Ben Paulson
CPE 133-80-2206

Half Adder, Full Adder, and Ripple Carry Adder

Objectives

The purpose of this lab was to implement digital logic equations in both sum-of-products (SOP) form and product-of-sums (POS) form in hardware. The equations were implemented using System Verilog language on a Basys 3 FPGA board. The first of these equations were a 1-bit half adder (HA) and a 1-bit full adder (FA). These base adder modules were then used to create a ripple carry adder (RCA) to demonstrate the notion of hierarchical digital design and code reuse.

Procedures

The first circuit to be built was the half adder. To determine the appropriate equation, the black box diagram in *Figure 1* and the truth table in *Figure 2* were drawn. The inputs A and B are both 1 bit, as are the outputs SUM and CO (carry out).

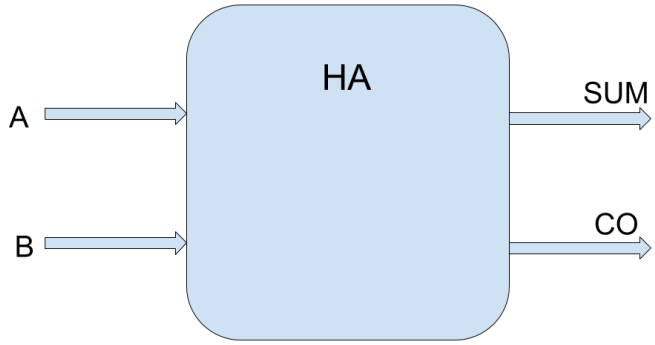


Figure 1 - Half Adder Black Box Diagram

A	B	SUM	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 2 - Half Adder Truth Table

Using the truth table, an equation in standard SOP form can be easily found for both SUM and CO outputs.

$$SUM = \bar{A}B + A\bar{B}$$
$$CO = AB$$

The truth table can also be used to find the standard POS form of both outputs by first writing SOP equations for the places where SUM and CO are 0 instead of 1 ($\overline{SUM}$ and $\overline{CO}$).

$$\overline{SUM} = \overline{A}\overline{B} + AB$$
$$\overline{CO} = \overline{A}\overline{B} + \bar{A}B + A\bar{B}$$

Using DeMorgan's Theorem, these can be simplified to give the final standard POS form.

$$SUM = (A + B)(\bar{A} + \bar{B})$$
$$CO = (A + B)(A + \bar{B})(\bar{A} + B)$$

These equations can then be used to implement the circuit in hardware. The schematics shown in figures 3 and 4 are the half adder in SOP and POS form, respectively.

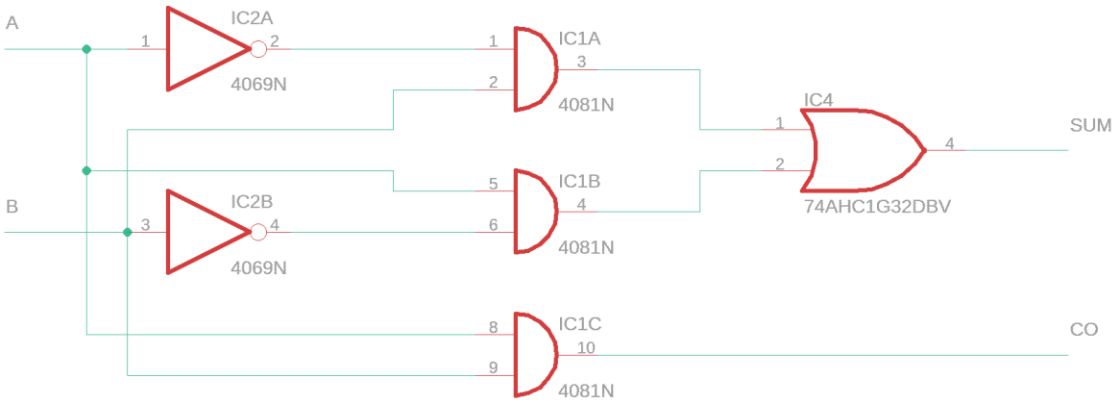


Figure 3 (Above) - Half Adder SOP Schematic

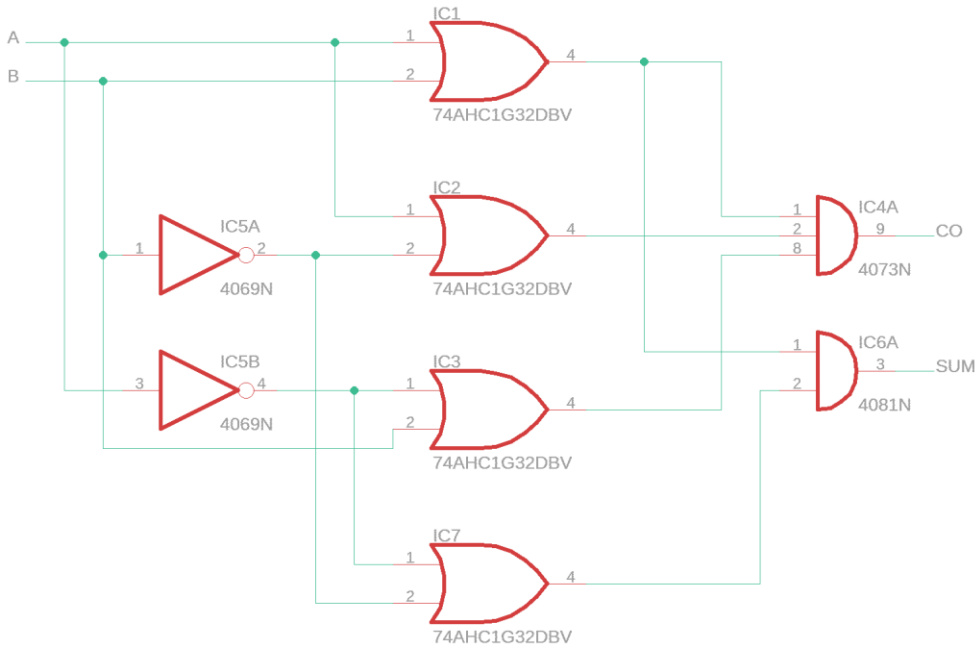


Figure 4 - Half Adder POS Schematic

To complete the half adder, the equations for both SOP and POS forms were implemented on the Basys 3 board. First, a SystemVerilog module HalfAdderSOP was created to assign inputs and output values for the SOP form of the HA (see *Appendix A*). To map the inputs and outputs in this module to physical hardware on the board, a constraints file had to be created (see *Appendix B*). The same process was then repeated for the POS form of the equations (see *Appendix D* and *E*). The SOP form of the HA was also implemented in the same files as the POS form to illustrate that they operate the same.

Next, the full adder was designed and built. Following the same design method as with the HA, the FA was created according to the black box diagram and truth table in *Figure 5* and *Figure 6*, respectively.

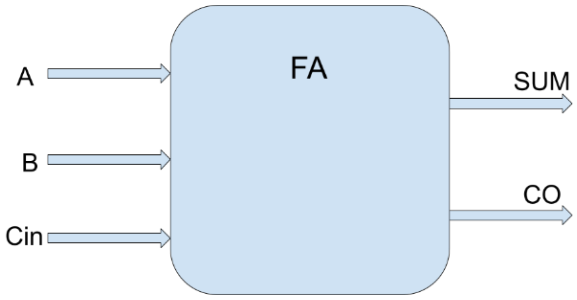


Figure 5 – Full Adder Black Box Diagram

A	B	Cin	SUM	CO
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

Figure 6 – Full Adder Truth Table

The key difference between the HA and the FA is the additional input, carry in, which is also added to the other inputs A and B. As with the HA, the truth table in *Figure 6* can be used to find standard SOP equations for both SUM and CO. The following equations were found (with C as Cin).

$$SUM = \overline{A\bar{B}}C + \bar{A}B\bar{C} + A\overline{B\bar{C}} + ABC$$
$$CO = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

Using the same process as was used with the HA, the POS equations for the FA were also found.

$$SUM = (A + B + C)(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + C)$$
$$CO = (A + B + C)(A + B + \bar{C})(A + \bar{B} + C)(\bar{A} + B + C)$$

The equations were used to create the schematics for the FA in both SOP and POS forms as shown in Figures *7* and *8*.

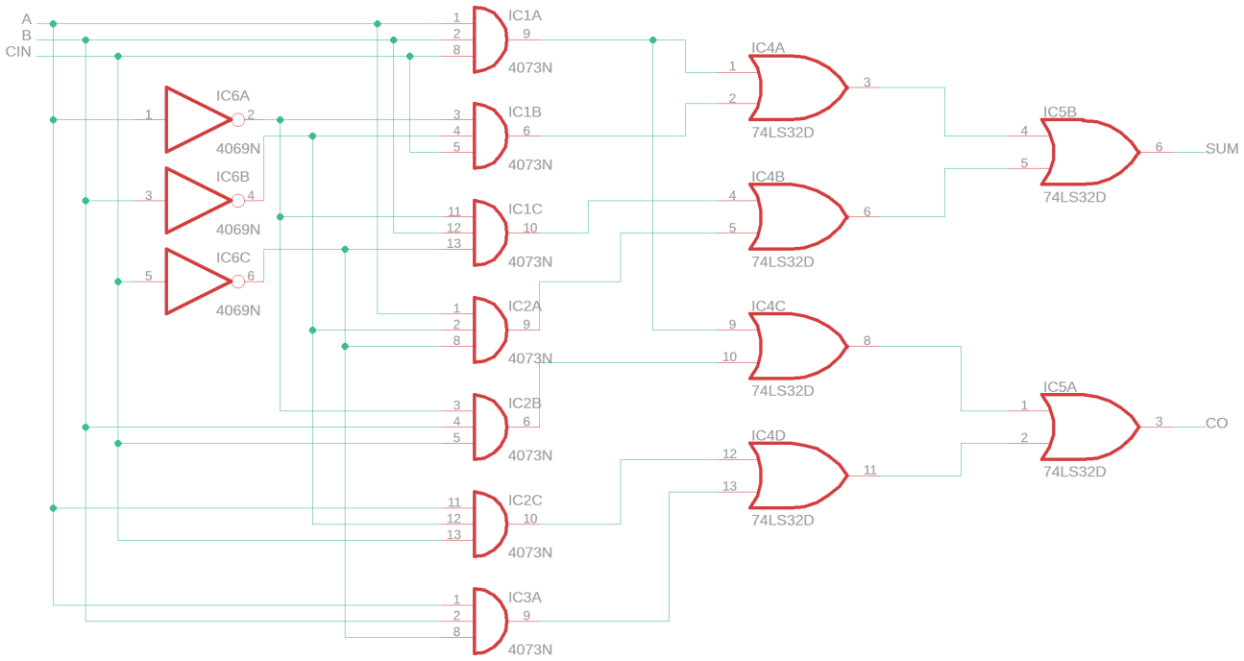


Figure 7 (Above) – Full Adder SOP Schematic

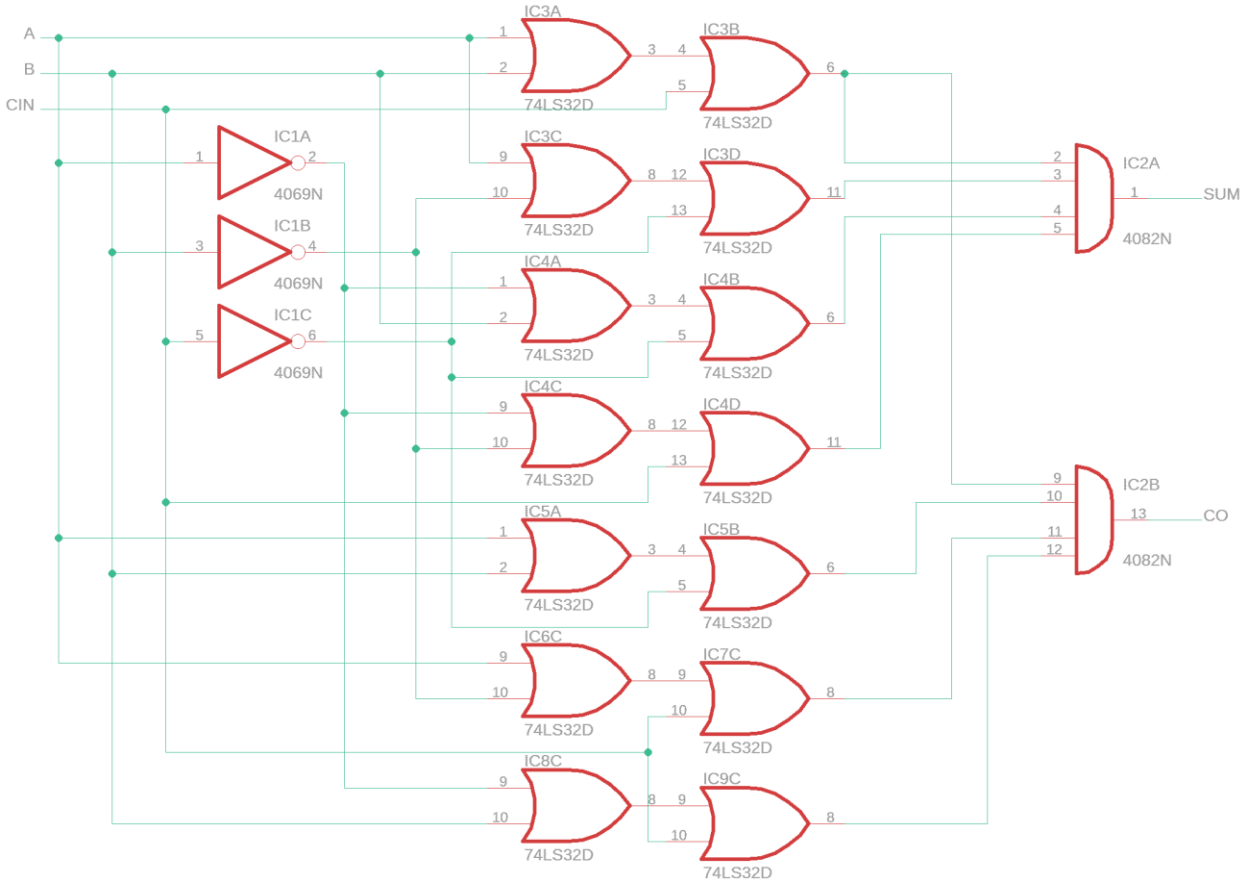


Figure 8 – Full Adder POS Schematic

The equations for the FA were then implemented on the Basys 3 board just as the HA was. A SystemVerilog module, FullAdderSOP, was created first using the SOP equation (see *Appendix G*). A constraints file mapping the inputs and outputs in the module to hardware on the board was then made (see *Appendix H*). The same process was repeated for the POS form of the FA (see *Appendix J* and *Appendix K*). As with the HA, the POS version of the source code also implemented the SOP equations to demonstrate their equivalency.

Finally, the HA and FA designs were combined to build a 5-bit ripple carry adder. The design process for the RCA was different from the brute force design process of the HA and FA. To represent all combinations of adding two 5-bit numbers in a truth table would be very inefficient. Instead, the RCA was formed by using a combination of 1 half adder and 4 full adders, with the CO output of one connected to the Cin input of the next. The high-level black box diagram of the RCA is shown in *Figure 9*, and the lower-level diagram, consisting of the internal HA and FA modules, is shown in *Figure 10*.

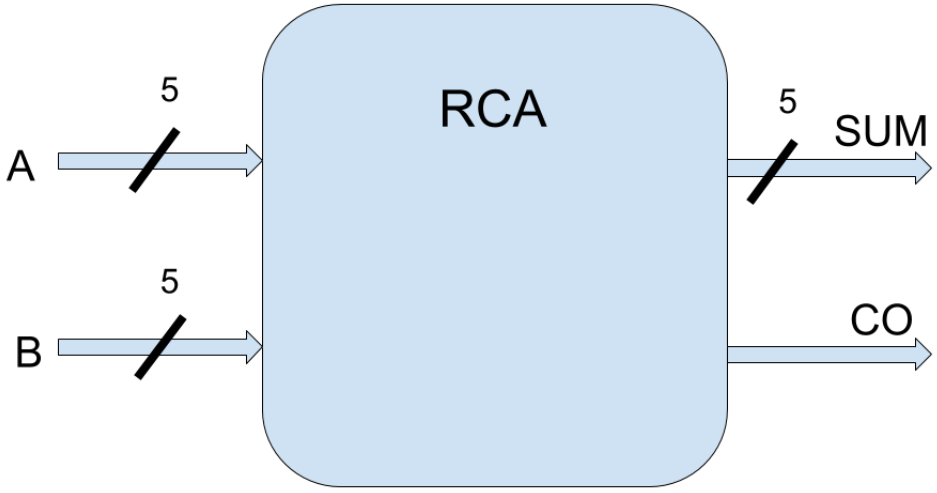


Figure 9 (Above) – Ripple Carry Adder Black Box Diagram (High Level)

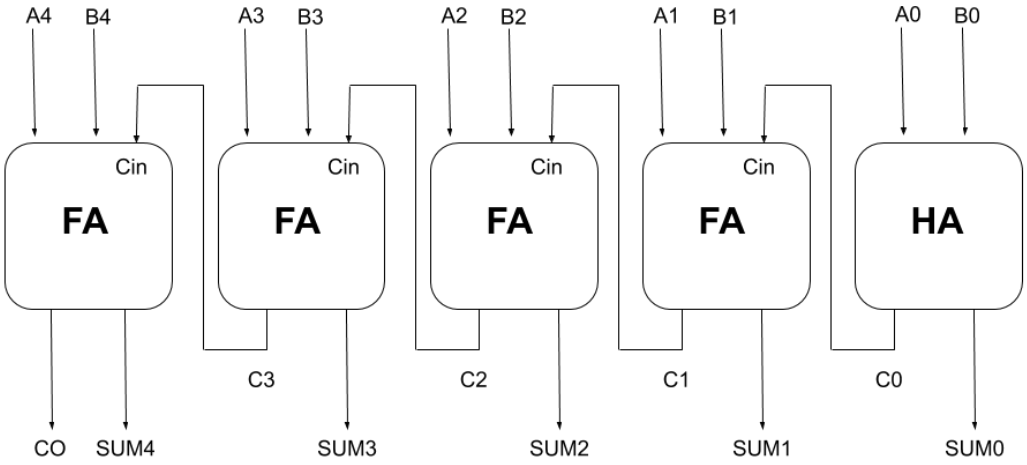


Figure 10 – Ripple Carry Adder Diagram (Lower Level)

Instead of generating equations from a truth table as was done with both the HA and FA, the RCA uses a hierarchical design by combining the HA and FA, so the next implementation step was to create a new SystemVerilog module, RippleCarryAdder (see *Appendix M*). This module uses the HalfAdderSOP and FullAdderSOP modules (see *Appendix A* and *Appendix G*) to incrementally change each bit of the 4-bit inputs A and B. The final FA in the sequence generates the CO output bit.

Testing

To test each adder module, a separate SystemVerilog file was written to simulate all possible combinations of inputs and their corresponding outputs. For simpler modules, like HalfAdderSOP, each possible combination of bits could be written by hand, however, for modules with many more combinations, like RippleCarryAdder, other techniques were used to generate all combinations. See *Appendix C, F, I, L,* and *O* for all simulation source code of all modules. The result of each simulation run is shown in *Figures 11-16*.

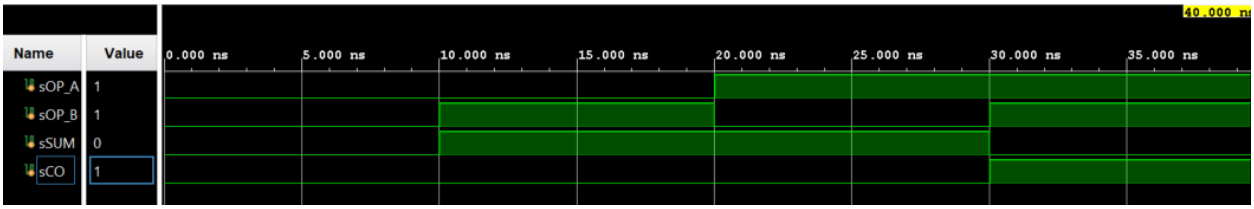


Figure 11 – HalfAdderSOP Simulation

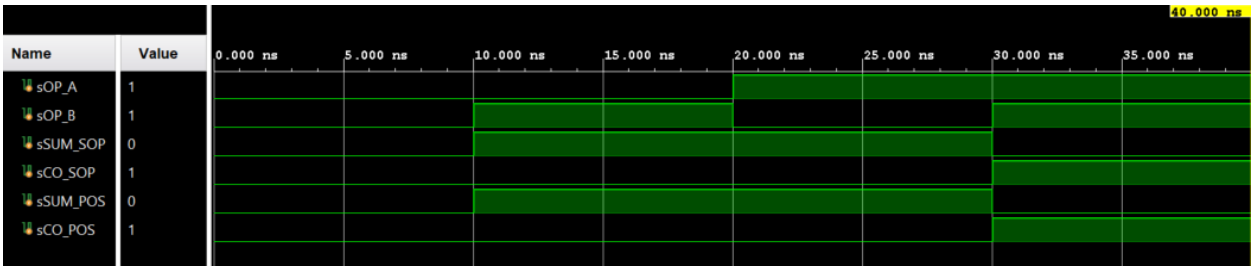


Figure 12 – HalfAdderPOS Simulation

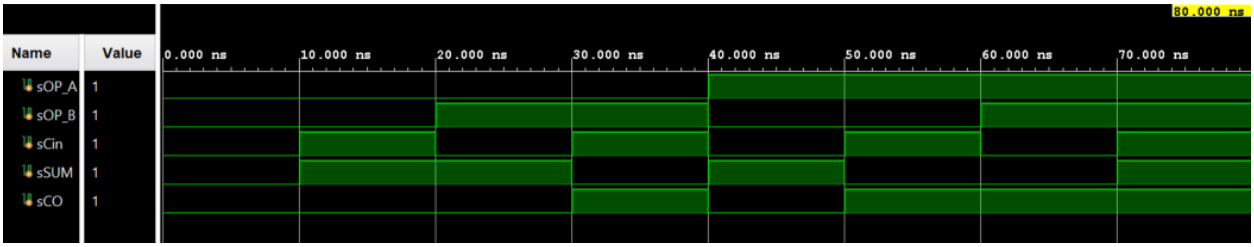


Figure 13 – FullAdderSOP Simulation

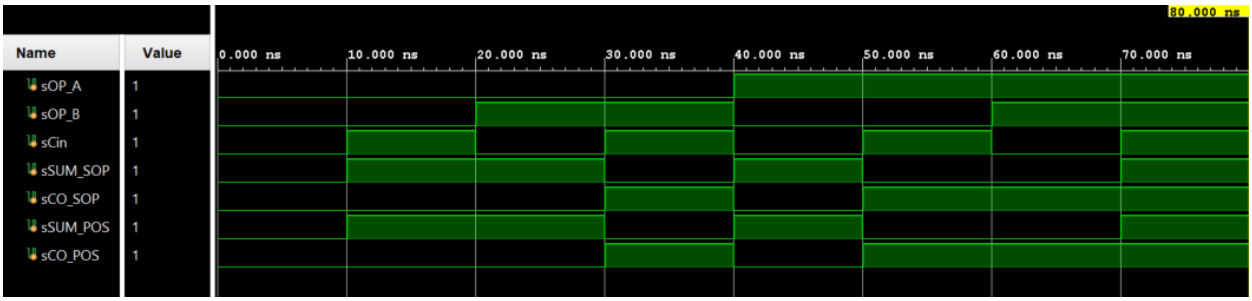


Figure 14 – FullAdderPOS Simulation

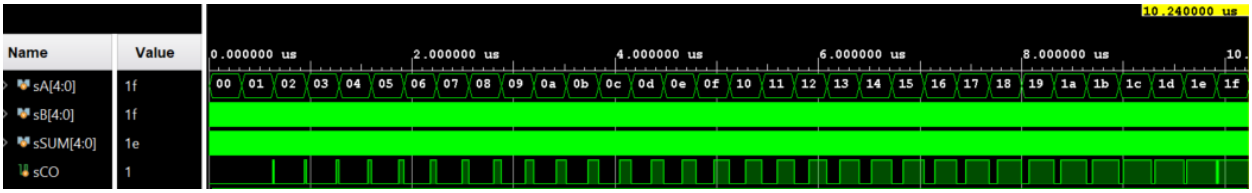


Figure 15 – RippleCarryAdder Simulation (Broad)

The simulation for RippleCarryAdder has too many combinations to see them all without zooming in. *Figure 15* shows the simulation in its entirety, while *Figure 16* shows it zoomed in to a particular value of A (0x0b in this case).

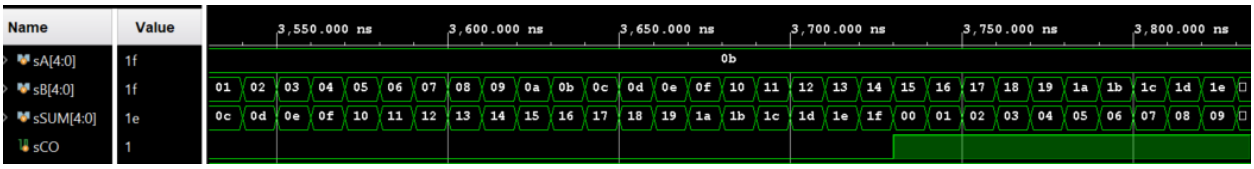


Figure 16 – RippleCarryAdder Simulation (Zoomed)

After simulating each module, it was uploaded to the Basys 3 board for further manual testing.

Conclusion

In this lab, half adder and full adder circuits were designed and implemented on a Basys 3 board using a brute force design method. Each of these circuits was shown to operate the same when written in a sum-of-products form as in a product-of-sums form. The SystemVerilog modules that implemented these circuits in SOP form were then used to design a 5-bit ripple carry adder using an iterative modular design method. After designing each adder module, a separate SystemVeriliog program was written to simulate all possible input combinations, which was used to verify correct operation.

Questions

In your own words, what is meant by the term "methodology"?

The term "methodology", in the case of this lab, refers to the process that was used to design and test a specific circuit. For example, the HA and FA modules used a brute force methodology, in which all possible combinations of inputs and outputs were written in a truth table, from which an equation was

derived. The RCA module, on the other hand, used an iterative modular design methodology. There were far too many input and output combinations to write a truth table and derive an equation from it, so the previous HA and FA modules were used iteratively to design the RCA module.

Briefly describe the purpose of the "constraints file", which is another name for the file with the ".xdc" extension.

The constraints file is essentially a mapping of input and output variables in the SystemVerilog code to physical hardware on the Basys 3 board. In this lab, input variables were mapped to different switches, while outputs were mapped to LEDs.

Briefly describe what you're doing when you "synthesize" your HDL model.

The synthesis of an HDL model is the process used to determine which gates on the FPGA board should be connected. We don't have to worry about physically connecting a circuit together because the board already has all the hardware we need to run our programs. The board needs to know which pieces of hardware to use, though, and that is what the HDL synthesis determines.

There is a programmable logic device (PLD) on the development board used for this class. What particular type of PLD is on the dev board?

The PLD on the Basys 3 board is a field programmable gate array (FPGA). The specific FPGA on the board is the Artix-7 chip.

How many assignment statements did you use in your HDL model for the HA and FA?

Two assignment statements were used for the HA and FA modules. One for the SUM, and one for the CO.

Briefly describe the basic limitation of the HA (relative to the FA) in the context of a mathematical digital circuit.

The HA is limited relative to the FA in that it has no carry-in. While this may not matter for 1-bit addition, this operation is not very useful. Ideally, an adder circuit should be able to add large numbers. The FA can be used iteratively, with the CO of one connected to the Cin of the next, to add very large numbers. The HA is unable to do the same.

Briefly described how you verified your circuit in this experiment was working properly.

The circuits in this experiment were verified by creating simulations for them by showing all possible inputs and outputs, as well as manual testing. For the HA and FA, manual testing was enough (although simulations were created for these as well). For the RCA, there are too many combinations to test all manually, so the testing process relied more heavily on the simulation of this module.

Briefly describe the main purpose of a "gate" in the context of this course.

A "gate", in the context of this course, is to give us an appropriate model to use for designing our digital circuits. Our circuits could be modeled by something like transistors instead of gates, but that model would be too low-level for the types of circuits we are designing. A higher-level model than gates may also work in some cases, but it would not give enough information to be helpful.

Since you now know there are many ways to implement digital circuits, describe some possible parameters involved in ascertaining the "best" way.

Looking specifically at the SOP vs. POS implementations of the HA and FA, it is clear that there are many ways to implement the same circuit with different equations. The "best" way will likely be different for every circuit. Some equations can be simplified better than others. Generally, it would probably be best to find the equation that can be simplified to use the least amount of hardware possible, to reduce cost.

Briefly comment on which form (SOP or POS) of the HA & FA was easier to design and implement. Provide a brief justification for your answer.

The SOP form was much easier to implement, especially for the HA. There were fewer total terms for the HA in SOP form than in POS form. For the FA, even though the amount of terms was the same between SOP and POS forms, it was still easier to design and implement the SOP form because the truth table is essentially already set up to give that form without much thought. The POS form takes more steps, including inverting the outputs and using DeMorgan's theorem.

Briefly describe how the full adder is somewhat limited in doing math operations.

The full adder is limited because it can only add single bit numbers, which is not very useful. In the same respect, however, it can be used iteratively to add numbers of any number of bits.

Briefly describe how you could configure the two full adders to become a two-bit adder.

Two full adders could be used to add two-bit numbers by connecting the carry out of the first to the carry in of the second. The carry bit will be carried through both operations.

Generate a table that compares the number of AND gates, OR gates, and inverters to implement the FA in both SOP and POS forms. Show your equations for this problem, meaning you should have a total of four equations. Draw the final circuit for these two functions also.

FA SOP Equations:

$$SUM = \overline{AB}C + \bar{A}B\bar{C} + A\overline{BC} + ABC$$
$$CO = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

FA POS Equations:

$$SUM = (A + B + C)(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + C)$$
$$CO = (A + B + C)(A + B + \bar{C})(A + \bar{B} + C)(\bar{A} + B + C)$$

Using strictly 2-input gates:

Type	AND	OR	INV
SOP	14	6	3
POS	6	14	3

The schematic for the SOP form can be seen in *Figure 7* and the POS form in *Figure 8*. Note that some AND gates in the schematics have 3 or 4 inputs rather than 2, which reduces the number of gates needed.

Briefly describe how you verified your circuit in this experiment was working properly.

The SOP and POS forms of both the HA and FA were verified to be working properly by creating a simulation as well as manual testing. The outputs SUM and CO in the simulations were displayed next to each other, so it was easy to verify that they were the same.

Write a "conclusion" for this experiment. Note that a conclusion describes what the point of the experiment was.

The purpose of comparing the SOP and POS forms of the HA and FA was to demonstrate that although they are completely different equations, they produce the same result. This equality gives the option to choose the most simple implementation of the circuit we are designing.

Briefly describe the two main attributes of modern digital design.

Digital design is modular (HA module and FA module) and hierarchical (RCA is a level above HA and FA).

Briefly describe why is it a good idea to avoid modifying previously designed modules in your new design?

It is a good idea to avoid modifying previously designed modules in a new design because if the previous modules have been proven to work and have been well-tested, modifying them may introduce unwanted errors. If tempted to change it due to unsolved problems in the new design, it would be better to look at the new design because that is most likely where the problem is.

In your own words, briefly but completely explain why we refer to the circuit in this lab activity to as a "ripple carry adder".

It is referred to as a "ripple carry adder" because the "carry bit" "ripples" through all addition operations due to the carry out of one adder being connected to the carry in of the next. The final CO is then available at the end.

If you needed to extend the RCA from this lab activity to a 10-bit RCA by using a structural model with two 5-bit RCAs, what changes would you need to apply to the 5-bit RCA?

The LSB in the 5-bit RCA would no longer be able to be a HA. The second 5-bit RCA in the 10-bit RCA (the one that would handle the upper 5 bits) would run into problems because its HA would not be able to take carry in input from the previous FA.

How many rows were there be in a truth table for a 32-bit RCA? Would it be feasible to design a 32-bit RCA using a truth table?

There would be a total of 64 inputs, so the total number of input combinations would be 2^64. Using a truth table to design this RCA would be extremely inefficient.

How many logic gates would it require to implement the 5-bit RCA using discrete logic? For this problem assume the logic is in standard SOP and the LSB uses a HA.

In standard SOP form with a HA for the LSB, and assuming all gates are 2-input gates, the total number of gates to implement the 5-bit RCA would be 98. The HA requires a total of 6 gates, while each FA requires 23. There are 4 FAs and 1 HA, so (23 * 4) + 6 = 98 gates in total.

Write a formula in closed for that describes the number of gates in a RCA as a function of the bitwidth of the RCA. Recall that the LSB of the RCA is a HA. For this question, assume the FA and HA are in reduced form (not in standard SOP form). Feel free to ask the instructor for the "reduced" form of both the HA and FA.

Reduced form HA:

$$SUM = A \oplus B$$
$$CO = AB$$

Reduced form FA:

$$SUM = (A \oplus B) \oplus C$$
$$CO = A(B + C) + BC$$

The HA requires 2 gates and the FA requires 6. Thus the total gates will be $6(bitwidth - 1) + 2$

For a RCA, the result could be available immediately, or the result could be delayed. Describe a case where the result is available immediately and also describe a case where the delay is the "worst case". State how long the worst case is in terms of "gate delays".

The result would be available immediately if there is no carry out in any of the bits. This is the best case scenario. In the worse case, every addition operation produces a carry out, in which case you would have to wait for the carry bit to pass through every adder for the correct result to be available.

Briefly describe the notion of concurrency in digital circuit design.

Concurrency is the notion of multiple events occurring at the same time. In the RCA example, if there is no carry, then all bits will be correct concurrently. If there is a carry bit in all adders though, then none of the bits will be available at the same time because we have to wait for the carry bit to pass through all adders.

Do the various module instantiations in a HDL model operate in a concurrent manner? Briefly explain why or why not.

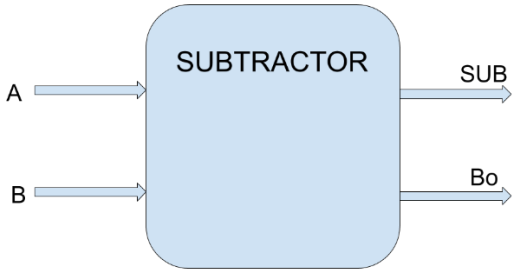
According to the textbook, yes: "Concurrency is one of the underlying factors in VHDL in that many of the statements in VHDL are interpreted as being concurrent in that they can describe multiple hardware entities that work in parallel, and thus supporting the concept of parallelism." It would make sense that module instantiations operate concurrently because otherwise we would not be able to get the result from the RCA immediately.

Consider two different HDL models that are functionally equivalent; in particular, they are mostly the same except for the notion that one used structural modeling and the other implemented a similar set of modules but did not use structural modeling. Would expect the synthesized circuit based on these models to use the equivalent amount of resources or will one approach use more or less resources. Support your answer with intelligent commentary.

I would assume that the synthesized circuit that used a structural model would use less resources than the one that did not use structural modeling. It makes sense that upon synthesizing, the software doing the synthesizing would look for ways to minimize the amount of hardware that is needed. Circuits that implement other modules would probably be able to be minimized more easily because it is clear which parts of hardware may be reused among different modules or multiple instances of the same module, whereas a design that does not use structural modeling would be harder to minimize hardware use because there is not a clearly defined structure.

Design Problems

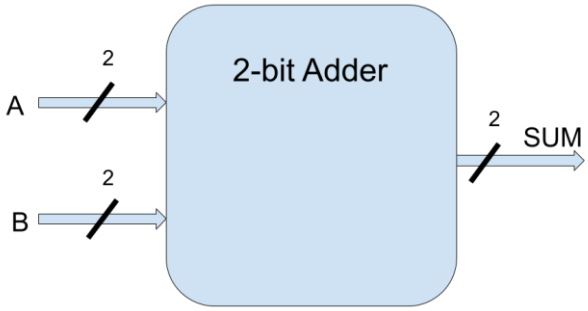
1-bit Subtractor



A	B	SUB	Bo
0	0	0	0
0	1	0	1
1	0	1	0
1	1	0	0

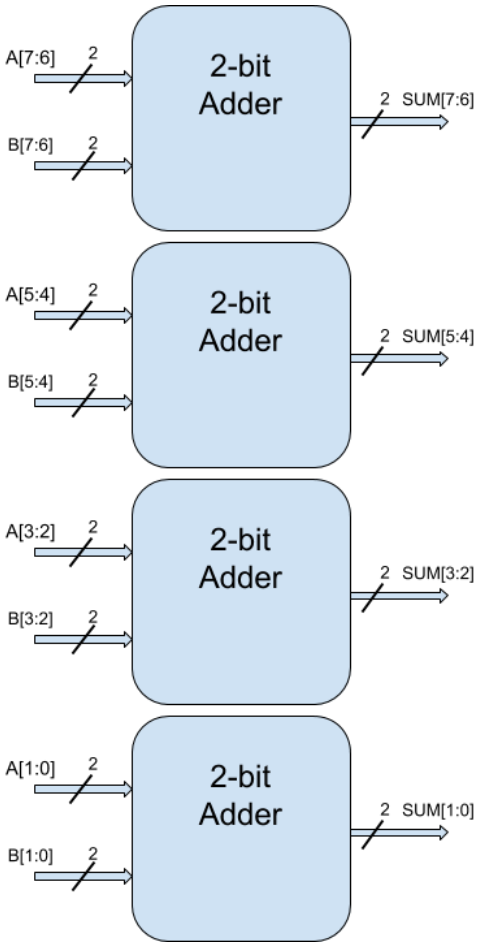
$$SUB = A\bar{B}$$
$$Bo = \bar{A}B$$

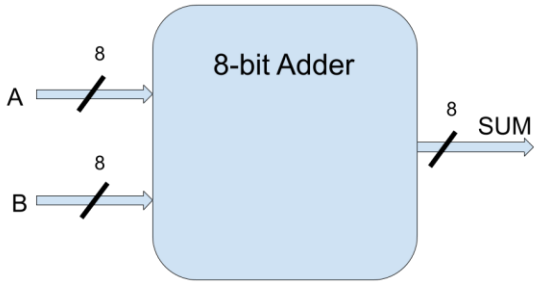
2-Bit "Modulo-2" Adder



A	B	SUM
00	00	00
00	01	01
00	10	10
00	11	11
01	00	01
01	01	00
01	10	11
01	11	10
10	00	10
10	01	11
10	10	00
10	11	01
11	00	11
11	01	10
11	10	01
11	11	00

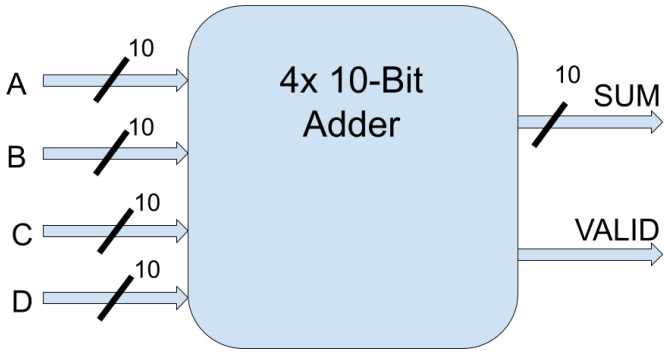
8-Bit "Modulo-2" Adder



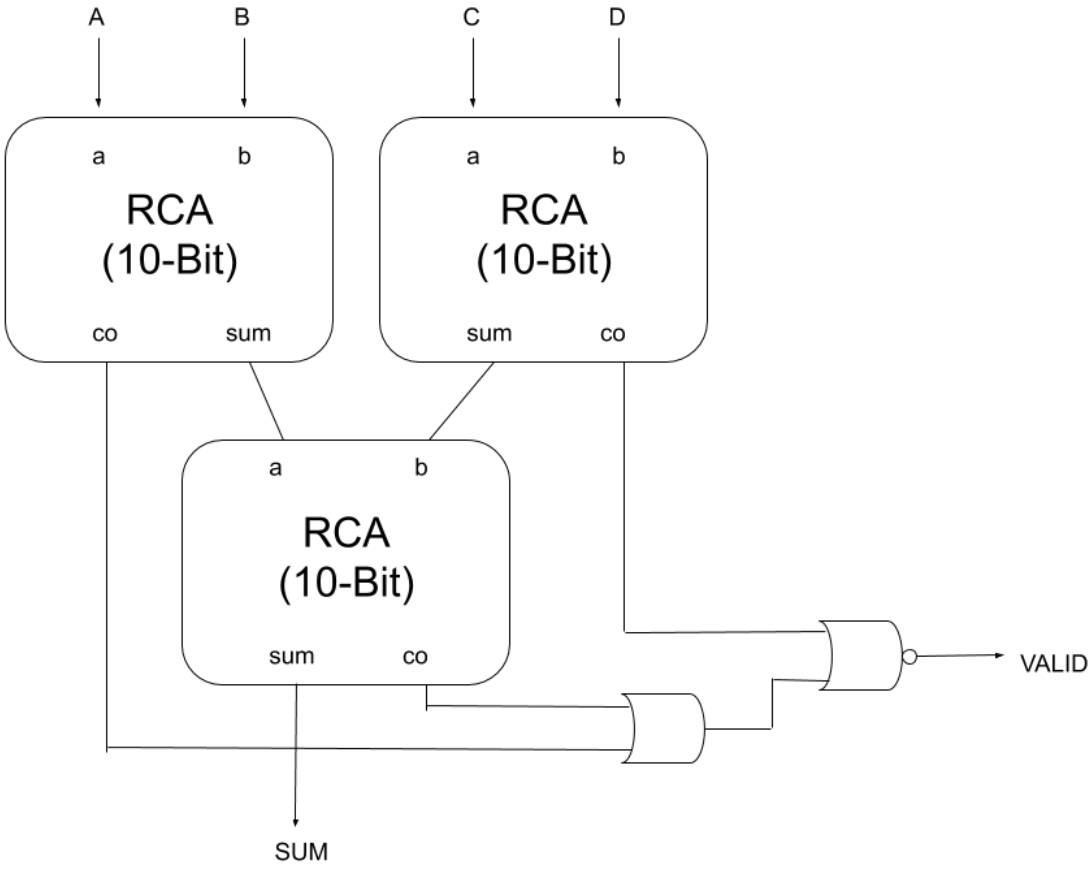


Four 10-bit Adder

High level overview black box diagram:



Lower level diagram, including 3 ripple carry adders, an OR gate, and a NOR gate:



The first part of the circuit is two 10-bit ripple carry adders. Two of the inputs go to one of the RCAs and the other two go to the other RCA. The results from these operations are then added together in the third RCA. The result of this RCA is the final 10-bit SUM output. The carry outs from two of the RCAs are connected to an OR gate, and the result of this gate is connected with the carry out of the remaining RCA as inputs to a NOR gate. Without the NOR (if it were only an OR), the VALID output would only turn on if the operation is invalid. Thus, the NOR inverts it to be correct according to the name of the output.

Appendix

A: Half adder SystemVerilog module source code (standard SOP form) – HalfAdderSOP.sv

```
`timescale 1ns / 1ps

module HalfAdderSOP(
    input OP_A,
    input OP_B,
    output SUM,
    output CO
    );

    assign SUM = (~OP_A & OP_B) | (OP_A & ~OP_B);
    assign CO = OP_A & OP_B;

endmodule
```

B: Half adder constraints file (standard SOP form) – HalfAdderSOP_Constraints.xdc

```
set_property PACKAGE_PIN R2 [get_ports {OP_A}]
set_property IOSTANDARD LVCMOS33 [get_ports {OP_A}]

set_property PACKAGE_PIN T1 [get_ports {OP_B}]
set_property IOSTANDARD LVCMOS33 [get_ports {OP_B}]

set_property PACKAGE_PIN L1 [get_ports {SUM}]
set_property IOSTANDARD LVCMOS33 [get_ports {SUM}]

set_property PACKAGE_PIN U16 [get_ports {CO}]
set_property IOSTANDARD LVCMOS33 [get_ports {CO}]
```

C: Half adder simulation source code (standard SOP form) – HalfAdderSOP_Simulation.sv

```
`timescale 1ns / 1ps

module HalfAdderSOP_Simulation();

    logic sOP_A, sOP_B, sSUM, sCO;

    HalfAdderSOP UUT (
        .OP_A(sOP_A), .OP_B(sOP_B),
        .SUM(sSUM), .CO(sCO));
```

```
    initial begin

        sOP_A = 0; sOP_B = 0;
        #10;

        sOP_B = 1;
        #10;

        sOP_A = 1; sOP_B = 0;
        #10;

        sOP_B = 1;
        #10;

    end

endmodule
```

D: Half adder SystemVerilog module source code (standard POS form) – HalfAdderPOS.sv

```
`timescale 1ns / 1ps

module HalfAdderPOS(
    input OP_A,
    input OP_B,
    output SUM_SOP,
    output CO_SOP,
    output SUM_POS,
    output CO_POS
    );

    // SOP equations
    assign SUM_SOP = (~OP_A & OP_B) | (OP_A & ~OP_B);
    assign CO_SOP = OP_A & OP_B;

    // POS equations
    assign SUM_POS = (OP_A | OP_B) & (~OP_A | ~OP_B);
    assign CO_POS = (OP_A | OP_B) & (OP_A | ~OP_B) & (~OP_A | OP_B);

endmodule
```

E: Half adder constraints file (standard POS form) – HalfAdderPOS_Constraints.xdc

```
set_property PACKAGE_PIN R2 [get_ports {OP_A}]
set_property IOSTANDARD LVCMOS33 [get_ports {OP_A}]

set_property PACKAGE_PIN T1 [get_ports {OP_B}]
set_property IOSTANDARD LVCMOS33 [get_ports {OP_B}]

set_property PACKAGE_PIN L1 [get_ports {SUM_SOP}]
set_property IOSTANDARD LVCMOS33 [get_ports {SUM_SOP}]

set_property PACKAGE_PIN E19 [get_ports {SUM_POS}]
set_property IOSTANDARD LVCMOS33 [get_ports {SUM_POS}]

set_property PACKAGE_PIN P1 [get_ports {CO_SOP}]
set_property IOSTANDARD LVCMOS33 [get_ports {CO_SOP}]

set_property PACKAGE_PIN U16 [get_ports {CO_POS}]
set_property IOSTANDARD LVCMOS33 [get_ports {CO_POS}]
```

F: Half adder simulation source code (standard POS form) – HalfAdderPOS_Simulation.sv

```
`timescale 1ns / 1ps

module HalfAdderPOS_Simulation();

    logic sOP_A, sOP_B, sSUM_SOP, sCO_SOP, sSUM_POS, sCO_POS;

    HalfAdderPOS UUT (
        .OP_A(sOP_A), .OP_B(sOP_B), .SUM_SOP(sSUM_SOP),
        .SUM_POS(sSUM_POS), .CO_SOP(sCO_SOP), .CO_POS(sCO_POS));

    initial begin

        sOP_A = 0; sOP_B = 0;
        #10;

        sOP_B = 1;
        #10;

        sOP_A = 1; sOP_B = 0;
        #10;
```

```
        sOP_B = 1;
        #10;

    end

endmodule
```

G: Full adder SystemVerilog module source code (standard SOP form) – FullAdderSOP.sv

```
`timescale 1ns / 1ps

module FullAdderSOP(
    input OP_A,
    input OP_B,
    input Cin,
    output SUM,
    output CO
    );

    assign SUM = (~OP_A & ~OP_B & Cin) | (~OP_A & OP_B & ~Cin) |
                 (OP_A & ~OP_B & ~Cin) | (OP_A & OP_B & Cin);
    assign CO = (~OP_A & OP_B & Cin) | (OP_A & ~OP_B & Cin) |
                (OP_A & OP_B & ~Cin) | (OP_A & OP_B & Cin);

endmodule
```

H: Full adder constraints file (standard SOP form) – FullAdderSOP_Constraints.xdc

```
set_property PACKAGE_PIN V17 [get_ports {OP_A}]
set_property IOSTANDARD LVCMOS33 [get_ports {OP_A}]

set_property PACKAGE_PIN V16 [get_ports {OP_B}]
set_property IOSTANDARD LVCMOS33 [get_ports {OP_B}]

set_property PACKAGE_PIN W16 [get_ports {Cin}]
set_property IOSTANDARD LVCMOS33 [get_ports {Cin}]

set_property PACKAGE_PIN L1 [get_ports {SUM}]
set_property IOSTANDARD LVCMOS33 [get_ports {SUM}]

set_property PACKAGE_PIN U16 [get_ports {CO}]
set_property IOSTANDARD LVCMOS33 [get_ports {CO}]
```

I: Full adder simulation source code (standard SOP form) – FullAdderSOP_Simulation.sv

```
`timescale 1ns / 1ps

module FullAdderSOP_Simulation();

    logic sOP_A, sOP_B, sCin, sSUM, sCO;

    FullAdderSOP UUT (
        .OP_A(sOP_A), .OP_B(sOP_B),
        .Cin(sCin), .SUM(sSUM), .CO(sCO));

    initial begin

        // Loop through all combinations instead of writing one by one
        for (byte a = 0; a < 2; a++) begin
            for (byte b = 0; b < 2; b++) begin
                for (byte cin = 0; cin < 2; cin++) begin
                    sOP_A = a; sOP_B = b;
                    sCin = cin;
                    #10;
                end
            end
        end

    end

endmodule
```

J: Full adder SystemVerilog module source code (standard POS form) – FullAdderPOS.sv

```
`timescale 1ns / 1ps

module FullAdderPOS(
    input Cin,
    input OP_A,
    input OP_B,
    output CO_SOP,
    output SUM_SOP,
    output CO_POS,
```

```
    output SUM_POS
    );

    // SOP Equations
    assign SUM_SOP = (~OP_A & ~OP_B & Cin) | (~OP_A & OP_B & ~Cin) |
                (OP_A & ~OP_B & ~Cin) | (OP_A & OP_B & Cin);
    assign CO_SOP = (~OP_A & OP_B & Cin) | (OP_A & ~OP_B & Cin) |
                (OP_A & OP_B & ~Cin) | (OP_A & OP_B & Cin);

    // POS equations
    assign SUM_POS = (OP_A | OP_B | Cin) & (OP_A | ~OP_B | ~Cin) &
                     (~OP_A | OP_B | ~Cin) & (~OP_A | ~OP_B | Cin);
    assign CO_POS = (OP_A | OP_B | Cin) & (OP_A | OP_B | ~Cin) &
                    (OP_A | ~OP_B | Cin) & (~OP_A | OP_B | Cin);

endmodule
```

K: Full adder constraints file (standard POS form) – FullAdderPOS_Constraints.xdc

```
set_property PACKAGE_PIN V16 [get_ports {OP_A}]
set_property IOSTANDARD LVCMOS33 [get_ports {OP_A}]

set_property PACKAGE_PIN V17 [get_ports {OP_B}]
set_property IOSTANDARD LVCMOS33 [get_ports {OP_B}]

set_property PACKAGE_PIN W16 [get_ports {Cin}]
set_property IOSTANDARD LVCMOS33 [get_ports {Cin}]

set_property PACKAGE_PIN P1 [get_ports {SUM_SOP}]
set_property IOSTANDARD LVCMOS33 [get_ports {SUM_SOP}]

set_property PACKAGE_PIN L1 [get_ports {CO_SOP}]
set_property IOSTANDARD LVCMOS33 [get_ports {CO_SOP}]

set_property PACKAGE_PIN U16 [get_ports {SUM_POS}]
set_property IOSTANDARD LVCMOS33 [get_ports {SUM_POS}]

set_property PACKAGE_PIN E19 [get_ports {CO_POS}]
set_property IOSTANDARD LVCMOS33 [get_ports {CO_POS}]
```

L: Full adder simulation source code (standard POS form) – FullAdderPOS_Simulation.sv

```
`timescale 1ns / 1ps

module FullAdderPOS_Simulation();

    logic sOP_A, sOP_B, sCin, sSUM_SOP, sCO_SOP, sSUM_POS, sCO_POS;

    FullAdderPOS UUT (
        .OP_A(sOP_A), .OP_B(sOP_B), .Cin(sCin), .SUM_SOP(sSUM_SOP),
        .SUM_POS(sSUM_POS), .CO_SOP(sCO_SOP), .CO_POS(sCO_POS));

    initial begin

        // Loop through all combinations instead of writing one by one
        for (byte a = 0; a < 2; a++) begin
            for (byte b = 0; b < 2; b++) begin
                for (byte cin = 0; cin < 2; cin++) begin
                    sOP_A = a; sOP_B = b;
                    sCin = cin;
                    #10;
                end
            end
        end

    end

endmodule
```

M: Ripple carry adder SystemVerilog module source code – RippleCarryAdder.sv

```
`timescale 1ns / 1ps

module RippleCarryAdder(
    input [4:0] A,
    input [4:0] B,
    output [4:0] SUM,
    output CO
    );

    // Make a wire variable to connect the carry out
    // from one adder to the carry in of another
    wire [3:0] Cout;
```

```
    // Use the HA and FA modules to successively change the outputs
    HalfAdderSOP Bit0(A[0], B[0], SUM[0], Cout[0]);
    FullAdderSOP Bit1(A[1], B[1], Cout[0], SUM[1], Cout[1]);
    FullAdderSOP Bit2(A[2], B[2], Cout[1], SUM[2], Cout[2]);
    FullAdderSOP Bit3(A[3], B[3], Cout[2], SUM[3], Cout[3]);
    FullAdderSOP Bit4(A[4], B[4], Cout[3], SUM[4], CO);

endmodule
```

N: Ripple carry adder constraints file – RippleCarryAdder_Constraints.xdc

```
# First set constraints for 5-bit input A
# These will be the 5 leftmost switches

set_property PACKAGE_PIN R3 [get_ports {A[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {A[0]}]

set_property PACKAGE_PIN W2 [get_ports {A[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {A[1]}]

set_property PACKAGE_PIN U1 [get_ports {A[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {A[2]}]

set_property PACKAGE_PIN T1 [get_ports {A[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {A[3]}]

set_property PACKAGE_PIN R2 [get_ports {A[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {A[4]}]

# Next set constraints for 5-bit input B
# These will be the 5 rightmost switches

set_property PACKAGE_PIN V17 [get_ports {B[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {B[0]}]

set_property PACKAGE_PIN V16 [get_ports {B[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {B[1]}]

set_property PACKAGE_PIN W16 [get_ports {B[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {B[2]}]

set_property PACKAGE_PIN W17 [get_ports {B[3]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {B[3]}]

set_property PACKAGE_PIN W15 [get_ports {B[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {B[4]}]

# Now set constraints for SUM output
# This will be the 5 rightmost LEDs

set_property PACKAGE_PIN U16 [get_ports {SUM[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SUM[0]}]

set_property PACKAGE_PIN E19 [get_ports {SUM[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SUM[1]}]

set_property PACKAGE_PIN U19 [get_ports {SUM[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SUM[2]}]

set_property PACKAGE_PIN V19 [get_ports {SUM[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SUM[3]}]

set_property PACKAGE_PIN W18 [get_ports {SUM[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SUM[4]}]

# Finally set carry out (the possible 6th bit)
# It will be on LED7

set_property PACKAGE_PIN V14 [get_ports {CO}]
set_property IOSTANDARD LVCMOS33 [get_ports {CO}]
```

O: Ripple carry adder simulation source code – RippleCarryAdder_Simulation.sv

```
`timescale 1ns / 1ps

module RippleCarryAdder_Simulation();

    // 5-bit inputs and output (include [4:0])
    logic [4:0] sA, sB, sSUM;
    // 1-bit output
    logic sCO;

    RippleCarryAdder UUT(
        .A(sA), .B(sB), .SUM(sSUM), .CO(sCO));
```

```
    // The number of bits in the adder
    localparam bits = 5;
    // Delay time - total runtime will be ((2 ** (2 * bits)) * delay) time units.
    // 10240 ns in this case.
    localparam delay = 10;

    initial begin

        // Use only 2 for loops instead of 10 (1 for each bit of A and B)
        // by assigning the entire variable at one time in this way.
        // It works the same as 10 loops but is much more readable
        // and less messy
        for (byte a = 0; a < 2 ** bits; a++) begin
            for (byte b = 0; b < 2 ** bits; b++) begin
                sA = a[4:0];
                sB = b[4:0];
                #delay;
            end
        end

    end

endmodule
```