

Tema 3. Lenguajes de descripción de hardware orientados a la síntesis lógica

Prof. Juan J. Pombo García

Introducción

- HDL (*hardware description language*)
- Describe *hardware* en forma de texto. Pero va más allá de ser una descripción tipo *Netlist*, en la que se describe un circuito por su conexión. Ahora, casi siempre, se intenta describir a través de su funcionamiento.
- Desde este punto de vista NO es “*software*”
- Es una alternativa a la representación con esquemáticos.
 - A partir de un cierto nivel de complejidad el sistema con esquemáticos no es suficientemente práctico, incluso cuando se utilizan bloques funcionales
- Además de describir el hardware también se contempla la simulación.
- A partir de la expresión en el HDL se han creado herramientas para automatizar la “**síntesis**” del circuito en determinadas plataformas: diseño independiente de la tecnología (FPGA, ASIC, etc..., y sin especificar marcas)

Introducción

- Existen dos variantes principales de HDL's:
 - Verilog
 - No es lo mismo que la subvariante System Verilog, la más moderna
 - VHDL
- Los HDL, a diferencia de un lenguaje de programación, incluyen la noción del cambio en el tiempo de las señales.
- A partir del lenguaje la síntesis se realiza utilizando unidades llamadas “primitivas de hardware”.
- La mayor parte de las expresiones se ejecutan concurrentemente.

Introducción

- Usaremos para comenzar Verilog.
 - Se puede expresar lo mismo en Verilog que en VHDL
 - Quizá Verilog sea más simple para comenzar por tener una sintaxis más compacta y no ser tan fuertemente tipado. VHDL puede ser más sencillo de leer.
 - Casi todos los entornos soportan ambos tipos de descripciones
 - Tanto Verilog como VHDL están considerados “estándares” y están bien soportados. Incluso se pueden combinar en un mismo proyecto.
- Puerta AND en Verilog

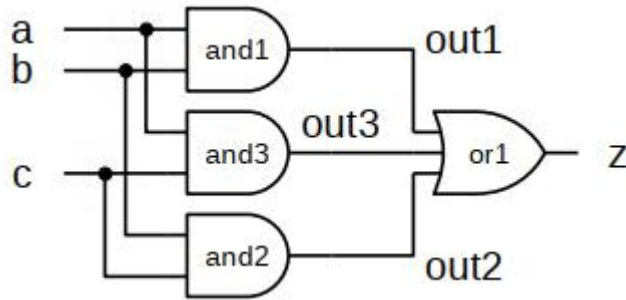
```
module AND_2(output Y, input A, B);  
  and(Y, A, B);  
endmodule;
```

Niveles de abstracción

- Existen varias maneras de expresar un circuito (niveles de abstracción)

- Nivel de puerta** (tb. Llamado modelo estructural)
 - Se describe el circuito con primitivas lógicas

Esto es una
instanciación de
un módulo, lo
veremos en
detalle más
adelante

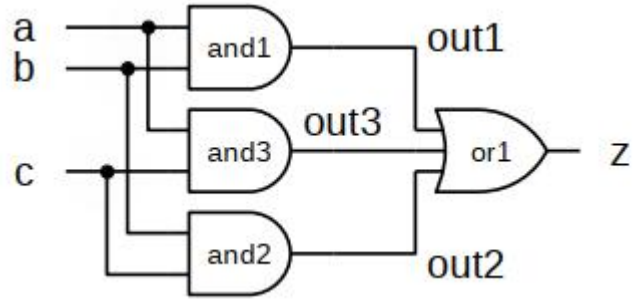


```
wire out1, out2, out3;  
  
and and1 (out1, a, b);  
and and2 (out2, b, c);  
and and3 (out3, a, c);  
or or1 (z, out1, out2, out3);
```

- El más intuitivo pero es el menos útil. No ganamos mucho respecto a los esquemáticos. Para circuitos complejos es muy costoso escribir todo a este nivel.

Niveles de abstracción

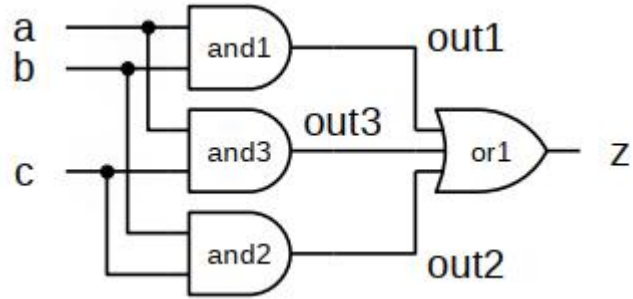
- **Nivel funcional** (tb. Llamado asignación continua)
 - Se describe con expresiones lógicas



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

Niveles de abstracción

- **Nivel Procedimental** (tb. Bloques “*always*”)
 - Estructura más similar al software. Usa expresiones de control.
 - Es el más útil cuando se alcanza cierto nivel de complejidad



```
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;
```

Estructura general de un diseño Verilog

Sentencia “module”:

Module <nombre> <definición de señales de interfaz>;

Entradas/salidas

Input/output <width> señal; ← También llamados puertos

Código del diseño

.....

Sentencia “endmodule”

endmodule

```
//Ejemplo de un circuito  
//con una única puerta EXOR
```

```
`timescale 1ns/1ps
```

```
module mi_exor(  
    input A,  
    input B,  
    output F );
```

```
    assign F = A ^ B;
```

```
endmodule
```


Representación numérica en Verilog

`<Tamaño>'<base><valor>`

La representación interna es en complemento a 2 para los números negativos.

Ejemplos:

se almacena como:

1 00000000000000000000000000000001 (32 bits)

8'hAA 10101010

8'b1 00000001 (se rellenan los 0's a la izquierda)

-8'd2 11111110 (el signo va antes del tamaño, aquí es un -2 en base decimal)

Tipos de datos

Básicamente existen 2 tipos de datos para trabajar en Verilog:

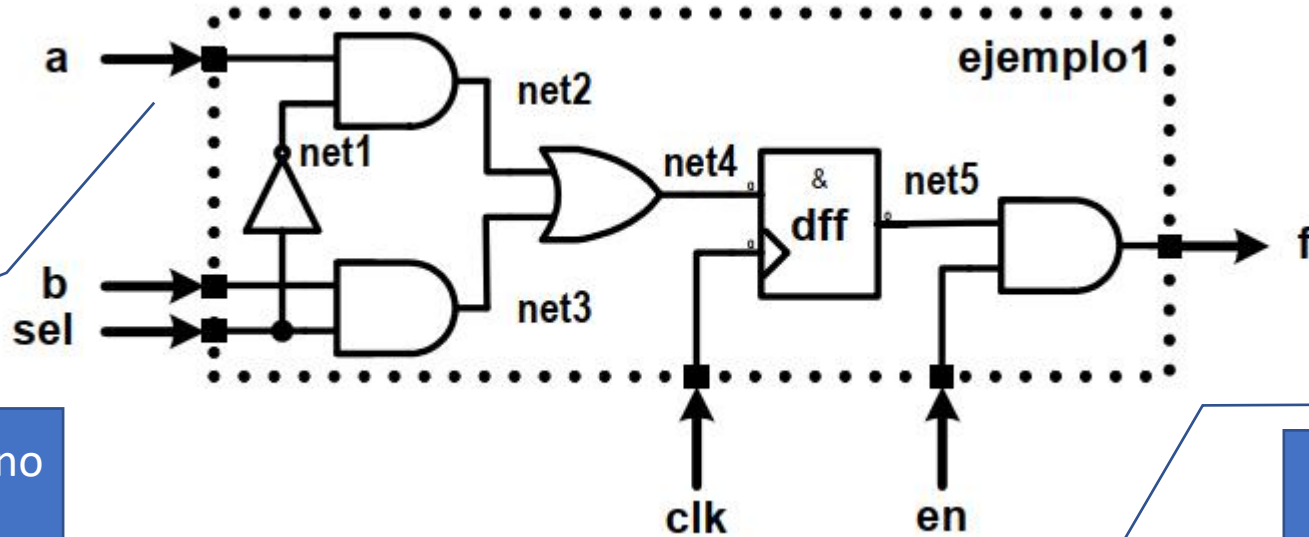
Nets.

Conexiones entre componentes. El más usual y que consideraremos por ahora es el “*wire*”

Registers.

Variables en las que podemos almacenar información
Los más básicos son el “*reg*” y el “*integer*”.

Ejemplo: Definición de nodos *wire* y *reg*



1. En la entrada no definimos los tipos. Se asume que es de tipo *wire*

En verilog se distinguen mayúsculas/minúsculas (*case sensitive*)

```
module mi_ejemplo1(a, b, sel, clk, en, f);
//entradas y salidas
input    a, b, sel, clk, en;
output   f;
wire     f;
//nodos internos de trabajo
reg      net5;
wire     net1, net2, net3, net4;
// ahora vendría el diseño
```

Usar un *reg* no quiere decir que necesariamente sea un registro en la implementación. Puede ser sólo algo usado para describir el comportamiento

Concepto de proceso

Aquí aparece una de las diferencias respecto a los lenguajes procedurales habituales en informática: el lenguaje introduce el concepto de ejecución de procesos en paralelo.

Toda descripción debe estar en el contexto de un proceso.

Tipos de procesos:

Initial. Se ejecuta una sólo vez en el tiempo cero. NO ES SINTETIZABLE. Se usa para ejecutar *testbenches*.

Si existiesen varios bloques Initial se ejecutarían concurrentemente.

Always, Se ejecuta continuamente como un bucle. Es sintetizable. Se ejecutará en determinados momentos que puede ser determinado por una señal temporizada (periódica) o ante un evento.

El conjunto de eventos que dispara un proceso se denomina **lista sensible**.

```
always <temporización>
```

```
always @<lista sensible>
```

Procesos Initial (ejemplo)

```
reg clk,reset;  
reg enable,data;  
initial  
begin  
    clk = 0;  
    reset = 0;  
    enable = 0;  
    data = 0;  
end
```

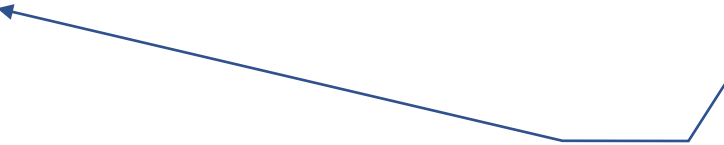
En Initial solo se puede asignar sobre variables tipo *reg*, nunca *wire* (esto es general para las asignaciones)

En general una asignación en Verilog es del tipo:

variable = función(wire, reg, constante)

Procesos always (ejemplo)

```
always @(a or b or sel)
begin
    if (sel == 1)
        y = a;
    else
        y = b;
end
```



También puede
ser una lista
separada por “,”

Cuando sólo hay una estructura de control los delimitadores “begin” y “end” pueden suprimirse.

Las variables a, b y sel constituyen la **lista sensible**. El circuito se activa ante cambios en la lista sensible exclusivamente

Asignación continua

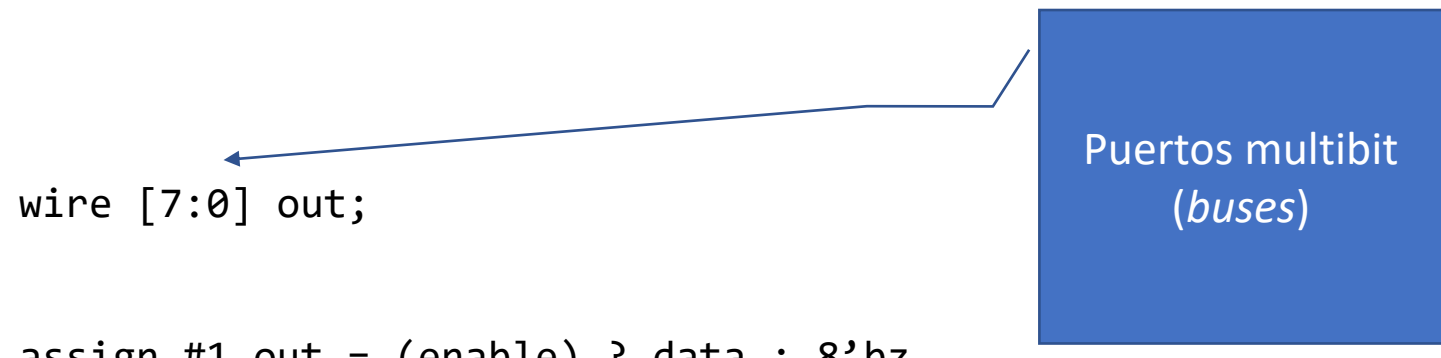
Modelan lógica combinacional.

Las asignaciones se realizan FUERA de los procesos (ni bloques *always* ni *initial*)

La variable ahora sí puede ser tipo *net* (*wire*, por ejemplo)

```
wire [7:0] out;
```

```
assign #1 out = (enable) ? data : 8'bz
```



Puertos multibit
(*buses*)

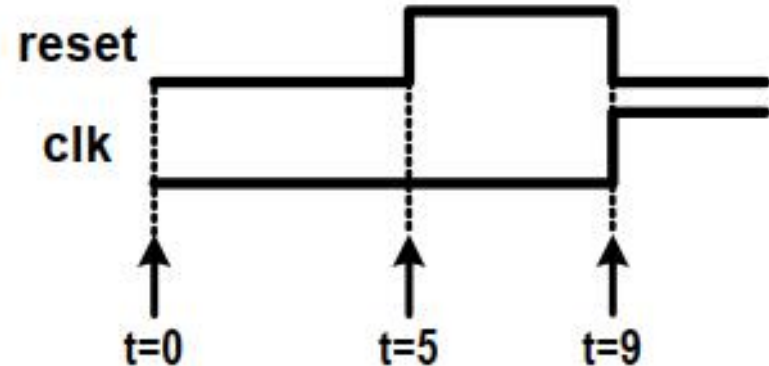
Temporizaciones

Nos permitirán indicar el instante en que se producen las asignaciones.

Para ello se introduce el concepto de **retardo**.

```
initial
begin
  clk = 0;
  reset = 0;
  #5 reset = 1;
  #4 clk = 1;
  reset = 0;
end
```

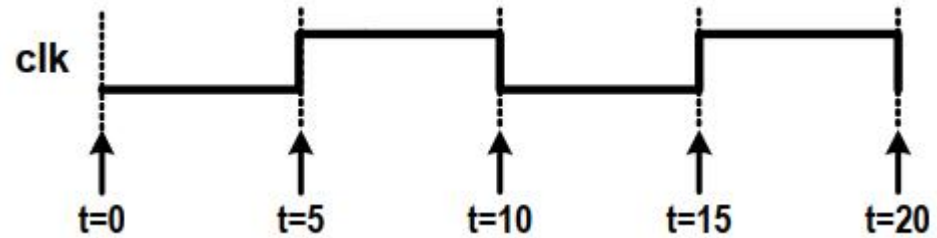
#4 unidades mas
tarde que la línea
anterior



Temporizaciones

Un ejemplo con retardos en un bloque always.

```
always  
  #5 clk = !clk
```



Uso de los eventos

Disparo del evento por cambio : @

Podremos hablar de eventos que se disparan por nivel o por flanco (posedge / negedge)

```
// por nivel  
always @(a)  
    b = b + c;
```

```
// por flanco  
always @(posedge clk)  
    b <= b+c;
```

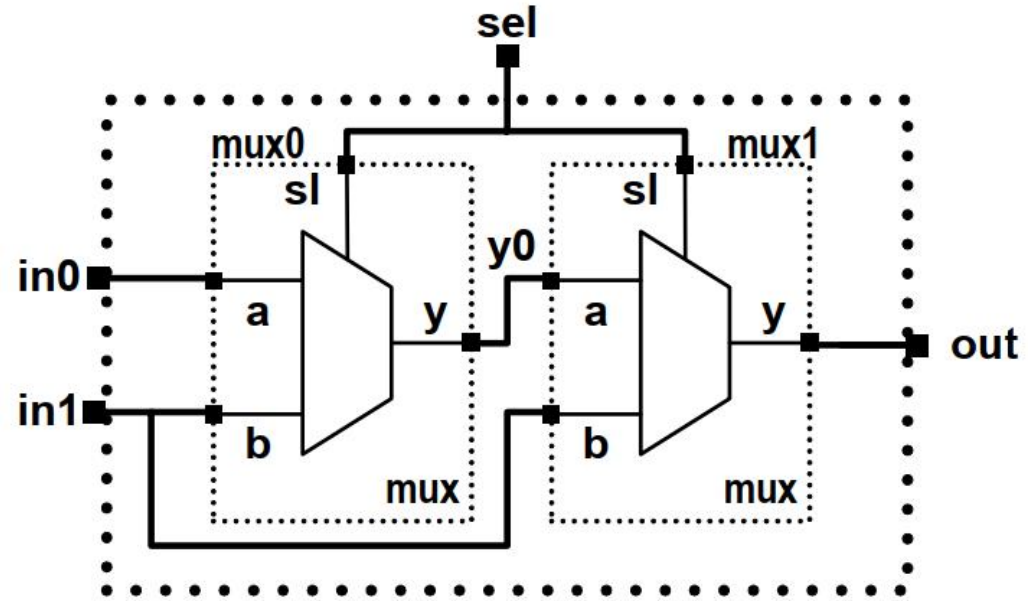
Módulos

Los diseños se van estructurando en módulos funcionales.

En el ejemplo siguiente el módulo general contiene submódulos de tipo mux.

Declaración : **mux(a,b,sl,y);**

```
module muxt(in0,in1,sel,out);  
  input [7:0] in0,in1;  
  input sel;  
  output [7:0] out;  
  wire [7:0] out;  
  //nodos internas  
  wire [7:0] y0;  
  //Conexionado por orden  
  mux mux0(in0,in1,sel,y0);  
  //Conexionado por nombre  
  mux  
  mux1(.y(out), .b(in1), .a(y0), .sl(sel));  
endmodule
```



se indican los puertos de
conexión: .puerto(señal)

Testbench (“banco de pruebas”)

Es una parte fundamental del proceso de diseño.

Frecuentemente la complejidad de esta parte es similar al diseño del circuito.

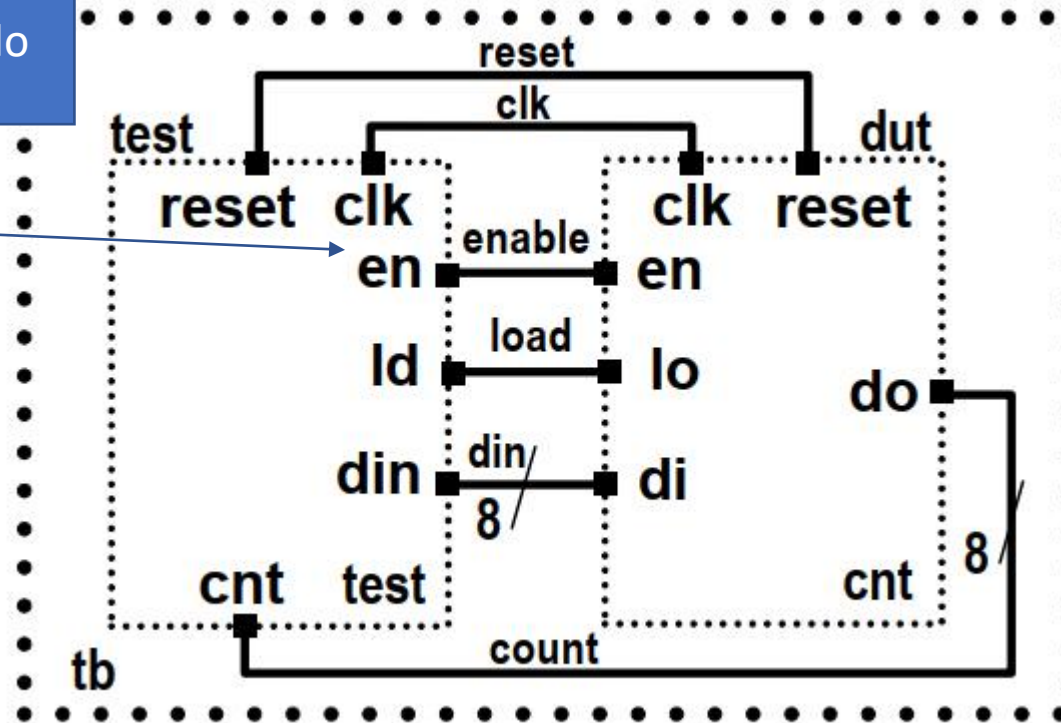
La parte de *testbench* se especifica como procedural y no tiene que ser sintetizable.

Un testbench consta de:

- Módulo DUT (design under verification)
- Módulo test (genera las entradas y salidas a testear sobre el DUT)
- Módulo tb: conexionado entre los módulos test y dut

Testbench

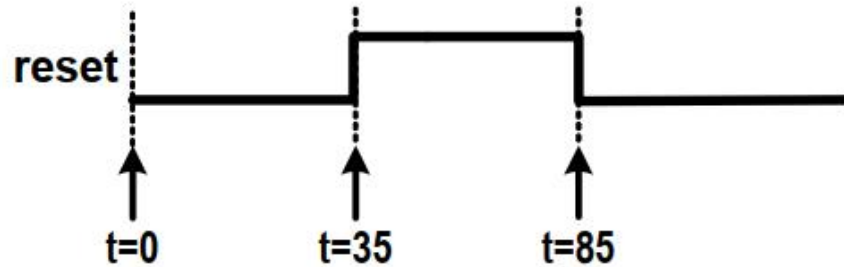
las señales de interfaz del módulo benchmark son iguales pero de sentido contrario al dut



Testbench

Cuando se está desarrollando un módulo de testeo tenemos que generar los estímulos.

Por ejemplo, supongamos que queremos generar una señal de *reset* de forma asíncrona con las características de la figura que se muestra:



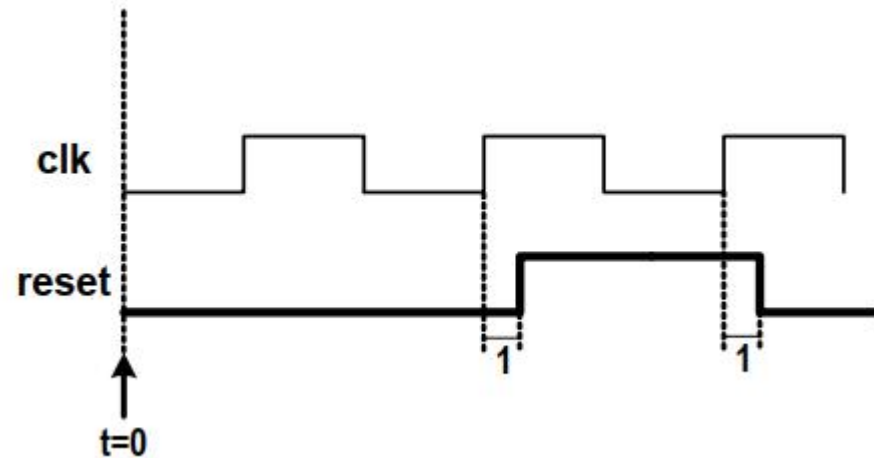
```
begin
  reset = 1'b0;
  #35
  reset = 1'b1;
  #50
  reset = 1'b0
end
```

Notar que usamos las
asignaciones con "="

Testbench

Si por el contrario quisiésemos hacerlo de forma síncrona respecto al *clk* del sistema podríamos hacer lo siguiente:

```
initial
begin
  reset = 1'b0;
  repeat(2)
    @(posedge clk) #1;
    reset = 1'b1;
    @(posedge clk) #1;
    reset = 1'b0;
end
```

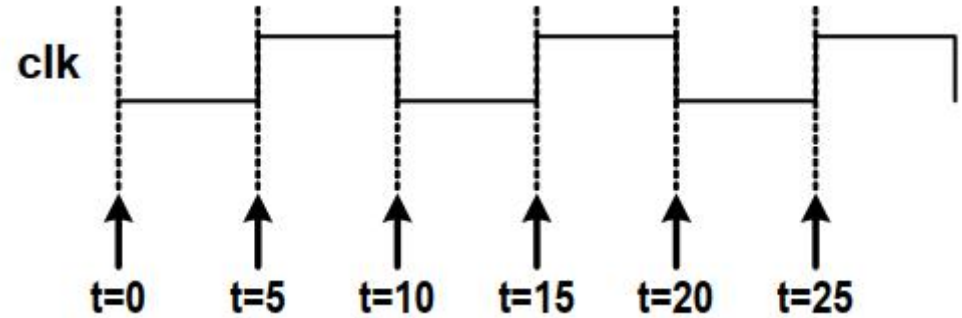


Testbench

Generar un clock sencillo:

```
`define PERIODO 5
...
always #`PERIODO clk=!clk;

initial
begin
    clk = 1'b0
end
```



Arrays en Verilog

En ocasiones necesitamos modelar contenidos en memorias más amplias que un simple registro. Verilog permite declarar arrays de 2 dimensiones.

```
reg [wordsize:0] nombre_array[arraysize:0]
```

```
//Memoria de 128 palabras de 8 bits  
reg [7:0] mi_memoria[127:0]
```

Operadores en Verilog

//operador +, - sobre un operando

a=+8'h01 El número resultado es 8'h01

a=-8'h01 El número resultado es 8'hFF

//operadores aritméticos de 2 operandos

a= b-c;

a= b*c;

a= b/c;

a= b%c;

//operadores de comparación

a == b

a != b

a === b

a !== b

Se consideran los estados x y z (4 state bits)
que se corresponden a “unknown” y “high impedance”

Operadores en Verilog

Operadores lógicos:

“!” negación

“&&” AND

“||” OR

Operadores Bit a Bit:

“~” negación

“&” AND

“|” OR

“^” OR exclusiva

y sus combinaciones: ~&, ~^, etc...

Ejemplo de combinaciones:
assign y = ~(a&b) //NAND

Operadores de reducción:

“&” usado sobre un único operando (&a) devuelve AND de todos los bits de a

Existen todas las versiones de estos: ~&, |, ~|, etc...

Operadores en Verilog

Operadores de desplazamiento:

"<<" y ">>". $a = b \ll 2$; Desplazamiento de 2 bits (se rellenan los ceros)

Concatenaciones de bits

{ } Ejemplos:

{a, b} , si a y b son de 8 bits el resultado son 16 bits

{ a, b[3:0] } , si a y b son de 8 bits el resultado son 12 bits

Sentencia if - else

```
//Condicional simple  
if (enable)  
    q<= #1 data;
```

```
//Condicional multiple  
if (reset == 1'b1)  
    count<= #1 8'b0;  
else if (enable == 1'b1 && up_en == 1'b1)  
    count<= #1 count + 1'b1;  
else if (enable == 1'b1 && down_en == 1'b1)  
    count<= #1 count - 1'b1;  
else  
    count<= #1 count;
```

Sentencia case

```
case (sel)
  2'b00: y = ina;
  2'b01: y = inb;
  2'b10: y = inc;
  2'b11: y = ind;
  default: y = 2'b00;
endcase
```

Nota explicativa: “sel” en el ejemplo a la izquierda sería un dato de 2 bits. Si en la definición del módulo tuviésemos

definidas 2 entradas:

input A,

input B

Aquí escribiríamos :

always @({A,B})

Existen unos “case” especiales: casez y casex que permiten que los valores de estado x y z puedan considerarse como indiferentes a la hora de seleccionar en el case.

La síntesis de una sentencia “case” será muy a menudo un multiplexor.

Bucles

```
//forever
```

```
initial
```

```
begin
```

```
    clk = 0;
```

```
    forever #5 clk = !clk;
```

```
end
```

```
//repeat : una rotación de bits
```

```
repeat (8) begin
```

```
    temp = data[7];
```

```
    data = {data <<1,temp};
```

```
end
```

```
//while
```

```
loc = 0;
```

```
if (data = 0)) loc = 32;
```

```
else while (data[0] == 1'b0) begin
```

```
    loc = loc + 1;
```

```
    data = data >> 1;
```

```
end
```

```
//FOR
```

```
for (i=0, i<64, i=i+1))
```

```
    ram[i] = 0
```

Directivas de compilación

```
`define VAR 5
```

```
`include mi_modulo.v
```

```
`timescale 1ns / 100ps      // <unidad> / <resolución>
```


Funciones del sistema

Son funciones (en general) no sintetizables utilizadas para los testbench.

Comienzan con el carácter \$.

Algunos ejemplos son:

\$display : muestra por pantalla mensajes.

Ej: \$display ("Valor : %d", data);

\$write: no añade retornos de carro ni avances de linea

\$monitor : similar a \$display pero imprime cuando cambie el valor de algunas de las variables de su lista. Tiene asociadas las funciones \$monitoron \$monitoroff

\$fopen

\$fclose

\$fdisplay

\$fwrite

etc...

Tareas (task)

Son lo que en otros lenguajes se llama procedimientos o subrutinas.

- Permiten una mayor organización del código.
- Pueden existir en ficheros separados y ser incluidas con la directiva “include”.
- Acceden a las variables globales del diseño, pero las declaradas en el interior son “locales”.
- Pueden contener retardos como #delay, posedge, negedge.
- Pueden llamarse de forma anidada
- Los parámetros de entrada / salida usan nodos que hayan sido previamente declarados en el módulo.
- Verilog no permite llamar a la tarea en una expresión.

```
task logic_oper;
input [7:0] numa;
input [7:0] numb;
output [7:0] out_and;
output [7:0] out_or;
begin
    out_and = numa & numb;
    out_or = numa | numb;
end
endtask
```

```
module operador(a,b);
input [7:0] a,b;
reg [7:0] aandb;
reg [7:0] aorb;
...
always @( a or b)
    logic_oper(a,b,aandb,aorb);
```

Funciones (function)

Se diferencia de las *task* en lo siguiente:

- No pueden incluirse en la definición elementos de retardo.
- Sólo una salida
- Modelan exclusivamente lógica combinacional
- Se pueden anidar llamadas de funciones, pero NO llamar a tareas desde funciones.

```
function logic_oper;  
input [7:0] numa;  
input [7:0] numb;  
begin  
    logic_oper = numa & numb;  
end  
endfunction
```

```
module operador(a,b);  
input [7:0] a,b;  
wire [7:0] aandb;  
  
...  
  
    assign aandb = alogic_oper(a,b);
```

Sólo puedo asignar sobre *wires* cuando es una asignación continua

Detalles y aclaraciones

Concurrencia:

Todos los *assign* se ejecutan de manera concurrente

Todos los bloques *always* y *assign* que haya en la descripción se ejecutan concurrentemente

```
module operac_lógica(  
    input a,b,c;  
    output fparc1, fparc2);  
begin  
    assign fparc1 = a & b;  
    assign fparc2 = fparc1 & c;  
  
    // si hubiese usado esta otra forma resultaría el mismo circuito  
    // assign fparc2 = a & b & c;  
  
end  
endmodule
```

Detalles y aclaraciones

Listas de sensibilidad:

Si estamos describiendo un bloque combinacional en una sentencia always la lista de sensibilidad debería incluir TODAS las variables de entrada del bloque. Se puede abreviar poniendo un “*”

```
module operac_combinacional (  
  input a,b,c,d,e,f;  
  output fparc1, fparc2 );  
  always @(a,b,c,d,e,f)           //<- Alternativa: @(*)  
  begin  
    // Expresiones combinacionales  
    // . . .  
  end  
endmodule
```

Detalles y aclaraciones

Tipos de asignaciones: bloqueantes y no bloqueantes

1. Bloqueante

El operador es el signo =

Es como una salida combinacional que se ejecuta **inmediatamente**, pero si hay varias consecutivas importará el orden en que se ejecuten, ya que unas van detrás de otras y podrían depender de los cambios producidos por las sentencias anteriores.

Está asociado generalmente a bloques combinacionales.

2. No Bloqueante

El operador es <=

Para TODAS las asignaciones se calculan los valores a la derecha y después todas las asignaciones se realizan simultáneamente. Por tanto NO importará el orden en que aparezcan en el código.

Es la forma normal de asignar en diseños de lógica secuencial.

Detalles y aclaraciones

Diferencia bloqueante / no bloqueante

Extraído <http://www.asic-world.com/verilog/index.html>

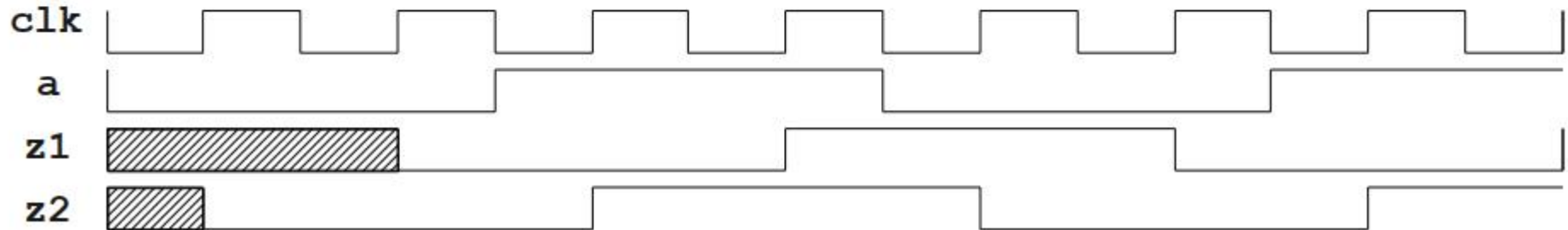
```
1 module block_nonblock();
2 reg a, b, c, d , e, f ;
3
4 // Blocking assignments
5 initial begin
6     a = #10 1'b1; // The simulator assigns 1 to a at time 10
7     b = #20 1'b0; // The simulator assigns 0 to b at time 30
8     c = #40 1'b1; // The simulator assigns 1 to c at time 70
9 end
10
11 // Nonblocking assignments
12 initial begin
13     d <= #10 1'b1; // The simulator assigns 1 to d at time 10
14     e <= #20 1'b0; // The simulator assigns 0 to e at time 20
15     f <= #40 1'b1; // The simulator assigns 1 to f at time 40
16 end
17
18 endmodule
```

Detalles y aclaraciones

Diferencia bloqueante / no bloqueante

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

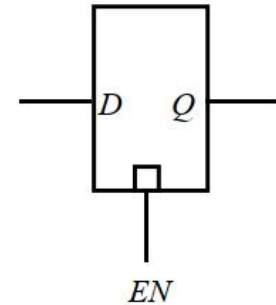
```
module bloqueante(input a,clk,  
  output reg z2);  
  reg q;  
  always @(posedge clk)  
    begin  
      q = a;  
      z2 = q;  
    end  
endmodule
```



Ejemplos

Flip-flop tipo D

```
module ff(d,clk,q,q_bar);  
  input d,clk;  
  output q,q_bar;  
  
  always @(posedge clk)  
  begin  
    q <= d;  
    q_bar <= !d;  
  end  
endmodule
```



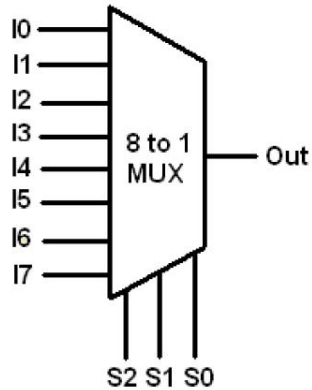
Ejemplos

Comparador entre palabras de 4 bits
(mayor/menor o igual – *greater/less/equal*)

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

Ejemplos

Multiplexor con 3 bits de selección



```
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
```

Detalles y aclaraciones

Dentro de los niveles de abstracción de circuitos basados en comportamiento existen 2 subvariantes:

Definición a nivel RTL (*Register Transfer Level*)

Trabaja sobre variables NET. Se utiliza la asignación continua para crear las funciones entre entradas y salidas.
Se especifican características de timing.

El código resultante es **siempre sintetizable**

De forma genérica, actualmente, se le llama código RTL a todo código que es sintetizable.

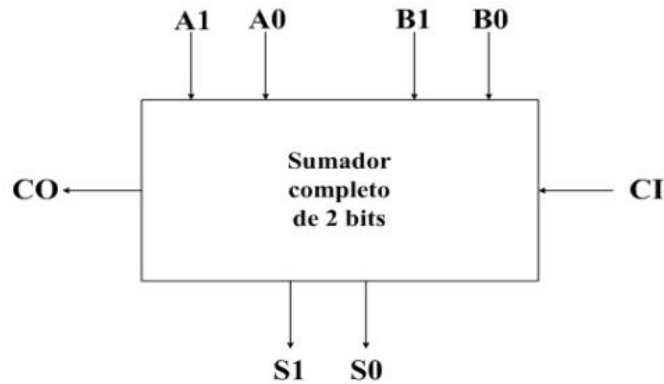
Un nivel de descripción superior sería el de arquitectura (de procesadores, por ejemplo). Son niveles inferiores, por ejemplo, las descripciones a nivel de puertas lógicas y por supuesto el nivel de transistor.

Descripción basada en algoritmos

Trabaja sobre datos tipo *reg* y usa las sentencias de control de flujo típicas de los lenguajes de programación.
El código resultado no es siempre sintetizable.

Ejemplo (RTL)

Sumador completo de 2 bits



```
module sumador (CO, S, A, B, CI);
```

```
output CO;
```

```
output [1:0] S;
```

```
input CI;
```

```
input [1:0] A, B;
```

```
wire [2:0] result;
```

```
assign result = A+B+CI;
```

```
assign CO = result[2];
```

```
assign S = result[1:0];
```

```
endmodule
```

Ejemplo

Buffer triestado

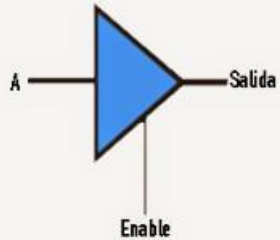
```
module buffer_triestado (out, in, enable);
```

```
output out;
```

```
input in, enable;
```

```
assign out = enable ? in : 1'bz;
```

```
endmodule
```



Simbolo

EN	A	SALIDA
0	0	HI-Z
0	1	HI-Z
1	0	0
1	1	1

Tabla de verdad

HI-Z=alta impedancia

Recursos

Simulador ON-line

EDA Playground

<https://www.edaplayground.com/>

Tutorial introducción

<https://www.youtube.com/watch?v=NXlqdrYga9M>

Libro de especificación del estándar de Verilog (antiguo pero válido)

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