Informe guía 3

Laboratorio de Electrónica Digital

Autor:
Simón Aulet

Octubre 2025

Universidad Nacional de Río Negro
Ingeniería Electrónica y en Telecomunicaciónes

Índice

1 Ejercicio 1

Diseñar un módulo que solo conecte una entrada a a una salida y.

1.1 Código del Módulo

```
module ej1(
  input a,
  output reg y);

always @(*)
begin
  y <= a;
end
endmodule
```

Listing 1: Módulo ej1.v

1.2 Testbench

```
module ej1_tb();
reg A;
wire Y;
integer i;

ej1 INOUT(.a(A), .y(Y));

initial
begin
  #3 A = 1;
end

initial
begin
  for(i=0; i<20; i=i+1)
  #10 A = ~A;
  $finish;
end
initial
$monitor($time, "A=%d, Y=%d", A, Y);
endmodule
```

Listing 2: Testbench ej1_tb.v

1.3 Waveform

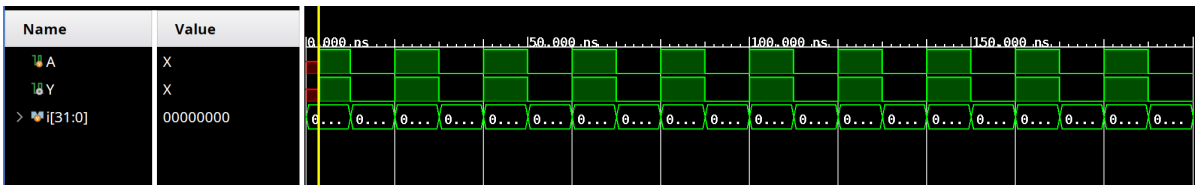

Figura 1: Waveform del ejercicio 1

2 Ejercicio 2

Diseñar una compuerta AND con dos entradas y una salida.

2.1 Código del Módulo

```
module and\_2(
input x1,
input x2,
output q);

assign q = aux; //pruebo para usar wires en la salida
reg aux;        //registro generado después del assign para probar
   la concurrencia
always@(*)
begin
  aux <= x1 && x2;
end

endmodule
```

Listing 3: Módulo and_2

2.2 Testbench

```
`timescale 1ns/100ps
module ej2_tb();

reg X1, X2;
wire Q;

and_2 AND_2_TB(.x1(X1), .x2(X2), .q(Q));

initial
begin
  #2 X1 = 0;
  #3 X2 = 0;
end
```

```
14
15 always
16 begin
17   #10 X1 <= 1'b0;
18   #11 X2 <= 1'b0;
19   #20 X1 <= 1'b1;
20   #30 X2 <= 1'b1;
21   #40 X1 <= 1'b0;
22   #60 X1 <= 1'b0;
23
24   $finish;
25 end
26 initial
27 $monitor($time, "X1=%d, X2=%d, Q=%d", X1, X2, Q);
28
29 endmodule
```

Listing 4: Testbench ej2_tb.v

2.3 Waveform

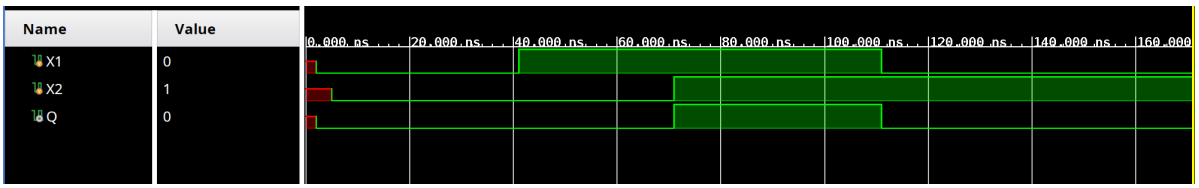

Figura 2: Waveform del ejercicio 2

3 Ejercicio 3

Módulo que implemente una OR de tres entradas.

3.1 Código del Módulo

```
1  module or_gate(
2    input wire  x1,
3    input wire  x2,
4    input wire  x3,
5    output wire q
6  );
7  reg salida;
8  assign q = salida;
9
10 always@(*)
11 begin
12   if(x1 || x2 || x3)
13     salida = 1'b1;
```

```
  else
    salida = 1'b0;
end

endmodule
```

Listing 5: Módulo or_gate

3.2 Testbench

```
`timescale 1ns/100ps
module or_gate_tb();

reg X1, X2, X3;
wire Y;

or_gate DUT(.x1(X1), .x2(X2), .x2(X2), .q(Y));

initial
begin
  X1 <= 1'b0;
  X2 <= 1'b0;
  X3 <= 1'b0;
end
always
begin
    #10 X1 <= 1'b0;
        X2 <= 1'b0;
        X3 <= 1'b0;
    #10 X1 <= 1'b0;
        X2 <= 1'b0;
        X3 <= 1'b1;
    #10 X1 <= 1'b0;
        X2 <= 1'b1;
        X3 <= 1'b0;
    #10 X1 <= 1'b0;
        X2 <= 1'b1;
        X3 <= 1'b1;
    #10 X1 <= 1'b1;
        X2 <= 1'b0;
        X3 <= 1'b0;
    #10 X1 <= 1'b1;
        X2 <= 1'b0;
        X3 <= 1'b1;
    #10 X1 <= 1'b1;
        X2 <= 1'b1;
        X3 <= 1'b0;
    #10 X1 <= 1'b1;
        X2 <= 1'b1;
        X3 <= 1'b1;
    #10
```

```
42        X1 <= 1'b0;
43        X2 <= 1'b0;
44        X3 <= 1'b0;
45      #10
46    $finish;
47  end
48
49  initial
50  $monitor("X1=%d, X2=%d, Y=%d", X1, X2, Y);
51
52  endmodule
```

Listing 6: Testbench or_gate_tb.v

3.3 Waveform

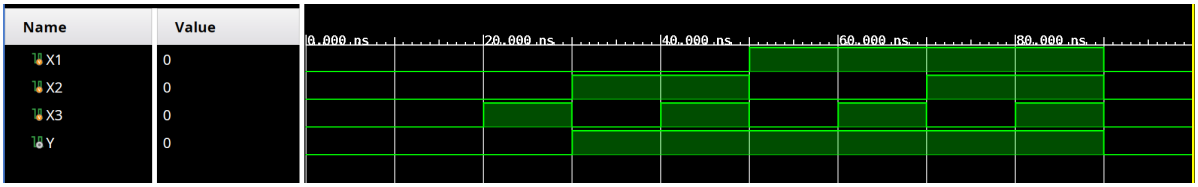

Figura 3: Waveform del ejercicio 3

4 Ejercicio 4

Diseñar un módulo con operación XOR.

4.1 Código del Módulo

```
1  module xor_gate(
2  input wire x1,
3  input wire x2,
4  output wire q
5  );
6
7  reg aux;
8  assign q = aux;
9  always@(*)
10 begin
11   aux = x1 || x2;
12   if(x1&&x2)
13     aux = 1'b0;
14 end
15
16 endmodule
```

Listing 7: Módulo xor_gate

4.2 Testbench

```
`timescale 1ns/100ps

module xor_gate_tb();
reg X1, X2;
wire Q;

xor_gate XOR_GATE_TB(.x1(X1), .x2(X2), .q(Q));

always
begin
  #10 X1 <= 1'b0;//X1=0
  #11 X2 <= 1'b0;//X1=0, X2=0
  #20 X1 <= 1'b1;//X1=1, X2=0
  #30 X2 <= 1'b1;//X1=1, X2=1
  #40 X1 <= 1'b0;//X1=0, X2=1
  #50 X2 <= 1'b0;//X1=0, X2=0
  #60

  $finish;
end

initial
$monitor("X1=%d, X2=%d, Q=%d", X1, X2, Q);

endmodule
```

Listing 8: Testbench xor_gate_tb

4.3 Waveform

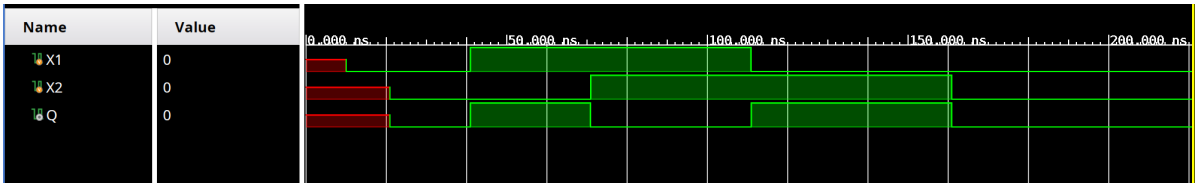

Figura 4: Waveform del ejercicio 4

5 Ejercicio 5

Implementar un MUX de 2 entradas y 1 salida con señal de selección.

5.1 Código del Módulo

```
module mux(
input  wire in1,
```

```
input  wire in2,
input  wire sel,
input  wire clk,
output wire out
);
reg aux;
assign out = aux;

always@(posedge clk)
begin
  if(sel)
    aux = in2;
  else
    aux = in1;
end
endmodule
```

Listing 9: Módulo mux

5.2 Testbench

```
`timescale 1ns/100ps

//verible_lint waive module-filename
module MUX_TB();
reg IN1, IN2, SEL, CLK;
wire OUT;
integer i;

mux DUT(.in1(IN1), .in2(IN2), .sel(SEL), .clk(CLK), .out(OUT));

initial
begin
  #5 CLK = 0;
  for(i=0; i<20; i=i+1)
  #10 CLK = ~CLK;

  $finish;
end

initial
begin
  IN1      = 1'b0;
  IN2      = 1'b1;
  #100 IN1 = 1'b1;
       IN2 = 1'b0;
end

always
begin

```

```
31    SEL = 1'b0;
32    #20 SEL = 1'b1;
33    #20 SEL = 1'b0;
34
35  end
36
37  initial
38  $monitor("IN1=%d, IN2=%d, SEL=%d, aux=%d, OUT=%d", IN1, IN2,
        SEL,DUT.aux, OUT);
39
40  endmodule
```

Listing 10: Testbench MUX_TB

5.3 Waveform

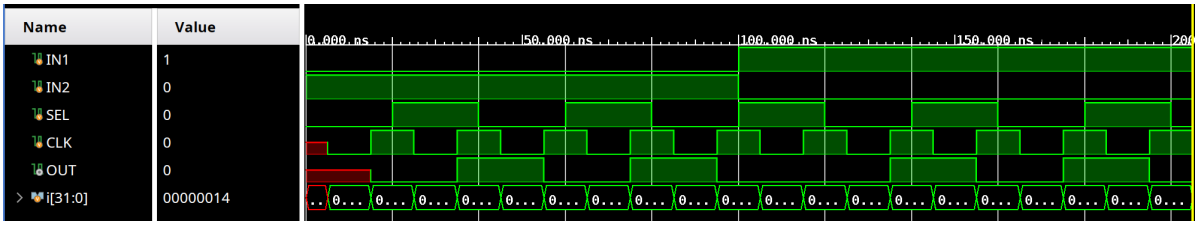

Figura 5: Waveform del ejercicio 5

6 Ejercicio 6

Decodificador 2:4: Con 2 bits de entrada y 4 salidas.

6.1 Código del Módulo

```
1   module decoder2_4(
2     input [1:0] x,
3     output reg [3:0] y
4   );
5
6   initial
7     y = 4'b0000;
8
9   always@(*)
10  begin
11    y    = 4'b0000;
12    y[x] = 1'b1;
13  end
14
15  endmodule
```

Listing 11: Módulo ej6.v

6.2 Testbench

```
`timescale 1ns/100ps

module DECODER2_4_TB();

reg [1:0] X;
wire [3:0] Y;

decoder2_4 DUT(.x(X), .y(Y));

initial
begin
  #10 X = 2'b00;
  #10 X = 2'b01;
  #10 X = 2'b10;
  #10 X = 2'b11;
  #10;
  $finish;
end

endmodule
```

Listing 12: Testbench ej6_tb.v

6.3 Waveform

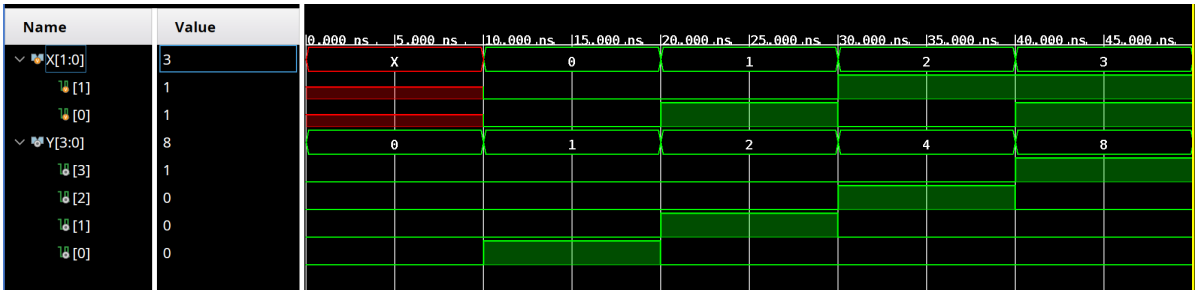

Figura 6: Waveform del ejercicio 6

7 Ejercicio 7

Codificador 8:3: Con 8 entradas y 3 salidas.

7.1 Código del Módulo

```
`timescale 1ns/100ps

module encoder83(
  input wire[7:0] x,
```

```
  output reg[2:0] y
);

always@(*)
casex(x) // Encoder de prioridad, solo toma la salida más alta
   ignorando el resto
  8'b1xxxxxxx: y = 7;
  8'b01xxxxxx: y = 6;
  8'b001xxxxx: y = 5;
  8'b0001xxxx: y = 4;
  8'b00001xxx: y = 3;
  8'b000001xx: y = 2;
  8'b0000001x: y = 1;
  8'b00000001: y = 0;
  default    : y = 0;
endcase

endmodule
```

Listing 13: Módulo ej7.v

7.2 Testbench

```
module ENCODER83_TB();

reg  [7:0] X;
wire [2:0] Y;

encoder83 DUT(.x(X), .y(Y));

initial
begin
    #10 X=8'b00000000;
    #10 X=8'b00000001;
    #10 X=8'b00000010;
    #10 X=8'b00000100;
    #10 X=8'b00001000;
    #10 X=8'b00010000;
    #10 X=8'b00100000;
    #10 X=8'b01000000;
    #10 X=8'b10000000;
    #10 X=8'b01000100;
    #10 X=8'b00000011;
    #10 X=8'b00000001;
    #10
    $finish;
end

endmodule
```

Listing 14: Testbench ej7_tb.v

7.3 Waveform

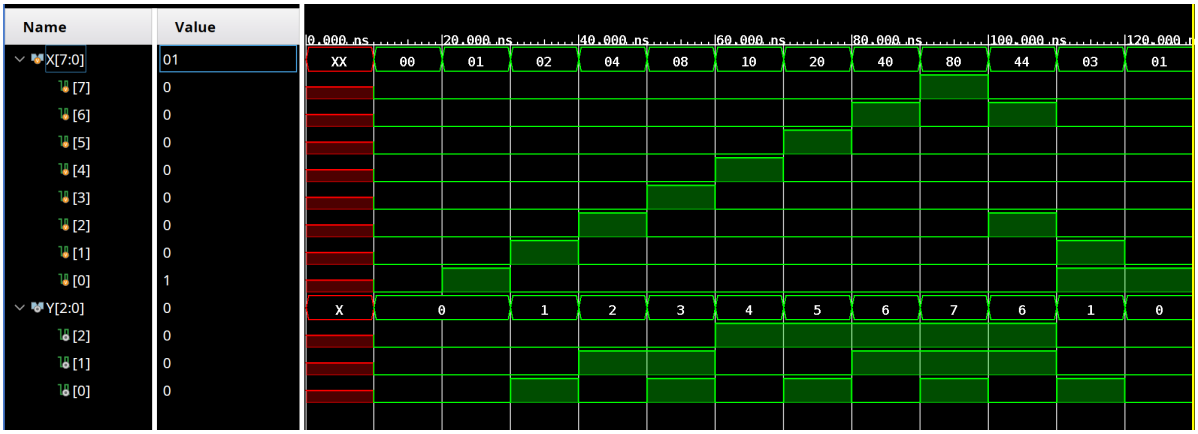

Figura 7: Waveform del ejercicio 7

8 Ejercicio 8

Comparador de 2 números (4 bits): Salidas para mayor, igual, menor.

8.1 Código del Módulo

```
module comparator(
input  wire [3:0] x1,
input  wire [3:0] x2,
output wire big,
output wire eq,
output wire les
);
reg [2:0] result;

assign les = result[2];
assign eq  = result[1];
assign big = result[0];

always@(*)
begin
  if (x1 < x2)
    result = 3'b100;//les on
  else if(x1 == x2)
    result = 3'b010;//eq on
  else if (x1 > x2)
  result =   3'b001;//big on
  else
    result = 3'b000;//no imput
end

endmodule
```

Listing 15: Módulo comparator

8.2 Testbench

```
`timescale 1ns / 100ps

module COMPARATOR_TB();

reg [3:0] X1;
reg [3:0] X2;

wire LES, EQ, BIG;

comparator COMP_DUT(.x1(X1), .x2(X2), .big(BIG), .eq(EQ),
    .les(LES));
initial
begin
  X1 = 0;
  X2 = 0;
  #10 X1 = 5; //X1>X2 -> big
  #10 X2 = 5; //X1=X2 -> eq
  #10 X1 = 1; //X1<X2 -> les
  #10 X2 = 0; //X1>X2 -> big
  #10;
  $finish;
end
endmodule
```

Listing 16: Testbench COMPARATOR_TB

8.3 Waveform

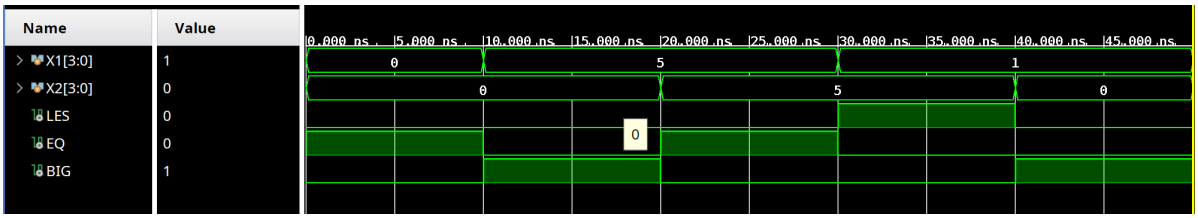

Figura 8: Waveform del ejercicio 8

9 Ejercicio 9

Sumador de 1 bit (half adder).

9.1 Código del Módulo

```
module half_adder(
input  wire a,
input  wire b,
input wire clk,
output wire s,
output wire c);

reg sum;
reg carry;

assign s = sum;
assign c = carry;

always@(posedge clk)
begin
  if((a && b))
  begin
    sum   <= 1'b0;
    carry <= 1'b1;
  end
  else
  begin
    carry <= 1'b0;
    if(a || b)
      sum <= 1'b1;
    else
      sum <= 1'b0;
  end
end

endmodule
```

Listing 17: Módulo half_adder

9.2 Testbench

```
`timescale 1ns/100ps

module HALF_ADDER_TB();

reg A, B;
wire S, C;
reg CLK;
integer i;

half_adder HALF_ADDER_DUT(.a(A), .b(B), .s(S), .c(C), .clk(CLK));

initial
begin
```

```
14    CLK = 1'b0;
15    for (i=0; i<10; i = i+1)
16      #10 CLK = ~CLK;
17    $finish;
18  end
19
20  initial
21  begin
22    A = 0;
23    B = 0;
24  end
25
26  always
27  begin
28    #5  A = 1; //s=1, c=0
29    #10 B = 1; //s=0, c=1
30    #10 A = 0; //s=1, c=0
31    #10 B = 0; //s=0, c=0
32  end
33
34  endmodule
```

Listing 18: Testbench HALF_ADDER_TB

9.3 Waveform

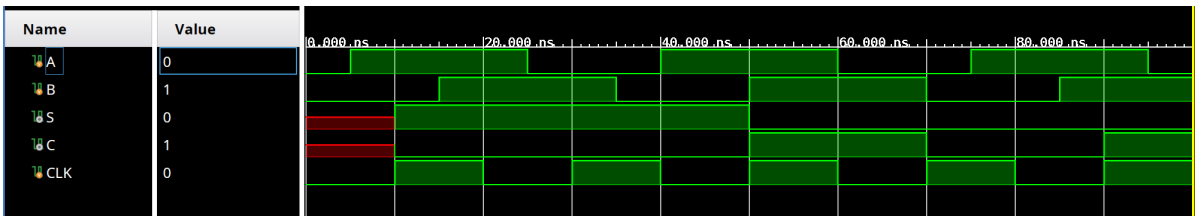

Figura 9: Waveform del ejercicio 9

10 Ejercicio 10

Sumador completo (full adder): con acarreo de entrada y salida.

10.1 Resolución

Se elige instanciar dos veces el medio sumador hecho en el ejercicio anterior y conectarlo como se ve en el diagrama.

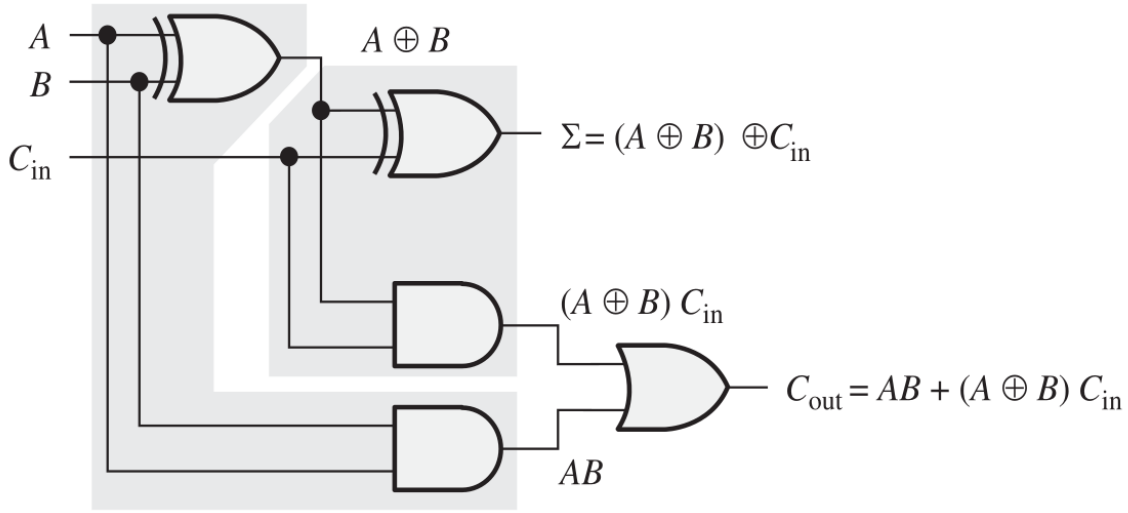

Figura 10: Diagrama del sumador completo implementado con dos half adders

10.2 Código del Módulo

```
module full_adder(
  input  wire a,
  input  wire b,
  input  wire cin,
  input  wire clk,

  output wire s,
  output wire cout
);

wire a1, b1, a2, b2;
wire s1, c1, s2, c2;

half_adder hadder1(.a(a1), .b(b1),   .s(s1), .c(c1), .clk(clk));
half_adder hadder2(.a(a2), .b(b2),   .s(s2), .c(c2), .clk(clk));

assign a1 = a;
assign b1 = b;
assign a2 = s1;
assign b2 = cin;

assign s  = s2;
assign cout = c1 || c2;

endmodule
```

Listing 19: Módulo full_adder

10.3 Testbench

```
`timescale 1ns/100ps

module FULL_ADDER_TB();

reg A, B, CIN;
wire S, COUT;
reg CLK;
integer i;

full_adder FULL_ADDER_DUT(.a(A), .b(B), .cin(CIN), .s(S),
   .cout(COUT), .clk(CLK));

initial
begin
  CLK = 1'b0;
  for (i=0; i<500; i = i+1)
    #5 CLK = ~CLK;
end

initial
begin
  A   = 0;
  B   = 0;
  CIN = 0;
end

always
begin
  #2;
      A   = 0;
      B   = 0;
      CIN = 0;
  #50;
      A   = 0;
      B   = 0;
      CIN = 1;
  #50;
      A   = 0;
      B   = 1;
      CIN = 0;
  #50;
      A   = 0;
      B   = 1;
      CIN = 1;
  #50;
      A   = 1;
      B   = 0;
      CIN = 0;
  #50;
```

```
49        A    = 1;
50        B    = 0;
51        CIN  = 1;
52    #50;
53        A    = 1;
54        B    = 1;
55        CIN  = 0;
56    #50;
57        A    = 1;
58        B    = 1;
59        CIN  = 1;
60    $finish;
61  end
62
63  endmodule
```

Listing 20: Testbench FULL_ADDER_TB

10.4 Waveform

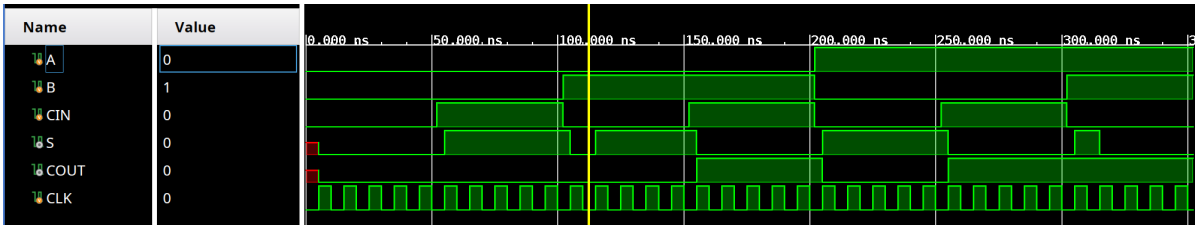

Figura 11: Waveform del ejercicio 10

11 Ejercicio 11

Sumador de 4 bits en cascada: usando full adders. Se toma el enfoque de sumadores sin acarreo anticipado.

11.1 Resolución

Para este ejercicio se conectan 4 full_adders en serie como se ve en la figura. No se usa acarreo anticipado si no que se usa la señal valid para indicar que ya pasaron 4 ciclos de reloj desde que se inició la suma; cubriendose así el peor caso posible en retardo de acarreo.

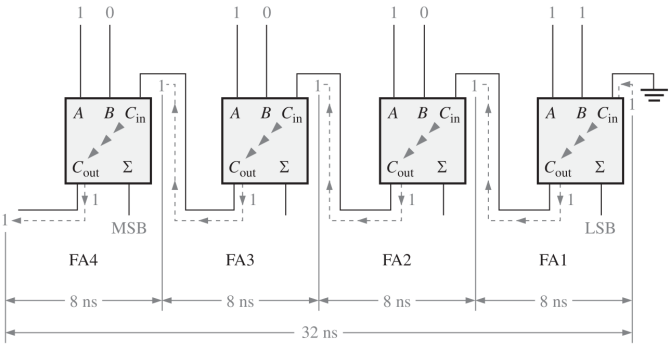

Figura 12: Diagrama del sumador de 4 bits implementado con full adders en cascada

11.2 Código del Módulo

```
module nibble_adder(
  input  wire [3:0] a,
  input  wire [3:0] b,
  input  wire       cin,
  output wire [3:0] s,
  output wire       cout,
  output wire       valid,//valid indica salida valida sin
     transiciones de accarreo ocurriendo
  input             clk
);

wire [3:0] cin_aux;
wire [3:0] cout_aux;

reg valid_state;
reg[1:0] valid_count;

full_adder FA4(.a(a[3]), .b(b[3]), .cin(cin_aux[3]), .s(s[3]),
   .cout(cout_aux[3]), .clk(clk));
full_adder FA3(.a(a[2]), .b(b[2]), .cin(cin_aux[2]), .s(s[2]),
   .cout(cout_aux[2]), .clk(clk));
full_adder FA2(.a(a[1]), .b(b[1]), .cin(cin_aux[1]), .s(s[1]),
   .cout(cout_aux[1]), .clk(clk));
full_adder FA1(.a(a[0]), .b(b[0]), .cin(cin_aux[0]), .s(s[0]),
   .cout(cout_aux[0]), .clk(clk));

assign cout       = cout_aux[3];
assign cin_aux[3] = cout_aux[2];
assign cin_aux[2] = cout_aux[1];
assign cin_aux[1] = cout_aux[0];
assign cin_aux[0] = cin;

assign valid = valid_state;

```

```
30 //Frente a un cambio en la entrada, deshabilita la validez de la
       salida
31 always@(a, b)
32 begin
33   valid_state = 1'b0;
34   valid_count = 2'b00;
35 end
36
37 //Cuento hasta 4 para habilitar la salida
38 always@(posedge clk)
39 begin
40   valid_state <= &valid_count; //valid_state = 1 cuando count=11
         (pasaron 4 clk desde cambio)
41   if(valid_count!=2'b11) //incremento cuando valid_state=0. evaluo
         valid_count por sincronizacion
42     valid_count <= valid_count+1;
43 end
44
45 endmodule
```

Listing 21: Módulo nibble_adder

11.3 Testbench

```
1  `timescale 1ns/100ps
2
3  module NIBBLE_ADDER_TB();
4
5  reg  [3:0] A, B;
6  reg        CIN;
7  wire [3:0] S;
8  wire COUT, VALID;
9  reg CLK;
10 integer i;
11
12 nibble_adder NIBBLE_ADDER_DUT( .a(A), .b(B), .cin(CIN), .s(S),
       .cout(COUT), .valid(VALID), .clk(CLK) );
13
14 initial
15 begin
16   CLK = 1'b0;
17   for (i=0; i<500; i = i+1)
18     #5 CLK = ~CLK;
19 end
20
21 initial
22 begin
23     A = 4'd0;
24     B = 4'd0;
25     CIN = 1'b0;
26 end
```

```
27
28 always
29 begin
30     #100;
31     A = 4'd7;
32     B = 4'd2;
33     #100;
34
35     A = 4'd3;
36     B = 4'd10;
37     #100;
38
39     A = 4'd5;
40     B = 4'd9;
41     #100;
42
43     A = 4'd12;
44     B = 4'd4;
45     #100;
46
47     A = 4'd1;
48     B = 4'd15;
49     #100;
50
51     A = 4'd8;
52     B = 4'd6;
53     #100;
54
55     A = 4'd15;
56     B = 4'd15;
57     #100;
58
59   $finish;
60 end
61
62 endmodule
```

Listing 22: Testbench NIBBLE_ADDER_TB

11.4 Waveform

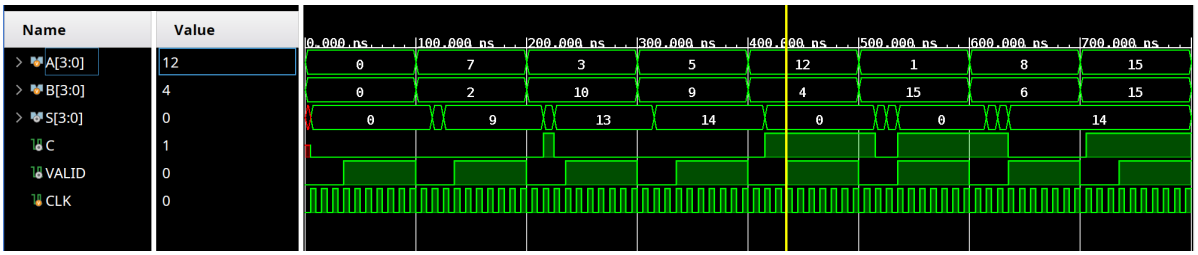

Figura 13: Waveform del ejercicio 11

12 Ejercicio 12

ALU simple de 4 bits: que realice suma, resta, AND y OR (seleccionado con un opcode).

12.1 Resolución

Para el AND y OR se implementan soluciónes en el mismo módulo. Para la suma se usa el nibble_adder del ejercicio anterior y para la resta se usa el nibble adder con entradas modificadas (b invertido y acarreo de entrada=1). Para interpretar la salida de la resta hay que considerar el carry_out invertido como bit de signo. Para poder ver eso se hace una salida auxiliar en el tb llamada S_RESTA que representa la salida con el bit de signo.

12.2 Código del Módulo

```
module nibble_alu(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire[1:0] op,
  input wire      clk,

  output wire[3:0] s,
  output wire      cout,
  output wire      valid
);

//variables manipulables segun operacion elegida
reg [3:0] a_reg, b_reg, s_reg;
reg       cin_reg;
reg       cout_reg, valid_reg;

//salidas wire a registros internos modificables
assign s     = s_reg;
assign cout  = cout_reg;
assign valid = valid_reg;

//cables auxiliares para salida de instancia de sumador
wire [3:0] s_sum;
wire cout_sum, valid_sum;

nibble_adder sumador(.a(a_reg), .b(b_reg), .cin(cin_reg),
  .s(s_sum), .cout(cout_sum), .valid(valid_sum), .clk(clk));

//codigos de operacion
reg add_op;    // op1 = add (00)
reg sub_op;    // op2 = sub (01)
reg and_op;    // op3 = and (10)
reg or_op;     // op4 = or  (11)


```

```
36 always@(posedge clk)
37 begin
38   add_op <= (op==2'b00) ? 1'b1 : 1'b0;
39   sub_op <= (op==2'b01) ? 1'b1 : 1'b0;
40   and_op <= (op==2'b10) ? 1'b1 : 1'b0;
41   or_op  <= (op==2'b11) ? 1'b1 : 1'b0;
42 end
43
44 always@(posedge clk)
45 begin
46   if(add_op)
47   begin
48     a_reg     <= a;
49     b_reg     <= b;
50     cin_reg   <= 1'b0;
51     s_reg     <= s_sum;
52     cout_reg  <= cout_sum;
53     valid_reg <= valid_sum;
54   end
55   else if(sub_op)
56   begin
57     a_reg     <= a;
58     b_reg     <= ~b;
59     cin_reg   <= 1'b1;
60     s_reg     <= s_sum;
61     cout_reg  <= ~cout_sum;
62     valid_reg <= valid_sum;
63   end
64   else if(and_op)
65   begin
66     a_reg     <= a;
67     b_reg     <= b;
68     s_reg      <= a_reg & b_reg;
69     cout_reg   <=1'b1;
70     valid_reg <= 1'b1;
71   end
72   else if(or_op)
73   begin
74     a_reg     <= a;
75     b_reg     <= b;
76     s_reg <= a_reg | b_reg;
77     cout_reg   <=1'b1;
78     valid_reg <= 1'b1;
79   end
80 end
81
82 endmodule
```

Listing 23: Módulo nibble_alu

12.3 Testbench

```
`timescale 1ns/100ps

module NIBBLE_ALU_TB();

reg  [3:0] A, B;
wire [3:0] S;
wire COUT, VALID;
reg[1:0] OP;
reg CLK;

wire[4:0] S_RESTA;//variable para visualizar resta en waveform

integer i;

nibble_alu NIBBLE_ALU_DUT( .a(A), .b(B), .op(OP), .s(S),
   .cout(COUT), .valid(VALID), .clk(CLK) );

initial
begin
  CLK = 1'b0;
  for (i=0; i<500; i = i+1)
    #5 CLK = ~CLK;
end

assign S_RESTA = {COUT, S};

initial
begin
    A  = 4'd0;
    B  = 4'd0;
    OP = 2'b00;
end

always
begin
    #100;
    OP = 2'b00; //suma
    A  = 4'd7;
    B  = 4'd2;

    #100;
    OP = 2'b01;//resta con salida positiva
    A  = 4'd10;
    B  = 4'd3;

    #100;
    OP = 2'b01;//resta con salida negativa
    A = 4'd5;
    B = 4'd9;
```

```
49      #100;
50
51      OP = 2'b10;//and bit a bit
52      A = 4'b1111;
53      B = 4'b0101;
54      #100;
55
56      OP = 2'b10;//and bit a bit
57      A = 4'b1010;
58      B = 4'b0110;
59      #100;
60
61      OP = 2'b11;//or bit a bit
62      A = 4'b0010;
63      B = 4'b1010;
64      #100;
65
66      A = 4'b0000;
67      B = 4'b1111;
68      #100;
69
70    $finish;
71  end
72
73  endmodule
```

Listing 24: Testbench NIBBLE_ALU_TB

12.4 Waveform

Waveform en formato decimal para interpretar las operaciones de suma y resta:

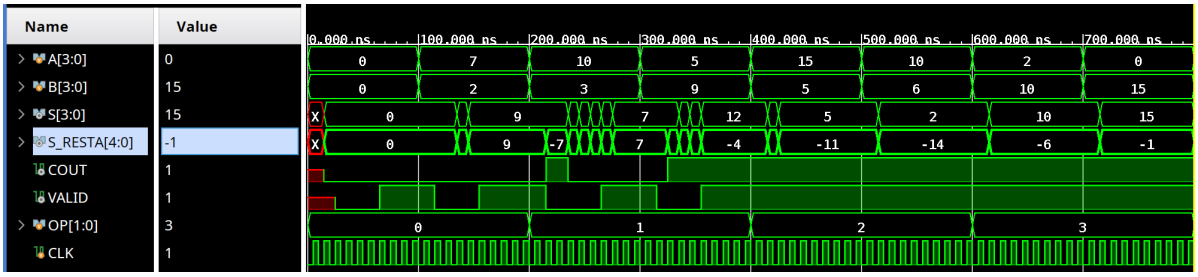

Figura 14: Waveform del ejercicio 12 - Formato decimal

Waveform en formato binario para ver las operaciones de AND y OR:

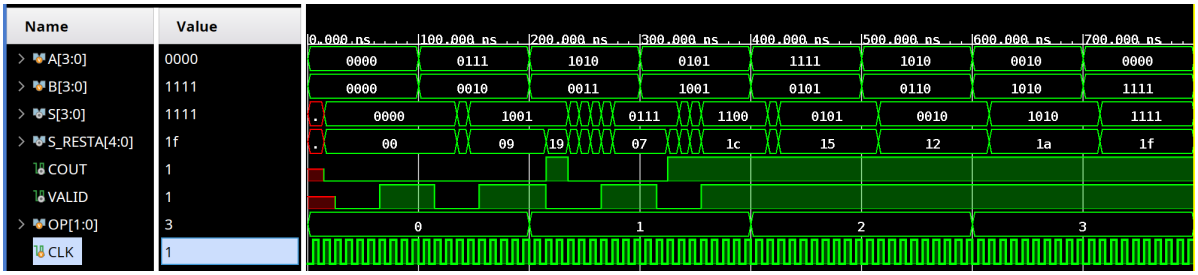

Figura 15: Waveform del ejercicio 12 - Formato binario

13 Ejercicio 13

Multiplexor 4:1.

13.1 Código del Módulo

```
module mux_41(
  input wire [3:0] x,
  input wire [1:0] sel,
  output wire y
);

assign y = x[sel];

endmodule
```

Listing 25: Módulo mux_41

13.2 Testbench

```
`timescale 1ns/100ps

module MUX41_TB();

reg  [3:0] X;
reg  [1:0] SEL;
wire       Y;

mux_41 mux( .x(X), .sel(SEL), .y(Y));

initial
begin
  X = 4'b0000;
  SEL = 2'b00;
end

always
begin
    #20;
    X       = 4'b0000;
```

```
21      #10 SEL = 2'b00;
22      #10 SEL = 2'b01;
23      #10 SEL = 2'b10;
24      #10 SEL = 2'b11;
25      #20;
26      X       = 4'b0101;
27      #10 SEL = 2'b00;
28      #10 SEL = 2'b01;
29      #10 SEL = 2'b10;
30      #10 SEL = 2'b11;
31      #20;
32      X       = 4'b1010;
33      #10 SEL = 2'b00;
34      #10 SEL = 2'b01;
35      #10 SEL = 2'b10;
36      #10 SEL = 2'b11;
37      #20;
38      X       = 4'b1111;
39      #10 SEL = 2'b00;
40      #10 SEL = 2'b01;
41      #10 SEL = 2'b10;
42      #10 SEL = 2'b11;
43      #20;
44      X       = 4'b0011;
45      #10 SEL = 2'b00;
46      #10 SEL = 2'b01;
47      #10 SEL = 2'b10;
48      #10 SEL = 2'b11;
49      #20;
50      X       = 4'b1100;
51      #10 SEL = 2'b00;
52      #10 SEL = 2'b01;
53      #10 SEL = 2'b10;
54      #10 SEL = 2'b11;
55      #20;
56      X       = 4'b0110;
57      #10 SEL = 2'b00;
58      #10 SEL = 2'b01;
59      #10 SEL = 2'b10;
60      #10 SEL = 2'b11;
61      #20;
62      X       = 4'b1001;
63      #10 SEL = 2'b00;
64      #10 SEL = 2'b01;
65      #10 SEL = 2'b10;
66      #10 SEL = 2'b11;
67
68    $finish;
69  end
70
71  endmodule
```

Listing 26: Testbench MUX41_TB

13.3 Waveform

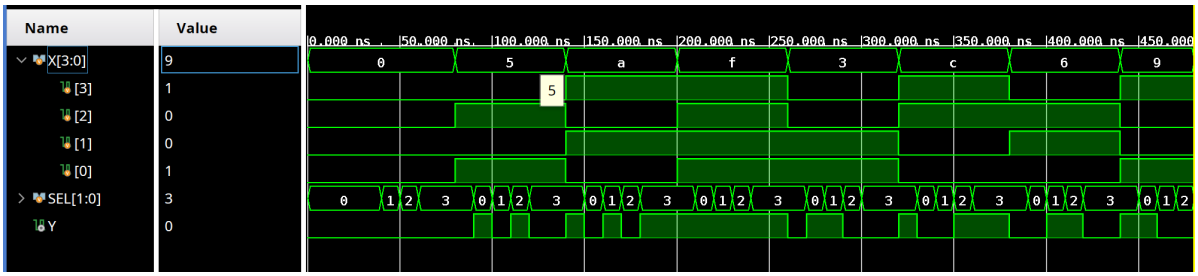

Figura 16: Waveform del ejercicio 13

14 Ejercicio 14

Demultiplexor 1:4.

14.1 Código del Módulo

```
module demux_14(//No se consideran los estados no-seleccionados; se
    dejan en estado previo
input  wire      x,
input  wire[1:0] sel,
input  wire      clk,

output reg[3:0] y
);
initial
  y = 4'b0000;

always@(posedge clk)
  y[sel] <= x;

endmodule
```

Listing 27: Módulo demux_14

14.2 Testbench

```
`timescale 1ns/100ps

module DEMUX_14_TB();

reg X, CLK;
reg [1:0] SEL;
```

```
7  wire [3:0] Y;
8
9  integer i;
10 demux_14 DEMUX_14_TB(.x(X), .sel(SEL), .y(Y), .clk(CLK));
11
12 initial
13 begin
14   X   = 1'b0;
15   CLK = 1'b0;
16   for(i=0; i<500; i=i+1)
17   begin
18     #5
19     CLK = ~CLK;
20   end
21 end
22
23 initial
24 begin
25   #20 X = 1'b1;
26       SEL = 2'b00;
27   #22 SEL = 2'b01;
28   #11 SEL = 2'b10;
29   #33 SEL = 2'b11;
30   #15 X   = 1'b0;
31   #17 SEL = 2'b10;
32   #20 SEL = 2'b01;
33   #33 SEL = 2'b00;
34   #20 $finish();
35 end
36
37 endmodule
```

Listing 28: Testbench DEMUX_14_TB

14.3 Waveform

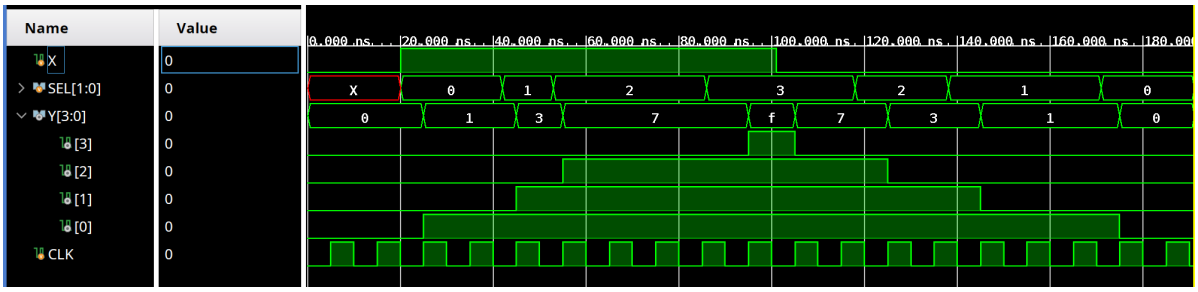

Figura 17: Waveform del ejercicio 14

15 Ejercicio 15

Latch tipo D: sensible al nivel.

15.1 Resolución

Si bien se podría implementar la lógica pura del latch, para este ejercicio se conectan las compuertas construyendo un latch tradicional. En el diagrama se ven los nombres de los wires usados para el codigo.

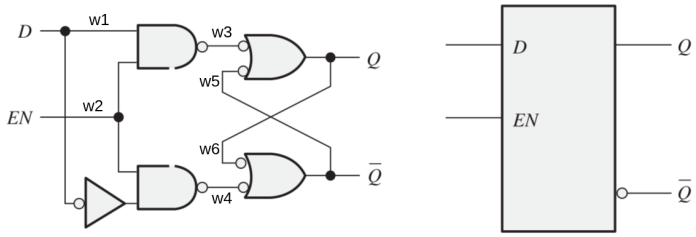

Figura 18: Diagrama del latch tipo D implementado con compuertas

15.2 Código del Módulo

```
module latchD(
input wire D,
input wire E,
output wire Q,
output wire Qnot
);

// Podría implementar solo la lógica pero como estoy manija
   implemento
// el mismo diagrama del libro :)))

wire w1, w2, w3, w4, w5, w6;

//Entradas
assign w1 = D;
assign w2 = E;
//Lógica del diagrama
assign w3 = ~(w1 && w2);
assign w4 = ~(w2 && ~w1);
assign w5 = (~w6 || ~w4);
assign w6 = (~w3 || ~w5);
//Salidas
assign Q = w6;
assign Qnot = w5;

endmodule
```

Listing 29: Módulo latchD

15.3 Testbench

```
`timescale 1ns/100ps

module ej15_tb();

reg d, e;
wire q, qNOT;

latchD latchD_tb(.D(d), .E(e), .Q(q), .Qnot(qNOT));

initial
begin
    #5;
    d = 1'b1;
    e = 1'b1;
    #10;
    d = 1'b0;
    e = 1'b0;
    #10;
    d  = 1'b0;
    e = 1'b1;
    #10;
    d  = 1'b1;
    e = 1'b0;
    #10;
    d  = 1'b1;
    e = 1'b1;
    #10;

    $finish;
end

endmodule
```

Listing 30: Testbench ej15_tb

15.4 Waveform

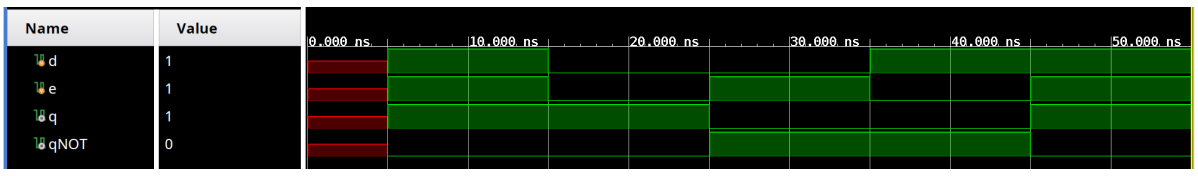

Figura 19: Waveform del ejercicio 15

16 Ejercicio 16

Flip-Flop tipo D: sensible al flanco de subida.

16.1 Resolución

En este caso se hace la implementación directamente. Se observa que, a diferencia del latch, hace falta un registro para mantener el estado.

Entradas		Salidas		
D	*CLK*	*Q*	$\overline{Q}$	*Comentarios*
1	↑	1	0	SET (almacena un 1)
0	↑	0	1	RESET (almacena un 0)
↑ = transición del reloj de nivel BAJO a nivel ALTO				

Figura 20: Tabla de verdad del flip-flop tipo D

16.2 Código del Módulo

```
module flipflopD(
  input wire  d,
  input wire  clk,
  output wire q,
  output wire qnot
);
//En este caso no hago todo el diagrama como el libro. Sin embargo, se observa que,
// por ser un flipflop, necesito si o si registros, no puedo asignar los wires

reg state;
assign q = state;
assign qnot = ~state;

always@(posedge clk)
  state <= d;

endmodule
```

Listing 31: Módulo flipflopD

16.3 Testbench

```
`timescale 1ns/100ps

module ej16_tb();

reg D, CLK;
wire Q, QNOT;

integer i;

```

```
flipflopD FLIPFLOPD_TB(.d(D), .clk(CLK), .q(Q), .qnot(QNOT));

initial
begin
  CLK = 1'b0;
  for(i=0; i<500; i=i+1)
    #5 CLK = ~CLK;
end

initial
begin
  D = 1'b1;
  #23 D = 1'b0;
  #21 D = 1'b1;
  #2  D = 1'b0;
  #44 D = 1'b1;
  #20;
  $finish;
end

endmodule
```

Listing 32: Testbench ej16_tb

16.4 Waveform

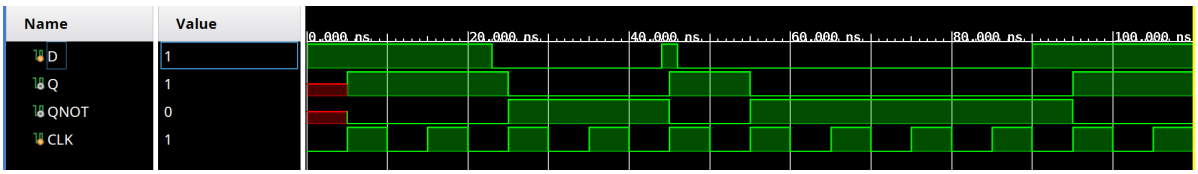

Figura 21: Waveform del ejercicio 16

17 Ejercicio 17

Contador binario de 4 bits : que cuente de 0 a 15 en cada flanco de reloj.

17.1 Código del Módulo

```
module contador_4(
  input  wire rst,
  input  clk,
output wire [3:0] q
);
reg [3:0] state;
assign q = state;

initial
```

```
10   state = 4'b0000;
11
12 always@(posedge clk or rst)
13   if(rst)
14     state <= 4'b0000;
15   else
16     state <= state+1;
17
18 endmodule
```

Listing 33: Módulo contador_4

17.2 Testbench

```
1  `timescale 1ns/100ps
2
3  module ej17_tb();
4
5  reg CLK, RST;
6  wire [3:0] Q;
7  integer i;
8
9  contador_4 CONTADOR_4_TB(.q(Q), .rst(RST), .clk(CLK));
10
11 initial
12 begin
13   CLK = 1'b0;
14   i   = 1'b0;
15   RST = 1'b0;
16 end
17
18 initial
19 begin
20   for(i=0; i<50; i=i+1)
21   begin
22     #10;
23     CLK = ~CLK;
24   end
25   $finish;
26 end
27 initial
28 begin
29   #20 RST = 1'b1;
30   #5  RST = 1'b0;
31   #50 RST = 1'b1;
32   #52 RST = 1'b0;
33 end
34
35 endmodule
```

Listing 34: Testbench ej17_tb

17.3 Waveform

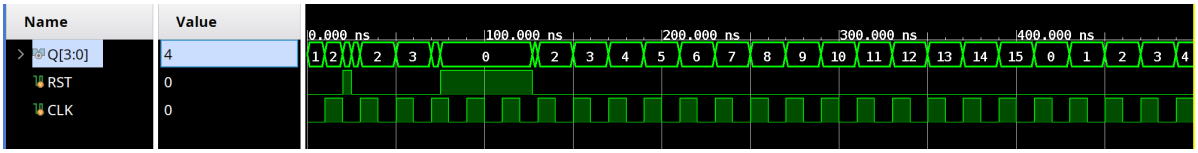

Figura 22: Waveform del ejercicio 17

18 Ejercicio 18

Contador módulo-N: parametrizable.

18.1 Resolución

Se elije un límite de 8 bits para el contador. Un cambio de módulo durante uso no resetea si no que simplemente incrementa el límite. La salida Q es la salida en binario.

18.2 Código del Módulo

```
module contador_N(
  input wire[7:0]  N,
  input wire       rst,
  input wire       clk,
  output wire[7:0] Q //Determino que cuenta hasta máximo 256
);

reg [7:0] state;
assign Q = state;

always@(posedge clk or rst)
begin
  if(rst)
    state <= 8'b00000000;
  if( (state < N) && (!rst) )
    state <= state+1;
  else
    state <= 8'b00000000;
end

endmodule
```

Listing 35: Módulo contador_N

18.3 Testbench

```
`timescale 1ns/100ps

```

```
3  module ej18_tb();
4
5  reg [7:0]N;
6  reg RST, CLK;
7  wire[7:0] Q;
8  integer i;
9
10 initial
11 begin
12   i   = 0;
13   RST = 1'b0;
14   CLK = 1'b0;
15   N   = 8'd5;
16 end
17
18 contador_N CONTADOR_N_TB(.N(N), .rst(RST), .clk(CLK), .Q(Q));
19
20 initial
21 begin
22   for(i=0; i<300; i=i+1)
23     #10 CLK = ~CLK;
24   $finish;
25 end
26 initial
27 begin
28   #210 N   = 8'd20;
29   #400 RST = 1'b1;
30   #2   RST = 1'b0;
31   #1   N   = 8'd100;
32 end
33 endmodule
```

Listing 36: Testbench ej18_tb

18.4 Waveform

Waveform 1: Se observa el inicio del contador que llega hasta 5 y vuelve a arrancar desde 0

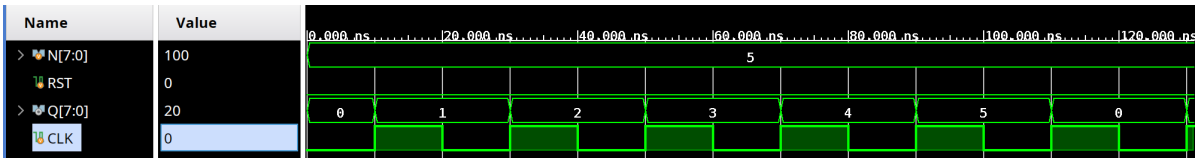

Figura 23: Waveform del ejercicio 18 - Inicio del contador

Waveform 2: Se puede apreciar el cambio de N durante la ejecución del contador. No se reinicia si no que simplemente sigue sumando

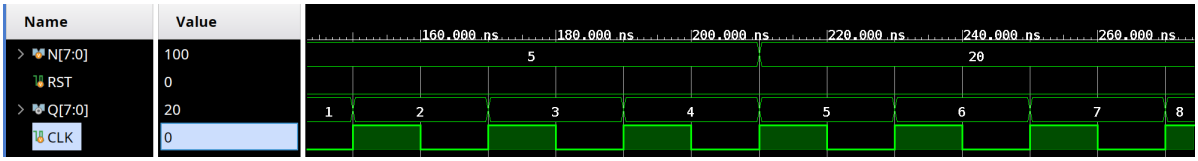

Figura 24: Waveform del ejercicio 18 - Cambio de módulo

Waveform 3: Misma corrida, se ve el limite superior

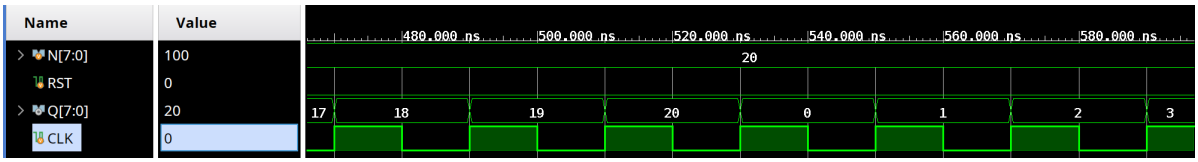

Figura 25: Waveform del ejercicio 18 - Límite superior

Waveform 4: Reset en funcionamiento

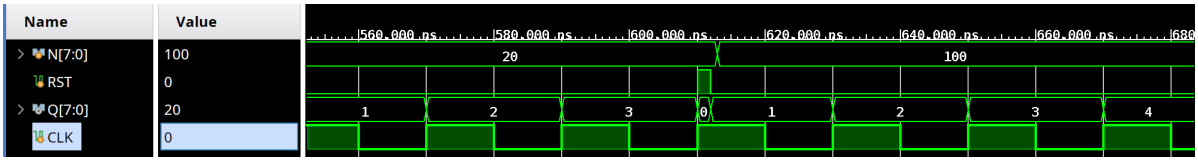

Figura 26: Waveform del ejercicio 18 - Reset

19 Ejercicio 19

Registro de desplazamiento de 8 bits: con carga paralela y desplazamiento hacia la izquierda.

19.1 Código del Módulo

```
module srg_4(
  input  wire[7:0] x,
  input  wire      sh_ld, //1=shift / 0=load
  input  wire      clk,
  output wire      y
);

reg [7:0]  buffer;
reg        serial_output;
wire       shift;
wire       load;

assign y = serial_output;
assign shift = sh_ld;
assign load  = ~sh_ld;

```

```
17 initial
18   buffer = 8'd0;
19
20 always@(posedge clk)
21 begin
22   if(load)
23     buffer <= x;
24   else if(shift)
25   begin
26     serial_output <= buffer[7];//desplazamiento a la izquierda,
         sale MSB
27     buffer <= buffer << 1;
28   end
29   else
30     buffer <= 8'b00000000;
31 end
32
33 endmodule
```

Listing 37: Módulo srg_4

19.2 Testbench

```
1  module ej19_tb();
2
3  reg [7:0] X;
4  reg       SH_LD;
5  reg       CLK;
6  wire      Y;
7
8  integer   i;
9
10 srg_4 SRG_TB(.x(X), .sh_ld(SH_LD), .clk(CLK), .y(Y));
11
12 initial
13 begin
14   CLK   = 1'b0;
15   X     = 8'd0;
16   SH_LD = 1'b1;
17 end
18
19 initial
20 begin
21   for (i=0; i<500; i=i+1)
22     #5 CLK = ~CLK;
23 end
24
25 initial
26 begin
27   #6  SH_LD = 1'b0;// LOAD
28   #1  X = 8'b11111111;
```

```
29    #20 SH_LD = 1'b1;// SHIFT
30    #100;
31    #3  SH_LD = 1'b0;// LOAD
32    #2  X = 8'b11111111;
33    #20 SH_LD = 1'b1;// SHIFT
34    #50 SH_LD = 1'b0;// LOAD
35    #2  X = 8'b10101010;
36    #20 SH_LD = 1'b1;// SHIFT
37    #120;
38    $finish();
39  end
40  endmodule
```

Listing 38: Testbench ej19_tb

19.3 Waveform

Se marcan 3 puntos en la salida de onda

- El primer marcador (desde la izquierda) indica un punto en el cual todos los bits fueron desplazados por cero y no se actualiza la entrada mientras que el selector sigue en SHIFT, con lo cual la salida entrega ceros
- En el segundo marcador se activo la entrada (SH_LD en bajo por un momento) con lo cual el registro volvió a empezar. Luego, justo en el marcador se vuelve a activar la entrada con otro codigo en X con lo cual se pisa la información anterior
- Finalmente en el ultimo se ve cómo el ultimo patrón ingresado terminó de salir y quedan ceros a la salida por no volver a activarse la entrada

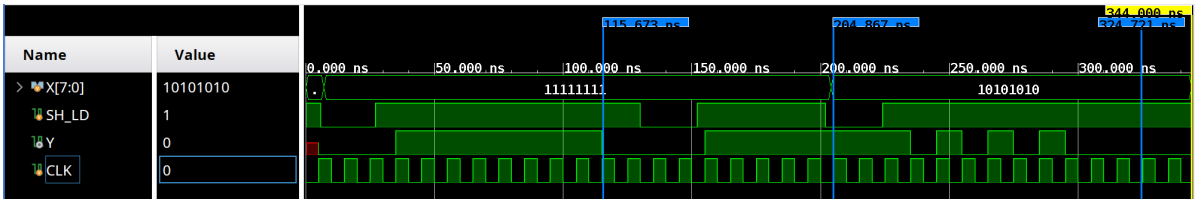

Figura 27: Waveform del ejercicio 19

20 Ejercicio 20

Diseñar una FSM Moore que detecte la secuencia ”101.[en] una entrada serial (din). La FSM tiene una entrada de datos (din), reloj (clk) y reset (rst). La salida (det) vale 1 solo cuando se ha detectado la secuencia ”101”. Si aparece la secuencia, la máquina sigue buscando solapamientos (ej: en 10101, detecta 2 veces).

20.1 Resolución

Para el detector de secuencias 101 se implementa el diagrama lógico de la figura. Nótese que la máquina de estados de Moore es con overlap; no necesariamente se re-inicia al entregar una secuencia.

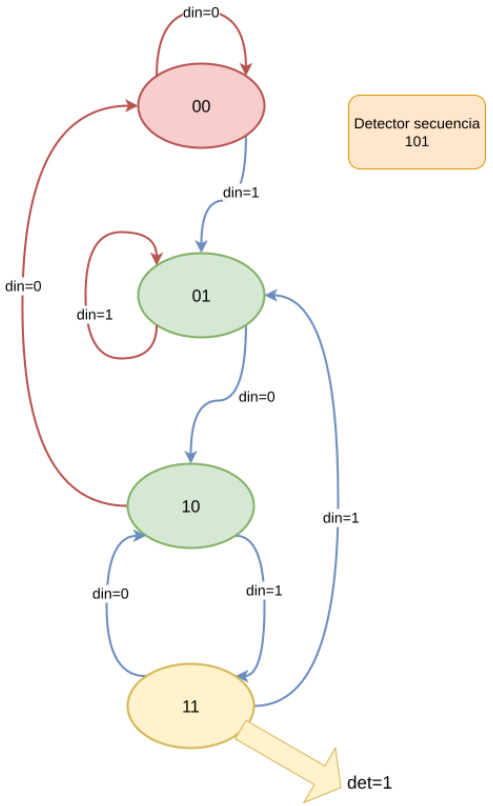

Diagrama de la FSM

20.2 Código del Módulo

```
module detector_secuencias(
  input  din,
  input  clk,
  input  rst,
  output det);

reg [1:0] estado; //cuatro estados posibles;  mirar diagrama

assign det = &estado; // la salida es 1 cuando el estado es 11

initial
estado <= 2'b00;

always@(posedge clk or posedge rst)
begin
  if(rst==1)
    estado <= 2'b00; // reset asíncrono
  else
  begin
    case (estado)
    2'b00: estado <= (din) ? 2'b01 : 2'b00;
    2'b01: estado <= (din) ? 2'b01 : 2'b10;
    2'b10: estado <= (din) ? 2'b11 : 2'b00;
```

```
    2'b11: estado <= (din) ? 2'b01 : 2'b10;
    default: estado <= 2'b00;
    endcase
  end
end

endmodule
```

Listing 39: Módulo detector_secuencias

20.3 Testbench

```
`timescale 1ns/100ps

module ej20_tb();

reg X, RST, CLK;
wire Y;
integer i;

detector_secuencias DETECTOR_SECUENCIAS_TB(
  .din(X),
  .rst(RST),
  .clk(CLK),
  .det(Y)
);

initial
begin
  X   = 0;
  RST = 0;
  CLK = 0;

  for(i=0; i<100; i=i+1)//25 períodos de reloj
    #10 CLK = ~CLK;

  $finish;
end

initial
begin
    @(negedge CLK); X   = 0;
    @(negedge CLK); X   = 0;
    @(negedge CLK); X   = 1;
    @(negedge CLK); X   = 1;
    @(negedge CLK); X   = 0;
    @(negedge CLK); X   = 1;
    @(negedge CLK); X   = 0;
    @(negedge CLK); X   = 1;
    @(negedge CLK); RST = 1;
    @(negedge CLK); RST = 0;
```

```
40      @(negedge CLK); X    = 0;
41      @(negedge CLK); X    = 1;
42      @(negedge CLK); X    = 1;
43      @(negedge CLK); X    = 0;
44      @(negedge CLK); X    = 1;
45      @(negedge CLK); X    = 0;
46      @(negedge CLK); X    = 1;
47      @(negedge CLK); X    = 0;
48      @(negedge CLK); X    = 0;
49      @(negedge CLK); X    = 1;
50      @(negedge CLK); X    = 0;
51      @(negedge CLK); X    = 0;
52      @(negedge CLK); X    = 1;
53      @(negedge CLK); X    = 0;
54      @(negedge CLK); X    = 0;
55  end
56
57  endmodule
```

Listing 40: Testbench ej20_tb

20.4 Waveform

Se puede ver el reset asincrono en funcionamiento durante la segunda salida positiva de Y enviando al estado inicial y re-iniciandose la secuencia.

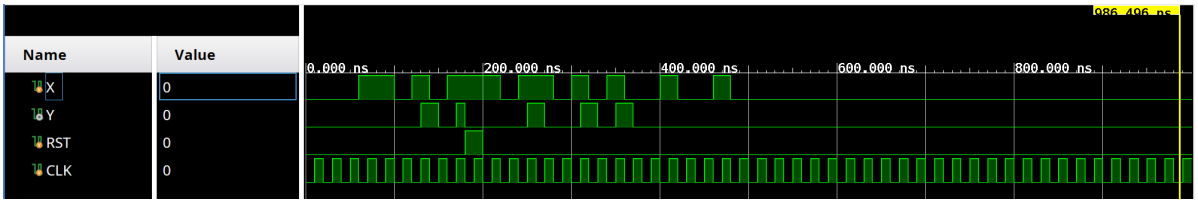

Figura 28: Waveform del ejercicio 20