

# Digital System Design Applications

---

## Experiment IV ARITHMETIC CIRCUITS 2024

### Preliminary

Students should know about arithmetic circuits.

### Objectives

- To learn how to design arithmetic circuits.
- Usage of testbench code for making simulation

### Requirements

Students are expected to be able to

- define hardwares with Verilog
- synthesize, simulate, implement designs, generate bitstreams and configure FPGA
- create project on Vivado

## Experiment Report Checklist

Each student is going to prepare his/her own report. Reports should include:

1. **Half Adder**
  - Verilog code of Half Adder
  - Testbench code and simulation results of Half Adder
2. **Full Adder**
  - Verilog code of Full Adder
  - Testbench code and simulation results of Full Adder
3. **Ripple Carry Adder**
  - Verilog code of Ripple Carry Adder
  - Testbench code and simulation results of Ripple Carry Adder
4. **Ripple Carry Adder "parametric"**
  - Verilog code of parametric Ripple Carry Adder
  - Testbench code and simulation results of parametric Ripple Carry Adder
  - RTL and Technology schematics
  - Utilization and timing reports
5. **Carry Lookahead Adder "parametric"**
  - Verilog code of Carry Lookahead Adder
  - Test code and simulation results of parametric Carry Lookahead Adder
  - RTL and Technology schematics
  - Utilization and timing reports
6. **Behavioral Adder**
  - Verilog code of Carry Lookahead Adder
  - Test code and simulation results of parametric Behavioral Adder
  - RTL and Technology schematics
  - Utilization and timing reports
  - Comparison of the RCA, CLA and Behavioral adders with and without DONT\_TOUCH attribute.
7. **Adder-Subtractor Circuit**
  - Adder-Subtractor Circuit with Overflow detection Verilog code
  - Test code and results for Adder-Subtractor for all cases
  - RTL and Technology schematics
  - Utilization and timing reports
8. **Research**
  - Research about DSP Block Implementation in Vivado.
  - Research about Fixed-point representation in FPGA Designs.

- **Projects and reports are to be done INDIVIDUALLY. High amounts of points will be deducted from similar works.**
- **Reports must be written in a proper manner. Divide your text to sections and sub-sections if needed, label your figures and connect your sections with proper explanations of your works. Reports filled with imprecisely placed tables and figures, with no verbal explanations in workflow, will not fare well.**
- **Check homeworks section in Ninova for submission dates.**

## Half Adder

1. Add a new source to your projects named **arithmetic\_circuits.v**.
2. Write down a module which is called **HA** into your arithmetic\_circuits.v file. This module is going to have 1-bit inputs **x** and **y** and 1-bit outputs **cout** (carry out) and **sum**. Ensure that your design provides the truth table shown in Figure 1.
3. Write a test code to show that your circuit is working correctly.
4. Synthesize and implement your design.

x	y	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

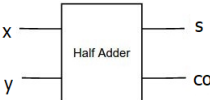


Figure 1: Half Adder Truth Table and block representation

## FULL Adder

1. Write down full adder circuit to a new module called **FA** into your arithmetic\_circuits.v file. This module is going to have 1-bit inputs **x**, **y**, **cin** (carry in) and 1-bit outputs **cout** (carry out) and **sum**.
2. Design your FA module as it contains 2 HA circuits and 1 OR gate. Ensure that your design provides the truth table shown in Figure 2.
3. Write a test code to show that your circuit is working correctly.
4. Synthesize and implement your design.

## Ripple Carry Adder

1. Write down your ripple carry adder circuit to a new module called **RCA** into your arithmetic\_circuits.v file.
2. This module is going to have 4-bit inputs **x**, **y**, 1-bit carry input **cin** and 1-bit output carry out **cout**, 4-bit output **sum**.

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

Figure 2: FULL Adder Truth Table

3. Design your RCA module so that it contains 4 FA circuits as shown in Figure 3. Ensure that your design works correctly.
4. Write a test code to show that your circuit is working correctly.
5. Synthesize and implement your design.

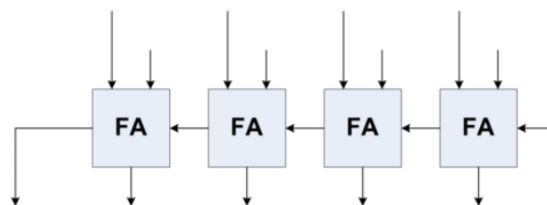


Figure 3: Ripple Carry Adder

### Ripple Carry Adder "parametric"

1. Write down your ripple carry adder circuit to a new module called **parametric\_RCA**. This module should include a parameter named **SIZE**, to represent number length. By only changing the **SIZE** parameter, a user must be able to easily synthesise an adder in a specific length. Thus, define your inputs **x**, **y**; and your output **sum**, with their length depending on **SIZE** parameter. The module also needs to have a one bit **cin** input and one bit **cout** output.
2. Design your RCA module similarly to the previous step. But this time, use **generate-for** statement and **SIZE** parameter to come up with a rule to connect multiple FA's, using Figure 3. How many FA's used should be solely determined by **SIZE** parameter. **generate-for statement** must make the proper connections automatically.
3. Set your size parameter to 8, and test your circuit for six different 8-bit number pairs at least. Show your simulation results clearly.
4. Make your pin configurations for 8-bit adder. Assign inputs to the switches, outputs to the LEDs and carry input to a button. Then synthesize and implement your design.
5. View RTL, Technology schematics, and Design Summary. How many LUTs does your design use? What are the delays of your circuit?
6. Generate **.BIT file** and program your FPGA. Show that your circuit is working correctly.

## Carry Lookahead Adder "parametric"

1. **Carry-lookahead adders (CLA)** calculate carry bits before the sum bits, whereas ripple-carry adders (RCA) calculate the carry bits alongside with the sum bits. In ripple-carry adders, each sum calculation must wait until the previous carry bit has been calculated. On the other hand, in carry-lookahead adders, carry and sum calculations are done independently and parallelly.
2. Carry-lookahead adders use **generating** and **propagating** carries strategy for each digit. If the addition of two 1-bit numbers produces a carry regardless of whether there is an carry input, it can be said that these two numbers **generates** a carry. If the addition of two 1-bit numbers produces a carry whenever there is an carry input, these numbers **propagates** a carry to the next digit. The calculation of the carries (C) of the binary numbers (X, Y) by using generate (G) and propagate (P) concept can be seen as follows:

$$G_i = X_i \cdot Y_i$$

$$P_i = X_i \oplus Y_i$$

$$C_i = G_i + P_i \cdot C_{i-1}$$

3. The calculation of the carries for 4-bit CLA is as follows:

$$C1 = G_0 + P_0 \cdot C_0$$

$$C2 = G_1 + P_1 \cdot C_1$$

$$C3 = G_2 + P_2 \cdot C_2$$

$$C4 = G_3 + P_3 \cdot C_3$$

$$C1 = G_0 + P_0 \cdot C_0$$

$$C2 = G_1 + P_1 \cdot G_0 + P_0 \cdot P_1 \cdot C_0$$

$$C3 = G_2 + P_2 \cdot G_1 + P_1 \cdot P_2 \cdot G_0 + P_0 \cdot P_1 \cdot P_2 \cdot C_0$$

$$C4 = G_3 + P_3 \cdot G_2 + P_2 \cdot P_3 \cdot G_1 + P_1 \cdot P_2 \cdot P_3 \cdot G_0 + P_0 \cdot P_1 \cdot P_2 \cdot P_3 \cdot C_0$$

4. Write down your carry lookahead adder circuit to a new module called CLA. This module should include a parameter named **SIZE**, to represent number length. By only changing the **SIZE** parameter, a user must be able to easily synthesise an adder in a specific length. Thus, define your inputs **x**, **y**; and your output **s**, with their length depending on **SIZE** parameter. The module also needs to have a one bit **cin** input and one bit **cout** output.
5. Define wires with the length of **SIZE** for generate (g), propagate (p) and carries (c). Obtain carries by using p, g and c expressions with assign keywords. Use **generate-for statement** to make the proper connections. Then obtain outputs sum and cout.
6. Set your size parameter to 8 and write a test code to show that your circuit is working correctly. Test your circuit for six different 8-bit number pairs at least.
7. Make your pin configurations for 8-bit adder. Assign inputs to the switches, outputs to the LEDs and carry input to a button. Then synthesize and implement your design.

8. View RTL, Technology schematics, and Design Summary. How many LUTs does your design use? What are the delays of your circuit?
9. Generate **.BIT file** and program your FPGA. Show that your circuit is working correctly.

## Behavioral Adder

1. Write down a behavioral adder by using the **arithmetic operator (+)**. This module should include a parameter named **SIZE**, to represent number length of inputs and the output sum. Thus, define your inputs **x, y**; and your output **sum**, with their length depending on **SIZE** parameter. The module also needs to have one bit **cout** output.
2. Design your circuit by just using the (+) operator and assign keywords to obtain outputs sum and cout.
3. Set your size parameter to 8 and write a test code to show that your circuit is working correctly. Test your circuit for six different 8-bit number pairs at least.
4. View RTL, Technology schematics, and Design Summary. How many LUTs does your design use? What are the delays of your circuit?
5. Compare the behavioral adder with the RCA and CLA adders in terms of resource usage and delays. Explain the similarities and differences of RTL and Technology schematics among them.
6. Add **\*DONT\_TOUCH = "yes"** attribute into the RTL codes of all these designs and re-implement them. In this case, make a comparison again in terms of their resource usage and delays. Explain the similarities and differences of RTL and Technology schematics among them. How does the **DONT\_TOUCH** attribute make an impact on these designs? Explain.

## Adder-Subtractor Circuit with Overflow detection

1. You will design a 4-bit adder-subtractor circuit to add or subtract the **signed binary** numbers. In Verilog, the signed numbers are always represented in two's complement form with a leftmost sign bit. You must design your circuit so that it detects the arithmetic overflow.
2. When two numbers with  $n$  digits each are added and the sum is a number with  $n + 1$  digits, it is said that an overflow occurred. For unsigned numbers, an overflow is detected from the carry out (the most significant position). For signed numbers, the sign bit is treated as a part of the number, thus the carry out does not indicate an overflow. An overflow cannot occur when addition of the numbers having opposite signs, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two input numbers. An overflow can occur if the two numbers added are both positive or both negative.
3. If the last two carries of the adder/subtractor are applied to an exclusive-OR gate, an overflow can be detected.
4. The circuit should be design as given in Figure 4. Write down a module which is called **Add\_Sub**. This module is going to have 4-bit inputs **A** and **B**, 1-bit carry input **cin**, 4-bit output **sum**, 1-bit output **cout** and 1-bit output **overflow**. Obtain the circuit using FA's from your arithmetic\_circuits.v file.

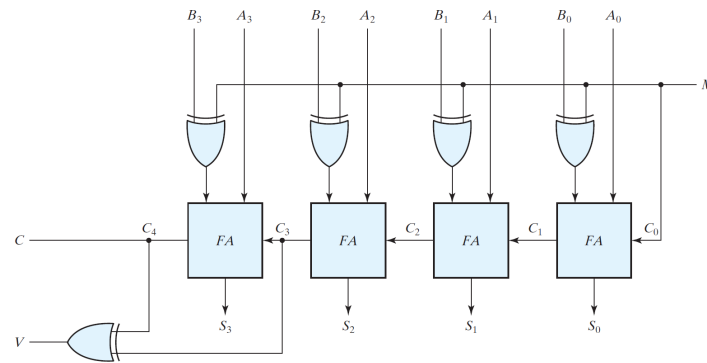


Figure 4: The 4-bit Adder Subtractor Circuit with Overflow Detector

5. Write a test code to show that your circuit is working correctly. In your test code:
  - Change the value of A and B inputs from -8 to +7.
  - Change the value of cin from 0 to 1.
  - Show all of the cases.
  - Show the carry outputs of the last two FA on simulation so that it can be understood if overflow signal is working as expected.
6. Make your pin configurations. Assign inputs to the switches, outputs to the LEDs and carry input to a button. Then synthesize and implement your design.
7. View RTL and Technology schematics. Obtain utilization and timing reports.
8. Generate **.BIT file** and program your FPGA. Show that your circuit is working correctly.

## Research

1. Make a research about the **DSP Block Implementation** in Vivado (from Xilinx User Guides). How the DSP blocks are inferred (synthesized)? Write an RTL code example. What is the advantage of DSP block implementation in comparison with the slice logic implementation? Explain.
2. Make a research about the **Fixed-point Representation** in FPGA Designs. How the fixed-point numbers are represented? What are the advantages of the fixed-point arithmetic over the floating-point arithmetic in FPGA Designs? Explain.

## References:

1. Digital Design, 5th Edition By M. Morris Mano And Michael Ciletti
2. Nexys4DDR Reference Manual
3. Constraints Guide