Lab 02
Arquitectura de Computadores
Sección 2

Joaquín Ramírez

Mayo 04, 2020

Nota: Todos los *time scales* de los *test benches* son $\frac{1ns}{1ns}$. Además, en cada *test bench* se genera un archivo *.vcd*, el cual será ejecutado en *GTKWave*.

1. Será implementado en el ejercicio 2.a
2. MUX
   (a) Implementación de un 2-to-1 MUX de 16-bits. La implementación es igual que un MUX 2x1 de 1 bit. En el *.v*, el *output* está en función del *select*. Si éste último es 0, entonces se le asignará a y el valor de a. En el otro caso, cuando $s = 1$, $y = b$.

```
module mux16_2x1(a, b, s, y);
        input   [15:0] a;
        input   [15:0] b;
        input   s;

        output [15:0]y;

        assign y = s? b : a;
endmodule
```

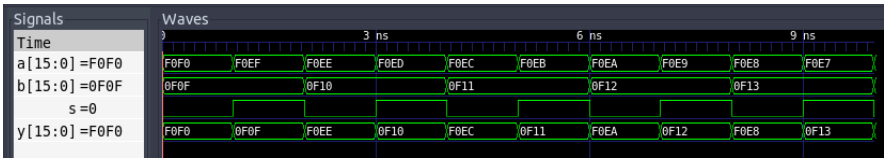
   El *test bench* analiza dies tiempos diferentes. El *select* varía cada dos tiempos, para que dentro de ese intervalo se pueda observar cómo en cada tiempo el *output* cambia entre a y b. Los valores iniciales son: $a = 1111000011110000$ y $b = 0000111100001111$. Cada tiempo, a disminuye en 1, y cada dos tiempos b se incrementa en 1.

```
`timescale 1ns/1ns
module mux16_2x1_tb;
        reg s;
        reg [15:0] a,b;
        wire  [15:0]y;
        mux16_2x1 g (a,b,s,y);
        initial begin
        $display("time\t\ta\t\t\tb\t\ts\t\ty");
                a = 16'b1111000011110000;
                b = 16'b0000111100001111;
                s = 16'b00;
                #10 $finish;
        end
        initial begin
                $monitor("%2d:\t%b\t%b\t%b\t%b",$time,a,b,s,y);
        end
        always #1 a = a - 1;
        always #1 s = !s;
        always #2 b = b + 1;

        initial begin
                $dumpfile("mux16_2x1.vcd");
                $dumpvars;
        end
endmodule
```

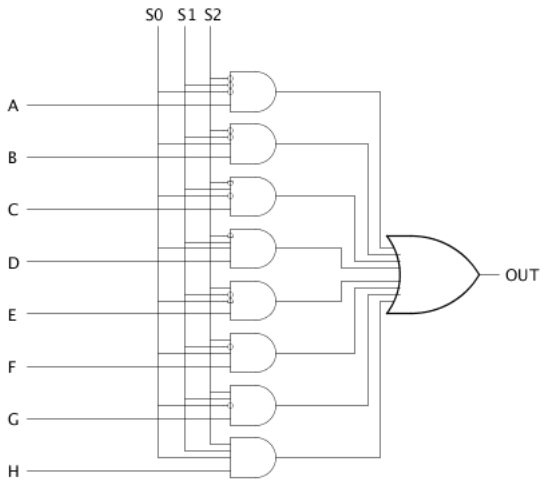
La tabla de verdad obedece el comportamiento planteado.

```
time            a                       b               s               y
VCD info: dumpfile mux16_2x1.vcd opened for output.
 0:     1111000011110000        0000111100001111        0       1111000011110000
 1:     1111000011101111        0000111100001111        1       0000111100001111
 2:     1111000011101110        0000111100010000        0       1111000011101110
 3:     1111000011101101        0000111100010000        1       0000111100010000
 4:     1111000011101100        0000111100010001        0       1111000011101100
 5:     1111000011101011        0000111100010001        1       0000111100010001
 6:     1111000011101010        0000111100010010        0       1111000011101010
 7:     1111000011101001        0000111100010010        1       0000111100010010
 8:     1111000011101000        0000111100010011        0       1111000011101000
 9:     1111000011100111        0000111100010011        1       0000111100010011
10:     1111000011100110        0000111100010100        0       1111000011100110
```

A través de los *waveforms* queda más claro el cambio de valores de y, entre a y b.



(b) Implementación de un 8-to-1 MUX de 16-bits.Se crearon *small modules* que fueron integrados en un *top*. Para la construcción del sistema se partió del siguiente sistema.



El módulo NOT se encarga de negar a cada *select*, para que se puedan formar todas las combinaciones. El módulo niega un input de 1 bit.

```
module mux16_8x1_not(f,h);
        output f;
        input h;
assign f = ~h;
endmodule
```

Ya que hay 3 *selects*, entonces habrán $2^3 = 8$ posibles combinaciones. Es así que hay una compuerta AND para cada una, la cual se activará únicamente con un determinado input de *selects*. El módulo AND valida que la combinación de *selects* sea "high". De ser el caso, el *wireoutput* será igual al valor del bus de 16 bits. Si no, será un cero lógico de 16 bits.

```
module mux16_8x1_and(a, b, c, d, e);
        output [15:0]a;
        input [15:0]b;
        input c,d,e;
        wire validacion;
        assign validacion = (c & d & e);
assign a = validacion? b: 16'b00;
endmodule
```

Todos los *wire outputs* previos se juntan en un único or de 8*inputs*. El *output* final tomará el valor del *wire* cuya combinación de *selects* se "activó".

```
module mux16_8x1_or(l, m, n, o, p, q, r, s, t);
        output [15:0]l;
        input [15:0]m, n, o, p, q, r, s, t;
assign l = m | n | o | p | q | r | s | t;
endmodule
```

Se juntan los módulos previos en el *.v*, el cual está basado en el diagrama de MUX 8x1 anterior.

```
module mux16_8x1(D0, D1, D2, D3, D4, D5, D6, D7, S0, S1, S2, out);
output [15:0]out;
input [15:0] D0,D1,D2, D3,D4,D5,D6,D7;
input S0,S1,S2;
input NS0, NS1, NS2;
wire [15:0]T1, T2, T3, T4, T5, T6, T7, T8;

mux16_8x1_not a1(NS0, S0);
mux16_8x1_not a2(NS1, S1);
mux16_8x1_not a3(NS2, S2);

mux16_8x1_and u1(T1, D0, NS0, NS1, NS2);
mux16_8x1_and u2(T2, D1, S0, NS1, NS2);
mux16_8x1_and u3(T3, D2, NS0, S1, NS2);
mux16_8x1_and u4(T4, D3, S0, S1, NS2);
mux16_8x1_and u5(T5, D4, NS0, NS1, S2);
mux16_8x1_and u6(T6, D5, S0, NS1, S2);
mux16_8x1_and u7(T7, D6, NS0, S1, S2);
mux16_8x1_and u8(T8, D7, S0, S1, S2);
mux16_8x1_or u12(out, T1, T2, T3, T4, T5, T6, T7, T8);
endmodule
```

En el *test bench* se asignan valores diferentes a cada uno de los 8 $16-bit inputs$. Dentro de un intervalo de 8 tiempos se permutan los *selects*, para ver que el *output* es igual a un diferente input en cada ocasión.

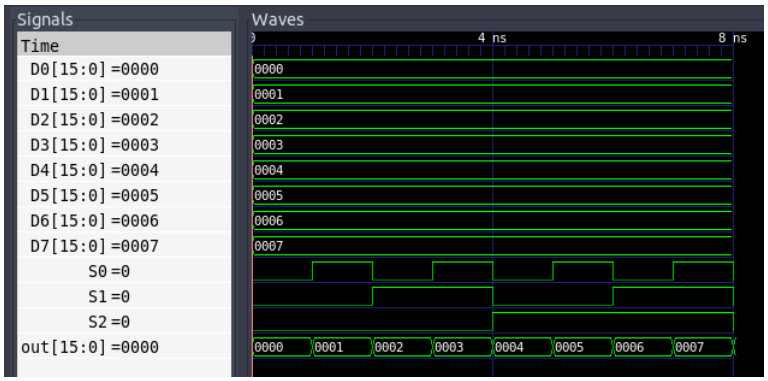
```
`timescale 1ns/1ns
module mux16_8x1_tb;
    wire[15:0]out;
        reg  [15:0]D0, D1, D2, D3, D4, D5, D6, D7;
        reg S0, S1, S2;

    mux16_8x1 g (D0, D1, D2, D3, D4, D5, D6, D7, S0, S1, S2, out);
    initial begin
                D0 = 16'b000;
                D1 = 16'b001;
                D2 = 16'b010;
                D3 = 16'b011;
                D4 = 16'b100;
                D5 = 16'b101;
                D6 = 16'b110;
                D7 = 16'b111;
                S0 = 0;
                S1 = 0;
                S2 = 0;

$display("time\t\tD0\t\t\tD1\t\t\tD2\t\t\tD3\t\t\tD4\t\t\tD5\t\t\tD6\t\t\tD7\t\tS2\tS1\tS0\t\tY");
                $monitor("%2d:\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b",$time,D0, D1, D2, D3, D4,
D5, D6, D7, S2, S1, S0, out);
                    #8 $finish;
    end
always #1 S0 = !S0;
always #2 S1 = !S1;
always #4 S2 = !S2;
    initial begin
        $dumpfile("mux16_8x1.vcd");
        $dumpvars;
    end
endmodule
```

La tabla de verdad confirma lo descrito, cada tiempo el *output* corresponde a un único *input*.

```
time            D0                  D1                  D2                  D3                  D4
VCD info: dumpfile mux16_8x1.vcd opened for output.
 0:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 1:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 2:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 3:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 4:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 5:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 6:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 7:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 8:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
```

```
        D5                  D6                  D7                  S2      S1      S0      Y

0000000000000101    0000000000000110    0000000000000111    0       0       0       0000000000000000
0000000000000101    0000000000000110    0000000000000111    0       0       1       0000000000000001
0000000000000101    0000000000000110    0000000000000111    0       1       0       0000000000000010
0000000000000101    0000000000000110    0000000000000111    0       1       1       0000000000000011
0000000000000101    0000000000000110    0000000000000111    1       0       0       0000000000000100
0000000000000101    0000000000000110    0000000000000111    1       0       1       0000000000000101
0000000000000101    0000000000000110    0000000000000111    1       1       0       0000000000000110
0000000000000101    0000000000000110    0000000000000111    1       1       1       0000000000000111
0000000000000101    0000000000000110    0000000000000111    0       0       0       0000000000000000
```

Los *waveforms* permiten ver con mayor claridad la asignación al *output* de un diferente *input* en cada tiempo.



(c) Implementación de un 16-to-1 MUX de 16-bits. La elaboración de este sistema sigue la misma idea del diagrama del MUX 8x1, pero varían ciertas cosas.

El módulo NOT continúa devolviendo la negación de 1 bit.

```
module mux16_16x1_not(f,h);
        output f;
        input h;
assign f = ~h;
endmodule
```

Ya que ahora hay 4 *selects*, entonces existirán $2^4 = 16$ posibles combinaciones. De esta manera, el AND tiene que validar 4 *selects* de 1 bit cada uno. Si la combinación es "high", entonces el *wire output* será igual que el input, si no, un 0 de 16 bits.

```
module mux16_16x1_and(a, b, c, d, e, w);
        output [15:0]a;
        input [15:0]b;
        input c,d,e,w;
        wire validacion;
        assign validacion = (c&d&e&w);
assign a = validacion? b : 16'b00;
endmodule
```

En este caso, el OR tiene 16 *wire outputs* anteriores, de los cuales dejará pasar a aquel cuya combinación de *selects* fue "high".

```
module mux16_16x1_or(l, m, n, o, p, q, r, s, t, a, b, c, d, e, f, g, h);
       output [15:0]l;
       input [15:0]m, n, o, p, q, r, s, t, a, b, c, d, e, f, g, h;
assign l = m | n | o | p | q | r | s | t| a | b | c | d | e | f | g | h;
endmodule
```

Se juntan los módulos previos en el *.v (top)*, que sigue la misma idea que el desarrollado para el MUX 8x1, solo que aumentando un *select*.

```
module mux16_16x1(D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, D14, D15, S0, S1, S2,S3,out);
output [15:0]out;
input [15:0] D0,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15;
input S0,S1,S2,S3;
input NS0, NS1, NS2,NS3;
wire [15:0]T1, T2, T3, T4, T5, T6, T7, T8,T9,T10,T11,T12,T13,T14,T15,T16;

mux16_16x1_not a1(NS0, S0);
mux16_16x1_not a2(NS1, S1);
mux16_16x1_not a3(NS2, S2);
mux16_16x1_not a4(NS3, S3);

mux16_16x1_and u1(T1, D0, NS0, NS1, NS2, NS3);
mux16_16x1_and u2(T2, D1, S0, NS1, NS2, NS3);
mux16_16x1_and u3(T3, D2, NS0, S1, NS2, NS3);
mux16_16x1_and u4(T4, D3, S0, S1, NS2, NS3);
mux16_16x1_and u5(T5, D4, NS0, NS1, S2, NS3);
mux16_16x1_and u6(T6, D5, S0, NS1, S2, NS3);
mux16_16x1_and u7(T7, D6, NS0, S1, S2, NS3);
mux16_16x1_and u8(T8, D7, S0, S1, S2, NS3);
mux16_16x1_and u9(T9, D8, NS0, NS1, NS2, S3);
mux16_16x1_and u10(T10, D9, S0, NS1, NS2, S3);
mux16_16x1_and u11(T11, D10, NS0, S1, NS2, S3);
mux16_16x1_and u12(T12, D11, S0, S1, NS2, S3);
mux16_16x1_and u13(T13, D12, NS0, NS1, S2, S3);
mux16_16x1_and u14(T14, D13, S0, NS1, S2, S3);
mux16_16x1_and u15(T15, D14, NS0, S1, S2, S3);
mux16_16x1_and u16(T16, D15, S0, S1, S2, S3);
mux16_16x1_or u17(out, T1, T2, T3, T4, T5, T6, T7, T8,T9,T10,T11,T12,T13,T14,T15,T16);
endmodule
```

En el *test bench* se asignan valores únicos para cada input, y durante 16 tiempos, se generan todas las combinaciones de *selects*.

```
 timescale 1ns/1ns
module mux16_16x1_tb;
    wire[15:0]out;
        reg  [15:0]D0, D1, D2, D3, D4, D5, D6, D7,D8, D9, D10, D11, D12, D13, D14, D15;
reg S0, S1, S2, S3;

    mux16_16x1 g (D0, D1, D2, D3, D4, D5, D6, D7,D8, D9, D10, D11, D12, D13, D14, D15,S0, S1, S2,
S3,out);
    initial begin
                D0 = 16'b0000;
                D1 = 16'b0001;
                D2 = 16'b0010;
                D3 = 16'b0011;
                D4 = 16'b0100;
                D5 = 16'b0101;
                D6 = 16'b0110;
                D7 = 16'b0111;
                D8 = 16'b1000;
                D9 = 16'b1001;
                D10 = 16'b1010;
                D11 = 16'b1011;
                D12 = 16'b1100;
                D13 = 16'b1101;
                D14 = 16'b1110;
                D15 = 16'b1111;
                S0 = 0;
                S1 = 0;
                S2 = 0;
                S3 = 0;

$display("time\t\tD0\t\t\tD1\t\t\tD2\t\t\tD3\t\t\tD4\t\t\tD5\t\t\tD6\t\t\tD7\t\t\tD8\t\t\tD9\t\t\t
D10\t\t\tD11\t\t\tD12\t\t\tD13\t\t\tD14\t\t\tD15\t\t\tS3\tS2\tS1\tS0\t\tY");
$monitor("%2d:\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b",
$time,D0, D1, D2, D3, D4, D5, D6, D7,D8, D9, D10, D11, D12, D13, D14, D15,S3, S2, S1, S0,out);
#16 $finish;
    end
always #1 S0 = !S0;
always #2 S1 = !S1;
always #4 S2 = !S2;
always #8 S3 = !S3;
    initial begin
        $dumpfile("mux16_16x1.vcd");
        $dumpvars;
    end
endmodule
```

La tabla de verdad corrobora lo planteado: el *output* Y varía en valores conforme cambian los *selects*.

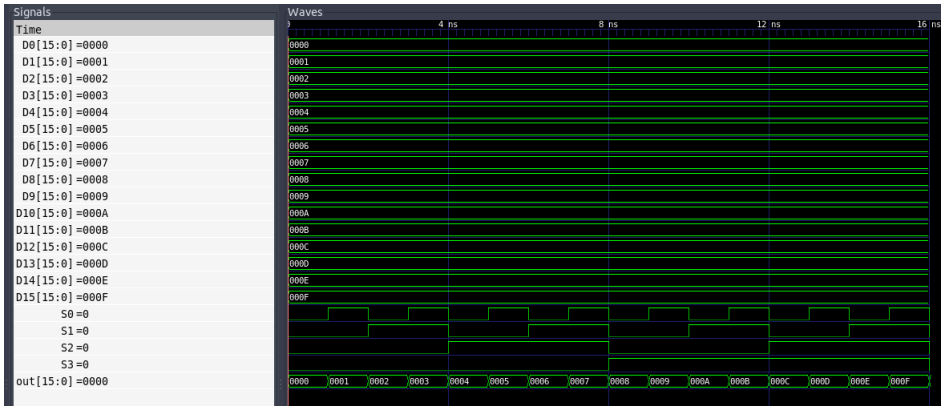
```
time            D0                  D1                  D2                  D3                  D4
VCD info: dumpfile mux16_16x1.vcd opened for output.
 0:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 1:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 2:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 3:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 4:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 5:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 6:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 7:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 8:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
 9:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
10:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
11:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
12:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
13:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
14:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
15:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
16:     0000000000000000    0000000000000001    0000000000000010    0000000000000011    0000000000000100
```

```
      D5                D6                D7                D8                D9

 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
 0000000000000101  0000000000000110  0000000000000111  0000000000001000  0000000000001001
```

```
      D10               D11               D12               D13               D14

 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
 0000000000001010  0000000000001011  0000000000001100  0000000000001101  0000000000001110
```

```
 S3    S2    S1    S0         Y

 0     0     0     0     0000000000000000
 0     0     0     1     0000000000000001
 0     0     1     0     0000000000000010
 0     0     1     1     0000000000000011
 0     1     0     0     0000000000000100
 0     1     0     1     0000000000000101
 0     1     1     0     0000000000000110
 0     1     1     1     0000000000000111
 1     0     0     0     0000000000001000
 1     0     0     1     0000000000001001
 1     0     1     0     0000000000001010
 1     0     1     1     0000000000001011
 1     1     0     0     0000000000001100
 1     1     0     1     0000000000001101
 1     1     1     0     0000000000001110
 1     1     1     1     0000000000001111
 0     0     0     0     0000000000000000
```

Los *waveforms* ilustran mejor la variación del *output* Y.



3. DEMUX

   (a) Implementación del DEMUX 1-to-2 de 16-bits. Se elaboró todo en el $.v$, pues no tenía mucha complejidad. A cada *output* se le asigna el valor del input si y solo si un determinado select está presente. A $y0$ se le asigna $D0$ si $s = 0$, y a $y1$ se le asigna $D0$ si $s = 1$. Cuando el *select* no le corresponde al *output*, se le asigna un 0 de 16 bits.

```
module demux16_1x2 (D0,s,y0,y1);
        input [15:0]D0  ;
        input s;
        output [15:0]y0,y1;

        assign y0 = (s == 0)? D0 : 16'b00;
        assign y1 = (s == 1)? D0 : 16'b00;
endmodule
```

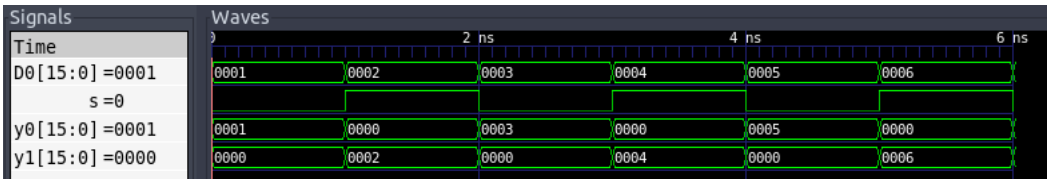
   En el *test bench* se asignan ún valor inicial $D0 = 16'b01$. Cada tiempo, por 6 tiempos, D0 aumentará en 1, y el *select* se negará, para ver que los outputs toman distintos valores dependiendo del valor del *select*.

```
`timescale 1ns/1ns
module demux16_1x2_tv;
        reg s;
        reg [15:0]D0;
        wire [15:0]y0,y1;
        demux16_1x2 g(D0,s,y0,y1);
        initial begin
        $display("time\t\tD0\t\ts\t\ty0\t\t\ty1");
                D0 = 16'b01;
                s = 0;
                #6 $finish;
        end
        initial begin
                $monitor("%2d:\t%b\t%b\t%b\t%b",$time,D0,s,y0,y1);
        end
        always #1 s = !s;
        always #1 D0 = D0 + 1;

        initial begin
                $dumpfile("demux16_1x2.vcd");
                $dumpvars;
        end
endmodule
```

La tabla de verdad confirma lo planteado.

```
time            D0              s               y0                      y1
VCD info: dumpfile demux16_1x2.vcd opened for output.
 0:     0000000000000001        0       0000000000000001        0000000000000000
 1:     0000000000000010        1       0000000000000000        0000000000000010
 2:     0000000000000011        0       0000000000000011        0000000000000000
 3:     0000000000000100        1       0000000000000000        0000000000000100
 4:     0000000000000101        0       0000000000000101        0000000000000000
 5:     0000000000000110        1       0000000000000000        0000000000000110
 6:     0000000000000111        0       0000000000000111        0000000000000000
```

A través de los *waveforms* se puede ver cómo el valor de $D0$ se le asigna a un determinado output según el valor de s.



(b) Implementación del DEMUX 1-to-8 de 16-bits. De acuerdo a la misma idea anterior, si una determinada combinación de *selects* se presenta para determinado output, entonces a éste se le asigna el valor de D0. De lo contrario, se le asigna un 0 de 16 bits. En este caso, el *select* es un bus de 3 bits, con el que se pueden generar $2^3 = 8$ posibles combinaciones.

```
module demux16_1x8(D0, s, y0, y1, y2, y3, y4, y5, y6, y7);
        input [15:0] D0;
        input [2:0]s;
        output [15:0]  y0, y1, y2, y3, y4, y5, y6, y7;
        assign y0 = (s == 3'b000) ? D0 : 16'b0;
        assign y1 = (s == 3'b001) ? D0 : 16'b0;
        assign y2 = (s == 3'b010) ? D0 : 16'b0;
        assign y3 = (s == 3'b011) ? D0 : 16'b0;
        assign y4 = (s == 3'b100) ? D0 : 16'b0;
        assign y5 = (s == 3'b101) ? D0 : 16'b0;
        assign y6 = (s == 3'b110) ? D0 : 16'b0;
        assign y7 = (s == 3'b111) ? D0 : 16'b0;
endmodule
```

En el *test bench* se asignan ún valor inicial $D0 = 16'b01$. Se analizan 8 tiempos, dentro de los cuales cada se generarán todas las combinaciones de 3 bits del *select*. Esto permitirá ver que en cada tiempo, solo un output tendrá el valor de $D0$.

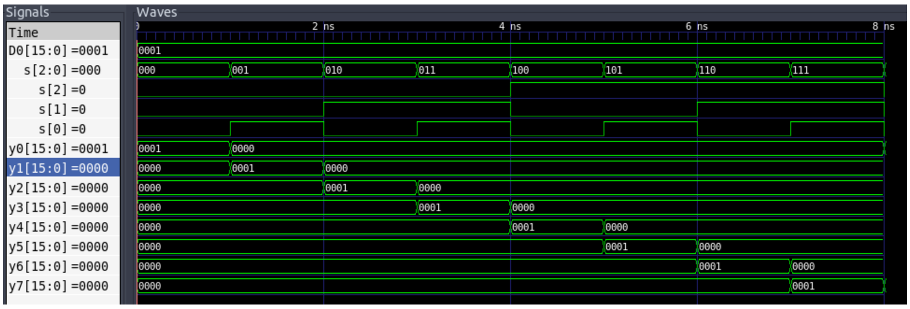
```
`timescale 1ns/1ns
module demux16_1x8_tb;
        reg [15:0]D0;
        reg [2:0]s;
        wire [15:0]  y0, y1, y2, y3, y4, y5, y6, y7;
        demux16_1x8 g(D0, s, y0, y1, y2, y3, y4, y5, y6, y7);
        initial begin
        $display("time\t\tD0\t\ts\t\ty0\t\t\t\ty1\t\t\t\ty2\t\t\ty3\t\t\ty4\t\t\ty5\t\t\ty6\t\t\ty7");
                D0 = 16'b01;
                s = 3'b000;
                #8 $finish;
        end
        initial begin
                $monitor("%2d:\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b",$time,D0, s, y0, y1, y2, y3, y4, y5, y6, y7);
        end
        always #1 s[0] = !s[0];
        always #2 s[1] = !s[1];
        always #4 s[2] = !s[2];
        initial begin
                $dumpfile("demux16_1x8.vcd");
                $dumpvars;
        end
endmodule
```

La tabla de verdad corrobora lo elaborado.

```
time            D0              s               y0                      y1                      y2
VCD info: dumpfile demux16_1x8.vcd opened for output.
 0:     0000000000000001        000     0000000000000001        0000000000000000        0000000000000000
 1:     0000000000000001        001     0000000000000000        0000000000000001        0000000000000000
 2:     0000000000000001        010     0000000000000000        0000000000000000        0000000000000001
 3:     0000000000000001        011     0000000000000000        0000000000000000        0000000000000000
 4:     0000000000000001        100     0000000000000000        0000000000000000        0000000000000000
 5:     0000000000000001        101     0000000000000000        0000000000000000        0000000000000000
 6:     0000000000000001        110     0000000000000000        0000000000000000        0000000000000000
 7:     0000000000000001        111     0000000000000000        0000000000000000        0000000000000000
 8:     0000000000000001        000     0000000000000001        0000000000000000        0000000000000000
```

```
        y3                      y4                      y5                      y6                      y7

0000000000000000        0000000000000000        0000000000000000        0000000000000000        0000000000000000
0000000000000000        0000000000000000        0000000000000000        0000000000000000        0000000000000000
0000000000000000        0000000000000000        0000000000000000        0000000000000000        0000000000000000
0000000000000001        0000000000000000        0000000000000000        0000000000000000        0000000000000000
0000000000000000        0000000000000001        0000000000000000        0000000000000000        0000000000000000
0000000000000000        0000000000000000        0000000000000001        0000000000000000        0000000000000000
0000000000000000        0000000000000000        0000000000000000        0000000000000001        0000000000000000
0000000000000000        0000000000000000        0000000000000000        0000000000000000        0000000000000001
0000000000000000        0000000000000000        0000000000000000        0000000000000000        0000000000000000
```

Con los *waveforms* se puede observar que en cada tiempo solo un *output* toma el valor de $D0$.



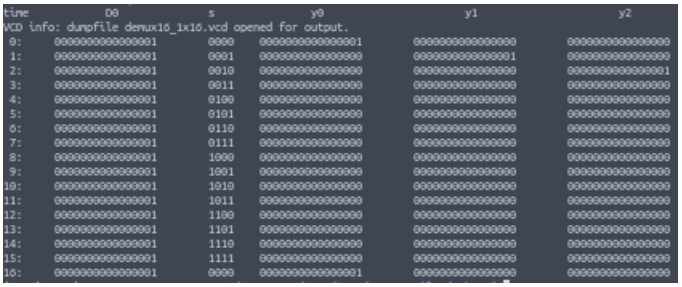
(c) Implementación del DEMUX 1-to-16 de 16-bits. Conforme a la misma idea de los dos ejercicios anteriores, si una determinada combinación de *selects* se presenta para determinado output, entonces a éste se le asigna el valor de D0. De lo contrario, se le asigna un 0 de 16 bits. En este caso, el *select* es un bus de 4 bits, con el que se pueden generar $2^4 = 16$ posibles combinaciones.

```
module demux16_1x16(D0, s, y0, y1, y2, y3, y4, y5, y6, y7,y8,y9,y10,y11,y12,y13,y14,y15);
        input [15:0] D0;
        input [3:0]s;
        output [15:0]  y0, y1, y2, y3, y4, y5, y6, y7,y8,y9,y10,y11,y12,y13,y14,y15;
        assign y0 = (s == 4'b0000) ? D0 : 16'b0;
        assign y1 = (s == 4'b0001) ? D0 : 16'b0;
        assign y2 = (s == 4'b0010) ? D0 : 16'b0;
        assign y3 = (s == 4'b0011) ? D0 : 16'b0;
        assign y4 = (s == 4'b0100) ? D0 : 16'b0;
        assign y5 = (s == 4'b0101) ? D0 : 16'b0;
        assign y6 = (s == 4'b0110) ? D0 : 16'b0;
        assign y7 = (s == 4'b0111) ? D0 : 16'b0;
        assign y8 = (s == 4'b1000) ? D0 : 16'b0;
        assign y9 = (s == 4'b1001) ? D0 : 16'b0;
        assign y10 = (s == 4'b1010) ? D0 : 16'b0;
        assign y11 = (s == 4'b1011) ? D0 : 16'b0;
        assign y12 = (s == 4'b1100) ? D0 : 16'b0;
        assign y13 = (s == 4'b1101) ? D0 : 16'b0;
        assign y14 = (s == 4'b1110) ? D0 : 16'b0;
        assign y15 = (s == 4'b1111) ? D0 : 16'b0;
endmodule
```

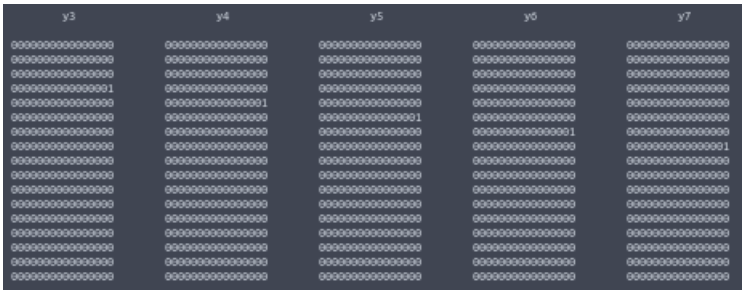
En el *test bench* se asignan ún valor inicial $D0 = 16'b01$. Se analizan 16 tiempos, dentro de los cuales cada se generarán todas las combinaciones de 4 bits del *select*. Esto permitirá ver que en cada tiempo, solo un output tendrá el valor de $D0$.

```
`timescale 1ns/1ns
module demux16_1x16_tb;
        reg [15:0]D0;
        reg [3:0]s;
        wire [15:0]  y0, y1, y2, y3, y4, y5, y6, y7,y8,y9,y10,y11,y12,y13,y14,y15;
        demux16_1x16 g(D0, s, y0, y1, y2, y3, y4, y5, y6, y7,y8,y9,y10,y11,y12,y13,y14,y15);
        initial begin
        $display("time\t\tD0\t\ts\t\ty0\t\t\ty1\t\t\ty2\t\t\ty3\t\t\ty4\t\t\ty5\t\t\ty6\t\t\ty7\t\t\ty8\t\t\ty9\t\t\ty10\t\t\ty11\t\t\ty12\t\t\ty13\t\t\ty14\t\t\ty15");
                D0 = 16'b01;
                s = 4'b0000;
                #16 $finish;
        end
        initial begin
                $monitor("%2d:\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b",$time,D0, s, y0, y1, y2, y3, y4, y5, y6, y7,y8,y9,y10,y11,y12,y13,y14,y15);
        end
        always #1 s[0] = !s[0];
        always #2 s[1] = !s[1];
        always #4 s[2] = !s[2];
        always #8 s[3] = !s[3];

        initial begin
                $dumpfile("demux16_1x16.vcd");
                $dumpvars;
        end
endmodule
```

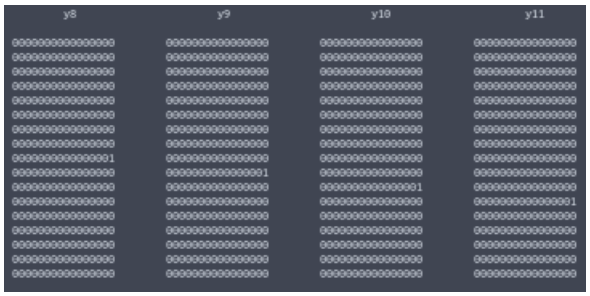
La tabla de verdad confirma el planteamiento anterior.



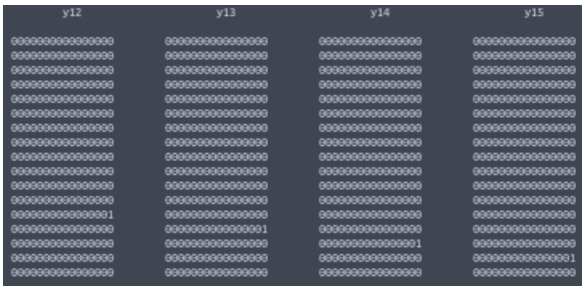
```
time          D0                s           y0                  y1                  y2
VCD info: dumpfile demux16_1x16.vcd opened for output.
 0:     0000000000000001     0000   0000000000000001    0000000000000000    0000000000000000
 1:     0000000000000001     0001   0000000000000000    0000000000000001    0000000000000000
 2:     0000000000000001     0010   0000000000000000    0000000000000000    0000000000000001
 3:     0000000000000001     0011   0000000000000000    0000000000000000    0000000000000000
 4:     0000000000000001     0100   0000000000000000    0000000000000000    0000000000000000
 5:     0000000000000001     0101   0000000000000000    0000000000000000    0000000000000000
 6:     0000000000000001     0110   0000000000000000    0000000000000000    0000000000000000
 7:     0000000000000001     0111   0000000000000000    0000000000000000    0000000000000000
 8:     0000000000000001     1000   0000000000000000    0000000000000000    0000000000000000
 9:     0000000000000001     1001   0000000000000000    0000000000000000    0000000000000000
10:     0000000000000001     1010   0000000000000000    0000000000000000    0000000000000000
11:     0000000000000001     1011   0000000000000000    0000000000000000    0000000000000000
12:     0000000000000001     1100   0000000000000000    0000000000000000    0000000000000000
13:     0000000000000001     1101   0000000000000000    0000000000000000    0000000000000000
14:     0000000000000001     1110   0000000000000000    0000000000000000    0000000000000000
15:     0000000000000001     1111   0000000000000000    0000000000000000    0000000000000000
16:     0000000000000001     0000   0000000000000001    0000000000000000    0000000000000000
```



```
       y3                  y4                  y5                  y6                  y7

0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000001    0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000001    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000001    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000001    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000001
0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000    0000000000000000
```

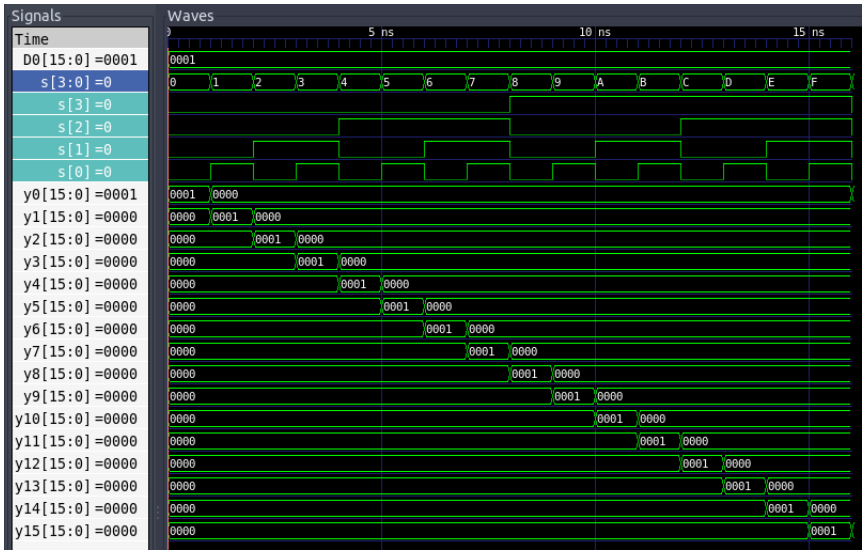


```
       y8                  y9                  y10                 y11

0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000001    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000001    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000001    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000001
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
```



```
       y12                 y13                 y14                 y15

0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000001    0000000000000000    0000000000000000    0000000000000000
0000000000000000    0000000000000001    0000000000000000    0000000000000000
0000000000000000    0000000000000000    0000000000000001    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000001
0000000000000000    0000000000000000    0000000000000000    0000000000000000
```

Los *waveforms* permiten ver el cambio en determinados outputs de una manera más eficiente.



4. Implementación de un decoder 3-to-8 ($m - 2^m$). Se tiene un *input* n de 3 bits, cuyos 8 posibles valores indican qué posición del output de 8 bits será 1. Si se presenta una combinación de particular de n, entonces una posición (7:0) del output será 1, y las demás 0. Se tiene un enable que verifica antes de todo la "autorización" del sistema. Cuando $ena = 1$, entonces proceden las validaciones de n. No obstante, en el caso de que $n = 0$, no es necesario ver el valor de n (se toman como *don't care*), pues automáticamente todas las posiciones del *output* serán 0.

```
module decoder3x8(n,ena,d);
        input [2:0] n;
        input ena;
        output [7:0]d;

        assign d[0] = (ena==1) ?((n ==3'b000) ? 1: 0):0;
        assign d[1] = (ena==1) ?((n ==3'b001) ? 1: 0):0;
        assign d[2] = (ena==1) ?((n ==3'b010) ? 1: 0):0;
        assign d[3] = (ena==1) ?((n ==3'b011) ? 1: 0):0;
        assign d[4] = (ena==1) ?((n ==3'b100) ? 1: 0):0;
        assign d[5] = (ena==1) ?((n ==3'b101) ? 1: 0):0;
        assign d[6] = (ena==1) ?((n ==3'b110) ? 1: 0):0;
        assign d[7] = (ena==1) ?((n ==3'b111) ? 1: 0):0;
endmodule
```

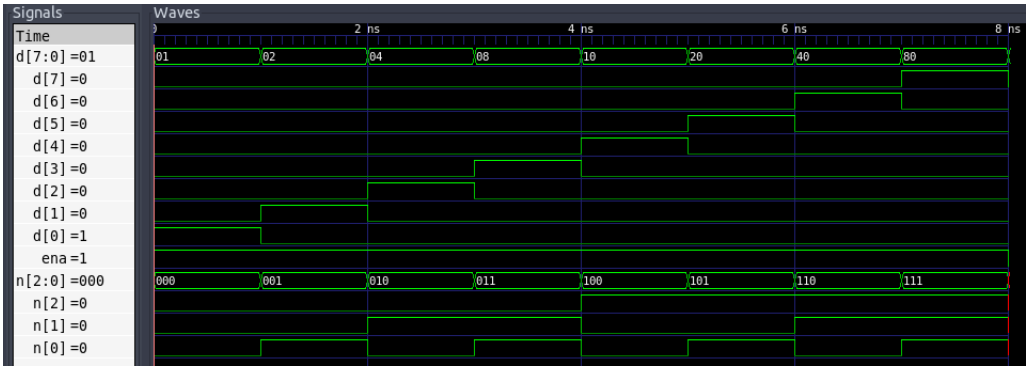
En el *test bench*, el valor inicial de n es 000. Se generarán todas las combinaciones de éste durante 8 tiempos. Esto permitirá analizar que en cada tiempo una posición diferente del *output* se vuelve 1.

```
`timescale 1ns/1ns
module decoder3x8_tb    ;
        reg [2:0] n;
        reg ena;
        wire [7:0]d;
        decoder3x8 g (n,ena,d);
        initial begin
        $display("time\tena\tn[2]\tn[1]\tn[0]\td[7]\td[6]\td[5]\td[4]\td[3]\td[2]\td[1]\td[0]");

                n = 3'b000;
                ena = 1;
                #8 $finish;
        end
        initial begin
$monitor("%2d:\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b",
$time,ena,n[2],n[1],n[0],d[7],d[6],d[5],d[4],d[3],d[2],d[1],d[0]);
        end
        always #1 n[0] = !n[0];
        always #2 n[1] = !n[1];
        always #4 n[2] = !n[2];
        always #8 ena = !ena;
        always #8 n = 3'bXXX;

        initial begin
                $dumpfile("decoder3x8.vcd");
                $dumpvars;
        end
endmodule
```

En la tabla de verdad se puede ver que en cada tiempo un bit diferente de d es 1, y los demás son 0. Asimismo, en el último tiempo analizado, el *enable* $= 0$, por lo que n está lleno de *don't cares*, y todos los bits de d son 0.

```
time    ena     n[2]    n[1]    n[0]    d[7]    d[6]    d[5]    d[4]    d[3]    d[2]    d[1]    d[0]
VCD info: dumpfile decoder3x8.vcd opened for output.
 0:     1       0       0       0       0       0       0       0       0       0       0       1
 1:     1       0       0       1       0       0       0       0       0       0       1       0
 2:     1       0       1       0       0       0       0       0       0       1       0       0
 3:     1       0       1       1       0       0       0       0       1       0       0       0
 4:     1       1       0       0       0       0       0       1       0       0       0       0
 5:     1       1       0       1       0       0       1       0       0       0       0       0
 6:     1       1       1       0       0       1       0       0       0       0       0       0
 7:     1       1       1       1       1       0       0       0       0       0       0       0
 8:     0       x       x       x       0       0       0       0       0       0       0       0
```

Los *waveforms* enclarecen el cambio de 0 a 1 de un determinado bit de d en cada tiempo.



5. Implementación de un*priority encoder* 8-to-3. La implementación de este sistema se hizo con *casex*. Éste es un tipo particular de *case statement*, que permite comparar bits "sin importar" ciertas posiciones. En este caso, compara el input d de 8 bits con 8 diferentes posibilidades, cada posición igual a 1, y las otras igual a 0. El *casex* analiza d (de izquierda a derecha) hasta encontrar un 1. El valor de los bits después del 1 no son de relevancia. Como el *priority encoder* es la operación inversa del ejercicio anterior (decoder), entonces no es necesario analizar lo que viene después del 1 en d, pues ya se sabe que no habrá otro 1. De igual manera como antes, se asigna 000 a n cuando $ena = 0$; de lo contrario, se analizan los bits.

```
module encoder8x3 (d,ena,n);
        input [7:0]d;
        input ena;
        output [2:0]n;
        reg [2:0]n;
        always @(d,ena) begin
        if(ena==1)begin
        casex(d)
        8'b1??????? : n = 3'b111;
        8'b01?????? : n = 3'b110;
        8'b001????? : n = 3'b101;
        8'b0001???? : n = 3'b100;
        8'b00001??? : n = 3'b011;
        8'b000001?? : n = 3'b010;
        8'b0000001? : n = 3'b001;
        8'b00000001 : n = 3'b000;
        default: n = 3'b000;
        endcase
        end
        else begin n = 3'b000;end
end
endmodule
```

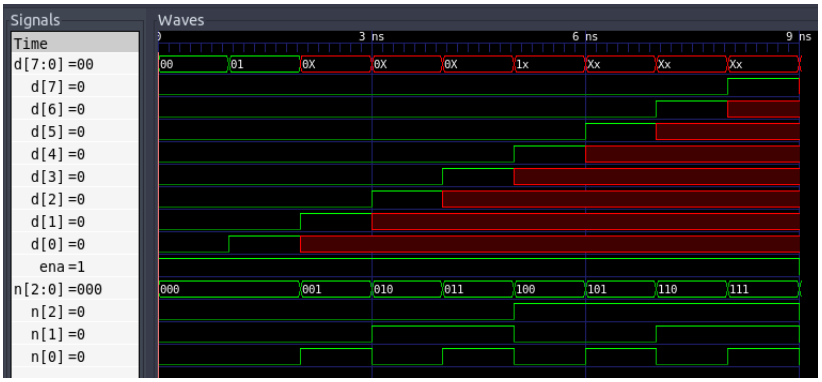
En el *test bench* se asigna como valor inicial $d = 8'b00$ y $ena = 1$. Durante 8 tiempos se cambia el valor de d, para observar el cambio en n. En el noveno tiempo se asigna $ena = 0$.

```
`timescale 1ns/1ns
module encoder8x3_tb;
        reg [7:0] d;
        reg ena;
        wire [2:0]n;
        encoder8x3 g (d,ena,n);
        initial begin
        $display("time\tena\td[7]\td[6]\td[5]\td[4]\td[3]\td[2]\td[1]\td[0]\tn[2]\tn[1]\tn[0]");
                d= 8'b00;
                ena = 1;
                #9 $finish;
        end
        initial begin
                $monitor("%2d:\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b",
$time,ena,d[7],d[6],d[5],d[4],d[3],d[2],d[1],d[0],n[2],n[1],n[0]);
        end
        initial begin
        #1 d = 8'b1;
        #1 d = 8'b1X;
        #1 d = 8'b1XX;
        #1 d = 8'b1XXX;
        #1 d = 8'b1XXXX;
        #1 d = 8'b1XXXXX;
        #1 d = 8'b1XXXXXX;
        #1 d = 8'b1XXXXXXX;
        #8 d = 8'bXXXXXXXX;
        end
        always #9 ena = !ena;
        always #9 d = 8'bXXXXXXXX;
        initial begin
                $dumpfile("encoder8x3.vcd");
                $dumpvars;
        end
endmodule
```

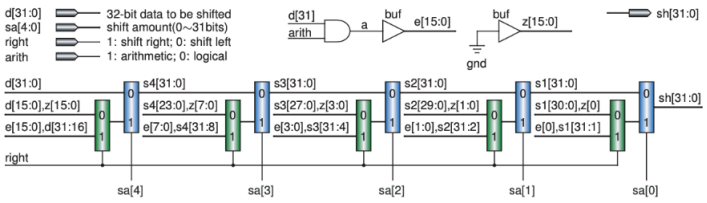
La tabla de verdad verifica el comportamiento planteado del sistema.

```
time    ena     d[7]    d[6]    d[5]    d[4]    d[3]    d[2]    d[1]    d[0]    n[2]    n[1]    n[0]
VCD info: dumpfile encoder8x3.vcd opened for output.
 0:     1       0       0       0       0       0       0       0       0       0       0       0
 1:     1       0       0       0       0       0       0       0       1       0       0       0
 2:     1       0       0       0       0       0       0       1       x       0       0       1
 3:     1       0       0       0       0       0       1       x       x       0       1       0
 4:     1       0       0       0       0       1       x       x       x       0       1       1
 5:     1       0       0       0       1       x       x       x       x       1       0       0
 6:     1       0       0       1       x       x       x       x       x       1       0       1
 7:     1       0       1       x       x       x       x       x       x       1       1       0
 8:     1       1       x       x       x       x       x       x       x       1       1       1
 9:     0       x       x       x       x       x       x       x       x       0       0       0
```

Los *waveforms* ayudan a la visualización del cambio en los valores de cada bit de d, entre 0, 1 y *don't care*. Para cada valor de d, hay un valor de n diferente.



6. Implementación del *barrel shifter* de 32-bits. Ésta operación tiene que ser capaz de mover un *input* d de 32 bits hacia la derecha o izquierda, llenando los espacios vaciós con ceros o con el valor de $d[31]$. Se partió del siguiente diagrama:



Primero se construye un *wire* e, a través del módulo ANDE. Cuando $arith = 1$, entonces e toma el valor de $d[31]$, concatenado 16 veces. En el caso contrario, $e = 16'b00$.

```
module ANDE(d,arith,e);
        input d;
        input arith;
        output [15:0]e;
        wire [15:0]temp;
        assign temp = {16{d}};
        assign e = arith? temp : 16'b00;
endmodule
```

Después se genera el módulo mux32 2x1, que se utilizará 10 veces a lo largo del *shifter*. Si el *select* s es 0, entonces el *wire* y tomará el valor de a. De lo contrario, tomará el valor de b.

```
module mux32_2x1(a, b, s, y);
        input   [31:0] a;
        input   [31:0] b;
        input   s;
        output [31:0]y;

        assign y = s? b : a;
endmodule
```

En el *.v* se "llaman" a los módulos previamente indicados siguiendo el "camino" del diagrama anterior. El *input sa* de 5 bits contiene los *selects* que se usarán a lo largo del *shifter*. Después de que los 10 MUX se "ejecuten", el valor final de la operación es asignado a *sh*.

```
module barrel_shifter(d,sa,right,arith,sh);
        input [31:0]d;
        input [4:0]sa;
        input right, arith;
        output [31:0]sh;
        wire [31:0]green0,green1,green2,green3,green4;
        wire [31:0]s4,s3,s2,s1;
        wire [15:0]e,z;

        assign z = 16'b00;
        ANDE ande(d[31],arith,e);

        mux32_2x1 firstgreenmux({d[15:0],z},{e,d[31:16]},right,green0);
        mux32_2x1 firstbluemux(d,green0,sa[4],s4);
        mux32_2x1 secondgreenmux({s4[23:0],z[7:0]},{e[7:0],s4[31:8]},right,green1);
        mux32_2x1 secondbluemux(s4,green1,sa[3],s3);
        mux32_2x1 thirdgreenmux({s3[27:0],z[3:0]},{e[3:0],s3[31:4]},right,green2);
        mux32_2x1 thirdbluemux(s3,green2,sa[2],s2);
        mux32_2x1 fourthgreenmux({s2[29:0],z[1:0]},{e[1:0],s2[31:2]},right,green3);
        mux32_2x1 fourthbluemux(s2,green3,sa[1],s1);
        mux32_2x1 fifthgreenmux({s1[30:0],z[0]},{e[0],s1[31:1]},right,green4);
        mux32_2x1 fifthbluemux(s1,green4,sa[0],sh);
endmodule
```

En el *test bench* se asigna el valor inicial $d = 11111111000000000000000011111111$. Se desea que se "muevan" los bits en bloques de 8, entonces $sa = 5'b100$. Para poder ver todas las posibilidades de *shifts*, se inicia con $right = 0$ y $arith = 0$, y durante 3 tiempos se realizan todas las combinaciones indicadas en la tarea, para poder ver como cambia sh.

```
`timescale 1ns/1ns
module barrel_shifter_tb;
        reg [31:0]d;
        reg [4:0]sa;
        reg right, arith;
        wire [31:0]sh;
        barrel_shifter ojalacorra(d,sa,right,arith,sh);
        initial begin
        $display("time\t\t\td\t\t\tsa\tright\tarith\t\t\tsh");
        d = 32'b11111111000000000000000011111111;
        sa = 5'b1000;
        right = 0;
        arith = 0;
        #3 $finish;
        end
        initial begin
                $monitor("%2d:\t%b\t%b\t%b\t%b\t%b",$time,d,sa,right,arith,sh);
        end
        initial begin
        #1 right = !right;
        #1 arith = !arith;
        #1 right = !right;arith = !arith;
        end
        initial begin
                $dumpfile("barrel_shifter.vcd");
                $dumpvars;
        end
endmodule
```

A través de la tabla de verdad se pueden observar cómo se dan los movimientos. En el tiempo 1, hay un *right logicalic shift*, porque se mueve a la derecha y se rellenan los espacios vacíos con ceros. En el tiempo 2, hay un *right arithmetic shift*, ya que se mueve a la derecha y se replica $d[31]$ en las posiciones vacías. Finalmente, en el tiempo 3 hay un *left logical shift*, pues se mueve a la izquierda y se llenan los espacios vacíos con ceros.

```
time                    d                       sa      right   arith                   sh
VCD info: dumpfile barrel_shifter.vcd opened for output.
 0:     11111111000000000000000011111111        01000   0       0       00000000000000001111111100000000
 1:     11111111000000000000000011111111        01000   1       0       00000000111111110000000000000000
 2:     11111111000000000000000011111111        01000   1       1       11111111111111110000000000000000
 3:     11111111000000000000000011111111        01000   0       0       00000000000000001111111100000000
```

A través de los *waveforms* se puede apreciar cómo cambian de valor ciertos bits de *sh*. Se ven con claridad los *shifts*.