ELEC 2607 LAB REPORT 2: A 3-Bit Adder/Subtractor Circuit

Title: A 3-Bit Binary Adder/Subtractor Circuit

Your Name: Dumany Lombe

Student Number: 101316658

Lab Section: L4

Date: 03/10/2024

1. Introduction

- ***Objective:***

This lab aimed to design and test a 3-bit binary adder/subtractor circuit using Verilog. The circuit was designed to perform binary addition and subtraction based on a control signal, with outputs including the sum or difference and an overflow indicator. The goal was to simulate, analyse, and identify discrepancies between expected and actual results.

- ***Background***:

This lab focuses on binary arithmetic and two's complement representation, fundamental concepts in digital circuits. Binary addition and subtraction are implemented using logic gates and are critical in digital systems such as processors and arithmetic logic units (ALUs). Overflow detection is a crucial concept in these systems, as it allows the circuit to identify when the result of an operation exceeds the representable range.

---

2. Specifications

- ***Circuit Overview****:*

The circuit is designed to take two 3-bit binary numbers, `X` and `Y`, and perform either addition or subtraction depending on a control signal, `M`. If `M = 0`, the circuit performs addition; if `M = 1`, it performs subtraction using XOR gates to invert the `Y` input. The output is a 4-bit result `S`, including the sum or difference and the carry-out or overflow bit.

- ***Circuit Functionality****:*

The addition uses a series of full adders, with carry bits propagated from the least significant to the most crucial bit. The circuit uses two’s complement arithmetic for subtraction by inverting `Y` and adding it to `X`. The circuit also checks for overflow by comparing the carry-in and carry-out of the most significant bit.

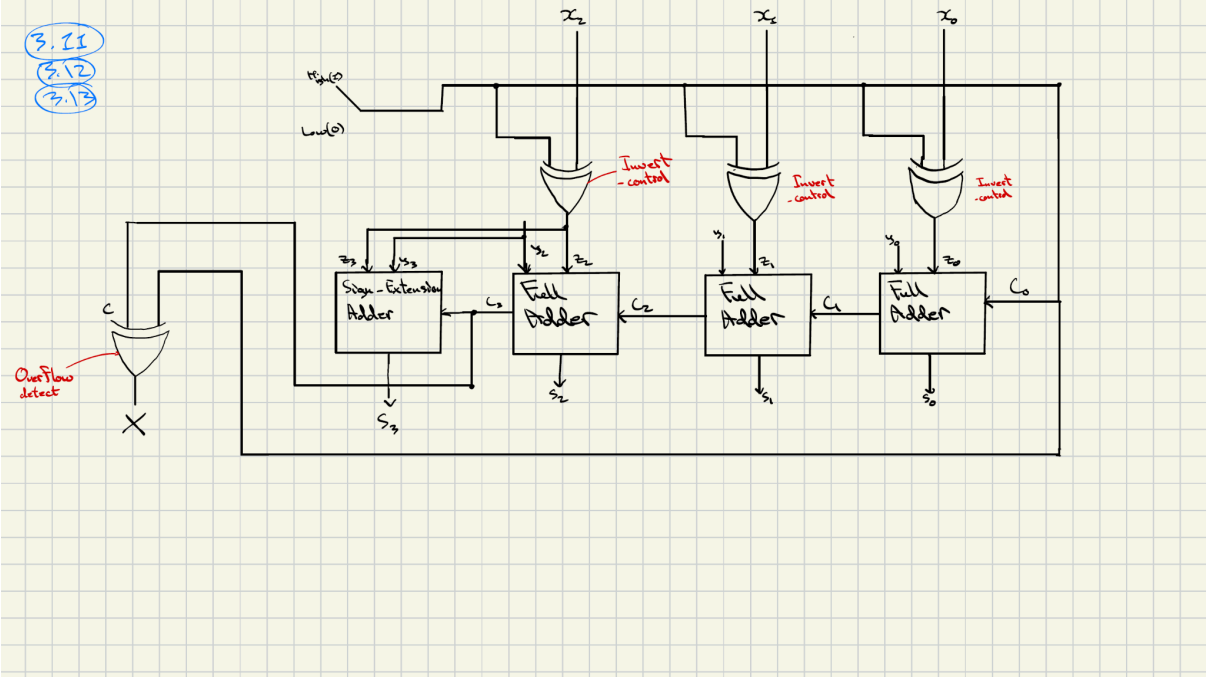
---

3. Design

- ***Circuit Design:***

The adder/subtractor circuit was designed with three full adders, each capable of performing addition and subtraction. The XOR gates were used to handle the inversion of the `Y` input for subtraction. The carry propagation was managed using AND, OR, and XOR gates and the final carry-out was used to detect overflow conditions.

- ***Schematic/Block Diagram:***



---

4. Pre-Lab Components

- ***Binary Arithmetic and Two's Complement Representation:***

The lab uses binary addition and subtraction, which can be easily handled using two's complement for subtraction. Here, a binary number's two's complement is found by inverting all the bits and adding 1 to the least significant bit.

Example of Two's Complement:

- Positive 3 (`011`)
- Negative 3 in two's complement: Invert `011` to get `100`, then add 1 to get `101`.

This same principle is applied in the circuit to perform subtraction.

- ***Full Adder Design:***

The circuit uses full adders to handle both addition and subtraction. A full adder sums two bits and a carry-in bit and outputs a sum and a carry-out. The `Y` input is XORed for subtraction with the control signal to invert its bits when `M = 1` (subtraction mode).

- ***Overflow Detection:***

Overflow is detected by comparing the carry into the most significant bit with the carryout. If these two values differ, an overflow condition is signalled. This is crucial in ensuring the binary arithmetic results fit within the designated bit width.

---

5. Simulation and Results

- ***Test Results:***

The circuit was tested with various input combinations to verify addition and subtraction. The following cases were considered:

- ***Test Case 1:***

Inputs: `X = 3 (011)`, `Y = 5 (101)`, `M = 0` (Addition)
Expected Output: Sum = 8 (1000)
Actual Output: Sum = 011 (Carry-out = 1, Incorrect result)

- ***Test Case 2:***

Inputs: `X = 3 (110)`, `Y = 2 (010)`, `M = 1` (Subtraction)
Expected Output: Difference = 1 (0001)
Actual Output: Difference = 001 (Incorrect due to carry mismanagement)

- ***Comparison of Results:***

In some test cases, the actual results differed from the expected results due to issues with carry propagation and subtraction logic. These errors could be attributed to incorrect carrying-out signal handling during subtraction.

---

6. Analysis and Discussions

- ***Analysis of Results:***

The results showed that the addition worked as expected for most cases, but the subtraction logic encountered problems. The main issue was carry propagation, particularly when switching between addition and subtraction modes. This is a common issue in digital circuits, where the same logic gates are used for both operations.

- ***Carry Propagation Errors:***

The primary problem encountered during subtraction was an incorrect carry-out, which led to wrong results. The XOR gate that inverts the `Y` input may not function correctly for all bits, leading to improper subtraction results.

- ***Overflow Detection:***

Overflow detection worked as expected in addition mode, but in subtraction mode, the results were inconsistent due to the carry propagation error. Proper debugging and a more thorough examination of the logic gates used in the circuit are needed to resolve this issue.

---

7. Conclusion

The lab helped solidify my understanding of binary arithmetic, two's complement representation, and overflow detection. While the circuit's addition worked well, the subtraction part encountered issues due to carry propagation errors. These problems were likely caused by time constraints during the lab, leading to incomplete testing and debugging. Improvements to the XOR gate implementation and better management of the carry signals are recommended to correct these errors.

---

8. Verilog Code

- ***Verilog Code Explanation****:*

The Verilog code implemented a 3-bit adder/subtractor with controlled inverters for subtraction and a full-adder architecture. A control signal determined whether the operation was addition or subtraction, and the result was stored in a 4-bit register.

Actual code:

- ***Code for 3-bit Adder/Subtractor:***

```
`timescale 1ns / 1ps // set time units

module three_bit_adder_xor (
  output [7:0] led,  // connect output LEDs 0-7 on FPGA
  input [7:0] swt   // connect input switches 0-7 on FPGA
 );

wire [2:0] X = swt[2:0];
wire [2:0] Y = swt[5:3];
wire [2:0] X_mod;
wire [2:0] S;
wire [3:1] C; // Please note that in this example we don't need C[0], because input carry = 0
wire I = swt [7];

// Calculate the sum for each bit using XOR gates



//Inverted Controlers
assign X_mod[0] = X[0] ^ I;
assign X_mod[1] = X[1] ^ I;
assign X_mod[2] = X[2] ^ I;

// Half adder
//assign X[0] = X[0] ^ I;
//assign C[1] = (X[0] & I);

// half Adder 2
//assign X[1] = X[1] ^ C[1];
//assign C[2] = (X[1] & C[1]);

// half Adder 3
//assign X[2] = X[2] ^ C[2];
//assign C[3] = (X[2] & C[2]);

//full part
// Half Adder 0
assign S[0] = X_mod[0] ^ Y[0];
assign C[1] = (X_mod[0] & Y[0]);

// Full Adder 1
assign S[1] = X_mod[1] ^ Y[1] ^ C[1];
```

```
assign C[2] = (X_mod[1] & Y[1]) | (Y[1] & C[1]) | (C[1] & X_mod[1]);

// Full Adder 2
assign S[2] = X_mod[2] ^ Y[2] ^ C[2];
assign C[3] = (X_mod[2] & Y[2]) | (Y[2] & C[2]) | (C[2] & X_mod[2]);

// Sign Extension Adder
assign S[2] = X_mod[2] ^ Y[2] ^ C[2];
assign C[3] = (X_mod[2] & Y[2]) | (Y[2] & C[2]) | (C[2] & X_mod[2]);


// sending results to the output
assign led[2:0] = S;  // Sum
assign led[3] = C[3]; // Carry-out
assign led[6] = I;
//assign led[4] = C[3]^(C[1] & C[2]);

//wire overflow;
//wire overflow_add = C[3]!= C[0]);
//wire overflow_sub = (X[2] != Y[2]) && (S[2] = X[2]);



endmodule
```

Expected code:

- ***Code for 3-bit Subtractor:***

```
`timescale 1ns / 1ps // set time units

module three_bit_adder_xor (
  output [7:0] led,  // connect output LEDs 0-7 on FPGA
  input [7:0] swt   // connect input switches 0-7 on FPGA
 );

wire [2:0] X = swt[2:0];
wire [2:0] Y = swt[5:3];
wire [2:0] Y_inv;
wire [2:0] S;
wire [3:1] C; // Please note that in this eample we don't need C[0], because input carry = 0

// Invert all bits of Y to subtract
assign Y_inv[0] = ~Y[0];
assign Y_inv[1] = ~Y[1];
assign Y_inv[2] = ~Y[2];

// Half Adder 0
```

```
assign S[0] = X[0] ^ Y_inv[0] ^ 1'b1; // Add 1 to LSB and use inverted Y instead of Y
assign C[1] = (X[0] & Y_inv[0]) | (X[0] & 1'b1) | (Y_inv[0] & 1'b1);

// Full Adder 1
assign S[1] = X[1] ^ Y_inv[1] ^ C[1]; // Use inverted Y instead of Y
assign C[2] = (X[1] & Y_inv[1]) | (X[1] & C[1]) | (Y_inv[1] & C[1]);

// Full Adder 2
assign S[2] = X[2] ^ Y_inv[2] ^ C[2]; // Use inverted Y instead of Y
assign C[3] = (X[2] & Y_inv[2]) | (X[2] & C[2]) | (Y_inv[2] & C[2]);

// sending results to the output
assign led[2:0] = S;  // Sum
assign led[3] = C[3]; // Carry-out

endmodule
```

- ***Testbench Code:***

```
`timescale 1ns / 1ps // set time units

module testbench_three_bit_adder_xor;

  wire [7:0] led;  // Add this line for LED output
  reg  [7:0] swt;  // Add this line for switch input

 // Instantiate the 3-bit adder module with correct port connections
 three_bit_adder_xor UUT (
   .led(led),
   .swt(swt)
 );

 // Stimulus generation
 initial begin
   swt[7:0] = 7'b0000000; // All inputs are initially set to 0

   // Test case 1: A=3 (binary 011), B=5 (binary 101)
   swt[2:0] = 3'b011;
   swt[5:3] = 3'b101;

   // Wait a bit before checking results
   #10;

        // Test case 2: A=3 (binary 110), B=5 (binary 010)
        swt[2:0] = 3'b110;
   swt[5:3] = 3'b010;
```

```
    // Wait a bit before checking results
    #10;

        // Test case 2: A=2 (binary 110), B=4 (binary 010)
        swt[2:0] = 3'b010;
    swt[5:3] = 3'b100;

    // Wait a bit before checking results
    #10;

 // Add more test cases as needed
    // ...

    // End simulation
    $finish;
  end

endmodule
```

- ***Code for 3-bit Adder/Subtractor:***

```
`timescale 1ns / 1ps // set time units

module three_bit_adder_subtractor (
  output [7:0] led,  // connect output LEDs 0-7 on FPGA
  input [7:0] swt    // connect input switches 0-7 on FPGA
);

wire [2:0] X = swt[2:0];
wire [2:0] Y = swt[5:3];
wire [2:0] Y_mod;
wire [2:0] S;
wire [3:1] C;
wire M = swt[6];

// Invert Y when M is high using XOR gates
assign Y_mod[0] = Y[0] ^ M;
assign Y_mod[1] = Y[1] ^ M;
assign Y_mod[2] = Y[2] ^ M;

// Half Adder 0
assign S[0] = X[0] ^ Y_mod[0] ^ M;
assign C[1] = (X[0] & Y_mod[0]) | (M & X[0])| (Y_mod[0] & M);

// Full Adder 1
assign S[1] = X[1] ^ Y_mod[1] ^ C[1];
assign C[2] = (X[1] & Y_mod[1]) | (Y_mod[1] & C[1]) | (C[1] & X[1]);
```

```
// Full Adder 2
assign S[2] = X[2] ^ Y_mod[2] ^ C[2];
assign C[3] = (X[2] & Y_mod[2]) | (Y_mod[2] & C[2]) | (C[2] & X[2]);

// Sending results to the output
assign led[2:0] = S;  // Result of addition/subtraction
assign led[3] = C[3]; // Carry-out
assign led[4] = M;    // Mode indicator (0 for addition, 1 for subtraction)


wire overflow;
wire overflow_add = (C[2] != C[3]); // Addition overflow
wire overflow_sub = (X[2] != Y[2]) && (S[2] != X[2]); // Subtraction overflow

assign overflow = M ? overflow_sub : overflow_add; // Choose overflow condition based on
mode

assign led[5] = overflow; // Indicate overflow status on LED[5]
assign led[6] = C[2];
assign led[7] = C[3]; // Final carry-out

endmodule
```

- ***Testbench Code:***

```
timescale 1ns / 1ps // set time units

module testbench_three_bit_adder_subtractor;

  wire [7:0] led;  // LED output
  reg  [7:0] swt;  // Switch input

 // Instantiate the 3-bit adder/subtractor module with correct port connections
 three_bit_adder_subtractor UUT (
   .led(led),
   .swt(swt)
 );

 // Stimulus generation
 initial begin
   swt[7:0] = 8'b00000000; // All inputs are initially set to 0

   // Test case 1: Addition A=3 (binary 011), B=2 (binary 010), M=0
   swt[2:0] = 3'b011; // X = 3
   swt[5:3] = 3'b010; // Y = 2
   swt[6] = 1'b0;     // M = 0 (addition)
   #10;
```

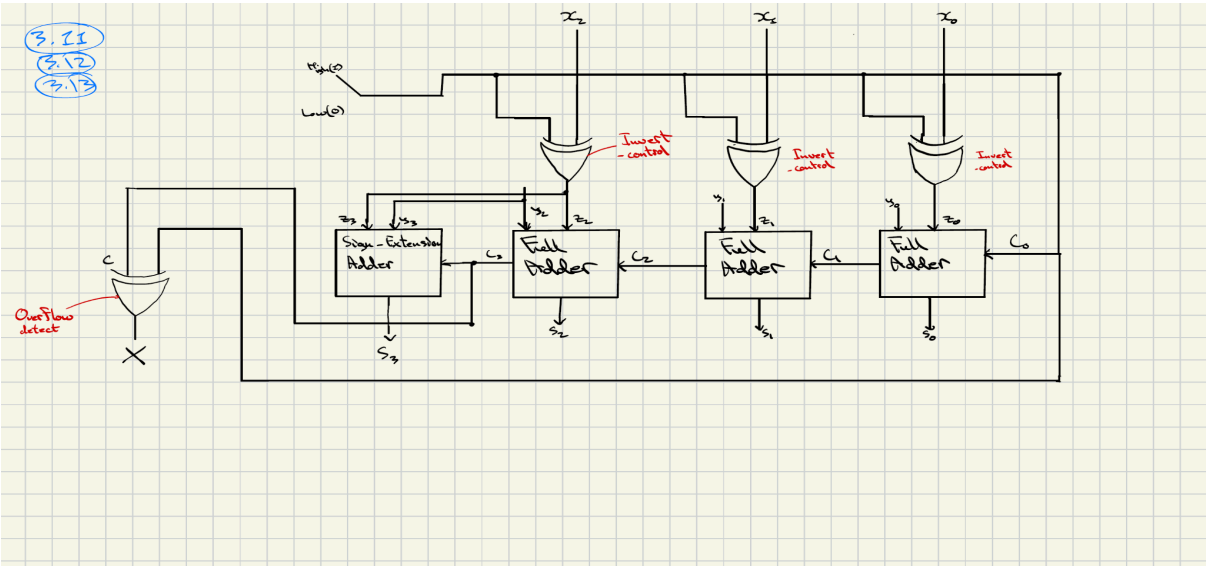
```
    // Test case 2: Subtraction A=3 (binary 011), B=2 (binary 010), M=1
    swt[2:0] = 3'b011; // X = 3
    swt[5:3] = 3'b010; // Y = 2
    swt[6] = 1'b1;     // M = 1 (subtraction)
    #10;


    // Test case 5: Addition A=1 (binary 001), B=3 (binary 011), M=0
    swt[2:0] = 3'b001; // X = 1
    swt[5:3] = 3'b011; // Y = 3
    swt[6] = 1'b0;     // M = 0 (addition)
    #10;

    // Test case 6: Subtraction A=1 (binary 001), B=3 (binary 011), M=1
    swt[2:0] = 3'b001; // X = 1
    swt[5:3] = 3'b011; // Y = 3
    swt[6] = 1'b1;     // M = 1 (subtraction)
    #10;

    // End simulation
    $finish;
  end

endmodule
```

`tide here).*

---

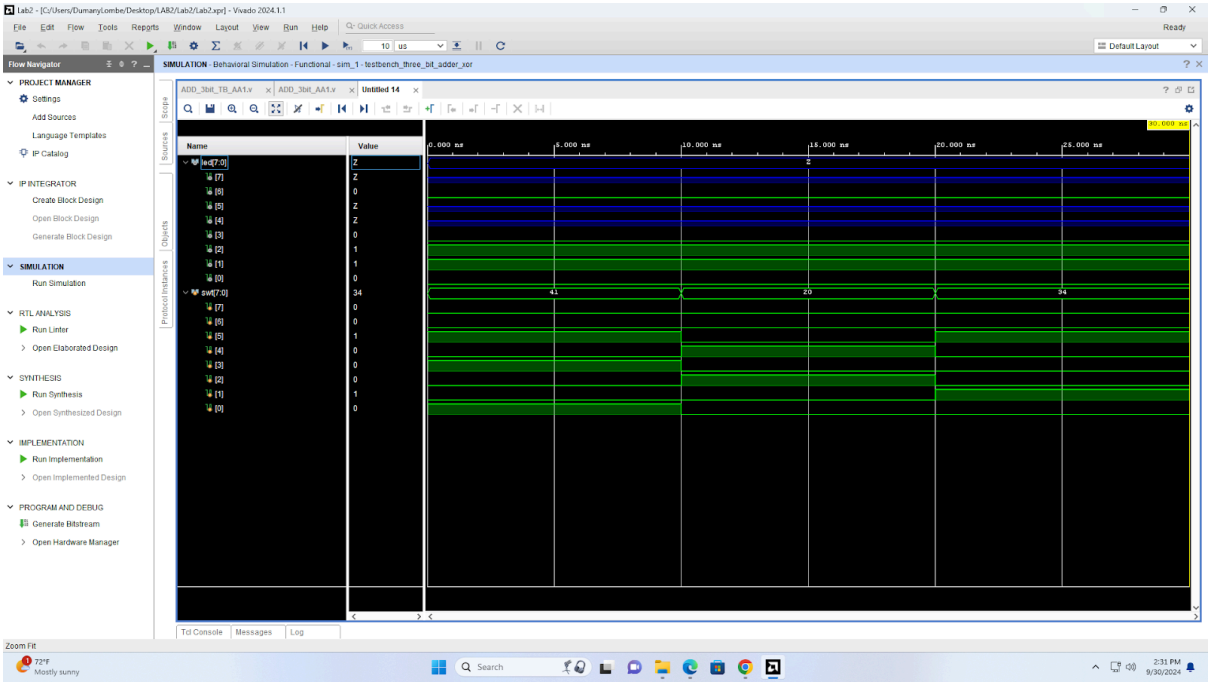
9. Appendix

- Pre-Lab Schematics



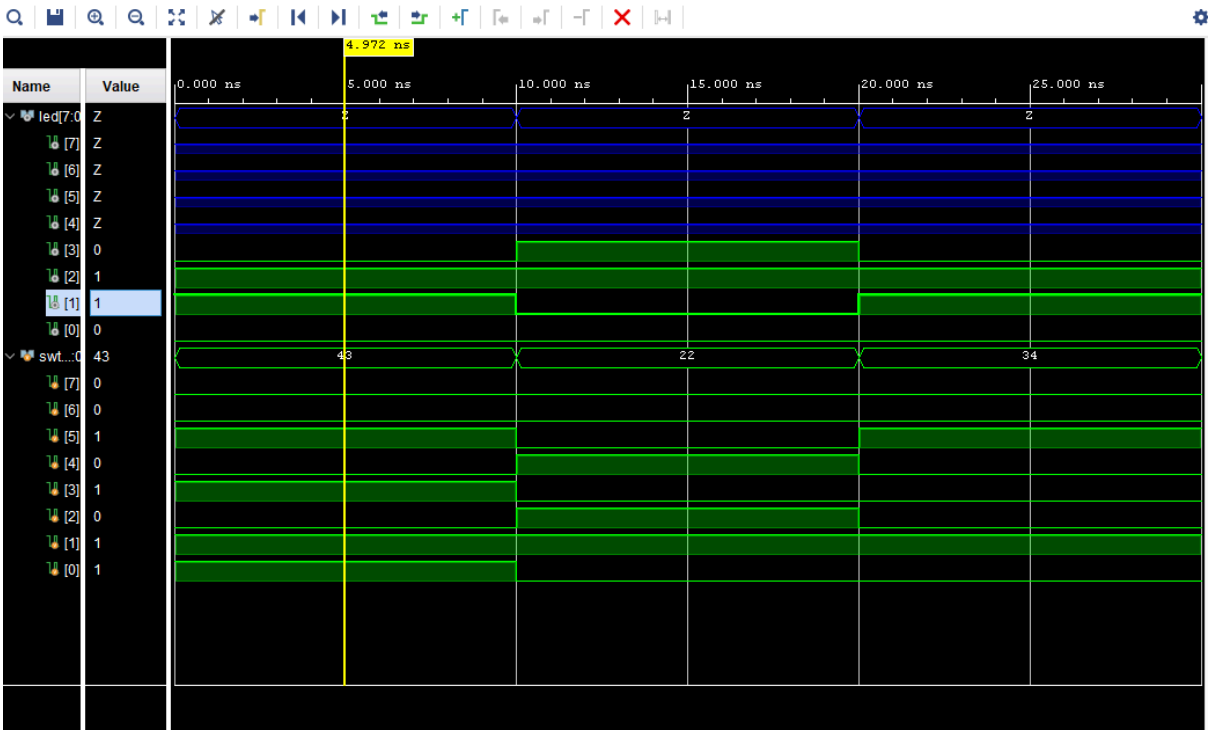
Additional Screenshots:

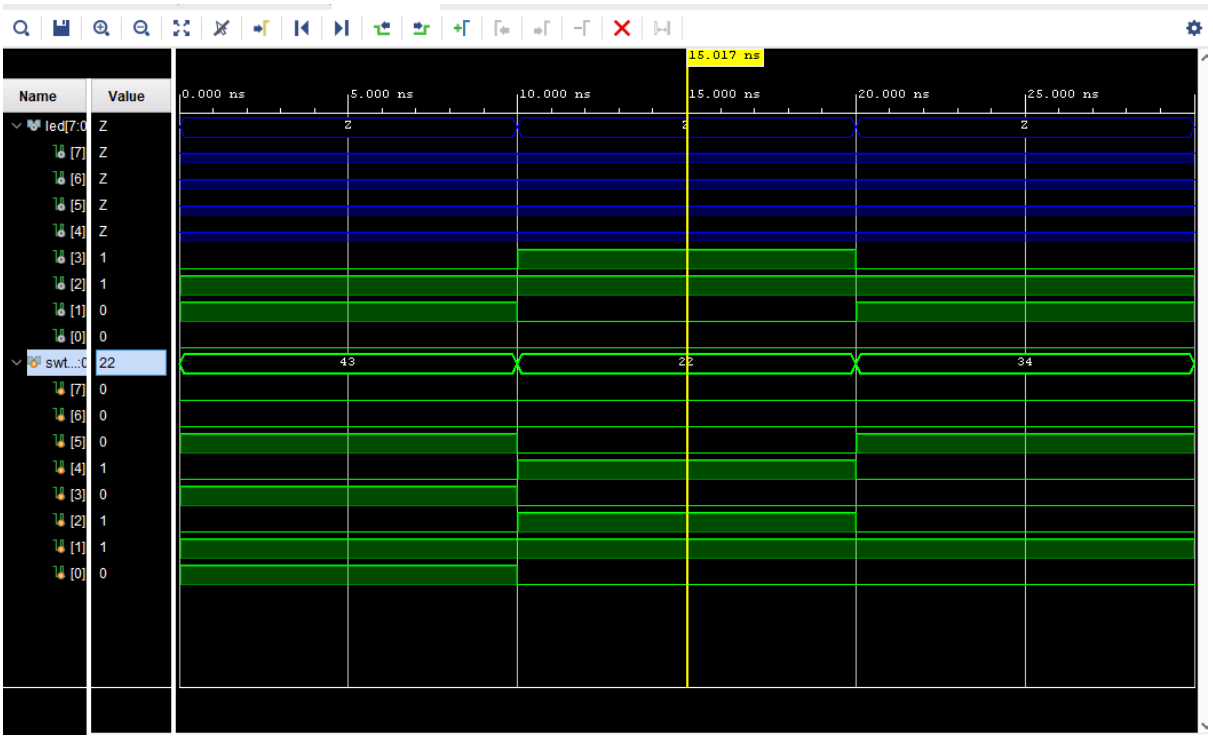
Actual results



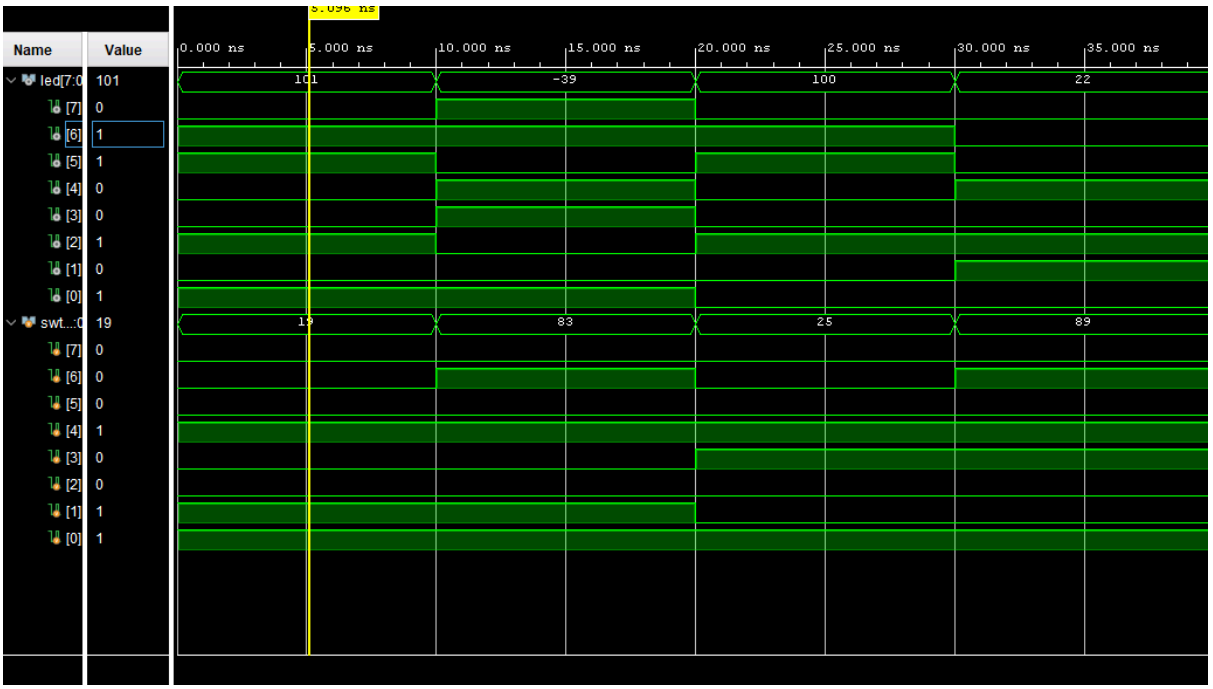
Expected Results

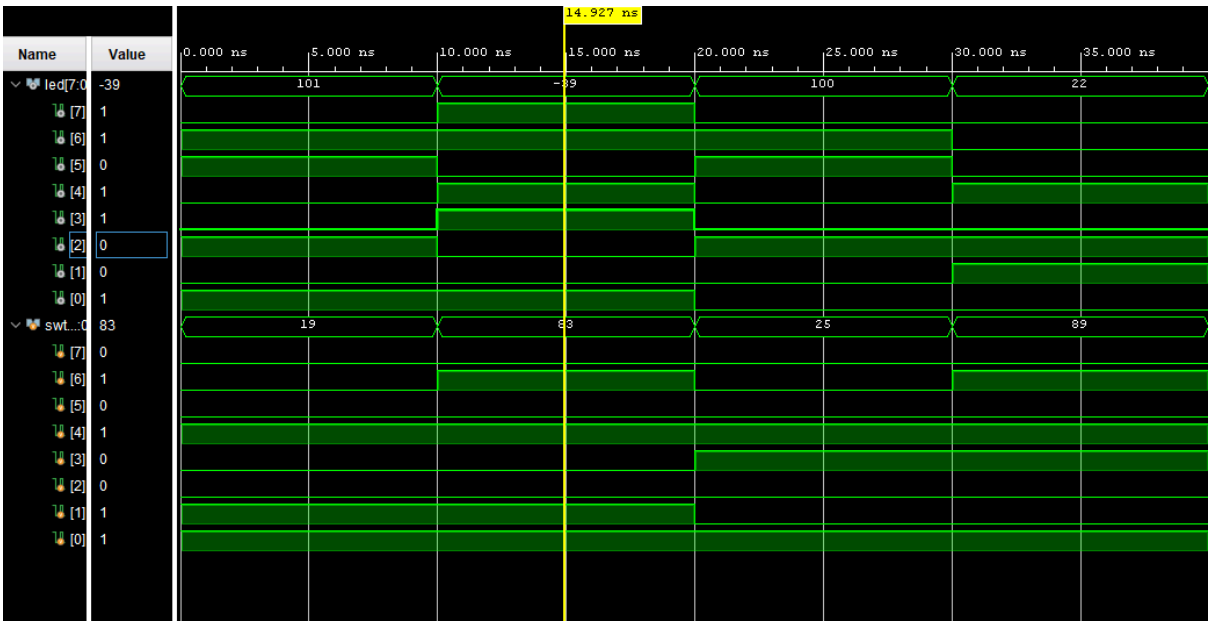
Subtracter Results:

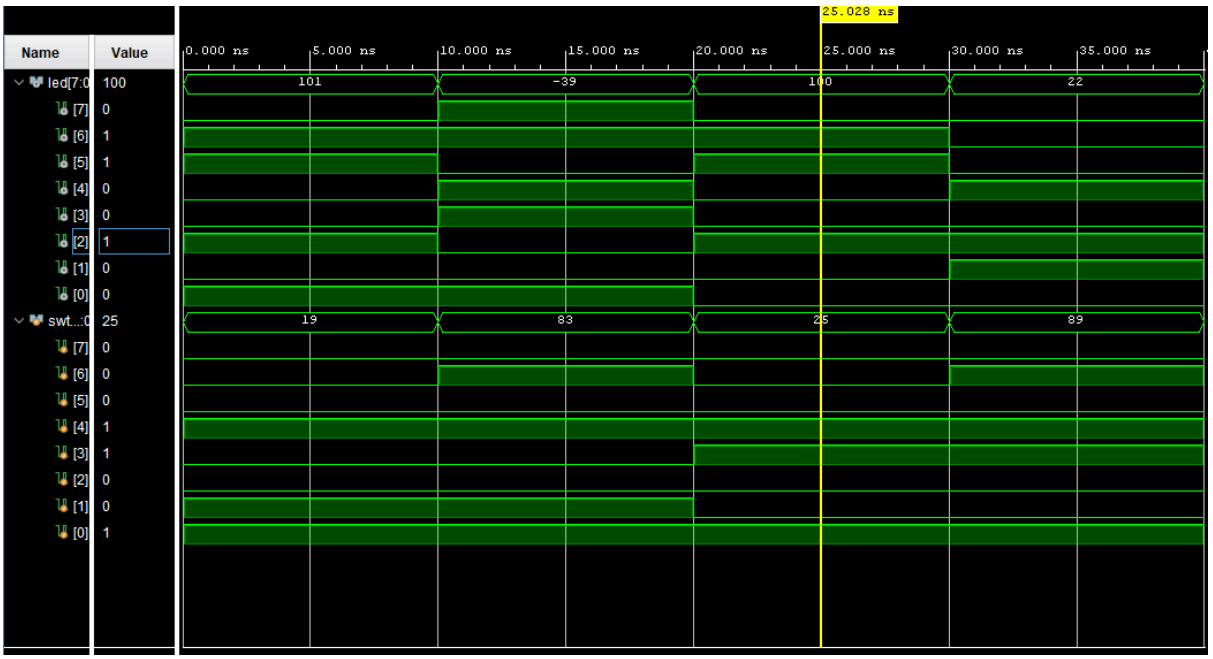


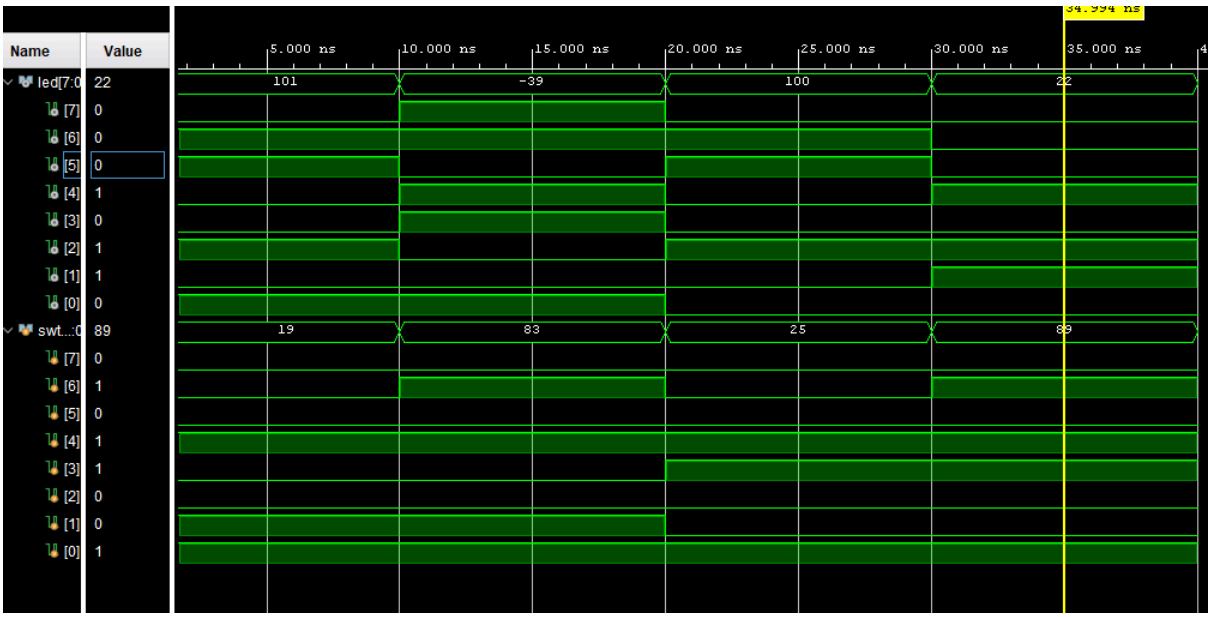


Adder/Subtracter Results




14.927 ns
Name
Value
0.000 ns
5.000 ns
10.000 ns
15.000 ns
20.000 ns
25.000 ns
30.000 ns
35.000 ns
led[7:0]
-39
[7] 1
[6] 1
[5] 0
[4] 1
[3] 1
[2] 0
[1] 0
[0] 1
swt...:0
83
[7] 0
[6] 1
[5] 0
[4] 1
[3] 0
[2] 0
[1] 1
[0] 1
101
-39
100
22
19
83
25
89


25.028 ns
Name
Value
0.000 ns
5.000 ns
10.000 ns
15.000 ns
20.000 ns
25.000 ns
30.000 ns
35.000 ns
led[7:0]
100
[7] 0
[6] 1
[5] 1
[4] 0
[3] 0
[2] 1
[1] 0
[0] 0
swt...:0
25
[7] 0
[6] 0
[5] 0
[4] 1
[3] 1
[2] 0
[1] 0
[0] 1
101
-39
100
22
19
83
25
89


Name
Value
led[7:0]
22
[7] 0
[6] 0
[5] 0
[4] 1
[3] 0
[2] 1
[1] 1
[0] 0
swt...0]
89
[7] 0
[6] 1
[5] 0
[4] 1
[3] 1
[2] 0
[1] 0
[0] 1
5.000 ns
10.000 ns
15.000 ns
20.000 ns
25.000 ns
30.000 ns
35.000 ns
101
-39
100
22
19
83
25
89