Bloque I. METODOLOGÍAS Y HERRAMIENTAS DE DISEÑO

Tema 2. VERILOG

2_1 Revisión Verilog

Organización Bloque I

Bloque I. Metodologías y Herramientas de diseño

- **Teoría**
  - Tema 1: Introducción.
  - Tema 2: Verilog
    - Revisión del lenguaje
      - Clase: resumen de los principales conceptos
      - Vosotros: revisión/estudio de la sintaxis
    - Verilog para síntesis y simulación

Bibliografía Tema 2

Más centrados en el lenguaje:

- F. Vahid y R. Lysecky “Verilog for Digital Design”, Wiley 2007, ISBN 978-0-470-05262-4
- Verilog HDL Quick Reference Guide (Verilog-2001 standard)

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

- M. D. Ciletti, “Modeling, synthesis, and rapid prototyping with the Verilog HDL”, Prentice Hall (exhaustivo, muy extenso)

Más centrados en el proceso de diseño & metodologías

- Navabi, Zainalabedin, “Verilog digital system design. RT level synthesis, testbench, and verification”
- Wolf, W. “FPGA-Based System Design”, Prentice Hall 2004

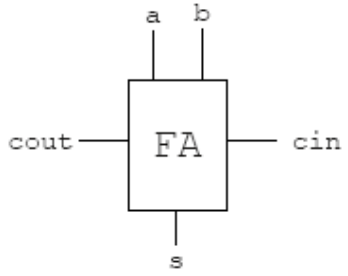
Índice revisiónVerilog

- Ejemplos
  - Modelado estructural versus modelado comportamiento
  - Procesos/procedures
  - Tipos de datos
  - Asignamientos Blocking y non Blocking
  - Construcciones de control de flujo
  - Funciones/tareas del sistema
  - Directivas de compilación

Revisión de Verilog. Introducción. Ejemplos

Ejemplo 1. Modelo de circuito combinacional
(tomado del material de Estructura de Computadores de 1º)

- 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
```

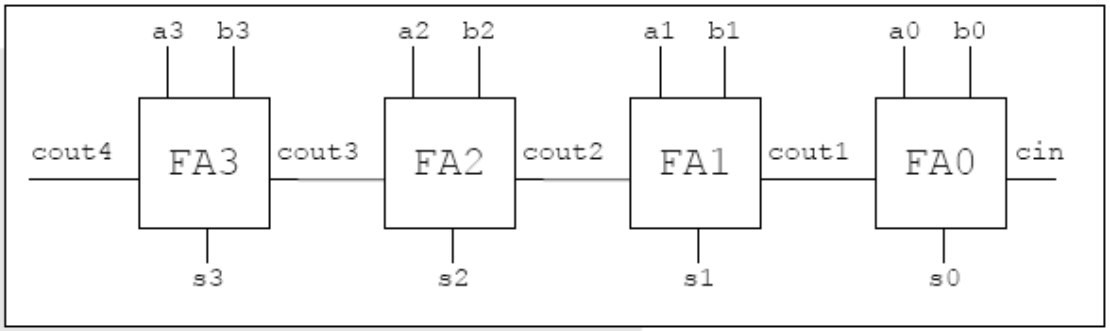
cin	a	b	cout	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Describimos el circuito capturando su funcionalidad, usando para ello operadores de verilog,

Revisión de Verilog. Introducción. Ejemplos

Ejemplo 2. Modelo estructural de circuito combinacional (y jerárquico)
(tomado del material de Estructura de Computadores de 1º)

- Unión de 4 FULL-ADDER



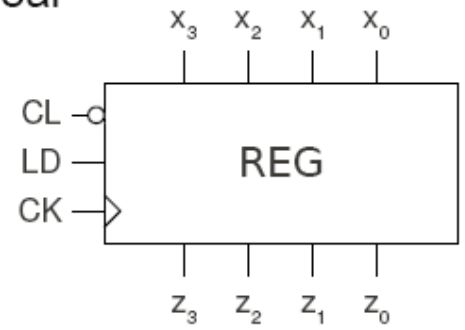

```
module fulladder4(
  input [3:0] a,
  input [3:0] b,
  input cin,
  output [3:0] s,
  output cout4);

 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
```

Describimos el circuito indicando sus componentes y cómo están conectadas (instanciamos)

Revisión de Verilog. Introducción. Ejemplos

Ejemplo 3. Modelo comportamiento de circuito secuencial
(tomado del material de Estructura de Computadores de 1º)

- Registro con carga en paralelo y clear



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

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

  reg [3:0] q;

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

  assign z = q;

endmodule
```

Describimos el circuito capturando su funcionalidad, usando construcciones propias de un lenguaje de programación

Revisión de Verilog. Introducción. Ejemplos

- Ejemplo 4. ***testbench***: un código verilog que instancia un diseño y describe qué aplicamos en sus entradas (estímulos) al simularlo

```
‘timescale 1ns/1ns
module comp _test
integer i, j; reg [1:0] a, b;
initial #1000 $finish ;
initial begin
for(i=0; i<=3; i=i+1)
begin
for(j=0; j<=3; j=j+1)
begin
a=i; b=j; # 10;
end
end
end
comp v_1(out1, out2, out3, a, b);
endmodule
```

Un HDL tiene implícita una semántica de tiempos asociada a sus objetos (señales) porque modela su evolución- Define unidad de tiempo y precisión

Control de la simulación. Simulamos hasta1000 ns

Describimos la evolución temporal de las señales a y b. Cada 10ns a y b cambian. Recorren todas las combinaciones posibles.

Estas señales a y b se usan como entradas del circuito que se simula (comp). Su descripción estaría en otro module

Modelado structural (1)

- Se captura en el Verilog un diagrama de bloques del circuito/sistema (qué componentes y cómo están interconectadas)
- Estos bloques pueden ser desde muy simples (una puerta lógica) a muy complejos (un procesador completo)
- Se dice que instanciamos “modules”
- Para que se pueda simular (o sintetizar) esos bloques tienen que tener también su descripción
- La única excepción es cuando se instancia una primitiva del lenguaje (puertas lógicas, inversores y buffers). En este caso se puede instanciar sin más.
- Todo ocurre concurrentemente
- Por eso, **no importa el orden en el que se instancien**
- **No los instanciamos dentro de un always o de un initial !**

Modelado structural (2). Assign

- Establece una asociación estática entre señales y expresiones
- Se ejecuta continuamente. Si la parte derecha del asignamiento cambia, se actualiza la parte izquierda

- Los asignamientos estáticos funcionan como un modelado estructural

- **assign suma = a + b + c;**

Es como instanciar una componente cuya salida denominamos "suma" y que implementa la funcionalidad que aparece en la expresión a la derecha del igual

- La ventaja es que puedo utilizar los operadores de verilog para describir la funcionalidad que queramos

- También ocurren concurrentemente y no importa el orden en el que se escriban en el código

Tipos de datos

- Nets (se usan en el modelado estructural)
  - Análogo a un cable (nodo) en un circuito
  - Sintaxis: tipo_net [msb:lsb] [#delay] identificador;
    - tipo_net: wire, supply0, supply1, tri, wand
  - Pueden ser asignados implícitamente por módulos al instanciarlos ("se conectan a las salidas")
  - Puede ser asignado en un **asignamiento continuo** o un force...release
  - Por defecto escalar, pueden definirse vectores y arrays
  - Existe el cualificador signed
  - Señales (datos) no explícitamente declaradas son wire
  - **Ejemplos**

```
wire y1, z_5; // declara dos wires
wire [7:0] data_bus; // declara un vector de 8 wires (bus)
wire A= B + C; // se hace una asignación al declarar A
wire #2 y; // asigna un retraso de 2 unidades
wire signed [3:0] signed_a;
```

- Variables (como en un lenguaje de programación, se usan en el modelado comportamiento)

Modelado comportamiento

Procedimientos (procedures, flujos de actividad)

- Conjunto de sentencias limitado por begin ...end que describe la evolución de los valores de las señales (**variables**) asociadas.
  - Un caso particular es el de una única sentencia en cuyo caso no es necesario utilizar los delimitadores
- Las sentencias dentro de un bloque secuencial que es parte de un procedimiento se ejecutan secuencialmente, en el orden en que aparecen (como en los lenguajes de programación), pero en la misma unidad de tiempo de simulación si no se especifica ninguna temporización
- Solo se asigna una variable cuando se ejecuta una sentencia de asignamiento en el flujo de actividad del procedimiento mientras conservan su valor
  - *(a diferencia de los asignamientos continuos o de los módulos instanciados que asignan valores a los nets **continuamente**)*
- Un módulo puede contener cualquier número de procedimientos, pero no se anidan
- Los procedimientos se ejecutan concurrentemente e interaccionan con otros procedimientos, asignamientos continuos y componentes instanciadas dentro del mismo módulo
- initial flujo de actividad que se ejecuta una vez, se activa en tsim= 0
- always flujo de actividad que se ejecuta cíclicamente, se activa en tsim= 0

Tipos de datos

- Nets (conectividad estructural)
- Variables (como en los lenguajes de programación)
  - Se asignan en sentencias procedurales (dentro de un initial o un always, funciones o *tasks)*
  - Conservan su valor hasta la siguiente asignación
  - Su valor inicial es desconocido (x)
  - reg
  - Almacena un valor lógico
  - No solo se usan para describir registros hardware !!!!
  - Existe el cualificador signed
  - Ejemplos

```
reg a:
reg [5:0] a, b;
reg [31:0] cce [0:1023]; // 1024 elementos de reg de 32 bits
reg signed [3:0] b_s
reg [1:0] IREG = 2'b11; // se asigna un valor inicial a IREG al declararla
```

  - integer, real, time, realtime

Revisión de Verilog. Modelado comportamiento

Procedimientos. Ejemplos

```
‘timescale 1ns/1ns
assign sig_net = a;
initial
begin
sig_a = a;
sig_b = 1;
sig_c = 1;
sig_d = 0;
// asignamientos procedurales
// se ejecutan secuencialmente
en tsim = 0 se asignan sig_a,
sig_b, sig_c y sig_d.
Incluso si a cambia luego sig_a no,
sig_net sí.
end
.....
```

```
‘timescale 1ns/1ns
module pp; reg Y, clk;
always begin
@(posedge clk) #5 Y=1;
// Espera 5 y Pone a 1 Y con el flanco de subida
@(posedge clk) #5 Y=0;
// Espera 5 y Pone a 0 Y al flanco siguiente
end
always #10 clk = ~clk ;
initial Y = 0;
initial clk = 0;
endmodule
```

```
T=0 clk =0 Y=0
T=10 clk =1 Y=0
T=15 clk =1 Y=1
T=20 clk =0 Y=1
T=30 clk =1 Y=1
T=35 clk =1 Y=0
T=40 clk =0 Y=0
T=50 clk =1 Y=0
T=55 clk =1 Y=1
T=60 clk =0 Y=1
```

Modelado comportamiento. Asignamientos procedurales

- Asignamiento procedural blocking (=)
  - variable = expresión; actualiza variable con el valor de expresión
    - Aparecen en procesos
    - No está soportado para nets
    - Debe ejecutarse antes de que se puedan ejecutar las que le siguen

```
initial
begin
sum[7] = b[7] ^ c[7]; // se ejecuta ahora

#10 hat = b&c;  // 10 unidades de tiempo después de que sum[7] cambia,
b&c se evalúa y hat cambia
```

Modelado comportamiento. Asignamientos procedurales

- Asignamiento procedural no blocking (<=,)
  - variable <= expresión; actualiza variable con el valor de expresión
    - Aparecen en procesos. No está soportado para nets
    - No bloquea la ejecución de las sentencias que le siguen
    - Las sentencias en una lista de asignaciones non-blocking que tienen que ejecutarse en el mismo tiempo de simulación, se ejecutan "concurrentemente" (primero se evaluan todas las expresiones y luego se actualizan todas las variables) y por ello son útiles en el modelado de operaciones de transferencias de registros síncronas

blocking

```
initial
begin
a = 1; b = 0;
a = b; // usa b = 0
b = a; // usa a = 0
```

No blocking

```
initial
begin
a = 1; b = 0;
a <= b; // usa b = 0
b <= a; // usa a = 1
```

Asignamientos Blocking/ Non Blocking

- Los asignamientos non-blocking capturan mejor las operaciones de transferencias de registros que se producen en los sistemas digitales síncronos. Se recomienda su uso cuando estemos modelando registros hardware

```
always @ (posedge clk)
        W = Y; Y = DATAIN;
```
Registro de desplazamiento

```
always @ (posedge clk)
        W <= Y; Y <= DATAIN;
```
Registro de desplazamiento

```
always @ (posedge clk)
Y = DATAIN; W = Y;
```
Dos flip-flops en paralelo

```
always @ (posedge clk)
Y< = DATAIN; W <= Y;
```
Registro de desplazamiento

```
always @ (posedge clk)
        R1 = R2; R2 = R1;
```
No modela el intercambio de contenidos

```
always @ (posedge clk)
        R1 <= R2; R2 < = R1;
```
modela el intercambio de contenidos

Revisión de Verilog. Modelado comportamiento. Temporización

- Control de retrasos (#): #Δt sentencia;
  - Retrasa la ejecución de la setencia Δt unidades de tiempo. Suspende el flujo de actividad
- Control por ocurrencia de eventos (@)

  @ (evento1) sentencia;;

  @ (evento1 or (,) evento2) sentencia;
  - Sincroniza la ejecución de una sentencia con la ocurrencia de un determinado evento
  - Evento: cambio en el valor de un identificador o una expresión que hacen referencia a un net o a una variable
  - Cuando se alcanza @ el flujo de actividad se suspende
  - cualificadores predefinidos
  - posedge : transiciones 0 -> 1, 0 -> x y x -> 1
  - negedge: transiciones 1 -> 0, 1 -> x y x -> 0

- Sentencia wait: wait (expresión) sentencia;
  - suspende el flujo de actividad hasta que una condición es verdadera

Revisión de Verilog. Modelado comportamiento. Control Flujo

- Condicionales
  - operador condicional (?...:)
  - sentencia condicional (if...else)
  - sentencia case (case)
- Bucles
  - construcción repeat
  - construcción for
  - construcción while
  - construcción forever
- Otros
  - Sentencia disable
  - Flujo de actividad paralelo (fork ..join)

Revisión de Verilog. Sintaxis básica. Operadores

Operator	Argument	Result
Arithmetic	Pair of operands (except minus)	Binary word
Bitwise	Pair of operands (except negation)	Binary word
Reduction	Single operand	bit
Logical	Pair of operands	Boolean value
Relational	Pair of operands	Boolean value
Shift	Single operand	Binary word
Conditional	Three operands	Expression

Revisión de Verilog. Otros procedimientos. Funciones

- Una **función** es un procedimiento que es usado en lugar de una expresión
  - Computa un valor. Normalmente cuando capturar este computo en una expresión es complejo, o se repite
  - Tiene al menos una entrada, ninguna salida, y **devuelve un único valor**
  - No puede contener sentencias de control temporal (operadores retraso o de control por eventos)
  - Puede llamar a otras funciones (pero no a una tarea)

Resultado_inmediato=Llama_a_una_**función**(All_Inputs);

```
module Func_compara;
reg [2:0] A, B,C, D, E, F;
initial begin A=1; B=0; D=2; E=3;
C=compara(A,B); F=compara(D, E);
end
function [2:0] compara;
input [2:0] a, b;
begin if (a<=b) compara=a;
else compara=b; end
endfunction
endmodule
```

Revisión de Verilog. Otros procedimientos. Tareas

- Una **tarea** es un procedimiento que es llamado en una sentencia procedural
  - Tiene cualquier número de entradas (incluida ninguna), salidas, y **no devuelve un valor**
  - Puede contener sentencias de control temporal (operadores retraso, de control por eventos)
  - Puede llamar a funciones y tareas

LLama_a_una_**tarea**_y_espera (Input1,Input2,Output);

```
module bit_counter(data_w1, data_w2,
bit_count1, bit_count2);
input [3:0] data_w1, data_w2;
output reg [2:0] bit_count1, bit_count2;
always @ *
count_ones(data_w1, bit_count1);
always @ *
count_ones(data_w2, bit_count2);
task count_ones;
input [3:0] reg_a;
output reg [2:0] count;
```

```
begin
count = reg_a[0] + reg_a[1]+reg_a[2] + reg_a[3];
end
endtask
endmodule
```

Revisión de Verilog. Contenidos del module

module el_diseño ();

- Declaraciones de terminales (entradas salidas)
- Declaración de parámetros
- Declaración de nets y variables internas
- Declaración de tareas y funciones
- **Instanciado de primitivas** (modelado estructural)
- **Instanciado de módulos** (modelado estructural)
- **Asignamientos continuos** (**assign**) (pseudo-estructural)
- **Procesos (flujos de actividad),** initial, always (modelado comportamiento)

  Asignamientos procedurales (blocking (=), non-blocking (<=), otros)

  Bucles (for, repeat, while, forever)

  Control de flujo (if, condicional, case, wait, disable)

  Funciones y tareas del sistema

  Funciones y tareas del usuario

endmodule

Revisión de Verilog

- El resto del documento profundiza en los contenidos anteriores.

Revisión de Verilog. Introducción

- La compañía **Gateway Design Automation** desarrolló **Verilog** como un lenguaje de simulación.
- En 1989 **Cadence** compró **Gateway** y en 1990 puso **Verilog** en dominio público.
- Posteriormente se desarrollo Verilog como un estándar del IEEE (IEEE 1364 (1995, 2001, 2005))
- **Verilog** es un lenguaje **tipo C**
- **Verilog** permite describir diseños en distintos estilos/niveles de abstracción
  - descripciones tipo **comportamiento** (similar a un código C con construcciones condicionales, loops, operadores…)
  - descripciones **estructurales** (interconexión de componentes)
  - **nivel de conmutadores** (los conmutadores son los transistores MOS que implementan las puertas lógicas, no lo vamos a cubrir aquí)

Revisión de Verilog. Module

- El module es la entidad de diseño principal en verilog
- Un sistema digital puede definirse como un conjunto de modules
- Sintaxis

```
module nombre_modulo (lista_de_terminales); // especifica nombre y terminales (puertos)
input [msb:lsb] lista_terminales_entrada; // especifica tipo y número de bits de cada terminal
output [msb:lsb] lista_terminales_salida;
inout [msb:lsb] lista_terminales_bidireccionales;
..... <module_items> ..... // comportamiento o instanciado de otros modules
endmodule
```


```
module suma_resta(suma, datoa, datob, out);
input suma;          //por defecto wire
input [7:0] datoa, datob;                    ;
output [7:0] out
.....sentencias .....
endmodule
```

```
module suma_resta(input suma, input [7:0] datoa, datob, output [7:0] out);
……..sentencias…….
endmodule
```

alternativa

Revisión de Verilog. Modelado estructural

- Los módulos se instancian dentro de otros módulos
- No se pueden instanciar modules en procedimientos
- No importa en orden en el que se instancien.
- Sintaxis

```
module_name
instance_name_1 (lista_conexiones_terminales),
instance_name_2 (lista_conexiones_terminales);
```

- **Ejemplo. Instanciado de componentes**

```
wire [3:0] in1, in2;
wire [3:0] o1, o2;
// C1 es una instancia del module and4
// los términales de C1 referenciados por posición
and4 C1(in1, in2, o1);
// C2 es otra instancia del modulo and4
// los terminales de C2 referenciados por nombre
and4 C2(.c(o2), .a(in1), .b(in2));
```

```
// Definición de and4
module and4(a, b, c);
input [3:0] a, b;
output [3:0] c;
assign c=a&b;
endmodule
```

Revisión de Verilog. Modelado estructural. Primitivas

- Puertas lógicas que son parte del lenguaje Verilog
- Se pueden especificar retrasos para estas puertas
- Sintaxis

- Puertas básicas: una salida

```
GATE #(retrasos) nonbre_instancia_1(out1, in1,....inN);
GATE: and, nand, or, nor, xor, xnor
and c1(o, a, b, c, d);// cuatro entradas
and c2(p, f, g);//dos entradas
or #(4,3) c3(s, a, b);// retraso subida (4 unidades), bajada (3)
xor #(5) c4(m, c, d); // retraso de subida y de bajada iguales (5)
```

- Buffers e Inversores (multi-salida)

```
GATE #(retrasos) nombre_instancia_1(out1, ..., outN, in);
GATE: buf o not
not #(4) not_1(a, c);        buf c1(o, p, q, r, in);
```

- Puertas tri-estado (multi-salida)

```
GATE puede ser bufif1, bufif0, notif1, notif0
bufif0  c1(bus, a, ctrl);
```

Revisión de Verilog. Sintaxis básica

❑ Números

❑ Constantes enteras

<size>'<base><número>

- ❑ <size> valor decimal que especifica cuantos bits representan el valor almacenado, opcional
- ❑ <base> **decimal** (d o D), **hex** (h o H), **octal** (o ó O) y **binary**(b o B)

  la base puede ir precedida de s (sd, sb, so, sh). Si se hacen operaciones con ellos, se interpretan con signo. Verilog utiliza complement a 2 para la representación de números negativos

  decimal por defecto
- ❑ <numero> valor en la base indicada
- ❑ Pueden utilizarse underscores para mejorar la legibilidad

Número	#bits	Base equivalente	dec. almacenado	
2'b10	2	binaria	2	10
3'd5	3	decimal	5	101
3'b5		no valido		
'ha	-	hexadecimal	10	depende de la máquina
-45	-	decimal	-45	dep. máquina

Revisión de Verilog. Sintaxis básica

❑ Números

- ❑ Constantes Reales
  - ❑ Pueden definirse usando notación decimal o científica
  - ❑ Al menos un dígito a cada lado del punto
  - ❑ Pueden utilizarse *underscores* para mejorar la legibilidad

```
3.14159
0.5
2e-5
1.25e10
1_.2_5e1_0
```

Revisión de Verilog. Sintaxis básica. Operadores

- Verilog tiene un conjunto robusto de operadores para manipular operandos. Un operando puede ser un *net*, *una variable*, un número, una selección de bits de un *net* o de una *variable,* una función o una concatenación de estos (operador concatenación {})
- Algunos se utilizan en expresiones (combinan operadores y operandos) que se asignan a *nets* o *variables*

```
// exor bit a bit de datos de ocho bits
module bitwise_xor(y, a, b)
input [7:0] a, b; output [7:0] y;
assign y = a ^ b; /* ^ operador bitwise, genera un resultado de 8
bits, el resultado se asigna al net vector y */
endmodule
```

- Otros se utilizan en expresiones booleanas utilizadas como condiciones

```
always @ (A or B) if (A == B) igual = 1; else igual = 0;
/*  A == B expresión booleana en un  condicional, el resultado es un valor
booleano (verdadero, falso ó ambiguo) */
```

Revisión de Verilog. Sintaxis básica. Operadores

❑ Aritméticos

+	addition
-	subtraction
*	multiplication
/	division
%	modulus
**	power

Revisión de Verilog. Sintaxis básica. Operadores

❑ Bitwise y reducción

~	Bitwise NOT
& (~&)	AND (NAND)
| (~|)	OR (NOR)
^	XOR
^~ or ~^	NXOR

bitwise NOT	~'b10xz es 'b01xx
bitwise AND	(010101) & (001100) es (000100)
reducción AND	&4'b1111 es 1'b1, &'b1x es 'bx
reduccion EXOR	^2'b01 es 1'b1

Revisión de Verilog. Sintaxis básica. Operadores

❑ Lógicos

!	Negation
&&	AND
||	OR
== (!=)	Equality (Inequality)
=== (!===)	Equality (Inequality) including x and z

Descripción	Ejemplo
Negación Lógica	!1'b1 es 1'b0,
AND lógica	3'b110 && 3'b000 es 0 (falso) 3'b110 && 3'b001 es 1 (verdadero) 3'b110 & 3'b001 es 000 (falso si se usa como condición)
Igualdad ==	(010101) == (001100) falso (10x) == (100) ambiguo
Igualdad ===	(00x) == (000) falso

Revisión de Verilog. Sintaxis básica. Operadores

❑ Relacionales

>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

- Devuelven verdadero si la comparación es verdadera y devuelven falso si es falsa
- Si hay bits desconocidos (x) o alta impedancia (z) en los operandos devuelve x

Revisión de Verilog. Sintaxis básica. Operadores

❑ Desplazamiento

>>	Desplazamiento lógico a la derecha
<<	Desplazamiento lógico a la zquierda
>>>	Desplazamiento aritmético a la derecha
<<<	Desplazamiento aritmético a la izquierda

R = R_in >> 2
R2 = R_in <<< 8

- Desplazan el primer operando tantas posiciones como indique el segundo operando
- En los desplazamientos lógicos y en el aritmético a la izquierda las posiciones vacantes se completan con ceros
- En el desplazamiento aritmético a la derecha las posiciones vacantes se completan con el valor original del bit más significativo

Revisión de Verilog. Sintaxis básica. Operadores

❑ Condicional

```
// multiplexor usando operadores condicionales
wire  select;
wire [15:0] d_1, d_2;
wire [15:0] bus_a
assign bus_a = select  ? d_1: d_2;
```

❑ Concatenación, {}

- Concatena los bits de dos o más expresiones (o nets o variables)
- A5 = {1'b0, A};
- {Co, S} = A + B +Ci;

Revisión de Verilog. Modelado comportamiento. Control Flujo

- Operador condicional (?...:)

  expresion_condicional = expresión ? exp_verdadera : exp falsa

  se usa tanto en asignamientos continuos como procedurales

Ejemplo. **Circuito sumador/restador**

```
module ejemplo7(y_out, clk, r, sel, a, b);
input  sel;
input [15:0] a, b;
output [15:0] y_out; reg [15:0] y_out;
always @ *
y_out = (sel) ? a + b : a -b;
endmodule
```

Revisión de Verilog. Modelado comportamiento. Control Flujo

Sentencia condicional (if...else)

if (expresión) sentencia1; else sentencia2;

- Selección entre sentencias alternativas en función del valor booleano de una expresión
- Una expresión es verdadera si es distinta de cero
- Sentencia puede ser una sentencia simple o un bloque secuencial
- Se pueden anidar

si todas las posibilidades no están especificadas, la variable no cambia

Ejemplo. Circuito sumador/restador con registro a la salida

```
module mux_bev(y_out, clk, r, sel, a, b);
input clk, r, sel;
input [15:0] a, b;
output [15:0] y_out; reg [15:0] y_out;
always @ (posedge clk or negedge r)
if( r == 0) y_out = 0; else y_out = (sel) ? a + b : a -b;
endmodule
```

Revisión de Verilog. Modelado comportamiento. Control Flujo

Sentencia case (case)

```
case (expresión) case1: sentencia1;
case2: sentencia2;
.....
default sentenciad;
endcase
```

- Ejecuta la sentencia o las sentencias asociadas al elemento case cuya expresión asociada (casei) coincida con expresión (también los posibles valores a x o a z deben coincidir)
- Se examinan de arriba abajo y una vez que se identifica una coincidencia los restantes elementos se ignoran
- Si ningún elemento coincide, se ejecuta la sentencia o sentencias asociadas a default, si existiese

Ejemplo. Mux

```
module mux4_case(a, b, c, d, select, y_out);
input a, b,  select; output y_out; reg y_out
always @(a or b or c or d or select).
case (select)
0: y_out = a;
1: y_out = b;
default y_out = 1'bx;
endcase  endmodule
```

Variantes:

casez: trata z como no importa
casex: trata x y z como no importa

Se puede asociar la misma sentencia o conjunto de sentencias a varias expresiones

```
case1, case2, case3: sentencia1;
```

Revisión de Verilog. Modelado comportamiento. Control Flujo

Construcción repeat

repeat (expresión) sentencia;

- Ejecuta una sentencia o un bloque de sentencias un número fijo de veces determinado por el valor de expresión

```
// inicializa un array de memoria
word_address = 0;
repeat (memory_size)
begin
memory[word_adress] = 0;
word_address = word_address + 1;
end
.....
```

Revisión de Verilog. Modelado comportamiento. Control Flujo

Construcción for

for (inicialización_índice; condición; actualización_indice) sentencia;

- Ejecuta una sentencia o un bloque de sentencias un número determinado de veces controlado por una variable índice
- Ejecuta una sentencia o un bloque de sentencias repetidamente, bajo la condición de que una expresión sea verdadera

```
.....
for (K = 4; K ; K = K -1;)
begin
demo_register [K + 10] = 0;
demo_register [[K + 2] = 1;
end
// posiciones asignadas: 14, 13,
// 12, 11, 6, 5, 4, 3
```

Revisión de Verilog. Modelado comportamiento. Control Flujo

Construcción while

while (expresión)) sentencia;

- Ejecuta una sentencia o un bloque de sentencias repetidamente mientras expresión sea verdadera

```
…..
// Cuenta el número de unos en reg_a
reg [7:0] temp_reg;
count = 0;
temp_reg = reg_a;
while (temp_reg)
begin
count= count + temp_reg[0];
temp_reg = temp_reg >> 1;
end
end
.....
```

Revisión de Verilog. Modelado comportamiento. Control Flujo

Construcción forever

forever sentencia;

ejecuta una sentencia repetidamente y de forma incondicional sujeto a disable

Sentencia disable

termina prematuramente un bloque secuencial etiquetado, sentencia procedural o task;

```
parameter half_cycle = 50;
initial
begin: clock_loop
clock = 0;
forever begin
#half_cycle clock = 1;
#half_cycle clock = 0;
end
end
initial
#350 disable clock_loop;
```

La señal *clock* conmuta cada 25 unidades de tiempo hasta las 350 unidades

Revisión de Verilog. Modelado comportamiento. Control Flujo

- Flujo de actividad paralelo fork ...join

```
fork
sentencia1;
sentencia2;
. ...
join
```

```
.....
fork // tsim = 0
#50 sig_wave = ´b1;
#100 sig_wave = ’b0;
#150 sig_wave = ‘b1;
#300 sig_wave = ‘b0; // se ejecuta en tsim= 300
join
 // mismo resultado si se cambia orden
```

```
.....
begin // bloque secuencial
#50 sig_wave = ‘b1;
#100 sig_wave = ’b0;
#150 sig_wave = ‘b1;
#300 sig_wave = ‘b0; // se ejecuta en tsim= 600
end
..... // resultado distinto si se cambia el orden
```

Revisión de Verilog. Funciones y tareas del sistema

- Funciones y tareas predefinidas (**comienzan por $)**
- Algunas muy utilizadas:
  - ❑ Visualización de los resultados de la simulación

    $monitor, $display, $write, $strobe
  - ❑ Control de la simulación

    $stop, $finish
  - ❑ Chequeo comportamiento temporal

    $hold, $setup, $setuphold, $period, $width, $skew, $nochange, $recovery
  - ❑ I/O ficheros

    $fclose, $fdisplay, $fmonitor, $fopen, $fstrobe, $fwrite, $readmemb, ...
  - ❑ Obtener tiempo de simulación:

    $time
  - ❑ Conversión:

    $signed, $unsigned, $bitstoreal, $realtobits, $itor, $rtoi

Revisión de Verilog. Directivas de compilación

- Instrucciones para el compilador (**comienzan por `**)
- Están activas desde el momento en que se declaran hasta que otra directiva la anula
- Dependen del compilador
- Algunas estándares:
  - `include: inserta el contenido de un archivo en otro durante la compilación
  - `define (`undef): se usa para definir (eliminar definiciones) macros de texto. Normalmente se usan en un archivo nombre.h .`define PRIMERO
  - `ifdef: inclusión opcional de líneas de código durante la compilación. Chequea si un determinado MACRO ha sido definido y en caso afirmativo compila las líneas de código asociadas.

```
`ifdef PRIMERO  sentencia 1;
else  sentencia 2;
```

  - `timescale: especifica la unidad de tiempo y la precisión de la simulación de los módulos que siguen a la directiva. La precisión indica como el simulador redondea los tiempos. Afecta a la velocidad de la simulación y a la memoria requerida. `timescale 1ns/10ps

Revisión de Verilog. Generate

La construcción generate permite simplificar la escritura de códigos
"Desaparecen" antes del que código se simule o se sintetice

generate for

```
module Reg64 (input [63:0] D, input clk,
output [63:0] Sal);
genvar i;
generate
for (i= 0; i < 64; i= i + 1)
begin: etapa
FFD flipflop (D[i], clk, Sal[i]);
end
endgenerate
endmodule
```

Instancia 64 flip-flops sin necesidad de instanciarlos uno a uno. Para el simulador o para la herramienta de síntesis es idéntico a haber instanciado los 64.

generate if

```
module multiplier (a, b, product);
parameter WIDTH = 8;
input [WIDTH -1:0] a, b;
output [2*WIDTH -1:0] product;
generate
if (WIDTH < 8)
CLA_multiplier #(WIDTH) C1(a, b, product);
else
Wallace_multiplier #(WIDTH) C1(a, b, product);
endgenerate
endmodule
```

Cuando en otro código instancio el módulo multiplier:
multiplier #(5) m1(….); //instancia el CLA_multiplier
multiplier #(18) m1(….); //instancia el Wallace_multiplier