ECE 385

Spring 2020

Experiment 4

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

Michael Faitz and Zohair Ahmed

Section ABJ: Friday 2:00-4:50

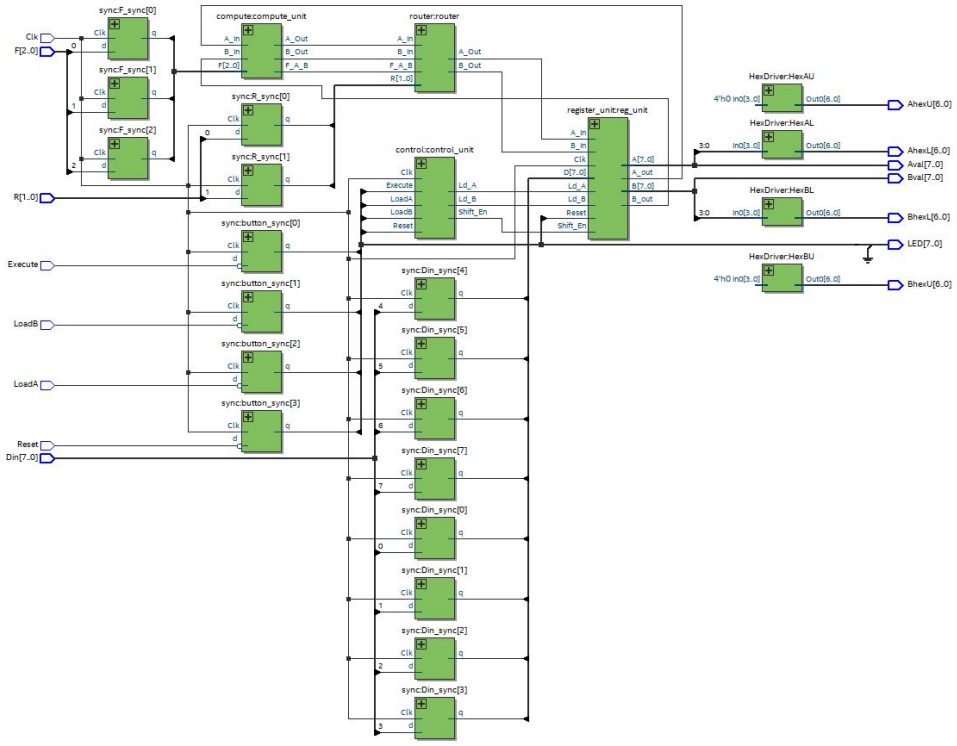
Yuming Wu and Lian Yu

Introduction

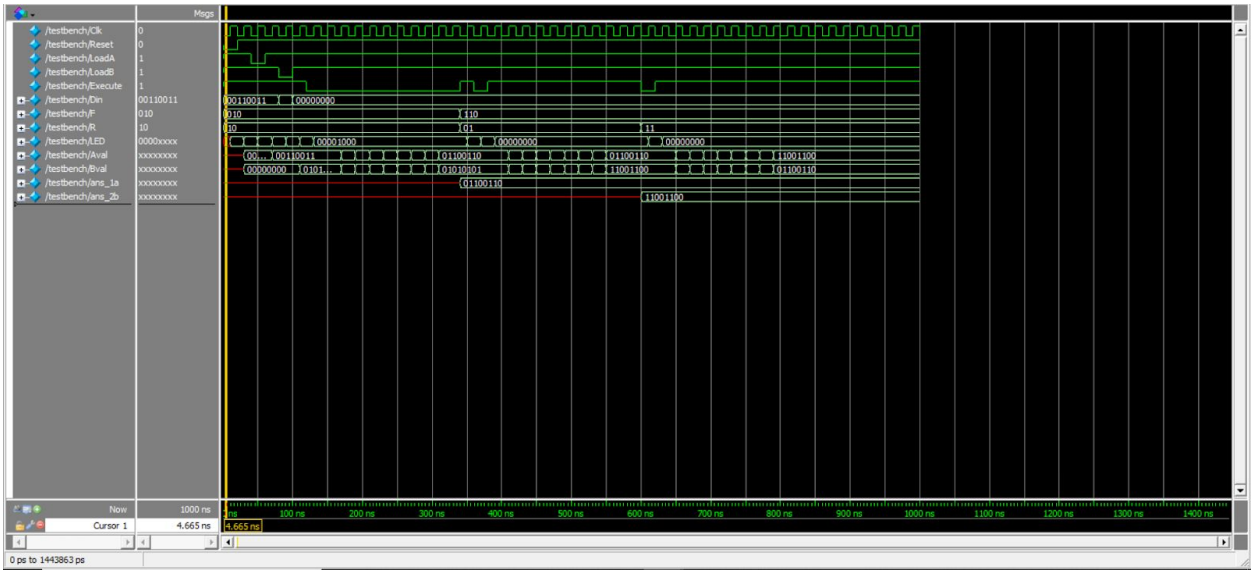
Our circuit is meant to perform a variety of logical operations (based on 3 input switches to choose the operation) on two sets of data (which can be loaded with values based on 8 input switches from the user), which will then be stored back into the registers (based on 2 input switches to choose the destinations of the register data and the function output). It can operate on 2 8-word shift registers, two bits at a time. The three adders allow the user to input 16 bits into two registers, along with a carry in bit. For the purposes of this lab, there was no carry in bit for the first adder (or set of adders). When the run switch is flipped, the adders will add the two input registers and calculate the sum and carry out bit. Each adder handles this differently: the carry ripple does it one bit at a time, the carry look ahead calculates the carry in bits for all adders as soon as possible, allowing for faster adding, and the carry select creates all possible outputs and uses the carry in bit to select the correct output when it is available.

Serial Logic Processor

1. High Level Block Diagram



2. We changed the number of data in switches from 4 to 8, as well as increased the register sizes from 4 to 8 bits. To make sure that all bits in the register would be included in computation, we added 4 more shift states (there was a reset state, a stop state, and 4 shift states prior).
3. Simulation

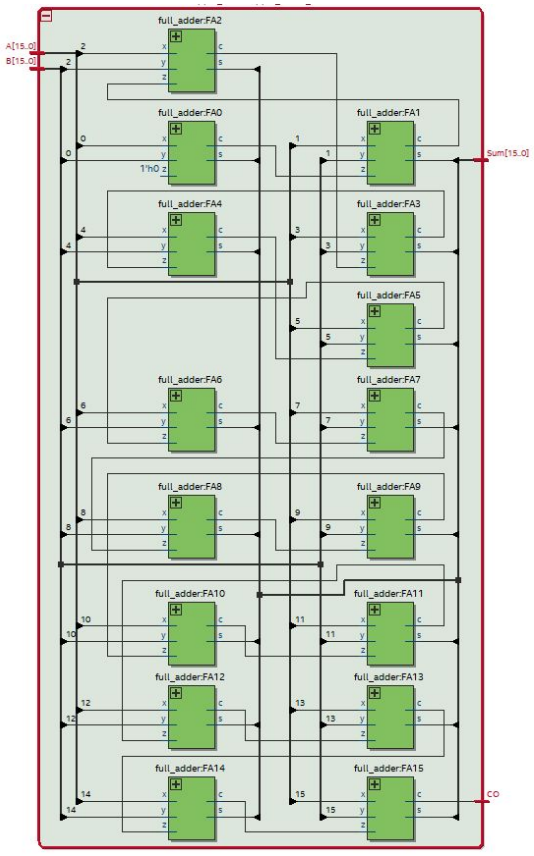


Operations being shown are loading 00110011 into A and loading 01010101 into B and then doing A XOR B and storing it into A. This result is shown on time 330ns on channel Aval and the expected answer is shown in ans_1a at time 380ns. These results are equivalent which tells us this first operation was successful. The second operation performed was A XNOR B which was stored in B. The A used here was the updated A from the operation prior. The result can be seen at time 550ns on channel Bval and the expected answer is shown at ans_2b time 600ns.

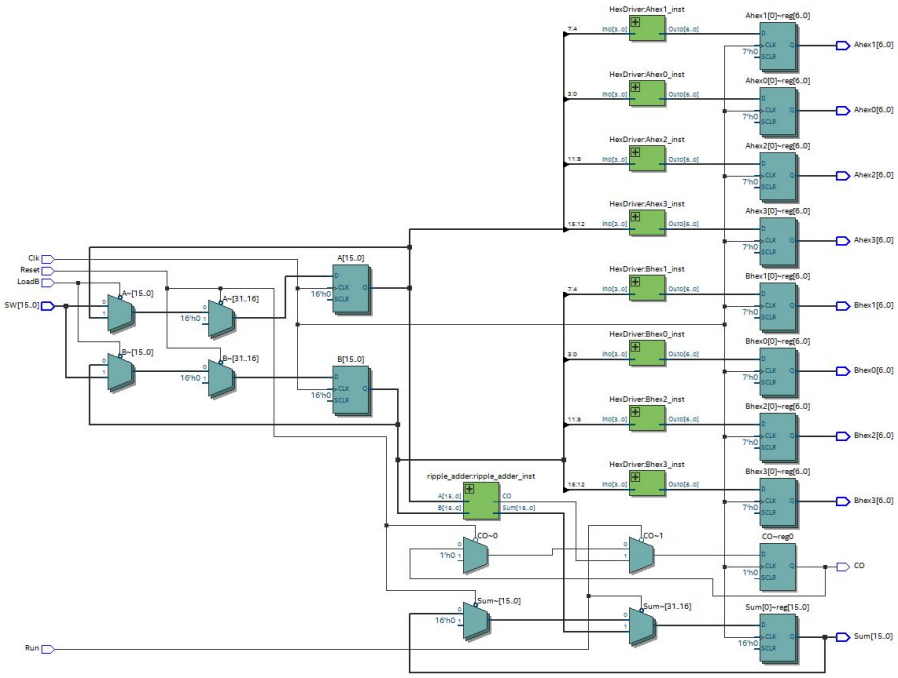
Adders

1. Ripple Carry Adder
    a. Every adder has the inputs of 1 bit from registers A and B, a carry in bit, and outputs a carry out bit and a sum bit. The first adder receives a carry in bit of 0, and the last adder's carry out bit is not an input to any other adder. In other cases, the carry in bit of any adder is the carry out bit of the previous.
    b. Block diagram

Inside Ripple Carry Adder



Ripple Carry Adder Toplevel



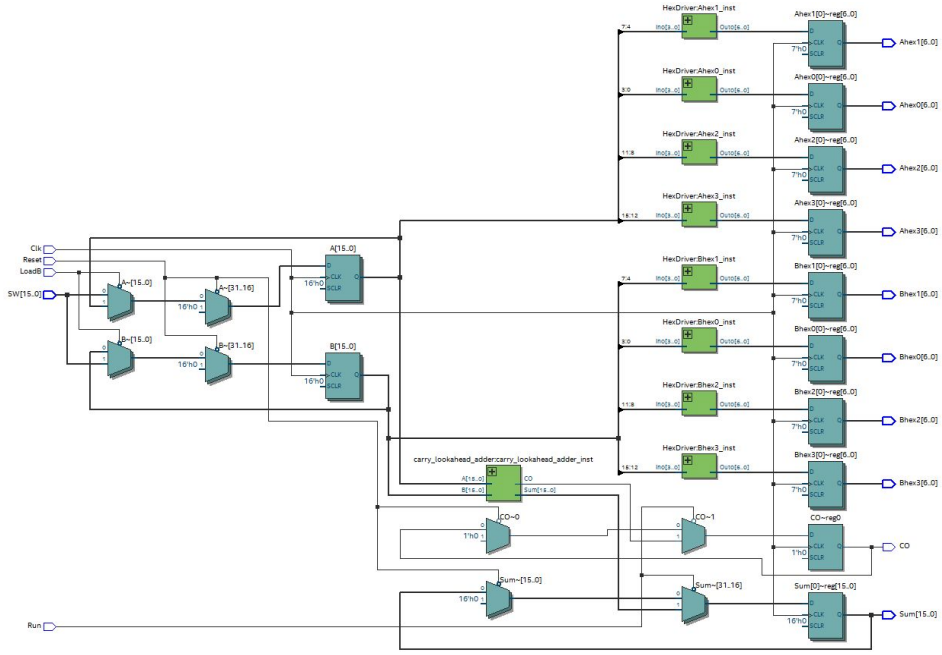
2. Carry Lookahead Adder
    a. The carry lookahead adder will calculate the carry in bits for every individual adder as soon as possible. It does this by figuring out if two of A, B and Cin of the

last adder are equal to 1 using P and G logic. This allows for faster computation as each adder is allowed to calculate sooner than the carry ripple, with the tradeoff of additional logic gates and wires.

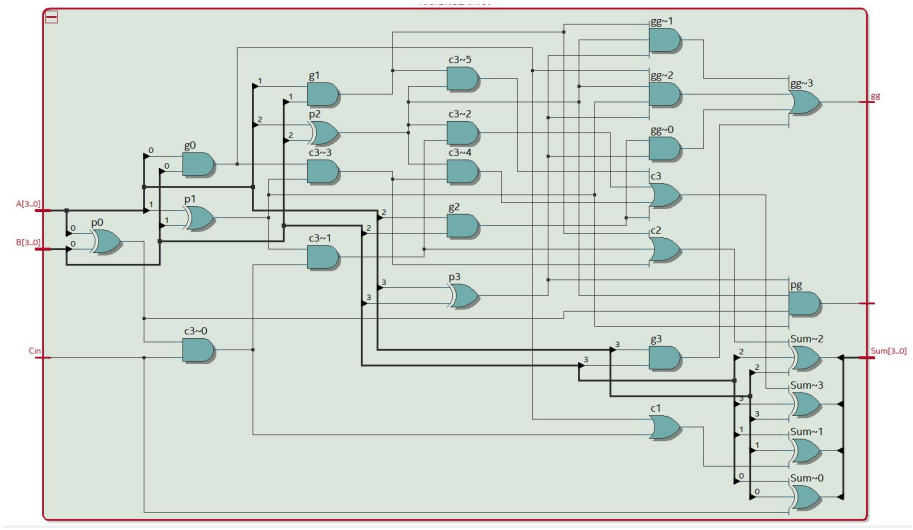
b. G logic is to determine a carry out based on the inputs from A and B. Basically, for any adder n, $G_n = A_n \& B_n$. This logical operation occurs faster than the adder can generate its own carry out bit. P logic is used when only one of A and B is 1. For any adder n, $P_n = A_n \hat{} B_n$. Both of these operations, for all adders n, can be done in parallel. Both G_n and P_n are combined with the carry in bit (C_n) to the adder to calculate the carry out bit (C_{n+1}), where $C_{n+1} = G_n + P_n C_n$. These are used to allow the adders to operate more in parallel than in series which speeds up the process compared to the ripple adder.

c. Despite the independence of calculating carry out bits using P and G logic, the operation will still effectively ripple if we connect every CLA together in series. By separating them into 4 groups of 4 CLAs, we can calculate the Cin of each group using the P and G logic, allowing for faster calculation. This effectively allows for 4 queues of 4 instead of the single queue of 16 operations. This works faster because instead of waiting for the output to come from full adders, we use the carry lookahead unit to calculate it.

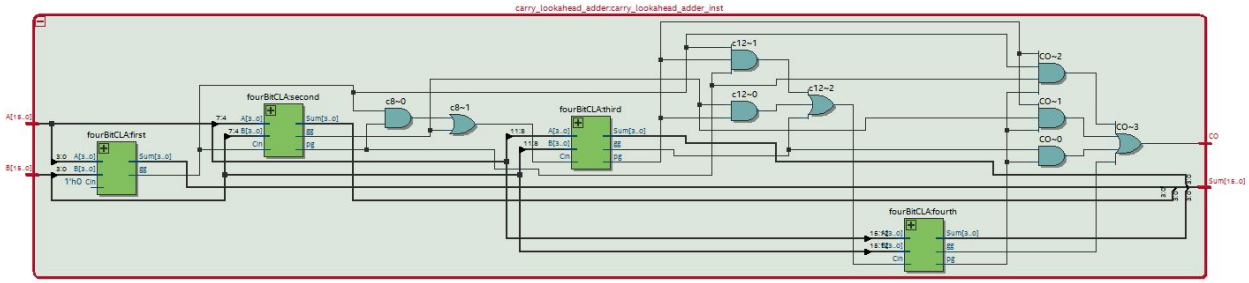
d. Full Block diagram



e. Block diagram inside a single CLA (4 bits)

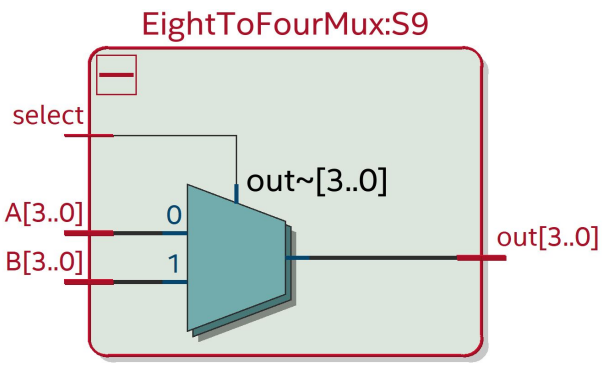


f. Block diagram of how each CLA was chained

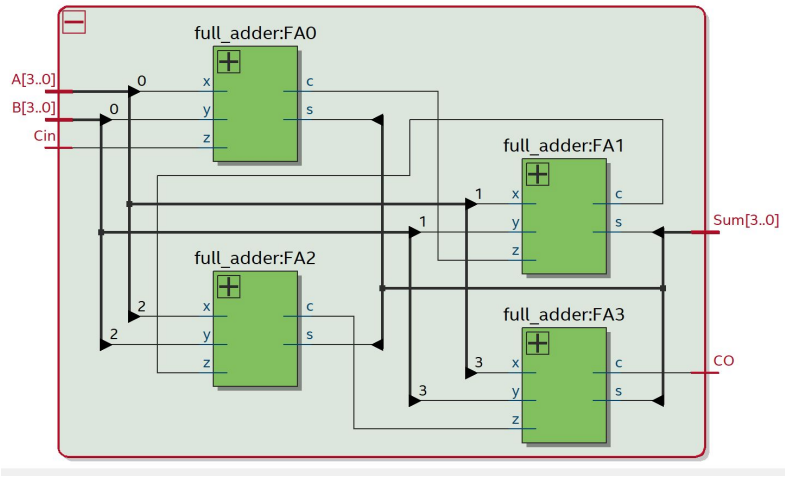


3. Carry Select Adder
    a. The carry select adder will create a lookup table of sorts, calculating every possible carry-out and sum output and waiting for the previous adder's carry out bit to choose which carry out and sum will be used. This allows for faster operation than the ripple and lookahead adders, with the downside of requiring almost twice as many adders and some additional gates.
    b. The CSA processes every group of 4 bits at the same time, calculating two sums, one with Cin of 0 and one with 1. This way, each pair of sums and Couts for all groups of 4 input bits are calculated at once, and using MUXes and minimal combinational logic, the actual sum and Cout is picked. This way, the most noticeable propagation delay is through the combinational logic, not the adders.
    c. Block diagram including adders, multiplexors, and glue logic

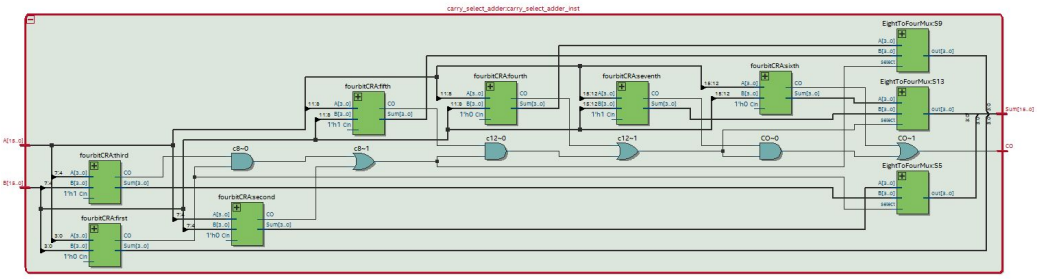
Eight to Four MUX



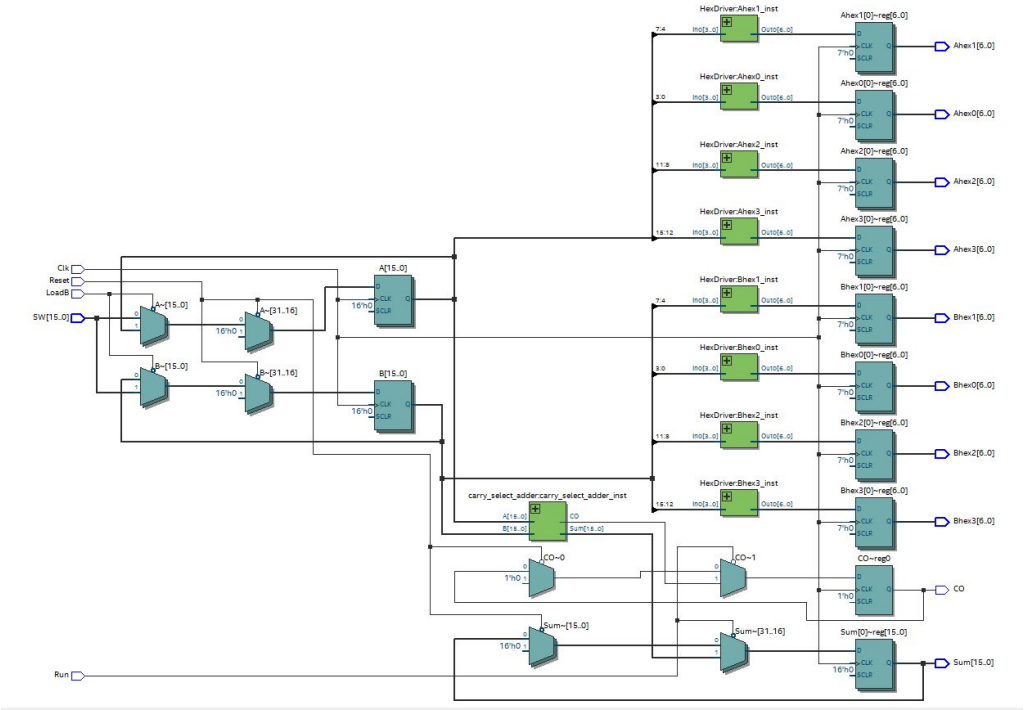
4 bit CRA



Inside Select



Top Level Block Diagram



4. The area and complexity of the carry ripple adder is lower than the other two. It is the slowest however, as you must wait for the last adder to finish operation before the next one can begin. The carry lookahead adder has the second highest area and highest complexity, using many logic gates to calculate carry bits which allows every adder to operate at almost the same time. It is the second fastest adder. The carry select adder has the highest area and second highest complexity, as it uses almost double the amount of adders to calculate a number of bits. This is the fastest adder, however it has by far the most area.

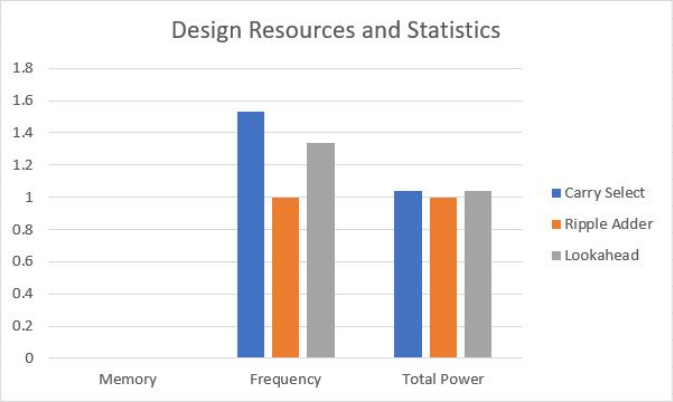
Data:

Carry Select Adder	
LUT	123
DSP	0
Memory(BRAM)	0
Flip-Flop	105
Frequency	95.18MHz
Static Power	98.57mW
Dynamic Power	6.34mW
Total Power	161.8mW

Ripple Adder	
LUT	114
DSP	0
Memory(BRAM)	0
Flip-Flop	105
Frequency	62.25MHz
Static Power	98.55mW
Dynamic Power	3.09mW
Total Power	156.09mW

Lookahead Adder	
LUT	131
DSP	0
Memory(BRAM)	0
Flip-Flop	105
Frequency	83.28MHz
Static Power	98.57mW
Dynamic Power	6.65mW
Total Power	162.31mW

Plot:



Written Description of all.sv Modules

Module: full_adder.sv
Inputs: x, y, z
Outputs: s, c
Description: Adds three bits together and outputs the sum and the carryout bit. Within this application it will add a carryin bit, and bits from registers A and B.
Purpose: Serves as a basic building block for the adder designs.

Module: ripple_adder.sv
Inputs: [15:0]A and [15:0]B
Outputs: [15:0] Sum and CO
Description: String of successive full adders using the prior carryout bit and an initial carryin bit of 0.
Purpose: To create the simplest design of a 16 bit adder for comparison with the other designs. Serves as a baseline and the worst model.

Module: lab4_adders_toplevel.sv
Inputs: Clk, Reset, LoadB, Run, [15:0] SW
Outputs: CO, [15:0] Sum, [6:0] Ahex0, [6:0] Ahex1, [6:0] Ahex2, [6:0] Ahex3, [6:0] Bhex0, [6:0] Bhex1, [6:0] Bhex2, [6:0] Bhex3
Description: Allows for positive edge clock resets of registers A and B when Reset is low and allows for the loading of data in SW switches to be loaded into A and B respectively following the clock cycles. When Run is low the data stored within Sum is output to the LED displays on the board. Also loads the new hex data into the hex driver when the clock cycle is on its rising edge. Finally, this module instantiates an instance of either the ripple_adder, carry_select_adder, or the carry_lookahead module.
Purpose: Serves as the backbone, driver, and connection to the clock for the other modules and allows them to be put all together and run on one clock.

Module: HexDriver
Inputs: [3:0] In0

Outputs: [6:0] Out0
Description: Takes in four bits of data and extends it into specific binary sequences that serve to turn on the 7 segment displays on the board.
Purpose: To drive the process that displays the content of the registers onto the 7 segment displays for ease of access for the user.

Module: carry_select_adder.sv
Inputs: [15:0] A, [15:0] B
Outputs: [15:0] Sum, CO
Description: A 4x4 hierarchical adder that uses pairs of 4-bit CRAs to calculate 4 groups of Couts 2 4-bit sums. Uses 8-to-4 MUXes and combinational logic to determine Couts and actual 4-bit sum groups.
Purpose: This module handled the 4x4 hierarchical CSA we were required to design for the lab.

Module: carry_lookahead_adder.sv
Inputs: [15:0] A, [15:0] B
Outputs: [15:0] Sum, CO
Description: 4 4-bit CLAs chained in parallel to create a hierarchical structure. Calculates the Cin bits faster than the carry-ripple adder, allowing for faster adding.
Purpose: This module handled the 4x4 hierarchical CLA we were required to design for the lab.

Module: 4bitCRA.sv
Inputs: [3:0] A, [3:0] B, Cin
Outputs: [3:0] Sum, CO
Description: 4 1-bit CRAs chained in series.
Purpose: Used to simplify the design of the 4x4 CSA. Could assist with the 16-bit CRA but we did not use that there.

Module: EightToFour.sv
Inputs: [3:0] A, [3:0] B, Select
Outputs: [3:0] out
Description: This is effectively a 2-to-1 MUX that we altered to pick between two 4-bit inputs.
Purpose: We used this two pick between the sum outputs from a pair of 4-bit adders in the CSA, using the Cout from the last 4-bit adder as the select bit for the MUX.

Module: fourBitCLA.sv
Inputs: [3:0] A, [3:0] B, Cin
Outputs: [3:0] Sum, pg, gg
Description: 4 1-bit CLAs chained in series, outputting a group propagate and generate which allows the next 4-bit CLA to calculate its Cin bits.
Purpose: This module is used to simplify the design of the 4x4 CLA that we designed. It gives the Sum, pg, and gg for a group of 4 1-bit CLAs which are used when calculating the Cin for other 4-bit CLAs.

Answers to post-lab questions

1) Compare the usage of LUT, Memory, and Flip-Flop of your bit-serial logic processor exercise in the IQT with your TTL design in Lab 3. Make an educated guess of the usage of these resources for TTL assuming the processor is extended to 8-bit. Which design is better, and why?

The TTL had no LUT, and used little memory due to the registers and a singular flip-flop, whereas the IQT used many LUTs, used no memory (BRAM), and a drastically higher (more than 100 more) number of flip-flops. To extend the TTL, there would be added complexity inside the control unit and additional switches to the registers, but would have no slowdown between single bit operations. There would be insignificant slowdown in the IQT by increasing the number of states, and would otherwise operate exactly the same. The design of the IQT is easier to extend to 8-bit or even further, with no slowdown, whereas the TTL requires a lot of rewiring and adding to different states, and would have slightly more slowdown as another counter bit would be necessary to track in the control unit. The IQT is better because of this.

2) 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?)

A 1x16 CSA, or a 16 bit CSA consisting of 16 1-bit CSAs chained together should be the fastest version of this adder, but contain the most area and complexity. The 4x4 has the closest tradeoffs between size and speed, while the 1x16 has greater speed but huge complexity. I think we would need information about the propagation delay tradeoffs between adders and gates, and we would need to test both adders in practice, testing first the speed of the 4-bit CRA against 4 bits of the 1x16.

3) Design Resources and Statistics

Carry Select Adder	
LUT	123
DSP	0
Memory(BRAM)	0
Flip-Flop	105
Frequency	95.18MHz
Static Power	98.57mW
Dynamic Power	6.34mW
Total Power	161.8mW

Ripple Adder	
LUT	114
DSP	0
Memory(BRAM)	0
Flip-Flop	105
Frequency	62.25MHz
Static Power	98.55mW
Dynamic Power	3.09mW
Total Power	156.09mW

Lookahead Adder	
LUT	131
DSP	0
Memory(BRAM)	0
Flip-Flop	105
Frequency	83.28MHz
Static Power	98.57mW
Dynamic Power	6.65mW
Total Power	162.31mW

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?

From the data above we can see that the CRA has the least components and Lookahead has the most components which makes sense as the CRA is the simplest design and the Lookahead has a significant amount of logic to calculate the carry out bits early which adds to complexity. Both DSP and Memory are zero for all the designs as there is no signal processing or external storage outside of the registers within each circuit. Each of the designs has the same number of flip-flops which also agrees with the theory as the flip-flops are in the shared design of each circuit and no design calls for additional flip-flops. We can also see that the static power drawn from each design is approximately the same which makes sense as the board will require a set amount of power and the dynamic power is what changes the total power within the design. From the table we can see that the total power and the dynamic power are the lowest within the ripple adder, once again from its simplistic design, and the most power is from the Lookahead which is a more complex system. Since both the Lookahead and the Carry Select are higher in complexity it makes sense that more dynamic power would be consumed within the circuit, however, we expected the Carry Select to be far above the other designs due to it performing almost double the operations as the CRA. We can see this within the doubling of the dynamic power between the two. The Lookahead adder consumes significantly more power than expected most likely due to the high number of logic elements required for each CLA to be linked together. Finally, the frequency operation of the CSA is the highest due to it effectively being individual units which can operate extremely more effectively than the ripples created by the other two. The CRA has the slowest maximum frequency due to the heavy ripple in the circuit from the simplistic design.

Conclusion

1. We didn't run into many bugs or countermeasures in this lab. We had difficulty with SystemVerilog syntax but those were the only issues we had.
2. It would have helped to have more guidance with how the 8 bit serial processor worked in the code and how it differed from the lab 3 implementation. Everything else was pretty rough when it came to getting used to Quartus and ModelSim, but we don't see how to improve on it.
3. No additional summary to be included here.