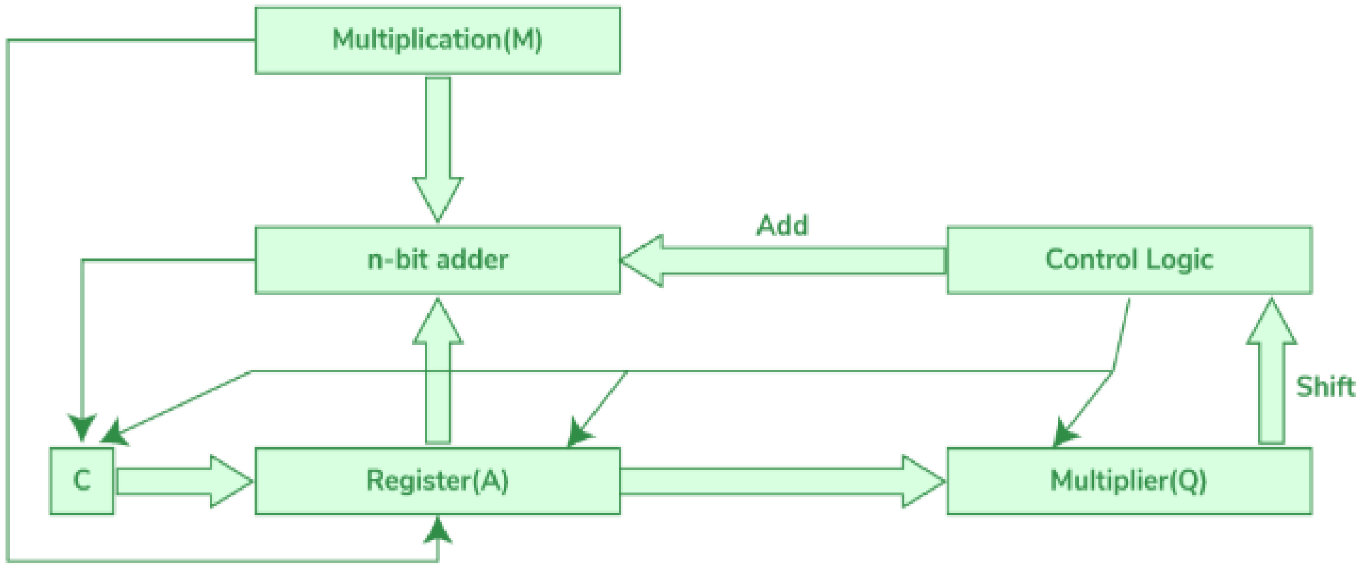
LAB 4: Parameterized N-Bit Structural Multiplier

Date: 15 de mayo de 2025



Proffesor: Dr. Francisco Javier Rodríguez Navarrete

Students: Cesar Eduardo Inda Ceniceros

Alonso Emmanuel López Macías

Team: 14

Introduction

In modern digital systems, binary multiplication is a fundamental operation found in arithmetic logic units (ALUs), processors, and digital signal processors (DSPs). Digital multipliers enable efficient and accurate computations in hardware architectures, making their structural implementation a key component in digital design.

The objective of this lab is to develop a 4-bit structural multiplier and then extend it to a parameterized N-bit version. Through this implementation, the student will become familiar with essential concepts such as the use of buses, structural logic description, module instantiation, and the use of generate blocks to efficiently replicate hardware structures. Additionally, the design will include a functional comparison between the structural multiplier and a reference model using the “*” operator, verifying its behavior through extensive testing. Finally, the lab includes the deployment of the design on an FPGA, where both inputs and the output will be displayed in hexadecimal format using 7-segment displays, integrating knowledge acquired in previous labs.

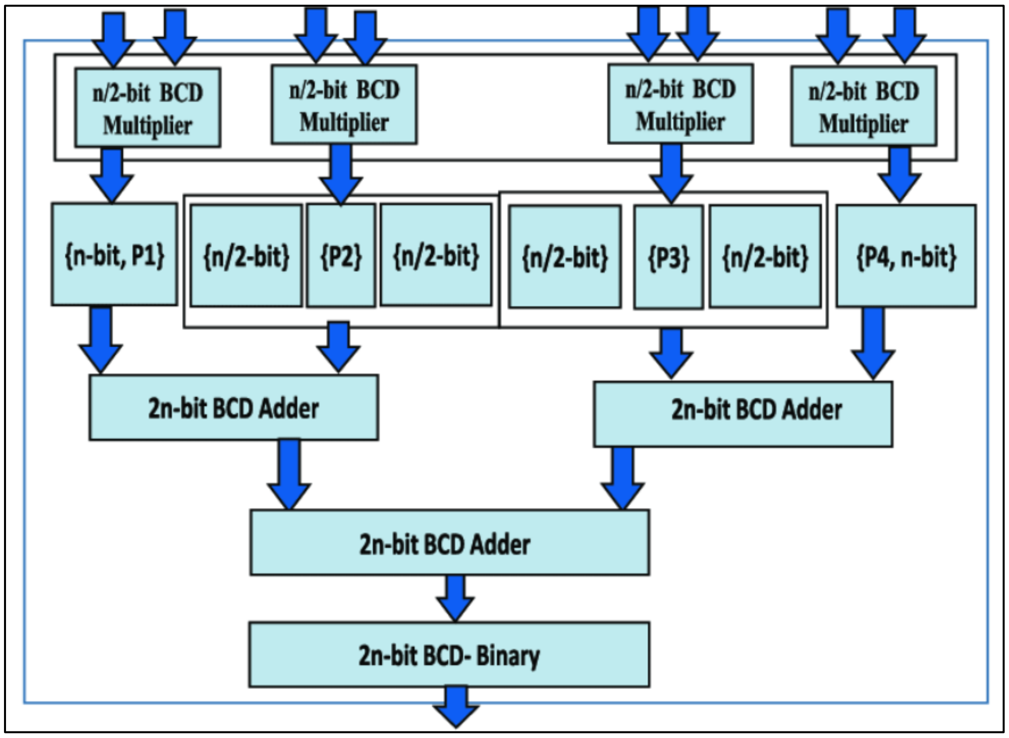


Figure: 1 N-bit Multiplier

Requirements

Step1.- Create a 4-bit multiplier following the following diagram:

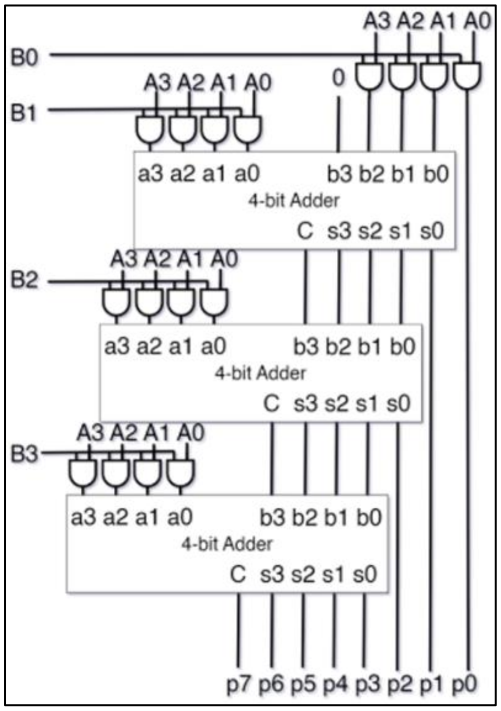


Figure: 2 Multiplier Diagram

Step2.- Once you have the 4-bit multiplier, you will create a function called "check multiplication" where we will use the * operator multiplier. We will create a testbench and instantiate both our structural multiplier and use our function that uses the operator. In this testbench we will send stimuli to both (at least 1000) and then we will compare the output of both circuits. We will use the * multiplier function as our reference model and then print a "TEST PASSED" if all comparations are ok or "TEST FAILED" if one of the comparations are not.

Step3.- After we have a working 4-bit multiplier, we will create a N-bit multiplier using this as a base. Create a draft of how the connections of the instances and wires would look like if they were named variables (for example: i, j, k) to figure out how you have to calculate the index for each of the iterations of the circuit for N-bits, use a generate block to generate the needed wires and instantiations.

We will add parameterization to the * operator multiplier as well.

In the testbench we will modify it to use parameterized inputs and outputs and then we will set the N of the multiplier to 32 and 64 and re-run the test to verify the functionality.

Step4.- Once it's complete we will use the circuit to implement the multiplier in the FPGA, we will use lab 1 once again for more readability in the output, however we will only use the decodifier (not the double dabble algorithm) because we want to see both input and output as hexadecimal. For this the decodifier will have to be changed to support a, b, c, d, e, f:

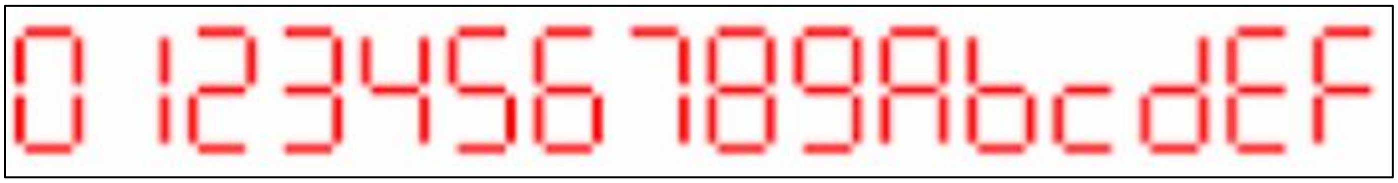


Figure: 3 Bcd Requeriment

Remember, each of these settings has a different parameterization and a different synthesized circuit. For this, a testbench that verifies all 3 types is mandatory since the 32-bit, 4 CHANNEL, and 18-bit, 16 CHANNELS won't fit in the I/O of the FPGA.

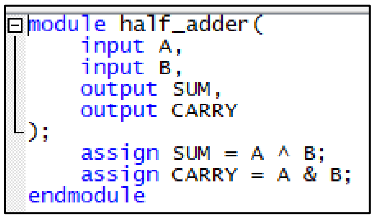
The decodifier will use case statements and procedural blocks for easier implementation. The multiplier will be changed to N=8.

Step5.- In the FPGA implementation we will have the following:

- Use the following ***switches*** for input ***A: SW [7] to SW [0]***.
- Use the following ***switches*** for input ***B: SW [15] to SW [8]***
- Use the following ***7-segment*** display for hexadecimal input ***A: HEX [7] to HEX [6]***.
- Use the following ***7-segment*** display for hexadecimal input ***B: HEX [5] to HEX [6]***.
- Use the following ***7-segment*** display for hexadecimal output product P: ***A: HEX[3] to HEX [0]***.

Development

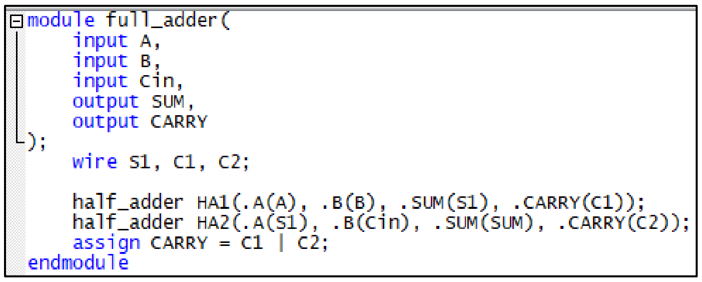
To create this project, I first had to define the implementation of a ***half adder and a full adder*** in order to ensure that the ***4-bit trial multiplier*** would function correctly during testing and implementation.



```
module half_adder(
        input A,
        input B,
        output SUM,
        output CARRY
);
        assign SUM = A ^ B;
        assign CARRY = A & B;
endmodule
```

Figure: 4 Half_Adder.v

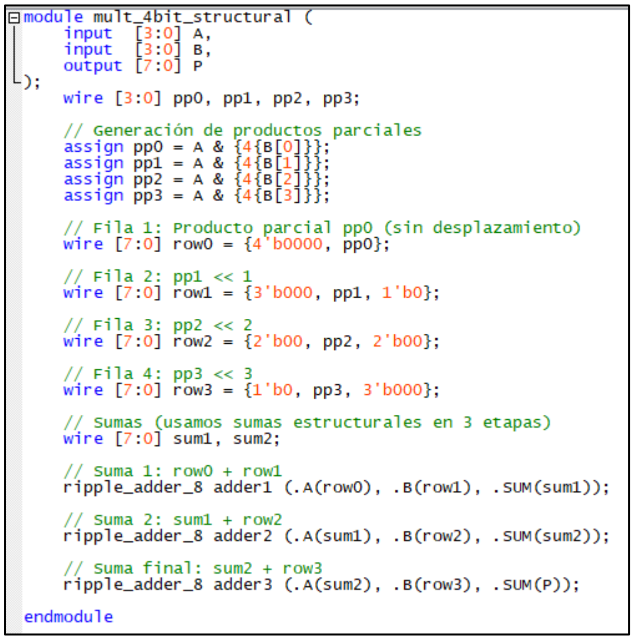
The Implementation for the ***full adder*** was:



```
module full_adder(
        input A,
        input B,
        input Cin,
        output SUM,
        output CARRY
);
        wire S1, C1, C2;

        half_adder HA1(.A(A), .B(B), .SUM(S1), .CARRY(C1));
        half_adder HA2(.A(S1), .B(Cin), .SUM(SUM), .CARRY(C2));
        assign CARRY = C1 | C2;
endmodule
```

Figure: 5 Full_Adder.v

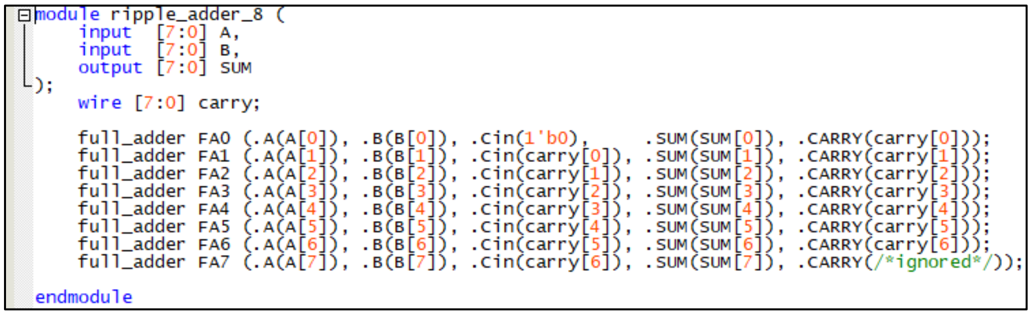
The logic for test 4-bit multiplier was ***"mut_4bit_structural"***



```
module mult_4bit_structural (
        input   [3:0] A,
        input   [3:0] B,
        output  [7:0] P
);
        wire [3:0] pp0, pp1, pp2, pp3;

        // Generación de productos parciales
        assign pp0 = A & {4{B[0]}};
        assign pp1 = A & {4{B[1]}};
        assign pp2 = A & {4{B[2]}};
        assign pp3 = A & {4{B[3]}};

        // Fila 1: Producto parcial pp0 (sin desplazamiento)
        wire [7:0] row0 = {4'b0000, pp0};

        // Fila 2: pp1 << 1
        wire [7:0] row1 = {3'b000, pp1, 1'b0};

        // Fila 3: pp2 << 2
        wire [7:0] row2 = {2'b00, pp2, 2'b00};

        // Fila 4: pp3 << 3
        wire [7:0] row3 = {1'b0, pp3, 3'b000};

        // Sumas (usamos sumas estructurales en 3 etapas)
        wire [7:0] sum1, sum2;

        // Suma 1: row0 + row1
        ripple_adder_8 adder1 (.A(row0), .B(row1), .SUM(sum1));

        // Suma 2: sum1 + row2
        ripple_adder_8 adder2 (.A(sum1), .B(row2), .SUM(sum2));

        // Suma final: sum2 + row3
        ripple_adder_8 adder3 (.A(sum2), .B(row3), .SUM(P));

endmodule
```

Figure: 6 mult_4bit_structural. v

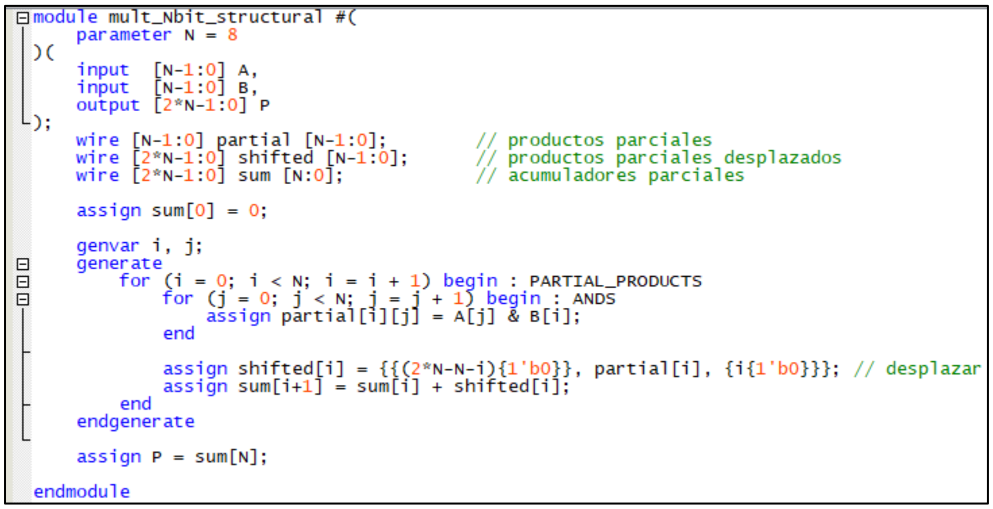
For the correct operation of the 4-bit multiplier, it was necessary to add a control mechanism for the partial products. To achieve this, an ***"rippler_adder_8"*** was used, which follows the following logic:



```
module ripple_adder_8 (
    input  [7:0] A,
    input  [7:0] B,
    output [7:0] SUM
);
    wire [7:0] carry;

    full_adder FA0 (.A(A[0]), .B(B[0]), .Cin(1'b0),     .SUM(SUM[0]), .CARRY(carry[0]));
    full_adder FA1 (.A(A[1]), .B(B[1]), .Cin(carry[0]), .SUM(SUM[1]), .CARRY(carry[1]));
    full_adder FA2 (.A(A[2]), .B(B[2]), .Cin(carry[1]), .SUM(SUM[2]), .CARRY(carry[2]));
    full_adder FA3 (.A(A[3]), .B(B[3]), .Cin(carry[2]), .SUM(SUM[3]), .CARRY(carry[3]));
    full_adder FA4 (.A(A[4]), .B(B[4]), .Cin(carry[3]), .SUM(SUM[4]), .CARRY(carry[4]));
    full_adder FA5 (.A(A[5]), .B(B[5]), .Cin(carry[4]), .SUM(SUM[5]), .CARRY(carry[5]));
    full_adder FA6 (.A(A[6]), .B(B[6]), .Cin(carry[5]), .SUM(SUM[6]), .CARRY(carry[6]));
    full_adder FA7 (.A(A[7]), .B(B[7]), .Cin(carry[6]), .SUM(SUM[7]), .CARRY(/*ignored*/));

endmodule
```

Figure: 7 rippler_adder_8. v

The logic for parameterized ***multiplier for N bits*** was the following:

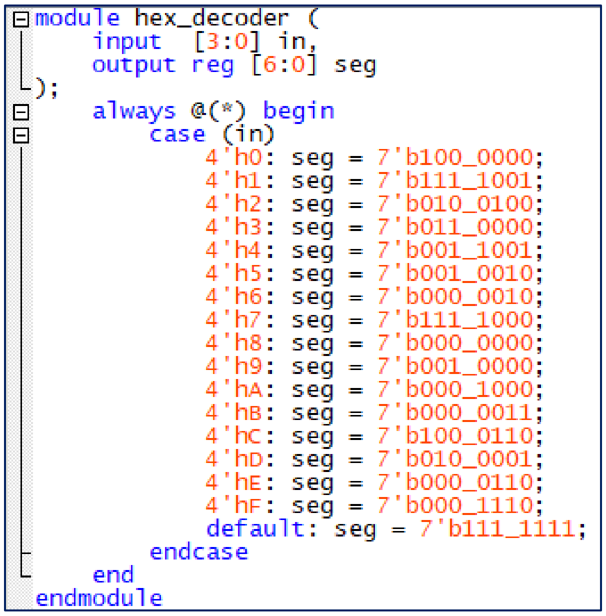


```
module mult_Nbit_structural #(
    parameter N = 8
)(
    input  [N-1:0] A,
    input  [N-1:0] B,
    output [2*N-1:0] P
);
    wire [N-1:0] partial [N-1:0];       // productos parciales
    wire [2*N-1:0] shifted [N-1:0];     // productos parciales desplazados
    wire [2*N-1:0] sum [N:0];           // acumuladores parciales

    assign sum[0] = 0;

    genvar i, j;
    generate
        for (i = 0; i < N; i = i + 1) begin : PARTIAL_PRODUCTS
            for (j = 0; j < N; j = j + 1) begin : ANDS
                assign partial[i][j] = A[j] & B[i];
            end

            assign shifted[i] = {{(2*N-N-i){1'b0}}, partial[i], {i{1'b0}}}; // desplazar
            assign sum[i+1] = sum[i] + shifted[i];
        end
    endgenerate

    assign P = sum[N];

endmodule
```

Figure: 8 mult_Nbit_structural.v. v

In this case before to do the Nbit parameterized multiplier was necessary test the 4bit multiplier and the logic system for improve and generate the scalabity for the Nbit multiplier. Once that part of the logic was obtained, it became possible to generate the logic for a parameterizable multiplier i.e., one capable of handling 8, 32, 64, or any other value assigned to the parameter. This was achieved by designing scalable logic that performed the necessary shifts and operations correctly.

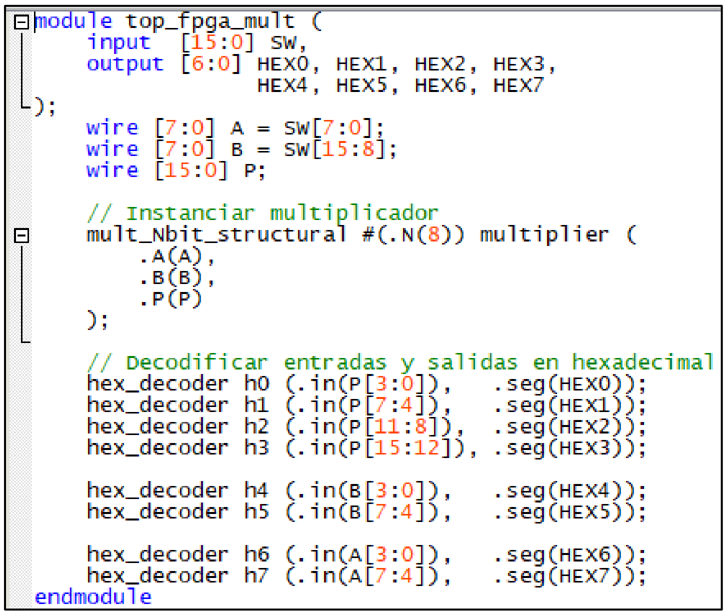
The consideration for ***bcd-7-segment***:



```
module hex_decoder (
     input   [3:0] in,
     output reg [6:0] seg
);
     always @(*) begin
         case (in)
             4'h0: seg = 7'b100_0000;
             4'h1: seg = 7'b111_1001;
             4'h2: seg = 7'b010_0100;
             4'h3: seg = 7'b011_0000;
             4'h4: seg = 7'b001_1001;
             4'h5: seg = 7'b001_0010;
             4'h6: seg = 7'b000_0010;
             4'h7: seg = 7'b111_1000;
             4'h8: seg = 7'b000_0000;
             4'h9: seg = 7'b001_0000;
             4'hA: seg = 7'b000_1000;
             4'hB: seg = 7'b000_0011;
             4'hC: seg = 7'b100_0110;
             4'hD: seg = 7'b010_0001;
             4'hE: seg = 7'b000_0110;
             4'hF: seg = 7'b000_1110;
             default: seg = 7'b111_1111;
         endcase
     end
endmodule
```

Figure: 9 hex_decoder.v.

To test the logic on the FPGA board, the following was considered: instantiating values of the parameterizable multiplier and making use of the hexadecimal decoder elements. This allowed the main logic to function correctly, while also optimizing certain elements by maintaining function calls or instances from other files.

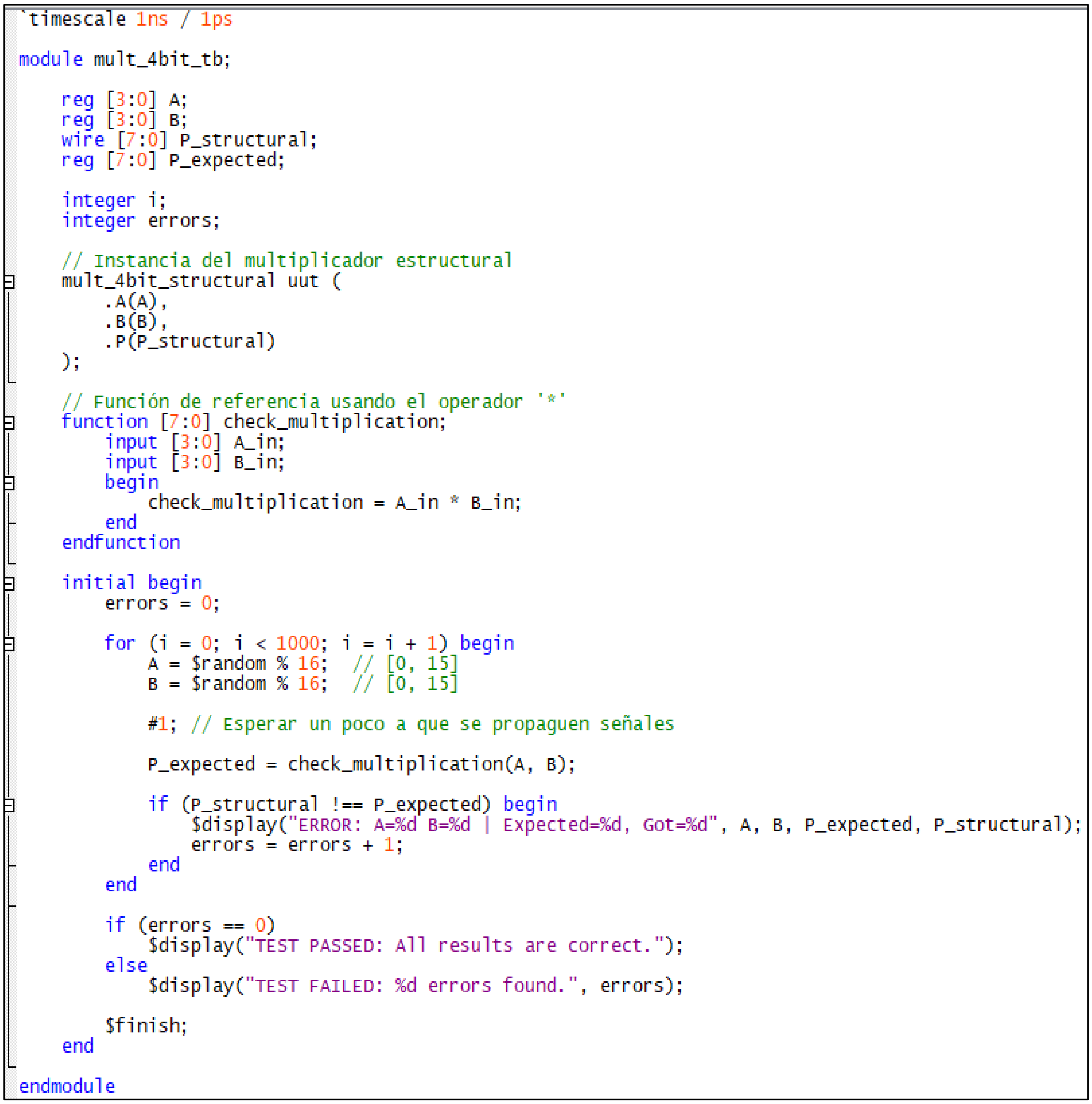


```
module top_fpga_mult (
    input  [15:0] SW,
    output [6:0] HEX0, HEX1, HEX2, HEX3,
                 HEX4, HEX5, HEX6, HEX7
);
    wire [7:0] A = SW[7:0];
    wire [7:0] B = SW[15:8];
    wire [15:0] P;

    // Instanciar multiplicador
    mult_Nbit_structural #(.N(8)) multiplier (
        .A(A),
        .B(B),
        .P(P)
    );

    // Decodificar entradas y salidas en hexadecimal
    hex_decoder h0 (.in(P[3:0]),   .seg(HEX0));
    hex_decoder h1 (.in(P[7:4]),   .seg(HEX1));
    hex_decoder h2 (.in(P[11:8]),  .seg(HEX2));
    hex_decoder h3 (.in(P[15:12]), .seg(HEX3));

    hex_decoder h4 (.in(B[3:0]),   .seg(HEX4));
    hex_decoder h5 (.in(B[7:4]),   .seg(HEX5));

    hex_decoder h6 (.in(A[3:0]),   .seg(HEX6));
    hex_decoder h7 (.in(A[7:4]),   .seg(HEX7));
endmodule
```

Figure: 10 top_fpga_mult.v.

Testbench

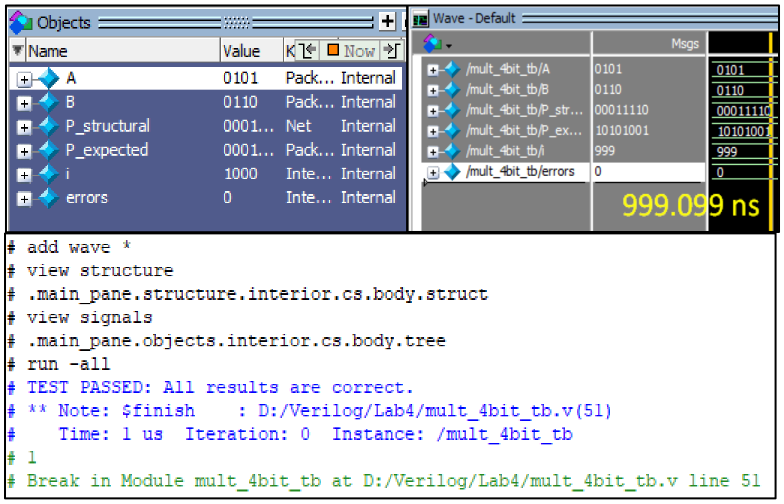
The correct assignment of the testbenches was carried out in the "Assignments" section, and the 4-bit multiplier was tested first—specifically, the structural file along with the logic of the ***"4-bit multiplier testbench":***



```verilog
`timescale 1ns / 1ps

module mult_4bit_tb;

    reg [3:0] A;
    reg [3:0] B;
    wire [7:0] P_structural;
    reg [7:0] P_expected;

    integer i;
    integer errors;

    // Instancia del multiplicador estructural
    mult_4bit_structural uut (
        .A(A),
        .B(B),
        .P(P_structural)
    );

    // Función de referencia usando el operador '*'
    function [7:0] check_multiplication;
        input [3:0] A_in;
        input [3:0] B_in;
        begin
            check_multiplication = A_in * B_in;
        end
    endfunction

    initial begin
        errors = 0;

        for (i = 0; i < 1000; i = i + 1) begin
            A = $random % 16;  // [0, 15]
            B = $random % 16;  // [0, 15]

            #1; // Esperar un poco a que se propaguen señales

            P_expected = check_multiplication(A, B);

            if (P_structural !== P_expected) begin
                $display("ERROR: A=%d B=%d | Expected=%d, Got=%d", A, B, P_expected, P_structural);
                errors = errors + 1;
            end
        end

        if (errors == 0)
            $display("TEST PASSED: All results are correct.");
        else
            $display("TEST FAILED: %d errors found.", errors);

        $finish;
    end

endmodule
```

Figure: 11 mult_4bit_tb.v

Debug in Model Sim:



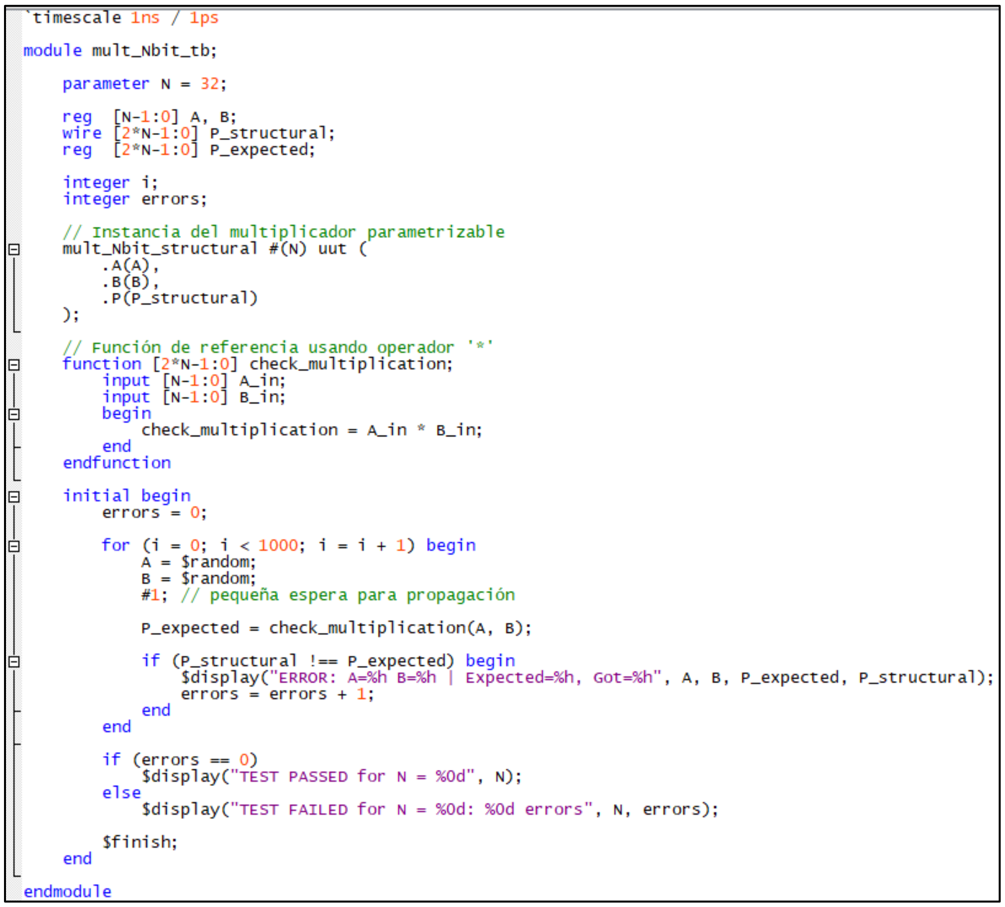
```
Objects
Name           Value    Kind     Mode
A              0101     Pack...  Internal
B              0110     Pack...  Internal
P_structural   0001...  Net      Internal
P_expected     0001...  Pack...  Internal
i              1000     Inte...  Internal
errors         0        Inte...  Internal
```

```
Wave - Default
                          Msgs
/mult_4bit_tb/A          0101        0101
/mult_4bit_tb/B          0110        0110
/mult_4bit_tb/P_str...   00011110    00011110
/mult_4bit_tb/P_ex...    10101001    10101001
/mult_4bit_tb/i          999         999
/mult_4bit_tb/errors     0           0
                         999.099 ns
```

```
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# TEST PASSED: All results are correct.
# ** Note: $finish    : D:/Verilog/Lab4/mult_4bit_tb.v(51)
#    Time: 1 us  Iteration: 0  Instance: /mult_4bit_tb
# 1
# Break in Module mult_4bit_tb at D:/Verilog/Lab4/mult_4bit_tb.v line 51
```

Figure: 12 mult_4bit_tb ModelSim

The logic for the parameterizable multiplier and its corresponding testbench were created to test it properly. For do adjustment in the parameterized ***Parameter N = Value***

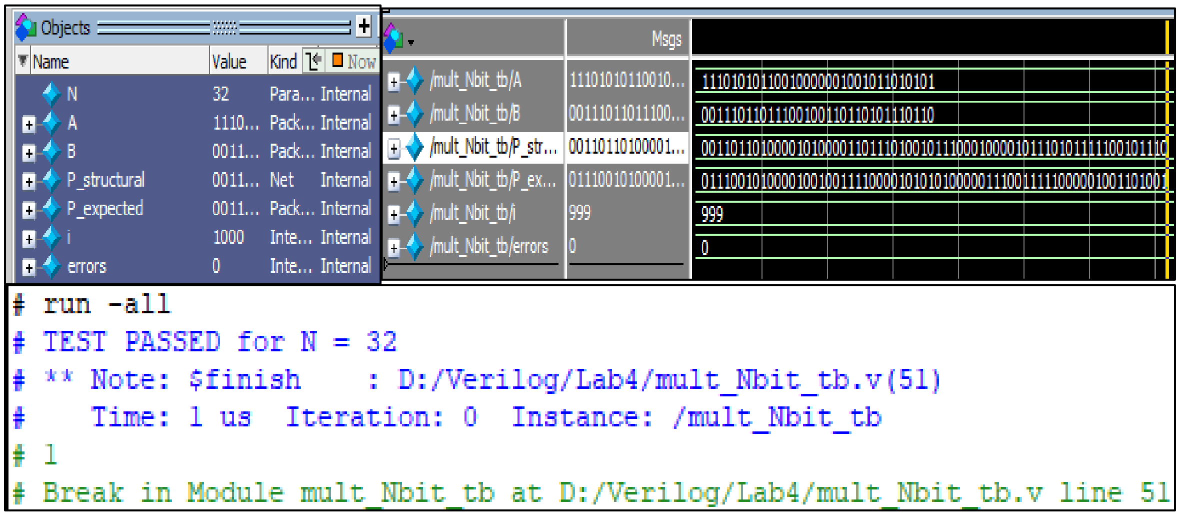


```
`timescale 1ns / 1ps

module mult_Nbit_tb;

    parameter N = 32;

    reg  [N-1:0] A, B;
    wire [2*N-1:0] P_structural;
    reg  [2*N-1:0] P_expected;

    integer i;
    integer errors;

    // Instancia del multiplicador parametrizable
    mult_Nbit_structural #(N) uut (
        .A(A),
        .B(B),
        .P(P_structural)
    );

    // Función de referencia usando operador '*'
    function [2*N-1:0] check_multiplication;
        input [N-1:0] A_in;
        input [N-1:0] B_in;
        begin
            check_multiplication = A_in * B_in;
        end
    endfunction

    initial begin
        errors = 0;

        for (i = 0; i < 1000; i = i + 1) begin
            A = $random;
            B = $random;
            #1; // pequeña espera para propagación

            P_expected = check_multiplication(A, B);

            if (P_structural !== P_expected) begin
                $display("ERROR: A=%h B=%h | Expected=%h, Got=%h", A, B, P_expected, P_structural);
                errors = errors + 1;
            end
        end

        if (errors == 0)
            $display("TEST PASSED for N = %0d", N);
        else
            $display("TEST FAILED for N = %0d: %0d errors", N, errors);

        $finish;
    end

endmodule
```

Figure: 13 mult_Nbit_tb.v

Debug in Model Sim:

In this process 999.409 nano seconds.



Name	Value	Kind	
N	32	Para...	Internal
A	1110...	Pack...	Internal
B	0011...	Pack...	Internal
P_structural	0011...	Net	Internal
P_expected	0011...	Pack...	Internal
i	1000	Inte...	Internal
errors	0	Inte...	Internal

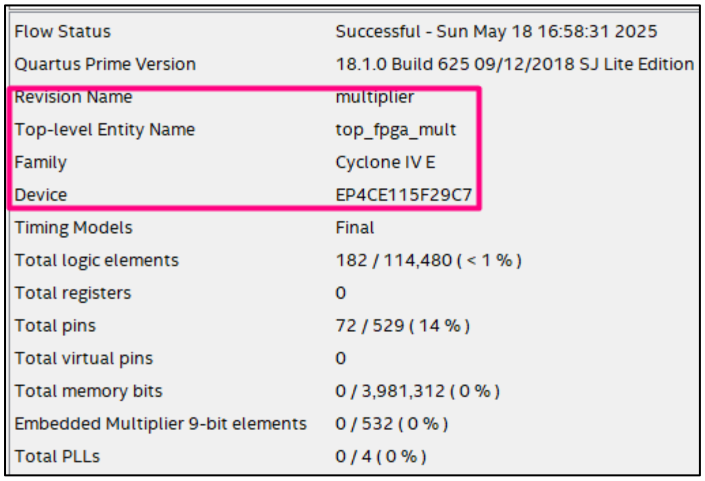
Signal	Msgs	
/mult_Nbit_tb/A	1110101011001O...	11101010110010000001001011010101
/mult_Nbit_tb/B	0011101101110O...	00111011011100100110110101110110
/mult_Nbit_tb/P_str...	0011011010O001...	0011011010000101000011011101001011100010000101110101111100101110
/mult_Nbit_tb/P_ex...	0111001010O001...	0111001010000100100111100001010101000001110011111000001001101001
/mult_Nbit_tb/i	999	999
/mult_Nbit_tb/errors	0	0

```
# run -all
# TEST PASSED for N = 32
# ** Note: $finish    : D:/Verilog/Lab4/mult_Nbit_tb.v(51)
#    Time: 1 us  Iteration: 0  Instance: /mult_Nbit_tb
# 1
# Break in Module mult_Nbit_tb at D:/Verilog/Lab4/mult_Nbit_tb.v line 51
```

Figure: 14 mult_Nbit_tb.v ModelSim

FPGA Implementation

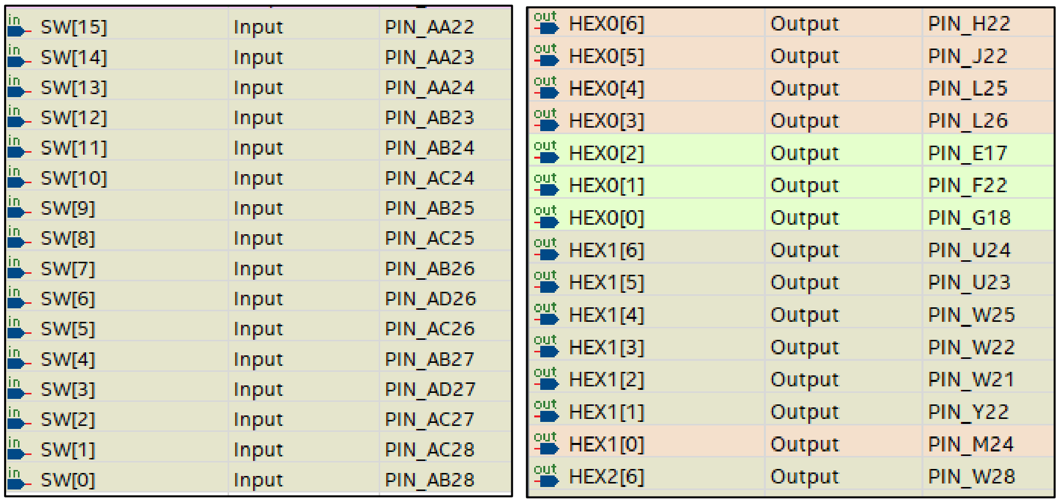
Once the program was completed, the compilation process was carried out in order to generate the pin assignment plan and correctly map the pins according to the program's requirements.



Flow Status	Successful - Sun May 18 16:58:31 2025
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	multiplier
Top-level Entity Name	top_fpga_mult
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	182 / 114,480 (< 1 %)
Total registers	0
Total pins	72 / 529 (14 %)
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: 15 Compilation Brief

The pins were assigned in this case we used all the bcd-7-segments in the images can see only a example, but is necessary set all bcd from the data sheet.



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

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

Figure: 16 PinPlanner

The RTL view can be observed as follows.

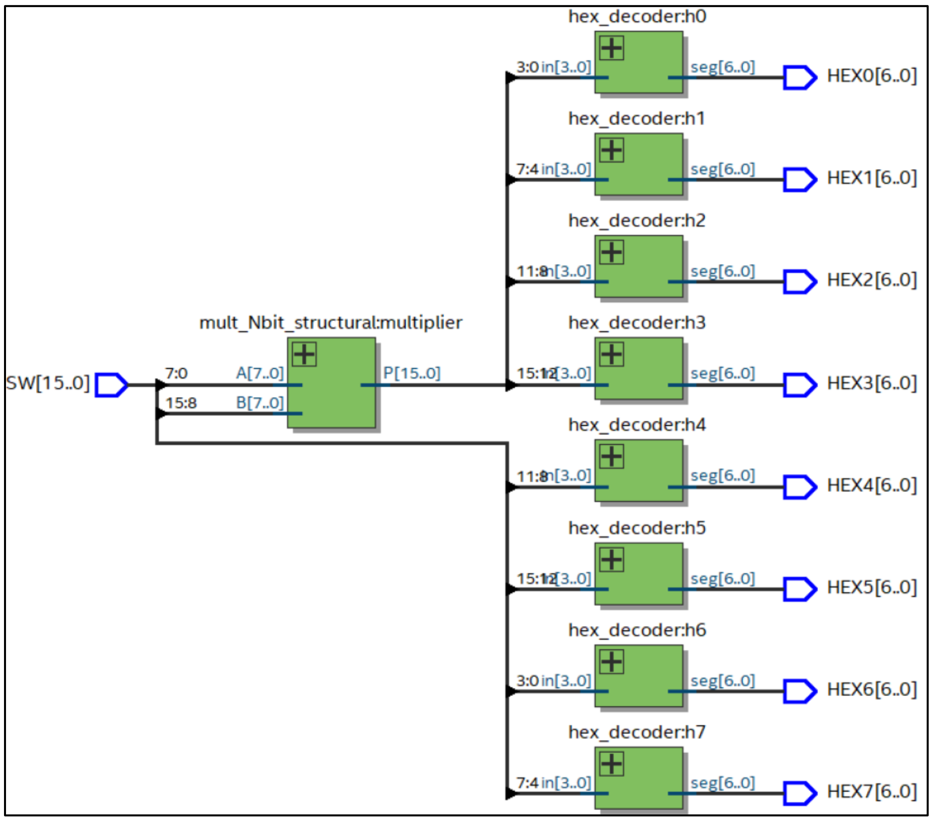


Figure: 17 RTL Viewer

FPGA Results

Once the testbenches were successfully tested, we proceeded to verify the correct operation of the parameterizable multiplier directly on the FPGA. In this case, ***8 bits were input for "A" and 8 bits for "B", with the result displayed in BCD format.***

1.- Test; A=11111111(FF); B=11111111(FF); R=FE01

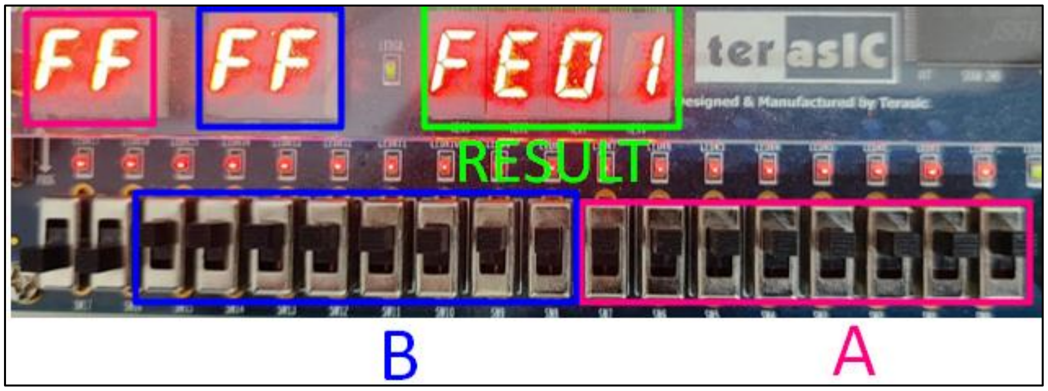


Figure: 18 Test1

2.- Test; A=11111111(FF); B=00000011(03); R=02FD

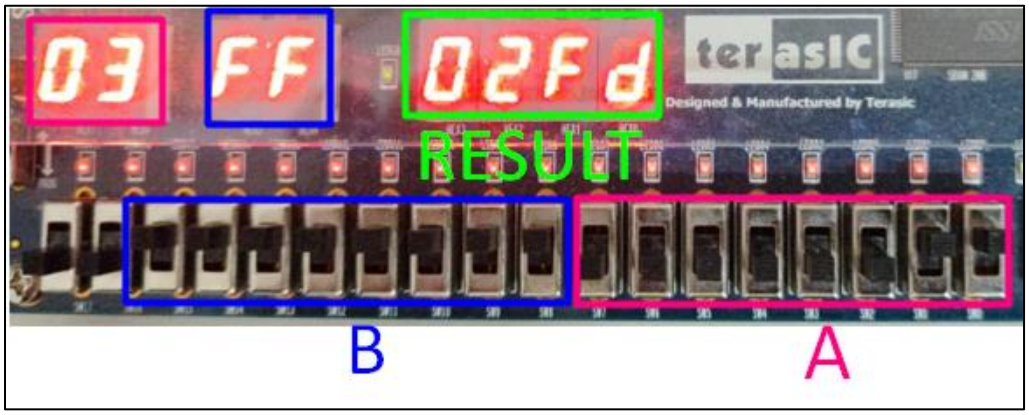


Figure: 19 Test2

GitHub Repository

https://github.com/CeicGitHub/Fundamentals_Of_Digital_Design_FPGA_2/tree/Lab4_Multiplier

Conclusion / Issues

Cesar Eduado Inda Ceniceros

One of the biggest challenges I faced while completing this lab was designing a multiplier that could scale to different values. For the 4-bit version, we used the half adder and full adder files, but for this implementation, we had to rely on shifting and logic without using those files. This approach allowed for a more optimized and readable implementation in terms of code. This practice helped me gain a deeper understanding of how a multiplier works, and I found it interesting to see how it can be made parameterizable simply by modifying a single value in the code. The consideration of generating callbacks to certain instances or what I commonly refer to as function calls in other programming contexts helped me better understand this part of Verilog and its combination with hardware logic.

Alonso Emmanuel Lopez Macias

Personally, we faced some difficulties understanding that we needed to add the ripple adder component to properly handle controlled additions. Initially, we were misinterpreting some values, which caused errors in the 4-bit multiplier. Later on, we had to start implementing improvements, and this made the implementation clearer for me. Working step by step through the generation of project files and seeing how these instances were called from other modules helped me realize that it's not necessary to load all the logic into a single Verilog file. Keeping the code modular is not only a good practice but also improves understanding and readability. Separating the testbenches for specific modules and the programming considerations for physical FPGA testing helped me understand the workflow more deeply—this was one of the key lessons I took from this lab.