INTRODUCCIÓN A HDL VERILOG

Departamento de Tecnología Electrónica

Universidad de Sevilla

Evaluable en las prácticas de laboratorio

Rev.FPG (feb 2020)

Índice

Introducción

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

Bibliografía y referencias

- Online: Verilog-1995 Quick Reference Guide by Stuart Sutherland of Sutherland HDL, Inc., Portland, Oregon, USA at

  http://www.sutherland-hdl.com/online_verilog_ref_guide/vlog_ref_top.html

- Verilog Tutorial:

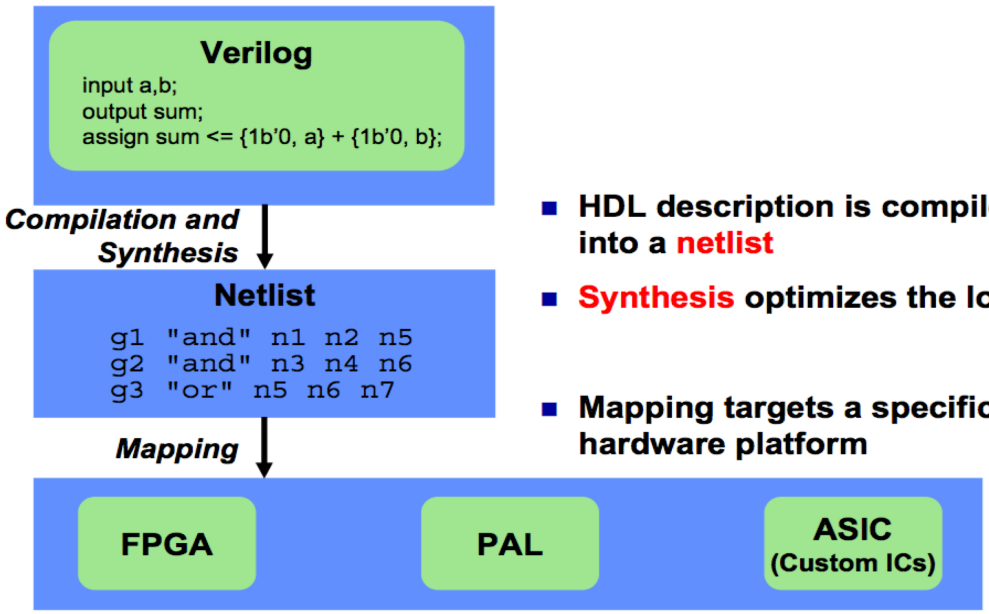
  http://www.asic-world.com/verilog/veritut.html

- Verilog HDL Quick Reference Guide (Verilog-2001 standard)

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

Synthesis and HDLs

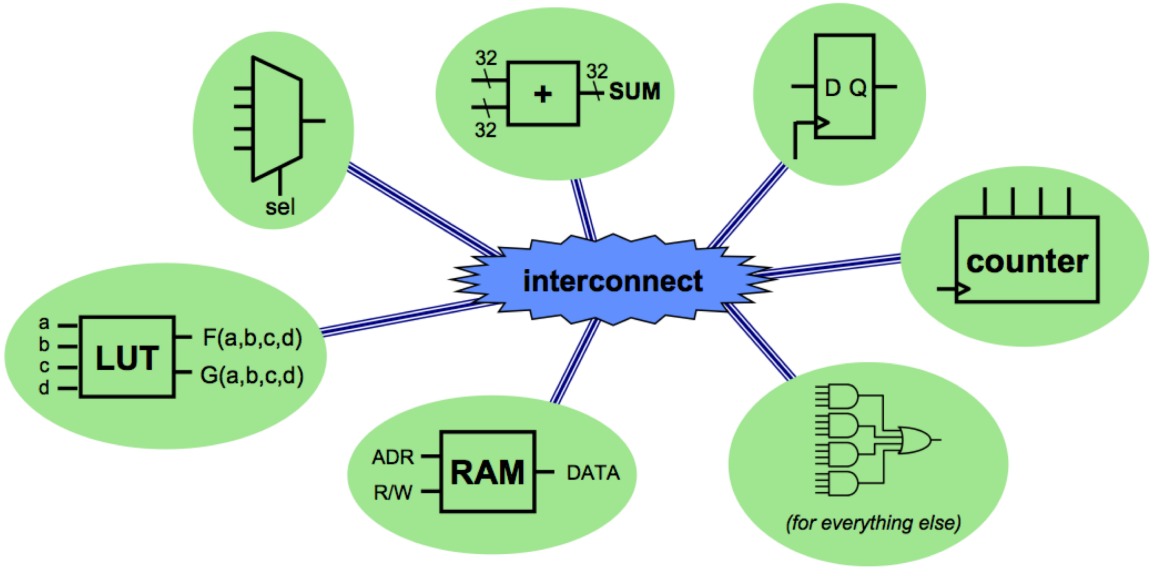
- Hardware description language (HDL) is a convenient, device-independent representation of digital logic



- HDL description is compiled into a netlist
- Synthesis optimizes the logic
- Mapping targets a specific hardware platform

The FPGA: A Conceptual View

- An FPGA is like an electronic breadboard that is wired together by an automated synthesis tool
- Built-in components are called macros



BLOQUE I

Diseño de Circuitos Combinacionales

Bloque I:
(Circuitos Combinacionales)

1. Estructura general de una descripción Verilog
2. Tipos de descripciones
3. Señales, puertos E/S y arrays
4. Sintaxis básica

Estructura de descripciones Verilog

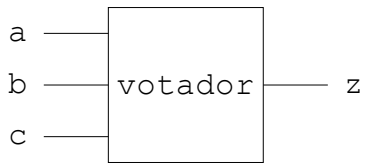
```
module mi_circuito (
  input x, y,
  input z,
  output f1, f2
  );

  wire cable_interno;
  reg  variable_a

  ...
  ...
  ...

endmodule
```

Declaración del módulo con sus entradas y salidas

Declaración de señales y variables que se utilizarán internamente en la descripción

Descripción del comportamiento del módulo.
Hay varias alternativas para realizarla

Ejemplo: circuito votador



- Expresión lógica:
  - z=ab+ac+bc

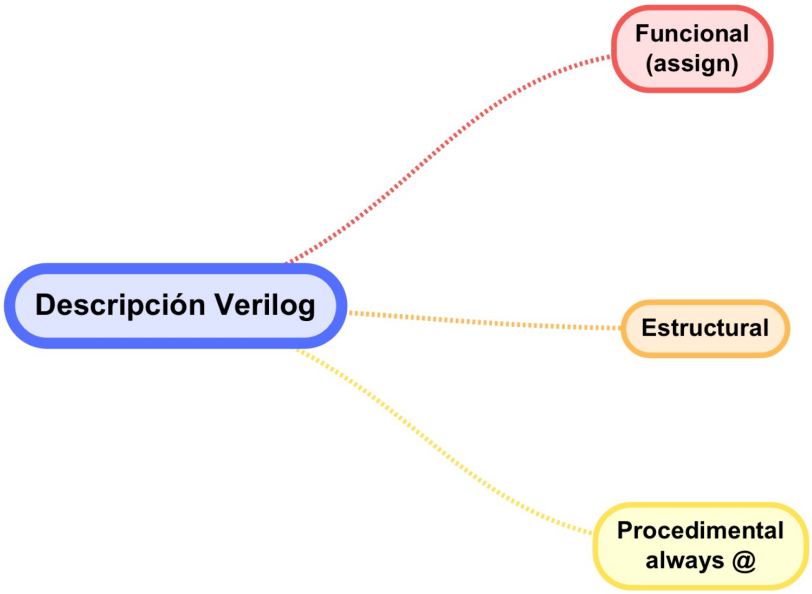
```
module votador (input a,b,c,output z);

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

endmodule
```

OR: | (alt gr + 1)

Bloque I:
(Circuitos Combinacionales)

1. Estructura general de una descripción Verilog
2. Tipos de descripciones
3. Señales, puertos E/S y arrays
4. Sintaxis básica

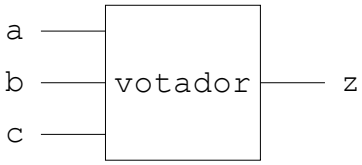
Tipos de descripciones



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.

```
/* Ejemplo de descripción funcional */
module votador(input a,b,c, output z);
  assign z = a&b | a&c | b&c;
endmodule
```



Operadores a nivel de bits (bit-wise)

Operador	Ejemplo de código verilog
&	c = a&b; //AND
|	c = a|b; //OR
^	c = a^b; //XOR
~	c = ~a; // NOT
~&	c = a ~& b; //NAND
~|	c = a ~| b; //NOR
~^	c = a ~^ b; //EXNOR

| (altgr+1)

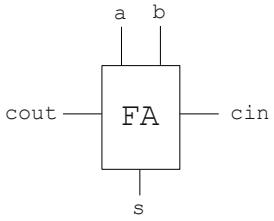
~ (altgr+4; esp)

- Estos operadores trabajan con todos los bits de a y de b

- Si las variables son de un bit, operan como los operadores del álgebra de conmutación

Ejemplo:
Descripción funcional de un Full-Adder de 1 bit

```
/* Descripción funcional de un sumador completo
de un bit */

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

/* todas las sentencias assing se ejecutan
Concurrentemente */
    assign  s = a ^ b ^ cin;
    assign  cout = a & b | a & cin | b & cin;
endmodule
```



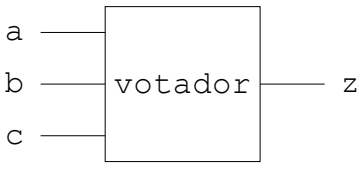
cin	a	b	cout	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$s = a \oplus b \oplus c$

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

Descripción procedimental

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

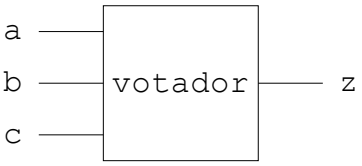


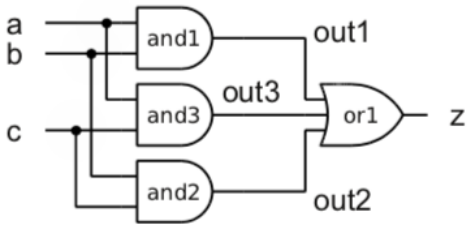

```
module votador(
  input a,b,c,
  output reg z);

  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
```

Descripción estructural

- Se conectan módulos que ya están definidos previamente
- Las puertas lógicas básicas ya están predefinidas en Verilog:

  and, or, nand, nor, xor, xnor, not
- Es muy útil para la interconexión de los módulos que se creen



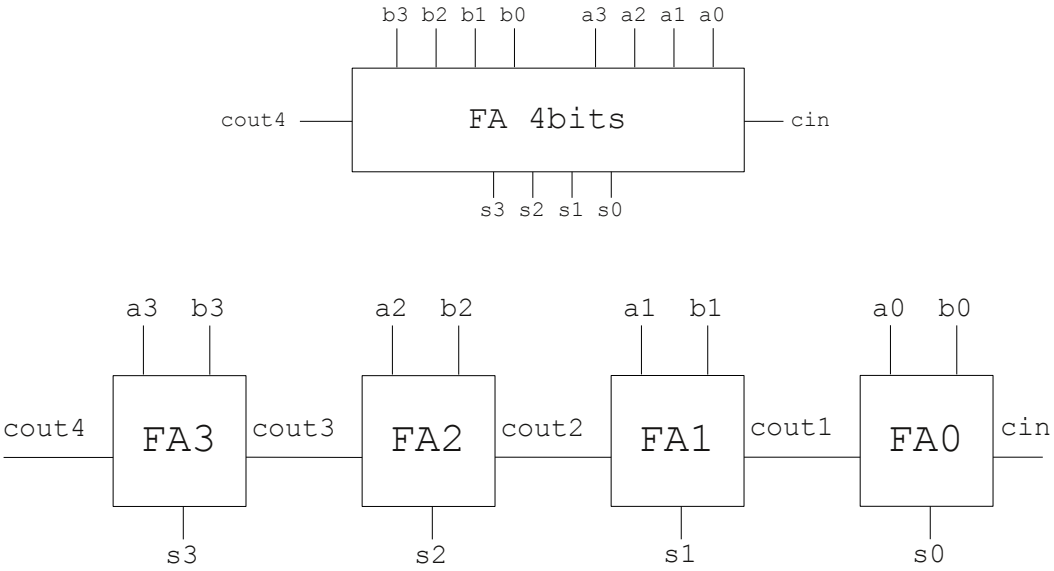



```
module votador(
  input a,b,c,
  output z);

  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
```


¿Qué tipo de descripción es mejor?

- La descripción funcional (assign) es útil para funciones combinacionales sencillas
- La descripción procedimental (always) es muy potente y permite describir circuitos combinacionales complejos y circuitos secuenciales; se describe el comportamiento observable del sistema
- 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.

Ejemplo:
Descripción de un FULL-ADDER de 4 bits a partir de varios FA de 1 bit

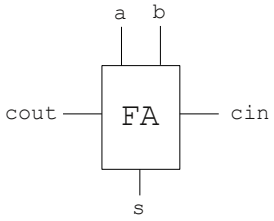


Ejemplo: FULL-ADDER de 4 bits (cont.)

- Pasos:
  - Descripción de un módulo para el FULL-ADDER de un bit (ya se ha hecho anteriormente).
  - Descripción de un módulo donde se utilizan 4 FULL-ADDER de un bit y se interconectan los cables de los módulos.

Descripción funcional de un Full-Adder de 1 bit

```
/* Descripción funcional de un sumador completo
de un bit */

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

/* todas las sentencias assing se ejecutan
Concurrentemente */
    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

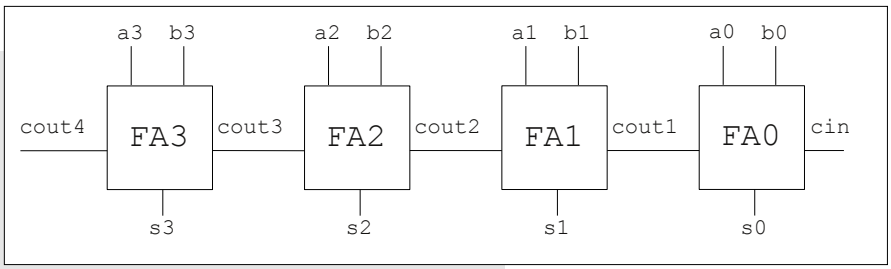
$s = a \oplus b \oplus c$

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

Ejemplo: FULL-ADDER de 4 bits (cont.)

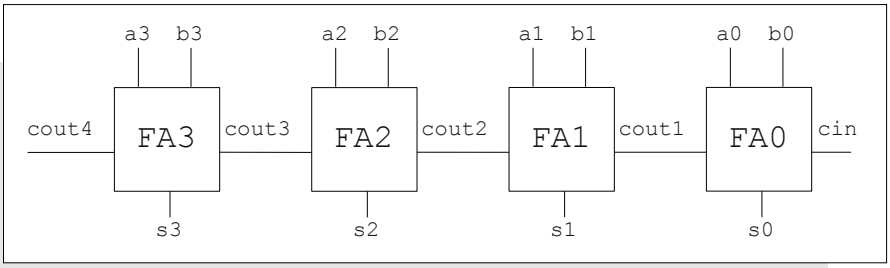
Unión de 4 FULL-ADDER: conexión posicional.

```
module fulladder4(
  input [3:0] a,
  input [3:0] b,
  input cin,
  output [3:0] s,
  output cout4);

/*doy nombre a los cables que unen los módulos*/
  wire cout1,cout2,cout3;
/*usamos ahora el módulo tipo fulladder ya definido*/
  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
```



Ejemplo: FULL-ADDER de 4 bits (cont.)

Unión de 4 FULL-ADDER: conexión nombrada




```
module fulladder4(
  input [3:0] a,
  input [3:0] b,
  input cin,
  output [3:0] s,
  output cout4);

/* doy nombre a los "cables" que unen los módulos*/
  wire cout1,cout2,cout3;

/*uso el módulo tipo fulladder que ya se había definido */
  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
```


Bloque I: Índice

Tipos de señales: wire y reg

- 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
- Todas las asignaciones que se realicen dentro de un procedimiento (always) deben ser sobre una señal tipo reg

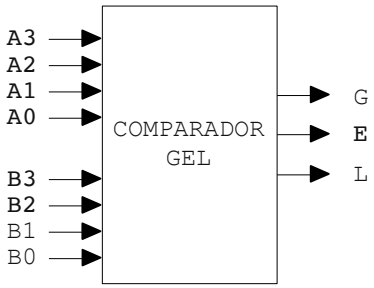
Puertos de entrada/salida

- Cuando se declaran módulos se puede especificar si un puerto es tipo wire o reg
  - Si no se indica nada es por defecto wire
  - Los cables (wire) son utilizados con la sentencia assign
  - Los registro (reg) son asignados en los procedimientos always

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

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
```


Bloque I: Índice

1. Estructura general de una descripción Verilog
2. Tipos de descripciones
3. Señales, puertos E/S y arrays
4. Sintaxis básica

Sintaxis básica

- Algunas consideraciones
- 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
```

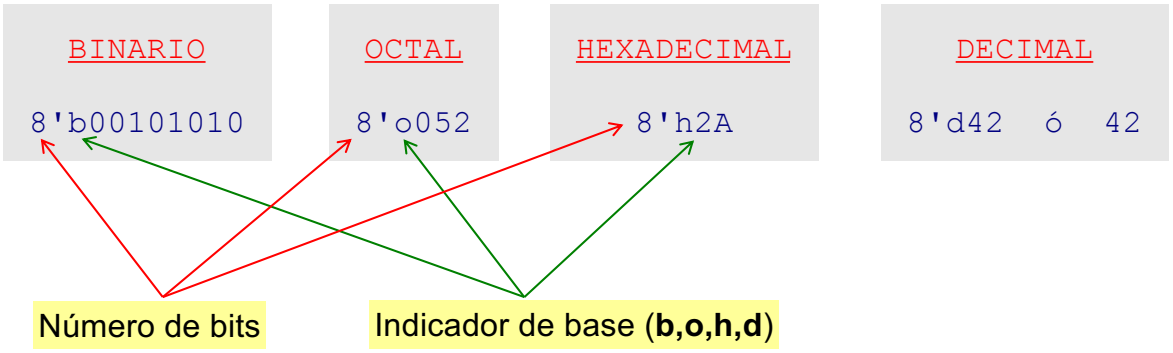
  - Comentario de varias líneas: comienza con /* y termina con */

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

wire a;
```

Sintaxis básica: números en Verilog

- Las constantes numéricas (literales) pueden expresarse en binario, octal, hexadecimal o decimal

Ejemplo: $00101010_{(2}$



Sintaxis básica: números en Verilog (ii)

- Ejemplo de literales: 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
```

Sintaxis básica: sentencia assign

- Todas las sentencias assign se ejecutan de manera concurrente
- En el ejemplo 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
```


Sintaxis básica: 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 sólo si cambia alguna de las señales de la lista de sensibilidad.
  - La sintaxis es:

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

Sintaxis básica: sentencia always (ii)

- 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
```


Sintaxis básica: sentencia always (iii)

- Siempre que se esté describiendo un componente combinacional, se debe incluir en la lista de sensibilidad todas las entradas del componente
- 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
```

Sintaxis básica: Operadores a nivel de bits (bit-wise)

Operador	Ejemplo de código verilog
&	c = a&b; //Operador AND de todos los bits
|	c = a|b; //Operador OR de todos los bits
^	c = a^b; //Operador XOR de todos los bits
~	c = ~a; // Inversión de todos los bits
~&	c = a ~& b; //Operador NAND de todos los bits
~|	c = a ~| b; //Operador NOR de todos los bits
~^	c = a ~^ b; //Operador EXNOR de todos los bits

- Estos operadores trabajan con todos los bits de a y de b

- Si las variables son de un bit, operan como los operadores del álgebra de conmutación

Ejemplo de uso de Operadores a nivel de bits (bit-wise)

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

```
/*Este módulo calcula 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
```

Sintaxis básica: Operadores relacionales

- Devuelven 1 si es verdadera 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`

Sintaxis básica: Operadores lógicos

(No confundidlos 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`

Sintaxis básica: 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`

Sintaxis básica: 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; // Concatenación: // Asigna una palabra a bits // individuales: a=1, b=0 y c=1`

Verilog dispone de más que se pueden encontrar en la bibliografía

Ejemplo: sumador completo de 32 bits

```
module fulladder32(
  input [31:0] a,
  input [31:0] b,
  input cin,
  output [31:0] sum,
  output cout);

/*utilizo el operador aritmético suma*/
  assign {cout, sum} = a + b + cin;
endmodule
```


Sentencias condicionales (i)

- 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

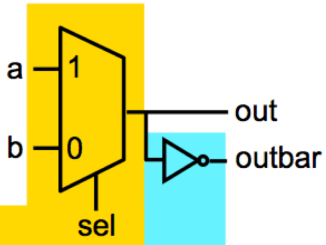
Mix-and-Match Assignments

- Procedural and continuous assignments can (and often do) co-exist within a module
- Procedural assignments update the value of `reg`. The value will remain unchanged till another procedural assignment updates the variable. This is the main difference with continuous assignments in which the right hand expression is constantly placed on the left-side

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input a, b, sel;
  output out, outbar;
  reg out;

  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end

  assign outbar = ~out;

endmodule
```

procedural description

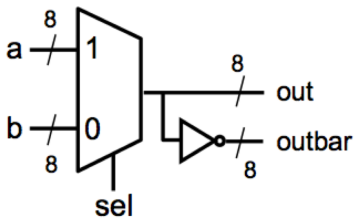
continuous description



The Power of Verilog: *n*-bit Signals

- Multi-bit signals and buses are *easy* in Verilog.
- 2-to-1 multiplexer with *8-bit operands*:

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input[7:0] a, b;
  input sel;
  output[7:0] out, outbar;
  reg[7:0] out;

  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end

  assign outbar = ~out;

endmodule
```



Sentencias condicionales (ii)

- 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

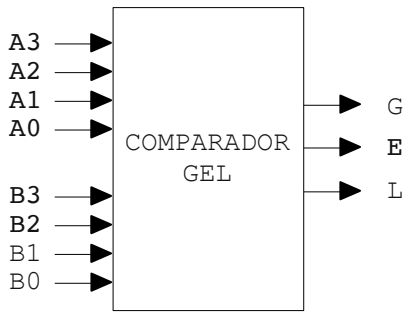
```
always @(a)
 begin
  if ( a > 0 )
   begin
    f1 = 1;
    f2 = 1;
   end
  else
   f1 = 0;
 end
```

Correcto

Sentencias condicionales (iii)
Ejemplo de uso de if... else

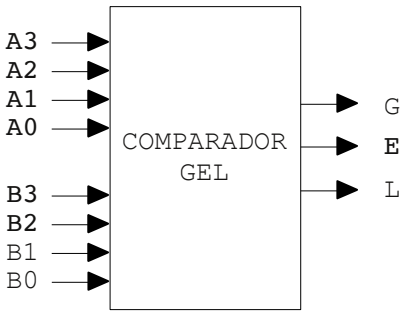
Ejemplo:

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);

//  reg g, e, 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
```

Otro ejemplo de uso de if... else




```
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);

//  reg g, e, l;
   always @(a, b)
     begin
       if (a > b)
          {g,e,l} =3'b100;
       else
                              if (a < b)
                              {g,e,l} =3'b001;
                     else
                              {g,e,l} =3'b010;
      end
endmodule
```

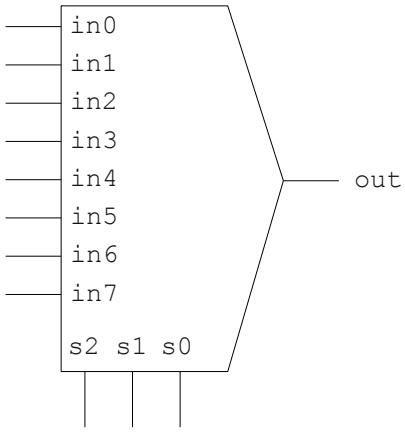

Sentencias condicionales (iv)

- 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 (v)
Ejemplo de uso de case

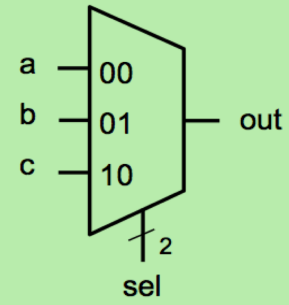
- Multiplexor 8:1
  - Ejemplo de acceso a elementos individuales de un array




```
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
```


Dangers of Verilog: Incomplete Specification

Goal:



3-to-1 MUX
('11' input is a don't-care)

Proposed Verilog Code:

```
module maybe_mux_3to1(a, b, c,
                      sel, out);
  input [1:0] sel;
  input a,b,c;
  output out;
  reg out;

  always @(a or b or c or sel)
  begin
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```

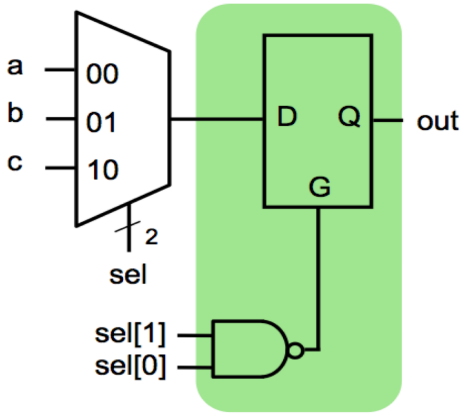
Is this a 3-to-1 multiplexer?

Incomplete Specification Infers Latches

```
module maybe_mux_3to1(a, b, c,
                      sel, out);
  input [1:0] sel;
  input a,b,c;
  output out;
  reg out;

  always @(a or b or c or sel)
  begin
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```

if `out` is not assigned during any pass through the always block, then the previous value must be retained!

Synthesized Result:



- Latch memory "latches" old data when G=0 (we will discuss latches later)
- In practice, we almost *never* intend this

BLOQUE II

Diseño de Circuitos Secuenciales

Bloque II: Índice

1. Sintaxis II
2. Biestables
3. Máquinas de estados
4. Registros
5. Contadores

Sintaxis II

- Definición de constantes
- Operador de concatenación
- Lista de sensibilidad con detección de flancos
- Asignaciones bloqueantes / no bloqueantes

Sintaxis II: Definición de constantes

- Dentro de un módulo se pueden definir constantes utilizando parameter
- Es útil en la definición de máquinas de estados (asignación de estados)
- Ejemplo:

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

reg [2:0] a;

a = ultimo;
```


Sintaxis II: Operador de concatenación {señal, señal, ...}

- 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: Lista de sensibilidad con detección de flancos posedge, negedge

- 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

Ejemplo de detección de flanco de bajada (o negativo) de la señal clk

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

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

Sintaxis II: Asignación bloqueante signo =

- Si en un proceso always se desea que la salida cambie inmediatamente, se debe utilizar una asignación bloqueante.
- Modelan, sobretodo, salidas combinacionales.
- Importa el orden en que se efectúan las asignaciones bloqueantes puesto que las acciones en un proceso se ejecutan secuencialmente, una detrás de otra, en el orden en que aparecen

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

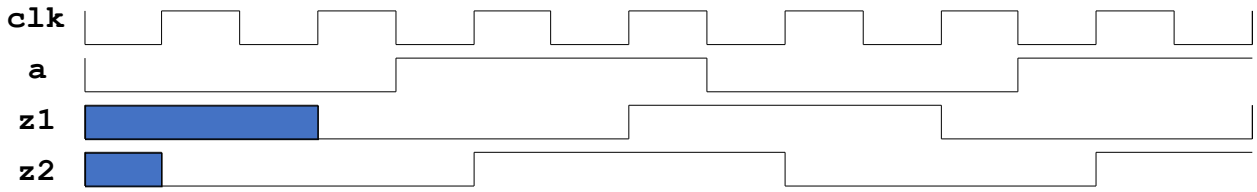

Sintaxis II: Asignación No bloqueante <=

- Modela los cambios de estados en flip-flops.
- Define "el próximo estado"
- Se calculan primero los valores de la derecha de la asignación de todas las asignaciones <= ; tras esto, se asignan todas simultáneamente.
- Cuando se tienen varias asignaciones no bloqueantes, no importa el orden en que son escritas.

Comparación entre asignación no bloqueante y asignación bloqueante

```
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
```

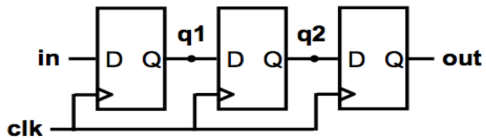


Comparación entre asignación no bloqueante y asignación bloqueante

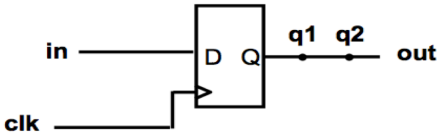
```
module no_bloqueante(input in,clk,
  output reg out);
  reg q1, q2;
  always @(posedge clk)
    begin
                q1  <= in;
                q2  <= q1;
                out <= q2;
    end
endmodule
```

```
module bloqueante(input in,clk,
  output reg out);
  reg q1, q2;
  always @(posedge clk)
    begin
                q1  = in;
                q2  = q1;
                out = q2;
    end
endmodule
```

"At each rising clock edge, *q1*, *q2*, and *out* simultaneously receive the old values of *in*, *q1*, and *q2*."

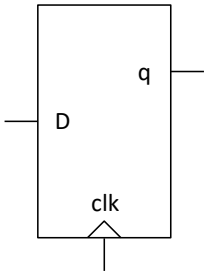


"At each rising clock edge, *q1* = *in*.
After that, *q2* = *q1* = *in*.
After that, *out* = *q2* = *q1* = *in*.
Therefore *out* = *in*."

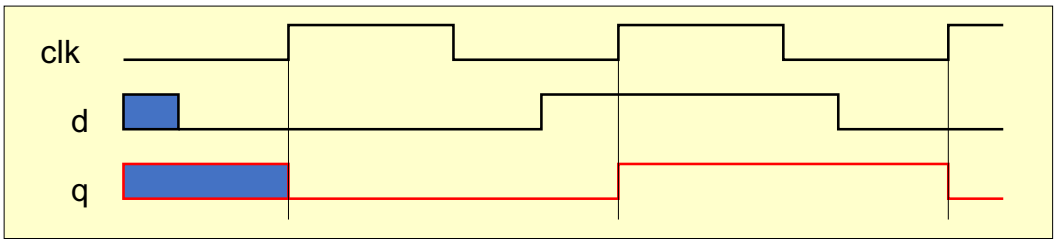


Biestables

- Ejemplo de biestables:

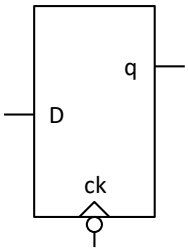



```
module biestable_d(
  input clk,d,
  output reg q);

always @ (posedge clk)
  q <= d;

endmodule
```

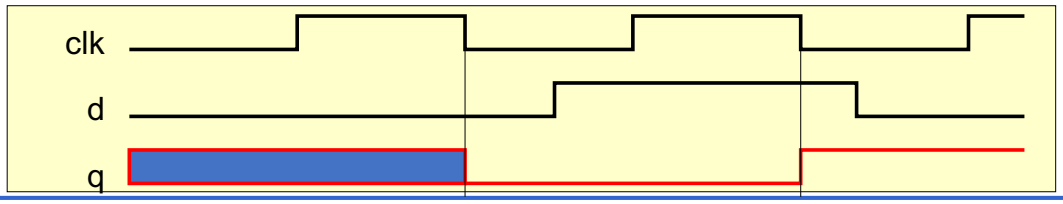


Biestables (ii)

Ejemplo de biestable D disparado en flanco negativo

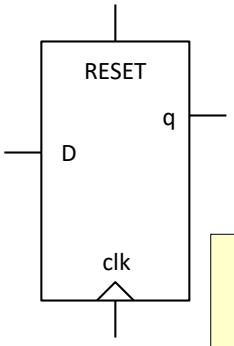



```
module biestable_d(
  input clk,d,
  output reg q);

  always @ (negedge clk)
    q <= d;

endmodule
```

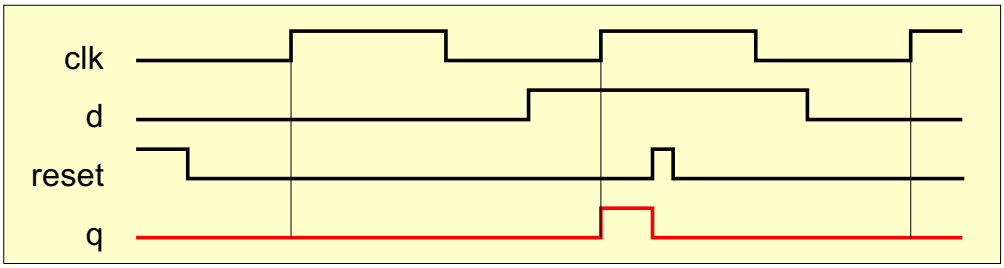


Biestables (iii)

Biestable D con reset asíncrono

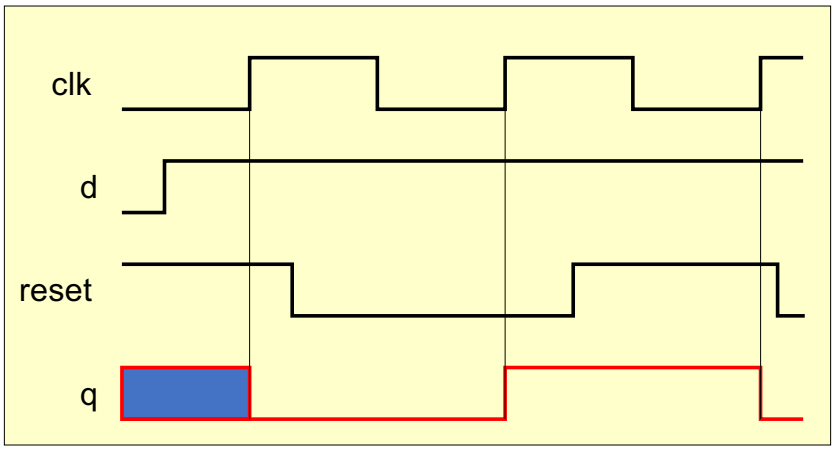



```
module biestable_d(
  input clk,d,reset,
  output reg q);

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

endmodule
```



Biestables (iv)

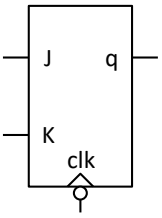
- Biestable D con reset síncrono




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

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

Biestables (v)

- Biestable JK



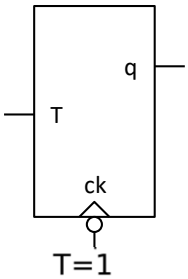
JK / q	00	01	11	10
0	0	0	1	1
1	1	0	0	1

Q

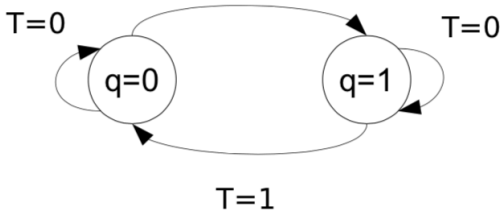
```
module jk_flip_flop (
  input       clk,
  input       j,
  input       k,
  output reg q);

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 (vi)

- Biestable T



T=1



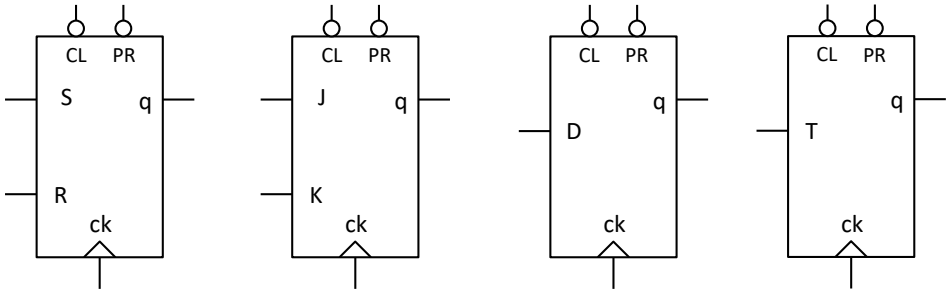
T=1

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

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

endmodule
```

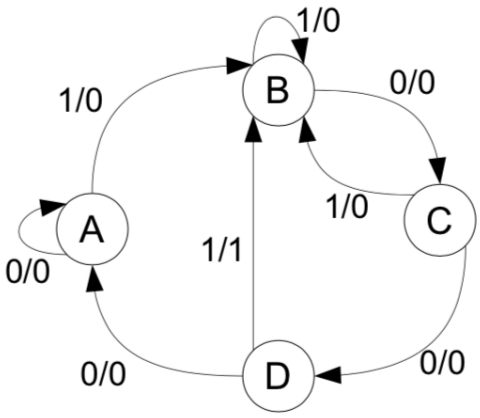
Biestables (y vii)

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:

- Un proceso always que establece cómo cambia de estado (por flanco de subida, flanco de bajada, reset asíncrono, etc)
- Otro proceso always de **cálculo de los próximos estados y salidas**

Máquinas de estado (ii)

```
module mi_diagrama_de_estados(
   input LISTA_DE_ENTADAS,
   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
```

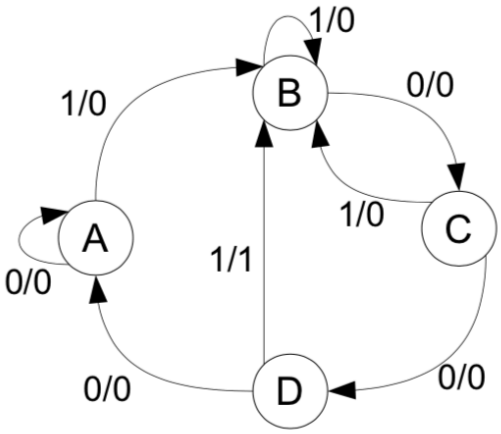

Máquinas de estado (iii)

- 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 (iv)

```
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 (v)

- El proceso de cálculo 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.

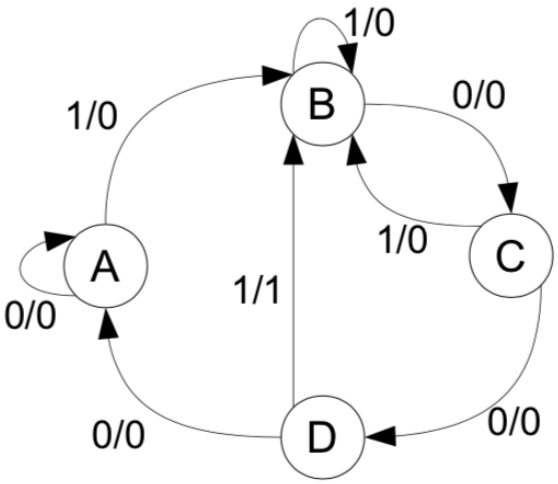
Máquinas de estado (vi)

```
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



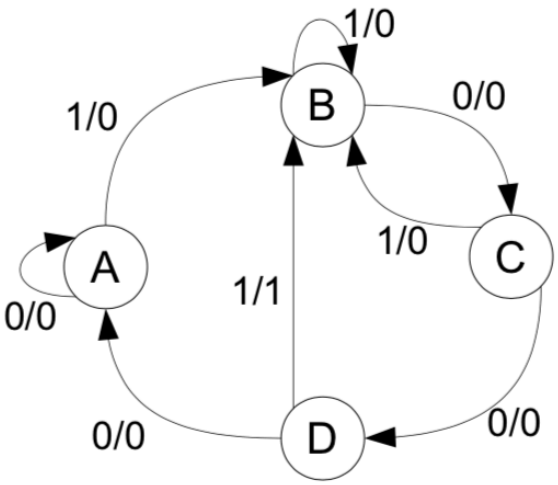
Máquinas de estado (vi)

```
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

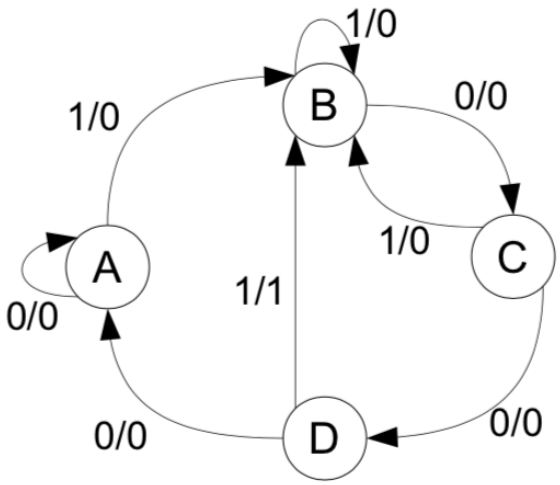


Máquinas de estado (vii)

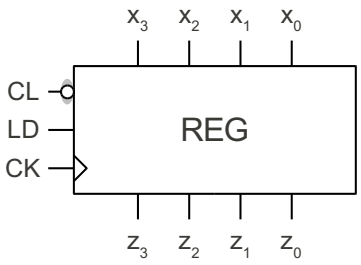
```
      C:
        if(x == 1)
          next_state = B;
        else
          next_state = D;
      D:
        if(x == 1)
          begin
            z = 1;
            next_state = B;
          end
        else
          next_state = A;
    endcase
  end
endmodule
```

Estado C

Estado D



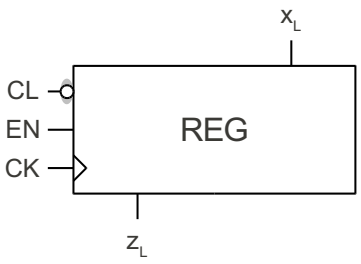
Ejemplo: Registro de carga en paralelo con clear asíncr.



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
```

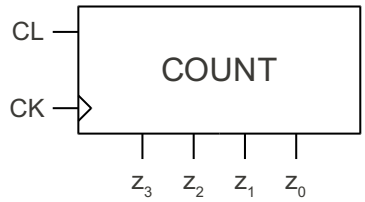
Ejemplo: Registro de desplazamiento



CL, EN	Operation	Type
0x	$q \leftarrow 0$	async.
11	$q \leftarrow SHL(q)$	sync.
10	$q \leftarrow q$	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], x

        assign zl = q[3];

endmodule
```


Ejemplo:
Contador ascendente con clear asíncrono

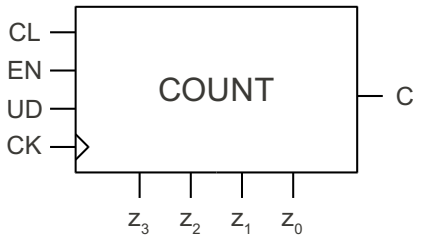


CL	Operation	Type
1	$q \leftarrow 0$	async.
0	$q \leftarrow q+1\|_{mod\ 16}$	sync.

```
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
```

Ejemplo: Contador ascendente/descendente con clear asíncrono

- 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 de circuitos con verilog

Bloque III: Simulación

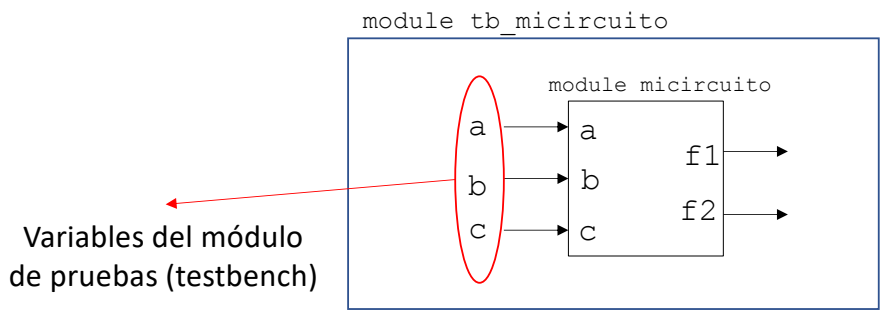
1. Test Bench
2. Simulación de módulos combinacionales
3. Simulación de módulos secuenciales

Simulación de circuitos con verilog

ISE Design Suite incluye la herramienta **ISim**, para simular el funcionamiento de nuestros diseños antes de implementarlo físicamente

Proceso: **New Source -> Verilog Test Module**, que debe ser asociado al módulo que queremos simular.

Se creará un fichero testbench al que sólo tendremos que añadir las órdenes para la simulación



Simulación de circuitos con verilog

Verilog tiene dos tipos de bloques procedimentales para usar en simulación que se ejecutan en paralelo:

always

Es un bucle que se ejecuta cíclicamente mientras dura la simulación

initial

sólo se ejecuta una vez, desde el inicio de la simulación; incluye sentencias que se ejecutan secuencialmente

Lo más importante es definir una buena estrategia de test para comprobar que el módulo realmente funciona como nosotros queremos

- En circuitos combinacionales, probaremos las 2^n combinaciones de entrada
- En CSS, suministraremos una secuencia de entrada para ver cómo responde el módulo

Simulación de módulos combinacionales

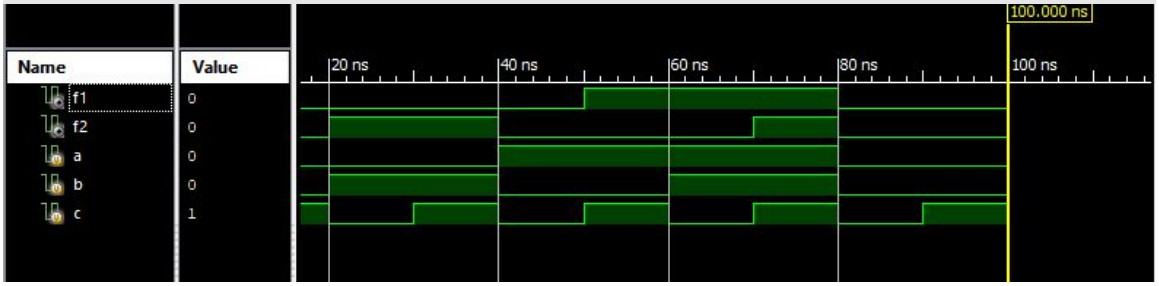
Un fichero de pruebas (testbench) contiene siempre un bloque **initial**, que define los valores iniciales de las variables, y cómo queremos que cambien con el tiempo. Por ejemplo:

```
//Initial Inputs
initial
begin
    a=0; //puedo definir, uno a uno, los valores iniciales
    b=0; //no importa el orden, son valores iniciales
    c=0;
    #10; //espero 10 ns
    c=1; //a y b no cambian
    #10; //espero 10 ns
    {a,b,c}=0'b010; //puedo asignar valores a los 3 con {}
    b=0; //no importa el orden, son valores iniciales
    c=0;
    #50; //espero 10 ns
    $finish; fin de la simulación
end
```

Simulación de módulos combinacionales (ii)

Para probar las 2^n combinaciones de las variables de entrada:, es muy útil el bloque **always** que se ejecuta en paralelo al bloque initial

```
always #10 {a,b,c} ={a,b,c} + 1; //se generan 000, 001, 010…. 111
                                 //cambiando de valor cada 10ns
//Initial Inputs
initial
begin
    a=0; //puedo definir, uno a uno, los valores iniciales
    b=0; //no importa el orden, son valores iniciales
    c=0;
    #100; //espero 100 ns
$finish; //fin de la simulación
end
```



Simulación de Circuitos Secuenciales

- Tenemos que dar valores a las entradas de datos (X), de reloj (clk) y de inicialización (clear, reset…) para comprobar que el módulo responde adecuadamente al diagrama de estados diseñado
- La señal clk se modela fácilmente en un bloque **always**:

  Ej. `always #5 clk = !clk; // clk T=10ns`

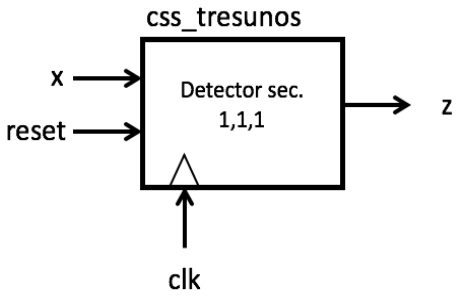
- La secuencia de valores de entrada se hace en el bloque **initial** y podemos:
  - asignar valor a variables:
  - Esperar un tiempo
  - Esperar a que llegue un flanco de la señal de reloj:

    `@(posedge clk)` ; espera a que llegue un flanco de subida

    `@(negedge clk);` espera a que llegue un flanco de bajada

Ejemplo de especificación y simulación de un CSS (i)

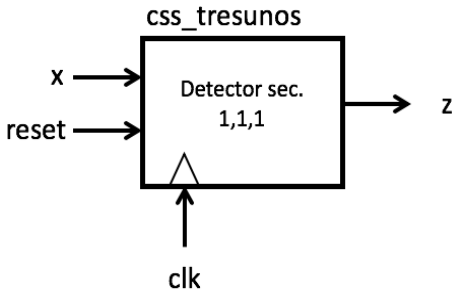
- **Circuito detector de la secuencia 1,1,1**



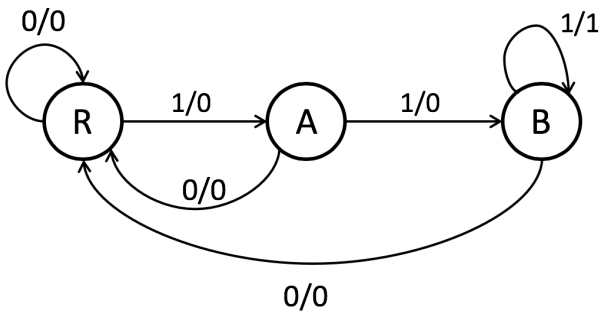
- La entrada x cambia sincronizada con los flancos de subida de clk, y el circuito genera salida z=1, si detecta tres o más unos consecutivos.
  - Si es de Mealy, coincidiendo con la aparición del tercer bit
  - Si es de Moore, en el ciclo siguiente.
- Necesitamos una entrada adicional (reset) para inicializar el circuito

Ejemplo de especificación y simulación de un CSS (ii)

- **Circuito detector de la secuencia 1,1,1**

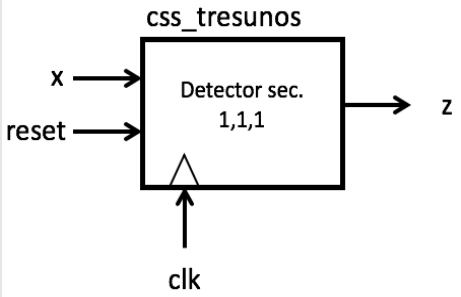


- Diagrama de estados:



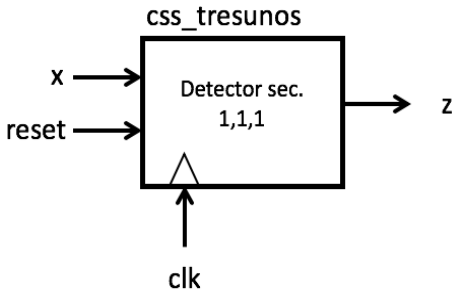
Ejemplo (iii)

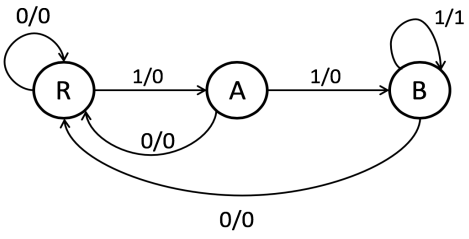
```
module css_tresunos(
  input x, clk, reset,
  output reg z);
  /*asignación de estados */
  parameter R = 2'b00,
            A = 2'b01,
            B = 2'b10;

  reg [1:0] current_state,next_state;
  /* Proceso de cambios de estado */
  always @(posedge clk, posedge reset)
    begin
     if(reset)
       current_state <= R;
     else
       current_state <= next_state;
    end

                  SIGUE ->
```



Ejemplo (iv)

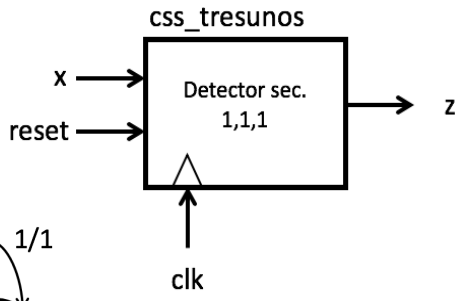
```
 always @(current_state,x)
   begin
    z = 0;
    case(current_state)
      R: if(x == 1)
           next_state = A;
      A: if(x == 1)
           next_state = B;
         else
           next_state = R;
      B: if(x == 1)
           begin
              z=1
             next_state = B;
           end
         else
           next_state = R;
      default: next_state = R;
    endcase
   end
endmodule
```



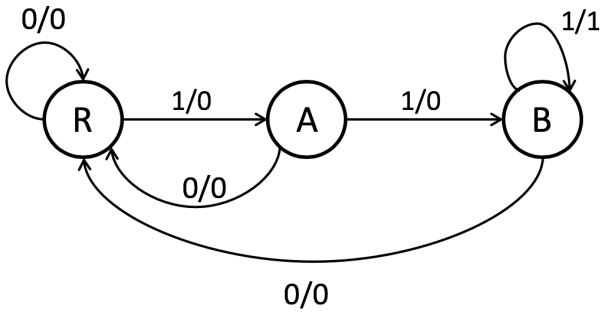


Ejemplo (v) (Simulación CSS)

- **Circuito detector de la secuencia 1,1,1**



- Diagrama de estados:

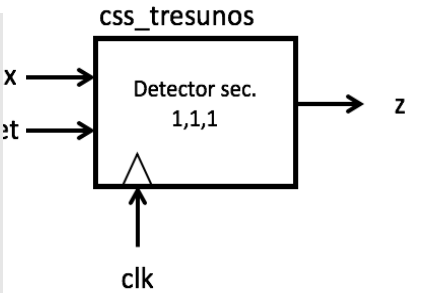


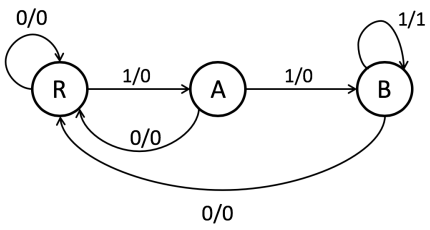
- Vamos a simular la respuesta a la secuencia de entrada x:

0,0,1,0,0,1,1,1,1,0,1,1,0,1,1,1,1,1,0

Ejemplo (vi) (Órdenes de simulación CSS)

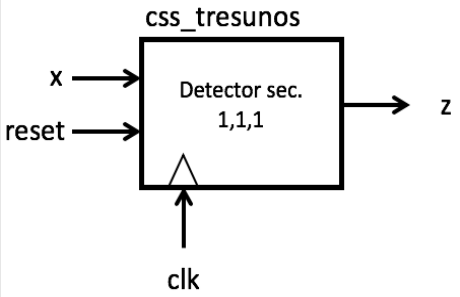
```
// aquí creo la señal de reloj
always #5 clk = !clk; T=10ns
initial
  begin
    x=0;
    clk =0;
    reset=0;
// genero pulso de reset del circuito
    #6;
    reset=1;
    #2;
    reset=0;
// secuencia 0,0,1,0,0,1,1,1,1,0,1,1,1,1,1,0,0

//sigue -->
```

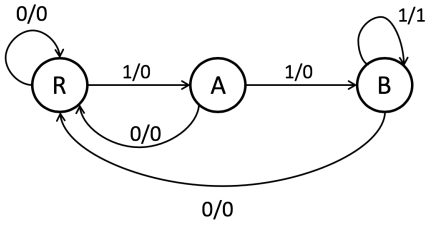




Ejemplo (vii) (Órdenes de simulación CSS)

```
// <--sigue
// secuencia
0,0,1,0,0,1,1,1,1,0,1,1,1,1,1,0,0
    @(posedge clk); //espero flanco de subida
    x=0;
    repeat (2) @(posedge clk);
    x=1;
    @(posedge clk);
    x=0;
    repeat (2) @(posedge clk);
    x=1;
    repeat (4) @(posedge clk);
    x=0;
    @(posedge clk);
    x=1;
    repeat (5) @(posedge clk);
    x=0;
    repeat (2) @(posedge clk);
    #100;
    $finish;
  end
endmodule
```





Ejemplo (viii) (Resultados simulación CSS)