Lab 4 Report

ECE 385

Friday (ABB)

11:00AM-1:50PM

Intro to SystemVerilog, FPGA,
CAD, and 16-Bit Adders

Ritvik Avancha and Steven Phan
(rra2 and sphan5)

Table of Contents

Introduction **2**

Part 1 - Serial Logic Processor **3**

Figure 1.1: 8-bit Processor High-Level Diagram 3

Figure 1.2: SLP Waveform Annotations 4

Part 2 - Adders **4**

Figure 2.1: Ripple Adder Block Diagram 4

Figure 2.2: 16-bit Carry Lookahead Adder (Chained) 5

Figure 2.3: 4-bit Carry Lookahead Adder Block Diagram 5

Figure 2.4: 16-bit Carry Select Adder (Chained) 6

Figure 2.6: Performance Chart 7

Figure 2.7: Normalized Performance Graph 8

Post-Lab Questions **8**

Figure 3.1: 8 Bit-Serial Processor from IQT vs TTL Design 8

Figure 3.2: Design Resources and Statistics 9

Conclusion **9**

Lab 4 Report

Introduction

After three labs of TTL on breadboards, Lab 4 taught the basics of RTL design on FPGAs in SystemVerilog and the connection between SystemVerilog and physical hardware. In addition to learning the basic syntax of SystemVerilog, this lab also teaches the fundamentals of quartus designs and test-benching the designs. Quartus is essential in learning about performance and drawbacks of certain implementations. In order to explore these concepts, different types of adders are implemented (ripple, select, lookahead), and the logic processor from lab 3 is extended to 8 bits on the FPGA.

At a high level, the 8-bit processor functions exactly the same as the one implemented on breadboard during lab 3. Three switches F[2:0] determine the bitwise logic operation that is performed on the values in the registers. The switches R[1:0] determines where the output/data in the registers are routed to after 8 computation cycles are done. Data can be parallel loaded into the registers through 7 switches D[7:0].

All three adders do the same function: add two inputs A and B. The way a ripple adder works is by chaining multiple full adders together and connecting the carry out of the nth adder to the (n+1)th adder. The lookahead adder is composed of full adders that generate two extra bits: propagate (P) and generate (G). Two generate G at a given adder, G = A * B. To generate P at a given adder, P = A ^ B. The carryout for the next adder can be calculated as $C_{n+1} = (C_n * P_n) + G_n$. This way, the ripple effects can be mitigated since the (n+1)th carry-in bit doesn't always require the previous nth carry-out bit to be calculated.

Part 1 - Serial Logic Processor

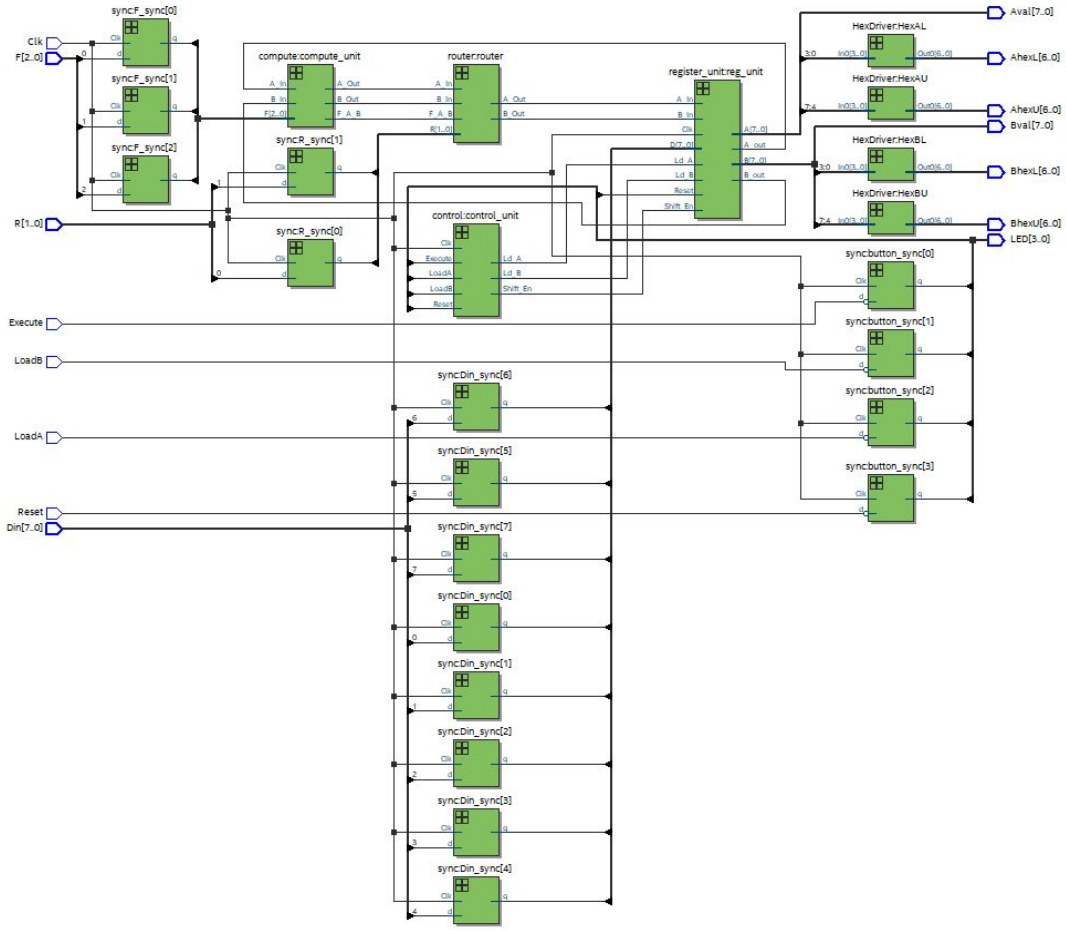

Figure 1.1: 8-bit Processor High-Level Diagram

The main changes made in the provided files were changing the D_in buses from [3:0] to [7:0]. In addition to extending the bits, the enumerated states the the control unit were extended from 4 states to 8 bits. In order to do a computation on 8 bits, 8 cycles are required. So, the enum logic[2:0] was extended to [3:0] in order to allow more states. The final state is changed to J (hold), so that was also modified in the control unit code. Another change was making sure that the upper nibbles of both registers are displayed on the hex drivers, so that was adjusted in the processor file. The last big change was dating the shifting data in the register unit from [3:1] to [7:1] to represent a shift in the left most bits.

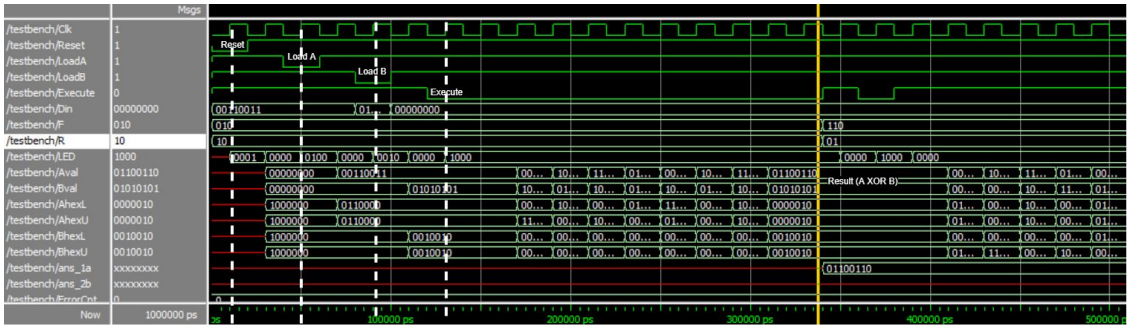

Figure 1.2: SLP Waveform Annotations

In this testbench waveform, 8h'33 was loaded to A, 8h'55 was loaded to B as scene in the waveforms (D_in[7:0] and the Load A/B signals). The function was set to 010 (XOR), routing to 10, then execute was flipped on. The expected value, 8h'66 is seen in register A after 8 clock cycles (between execute and A^B), then the final result is between the yellow dashed lines.

Part 2 - Adders

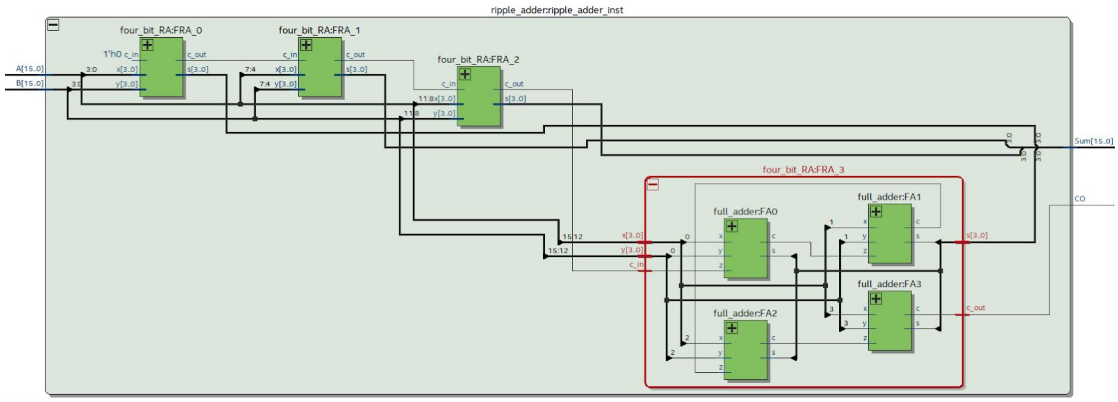

Figure 2.1: Ripple Adder Block Diagram

This 16-bit carry ripple adder (CRA for short) is composed of four 4-bit CRAs. Each 4-bit CRA is composed of four full adders, where the carry-out of one adder is connected to the carry-in of the next adder. With this architecture, computation time is significantly longer due to propagation within the adders in order to compute the sum at an adder later on in the sequence.

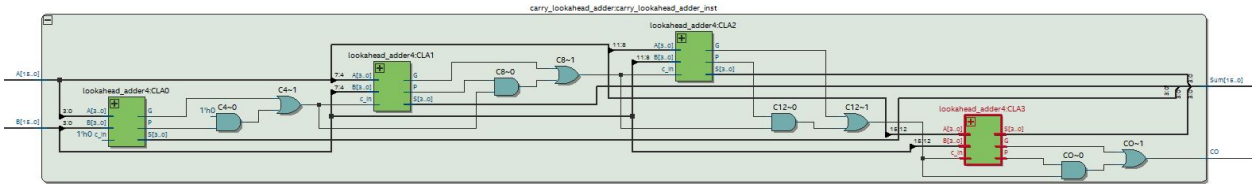

Figure 2.2: 16-bit Carry Lookahead Adder (Chained)

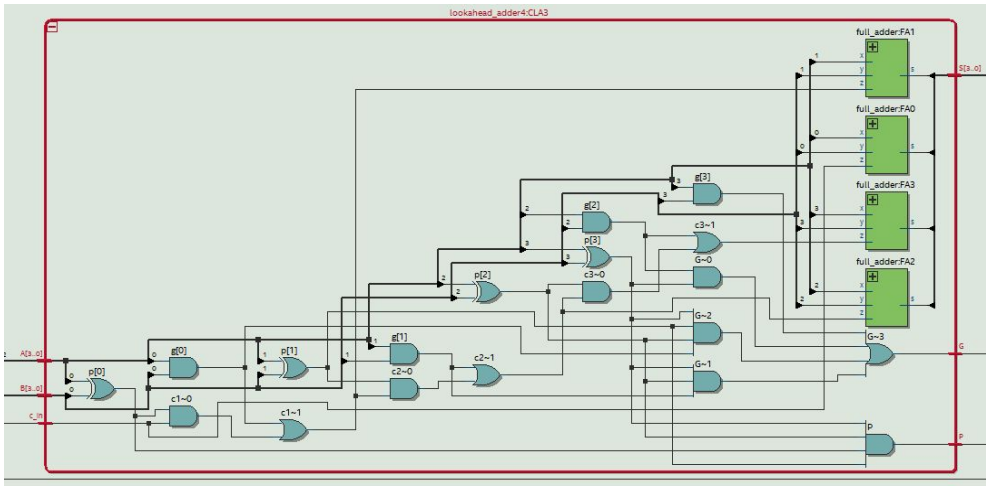

Figure 2.3: 4-bit Carry Lookahead Adder Block Diagram

This 16-bit carry lookahead adder (CLA for short) is composed of four 4-bit CLAs. Essentially, four 4-bit CLA modules, illustrated in Figure 2.3, are chained together, illustrated in Figure 2.2, to complete the 16-bit CLA. One 4-bit CLA is composed of 4 full adders and generates a set of bits called a propagate "P" and generates "G". P is A^B, and G is A*B. This logic is used to calculate the carry for the next adder unit. For example, if two inputs A = B = 1, then G is 1. The idea behind a generate bit is that since both inputs are 1, the carry out is a 1 regardless of what the carry in is. This significantly improves adder computation time since the previous carries are not needed to compute an accurate output. However, if A = 1 and B = 0, then the carry out depends on the carry in, hence the overall carry-in of an nth adder can be expressed as $C_n = G_{n-1} + (C_{n-1}*P_{n-1})$. Worse case, G_{n-1} ends up being 0, so ripple carry is required, otherwise, the carry can always be calculated relatively soon. Contrasting this with the regular ripple adder, the lookahead adder generates carry logic based on the current inputs, and does not always require the previous adder's carry out. Notice how this design is hierarchical - the 16 bits of the inputs A and B are divided into four groups of 4 bits. The first four bits go into the first 4-bit CLA unit, the second set goes into the second 4-bit CLA unit and so on. Each CLA unit then

generates a set of bits P_{group} and G_{group} where $P_{group} = P_0*P_1*P_2*P_3$, and $G_g=G_3+G_2*P_3+G_1*P_3*P_2+G_0*P_3*P_2*P_1$. Then, the overall carries between the four 4-bit CLA units are computed using the equation for C_n, where G and P are replaced by their group variants. While this improves the computation time significantly, it takes up much more space than the ripple adder. By creating submodules composed of smaller units, a hierarchy is created. So if necessary, extending this to a 64 bit CLA would take little effort since it would be easy to chain another module to the already existing module(s) and avoid slow rippling of the carry bits.

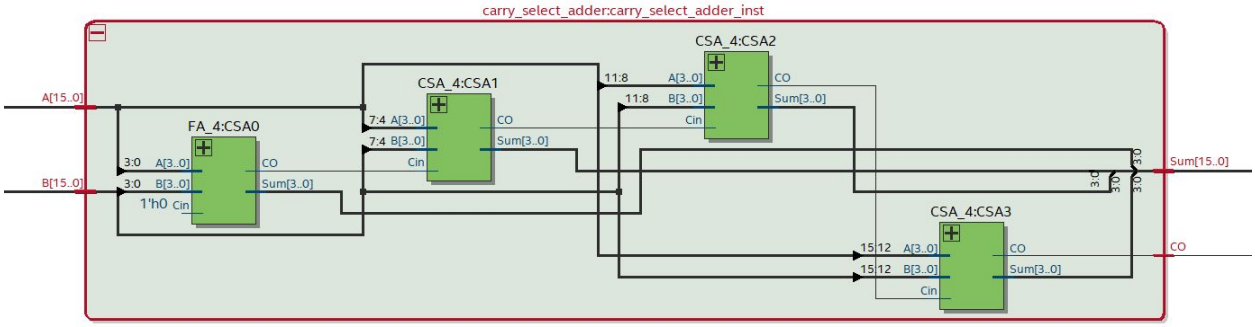

Figure 2.4: 16-bit Carry Select Adder (Chained)

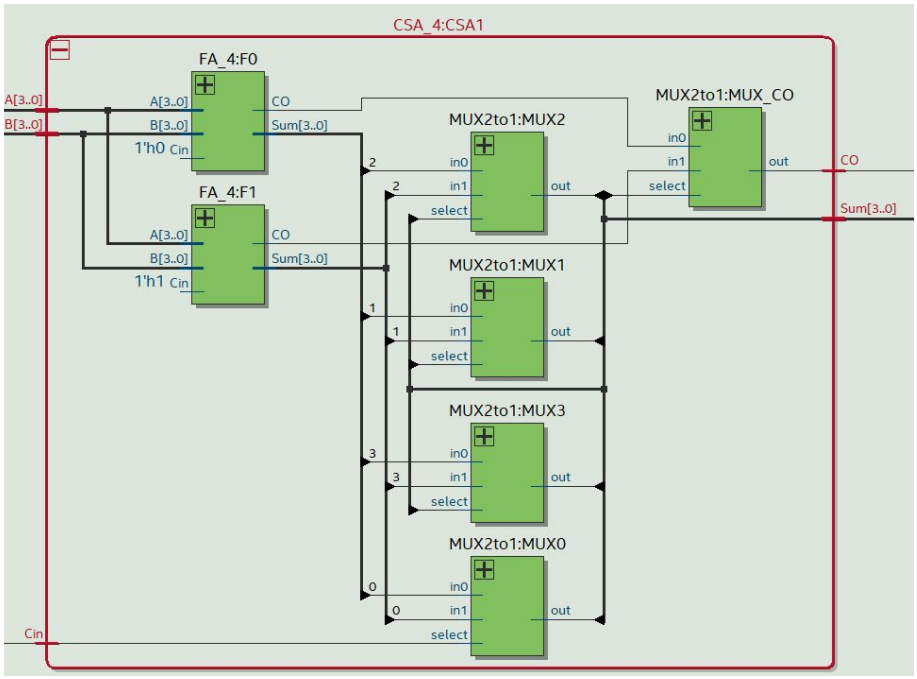

Figure 2.5: 4-bit Carry Select Adder

Figure 2.4 illustrates our design for a 16 bit CSA (Carry Select Adder). The 16-bit CSA is made up of three 4-bit CSA and a 4-bit full adder. Each 4-bit CSA module is made up of two 4-bit full adders as seen in Figure 2.5. The 4 bit adders are hooked up to a logical low and a logical high for carry in respectively. The sum of each of the four bit adders is sent into a 2 to 1 multiplexer with the select driven by the sum of the previous bit where the first bit is selected by the sum of the full adder. A CSA allows for the computations of all possible outcomes in one clock signal. The speed is only limited by the rippling selection speed of each multiplexer. A CRA (Carry Ripple Adder), on the other hand, is dependent on the rippling effect of the carry out signal of the previous adder along with the compounding effect of waiting for the rising edge of a clock high leading to a slower computation speed.

Between these three adder implementations, there are tradeoffs in performance, area, and complexity associated with each implementation as cited in Figure 2.6. While the CRA is the simplest and takes up the smallest area of the three implementations, the CRA is the slowest. The CLA is the fastest of the three but the speed comes with the tradeoff of a large area and the highest complexity. The CSA functions as a compromise between the CRA and CLA in terms of performance, area, and speed.

	Carry Ripple	Carry Select	Carry Lookahead
Mem (BRAM)	0	0	0
Frequency (MHz)	62.1	63.36	79.43
Total Power (mW)	155.80	156.79	162.26

Figure 2.6: Performance Chart

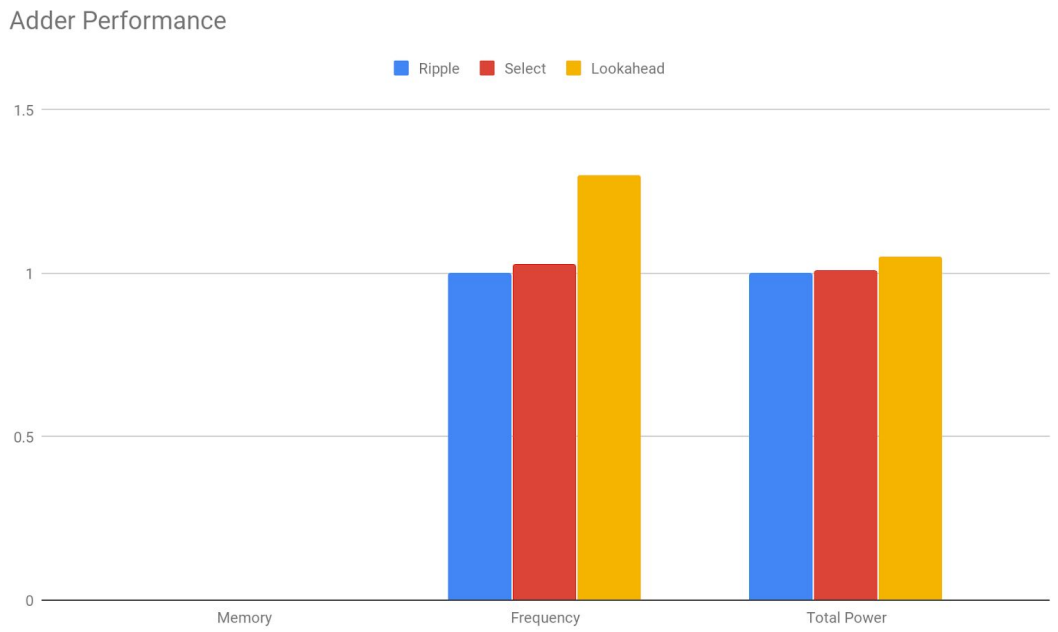

Figure 2.7: Normalized Performance Graph

Post-Lab Questions

1. The TTL design makes a more efficient use of resources compared to the synthesised circuit in the IQT. A potential reason for this difference is a lack of constraints regarding efficiency placed on the synthesiser. The synthesiser did not go through a sufficient redesigning state as it had no constraints for efficiency. Instead the synthesiser output the first circuit which could perform the computation without any regard for efficiency.

	IQT	TTL (4-Bit)	TTL (8-Bit)
LUT	72	16	*18
Memory (BRAM)	0	0	*0
Flip-Flop	43	9	*17

Figure 3.1: 8 Bit-Serial Processor from IQT vs TTL Design

* : Extrapolated data from 4-Bit TTL

2. The 4x4-bit hierarchy is not ideal for our case as a 4x4-bit experiences ripple effects between the 4 modules. In order to create an ideal hierarchy for this lab, a 1x16-bit hierarchy would allow for the most efficient system as each set of four full adders is not

dependent on the rippling effect of the carry out bit from the previous set of four full adders. To test if this design is the ideal hierarchy, we would synthesize the circuit for the FPGA and check the design resources for the max frequency and the LUT usage to guage the speed and area usage.

3. Figure 3.2 breaks down the difference between the three adders in various categories. The general data plot follows expectations. As displayed in Figure 3.2, the CRA has the lowest total power draw of 155.8mW along with the lowest usage of the LUT. Consequently the CRA also has the slowest frequency of 62.1 MHz between all three adders. On the other hand, the CLA appears to be the fastest and the highest total power draw of all the adders with a frequency of 79.43MHz and a total power draw of 162.26mW. The CSA manages to straddle in between both the adders with a frequency of 63.36MHz and a total power draw of 156.79mW.

	CRA	CLA	CSA
LUT	114	128	132
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flop	105	105	105
Frequency (MHz)	62.1	79.43	63.36
Static Power (mW)	98.55	98.57	98.55
Dynamic Power (mW)	54.33	6.62	2.52
Total Power (mW)	155.80	162.26	156.79

Figure 3.2: Design Resources and Statistics

Conclusion

During the course of the lab, a number of bugs were encountered. A significant portion of the bugs were caught by compilation errors from Quartus. Bugs such as assigning a packed array to an unpacked array, lacking a module declaration, and assigning a wire to logic type were encountered. Many of the bugs can be credited to a lack of experience with Quartus in terms of syntax and proper convention. For future reference, we would recommend providing a document

compiled with common Quartus issues such as undefined toplevel or packed arrays vs unpacked arrays along with general fixes for said bugs. The IQT was a good introduction to the Quartus environment but a similar tutorial for testbenches exploring the full functionality of test benches would have been much appreciated. The provided testbench tutorial video seemed to fall short in that regard.