ECE 385

Spring 22

Experiment #3

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

Siddarth Iyer, Geitanksha Tandon
TA: Ruihao Yao (RY)

1. Introduction:

- *High-level function performed by the three adders:*

In this experiment, we designed 3 different 16-bit binary adders using SystemVerilog. The adders implemented were Ripple Adder, Carry Lookahead Adder, and Carry Select Adder. Each adder took in two 16-bit values and a Carry in as inputs, and outputted a 16-bit sum and a Carry Out. We designed 16-bit adders with a 10-bit input set by the switches. The sum is stored in a 17-bit register(to account for overflow) and displayed on 4 hex displays. When the run_accumalate button is pressed, the 10-bit input from the switches is added to the current sum, with each button press accumulating the switch value to the previous sum. The sum can be cleared by pressing the Reset_Clear button. In the case overflow occurs, LED[9]

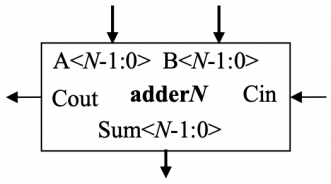


Figure 1: *N*-bit Binary Adder Block Diagram

---

2. The Three Varieties of Adders: Ripple Carry, Carry Lookahead, Carry Select

A. Ripple Carry Adder (RCA)

- *Architecture of the Adder:*

The Ripple Carry Adder is the simplest adder, constructed with 16 Full Adders.
A full adder (*Figure 2*) is a 1-bit adder, which takes in two 1-bit inputs and a carry-in, adds them together, and produces a 1-bit sum and carry out as the output. The 16 full adders are serially connected.
If we consider the full adder corresponding to the LSB, it takes the LSB of inputs A and B, the carry-in, and gives an output of S0 (LSB of sum). The carry out of the first Full Adder is connected to the carry-in of the second Full Adder, and similarly each intermediate Full Adder N has its Carry out connected to the N+1 full adder's carry in. The 16th Full Adder outputs the MSB of the sum and outputs the carry out. In this way, the adder ripples through the 16 full adders, with each subsequent Full Adder dependent on the current Full Adder's outputs.

In our implementation, we first designed a single full adder by deriving the necessary boolean circuitry using Kmaps (C_out = AB + BC_in + AC_in; Sum = A⊕B⊕cin)

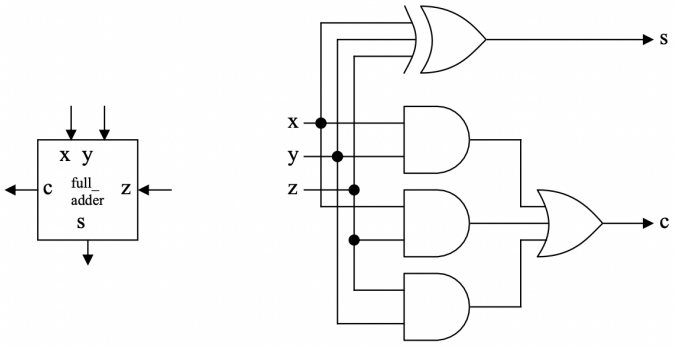


Figure 2: Full-Adder Block Diagram

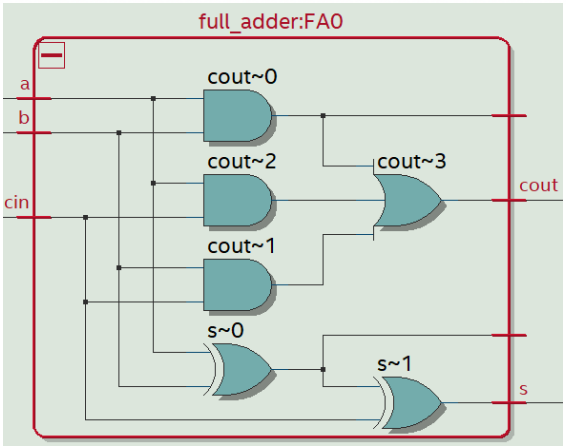


Figure: Full adder implementation in our structure.

We then created a 4-bit Carry Ripple adder using four Full Adders placed in series:

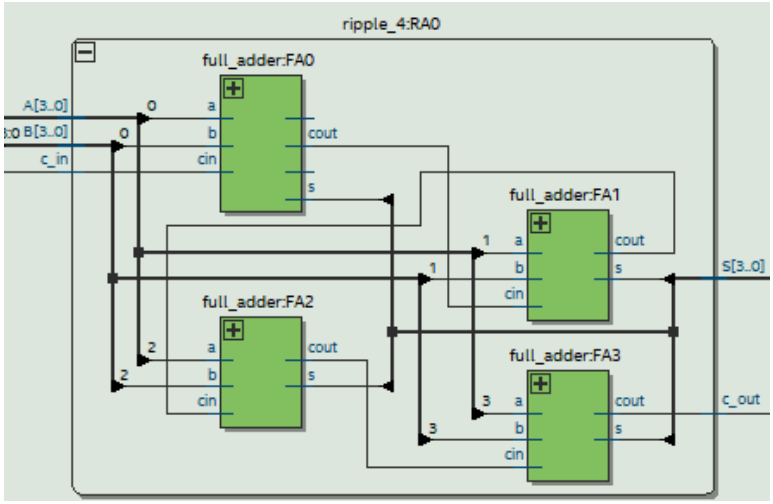


Figure: 4-Bit Carry Ripple Adder implementation in our structure.

- *Block Diagram of the Carry Ripple Adder:*

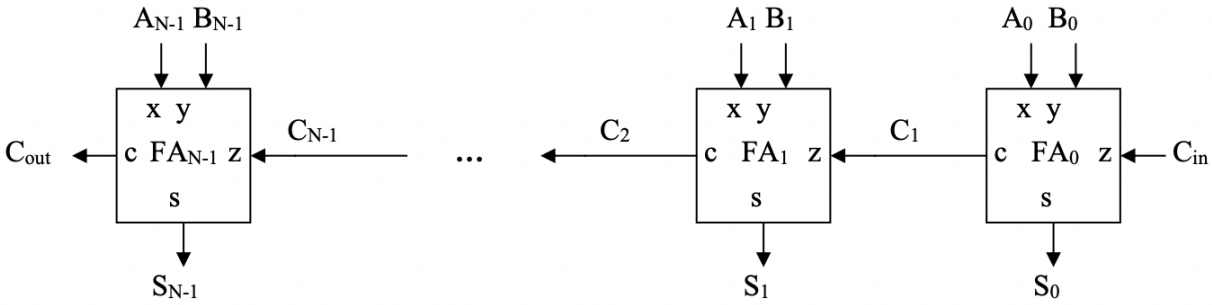


Figure 3: *N*-bit Carry-Ripple Adder Block Diagram

We then created the 16-Bit Adder by connecting four 4-bit ripple adders in series, it forms a structure to that shown in Figure 3.

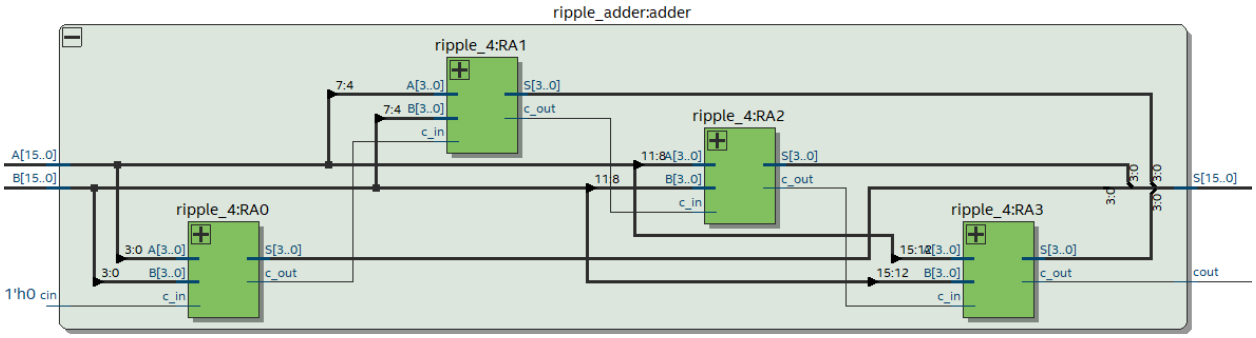


Figure: 16-Bit Full adder implementation in our structure.

B. Carry Lookahead Adder (CLA)

- *Architecture of the Adder:*

The Carry Lookahead Adder is a relatively time-efficient adder, which makes use of Generating and Propagating logic to anticipate and determine the carry-in bits of each Full Adder quicker. Our implementation is a 4x4 hierarchical adder making use of four 4-bit CLAs.

- *P and G Logic implemented in the Adder:*

The carry-out of the Nth adder, which serves as the carry-in bit for the subsequent N+1th full adder is determined by the 1-bit inputs A and B and the Carry-in bit.
The Carry-out is guaranteed to be 1 if A and B are both 1, or if either A or B are 1 and the Carry-in is 1. Based on this, each Full Adder outputs a Propagate Signal. A Carry-out is guaranteed when A and B are one, which then creates a Generate signal. Using this information, we can determine that:

- Propagate Signal (P) = A⊕B
- Generate Signal (G) = A.B

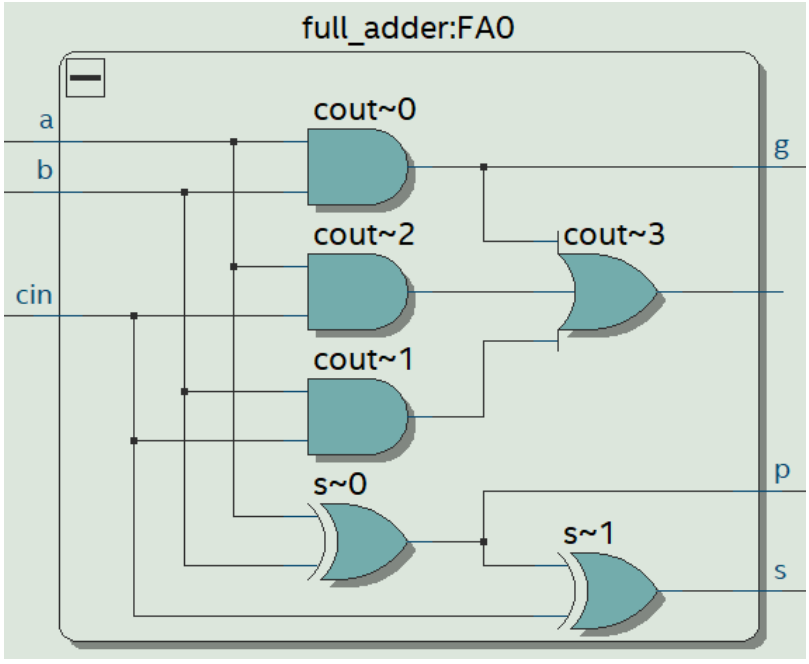


Figure: Full adder implementation with P, G bits.

Hence, the carryout now depends on P, G, C_in: C_out = G + P(C_in). C_in depends on the previous Full Adder, which in turn depends on the one before itself, up until the original C_in. Hence each C_out can be computed using the current P, G, the previous bits' P and G and C_in.
Hence, the C0 bits for the first few carries are written as below (taken from Experiment PDF):

$$C_0 = C_{in}$$
$$C_1 = C_{in} \cdot P_0 + G_0$$
$$C_2 = C_{in} \cdot P_0 \cdot P_1 + G_0 \cdot P_1 + G_1$$
$$C_3 = C_{in} \cdot P_0 \cdot P_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + G_1 \cdot P_2 + G_2$$
$$\dots$$

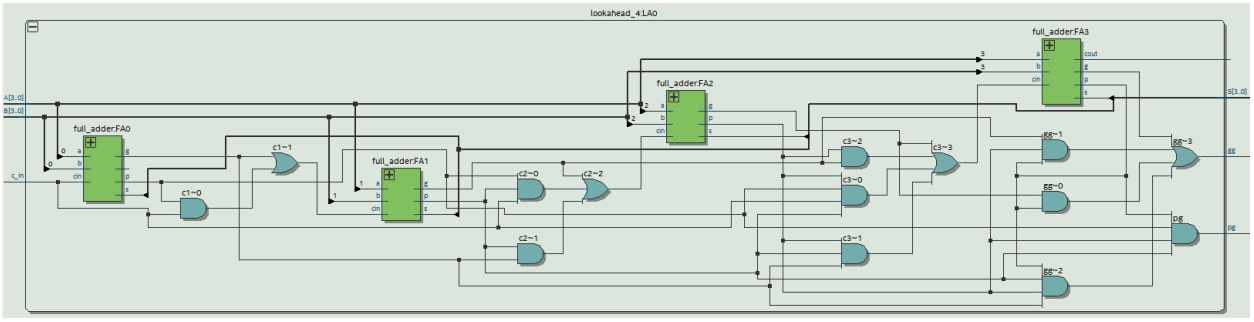


Figure: 4-bit version of the Carry Lookahead Adder

- *Hierarchical version of the 16-bit adder, using the 4x4 structure*

The 4x4 adder was designed using four 4-bit Carry Lookahead adders. Each Full Adder in the 4 bit adder generates its corresponding P and G, and the carry out of each FA is then determined by the 4 P and G signals, and Carry in as below.

Since implementing a 16-bit adder in this manner would require extremely large gates, we create a hierarchical design. Each 4-bit carry look-ahead adder, generates 2 additional signals Group Propagate and Group Generate, with expressions as shown below:

$$P_G = P_0 \cdot P_1 \cdot P_2 \cdot P_3$$
$$G_G = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1$$

We continue by assigning the C_in for the other 4-bit adders using PG and GG as shown below:

$$C_4 = G_{G0} + C_0 \cdot P_{G0}$$
$$C_8 = G_{G4} + G_{G0} \cdot P_{G4} + C_0 \cdot P_{G0} \cdot P_{G4}$$
$$C_{12} = G_{G8} + G_{G4} \cdot P_{G8} + G_{G0} \cdot P_{G8} \cdot P_{G4} + C_0 \cdot P_{G8} \cdot P_{G4} \cdot P_{G0}$$

This follows similar logic, as how the 4-bit adder determines C_in using individual P and G.

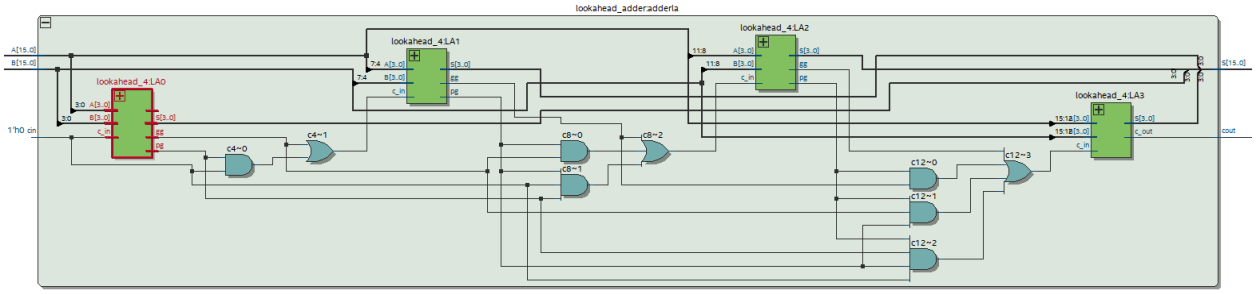


Figure: 4x4 Hierarchical Version of the Carry Lookahead Adder

---

- *Block Diagram of the Carry Ripple Adder:*

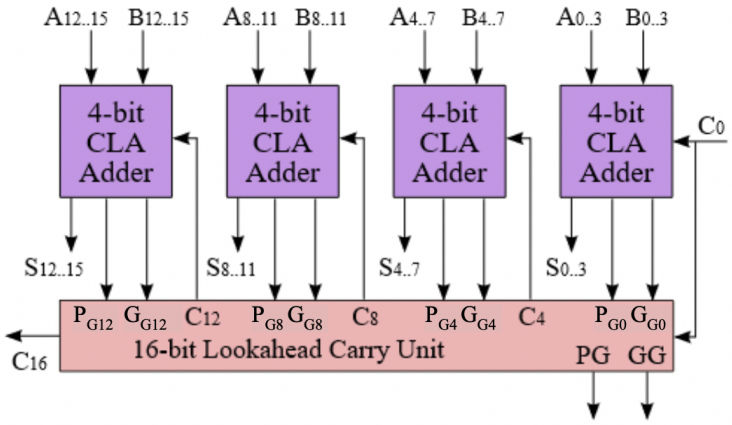


Figure 5: A 4x4-bit Hierarchical Carry-Lookahead Adder Block Diagram

---

C. Carry Select Adder (CSA)

- *Architecture of the Adder:*

A carry select adder makes use of 2 full adders or a group of 2 ripple adders and a multiplexer. The first full adder pre-computes the sum using the carry in as 1, while the other assumes the carry in as 0. When the actual carry in arrives, the sums have already been computed and the mux is used to determine which sum to output as the final sum. In our implementation, we design a hierarchical adder consisting of 7 carry ripple adders and divide the 16 bits into groups of 4. The first unit corresponding to the lowest bits requires only 1 CRA since the carry in is known. Each other unit contains a multiplexer 2 CRAs which compute the sum using 0 and 1 as the carry in. The carry-in of the previous unit, when computed, is used as the select signal to determine which Full Adders sum to output.

- *How does the CSA speculatively compute multiple sums in parallel and rapidly choose the correct one later?*

Our CSA uses seven 4-bit CRAs. Since each CRA is provided with a Carry In, the outputs can immediately be calculated. Hence, in the time it takes for the first CRA to compute the outputs, the other 6 CRAs also compute the outputs parallely. The adders also, in parallel, generate a carryout which is used to determine the carry in for the next unit using combinational logic. With the sum pre-computed, as soon as the Carry In signal is available the corresponding sum is selected by the MUX.

- *Block Diagram of the CSA:*

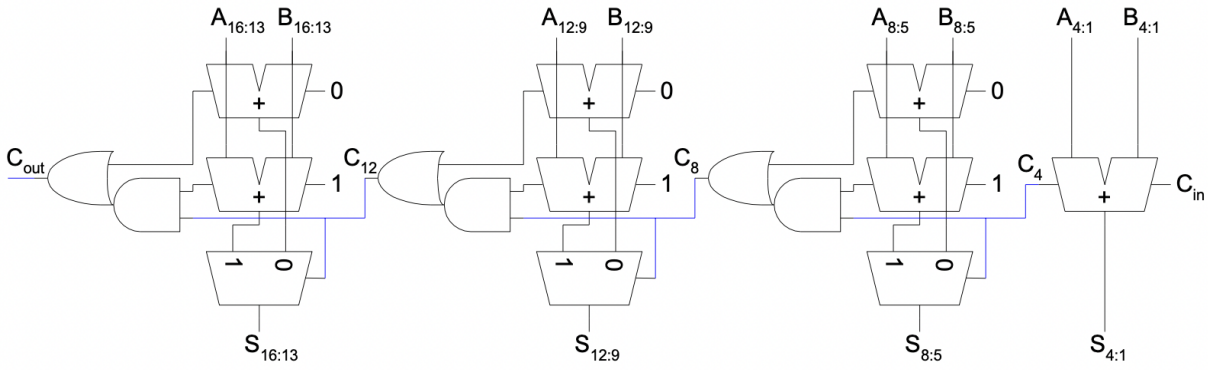


Figure 5: 16-bit Carry-Select Adder Block Diagram

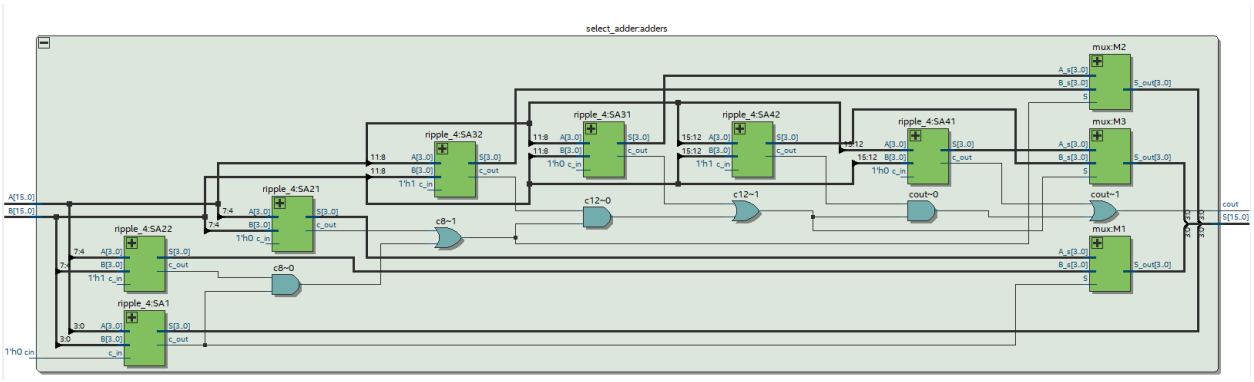


Figure: The 4x4 Hierarchical 16-Bit version of the Carry Select adder.

Module Descriptions:

Module: full_adder.sv

Inputs: a, b, C_in

Outputs: s, C_out, p, g

Description: This is a full adder that takes in a, b, C_in as inputs, sums them together using combinational logic, and outputs the sum s and C_out. It additionally generates outputs p and g used in the carry-lookahead adder.
Purpose: This module is used to design a Full Adder which is instantiated to create a 4-bit carry ripple adder and a 4-bit carry-lookahead adder.

Module: ripple_four.sv

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

Outputs: [3:0] S, c_out

Description:This module instantiates 4 full adders to create a 4-bit ripple adder. Each full adder N takes in bit A[N], B[N] and outputs sum S[N] depending on the carry in. Input c_in is passed as the carry in to the Full Adder corresponding to the LSB, and output c_out is set to the carry out of the Full Adder corresponding to the MSB.

Purpose: This module is used to design a 4-bit carry ripple adder which is instantiated to create a 16-bit Carry ripple adder and a 16-bit carry select adder.

Module: lookahead_4.sv

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

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

Description: This module instantiates 4 full adders to create a 4-bit carry select adder. Each full adder N takes in bit A[N], B[N] and outputs sum S[N] depending on the carry in. Input c_in is set as the carry in for the full adder corresponding to the LSB. The remaining carry-ins are determined by the p and g signal from the full adders. Additionally, pg and gg is calculated and passed as outputs.

Purpose: This module describes a 4-bit carry-lookahead adder, which is instantiated to create a 16-bit Carry ripple adder.

Module: ripple_adder

Inputs: [15:0] A, [15:0] B, C_in
Outputs: [15:0] S, C_out

Description: This module instantiates four 4-bit ripple adders to create a 16-bit ripple adder. Each ripple adder receives 4-bit inputs A and B, the carry-in from the previous adder or C_in for the first adder, and produces a 4-bit Sum. The four 4-bit S outputs from each adder serves as the final 16 bit sum. Additionally, the carry out of the last adder is set to C_out

Purpose: This module describes a 16-bit Carry Ripple Adder that takes in two 16 bit inputs, a carry in, and outputs a 16-bit sum and a carry out.

Module: mux.sv

Inputs: [3:0] A_s, [3:0] B_s, S
Outputs: [3:0] S_out

Description: This module takes in two 4-bit values A_s and B_s, and a select bit S. When S is 0, output S_out is set to A_s, and when S is 1, output S_out is set to B_s

Purpose: This module creates a quad 2-to-1 MUX used in the carry select adder to choose the correct sum.

Module: select_adder.sv

Inputs: [15:0] A, [15:0] B, C_in
Outputs: [15:0] S, C_out

Description: This module instantiates seven 4-bit ripple adders to create a 16-bit carry select adder. The first ripple adder takes C_in as the input. The remaining 6 adders are split into 3 units, with each unit having one full adder with carryin set to 0 and the other set to 1. Each unit receives 4-bit inputs A and B which go into the 2 full adders and the two sums are inputted into a mux. The mux uses the carry in from the previous unit to determine which sum will be the final output.

Purpose: This module describes a 16-bit Carry Select Adder that takes in two 16 bit inputs, a carry in, and outputs a 16-bit sum and a carryout.

Module: lookahead_adder.sv

Inputs: [15:0] A, [15:0] B, C_in
Outputs: [15:0] S, C_out

Description: This module instantiates four 4-bit Carry Lookahead adders to create a 16-bit carry lookahead adder. Each CLA receives 4 bits of A and B and produces a 4 bit sum S as the output. The carry in for each CLA is determined by the pg, gg, and C_in signals. The four 4-bit S outputs from each adder serves as the final 16 bit sum. Additionally, the carry out of the last adder is set to C_out

Purpose: This module describes a 16-bit Carry Lookahead Adder that takes in two 16 bit inputs, a carry in, and outputs a 16-bit sum and a carryout.

Module: reg_17.sv

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

Description: This is a positive-edged 17-bit parallel load register. When Reset is high, the register asynchronously resets and sets Data_Out to 0. When Load is high, D loads into Data_Out.

Purpose: This module is used to create a 17 bit register to store the 16-bit sum and an additional bit to detect overflow

---

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This makes use of a unique case switch to set the 7 bits of Out0 to the corresponding 4 bit In0, to display the 4 bit hex number on the 7-segment LED.

Purpose: This module maps a hex number to its corresponding value so it can be displayed on a 7-segment LED.

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. The control unit, 3 adders, register, router, Hex_driver are instantiated and the necessary I/O connections made to attach all the modules together

Purpose: Top-level entity that describes the overall hardware. Contains all the inputs, outputs, and modules for the subunits of the circuit. This is where the user selects which adder to implement.

Module: router.sv

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

Description: This module takes in a signal R. When R is 0, Q_Out[15:0] is set to A_In and Q_Out[16] is set to 0. When R is 1, Q_Out is set to B_In.

Purpose: This module is used to decide where the value set by the switches is loaded.

Module: control.sv

Inputs: Clk, Reset, Run

Output: Run_0

Description: Describes an FSM with 3 states. All states transition back to A state when reset is high. When run is pressed the FSM transitions to state B, and then to state C and stays there till run is released after which it transitions back to A. Run_0 is set to high in state B and 0 otherwise.

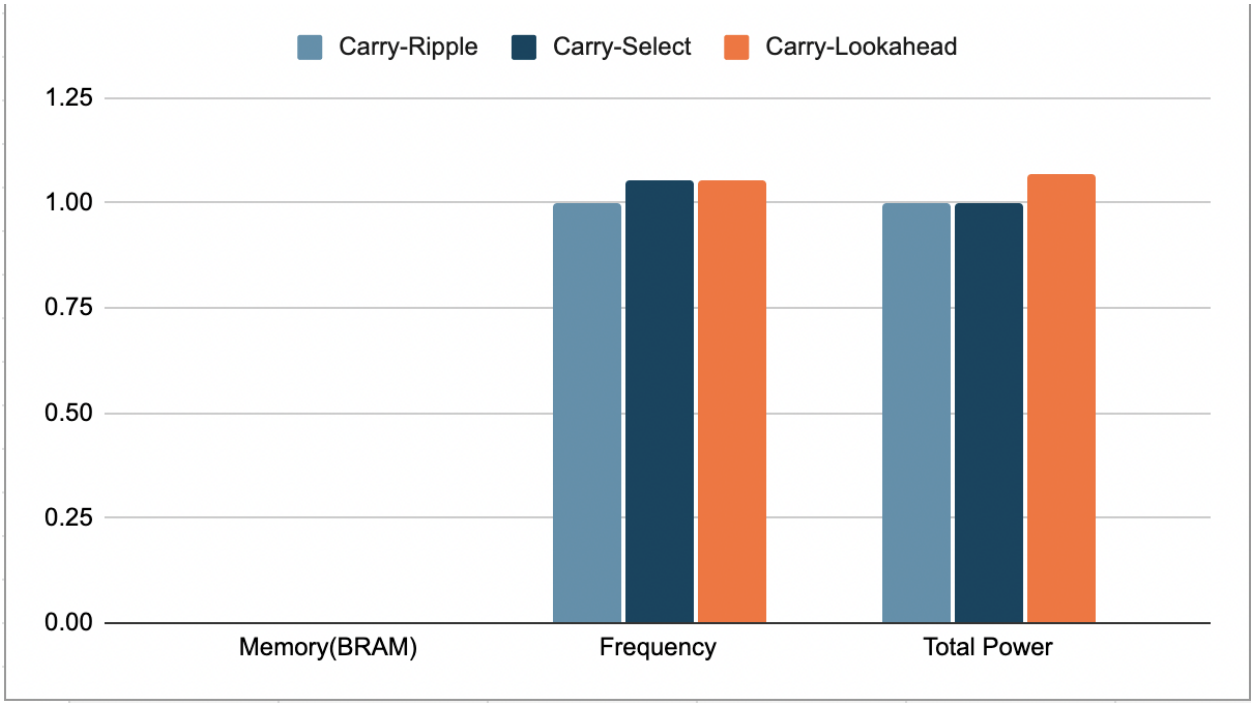
Purpose: Describes an FSM used to implement addition.

- *Area, complexity, and performance tradeoffs between the adders.*

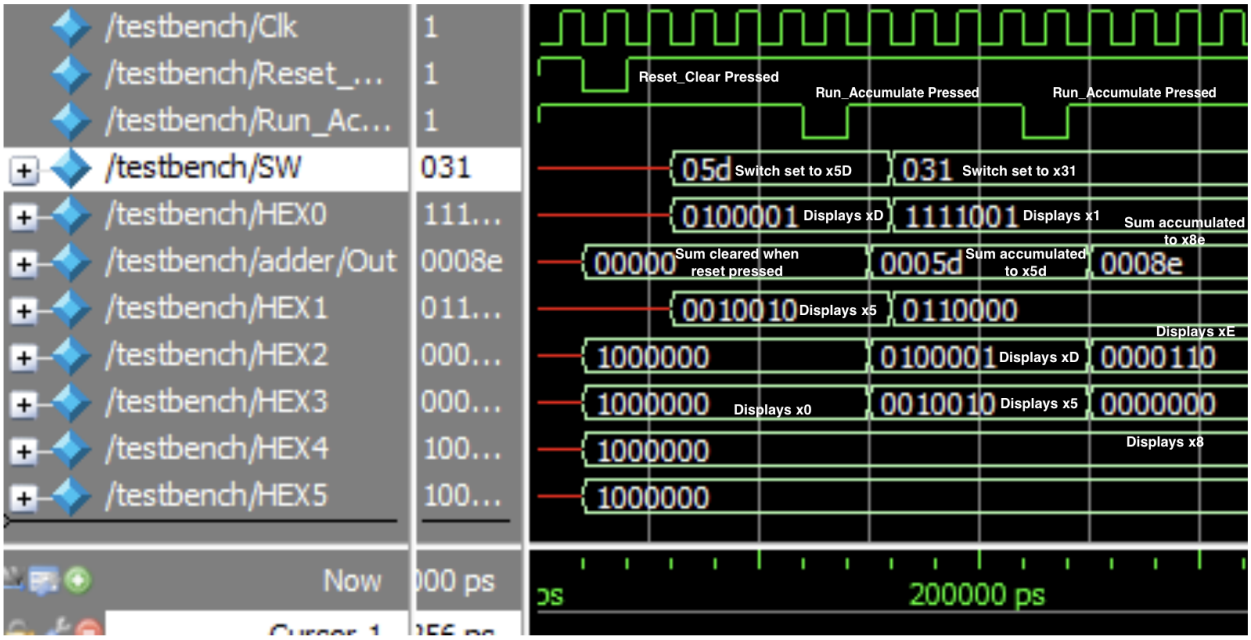
Tradeoffs:	Carry Ripple Adder	Carry Lookahead Adder	Carry Select Adder
Area	Requires Least Area (Implementation of only 4N adders for 4N-bits	Requires more area than CRA, additional combinational logic for P, G signals. 4N adders for 4N-bits	Most area required, implementation of additional adders, space required by MUXes. 8N-1 full adders for 4N-bits
Complexity	Relatively simple, uses direct-flow logic for carry bits.	Highly complex design, more combinational logic is require to extend the structure, and generate bits require significant computing.	Simpler than the CLA, more complex than the CRA. Uses consistent logic with MUX choosing between one of the 2 adders, and can be easily extended.
Performance	Low performance, slow execution, with large amount of latency. With an increase in number of bits, latency increases significantly. O(n)	Best performance, ensures that we are able to compute the Carry bits of the adder using combinational logic instead of waiting for the rippling scheme as in the CRA. O(logn)	Intermediate performance among the adders because each 4bit CRA parallelly computes sum, and outputs sum once c_in is determined O(√n)

- *Performance of our adders:*

	Carry-Ripple	Carry-Select	Carry-Lookahead
Memory (BRAM)	0	0	0
Frequency	64.47 MHz	68 MHz	67.8 MHz
Total Power	98.71 mW	98.71 mW	105.34 mW



- *Annotated simulation trace. We have only included a single diagram for all adders as RTL simulation does not account for the gate delays.*



SW: 10 bit value simulating switches
Out: Stores the sum displayed on the HEX display
HEX1, HEX0: Displays value set by switches
HEX5, HEX4, HEX3, HEX2: Displays Out[15:0]

Our testbench is designed to add x5D and x31. First Reset_Clear is pressed at t=20 ns and the Out which stores our sum is cleared. Next, the switch is set to x5D and Run_Accumulate is pressed at t = 120 ns. The sum is accumulated and stored in Out and the value reflects on the next clock cycle. The switches are then set to x31, and Run_Accumulate is pressed at t = 220 ns. The sum is accumulated and Out stores the sum x8E.

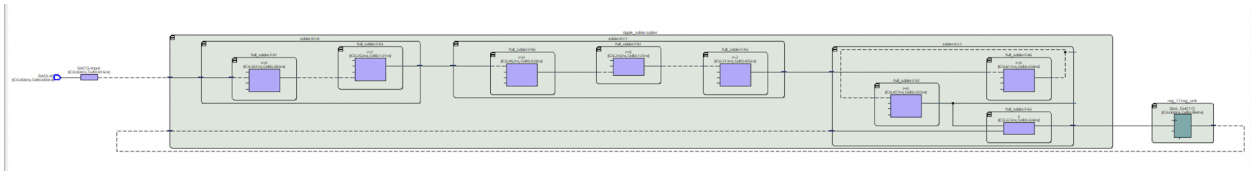
Extra Credit:

- *Critical path Analysis:*

We were instructed to find the critical path for each of the Adders we had implemented.

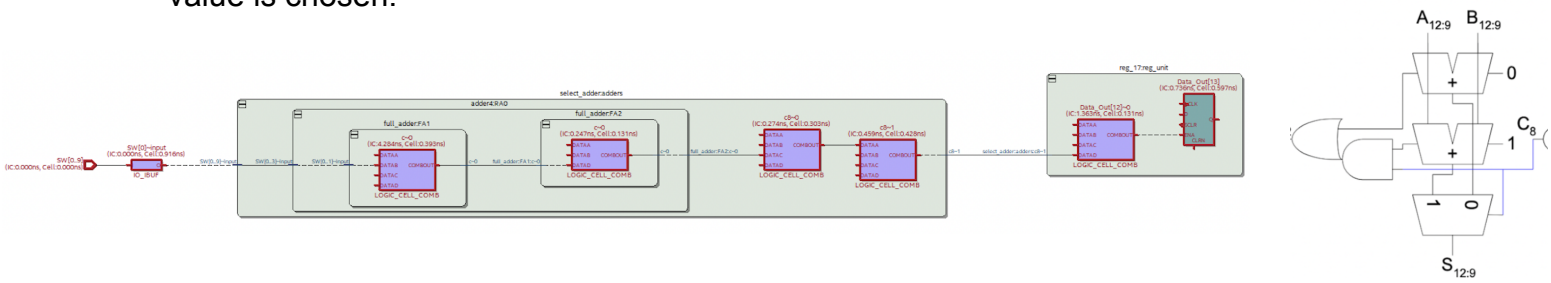
Carry Ripple Adder:

In this critical path tracing performed by Quartus, we notice that the critical path that is displayed is the one which goes through 8 of the full adder functions. This means that this specified path is the one that happens to slow down the functionality of the CRA. This is what we expected to see because in a Carry Ripple adder, the main functionality of traversing the carry bit through the adders is the slowest path for any addition that we perform.



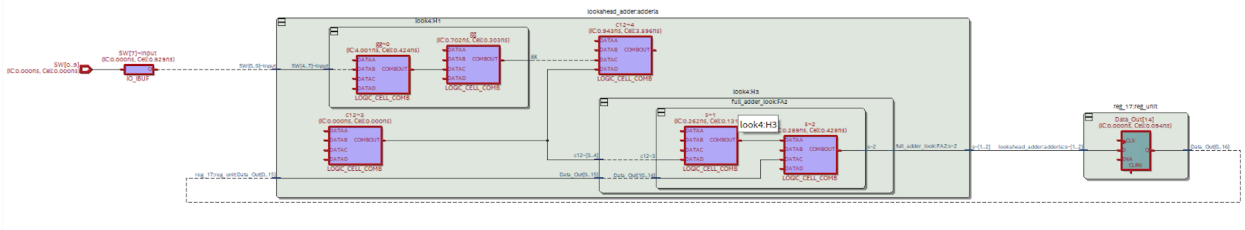
Carry Select Adder:

In the critical path diagram, we notice that the path traverses through 2 full adders, and then goes through combinational logic (of the MUX) before it gets stored in Data_out. This is the implementation of any of the paths in the Carry select adder, as shown below. This is what we expected the longest path of our CSA to look like, and this to be the slowest path. In this diagram, the switches send their inputs into the 2 Full adders simultaneously, the outputs of which are sent into the MUX, and the correct value is chosen.



Carry Lookahead Adder:

This adder was more confusing to analyze, simply because it involved far more combinational logic than any of the other 2 adders. In this, what we would expect is for the longest path length to traverse through a lot of logic required for the P, G, PG, GG logic. However, we would see a lot of parallelism in our structure as visible here.



3. Post-lab Questions:

1) *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?)*

When designing a CSA, there is a trade-off between the size of the CRA per unit, and total number of units. In our current implementation, we make use of 4 units with each unit comprising 2 4-bit CRA. The time taken for execution is dependent on the delay of the CRA (all the CRAs compute the sum in parallel) and the delay in propagation of the glue logic, and muxes. The 4x4 hierarchy may not be ideal. An ideal CSA should have a low propagation time through the ripple adder (fewer bits per CRA and in turn more units) but also have a lower number of muxes(fewer units and in turn more bits per CRA). To find this balance, we would need to know the propagation time for an n-bit carry ripple adder, and the time delay as a result of the mux and additional logic. To experiment, we would run the timing analyzer to determine the delays and create variants such as 2x8, 8x2, 1x16 CSAs.

---

2) *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.*

Design Statistics Table:

- *Carry Ripple Adder:*

LUT	79
DSP	0
Memory (BRAM)	0
Flip-Flop	20
Frequency	64.47 MHz
Static Power	89.94 mW
Dynamic Power	0 mW
Total Power	98.71 mW

- *Carry Lookahead Adder:*

LUT	86
DSP	0
Memory (BRAM)	0
Flip-Flop	20
Frequency	68 MHz
Static Power	89.94 MHz
Dynamic Power	0 mW
Total Power	98.71 mW

- *Carry Select Adder:*

LUT	82
DSP	0
Memory (BRAM)	0
Flip-Flop	20
Frequency	67.8 MHz
Static Power	89.97 MHz
Dynamic Power	1.58 mW
Total Power	105.34 mW

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?

When analyzing the Data plots, we can infer that the Carry Ripple Adder took the least number of LUTs, whereas the Carry Lookahead Adder took the most. This is what we could have expected due to the relative simplicity of the CRA, compared to the complexity of the combinational logic required to be implemented, which would use up more LUTs.
When we observe the maximum operating frequencies of the adders (F_max), we notice that the CLA has the highest operating frequency, significantly higher than the CRA. This is due to the fact that we have optimized the circuit to produce the result of the Sum bits and the C_out bits much faster in the CLA as compared to the CRA. When we compare these values to the CSA, we know that even though the CSA is significantly more optimized than the CRA, in comparison to the CLA, it operates at a similar level of efficiency. However, it still operates at a frequency lower than the CLA because the CLA involves a of combinational logic for its P, G bits, in comparison to the CSA. Theoretically, the CLA should be the fastest because it determines all values in parallel, but the overall combinational logic interferes and brings the frequency down.
The next difference we noticed was in the fact that the CRA, CLA use the same amount of total power. This is not what we expected as we thought that the presence of significant amounts of combinational logic in CLA would mean it dissipates more power. We however noticed that the CSA uses significantly more power which is what we expected, as the CSA has 28 Full Adders.

---

4. Conclusion:

In this lab, we utilize the knowledge we have gained about SystemVerilog to create various types of adders, synthesizing them on Quartus using ModelSim, and implementing them on our FPGA board. Whilst executing this lab, the section we struggled with was the creation of the Carry Select adder, implementing the hierarchical structure and incorporating our MUX structure. In addition, we found it the most difficult to identify a typing error that crept into our PG, GG statements, and we managed to utilize our testbench with only our CLA to fix the error.

In this lab, the only ambiguous section was understanding the diagram of the critical path that was generated. Despite having generated the image as required, we could not understand the structure that was generated by Quartus. In addition, working with files that we can directly import for pin assignments would save a significant amount of time.
