ECE 385

Spring 2019

Experiment #4

Introduction to System Verilog, FPGA, CAD, and 16-bit Adders

Rob Audino and Soham Karanjikar

Section ABJ, Friday 2:00-4:50

TA: David Zhang

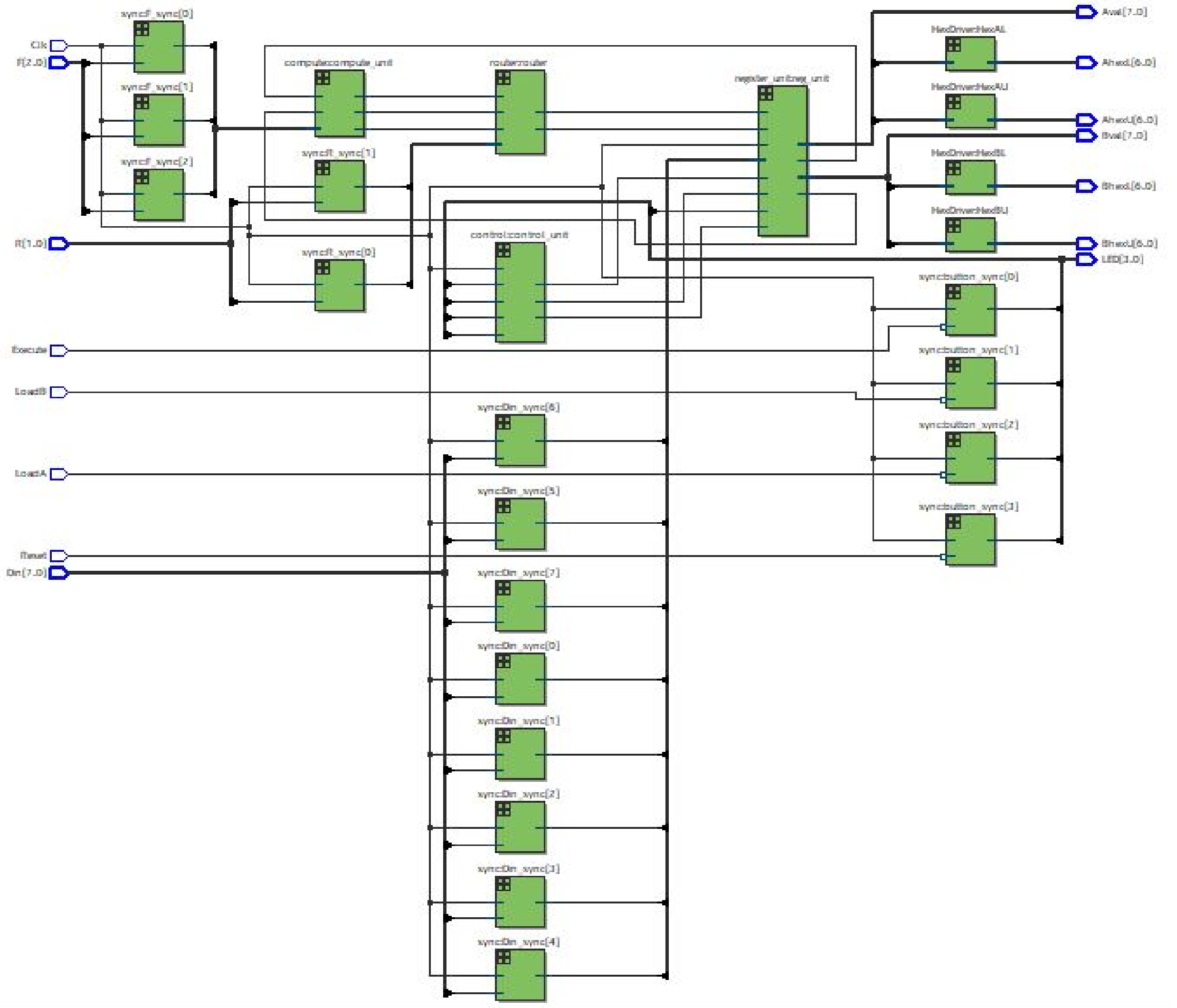
Introduction:

Lab 4 served as an introduction to System Verilog, and as a result, wasn't as in depth as the TTL labs. Our first task was to recreate an 8 bit version of the 4 bit logic processor we had created in Lab 3, with the same loading, computing, and routing features. The processor had to have loading capabilities to either register from switches, the same 8 logical operations (AND, OR, XOR, NAND, NOR, XNOR, All 1's, All 0's), and have the option to route the result of the operation to the A register, B register, or ignore the result completely and either return the contents of the original A and B registers to their respective places or switch places (e.g. Old A into B, Old B into A). We were given code to begin that created the exact circuit we had made in Lab 3, and simply had to modify it to accommodate operation on and storage of 8 bit words.

Our second task was to create a series of three 16 bit adders, each adding through a different algorithm and with different hardware. All of the three adders (Ripple Carry, Carry Lookahead, Carry Select) are implemented through combining the results of four 4-bit adders, creating a 16 bit sum from two initial 16 bit numbers. The Ripple Carry Adder operated similarly to the way people add two numbers on paper, adding digit by digit and carrying over the extra digits to the next bit addition. The Carry Lookahead Adder creates two additional signals (G and P) that predict if a carry is used based on a possible carry from a less significant bit position or if it is eliminated in the next position (if the result of the carry is a 0). The Carry Select Adder calculates two iterations of each 4 bit sum: one assuming a carry and one assuming no carry. The actual carry bit determines which sum is used, but the sums are pre-calculated instead of having to wait for the carry to calculate.

Part 1 - Serial Logic Processor:

Top Level Design:

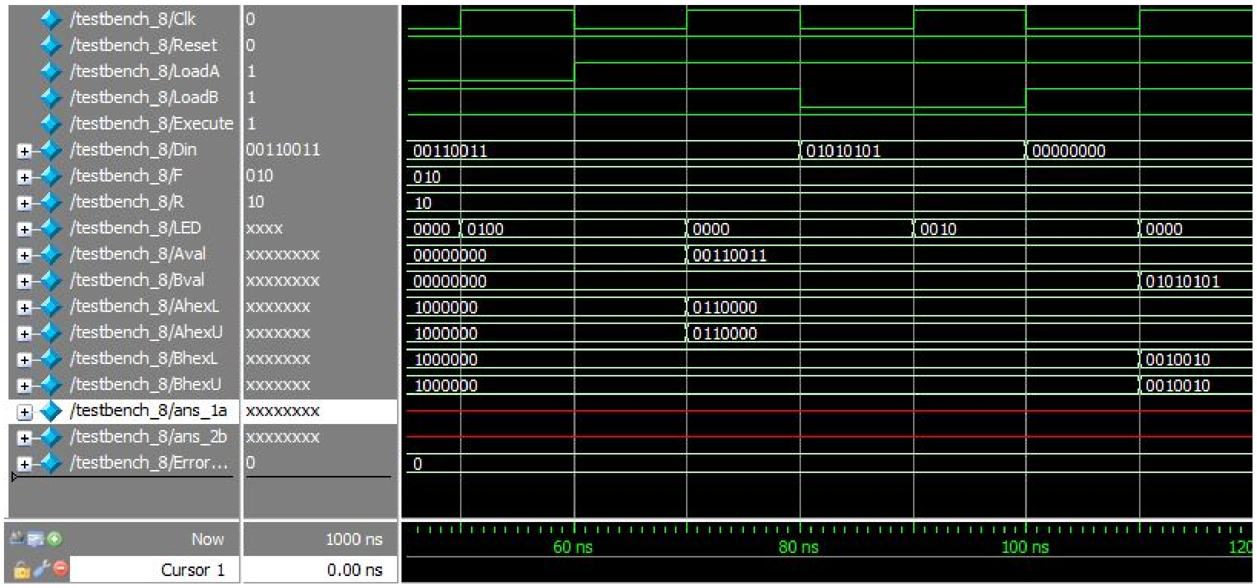


Explanation of Change from 4 to 8 Bits:

Changing the processor from 4 bits to 8 bits mainly meant adding more states. Since the 4 bits only has 6 changes from states, the 8 bit requires 4 more counter ticks, so we had to add 4 more changes in states. Other than that, extending the 4-bit registers/arrays to 8 bits meant changing a bunch of [3:0] arrays to [7:0] arrays (including DIN, RegA, RegB, DOUT etc.). We also had to add an additional HEX driver part to drive the upper 4 bits of the 8 bits, since each hex value can only handle 4 bits.

Processor Simulation:

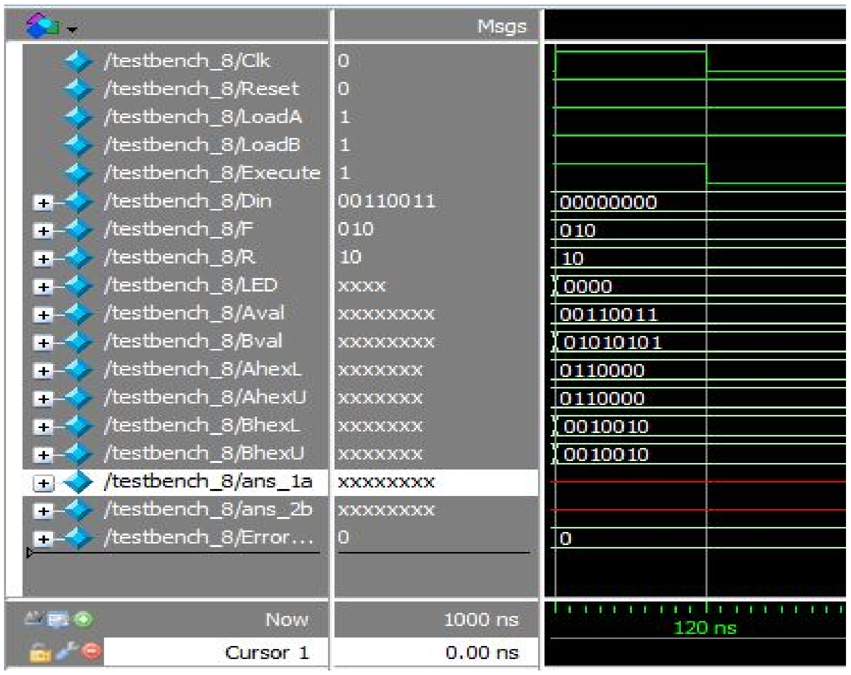
Step 1:



```
/testbench_8/Clk       0
/testbench_8/Reset     0
/testbench_8/LoadA     1
/testbench_8/LoadB     1
/testbench_8/Execute   1
/testbench_8/Din       00110011
/testbench_8/F         010
/testbench_8/R         10
/testbench_8/LED       xxxx
/testbench_8/Aval      xxxxxxxx
/testbench_8/Bval      xxxxxxxx
/testbench_8/AhexL     xxxxxxx
/testbench_8/AhexU     xxxxxxx
/testbench_8/BhexL     xxxxxxx
/testbench_8/BhexU     xxxxxxx
/testbench_8/ans_1a    xxxxxxxx
/testbench_8/ans_2b    xxxxxxxx
/testbench_8/Error...  0
Now                    1000 ns
Cursor 1               0.00 ns
```

The first step of the operation shows the loading of both A and B. A is loaded with 001100011, and B is loaded with 01010101.

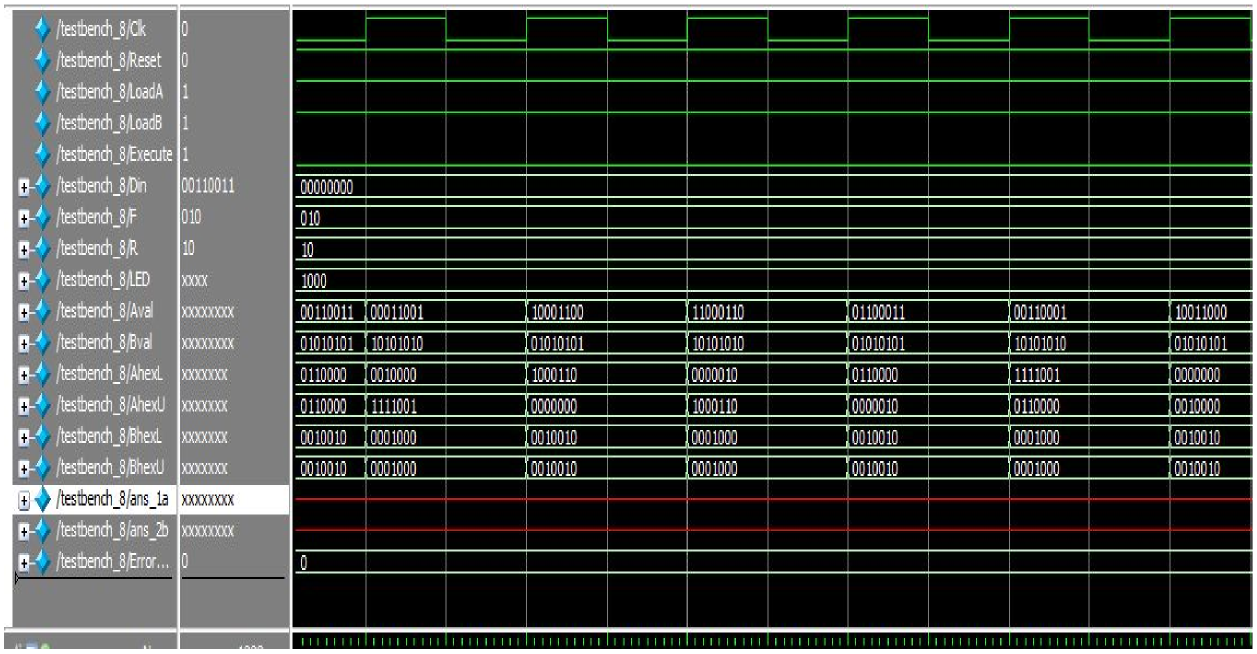
Step 2:



```
                       Msgs
/testbench_8/Clk       0
/testbench_8/Reset     0
/testbench_8/LoadA     1
/testbench_8/LoadB     1
/testbench_8/Execute   1
/testbench_8/Din       00110011
/testbench_8/F         010
/testbench_8/R         10
/testbench_8/LED       xxxx
/testbench_8/Aval      xxxxxxxx
/testbench_8/Bval      xxxxxxxx
/testbench_8/AhexL     xxxxxxx
/testbench_8/AhexU     xxxxxxx
/testbench_8/BhexL     xxxxxxx
/testbench_8/BhexU     xxxxxxx
/testbench_8/ans_1a    xxxxxxxx
/testbench_8/ans_2b    xxxxxxxx
/testbench_8/Error...  0
Now                    1000 ns
Cursor 1               0.00 ns
```

This picture represents the first real part of the operation in ModelSim. The A and B registers have both been loaded with 00110011 and 01010101 respectively. The F signal is selected to 010, which means that the result will be A XOR B. In addition, the routing has been selected so that the result of the XOR returns to the A register, since R = 10. Additionally, we can see that while execute is now 1, it is about to be triggered to 0 to start the XOR process.

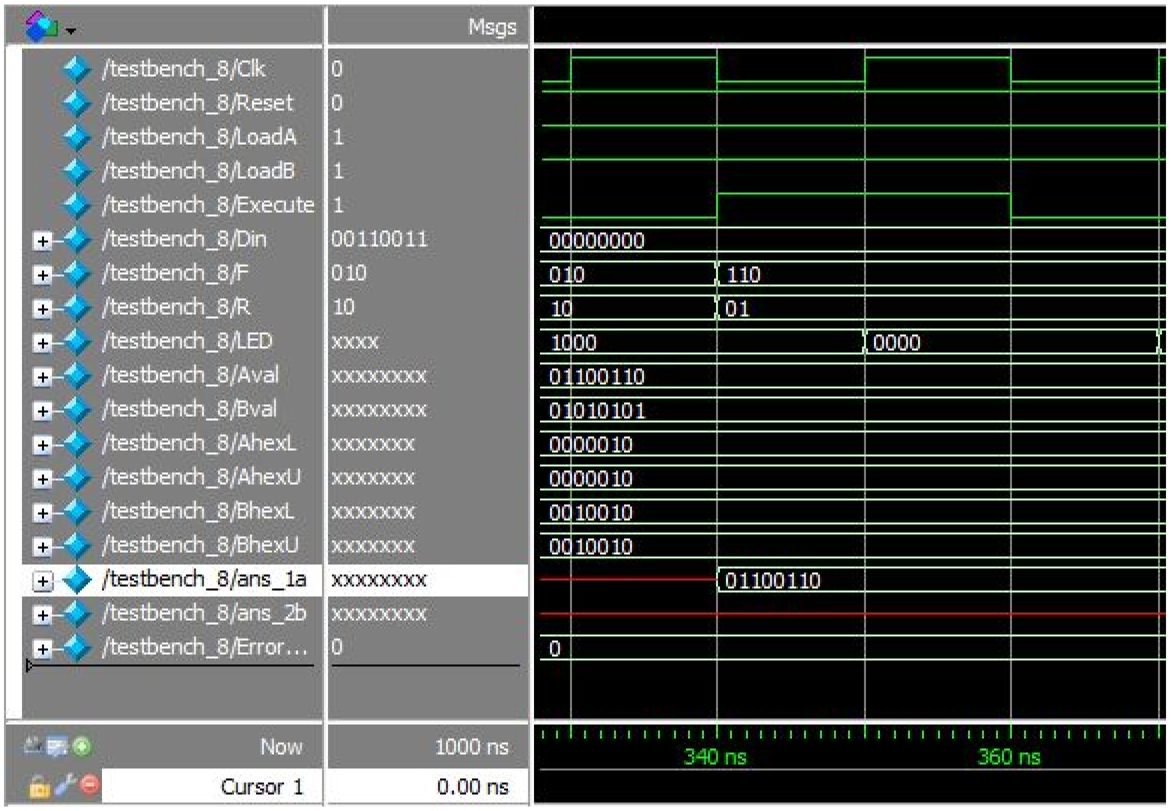
Step 3:



```
/testbench_8/Clk        0
/testbench_8/Reset      0
/testbench_8/LoadA      1
/testbench_8/LoadB      1
/testbench_8/Execute    1
/testbench_8/Din        00110011   00000000
/testbench_8/F          010        010
/testbench_8/R          10         10
/testbench_8/LED        xxxx       1000
/testbench_8/Aval       xxxxxxxx   00110011  00011001  10001100  11000110  01100011  00110001  10011000
/testbench_8/Bval       xxxxxxxx   01010101  10101010  01010101  10101010  01010101  10101010  01010101
/testbench_8/AhexL      xxxxxxx    0110000   0010000   1000110   0000010   0110000   1111001   0000000
/testbench_8/AhexU      xxxxxxx    0110000   1111001   0000000   1000110   0000010   0110000   0010000
/testbench_8/BhexL      xxxxxxx    0010010   0001000   0010010   0001000   0010010   0001000   0010010
/testbench_8/BhexU      xxxxxxx    0010010   0001000   0010010   0001000   0010010   0001000   0010010
/testbench_8/ans_1a     xxxxxxxx
/testbench_8/ans_2b     xxxxxxxx
/testbench_8/Error...   0          0
```

The third step describes how each bit of the A and B registers are being shifted out from the right as the XOR result is computed. We can see that each clock cycle, one bit is being shifted out from the right of both of the A and B registers. However, if you look closely, you can see two things: first, that the bit of B that shifts out to the right returns on the left side; and second, that the result of the XOR of the bit that shifted out of the right of the A and B registers is pushed into the left of the register A.

Step 4:



This is the final step of the adder. While a new operation is about to be started, we can view the final results of our operation here. The goal of the operation was to store the result of the operation A XOR B into the A register, which was dictated by F = 010 and R = 10. Comparing the contents of the register “Aval” and the control A XOR B value “ans_1a”, we can clearly see that the operation was successful and accurate. Noticing also that the result of A XOR B has been loaded into the A register (Aval), and that the original B value has been loaded back into the B register (Bval), we can say that our routing has also been a success. The lack of an increment in the Error count also backs up this success.

Module Descriptions for 8 Bit Logic Processor:

Module: testbench_8
Inputs: None
Outputs: None
Description: This module creates simulation conditions for processor
Purpose: This module simulates the processor operation for various logical and routing operations.

Module: compute
Inputs: [2:0] F, A_In, B_In

Outputs: A_Out, B_Out, F_A_B
Description: This module does the actual computation for the 8 logical operations.
Purpose: The purpose of this module is to be the computational unit for the circuit.

Module: control
Inputs: Clk, Reset, LoadA, LoadB, Execute
Outputs: Shift_En, Ld_A, Ld_B
Description: This module is the state machine for the whole circuit.
Purpose: This module makes sure the correct number of shifts are performed.

Module: HexDriver
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This module drives the LED display on the FPGA board.
Purpose: The purpose of this module is to ensure that we can use our FPGA board to test the functionality of our 8 bit processor.

Module: Processor
Inputs: Clk, Reset, LoadA, LoadB, Execute, [7:0] Din, [2:0] F, [1:0] R
Outputs: [3:0] LED, [7:0] Aval, [7:0] Bval, [6:0] AhexL, [6:0] BhexL, [6:0] AhexU, [6:0] BhexU
Description: This module instantiates all of the variables for inputs and output of the circuit as a whole. In addition, this module instantiates all of the other modules in this project.
Purpose: This module allows us to use the switches on the FPGA board as inputs to the FPGA circuit.

Module: reg_4
Inputs: Clk, Reset, Shift_In, Load, Execute, [7:0] D
Outputs: Shift_Out, [7:0] Data_Out
Description: This module regulates the shifting into and out of registers A and B
Purpose: This module allows both the computation of the result of the operation and the storage of the result to happen.

Module: register_unit
Inputs: Clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, [7:0] D
Outputs: A_out, B_out, [7:0] A, [7:0] B
Description: This module creates an 8 bit shift register.
Purpose: This module allows for the storage of A and B.

Module: router
Inputs: [1:0] R, A_In, B_In, F_A_B
Outputs: A_Out, B_Out
Description: This module represents the routing unit of the 8 bit processor.
Purpose: This module controls where the result of the computational unit is stored.

Module: sync
Inputs: Clk, d
Outputs: q
Description: This module creates a synchronizer with no reset.
Purpose: This module is useful for switches and buttons (typically asynchronous signals).

Module: sync_r0
Inputs: Clk, Reset, d
Outputs: q
Description: This module creates a synchronizer with a reset to 0.
Purpose: This module is useful for helping to bring asynchronous signals into the FPGA.

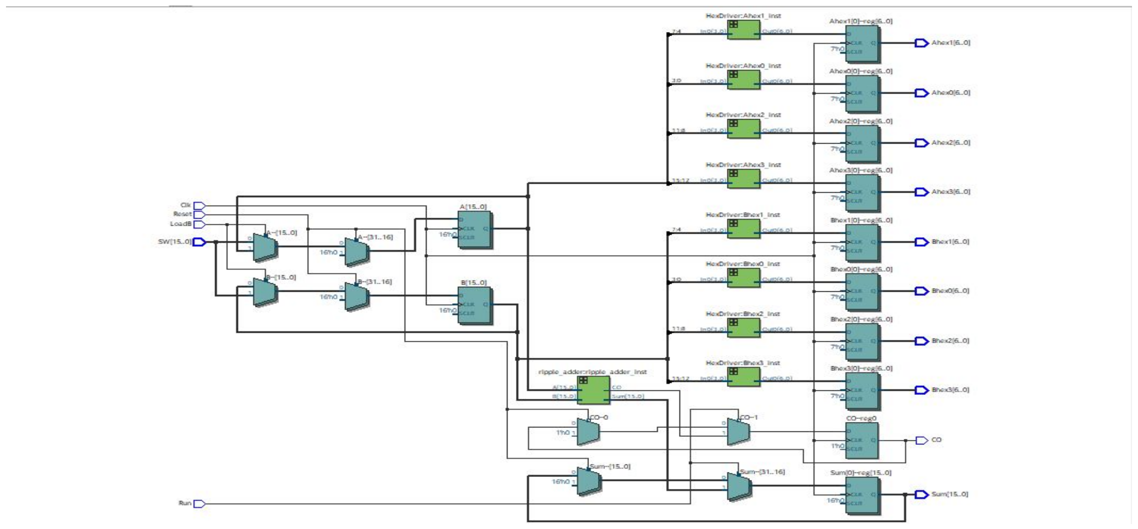
Module: sync_r1
Inputs: Clk, Reset, d
Outputs: q
Description: This module creates a synchronizer with a reset to 1.
Purpose: This module is useful for helping to bring asynchronous signals into the FPGA.

Part 2 - Adders:

Ripple Carry Adder:

The architecture of the 16 bit Ripple Carry Adder was the simplest of the three adders. The 16 bit Ripple Carry Adder can be divided into four 4-bit Ripple Carry adders. Each of these 4 bit Ripple Carry Adders consists of four 1 bit Full Adders, which add very similar to how we add in real life, creating a carry if the sum of the two bits exceeds 1. Between the 4-bit Ripple Carry Adders also exists a connection to account for the carry bits from the addition of the previous 4 bits.

<u>Block diagram:</u>



<u>Modules in Ripple Carry Adder:</u>
Module: ripple_adder
Inputs: [15:0] A, [15:0] B
Outputs: [15:0] Sum, CO
Description: This module integrates four 4-bit ripple carry adders into a single 16 bit ripple carry adder.
Purpose: This module produces the final 16 bit ripple carry adder.

Module: fourbra
Inputs: [3:0] x, [3:0] y, cin
Outputs: [3:0] sum, cout
Description: This module creates a four bit ripple carry adder by combining four full adders.
Purpose: This module is the intermediate between the single bit full adder and the 16 bit ripple carry adder. This module is used in the ripple_adder module to combine each of the four 4-bit sized blocks into the final 16 bit sum.

Module: fulladder
Inputs: x, y, cin
Outputs: s, cout
Description: This module creates a full adder to add two 1-bit numbers.
Purpose: This module serves as the building block to all of the adders with more bits involved. It is used in all three of the adders.

Carry Lookahead Adder:

Written description of the architecture of the adder:

The carry look ahead adder works through computing the carry bits prior to doing the actual computation. It does this through looking at the bits of A and B: if A and B bits are 1, then the carry is automatically 1; if A xor B is 1, and carry in bit is 1, then the carry bit is automatically also 1. The computation time is significantly decreased as a result of this logic. The same logic is then implemented into 4 bit look ahead adders, and then combined to make a 16 bit adder.

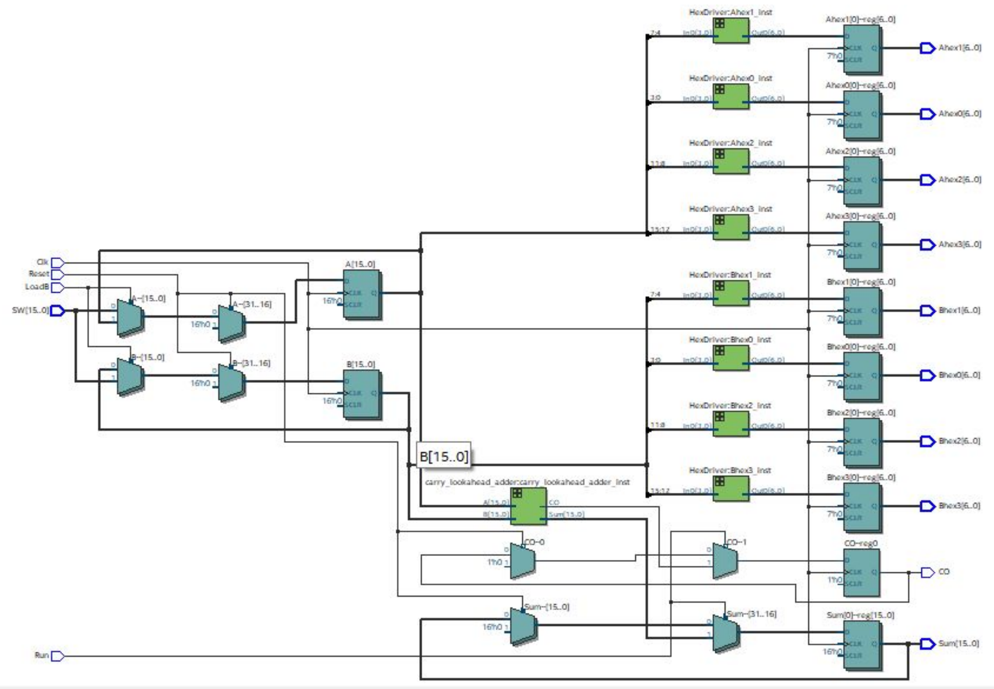
Description of the P and G logic:

P and G are variables used to determine the carry out bits of the adder. These give this type of adder the "lookahead" factor. Since the carry out bits can all be calculated before even looking at the final computation, this improves the performance time. 'P = A AND B' and 'G=(A XOR B) AND CIN.' The final carry out is 'P OR B'.

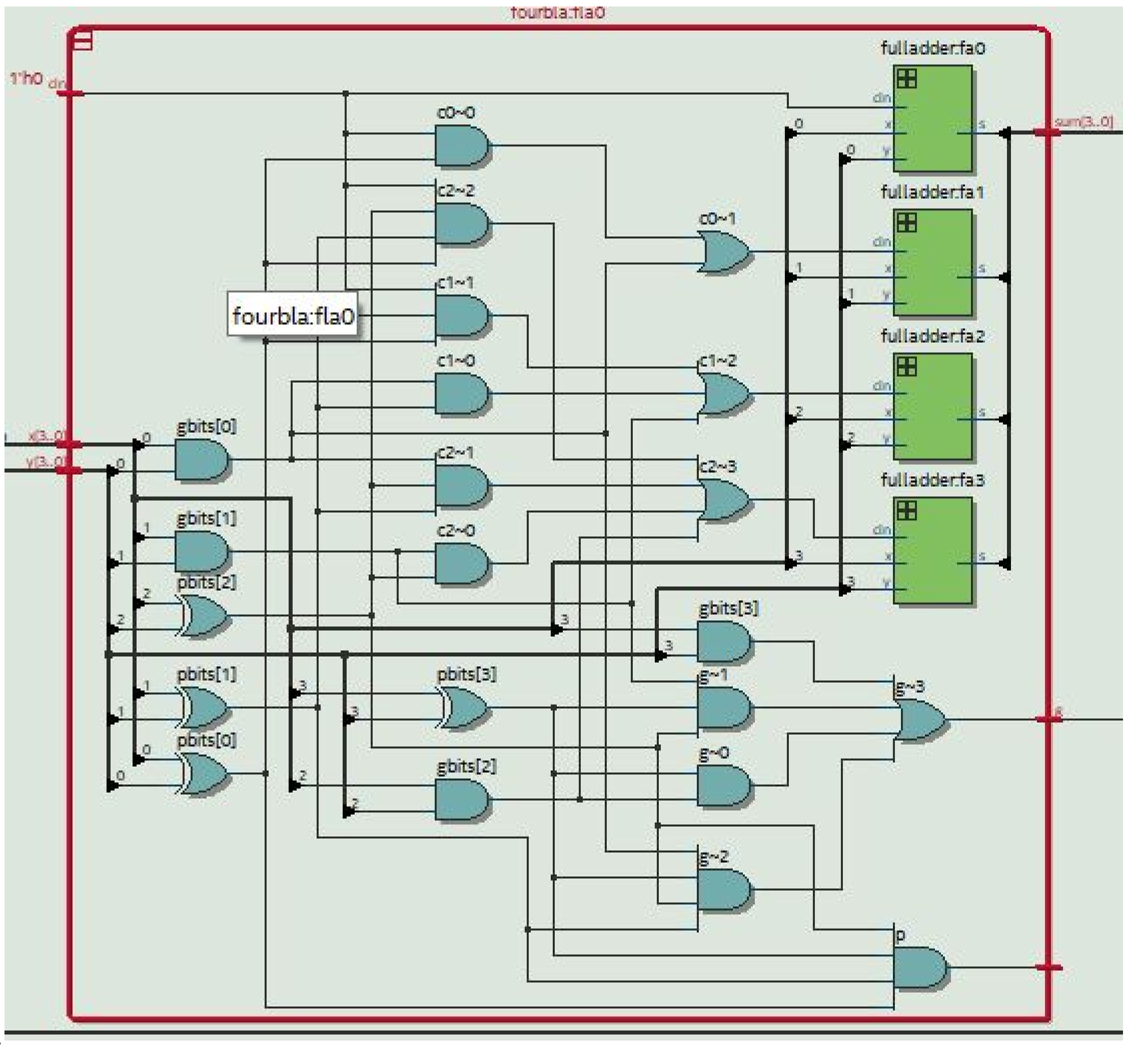
Description of Adder Partition:

We chained four 4 Bit CLA's to make a 16-bit block. This simply meant that we had to manipulate P and G depending on the prior CLA, and then generate a carry-out using the carry in.

Block Diagram:

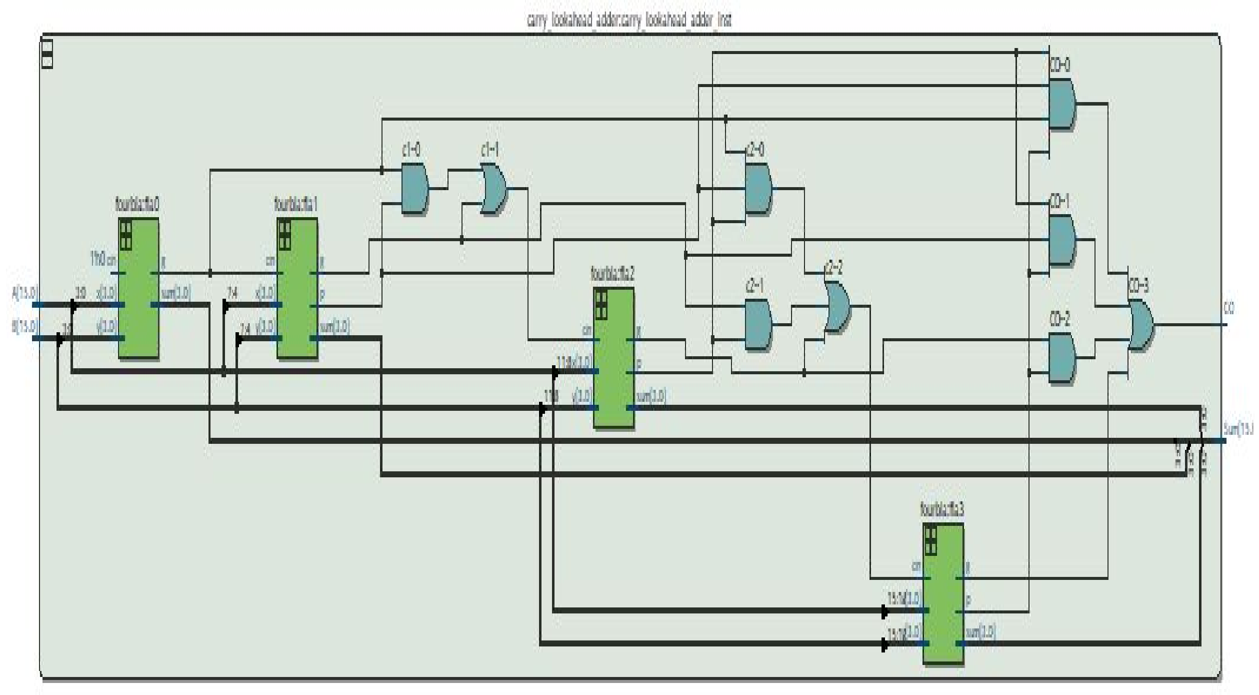


Block diagram inside a single CLA:



:

Block diagram of how each CLA was chained together:



Modules in Carry Lookahead Adder:

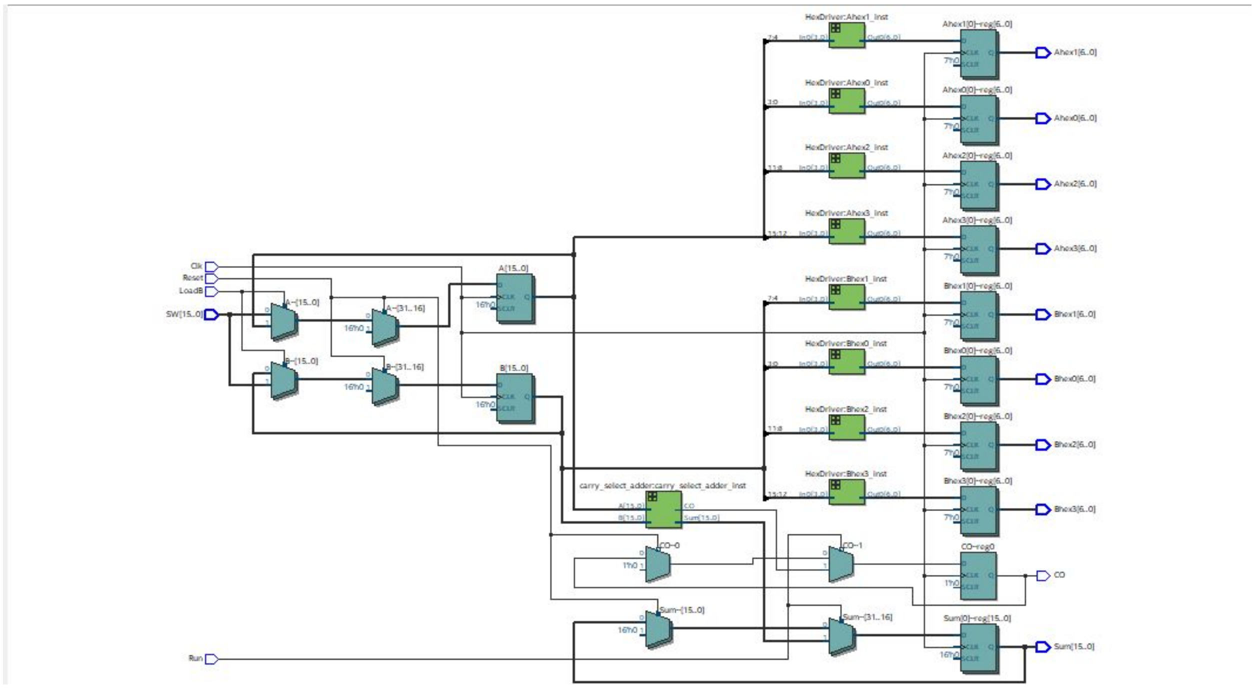
Module: carry_lookahead_adder
Inputs: [15:0] A, [15:0] B
Outputs: [15:0] Sum, CO
Description: This module integrates four 4-bit Carry Lookahead adders into a single 16-bit Carry Lookahead Adder.
Purpose: This module creates the final result: a 16 bit Carry Lookahead Adder.

Module: fourbla
Inputs: [3:0] x, [3:0] y, cin
Outputs: [3:0] sum, g, p
Description: This module combines four full adders into a single 4-bit Carry Lookahead Adder. It uses the variables G and P to predict whether a carry bit will be 0 or 1.
Purpose: This module is used four times in the carry_lookahead_adder module to create a larger Carry Lookahead Adder.

Carry Select Adder:

The 16 bit Carry Select Adder consists of seven 4 bit Ripple Carry Adders. What makes this adder different from the 16 bits Ripple Carry Adder is that the results of 4 bit additions for the bits [15:12], [11:8], and [7:4] are precalculated. Two iterations of each of these additions are calculated: one assuming a carry in of 0 and the other assuming a carry in of 1. The four least significant bits [3:0] are added with a single 4-bit Ripple Carry Adder, and the carry out bit from this Adder goes into a MUX, which selects the appropriate precalculated sum from the bits [7:4] based on whether the carry out bit ended up being 0 or 1. This same process carries through the rest of the bits until the final sum is calculated.

Block Diagram of the whole CSA circuit containing adders, multiplexers, and glue logic:



Modules in Carry Select Adder:
Module: carry_select_adder
Inputs: [15:0] A, [15:0] B
Outputs: [15:0] Sum, CO
Description: This module integrates seven 4-bit Ripple Adder modules and 3 MUX's to create the Carry Select Adder.
Purpose: The purpose of this module is to create the carry select adder.

Module: MUX
Inputs: [3:0] D1, [3:0] D2, select, c1, c2
Outputs: [3:0] Dout, c

Description: This module essentially creates a 2:1 Multiplexer.
Purpose: This module selects between the precalculated sums with 0 carried in and 1 carried in.

<u>Other Modules:</u>
Module: lab4_adders_toplevel
Inputs: Clk, Reset, LoadB, Run, [15:0] SW
Outputs:
Description:
Purpose:

Module: HexDriver
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This module drives the Hex LED's on the FPGA board that display the contents of the registers A and B.
Purpose: This module essentially allows us to use the display on the FPGA board to verify the correct operation of our adders.

<u>Comparison between Adders:</u>

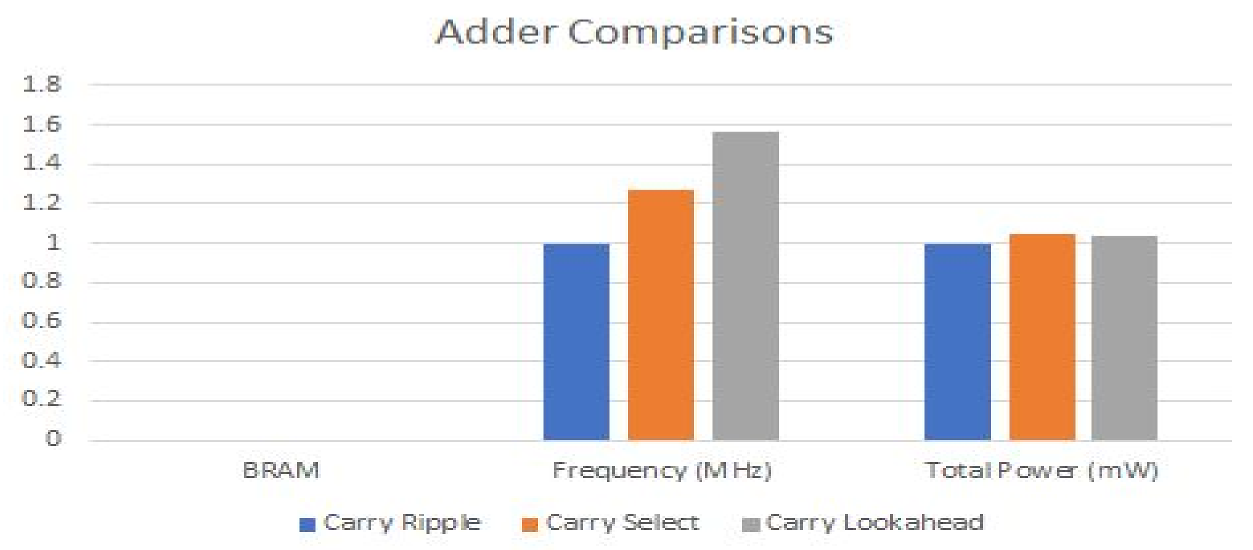
These three adders are quite distinct in function, and they all come with their own pros and cons. The 16 bit Ripple adder is the slowest of the three adders, as its speed depends on each carry bit travelling between the 4-bit blocks, but since the 16 bit Ripple Adder only requires 16 full adders, it takes the smallest area of the three adders.

The 16 bit Carry Lookahead adder is the fastest of the three adders, since its design allows it to function without waiting for a carry bit between the adders. However, this makes it the most complicated of the three, since there is a significant amount of logic that has to be implemented in order to "predict" the carry bit. This circuit is not as large as the Carry Select Adder (because it doesn't have as many adders), but the sheer amount of logic necessary for G and P make it larger than the Ripple Carry Adder, even though it has the same number of Full Adders. Lastly, the Carry Select Adder somewhat of a compromise between the Ripple Carry Adder and the Carry Lookahead Adder in terms of speed and complexity.

The Carry Select Adder pre-calculates an iteration of each 4-bit intermediate sum for each possible carry, so it is faster than the Ripple Carry Adder, but it still has to wait for the carry bit from the initial 4-bit intermediate sum to operate, so it is not as fast as the Carry Lookahead adder. The Carry Select Adder has some pre-calculation, so it is more complex than the Ripple Carry Adder, but the G and P signals in the Carry Lookahead Adder are more complex than simply precalculating the intermediate 4-bit sums for each possible carry. However, even though the Carry Select Adder doesn't have more logic than the Carry Lookahead Adder, the 16 bit

Carry Select Adder is comprised of 28 full adders, which take up far more space than the 16 full adders involved in each of the Carry Lookahead and Ripple Carry Adders.

	Carry Ripple	Carry Select	Carry Lookahead
Memory (BRAM)	0	0	0
Frequency	62.1 MHz	78.9 MHz	97.24 MHz
Total Power	155.80 mW	162.62 mW	162.02 mW



Post Lab Questions:
1. Q: How do we gain “speed up” from the Carry Select Adder?
A: The reason why the ripple adder takes time is because the more significant bits have to wait from the carry in bits from the less significant bits. The carry select adder pre-calculates the sum of the more significant bits; one sum predicting a carry in bit of 1 and another predicting the carry in bit of 0. Instead of the adder waiting for the carry in bit to calculate, the carry in bit is used as the select bit of a mux that chooses between the “zero carry in” and “one carry in” sums that were pre-calculated. This is where the time is saved in using the carry select adder.

2. Q: 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?

A: The IQT has resources as follows: LUT = 72, Memory = 0, Flip Flops = 43. By our estimation, the TTL 8 Bit Processor would have these resources: LUT = 143 , Memory = 0, Flip Flop = 24. This means that the IQT is a better design, since it utilizes less total flip flops and gates to produce the same result. This makes sense, as a design created and optimized by a program like Quartus is likely to be more efficient than a design wired by hand, which probably uses more gates than necessary and is susceptible to human error.

3. Q: 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.
A:

	Carry Ripple	Carry Select	Carry Lookahead
LUT	114	125	137
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flop	105	105	105
Frequency	62.1 MHz	78.9 MHz	97.24 MHz
Static Power	98.55 mW	98.57 mW	98.57mW
Dynamic Power	2.92 mW	6.89 mW	6.42 mW
Total Power	155.80 mW	162.62 mW	162.02 mW

4. Q: 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?
A: The breakdowns from the data plot make perfect sense to me. Since we didn’t use any BRAM in our design, there is no BRAM being used by any of the 3 adders. In addition, the design with the most components (Carry Select) consumes the most power. Since the Carry Lookahead adder has more predictive power than either of the other two adders, it can compute the sum much faster, which we see is true in the frequency measurements in the above table. It makes sense then that the Ripple Carry adder is the slowest, since it has the simplest design with no shortcuts to make adding faster like the other two designs have. I was surprised that the Carry Lookahead adder consumed almost as much power as the Carry Select; even though it has less adders, the combinational logic of G and P must be the source of this added power consumption. It also makes sense that the Ripple Carry adder consumes the least amount of

power, since it is the simplest design with the least number of components. In summation, all of these measurements agree with the general theoretical design expectations.

Conclusion:

We encountered slightly more issues in this lab than previous labs simply because we had to get accustomed to using the syntax of a programming language that was unfamiliar to us. One such error involved the full adder module. Knowing that we had to use the full adder for all three adders, we copied the full adder module from the Ripple Carry sv file into the Carry Lookahead sv file. However, we didn’t realize that we didn’t need to have this module in both of the files; as long as the module was in the project once, we could cite and use instances of it anywhere in the code, even in a different file. We also had trouble with ‘If’ statements in trying to implement the MUXes for the Carry Select Adder, but learned that we had to add the ‘begin’ and ‘end’ statements to the ‘If’ statements.

The other issue we had was with programming the FPGA itself. I had no issues installing any of the software and drivers associated with ModelSim and Quartus, but we couldn’t get the FPGA to program. It turns out, since we had created the 8 bit processor first, we were pressing the Analysis button instead of the Compile button, since only analysis is necessary for the ModelSim to work. After a few excruciating hours, the TA in open lab pointed this out to us, and we finally got the correct software loaded into the FPGA, which worked immediately.

In the end, this lab was similar to lab 1 in that it was mostly an introduction to a relatively unfamiliar process (TTL in that case, SV in this lab) that eased us into technology that we would have to use for future labs. We learned the inner workings of three different kinds of adders, and also learned how to modify past circuits to be able to handle and operate on more bits. My only recommendation for the lab manual for next semester would be to include more examples on how to use the syntax of SV, especially the ‘If’ statement.