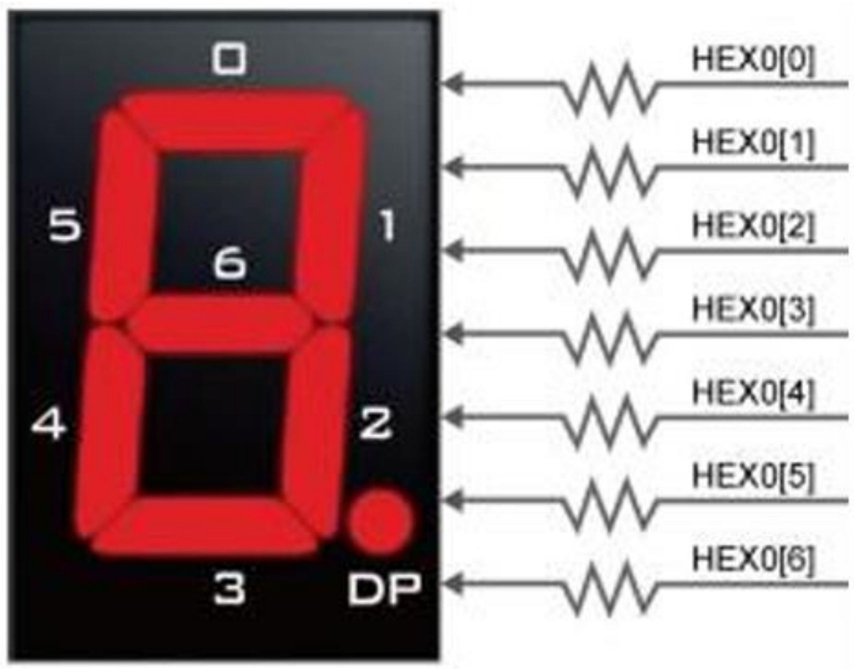
LAB 1: BCD-7segment Decoder

Date: 7 de mayo de 2025



Proffesor: Dr. Francisco Javier Rodríguez Navarrete

Students: Cesar Eduardo Inda Ceniceros

Alonso Emmanuel López Macías

Team: 14

Introduction

This lab focuses on the design and implementation of a BCD to 7-segment display decoder using an FPGA as the development platform. Decoders are fundamental digital circuits that identify specific combinations of input bits and generate corresponding outputs based on defined conditions. In this case, the system receives a binary-coded decimal (BCD) input and converts it into a human-readable representation using 7-segment displays, which are commonly used in embedded systems. The objective of this activity is to help students understand the usefulness of decoders in digital hardware, strengthen their logic design skills, and become familiar with the Double Dabble encoding technique used to represent numbers greater than a single decimal digit. The programming was carried out using the language studied during the course, Verilog HDL, with the aim of continuing to practice its syntax and hardware-oriented structure. This allowed for the implementation of each required design throughout the course, applying different programming approaches and adhering to best practices in hardware description using this language.

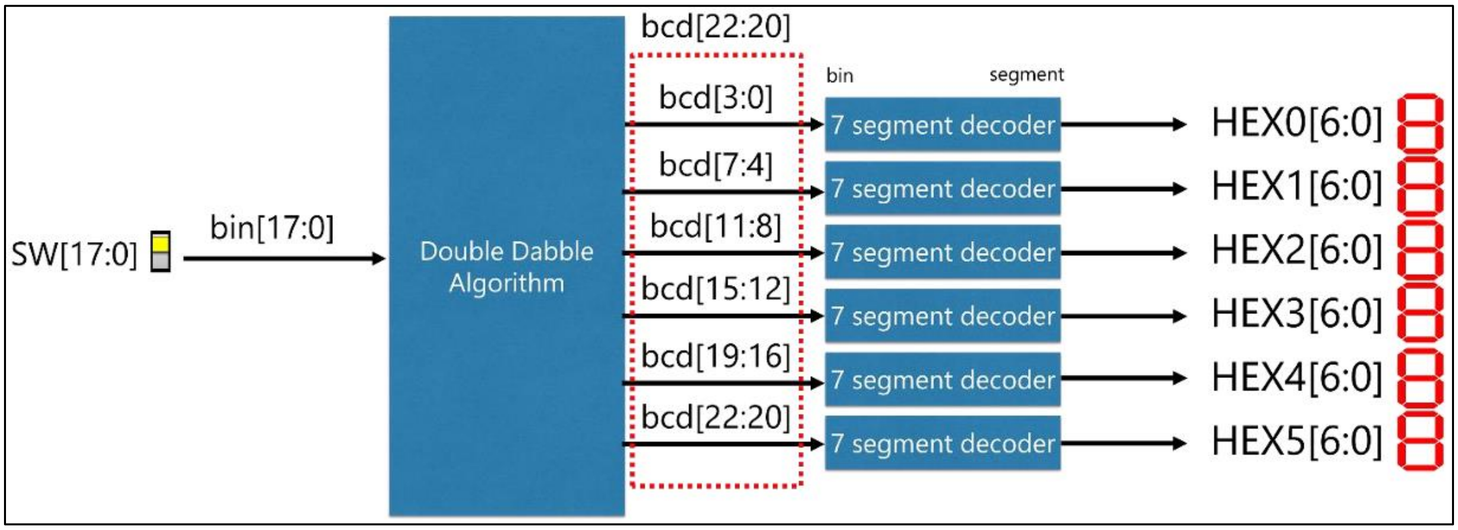


Figure: 1 Decoder - Bcd 7 Segment

Requirements

Step1.- The DE2-115 7 segment displays are common anode, this means that the corresponding connected line must be set to 0 in order for the display to light up that segment.

Step2.- Connect the 4-bit input to the 4-bit output of the double dabble algorithm that can split any number from 0 to 9,999,999, and use the 18 slide switches to represent any number from 0 to 262,143 as the input of the double dabble algorithm.

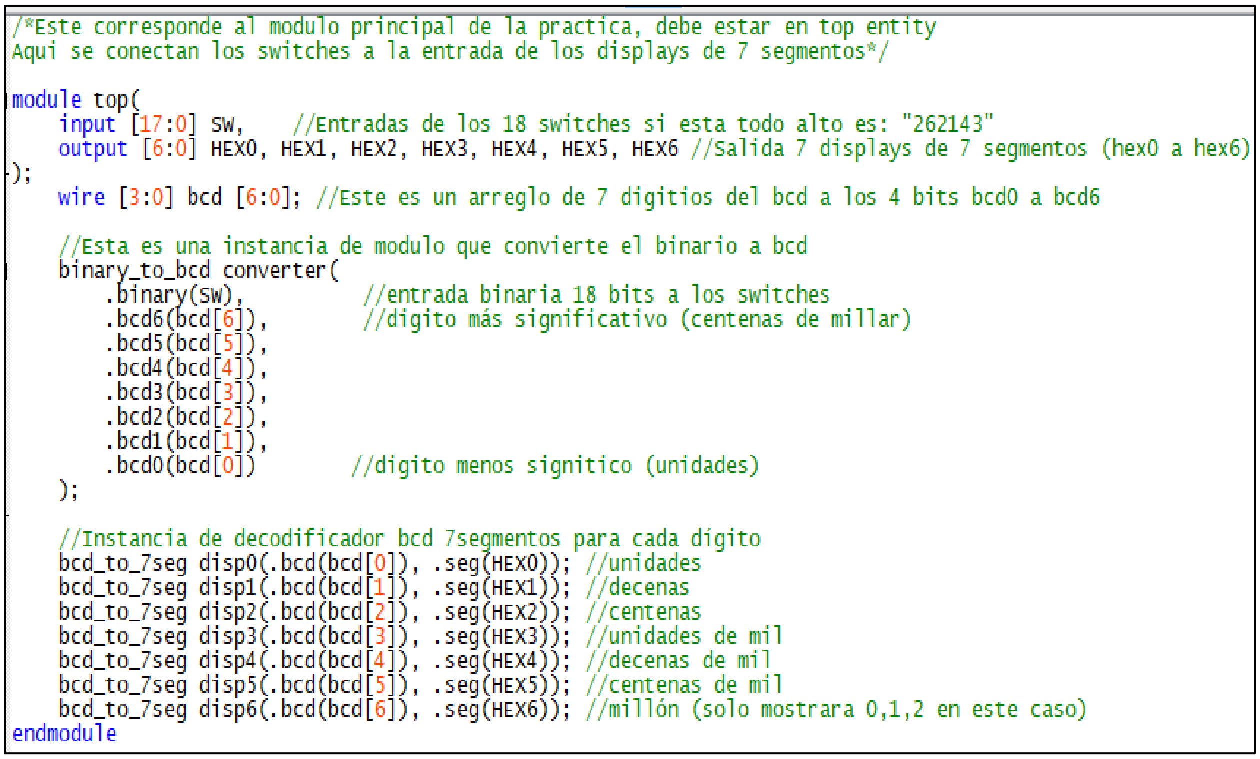
Step3.- The circuit should look a bit.

Step4.- Implement the circuit on the FPGA.

• Use the following switches for input bin: SW [17] to SW [0]
• Use the following 7 segment displays for outputs: HEX7 to HEX0

Development

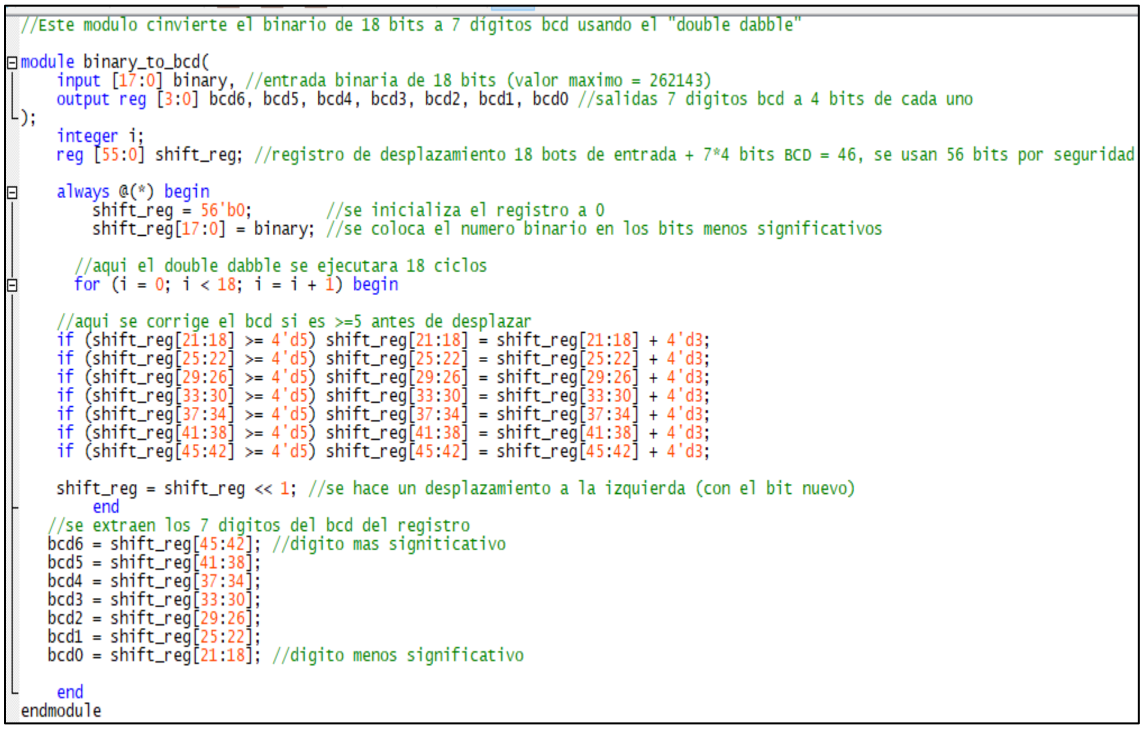
The first step was to create the main logic of the practice, is the main entity.



```
/*Este corresponde al modulo principal de la practica, debe estar en top entity
Aqui se conectan los switches a la entrada de los displays de 7 segmentos*/

module top(
    input [17:0] SW,    //Entradas de los 18 switches si esta todo alto es: "262143"
    output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6 //Salida 7 displays de 7 segmentos (hex0 a hex6)
);
    wire [3:0] bcd [6:0]; //Este es un arreglo de 7 digitios del bcd a los 4 bits bcd0 a bcd6

    //Esta es una instancia de modulo que convierte el binario a bcd
    binary_to_bcd converter(
        .binary(SW),          //entrada binaria 18 bits a los switches
        .bcd6(bcd[6]),        //digito más significativo (centenas de millar)
        .bcd5(bcd[5]),
        .bcd4(bcd[4]),
        .bcd3(bcd[3]),
        .bcd2(bcd[2]),
        .bcd1(bcd[1]),
        .bcd0(bcd[0])        //digito menos signitico (unidades)
    );

    //Instancia de decodificador bcd 7segmentos para cada digito
    bcd_to_7seg disp0(.bcd(bcd[0]), .seg(HEX0)); //unidades
    bcd_to_7seg disp1(.bcd(bcd[1]), .seg(HEX1)); //decenas
    bcd_to_7seg disp2(.bcd(bcd[2]), .seg(HEX2)); //centenas
    bcd_to_7seg disp3(.bcd(bcd[3]), .seg(HEX3)); //unidades de mil
    bcd_to_7seg disp4(.bcd(bcd[4]), .seg(HEX4)); //decenas de mil
    bcd_to_7seg disp5(.bcd(bcd[5]), .seg(HEX5)); //centenas de mil
    bcd_to_7seg disp6(.bcd(bcd[6]), .seg(HEX6)); //millón (solo mostrara 0,1,2 en este caso)
endmodule
```

Figure: 2 Module Top

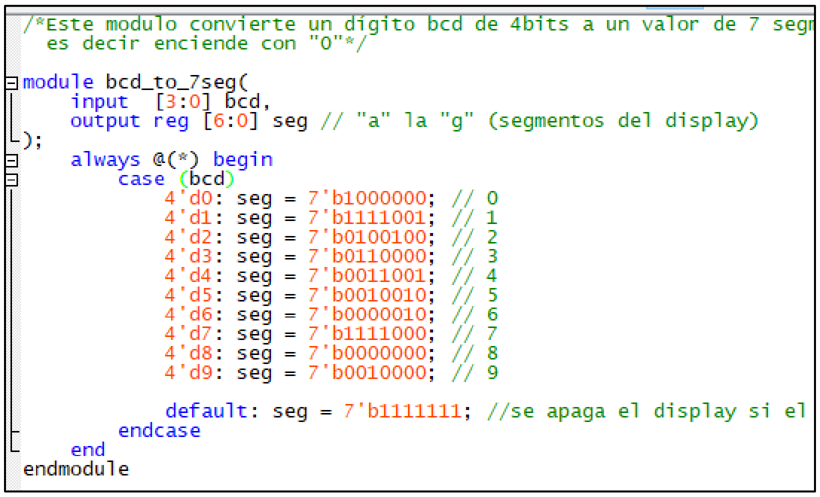
The next aspect to consider is the part of the module that converts the 18-bit binary value into the 7 BCD digits using the Double Dabble algorithm, in order to generate the required binary inputs and logical shifts.



```
//Este modulo cinvierte el binario de 18 bits a 7 digitos bcd usando el "double dabble"

module binary_to_bcd(
    input [17:0] binary, //entrada binaria de 18 bits (valor maximo = 262143)
    output reg [3:0] bcd6, bcd5, bcd4, bcd3, bcd2, bcd1, bcd0 //salidas 7 digitos bcd a 4 bits de cada uno
);
    integer i;
    reg [55:0] shift_reg; //registro de desplazamiento 18 bots de entrada + 7*4 bits BCD = 46, se usan 56 bits por seguridad

    always @(*) begin
        shift_reg = 56'b0;        //se inicializa el registro a 0
        shift_reg[17:0] = binary; //se coloca el numero binario en los bits menos significativos

      //aqui el double dabble se ejecutara 18 ciclos
      for (i = 0; i < 18; i = i + 1) begin

    //aqui se corrige el bcd si es >=5 antes de desplazar
    if (shift_reg[21:18] >= 4'd5) shift_reg[21:18] = shift_reg[21:18] + 4'd3;
    if (shift_reg[25:22] >= 4'd5) shift_reg[25:22] = shift_reg[25:22] + 4'd3;
    if (shift_reg[29:26] >= 4'd5) shift_reg[29:26] = shift_reg[29:26] + 4'd3;
    if (shift_reg[33:30] >= 4'd5) shift_reg[33:30] = shift_reg[33:30] + 4'd3;
    if (shift_reg[37:34] >= 4'd5) shift_reg[37:34] = shift_reg[37:34] + 4'd3;
    if (shift_reg[41:38] >= 4'd5) shift_reg[41:38] = shift_reg[41:38] + 4'd3;
    if (shift_reg[45:42] >= 4'd5) shift_reg[45:42] = shift_reg[45:42] + 4'd3;

    shift_reg = shift_reg << 1; //se hace un desplazamiento a la izquierda (con el bit nuevo)
        end
   //se extraen los 7 digitos del bcd del registro
   bcd6 = shift_reg[45:42]; //digito mas significativo
   bcd5 = shift_reg[41:38];
   bcd4 = shift_reg[37:34];
   bcd3 = shift_reg[33:30];
   bcd2 = shift_reg[29:26];
   bcd1 = shift_reg[25:22];
   bcd0 = shift_reg[21:18]; //digito menos significativo

    end
endmodule
```

Figure: 3 Module Binary-Bcd

After that, the module responsible for converting a 4-bit BCD value to a 7-segment output for a common anode display was defined. This section includes part of the always block mechanism used to handle the different display segment cases.



```
/*Este modulo convierte un digito bcd de 4bits a un valor de 7 seg
  es decir enciende con "0"*/

module bcd_to_7seg(
    input  [3:0] bcd,
    output reg [6:0] seg // "a" la "g" (segmentos del display)
);
    always @(*) begin
        case (bcd)
            4'd0: seg = 7'b1000000; // 0
            4'd1: seg = 7'b1111001; // 1
            4'd2: seg = 7'b0100100; // 2
            4'd3: seg = 7'b0110000; // 3
            4'd4: seg = 7'b0011001; // 4
            4'd5: seg = 7'b0010010; // 5
            4'd6: seg = 7'b0000010; // 6
            4'd7: seg = 7'b1111000; // 7
            4'd8: seg = 7'b0000000; // 8
            4'd9: seg = 7'b0010000; // 9

            default: seg = 7'b1111111; //se apaga el display si el
        endcase
    end
endmodule
```

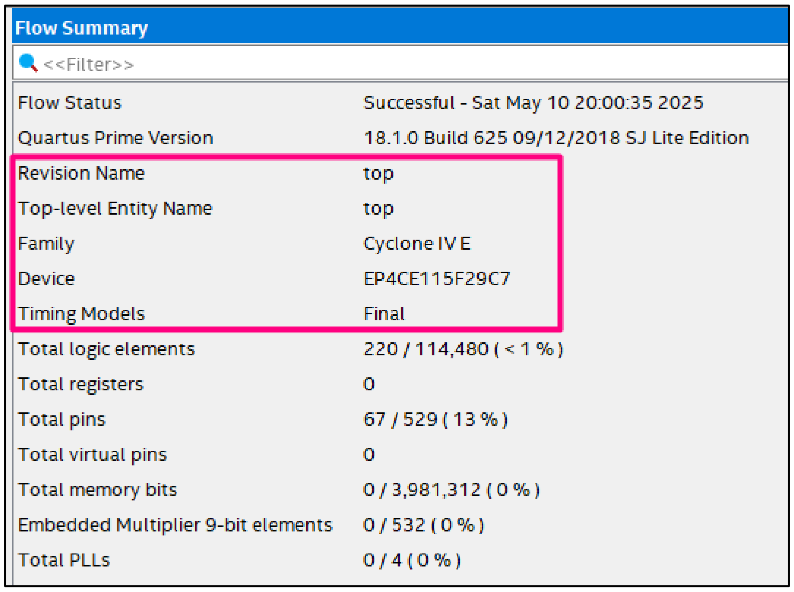
Figure: 4 Module bcd-7segment

Testbench

On this occasion, the code was loaded directly onto the FPGA board; therefore, a testbench was not used.

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.



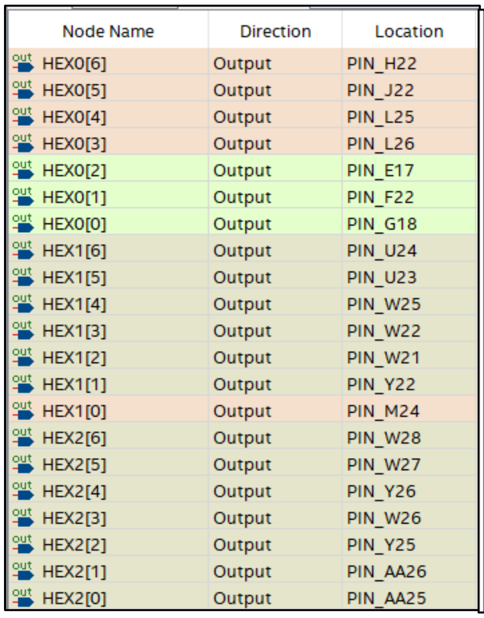
Flow Summary

<<Filter>>

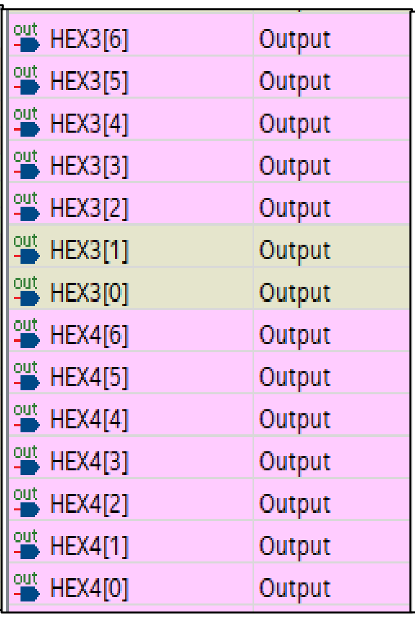
Flow Status	Successful - Sat May 10 20:00:35 2025
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	top
Top-level Entity Name	top
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	220 / 114,480 (< 1 %)
Total registers	0
Total pins	67 / 529 (13 %)
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: 5 Compilation Brief

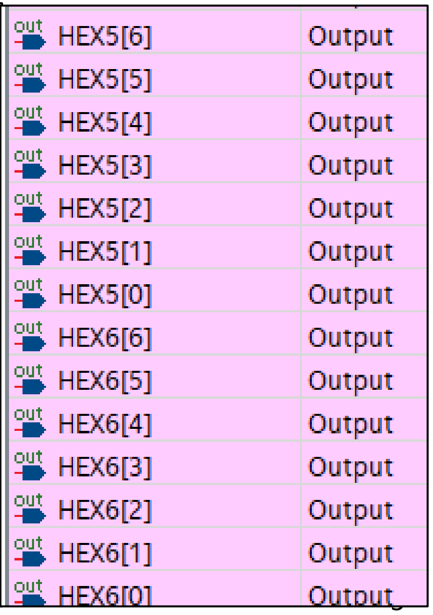
Then the pines were assigned



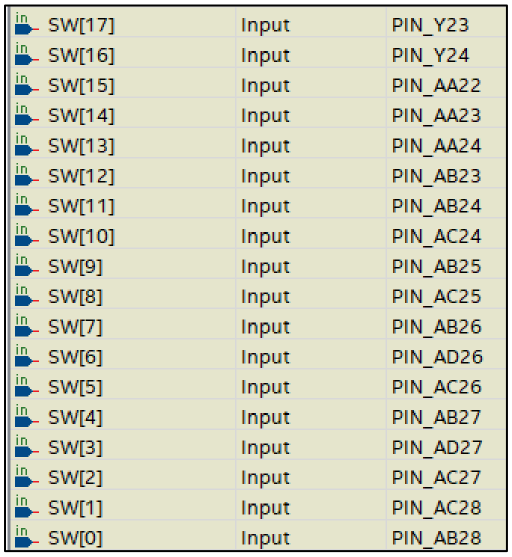
Node Name	Direction	Location
HEX0[6]	Output	PIN_H22
HEX0[5]	Output	PIN_J22
HEX0[4]	Output	PIN_L25
HEX0[3]	Output	PIN_L26
HEX0[2]	Output	PIN_E17
HEX0[1]	Output	PIN_F22
HEX0[0]	Output	PIN_G18
HEX1[6]	Output	PIN_U24
HEX1[5]	Output	PIN_U23
HEX1[4]	Output	PIN_W25
HEX1[3]	Output	PIN_W22
HEX1[2]	Output	PIN_W21
HEX1[1]	Output	PIN_Y22
HEX1[0]	Output	PIN_M24
HEX2[6]	Output	PIN_W28
HEX2[5]	Output	PIN_W27
HEX2[4]	Output	PIN_Y26
HEX2[3]	Output	PIN_W26
HEX2[2]	Output	PIN_Y25
HEX2[1]	Output	PIN_AA26
HEX2[0]	Output	PIN_AA25



HEX3[6]	Output
HEX3[5]	Output
HEX3[4]	Output
HEX3[3]	Output
HEX3[2]	Output
HEX3[1]	Output
HEX3[0]	Output
HEX4[6]	Output
HEX4[5]	Output
HEX4[4]	Output
HEX4[3]	Output
HEX4[2]	Output
HEX4[1]	Output
HEX4[0]	Output



HEX5[6]	Output
HEX5[5]	Output
HEX5[4]	Output
HEX5[3]	Output
HEX5[2]	Output
HEX5[1]	Output
HEX5[0]	Output
HEX6[6]	Output
HEX6[5]	Output
HEX6[4]	Output
HEX6[3]	Output
HEX6[2]	Output
HEX6[1]	Output
HEX6[0]	Output



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

Figure: 6 Pin Planner

The RTL view can be observed as follows.

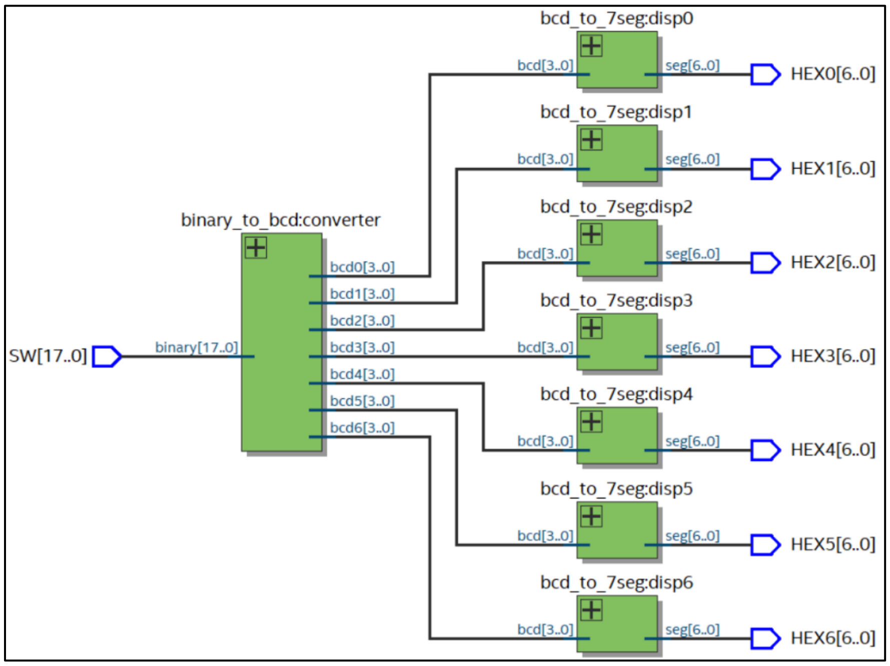


Figure: 7 RTL

GitHub Repository

https://github.com/CeicGitHub/Fundamentals_Of_Digital_Design_FPGA_2/tree/Lab1_BcdDecoder

Conclusion / Issues

Cesar Eduado Inda Ceniceros

Some of the challenges I faced while completing this lab involved understanding how to properly create the instances within the BCD decoder. Unlike previous labs from the first module, where we only had to consider values from 0 to 15, this time we needed to handle a much wider range—from units up to the millions. Another complex part was understanding the Double Dabble algorithm and correctly implementing the for loop to execute the required number of shifts. It was also challenging to manage the shifting operations and to grasp the purpose and usage of the 56-bit register defined as "reg [55:0]". Personally, I had previous experience programming in Verilog thanks to a former master's colleague from ITESO who worked at Intel. However, I had not had the opportunity to explore the deeper conceptual aspects, including the use of testbenches covered in other labs. Through this project, I learned more about logic design and realized that there are multiple ways to achieve the same functionality on an FPGA. I also gained a clearer understanding of the Double Dabble algorithm and how to leverage multiple modules throughout a project to support implementation.

Alonso Emmanuel Lopez Macias

This practice personally allowed me to deepen my understanding of Verilog. It became much clearer to me that programming is not limited to the method we used previously with BDF diagrams in the earlier module. I also realized how certain instructions can be defined within the code—for example, the use of integer in this lab for the for loop, which is similar to how it is used in C. Additionally, the concept of shifting operations became clearer to me, as did the use of the always block to create sequential or conditional behavior. In the context of registers and event-driven logic, this block plays a crucial role. I also understood that, because this was a more complex and robust implementation, the learning impact was greater compared to previous labs.