CRN: 22840

CET3136 – Logic Devices Programming

Spring 2025

Experiment 5

Numbers and Displays, Part 2

Performed By:

Anthony Paul Sevarino

Submitted to:

Prof. Ashley Evans

Department of Electrical & Computer Engineering Technology (ECET)

School of Engineering, Technology, and Advanced Manufacturing (ETAM)

Valencia College

Date Submitted

02/12/2025

Introduction

The objective of this lab is to create a four-bit ripple carry adder, using the switches and LEDs on the MAX10 development board. While the experiment statement explains that any coding style may be used to perform this task, it specifies that the program should be written as efficiently as possible. Thus, a combination of dataflow and structural coding styles will be used. The use of Karnaugh mapping will also assist in the design of the individual full-adder segment to be reused for each component of the four-bit full adder. Note that the fundamental function of a four-bit ripple carry adder is to add 2 four-bit binary values together, with an optional carry value added when desired. The result should display as a four bit binary value, with a fifth bit reserved for the carry value (overflow).

LIST OF EQUIPMENT/PARTS/COMPONENTS/SOFTWARE

- DE10-Lite MAX-10 Dev Board
- USB 2.0 Type B Cable
- Windows Desktop
- Quartus FPGA Design Software

PROCEDURE / DISCUSSION

Part 1 – The Full-Adder

First, creating a logic schematic for a full adder circuit will aid in designing the VHDL program efficiently and correctly. Below us such a schematic, provided by the Laboratory Exercise Manual[1].

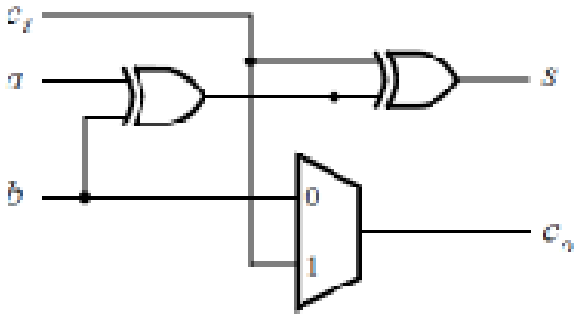

Figure 1 - Logic Diagram for a Full-Adder

A dataflow coding style may be most efficient for this component, however making a truth table, along with Karnaugh mapping it's logic will help determine whether or not this may be the case. Ensure that within this dataflow "building block" so-to-speak, there are three scalar inputs (A, B, and C_i) as well as two scalar outputs (C_o and S). Below is a created truth table and resulting Karnaugh maps for the system:

Table 1 – Diagram Truth Table and Karnaugh Maps

B	A	Ci	Co	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Co	Ci'	Ci
B'A'	0	0
BA'	0	1
BA	1	1
B'A	0	1

S	Ci'	Ci
B'A'	0	1
BA'	1	0
BA	0	1
B'A	1	0

Using the above Karnaugh mappings, the resulting logical Boolean statements can be derived and implemented into the VHDL program:

$$C_o <= (B\ and\ A)\ or\ (C_i\ and\ A)\ or\ (C_i\ and\ B);$$
$$S <= (not\ Ci\ and\ not\ B\ and\ A) or (not\ Ci\ and\ B\ and\ not\ A) or (Ci\ and\ not\ B\ and\ not\ A) or (Ci\ and\ B\ and\ A);$$

After implementing these logic statements in the dataflow-based *fulladder* block within the program, the *fourbitfulladder* main block can be created using the structural coding style.

Code Explanation Part 1

```
37  ----------------------------------------------------------------------------
38  ----------------------------Full Adder Block--------------------------------
39
40   library ieee;
41   use ieee.std_logic_1164.all;
42
43  entity fulladder is
44  port  (
45            A, B, Ci : in std_logic;      --adder inputs
46            Co, S : out std_logic         --adder outputs
47          );
48   end fulladder;
49
50  architecture dataflow of fulladder is
51
52  begin
53
54        Co <= (B and A) or (Ci and A) or (Ci and B);    --carry output logic
55        S <= (not Ci and not B and A) or (not Ci and B and not A) or (Ci and not B and not A) or (Ci and B and A);  --sum output logic
56
57   end dataflow;
58
59  ----------------------------Full Adder Block--------------------------------
60  ----------------------------------------------------------------------------
```

Figure 2 - Completed Code for fulladder

```
----------------------------------------------------------------------------
----------------------------Full Adder Block----------------------------

library ieee;
use ieee.std_logic_1164.all;

entity fulladder is
port	(
				A, B, Ci : in std_logic;		--adder inputs
				Co, S : out std_logic			--adder outputs
			);
end fulladder;

architecture dataflow of fulladder is

begin

			Co <= (B and A) or (Ci and A) or (Ci and B);		--carry output logic
			S <= (not Ci and not B and A) or (not Ci and B and not A) or (Ci and not B and not A) or (Ci and B
and A);  --sum output logic

end dataflow;

----------------------------Full Adder Block----------------------------
----------------------------------------------------------------------------
```

The entity portion of this code takes three inputs; A, B, and C_i as scalars, and two outputs; C_o and S as scalars.

The architecture section, which denotes a *dataflow* coding style, uses the previously discovered Boolean logic statements to create a simple logic definition for the two outputs for each *fulladder* which is implemented.

Using dataflow for this component of the program allows for a concise, efficient Boolean logic-flow which is both easy to follow, and runs quickly (simultaneous execution nature of *dataflow* style).

Below is the constructed block diagram for a single *fulladder*, provided by the Laboratory Exercise Manual[1]. Note the three inputs and two outputs types, which will be significant in the second part of this experiment.

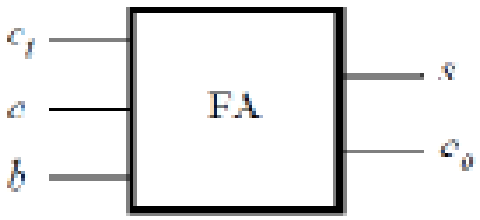

Figure 3 - Block Diagram for a Full Adder

Part 2 – The Four-bit Ripple-Carry Adder

Luckily, the *fulladder* was completed in the **Part 1** of the study, and a four-bit ripple carry adder needs to iterate this component four times (thus no truth table is required). The logical schematic for this is shown below, provided by the Lab Experiments Manual[1]:

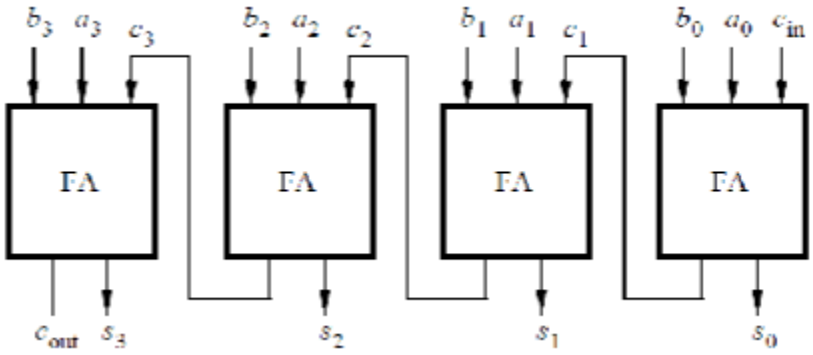

Figure 4 - Logic Schematic for Four-bit Ripple-Carry Adder

The noteworthy behavior for this logic-flow is that the *Carry* value is carried between each *fulladder* component. Thus, the *fulladder* component cannot be simply added four times to this VHDL program to function properly. Note that there is a separate C_{in} for the first *fulladder* component, and a separate C_{out} for the last *fulladder* component. The final carry will be the result displayed on the board, and the first carry has no *fulladder* component to carry from, thus will need to serve as the carry input from the board.

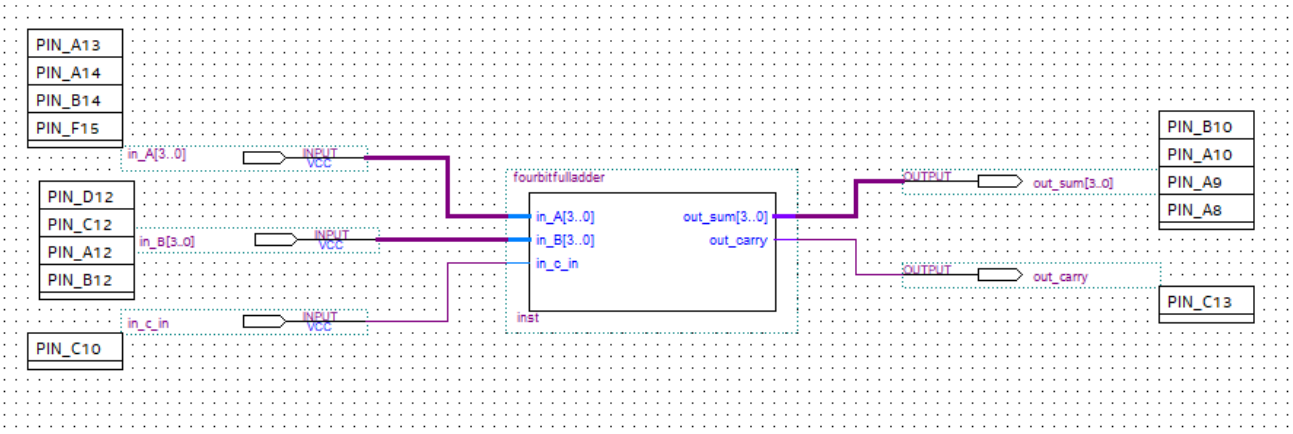

Figure 5 - Block Diagram for Completed Experiment

Pictured above in **figure** 5 is the completed block diagram for the cumulative experiment, with pin assignments included. Below are the specific pin assignments correlated to their VHDL counterpart names:

out_sum[3]	Output	PIN_B10	7	B7_N0	PIN_B10	3.3-V LVTTL
out_sum[2]	Output	PIN_A10	7	B7_N0	PIN_A10	3.3-V LVTTL
out_sum[1]	Output	PIN_A9	7	B7_N0	PIN_A9	3.3-V LVTTL
out_sum[0]	Output	PIN_A8	7	B7_N0	PIN_A8	3.3-V LVTTL
out_carry	Output	PIN_C13	7	B7_N0	PIN_C13	3.3-V LVTTL
in_c_in	Input	PIN_C10	7	B7_N0	PIN_C10	3.3-V LVTTL
in_B[3]	Input	PIN_D12	7	B7_N0	PIN_D12	3.3-V LVTTL
in_B[2]	Input	PIN_C12	7	B7_N0	PIN_C12	3.3-V LVTTL
in_B[1]	Input	PIN_A12	7	B7_N0	PIN_A12	3.3-V LVTTL
in_B[0]	Input	PIN_B12	7	B7_N0	PIN_B12	3.3-V LVTTL
in_A[3]	Input	PIN_A13	7	B7_N0	PIN_A13	3.3-V LVTTL
in_A[2]	Input	PIN_A14	7	B7_N0	PIN_A14	3.3-V LVTTL
in_A[1]	Input	PIN_B14	7	B7_N0	PIN_B14	3.3-V LVTTL
in_A[0]	Input	PIN_F15	7	B7_N0	PIN_F15	3.3-V LVTTL

Figure 6 - Pinout Assignments

Finally, the project can be compiled, and ensuring that the USB-Blaster hardware is added to the FPGA environment, and programmed to the development board. The results are discussed in the **Validation of Data** section of the report. This section of the report covers the resulting development board behavior, with supporting images and evidence.

Code Explanation Part 2

Included below are both an image, and text inclusion of the complete code segment for part 2 of the experiment:

```
--------------------------------------------------------------------------------
----------The Four Bit Rpple Carry Full Adder Main Block----------------

 library ieee;
 use ieee.std_logic_1164.all;

entity fourbitfulladder is
port  (
         in_A, in_B : in STD_LOGIC_VECTOR(3 downto 0);   --each input is 4 bits, one bit for each adder
         in_c_in : in STD_LOGIC;                         --the single Cin for the first adder only
         out_sum : out STD_LOGIC_VECTOR(3 downto 0);     --4 bit vector for each of the four sum (S) from adders
         out_carry : out STD_LOGIC                       --final carry value from fourth adder
      );
end fourbitfulladder;

architecture structure of fourbitfulladder is

component fulladder is
   port  (
            A, B, Ci : in std_logic;      --adder inputs
            Co, S : out std_logic         --adder outputs
         );
end component;

signal tempcarry : std_logic_vector(2 DOWNTO 0); --vector which holds carried value between adders

begin

   adder0 : fulladder port map(A => in_A(0), B => in_B(0), Ci => in_c_in, Co => tempcarry(0), S => out_sum(0));   --port map statements for each adder
   adder1 : fulladder port map(A => in_A(1), B => in_B(1), Ci => tempcarry(0), Co => tempcarry(1), S => out_sum(1));
   adder2 : fulladder port map(A => in_A(2), B => in_B(2), Ci => tempcarry(1), Co => tempcarry(2), S => out_sum(2));
   adder3 : fulladder port map(A => in_A(3), B => in_B(3), Ci => tempcarry(2), Co => out_carry, S => out_sum(3));

end structure;

----------The Four Bit Rpple Carry Full Adder Main Block----------------
--------------------------------------------------------------------------------
```

Figure 7 -Completed Code for fourbitfulladder

The entity statement, provided by the Lab Experiment Manual[1], defines two 4-bit input vectors (the values to be added together), a scalar Carry input, a 4-bit output summation of the program (sum from each individual *fulladder* component), and a final out carry scalar for the fourth resulting carry value.

The architectural section of the program adds a *fulladder* component described previously in **Part 1** of the report, and a signal called *tempcarry* which will serve as the bridge for carry values between *fulladder* implementations. Next, instantiations of the *fulladder* component (adder0 --> adder 3) use

port maps to bring values from the board into the logic for each *fulladder*. Note that in each instantiation the adders, *tempcarry* (starting at the second adder) connects to the Carry input, as well as the Carry output (until the final adder). This serves as the bridge displayed in the logic schematic earlier described.

Below is the text included finalized code for the second part of this report.

```
----------------------------------------------------------------------------------
----------The Four Bit Rpple Carry Full Adder Main Block----------

library ieee;
use ieee.std_logic_1164.all;

entity fourbitfulladder is
port	(
			in_A, in_B : in STD_LOGIC_VECTOR(3 downto 0);	--each input is 4 bits, one bit for
each adder
			in_c_in : in STD_LOGIC;
	--the single Cin for the first adder only
			out_sum : out STD_LOGIC_VECTOR(3 downto 0);		--4 bit vector for each of the
four sum (S) from adders
			out_carry : out STD_LOGIC
	--final carry value from fourth adder
		);
end fourbitfulladder;

architecture structure of fourbitfulladder is

component fulladder is
	port	(
				A, B, Ci : in std_logic;		--adder inputs
				Co, S : out std_logic			--adder outputs
			);
end component;

signal tempcarry : std_logic_vector(2 DOWNTO 0); --vector which holds carried value between adders

begin

	adder0 : fulladder port map(A => in_A(0), B => in_B(0), Ci => in_c_in, Co => tempcarry(0), S =>
out_sum(0));	--port map statements for each adder
	adder1 : fulladder port map(A => in_A(1), B => in_B(1), Ci => tempcarry(0), Co => tempcarry(1), S =>
out_sum(1));
	adder2 : fulladder port map(A => in_A(2), B => in_B(2), Ci => tempcarry(1), Co => tempcarry(2), S =>
out_sum(2));
	adder3 : fulladder port map(A => in_A(3), B => in_B(3), Ci => tempcarry(2), Co => out_carry, S =>
out_sum(3));

end structure;

----------The Four Bit Rpple Carry Full Adder Main Block----------
----------------------------------------------------------------------------------
```

VALIDATION OF DATA

Bench Analysis

https://youtu.be/Cqov7iWXljI

Functional Simulation Analysis

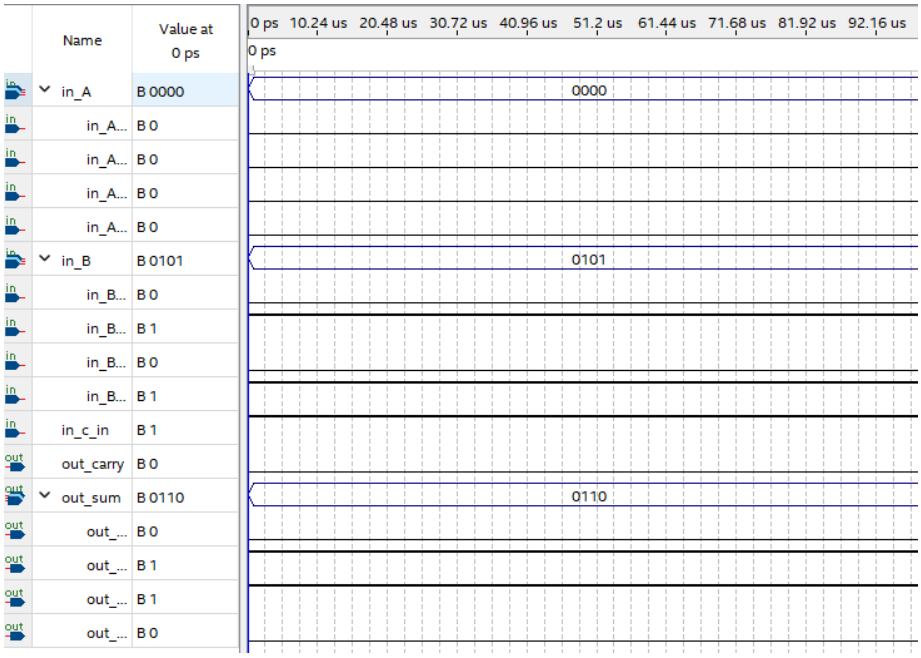

Figure 8 - Functional Simulation (0 + 4 + (carry(1)) = 5)

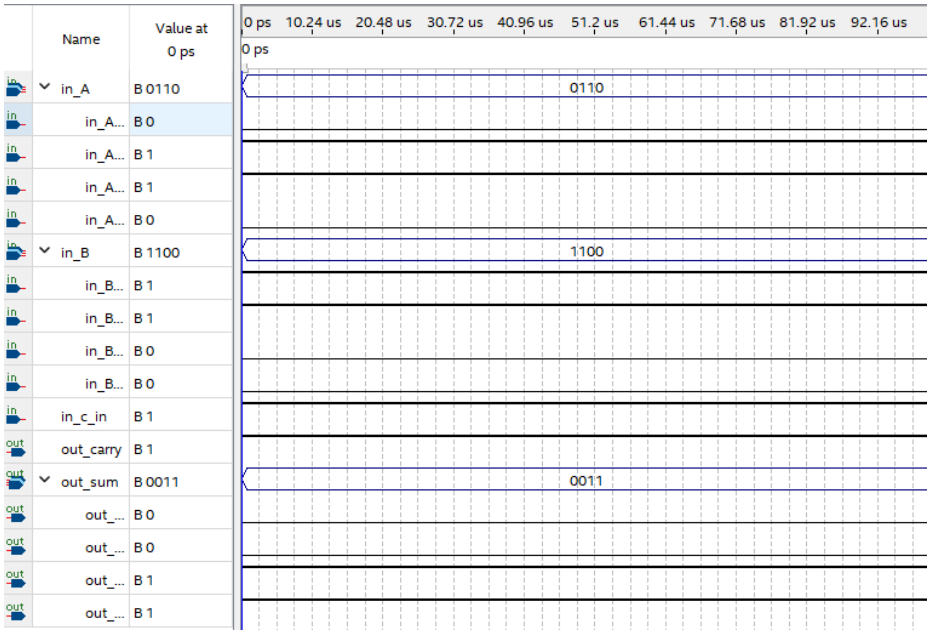

Figure 9- Functional Simulation (6 +12 + (carry(1)) = 18)

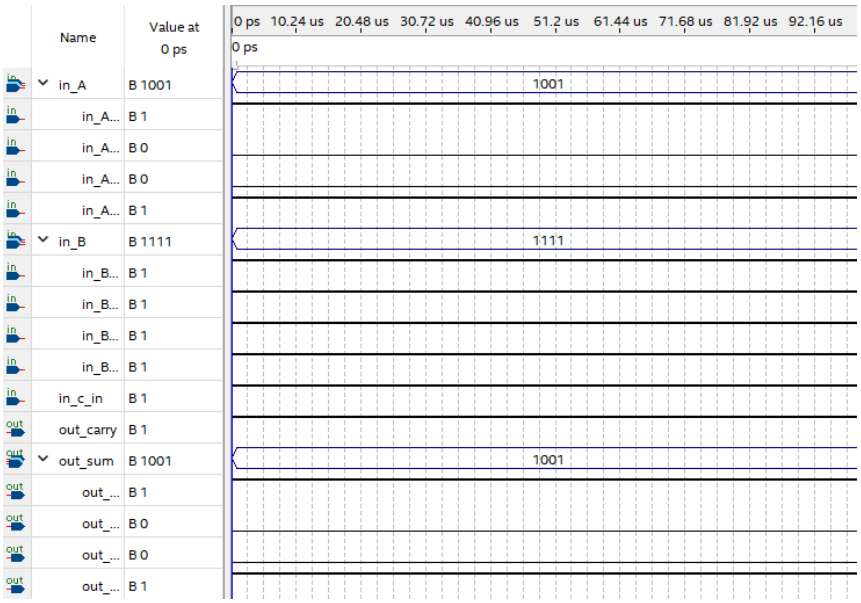

Figure 10 - Functional Simulation (9 + 15 + carry(1) = 24)

The three provided functional simulations display the behavior of the system under three arithmetic conditions. The first image (top left) sets the A four-bit vector to *0000*, and B to *0101*. This is equivalent to the decimal equation 0 + 4. The carry is set to one and therefore the final value is 5. Follow this same reasoning using the image captions to understand the second two functional simulation waveforms.

CONCLUSION

The experiment was conducted successfully and efficiently demonstrated the use of dataflow and structural coding styles to efficiently produce a system acting as a four-bit ripple-carry adder. The logic practice of designing a truth table with resulting Karnaugh maps assisted in generating the Boolean logic required for the dataflow section of the experiment, and served as a strong component to be used in the main VHDL block. The program in essence takes two, four-bit values and adds them together, utilizing a bridged carry between adders to allow for a fifth bit, representing the overflow carry bit. The structural coding style allowed for a simple full adder to be created as a component, and instantiated into the top level architecture 4 times, rather than creating four individual adder designs. This makes for a more responsive, modular system which is more efficient and faster. The use of dataflow in the *fulladder* component allowed for an efficient and simple Boolean logic based design for the individual segment. If this experiment were to be conducted again, more advanced arithmetic sequences such as multiplication and division along with carry could be designed and observed.

REFERENCES

[1] Professor Ashley Evans, *Logic Devices Programming Lab Manual*, 1st ed. Orlando, FL: Valencia College, 2025.