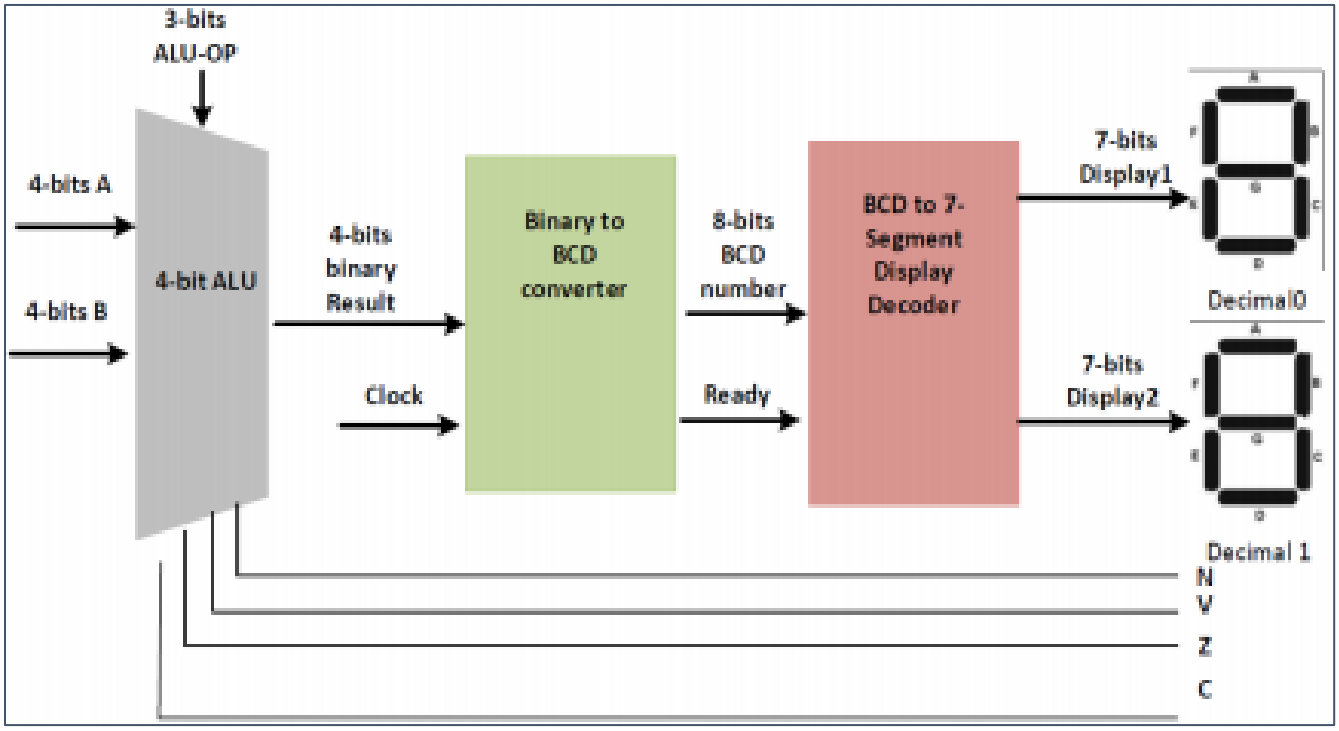
LAB 5: 4Bit ALU - BCD

Date: 17 de mayo de 2025



Proffesor: Dr. Francisco Javier Rodríguez Navarrete

Students: Cesar Eduardo Inda Ceniceros

Alonso Emmanuel López Macías

Team: 14

Introduction

An Arithmetic Logic Unit (ALU) is a fundamental block within the Central Processing Unit (CPU) of any digital system. Its main function is to perform arithmetic and logical operations on binary data. This lab focuses on the design and implementation of a 4-bit ALU using knowledge acquired in previous practices, integrating previously developed modules such as adders and multipliers.

The objective is for the student to understand the internal functioning of an ALU, reinforce the use of modular blocks, and experiment with operation control through selection signals. Additionally, a visual interface is implemented using 7-segment displays, either to represent values in hexadecimal or in decimal format using the Double Dabble algorithm, making the results easier to interpret on the FPGA. This lab represents an important step in consolidating skills in digital design, module reuse, and physical deployment on an FPGA, simulating the real behavior of an arithmetic embedded system. In the previous module, we had the opportunity to implement a 4-bit ALU, but this time we were required to display those values in BCD. This added complexity, which allowed us to gain more hands-on experience and apply additional logic to handle each of the necessary considerations.

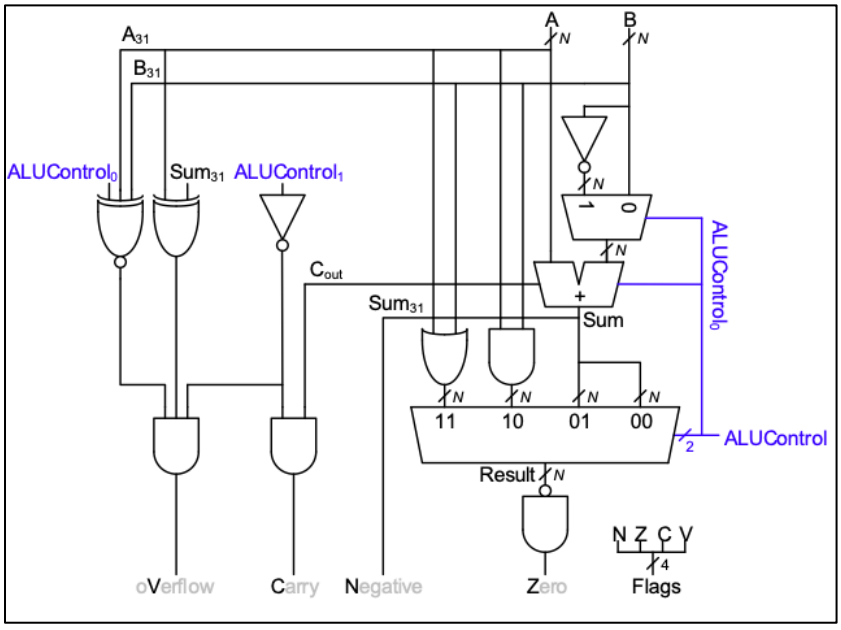


Figure: 1 4Bit ALU

Requirements

Design and implement a 4-bit ALU using the previous labs.

Step1.- Create a 4-bit ALU using the following diagram as a base:

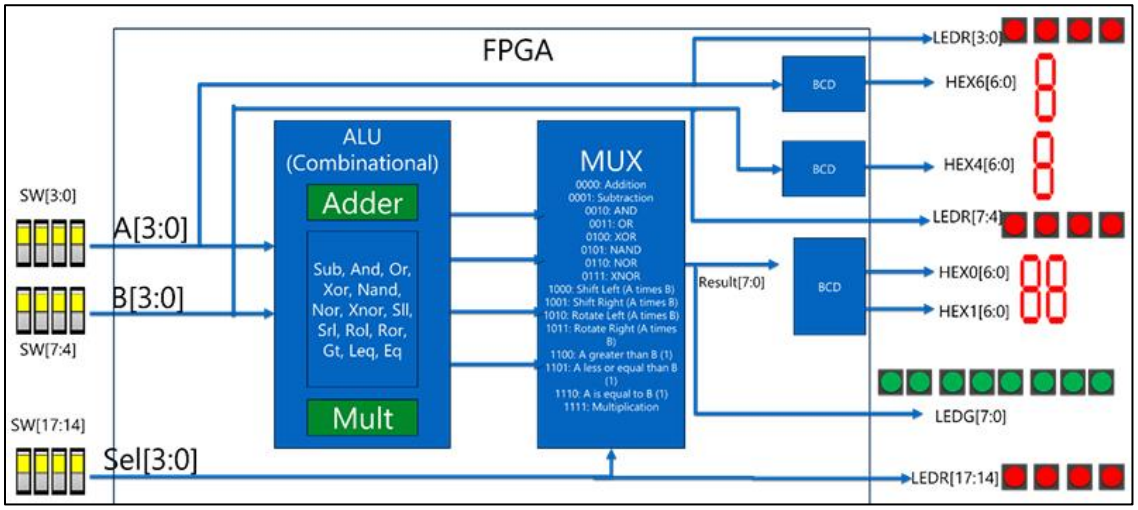


Figure: 2 ALU Diagram Requeriment

You will utilize the lab 4 for the multiplier and the lab 2 for the binary adder, the other operations will be added to the ALU as procedural with operators as needed. You can also opt to make the ALU purely behavioral (via a procedural block and case for example).

The ALU will implement all the following operations and will output based on sel signal:

- 0000: Addition
- 0001: Subtraction
- 0010: AND
- 0011: OR
- 0100: XOR
- 0101: NAND
- 0110: NOR
- 0111: XNOR
- 1000: Shift Left (A times B)
- 1001: Shift Right (A times B)
- 1010: Rotate Left (A times B)
- 1011: Rotate Right (A times B)
- 1100: A greater than B (if condition is met, output will be 1)
- 1101: A less or equal than B (if condition is met, output will be 1)
- 1110: A is equal to B (if condition is met, output will be 1)
- 1111: Multiplication

Step2.- Once you have the implementation you will create a testbench to test all operations and then implement it in FPGA and test combinations of the values to test out all the operations. After you have the implementation of the FPGA, instance it on a top module to add the double dabble BCD circuit you did, for 4 bits the max value is 15 and for 8 bits is 255, so you will be able to fit that into the FPGA without problem and observe base-10 outputs. You can also opt to use the 7segment decoder expanded to display HEX values instead from 0 to F on the 7segment, and it will mean you only need one 7segment to display inputs of 4-bits (0 to F) or for 8-bits (0 to FF) for the result, the choice is up to you. Here's an example of the pinout:

- Use the following ***switches*** for input ***A: SW [3] to SW [0]***.
- Use the following ***switches*** for input ***B: SW [7] to SW [4]***
- Use the following ***switches*** for input ***SEL: SW [17] to SW [14]***

For double dabble (BCD) + 7 Segment:

- Use the following ***7-segment*** display for decimal input ***A: HEX [7]***
- Use the following ***7-segment*** display for decimal input ***A: HEX [6]***
- Use the following ***7-segment*** display for decimal input ***B: HEX [5]***
- Use the following ***7-segment*** display for decimal input ***B: HEX [4]***
- Use the following ***7-segment*** display for hexadecimal output result ***HEX [2] to HEX [0]***

For 7 segment and HEX output from 0 to F:

- Use the following ***7-segment*** display for decimal input ***A: HEX [6]***
- Use the following ***7-segment*** display for decimal input ***B: HEX [4]***
- Use the following ***7-segment*** display for hexadecimal output result ***HEX [1] to HEX [0]***

- Use the following ***green leds*** for display of the output in binary format: ***LEDG [7] to LEDG [0]***
- Use the following ***red leds*** for display binary format of input ***A: LEDR [3] to LEDR [0]***
- Use the following ***red leds*** for display binary format of input ***B: LEDR [7] to LEDR [4]***
- Use the following ***red leds*** for display binary format of input ***SEL: LEDR [17] to LEDR [14]***

Development

The main considerations based on the selector-controlled operations are as follows.

0000: A + B	1000: Shift Left A << B
0001: A - B	1001: Shift Right A >> B
0010: A & B	1010: Rotate Left A <<< B
0011: A | B	1011: Rotate Right A >>> B
0100: A ^ B	1100: A > B → 1
0101: ~(A & B) (NAND)	1101: A <= B → 1
0110: ~(A | B) (NOR)	1110: A == B → 1
0111: ~(A ^ B) (XNOR)	1111: A * B

Figure: 3 Selector Combination

To develop this project, the first step was to create the logic for the operations based on the input values and the selector. This was done by considering the use of procedural elements when handling each of the operations for the ***4-bit ALU***. It was also taken into account that, for subtraction operations where the value of A was less than B, the result would be considered a signed negative value to indicate this on the display.

```
module alu4bit (
    input  [3:0] A,
    input  [3:0] B,
    input  [3:0] SEL,
    output reg [7:0] RESULT
);
    always @(*) begin
        case (SEL)
            4'b0000: RESULT = A + B;
            4'b0001: RESULT = A - B;
            4'b0010: RESULT = {4'b0000, A & B};
            4'b0011: RESULT = {4'b0000, A | B};
            4'b0100: RESULT = {4'b0000, A ^ B};
            4'b0101: RESULT = {4'b0000, ~(A & B) & 4'b1111}; // NAND
            4'b0110: RESULT = {4'b0000, ~(A | B) & 4'b1111}; // NOR
            4'b0111: RESULT = {4'b0000, ~(A ^ B) & 4'b1111}; // XNOR
            4'b1000: RESULT = A << B;
            4'b1001: RESULT = A >> B;
            4'b1010: RESULT = {4'b0000, (A << B) | (A >> (4 - B))}; // Rotate Left
            4'b1011: RESULT = {4'b0000, (A >> B) | (A << (4 - B))}; // Rotate Right
            4'b1100: RESULT = (A > B)  ? 8'b00000001 : 8'b00000000;
            4'b1101: RESULT = (A <= B) ? 8'b00000001 : 8'b00000000;
            4'b1110: RESULT = (A == B) ? 8'b00000001 : 8'b00000000;
            4'b1111: RESULT = A * B;
            default: RESULT = 8'b00000000;
        endcase
    end
endmodule
```

Figure: 4 alu4bit.v

The consideration for implementing the logic of the 7-segment BCDs was the following: it was based on the need to display both the input values and the result values.

```
module hex7seg(
    input  [3:0] num,
    output reg [6:0] seg
);
    always @(*) begin
        case (num)
            4'h0: seg = 7'b1000000;
            4'h1: seg = 7'b1111001;
            4'h2: seg = 7'b0100100;
            4'h3: seg = 7'b0110000;
            4'h4: seg = 7'b0011001;
            4'h5: seg = 7'b0010010;
            4'h6: seg = 7'b0000010;
            4'h7: seg = 7'b1111000;
            4'h8: seg = 7'b0000000;
            4'h9: seg = 7'b0010000;
            4'hA: seg = 7'b0001000;
            4'hB: seg = 7'b0000011;
            4'hC: seg = 7'b1000110;
            4'hD: seg = 7'b0100001;
            4'hE: seg = 7'b0000110;
            4'hF: seg = 7'b0001110;
            default: seg = 7'b1111111;
        endcase
    end
endmodule
```

Figure: 5 hex7seg.v

For this module, the "***Double Dabble algorithm also known as Shift-and-Add-3"*** was used to convert a binary number into its BCD representation. A 20-bit shift register initialized to zero is established, and 8 shifting cycles are performed. Before each shift, each group of 4 bits is evaluated; if a group is greater than or equal to 5, 3 is added to it. The values for the ones, tens, and hundreds of digits are also considered accordingly.

```
module binary_to_bcd (
    input [7:0] binary,
    output reg [3:0] hundreds, tens, ones
);
    integer i;
    reg [19:0] shift_reg;

    always @(binary) begin
        shift_reg = 20'd0;
        shift_reg[7:0] = binary;
        for (i = 0; i < 8; i = i + 1) begin
            if (shift_reg[11:8] >= 5) shift_reg[11:8] = shift_reg[11:8] + 3;
            if (shift_reg[15:12] >= 5) shift_reg[15:12] = shift_reg[15:12] + 3;
            if (shift_reg[19:16] >= 5) shift_reg[19:16] = shift_reg[19:16] + 3;
            shift_reg = shift_reg << 1;
        end
        ones = shift_reg[11:8];
        tens = shift_reg[15:12];
        hundreds = shift_reg[19:16];
    end
endmodule
```

Figure: 6 binary_to_bcd.v.

The main consideration for the functional logic in this part, for test in the FPGA we use this file of "top". This file contains all the implementation from the converters and the system for operations in the ALU.

```
module top (
    input [17:0] SW,
    output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7,
    output [7:0] LEDG,
    output [17:0] LEDR
);
    wire [3:0] A = SW[3:0];
    wire [3:0] B = SW[7:4];
    wire [3:0] SEL = SW[17:14];
    wire mode = SW[13];
    wire [7:0] RESULT;
    wire is_negative = RESULT[7];
    wire [7:0] abs_result = is_negative ? (~RESULT + 1'b1) : RESULT;

    wire [3:0] a_ones, a_tens;
    wire [3:0] b_ones, b_tens;
    wire [3:0] r_hundreds, r_tens, r_ones;

    alu4bit alu_inst(.A(A), .B(B), .SEL(SEL), .RESULT(RESULT));

    binary_to_bcd bcd_a(.binary({4'b0000, A}), .hundreds(), .tens(a_tens), .ones(a_ones));
    binary_to_bcd bcd_b(.binary({4'b0000, B}), .hundreds(), .tens(b_tens), .ones(b_ones));
    binary_to_bcd bcd_r(.binary(abs_result), .hundreds(r_hundreds), .tens(r_tens), .ones(r_ones));

    assign LEDG = RESULT;
    assign LEDR[3:0] = A;
    assign LEDR[7:4] = B;
    assign LEDR[17:14] = SEL;

    // HEX displays depending on mode
    // Result
    hex7seg h0(.num(mode ? r_ones     : RESULT[3:0]), .seg(HEX0));
    hex7seg h1(.num(mode ? r_tens     : RESULT[7:4]), .seg(HEX1));
    hex7seg h2(.num(mode ? r_hundreds : 4'd0),        .seg(HEX2));
    assign HEX3 = mode ? (is_negative ? 7'b0111111 : 7'b1111111) : 7'b1111111; // muestra guión si negativo

    // Input B
    hex7seg h4(.num(mode ? b_ones : B), .seg(HEX4));
    hex7seg h5(.num(mode ? b_tens : 4'd0), .seg(HEX5));

    // Input A
    hex7seg h6(.num(mode ? a_ones : A), .seg(HEX6));
    hex7seg h7(.num(mode ? a_tens : 4'd0), .seg(HEX7));
endmodule
```

Figure: 7 top. v

Testbench

The testbench was defined as follows, in a way that allows testing the DUT based on its behavior with respect to the selector and the defined input values. In this case, the" ***alu4bit uut"*** module is instantiated. Once this is completed, the initialization section allows defining the time values in the monitor, thereby establishing many key points to assess the representation of the different values.

```
module tb_alu4bit;
    reg [3:0] A, B, SEL;
    wire [7:0] RESULT;

    alu4bit uut (
        .A(A), .B(B), .SEL(SEL), .RESULT(RESULT)
    );

    initial begin
        $display("Time |  A  |  B  | SEL  | RESULT");
        for (integer i = 0; i < 16; i = i + 1) begin
            A = 4'd3;
            B = 4'd2;
            SEL = i[3:0]; // Aquí usamos el índice como selector
            #10 $display("%4t | %2d  | %2d  | %4b | %3d", $time, A, B, SEL, RESULT);
        end
        $stop;
    end
endmodule
```

Figure: 8 tb_alu4bit.v

Debug in Model Sim:

We can observe the result in the Model Sim, after add the testbench in the "assignments". For test this part we need set ***"alu4bit" as a top and "tb_alu4bit"*** in testbench section. This is important because we can observe a error in the simulation if not are set correctly the main program section and the files correctly configured for test the different operations.

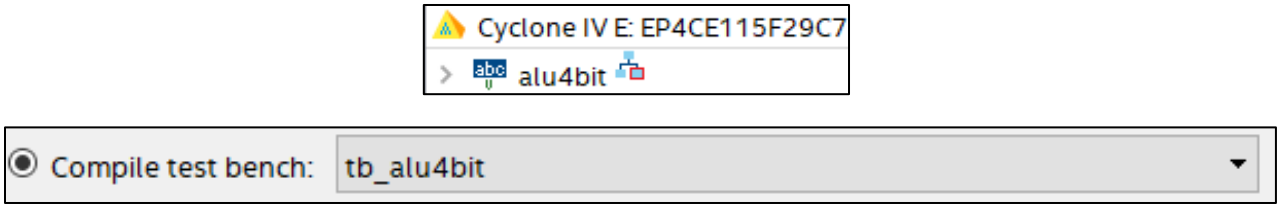


Figure: 9 Testbench Considerations

We can observe in ***ModelSim simulation*** the following:

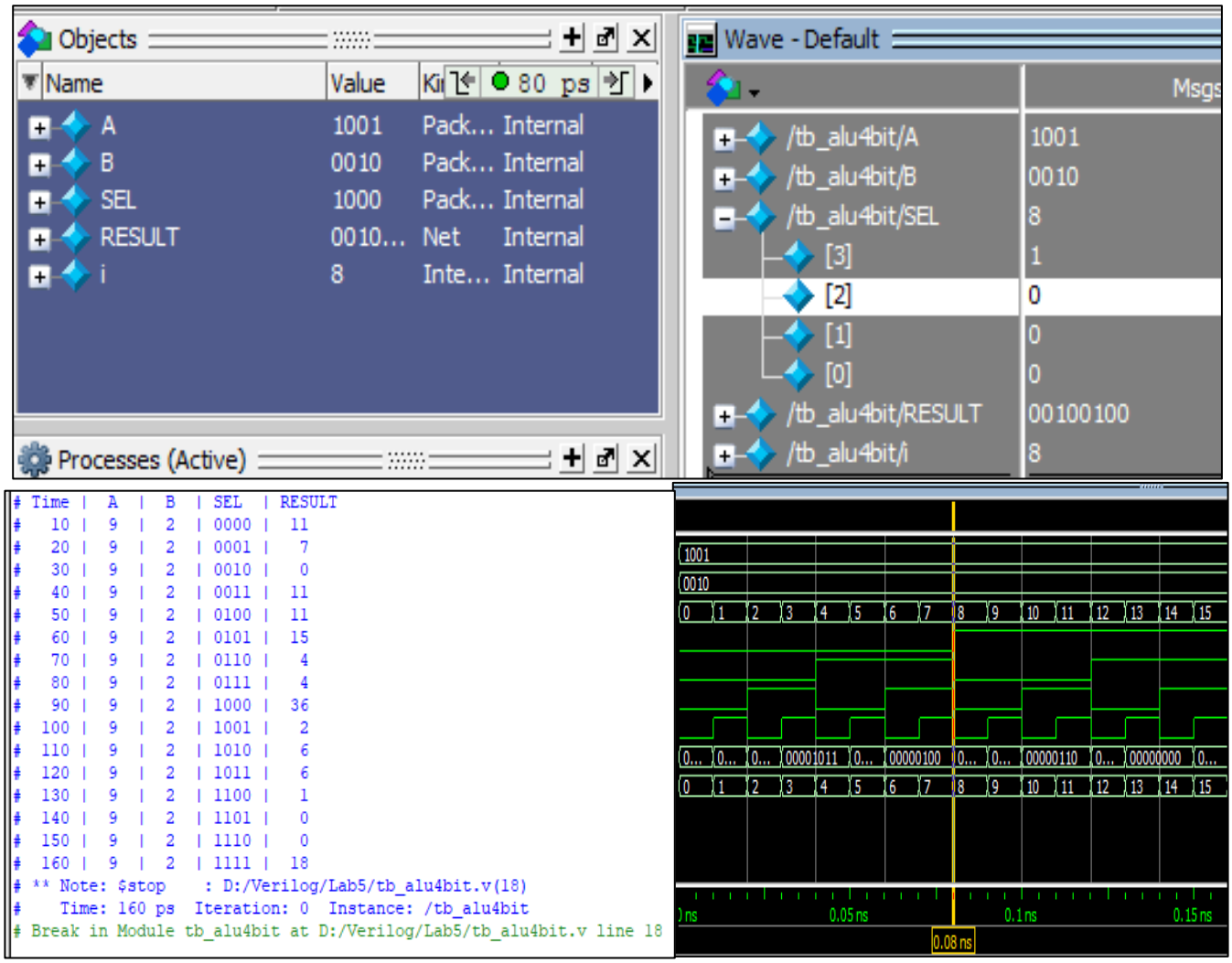


Figure: 10 ModelSim_ALU_4Bit

We can observe in the monitor display that the operations are correctly set for each one from the values this represent a correct instantiation and correct definition from each one alu section with the different operations. This established the correct format and parameters for consider the recommendations in the values and properties for apply the shifter and logical operations.

FPGA Implementation

Once the program was completed, the compilation process was carried out in order to generate the pin assignment plan and correctly map the pins according to the program's requirements.

Flow Status	Successful - Wed May 21 21:30:56 2025
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	alu_bcd
Top-level Entity Name	alu4bit
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	137 / 114,480 (< 1 %)
Total registers	0
Total pins	20 / 529 (4 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figure: 11 Compilation Brief

The pins consideration was assigned as following, in this case for bcd 7 segment we considered the 7 bcd's but only for a brief illustration we can set a section from "HEX0", but in the planner are set from ***HEX0 to HEX7.***

HEX0[1]	Unknown	PIN_F22
HEX0[2]	Unknown	PIN_E17
HEX0[3]	Unknown	PIN_L26
HEX0[4]	Unknown	PIN_L25
HEX0[5]	Unknown	PIN_J22
HEX0[6]	Unknown	PIN_H22

LEDG[0]	Unknown	PIN_E21
LEDG[1]	Unknown	PIN_E22
LEDG[2]	Unknown	PIN_E25
LEDG[3]	Unknown	PIN_E24
LEDG[4]	Unknown	PIN_H21
LEDG[5]	Unknown	PIN_G20
LEDG[6]	Unknown	PIN_G22

LEDR[0]	Unknown	PIN_G19
LEDR[1]	Unknown	PIN_F19
LEDR[2]	Unknown	PIN_E19
LEDR[3]	Unknown	PIN_F21
LEDR[4]	Unknown	PIN_F18
LEDR[5]	Unknown	PIN_E18
LEDR[6]	Unknown	PIN_J19
LEDR[7]	Unknown	PIN_H19
LEDR[8]	Unknown	PIN_J17
LEDR[9]	Unknown	PIN_G17
LEDR[10]	Unknown	PIN_J15
LEDR[11]	Unknown	PIN_H16
LEDR[12]	Unknown	PIN_J16
LEDR[13]	Unknown	PIN_H17
LEDR[14]	Unknown	PIN_F15
LEDR[15]	Unknown	PIN_G15
LEDR[16]	Unknown	PIN_G16
LEDR[17]	Unknown	PIN_H15

SW[0]	Unknown	PIN_AB28
SW[1]	Unknown	PIN_AC28
SW[2]	Unknown	PIN_AC27
SW[3]	Unknown	PIN_AD27
SW[4]	Unknown	PIN_AB27
SW[5]	Unknown	PIN_AC26
SW[6]	Unknown	PIN_AD26
SW[7]	Unknown	PIN_AB26
SW[8]	Unknown	PIN_AC25
SW[9]	Unknown	PIN_AB25
SW[10]	Unknown	PIN_AC24
SW[11]	Unknown	PIN_AB24
SW[12]	Unknown	PIN_AB23
SW[13]	Unknown	PIN_AA24
SW[14]	Unknown	PIN_AA23
SW[15]	Unknown	PIN_AA22
SW[16]	Unknown	PIN_Y24
SW[17]	Unknown	PIN_Y23

Figure: 12 PinPlanner

The RTL view can be observed as follows.

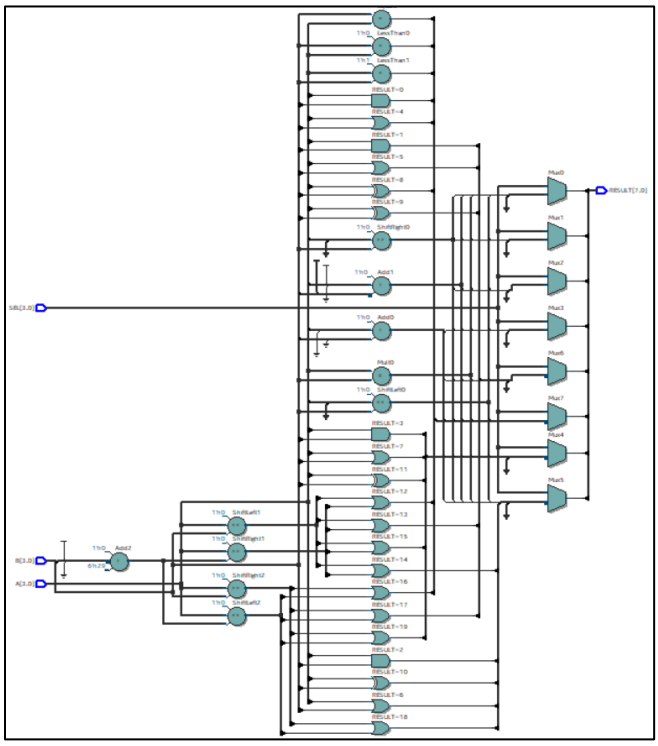


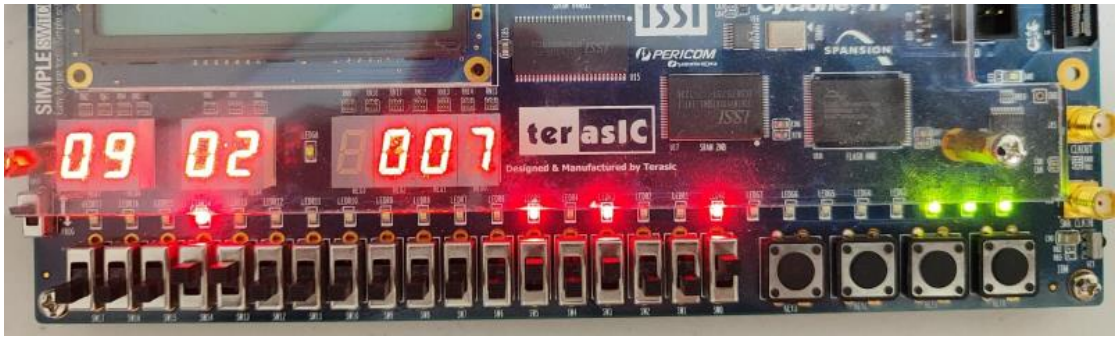
Figure: 13 RTL_Viewver

FPGA Results

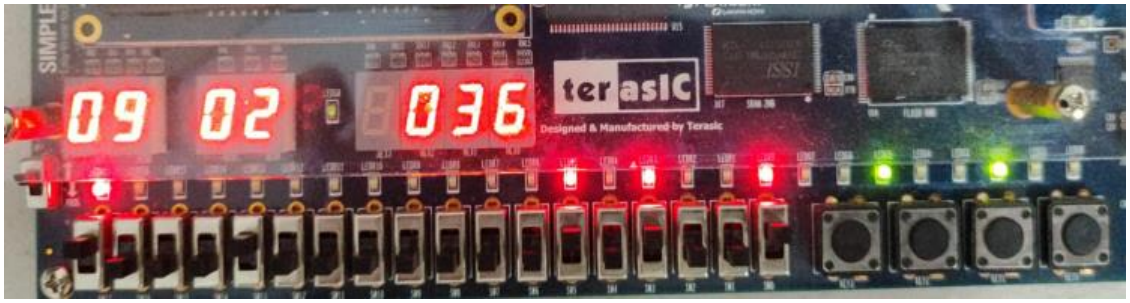
To perform tests on the FPGA board, it is necessary ***to set "top" as the main entity.*** Once the program is loaded, the corresponding tests can be carried out. For these considerations, only a few photographs will be taken to represent some of the many ALU operations. An important point to consider is that ***SW13 will allow switching between hexadecimal and decimal values. When SW13 is set high, it will be possible to correctly observe the values on the BCD displays.***

All the values for "***A" and "B***" for practice test ***are constants in this A=1001(9) and B=0010(2)***

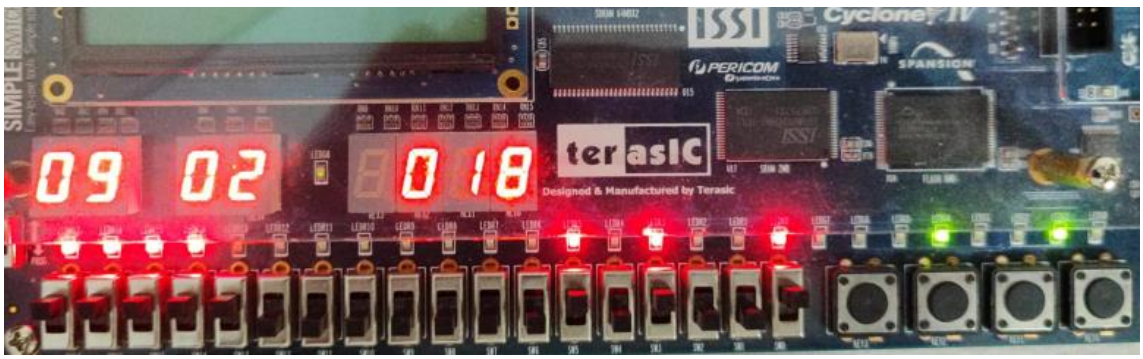
1.- SELECTOR = 0001 (Subtraction) A=1001, B=0010



2.- SELECTOR = 1000 (Shift Left (A times B) A=1001, B=0010



3.- SELECTOR = 1111 (Multiplication) A=1001, B=0010



GitHub Repository

https://github.com/CeicGitHub/Fundamentals_Of_Digital_Design_FPGA_2/tree/Lab5_ALU_BCD

Conclusion / Issues

Cesar Eduado Inda Ceniceros

One of the main challenges was the correct integration of all the modules from previous labs and how to incorporate them into a single Arithmetic Logic Unit (ALU). Managing the operations within the procedural case block, as well as designing shifts and rotations according to the operand-specified amount, were also complex tasks. Instantiating the code and understanding the hierarchy for file communication, along with analyzing the selector combinations for each operation, required careful attention. This lab allowed me to reinforce my understanding of the internal functioning of an ALU and of design in Verilog HDL. In addition, this time the ALU included a BCD module, which made the project more challenging. It also helped me better understand how to create and use different testbenches to simulate the design using Altera's modeling system.

Alonso Emmanuel Lopez Macias

One of the biggest challenges I personally experienced was when dealing with the file structure for control logic or shifts, as there are many operations to consider regarding the arithmetic logic unit. The main difficulty was understanding that much of its functionality is based precisely on grasping the principles of basic operations. Moreover, this process allowed me to learn a lot from a different perspective building the ALU now from the logic of Verilog programming. It was different, even though in Module 1 we used diagrams, which made understanding the functionality more visual. This time, it required me to begin understanding it through callbacks or instances integrated across different files, due to the considerable number of operations involved. Thanks to this practice, the functioning of testbenches and the process of generating the program logic became clearer to me. It's interesting to now have both perspectives with this module.