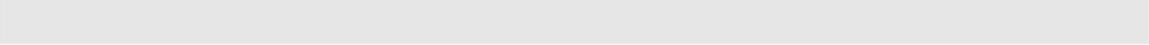


*Este tutorial pretende ser una guía de aprendizaje para el diseño HDL usando Verilog. Los conceptos del diseño se explican a lo largo de los ejemplos que se van desarrollando. Cada apunte a la sintaxis, representación de constantes, directivas, etc... se introduce a medida que van siendo necesarios para el desarrollo del ejemplo. Por todo esto y debido a la estructura de su contenido, no se puede considerar este tutorial una guía de consulta sino de aprendizaje de este lenguaje.*

<b>1</b>	<b>INTRODUCCIÓN</b> .....	<b>4</b>
1.1	ACERCA DEL LENGUAJE .....	4
1.2	NIVELES DE ABSTRACCIÓN EN VERILOG.....	4
<b>2</b>	<b>MI PRIMER DISEÑO</b> .....	<b>5</b>
2.1	INTRODUCCIÓN .....	5
2.2	ALGUNAS CONSIDERACIONES ACERCA DEL LENGUAJE .....	6
2.3	NÚMEROS EN VERILOG .....	6
2.4	TIPOS DE DATOS .....	6
<b>3</b>	<b>PROCESOS</b> .....	<b>7</b>
3.1	ASIGNACIÓN CONTINUA .....	9
3.2	TEMPORIZACIONES.....	9
3.3	EVENTOS.....	10
<b>4</b>	<b>MÓDULOS Y JERARQUÍAS</b> .....	<b>10</b>
4.1	CONEXIONADO .....	10
<b>5</b>	<b>TESTBENCH</b> .....	<b>11</b>
5.1	ESTRUCTURA DE UN TESTBENCH.....	11
5.2	MÓDULO TEST.....	11
5.2.1	<i>Interfaz de entrada/salida</i> .....	12
5.2.2	<i>Generación de estímulos</i> .....	12
<b>6</b>	<b>MODELADO DE MEMORIAS EN VERILOG</b> .....	<b>14</b>
6.1	USO DE PARÁMETROS.....	15
<b>7</b>	<b>OPERADORES EN VERILOG</b> .....	<b>15</b>
7.1	OPERADORES ARITMÉTICOS .....	15
7.2	OPERADORES RELACIONALES.....	16
7.3	OPERADORES DE IGUALDAD .....	16
7.4	OPERADORES LÓGICOS .....	16
7.5	OPERADORES BIT A BIT (BIT-WISE).....	16
7.6	OPERADORES DE REDUCCIÓN .....	16
7.7	OPERADORES DE DESPLAZAMIENTO .....	17
7.8	OPERADOR DE CONCATENACIÓN .....	17
7.9	OPERADOR CONDICIONAL .....	17
7.10	PRECEDENCIA DE LOS OPERADORES .....	17
<b>8</b>	<b>ESTRUCTURAS MÁS COMUNES</b> .....	<b>17</b>
8.1	SENTENCIAS CONDICIONALES IF – ELSE.....	17
8.2	SENTENCIA CASE.....	18
8.3	SENTENCIA CASEZ Y CASEX.....	19
8.4	SENTENCIAS DE BUCLE .....	19
8.4.1	<i>Sentencia forever</i> .....	19
8.4.2	<i>Sentencia repeat</i> .....	19
8.4.3	<i>Sentencia while</i> .....	19
8.4.4	<i>Bucle for</i> .....	20
<b>9</b>	<b>DIRECTIVAS DE COMPILACIÓN</b> .....	<b>20</b>
9.1	DEFINE .....	20
9.2	INCLUDE.....	20
9.3	IFDEF – ELSE - ENDIF .....	20
9.4	TIMESCALE.....	21
<b>10</b>	<b>FUNCIONES DEL SISTEMA</b> .....	<b>21</b>
<b>11</b>	<b>TAREAS EN VERILOG</b> .....	<b>22</b>

11.1	SINTAXIS Y LLAMADA A UNA TAREA .....	22
<b>12</b>	<b>FUNCIONES EN VERILOG .....</b>	<b>23</b>
12.1	SINTAXIS Y LLAMADA A UNA FUNCIÓN.....	23

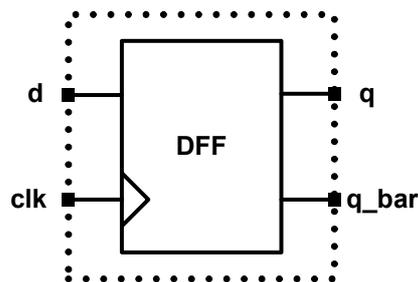


## 1 Introducción

### 1.1 Acerca del lenguaje

Verilog es un lenguaje de descripción hardware (*Hardware Description Language, HDL*) utilizado para describir sistemas digitales, tales como procesadores, memorias o un simple flip-flop. Esto significa que realmente un lenguaje de descripción hardware puede utilizarse para describir cualquier hardware (digital) a cualquier nivel.

La descripción del sistema puede ser tan sencilla como la de un flip-flop, tal y como se refleja en la Figura 1-1, o un sistema complejo de más de un millón de transistores, tal es el caso de un procesador. Verilog es uno de los estándares HDL disponibles hoy en día en la industria para el diseño hardware. Este lenguaje nos permite la descripción del diseño a diferentes niveles, denominados niveles de abstracción.



```

modul e ff(d, cl k, q, q_bar);
i nput      d, cl k;
output     q, q_bar;

al ways @(posedge cl k)
begi n
q      <= d;
q_bar <= !d;
end
endmodul e

```

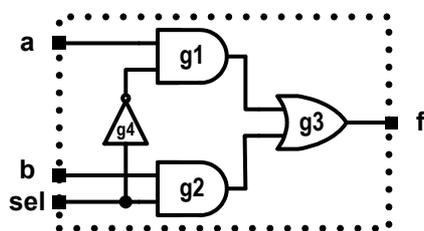
▪ Figura 1-1 Descripción Verilog de un *flip-flop*.

### 1.2 Niveles de abstracción en Verilog

Verilog soporta el diseño de un circuito a diferentes niveles de abstracción, entre los que destacan:

- **Nivel de puerta.** Corresponde a una descripción a *bajo nivel* del diseño, también denominada modelo estructural. El diseñador describe el diseño mediante el uso de primitivas lógicas (AND, OR, NOT, etc...), conexiones lógicas y añadiendo las propiedades de tiempo de las diferentes primitivas. Todas las señales son discretas, pudiendo tomar únicamente los valores '0', '1', 'X' o 'Z' (siendo 'X' estado indefinido y 'Z' estado de alta impedancia).

La Figura 1-2 representa la descripción de un multiplexor a nivel de puertas. Este nivel de descripción no resulta ni mucho menos adecuado por lo que no se volverá a tratar a lo largo del presente texto.



```

modul e mux(f, a, b, sel );
i nput      a, b, sel ;
output     f;

and      #5      g1(f1, a, nsel ),
          #5      g2(f2, b, sel );

or       #5      g3(f, f1, f2);

not      #5      g4(nsel , sel );

endmodul e

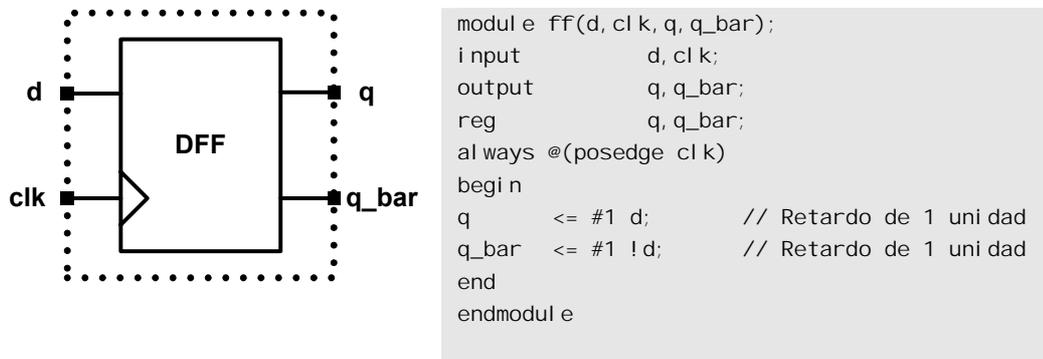
```

▪ Figura 1-2 Descripción a nivel de puerta de un multiplexor.

- **Nivel de transferencia de registro o nivel RTL.** Los diseños descritos a nivel RTL especifican las características de un circuito mediante operaciones y la transferencia de

datos entre registros. Mediante el uso de especificaciones de tiempo las operaciones se realizan en instantes determinados. La especificación de un diseño a nivel RTL le confiere la propiedad de *diseño sintetizable*, por lo que hoy en día una moderna definición de diseño a nivel RTL es *todo código sintetizable se denomina código RTL*.

La Figura 1-3 corresponde a la descripción a nivel RTL de un *flip-flop*. Este nivel de descripción, por la propiedad de ser sintetizable, será el nivel utilizado por excelencia en el diseño HDL.



▪ Figura 1-3 Descripción a nivel RTL de un *flip-flop*.

- ✦ **Nivel de comportamiento (*Behavioral level*)**<sup>1</sup>. La principal característica de este nivel es su total independencia de la estructura del diseño. El diseñador, más que definir la estructura, define el comportamiento del diseño. En este nivel, el diseño se define mediante algoritmos en paralelo. Cada uno de estos algoritmos consiste en un conjunto de instrucciones que se ejecutan de forma secuencial. La descripción a este nivel puede hacer uso de sentencias o estructuras no sintetizables, y su uso se justifica en la realización de los denominados *testbenches* (definidos más adelante).

## 2 Mi primer diseño

### 2.1 Introducción

La descripción de un diseño en Verilog comienza con la sentencia:

```
✦ modul e <nombre_módul o> <(defi ni ci ón l as se ñal es de i nterfaz)>;
```

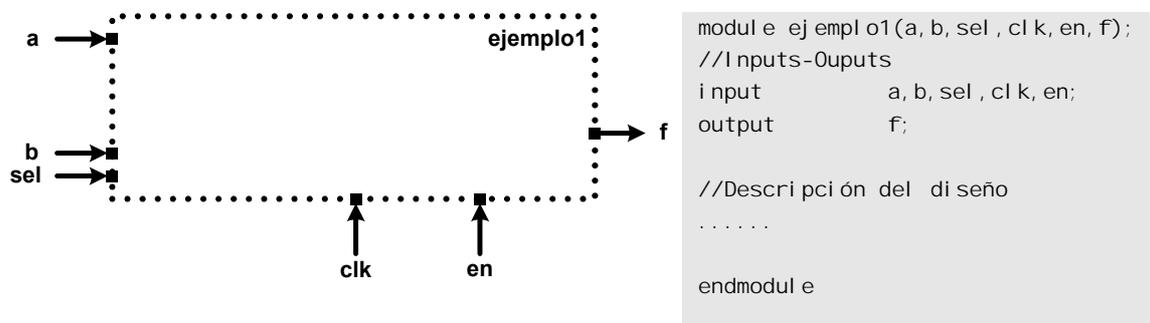
En segundo lugar se declaran las entradas/salidas:

```
✦ i nput/output <wi dth> se ñal ;
```

Seguidamente se describe el módulo/diseño (tarea que veremos más adelante) y se termina la descripción con:

```
✦ endmodul e //Notar que a di fe re nci a de l as de más se nci as, no se i ntr oduce ";"
```

La Figura 2-1 muestra las diferentes partes en la descripción de un diseño.



▪ Figura 2-1 Descripción del interfaz.

<sup>1</sup> Si bien se define el nivel de comportamiento como un nivel diferente al nivel RTL, hay que decir que éste último se puede englobar dentro del nivel de comportamiento como una descripción a *bajo nivel*.

## 2.2 Algunas consideraciones acerca del lenguaje

Antes de proseguir con la definición del diseño es importante hacer notar las siguientes consideraciones.

- **Comentarios.** De una sola línea: pueden introducirse precedidos de `//`. De varias líneas: pueden introducirse con el formato `/* comentario */`.
- **Uso de Mayúsculas.** Verilog es sensible al uso de mayúsculas. Se recomienda el uso únicamente de minúsculas.
- **Identificadores.** Los identificadores en Verilog deben comenzar con un carácter, pueden contener cualquier letra de la **a** a la **z**, caracteres numéricos además de los símbolos “\_” y “\$”. El tamaño máximo es de 1024 caracteres.

## 2.3 Números en Verilog

Las constantes numéricas en Verilog pueden especificarse en decimal, hexadecimal, octal o binario. Los números negativos se representan en complemento a 2 y el carácter “\_” puede utilizarse para una representación más clara del número, si bien no se interpreta. La sintaxis para la representación de una constante numérica es:

➤ `<Tamaño>' <base><val or>`

Si bien el lenguaje permite el uso de números enteros y reales, por brevedad y simplicidad sólo utilizaremos la representación de constantes numéricas enteras.

Entero	Almacenado como	Descripción
1	00000000000000000000000000000001	Unzised 32 bits
8'hAA	10101010	Sized hex
6'b10_0011	100011	Sized binary
'hF	00000000000000000000000000001111	Unzised hex 32 bits
6'hCA	001010	Valor truncado
6'hA	001010	Relleno de 0's a la izquierda
16'bz	zzzzzzzzzzzzzzzz	Relleno de z's a la izquierda
8'bx	xxxxxxx	Relleno de x's a la izquierda
8'b1	00000001	Relleno de 0's a la izquierda

▪ Figura 2-2 Ejemplo de definición de números enteros en Verilog.

La Figura 2-2 muestra algunos ejemplos de definición de números enteros. Verilog expande el valor hasta rellenar el tamaño especificado. El número se expande de derecha a izquierda siguiendo los siguientes criterios:

- Cuando el tamaño es menor que el valor se truncan los bits más significativos.
- Cuando el tamaño es mayor que el valor, se rellenan con el valor del bit más significativo del número, siguiendo el convenio que se muestra en la Figura 2-3.

Bit más significativo	Se rellena con
0	0
1	0
z	z
x	x

▪ Figura 2-3 Bit de relleno.

Los **números negativos** se especifican poniendo el signo “-” delante del tamaño de la constante, tal y como se especifica en el siguiente ejemplo:

➤ `-8' d2 → 11111110`

La representación interna de los números negativos en Verilog se realiza en complemento a 2.

## 2.4 Tipos de datos

Antes de seguir con el diseño de nuestro primer módulo es necesario explicar los tipos de datos con los que Verilog trabaja. Existen en Verilog dos tipos de datos principalmente:

- **Nets.** Representan conexiones estructurales entre componentes. No tienen capacidad de almacenamiento de información.
  - De los diferentes tipos de *nets* sólo utilizaremos el tipo *wire*.
- **Registers.** Representan variables con capacidad de almacenar información.

- De los diferentes tipos de *registers* sólo utilizaremos el tipo *reg* y el tipo *integer* (estos últimos solo en la construcción de los testbenches).

La Figura 2-4 introduce la definición de los nodos, tanto *wire* como *reg* para el ejemplo1.

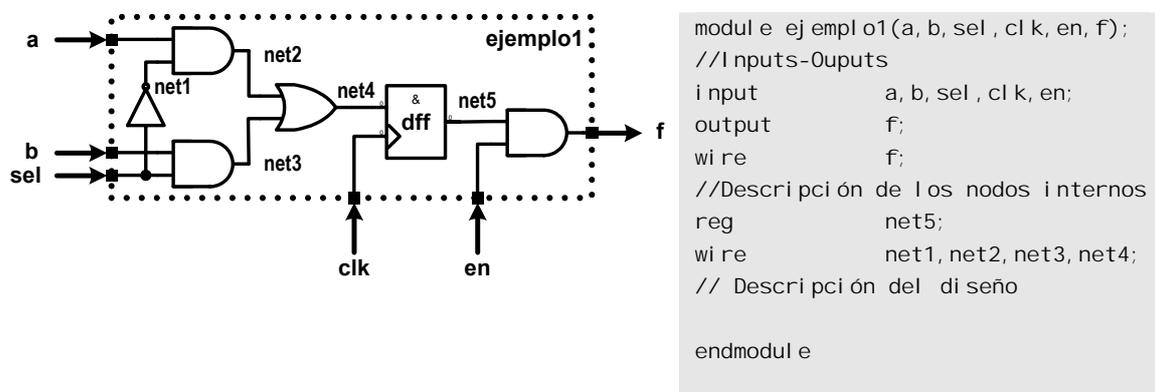


Figura 2-4 Definición de las señales de interfaz y nodos internos.

Las señales de interfaz y los nodos internos se declaran de la siguiente forma:

- Inputs.** El tipo de las señales de entrada NO SE DEFINEN, por defecto se toman como *wire*.
- Outputs.** Las salidas pueden ser tipo *wire* o *reg*, dependiendo si tienen capacidad de almacenamiento de información. OJO, en Verilog, un nodo tipo *wire* puede atacar a una salida.
- Nodos internos.** Siguen la misma filosofía que las salidas.

No piense el usuario que la declaración de un nodo tipo *reg* lleva asociado la síntesis del mismo mediante un elemento secuencial, tal y como se observa en la Figura 2-4. La Figura 2-5 muestra la descripción del ejemplo1 declarando *net4* tipo *reg*. Si observamos el diseño final vemos que es idéntico. La diferencia está en la forma de definir el código.

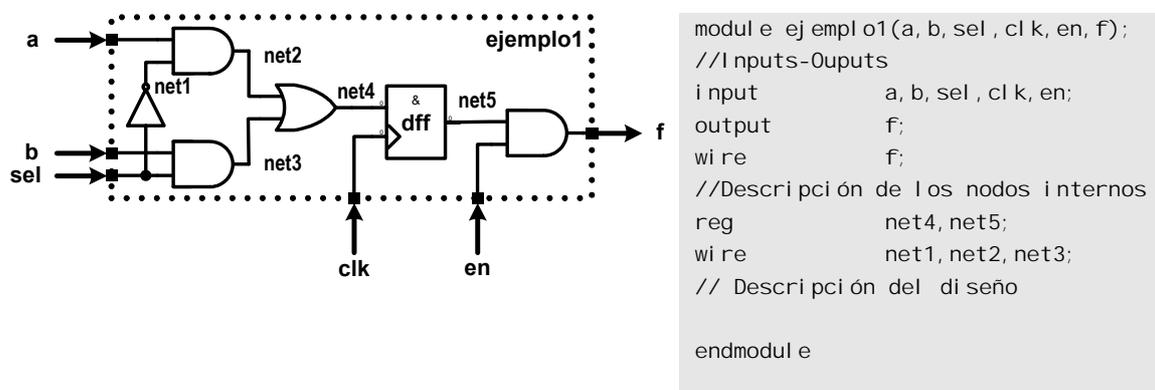


Figura 2-5 Definición de las señales de interfaz y nodos internos.

### 3 Procesos

El concepto de procesos que se ejecutan en paralelo es una de las características fundamentales del lenguaje, siendo ese uno de los aspectos diferenciales con respecto al lenguaje procedural como el lenguaje C.

Toda descripción de comportamiento en lenguaje Verilog debe declararse dentro de un proceso, aunque existe una excepción que trataremos a lo largo de este apartado. Existen en Verilog dos tipos de procesos, también denominados *bloques concurrentes*.

- Initial.** Este tipo de proceso se ejecuta una sola vez comenzando su ejecución en tiempo cero. Este proceso NO ES SINTETIZABLE, es decir no se puede utilizar en una descripción RTL. Su uso está íntimamente ligado a la realización del *testbench*.

- ✦ **Always.** Este tipo de proceso se ejecuta continuamente a modo de bucle. Tal y como su nombre indica, se ejecuta siempre. Este proceso es totalmente sintetizable. La ejecución de este proceso está controlada por una temporización (es decir, se ejecuta cada determinado tiempo) o por eventos. En este último caso, si el bloque se ejecuta por más de un evento, al conjunto de eventos se denomina *lista sensible*. La sintaxis de este proceso es pues:

```
always <temporización> o <@(lista sensible)>
```

La Figura 3-1 muestra dos ejemplos de utilización de los procesos *initial* y *always*. De estos ejemplos se pueden apuntar las siguientes anotaciones:

<pre>initial begin   clk = 0;   reset = 0;   enable = 0;   data = 0; end</pre>	<pre>always @(a or b or sel) begin //Sobra en este caso   if (sel == 1)     y = a;   else     y = b; end //Sobra en este caso</pre>
--	---

▪ Figura 3-1 Procesos *initial* y *always*.

- ✦ **Begin, end.** Si el proceso engloba más de una asignación procedural (=) o más de una estructura de control (*if-else*, *case*, *for*, etc...), estas deben estar contenidas en un bloque delimitado por *begin* y *end*.
- ✦ **Initial.** Se ejecuta a partir del instante cero y, en el ejemplo, en tiempo 0 (no hay elementos de retardo ni eventos, ya los trataremos), si bien las asignaciones contenidas entre *begin* y *end* se ejecutan de forma secuencial comenzando por la primera. En caso de existir varios bloques *initial* todos ellos se ejecutan de forma concurrente a partir del instante inicial.
- ✦ **Always.** En el ejemplo, se ejecuta cada vez que se produzcan los eventos variación de la variable *a* o variación de *b* o variación de *sel* (estos tres eventos conforman su lista de sensibilidad) y en tiempo 0. En el ejemplo, el proceso *always* sólo contiene una estructura de control por lo que los delimitadores *begin* y *end* pueden suprimirse.

Todas las asignaciones que se realizan dentro de un proceso *initial* o *always* se deben de realizar sobre variables tipo *reg* y NUNCA sobre nodos tipo *wire*. La Figura 3-2 muestra un ejemplo de asignación errónea sobre nodos tipo *wire* en un proceso *initial*.

<pre>wire      clk, reset; reg       enable, data; initial begin   clk = 0;           //Error   reset = 0;        //Error   enable = 0;   data = 0; end</pre>	<pre>reg       clk, reset; reg       enable, data; initial begin   clk = 0;   reset = 0;   enable = 0;   data = 0; end</pre>
---	--

▪ Figura 3-2 a) Error de asignación de valores a variables tipo *wire*. B) Código corregido.

En general, la asignación dentro de un proceso *initial* o *always* tiene la siguiente sintaxis:

- ✦ `variable = f(wire, reg, constante numérica)`

La variable puede ser tanto una variable interna como una señal del interfaz del módulo. La asignación puede ser de un nodo tipo *wire*, de una variable tipo *reg*, de una constante numérica o una función de todas ellas.

### 3.1 Asignación continua

La asignación continua se utiliza exclusivamente para modelar lógica combinacional. A diferencia de la asignación procedural se ejecuta de forma continua por lo que no necesita de una lista sensible. La sintaxis de este tipo de asignación es:

```
➤ assign variable #<delay> = f(wire, reg, constante numérica)
```

La variable sólo puede estar declarada tipo net (en nuestro caso tipo wire). La asignación continua debe estar declarada fuera de cualquier proceso y nunca dentro de bloques *always* o bloques *initial*. La Figura 3-3 muestra una serie de ejemplos de asignaciones continuas.

```
//El siguiente ejemplo corresponde a un buffer triestado
wire [7:0] out;
.....
assign #1 out = (enable) ? data : 8'bz;

//Asignación de una suma de operandos de 4bits a una concatenación de suma y carry de salida
reg [3:0] a, b;
wire cout;
wire [3:0] sum;
.....
assign #1 {cout, sum} = a + b cin
```

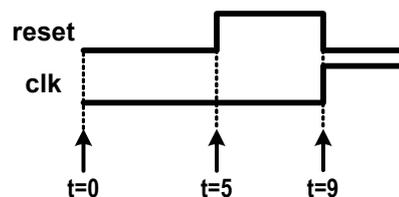
▪ Figura 3-3 Ejemplo de uso de asignación continua.

A pesar de que las asignaciones, procedurales o continuas, se ejecutan secuencialmente, es posible controlar el instante en que se producen. Esto se consigue mediante el uso de temporizaciones o eventos.

### 3.2 Temporizaciones

Las temporizaciones se consiguen mediante el uso de retardos. El retardo se especifica mediante el símbolo # seguido de las unidades de tiempo de retardo. La Figura 3-4 muestra el efecto del operador retardo. En ella se observa que los retardos especificados son acumulativos.

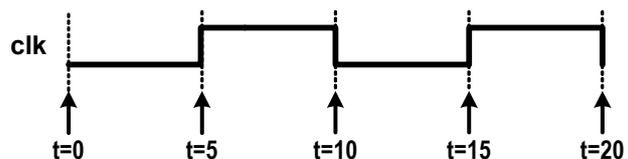
```
initial
begin
  clk = 0;
  reset = 0;
  #5 reset = 1;
  #4 clk = 1;
  reset = 0;
end
```



▪ Figura 3-4 Uso de temporizaciones en un bloque *initial*.

En el ejemplo anterior, las dos primeras asignaciones se ejecutan en tiempo 0. Cinco unidades de tiempo más tarde se realiza la tercera asignación y cuatro unidades más tarde de esta última se ejecutan la cuarta y quinta asignación. La Figura 3-5 muestra un ejemplo de control de asignación en un proceso *always*.

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



▪ Figura 3-5 Uso de temporizaciones en un bloque *always*.

### 3.3 Eventos

La segunda manera de controlar el instante en que se produce una asignación procedural o la ejecución de un proceso *always* es por el cambio de una variable, denominándose control por eventos. Para ello se emplea el carácter @ seguido del evento.

Se distinguen dos tipos de eventos:

- **Eventos de nivel.** Este evento se produce por el cambio de valor de una variable simple o de una lista sensible. Veamos los siguientes ejemplos:

Evento	Descripción
always @(a) b = b+c;	Cada vez que varía a se evalúa la expresión
always @(a or b or c) d = a+b;	Cada vez que varía a o b o c se evalúa la expresión. En este caso, a, b y c conforman la lista sensible.

- **Eventos de flanco.** Este evento se produce por la combinación de flanco/s de subida y/o bajada de una variable simple o de una lista sensible. Veamos los siguientes ejemplos:

Evento	Descripción
always @(posedge clk or posedge mr) b <= b+c;	Cada vez que se produce un flanco de subida de clk o de mr se evalúa la expresión
always @(posedge clk or negedge mr) b <= b+c;	Cada vez que se produce un flanco de subida de clk o de bajada de mr se evalúa la expresión

## 4 Módulos y jerarquías

Tal y como se comentó en el apartado 2.1, el identificador `module` se utiliza para la definición de módulos/diseños. Estos módulos pueden utilizarse para construir diseños de mayor complejidad, creando lo que se denomina un diseño con jerarquía. La manera de incluir un módulo en un diseño es:

- `master_name instante_name(port_list);`

La Figura 4-1 muestra un ejemplo de un módulo, denominado *muxt*, formado por dos módulos, denominados *mux*, y cuya declaración de interfaz es la que a continuación se muestra:

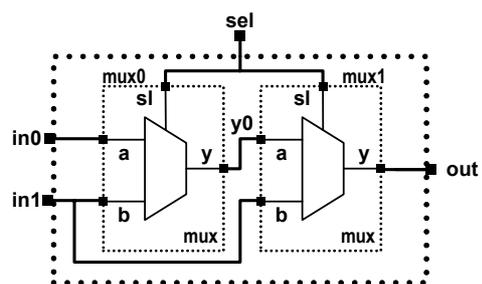
- `Module mux(a,b,sl,y); //a, b, out son señales de 8 bits`

```

module muxt(i n0, i n1, sel , out);
input [7:0] i n0, i n1;
input      sel ;
output [7:0] out;
wire [7:0] out;

//nodos internas
wire [7:0] y0;
//Conexionado por orden
mux mux0(i n0, i n1, sel , y0);
//Conexionado por nombre
mux mux1(.y(out), .b(i n1), .a(y0), .sl(sel));
endmodule

```



- Figura 4-1 Llamada a un módulo.

### 4.1 Conexionado

El conexionado de un módulo dentro de un diseño se puede realizar de dos formas: **por orden** (conexión implícita) o **por nombre** (conexión explícita).

En el conexionado por orden, utilizado en mux0 en el ejemplo anterior, las señales utilizadas en la llamada al bloque se asignan a las señales internas por **orden**. En ese caso, si se tiene en cuenta la declaración del módulo:

```
* module mux(a, b, sl, y); //a, b, out son señales de 8 bits
```

y la llamada al mismo:

```
* mux    mux0(i n0, i n1, sel, y0);
```

se producen la siguientes asignaciones:

```
* i n0 → a
* i n1 → b
* sl   → sel
* y    → y0
```

En el conexionado por nombre, utilizado en mux1 en el ejemplo anterior, se debe especificar, además de la señal, el puerto al que va conectado siguiendo la siguiente sintaxis:

```
* .(puerto) señal
```

## 5 Testbench

El propósito de un *testbench* no es otro que verificar el correcto funcionamiento de un módulo/diseño. La escritura de un *testbench* es casi tan compleja como la realización en RTL del módulo a verificar, a partir de ahora DUT (*Desing Under Test*). Una de las ventajas que presenta la escritura de un *testbench* es la posibilidad de no tener que ser sintetizable y, por tanto RTL.

Para escribir un *testbench* el diseñador debe tener siempre presente las especificaciones de diseño, en la que quedan reflejadas las funciones del diseño y, por tanto, las funciones a verificar.

### 5.1 Estructura de un testbench

La estructura de un *testbench* es la que se refleja en la Figura 5-1. Se compone de:

- **Módulo dut:** Diseño a verificar
- **Módulo test:** Módulo generador/analizador. Este módulo es el encargado de generar las señales de entrada al módulo *dut* y de analizar sus salidas.
- **Módulo tb:** Este módulo incluye los módulos anteriores. Se caracteriza por no tener entradas ni salidas. Corresponde a una declaración estructural del conexionado de los módulos *dut* y *test*. En el ejemplo de la Figura 5-1 se ha incluido un proceso (*initial*) cuya única finalidad es la de abrir una base de datos con las formas de ondas del *testbench*.

### 5.2 Módulo test

El módulo test será el encargado de proporcionar los estímulos de entrada al *dut* y de analizar sus salidas. Su realización se hace a nivel de comportamiento (behavioral) y no necesariamente utilizando código RTL. Tal y como se indicó anteriormente, la realización de este módulo implica el conocer en detalle el diseño a verificar. En este apartado se realizará la verificación del contador, denominado *cnt*, que se muestra en la Figura 5-1, por lo que en primer lugar se ha de conocer las especificaciones de este módulo.

Las especificaciones del módulo *cnt* son:

- Funcionamiento general. Se trata de un contador de 8 bits con señal de habilitación de conteo (*enable*) y señal de carga (*load*). El contador no es cíclico, por lo que al llegar al valor 8'HFF debe detenerse.
- Señal de *reset*. Cuando se activa se produce el reset asíncrono del contador a 0.
- Señal de *enable*. El contador realiza el conteo cuando la señal de *enable* está activa. En caso contrario el conteo queda detenido.
- Señal de *load*. La señal de *load* habilita la carga del conteo inicial, presente en *din*. Esta señal no tiene efecto si el conteo está habilitado. Una vez cargado, el contador debe comenzar el conteo con el valor cargado.

```

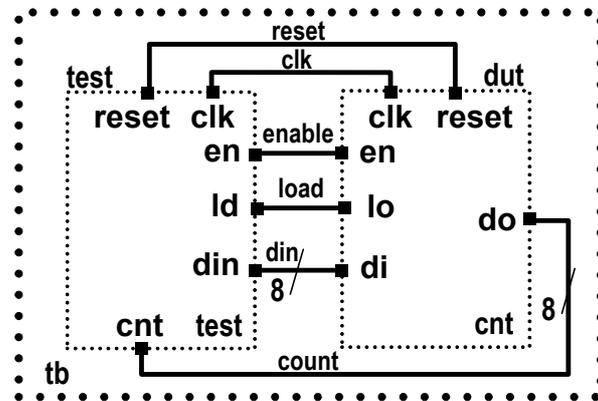
module tb;

//nodos internas
wire [7:0] din, count;
wire      reset, clk, enable, load;
//Dut
cnt dut(.clk (clk),
        .reset (reset),
        .di (din),
        .en (enable),
        .lo (load),
        .do (count));

//Test
test test(.clk (clk),
          .reset (reset),
          .en (enable),
          .din (din),
          .ld (load),
          .cnt (count));

initial
begin
    $shm_open("waves.shm");
    $shm_probe(tb, "AS");
end
endmodule

```



▪ Figura 5-1 Estructura de un test-bench.

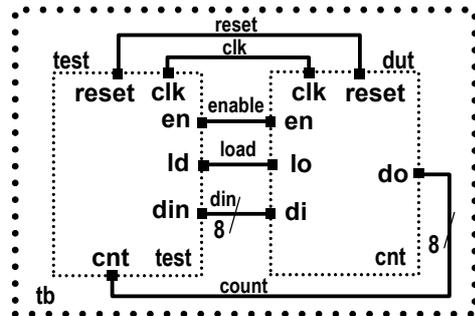
### 5.2.1 Interfaz de entrada/salida

La descripción del módulo de test comienza con la definición de las señales de interfaz. Éstas deben ser iguales, pero de sentido contrario, a las señales de interfaz del módulo *dut*. La Figura 5-2 muestra la descripción del interfaz del módulo *test* para nuestro ejemplo.

```

module test(clk, reset, en, lo, di, do);
//Señales globales
input      clk, reset;
//Entradas
input [7:0] di;
input      en;
input      lo;
//Salidas
output [7:0] do;
reg [7:0] do;
...
endmodule

```



▪ Figura 5-2 Módulo test, definición del interfaz.

### 5.2.2 Generación de estímulos

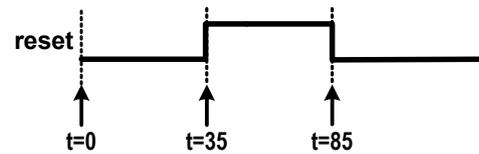
Para la generación de estímulos se utilizan los procesos *initial* o *always*. La elección del tipo de proceso viene determinada por las características de las señales a generar. Para aclarar este punto veamos la generación de diferentes señales. Dependiendo de la naturaleza de la señal, éstas se pueden generar de forma síncrona o asíncrona. La generación asíncrona se basa en el uso de retardos para activar y desactivar las señales. La generación síncrona se basa en el uso de eventos, disparados por flancos (normalmente de la señal de reloj) para activar y desactivar las señales. En este último caso hay que hacer notar un detalle muy importante. **Todas las señales que se activen de forma síncrona se harán con un retardo.**

- Señal de *reset* (asíncrona). La Figura 5-3 muestra la generación de una señal de reset de forma asíncrona. Esta señal se inicializa a '0' en tiempo 0. Después de 35 unidades de tiempo se activa a '1' y tras 50 unidades de tiempo, respecto al último evento, se pone de nuevo a '0'.

```

i n i t i a l
b e g i n
  r e s e t = 1' b0;
  #35
  r e s e t = 1' b1;
  #50
  r e s e t = 1' b0;
e n d

```



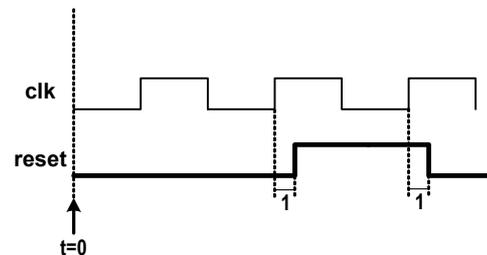
- Figura 5-3 Generación de la señal de reset (asíncrona).

- Señal de *reset* (síncrona). La Figura 5-4 muestra la generación de la señal de reset de forma síncrona. Hay que destacar que la activación/desactivación de esta señal se controla mediante eventos de, en este caso, de la señal de reloj. Tras su inicialización, se activa después de dos flancos de reloj y con un retardo de una unidad de tiempo. Se desactiva al siguiente flanco de reloj.

```

i n i t i a l
b e g i n
  r e s e t = 1' b0;
  r e p e a t (2)
  @ ( p o s e d g e c l k ) #1;
  r e s e t = 1' b1;
  @ ( p o s e d g e c l k ) #1;
  r e s e t = 1' b0;
e n d

```



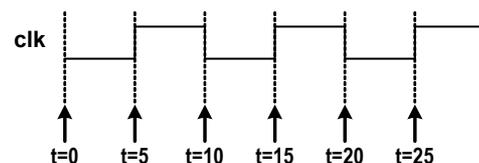
- Figura 5-4 Generación de la señal de reset (síncrona).

- Señal de *reloj*. La señal de reloj, por sus características, se genera usando otro tipo de procesos, generalmente *always*. La Figura 5-5 muestra la generación de una señal de reloj de 5 unidades de tiempo de semiperiodo. Hacer notar que para que la señal de reloj se genere de forma correcta hay que inicializar la variable *clk* a cero.

```

`d e f i n e      S P E R 5
. . .
a l w a y s # ^ S P E R c l k = ! c l k ;
i n i t i a l
  b e g i n
    c l k = 1' b0
  e n d

```



- Figura 5-5 Generación de la señal de reloj.

- Señales de *control/datos*. Al igual que ocurría con el reset, estas señales pueden ser asíncronas o síncronas. En este último caso habrá que tener en cuenta la señal de reloj para generarlas. La Figura 5-6 muestra un test para el contador que estamos desarrollando.

Hay que destacar como en el módulo de test presentado existen dos procesos que se ejecutan en paralelo. El primero de ellos corresponde al *always* para la generación de la señal de reloj. El segundo corresponde al bloque *initial*. Debido a la existencia de un bloque *always* y para poder parar la simulación se hace necesario el uso del comando *\$finish* que, al ejecutarse, para la simulación.

```

//Señal de reloj
always #`SPER clk = !clk;
//Reset, señales de control y datos
initial
begin
  reset = 1'b0;
  #35
  reset = 1'b1;
  en    = 1'b0;
  ld    = 1'b0;
  din   = 8'b0;
  #50
  reset = 1'b0
//Enable count
  @(posedge clk) #1;
  en    = 1'b1;
//Dejar 30 ciclos contando
  repeat(30)
    @(posedge clk) #1;
//Deshabilitar contaje durante 10 ciclos
  en    = 1'b0;
  repeat(10)
    @(posedge clk) #1;
//Volvemos a habilitar contaje
  @(posedge clk) #1;
  en    = 1'b1;
//Habilitamos carga de nuevo dato
  @(posedge clk) #1;
  ld    = 1'b1;
  din   = 8'h4A;
  @(posedge clk) #1;
  ld    = 1'b0;
//Esperamos 10 ciclos de reloj y repetimos la operación
//deshabilitando el contaje
  @(posedge clk) #1;
  en    = 1'b0;
  ld    = 1'b1;
  din   = 8'hF0;
  @(posedge clk) #1;
  en    = 1'b1;
//Habilitamos 30 ciclos de reloj y finalizamos
  repeat(30)
    @(posedge clk) #1;
  $finish;
end

```

▪ Figura 5-6 Test del contador cnt.

## 6 Modelado de memorias en Verilog

Con la intención de facilitar el modelado de memorias, Verilog acepta la declaración de arrays de dos dimensiones. En Verilog las memorias se modelan mediante arrays de registros. Cualquier palabra de la memoria puede ser accedida mediante un índice. El acceso a un determinado bit de una palabra se realiza mediante la declaración de una variable temporal. La sintaxis de la declaración es:

```

reg [wordsi ze: 0] array_name[arraysi ze: 0]

```

Así, para el caso de querer modelar una memoria de 128 palabras de 8 bits, la declaración sería:

```
reg [7:0] core[127:0]
```

La Figura 6-1 muestra el modelado de una memoria RAM.

```
module reg_sram (data, q, clk, we, addr);
parameter width = 8;
parameter depth = 128;
parameter addr = 7;

input clk, we;
input [addr-1:0] addr;
input [width-1:0] data;
output [width-1:0] q;
wire [width-1:0] q;

reg [width-1:0] core[depth-1:0]

always @(posedge clk)
if (we core[addr] <= #1 data;

assign #1 q = core[addr]
endmodule
```

▪ Figura 6-1 Memoria RAM.

## 6.1 Uso de parámetros

En el ejemplo anterior se ha hecho uso de los parámetros consiguiendo así una memoria parametrizable. La parametrización de un módulo se consigue mediante el uso del token *parameter*. La sintaxis de declaración de un parámetro es:

```
parameter <nombre> = <valor por defecto>
```

Los parámetros pueden especificarse durante la llamada al módulo. En caso de no especificar el parámetro, éste toma el valor por defecto. La muestra dos llamadas al módulo *reg\_sram*. La primera de ellas, con valores por defecto, consigue una memoria de 128x8 bits, mientras que la segunda, la memoria es de 1024x16 bits. Los parámetros de llamada al módulo se deben especificar en el mismo orden en que fueron declarados dentro del módulo.

```
...
reg_sram ram128x8(.data(), .q(), .clk(), .we(), .addr()); //128x8 bits
reg_sram #(16, 1024, 10)ram1024x16(.data(), .q(), .clk(), .we(), .addr()); //1024x16 bits
```

▪ Figura 6-2 Memoria RAM.

## 7 Operadores en Verilog

Los operadores aritméticos y/o lógicos se utilizan para construir expresiones. Estas expresiones se realizan sobre uno o más operandos. Estos últimos pueden ser nodos, registros constantes, etc..

### 7.1 Operadores aritméticos

De un operando:

+	a = +8'h01	Resultado: a = 8'h01
-	a = -8'h01	Resultado: a = 8'hFF

De dos operandos:

+	a = b + c;
---	------------

➤	“-“	$a = b - c;$	
➤	“*“	$a = b * c;$	
➤	“/“	$a = b / c;$	Resultado: Se devuelve la parte entera de la división
➤	“%*“	$a = b \% c;$	Resultado: Se devuelve el módulo de la división

El resultado del módulo toma el signo del primer operando.

Si algún bit del operando tiene el valor “x”, el resultado completo es “x”.

Los números negativos se representan en complemento a 2.

## 7.2 Operadores relacionales

➤	“<“	$a < b$	a es menor que b
➤	“>“	$a > b$	a es mayor que b
➤	“<=“	$a <= b$	a es menor o igual que b
➤	“>=“	$a >= b$	a es mayor o igual que b

El resultado que se devuelve es:

0	si la condición no se cumple
1	si la condición se cumple
x	si alguno de los operandos tiene algún bit a “x”

## 7.3 Operadores de igualdad

➤	“==“	$a == b$	a es igual a b
➤	“!=“	$a != b$	a es diferente de b
➤	“===“	$a === b$	a es igual a b, incluyendo “x” y “z”
➤	“!==“	$a !== b$	a es diferente a b, incluyendo “x” y “z”

Los operandos se comparan bit a bit, rellenando con ceros para ajustar el tamaño en caso de que no sean de igual ancho. Estas expresiones se utilizan en condicionales. El resultado es:

0	si se cumple la igualdad
1	si no se cumple la igualdad
x	únicamente en las igualdades “==” o “!=” si algún operando contiene algún bit a “x” o “z”

## 7.4 Operadores lógicos

➤	“!”	negación lógica
➤	“&&“	AND lógica
➤	“  “	OR lógica

Las expresiones con “&&” o “||” se evalúan de izquierda a derecha. Estas expresiones se utilizan en condicionales. El resultado es:

0	si la relación es falsa
1	si la relación es verdadera
x	si algún operando contiene algún bit a “x” o “z”

## 7.5 Operadores bit a bit (bit-wise)

➤	“~“	negación
➤	“&“	AND
➤	“ “	OR
➤	“^“	OR exclusiva

Los citados operadores pueden combinarse obteniendo el resto de operaciones, tales como ~& (AND negada), ~^ (OR exclusiva negada), etc... El cálculo incluye los bits desconocidos (x) en la siguiente manera:

~x	= x
0&x	= 0
1&x	= x&x = x
1 x	= 1
0 x	= x x = x
0^x	= 1^x = x^x = x

## 7.6 Operadores de reducción

➤	“&“	AND	Ejemplo: &a	Devuelve la AND de todos los bits de a
---	-----	-----	-------------	--

“~&”	AND negada	Igual que el anterior
“ ”	OR	Ejemplo:  a Devuelve la OR de todos los bits de a
“~ ”	OR negada	Igual que el anterior
“^”	OR exclusiva	Ejemplo: ^a Devuelve la OR exclusiva de todos los bits de a
“~^”	OR exclusiva negada	Igual que el anterior

Estos operadores de reducción realizan las operaciones bit a bit sobre un solo operando. Las operaciones negadas operan como AND, OR o EXOR pero con el resultado negado.

## 7.7 Operadores de desplazamiento

“<<”	Desplazamiento a izquierda	Ejemplo: a = b << 2;
“>>”	Desplazamiento a derecha	Ejemplo: a = b >> 2;

Notas: Los bits desplazados se rellenan con ceros.

## 7.8 Operador de concatenación

“{ }”	Concatenación de operandos. Los operandos se separan por comas
-------	--

Ejemplos: {a,b[3:0],c} Si a y c son de 8 bits, el resultado es de 20 bits  
 {3{a}} Es equivalente a {a,a,a}  
 {b,3{c,d}} Es equivalente a {b,c,d,c,d,c,d}

Nota: No se permite la concatenación con constantes sin tamaño.

## 7.9 Operador condicional

“?”	El formato de uso de dicho operador es (condición) ? trae_expr : false_expr
-----	---

Ejemplos: out = (enable) ? a : b; La salida out toma el valor de a si enable está a 1, en caso contrario toma el valor de b

## 7.10 Precedencia de los operadores

El orden de precedencia de los diferentes operadores es el siguiente:

Operador	Símbolo
Unary, Multiplicación, División, Módulo	+, -, *, %
Suma, Resta, Desplazamientos	+, -, <<, >>
Relación, Igualdad	<, >, <=, >=, ==, !=, ===, !==
Reducción	&, !&, ^, ~^,  , ~
Lógicos	&&,
Condicional	?:

- Figura 7-1 Orden de precedencia de los operadores en Verilog.

## 8 Estructuras más comunes

En este apartado se presentan las estructuras más comunes para la descripción de comportamiento en Verilog. A continuación se presentan cada una de ellas por separado indicando, además de un ejemplo, si son sintetizables o no.

### 8.1 Sentencias condicionales if – else

La sentencia condicional *if – else* controla la ejecución de otras sentencias y/o asignaciones (estas últimas siempre procedurales). El uso de múltiples sentencias o asignaciones requiere el uso de *begin – end*. La sintaxis de la expresión es:

```

if (condición) //Condición simple
    sentencias;
if (condición) //Condición doble
    sentencias;
else
    sentencias;
if (condición1) // Múltiples condiciones
    sentencias;
else if (condición2)

```

```

sentenci as;
. . . .
el se
sentenci as;

```

La siguiente figura muestra un ejemplo de uso del condicional *if – else*.

```

//Condi ci onal si mpl e
i f (enabl e)
  q<= #1 data;
//Condi ci onal mul ti pl e
i f (reset == 1' b1)
  count<= #1 8' b0;
el se i f (enabl e == 1' b1 && up_en == 1' b1)
  count<= #1 count + 1' b1;
el se i f (enabl e == 1' b1 && down_en == 1' b1)
  count<= #1 count – 1' b1;
el se
  count<= #1 count;

```

- Figura 8-1 Ejemplos de uso de sentencias condicionales.

## 8.2 Sentencia case

La sentencia case evalúa una expresión y en función de su valor ejecuta la sentencia o grupos de sentencias agrupadas en el primer caso en que coincida. El uso de múltiples sentencias o asignaciones requiere el uso de *begin – end*. En caso de no cubrir todos los posibles valores de la expresión a evaluar, es necesario el uso de un caso por defecto (*default*). Este caso se ejecutará siempre y cuando no se cumplan ninguno de los casos anteriores. La sintaxis de la expresión es:

```

case (expresi ón)
  <caso1>: sentenci as;
  <caso2>: begi n
    sentenci as;
    sentenci as;
    end
  <caso3>: sentenci as;
  . . . .
  . . . .
  <defaul t>: sentenci as;
endcase

```

En el ejemplo de la Figura 8-2 hay que destacar que no se están cubriendo todos los casos ya que no se evalúan los posibles valores de la variable *sel* en estado de alta impedancia (z) o desconocido (x). La sentencia case reconoce tanto el valor z como x como valores lógicos.

```

//Ej empl o
case (sel )
  2' b00: y = i na;
  2' b01: y = i nb;
  2' b10: y = i nc;
  2' b11: y = i nd;
  defaul t: $di spl ay("Val or de sel erróneo");
endcase

```

- Figura 8-2 Ejemplos de uso de sentencia case.

### 8.3 Sentencia casez y casex

Estas sentencias corresponden a versiones especiales del case y cuya particularidad radica en que los valores lógicos z y x se tratan como valores indiferentes.

- ✦ **casez**: usa el valor lógico z como valor indiferente
- ✦ **casex**: toma como indiferentes tanto el valor z como el valor lógico x

La Figura 8-3 muestra un ejemplo de uso de casez.

```
casez (opcode)
  4'b1zzz: out = a; // Esta sentencia se ejecuta siempre que el bit más significativo
                 sea 1.
  4'b01??: out = b; // El símbolo ? siempre se considera como "indiferente", luego en
                 este caso es equivalente a poner 4'b01zz
  4'b001?: out = c;
  default: $display("Error en el opcode");
endcase
```

- Figura 8-3 Ejemplo de uso de casez.

### 8.4 Sentencias de bucle

Todas las sentencias de bucles en Verilog deben estar contenidas en bloques procedurales (*initial* o *always*). Verilog soporta cuatro sentencias de bucles.

#### 8.4.1 Sentencia forever

El bucle *forever* se ejecuta de forma continua, sin condición de finalización. Su sintaxis es:

```
✦ forever <sentencias>
```

En caso de englobar más de una sentencia o asignación, éstas se deben incluir entre *begin – end*.

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

- Figura 8-4 Ejemplo de uso de *forever*.

#### 8.4.2 Sentencia repeat

El bucle *repeat* se ejecuta un determinado número de veces, siendo este número su condición de finalización. Su sintaxis es:

```
✦ repeat (<número>) <sentencias>
```

En caso de englobar más de una sentencia o asignación, éstas se deben incluir entre *begin – end*.

```
if (opcode == 10) //Realización de una operación de rotación
  repeat (8) begin
    temp = data[7];
    data = {data <<1, temp};
  end
```

- Figura 8-5 Ejemplo de uso de *repeat*.

#### 8.4.3 Sentencia while

El bucle *while* se ejecuta mientras la condición que evalúa sea cierta. La condición que se especifica expresa la condición de ejecución del bucle. Su sintaxis es:

```
✦ while (<expresión>) <sentencias>
```

En caso de englobar mas de una sentencia o asignación, éstas se deben incluir entre *begin – end*.

```
loc = 0;
if (data = 0) //Ejemplo de cálculo del bit más significativo
    loc = 32;
else while (data[0] == 1'b0) begin
    loc = loc + 1;
    data = data >> 1;
end
```

- Figura 8-6 Ejemplo de uso de *while*.

#### 8.4.4 Bucle *for*

El bucle *for* es similar a aquellos utilizados en los lenguajes de programación. Su sintaxis es:

```
* for (<valor inicial>, <expresión>, <incremento>) <sentencias>
```

En caso de englobar mas de una sentencia o asignación, éstas se deben incluir entre *begin – end*.

```
for (i=0, i<64, i=i+1) //Inicialización de memoria RAM
    ram[i] = 0;
```

- Figura 8-7 Ejemplo de uso de *for*.

## 9 Directivas de compilación

Verilog ofrece una serie de directivas de compilación que permiten, entre otras cosas, obtener diferentes códigos a partir de una única descripción. A continuación se detallan las más comunes. La sintaxis para la definición de una directiva es:

```
* `directiva <nombre> <valor>
```

### 9.1 Define

La directiva *define* permite definir un valor. Ejemplo:

```
`define TD 1
...
assign #TD q = data_out;
```

- Figura 9-1 Ejemplo de definición y uso de la directiva *define*.

### 9.2 Include

La directiva *include* permite incluir un fichero. En este caso, el fichero a incluir puede contener, por ejemplo, la definición de otros módulos.

```
`include "modulos.v"
...

```

- Figura 9-2 Ejemplo de definición de la directiva *include*.

### 9.3 Ifdef – else - endif

La directiva *ifdef* permite compilar un código siempre y cuando se haya definido el símbolo al que hace referencia.

```
`define SUMA
...

```

```
always @(posedge clk or posdeg reset)
if (reset) data <= #1 0;
else
`ifdef SUMA
data <= #1 a + b;
`else
data <= #1 a - b;
`endif
```

- Figura 9-3 Ejemplo de definición de la directiva *ifdef* – *else* - *endif*.

En el ejemplo anterior el resultado de la compilación será la asignación de  $a + b$  a *data* ya que el símbolo SUMA ha sido definido previamente.

## 9.4 Timescale

La directiva *timescale* permite definir las unidades de tiempo con las que se va a trabajar. La sintaxis de esta directiva es:

```
* `timescale <unidad de tiempo> / <resolución>
```

```
`timescale 1ns /100ps //Definimos el nanosegundo como unidad de tiempo.
...
a = #1 b + c; //La asignación tiene lugar tras 1 nanosegundo
```

- Figura 9-4 Ejemplo de definición de la directiva *timescale*.

## 10 Funciones del sistema

Verilog ofrece una serie de funciones específicas. Estas funciones no son sintetizables y su uso está enfocado a la construcción de *testbench*. Sin embargo, existen directivas que permiten el uso de estas funciones en el código RTL. Estas directivas no se tratan en el presente tutorial. Como nota común destacar que todas las funciones en Verilog comienzan con el carácter \$. A continuación se muestran algunas de las funciones más comunes. Todas las uncciones deben estar declaradas dentro de un bloque.

- **\$time:** Devuelve el tiempo actual de la simulación. Ejemplo:  
A = \$time;
- **\$display:** Esta función imprime, en el momento de ejecutarse, en pantalla un mensaje. Se puede añadir, además, una lista de variables. El mensaje debe ser declarado entre "", seguido de la lista de variables a imprimir. Al final del mensaje se introduce un retorno de carro y avance de línea. Ejemplo:  
\$display ("Tiempo: %d, Valor: %h, %o, %b", \$time, data);  
%b = binario; %o = octal; %h = hexadecimal; %d = decimal;  
%c = carácter; %s = cadena de caracteres;
- **\$monitor:** Esta función tiene la misma sintaxis que \$display. Se utiliza para sacar por pantalla un mensaje pero siempre que cambie una de las variables de su lista de variables. Su lectura se realiza una única vez y el mensaje se imprime cada vez que varía el valor de alguna de sus variables. Ejemplo:  
\$monitor ("Input: %d, Valor: %h, %o, %b", in, data); //Se imprime el mensaje cada vez que varía in o data
- **\$monitoroff:** Esta función detiene la monitorización de variables que se ejecuta debido al comando \$monitor. Ejemplo:  
\$monitoroff;
- **\$monitoron:** Esta función habilita la monitorización de variables deshabilitada al ejecutar \$monitoroff. Ejemplo:  
\$monitoron;
- **\$fopen:** Esta función permite abrir un fichero, devolviendo, sobre una variable definida como integer el identificador del fichero (file\_id). Ejemplo:  
id = \$fopen("hola.txt"); //"id" debe estar definido como integer. Se crea el fichero "hola.txt"
- **\$fclose:** Esta función permite cerrar un fichero. El fichero a cerrar se especifica mediante el identificador. Ejemplo:

- `$fclose(id);`
- **\$display:** Esta función imprime, en el momento de ejecutarse, en un fichero un mensaje. Se puede añadir, además, una lista de variables. El mensaje debe ser declarado entre "", seguido de la lista de variables a imprimir. Al final del mensaje se introduce un retorno de carro y avance de línea. Ejemplo:  
`$display (file_id, "Tiempo: %d, Valor: %h, %o, %b", $time, data);`
- **\$write:** Esta función es idéntica al `$display` pero no introduce avance de línea ni retorno de carro. Ejemplo:  
`$write ("Tiempo: %d, Valor: %h, %o, %b", $time, data);`
- **\$fdisplay:** Esta función es idéntica al `$display` pero no introduce avance de línea ni retorno de carro. Ejemplo:  
`$fwrite (file_id, "Tiempo: %d, Valor: %h, %o, %b", $time, data);`

## 11 Tareas en Verilog

Las tareas son usadas en la mayor parte de los lenguajes de programación, conocidas comúnmente como procedimientos o subrutinas. Las tareas tienen la finalidad de reducir el código del diseño y sus llamadas se realizan en tiempo de compilación. Las tareas ofrecen la posibilidad de permitir parámetros de entrada y/o salida usando para ello nodos declarados en el módulo que realiza las llamadas. Las tareas:

- Se definen en el módulo en el que se utilizan, si bien pueden estar definidas en un fichero aparte y ser incluidas mediante la directiva *include* de Verilog.
- Pueden incluir retardos del tipo *#delay*, *posedge* o *negedge*.
- Pueden tener cualquier número de entradas/salidas.
- La definición de entradas y/o salidas marcan el orden en que éstas deben pasarse a la tarea.
- Las variables que se declaran dentro de una tarea son locales a dicha tarea.
- Las tareas pueden usar y/o asignar valores a cualquier señal declarada como global en el diseño.
- Una tarea puede llamar a otra tarea o función.
- Pueden utilizarse para modelar tanto lógica combinacional como secuencial.
- La llamada a una tarea no se puede realizar dentro de una expresión.

### 11.1 Sintaxis y llamada a una tarea

La sintaxis de una tarea es la que se muestra en la siguiente figura.

```

task <name>      //Nombre de la tarea
inputs;         //Entradas
outputs;        //Salidas

<internal >;   //Variables internas
begin
  código de la tarea;
end
endtask

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

```

- Figura 11-1 Descripción de una tarea. Ejemplo de definición de una tarea.

La Figura 11-2 muestra una posible llamada a la función declarada en el ejemplo anterior.

```

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);
...

```

```
endmodule
```

- Figura 11-2 Ejemplo de llamada a una tarea.

## 12 Funciones en Verilog

Una función en Verilog se asemeja a una tarea con las siguientes salvedades:

- No pueden incluir elementos de retardo.
- Pueden tener cualquier número de entradas pero sólo una salida.
- Solo se pueden utilizar para modelar lógica combinacional.
- Una función puede llamar a otra función pero no a una tarea.

### 12.1 Sintaxis y llamada a una función

La sintaxis de una función es la que se muestra en la siguiente figura.

<pre>function &lt;name&gt;          //Nombre de la función inputs;                  //Entradas  &lt;internal&gt;;             //Variables internas begin   código de la función; end endfunction</pre>	<pre>function logic_oper; input  [7:0]  numa; input  [7:0]  numb; begin   logic_oper = numa &amp; numb; end endfunction</pre>
--	---

- Figura 12-1 Descripción de una función. Ejemplo de definición de una función.

La Figura 12-2 muestra una posible llamada a la función declarada en el ejemplo anterior.

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

- Figura 12-2 Ejemplo de llamada a una función.