Laboratorio Nro 1 - ARCH

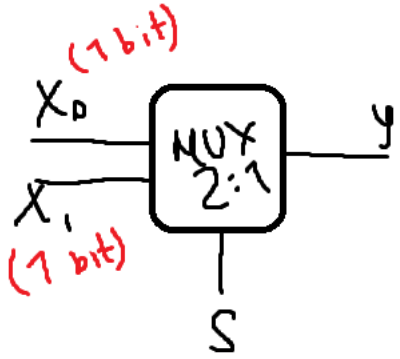
Grover Eduardo Ugarte Quispe - 202020159

Contents

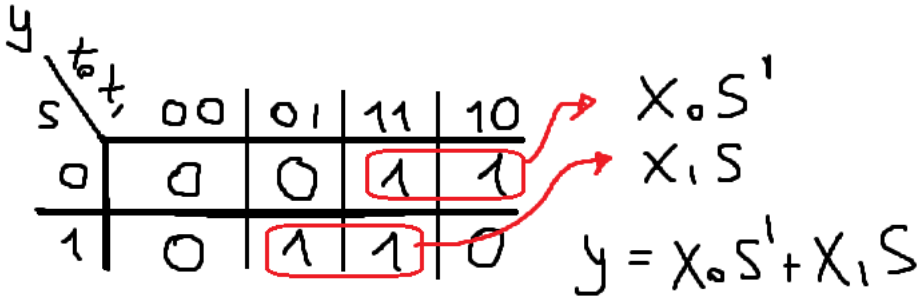
1 Justificación ejercicio 1

1.1 Planteamiento

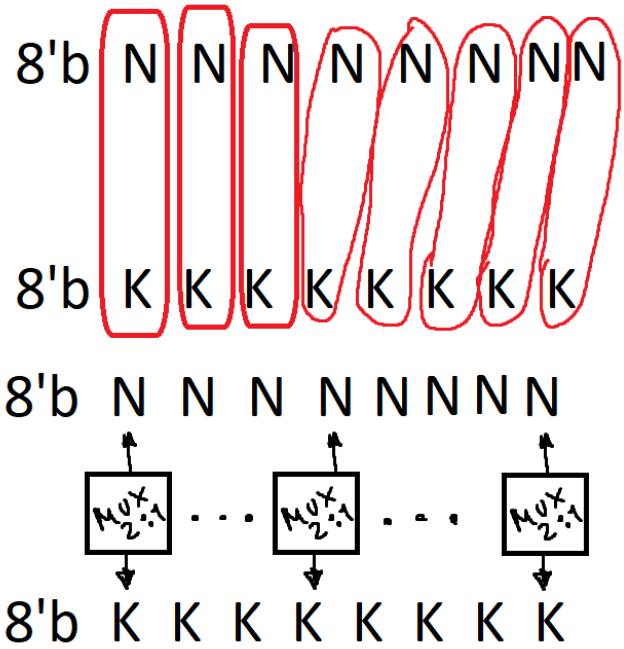
En primer lugar, debemos crear un módulo para representar el MUX2:1. Creamos una tabla con las entradas y salidas para luego generar el mapa de Karnaugh, retornandonos la expresión más reducida. Este MUX está encargado de evaluar dos entradas de 1 bit.



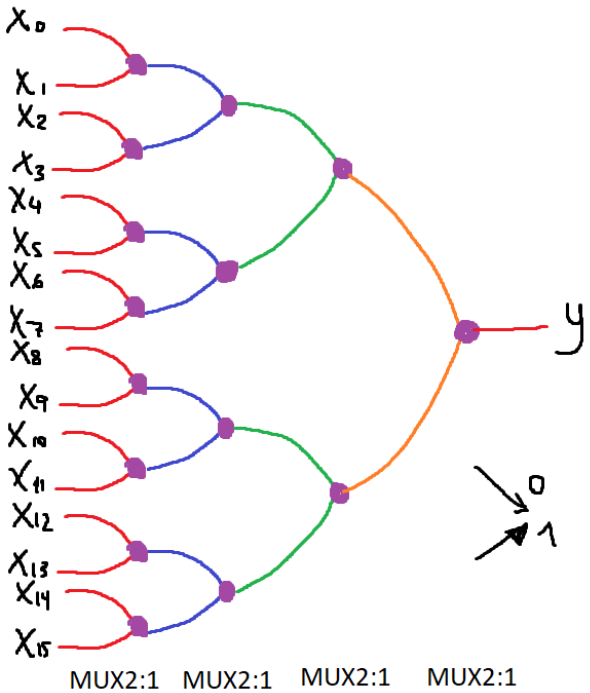
X_0	X_1	S	y
0	0	0	0
0	0	1	0
1	0	0	1
1	0	1	0
0	1	0	0
0	1	1	1
1	1	0	1
1	1	1	1



Con las herramientas actuales, no es posible que el módulo MUX evalúe entradas de 8 bits. Por ese motivo, se creará un módulo que aceptará estas entradas (Buses [7:0]) y empleará 8 veces el módulo MUX previo para de esa forma otorgarnos la señal deseada.



Si tratásemos de crear un MUX16:1 usando solo MUX2:1, podemos ver los MUX2:1 como nodos donde se originan decisiones dentro de un árbol que tiene alcanza las 16 ramas.



Con respecto a los selectores de los 15 MUX2:1 utilizados definir si se utilizarán las 15 entradas de selección o solo algunas. Las observaciones obtenidos son las siguientes:

1. En la primera columna-nodo basta con declarar solo 1 parámetro de selector ya que no es relevante la rama escogida en los otros pares.

2. En la segunda columna-nodo utilizamos únicamente 1 solo parámetro utilizando la misma justifiación que en el punto 1.

3. Tomando en cuenta la irrelevancia de los selectores de los MUX2:1 que no contengan la rama a retornar, nos limitaremos únicamente a 4 estados de selectores. (4 bits individuales)

Con esto en cuenta, creamos el módulo de MUX16:1 evaluando lo presentado en el árbol de deciones. Con respecto al índice de selección S0, se encuentra como evaluador de las 16 ramas y se reduce hasta llegar a S3 que evalúa las 2 ramas restantes.
La jerarquía del módulo mediante el alcance **structural** es el siguiente:

1. Módulo mux16-1-8b-struc (entradas [7:0]x0-15, s3,s2,s1,s0 y salida [7:0]Y) (mux de 16 a 1 exclusivo de 8 bits)

2. Módulo mux2-1-8b-struc (entradas [7:0]x0, [7:0]x1, s y salida [7:0]Y) (mux de 2 a 1 exclusivo de 8 bits) (Al ser behavioral, podemos repetir directamente operador booleanos sin causar demasiado código a comparación del método structural) (Cuenta con 8 módulos mux-2-1-struc que evaluan 1 bit en la misma posición de las entradas)

3. Módulo mux-2-1-struc (entradas x0, x1, s y salida Y) (mux de 2 a 1 de 1 solo bit)

La jerarquía del módulo mediante el alcance **behavioral** es el siguiente:

1. Módulo mux16-1-8b-behav (entradas [7:0]x0-15, s3,s2,s1,s0 y salida [7:0]Y) (mux de 16 a 1 exclusivo de 8 bits)

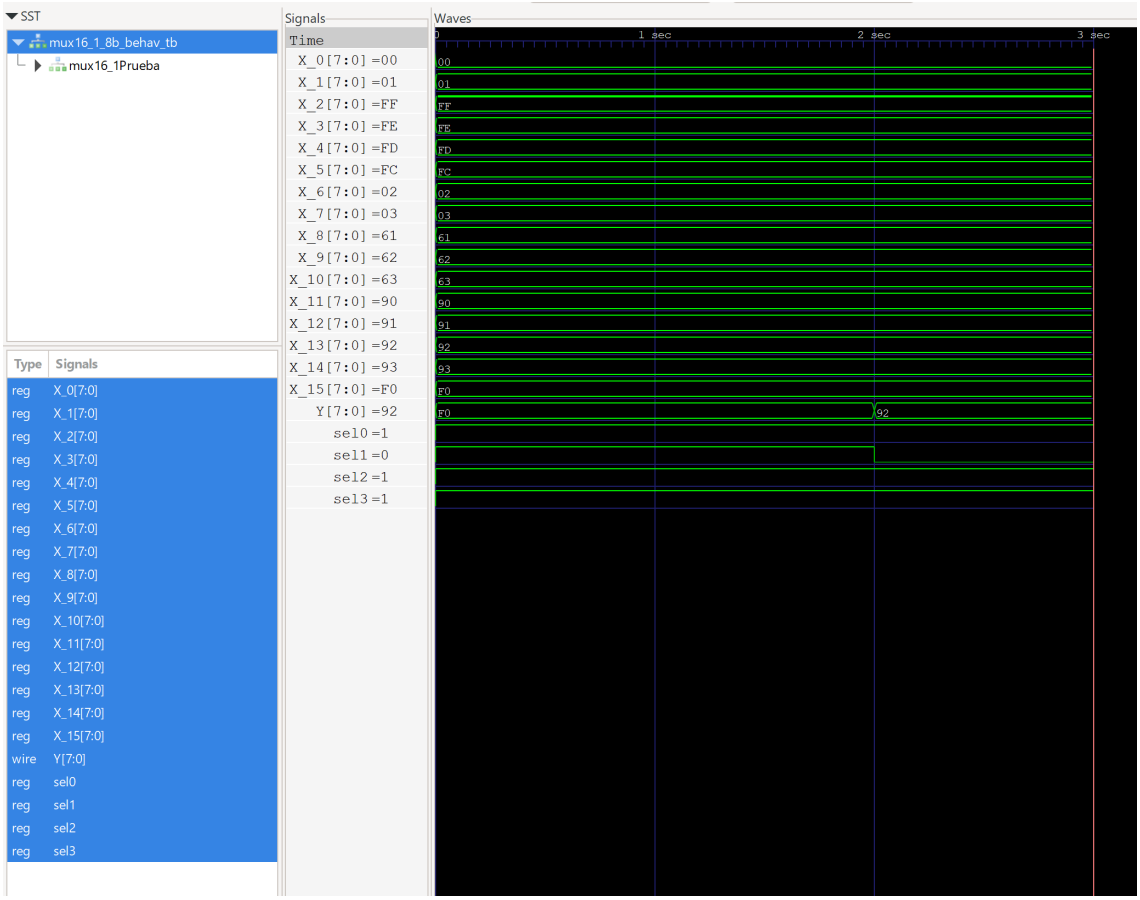
2. Módulo mux2-1-8b-behav (entradas [7:0]x0, [7:0]x1, s y salida [7:0]Y) (mux de 2 a 1 exclusivo de 8 bits) (Al ser behavioral, podemos repetir directamente operador booleanos sin causar demasiado código a comparación del método structural)

1.2 Testbench

Tratamos de que el MUX retorne valores de dos ramas siguiendo la composición del gráfico previo. Hacemos este procedimiento por cada método.

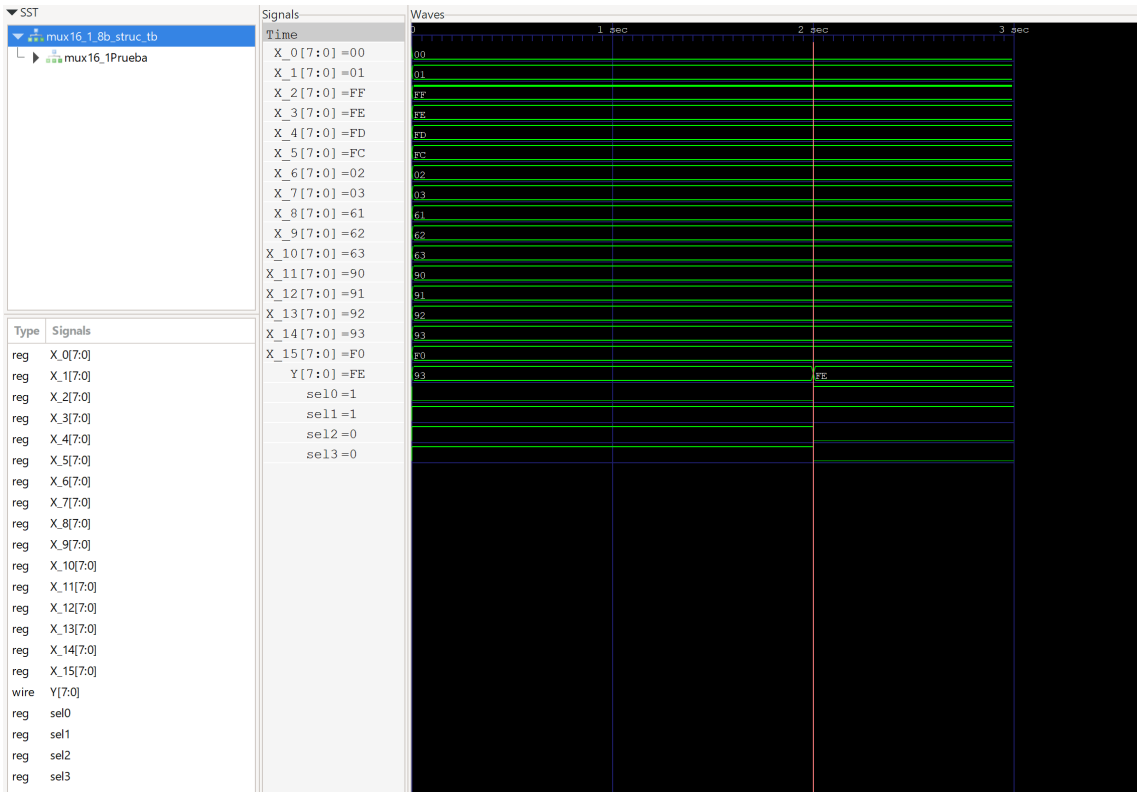
1.2.1 BEHAVIORAL

```
Grove@LAPTOP-2A658HOR MINGW64 ~/Documents/__UTEC__/Grupo 4/CS2201/Lab/tarea1/entregable/Ejercicio 1 (master)
$ iverilog mux16_1_8b_behav_tb.v mux16_1_8b_behav.v && vvp a.out
x0:   0, x1:   1, x2: 255, x3: 254, x4: 253, x5: 252, x6:   2, x7:   3, x8:  97, x9:  98, x10:  99, x11: 144, x12: 145, x13: 146, x14: 147, x15: 240
VCD info: dumpfile mux16_1_8b_behav.vcd opened for output.
S3: 1, S2: 1, S1: 1, S0: 1 -> Y: 240
S3: 1, S2: 1, S1: 0, S0: 1 -> Y: 146
```



1.2.2 STRUCTURAL

```
Grove@LAPTOP-2A658HOR MINGW64 ~/Documents/__UTEC__/Grupo 4/CS2201/Lab/tarea1/entregable/Ejercicio 1 (master)
$ iverilog mux16_1_8b_struc_tb.v mux16_1_8b_struc.v && vvp a.out
x0:   0, x1:   1, x2: 255, x3: 254, x4: 253, x5: 252, x6:   2, x7:   3, x8:  97, x9:  98, x10:  99, x11: 144, x12: 145, x13: 146, x14: 147, x15: 240
VCD info: dumpfile mux16_1_8b_struc.vcd opened for output.
S3: 1, S2: 1, S1: 1, S0: 0 -> Y: 147
S3: 0, S2: 0, S1: 1, S0: 1 -> Y: 254
```

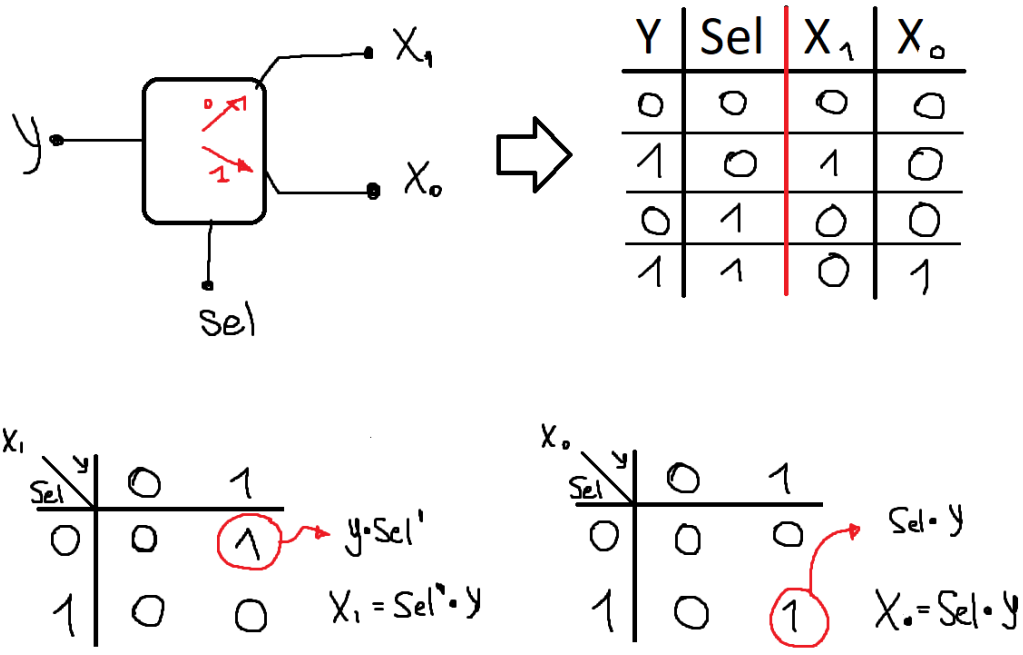

SST
mux16_1_8b_struc_tb
mux16_1Prueba
Type
Signals
reg X_0[7:0]
reg X_1[7:0]
reg X_2[7:0]
reg X_3[7:0]
reg X_4[7:0]
reg X_5[7:0]
reg X_6[7:0]
reg X_7[7:0]
reg X_8[7:0]
reg X_9[7:0]
reg X_10[7:0]
reg X_11[7:0]
reg X_12[7:0]
reg X_13[7:0]
reg X_14[7:0]
reg X_15[7:0]
wire Y[7:0]
reg sel0
reg sel1
reg sel2
reg sel3
Signals
Time
X_0[7:0] =00
X_1[7:0] =01
X_2[7:0] =FF
X_3[7:0] =FE
X_4[7:0] =FD
X_5[7:0] =FC
X_6[7:0] =02
X_7[7:0] =03
X_8[7:0] =61
X_9[7:0] =62
X_10[7:0] =63
X_11[7:0] =90
X_12[7:0] =91
X_13[7:0] =92
X_14[7:0] =93
X_15[7:0] =F0
Y[7:0] =FE
sel0 =1
sel1 =1
sel2 =0
sel3 =0
Waves
1 sec
2 sec
3 sec

2 Justificación ejercicio 2

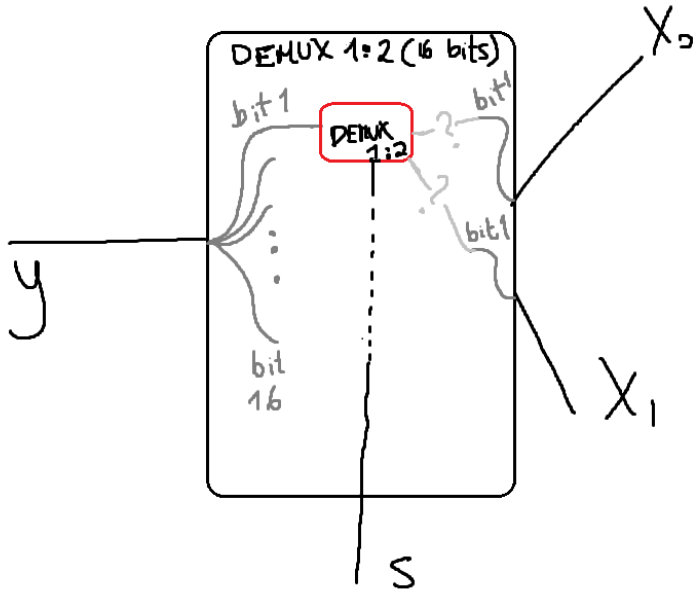
2.1 PARTE A

2.1.1 Planteamiento

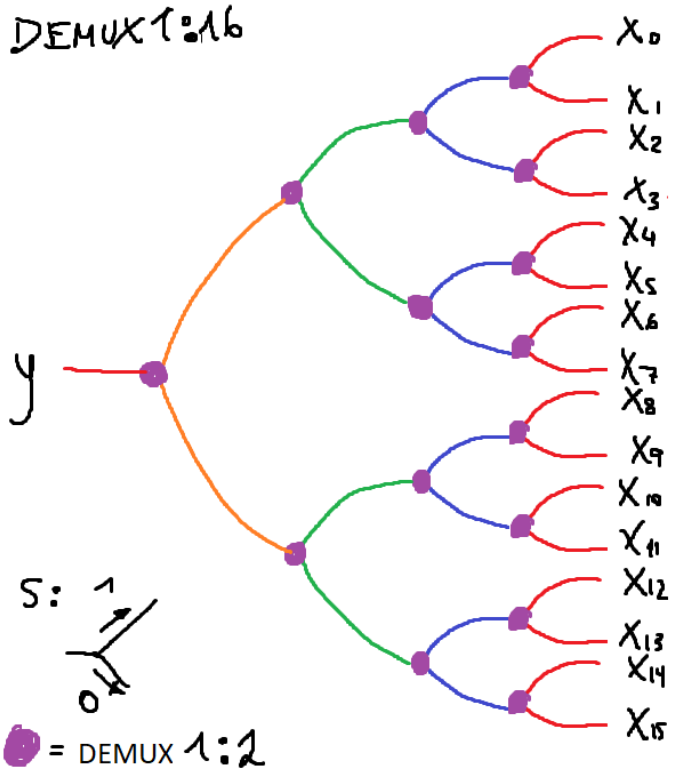
Planteamos el funcionamiento del DEMUX1:2 como un módulo capaz de enviar una señal de entrada únicamente por una sola de sus dos salidas por medio de un selector. Teniendo esto en cuenta, generamos una tabla con todas las salidas y entradas para obtener el K-map y la expresión para determinar el estado de cada una de sus salidas.



Similar al ejercicio anterior, debemos crear un módulo capaz de aceptar entradas con entradas de 16 bits. Para diseñarlo se utilizará la idea del anterior ejercicio la cual era evaluar un bit a la vez durante el transcurso de sus 16 bits.



Utilizando lo visto previamente, ahora se aplicará el concepto de árbol de decisiones pero aplicado inversamente.



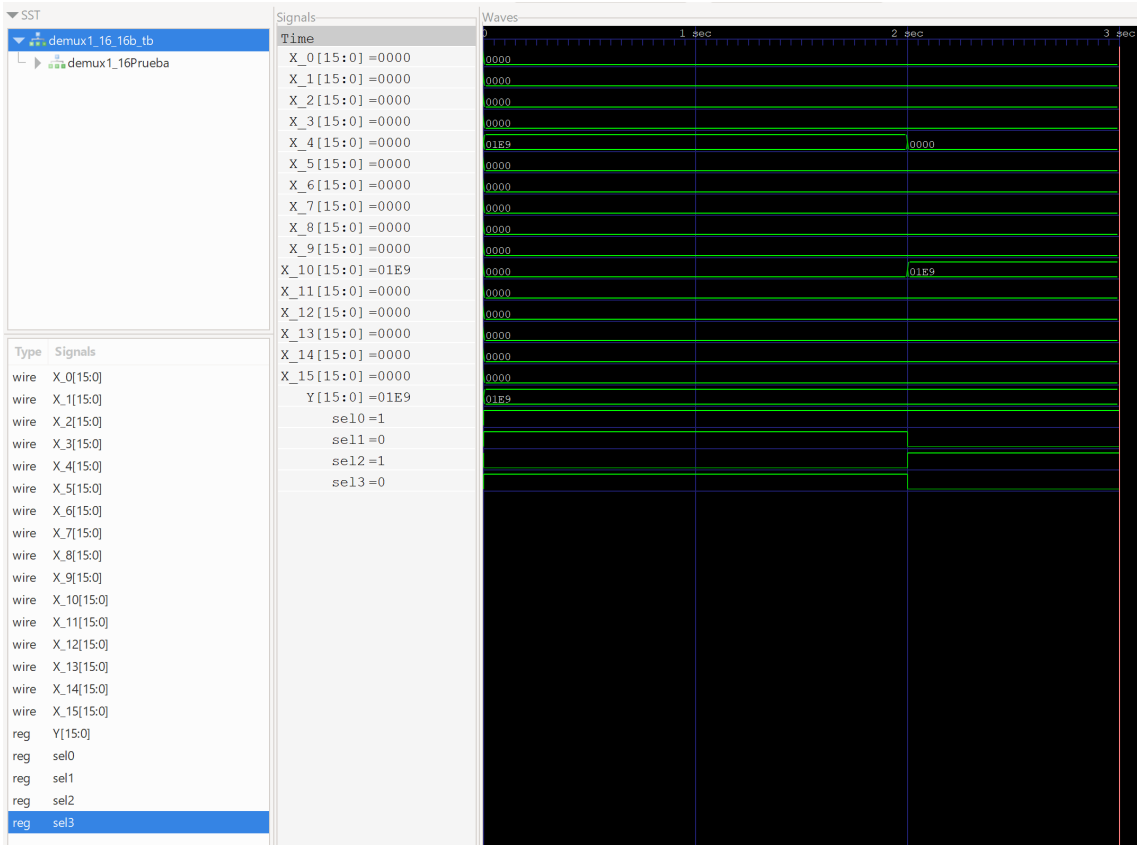
Con respecto al manejo de entradas seleccion, nos planteamos un ejemplo de querer retornar la entrada mediante la rama X4. La seleccion del primer, segundo, tercero y cuarto DEMUX1:2 tendrá los valores 1,0,1,1 respectivamente. Nos percatamos de que si aplicasemos aquellos estados en los DEMUX restantes de cada columna, la salida objetivo se vería alterada. Por lo que para evitar redundancia, se diseñara aquel DEMUX1:16 teniendo solamente con 4 valores de seleccion. S3(primera rama), S2(segunda rama), S1(tercera rama), S0(cuarta rama)
Nuestra jerarquía del módulo demux1-16 queda de la siguiente forma:

1. Módulo demux1-16-16b(entrada [15:0]Y,S3,S2,S1,S0 y salidas [15:0]x0-15) (demux de 1 a 16 que solo acepta 16 bits)
2. Módulo demux1-2-16b (entrada [15:0]Y,S y salidas [15:0]x0, [15:0]x1) (demux de 1 a 2 que solo acepta 16 bits)
3. Módulo demux1-2 (entrada Y,S y salidas x0, x1) (demux de 1 a 2 que solo acepta 1 bit)

2.1.2 Testbench

Probamos el DEMUX1:16 al asignar los estados a las entradas de selección y se verifica si se llega a una rama establecida según la imagen del arbol presentada anteriormente. (Se probó con dos ramas x4 y x10)

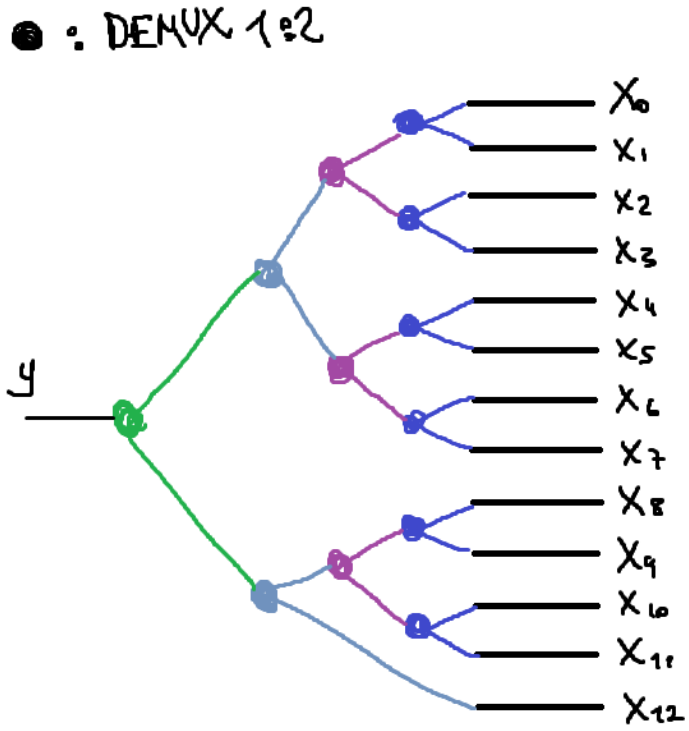
```
Grove@LAPTOP-2A658HOR MINGW64 ~/Documents/__UTEC__/Grupo 4/CS2201/Lab/tarea1/entregable/Ejercicio 2 (master)
$ iverilog demux1_16_16b_tb.v demux1_16_16b.v && vvp a.out
input: 01e9, s3: 1, s2: 0, s1: 1, s0: 1
VCD info: dumpfile demux1_16_16b.vcd opened for output.
x0: 0000, x1: 0000, x2: 0000, x3: 0000, x4: 01e9, x5: 0000, x6: 0000, x7: 0000, x8: 0000, x9: 0000, x10: 0000, x11: 0000, x12: 0000, x13: 0000, x14: 0000, x15: 0000
input: 01e9, s3: 0, s2: 1, s1: 0, s0: 1
x0: 0000, x1: 0000, x2: 0000, x3: 0000, x4: 0000, x5: 0000, x6: 0000, x7: 0000, x8: 0000, x9: 0000, x10: 01e9, x11: 0000, x12: 0000, x13: 0000, x14: 0000, x15: 0000
```



2.2 PARTE B

Un DEMUX de 13 salidas **puede** ser creado utilizando únicamente DEMUX1:2. Sin embargo, se tienen las siguientes observaciones:

- El árbol para plantear la estructura no sería simétrico ya que la simetría ocurre cuando el número de salidas es una potencia de 2.
- Una rama no necesitaría de 4 selectores, basta con 2. (En el caso del gráfico, sería la rama X12)
- Son necesarios 12 DEMUX1:2 y 4 entradas de selección ya que las columnas-nodo no son afectadas.



3 Justificación ejercicio 3

3.1 Planteamiento

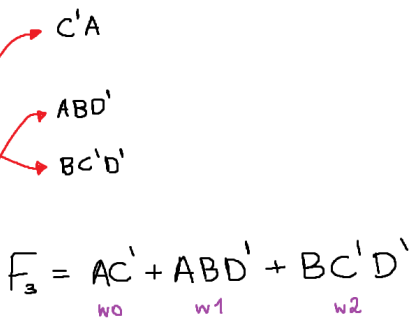
Debemos crear mapas de Karnaugh con respecto a los resultados de "mayor a", "menor a" e "igual" con el fin de obtener expresiones booleanas minimizadas.

3.1.1 Caso AB>CD (F3)

Armamos y resolvemos el K-Map, asegurandonos que la última expersión hallada tenga la menor cantidad de operaciones posibles.

AB > CD

CD \ AB	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

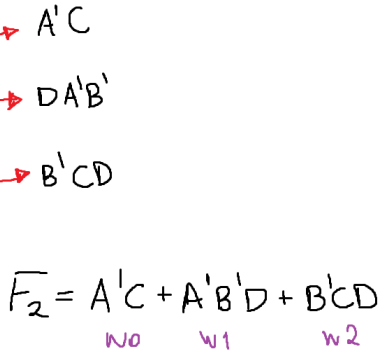


3.1.2 Caso AB<CD (F2)

Armamos y resolvemos el K-Map, asegurandonos que la última expersión hallada tenga la menor cantidad de operaciones posibles.

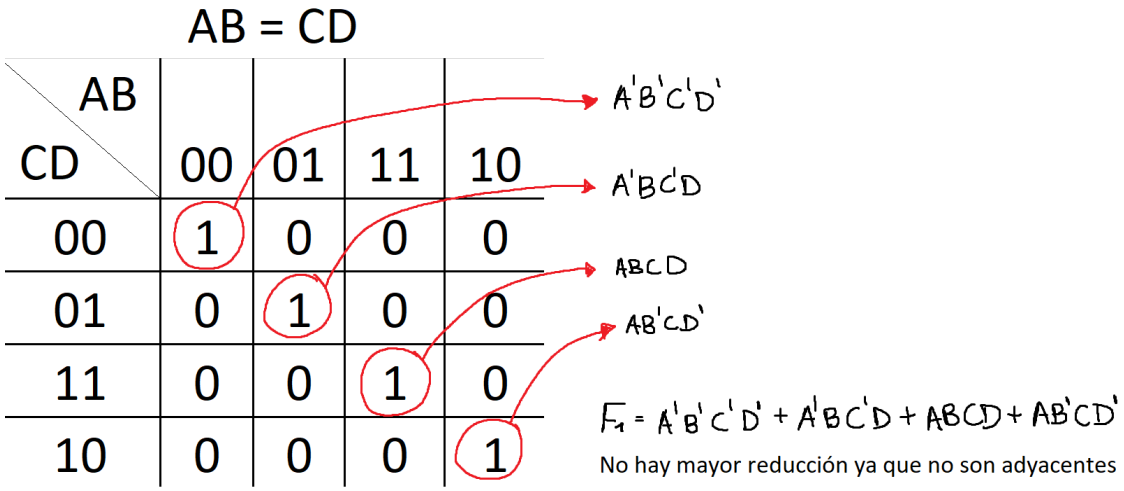
AB < CD

CD \ AB	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0



3.1.3 Caso AB=CD (F1)

Armamos y resolvemos el K-Map, asegurandonos que la última expersión hallada tenga la menor cantidad de operaciones posibles.



Como son términos adyacentes diagonales, no es posible reducir más la expresión hallada.

Utilizando las expresiones lógicas, se crea un módulo para AB > CD, AB = CD y AB < CD. Finalmente, se ensambla el módulo comparador que dará las respectivas salidas F1, F2 y F3. (Como observación es posible no crear un módulo para = si es que > y < ya estan creados. Solo basta con saber que ambos tienen un estado 0 para determinar que = sería 1 utilizando una compuerta NOR).

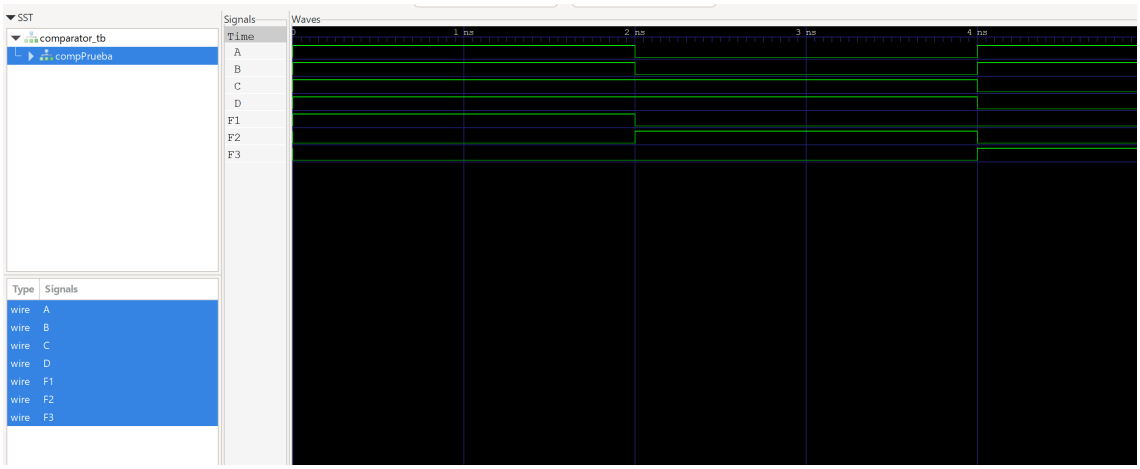
Para generar un módulo de acuerdo al ejercicio se tiene la siguiente jerarquía:

1. Comparator (entradas A,B,C,D y salidas F1-,F2-,F3-)
2. F1 (entradas A,B,C,D y salida F1-)
3. F2 (entradas A,B,C,D y salida F2-)
4. F3 (entradas A,B,C,D y salida F3-)

3.2 Testbench

Debido a que son múltiples combinaciones, optamos por evaluar los tres casos de comparación.

```
Grove@LAPTOP-2A658HOR MINGW64 ~/Documents/__UTEC__/Grupo 4
$ iverilog comparator_tb.v comparator.v  && vvp a.out
AB: 11  CD: 11
VCD info: dumpfile comparator.vcd opened for output.
F1(AB = CD): 1   F2(AB < CD): 0          F3(AB > CD): 0
AB: 00  CD: 01
F1(AB = CD): 0   F2(AB < CD): 1          F3(AB > CD): 0
AB: 11  CD: 10
F1(AB = CD): 0   F2(AB < CD): 0          F3(AB > CD): 1
```



4 Justificación ejercicio 4

4.1 Planteamiento

Para resolver los mapas de Karnaugh formados a partir de las tablas presentadas, debemos aprovechar las casillas marcadas sin interés con el propóito de conseguir expresiones más cortas.

W

AB / CD	00	01	11	10
00	0	0	X	1
01	0	0	X	0
11	0	1	X	X
10	0	0	X	X

AC'D'

BCD

W = AC'D' + BCD

X

AB / CD	00	01	11	10
00	0	1	X	0
01	0	1	X	0
11	1	0	X	X
10	0	1	X	X

BC'

B'CD

BCD'

X = BC' + B'CD + BCD'

X = BC' + C·(B ⊕ D)

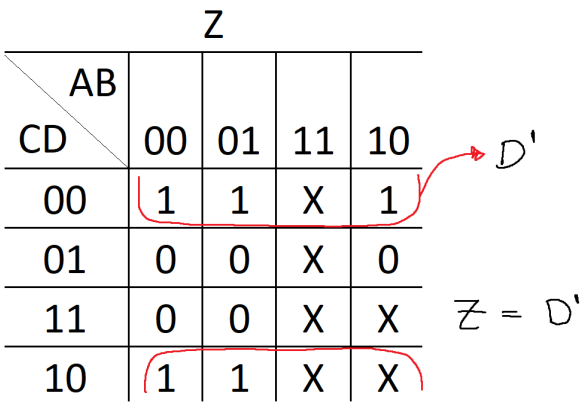
Y

AB / CD	00	01	11	10
00	0	0	X	0
01	1	1	X	0
11	0	0	X	X
10	1	1	X	X

A'C'D

CD'

Y = A'C'D + CD'



Para generar un módulo de acuerdo al ejercicio se tiene la siguiente jerarquía:

1. Módulo BCD (entradas A,B,C,D y salidas W,X,Y,Z)
2. Módulo dígito-W (entradas A,B,C,D y salida W)
3. Módulo dígito-X (entradas A,B,C,D y salida X)
4. Módulo dígito-Y (entradas A,B,C,D y salida Y)
5. Módulo dígito-Z (entradas A,B,C,D y salida Z)

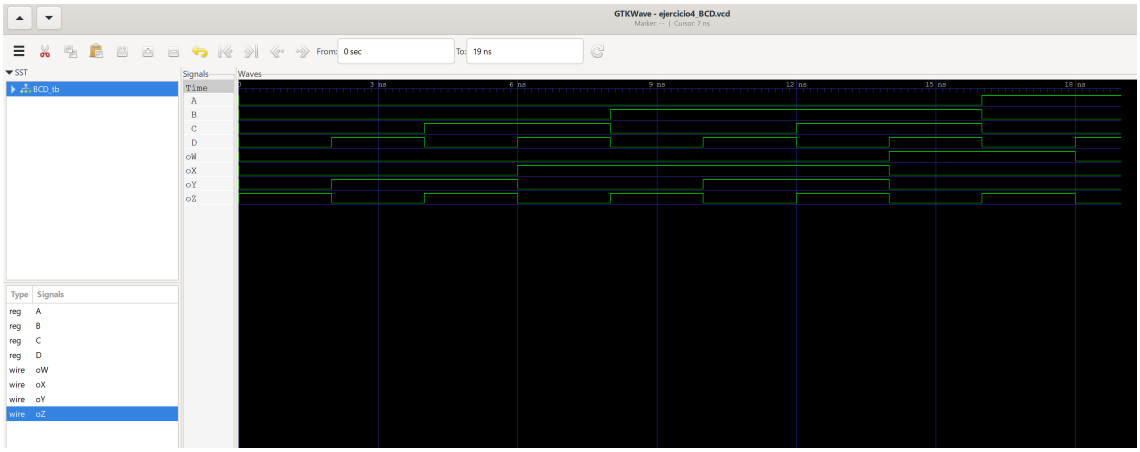
Debido a que se empleaban compuertas lógicas de múltiples entradas, se optó por crear cada submódulo del dígito W, X, Y y Z de modo behavioral. Tras crear aquellos submódulos, se les incorporó al módulo que realiza el BCD.

4.2 Testbench

Debido a que son solo 10 posibles entradas, creamos nuestro testbench con cada combinación.

```
Grove@LAPTOP-2A658HOR MINGW64 ~/Documents/__UTEC__/Grupo 4/CS2201/Lab/tarea1/entregable/ejercicio4 (master)
$ iverilog BCD_tb.v BCD.v

Grove@LAPTOP-2A658HOR MINGW64 ~/Documents/__UTEC__/Grupo 4/CS2201/Lab/tarea1/entregable/ejercicio4 (master)
$ vvp a.out
ABCD: 0000
VCD info: dumpfile BCD.vcd opened for output.
ABCD: 0000 -> WXYZ: 0001
ABCD: 0001
ABCD: 0001 -> WXYZ: 0010
ABCD: 0010
ABCD: 0010 -> WXYZ: 0011
ABCD: 0011
ABCD: 0011 -> WXYZ: 0100
ABCD: 0100
ABCD: 0100 -> WXYZ: 0101
ABCD: 0101
ABCD: 0101 -> WXYZ: 0110
ABCD: 0110
ABCD: 0110 -> WXYZ: 0111
ABCD: 0111
ABCD: 0111 -> WXYZ: 1000
ABCD: 1000
ABCD: 1000 -> WXYZ: 1001
ABCD: 1001
ABCD: 1001 -> WXYZ: 0000
```


GTKWave - ejercicio4_BCD.vcd
From: 0 sec
To: 19 ns
SST
BCD_tb
Signals
Waves
Time
A
B
C
D
oW
oX
oY
oZ
3 ns
6 ns
9 ns
12 ns
15 ns
18 ns
Type
Signals
reg A
reg B
reg C
reg D
wire oW
wire oX
wire oY
wire oZ

5 Anexos: Códigos

5.1 Ejercicio 1

5.1.1 Ensamble *structural*

MUX2:1 de 1 bit

```
module mux2_1_struc (
    input  in0, input  in1, input sel, output  out
);
    wire sel_not, op_in0, op_in1;
    not invert_sel(sel_not, sel);

    and choose_in0(op_in0, sel_not, in0);
    and choose_in1(op_in1, sel, in1);

    or salida(out, op_in1, op_in0);
endmodule
```

MUX2:1 de 8 bits

```
//`include "mux2_1_struc.v"
module mux2_1_8b_struc (
    input [7:0] in0, input [7:0] in1, input sel, output [7:0] out
);
    mux2_1_struc bit0(in0[0], in1[0], sel, out[0]);
    mux2_1_struc bit1(in0[1], in1[1], sel, out[1]);
    mux2_1_struc bit2(in0[2], in1[2], sel, out[2]);
    mux2_1_struc bit3(in0[3], in1[3], sel, out[3]);
    mux2_1_struc bit4(in0[4], in1[4], sel, out[4]);
    mux2_1_struc bit5(in0[5], in1[5], sel, out[5]);
    mux2_1_struc bit6(in0[6], in1[6], sel, out[6]);
    mux2_1_struc bit7(in0[7], in1[7], sel, out[7]);
endmodule
```

MUX16:1 de 8 bits

```
`include "mux2_1_struc.v"
`include "mux2_1_8b_struc.v"
module mux16_1_8b_struc (
    input [7:0] in0, input [7:0] in1, input [7:0] in2, input [7:0] in3, input [7:0] in4,
    input [7:0] in5, input [7:0] in6, input [7:0] in7, input [7:0] in8, input [7:0] in9,
    input [7:0] in10, input [7:0] in11, input [7:0] in12, input [7:0] in13, input [7:0] in14
    , input [7:0] in15, input sel3, input sel2, input sel1, input sel0, output [7:0] out
);
    //1era evaluacion
    wire [7:0] eva1[7:0];
    mux2_1_8b_struc mux01(in0, in1, sel0, eva1[0]);
    mux2_1_8b_struc mux23(in2, in3, sel0, eva1[1]);
    mux2_1_8b_struc mux45(in4, in5, sel0, eva1[2]);
    mux2_1_8b_struc mux67(in6, in7, sel0, eva1[3]);
    mux2_1_8b_struc mux89(in8, in9, sel0, eva1[4]);
    mux2_1_8b_struc mux1011(in10, in11, sel0, eva1[5]);
    mux2_1_8b_struc mux1213(in12, in13, sel0, eva1[6]);
    mux2_1_8b_struc mux1415(in14, in15, sel0, eva1[7]);

    //2da evaluacion
    wire [7:0] eva2[3:0];
    mux2_1_8b_struc eva1_01(eva1[0], eva1[1], sel1, eva2[0]);
    mux2_1_8b_struc eva1_23(eva1[2], eva1[3], sel1, eva2[1]);
    mux2_1_8b_struc eva1_45(eva1[4], eva1[5], sel1, eva2[2]);
    mux2_1_8b_struc eva1_67(eva1[6], eva1[7], sel1, eva2[3]);

    //3era evaluacion
    wire [7:0] eva3[1:0];
```

```
    mux2_1_8b_struc eva2_01(eva2[0], eva2[1], sel2, eva3[0]);
    mux2_1_8b_struc eva2_23(eva2[2], eva2[3], sel2, eva3[1]);

    //ultima evaluacion
    mux2_1_8b_struc eva_final(eva3[0], eva3[1], sel3, out);

endmodule
```

TestBench del MUX16:1 de 8 bits

```
module mux16_1_8b_struc_tb;

    reg [7:0] X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_10, X_11, X_12, X_13, X_14
    , X_15;
    reg sel3, sel2, sel1, sel0;
    wire [7:0] Y;

    mux16_1_8b_struc mux16_1Prueba(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_10,
    X_11, X_12, X_13, X_14, X_15, sel3, sel2, sel1, sel0, Y);

    initial begin
        X_0=8'b00000000;
        X_1=8'b00000001;
        X_2=8'b11111111;
        X_3=8'b11111110;
        X_4=8'b11111101;
        X_5=8'b11111100;
        X_6=8'b00000010;
        X_7=8'b00000011;
        X_8=8'b01100001;
        X_9=8'b01100010;
        X_10=8'b01100011;
        X_11=8'b10010000;
        X_12=8'b10010001;
        X_13=8'b10010010;
        X_14=8'b10010011;
        X_15=8'b11110000;

        $display("x0: %d, x1: %d, x2: %d, x3: %d, x4: %d, x5: %d, x6: %d, x7: %d, x8: %d, x9
    : %d, x10: %d, x11: %d, x12: %d, x13: %d, x14: %d, x15: %d", X_0, X_1, X_2, X_3, X_4,
    X_5, X_6, X_7, X_8, X_9, X_10, X_11, X_12, X_13, X_14, X_15);
        sel0=0;
        sel1=1;
        sel2=1;
        sel3=1;
        #1
        $display("S3: %b, S2: %b, S1: %b, S0: %b -> Y: %d",sel3, sel2, sel1, sel0, Y);
        #1
        sel0=1;
        sel1=1;
        sel2=0;
        sel3=0;
        #1
        $display("S3: %b, S2: %b, S1: %b, S0: %b -> Y: %d",sel3, sel2, sel1, sel0, Y);

    end
    initial begin
        $dumpfile("mux16_1_8b_struc.vcd");
        $dumpvars;
    end


endmodule
```

5.1.2 Ensamble *behavioral*

MUX2:1 de 8 bits

```
module mux2_1_8b_behav (
    input [7:0] in0, input [7:0] in1, input sel, output [7:0] out
);
    assign out[0] = in0[0]&~sel | in1[0]&sel;
    assign out[1] = in0[1]&~sel | in1[1]&sel;
    assign out[2] = in0[2]&~sel | in1[2]&sel;
    assign out[3] = in0[3]&~sel | in1[3]&sel;
    assign out[4] = in0[4]&~sel | in1[4]&sel;
    assign out[5] = in0[5]&~sel | in1[5]&sel;
    assign out[6] = in0[6]&~sel | in1[6]&sel;
    assign out[7] = in0[7]&~sel | in1[7]&sel;
endmodule
```

MUX16:1 de 8 bits

```
`include "mux2_1_8b_behav.v"
module mux16_1_8b_behav (
    input [7:0] in0, input [7:0] in1, input [7:0] in2, input [7:0] in3, input [7:0] in4,
    input [7:0] in5, input [7:0] in6, input [7:0] in7, input [7:0] in8, input [7:0] in9,
    input [7:0] in10, input [7:0] in11, input [7:0] in12, input [7:0] in13, input [7:0] in14
    , input [7:0] in15, input sel3, input sel2, input sel1, input sel0, output [7:0] out
);
    //1era evaluacion
    wire [7:0] eva1[7:0];
    mux2_1_8b_behav mux01(in0, in1, sel0, eva1[0]);
    mux2_1_8b_behav mux23(in2, in3, sel0, eva1[1]);
    mux2_1_8b_behav mux45(in4, in5, sel0, eva1[2]);
    mux2_1_8b_behav mux67(in6, in7, sel0, eva1[3]);
    mux2_1_8b_behav mux89(in8, in9, sel0, eva1[4]);
    mux2_1_8b_behav mux1011(in10, in11, sel0, eva1[5]);
    mux2_1_8b_behav mux1213(in12, in13, sel0, eva1[6]);
    mux2_1_8b_behav mux1415(in14, in15, sel0, eva1[7]);

    //2da evaluacion
    wire [7:0] eva2[3:0];
    mux2_1_8b_behav eva1_01(eva1[0], eva1[1], sel1, eva2[0]);
    mux2_1_8b_behav eva1_23(eva1[2], eva1[3], sel1, eva2[1]);
    mux2_1_8b_behav eva1_45(eva1[4], eva1[5], sel1, eva2[2]);
    mux2_1_8b_behav eva1_67(eva1[6], eva1[7], sel1, eva2[3]);

    //3era evaluacion
    wire [7:0] eva3[1:0];
    mux2_1_8b_behav eva2_01(eva2[0], eva2[1], sel2, eva3[0]);
    mux2_1_8b_behav eva2_23(eva2[2], eva2[3], sel2, eva3[1]);

    //ultima evaluacion
    mux2_1_8b_behav eva_final(eva3[0], eva3[1], sel3, out);

endmodule
```

TestBench del MUX16:1 de 8 bits

```
module mux16_1_8b_behav_tb;

    reg [7:0] X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_10, X_11, X_12, X_13, X_14
    , X_15;
    reg sel3, sel2, sel1, sel0;
    wire [7:0] Y;

    mux16_1_8b_behav mux16_1Prueba(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_10,
    X_11, X_12, X_13, X_14, X_15, sel3, sel2, sel1, sel0, Y);

    initial begin
```

```
        X_0=8'b00000000;
        X_1=8'b00000001;
        X_2=8'b11111111;
        X_3=8'b11111110;
        X_4=8'b11111101;
        X_5=8'b11111100;
        X_6=8'b00000010;
        X_7=8'b00000011;
        X_8=8'b01100001;
        X_9=8'b01100010;
        X_10=8'b01100011;
        X_11=8'b10010000;
        X_12=8'b10010001;
        X_13=8'b10010010;
        X_14=8'b10010011;
        X_15=8'b11110000;

        $display("x0: %d, x1: %d, x2: %d, x3: %d, x4: %d, x5: %d, x6: %d, x7: %d, x8: %d, x9
    : %d, x10: %d, x11: %d, x12: %d, x13: %d, x14: %d, x15: %d", X_0, X_1, X_2, X_3, X_4,
    X_5, X_6, X_7, X_8, X_9, X_10, X_11, X_12, X_13, X_14, X_15);
        sel0=1;
        sel1=1;
        sel2=1;
        sel3=1;
        #1
        $display("S3: %b, S2: %b, S1: %b, S0: %b -> Y: %d",sel3, sel2, sel1, sel0, Y);
        #1
        sel0=1;
        sel1=0;
        sel2=1;
        sel3=1;
        #1
        $display("S3: %b, S2: %b, S1: %b, S0: %b -> Y: %d",sel3, sel2, sel1, sel0, Y);

    end
    initial begin
        $dumpfile("mux16_1_8b_behav.vcd");
        $dumpvars;
    end


endmodule
```

5.2 Ejercicio 2

DEMUX1:2 de 1 bit

```
module demux1_2 (
    input i_Y, input i_Sel,  output o_X0, output o_X1
);
    assign o_X1 = ~i_Sel & i_Y;
    assign o_X0 = i_Sel & i_Y;

endmodule
```

DEMUX1:2 de 16 bits

```
//`include "demux1_2.v"

module demux1_2_16b (
    input [15:0] i_Y, input i_Sel,  output [15:0] o_X0, output [15:0] o_X1
);
    demux1_2 demuxSingular1(i_Y[0], i_Sel, o_X0[0], o_X1[0]);
    demux1_2 demuxSingular2(i_Y[1], i_Sel, o_X0[1], o_X1[1]);
```

```
    demux1_2 demuxSingular3(i_Y[2], i_Sel, o_X0[2], o_X1[2]);
    demux1_2 demuxSingular4(i_Y[3], i_Sel, o_X0[3], o_X1[3]);
    demux1_2 demuxSingular5(i_Y[4], i_Sel, o_X0[4], o_X1[4]);
    demux1_2 demuxSingular6(i_Y[5], i_Sel, o_X0[5], o_X1[5]);
    demux1_2 demuxSingular7(i_Y[6], i_Sel, o_X0[6], o_X1[6]);
    demux1_2 demuxSingular8(i_Y[7], i_Sel, o_X0[7], o_X1[7]);
    demux1_2 demuxSingular9(i_Y[8], i_Sel, o_X0[8], o_X1[8]);
    demux1_2 demuxSingular10(i_Y[9], i_Sel, o_X0[9], o_X1[9]);
    demux1_2 demuxSingular11(i_Y[10], i_Sel, o_X0[10], o_X1[10]);
    demux1_2 demuxSingular12(i_Y[11], i_Sel, o_X0[11], o_X1[11]);
    demux1_2 demuxSingular13(i_Y[12], i_Sel, o_X0[12], o_X1[12]);
    demux1_2 demuxSingular14(i_Y[13], i_Sel, o_X0[13], o_X1[13]);
    demux1_2 demuxSingular15(i_Y[14], i_Sel, o_X0[14], o_X1[14]);
    demux1_2 demuxSingular16(i_Y[15], i_Sel, o_X0[15], o_X1[15]);
endmodule
```

DEMUX1:16 de 16 bits

```
`include "demux1_2.v"
`include "demux1_2_16b.v"
module demux1_16_16b (
    input [15:0] Y, input Sel3, Sel2, Sel1, Sel0, output [15:0] x_0, x_1, x_2, x_3, x_4, x_5
    , x_6, x_7, x_8, x_9, x_10, x_11, x_12, x_13, x_14, x_15
);
    //primera evaluacion demux
    wire[15:0] y0, y1;
    demux1_2_16b eva(Y, Sel3, y0, y1);
    //segunda evaluacion ''
    wire[15:0] y00, y01, y10, y11;
    demux1_2_16b eva0(y0, Sel2, y00, y01);
    demux1_2_16b eva1(y1, Sel2, y10, y11);
    //tercera '' ''
    wire[15:0] y000, y001, y010, y011, y100, y101, y110, y111;
    demux1_2_16b eva00(y00, Sel1, y000, y001);
    demux1_2_16b eva01(y01, Sel1, y010, y011);
    demux1_2_16b eva10(y10, Sel1, y100, y101);
    demux1_2_16b eva11(y11, Sel1, y110, y111);
    //final '' ''
    demux1_2_16b eva000(y000, Sel0, x_0, x_1);
    demux1_2_16b eva001(y001, Sel0, x_2, x_3);
    demux1_2_16b eva010(y010, Sel0, x_4, x_5);
    demux1_2_16b eva011(y011, Sel0, x_6, x_7);
    demux1_2_16b eva100(y100, Sel0, x_8, x_9);
    demux1_2_16b eva101(y101, Sel0, x_10, x_11);
    demux1_2_16b eva110(y110, Sel0, x_12, x_13);
    demux1_2_16b eva111(y111, Sel0, x_14, x_15);
endmodule
```

TestBench del DEMUX1:16 de 16 bits

```
module demux1_16_16b_tb;

    reg [15:0] Y;
    reg sel3, sel2, sel1, sel0;
    wire [15:0] X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_10, X_11, X_12, X_13,
    X_14, X_15;

    demux1_16_16b demux1_16Prueba(Y, sel3, sel2, sel1, sel0, X_0, X_1, X_2, X_3, X_4, X_5,
    X_6, X_7, X_8, X_9, X_10, X_11, X_12, X_13, X_14, X_15);

    initial begin
        Y = 16'b111101001;//%H
        sel3 = 1;
        sel2 = 0;
        sel1 = 1;
        sel0 = 1;
```

```
        $display("input: %H, s3: %b, s2: %b, s1: %b, s0: %b", Y, sel3, sel2, sel1, sel0);
        #1
        $display("x0: %h, x1: %h, x2: %h, x3: %h, x4: %h, x5: %h, x6: %h, x7: %h, x8: %h, x9
    : %h, x10: %h, x11: %h, x12: %h, x13: %h, x14: %h, x15: %h", X_0, X_1, X_2, X_3, X_4,
    X_5, X_6, X_7, X_8, X_9, X_10, X_11, X_12, X_13, X_14, X_15);
        #1
        sel3 = 0;
        sel2 = 1;
        sel1 = 0;
        sel0 = 1;
        $display("input: %H, s3: %b, s2: %b, s1: %b, s0: %b", Y, sel3, sel2, sel1, sel0);
        #1
        $display("x0: %h, x1: %h, x2: %h, x3: %h, x4: %h, x5: %h, x6: %h, x7: %h, x8: %h, x9
    : %h, x10: %h, x11: %h, x12: %h, x13: %h, x14: %h, x15: %h", X_0, X_1, X_2, X_3, X_4,
    X_5, X_6, X_7, X_8, X_9, X_10, X_11, X_12, X_13, X_14, X_15);
    end
    initial begin

        $dumpfile("demux1_16_16b.vcd");
        $dumpvars;
    end


endmodule
```

5.3 Ejercicio 3

F1

```
module F1 (
    input A, input B, input C, input D, output out
);
    assign out = ~A&~B&~C&~D | ~A&B&~C&D | A&B&C&D | A&~B&C&~D;
endmodule
```

F2

```
module F2 (
    input A, input B, input C, input D, output out
);
    assign out = ~A&C | ~A&~B&D | ~B&C&D;
endmodule
```

F3

```
module F3 (
    input A, input B, input C, input D, output out
);
    assign out = ~C&A | A&B&~D | B&~C&~D;
endmodule
```

Comparador

```
`include "F1.v"
`include "F2.v"
`include "F3.v"
module comparator (
    input A, input B, input C, input D, output F1, output F2, output F3
);

    F3 salidaF3(.A(A), .B(B), .C(C), .D(D), .out(F3));
    F2 salidaF2(.A(A), .B(B), .C(C), .D(D), .out(F2));
    F1 salidaF1(.A(A), .B(B), .C(C), .D(D), .out(F1));

endmodule
```

TestBench del comparador

```
`timescale 1ns/1ns
module comparator_tb;

    reg A, B, C, D;
    wire oF1, oF2, oF3;

    comparator compPrueba(.A(A), .B(B), .C(D), .D(D), .F1(oF1), .F2(oF2), .F3(oF3));

    initial begin
        A=1;
        B=1;
        C=1;
        D=1;
        $display("AB: %b%b\tCD: %b%b", A, B, C, D);
        #1
        $display("F1(AB = CD): %b \t F2(AB < CD): %b \t F3(AB > CD): %b", oF1, oF2, oF3);
        #1
        A=0;
        B=0;
        C=0;
        D=1;
        $display("AB: %b%b\tCD: %b%b", A, B, C, D);
        #1
        $display("F1(AB = CD): %b \t F2(AB < CD): %b \t F3(AB > CD): %b", oF1, oF2, oF3);
        #1
        A=1;
        B=1;
        C=1;
        D=0;
        $display("AB: %b%b\tCD: %b%b", A, B, C, D);
        #1
        $display("F1(AB = CD): %b \t F2(AB < CD): %b \t F3(AB > CD): %b", oF1, oF2, oF3);
    end

    initial begin
        $dumpfile("comparator.vcd");
        $dumpvars;
    end
endmodule
```

5.4 Ejercicio 4

Dígito W

```
module digito_W (
    input A, input B, input C, input D, output oW
);
    assign oW = B & C & D | A & ~C & ~D;

endmodule
```

Dígito X

```
module digito_X (
    input A, input B, input C, input D, output oX
);
    assign oX = B & ~C | C & (B^D);

endmodule
```

Dígito Y

```
module digito_Y (
```

```
    input A, input B, input C, input D, output oY
);
    assign oY  = ~A & ~C & D | C & ~D;

endmodule
```

Dígito Z

```
module digito_Z (
    input A, input B, input C, input D, output oZ
);
    assign oZ = ~D;

endmodule
```

BCD

```
`include "digito_W.v"
`include "digito_X.v"
`include "digito_Y.v"
`include "digito_Z.v"

module BCD (
    input in_A, input in_B, input in_C, input in_D, output out_W, output out_X, output out_Y
    , output out_Z
);
    digito_W dW(.oW(out_W), .A(in_A), .B(in_B), .C(in_C), .D(in_D));
    digito_X dX(.oX(out_X), .A(in_A), .B(in_B), .C(in_C), .D(in_D));
    digito_Y dY(.oY(out_Y), .A(in_A), .B(in_B), .C(in_C), .D(in_D));
    digito_Z dZ(.oZ(out_Z), .A(in_A), .B(in_B), .C(in_C), .D(in_D)); //Podriamos utilizar
    solamente: not dZ(out_Z, D);

endmodule
```

TestBench del BCD

```
`timescale 1ns/1ns
module BCD_tb;
    reg A, B, C, D;
    wire oW, oX, oY, oZ;

    BCD testBCD(.in_A(A), .in_B(B), .in_C(C), .in_D(D), .out_W(oW), .out_X(oX), .out_Y(oY),
    .out_Z(oZ));
    initial begin
        A=0;
        B=0;
        C=0;
        D=0;
        $display("ABCD: %b%b%b%b", A, B, C, D);
        #1
        $display("ABCD: %b%b%b%b -> WXYZ: %b%b%b%b", A, B, C, D, oW, oX, oY, oZ);
        #1
        A=0;
        B=0;
        C=0;
        D=1;
        $display("ABCD: %b%b%b%b", A, B, C, D);
        #1
        $display("ABCD: %b%b%b%b -> WXYZ: %b%b%b%b", A, B, C, D, oW, oX, oY, oZ);
        #1
        A=0;
        B=0;
        C=1;
        D=0;
        $display("ABCD: %b%b%b%b", A, B, C, D);
        #1
        $display("ABCD: %b%b%b%b -> WXYZ: %b%b%b%b", A, B, C, D, oW, oX, oY, oZ);
```

```
        #1
        A=0;
        B=0;
        C=1;
        D=1;
        $display("ABCD: %b%b%b%b", A, B, C, D);
        #1
        $display("ABCD: %b%b%b%b -> WXYZ: %b%b%b%b", A, B, C, D, oW, oX, oY, oZ);
        #1
        A=0;
        B=1;
        C=0;
        D=0;
        $display("ABCD: %b%b%b%b", A, B, C, D);
        #1
        $display("ABCD: %b%b%b%b -> WXYZ: %b%b%b%b", A, B, C, D, oW, oX, oY, oZ);
        #1
        A=0;
        B=1;
        C=0;
        D=1;
        $display("ABCD: %b%b%b%b", A, B, C, D);
        #1
        $display("ABCD: %b%b%b%b -> WXYZ: %b%b%b%b", A, B, C, D, oW, oX, oY, oZ);
        #1
        A=0;
        B=1;
        C=1;
        D=0;
        $display("ABCD: %b%b%b%b", A, B, C, D);
        #1
        $display("ABCD: %b%b%b%b -> WXYZ: %b%b%b%b", A, B, C, D, oW, oX, oY, oZ);
        #1
        A=0;
        B=1;
        C=1;
        D=1;
        $display("ABCD: %b%b%b%b", A, B, C, D);
        #1
        $display("ABCD: %b%b%b%b -> WXYZ: %b%b%b%b", A, B, C, D, oW, oX, oY, oZ);
        #1
        A=1;
        B=0;
        C=0;
        D=0;
        $display("ABCD: %b%b%b%b", A, B, C, D);
        #1
        $display("ABCD: %b%b%b%b -> WXYZ: %b%b%b%b", A, B, C, D, oW, oX, oY, oZ);
        #1
        A=1;
        B=0;
        C=0;
        D=1;
        $display("ABCD: %b%b%b%b", A, B, C, D);
        #1
        $display("ABCD: %b%b%b%b -> WXYZ: %b%b%b%b", A, B, C, D, oW, oX, oY, oZ);
    end
    initial begin
        $dumpfile("BCD.vcd");
        $dumpvars;
    end
endmodule
```