Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica
IE0424 – Laboratorio Digitales I
II ciclo 2016

.

Reporte #2

Laboratorio 2
Desarrollo de circuitos combinatorios

Daniel Díaz, B22245
José J Rodríguez, B25706
Carla Vega, B06763
Profesor: Diego Valverde G.

18 de Setiembre del 2016

Índice

Índice de figuras

Objetivos

Objetivo General

Utilizar el FPGA para el desarrollo de circuitos combinatorios.

Objetivo Específico

- Investigar el funcionamiento de la tarjeta de desarrollo FPGA Spartan 3E.
- Utilizar las herramientas del Xilinx ISE.
- Conocer y aplicar el flujo del diseño para sistemas basados en FPGA.

1. Ejercicio 1

1.1. Función desarrollada

Se implementaron modificaciones al código en el archivo *MiniAlu.v*, *Definitions.v* y en *Module_ROM.v* que se detallan a continuación.

En el siguiente código se implementa una operación similar a la de la suma ya incluida en el código original, con la diferencia que se utiliza el operador [*] para realizar una multiplicación de las entradas en lugar de una suma. Una nota importante de esta función es que las variables utilizadas (wData1 y wData0) se declaran especificando si se utilizan con signo o no.

MiniAlu.v

```
//--------------------------------------
    `SMUL:
    begin
    rResultMult <= wData1*wData0;
    end
```

Definitions.v

```
`timescale 1ns / 1ps
`ifndef DEFINTIONS_V
`define DEFINTIONS_V

`default_nettype none
`define NOP   4'd0
`define LED   4'd2
`define BLE   4'd3
`define STO   4'd4
`define ADD   4'd5
`define JMP   4'd6
`define SMUL  4'd7

`define R0 8'd0
`define R1 8'd1
`define R2 8'd2
`define R3 8'd3
`define R4 8'd4
`define R5 8'd5
`define R6 8'd6
`define R7 8'd7

`endif
```

1.2. Correcciones realizadas al anteproyecto

En el anteproyecto no se especificaron los cambios que se iban a realizar en el programa de la ROM. En la subsección siguiente se presentan estos cambios realizados. El código de la MiniAlu es muy similar al planteado en el anteproyecto.

1.3. Correcciones realizadas al código

Se realizó una modificación al programa de la ROM, para comprobar que este efectuara una multiplicación utilizando la nueva operación. El código se puede encontrar a continuación. En este se multiplica el valor de R7 con el valor de R3 (2*3 en el primer ciclo).

ModuleROM.v

```
`timescale 1ns / 1ps
`include "Defintions.v"

```

```
`define LOOP1 8'd8
`define LOOP2 8'd5
module ROM
(
    input  wire[15:0]       iAddress,
    output reg [27:0]       oInstruction
);
always @ ( iAddress )
begin
    case (iAddress)

    0: oInstruction = { `NOP ,24'd4000     };
    1: oInstruction = { `STO, `R7, 16'd2  };
    2: oInstruction = { `STO, `R3, 16'd3  };
    3: oInstruction = { `STO, `R4,16'd1000 };
    4: oInstruction = { `STO, `R5,16'd0    };  //j
//LOOP2:
    5: oInstruction = { `LED ,8'b0,`R7,8'b0 };
    6: oInstruction = { `STO ,`R1,16'h0     };
    7: oInstruction = { `STO ,`R2,16'd5000 };
//LOOP1:
    8: oInstruction = { `ADD ,`R1,`R1,`R3    };
    9: oInstruction = { `BLE ,`LOOP1,`R1,`R2 };

    10: oInstruction = { `ADD ,`R5,`R5,`R3    };
    11: oInstruction = { `BLE ,`LOOP2,`R5,`R4 };
    12: oInstruction = { `NOP ,24'd4000       };
    13: oInstruction = { `SMUL, `R7, `R7, `R3};
    14: oInstruction = { `JMP ,  8'd2,16'b0   };
    default:
        oInstruction = { `LED ,  24'b10101010 };        //NOP
    endcase
end

endmodule
```

En la figura 2 se puede observar la señal que contiene el resultado de la multiplicación.

1.4. Justificación de los resultados obtenidos

En el siguiente código se muestra la frecuencia máxima obtenida mediante el reporte de síntesis.

```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 7.345ns (Maximum Frequency: 136.147MHz)
   Minimum input arrival time before clock: 8.859ns
   Maximum output required time after clock: 4.283ns
   Maximum combinational path delay: No path found

=========================================================================

No asynchronous contProcess "Synthesize␣-␣XST" completed successfully
```

En la figura 1 se muestra el reporte de LTUs, slices, FF y latches obtenidos al sintetizar la nueva operación de multiplicación.

En la figura 2 se puede observar la señal de rResult la cual contiene el resultado de la multiplicación descrita en la subsección anterior.

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	98	9,312	1%	
Number used as Flip Flops	79			
Number used as Latches	19			
Number of 4 input LUTs	188	9,312	2%	
Number of occupied Slices	103	4,656	2%	
Number of Slices containing only related logic	103	103	100%	
Number of Slices containing unrelated logic	0	103	0%	
Total Number of 4 input LUTs	188	9,312	2%	
Number used as logic	124			
Number used for Dual Port RAMs	64			
Number of bonded IOBs	10	232	4%	
Number of BUFGMUXs	1	24	4%	
Average Fanout of Non-Clock Nets	4.25			

Figura 1: Cuadro resumen LTUs, slices, FF y latches del ejercicio 1

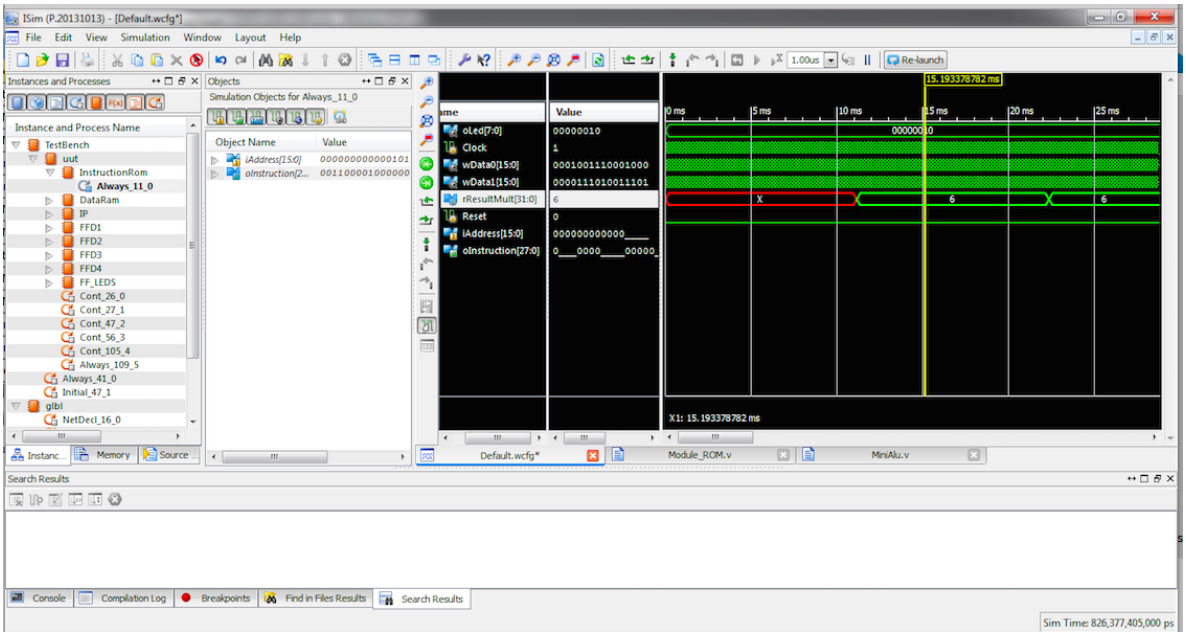

Figura 2: Resultado de la multiplicación

2. Ejercicio 2

2.1. Función desarrollada

La función desarrollada en esta sección es un multiplicador basado en sumadores esto se colocaron para crear un multiplicador de que recibe dos entradas de 4 bits y retorna como resultado su salida de 8 bits como se muestra en la figura 3

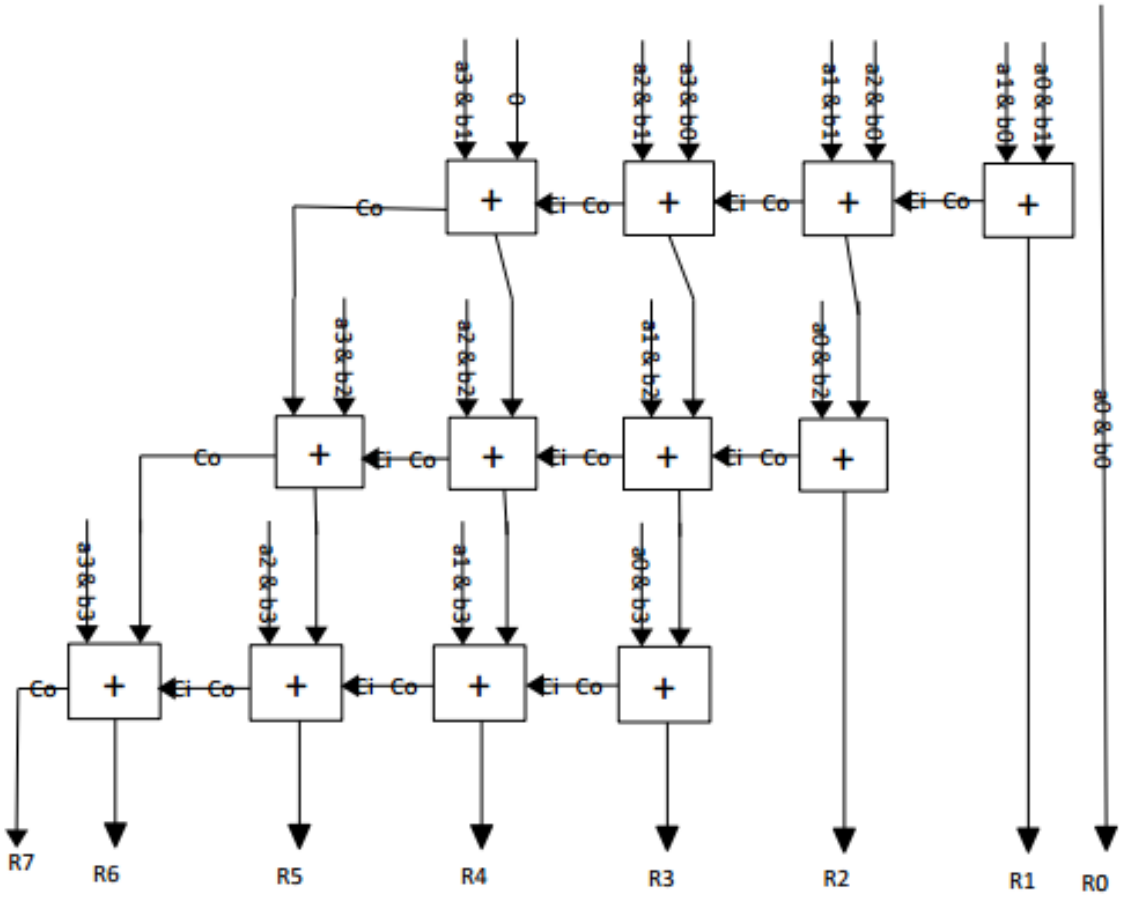

Figura 3: Sumadores colocados para formar un multiplicador

2.2. Correcciones realizadas al código

Se crearon tres archivos para mantener el codigo de manera más ordenada y con mayores posibilidades de ser reutilizable estos achivos fueron un sumador el cual esta compuesto de tres entadas que son los bits que se desean sumar y dos salidas donde una es el carry y la otra la respuesta de la suma

```
module Sumador(
    input wire wA,
    input wire wB,
     input wire wC,
    output reg rOut,
    output reg rCarry
    );

     always @(*)
            begin {rOut,rCarry} = wA+wB+wC ;
     end
endmodule
```

Luego se unieron cuatro sumadores para crear una etapa del multiplicador con esto disminuimos el código que se necesita para crear el multiplicador y ademas tenemos una forma más fácil para debugear en caso de tener problemas

```
module sumador4(
    input wInA0,
```

```
    input wInA1,
    input wInA2,
    input wInA3,
    input wInB0,
    input wInB1,
    input wInB2,
    input wInB3,
    output rCarry,
    output rout0,
     output rout1,
    output rout2,
    output rout3
     );
      wire carry1,carry2,carry3,carry0;
Sumador sum0 (
        .wA(wInA0),
        .wB(wInB0),
        .wC(1'b0),
        .rOut(rout0),
        .rCarry(carry0)
        );
Sumador sum2 (
        .wA(wInA2),
        .wB(wInB2),
        .wC(carry1),
        .rOut(rout2),
        .rCarry(carry2)
        );

Sumador sum1 (
        .wA(wInA1),
        .wB(wInB1),
        .wC(carry0),
        .rOut(rout1),
        .rCarry(carry1)
        );
Sumador sum3 (
        .wA(wInA3),
        .wB(wInB3),
        .wC(carry2),
        .rOut(rout3),
        .rCarry(rCarry)
        );


endmodule
```

Como ya se poseen cada una de las etapas del multiplicador en este solo queda unir estas para poder formar un multiplicador de el cual tiene dos entradas de 4 bit y una salida de 8 bits.

```
module Mult(
    input wire[3:0] wMulA,
    input wire[3:0] wMulB,
    output reg [7:0] rOut
     );
     wire wC0;
     wire wz0;
     wire wz1;
     wire wz2;
     wire wz3;

sumador4 sum0(
     .wInA0(wMulA[0]&wMulB[1]),
     .wInA1(wMulA[2]&wMulB[0]),
     .wInA2(wMulA[3]&wMulB[0]),
     .wInA3(1'b0),
     .wInB0(wMulA[1]&wMulB[0]),
     .wInB1(wMulA[1]&wMulB[1]),
```

```
      .wInB2(wMulA[2]&wMulB[1]),
      .wInB3(wMulA[3]&wMulB[1]),
      .rCarry(wC0),
      .rout0(wz0),
       .rout1(wz1),
      .rout2(wz2),
      .rout3(wz3)
      );
sumador4 sum1(
      .wInA0(wz1),
      .wInA1(wz2),
      .wInA2(wz3),
      .wInA3(wMulA[3]&wMulB[2]),
      .wInB0(wMulA[0]&wMulB[2]),
      .wInB1(wMulA[1]&wMulB[2]),
      .wInB2(wMulA[2]&wMulB[2]),
      .wInB3(wC0),
      .rCarry(wC1),
      .rout0(wy0),
       .rout1(wy1),
      .rout2(wy2),
      .rout3(wy3)
      );

sumador4 sum2(
      .wInA0(wy1),
      .wInA1(wy2),
      .wInA2(wy3),
      .wInA3(wC1),
      .wInB0(wMulA[0]&wMulB[3]),
      .wInB1(wMulA[1]&wMulB[3]),
      .wInB2(wMulA[2]&wMulB[3]),
      .wInB3(wMulA[3]&wMulB[3]),
      .rCarry(wC2),
      .rout0(ww0),
       .rout1(ww1),
      .rout2(ww2),
      .rout3(ww3)
      );

assign rOur = {wC2,ww3,ww2,ww1,ww0,wy0,wz0,wMulA[0]&wMulB[0]};

endmodule
```

Luego se creo una instancia en miniALU para que se pudiera llamar a esta

```
IMUL mult1
(
    .iMulA(wSourceData0[3:0]),
     .iMulB(wSourceData1[3:0]),
     .oMult(wMultResult)
) ;
```

Y se le asigna un definición la cual es IMUL para poder llamar esta con mas facilidad desde la rom como una función.

```

    `IMUL:
    begin
    rResult <= {8'b0,wMultResult};
    end
```

2.3. Correcciones al anteproyecto

En este no se planeo crear un bloque con cuatro sumadores el cual facilita mucho la implementación del código ya que cada etapa no se tiene que volver a colocar cada uno de lso sumadores sino que

solamente se tienen tres etapas de con 4 sumadores cada uno pero esto es un bloque el cual nos es más fácil manipular y ayuda a reducir las lineas de codigo, pero con respecto al resto del pseudo código se implemento en verilog para poder ver los resultados,

2.4. Justificación de los resultados obtenidos

Este modelo fue realizado a desde compuertas muy básicas en comparación con el ejercicio 1 que se creo el sumador de forma automática por la herramienta de síntesis por eso se tiene un mayor control de como se realizan . Después de tener la función multiplicar para que ponga el numero en los led es bastante sencillo solo hay que colocar en el registro R7 el resultado. Esta implementación ocupa menos registros tanto en flip-flops con en latches que la anterior pero conlleva mucho más trabajo ya que el otro es generado automáticamente por la herramienta y como se tiene casi todo construido con lógica combinacional los LUTs pertenecen solo a la parte de la ROM y al resto del circuito pero no al multiplicador.

```
1
2
3  Advanced HDL Synthesis Report
4
5  Macro Statistics
6  # RAMs                                              : 2
7   9x16-bit dual-port distributed RAM                : 2
8  # ROMs                                              : 1
9   16x28-bit ROM                                     : 1
10 # Adders/Subtractors                                : 26
11  1-bit adder carry out                             : 12
12  16-bit adder                                      : 2
13  2-bit adder                                       : 12
14 # Counters                                          : 1
15  16-bit up counter                                 : 1
16 # Registers                                         : 72
17  Flip-Flops                                        : 72
18 # Latches                                           : 3
19  1-bit latch                                       : 3
20 # Comparators                                       : 1
21  16-bit comparator lessequal                       : 1
```

3. Ejercicio 3

Se hizo una extensión del ejercicio anterior para multiplicar dos números de 16 bits y para ello se implementó la función *generate*. Para el generate se utilizaron dos lazos *for* dentro de los cuales se asignan valores a cada casilla de las matrices de cables que conectan los bloques sumadores.

3.1. Función desarrollada

La función *generate* de Verilog es muy útil cuando se tiene grandes cantidades de bits y los bloques que se desean implementar tienen una secuencia,y usando este método podemos implementar el multiplicador de manera muy similar al realizado en la parte anterior pero con la ventaja de que agregar o quitar etapas es mucho más fácil y rápido.

La función se implementó en el código del archivo *Minialu.v*. Se utilizó una matriz de cables para las señales de carry, una para los resultados de cada bloque sumador y un arreglo para el resultado final de la multiplicación. Se siguió el mismo diagrama de bloques sumadores utilizados en el ejercicio 2. El código implementado se describe a continuación.

```
1 wire[4:0] wCarryIn[2:0]
2 wire[4:0] wRes[3:0]
3
4
```

```
assign wCarryIn[0][0]=1'b0;
assign wCarryIn[1][0]=1'b0;
assign wCarryIn[2][0]=1'b0;
assign wRes[0][1]= wSourceData0[1]&wSourceData1[0];
assign wRes[0][2]= wSourceData0[2]&wSourceData1[0];
assign wRes[0][3]= wSourceData0[3]&wSourceData1[0];
assign wResult[0][4]=1'b0;
assign wRes[1][4]=wCarryIn[0][4];
assign wRes[2][4]=wCarryIn[1][4];


genvar row,col;
generate
    for (row = 0, row < 3 , row=row+1)
        begin:ROW
            for (col = 0 , col < 4 , col+1)
                begin:COL
    Sumador genSumador(
    .iA(wSourceData0[col]&wSourceData1[row+1]),
    .iB(wResult[row][col+1]),
    .iC(wCarryIn[row][col]),
    .oOut(wRes[row+1][col]),
    .oCarry(wCarryIn[row][col])
    );
        end
                end
end generate

assign wR = {wCarryIn[2][4], wRes[3][3], wRes[3][2], wRes[3][1], wRes[3][0], wRes
    [2][0], wRes[1][0], wSourceData0[0]&wSourceData1[0]};
```

Los primeros 3 *assign*'s del código fuera del bloque generate corresponden a los carry de entrada de los primeros bloques de cada fila, citando el enunciado del laboratorio *"Además recuerde que Ci de los sumadores de las columnas de la izquierda son cero"*. Los siguientes 3 assign corresponden a una de las dos entradas de los bloques sumadores de la primera fila. Esto debido a que los bloques de esta fila usan dicha entrada diferente al resto de bloques. Lo mismo sucede con los bloques sumadores de la última columna, estos se deben conectar como entrada de otros bloques, de manera diferente a la que se hace con el resto, por eso de codifica el assign fuera del bloque generate.

Dentro del bloque generate se asignan las matrices de cable a las entradas y salidas de un bloque sumador genérico, de manera que se cumpla con el diagrama propuesto en el enunciado y en el anteproyecto de esta bitácora. (figura 3

De manera similar de utiliza un assign fuera del generate al final del código, en este se asignan los valores a un vector de cables que funciona como el resultado total de la multiplicación.

3.2. Correcciones realizadas al anteproyecto

En el anteproyecto, se planteó un código que tenía un único **for**, sin embargo durante la sesión de laboratorio determinamos que se necesitan dos *for* para cumplir con los requerimientos de este ejercicio de recorrer filas y columnas y así crear la matriz de bloques implementada.

3.3. Correcciones realizadas al código

El código implementado finalmente y tomando en cuenta lo descrito en la subsección anterior se utilizaron dos arreglos de cables, uno para el acarreo: wCarryIn de cada bloque y uno para los resultados de estos wResult que son las entradas de los bloques de la siguiente fila. Además de esto, se agregó el código descrito en la subsección *Función desarrollada*. También se modifico el código del programa en la ROM para que utilizara la nueva operación.

3.4. Justificación de los resultados obtenidos

Se obtuvieron resultados bastantes similares a los del ejercicio anterior (Array Multiplier), sin embargo se nota que una vez entendido el funcionamiento de la directiva *generate*, es más sencillo y compacto utilizar esta sobre la descripción bloque por bloque utilizada en el ejercicio anterior. Entre las desventajas que se pueden presentar al utilizar un bloque generate es que el sintetizador puede hacer un uso poco eficiente de memoria y es más fácil para el programador cometer errores al codificar.

3.5. Cómo se compara la frecuencia y número de LUTs, slices y flip flops con el ejercicio 1?

Respecto a ejercicio 1 y 2, la máxima frecuencia es menor, sin embargo el número de slices y LUTs es mayor.

3.6. Son los bloques *generate* sintetizables?

Sí, los bloques *generate* son sintetizables. Un generate es básicamente una directiva al sintetizador, el cual se encarga de hacer una instancia diferente para cada ciclo del loop. El número de ciclos debe poder ser calculado en tiempo de ejecución.

3.7. Cuántas etapas tiene este nuevo circuito de multiplicación? Qué ocurre con el período del reloj si se añaden más y más etapas de lógica combinatoria?

3.8. Qué ocurre con la frecuencia si se añaden latches entre cada etapa de sumadores?

Al agregar latches entre cada etapa de sumadores se disminuye la frecuencia máxima a la cual puede trabajar el circuito, debido a que los latches agregan tiempo mínimo de propagación por sus retardos entre entrada y salida (tiempo de setup por ejemplo).

4. Ejercicio 4

4.1. Función desarrollada

Para este ejercicio se implementó la multiplicación con un algoritmo con Look up Table. La forma en que se hizo fue implementar una LUT que multiplica un operando de 2 bits con otro operando de longitud cualquiera. Luego, utilizando varios de éstos, se suman los resultados parciales con el corrimiento necesario para obtener la multiplicación.

4.2. Correcciones realizadas al anteproyecto

Solo fueron necesarias algunas correcciones al código que se detallan en la siguiente sección.

4.3. Correcciones realizadas al código

A continuación se muestra el código para la LUT. Fue necesario cambiar la definición de la señal wD0, para que fuera del tamaño correcto.

```
1 module LUT # (parameter SIZE=8)
2 (
3  input wire [SIZE-1:0] A,
4  input wire [1:0] B,
5  output reg [SIZE+1:0] Result
6 );
7
8 wire [SIZE+1:0] wD0, wD1, wD2, wD3;
```

```

assign wD1 = {2'b0, A};
assign wD0 = wD1 ^ wD1; // Esto siempre es cero, de la longitud adecuada.
assign wD2 = {1'b0, A, 1'b0};
assign wD3 = wD2 + wD1;

always @ (A, B)
begin
    case (B)
    //-------------------------------------
    2'b00:
    begin
        Result <= wD0;
    end
    //-------------------------------------
    2'b01:
    begin
        Result <= wD1;
    end
    //-------------------------------------
    2'b10:
    begin
        Result <= wD2;
    end
    //-------------------------------------
    2'b11:
    begin
        Result <= wD3;
    end
    //-------------------------------------
    endcase
end

endmodule
```

Luego se muestra un módulo que realiza la multiplicación de dos números de 4 bits.

```
module Mult4x4
(
 input wire [3:0] A,
 input wire [3:0] B,
 output wire [7:0] Result
);

wire [5:0] wResInt1,wResInt2;

LUT # ( 4 ) LUT1
(
    .A(A),
    .B(B[1:0]),
    .Result(wResInt1)
);

LUT # ( 4 ) LUT2
(
    .A(A),
    .B(B[3:2]),
    .Result(wResInt2)
);

assign Result = {wResInt2, 2'b0} + {2'b0, wResInt1};


endmodule
```

Luego para el multiplicador de 16 bits se hizo el siguiente módulo.

```
module Mult16x16
(
```

```
 input wire [15:0] A,
 input wire [15:0] B,
 output wire [31:0] Result
);

wire [17:0] wResInt1,wResInt2,wResInt3,wResInt4,wResInt5,wResInt6,wResInt7,wResInt8;

LUT # ( 16 ) LUT1
(
    .A(A),
    .B(B[1:0]),
    .Result(wResInt1)
);

LUT # ( 16 ) LUT2
(
    .A(A),
    .B(B[3:2]),
    .Result(wResInt2)
);

LUT # ( 16 ) LUT3
(
    .A(A),
    .B(B[5:4]),
    .Result(wResInt3)
);

LUT # ( 16 ) LUT4
(
    .A(A),
    .B(B[7:6]),
    .Result(wResInt4)
);

LUT # ( 16 ) LUT5
(
    .A(A),
    .B(B[9:8]),
    .Result(wResInt5)
);

LUT # ( 16 ) LUT6
(
    .A(A),
    .B(B[11:10]),
    .Result(wResInt6)
);

LUT # ( 16 ) LUT7
(
    .A(A),
    .B(B[13:12]),
    .Result(wResInt7)
);

LUT # ( 16 ) LUT8
(
    .A(A),
    .B(B[15:14]),
    .Result(wResInt8)
);


assign Result = ({wResInt8, 14'b0} + {2'b0, wResInt7, 12'b0}
    + {4'b0, wResInt6, 10'b0}  + {6'b0, wResInt5, 8'b0}
    + {8'b0, wResInt4, 6'b0}   + {8'b0, wResInt3, 4'b0}
    + {10'b0, wResInt2, 2'b0}  + {14'b0, wResInt1});
```

```
71
72 endmodule
```

4.4. Justificación de los resultados obtenidos

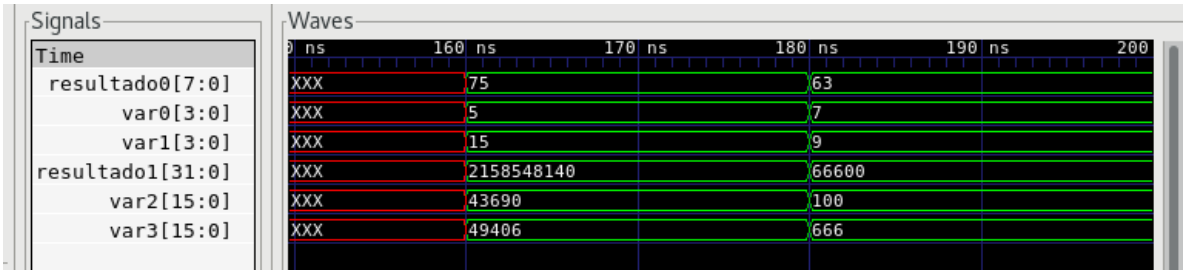

Figura 4: Señales del multiplicador.

La figura 4 muestra una captura de señales de la simulación de ambos módulos multiplicadores. La señales `var0`, `var1` y `resultado0` son respectivamente entradas y salida del multiplicador 4x4. Ídem para `var2`, `var3` y `resultado1` con el multiplicador de 16x16

4.5. Recomendaciones

Una cosa que dio problemas en la implementación del LUT fue definir el cero de la señal `wD0` para que tuviera la longitud adecuada dependiendo del parámetro que recibe, debía ser de tamaño **SIZE+1**.

4.6. Preguntas

4.6.1. ¿Cuántas filas y columnas tendría una LUT que permita multiplicar dos números de 32 bits?

Tendría 2^{32} filas e igual cantidad de columnas.

5. Bibliografía