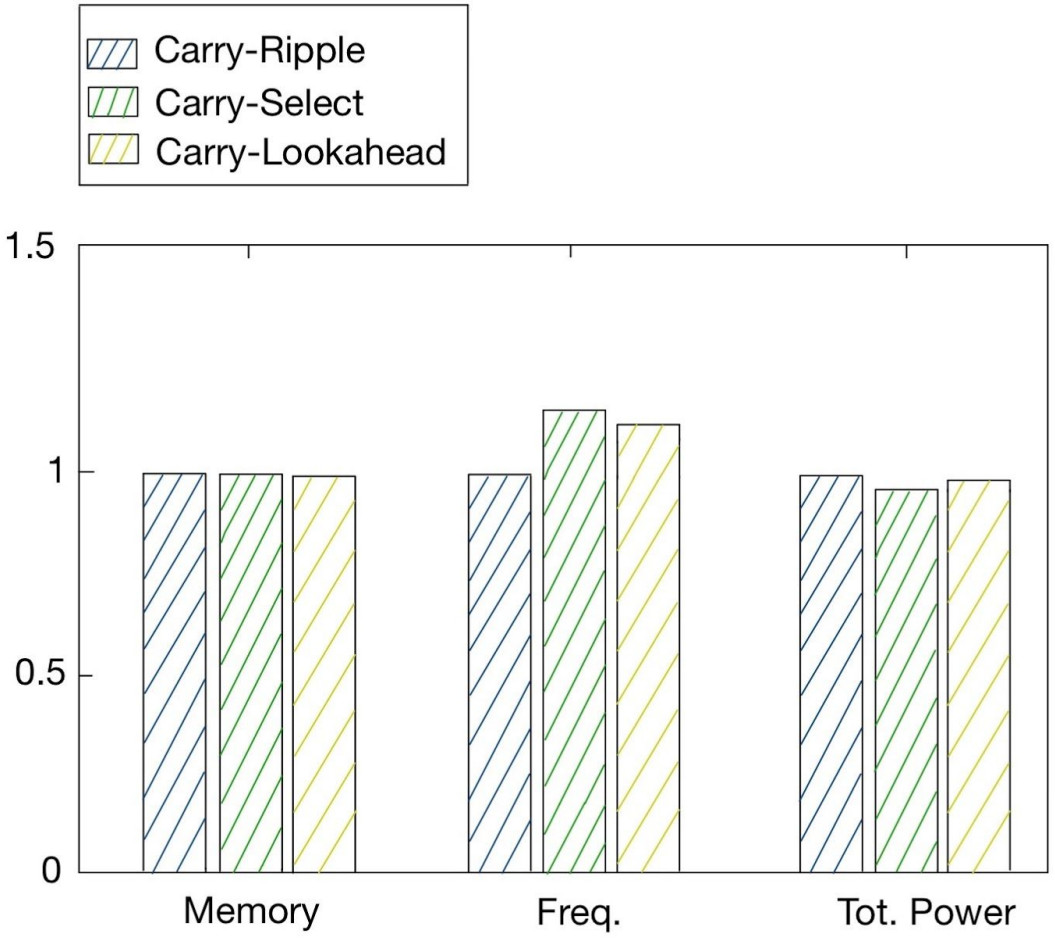
Clearly, Carry Ripple Adder is the simplest design because it is just a serial connection of 16 full adders. It also takes the least space resources for the simple design. The problem with the Carry Ripple Adder is it needs to wait for 16 operations of a full adder until it can get the final result. The time delay for the Carry Ripple Adder is big, so the maximum possible frequency was lowered for the system.

Carry Lookahead Adder is more complex than Carry Ripple Adder because it used additional logic gates for computing the carry-out bit for a single unit. The lookahead is actually using many logic gates as shown in the diagram for the four-bit design. That is a large consumption of power and space. However, the complexity trades speed. Our design is made of four units and each unit is made of four full adders. Those full adders can operate parallelly. Once the single parallel operation is finished, the carry-bit can be calculated from the logic directly. So the time taken for a single unit of CLA is one full adder operation plus the time for a lookahead unit. Since the 4 units were feeding into another lookahead unit, those 4 units' operations can be performed parallelly. As a result, the total time taken for the CLA is the sum of time for one 4-bit unit plus the time for one lookahead unit.

Carry Select Adder the most complex one among the three adders. It performs twice the arithmetic operations needed for all the possible outcomes for given inputs. When the previous unit finished computing, the only decision to make was just which result to choose. It takes a really short amount of time to make the choice and then pass the resulting carry-out bit into the next unit. I think this design is most space-consuming and power-consuming but performs the best for 16-bit among the three adders.

Performance Graph

	Carry-Ripple	Carry-Select	Carry-Lookahead
Memory(BRAM)	0 bit	0 bit	0 bit
Frequency	89.98 MHz	100.3 MHz	98.45 MHz
Total Power	149.42 mW	142.43 mW	147.06 mW



Post-lab Questions

1) Looking at the design in Lab3, if we extend the design into 8 bits, we would need to rebuild the entire logic, but we don't need to add states to the Mealy State Machine(more circles but two states are the same). So we would still need 1 flip-flop for the control unit. Since the inputs were extended into 8 bits, we would need four 4-bit shift registers, each composed of 4 flip-flops. The inside of a 4-bit counter has 4 flip-flops(we will only make use of three

of them). A total of 21 flip-flops will be used in the TTL design. However, the resource information shows that is the SystemVerilog design, we used 43 flip-flops. The reason behind that is we used too many states for the state machine in the control module(A-J). The LUT for the SystemVerilog design, in this case, is 72, but we would not need this much even if we extend the design to 8 bits. As a result, I guess the TTL design is better since it uses fewer resources but harder to design. The SystemVerilog design is simple to make(through hard coding) but used more resources.

2) The 4x4 hierarchy design of the CSA we used for this lab is possibly not ideal. We chose one unit of the CSA to be 4 bits, which is possibly large compared to the total of 16 bits. There are other choices we could make since we could put different numbers of full adders in each unit or increase the number of units. I would need to know the time delays for the gates and MUXs to compute total operation time for the CSA. Since we do not know the time delays for the full-adder and the MUXs, who do not know whether a unit or the selecting logic and the MUXs operate faster. In the ideal case, the adders finish operation when the MUXs finish the operations. We need to do the experiments to test those delays to balance the operation time of MUXs and adders.

3)

	Carry-Ripple	Carry-Select	Carry-Lookahead
LUT	114	123	127
DSP	None	None	None
Memory(BRAM)	0 bit	0 bit	0 bit

Flip-Flop	105	105	105
Frequency	89.98 MHz	100.3 MHz	98.45 MHz
Static Power	83.83 mW	83.81 mW	83.82 mW
Dynamic Power	2.26 mW	2.81 mW	2.38 mW
I/O Thermal Power	63.33 mW	55.81 mW	60.86 mW
Total Power	149.42 mW	142.43 mW	147.06 mW

Bugs Encountered and Countermeasures Taken

At first, since we are not very familiar with SystemVerilog, one of the members wrote codes that have nested modules, which is not allowed. Also, there are also some problems caused by sharing the same name, which caused compiler errors. We fixed the problems by moving modules outside and making them independent, while changing variable names to avoid repetition. As long as the compilation finished and we loaded the program into the FPGA, no more bugs were found.

Conclusion

In the process of the lab, we found the lab manual and tutorials on the website regarding Quartus and SystemVerilog extremely helpful. They provide detailed instructions and procedures on how to conduct this lab successfully. Among all, the pin assignment table is most useful because it saves us a lot of time to figure out which signal should be connected to which pin in order for the desired design. In this lab, we had our very first interactions with FPGA and SystemVerilog. We found it really intriguing and a nice experience. The serial processor we need to modify has a very similar structure as the logic processor we built in Lab 3. This

provides a direct vision of what the code in SystemVerilog means in the actual circuit wiring. We also learned more ideas and thoughts on how to build more effective adders instead of simply the straightforward ripple carry adder. All these are really inspiring.