ECE 385

Spring 2020

Experiment # 4

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

Megh Shah, Ishaan Patel
Lab Section: ABC, Tuesday 11:30 AM
TA: Gene Shiue, David Zhang

Introduction:

There are many ways one can build a circuit or design. Whether it is by physically building the entire circuit, or using software to emulate it, there is no doubt that there are many different approaches. In this lab, we transferred from building low-level physical circuits to creating software-based simulations in System Verilog. One circuit that we built is an 8-bit serial logic processor which takes in two 8-bit data and performs one of eight logical operations on them, and ultimately stores the result into one of the two registers. Other circuits that we were tasked to build included three different types of ripple adders. Each of these ripple adders had the same output and result, however they operated different as some, such as the lookahead and select adder, were more efficient and faster while others, such as the basic ripple adder, were much simpler. All in all, we believe that this lab provided activities that helped us transition from building physical circuits to creating software simulations.

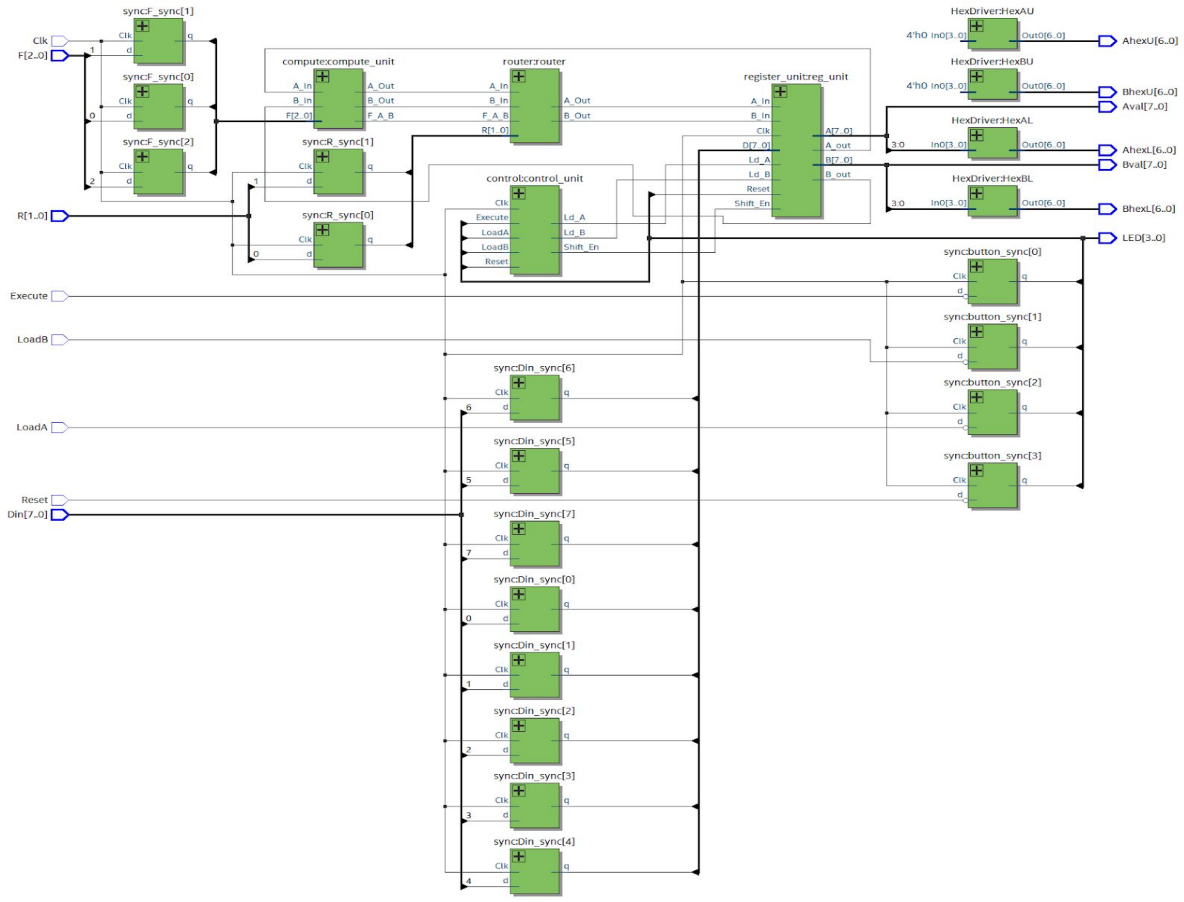
Answers to Pre-Lab Questions:

a. **SEE SERIAL LOGIC PROCESSOR FOR DIAGRAM AND WAVEFORM**

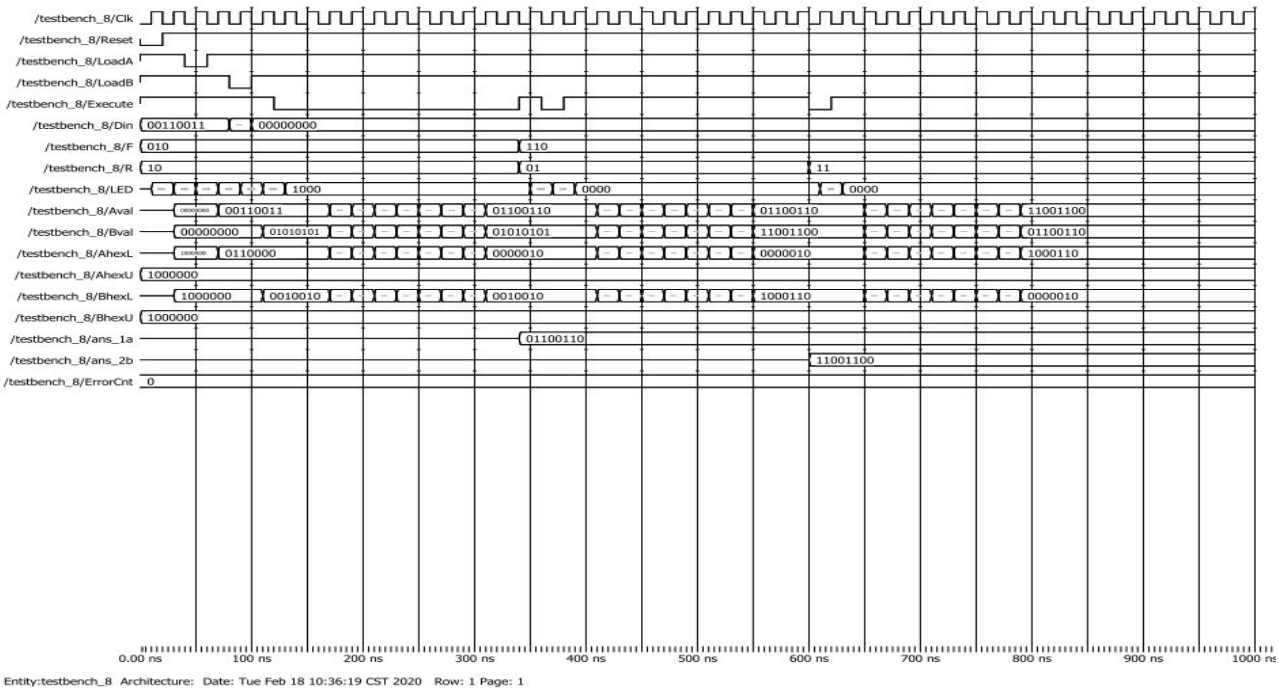
c. **SEE PART E FOR PERFORMANCE AND DOCUMENT ANALYSIS**

Serial Logic Processor:

a. Top-Level Diagram of the 8-Bit Serial Logic Processor



b. To extend the provided code to operate on 8-bits from 4-bits, we first had to ensure that the data switches were extended to 8 switches instead of 4. After that, we had to extend the shift registers, so they could hold and shift 8-bits instead of 4-bits. We also had to extend registers A and register B to hold 8-bits worth of data. In order for the logic operation to work as intended, we have to make sure that we wait extra clock cycles, so that the logic and routing unit can operate on each of the new bits as well as the old one.



c. Waveform Simulation of the 8-bit Serial Processor

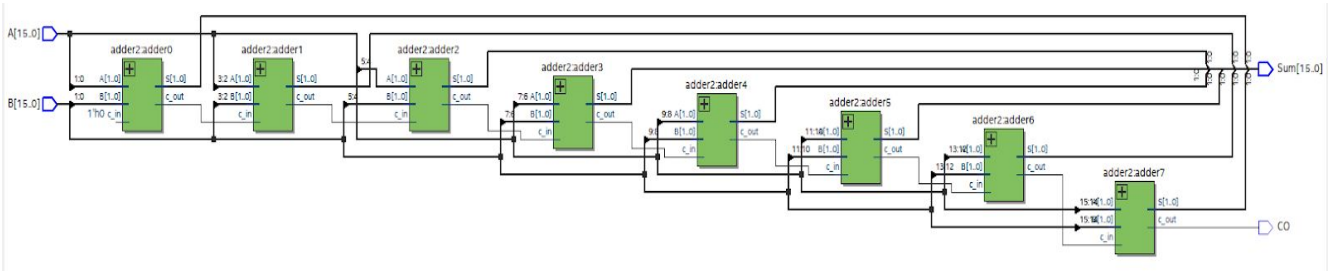
Regarding this test simulation that is provided above, we are performing an XOR operation on the values stored in register A and B. First, we set the data switches to our desired 8-bit data. Afterwards, when we flip Load A, that data gets stored into register A. We do this same process, but with a different 8-bit data for register B. We then set the function switches to “010” to represent the XOR operation and routing to “01” to store the result in register B and keep register A unchanged. After enough clock cycles have passed, we can see that the correct result of XORing the data in A and B is stored in register B, and register A remains the same.

Adders

a. Ripple Carry Adder:

i. The 16-bit ripple carry adder is designed and consists of 8 2-bit full adders. These full adders are connected in series with each of them being linked with one another through their carry-out and carry-in bits. This adder operates by taking in pairs individual bits of the two data, performing addition on them, then sending the carry-out bit, if there is one, into the adder of the next significant bit. This process continues until 16-bits are added. The outputs for this adder are the final carry-out bit and the 16-bit sum.

ii. Block Diagram for the Ripple Carry Adder



b. Carry Lookahead Adder:

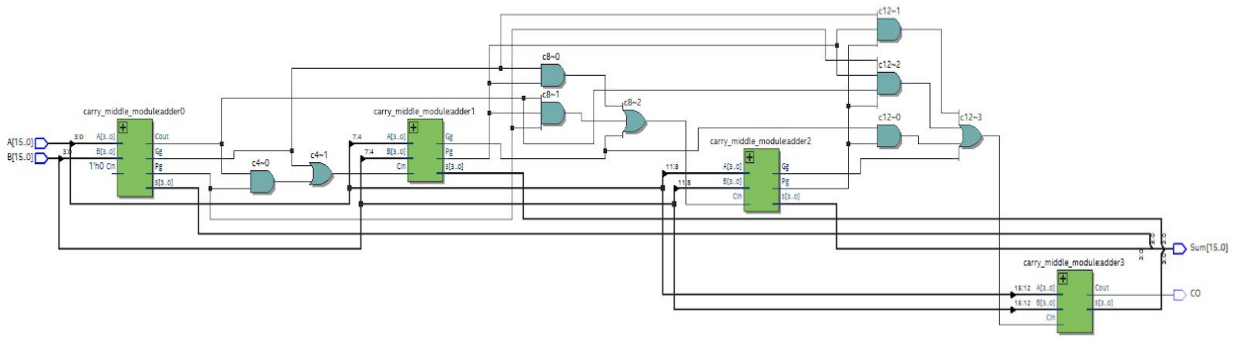
i. The carry lookahead adder operates differently as it computes the carry-in bits ahead of the actual carry bit by using the propagating and generating bits. This adder is designed by using a 4x4 hierarchical structure. This allows only four carry-in bits to be computed as each group its one set of adders which is in charge of adding and computing the carry bit for 4-bits. Each group is chained together with their carry-bits. In the end, the final carry-bit of the entire structure is the carry-out bit for the entire addition of the two 8-bit data.

ii. The P and G logic is used to predict what the carry-out bit from the previous group of adders will be. The generating bit, G, will be a "1" if and only if both A and B are "1", regardless of the carry-in bit. This is because if both bits are "1", then the sum will always produce a carry-out bit of "1". The propagating bit, P, represents the possibility of the carry-out bit being a "1" if either A or B is "1". This is because it depends on the carry-in bit as that could change the actual carry-out bit from a "0" to a "1". The propagating and generating bits of all the groups of the CLA adders are used to predict the carry-bit of all future groups of carry lookahead adders.

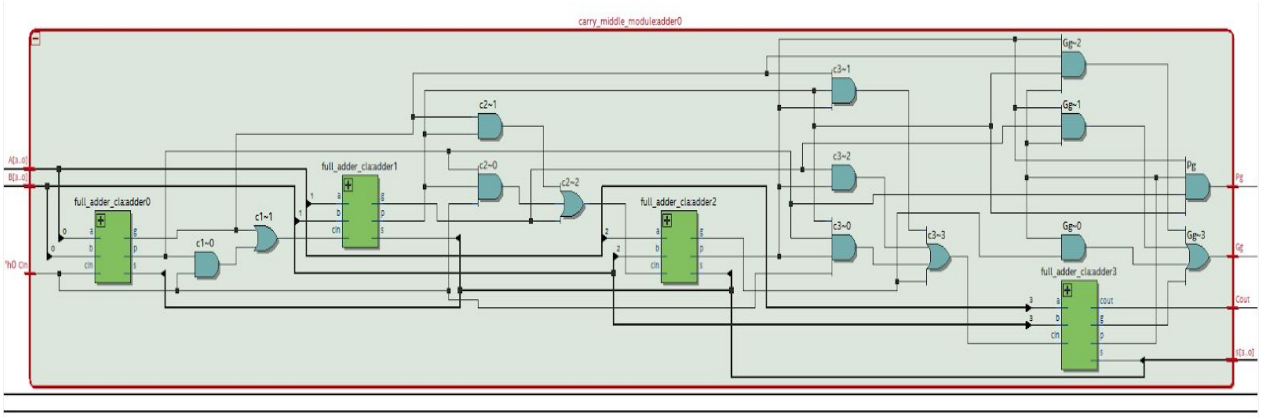
iii. We created the hierarchal 4x4 design for the carry lookahead adders by dividing the 16-bit adders into 4 sections where each section was in charge of computing the carry-out bit, propagating bit, and generating bit for that

group of 4-bits. Those three outputs for this group, and all other groups, would then go into a 16-bit lookahead carry unit that took these carry bits and propagating and generating bits and used it to compute the final carry-out bit, and final propagating and generating bits.

iv. Block Diagram



v. Block Diagram inside a Single CLA (4-bits)



vi. Block Diagram of how each CLA was chained together
SEE PART IV. SAME BLOCK DIAGRAM

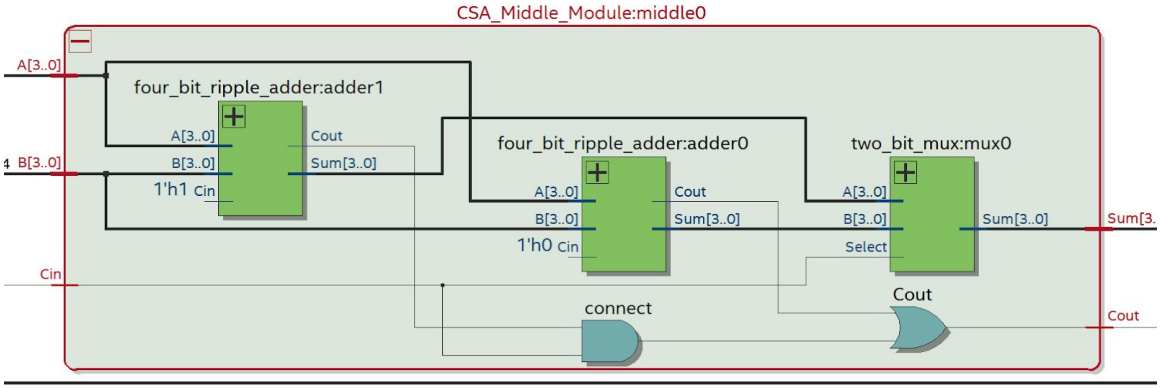
c. Carry Select Adder:

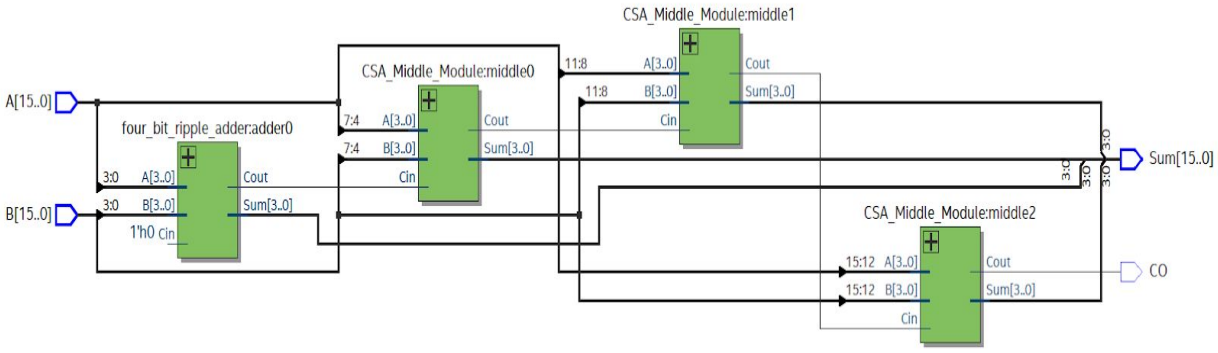
i. The carry select adder operates by having twice as many adders and modules as the lookahead and ripple adder. However, this adder computes the addition of the 16-bits the fastest. This is because each group of 4-bits, as we are still following the 4x4 hierarchical design as used in the lookahead adder, computes its sum and carry-out with both a carry-in bit of a “1” and “0” which serve as the feed in for a 2-to-1 MUX. Then, the actual carry-in bit operates as the select line for the MUX and selects the

correct sum and carry-out based on the last group of CSA results. Essentially, this method is the fastest as it computes all possible outputs and results beforehand, and then the circuit simply goes ahead and selects the correct outputs to display.

ii. The carry select adder computes multiple sums in parallel as it divides the two 16-bit data into subgroups of 4-bits. Afterwards, each group is then parallel computed with both a carry-bit of a "1" and "0" which then go into a 2-to-1 MUX. Finally, the actual carry-bits of each group serve as the select line for the MUX which simply combines all the bits of the results together. This operation is fast and efficient as all the possible outputs and combinations are computed all together, and the circuit simply has to select and combine the correct results together.

iii. Block Diagram of the whole CSA circuit





d. Area, Complexity, and Performance Tradeoffs between each Adder

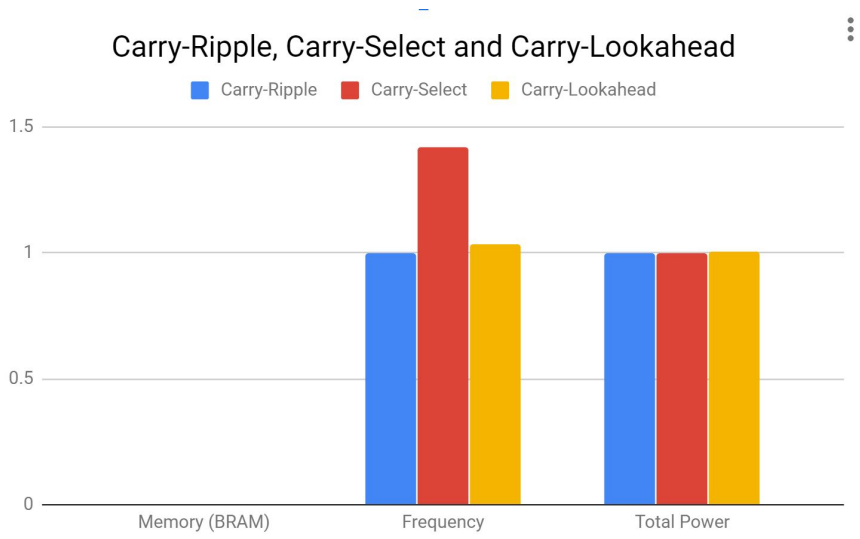
i. **Area** - Regarding all three adders, the carry select adder will have the greatest area out of all three. This is because the select adder has twice as many full-adder modules as the other adders due to it needing the compute all possible results all together. The lookahead adder is next when it comes to the most area taken up as it not only has all the adders as the ripple adder, but it also includes "middle modules" to take in and compute the propagation and generation bits alongside the carry-bit. The ripple adder is last in terms of area as it simply contains full-adders and nothing else.

ii. **Complexity** - In terms of complexity, the most complex design is the lookahead adder. This is because this design involves a lot of prediction and math to compute projected carry bits. It is essentially trying to predict the carry-in bit and the projected result before the actual carry-bit reaches the group. Afterwards follows the carry select adder. The carry select adder is not too complex as it simply contains twice as many adders as the ripple adder and a few 2-to-1 MUXes. The least complex design is the ripple adder is it consists of simply full-adders which are linked together in series.

iii. **Performance Tradeoff** - When it comes to performance tradeoffs, when we are adding small amounts of data, the carry ripple adder will operate slightly faster than the other two. This is because since we are dealing with less bits and the ripple adder has the least math to do (and less propagation delay), it can compute the sum for the bits the fastest. However, as we increase the amount of data and bits, the carry select adder will perform the fastest as it is computing all outputs in parallel then simply selecting the correct ones. Afterwards, the lookahead adder should be the second fastest as it will try to predict carry-out bits, and try to compute the correct output depending on its predictions. The ripple adder will be the slowest since it has to wait for each previous full-adder to execute and produce its carry-bit, so the next full-adder can operate.

e. **Performance and Design Analysis of the three Adders (Normalized)**

	Carry-Ripple	**Carry-Select**	**Carry-Lookahead**
Memory (BRAM)	0	0	0
Frequency	1	1.418	1.033
Total Power	1	1	1.002



Post-Lab Questions

1. Since we are using TTL logic we will never need to use look up tables or memory. However, there will still be an increase in combinational logic due to the circuit needing to account for the extra bits in the shift register and control unit. We would also double the flip flop usage for the 8-bit processor in TTL since the shift registers each go from 4 bits to 8 bits. In Quartus, the design of the processor is different in that all the combinational logic is being mimicked through the lookup table. We don't use any of the memory since all the data is stored in the flip flops of the FPGA. In terms of which design is better, we believe the FPGA design is better and faster since we never have to calculate any of the combinational logic during runtime, instead we just look up the correct output.
2. The 4x4 CSA hierarchy isn't necessarily ideal. For example, if we were given the propagation delay of the first two four-bit modules. We could then increase, from 4 bits, the number of bits the third module would have since we could calculate more bits for the same wait time of the second carry input. The number of extra bits the third module could have, depends on the delay of the first two, but this design would be faster than have two modules of just four bits.
3.

Resources	Ripple	CLA	CSA
LUT	114	132	123
DSP	0	0	0
Memory(BRAM)	0	0	0

Flip-Flop	105	105	105
Frequency	194.7 MHz	201.17 MHz	276.17 Mhz
Static Power	98.50 mW	98.5 mW	98.5 mW
Dynamic Power	0 mW	0 mW	0 mW
Total Power	139.11 mW	139.48 mW	139.11 mW

a. The results from quartus make sense. We can see that the carry select adder has a higher Fmax than CLA and the carry look ahead adder has a higher Fmax than the Ripple. We initially thought the Carry select would consume more power, however, we saw that the CLA consumed a little bit more power. Our reasoning is that the CLA obtains its speed through more combinational logic which in turn increases the number of lookup tables required, thus slightly increasing the power usage.

Conclusion:

Regarding this lab, we did not have many bugs or technical issues we ran into. Perhaps the most common issue we had was syntax errors in System Verilog. This is because we are still learning this new language and this lab was the first lab where we had to create all the files and modules from scratch. Another issue we ran into was that we found out that we had one less state in our state machine than what was needed. This was a logic issue and a design fault committed by us, and we were able to solve this by adding an extra state in the control unit file. Regarding ambiguity or unclearness, we do not believe that there was much that was unclear or incorrect for this lab. The lectures did a thorough job of explaining what the lab is about, and the lab manual is always a helpful, and detailed, guide to reference and learn from. Overall, this lab was a great way for us to grasp a firm understanding of System Verilog and creating modules and circuits in Quartus Prime from scratch.