INTRODUCCIÓN A HDL VERILOG

Departamento de Tecnología Electrónica

Universidad de Sevilla

Rev. 9 (ene 2019)

Autor: Paulino Ruiz de Clavijo
Modificaciones al texto original:
I.Gómez, A. Molina, P. Parra y S. Martín

Índice

Introducción a HDL Verilog

Bloque I: Diseño de circuitos combinacionales

Bloque II: Diseño de circuitos secuenciales

Bloque III: Simulación y verificación

Bloque IV: Implementación

Introducción

- Verilog es un lenguaje formal para describir e implementar circuitos electrónicos.
- Es similar a un lenguaje de programación imperativo: formado por un conjunto de sentencias que indican como realizar una tarea.
- Algunas diferencias:
  - La mayoría de las sentencias se ejecutan concurrentemente
  - Cada sentencia corresponde a un bloque de circuito

BLOQUE I

Diseño de Circuitos Combinacionales

Bloque I: Índice

Estructura general de una descripción Verilog

Tipos de descripciones

Señales, puertos E/S y arrays

Sintaxis básica

Estructura de descripciones Verilog

```
module mi_circuito (
   input x, y,
   input z,
   output f1, f2
   );

   wire cable_interno;
   reg  variable_a

   ...
   ...
   ...

endmodule
```

Falta un ";" aquí

Declaración del módulo con sus entradas y salidas: input, output, inout

Declaración de señales y variables que se utilizarán internamente en la descripción: wire, reg

Descripción del módulo. Hay varias alternativas para realizarla: funcional, procedimental, estructural

Ejemplo: circuito votador

Expresión lógica:

$$z=ab+ac+bc$$

```
module votador (input a,b,c,output z);

  assign z= (a & b) | (a & c) | (b & c);

endmodule
```

Tipos de descripciones

Descripción funcional

- Modela circuitos combinaciones.
- Consiste en asignaciones de las salidas de manera continua utilizando assign.
- Todas las sentencias assign se ejecutan de manera concurrente.

```
module votador(input a,b,c, output z);

  assign z = a&b | a&c | b&c;

endmodule
```

z=ab+ac+bc

Tipos de descripciones

Falta un ";" aquí

Descripción procedimental

- Permite el uso de estructuras de control
- La descripción es algorítmica, igual que el software
- Facilita la creación de funciones complejas
- Se basa en la sentencia always
- Todas las sentencias always se ejecutan de forma concurrente

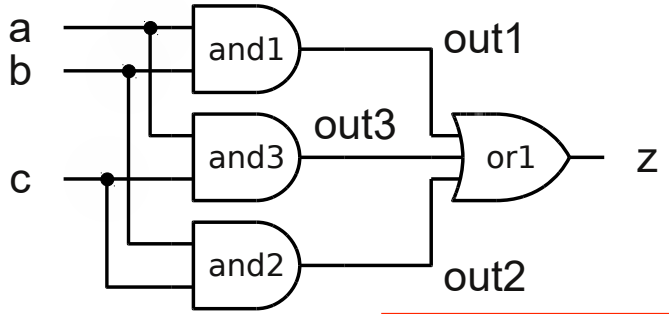
```
module votador(
  input a,b,c,
  output reg z)

  always @(a,b,c)
    if(a==1)
      if(b==1 || c==1)
        z=1;
      else
        z=0;
    else
       if(b==1 && c==1)
         z=1;
       else
         z=0;
endmodule
```

z=ab+ac+bc

Tipos de descripciones

Descripción estructural

- Se conectan módulos que ya están definidos previamente
- Las puertas lógicas básicas ya están predefinidas en Verilog (and, nand, or, nor, xor, xnor, not, buf, etc.)
- Es muy útil para la interconexión de los módulos que se creen
- Observe que se utilizan wires para conectar salidas y entradas de puertas




```
module votador(
  input a,b,c,
  output z)

  wire out1,out2,out3;

  and and1(out1,a,b);
  and and2(out2,b,c);
  and and3(out3,a,c);
  or or1(z,out1,out2,out3);

endmodule
```

Falta un ";" aquí

Sintaxis puertas: *GATE instance_name(outputs, inputs)*

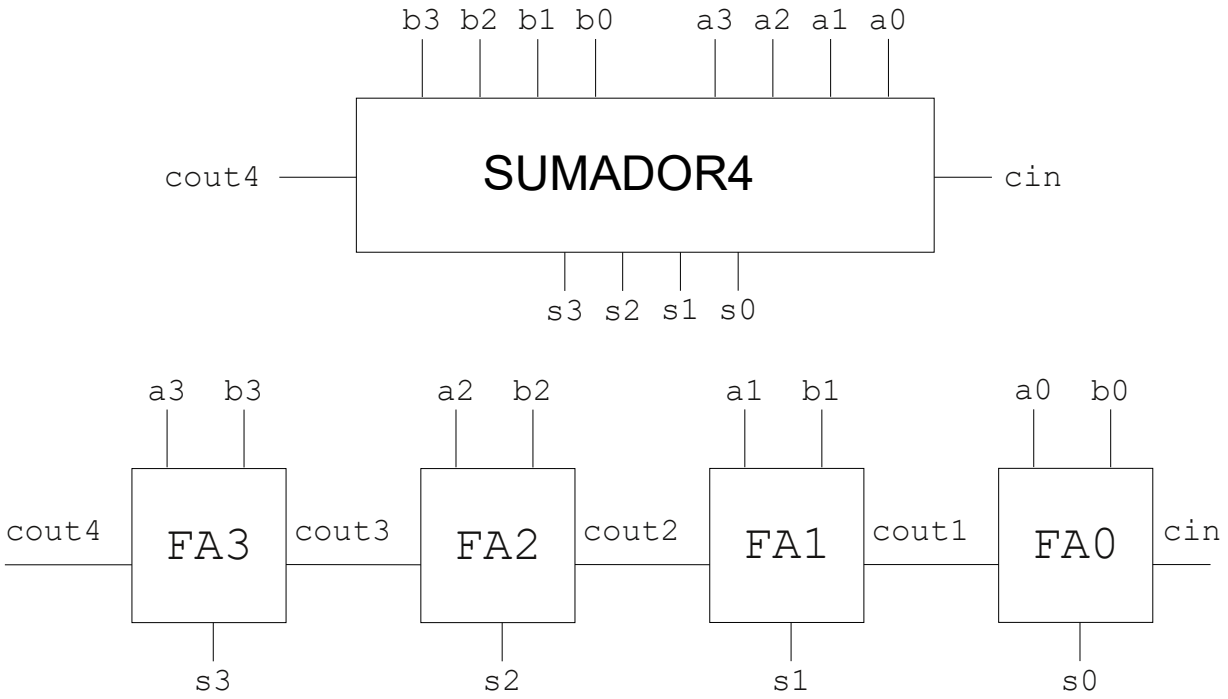
z=ab+ac+bc

Tipos de descripciones

- Todas las sentencias assign y always se ejecutan de manera concurrente.
- La descripción estructural se utiliza para la interconexión de los diferentes módulos que se creen.
- Las descripciones estructurales conforman la jerarquía del sistema que se está diseñando.

Tipos de descripciones

Ejemplo de descripción de un SUMADOR DE 4 bits a partir de varios FULL-ADDER de un bit



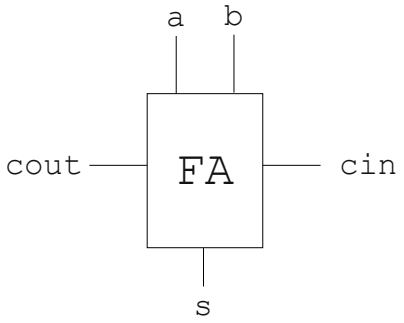
Tipos de descripciones

Pasos:

1) Descripción de un módulo para el FULL-ADDER de un bit.
2) Descripción de un módulo donde se utilizan 4 FULL-ADDER de un bit y se interconectan los cables de los módulos.

Tipos de descripciones

Descripción funcional del FA de un bit




```
module fulladder(
    input a,
    input b,
    input cin,
    output s,
    output cout);

    assign  s = a ^ b ^ cin;
    assign  cout = a & b | a & cin | b & cin;
endmodule
```

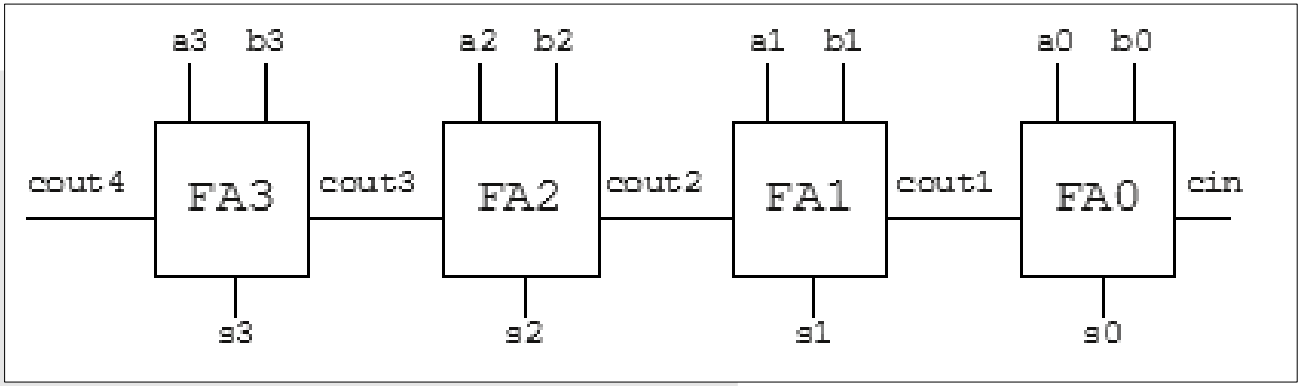
cin	a	b	cout	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$s = a \oplus b \oplus c$$

$$cout = a \cdot b + a \cdot cin + b \cdot cin$$

Tipos de descripciones

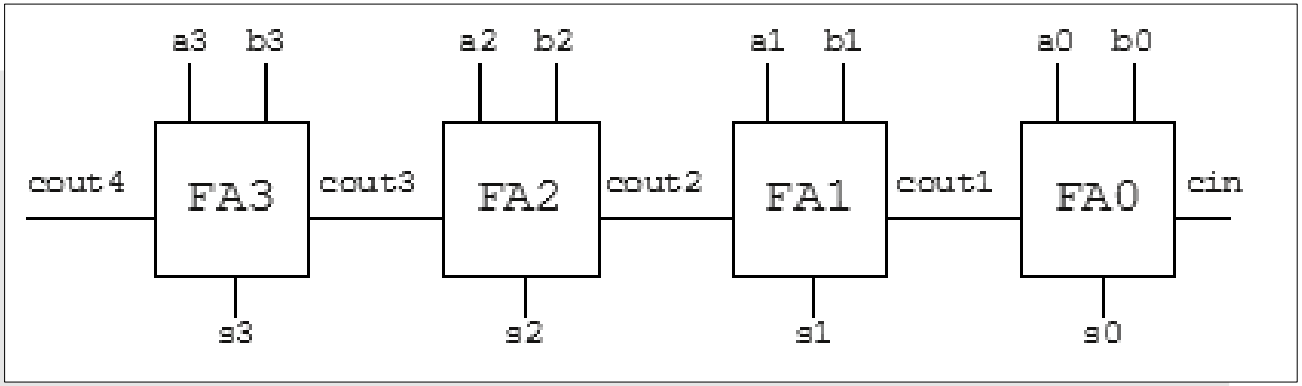
Unión de 4 FULL-ADDER: conexión posicional.




```
module sumador4(
  input [3:0] a,
  input [3:0] b,
  input cin,
  output [3:0] s,
  output cout4);

  wire cout1,cout2,cout3;

  fulladder fa0 (a[0], b[0], cin,   s[0], cout1);
  fulladder fa1 (a[1], b[1], cout1, s[1], cout2);
  fulladder fa2 (a[2], b[2], cout2, s[2], cout3);
  fulladder fa3 (a[3], b[3], cout3, s[3], cout4);

endmodule
```

Tipos de descripciones

Unión de 4 FULL-ADDER: conexión nombrada.




```
module sumador4(
  input [3:0] a,
  input [3:0] b,
  input cin,
  output [3:0] s,
  output cout4);

  wire cout1,cout2,cout3;

  fulladder fa0 (.a(a[0]), .b(b[0]), .cin(cin),   .s(s[0]), .cout(cout1));
  fulladder fa1 (.a(a[1]), .b(b[1]), .cin(cout1), .s(s[1]), .cout(cout2));
  fulladder fa2 (.a(a[2]), .b(b[2]), .cin(cout2), .s(s[2]), .cout(cout3));
  fulladder fa3 (.a(a[3]), .b(b[3]), .cin(cout3), .s(s[3]), .cout(cout4));

endmodule
```

Tipos de descripciones

Descripción procedimental

```
module sumador4(
  input [3:0] a,
  input [3:0] b,
  input cin,
  output [3:0] s,
  output cout4);

  reg [4:0] res;
  always @(a,b,cin)
    res = a + b + cin;

  assign cout=res[4];
  assign s = res[3:0];

endmodule
```

- El tipo de descripción utilizada para el módulo no influye en cómo se comporta éste.
- Se pierde la estructura interna. La herramienta de síntesis genera un hardware equivalente.

Tipos de señales

- Existen dos tipos básicos de señales

  wire: corresponden a cables físicos que interconectan componentes, por tanto, no tienen memoria.

  reg: (también llamada variable). Son utilizados para almacenar valores, tienen memoria.

- Los tipos (reg) se utilizan para modelar el almacenamiento de datos
- Todos los asignamientos que se realicen dentro de un procedimiento (always) deben ser sobre una señal tipo reg

Puertos de entrada/salida

Cuando se declaran módulos se puede especificar si un puerto es tipo wire o reg

- Si no se indica nada es wire por defecto
- Los cables (wire) son utilizados con la sentencia assign
- Los registro (reg) son asignados en los procedimientos

```
module mi_circuito (
  input wire x,
  input z,
  output reg mem
  );
  ...
endmodule
```

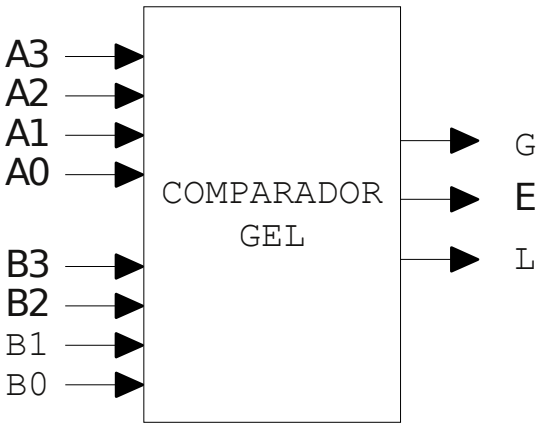
"Vector" es un término más correcto para este tipo de agrupaciones de bits

Arrays

Los arrays son agrupaciones de bits, motivos:

- Los puertos de entrada/salida se agrupan (buses) para trabajar con mayor comodidad
- Los registros pueden ser de varios bits

Sintaxis: [M:N]




```
module comparador_gel (
   input wire [3:0] a,
   input [3:0] b,
   output g,e,l
   );
   ...
endmodule
```

Sintaxis básica

Literales

Sentencia assign

Sentencia always

Expresiones y operadores

Sentencias condicionales

Sintaxis básica

Verilog distingue entre mayúsculas y minúsculas

Se pueden escribir comentarios:

Comentario en linea: precedido de doble barra “//”

```
wire a; // Este cable se conecta con f2
```

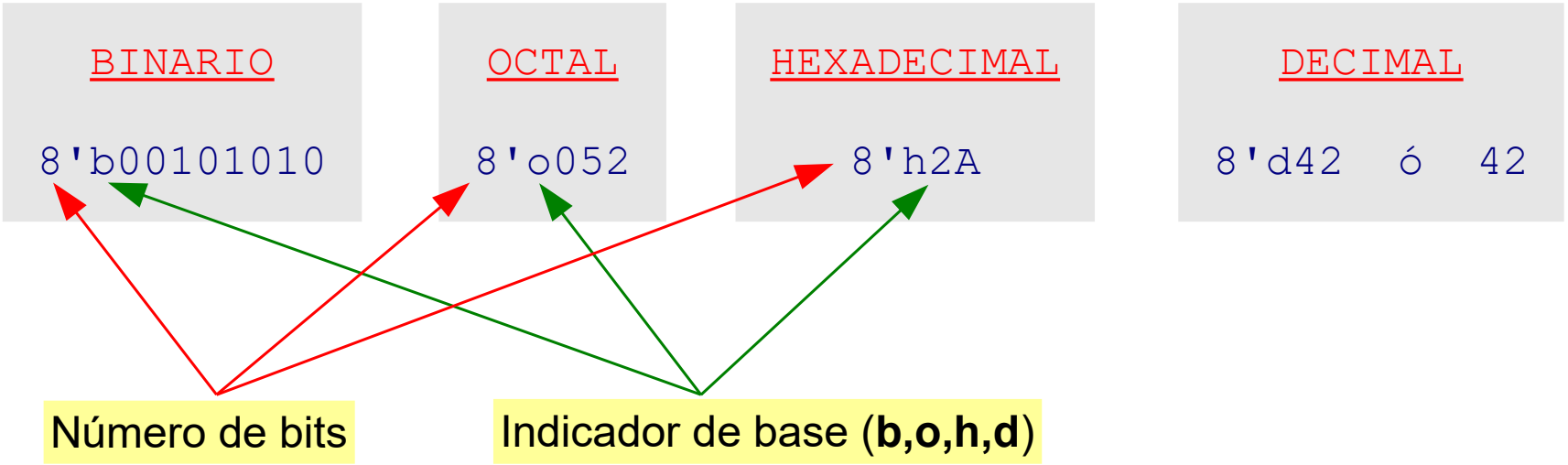
Comentario de varias líneas: comienza con `/*` y termina con `*/`

```
/* Este cable conecta muchos componentes
   y necesito varias lineas para explicarlo
   correctamente */

wire a;
```

Sintaxis básica

Literales: Se puede expresar en varios formatos

Ejemplo: “00101010” en binario



Sintaxis básica

Ejemplo de uso:

Circuito que siempre tiene sus salidas a uno

```
module siempre_uno (
  input  x,
  output [7:0] salida1,
  output [3:0] salida2
  );

  assign salida2 = 4'b1111;
  assign salida1 = 8'hFF;

endmodule
```

Sentencia assign

Todas las sentencias assign se ejecutan de manera concurrente

En el ejemplo la descripción de la salida f2 es equivalente a:

```
assign f2 = x & y & z;
```

```
module otro_ejemplo (
  input  x, y, z,
  output f1, f2
  );

  assign f1 = x & y;
  assign f2 = f1 & z;

endmodule
```

Sentencia always

Un bloque always se ejecuta concurrentemente con los demás bloques always y assign que hay en la descripción HDL

Los bloques always tienen una lista de sensibilidad:

La lista de sensibilidad consiste en una lista de señales.

El código del bloque always se ejecuta solo si cambia alguna de las señales de la lista de sensibilidad.

La sintaxis es:

```
always @(a,b)
  c = a | b;
```

Sentencia always

Una sentencia always suele contener varias sentencias, en cuyo caso, debe utilizar un bloque "begin" … "end"

Los bloques begin/end se utilizan para agrupar un conjunto de sentencias.

Son ampliamente utilizados

```
module (input a, b, c, d
        output reg f1,
        output reg f2);
  always @(a,b,c,d)
    begin
      f1 = a | b;
      f2 = c & d;
    end
endmodule
```

Sentencia always

Importante regla general sobre la lista de sensibilidad:

Siempre que se esté describiendo un componente combinacional, se deben incluir en la lista de sensibilidad ~~todas las entradas del componente~~

del always todas las señales que se "leen" dentro del always

Se puede simplificar la sintaxis mediante: **`always @(*)`**

```
module (input a, b, c, d,
        input e, f, g, h,
        output f1, f2);
  always @(a,b,c,d,e,f,g,h)
    begin
      ...
    end
endmodule
```

=

```
module (input a, b, c, d,
        input e, f, g, h,
        output f1, f2);
  always @(*)
    begin
      ...
    end
endmodule
```

Operadores

Operadores a nivel de bits:

Operador	Ejemplo de código Verilog
&	`c = a&b; // Operación AND de todos los bits`
|	`c = a|b; // Operación OR de todos los bits`
^	`c = a^b; // Operación XOR de todos los bits`
~	`b = ~a; // Inversión de todo los bits`

- ✔ Estos operadores trabajan con todos los bits.
- ✔ Si la variable es de un único bit operan como los operadores del álgebra de conmutación.

Operadores

Más operadores a nivel de bits

Operador	Ejemplo de código Verilog
~&	d = a ~& b; // Operador NAND a nivel de bits
~|	d = a ~| b; // Operador NOR a nivel de bits
~^	d = a ~^ b; // Operador EXNOR a nivel de bits

Todos estos operadores pueden trabajar con una única variable (reduction operators). En tal caso actúan sobre los bits del operando.
Ej: d = &a (reduction AND) o d=^ b (reduction xor)

Operadores

Ejemplo de uso de operadores a nivel de bits:

Módulo que realiza el complemento a uno de una palabra de 16 bits

```
module complemento_a1(
    input [15:0]  palabra,
    output [15:0] complemento_1);

    assign complemento_1 = ~palabra;

endmodule
```

Operadores

Operadores relacionales:
devuelven 1 si se cumple la condición

Operador	Ejemplo de código en Verilog
<	`a < b;  //¿Es a menor que b?`
>	`a > b;  //¿Es a mayor que b?`
>=	`a >= b; //¿Es a mayor o igual que b?`
<=	`a <= b; //¿Es a menor o igual que b?`
==	`a == b; //Devuelve 1 si a es igual que b`
!=	`a != b; //Devuelve 1 si a es distinto de b`

Operadores

Operadores lógicos:

No deben confundirse con los operadores a nivel de bits.

Operador	Ejemplo de código Verilog
&&	`a && b; // Devuelve 1 si a y b son verdaderos`
||	`a || b; // Devuelve 1 si a ó b es verdadero`
!	`!a; // Devuelve 1 si a es falso ó 0 si a es verdadero`

Operadores

Operadores aritméticos:

Operador	Ejemplo de código Verilog
*	`c = a * b;  // Multiplicación`
/	`c = a / b;  // División`
+	`sum = a + b; // Suma de a+b`
-	`resta = a - b; // Resta`

Operadores

Otros operadores:

Operador	Ejemplos en código Verilog
<<	`b = a << 1; //Desplazamiento a la //izq. de un bit`
>>	`b = a >> 1; //Desplazamiento a la //der. de un bit`
?:	`c = sel ? a : b; // si sel es uno //entonces c = a,sino entonces c = b`
{}	`{a, b, c} = 3'b101; // Conatenación: // Asigna una palabra a bits // individuales: a=1, b=0 y c=1`

Aparte de estos operadores existen más que se pueden encontrar en la bibliografía

Sentencias condicionales

La sentencia condicional más común es la sentencia:

`if` … `else` …

```
if ( a > 0 )
   Sentencia
else
   Sentencia
```

```
if ( a == 0 )
   Sentencia
else if( b != 1 )
   Sentencia
```

Sólo se pueden usar en procedimientos “always”

En las condiciones de esta sentencia se pueden utilizar todos los operadores lógicos y relacionales

Sentencias condicionales

Si hay más de una sentencia tras una condición, hay que utilizar bloques “begin” … “end”

```
always @(a)
 begin
  if ( a > 0 )
   f1 = 1;
   f2 = 1;
  else
   f1 = 0;
 end
```

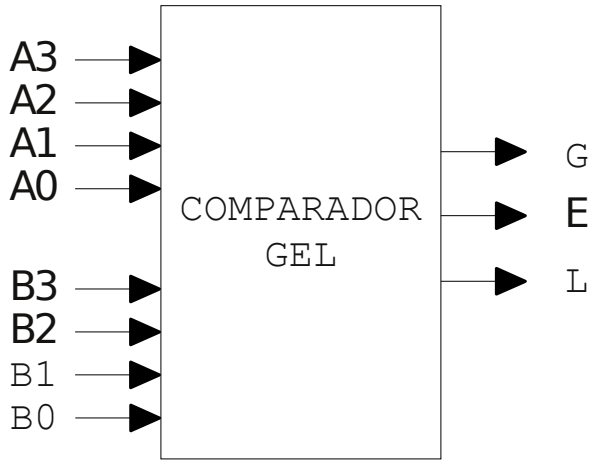
ERROR

```
always @(a)
 begin
  if ( a > 0 )
   begin
    f1 = 1;
    f2 = 1;
   end
  else
   f1 = 0;
 end
```

Correcto

Sentencias condicionales

Ejemplo:
comparador GEL




```
module comparador_gel(
    input [3:0] a,
    input [3:0] b,
    output reg g, // si a < b => (g,e,l) = (0,0,1)
    output reg e, // si a = b => (g,e,l) = (0,1,0)
    output reg l);


    always @(a, b)
      begin
        g = 0;
        e = 0;
        l = 0;
        if (a > b)
           g =1;
        else if (a < b)
           l = 1;
        else
           e = 1;
      end
endmodule
```

Sentencias condicionales

Sentencia case

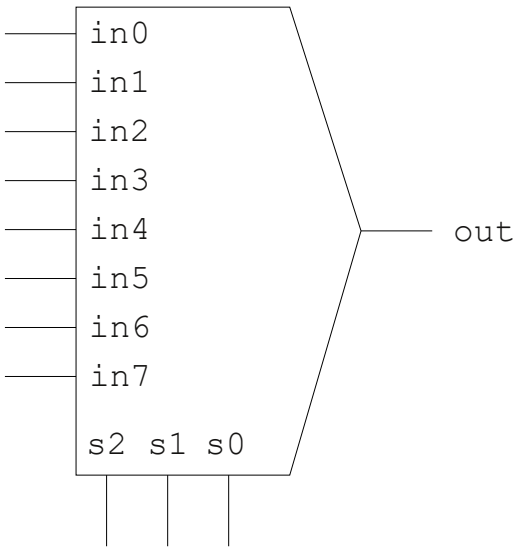
- Se utiliza dentro de un proceso “always”
- Si alguno de los casos tiene más de una sentencia hay que utilizar un bloque “begin” … “end”
- Se puede utilizar default para los casos no enumerados

```
reg [1:0] x;
always @(x)
 begin
      case(x)
       0:
        salida_1 = 1;
       1:
        begin
         salida_1 = 1;
         salida_2 = 0;
        end
       2:
        salida_2 = 1;
       3:
        salida_1 = 0;
      endcase
 end
```

Sentencias condicionales

Multiplexor 8:1

Ejemplo de acceso a elementos individuales de un array




```
module mux8_1(
    input [2:0] s,
    input [7:0] in,
    output reg out);


    always @(s, in)
        case (s)
                3'h0: out = in[0];
                3'h1: out = in[1];
                3'h2: out = in[2];
                3'h3: out = in[3];
                3'h4: out = in[4];
                3'h5: out = in[5];
                3'h6: out = in[6];
            default: out = in[7];
        endcase
endmodule
```

BLOQUE II

Diseño de Circuitos Secuenciales

Bloque II: Índice

Sintaxis II

Biestables

Máquinas de estados

Registros

Contadores

Sintaxis II

Definición de constantes

Operador de concatenación

Lista de sensibilidad con detección de flancos

Asignaciones bloqueantes / no bloqueantes

Sintaxis II

Dentro de un módulo se pueden definir constantes utilizando parameter

Es útil en la definición de máquinas de estados

Ejemplo:

```
parameter uno_con_tres_bits = 3'b001,
          ultimo = 3'b111;

reg [2:0] a;

a = ultimo;
```

Sintaxis II

El operador concatenar se utiliza para agrupar señales para que formen un array

Sintaxis: {señal, señal, ...}

Ejemplo:

Detector del número 3

```
module concatena(
  input a,b,c,
  output reg igual_a_3
  );

  always @(*)
    case({a,b,c})
      3'b011:
         igual_a_3 = 1;
      default:
         igual_a_3 = 0;
    endcase
endmodule
```

Sintaxis II

Detección de flanco

Sirve para que un proceso sólo se ejecute en determinados flancos de reloj de una o varias señales de entrada.

Se indica en la lista de sensibilidad de un proceso mediante un prefijo a la señal:

El prefijo posedge detecta el flanco de subida

El prefijo negedge detecta el flanco de bajada

Sintaxis II

Ejemplo de detección de flanco negativo de un reloj

```
module detector_flanco(
   input clk,
   output reg z);

   always @(negedge clk)
        ....
endmodule
```

Sintaxis II

Asignamiento bloqueante signo =

Si en un proceso se desea que la salida cambie inmediatamente, se debe utilizar una asignación bloqueante.

Esto modela una salida combinacional.

Importa el orden en que se efectúan las asignaciones bloqueantes puesto que las acciones en un proceso se ejecutan secuencialmente

Sintaxis II

Asignamiento no bloqueante signo <=

La asignación no bloqueante modela las escrituras en flip-flops.

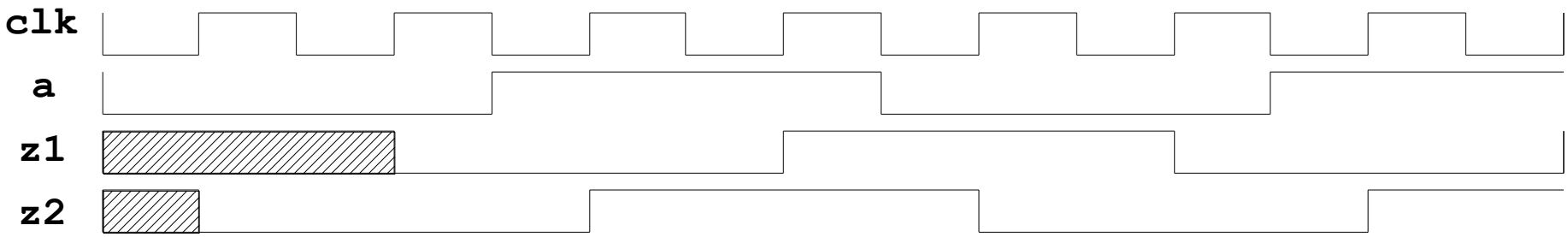
Se calculan primero los valores de la derecha de la asignación de todas las asignaciones <=, tras esto, se asignan todas simultáneamente.

Cuando se tiene una serie de asignaciones no bloqueantes, no importa el orden en que son escritas.

Sintaxis II

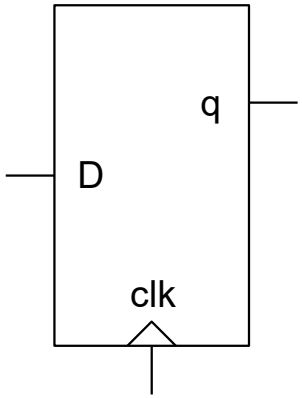
```
module no_bloqueante(input a,clk,
  output reg z1);
  reg q;
  always @(posedge clk)
    begin
      q  <= a;
      z1 <= q;
    end
endmodule
```

```
module bloqueante(input a,clk,
  output reg z2);
  reg q;
  always @(posedge clk)
    begin
      q  = a;
      z2 = q;
    end
endmodule
```

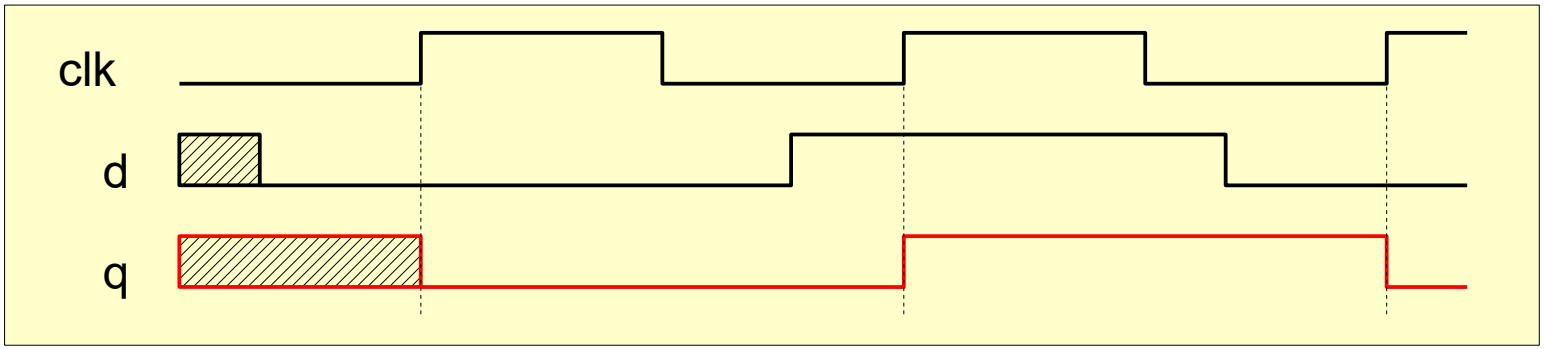


Biestables

Ejemplo de biestables:

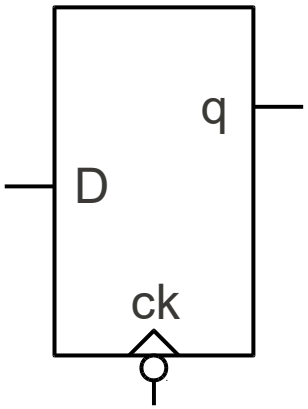



```
module biestable_d(
   input ck,d,
   output reg q);

always @ (posedge clk)
   q <= d;

endmodule
```

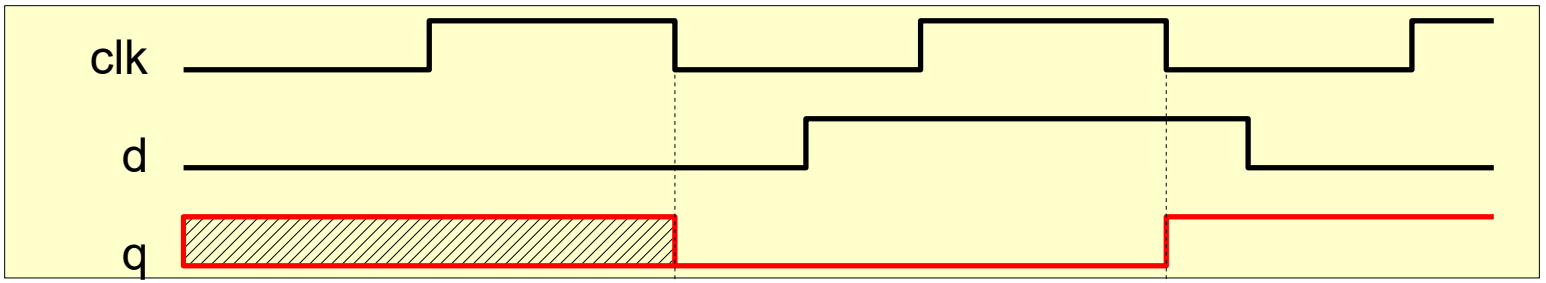


Biestables

Ejemplo de biestable D disparado en flanco negativo

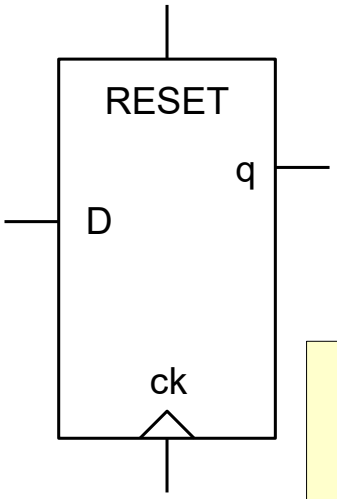



```
module biestable_d(
   input ck,d,
   output reg q);

   always @ (negedge clk)
     q <= d;

endmodule
```



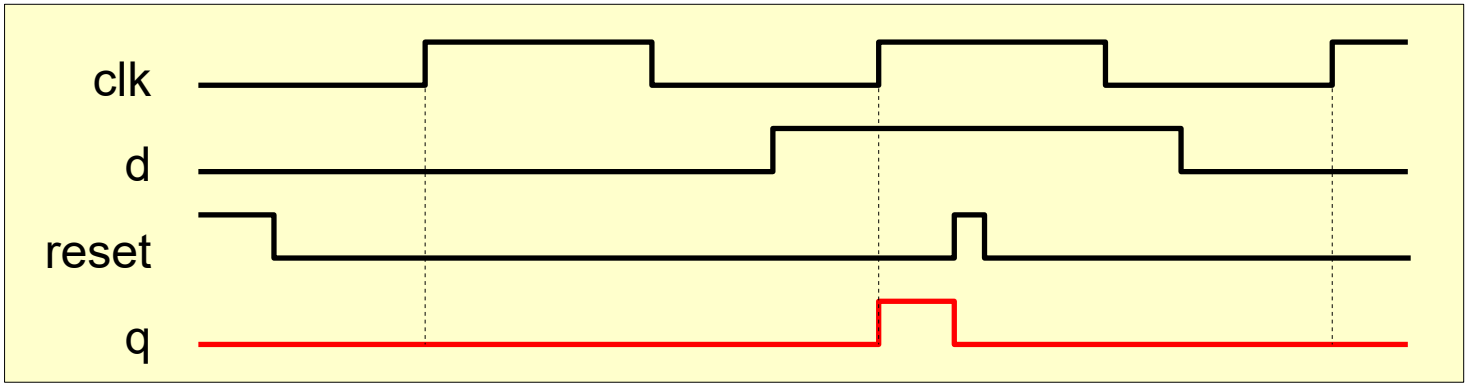
Biestables

Biestable D con reset asíncrono



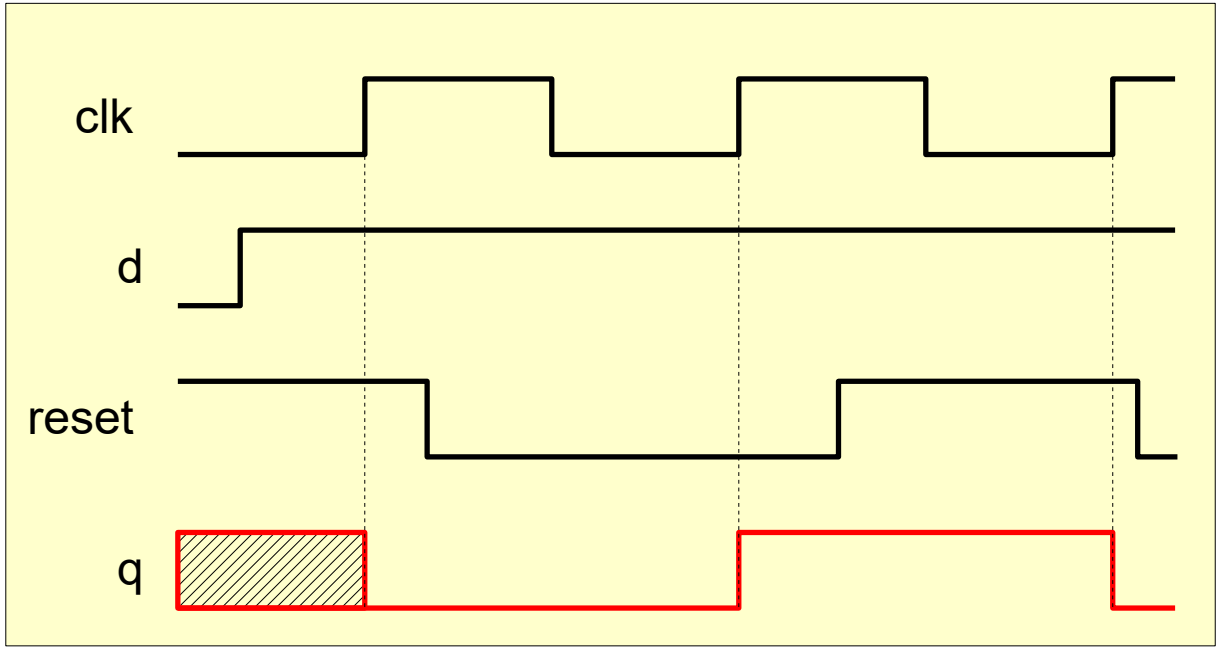

```
module biestable_d(
  input ck,d,reset,
  output reg q);

always @ (posedge clk or posedge reset)
  if (reset)
    q <= 1'b0;
  else
    q <= d;

endmodule
```

El "or" en la lista de sensibilidad es equivalente a poner una coma ","



Biestables

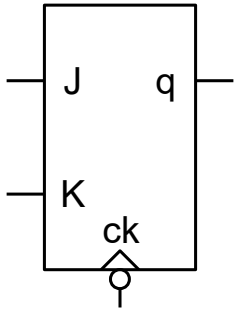
Biestable D con reset síncrono




```
module biestable_d(
  input ck,d,reset,
  output reg q);

always @ (posedge clk)
  if (reset)
    q <= 1'b0;
  else
    q <= d;
endmodule
```

Biestables

Biestable JK



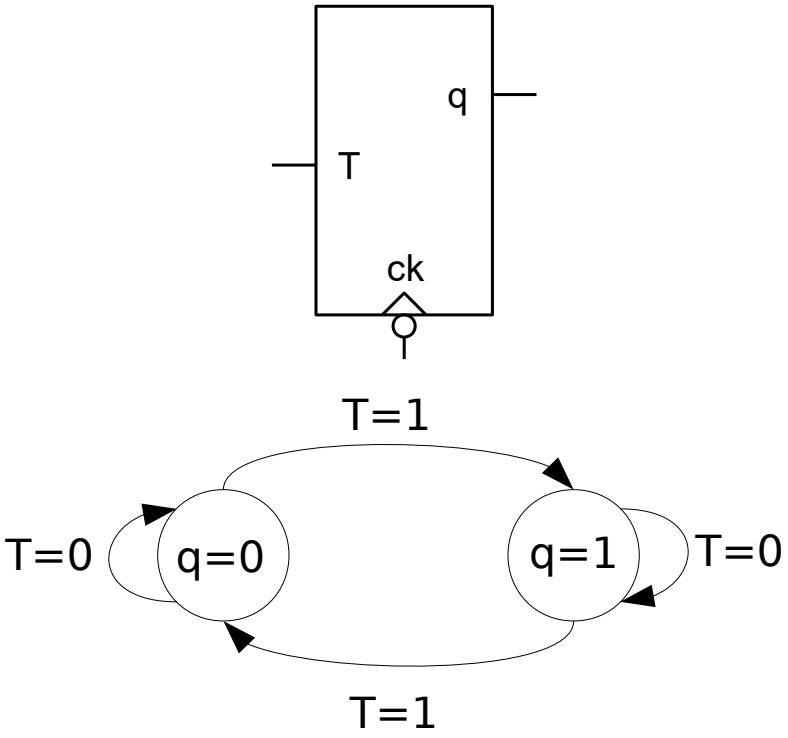
JK / q	00	01	11	10
0	0	0	1	1
1	1	0	0	1

Q

```
module jk_flip_flop (
  input       clk,
  input       j,
  input       k,
  output reg q);

always @(negedge clk)
  case ({j,k})
    2'b11 : q <= ~q;     // Cambio
    2'b01 : q <= 1'b0;   // reset.
    2'b10 : q <= 1'b1;   // set.
    2'b00 : q <= q;      //
  endcase

endmodule
```

Biestables

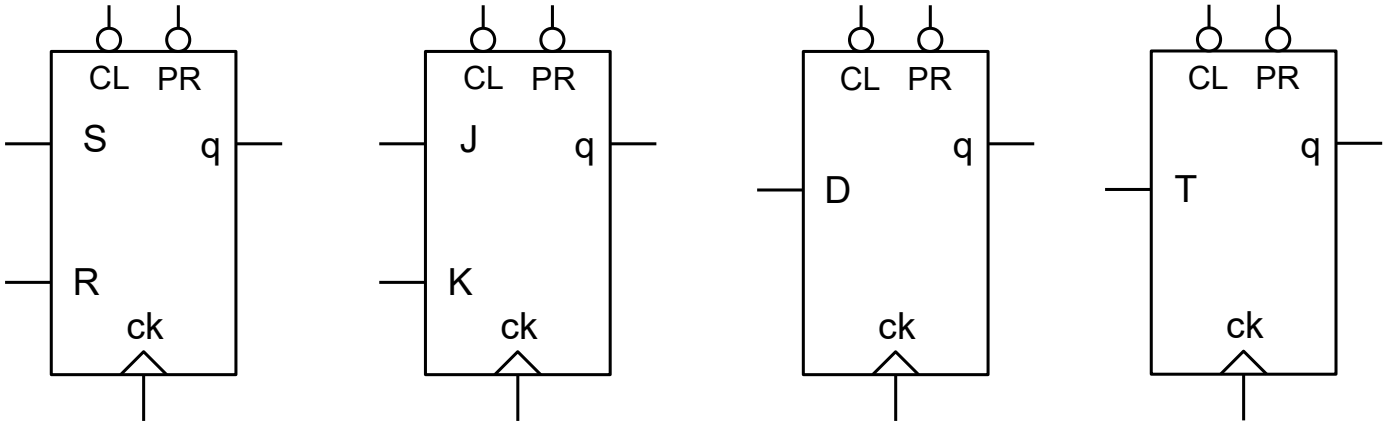
- Biestable T



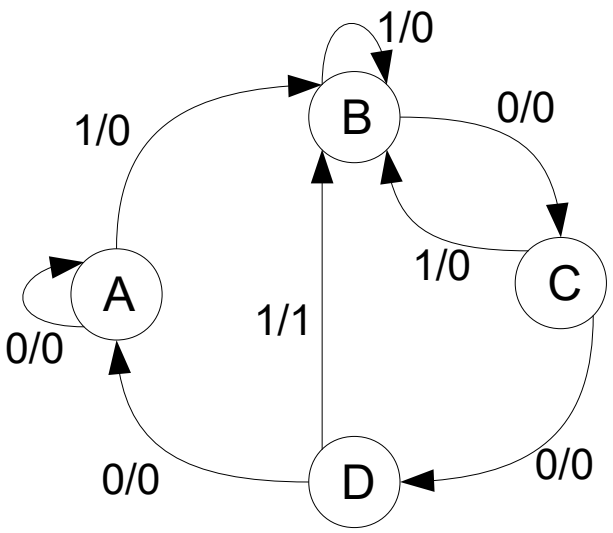

```
module biestable_t(
  input ck,
  input t,
  output reg q);

  always @(negedge ck)
    if (t == 1)
      q <= ~q;

endmodule
```

Biestables

Ejercicios: Realice los siguientes biestables, sabiendo las señales CL y PR son síncronas



Máquinas de estado



- Se utilizará una estructura general del código en la que hay 2 procesos
  - Uno de asignación de siguientes estados
  - Otro de cálculo de siguiente estado y salidas

Máquinas de estado

```
module mi_diagrama_de_estados(
   input LISTA_DE_ENTRADAS,
   output reg LISTA_DE_SALIDAS);

// DEFINICION Y ASIGNACIÓN DE ESTADOS
   parameter LISTA_DE_ESTADOS

// VARIABLES PARA ALMACENAR EL ESTADO PRESENTE Y SIGUIENTE
   reg [N:0] current_state, next_state;

// PROCESO DE CAMBIO DE ESTADO
always @(posedge clk or posedge reset)
    .......
// PROCESO SIGUIENTE ESTADO Y SALIDA
always @(current_state, LISTA_DE_ENTRADAS)
    .......
endmodule
```

"clk" y "reset" también deben aparecer aquí como "input", junto con las entradas propiamente dichas de la máquina de estados

Máquinas de estado

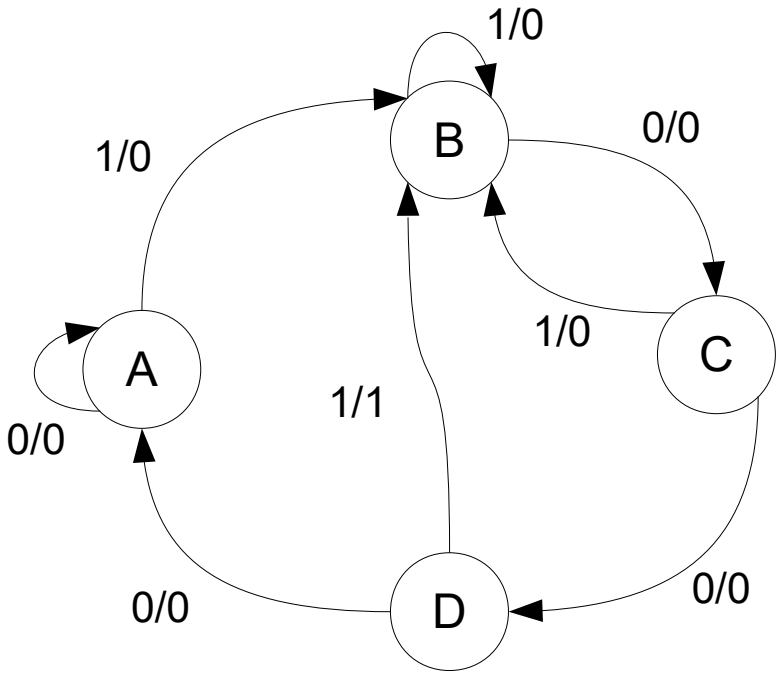
En la estructura general hay que completar 4 partes de código:

- ✔ Definición y asignación de estados, según el número de estados utilizaremos mas o menos bits.
- ✔ Definición de registros para almacenar el estado actual y el siguiente. Deben ser del mismo tamaño en bits que el utilizado en el punto anterior
- ✔ Proceso de cambio de estado: Siempre es el mismo código
- ✔ Proceso de cálculo de siguiente estado y salida: Hay que rellenar el código correspondiente las transiciones del diagrama de estados

Máquinas de estado

```
module maquina_estados(
  input x, clk, reset,
  output reg z);

  parameter A = 2'b00,
            B = 2'b01,
            C = 2'b10,
            D = 2'b11;

  reg [1:0] current_state,next_state;

  always @(posedge clk, posedge reset)
    begin
     if(reset)
       current_state <= A;
     else
       current_state <= next_state;
    end

              SIGUE ->
```

Asignación de estados



Proceso de asignación de siguiente estado

Máquinas de estado

El proceso de calculo del siguiente estado y salida se realiza con una única sentencia **case**

- La sentencia case debe contemplar todos los estados del diagrama de estados
- Antes de la sentencia **case** se recomienda establecer por defecto a cero todas las salidas.

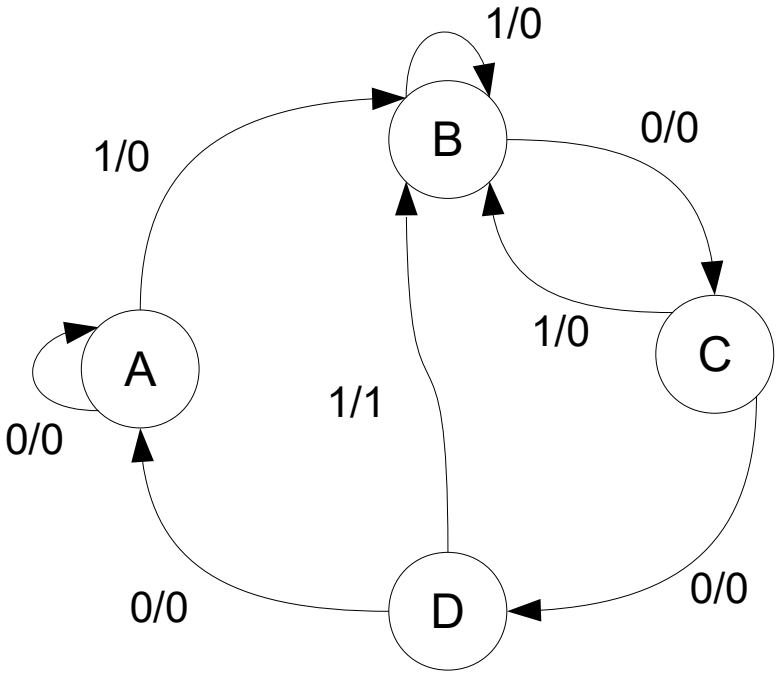
Máquinas de estado

```
always @(current_state,x)
  begin
    z = 0;
    case(current_state)
      A:
        if(x == 1)
          next_state = B;
        else
          next_state = A;
      B:
        if(x == 1)
          next_state = B;
        else
          next_state = C;
```

Salida a cero

Estado A

Estado B

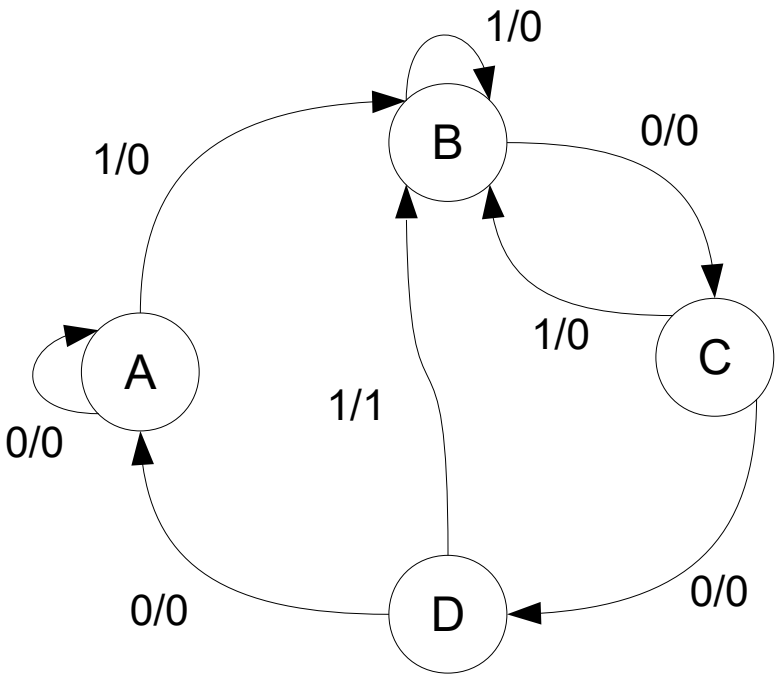


Máquinas de estado

```
        C:
          if(x == 1)
            next_state = B;
          else
            next_state = D;
        D:
          if(x == 1)
            begin
              z = 1;
              next_state = B;
            end
          else
            next_state = A;
      endcase
    end
endmodule
```

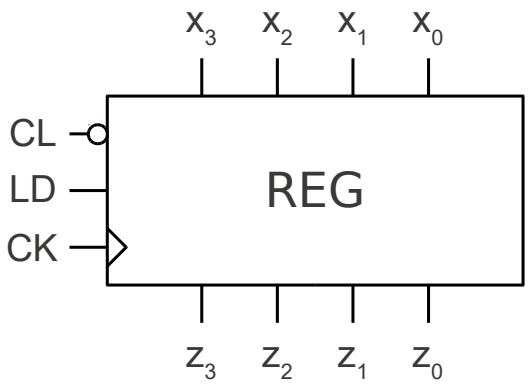
Estado C

Estado D



Registros

Registro con carga en paralelo y clear

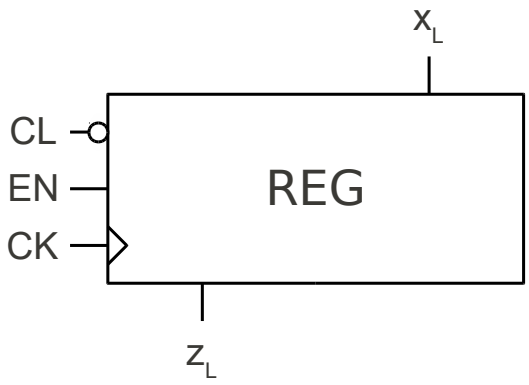


CL, LD	Operation	Type
0x	q ← 0	async.
11	q ← x	sync.
10	q ← q	sync.

```
module registro(
  input ck,
  input cl,
  input ld,
  input [3:0] x,
  output [3:0] z
  );

  reg [3:0] q;

  always @(posedge ck, negedge cl)
    if (cl == 0)
      q <= 0;
    else if (ld == 1)
      q <= x;

  assign z = q;

endmodule
```

Registros

Registro de desplazamiento



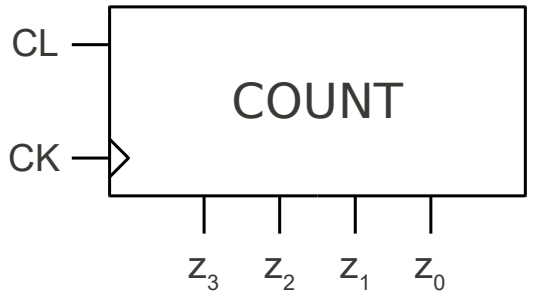
CL, EN	Operation	Type
0x	q ← 0	async.
11	q ← SHL(q)	sync.
10	q ← q	sync.

```
module reg_shl(
    input ck,
    input cl,
    input en,
    input xl,
    output zl
    );

    reg [3:0] q;

    always @(posedge ck, negedge cl)
        if (cl == 0)
            q <= 0;
        else if (en == 1)
            q <= {q[2:0], xl};

    assign zl = q[3];

endmodule
```

Lo correcto aquí es
q ← SHL(q,XL)

Contadores

Contador ascendente con clear

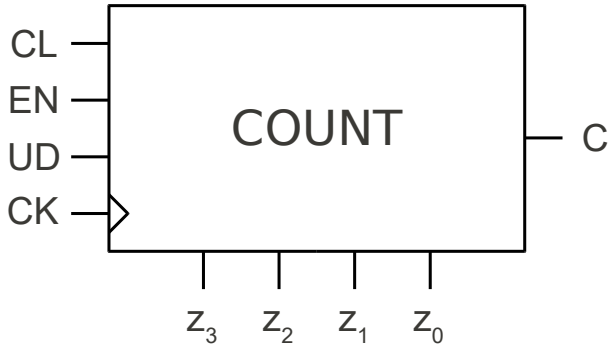


CL	Operation	Type
1	$q \leftarrow 0$	async.
0	$q \leftarrow q+1\|_{\text{mod } 16}$	sync.

```
module count_mod16(
  input ck,
  input cl,
  output [3:0] z);

  reg [3:0] q;

  always @(posedge ck, posedge cl)
    if (cl == 1)
        q <= 0;
    else
        q <= q + 1;

  assign z = q;

endmodule
```

Contadores

Contador ascendente/ descendente con clear



CL, EN, UD	Operation	Type
1xx	$q \leftarrow 0$	async.
00x	$q \leftarrow q$	sync.
010	$q \leftarrow q+1\|_{\text{mod } 16}$	sync.
011	$q \leftarrow q-1\|_{\text{mod } 16}$	sync.

```
module rev_counter1(
  input ck,
  input cl,en, ud,
  output [3:0] z, output c);

  reg [3:0] q;

  always @(posedge ck, posedge cl)
    begin
       if (cl == 1)
          q <= 0;
       else if (en == 1)
          if (ud == 0)
             q <= q + 1;
          else
             q <= q - 1;
    end

    assign z = q;
    assign c = ud ? ~(|q) : &q;
endmodule
```

Bloque III
Simulación y verificación

Bloque III: Índice

- Prueba y verificación de circuitos.
- Prueba y verificación de módulos Verilog.
- Estructura general del testbench.
- El testbench de un circuito combinacional.
  - Ejemplo: testbench de un decodificador 2:4
- El testbench de un circuito secuencial.
  - Ejemplo: testbench de un contador módulo 16.

Prueba y verificación de circuitos

- Introducción
  - Los circuitos reales ya implementados deben ser probados para verificar que su funcionamiento se ajusta a las especificaciones. Esto permitirá detectar problemas de en su funcionamiento.
  - Lo mismo ocurre con las descripciones Verilog de los circuitos. Pueden tener errores y es necesario detectarlos, antes incluso de pasar a la fase de implementación del circuito a partir de su descripción en Verilog.
  - Esto se consigue gracias a la simulación del funcionamiento y posterior verificación de los circuitos descritos en Verilog.

Prueba y verificación de circuitos

- ¿Cómo se prueba un circuito real?
  - Obviamente, no basta con conectarlo a una fuente de alimentación adecuada y observar su comportamiento.
  - Además será necesario darle diferentes valores a sus entradas e ir viendo si el valor de sus salidas es el correcto en cada momento.
  - Para ello se debe conectar el circuito a probar a otros equipos adicionales, que generaran señales especiales como la de reloj, o a circuitos auxiliares que generen señales de entrada de diversos valores.
  - Mediante el osciloscopio u otros equipos, el usuario verifica que las salidas obtenidas para las distintas entradas son las correctas.

Prueba y verificación de módulos Verilog

- ¿Cómo se prueba un módulo Verilog?
  - La descripción de un módulo en Verilog no es un circuito real, pero hay herramientas que simulan su funcionamiento.
  - Para probar un módulo Verilog no basta con introducirlo en una herramienta de simulación y darle al botón “simular funcionamiento”. Esto sería simplemente el equivalente a conectar un circuito real a una fuente de alimentación.
  - Debemos diseñar en Verilog un módulo especial llamado testbench y ese será el que le suministraremos al simulador para que simule su funcionamiento.
  - El modulo testbench contiene al modulo a probar y se encarga de suministrarle entradas con distintos valores para que luego el usuario pueda comprobar si las salidas del módulo son las correctas.

Estructura general del testbench

El testbench (o tb) es un módulo Verilog sin entradas ni salidas.

```
module micircuito_tb;

  reg tb_ent1, tb_ent2, … ;
  wire tb_sal1, tb_sal2, … ;

  micircuito instancia_de_micircuito(
    .ent1(tb_ent1), .ent2(tb_ent2), …,
    .sal1(tb_sal1), .sal2(tb_sal2), …);

 /* Sentencias procedimentales que
    modifican las variables tipo
    "reg" tb_ent1, tb_ent2, … */

endmodule
```

Debe tener tantas variables tipo reg como entradas tenga el circuito a probar.

Y tantas variables tipo wire como salidas tenga el circuito a probar.

El testbench contiene una instancia del módulo a probar, conectada a las variables reg y wire declaradas anteriormente.

Mediante sentencias procedimentales del tipo always y del tipo initial se hace que las entradas del módulo a probar vayan cambiando de valor.

El testbench de un circuito combinacional

- ¿Cómo se prueba un circuito combinacional?
  - Las salidas de un circuito combinacional dependen, en cada instante, exclusivamente, del valor actual de las entradas.
  - El testbench de un circuito combinacional lo único que debe hacer es generar todos los valores posibles que puedan tomar las entradas del circuito.
  - Si la suma del número de bits de todas las entradas del circuito es n, el testbench debe generar un total de 2^n valores diferentes.
  - Lo habitual es que los valores se generen mediante un bucle o algo similar, evitando que haya que generar "a mano" los valores.
  - Se puede usar la tarea del sistema $monitor para que nos muestre en pantalla los cambios que vayan experimentando las variables.

El testbench de un circuito combinacional

- Ejemplo: testbench de un decodificador 2:4

```
// Decodificador 2:4 con salidas en alto y habilitación en alto.

module dec_2_a_4(
    input EN,    // ENABLE
    input [1:0] A,
    output [3:0] Q
    );

    assign Q[0]= EN & ~A[1] & ~A[0];
    assign Q[1]= EN & ~A[1] &  A[0];
    assign Q[2]= EN &  A[1] & ~A[0];
    assign Q[3]= EN &  A[1] &  A[0];
endmodule
```

Este es el módulo combinacional que queremos probar.

Tiene dos entradas, una de 1 bit y otra de 2 bits. Habrá que generar $2^3 = 8$ valores de entrada diferentes, de 3 bits cada uno, para probar el circuito, pues es un circuito combinacional.

Ejemplo: testbench de DEC 2:4

Establecemos 1ns como unidad base para medir retrasos y tiempos de espera.

El testbench (o tb) es un módulo sin entradas ni salidas.

Declaramos las variables necesarias para conectarlas a continuación a las entradas y salidas de la instancia del circuito, teniendo cuidado de que sean del mismo tamaño.

Creamos la instancia del decodificador, conectando sus entradas a las variables reg y sus salidas a las variables wire que hemos declarado anteriormente.

```
`timescale 1ns / 1ps

module dec_2_a_4_tb;

  reg tb_EN;
  reg [1:0] tb_A;
  wire [3:0] tb_Q;

  dec_2_a_4 instancia_de_dec_2_a_4 (
    .EN(tb_EN),
    .A(tb_A),
    .Q(tb_Q)
  );
 /* Sigue en la página siguiente */
```

Ejemplo: testbench de DEC 2:4

Bloque initial. Se ejecuta una sola vez, al comienzo, en paralelo con el always.

Con $monitor se mostrarán en pantalla los valores de las variables especificadas, cada vez que alguna cambie de valor, indicando también del instante en que cambia.

```
/* Viene de la página anterior */

  initial begin
    $monitor("EN=%b A=%b Q=%b tiempo=%0dns",tb_EN,tb_A,tb_Q,$time);
    {tb_EN, tb_A[1:0]} = 3'b000;
    #800;
    $finish;
  end

  always begin
    #100; // Espera 100ns
    {tb_EN, tb_A[1:0]} = {tb_EN, tb_A[1:0]} + 3'b001;
  end
endmodule
```

Asignamos un valor inicial a las entradas, tratándolas como un único vector de 3 bits.

Esta sentencia introduce una espera de 800ns antes de pasar a la siguiente sentencia.

$finish detiene la simulación.

Bloque always. Repite "siempre" las mismas sentencias indefinidamente, en un bucle infinito.

Cada 100ns cambiamos el valor de las entradas de forma cómoda, considerándolas como un único vector de tres bits al que le sumamos una unidad.

Ejemplo: testbench de DEC 2:4

Estos son los mensajes que muestra en pantalla la herramienta de simulación cuando le pedimos que simule el funcionamiento del módulo de testbench.

```
Simulator is doing circuit initialization process.
Finished circuit initialization process.
EN=0 A=00 Q=0000 tiempo=0ns
EN=0 A=01 Q=0000 tiempo=100ns
EN=0 A=10 Q=0000 tiempo=200ns
EN=0 A=11 Q=0000 tiempo=300ns
EN=1 A=00 Q=0001 tiempo=400ns
EN=1 A=01 Q=0010 tiempo=500ns
EN=1 A=10 Q=0100 tiempo=600ns
EN=1 A=11 Q=1000 tiempo=700ns
Stopped at time : 800 ns
```

Estas 8 líneas salen en pantalla gracias a la sentencia $monitor, informándonos de los cambios en los valores de las variables de entrada (EN y A) y de salida (Q) del circuito a probar.

Nótese que el testbench cambia de valor las entradas cada 100ns.

Podemos verificar, analizando estas líneas con cuidado, que el DEC 2:4 se comporta correctamente.

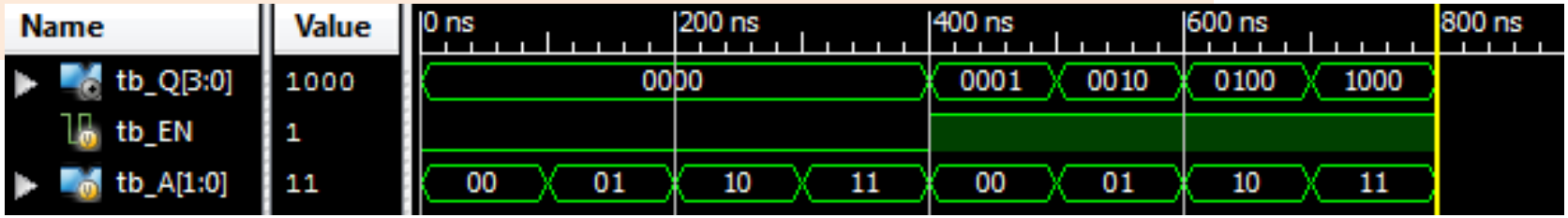
La simulación se ha detenido a los 800ns gracias a la sentencia de espera #800 que pusimos justo antes de la sentencia $finish.

Ejemplo: testbench de DEC 2:4

```
Simulator is doing c
Finished circuit ini
EN=0 A=00 Q=0000 tie
EN=0 A=01 Q=0000 tie
EN=0 A=10 Q=0000 tie
EN=0 A=11 Q=0000 tie
EN=1 A=00 Q=0001 tie
EN=1 A=01 Q=0010 tiempo=500ns
EN=1 A=10 Q=0100 tiempo=600ns
EN=1 A=11 Q=1000 tiempo=700ns
Stopped at time : 800 ns
```

Además de informar al usuario mediante información textual en pantalla, la herramienta de simulación también muestra las formas de onda de las variables declaradas en el módulo testbench, que son: tb_EN, tb_A y tb_Q.

Podemos verificar, analizando las formas de onda detenidamente, que el testbench genera todos los valores posibles de las entradas EN y A y que el valor de la salida Q del DEC 2:4 es correcto en todo momento.



El testbench de un circuito secuencial

- ¿Cómo se prueba un circuito secuencial?
  - Las salidas de un circuito secuencial dependen, en cada instante, del valor actual de las entradas y de su historia pasada (su estado).
  - Si se trata, como es habitual, de un circuito secuencial síncrono con una señal de reloj, el testbench debe generar dicha señal de reloj.
  - El testbench de un circuito secuencial es más complejo que el de un circuito combinacional. Para probar el circuito no basta con generar todos los valores posibles de las entradas.
  - En su lugar, hay que pensar bien qué tipo de circuito se está probando y el testbench se debe diseñar para que ponga “a prueba” las operaciones clave de dicho circuito, activando las señales de control correspondientes y colocando diversos valores en las entradas de datos.

El testbench de un circuito secuencial

- Ejemplo: testbench de un contador módulo 16

```
module contador_mod_16(
  input             CK,
  input             LD,
  input             EN,
  input       [3:0] X,
  output reg  [3:0] Q,
  output            CY
  );
    assign CY = &Q;
    always @(posedge CK)
      if (LD == 1)
        Q <= X;
      else if (EN == 1)
        Q <= Q + 4'b0001;
endmodule
```

Este es el módulo secuencial que queremos probar. Es un contador ascendente de 4 bits con carga síncrona en paralelo y señal de fin de cuenta (la salida CY).

Aquí vemos la entrada de la señal de reloj, CK, propia de un circuito secuencial.

La entrada de control LD (carga o LOAD) debe provocar la carga síncrona en el contador del valor presente en la entrada X.

La entrada de control EN (ENABLE) habilita la operación de cuenta ascendente.

El contador está sincronizado con los flancos de subida del reloj (posedge CK)

LD tiene prioridad sobre EN si se activan ambas. Si no se activa ninguna de las dos, la salida Q no cambia.

Ejemplo: testbench contador mód. 16

```
`timescale 1ns / 1ps

module contador_mod_16_tb;
  reg tb_CK;
  reg tb_LD;
  reg tb_EN;
  reg [3:0] tb_X;
  wire [3:0] tb_Q;
  wire tb_CY;
  contador_mod_16 instancia_de_contador_mod_16 (
      .CK(tb_CK),
      .LD(tb_LD),
      .EN(tb_EN),
      .X(tb_X),
      .Q(tb_Q),
      .CY(tb_CY)
  );
/* Sigue en la página siguiente */
```

El módulo de testbench de un circuito secuencial es diferente al de un circuito combinacional, pero hay cosas que se hacen siempre de igual forma: Se declara como un módulo sin entradas ni salidas.

Igual que siempre, se declaran las variables necesarias de tipo reg y de tipo wire.

Como siempre, se crea una instancia del módulo a probar (el contador), conectándola a las variables previamente declaradas.

Ejemplo: testbench contador mód. 16

```
/* Viene de la página anterior */

  always begin
    #50;
    tb_CK = ~tb_CK;
  end

  initial begin
    $monitor("CK=%b LD=%b EN=%b X=%b Q=%b CY=%b tiempo=%0dns",
             tb_CK,tb_LD,tb_EN,tb_X,tb_Q,tb_CY,$time);
    tb_CK = 0;
    tb_LD = 1;
    tb_EN = 0;
    tb_X = 4'b1100;

/* Sigue en la página siguiente */
```

La sentencia always repite indefinidamente, en bucle, las acciones de interior del bloque begin-end.

La variable tb_CK invierte su valor cada 50ns, generándose de esta forma una señal de reloj de período 100ns que se inyecta como entrada a la instancia del contador.

El bloque initial se ejecuta en paralelo con el always.

Al usar $monitor veremos en pantalla los cambios de valor de las variables.

Le damos a las entradas del contador los valores iniciales que creamos oportunos.

Ejemplo: testbench contador mód. 16

```
/* Viene de la página anterior */
  /* Sigue el bloque initial */
    @(negedge tb_CK);
    tb_X = 4'b0011;
    @(negedge tb_CK);
    tb_X = 4'b1101;
    tb_EN = 1;
    @(negedge tb_CK);
    tb_LD = 0;
    tb_X = 4'b1001;
    repeat ( 5 )
      @(negedge tb_CK);
    tb_EN = 0;
    @(negedge tb_CK);
    $finish;
  end /* del bloque initial */
endmodule
```

En el bloque initial, tras darle valores iniciales a las variables, iremos cambiando sus valores cada un cierto tiempo.

Esta sentencia se queda esperando un tiempo, hasta que tb_CK cambie de 1 a 0 (flanco de bajada).

Por tanto la variable tb_X tomará el valor 4'b0011 justo después del flanco de bajada de tb_CK.

Escribir repeat (5) delante de la sentencia @(negedge tb_CK); es más cómodo y equivale a escribir 5 veces seguidas la sentencia @(negedge tb_CK);

Tras haber cambiado las entradas del contador de la manera que hayamos estimado conveniente, ordenamos con $finish que finalice la simulación.

Ejemplo: testbench contador mód. 16

```
CK=0 LD=1 EN=0 X=1100 Q=xxxx CY=x tiempo=0ns
CK=1 LD=1 EN=0 X=1100 Q=1100 CY=0 tiempo=50ns
CK=0 LD=1 EN=0 X=0011 Q=1100 CY=0 tiempo=100ns
CK=1 LD=1 EN=0 X=0011 Q=0011 CY=0 tiempo=150ns
CK=0 LD=1 EN=1 X=1101 Q=0011 CY=0 tiempo=200ns
CK=1 LD=1 EN=1 X=1101 Q=1101 CY=0 tiempo=250ns
CK=0 LD=0 EN=1 X=1001 Q=1101 CY=0 tiempo=300ns
CK=1 LD=0 EN=1 X=1001 Q=1110 CY=0 tiempo=350ns
CK=0 LD=0 EN=1 X=1001 Q=1110 CY=0 tiempo=400ns
CK=1 LD=0 EN=1 X=1001 Q=1111 CY=1 tiempo=450ns
CK=0 LD=0 EN=1 X=1001 Q=1111 CY=1 tiempo=500ns
CK=1 LD=0 EN=1 X=1001 Q=0000 CY=0 tiempo=550ns
CK=0 LD=0 EN=1 X=1001 Q=0000 CY=0 tiempo=600ns
CK=1 LD=0 EN=1 X=1001 Q=0001 CY=0 tiempo=650ns
CK=0 LD=0 EN=1 X=1001 Q=0001 CY=0 tiempo=700ns
CK=1 LD=0 EN=1 X=1001 Q=0010 CY=0 tiempo=750ns
CK=0 LD=0 EN=0 X=1001 Q=0010 CY=0 tiempo=800ns
CK=1 LD=0 EN=0 X=1001 Q=0010 CY=0 tiempo=850ns
Stopped at time : 900 ns
```

Gracias a la sentencia $monitor, al simular el testbench se nos informa de los cambios en los valores de las variables de entrada (CK, LD, EN, y X) y de salida (Q y CY) del circuito a probar.

Podemos verificar, analizando el resultado de la simulación, si el circuito bajo prueba, el contador módulo 16, se comporta correctamente.

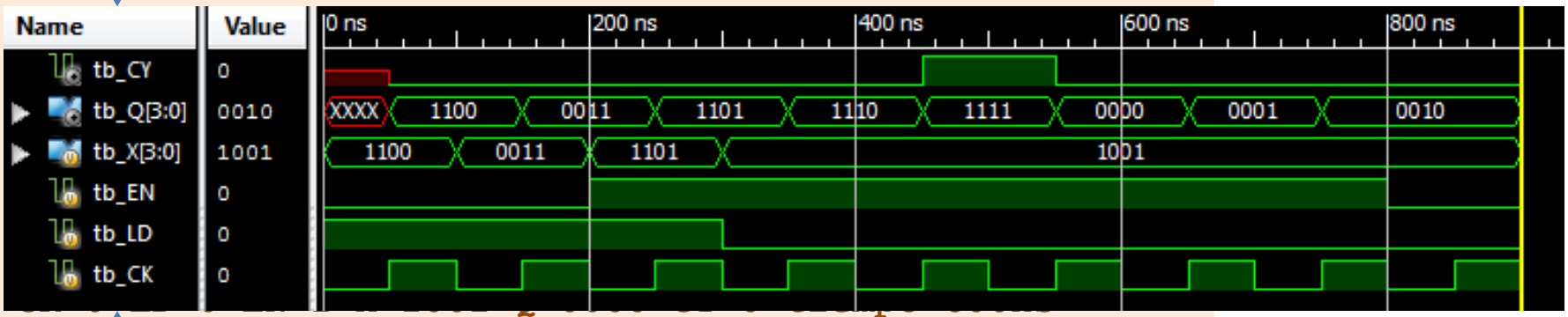
Por ejemplo: Si LD=0 y EN=0, aunque CK pasa de valer 0 a valer 1, vemos que la salida Q no cambia de valor (correcto).

Ejemplo: testbench contador mód. 16

```
CK=0 LD=1 EN=0 X=1100 Q=xxxx CY=x tiempo=0ns
```

Además de informar al usuario mediante información textual en pantalla, la herramienta de simulación también muestra las formas de onda de las variables declaradas en el módulo testbench, que son: tb_CK, tb_LD, tb_EN, tb_X, tb_Q y tb_CY.

```
CK=1 LD=1 EN=1 X=1101 Q=1101 CY=0 tiempo=250ns
CK=0 LD=0 EN=1 X=1001 Q=1101 CY=0 tiempo=300ns
```



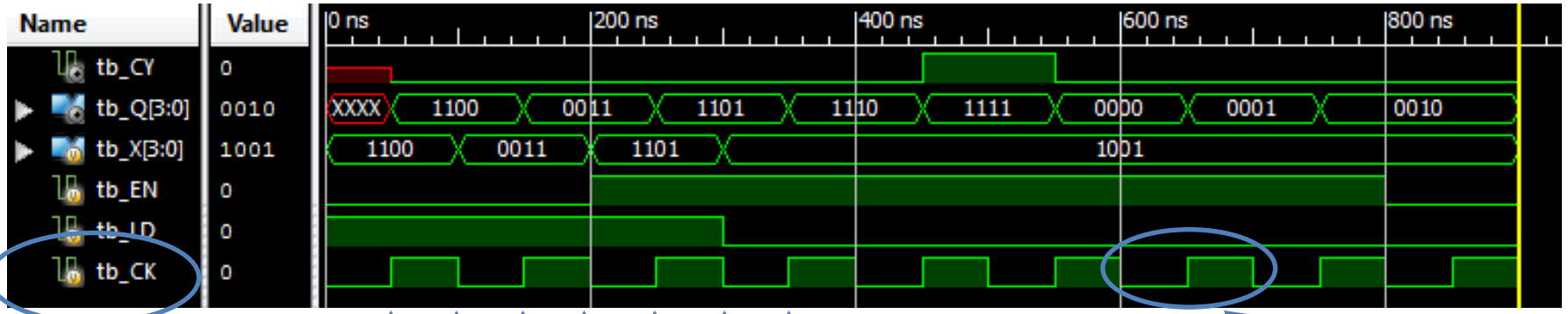

```
CK=1 LD=0 EN=1 X=1001 Q=0001 CY=0 tiempo=650ns
```

Podemos verificar, analizando las formas de onda detenidamente, que el testbench está cambiando los valores de las entradas del circuito bajo prueba según habíamos previsto y que las salidas de dicho circuito evolucionan también de forma correcta.

```
Stopped at time : 900 ns
```

Ejemplo: testbench contador mód. 16

La generación de una señal de reloj periódica es una tarea necesaria en los testbench de circuitos secuenciales.



Comprobamos que el testbench provoca un cambio en tb_CK cada 50ns, tal y como habíamos programado con la sentencia always.

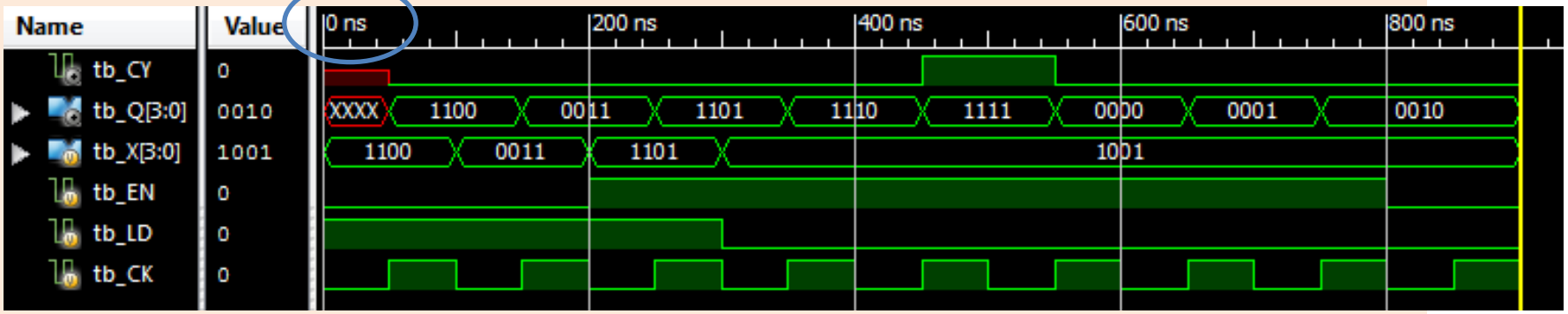
Un ciclo de reloj dura 100ns por lo que la frecuencia de reloj es de 10MHz

```
always begin
   #50;
   tb_CK = ~tb_CK;
end
```

Ejemplo: testbench contador mód. 16

```
initial begin
  $monitor("CK=%b LD=%b EN=%b X=%b Q=%b CY=%b tiempo=%0dns",
           tb_CK,tb_LD,tb_EN,tb_X,tb_Q,tb_CY,$time);
  tb_CK = 0;
  tb_LD = 1;
  tb_EN = 0;
  tb_X = 4'b1100;

  @(negedge tb_CK);
  tb_X = 4'b0011;
  @(negedge tb_CK);
  tb_X = 4'b1101;
```

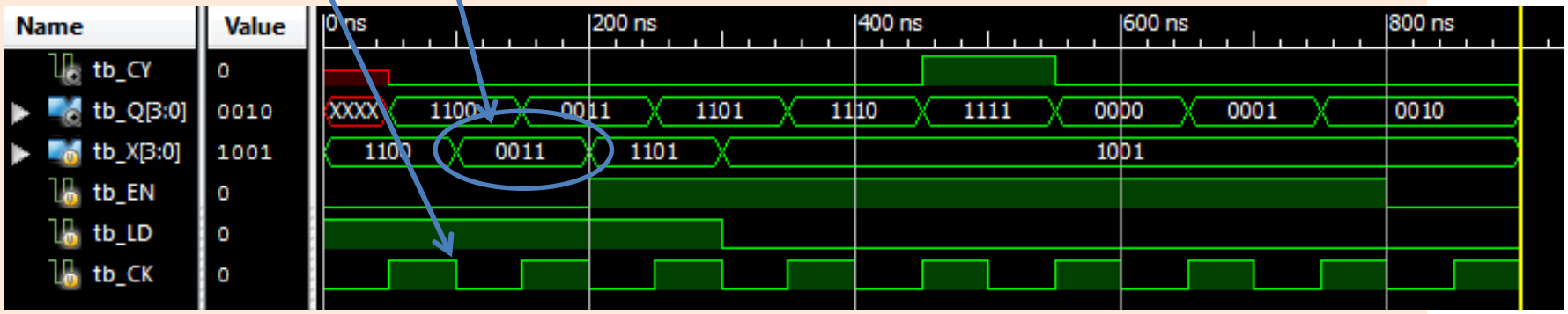
Comprobamos que los valores de las variables tb_CK, tb_LD, tb_EN y tb_X en el instante 0ns son los que programamos al principio del bloque initial del testbench.



Ejemplo: testbench contador mód. 16

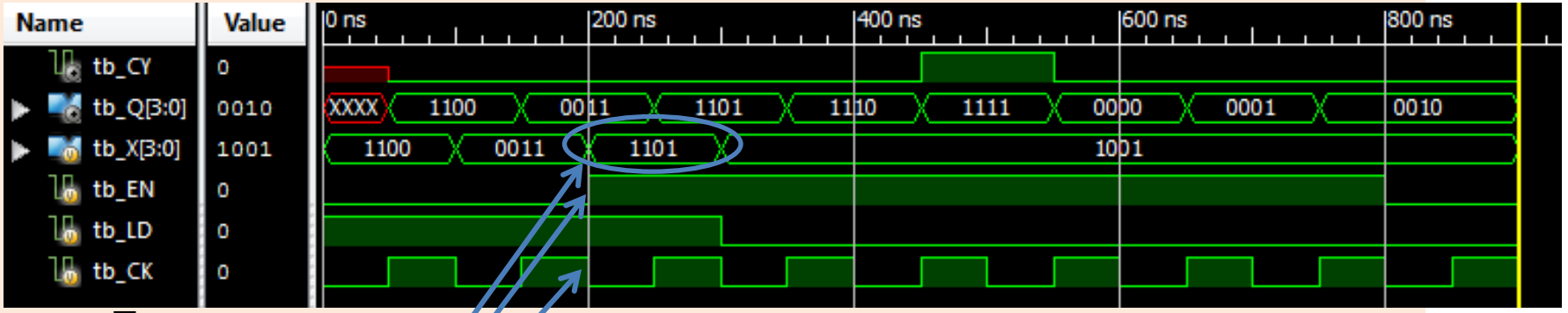
```
initial begin
  $monitor("CK=%b LD=%b EN=%b X=%b Q=%b CY=%b tiempo=%0dns",
           tb_CK,tb_LD,tb_EN,tb_X,tb_Q,tb_CY,$time);
  tb_CK = 0;
  tb_LD = 1;
  tb_EN = 0;
  tb_X = 4'b1100;

  @(negedge tb_CK);
  tb_X = 4'b0011;
  @(negedge tb_CK);
  tb_X = 4'b1101;
```

Comprobamos que, tal y como habíamos programado usando la sentencia @(negedge tb_CK), el testbench no pone la variable tb_X a valor 0011 (binario) hasta que no se produce un flanco de bajada en tb_CK.



Ejemplo: testbench contador mód. 16

```
initial begin
```




```
    @(negedge tb_CK);
    tb_X = 4'b0011;
    @(negedge tb_CK);
    tb_X = 4'b1101;
    tb_EN = 1;
    @(negedge tb_CK);
    tb_LD = 0;
    tb_X = 4'b1001;
```

Del mismo modo, comprobamos que una nueva sentencia @(negedge tb_CK), hace que el testbench espere un nuevo flanco de bajada en tb_CK antes de poner tb_X a valor 1101 (binario) y tb_EN a 1.

A continuación puede verse que hasta el tercer flanco de bajada del reloj no se modifican estas dos variables.

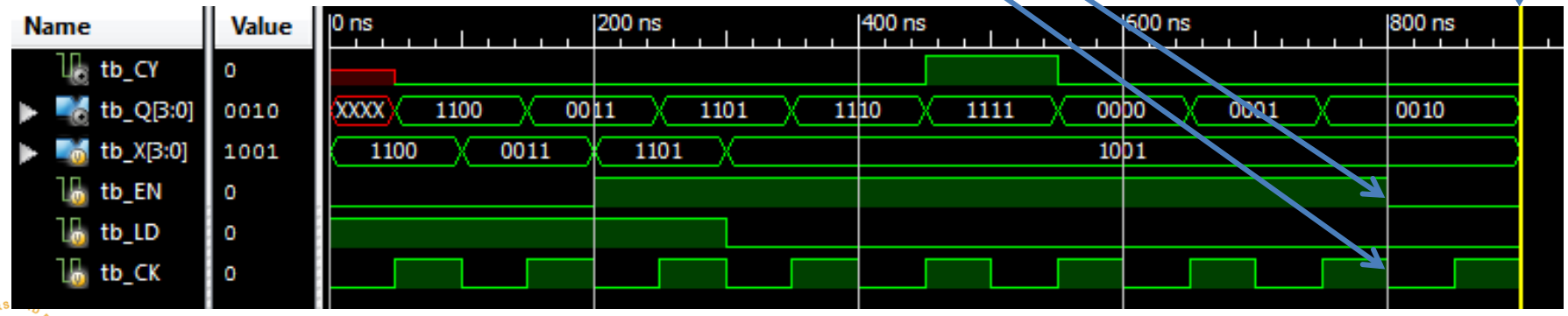
/* Sigue en la página siguiente */

Ejemplo: testbench contador mód. 16

```
/* Viene de la página anterior */
  /* Sigue el bloque initial
   repeat ( 5 )
       @(negedge tb_CK);
     tb_EN = 0;
     @(negedge tb_CK);
     $finish;
  end /* del bloque initial */
endmodule
```

Esperar 5 flancos de bajada de tb_CK cuando ya se han producido 3 flancos, quiere decir que hasta el octavo flanco de bajada (a los 800ms) no se pone a 0 el valor de tb_EN. Comprobamos que ha sido así, inspeccionando las formas de onda generadas.

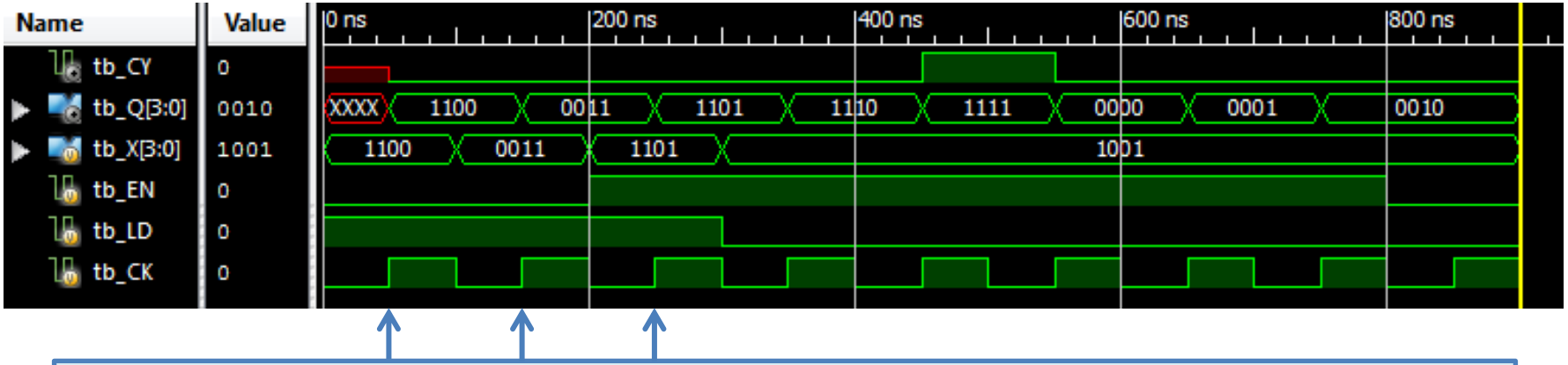
La simulación termina con $finish tras esperar un flanco de bajada más (el noveno) a los 900ns.



Ejemplo: testbench contador mód. 16

Aún no hemos comprobado si las formas de onda de salida son correctas.
Solo hemos comprobado que las entradas del contador cambian como habíamos planeado al diseñar el módulo de testbench.

Ahora habría que comprobar el correcto funcionamiento de las operaciones del contador (carga en paralelo, cuenta e inhibición).

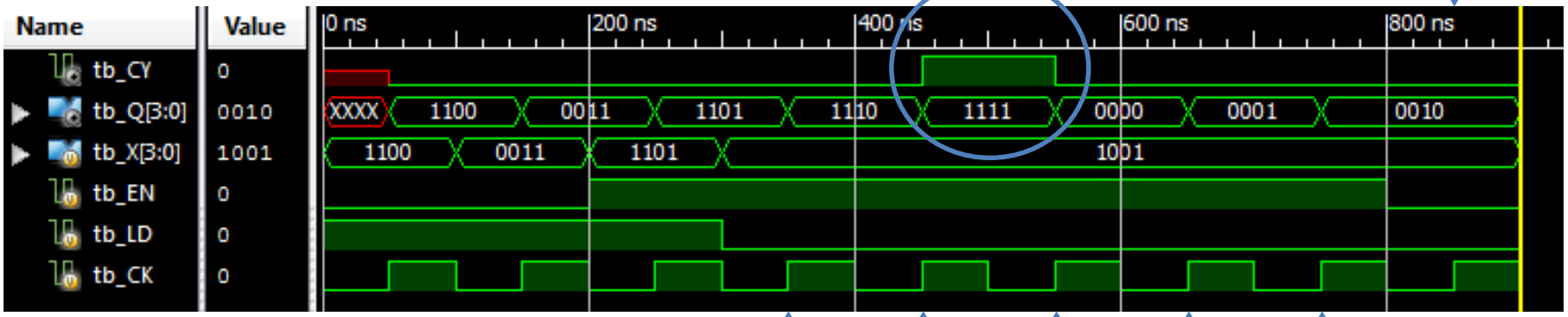


Por ejemplo, fijándonos en estos tres flancos de subida comprobamos que, independientemente del valor de la entrada EN, si al llegar el flanco de subida la entrada LD vale 1, el contador se carga con el dato de la entrada X y podemos ver dicho valor inmediatamente reflejado en la salida Q del contador.
Hemos verificado que la operación de carga en paralelo se realiza correctamente.

Ejemplo: testbench contador mód. 16

En el último flanco de subida mostrado en la simulación vemos que el valor del contador no cambia. Eso solo ocurre cuando las entradas LD y EN valen ambas 0.
Hemos verificado que la operación de inhibición se realiza correctamente

Verificamos que la salida CY también funciona correctamente, pues vale 1 solo si Q vale 1111.



Fijándonos en estos cinco flancos de subida comprobamos que si al llegar el flanco de subida la entrada LD vale 0 y la entrada EN vale 1 el contador incrementa en una unidad el valor almacenado, lo cual se ve reflejado inmediatamente en la salida Q. Nótese que, al ser módulo 16, tras el 15 (1111 binario) viene el 0000.
Hemos verificado que la operación de cuenta se realiza correctamente.

BLOQUE IV

Implementación

Bloque IV: Índice

Introducción

Definición de FPGA

Principales fabricantes y modelos de FPGA

Recursos internos de una FPGA

Estructura general de la FPGA modelo Virtex-II de Xilinx

Unidad básica de programación

Introducción

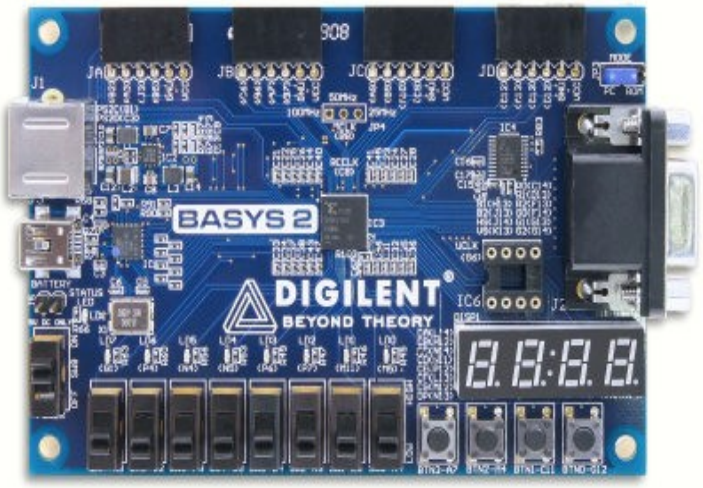
- El lenguaje verilog que hemos visto en las transparencias anteriores, permite la descripción de circuitos simples pero también de otros mucho más complejos cómo podría ser un microprocesador.
- Dichas descripciones tienen dos objetivos principales:
  - Simular el comportamiento del circuito.
  - Verificar que se cumplen las especificaciones deseadas.
  - Implementar el diseño.
- La implementación final puede llevarse a cabo mediante dispositivos programables o mediante ASIC.
- En esta asignatura se utilizará un dispositivo programable denominado FPGA para la realización de los diseños.
- Se dispone de herramientas de desarrollo que permiten realizar una implementación automática de un diseño descrito en verilog sobre una FPGA

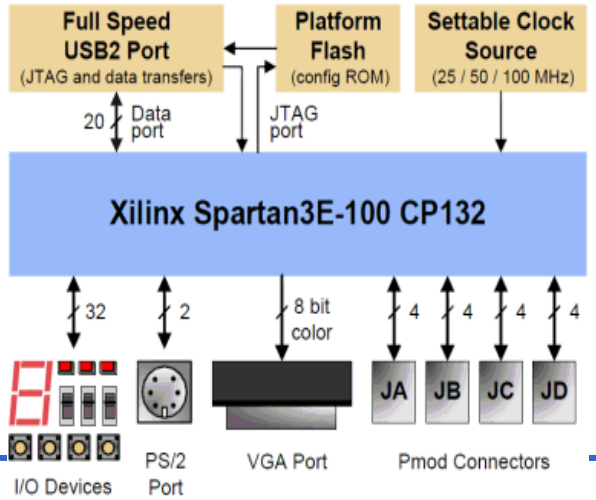
Definición de FPGA

- Una FPGA

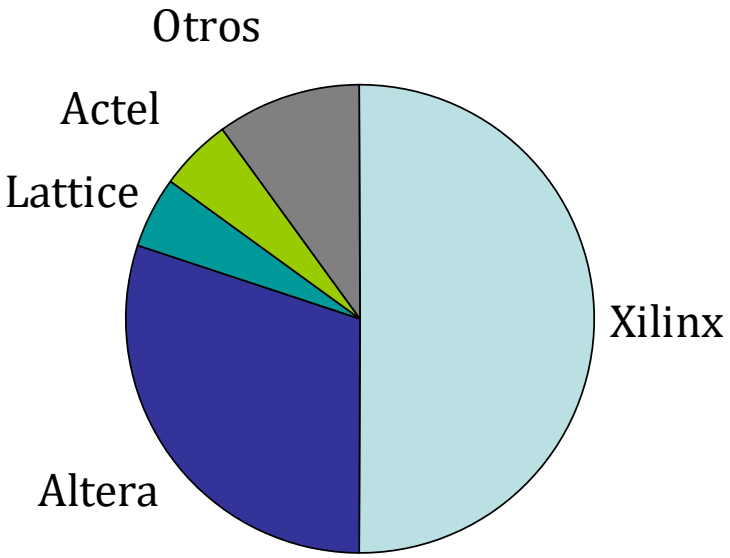
(*Field-Programmable Gate Array*) es un dispositivo programable:

  - –Permite implementar cualquier circuito digital.
  - –La única limitación es la cantidad de puertas del circuito.
  - –El desarrollo se realiza sobre una placa de entrenamiento.





Principales fabricantes y modelos de FPGA



Cuota de mercado

- Xilinx:
  - –Spartan
  - –Virtex
- Altera:
  - –Arria
  - –Cyclone
  - –Stratix
- Actel:
  - –Igloo
  - –ProASIC
  - –SmartFusion

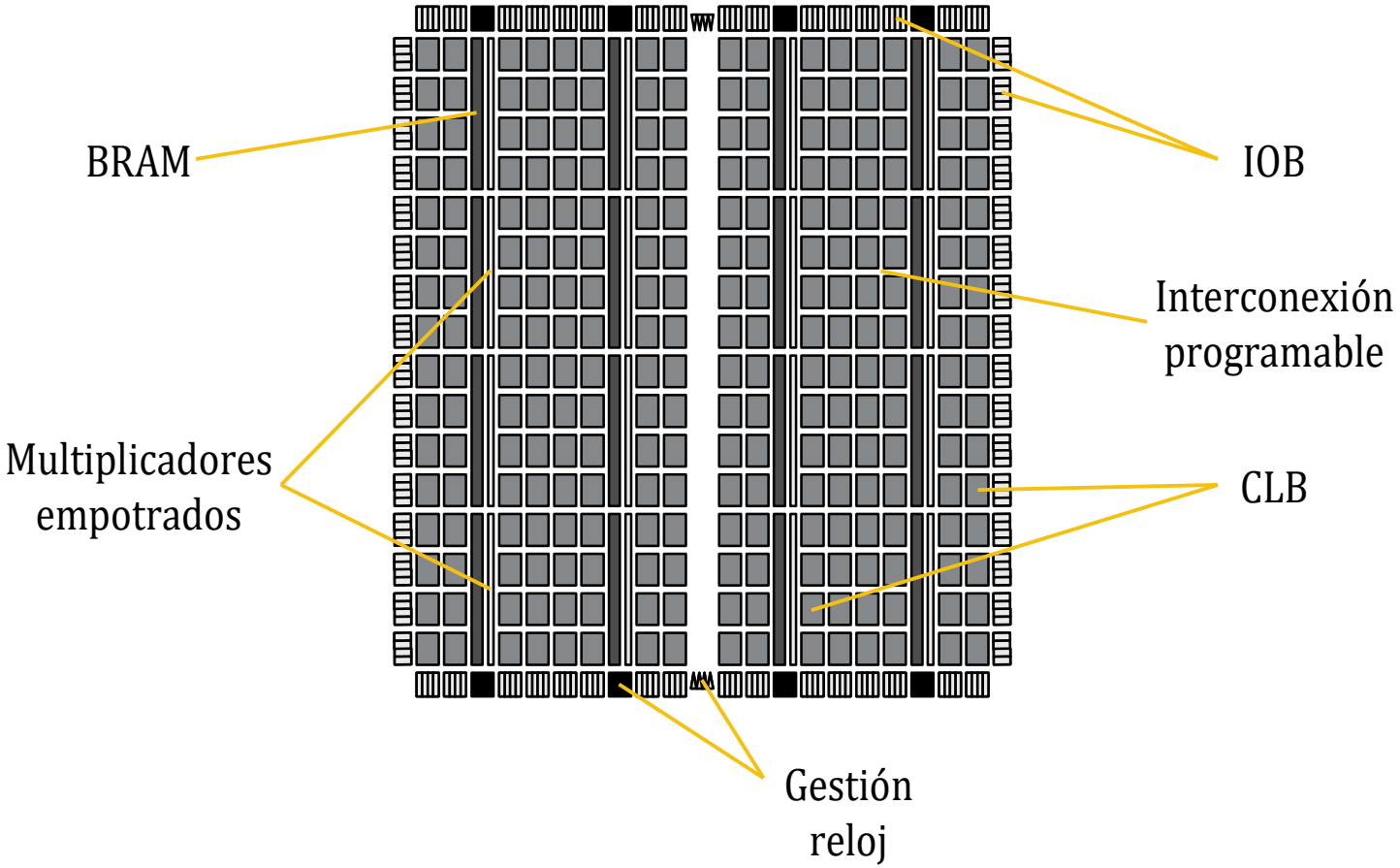
Recursos internos de una FPGA

En general, una FPGA contiene los siguientes recursos internos:

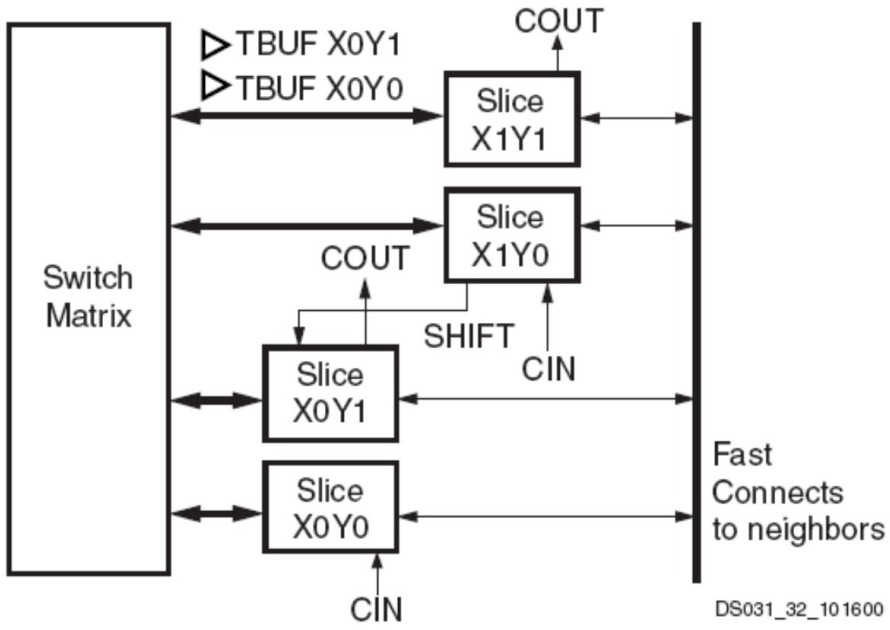
- Recursos lógicos:
  - *Slices*, agrupados en CLB (*Configurable Logic Blocks*).
  - Memoria BRAM (*Block* RAM).
  - Multiplicadores empotrados.
- Recursos de interconexión:
  - Interconexión programable.
  - Bloques de entrada/salida IOB (*Input/Output Blocks*).
- Otros recursos:
  - Búferes de reloj.
  - Lógica de escaneo de pines (*boundary scan logic*) normalmente mediante conexión JTAG (*Join Test Action Group*).

En las siguientes diapositivas se ilustra la estructura de la Virtex-II de Xilinx.

Estructura general de la FPGA modelo Virtex-II de Xilinx

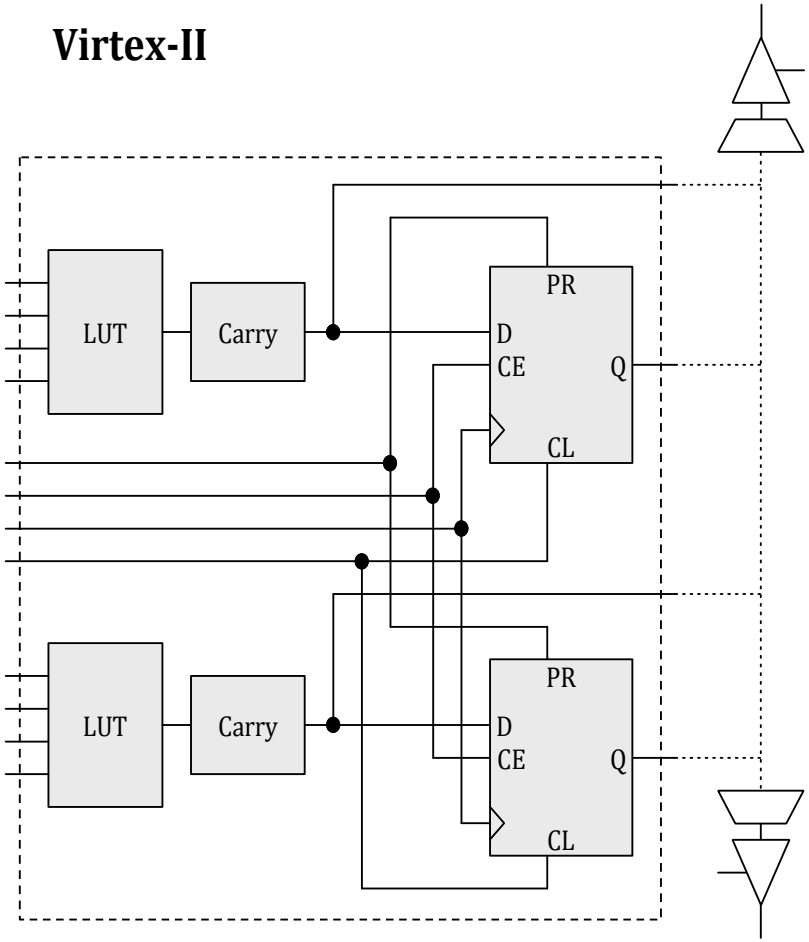


Estructura general de la FPGA modelo Virtex-II de Xilinx



- Cada *CLB* de la Virtex-II permite generar funciones combinacionales y secuenciales.
- Contiene:
  - 4 Slices
  - Conexionado hacia los CLB's vecinos
- Una matriz de conexión que permite su conexión con el resto de elementos de la FPGA.

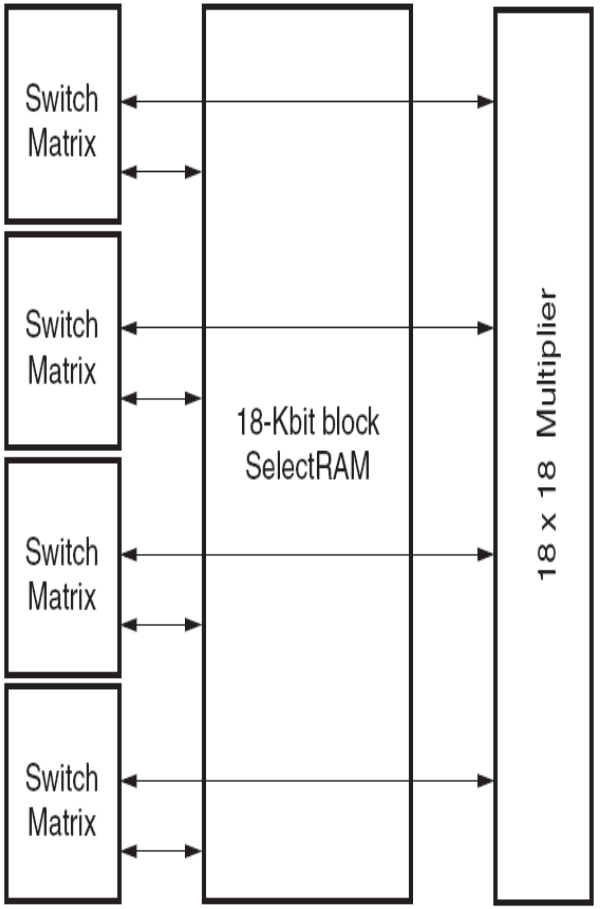
Unidad básica de programación: *slice*



- Cada *slice* incluye básicamente:
  - –2 biestables D.
  - –2 bloques de *carry*.
  - –2 bloques LUT (*Look-Up Table*).
  - – Multiplexores que permiten diferentes configuraciones
- Los bloques LUT
  - – Son programables (equivalen a una ROM 16x1) y pueden implementar cualquier función de 4 variables
  - – Las dos LUTs del mismo slice se pueden combinar para formar funciones combinacionales de más variables.
- Los biestables D
  - – Dispone de Pr y Cl (configurables en modo asíncronos o síncronos)
  - –Pueden trabajar en modo latch o en modo registro.

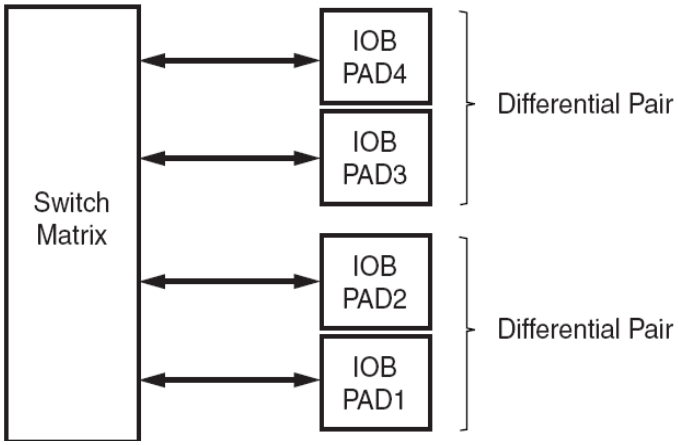
Unidad básica de programación: *BRAM y Multiplicador*

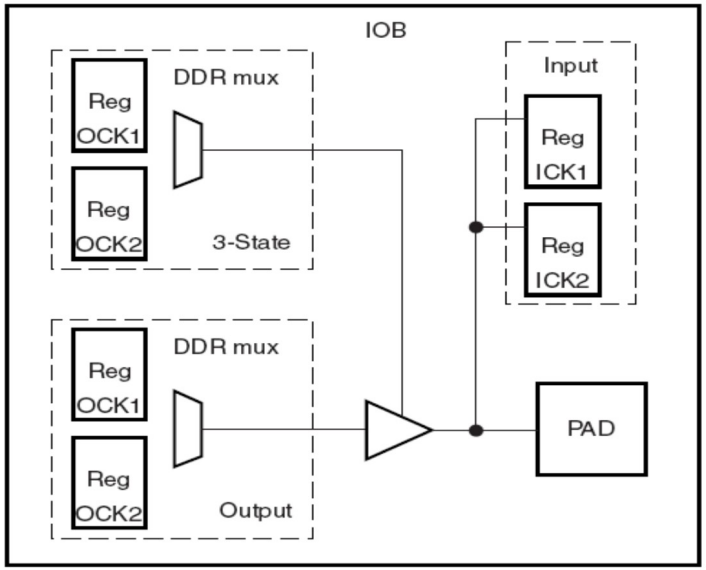
- BRAM 18Kx1 configurable como:
  - Memoria de uno o dos puertos
  - 16kx1, 8kx2, 1kx18,…
  - Conectada a la red general a través de 4 matrices de conexión.
- Multiplicador:
  - 18 x 18 bits
  - Complemento a 2.



Unidad básica de programación: *IOB*

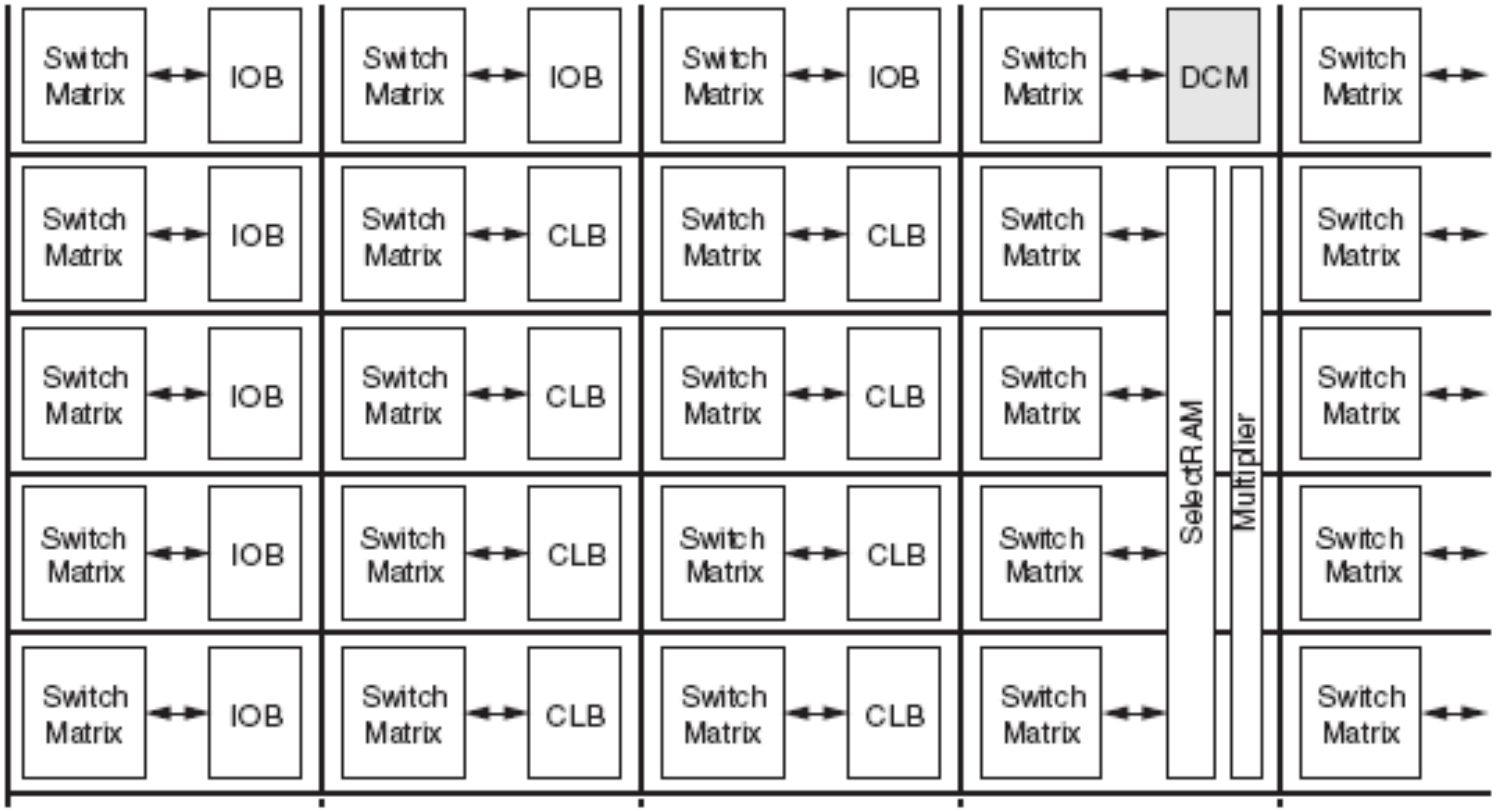
- IOB
  - – Admiten diferentes tipos de señales:
    - Diferencial (dos pads consecutivos)
    - Single-ended
  - – Tres partes:
    - Entrada
    - Salida
    - Control de salida (triestado)
  - – Cada parte tiene dos biestables configurables como latch o registro.
  - – Cada pin o pad puede configurarse como entrada, salida o bidireccional
  - – La impedancia (o resistencia) de salida se puede controlar digitalmente.
  - – Cada 2 o 4 pads tiene un matriz de conexión que los conecta al conjunto.
  - – Permite transferencias al doble de velocidad DDR.





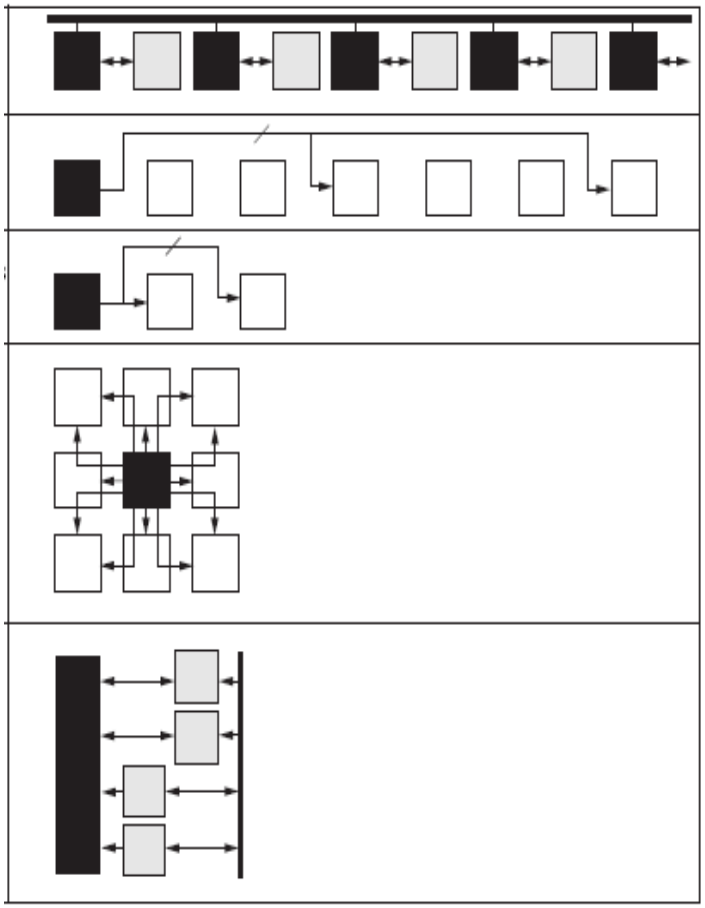
Unidad básica de programación: *Interconexión*

- La mayoría de las señales se envían por la red de líneas horizontales y verticales a la que los diferentes CLB, IOB, etc, tienen acceso a través de la matriz de interconexión programable.



Unidad básica de programación: *Interconexión*

- Long lines
  - Bidireccionales
  - 24 horizontales por cada fila y columna
  - Abarcan toda la FPGA
- Hex lines
  - Unidireccionales
  - 120 por cada fila y columna.
  - Conectan un bloque con su tercero o sexto
- Double lines
  - Unidireccionales
  - 40 por cada fila y columna
  - Conectan un bloque con su contiguo o al siguiente.
- Direct connect lines
  - Conectan un CLB con sus contiguos (incluyendo la diagonal).
  - 16 en total
- Fast connect lines
  - Internas al CLB, salidas de las LUTs a las entradas de otras LUTs



Unidad básica de programación:
Programación

- Una FPGA puede estar en dos estados: “Configuración” y “Operación”.
- Cuando la FPGA despierta después del encendido, se encuentra en modo configuración con todas sus salidas inactivas.
- La configuración requiere el envío de un patrón de bits con el mapa de conexiones internas por unos pines especiales que ésta posee (JTAG o “serial synchronous”).
- Una vez configurada, la FPGA pasa al modo operación, realizando la función lógica con la que se configuró.
- Hay varias formas de configurar la FPGA:

  - Directamente a través de un cable especial desde el PC.
  - Utilizando una boot-PROM en la placa donde está la FPGA y que contiene la inforación necesaria para que ésta se configure. La FPGA puede leer automáticamente la PROM tras su encendido.

Bibliografía y referencias

Verilog HDL Quick Reference Guide (Verilog-2001 standard)

http://sutherland-hdl.com/pdfs/verilog_2001_ref_guide.pdf

FPGA prototyping by verilog examples. Pong Chu. Wiley (Recurso-e de la biblioteca de la US)

Verilog Tutorial (1995): http://www.asic-world.com/verilog/veritut.html

Introduction to Verilog. Peter Nyasulu. (1995) URL: http://www.doe.carleton.ca/~jknight/97.478/PetervrlK.pdf