Tema 1

Introducción a Verilog

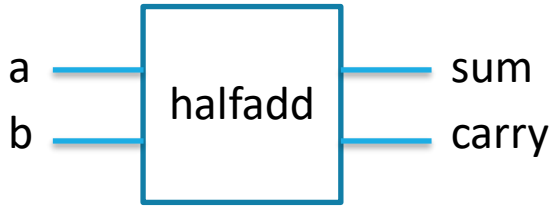
Objetivos

- Presentar las características básicas del lenguaje de descripción de hardware Verilog
- Introducir las palabras claves del lenguaje y sus reglas
- Introducir el diseño jerárquico
- Presentar los tipos de datos, operadores y estructuras del lenguaje
- Mostrar la descripción de módulos básicos combinacionales y secuenciales
- Mostrar la descripción de estímulos para simulación (testbench)

Esquema de la unidad

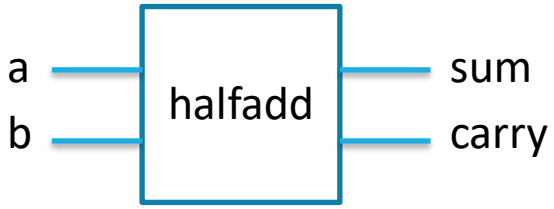
- Aspectos básicos
- Tipos de datos
- Vectores y arrays
- Literales y bases
- Procedimientos:
  - sentencias condicionales
  - bucles
- Operadores
- Parámetros
- Máquinas de estado
- Directivas define e include
- Simulación funcional

- Descripción de módulos de diseño
  - Comienza con la palabra clave module
  - Se declaran la lista de entradas y salidas del módulo
  - Se pueden agrupar entradas (o salidas) si son del mismo tamaño
  - En el cuerpo del módulo se describe el comportamiento del mismo
  - Se termina con la palabra clave endmodule




```
module halfadd (
    input wire a,b,
    output wire sum, carry);

    assign sum = a^b;
    assign carry= a&b;

endmodule
```

- Descripción de módulos de diseño
  - assign se utiliza para definir una expresión lógica
  - ^ es la op. exclusive-OR
  - & es la op. and
  - Las palabras clave son siempre en minúsculas
  - Verilog es “case-sensitive”: diferencia mayúsculas y minúsculas

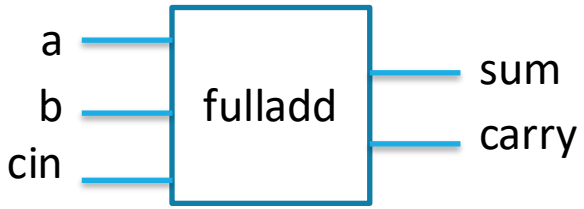


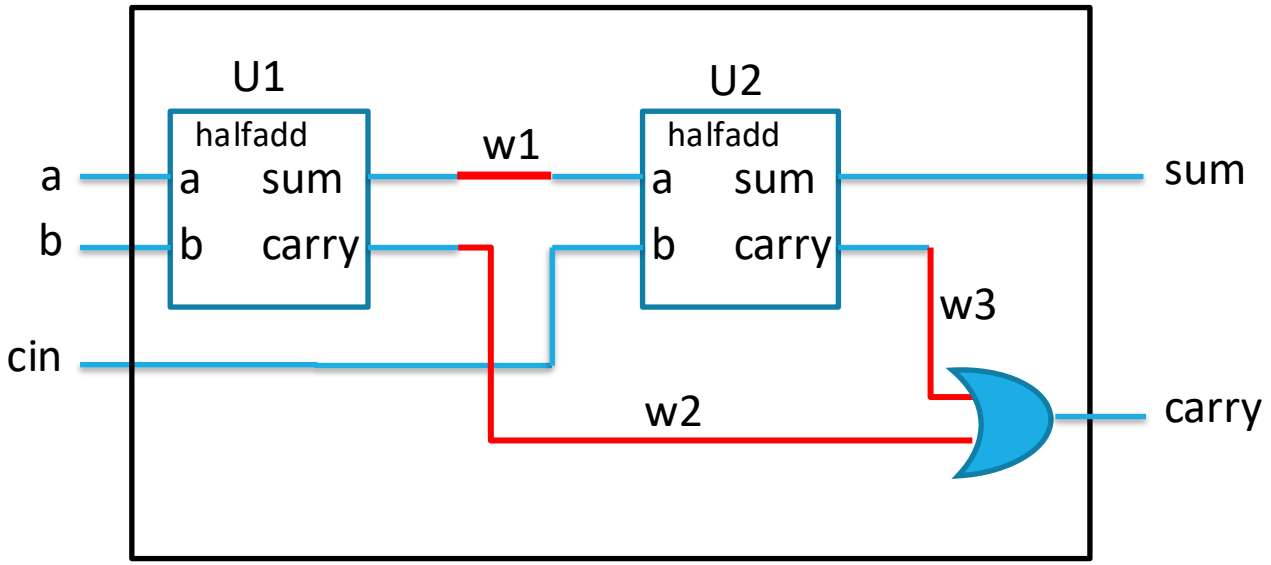

```
module halfadd (
    input wire a,b,
    output wire sum,carry);

    assign sum = a^b;
    assign carry= a&b;

endmodule
```

Aspectos básicos de Verilog

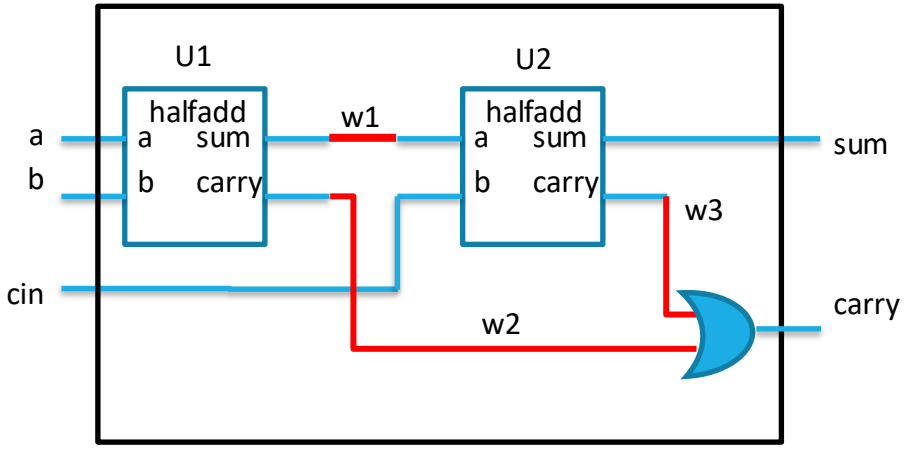
- Reglas para los comentarios y el formato:
  - Se pueden escribir comentarios de línea y de bloque
    - Los comentarios de línea comienzan a partir de dos barras consecutivas (//) y terminan al final de la línea.
    - Puede ser una línea completa o solo la parte final
    - Los comentarios de bloque comienzan con (/*) y terminan con (*/)
  - Verilog es un lenguaje sin formato
    - Se puede organizar el código como se quiera
    - Se recomienda usar espacios, tabulaciones y fin de línea, de modo que el código sea legible
    - En general, no se debería poner más de una sentencia ejecutable por línea.

- Diseño jerárquico:
  - Podemos construir nuevos módulos a partir de otros





- Diseño jerárquico:
  - Es necesario:
    - Declarar las variables locales
    - Instanciar los módulos
      - Dar a cada instancia un nombre
      - Conectar los puertos de entrada y salida de cada instancia




```
module fulladd (input wire a, b, cin,
                output wire sum, carry);

   wire w1,w2,w3;

   halfadd  U1( .a(a), .b(b), .sum(w1), .carry(w2) );
   halfadd  U2( .a(w1), .b(cin), .sum(sum), .carry(w3) );

   assign carry = w2 | w3;

endmodule
```

- Conexión posicional vs conexión nombrada
  - Se pueden conectar puertos y variables respetando el orden de la declaración de los puertos en el módulo (conexión posicional)
  - No es recomendable, es fácil cometer errores
  - Es preferible usar la conexión nombrada (la vista anteriormente)

```
module halfadd (
    input wire a,b,
    output wire sum,carry);

   assign sum a^b;
   assign carry= a&b;

endmodule
```

```
module fulladd (input wire a, b, cin,
                output wire sum, carry);

   wire w1,w2,w3;

   halfadd  U1(a,b,w1,w2);
   halfadd  U2(w1,cin,sum,w3);

   assign carry = w2 | w3;

endmodule
```

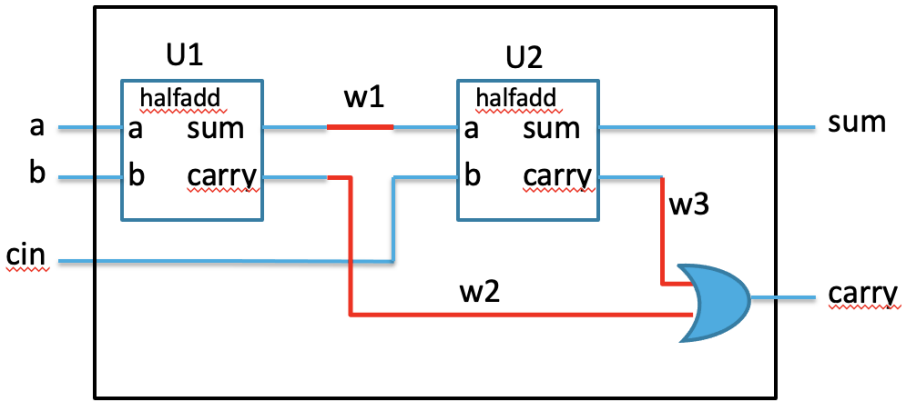
Aspectos básicos de Verilog

- Tipos de descripción:
  - Descripción funcional
  - Descripción estructural
  - Descripción procedimental

- Tipos de descripción:
  - Descripción funcional:
    - Se realizan asignaciones de forma continua utilizando assign
    - Representa conexiones directas de hardware que están constantemente activas y responden inmediatamente a cambios en sus entradas
    - Modela lógica combinacional
    - Todas las sentencias assign se ejecutan concurrentemente

```
module halfadd (
    input wire a,b,
    output wire sum,carry
    );
   assign sum = a^b;
   assign carry= a&b;
endmodule
```

- Tipos de descripción:
  - Descripción estructural:
    - Se conectan módulos que ya están definidos previamente mediante instanciación
    - Las puertas lógicas básicas ya están predefinidas (and, nand, or,nor, xor, xnor, not, buf, etc.)
    - Es muy útil para la interconexión de módulos creados previamente (diseño jerárquico)

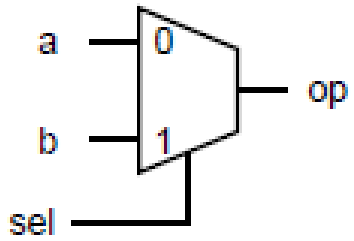



```
module fulladd (input wire a, b, cin,
                output wire sum, carry);
   wire w1,w2,w3;

   halfadd  U1(a,b,w1,w2);
   halfadd  U2(w1,cin,sum,w3);

   assign carry = w2 | w3;
endmodule
```

- Tipos de descripción:
  - Descripción procedimental:
    - Permite el uso de estructuras de control similares a las de los lenguajes de programación
    - Se estructura en bloques procedimentales o procesos
    - Todos los bloques procedimentales se ejecutan concurrentemente
    - Dentro de cada proceso las instrucciones se ejecutan en el orden en que están escritas (flujo secuencial)

- Bloques procedimentales
  - Pueden definir comportamientos complejos, como lógica combinacional repetitiva o condicional
  - Pueden definir circuitos con memoria (secuenciales)
  - Pueden definir bancos de estímulos (testbenches) para simulaciones
  - Fundamentalmente hay dos tipos de bloques procedimentales:
    - tipo always: Se ejecutan continuamente (cíclicamente)
    - tipo initial: Se ejecutan linealmente (de principio a fin una sola vez)

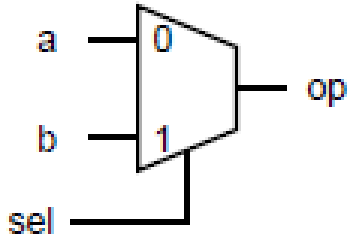
```
always @(a, b, sel)
   if(sel== 1)
     op = b;
   else
     op = a;
```




```
initial
   begin
        a = 1;
        b = 0;
        sel = 1;
        #10ns;
        sel= 0;
        ...
   end
```

- Bloques procedimentales
  - Cuando hay más de una sentencia en un bloque, estas han de agruparse mediante begin y end

```
always @(a, b, sel)
   if(sel== 1)
      op = b;
   else
      op = a;
```




```
initial
   begin
        a = 1;
        b = 0;
        sel = 1;
        #10ns;
        sel= 0;
        ...
   end
```

- Ejemplo de concurrencia
  - Un módulo puede contener múltiples bloques procedimentales (initial, always) y múltiples sentencias assign y no hay orden de ejecución entre ellos sino que se ejecutan en paralelo (concurrencia)
  - Dentro de un procedimiento las sentencias se ejecutan secuencialmente en orden de aparición

definen la misma función

```
module ejemplo1 (
   input wire x,
   input wire y,
   input wire z,
   output wire f1);

 assign f1 = x & y & z;

endmodule
```

```
module ejemplo2 (
   input wire x,
   input wire y,
   input wire z,
   output wire f1);

  wire f0;
  assign f0 = x & y;
  assign f1 = f0 & z;

endmodule
```

```
always begin   // a periodica  T=10
       #5 a = ~a;
       end

always begin
       #10 b = ~b; // b periodica  T=20
       end

initial begin
       a=0;
       b=0;
       #500;
      $finish;
    end
```

- Bloques procedimentales
  - Cada bloque always se ejecuta (dispara) cuando hay un evento en alguna variable de su lista de sensibilidad que viene determinada por una @
  - En lógica combinacional la lista de sensibilidad debe incluir todas las variables
  - Para la lógica secuencial el evento suele ser el flanco de una señal, es decir, se disparan en una transición específica de una determinada señal.
  - Se utilizan las palabras clave posedge y negedge

```
always @(a, b, sel)
   if(sel== 1)
      op = b;
   else
      op = a;
```

```
always @(posedge clock)
      q<=d;
```

Lista de sensibilidad

Esquema de la unidad

- Aspectos básicos
- Tipos de datos
- Vectores y arrays
- Literales y bases
- Procedimientos:
  - sentencias condicionales
  - bucles
- Operadores
- Parámetros
- Máquinas de estado
- Directivas define e include
- Simulación funcional

Tipos de datos en Verilog

- *Nets*, para representar conexiones. No almacenan valores.
  - Se usan sobre todo para conexiones entre módulos: *wire* (el más común y el tipo por defecto).
  - Se usan en asignaciones continuas (assign), nunca en bloque procedimentales.

- *Variables*, para almacenar valores.
  - El más importante: *reg*. Otros: *integer* (entero con signo de 32 bits), *real*, *time*…
  - Se usan en bloques procedimentales (always, initial)

- Tipos compuestos:
  - *vectores*: cualquier wire o reg puede ser vectorial,
    - reg [15:0] data;
    - wire [3:0] opcode;
  - *arrays*: para memorias o bancos de registros,
    - reg [7:0] memoria[0:255];

Constantes en Verilog

- *parameter*: constante válida en un módulo que permite crear bloques genéricos personalizables, su valor puede modificarse al instanciar el módulo
- *localparam*: constante cuyo valor es fijo y no se puede modificar

Tipos de datos en Verilog

- Tipos por defecto y declaraciones implícitas
  - Una declaración sin tipo es *wire* por defecto
  - Una errata en un nombre de variable puede llevar a una declaración implícita como *wire*
  - Lo recomendable es declarar completamente todas las variables
  - También se puede usar la directiva `default_nettype none de esta forma se genera un error si se intenta usar una variable no declarada

```
module halfadd (
    input wire a, b,
    output sum, carry);

  assign sum = a^b;
  assign carry = a&b;

endmodule
```

```
module fulladd (input wire a, b, cin,
                output wire sum, carry);

  wire w1,w2,w3;

  halfadd  U1( .a(a), .b(b), .sum(w1), .carry(ww2) );
  halfadd  U2( .a(w1), .b(cin), .sum(sum), .carry(w3) );

assign carry = w2 | w3;

endmodule
```

No declarada

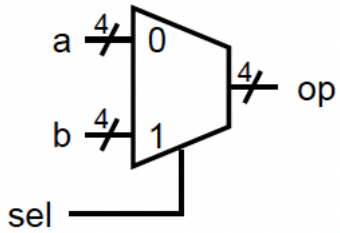
Esquema de la unidad

- Aspectos básicos
- Tipos de datos
- Vectores y arrays
- Literales y bases
- Procedimientos:
  - sentencias condicionales
  - bucles
- Operadores
- Parámetros
- Máquinas de estado
- Directivas define e include
- Simulación funcional

Vectores y arrays

- Al declarar un vector se define su tamaño (rango máximo)
  - Puede ser descendente o ascendente
  - Puede contener expresiones (constantes y conocidas antes de la simulación)
  - Por defecto las componentes del vector no tienen signo, si se quieren cantidades con signo, serán en notación complemento a 2 y se ha de declarar como reg signed

```
module mux4 (
   input wire [3:0] a,b,
   input wire  sel,
   output reg [3:0] op
);
   always @(a, b, sel)
    if(sel== 1)
       op = b;
    else
       op = a;
endmodule
```



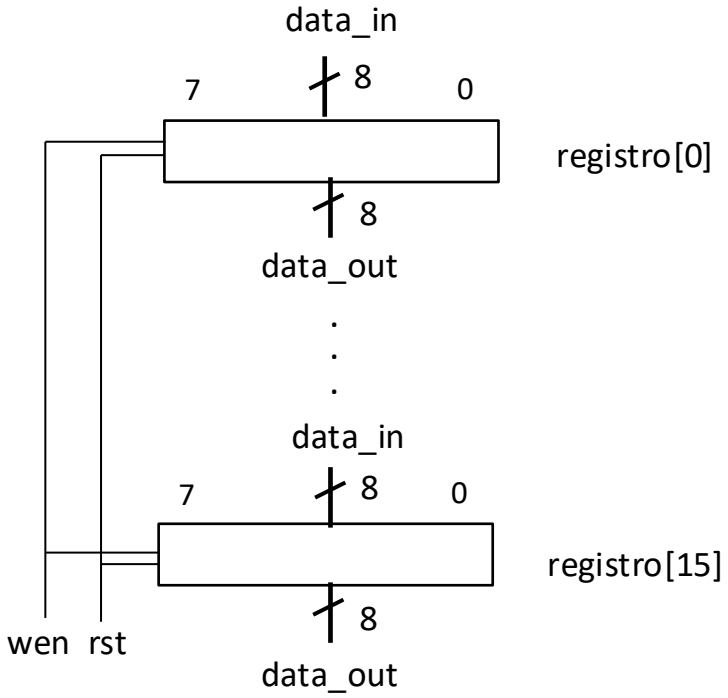

```
reg signed [7:0] svec8;
reg [7:0] usvec8;

initial begin
svec8 = 8'b11001101;   // -51
usvec8 = 8'b11001101;  // 205
   . . .
```

```
parameter N=8;
reg [N-1:0] in;
```

Vectores y arrays

- Verilog permite arrays (matrices)
  - De cualquier tipo
  - Con cualquier número de dimensiones (a nivel RT se usan dos dimensiones)
- Los arrays multidimensionales se usan para
  - Modelar memorias
  - Declarar grupos de registros

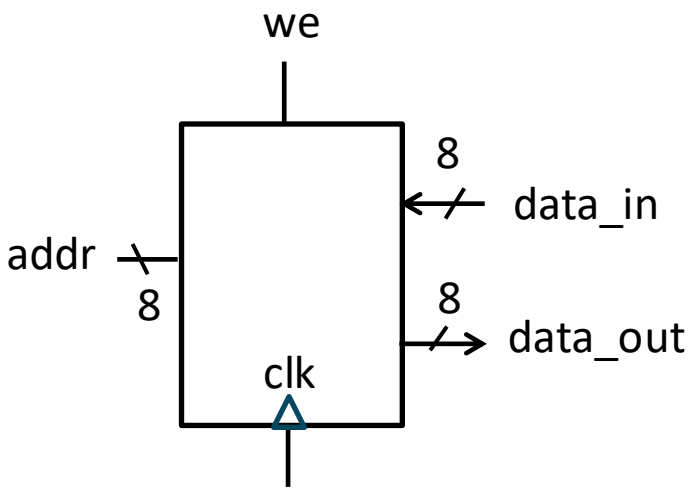
```
/* Declaración de un array de 16 registros de
8 bits */
reg [7:0] registro [0:15];

// registro   == array de bits
// registro [5]   == uno de los registros de 8
//                   bits  (el 6º)
// registro[5][2] == bit 2 del registro[5]
```



Vectores y arrays

- Ejemplo:

```
module memoria_simple (
    input wire clk,
    input wire we,
    input wire [7:0] addr,
    input wire [7:0] data_in,
    output wire [7:0] data_out
    );

    reg [7:0] mem [255:0];

    always @(posedge clk) begin
        if (we) begin
            mem[addr] <= data_in;
        end
    end
    assign data_out = mem[addr];
endmodule
```



Esta memoria contiene 256 palabras de 8 bits

Asignamos data_in (8 bits) al byte dado por mem[addr],

por ejemplo, si addr que es un vector de 8 bits contiene el valor 00110110 -> 54 haríamos mem[54] <- data_in (ambos son vectores de 8 bits)

Esquema de la unidad

- Aspectos básicos
- Tipos de datos
- Vectores y arrays
- Literales y bases
- Procedimientos:
  - sentencias condicionales
  - bucles
- Operadores
- Parámetros
- Máquinas de estado
- Directivas define e include
- Simulación funcional

Literales y bases

- Un **literal** numérico es una representación explícita y fija de un valor
  - Permiten especificar el ancho, la base y el valor
    - 8'd255: Literal de 8 bits en base decimal con valor 255 (1111 1111)
    - 16'hA5A5: Literal de 16 bits en base hexadecimal con valor A5A5 (1010 0101 1010 0101)
    - 4'b1010: Literal de 4 bits en base binaria con valor 1010
    - 8'b1010_0101: Se admiten "_" para mejorar la legibilidad
    - 8'o123: Literal de 8 bits en base octal con valor 123 (01010011)
    - Si se omite, la base por defecto es 10
- Cuando asignamos un literal a un vector no es necesario que tengan el mismo tamaño
  - El valor se extiende con 0 o se trunca al tamaño del vector

Esquema de la unidad

- Aspectos básicos
- Tipos de datos
- Vectores y arrays
- Literales y bases
- Procedimientos:
  - sentencias condicionales
  - bucles
- Operadores
- Parámetros
- Máquinas de estado
- Directivas define e include
- Simulación funcional

- Sentencia condicional **if-else**
  - Su sintaxis es:

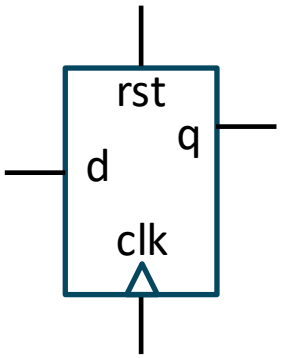
```
if (expresion)
   comando1;
else
   comando2;
```

  - También:

```
if (expresion1)
   comando1;
else if (expresion2)
   comando2;
else if (expresion3)
   comando3;
else comando4;
```

- Sentencia condicional **if-else**, ejemplo (I):

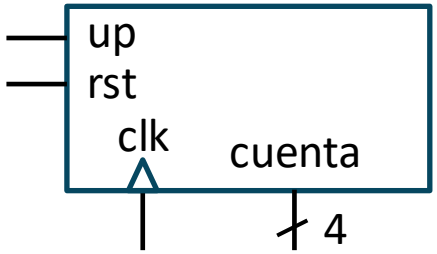
```
module flip_flop_d_reset(
        input wire clk, rst, d,
        output reg q
        );

 always@(posedge clk) begin
     if (rst == 1'b1)
         q <= 1'b0;
     else
         q <= d;
  end
endmodule
```



rst	d	q
1	X	0
0	0	0
0	1	1

- Sentencia condicional **if-else**, ejemplo (II):

```
module contador_con_reset (
        input wire clk, rst, up,
        output reg [3:0] cuenta );

 always@(posedge clk) begin
     if (rst == 1'b1)
         cuenta <= 4'b0000;
     else if (up == 1'b1)
         cuenta <= cuenta + 1;
  end
endmodule
```



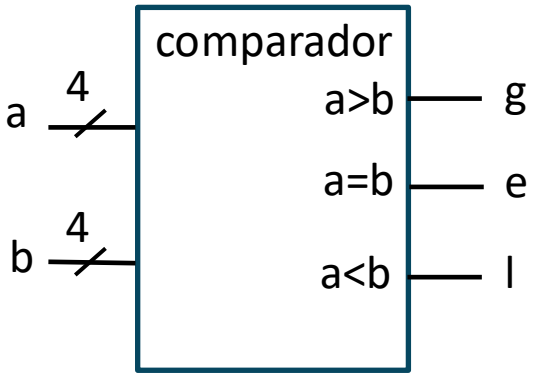
rst	up	cuenta
1	X	0
0	1	cuenta + 1
0	0	cuenta

Para *rst asíncrono*, se debe incluir el mismo en la lista de sensibilidad:

always @(posedge clk, posedge rst)

- Sentencia condicional **if-else**, ejemplo (III):

```
module comparador(
        input wire  [3:0] a, b,
        output reg g, e, l);

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



	g e l
a > b	1 0 0
a = b	0 1 0
a < b	0 0 1

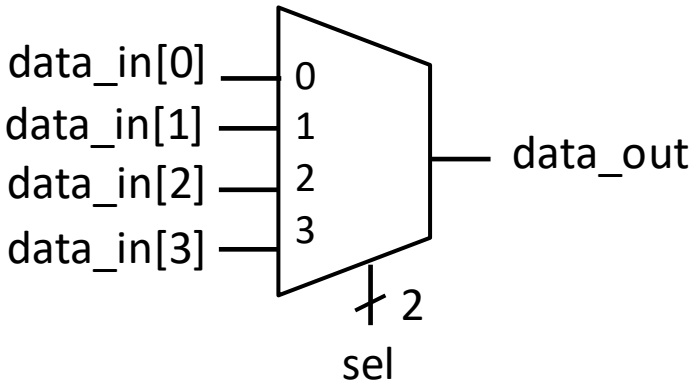
- Sentencia **case**
  - Su sintaxis es:

```
case (expresion)
    valor1: sentencia1;
    valor2: sentencia2;
    valor3: sentencia3;
    default: sentencia4;
endcase
```

  - default es opcional, aunque recomendable

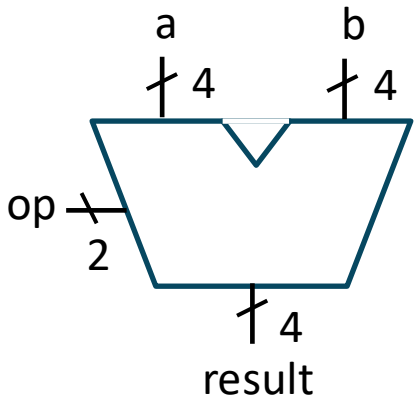
- Sentencia **case**, ejemplo (I):

```
module multiplexor_4a1 (
        input wire [1:0] sel,
        input wire [3:0] data_in,
        output reg data_out );

always @* begin
        case (sel)
             2'b00: data_out = data_in[0];
             2'b01: data_out = data_in[1];
             2'b10: data_out = data_in[2];
             2'b11: data_out = data_in[3];
            default: data_out = 1'bx;  // Valor 'x' para el caso por defecto
        endcase                        // p.ej: si sel valiese 2'b0x, 2'bx1, 2'bxz, etc.
end
endmodule
```



- Sentencia **case**, ejemplo (II):

```
module alu_simple (
        input wire [1:0] op,
        input wire [3:0] a,
        input wire [3:0] b,
        output reg [3:0] result);
 always @* begin
     case (op)
         2'b00: result = a + b; // Suma
         2'b01: result = a - b; // Resta
         2'b10: result = a & b; // AND
         2'b11: result = a | b; // OR
        default: result = 4'bx; // Valor 'x' del caso por defecto
     endcase
  end
endmodule
```



op[1:0]	result
0 0	a+b
01	a-b
10	AND(a,b)
11	OR(a,b)

Procedimientos: bucles

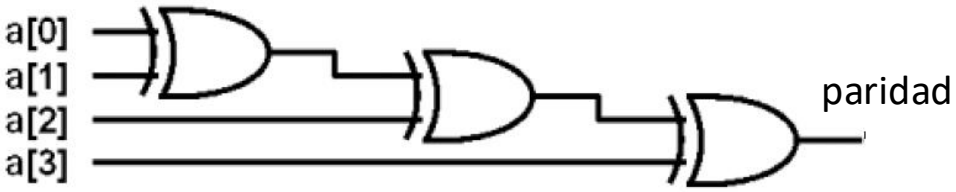
- Bucle **for**
  - Su sintaxis es:

```
for (initializacion; expresion; step)
    sentencia;
```

  - El bucle for comienza con la inicialización y evalúa la expresión, si esta se cumple realiza la sentencia y ejecuta la función step.
  - Permite escribir código más compacto y legible si se trabaja con estructuras repetitivas

- Bucle **for**, ejemplo (I):

```
module paridad_impar_4bits(
  input wire [3:0] a,
  output reg paridad);

integer i;
always @* begin
      paridad = 1'b0;
      for (i = 0; i <= 3; i=i+1)
          paridad = paridad ^ a[i];
     end
endmodule
```



Estos valores va tomando la salida en cada iteración

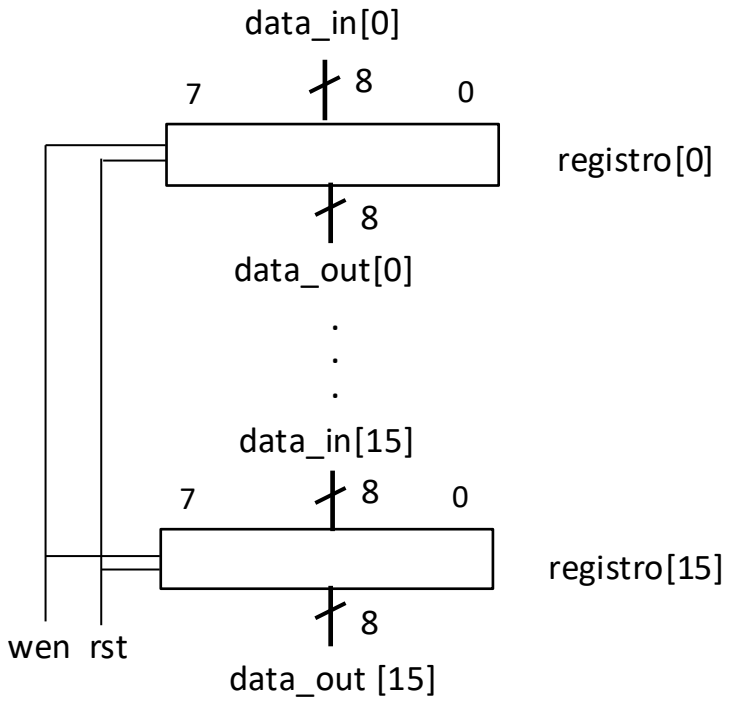
0
a[0]
a[1] xor a[0],
a[2] xor a[1] xor a[0]
a[3] xor a[2] xor a[1] xor a[0]

a[3:0]	paridad
0 0 0 0	0
0 0 0 1	1
0 0 1 0	1
0 0 1 1	0
. . .	. . .
1 1 0 1	1
1 1 1 0	1
1 1 1 1	0

Procedimientos: bucles

- Bucle **for**, ejemplo (II):

```
module banco_registros (
    input wire clk, rst, wen,
    input wire [3:0] addr [0:15],
    input wire [7:0] data_in [0:15],
    output wire [7:0] data_out [0:15] );

reg [7:0] registro [0:15]; // Declaración de un array de 16 registros de 8 bits
integer i;

// Lógica de escritura
 always @(posedge clk) begin
    if(rst)
        for (i = 0; i < 15; i=i+1)
            registro[i] <= 8'b00000000;
        else if (wen)
                registro <= data_in;
 end

// Lógica de lectura
  assign data_out = registro;
endmodule
```



Procedimientos: bucles

- Bucle **repeat**
  - Su sintaxis es:

```
repeat (expresion)
    sentencia;
```

  - El bucle repeat evalúa la expresión para obtener un número y ejecuta la sentencia el número de veces especificado.
  - Muy usado en simulación (testbenches)

Ejemplo:

```
// Generación de reloj:
//20 ciclos de periodo 10
repeat (20) begin
    #5 clk = ~clk;
end
```

Procedimientos: bucles

- Bucle **forever**:
  - Su sintaxis es:

```
forever sentencia;
```

  - se ejecuta siempre
  - sirve para definir señales en test benches para simulación
  - Ejemplo:

```
// Generación de reloj: periodo 10 unidades
initial begin
    clk = 0;
    forever #5 clk = ~clk;
end
```

Procedimientos: bucles

- Bucle while:
  - Su sintaxis es:

```
while (condición)
    sentencia;
```

  - ejecuta una sentencia hasta que una determinada expresión es falsa

  - Ejemplo:

```
//Espera los ciclos de reloj necesarios hasta que completed ==1
while (completed != 1)
    @(negedge clk);
```

Esquema de la unidad

- Aspectos básicos
- Tipos de datos
- Vectores y arrays
- Literales y bases
- Procedimientos:
  - sentencias condicionales
  - bucles
- Operadores
- Parámetros
- Máquinas de estado
- Directivas define e include
- Simulación funcional

Operadores

- Existen muchos operadores, algunos ya han ido apareciendo, se pueden clasificar en tipos
  - Aritméticos
  - Relacionales
  - Lógicos
    - De bits, de palabras (bit a bit), de reducción
  - De desplazamiento
  - De concatenación y replicación
  - Condicional ternario
  - De asignación bloqueante y no bloqueante

- **Operadores aritméticos**
  - **Suma**: +
    - a + b
    - Suma de dos valores.
  - **Resta**: -
    - a - b
    - Resta de dos valores.
  - **Multiplicación**: *
    - a * b
    - Multiplicación de dos valores.
  - **División**: /
    - a / b
    - División de dos valores (si ambos son enteros, es división entera).
  - **Módulo (Residuo)**: %
    - a % b
    - Devuelve el residuo de la división.

- **Operadores Relacionales y de Comparación**
  - **Igualdad lógica**: ==
    - a == b
    - Devuelve 1 (true) si a es igual a b.
  - **Desigualdad lógica**: !=
    - a != b
    - Devuelve 1 (true) si a es diferente de b.
  - **Mayor que**: >
    - a > b
  - **Menor que**: <
    - a < b
  - **Mayor o igual que**: >=
    - a >= b
  - **Menor o igual que**: <=
    - a <= b

- **Operadores lógicos (booleanos)**
  - **AND lógico**: &&
    - a && b
    - True si ambos a y b son true (1).
  - **OR lógico**: ||
    - a || b
    - True si al menos uno de a o b es true (1).
  - **NOT lógico**: !
    - !a
    - True si a es false (0).

- **Operadores bit a bit**
  - **AND bit a bit**: &
    - a & b
    - Operación AND entre cada bit de a y b.
  - **OR bit a bit**: |
    - a | b
    - Operación OR entre cada bit de a y b.
  - **XOR bit a bit**: ^
    - a ^ b
    - Operación XOR entre cada bit de a y b.
  - **NOT bit a bit**: ~
    - ~a
    - Niega cada bit de a
  - **Combinaciones de los mismos:**
    - **NAND** ~&, **NOR** ~|, **XNOR** ~^ **o** ^~

- Ejemplos

```
reg [3:0] vector1, vector2, vector3;
reg [3:0] num;
initial begin
   vector1 = 4'b1001;
   vector2 = 4'b1010;
   vector3 = 4'b11x0;
   num = ~ vector1;              // num = 0110
   num = vector1 & 4'b0111;      // num = 0001
   num = vector1 & vector2;      // num = 1000
   num = vector1 | vector2;      // num = 1011
   num = vector2 & vector3;      // num = 10x0
   num = vector2 | vector3;      // num = 1110
end
```

- **Operadores de reducción**
  - Aplican una operación bit a bit y generan un único bit de resultado:
    - **AND de reducción**: &a
    - **OR de reducción**: |a
    - **XOR de reducción**: ^a
    - **NAND de reducción**: ~&a
    - **NOR de reducción**: ~|a
    - **XNOR de reducción**: ~^a o ^~a
  - Ejemplos:
    - Si a = 4'b1010, &a = 1 & 0 & 1 & 0 = 0
    - Con reg [3:0] q; &q = q[3]& q[2]& q[1]& q[0]
  - En el ejemplo de la transparencia 37, bastaría hacer:
    - assign paridad = ^a;

- **Operadores de desplazamiento**
  - **Desplazamiento lógico izquierda** : <<
    - a = a << n; (despl. a izq., rellena con 0 por la dcha.)
  - **Desplazamiento lógico derecha** : >>
    - a = a >> n; (despl. a dcha., rellena con 0 por la izq.)
  - **Desplazamiento aritmético derecha con asignación**: >>>
    - a = a >>> n; (despl. a dcha., rellena con el bit de signo por la izq.)

- Ejemplos:

```
reg [7:0] vector1 , vector2;
reg signed [7:0] vectorsigno;
initial begin
        vector1 = 8'b10011001;
        vectorsigno = 8'b10011001;
        vector2 = vector1 << 3;      // 11001000 (entran 0 por la dcha.)
        vector2 = vector1 >> 1; // 01001100 (entra un 0 por la izq.)
        vector2 = vectorsigno >>> 1; // 11001100 (preserva signo)
end
```

- **Operadores de Concatenación y Replicación**
  - **Concatenación**: {}
    - {a, b}
    - Combina los bits de a y b en un único vector.
    - Por ejemplo, si a es de 4 bits y b es de 4 bits, {a, b} será de 8 bits.
    - c = {c[4:0] , c[7:5]} rota el vector c[7:0] tres posiciones a la izquierda.

{a, b} = 8'b10011100

  - **Replicación**: {n{}}
    - {4{a}}
    - Repite el valor de a n veces.
    - Por ejemplo, si a = 2'b01, entonces {4{a}} = 8'b01010101

- **Operador condicional ternario:**
  - Evalúa una condición y asigna un valor diferente según si dicha condición se cumple o no.
    - result = <condition> ?<true value> :<false value>;

- **Operadores de Asignación en Procedimientos**
  - **Asignación bloqueante: =**
    - Se ejecuta de forma secuencial dentro de un bloque initial o always.
  - **Asignación no bloqueante: <=**
    - Permite que las asignaciones se programen en paralelo dentro de un bloque secuencial.
    - Muy utilizado en el modelado de registros y secuencias de diseño sin introducir dependencias de orden.

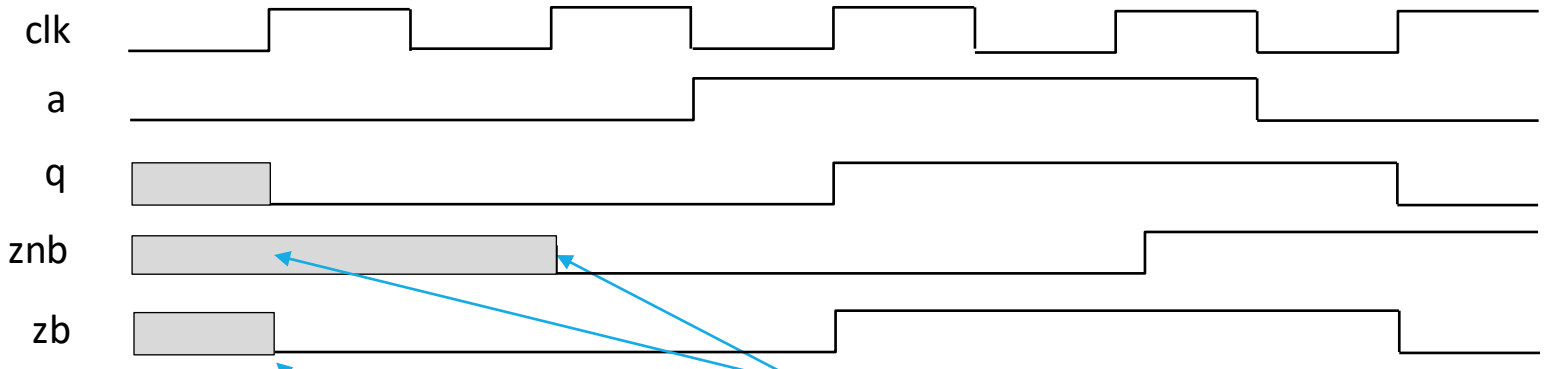
```
module bloqueante(
        input wire a, clk,
        output reg zb
        );
 reg q;
 always @(posedge clk) begin
     q = a;
     zb = q;
end
endmodule
```

Cuando llega el flanco activo de reloj, zb toma el valor de a

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

Cuando llega el flanco activo de reloj, znb toma el valor de q y, en el siguiente flanco, es cuando toma el valor de a

Operadores




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

Cuando llega el flanco activo de reloj, zb toma el valor de a

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

Cuando llega el flanco activo de reloj, znb toma el valor de q y, en el siguiente flanco, es cuando toma el valor de a

- **Resumen**
  - **Aritméticos**: +, -, *, /, %,
    Comparación: ==, !=, >, <, >=, <=
  - **Lógicos**: &&, ||, !
  - **Bit a bit**: &, |, ^, ^~, ~^, ~ (incluyendo reducción).
  - **Desplazamiento**: <<, >>, >>>
  - **Concatenación y replicación**: {}, {n{}}
  - **Condicional ternario**: ? :
  - **Asignación**: bloqueante (=) y no bloqueante (<=)

Esquema de la unidad

- Aspectos básicos
- Tipos de datos
- Vectores y arrays
- Literales y bases
- Procedimientos:
  - sentencias condicionales
  - bucles
- Operadores
- Parámetros
- Máquinas de estado
- Directivas define e include
- Simulación funcional

Constantes en Verilog

- *parameter*: constante válida en un módulo que permite crear bloques genéricos personalizables, su valor puede modificarse al instanciar el módulo
- *localparam*: constante cuyo valor es fijo y no se puede modificar
  - Permiten definir el ancho de un bus, el tamaño de un contador, los elementos de un vector, …
  - Se les asigna determinados valores que pueden ser modificados al instanciar el módulo, sin modificar su código.
  - Su uso facilita la reutilización y adaptación del módulo a distintos contextos
  - Se recomienda nombrarlos con mayúsculas para facilitar la legibilidad

Módulos parametrizables

- Ejemplo (I):

```
module mux #(parameter WIDTH = 8) (
        input wire [WIDTH-1:0 ] a,b,
        input wire sel,
        output reg [WIDTH-1:0] op
         );

        always @*
            if (sel)
                op=a;
            else
                op=b;
endmodule
```

Definimos variables de 8 y 4 bits para comparar las instanciaciones

```
wire [7:0] a8, b8;
reg [7:0] op8;
wire [3:0] a4, b4;
reg [3:0] op4;
wire sel;

//instanciación por defecto: WIDTH se mantiene a 8
mux mux8 (.a(a8),.b(b8),.sel,.op(op8));

// Instanciación con un ancho diferente
mux #(.WIDTH(4)) mux4 (.a(a4),.b(b4),.sel,.op(op4));
```

Módulos parametrizables

- Módulos parametrizables, ejemplo (II):

```
module contador #(parameter WIDTH = 8) (
            input wire clk,
            input wire rst,
            output reg [WIDTH-1:0] count
             );
  // Lógica del contador
always @(posedge clk)
     if (rst)
       count <= 'd0;
     else
       count <= count + 1;
endmodule
```

```
// Instanciación con un tamaño diferente

 contador #(.WIDTH(16)) contador16bits (
       .clk(clk),
       .rst(rst),
       .count(count16)
 );
```

Módulos parametrizables

- Sintaxis:
  - Declaración:

```
module nombre_modulo  #(parameter parametro1 = valor1,
                        parametro2 = valor2, ...) (...);
```

  - Redefinición al instanciar el módulo:

```
nombre_modulo  #(.parametro1(valor1),
                 .parametro2(valor2), ...)
                 nombre_instancia (...);
```

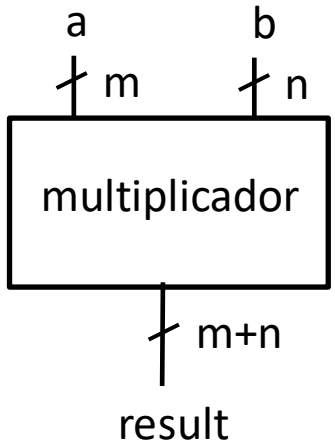
  - Se puede usar conexión posicional.

```
nombre_modulo #(valor1,valor2, ...)
                 nombre_instancia (...);
```

Constantes en Verilog

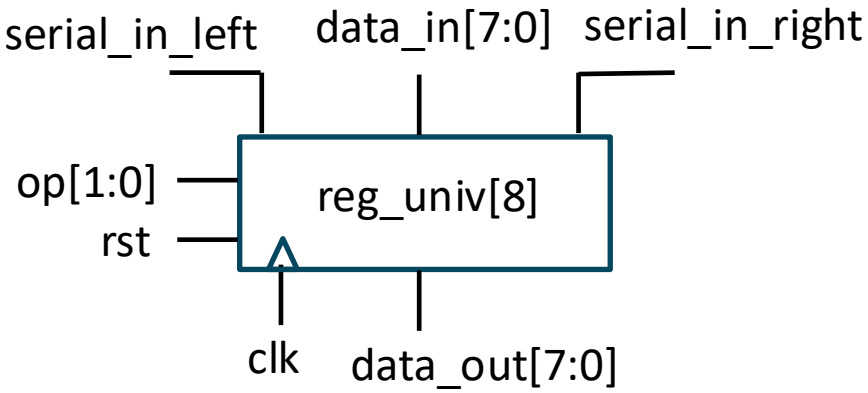
- localparam define una constante invariable
  - A diferencia de parameter su valor no puede ser redefinido al instanciarse
  - Sin embargo, puede estar definido a partir de los valores de otras constantes tipo parameter

```
module multiplicador
        #(parameter WIDTH_A = 4, WIDTH_B = 4)

         (input wire [WIDTH_A-1:0] a,
          input wire [WIDTH_B-1:0] b,
          output reg [WIDTH_OP-1:0] result);

          localparam WIDTH_OP = WIDTH_A + WIDTH_B;
          assign result = a * b;

endmodule
```



Ejemplo: registro universal

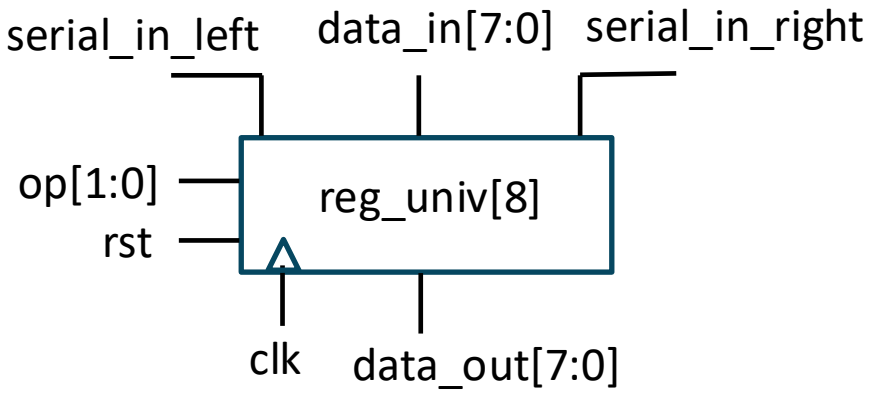
- El registro universal puede realizar todas las operaciones:
  - desplazamiento a derecha e izquierda,
  - carga en paralelo (operación de escritura)
  - inhibición (guardar el dato)
  - puesta a 0



op[1:0]	operacion
0 0	inhibición
0 1	shift left
1 0	shift right
1 1	escritura

Ejemplo: registro universal

```
module reg_univ #( parameter WIDTH = 8)(
                input wire clk, rst,
                input wire [1:0] op,
                input wire serial_in_left,
                input wire serial_in_right,
                input wire [WIDTH-1:0] data_in,
                output reg [WIDTH-1:0] data_out
                );
always @(posedge clk, posedge rst) begin
                if (rst)
                    data_out <= ‘0;
                else
                    case (op)
                        2'b00: data_out <= data_out;
                        2'b01: data_out <= {data_out[WIDTH-2:0], serial_in_right};
                        2'b10: data_out <= {serial_in_left, data_out [WIDTH-1:1]};
                        2'b11: data_out <= data_in;
                       default: data_out <= ‘0;
                    endcase
        end
endmodule
```




```
begin
    data_out <= data_out<<1;
    data_out[0] <= serial_in right
end
```

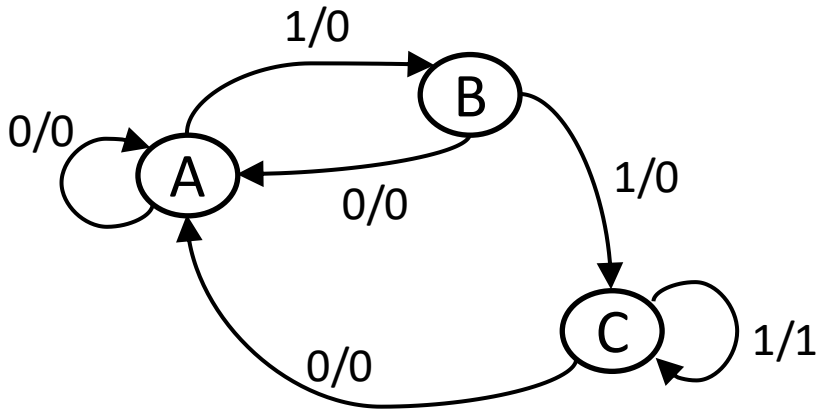
Esquema de la unidad

- Aspectos básicos
- Tipos de datos
- Vectores y arrays
- Literales y bases
- Procedimientos:
  - sentencias condicionales
  - bucles
- Operadores
- Parámetros
- Máquinas de estado
- Directivas define e include
- Simulación funcional

Descripción de máquinas de estado

- Se utilizará una estructura general del código en la que hay 2 tipos de procesos
  - Un proceso always para establecer las condiciones de cambio de estado
  - Uno o varios procesos always para calcular el próximo estado y las salidas. También pueden usarse sentencias assign.

- Se incorpora reset asíncrono (puede ser síncrono) para llevar a la máquina a un estado inicial conocido.



El estado inicial que conviene definir en este caso es el estado A

Descripción de máquinas de estado

```
module detector_3_unos (
   input wire clk,
   input wire rst,
   input wire in,
   output wire z );

   parameter  A=2'b00, B=2'b01, C=2'b10;
   reg [1:0] state, next_state;

   always @(posedge clk or posedge rst) begin
      if (rst)
         state <= A;
      else
         state <= next_state;
   end
```

se declaran entradas y salidas

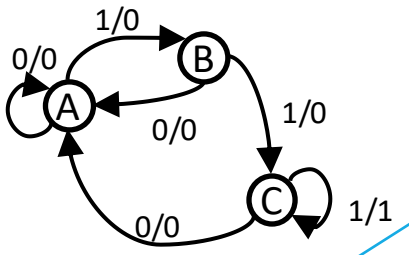
se asignan códigos a los estados y se declaran las variables de estado

procedimiento de cambio de estado:

- valor 1 en rst lleva a la máquina al estado inicial

- un flanco positivo en clk lleva a cambio de estado

Descripción de máquinas de estado



procedimiento combinacional de cálculo del próximo estado

```
always begin
      case (state)
         A: next_state = in ? B : A;
         B: next_state = in ? C : A;
         C: next_state = in ? C : A;
         default: next_state = A;
      endcase
   end
assign z = (state ==C && in);
endmodule
```

podría realizarse con if/else en lugar de la sentencia condicional

```
case (state)
        A: if (in)
              next_state =  B;
           else
              next_state = A;
        B:  if (in)
              next_state =  C;
           else
              next_state = A;
        C: if (in)
              next_state =  C;
           else
              next_state = A;
        default: next_state =  C;
endcase
```

cálculo de la salida podría realizarse con otro always en lugar del assign

```
always
      if (in==1 && state == C)
            z=1;
      else
            z=0;
```

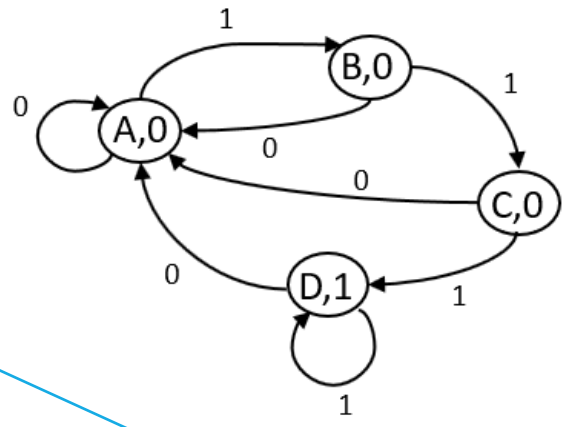
Descripción de máquinas de estado

```
module detector_3_unos (
   input wire clk,
   input wire rst,
   input wire in,
   output wire z );

    parameter  A=2'b00, B=2'b01, C=2'b10;
    reg [1:0] state, next_state;

   always @(posedge clk or posedge rst) begin
      if (rst)
         state <= A;
      else
         state <= next_state;
   end
```

El bloque de declaración del módulo no cambia

La misma especificación puede realizarse mediante una máquina de Moore

En este caso la salida está asociada al estado presente ya que no depende del valor de la entrada de forma directa

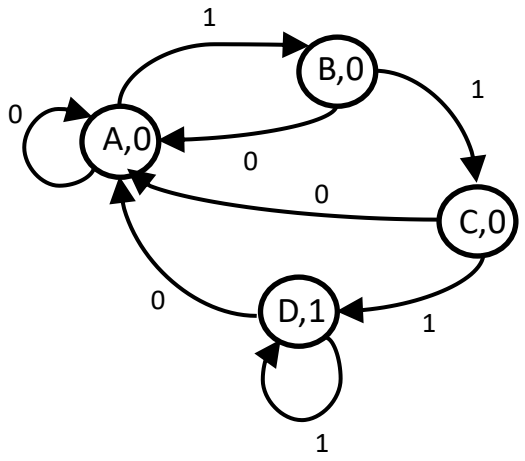


El procedimiento de cambio de estado, tampoco cambia:

- valor 1 en rst lleva a la máquina al estado inicial

- un flanco positivo en clk lleva a cambio de estado

Descripción de máquinas de estado




```
always begin
      case (state)
         A: next_state = in ? B : A;
         B: next_state = in ? C : A;
         C: next_state = in ? D : A;
         D: next_state = in ? D : A;
         default: next_state = A;
      endcase
   end
   assign z = (state ==D);
endmodule
```

El cálculo de la salida podría realizarse con otro always en lugar del assign

```
always
      if (state == D)
            z=1;
      else
            z=0;
```

Esquema de la unidad

- Aspectos básicos
- Tipos de datos
- Vectores y arrays
- Literales y bases
- Procedimientos:
  - sentencias condicionales
  - bucles
- Operadores
- Parámetros
- Máquinas de estado
- Directivas define e include
- Simulación funcional

La directiva `define

- Se puede usar la directiva `define para definir constantes y macros que se reemplazan antes de la compilación del código.
- Permite cambiar fácilmente los valores de los símbolos sin tener que editar el código.
- Sintaxis: `define SIMBOLO valor
  - Ejemplo:

```
`define CLOCK_PERIOD 10  // Define una constante: el período de reloj
module testbench;
   reg clk;
    . . .
   initial begin
      clk = 0;
      forever #(`CLOCK_PERIOD / 2) clk = ~clk;  // Generación del reloj
   end
 . . .
endmodule
```

La directiva `include

- Se puede usar la directiva `include para incluir archivos fuente externos en un código.
  - Se usa principalmente en testbenches para organizar mejor el código y reutilizar definiciones evitando el tener que repetirlas en múltiples archivos.
- Sintaxis: `include "archivo.sv"
  - Ejemplo:

```
// constantes.sv
`define WIDTH 8
`define NUM_ENTRADAS 16
```

```
module test;
  `include "constantes.sv"  // Incluir archivo de constantes

  reg [`WIDTH-1:0] data[`NUM_ENTRADAS-1:0];

  initial begin
    $display("Ancho de datos: %0d", `WIDTH);
    $display("Número de entradas: %0d", `NUM_ENTRADAS);
  end
endmodule
```

Esquema de la unidad

- Aspectos básicos
- Tipos de datos
- Vectores y arrays
- Literales y bases
- Procedimientos:
  - sentencias condicionales
  - bucles
- Operadores
- Parámetros
- Máquinas de estado
- Directivas define e include
- Simulación funcional

Simulación funcional en Verilog

- Pasos para la simulación:
  - se debe disponer de la descripción del módulo a simular en Verilog
  - hay que crear un módulo especial llamado testbench que:
    - incluirá una instancia del circuito a probar (dut: design under test)
    - generará los estímulos necesarios para probar su funcionamiento
  - el testbench puede incluir bloques procedimentales: initial, always, …
  - para observar las formas de onda durante la simulación, se usan comandos específicos:
    - $dumpfile, $dumpvars, $dumpon, $dumpoff…

- Nota: algunos entornos de diseño como ISE o Vivado no requieren estos comandos pues usan su propio formato de volcado y permiten añadir las ondas manualmente al simular
- Comando $dumpfile:
  - Sintaxis: $dumpfile("nombre.vcd")
  - Este comando indica el nombre del archivo donde se guardará la información de la simulación, generalmente en formato VCD (Value Change Dump). Este contendrá todos los cambios de las señales que se especifiquen.
- Comandos $dumpon y $dumpoff:
  - Permiten iniciar y detener el registro de datos durante la simulación para generar un archivo vcd más pequeño

- Comando $dumpvars:
  - Sintaxis: $dumpvars(n, modulo)
  - Con este comando se seleccionan las variables y señales que se van a volcar en el archivo definido con $dumpfile.
  - Con $dumpvars(0, top) se vuelcan todas las variables de todas las instancias y módulos que forman parte de la simulación, es decir, se realiza un volcado global
  - También se puede limitar el volcado especificando un módulo concreto y el número de niveles (n) de jerarquía al que se quiere descender: $dumpvars(n, instancia)
  - $dumpvars; es equivalente a $dumpvars(0, top) en Verilog

Organización de un testbench

El testbench es un módulo sin entradas ni salidas

Se declaran señales de prueba para conectarlas al módulo a probar

Se instancia el módulo y se conectan a él las señales de prueba

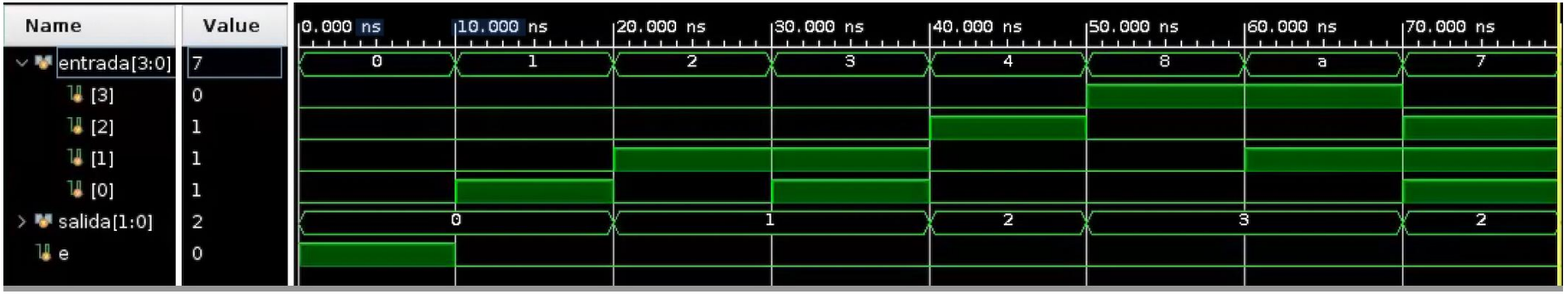
En este bloque initial se incorporan los comandos para que se generen formas de onda.

```
module tb_codificador_de_prioridad;

 reg [3:0] entrada;
 wire [1:0] salida;
 wire       e;

 codificador_de_prioridad dut (
  .entrada(entrada),
  .salida(salida),
  .e(e)
 );

initial begin
  $dumpfile("codificador_de_prioridad.vcd");
  $dumpvars(0, tb_codificador_de_prioridad);
 end

 initial begin
   entrada = 4'b0000;
   #10;
  entrada = 4'b0001;
   #10;
   entrada = 4'b0010;
   #10;
   entrada = 4'b0011;
   #10;
   entrada = 4'b0100;
   #10;
   entrada = 4'b1000;
   #10;
   entrada = 4'b1010;
   #10;
   entrada = 4'b0111;
   #10;
   $finish;
 end
endmodule
```

En este bloque initial se fijan los valores de las entradas y se establece el final de la simulación.

Resultado de la simulación

- El simulador nos muestra entradas y salidas para los casos probados: 0000,0001,0010,0011,0100,1000,1010,0111
- Cada valor se mantiene durante 10ns (ver más adelante unidades)
- La salida nos muestra en cada caso el código de la entrada más prioritaria que valga 1.
- El bus de entrada ha sido desplegado para apreciar mejor las entradas
- El bus de salida se muestra en base 10



Organización de un testbench

Otra opción para la organización del testbench

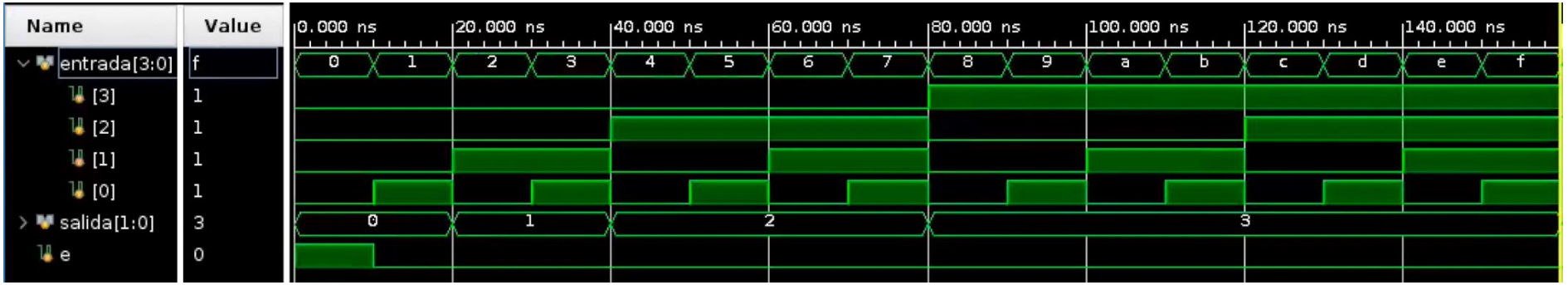
```
module tb_codificador_de_prioridad;

 reg [3:0] entrada;
 wire [1:0] salida;
 wire       e;

 codificador_de_prioridad dut (
  .entrada(entrada),
  .salida(salida),
  .e(e)
 );
initial begin
  $dumpfile("codificador_de_prioridad.vcd");
  $dumpvars(0, tb_codificador_de_prioridad);
 end

initial begin
    entrada = 4'b0000;
    #160;
    $finish;
end

always begin
     #10;
     entrada = entrada + 1'b1;
 end
endmodule
```

En este bloque initial solo se proporciona el valor inicial de las entradas y se establece el final de la simulación

En este bloque always se cubren todas las posibilidades de entrada.

Resultado de la simulación

- El simulador nos muestra entradas y salidas para todos los casos posibles de entrada (son 16): 0000,0001,0010, …, 1101,1110,1111
- Cada valor se mantiene durante 10ns (ver más adelante unidades)
- La salida nos muestra en cada caso el código de la entrada más prioritaria que valga 1.
- El bus de entrada ha sido desplegado para apreciar mejor las entradas
- El bus de salida se muestra en base 10



Unidades de tiempo

- El tiempo se puede expresar con el símbolo # seguido de un valor numérico entero o en punto fijo seguido (sin espacio) por una unidad de tiempo (fs ps ns us ms s)
  - Ejemplo: #10ns

- Existe una directiva `timescale que fija las unidades de tiempo y la precisión en la simulación
  - Sintaxis: ‘timescale unidad/precisión
    - Ejemplo: ‘timescale 1ns/100ps (usar 1, 10, 100)

- Si al expresar el tiempo no se incluyen unidades se entiende que se usa la indicada en la directiva `timescale
  - El simulador redondea las cantidades a la precisión establecida.

Salida por pantalla

- Se puede imprimir información durante la simulación mediante las funciones $display y $monitor. Ambas escriben con formato.

- $display imprime el mensaje una sola vez: en el punto donde se incluye.
  - Ejemplo:
    - $display ("Valor de la señal A = %0d", A);

- $monitor se escribe una sola vez, pero imprime el mensaje cada vez que cambia alguna de las variables incluidas en su lista.
  - Ejemplo:
    - $monitor ("Tiempo: %0d , A = %0d , B = %0d", $time, A, B);

Salida por pantalla: $display, ejemplo

```
`timescale 1ns/1ns
module tb_codificador_de_prioridad;
                . . .
                . . .
initial begin
   entrada = 4'b0000;
   $display("Tiempo %0d: entrada = %b, salida = %b, e = %b", $time, entrada, salida, e);
   #10;
  entrada = 4'b0001;
  $display("Tiempo %0d: entrada = %b, salida = %b, e = %b", $time, entrada, salida, e);
   #10;
   entrada = 4'b0010;
 $display("Tiempo %0d: entrada = %b, salida = %b, e = %b", $time, entrada, salida, e);
   #10;
  entrada = 4'b0011;
  $display("Tiempo %0d: entrada = %b, salida = %b, e = %b", $time, entrada, salida, e);
   #10;
  entrada = 4'b0100;
  $display("Tiempo %0d: entrada = %b, salida = %b, e = %b", $time, entrada, salida, e);
   #10;
  entrada = 4'b1000;
  $display("Tiempo %0d: entrada = %b, salida = %b, e = %b", $time, entrada, salida, e);
  #10;
  entrada = 4'b1010;
  $display("Tiempo %0d: entrada = %b, salida = %b, e = %b", $time, entrada, salida, e);
  entrada = 4'b0111;
  #10;
  $display("Tiempo %0d: entrada = %b, salida = %b, e = %b", $time, entrada, salida, e);
                . . .
                . . .
```

Incluimos la directiva `timescale para fijar las unidades de tiempo y la precisión

```
Tiempo 10: entrada = 0000, salida = 00, e = 1
Tiempo 20: entrada = 0001, salida = 00, e = 0
Tiempo 30: entrada = 0010, salida = 01, e = 0
Tiempo 40: entrada = 0011, salida = 01, e = 0
Tiempo 50: entrada = 0100, salida = 10, e = 0
Tiempo 60: entrada = 1000, salida = 11, e = 0
Tiempo 70: entrada = 1010, salida = 11, e = 0
Tiempo 80: entrada = 0111, salida = 10, e = 0
testbench.sv:66: $finish called at 80 (1ns)
```

La salida por pantalla muestra los resultados con el formato solicitado en el testbench

Salida por pantalla: $monitor, ejemplo

```
`timescale 1ns/1ns
module tb_codificador_de_prioridad;
                . . .
                . . .
initial begin
  $monitor ("Tiempo %0d: entrada = %b, salida = %b, e = %b", $time, entrada, salida, e);
  entrada = 4'b0000;
  #10;
  entrada = 4'b0001;
   #10;
  entrada = 4'b0010;
   #10;
  entrada = 4'b0011;
   #10;
  entrada = 4'b0100;
  #10;
  entrada = 4'b1000;
  #10;
  entrada = 4'b1010;
   #10;
  entrada = 4'b0111;
                . . .
                . . .
```

Con una única línea conseguimos el mismo efecto

```
Tiempo 10: entrada = 0000, salida = 00, e = 1
Tiempo 20: entrada = 0001, salida = 00, e = 0
Tiempo 30: entrada = 0010, salida = 01, e = 0
Tiempo 40: entrada = 0011, salida = 01, e = 0
Tiempo 50: entrada = 0100, salida = 10, e = 0
Tiempo 60: entrada = 1000, salida = 11, e = 0
Tiempo 70: entrada = 1010, salida = 11, e = 0
Tiempo 80: entrada = 0111, salida = 10, e = 0
testbench.sv:26: $finish called at 80 (1ns)
```

Simulación (ejemplo I)

- Contador módulo 256 y testbench:

```
module simple_counter (
      input wire clk, rst,
      output reg [7:0] count );

always @(posedge clk or posedge rst)
    begin
       if (rst)
         count <= 8'd0;
       else
         count <= count + 1;
    end
endmodule
```

```
module test_repeat_loop;
    reg clk,rst;
    wire [7:0] count;

simple_counter uut (
       .clk (clk),
       .rst (rst),
       .count (count ));


always begin
       #5 clk = ~clk;
       end

initial begin
       clk=0;
       rst = 1;
       #12;
       rst = 0;
       repeat (20)  @(posedge clk);
       $finish;
  end
endmodule
```

La organización es la ya vista:
- creación de un módulo,
- declaración de señales,
- instanciación y conexión del módulo a probar

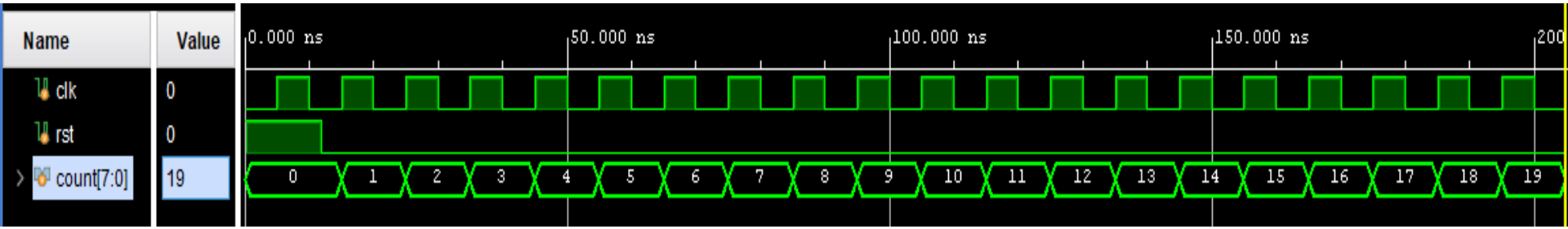
generación de la señal de reloj

generación de estímulos

con esta línea esperamos 20 ciclos de reloj

Resultado de la simulación

- El simulador muestra las entradas: *clk* y *rst* y la salida de 8 bits del contador *count*
- El reloj es de periodo 10 tal y como se ha definido.
- En la salida se puede apreciar el *reset* inicial, y durante 20 ciclos de reloj la evolución del estado del contador en decimal: 0->19



Simulación (ejemplo II)

- Contador módulo 8 con acarreo:

```
module mod8_counter (
      input wire clk, rst,
      output reg [2:0] count,
      output reg cy);

always@(posedge clk or posedge rst)
     begin
        if (rst)
          count <= 3'd0;
       else
         count <= count + 1;
     end
    assign cy = (count == 3'b111);
endmodule
```

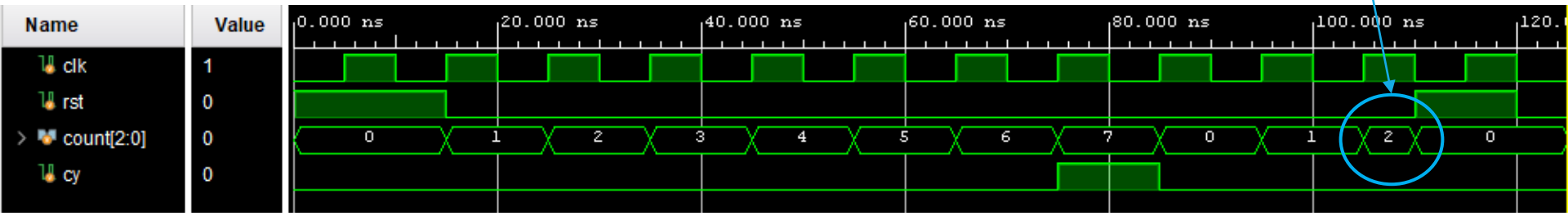
```
module test_counter;
     reg clk,rst;
     wire [2:0] count;
     wire cy;

mod8_counter uut (
       .clk (clk),
       .rst (rst),
       .count (count),
       .cy(cy));

always begin
       #5 clk = ~clk;
       end
```

```
initial begin
          clk=0;
          rst = 1;
          #15;
          rst = 0;
          repeat (10) @(negedge clk);
          rst=1;
          @(negedge clk);
          rst=0;
          repeat(5) @(posedge clk)
          $finish;
  end
endmodule
```

Podemos usar referencias a ambos flancos de reloj

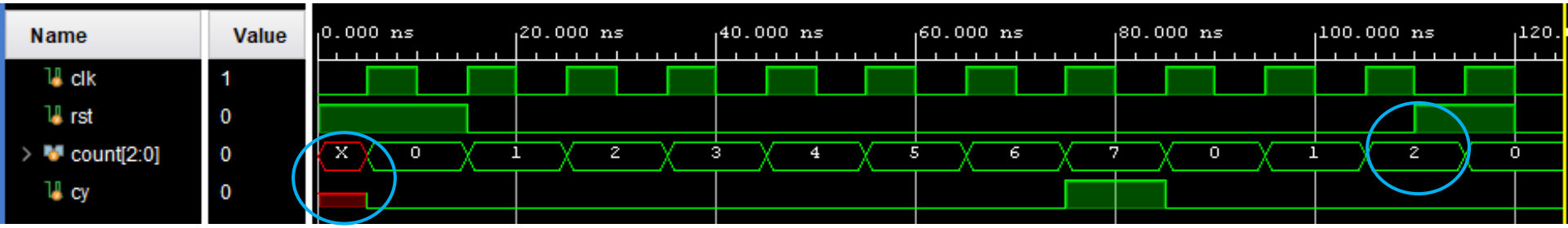
Resultado de la simulación

- El simulador muestra las entradas *clk* y *rst* y las salidas: los 3 bits del contador (*count*) y la salida de carry (*cy)*
- En la salida se aprecian:
  - el *reset* inicial y el ciclo completo de cuenta
  - la activación de la salida de *carry* en el estado 7 (último estado de cuenta)
  - otro *reset* que lleva al contador de nuevo al valor inicial (0)
- Es importante destacar que el reset se produce de forma asíncrona, sin necesidad de esperar a la señal de reloj, haciendo que el estado 2 no dure un ciclo de reloj.
  - Esto es debido a que dicha señal se ha incluido en la lista de sensibilidad del always al definir el módulo.



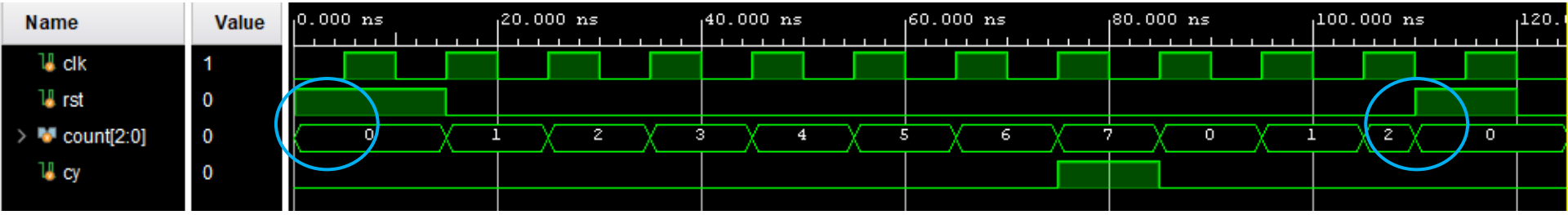
Resultado de la simulación

- Si se desea que la señal de *reset* sea síncrona basta con excluirla de la lista de sensibilidad.
- La respuesta en este caso es:



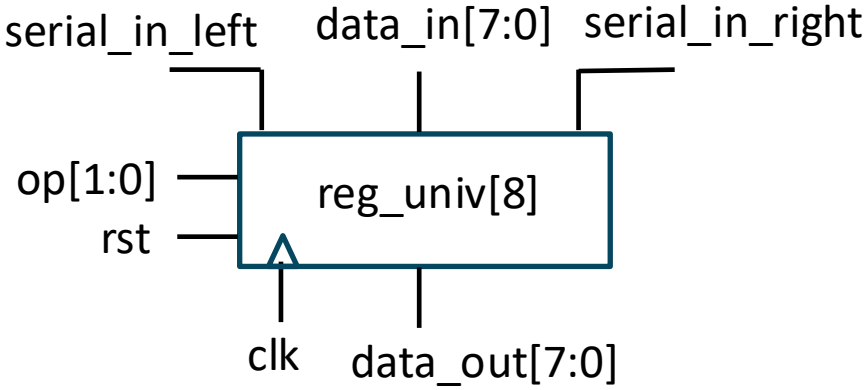
Con always @(posedge clk)
rst es síncrona

Con always @(posedge clk or posedge rst)
rst es asíncrona



- El registro universal puede realizar todas las operaciones:
  - desplazamiento a derecha e izquierda,
  - carga en paralelo (operación de escritura)
  - inhibición (guardar el dato)
  - puesta a 0

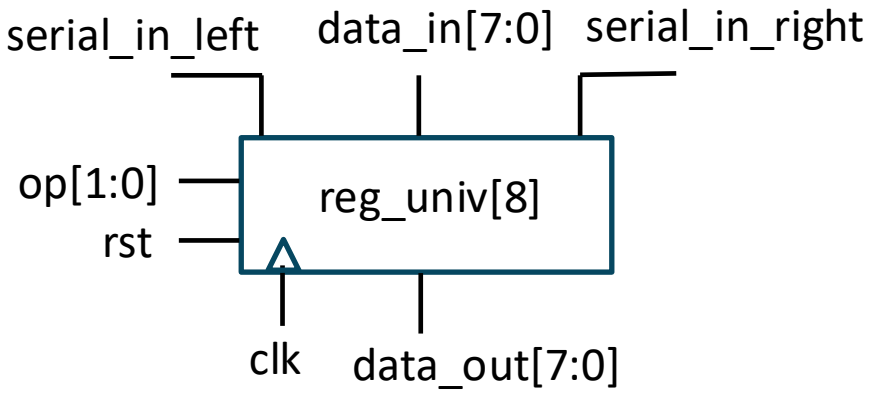
Recordamos el módulo



op[1:0]	operacion
0 0	inhibición
0 1	shift left
1 0	shift right
1 1	escritura

Simulación(ejemplo III)

```
module reg_univ #( parameter WIDTH = 8)(
                input wire clk, rst,
                input wire [1:0] op,
                input wire serial_in_left,
                input wire serial_in_right,
                input wire [WIDTH-1:0] data_in,
                output reg [WIDTH-1:0] data_out
                );
always @(posedge clk, posedge rst) begin
                if (rst)
                    data_out <= ‘0;
                else
                    case (op)
                        2'b00: data_out <= data_out;
                        2'b01: data_out <= {data_out[WIDTH-2:0], serial_in_right};
                        2'b10: data_out <= {serial_in_left, data_out [WIDTH-1:1]};
                        2'b11: data_out <= data_in;
                       default: data_out <= ‘0;
                    endcase
        end
endmodule
```



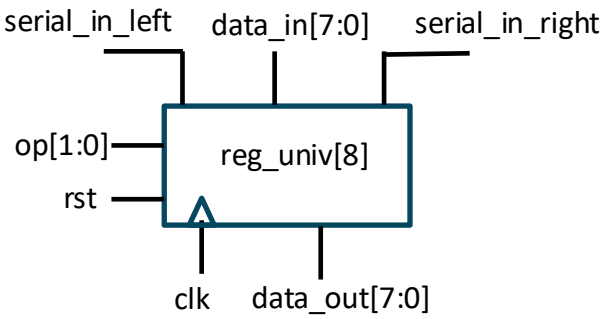
Recordamos el módulo

```
begin
    data_out <= data_out<<1;
    data_out[0] <= serial_in right
end
```

Simulación(ejemplo III)

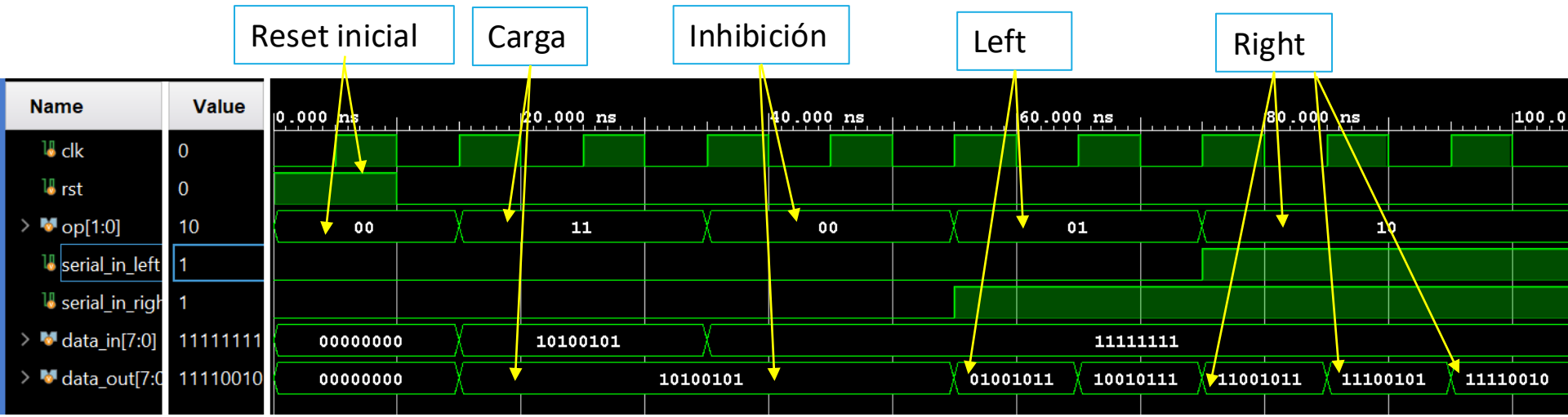
```
module tb_reg_univ;
  parameter WIDTH = 8;

  reg clk;
  reg rst;
  reg [1:0] op;
  reg serial_in_left;
  reg serial_in_right;
  reg [WIDTH-1:0] data_in;
  wire [WIDTH-1:0] data_out;

reg_univ #(.WIDTH(WIDTH)) dut (
    .clk(clk),
    .rst(rst),
    .op(op),
    .serial_in_left(serial_in_left),
    .serial_in_right(serial_in_right),
    .data_in(data_in),
    .data_out(data_out)
  );

initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end
```

```
initial begin
    $dumpfile ("dump.vcd");
    $dumpvars (0, tb_reg_univ);
end

initial begin
    rst = 1;
    op = 2'b00;
    data_in = '0;
    serial_in_left = 0;
    serial_in_right = 0;

    #10;
    rst = 0; //desactiva reset

    @(posedge clk);
    op = 2'b11;        // carga en paralelo
    data_in = 8'hA5;   // Valor de prueba
    @(posedge clk);

    @(posedge clk);
    op = 2'b00;    // inhibición
    data_in = 8'hFF; // data_in no afecta
    @(posedge clk);
```




```
    @(posedge clk);
    op = 2'b01;         // Shift left
    serial_in_right = 1;   // Bit que entra
    @(posedge clk);


    @(posedge clk);
    op = 2'b10;         // Shift right
    serial_in_left = 1;    // Bit que entra
    @(posedge clk);

    // Finalización de la simulación
    #20;
    $finish;
  end
endmodule
```

Test del registro universal: resultado



op[1:0]	operacion
0 0	inhibición
0 1	shift left
1 0	shift right
1 1	escritura