ECE 385

Fall 2022

Lab 3

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

Zhuoyuan Li, Shihua Cheng

Tianhao Yu

Introduction

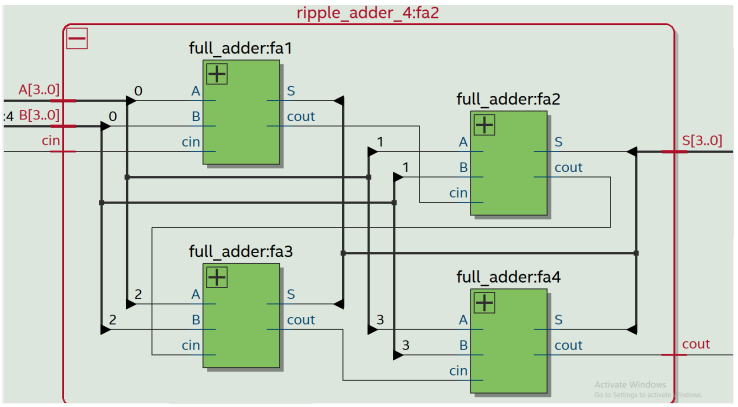
In this lab, we used SystemVerilog to build three 16-bit adders (ripple carry adder, carry lookahead adder, carry select adder) that can perform the addition of two 16-bit unsigned binary numbers and output a 16-bit unsigned result. SystemVerilog is used to implement an RTL design on an FPGA board.. These adders are binary adders whose input are two binary numbers of 16 bits, [15:0]A and [15:0]B, together with a carry-in C_{in}. They have two outputs, a 16 bit sum [15:0]S and a carry out C_{out}. Three adders have different performance due to differences in implementation. Quartus Prime is used for simulation and optimization in this lab.

Adders

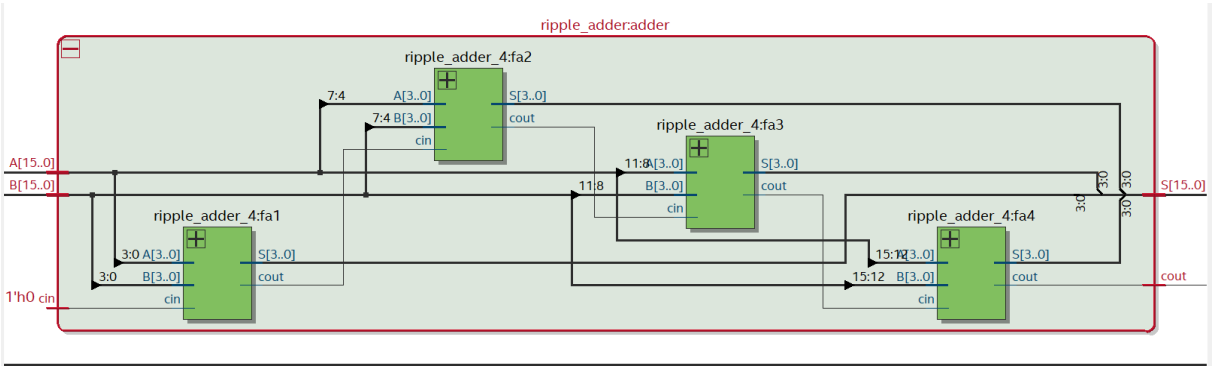
Ripple Carry Adder

The 16-bit ripple carry adder is simply a combination of four 4-bit binary adders, each of which consists of 4 single bit adders. Each single bit adder takes in 1 bit C_{in}, 1 bit from input A, and 1 bit from input B perform an addition to produce one 1-bit output S and 1-bit output C_{out}. The C_{out} of each 1-bit adder is then passed into the next 1-bit adder as C_{in} .

The 16-bit ripple carry adder is built from four 4-bit ripple carry adders. The RTL block diagram of a 4-bit ripple carry adder is shown below:



The RTL block diagram of the 16-bit ripple carry adder:



Carry Lookahead Adder

The carry lookahead adder consists of 4 4-bit carry lookahead adders. Within each 4-bit carry lookahead adder is the combination of four single bit carry lookahead adders. The carry lookahead adder involves the usage of a lookahead unit, the generating logic (G) and propagating (P) logic. The carry of a bit has a boolean expression expressed in terms of C_{in}, P and G. P indicates that the C_{in} is propagated to the next 1 bit adder, and it's calculated as $P(A, B) = A \oplus$ B. G indicates that the carry-out is generated as A and B are both 1. This module is much faster than carry ripple adder, because, instead of waiting for Cin to calculate Cout in ripple carry adder, carry lookahead adder enables C_0 ,C_1 ,C_2 ,C_3 to be calculated by P, G, and C_{in}. So carry out does not depend on carry in besides C_0, which depends on C_{in}. The way carry out are expressed by Cin, P and G are listed below:

$C0 = Cin$

$C1 = Cin \cdot P0 + G0$

$C2 = Cin \cdot P0 \cdot P1 + G0 \cdot P1 + G1$

$C3 = Cin \cdot P0 \cdot P1 \cdot P2 + G0 \cdot P1 \cdot P2 + G1 \cdot P2 + G2$

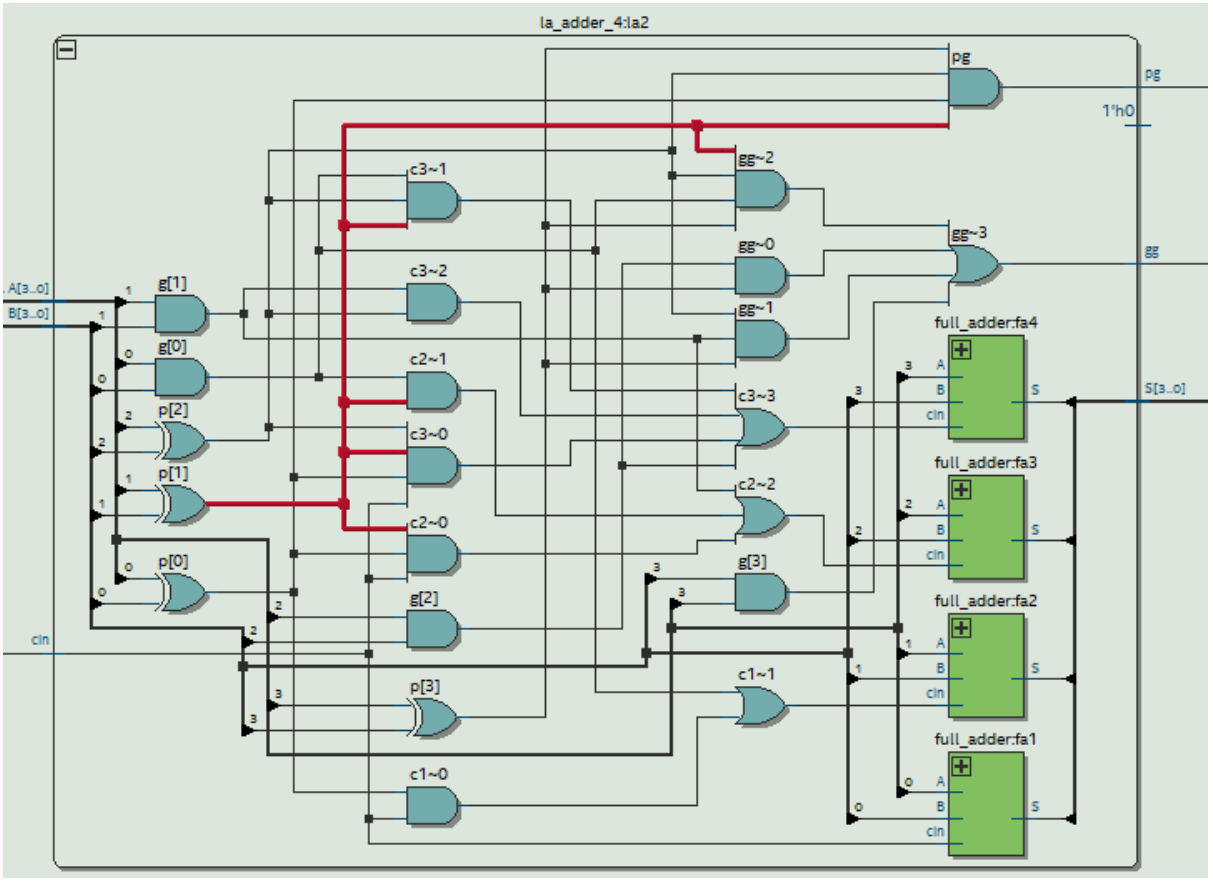
C0 depends on Cin of the 4 bit carry lookahead adder.

We construct a 4-bit carry lookahead adder module, and use 4 of them to construct a large 16 bit carry lookahead module, which is a 4 * 4 hierarchical adder. Each of the 4 4-bit carry lookahead adders will produce a P_G and a G_G that are used to calculate the Cin for the next 4-bit adder. $P_G = P0 \cdot P1 \cdot P2 \cdot P3$, if P_G is 1, this indicates that the carry is propagated throughout the 4-bit carry lookahead adder. $G_G = G3 + G2 \cdot P3 + G1 \cdot P3 \cdot P2 + G0 \cdot P3 \cdot P2 \cdot P1$, if G_G is 1, this indicates that carry is generated by A and B within the 4-bit carry lookahead adder and propagated to form C_{out}. Thus the carry in of the four 4-bit carry lookahead adder can be calculated by previous P_G, G_G, and C_0 using the following formula:
$C4 = GG0 + C0 \cdot PG0$

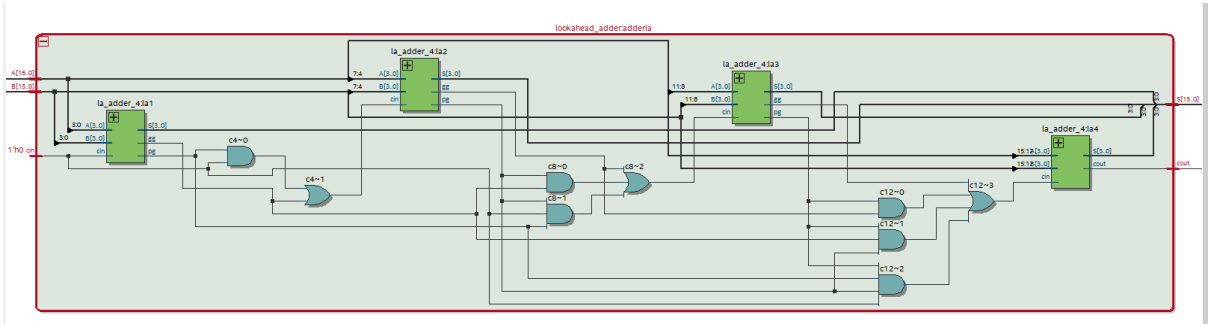
$C8 = GG4 + GG0 \cdot PG4 + C0 \cdot PG0 \cdot PG4$

$C12 = GG8 + GG4 \cdot PG8 + GG0 \cdot PG8 \cdot PG4 + C0 \cdot PG8 \cdot PG4 \cdot PG0$

The RTL block diagram of a 4-bit lookahead adder is shown below:



The RTL block diagram of a 16-bit lookahead adder is shown below:



Carry Select Adder

The carry select adder of 16 bit is made of three 4-bit carry select adders and one carry ripple adder. Within each of the carry select adders, two 4-bit carry ripple adders are used: one does the addition with assumption $C_{in} = 1$, while the other does the addition with $C_{in} = 0$. The sum result and the carry out is then selected by C_{in} . The sum is simply selected by a 2:1 mux, so when C_{in} is 1, the sum result with assumption C_{in} is 1 is outputted. The C_{out} for each 4-bit carry select adder is the result of the following expression:

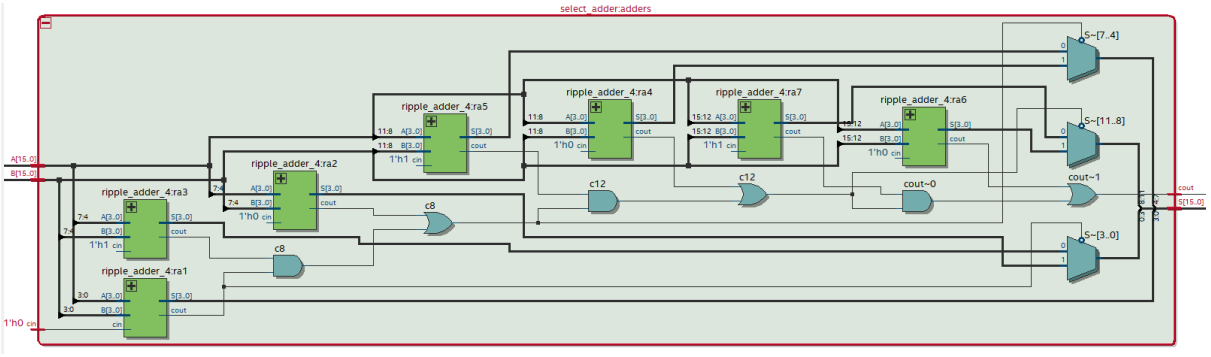
c8 = c1 | (c2 & c4); (c1 is the C output for carry ripple adder with Cin = 0, c2 is the C output for carry ripple adder with Cin = 1)

c12 = c5 | (c6 & c8); (c5 is the C output for carry ripple adder with Cin = 0, c6 is the C output for carry ripple adder with Cin = 1)

cout = c9 | (c10 & c12); (c9 is the C output for carry ripple adder with Cin = 0, c10 is the C output for carry ripple adder with Cin = 1）

If the adder assuming Cin = 1 produces a Cout of 1 when the actual Cin is indeed 1, then the Cout will be 1, thus an AND gate is used. If the adder assuming Cin = 0 produces a Cout of 1, then Cout will always be 1 no matter what Cin is, thus an OR gate is used. This works similar to a simplified 2:1 mux. This application is much faster in performance. All of its ripple carry adders are operating at the same time, causing 4 adder delay, and each mux causes 1 mux delay. Therefore, the total delay is 4 adder delay plus 3 mux delay.

The RTL block diagram of a 16-bit carry select adder is shown below:



Written Description of all modules

Module: reg_17.sv

Inputs: Clk, Reset, Load, [16:0] D

Outputs: [16:0] Data_Out

Description: This is a positive-edge triggered 17-bit register with asynchronous reset and synchronous load. When Load is high, it loads [16:0] D into the register on positive edge of Clk.

Purpose: This module is used to create the register that stores the result of 16-bit addition.

Module: router.sv

Inputs: R, [15:0] A_In, [16:0] B_In

Outputs: [16:0] Q_Out

Description: This is a 17-bit 2-to-1 parallel multiplexer. If R is 1'b0, then Q_Out will be the zero-extended value of A_In. If R is 1'b1, then Q_Out will be B_In.

Purpose: User can choose whether to load value from switches or value from the adder at each RunAccumulate button press in SystemVerilog top-level entity adder2.sv.

Module: control.sv

Inputs: Clk, Reset, Run

Outputs: Run_0

Description: This control unit is a three-state finite state machine. In the first state, Run_0 is set to low. The second state is reached from the first state only when Run is high. Only during the second state, the output Run_0 is set to high for the register to load once, then the machine goes into the third state where Run_0 is set to low again. The first state is then reached only if Run is low.

Purpose: The control unit ensures that during one press of the button, the load signal Run_0 is only high for one clock cycle. In other words, only one add operation is performed in one button press.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: HexDriver converts a 4-bit unsigned integer (0-F) to hexadecimal form that is mapped to the LED on the FPGA board.

Purpose: The content of the register can be visualized on the FPGA board.

Module: full_adder (in ripple_adder.sv)

Inputs: A, B, cin

Outputs: S, cout

Description: This is a 1-bit full adder. A, B are 1-bit operands and S is the 1-bit result, S = A+B. Cin is the input carry-in and cout is the carry-out. The addition is achieved with logic operations: S = A XOR B XOR cin, cout = (A&B) | (B&cin) | (A&cin).

Purpose: The 1-bit full adder is used to build 4-bit ripple adders and 4-bit lookahead adders, which are then chained together to build the three 16-bit adders.

Module: ripple_adder_4 (in ripple_adder.sv)

Inputs: [3:0] A, [3:0] B, cin

Outputs: [3:0] S, cout

Description: This is a 4-bit ripple adder built from four 1-bit full adders. Four full adder instances are created and chained in this module. In the chain, the cout from the previous full adder is connected to the cin of the current full adder. The 4-bit inputs are divided into four 1-bit inputs, each provided to a single full adder. The four 1-bit outputs of the full adders are concatenated into a 4-bit output, S. The cout of the last full adder is the cout of this module.

Purpose: Perform addition of two 4-bit inputs and output the 4-bit result and 1-bit carry-out.

Module: ripple_adder.sv

Inputs: [15:0] A, [15:0] B, cin

Outputs: [15:0] S, cout

Description: A, B are 16-bit inputs and cin is the carry-in provided to the adder. S is the 16-bit output of A+B, cout is the carry-out. The 16-bit ripple adder uses 4 instances of 4-bit ripple adders. The cout from the previous 4-bit ripple adder is connected to the cin of the current ripple adder. The 16-bit inputs A, B are divided into four 4-bit parts and provided to the four 4-bit adders. The 16-bit result, S, is the concatenation of the four 4-bit outputs from the 4-bit adders. The output cout comes from the cout from the last 4-bit ripple adder.

Purpose: Perform addition of two 16-bit inputs and output the 16-bit result and 1-bit carry-out.

Module: la_adder_4 (in lookahead_adder.sv)

Inputs: [3:0] A, [3:0] B, cin

Outputs: [3:0] S, cout, pg, gg

Description: This is a 4-bit lookahead adder. It creates four full_adder instances. Before the addition, it calculates the cin of each full adder (c0, c1, c2, c3) in advance, and then lets the

four full adders calculate in parallel. It does this by first calculating 4-bit numbers P = A XOR B and G = A AND B. Then, the carry-ins are calculated:

```
c0 = cin;
c1 = (g[0]) | (cin & p[0]);
c2 = (cin & p[0] & p[1]) | (g[0] & p[1]) | (g[1]);
c3 = (cin&p[0]&p[1]&p[2]) | (g[0]&p[1]&p[2]) | (g[1]&p[2]) | (g[2]);
```

These carry-ins are independent of each other so they can be provided to the four full adders simultaneously, making it faster than the ripple adder. The four 1-bit outputs from the full adders are concatenated to provide the output [3:0] S. The outputs pg and gg are then calculated:

```
pg = p[0]&p[1]&p[2]&p[3];
gg = g[3] | (g[2]&p[3]) | (g[1]&p[3]&p[2]) | (g[0]&p[3]&p[2]&p[1]);
```

Purpose: Perform 4-bit addition of two 4-bit inputs by first calculating carry-ins in parallel and let the full adders compute in parallel for faster calculation. Output a 4-bit result as well as two 1-bit logic, pg and gg. These outputs pg and gg are used later in the 16-bit lookahead adder to calculate carry-ins of each of the four 4-bit lookahead adders.

Module: lookahead_adder.sv

Inputs: [15:0] A, [15:0] B, cin

Outputs: [15:0] S, cout

Description: This module is a 16-bit lookahead adder that performs the addition of two 16-bit inputs with an input carry-in and outputs a 16-bit result as well as a 1-bit cout. It creates four la_adder_4 instances and divides the 16-bit inputs into four 4-bit parts and feeds them to the four 4-bit lookahead adders. This adder is partially serialized because the second 4-bit lookahead adder needs the outputs pg and gg from the first 4-bit lookahead adder to calculate its carry-in. In other words, four bits are calculated in parallel at a time. The carry-ins are calculated using the logic:

```
la_adder_4 la1(.A(A[3:0]),.B(B[3:0]),.cin(cin),.S(S[3:0]),.cout(c0),.pg(pg0),.gg(gg0));
assign c4 = gg0 | (cin & pg0);
la_adder_4 la2(.A(A[7:4]),.B(B[7:4]),.cin(c4), .S(S[7:4]),.cout(c1),.pg(pg4),.gg(gg4));
assign c8 = gg4 | (gg0&pg4) | (cin&pg0&pg4);
la_adder_4 la3(.A(A[11:8]),.B(B[11:8]),.cin(c8),.S(S[11:8]),.cout(c2),.pg(pg8),.gg(gg8));
assign c12 = gg8 | (gg4&pg8) | (gg0&pg8&pg4) | (cin&pg8&pg4&pg0);
la_adder_4 la4(.A(A[15:12]),.B(B[15:12]),.cin(c12),.S(S[15:12]),.cout(cout),.pg(pg12),.gg(gg12));
```

Here, c4 is the carry-in of the second 4-bit adder, c8 is third, and c12 is fourth which is the last adder. pg0, gg0 are outputs from the first 4-bit adder, pg4, gg4 are outputs from the second 4-bit adder, etc. The 4-bit output of each 4-bit adder is stored at the corresponding

location of the 16-bit output S. Finally, the output carry-out does not need pg and gg from the last adder because it can be directly obtained as cout from the last adder.

Purpose: Perform 16-bit addition of two 16-bit inputs and output the 16-bit result and the 1-bit carry out. Uses lookahead method to compute 4 bits parallel at a time to make the calculation faster.

Module: select_adder.sv

Inputs: [15:0] A, [15:0] B, cin

Outputs: [15:0] S, cout

Description: This is a 16-bit carry-select adder that performs the addition of two 16-bit unsigned inputs and outputs a 16-bit result. In this module, seven ripple_adder_4 instances are created. S[3:0] is calculated using one ripple_adder_4, and two ripple_adder_4 are used to calculate four bits from S[15:4]: one ripple_adder_4 assumes a carry-in of 0, and the other assumes a carry-in of 1. The outputs of the two ripple adders are connected to a MUX, with the select of the MUX being the carry-out from the previous 4-bit unit. The select of each 4-bit MUX as well as the output cout is calculated as {(Cout of previous adder with cin=0) OR (Cout of previous adder with cin=1 AND previous select)}. All the four 4-bit ripple adders compute in parallel, however higher bits experience MUX delays. Each of the three MUXes is connected to 4 bits of S[15:4] and the output of the first 4-bit ripple adder is connected to S[3:0], providing the 16-bit output S.

Purpose: Perform the addition of two 16-bit inputs and output a 16-bit result and a 1-bit carry-out. Divides 16-bit inputs into four 4-bit parts and computes all four 4-bit parts in parallel for faster speed.

Module: adder2.sv

Inputs: Clk, Reset_Clear, Run_Accumulate, [9:0] SW,

Outputs: [9:0] LED, [6:0] (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5)

Description: This is the top-level entity that creates instances of other modules and is compiled and loaded to the FPGA board. Reset_Clear and Run_Accumulate are mapped to the two buttons on the FPGA board: Pressing Reset_Clear will clear the register, and pressing Run_Accumulate will add the value from the switches to the register. It maps the 16-bit value of the register to four LEDs on the board using HexDriver, and maps the lower 8 bits of the switches to two LEDs for visualization and debugging. The choice of which adder to use is made in this module.

Purpose: Link all modules together in one top-level entity, select the type of adder, and make the program compilable to be loaded to the FPGA board.

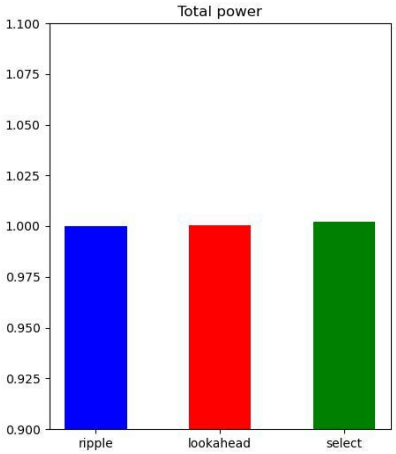
Tradeoffs between the adders

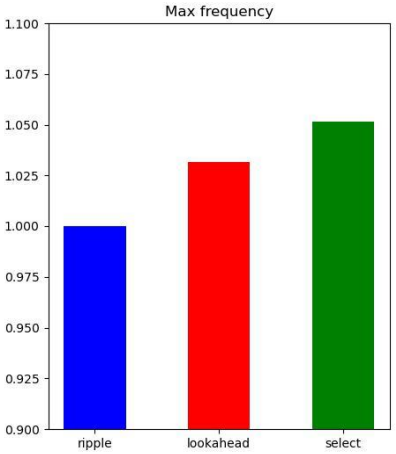
	Ripple carry adder	Lookahead adder	Carry select adder
Area	Smallest area. Chains 16 1-bit full adders together, simple logic inside each full adder, no logic between full adders	Larger area. Also uses 16 1-bit full adders, but more logic elements are used due to complex logic calculation of PG, GG, and the carry-ins	Largest area. Although fewer logic elements than lookahead adder, seven 4-bit ripple adders as well as three muxes are used.
Complexity	Least complex. Connects 16 1-bit full adders in series with no extra logic in between.	Most complex. Involves many logic operations, and these operations become even more complex in higher bits	More complex. A simple 2-gate logic is needed to calculate the carry-in bit of each 4-bit dual adder unit. Also, 2:1 MUXes are needed.
Performance	Worst. The 16 1-bit full adders compute in series, which means each adder needs the carry-out of the previous adder to compute.	Better. Each 4-bit adder can compute the 4-bit result in parallel, but it is still serialized because each 4-bit adder needs the output from the previous 4-bit adder to compute.	Best. All 4-bit adders can compute in parallel. The only delay is from the logic gates and MUXes that choose the carry-ins for each 4-bit adder, but these delays are much shorter than adder delays.

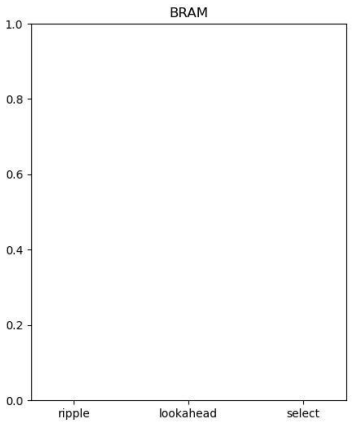
Performance of each adder

We wrote a .sdc file to constrain the clock at 50MHz and obtained the max frequency from “Slow 1200mV 85C” condition. Total power is obtained from Power Analyzer, and BRAM is obtained from resource usage summary in the compilation report. Following is a table and a graph showing these data.

	Ripper Carry Adder	Lookahead Adder	Carry Select Adder
BRAM	0	0	0
Max Frequency	64.47MHz	66.5MHz	67.8MHz
Total Power	105.11mW	105.17mW	105.34mW



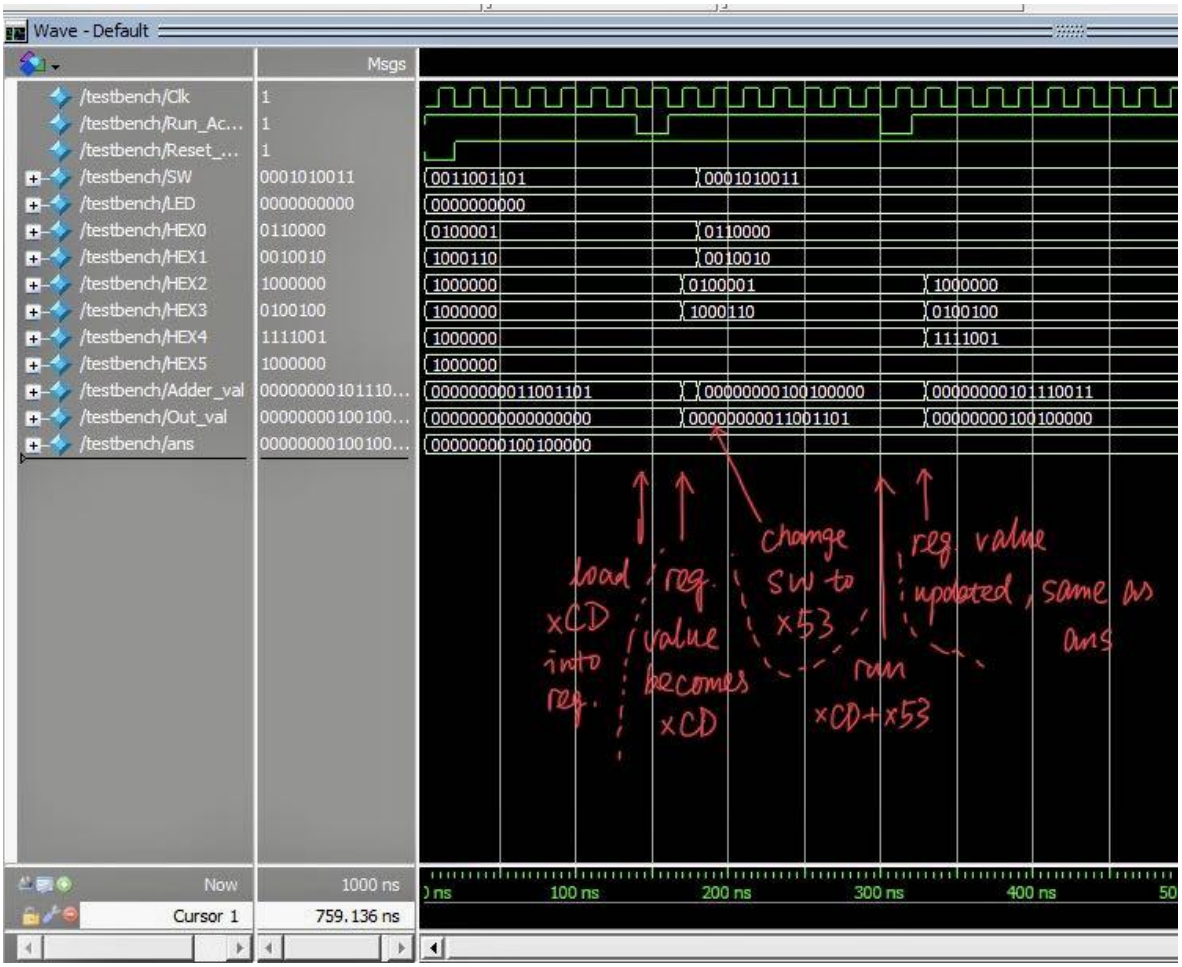




According to the data, all three adders do not use block RAM, the total power of the three adders are almost the same with Carry Select Adder slightly larger than Lookahead Adder and Lookahead Adder slightly larger than Ripple Adder. The max frequency of carry select adder is noticeably higher than Lookahead Adder, and both are higher than Ripple Adder.

Simulation Trace

Following is the simulation trace of all three adders performing the operation xCD + x53.



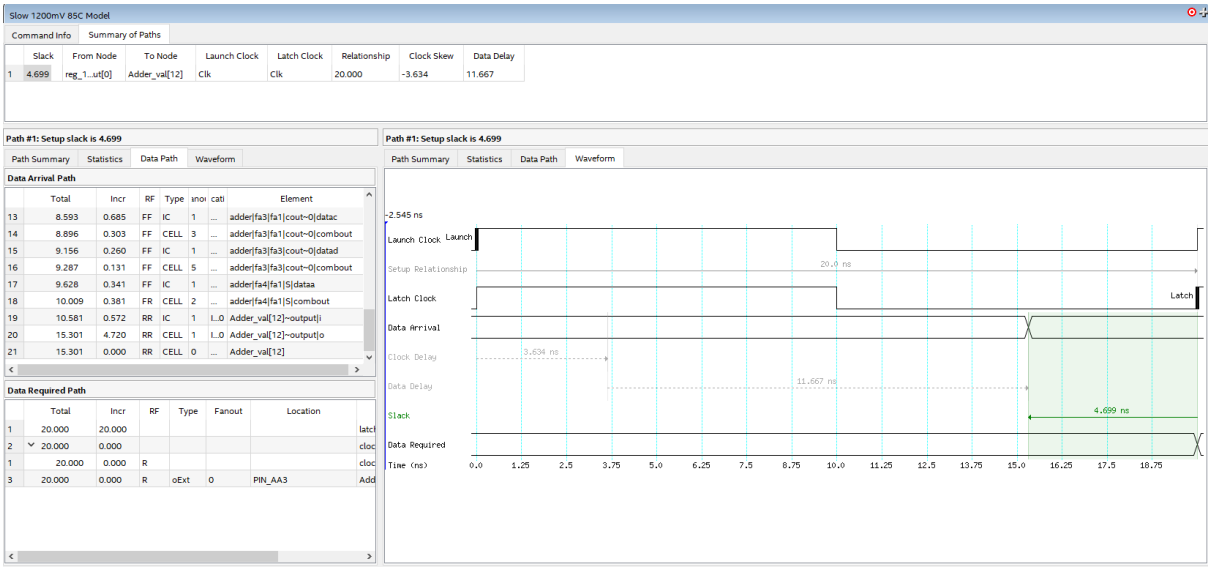
Out_val is the register value, Adder_val is the output of the adder. Ans stores the correct result of xCD + x53. As shown in the trace, the final register value is the same as the correct value. Since ModelSim does not simulate gate delays, the traces of all three adders are the same.

Critical Path Analysis

For this part, we followed the critical path analysis tutorial, and we chose to use the second method which is to add false paths from SW* ports/to HEX* ports. We did this by going to the Timing Analyzer, then clicking Constraints - Set False Path. Then, we set “From” ports to all SW ports and left “To” ports empty, and clicked “Run”. We did the same process again by setting “To” ports to all HEX ports and leaving “From” ports empty.

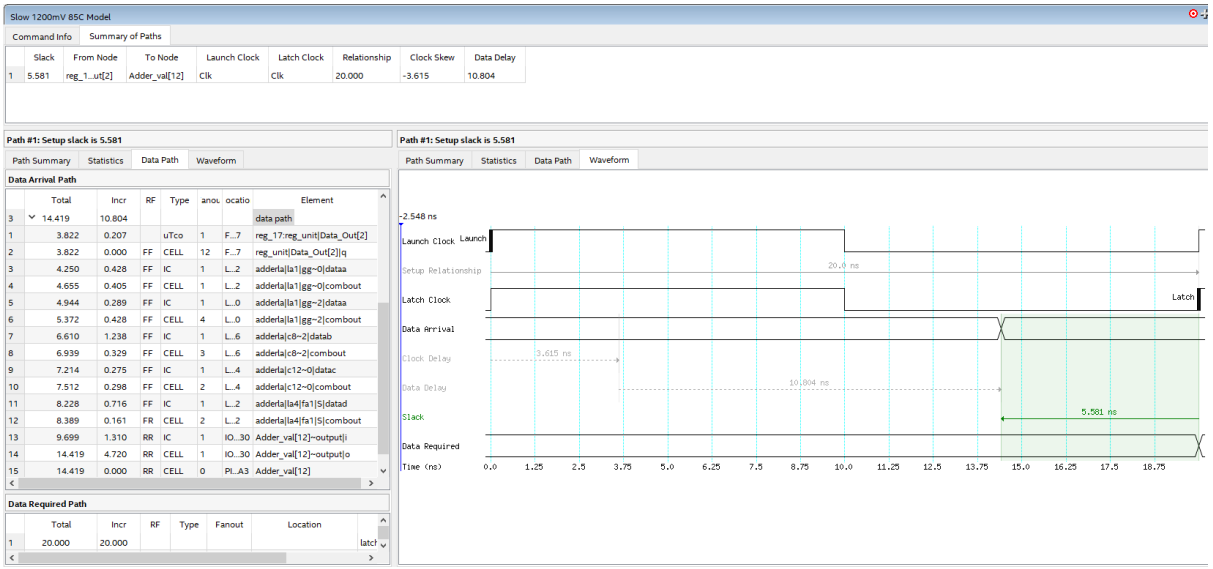
Since the false paths have been set, we set the “Report Number of Paths” in “Report Timing” to 1. This will analyze the slowest critical path **not including** paths that start from or end at any I/O port. Therefore, the result will be more accurate since I/O ports are limited to around 150MHz and will affect performance.

The critical path analysis of the Ripple Carry Adder is shown below:



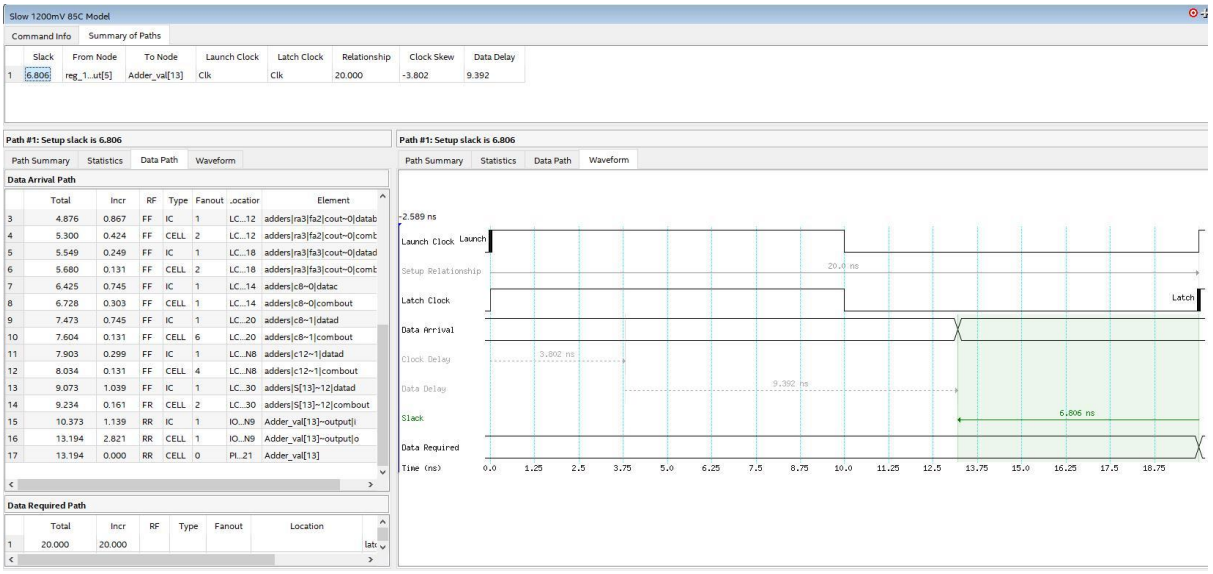
As shown in the graph, the total critical data delay (clock delay + data delay) for the Ripple Carry Adder is 15.301ns, with a clock delay of 3.634ns and a data delay of 11.667ns. This is indicated either in the “Total” column under “Data Arrival Path” on the left, or in the waveform on the right. The corresponding max frequency is 1/15.301ns = 65.36MHz.

The critical path analysis of the Lookahead Adder is shown below:



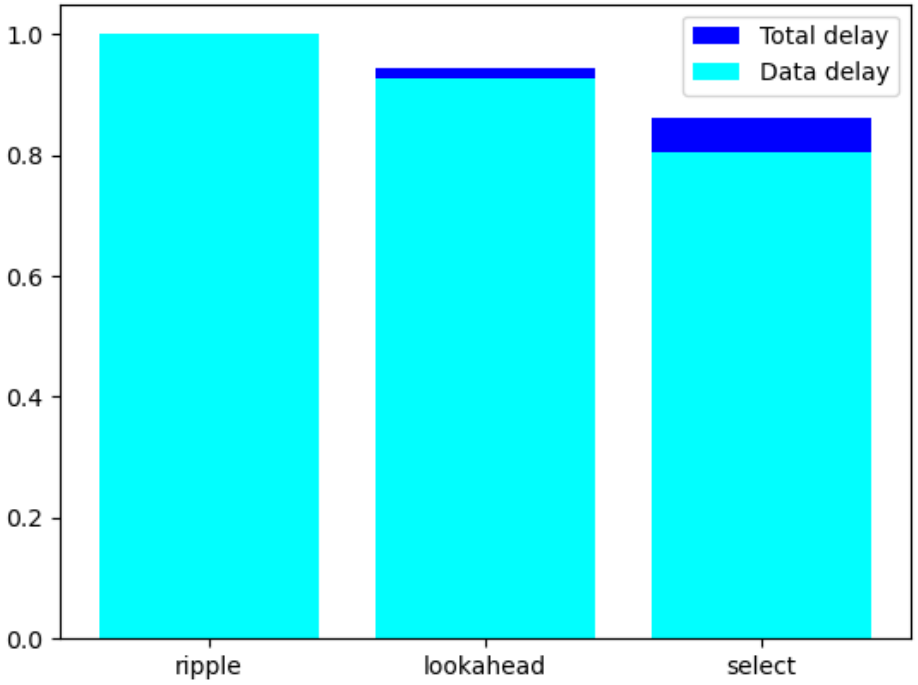
The total critical data delay (clock delay + data delay) for the Lookahead Adder is 14.419ns, with a clock delay of 3.615ns and a data delay of 10.804ns. The corresponding max frequency is 1/14.419ns = 69.35MHz.

The critical path analysis of the Carry Select Adder is shown below:



The total critical data delay (clock delay + data delay) for the Carry Select Adder is 13.194ns, with a clock delay of 3.802ns and a data delay of 9.392ns. The corresponding max frequency is 1/13.194ns = 75.79MHz.

It can be seen that the total delay of the Carry Select Adder is the smallest, Lookahead adder is the second smallest, and the Ripple Adder has the longest delay. On the other hand, all three adders have similar clock delays. Therefore, the difference in the actual data delay is even more significant than total delay. A graph is shown below to visualize this. Note that the delays are normalized so that the ripple adder is set to 1.



Through this critical path analysis, we gained a better understanding of how the performances of the three adders differ from each other, and quantified the results.

Answers to the post-lab questions

For the Carry Select Adder, the ideal situation is to gradually increase group size from lower bits to higher bits, so that the delay of the current group is equal to the delay of the previous group plus one MUX delay. Therefore, the 4x4 configuration is not the optimal configuration. If the MUX delay is large, for example equal to an Adder delay, then the optimal configuration would be 2-2-3-4-5, so that the total delay is 2 Adder delays + 4 MUX delays, which is faster than 4 Adder delays + 3 MUX delays of the 4x4 configuration. If the MUX delay is very small, then we can use smaller group size and more MUXes (for example, 2-bit adders with 7 MUXes) to achieve faster speed. We could do an experiment to obtain the MUX delay information: first use a 4x4 configuration (4 adder delays + 3 mux delays) and record the total delay, then use a 8x2 configuration (2 adder delays + 7 mux delays) and record the total delay. Since we have 2 equations and 2 unknowns, the adder delay and the mux delay can be calculated, then we could use the information to calculate an optimal configuration.

The data of the three adders are shown in the table below:

	Ripple Carry Adder	Lookahead Adder	Carry Select Adder
LUT	78	86	82
DSP	0	0	0
BRAM	0	0	0
Flip-Flop	20	20	20
Max Frequency	64.47MHz	66.5MHz	67.8MHz
Static Power	89.97mW	89.97mW	89.97mW
Dynamic Power	1.39mW	1.60mW	1.58mW
Total Power	105.11mW	105.17mW	105.34mW

The lookahead adder uses the most LUTs, which makes sense because it involves many different logic operations. The carry select adder involves simpler logic that is only applied to the select signals, so it uses fewer LUTs. The ripple carry adder does not have additional logic besides the 1-bit full adder, so it uses the least number of LUTs. Note that the number of logic gates in the CSA should be largest since it has 7 4-bit ripple adders, but in FPGA, LUT is used instead of logic gates and the 4-bit adders share the same LUT, so the number makes sense. All three adders do not use on-chip memory so DSP and BRAM are 0. All three adders have the same number of flip-flops. The max frequencies are Carry Select Adder >

Lookahead Adder > Ripple Carry Adder. This is expected because Carry Select Adder uses more area in exchange for highest speed, and lookahead adder uses more logic operations for slightly higher speed. The total power of all three adders are very close. Carry Select Adder consumes the most power, which is expected because it has more adders thus more gates. Lookahead adder is second, which also makes sense because there are more logic gates to compute carry-in, the total number of gates should be smaller than CSA but larger than Ripple Adder.

Conclusion

During Lab 3, we did not encounter any technical issues or bugs. Both the ripple adder and the select adder worked as expected as soon as the program was compiled and loaded to the FPGA board. However, the lookahead adder initially did not work as expected. We used SignalTap to locate the problem: the carry-out of the first bit was always incorrect. It took us an hour to finally resolve the problem. It was the misspell of a logic variable “c1”. We misspelled it as “C1” inside the adder instance. We learned that SystemVerilog does not give a compilation error when an undeclared variable is used, unlike some languages such as C.

We think the structure of Lab 3 is organized. Both the provided code and the lab manual are great and easy to work with, these do not need any amendments. A possible improvement is to add a brief introduction of the modules provided and specify which files to work on. (If it is intended for us to figure out the hierarchy and the use of each file, then no improvements are needed)

In conclusion, we used SystemVerilog to build three different adders in this lab: ripple carry adder, lookahead adder and carry select adder. We analyzed the advantages and disadvantages of each adder in terms of area, frequency, and power consumption. Finally, we were able to correctly perform 16-bit addition with all the three adders on the FPGA board.