

---

# INTRODUCCIÓN A HDL VERILOG

**Departamento de Tecnología Electrónica  
Universidad de Sevilla**

**Paulino Ruiz de Clavijo Vázquez**  
**<Paulino@dte.us.es>**

*Rev.7 (nov 2012)*

---

# Índice

- ▶ Introducción a HDL Verilog
- ▶ Bloque I: Diseño de circuitos combinacionales
- ▶ Bloque II: Diseño de circuitos secuenciales

---

# 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

---

# 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](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](http://sutherland-hdl.com/online_verilog_ref_guide/verilog_2001_ref_guide.pdf)

---

# 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

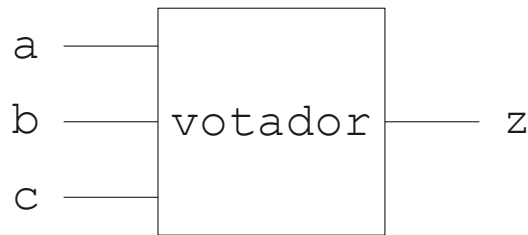
```
module mi_circuito (  
  input x, y,  
  input z,  
  output f1, f2  
);  
  
  wire cable_interno;  
  reg variable_a;  
  
  ...  
  ...  
  ...  
  
endmodule
```

Declaración del módulo con sus  
entradas y salidas

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

Descripción 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
```



---

# 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
```

# Tipos de descripciones

## ▶ Descripción procedimental

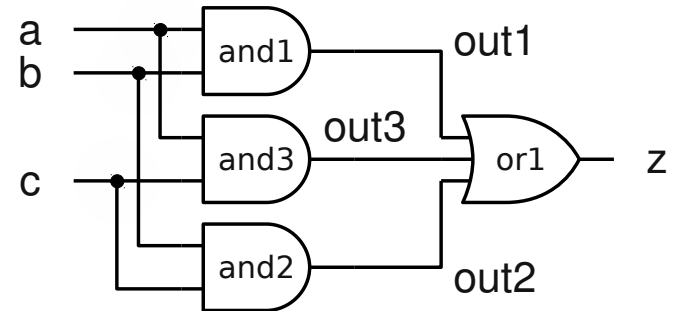
- ▶ Permite el uso de estructuras de control
- ▶ La descripción es algorítmica, igual que el software
- ▶ Facilita la creación de funciones complejas
- ▶ Se basa en la sentencia **always**

```
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
```

# Tipos de descripciones

## ► Descripción estructural

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

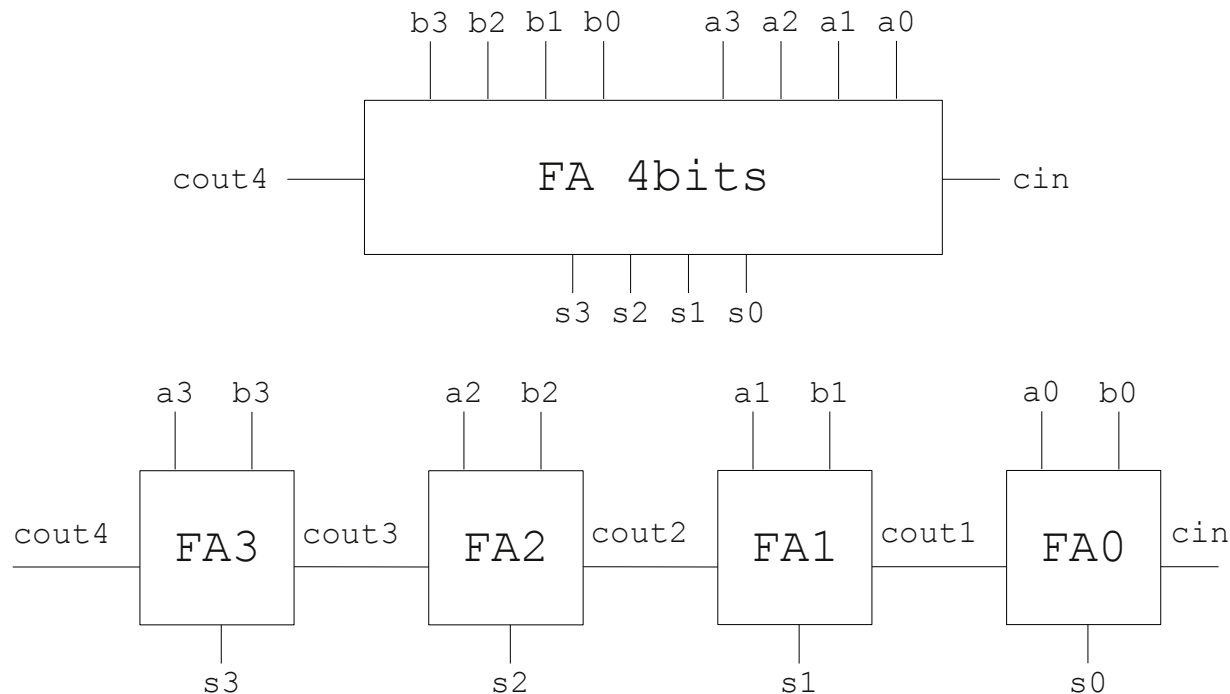
---

# 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 FULL-ADDER 4 bits a partir de varios FULL-ADDER de un bit



---

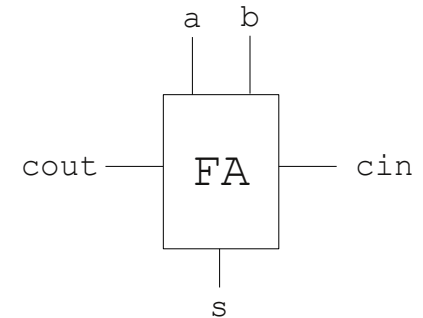
# Tipos de descripciones

▶ 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 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

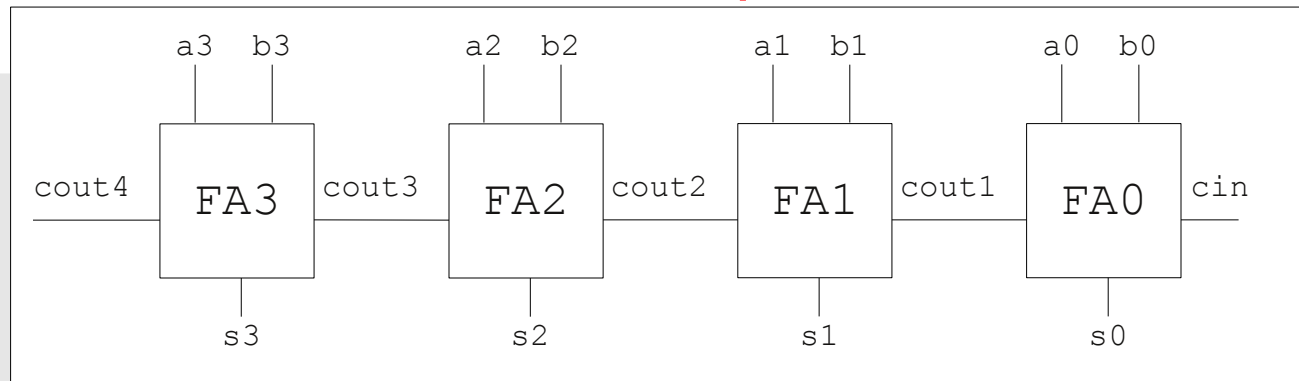
- Unión de 4 FULL-ADDER: **conexión posicional.**

```
module fulladder4(  
  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

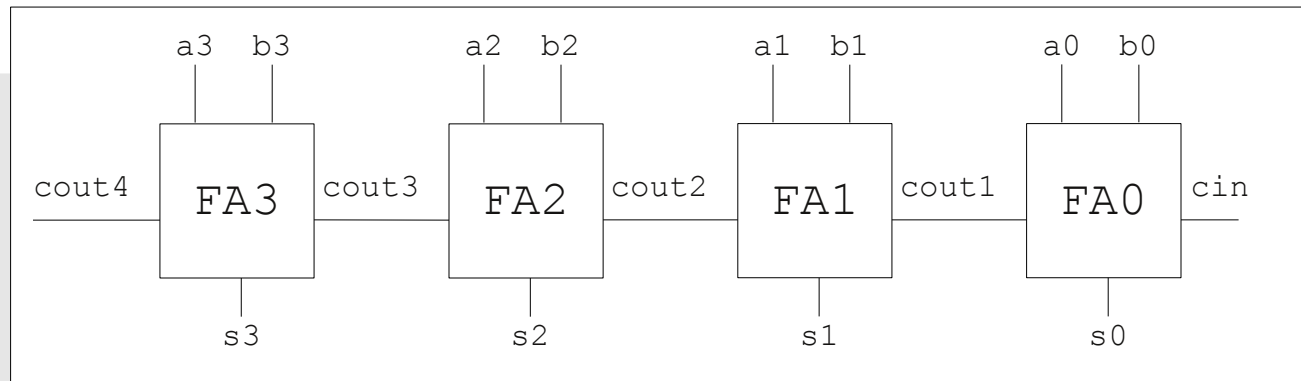
## ► Unión de 4 FULL-ADDER: **conexión nombrada.**

```
module fulladder4(  
  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 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
- ▶ Todas las asignaciones que se realicen dentro de un procedimiento (always) deben ser sobre una señal tipo reg

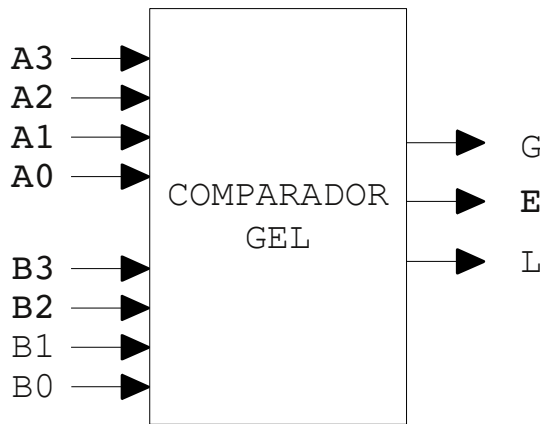
# Puertos de entrada/salida

- ▶ 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

```
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
```

---

# 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 línea: precedido de doble barra “//”

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

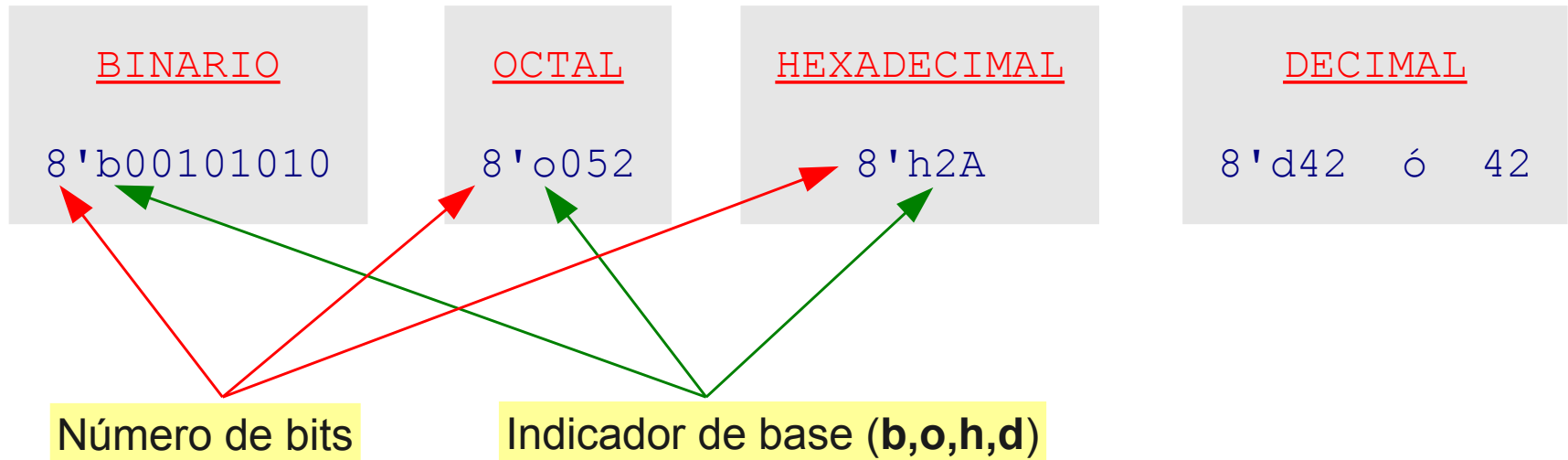
▶ Comentario de varias líneas: comienza con /\* y termina con \*/

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

```
wire a;
```

# Sintaxis básica

- ▶ Literales: Se puede expresar en varios formatos
  - ▶ Ejemplo: “00101010” en binario



---

# Sintaxis básica

- ▶ 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
```



# 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
```

# 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;
```

# 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 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
```

# Operadores

## ▶ Operadores a nivel de bits:

Operador	Ejemplo de código Verilog
&	<code>c = a&amp;b; // Operación AND de todos los bits</code>
	<code>c = a b; // Operación OR de todos los bits</code>
^	<code>c = a^b; // Operación XOR de todos los bits</code>
~	<code>b = ~a; // Inversión de todo los bits</code>

- ▶ 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
<code>~&amp;</code>	<code>d = a ~&amp; b; // Operador NAND a nivel de bits</code>
<code>~ </code>	<code>d = a ~  b; // Operador NOR a nivel de bits</code>
<code>~^</code>	<code>d = a ~^ b; // Operador EXNOR a nivel de bits</code>

---

# 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 es verdadera la condición

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



# Operadores

- ▶ Operadores lógicos: No confundirlos con los operadores a nivel de bits.

Operador	Ejemplo de código Verilog
&&	<code>a &amp;&amp; b; // Devuelve 1 si a y b son verdaderos</code>
	<code>a    b; // Devuelve 1 si a ó b es verdadero</code>
!	<code>!a; // Devuelve 1 si a es falso ó 0 si a es verdadero</code>

# Operadores

## ► Operadores aritméticos

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

# 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 cierto  
        //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
```

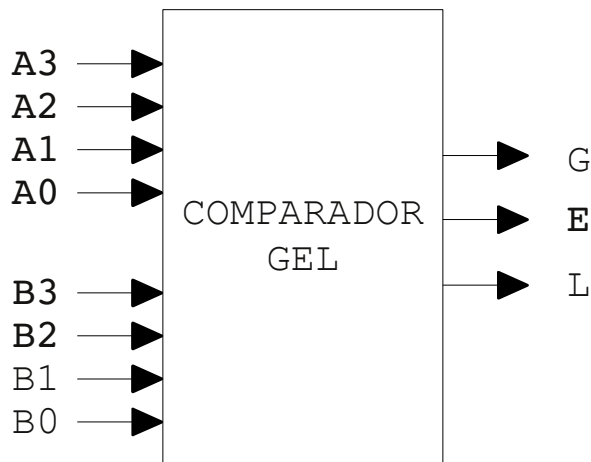
**ERROR**

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

**Correcto**

# Sentencias condicionales

## ▶ Ejemplo de comparador GEL



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

# Sentencias condicionales

## ▶ Sentencia **case**

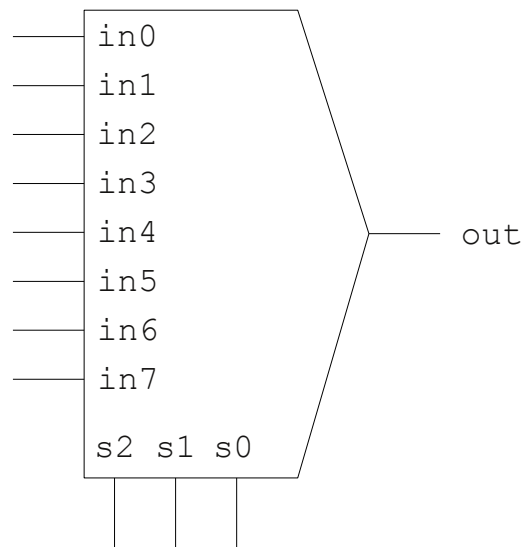
- ▶ Se utiliza dentro de un proceso “always”
- ▶ Si alguno de los casos tiene más de una sentencia hay que utilizar un bloque “begin” ... “end”
- ▶ Se puede utilizar **default** para los casos no enumerados

```
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 out);  
  
    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



---

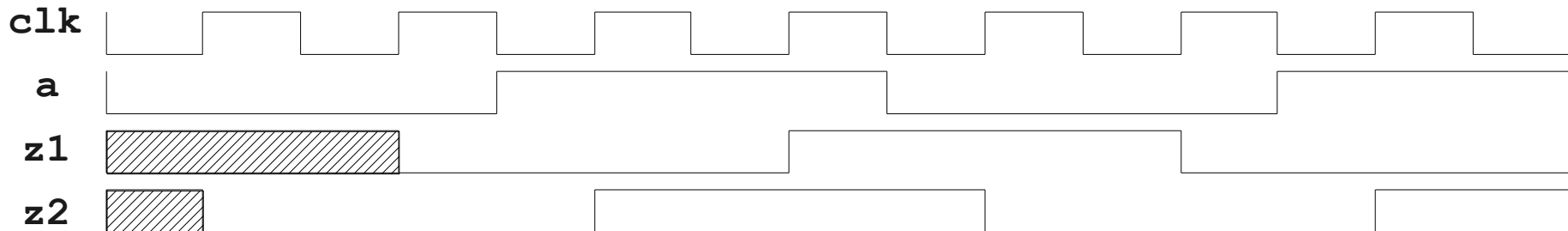
# Sintaxis II

- ▶ Asignamiento **no bloqueante** signo  $\leq$ 
  - ▶ 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  $\leq$ , 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

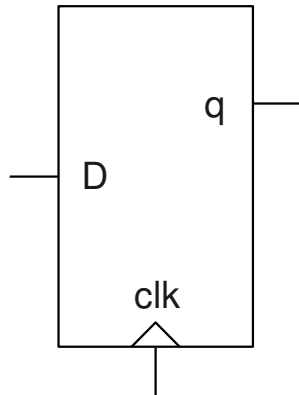
```
module no_bloqueante(input a,clk,  
output reg z1);  
reg q;  
always @(posedge clk)  
begin  
    q <= a;  
    z1 <= q;  
end  
endmodule
```

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

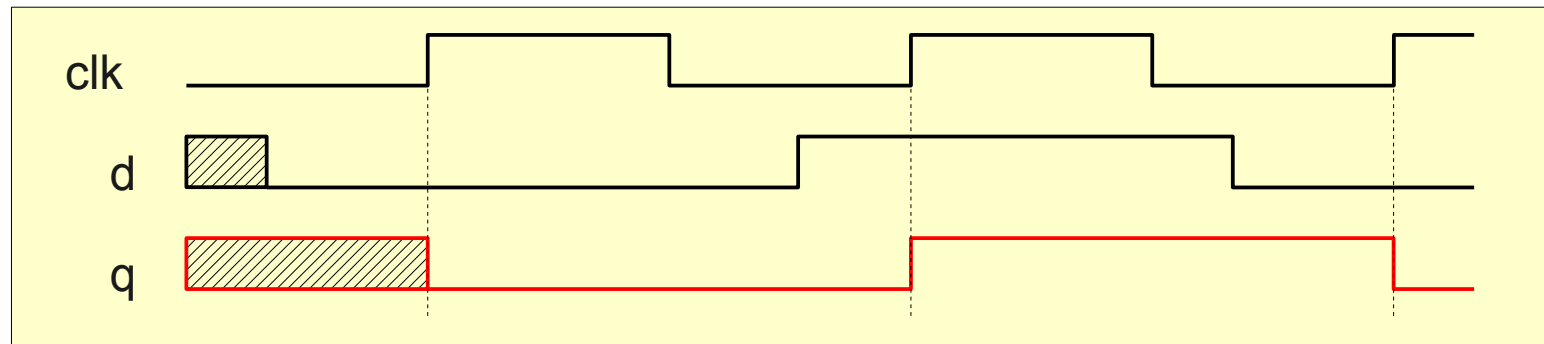


# Biestables

## ► Ejemplo de biestables:

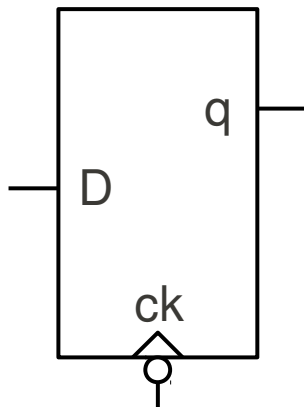


```
module biestable_d(  
    input clk,d,  
    output reg q);  
  
    always @ (posedge clk)  
        q <= d;  
  
endmodule
```

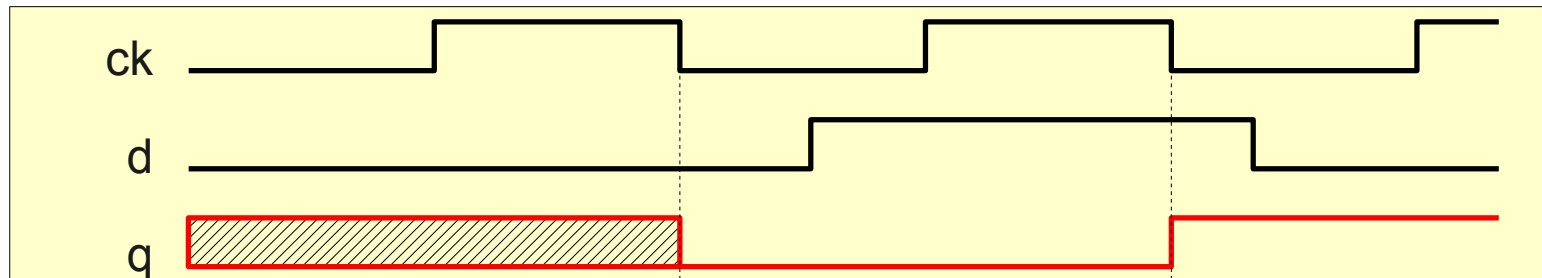


# Biestables

- ▶ Ejemplo de biestable D disparado en flanco negativo

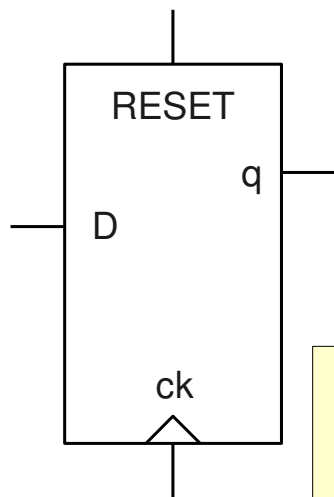


```
module biestable_d(  
    input ck,d,  
    output reg q);  
  
    always @ (negedge ck)  
        q <= d;  
  
endmodule
```

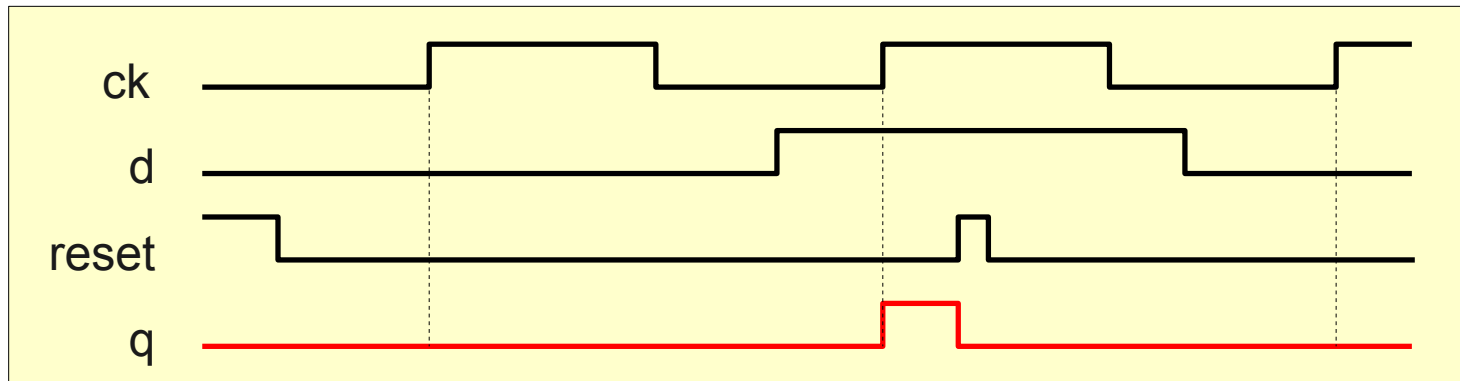


# Biestables

## ► Biestable D con reset asíncrono

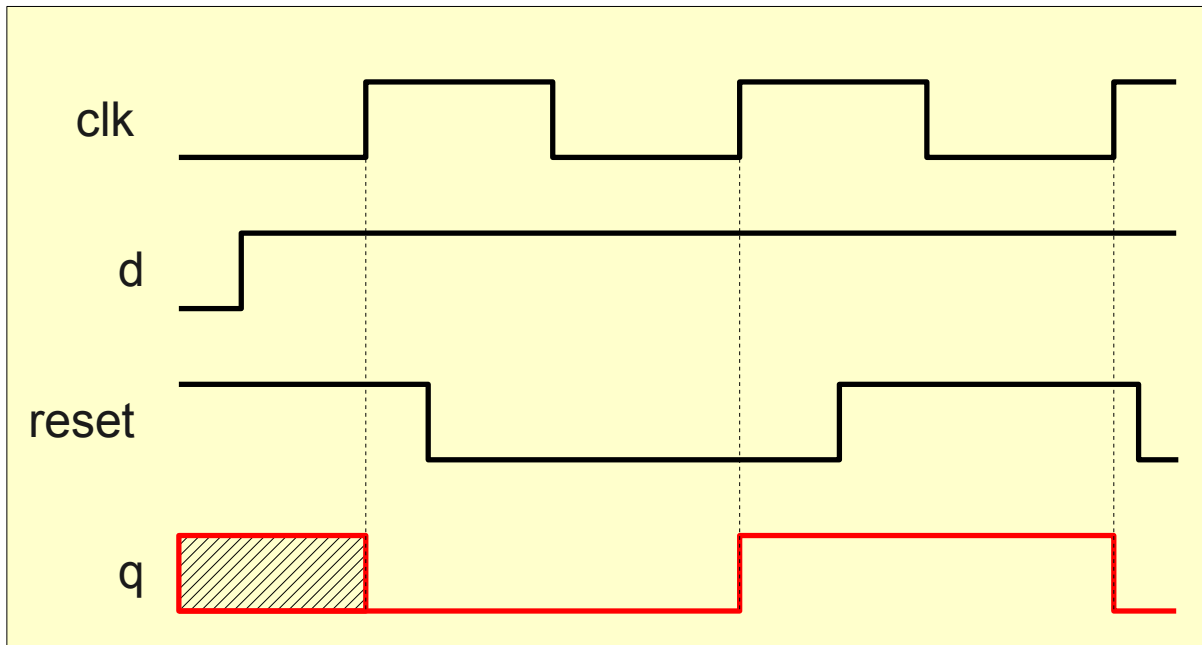


```
module biestable_d(  
    input ck,d,reset,  
    output reg q);  
  
    always @ (posedge ck or posedge reset)  
        if (reset)  
            q <= 1'b0;  
        else  
            q <= d;  
  
endmodule
```



# Biestables

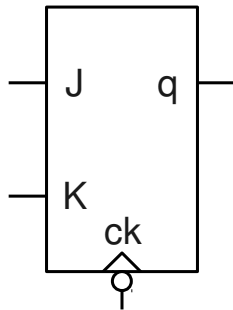
## ► Biestable D con reset síncrono



```
module biestable_d(  
    input clk,d,reset,  
    output reg q);  
  
    always @ (posedge clk)  
        if (reset)  
            q <= 1'b0;  
        else  
            q <= d;  
endmodule
```

# Biestables

## ► Biestable JK



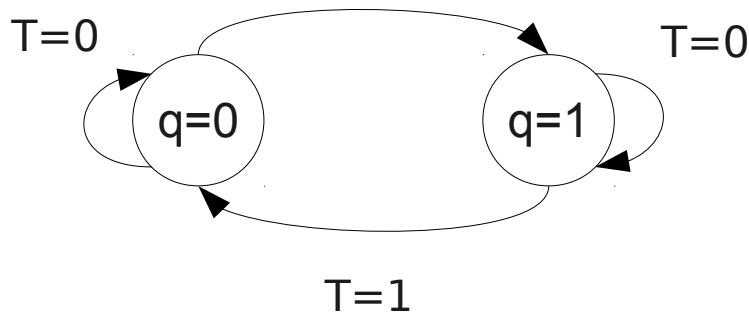
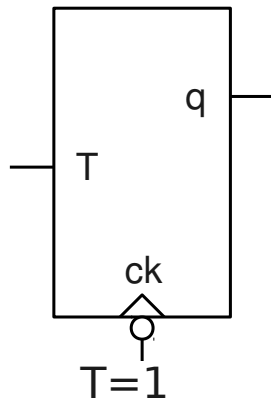
JK	00	01	11	10
q				
0	0	0	1	1
1	1	0	0	1

Q

```
module jk_flip_flop (  
    input      ck,  
    input      j,  
    input      k,  
    output reg q);  
  
    always @(negedge ck)  
        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
```

# Bistables

## ► Bistable T

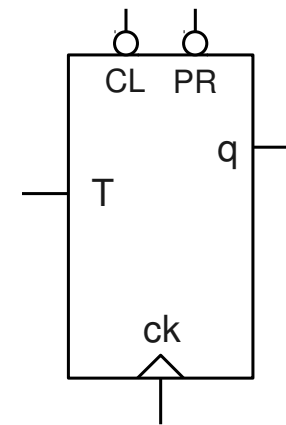
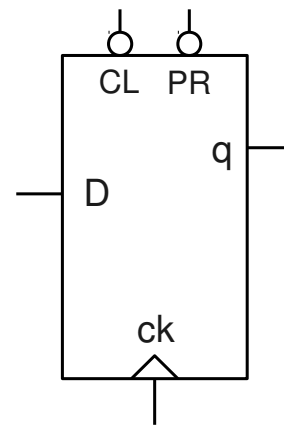
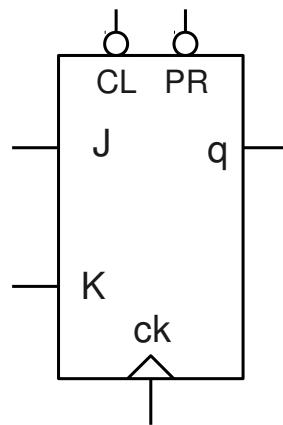
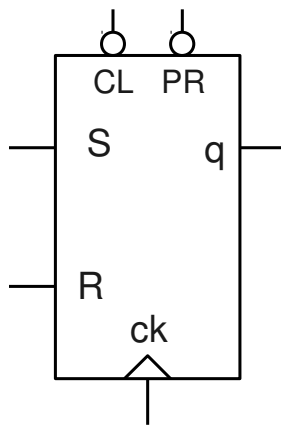


```
module biestable_t(  
    input ck,  
    input t,  
    output reg q);  
  
    always @(negedge ck)  
        if (t == 1)  
            q <= ~q;  
  
endmodule
```

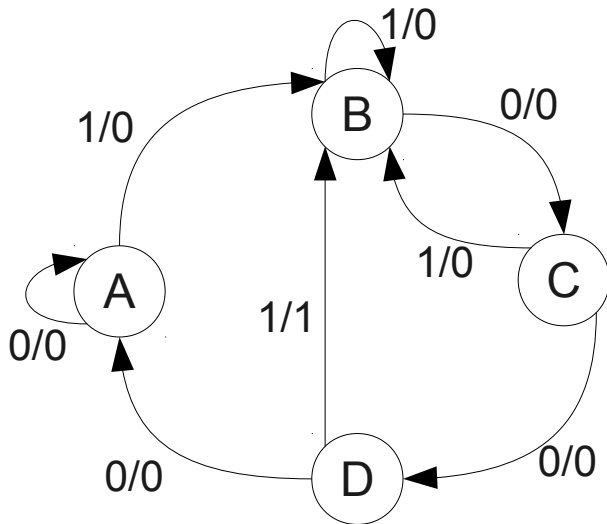


# Biestables

- ▶ 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 calculo 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
```

---

# Máquinas de estado

- ▶ En la estructura general hay que completar 4 partes de código:
  1. Definición y asignación de estados, según el número de estados utilizaremos mas o menos bits.
  2. 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
  3. Proceso de cambio de estado: Siempre es el mismo código
  4. 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);
```

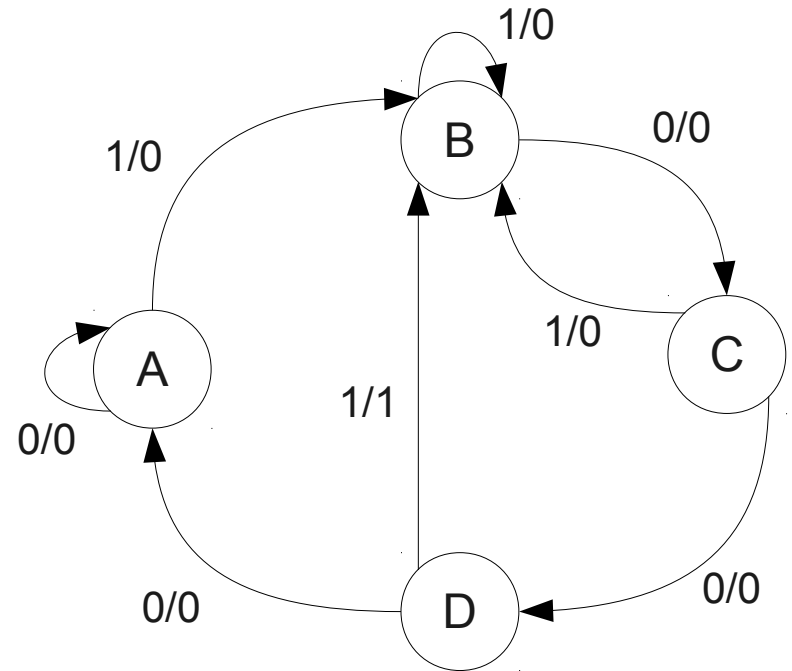
```
parameter A = 2'b00,  
          B = 2'b01,  
          C = 2'b10,  
          D = 2'b11;
```

Asignación  
de estados

```
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 ->



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.

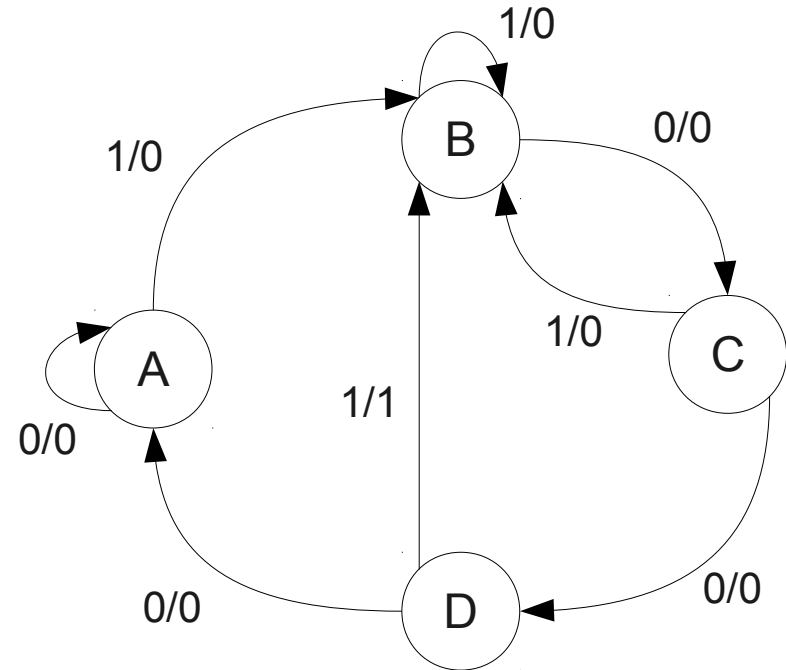
# Máquinas de estado

```
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

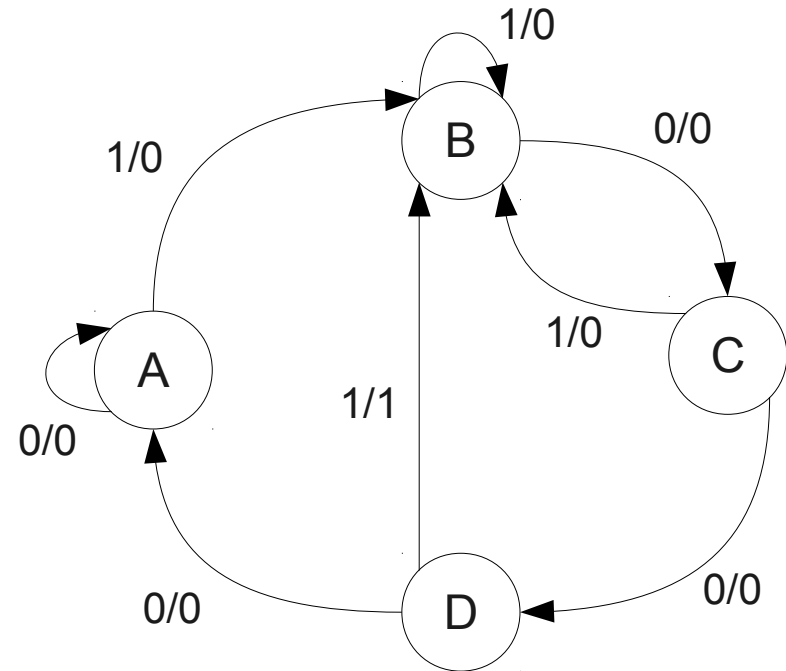


# Máquinas de estado

```
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

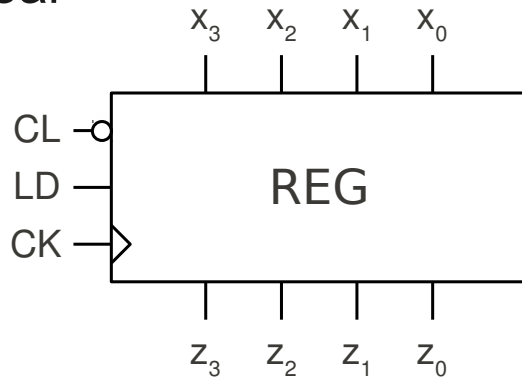
Estado D





# Registros

- ▶ Registro con carga en paralelo y clear

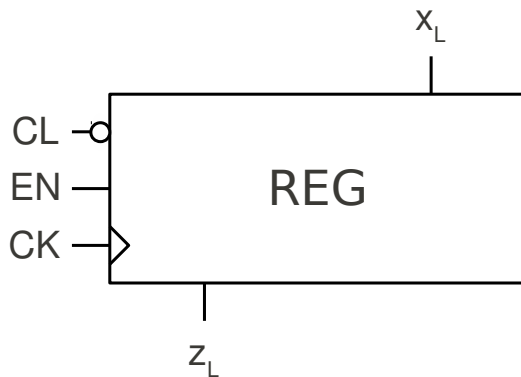


CL, LD	Operation	Type
0x	$q \leftarrow 0$	async.
11	$q \leftarrow x$	sync.
10	$q \leftarrow q$	sync.

```
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

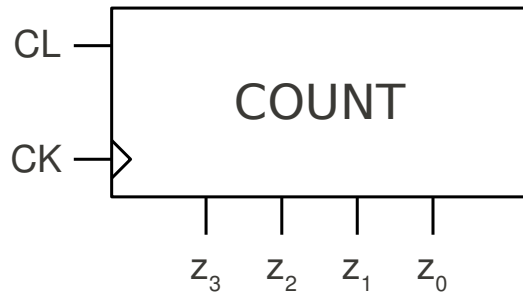


CL, EN	Operation	Type
0x	$q \leftarrow 0$	async.
11	$q \leftarrow \text{SHL}(q)$	sync.
10	$q \leftarrow q$	sync.

```
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], xl};  
  
    assign zl = q[3];  
  
endmodule
```

# Contadores

## ▶ Contador ascendente con clear

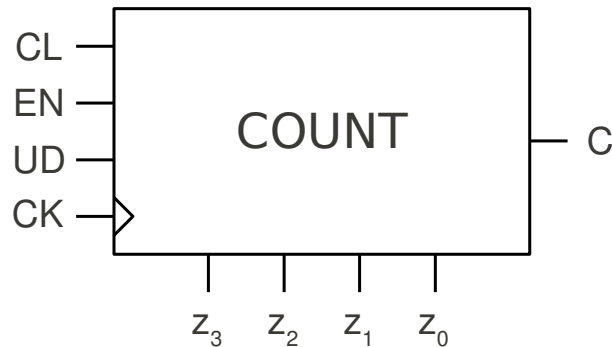


CL	Operation	Type
1	$q \leftarrow 0$	async.
0	$q \leftarrow q+1 \bmod 16$	sync.

```
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	$q \leftarrow q+1 \bmod 16$	sync.
011	$q \leftarrow q-1 \bmod 16$	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
    
```