Combinational Logic (Computer Arithmetic)

Christopher Hunt

Objectives

The objective of this lab is to design and implement a 4-bit ripple-carry adder using 8 switches as inputs and LEDs on the FPGA to display the 5-bit output. By completing this project, we will gain a deeper understanding of creating block diagrams, building logic schematics using Quartus Prime, and simulating digital logic designs. This lab will provide hands-on experience in implementing a basic adder using combinational logic, which is a critical component of processors and other digital systems.

Equipment

- Quartus Prime Lite Edition V. 20.1.1
- DE10-Lite kit with MAX10 10M50DAF484C7G FPGA
- USB to USB-B cable
- Chapter 5 of Digital Design and Computer Architecture: ARM Edition by Sarah L. Harris and David Harris

---

Design

In this lab we are designing and implementing a 4-bit ripple-carry adder. This is a digital circuit used to add two binary numbers and is the basic building block for ALU's. A ripple-carry adder works cascading full adders (4 bits requires 4 adders). By adding the least significant bits first and then propagating the carry to the next bit position the combinational logic design is able to add two binary numbers.

The fuller adder will have 3 inputs, A, B, and C_in and to outputs Sum and C_out (fig.1). The boolean logic for each output is as follows:

$$Sum = A \oplus B \oplus C_{in}$$
$$C_{out} = A * B + A * C_{in} + B * C_{in}$$

This logic design creates the truth table that can be viewed in Table 1.

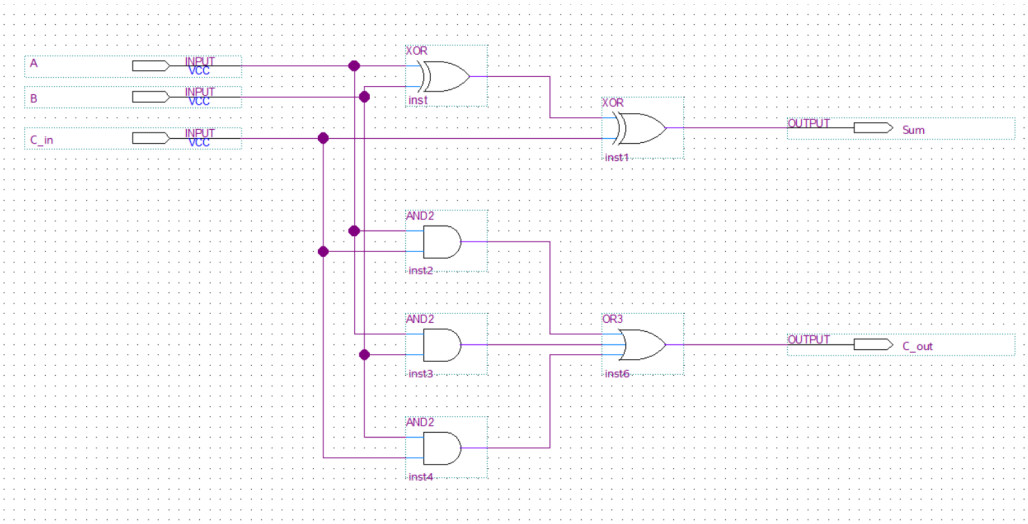

Figure 1: Fuller Adder Schematic

A	B	Carry In	Value (2-bit Binary)	Value (Decimal)
0b0	0b0	0b0	0b00	0
0b0	0b0	0b1	0b01	1
0b0	0b1	0b0	0b01	1
0b0	0b1	0b1	0b10	2
0b1	0b0	0b0	0b01	1
0b1	0b0	0b1	0b10	2
0b1	0b1	0b0	0b10	2
0b1	0b1	0b1	0b11	3

Table 1: Full-Adder Truth Table

Once the the full adder is designed, this module can be utilized in the overall block diagram for the 4-bit ripple carry adder (fig. 2). This adder will be able to display up to 5-bits of information (up to the value of 31 in decimal). Table 2 illustrates a partial truth table for the 4-bit adder.

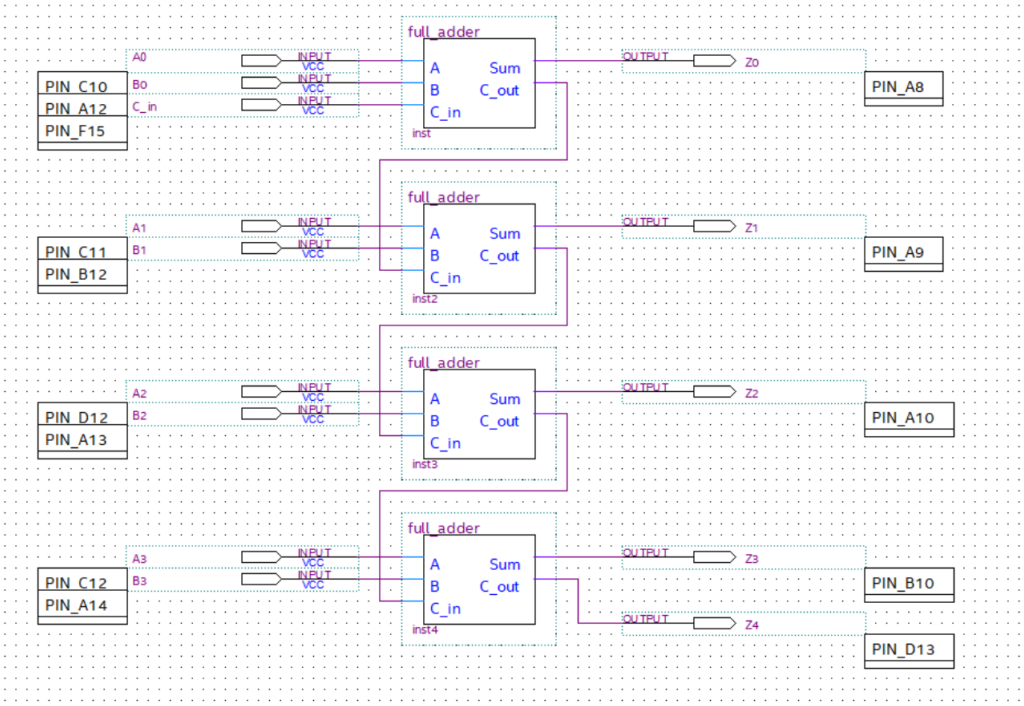

Figure 2: 4-bit Ripple Carry Adder

4-Bit Input A	4-Bit Input B	Output Value (5-bit Binary)	Value (Decimal)	Value (Hex)
0b0000	0b0000	0b00000	0	0h00
0b0000	0b0001	0b00001	1	0h01
0b0000	0b0010	0b00010	2	0h02
0b0000	0b0011	0b00011	3	0h03
0b0000	0b0100	0b00100	4	0h04
0b0000	0b0101	0b00101	5	0h05
0b1000	0b0000	0b01000	8	0h08
0b1000	0b0001	0b01001	9	0h09
0b1000	0b0010	0b01010	10	0h0A
0b1000	0b0011	0b01011	11	0h0B
0b1111	0b0011	0b10010	18	0h12
0b1111	0b1000	0b10111	23	0h17
0b1111	0b1010	0b11001	25	0h19
0b1111	0b1011	0b11010	26	0h1A

Table 2: Partial 4-bit Adder Truth Table

The 4-bit inputs, A and B, and the first Carry in were assigned to the switches on the DE10-Lite, while the 5-bit output was sent to the LED's. The exact pin placement can be viewed in tables 3 and 4.

Input PIN	FPGA PIN
A0	PIN_C10
A1	PIN_C11
A2	PIN_D12
A3	PIN_C12
B0	PIN_A12
B1	PIN_B12
B2	PIN_A13
B3	PIN_A14
C_in	PIN_F15

Table 3: Input PIN to FPGA PIN Mapping

Output PIN	FPGA PIN
Z0	PIN_A8
Z1	PIN_A9
Z2	PIN_A10
Z3	PIN_B10
Z4	PIN_D13

Table 4: Output PIN to FPGA PIN Mapping

Simulation

The 4-bit ripple carry adder design was compiled using Quartus Prime Lite and a verilog file was exported. This was then used in ModelSim to simulate the design before hardware implementation. The simulation (fig. 3) produced the expected output as desrcibed from Table 2.

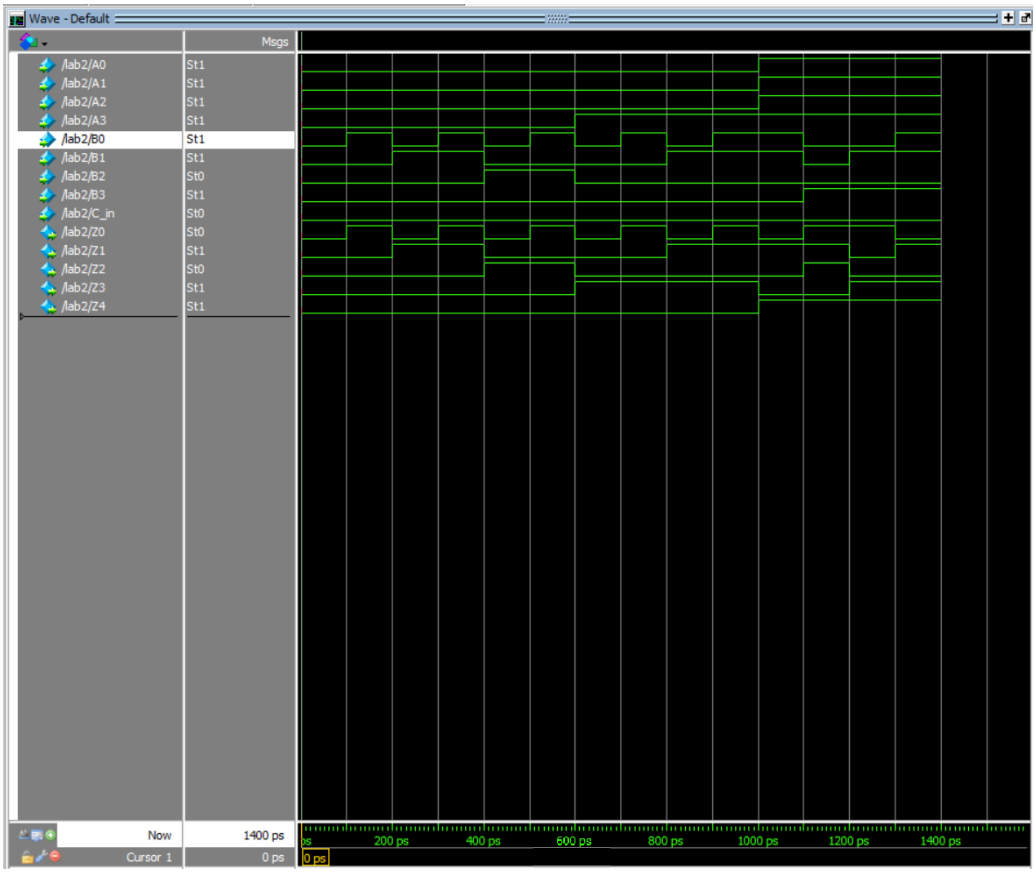

Figure 3: Simulation of Logic Design in ModelSim

Implementation

Upon successful completion of the design in Quartus Prime and simulation using ModelSim, the HDL program was uploaded to the DE10-Lite. All switch pin and LED assignments functioned according to the design and the output correctly matched the expected truth tables from above.

Observations

This lab went smoothly for every section. I had to review some content from lab 1 to remember how to appropriately use Quartus Prime. This was to be expected and helped reinforce the basics that were taught in lab 1.

Conclusion

The lab focused on the design and implementation of a 4-bit ripple-carry adder using combinational logic. The objectives were to gain a deeper understanding of creating block diagrams, building logic schematics using Quartus Prime, and simulating digital logic designs. Additionally, the lab aimed to provide hands-on experience in implementing a basic adder using combinational logic, which is an essential component of processors and other digital systems.

During the lab, the 4-bit ripple-carry adder was designed by cascading full adders. The full adder logic involved three inputs (A, B, and Cin) and two outputs (Sum and Cout). The truth table for the full adder was constructed, and subsequently, the 4-bit ripple carry adder was built using the full adder module. The adder was capable of displaying up to 5-bit output values.

The design was simulated using ModelSim, ensuring that the logic behaved as expected. The simulation confirmed that the outputs matched the truth tables derived from the design.

Following successful simulation, the design was implemented on the DE10-Lite FPGA board. The inputs were connected to switches, and the outputs were displayed using LEDs. The physical implementation validated that the design and pin assignments were correct, and the outputs matched the expected truth tables.

Study Questions

1. Explain how you would convert your 4-bit adder to a 4-bit adder/subtractor.

To convert this to an adder/subtractor the design would have to be modified in this way:

- First the counting system would need to be in twos complement.
- The full adder would be given a subtract control pin, this would determine what mode the adder is on, either add or subtract.
- If the control switch was in add mode (SUB pin is LOW) the full adders would work as before.
- If the control switch was in subtract mode (SUB pin HIGH), the full adders would be required to invert the B bit using an XOR gate with B and SUB as inputs before B is fed into the adder. If the full adder is in the position of the least significant bit, add one. Addition would then be performed as before.

2. In your own words, explain what pull resistors do in the FPGA.

Pull resistors allow designers to set default values to FPGA pins, preventing floating signals and ensuring predictable behavior. By setting the pins to a known state, designers can avoid potential issues that may arise from undefined logic levels, such as unintended switching or interference.

3. Explain your selection for the pull mode for the pin connected to the least-significant full-adder's carry-in.

I chose to place the least significant bits Carry In pin to a switch, this allowed me to select which value I wanted it to be set to and aided in testing the implementation of the program on the device.

Appendix

lab2_fulladder.v

```
1  // Copyright (C) 2020  Intel Corporation. All rights reserved.
2  // Your use of Intel Corporation's design tools, logic functions
3  // and other software and tools, and any partner logic
4  // functions, and any output files from any of the foregoing
5  // (including device programming or simulation files), and any
6  // associated documentation or information are expressly subject
7  // to the terms and conditions of the Intel Program License
8  // Subscription Agreement, the Intel Quartus Prime License Agreement,
9  // the Intel FPGA IP License Agreement, or other applicable license
10 // agreement, including, without limitation, that your use is for
```

```
// the sole purpose of programming logic devices manufactured by
// Intel and sold by Intel or its authorized distributors.  Please
// refer to the applicable agreement for further details, at
// https://fpgasoftware.intel.com/eula.

// PROGRAM              "Quartus Prime"
// VERSION              "Version 20.1.1 Build 720 11/11/2020 SJ Lite Edition
    "
// CREATED              "Thu Apr 27 12:10:55 2023"

module lab2_fulladder(
        A,
        B,
        C_in,
        Sum,
        C_out
);

input wire      A;
input wire      B;
input wire      C_in;
output wire     Sum;
output wire     C_out;

wire    SYNTHESIZED_WIRE_0;
wire    SYNTHESIZED_WIRE_1;
wire    SYNTHESIZED_WIRE_2;
wire    SYNTHESIZED_WIRE_3;

assign  SYNTHESIZED_WIRE_0 = A ^ B;
assign  Sum = SYNTHESIZED_WIRE_0 ^ C_in;
assign  SYNTHESIZED_WIRE_3 = A & B;
assign  SYNTHESIZED_WIRE_1 = A & C_in;
assign  SYNTHESIZED_WIRE_2 = B & C_in;
assign  C_out = SYNTHESIZED_WIRE_1 | SYNTHESIZED_WIRE_2 | SYNTHESIZED_WIRE_3
    ;

endmodule
```

lab2_block_schem.v

```
    // Copyright (C) 2020  Intel Corporation. All rights reserved.
// Your use of Intel Corporation's design tools, logic functions
// and other software and tools, and any partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Intel Program License
// Subscription Agreement, the Intel Quartus Prime License Agreement,
// the Intel FPGA IP License Agreement, or other applicable license
// agreement, including, without limitation, that your use is for
// the sole purpose of programming logic devices manufactured by
// Intel and sold by Intel or its authorized distributors.  Please
// refer to the applicable agreement for further details, at
// https://fpgasoftware.intel.com/eula.
```

```
15
16 // PROGRAM                "Quartus Prime"
17 // VERSION                "Version 20.1.1 Build 720 11/11/2020 SJ Lite Edition
       "
18 // CREATED                "Thu Apr 27 12:11:03 2023"
19
20 module lab2_block_schem(
21         A0,
22         B0,
23         C_in,
24         A1,
25         A2,
26         B2,
27         A3,
28         B3,
29         B1,
30         Z0,
31         Z1,
32         Z2,
33         Z3,
34         Z4
35 );
36
37
38 input wire      A0;
39 input wire      B0;
40 input wire      C_in;
41 input wire      A1;
42 input wire      A2;
43 input wire      B2;
44 input wire      A3;
45 input wire      B3;
46 input wire      B1;
47 output wire     Z0;
48 output wire     Z1;
49 output wire     Z2;
50 output wire     Z3;
51 output wire     Z4;
52
53 wire    SYNTHESIZED_WIRE_0;
54 wire    SYNTHESIZED_WIRE_1;
55 wire    SYNTHESIZED_WIRE_2;
56
57
58
59
60
61 lab2_fulladder  b2v_inst(
62         .A(A0),
63         .B(B0),
64         .C_in(C_in),
65         .Sum(Z0),
66         .C_out(SYNTHESIZED_WIRE_0));
67
68
```

```
69 lab2_fulladder  b2v_inst1(
70         .A(A1),
71         .B(B1),
72         .C_in(SYNTHESIZED_WIRE_0),
73         .Sum(Z1),
74         .C_out(SYNTHESIZED_WIRE_1));
75
76
77 lab2_fulladder  b2v_inst2(
78         .A(A2),
79         .B(B2),
80         .C_in(SYNTHESIZED_WIRE_1),
81         .Sum(Z2),
82         .C_out(SYNTHESIZED_WIRE_2));
83
84
85 lab2_fulladder  b2v_inst3(
86         .A(A3),
87         .B(B3),
88         .C_in(SYNTHESIZED_WIRE_2),
89         .Sum(Z3),
90         .C_out(Z4));
91
92
93 endmodule
```