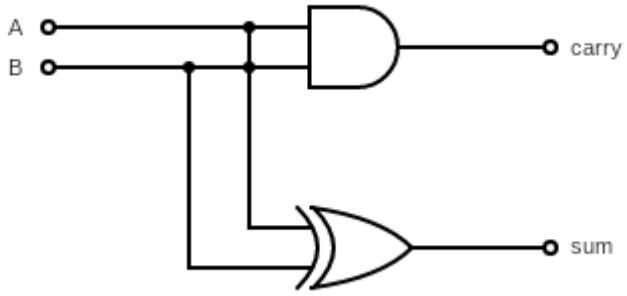
Computer Organization Laboratory CS39001
Assignment 01 - Introduction to Verilog Programming

Nakul Aggarwal (19CS10044) *Hritaban Ghosh (19CS30053)*

Problem 1A

Circuit diagram of half adder implemented in verilog



Half adder takes as input two bits *A* and *B* and produces a sum bit *sum* and a carry bit *carry*.

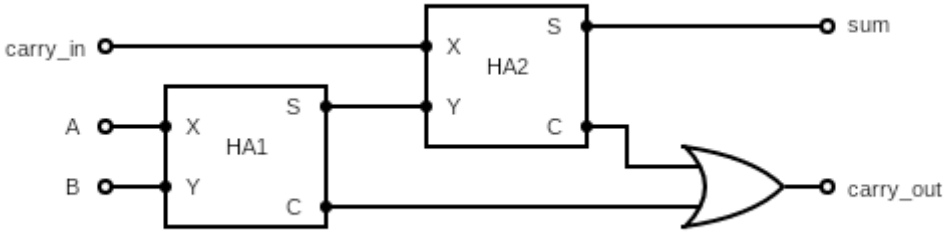
Truth table for the assignments of sum and carry bits in half adder

a	b	sum	carry_out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Please find the source code for half adder in file Half_Adder.v and the testbench for the same in Test_Bench_Half_Adder.v.

Problem 1B

Circuit diagram of full adder implemented in verilog



Full adder takes as input three bits *A*, *B* and an input carry bit *carry_in* and produces a sum bit *sum* and an output carry bit *carry_out*.

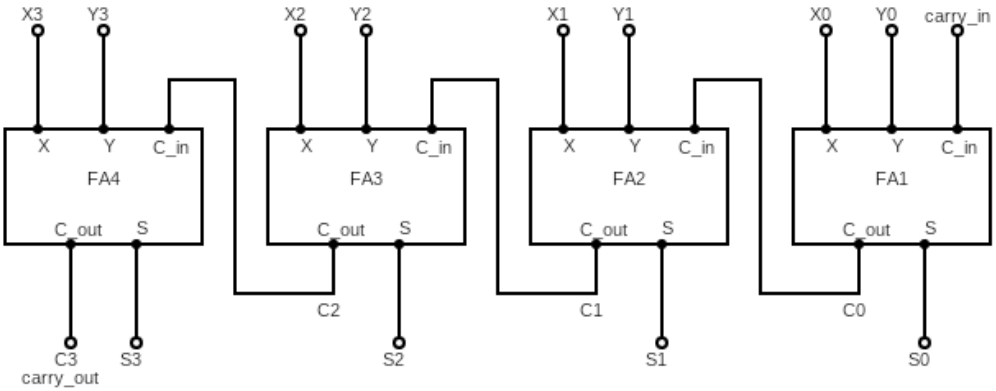
Truth table for the assignments of sum and carry bits in full adder

a	b	carry_in	sum	carry_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Please find the source code for full adder in file Full_Adder.v and the testbench for the same in Test_Bench_Full_Adder.v.

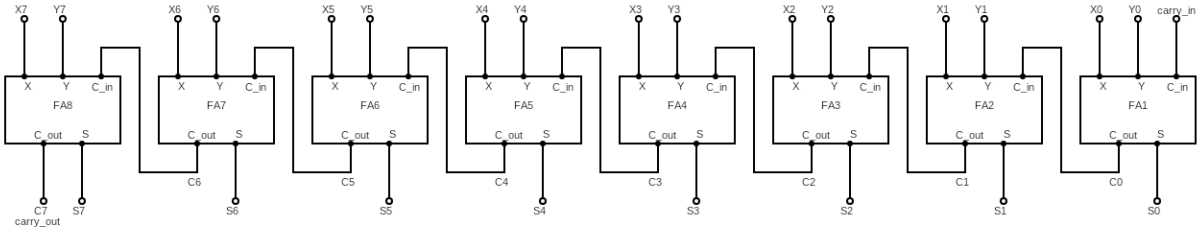
Problem 1C

Circuit diagram of 4-bit ripple carry adder implemented in verilog



4-bit Ripple Carry Adder takes in two 4-bit binary strings *X* and *Y* and an input carry bit *carry_in* as inputs, and outputs a 4-bit binary string *S* and an output carry bit *carry_out*. The circuit is designed in a hierarchical manner by cascading 4 *Full Adder* components such that the carry-out bit of one full adder is the carry-in of the next (carry bits are getting *rippled*).

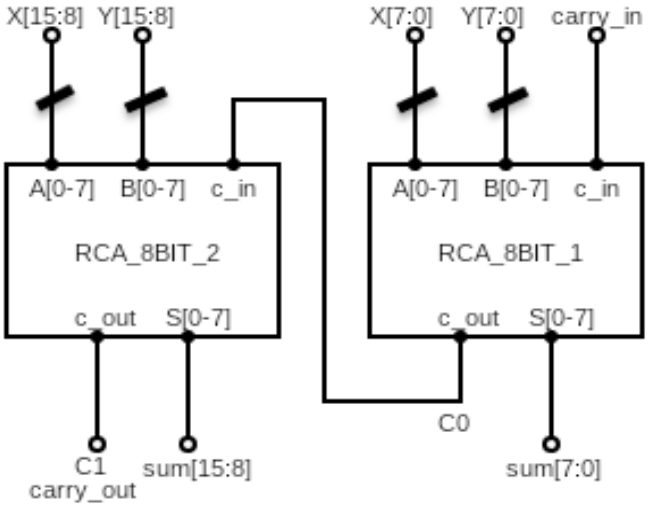
Circuit diagram of 8-bit ripple carry adder implemented in verilog



8-bit Ripple Carry Adder takes in two 8-bit binary strings *X* and *Y* and an input carry bit *carry_in* as inputs, and outputs an 8-bit binary string *S* and an output carry bit *carry_out*. The circuit is designed in a hierarchical

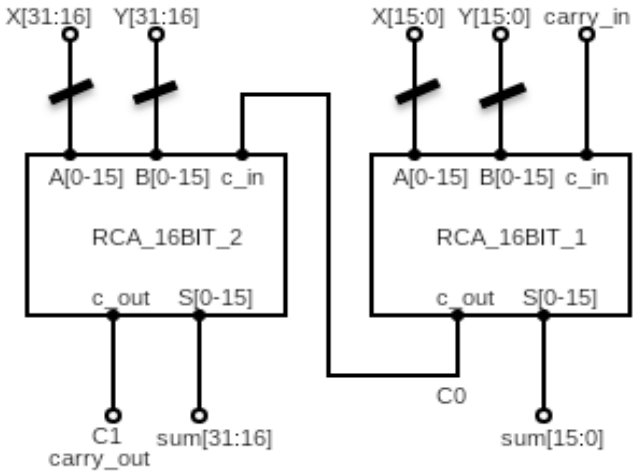
manner by cascading 8 *Full Adder* components such that the carry-out bit of one full adder is the carry-in of the next (carry bits are getting *rippled*).

Circuit diagram of 16-bit ripple carry adder implemented in verilog



16-bit Ripple Carry Adder takes in two 16-bit binary strings *X* and *Y* and an input carry bit *carry_in* as inputs, and outputs a 16-bit binary string *sum* and an output carry bit *carry_out*. The circuit is designed in a hierarchical manner by cascading 2 *8-bit Ripple Carry Adder* components such that the carry-out bit of one is the carry-in of the next. If the circuit is looked at a lower level, it is equivalent to 16 *Full Adder* components cascaded together. Hierarchical design reuses the circuitry implemented for *8-bit Ripple Carry Adder* without having to implement everything from scratch.
The 8 least significant bits of *X* and *Y* are the input binary strings for the first ripple carry adder. The 8 most significant bits of *X* and *Y* are the input binary strings for the second ripple carry adder. The first ripple carry adder outputs the 8 least significant bits of the sum *(X+Y)* and the second adder outputs the 8 most significant bits of the sum.

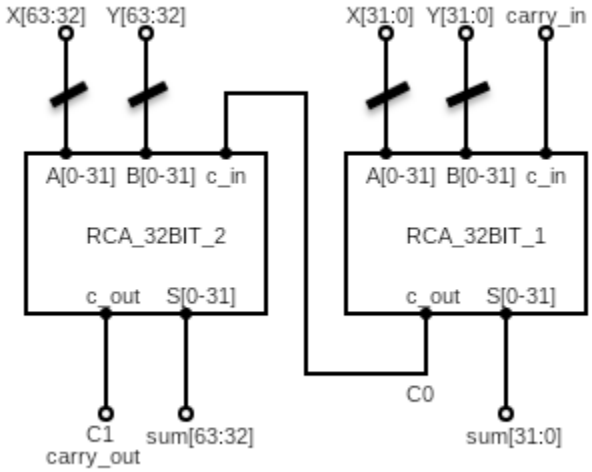
Circuit diagram of 32-bit ripple carry adder implemented in verilog



32-bit Ripple Carry Adder takes in two 32-bit binary strings *X* and *Y* and an input carry bit *carry_in* as inputs, and outputs a 32-bit binary string *sum* and an output carry bit *carry_out*. The circuit is designed in a hierarchical manner by cascading 2 *16-bit Ripple Carry Adder* components such that the carry-out bit of one is the carry-in of the next. If the circuit is looked at a lower level, it is equivalent to 32 *Full Adder* components cascaded together. Hierarchical design reuses the circuitry implemented for *16-bit Ripple Carry Adder* without having to implement everything from scratch.

The 16 least significant bits of *X* and *Y* are the input binary strings for the first ripple carry adder. The 16 most significant bits of *X* and *Y* are the input binary strings for the second ripple carry adder. The first ripple carry adder outputs the 16 least significant bits of the sum *(X+Y)* and the second adder outputs the 16 most significant bits of the sum.

Circuit diagram of 64-bit ripple carry adder implemented in verilog



64-bit Ripple Carry Adder takes in two 64-bit binary strings *X* and *Y* and an input carry bit *carry_in* as inputs, and outputs a 64-bit binary string *sum* and an output carry bit *carry_out*. The circuit is designed in a hierarchical manner by cascading 2 *32-bit Ripple Carry Adder* components such that the carry-out bit of one is the carry-in of the next. If the circuit is looked at a lower level, it is equivalent to 64 *Full Adder* components cascaded together. Hierarchical design reuses the circuitry implemented for *32-bit Ripple Carry Adder* without having to implement everything from scratch.
The 32 least significant bits of *X* and *Y* are the input binary strings for the first ripple carry adder. The 32 most significant bits of *X* and *Y* are the input binary strings for the second ripple carry adder. The first ripple carry adder outputs the 32 least significant bits of the sum *(X+Y)* and the second adder outputs the 32 most significant bits of the sum.

Longest delays in the 4-bit, 8-bit, 16-bit, 32-bit and 64-bit ripple carry adder circuits.
Longest Delay in 4-bit Ripple Carry Adder = 5.795ns (Levels of Logic = 20)
Longest Delay in 8-bit Ripple Carry Adder = 10.639ns (Levels of Logic = 36)
Longest Delay in 16-bit Ripple Carry Adder = 20.327ns (Levels of Logic = 70)
Longest Delay in 32-bit Ripple Carry Adder = 39.703ns (Levels of Logic = 138)
Longest Delay in 64-bit Ripple Carry Adder = 78.455ns (Levels of Logic = 274)

Conclusion: The longest delays in *n-bit ripple carry adders* are observed to increase monotonically with *n*. Hence, we can conclude that the longer the carry chain i.e. the longer the path for the rippling carry bit, more time is required for the adder to complete its computation. We can see that the subsequent full adders have to wait for the carry bit to arrive and thus the levels of logic keep increasing as more and more full adders are added in the carry chain.

Please find the source code for 4-bit, 8-bit, 16-bit, 32-bit, 64-bit ripple carry adders in file RCA_4_bit.v, RCA_8_bit.v, RCA_16_bit.v, RCA_32_bit.v, RCA_64_bit.v respectively and the testbenches for the same in Test_Bench_RCA_4_bit.v, Test_Bench_RCA_8_bit.v, Test_Bench_RCA_16_bit.v, Test_Bench_RCA_32_bit.v, Test_Bench_RCA_64_bit.v.

Problem 1D

How can you use the above circuit, to compute the difference between two n-bit numbers?

Difference between two *n*-bit binary strings *x* and *y* (i.e, *x*-*y*) can be interpreted as the sum of binary strings *x* and *y'* where *y'* is the 2's complement of *y*. This implies that we can re-use a significant amount of circuitry implemented in addition to perform subtraction between two binary strings as well.

Let us introduce a control line *K* as another input for the ripple carry adder that sends a single bit (0 or 1) as input. The input *K* is 1 if and only if subtraction is desired.

Algorithmic Requirements

- When *K* is 0, the circuit should behave exactly the same as it did when *K* control line was not introduced (addition behavior).
- When *K* is 1, all the *n* bits in *y* must be flipped before adding it to *x*. Post addition, 1 must be added to the *n*-bit output binary string to obtain the final *n*-bit difference (subtraction behavior).

Motivation

The motivation behind the addition algorithm is the same as for the ripple carry adder circuit that is already designed and implemented in *verilog*.

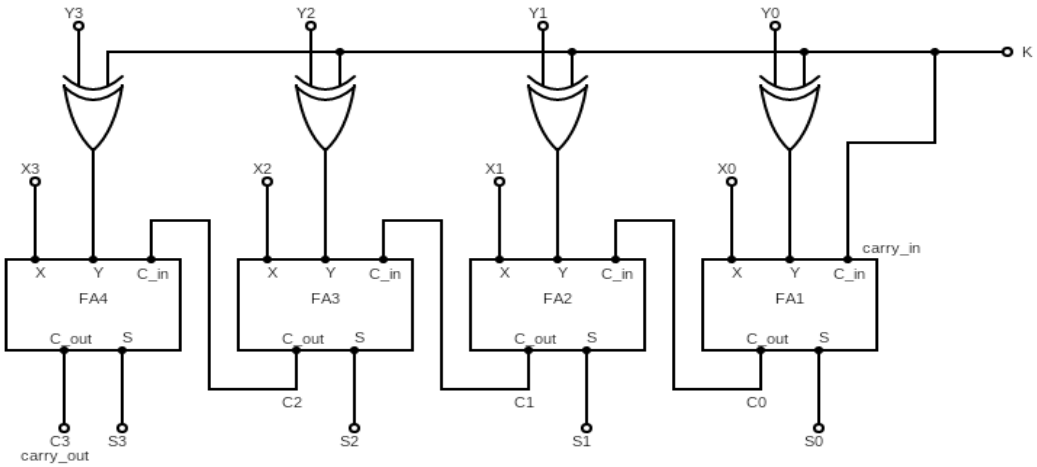
The motivation behind the subtraction algorithm is the fact that the 2's complement of an **n**-bit binary string *x* is *x' = x'' + 1* where *x''* is the 1's complement of *x*, i.e, another *n*-bit binary string such that *i*-th bit of *x'* is 1 if and only if *i*-th bit of x is 0.

(x - y) can be computed as *(x + y')* where *y'* is the 2's complement of *y*, which in turn can be computed as *(x + y'' + 1)* where *y''* is the 1's complement of *y*. Therefore, before utilizing the circuitry of addition, all the bits in *y* must be flipped to obtain its 1's complement *y''*. Now post addition, in order to account for the third operand in *(x+y''+1)*, 1 must be added to the *n*-bit intermediate binary output obtained in addition. This is the final *n*-bit binary string which is equal to the difference *(x - y)*.

Modified Design -- Description

The design of all the components/modules will remain the same except that of the ripple carry adder, i.e, the half adder and full adder modules can be used intact by the modified ripple carry adder. Here we are explaining the changes particularly wrt the 8-bit ripple carry adder (module *Ripple_Carry_Adder_8Bit*). The only change for any *n*-bit ripple carry adder would be to cascade *n* full-adder components, in a similar way as the 8 full adder components were cascaded in *Ripple_Carry_Adder_8Bit*.

- Consider the design of the *Ripple_Carry_Adder_nBit* module having n full adder components (for any *n*).
- Remove the *input-carry-port* from *Ripple_Carry_Adder_nBit*.
- Introduce a new input port to accept *K* as an input.
- Introduce *n XOR* gates.
- Set the *y* input bits (*y[0], y[1], ..., y[n-1]*) of each of the *n* full adder components as the first inputs of each of the *n XOR* gates.
- Connect the output ports of each of the *n XOR* gates with the *Y*-input-ports of each of the *n* full adder components, respectively.
- Set the second inputs of all the *n XOR* gates as the *K* input bit.
- Set *K* as the input-carry for the first full adder component (*FA1*) in place of *carry_in*.



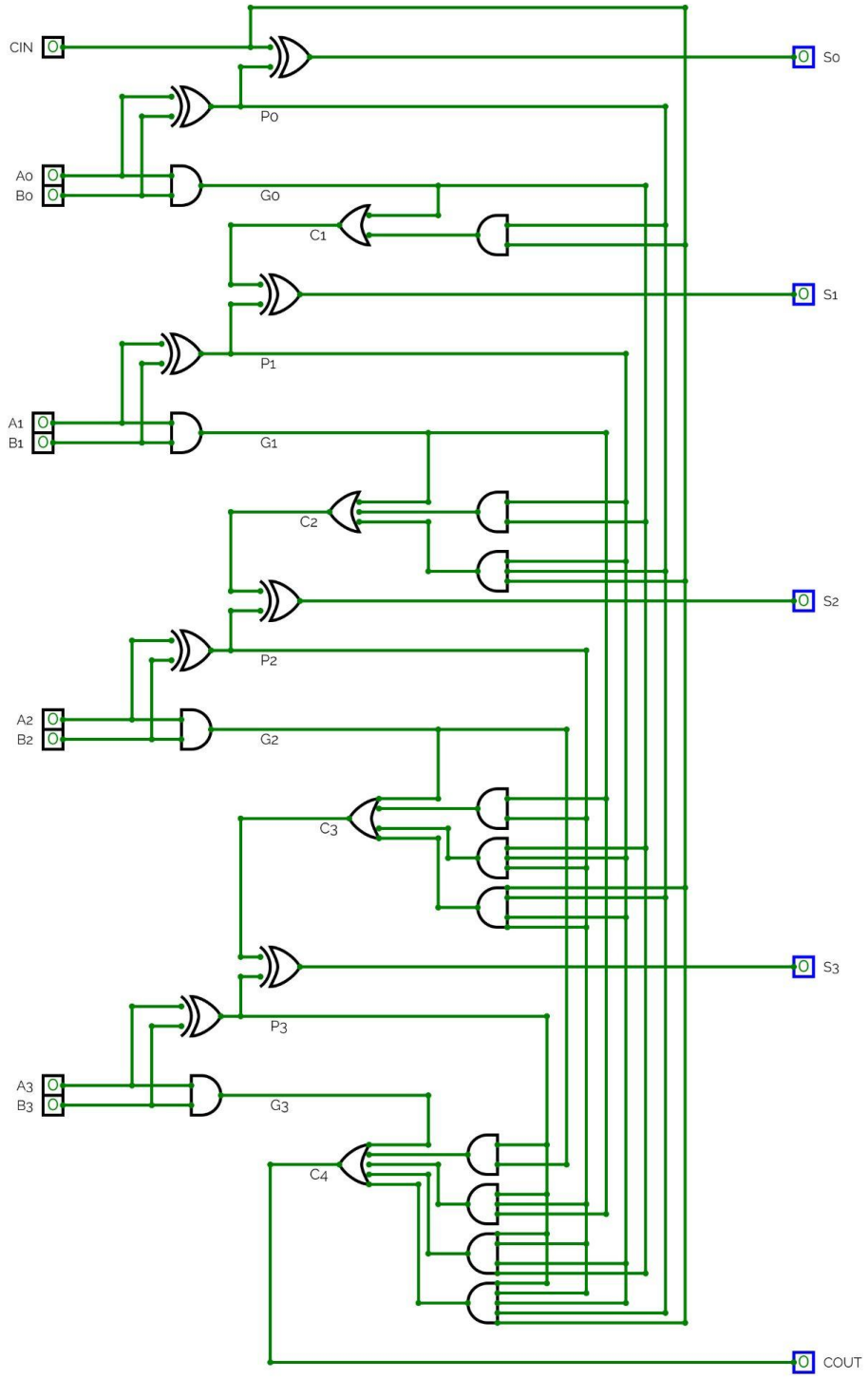
Modified Design -- Result

- If *K* input bit is 0 (addition operation), the outputs of the *XOR* gates will be the same as their first inputs, i.e, the unchanged bits of *y* binary string. Besides, the input carry bit will also be 0. Clearly, in this case the circuit behaves like the typical ripple carry adder, like is implemented in *verilog* in this assignment.
- If *K* input bit is 1 (subtraction operation), the outputs of all the *n XOR* gates will be different from their respective first inputs, i.e, the flipped bits of *y* binary string. Besides, the input carry bit will also be 1. Clearly, in this case the circuit behaves exactly like it was desired in the *Algorithmic Requirements*. In other words, when *K* is 1, the circuit adds *x* to the 2's complement of *y* (first to 1's complement of *y* and then to 1).

Hence, this modified design of *n-bit ripple carry adder* can be used to implement *n-bit ripple carry adder subtractor* by re-using the original circuitry.

Problem 2A

Circuit diagram of 4-bit carry look-ahead adder implemented in verilog



4-bit Carry Look-ahead Adder takes in two 4-bit binary strings *A* and *B* and an input-carry bit *cin* as inputs, and outputs a 4-bit binary string *S* and an output-carry bit *cout*.

This adder uses additional hardware to generate the carry bits and that is why it is called Carry Look-ahead Adder.

One of the weaknesses in the *Ripple Carry Adder* is the large delay because of the rippling of the carry, through what we call as the carry-chain. We endeavour to design a high-speed adder using the technique of what we call the *Carry Look-ahead Adder (CLA)*. It uses additional hardware to compute the carry bits.

The key idea to generate the carry bits involves

- A carry is generated in the current step if both *a[i]* and *b[i]* are 1 (Generate Signals)
  (or)
- either *a[i]* is 1 or *b[i]* is 1 (but not both) and there is a carry from the previous step (Propagate Signals)

State the boolean equations for the carry look-ahead adder that determines the values of *sum* bits (S0, S1, S2, S3) in terms of carry bits (C0, C1, C2, C3) and propagate signals (P0, P1, P2, P3).

$S0 = P0 \text{ ^ } C0$
$S1 = P1 \text{ ^ } C1$
$S2 = P2 \text{ ^ } C2$
$S3 = P3 \text{ ^ } C3$
** C0 is the input carry bit.*

State the boolean equations of the look-ahead carry generation for the 4 carry bits (C1, C2, C3, C4) in terms of the generate (G0, G1, G2, G3) and propagate (P0, P1, P2, P3, P4) signals.

$C1 = G0 + P0.C0$
$C2 = G1 + P1.G0 + P1.P0.C0$
$C3 = G2 + P2.G1 + P2.P1.G0 + P2.P1.P0.C0$
$C4 = G3 + P3.G2 + P3.P2.G1 + P3.P2.P1.G0 + P3.P2.P1.P0.C0$
** C0 is the input carry bit.*

State the equations for the generate and propagate signals in look-ahead carry generation.

$G0 = X[0] \,\&\, Y[0]$
$G1 = X[1] \,\&\, Y[1]$
$G2 = X[2] \,\&\, Y[2]$
$G3 = X[3] \,\&\, Y[3]$

$P0 = X[0] \text{ ^ } Y[0]$
$P1 = X[1] \text{ ^ } Y[1]$
$P2 = X[2] \text{ ^ } Y[2]$
$P3 = X[3] \text{ ^ } Y[3]$
** X and Y are 4-bit binary strings. X[0] denotes the least and X[3] denotes the most significant bit.*

Problem 2B

Please find the source code for 4-bit carry look-ahead adder in file CLA_4_bit.v and the testbench for the same in Test_Bench_CLA_4_bit.v.

Compare the speeds of 4-bit carry look-ahead adder with the 4-bit ripple carry adder.

Longest Delay in 4-bit Carry Look Adder = 2.123ns (Levels of Logic = 4)
Longest Delay in 4-bit Ripple Carry Adder = 5.795ns (Levels of Logic = 20)

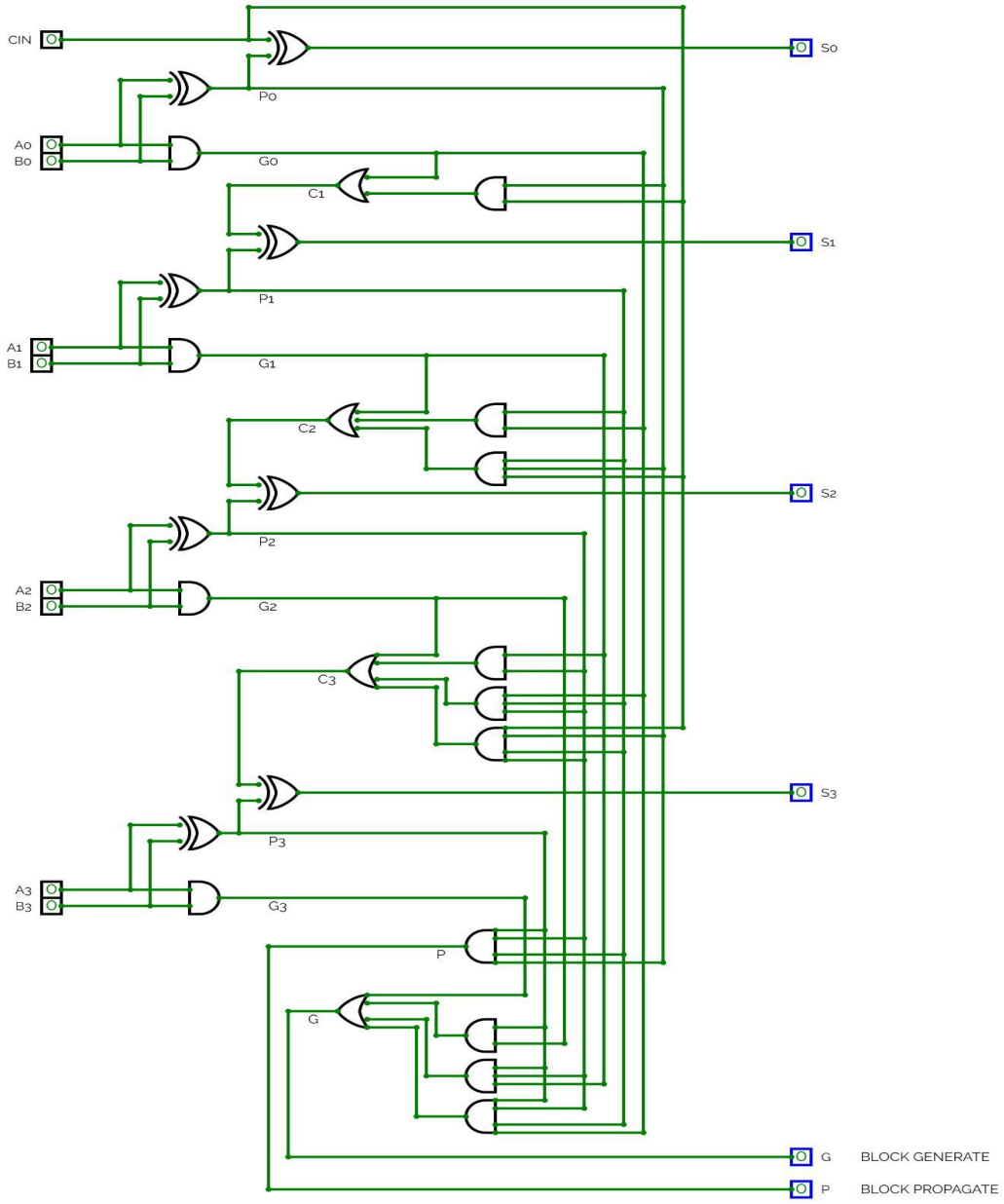
Speed Up of the 4-bit CLA over the 4-bit RCA = 5.795 / 2.123 = 2.730

Conclusion: Hence, it is clear that a 4-bit RCA where the carry is rippled in throughout the carry chain requires more time for its computation to complete, than a 4-bit CLA which uses additional hardware to compute the carry bits. In a 4-bit RCA the subsequent full adders have to wait for the carry bit to arrive and thus the level of logic increases as more and more full adders are cascaded into the carry chain. On the other hand in a 4-bit CLA, additional hardware is used to compute the carry bits, but with the help of concurrent execution, the level of logic is still 4, making the 4-bit CLA a much faster adder.

For each bit in a binary sequence to be added, the carry-lookahead logic will determine whether that bit pair will generate a carry or propagate a carry. This allows the circuit to "pre-process" the two numbers being added to determine the carry ahead of time. Then, when the actual addition is performed, there is no delay from waiting for the ripple-carry effect (or time it takes for the carry from the first full adder to be passed down to the last full adder).

Problem 2C.i

Circuit diagram of 4-bit carry look-ahead adder augmented to compute the block propagate and generate signals, implemented in verilog

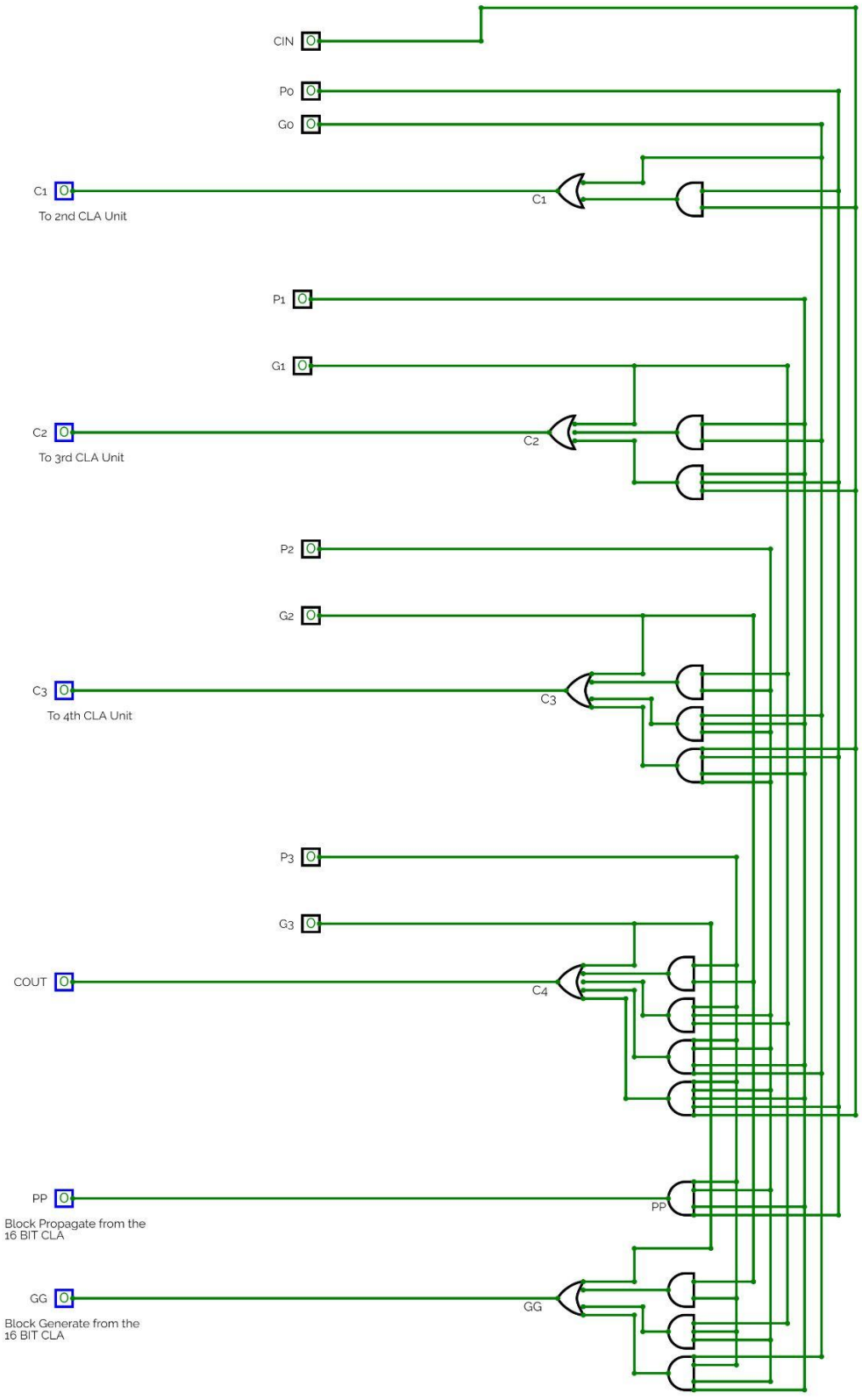


4-Bit Carry Look-ahead Adder (augmented to output block propagate and block generate signals) takes in two 4-bit binary strings *A* and *B* and an input-carry bit *cin* as inputs, and outputs a 4-bit binary string *S* along with block propagate bit *PP* and block generate bit *GG*. The circuit is implemented using primitive logic gates only.

Please find the source code for 4-bit carry look-ahead adder augmented to compute the block propagate and generate signals in file CLA_4_bit_Augmented.v and the testbench for the same in Test_Bench_CLA_4_bit_Augmented.v.

Problem 2C.ii

Circuit diagram of look-ahead carry unit implemented in verilog



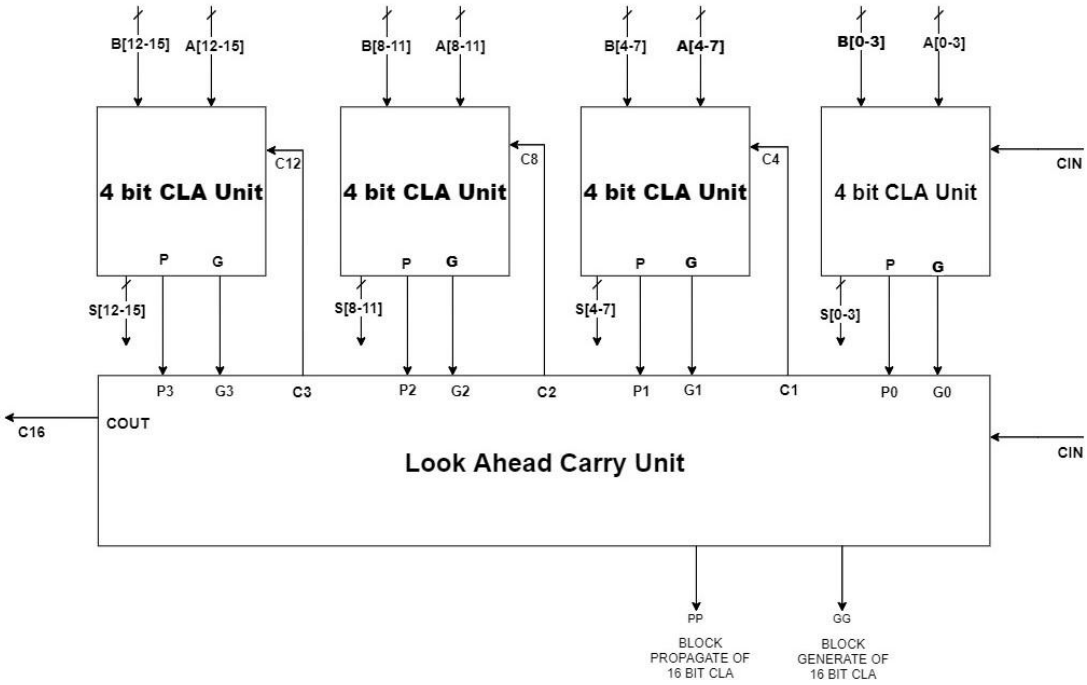
Look-ahead carry unit generates carry bits at a higher level of hierarchy. It takes 2 4-bit binary strings, *block propagate signals P* and *block generate signals G* along with an input carry bit *cin* as inputs and gives 4 carry bits along with block propagate and block generate signals (produced for the next level of hierarchy) as outputs. The circuit is implemented using primitive logic gates only. The boolean equations governing the values of the carry bits are as follows.

$C0 = G0 \mid (P0 \ \& \ cin)$
$C1 = G1 \mid (P1 \ \& \ C0)$
$C2 = G2 \mid (P2 \ \& \ C1)$
$C3 = G3 \mid (P3 \ \& \ C2)$

Expanded form of the above equations is as follows.
$C0 = G0 \mid (P0 \ \& \ cin)$
$C1 = G1 \mid (P1 \ \& \ (G0 \mid (P0 \ \& \ cin)))$
$C2 = G2 \mid (P2 \ \& \ (G1 \mid (P1 \ \& \ (G0 \mid (P0 \ \& \ cin)))))$
$C3 = G3 \mid (P3 \ \& \ (G2 \mid (P2 \ \& \ (G1 \mid (P1 \ \& \ (G0 \mid (P0 \ \& \ cin)))))))$

The *block propagate* and *block generate* signals for the next level of hierarchy are determined by the following equations.
$P = P3.P2.P1.P0$
$G = G3 + P3.G2 + P3.P2.G1 + P3.P2.P1.G0$

Circuit diagram of 16-bit carry look-ahead adder integrated with look-ahead carry unit, implemented in verilog

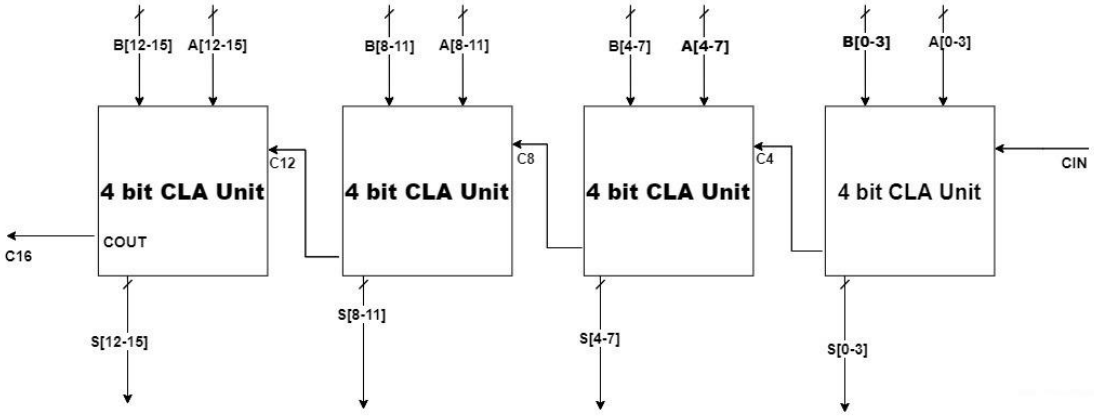


16-Bit Carry Look-ahead Adder integrated with look-ahead carry unit takes in two 4-bit binary strings *A* and *B* and an input-carry bit *cin* as inputs; and outputs a 16-bit binary string *S* and an output-carry bit *C16* and also Block Propagate PP and Block Generate GG for the next level of hierarchy.
The circuit is designed in hierarchical fashion by cascading 4 *4-bit carry look-ahead adders*, each one of which is augmented to produce block propagate and generate signals. These signals are fed into the look-ahead carry unit to obtain the carry bit that is fed into the next *CLA* component.

Please find the source code for the look-ahead carry unit in file LookAhead_Carry_Unit.v and the testbench for the same in Test_Bench_LookAhead_Carry_Unit.v. Find the source code for 16-bit CLA integrated with a look-ahead carry unit in file CLA_16_bit.v and the testbench for the same in Test_Bench_CLA_16_bit.v.

Problem 2C.iii

Circuit diagram of 16-bit carry look-ahead adder with rippling carry, implemented in verilog



16-Bit Carry Look-ahead Adder (with rippling carry) takes in two 4-bit binary strings *A* and *B* and an input-carry bit *cin* as inputs; and outputs a 16-bit binary string *S* and an output-carry bit *C16*.
The circuit is designed in hierarchical fashion by cascading 4 *4-bit carry look-ahead adders* such that the output carry of one *CLA* unit is the input carry of the next unit.

Please find the source code for 4-bit carry look-ahead adder in file CLA_4_bit.v and the testbench for the same in Test_Bench_CLA_4_bit.v. Find the source code for 16-bit CLA with rippling carry in file CLA_16_bit_with_Rippling_Carry.v and the testbench for the same in Test_Bench_CLA_16_bit_with_Rippling_Carry.v.

Compare the delays for obtaining the sum and final carry-out bits for the 16-bit carry look-ahead adder, one with integrated look-ahead carry unit and the other with rippling carry.
Longest Delay in 16-bit Carry Look Adder with Look Ahead Carry Unit = 5.447ns (Levels of Logic = 11)
Longest Delay in 16-bit Carry Look Adder with Rippling Carry = 6.167ns (Levels of Logic = 14)

Conclusion: Therefore, we can conclude that as the carry is rippled in from one CLA unit to another in 16-bit Carry Look Adder with Rippling Carry it takes more time, compared to when carry bits are computed separately using additional hardware(Look Ahead Carry Unit) in 16-bit Carry Look Adder with Look Ahead Carry Unit.

Problem 2C.iv

Compare the speeds and LUT (look-up table cost of the FPGA) of 16-bit carry look-ahead adder with integrated look-ahead carry unit and 16-bit ripple carry adder.
Longest Delay in 16-bit Carry Look Adder with Look Ahead Carry Unit = 5.447ns (Levels of Logic = 11)
Longest Delay in 16-bit Ripple Carry Adder = 20.327ns (Levels of Logic = 70)

Speed up of 16-bit CLA over the 16-bit RCA = 20.327/5.447 = 3.731

Conclusion: Hence, it is clear that a 16-bit RCA where the carry is rippled in throughout the carry chain requires more time for its computation to complete, than a 16-bit CLA which uses additional hardware(Look Ahead Carry Unit) to compute the carry bits. In a 16-bit RCA the subsequent full adders have to wait for the carry bit to arrive and thus the level of logic increases as more and more full adders are cascaded into the carry chain. On the other hand in a 16-bit CLA, additional hardware is used to compute the carry bits, but with the help of concurrent execution, the level of logic is still 11, making the 16-bit CLA a much faster adder.

LUT (look-up table cost of the FPGA) Comparisons of 16-bit carry look-ahead adder with integrated look-ahead carry unit and 16-bit ripple carry adder.

Device utilization summary: **----------**	**16-bit carry look-ahead adder with integrated look-ahead carry unit**	**16-bit ripple carry adder**
Slice Logic Utilization:		
Number of Slice LUTs:	43 out of 63400 (0%)	80 out of 63400 (0%)
Number used as Logic:	43 out of 63400 (0%)	80 out of 63400 (0%)
Slice Logic Distribution:		
Number of LUT Flip Flop pairs used:	43	80
Number with an unused Flip Flop:	43 out of 43 (100%)	80 out of 80 (100%)
Number with an unused LUT:	0 out of 43 (0%)	0 out of 80 (0%)
Number of fully used LUT-FF pairs:	0 out of 43 (0%)	0 out of 80 (0%)
Number of unique control sets:	0	0
IO Utilization:		
Number of IOs:	52	50
Number of bonded IOBs:	52 out of 210 (24%)	50 out of 210 (23%)

Conclusion : An LUT, which stands for LookUp Table, in general terms is basically a table that determines what the output is for any given input(s). In the context of combinational logic, it is the truth table. This truth table effectively defines how your combinatorial logic behaves.
In other words, whatever behavior you get by interconnecting any number of gates (like AND, NOR, etc.), can be implemented by an LUT. Therefore as the complexity of hardware increases the number of LUTs increases. Thus the LUT cost of a 16-bit RCA is much higher than that of a 16-bit CLA with Look Ahead Carry Unit.
Number of IOs utilized is explained as follows.
For the 16-bit Ripple Carry Adder

50 = 32-bit input (2 16-bit input values) + 1 carry-in bit + 16-bit output + 1 carry-out bit

For the 16-bit Carry Look Ahead Adder with look-ahead carry unit

52 = 32-bit input (2 16-bit input values) + 1 carry-in bit + 16-bit output + 1 carry-out bit + 1-bit Block Propagate + 1-bit Block Generate

Compare the speeds and LUT (look-up table cost of the FPGA) of 16-bit carry look-ahead adder with rippling carry and 16-bit ripple carry adder.
Longest Delay in 16-bit Carry Look Adder with Rippling Carry = 6.167ns (Levels of Logic = 14)
Longest Delay in 16-bit Ripple Carry Adder = 20.327ns (Levels of Logic = 70)

Speed up of 16-bit CLA with Rippling Carry over the 16-bit RCA = $20.327 / 6.167 = 3.296$

Conclusion: Hence, it is clear that a 16-bit RCA where the carry is rippled in throughout the carry chain requires more time for its computation to complete, than a 16-bit CLA with Rippling Carry which uses 4-bit CLAs as its component module which is a much faster adder compared to a 4-bit RCA. In a 16-bit RCA the subsequent full adders have to wait for the carry bit to arrive and thus the level of logic increases as more and more full adders are cascaded into the carry chain. On the other hand in a 16-bit CLA with Rippling Carry, which uses faster adders (4-bit CLA as components) thus reducing the overall computation time even though the carry is rippled from one CLA unit to another, and limiting the level of logic to 14.

LUT (look-up table cost of the FPGA) Comparisons of 16-bit carry look-ahead adder with rippling carry and 16-bit ripple carry adder.

Device utilization summary: **----------**	**16-bit carry look-ahead adder with rippling carry**	**16-bit ripple carry adder**
Slice Logic Utilization:		
Number of Slice LUTs:	24 out of 63400 (0%)	80 out of 63400 (0%)
Number used as Logic:	24 out of 63400 (0%)	80 out of 63400 (0%)
Slice Logic Distribution:		
Number of LUT Flip Flop pairs used:	24	80
Number with an unused Flip Flop:	24 out of 24 (100%)	80 out of 80 (100%)
Number with an unused LUT:	0 out of 24 (0%)	0 out of 80 (0%)
Number of fully used LUT-FF pairs:	0 out of 24 (0%)	0 out of 80 (0%)
Number of unique control sets:	0	0
IO Utilization:		
Number of IOs:	50	50
Number of bonded IOBs:	50 out of 210 (23%)	50 out of 210 (23%)

Conclusion : An LUT, which stands for LookUp Table, in general terms is basically a table that determines what the output is for any given input(s). In the context of combinational logic, it is the truth table. This truth table effectively defines how your combinatorial logic behaves.
In other words, whatever behavior you get by interconnecting any number of gates (like AND, NOR, etc.), can be implemented by an LUT. Therefore as the complexity of hardware increases the number of LUTs increases. Thus the LUT cost of a 16-bit RCA is much higher than that of a 16-bit CLA with Rippling Carry.
Number of IOs utilized is explained as follows.
50 = 32-bit input (2 16-bit input values) + 1 carry-in bit + 16-bit output + 1 carry-out bit
This is the same for both 16-bit CLA with rippling carry and 16-bit RCA.

References

The following online platforms were used for drawing the circuit diagrams

- https://www.circuit-diagram.org/
- https://online.visual-paradigm.com/
- https://circuitverse.org/