1. Introducción

Verilog es un lenguaje de descripción de hardware. Fue desarrollado a mediados de la década de 1980 y luego transferido al IEEE (Instituto de Ingenieros Eléctricos y Electrónicos). El lenguaje está definido formalmente por el estándar IEEE 1364. El estándar fue ratificado en 1995 (referido como Verilog-1995) y revisado en 2001 (referido como Verilog-2001). Muchas mejoras útiles se agregan en la versión revisada. En estos ejemplos utilizaremos Verilog-2001.

Verilog está diseñado para describir y modelar un sistema digital en varios niveles y es un lenguaje extremadamente complejo. El enfoque de este libro está en el diseño de hardware más que en el lenguaje. En lugar de cubrir todos los aspectos de Verilog, presentamos las construcciones clave de síntesis de Verilog mediante un conjunto de ejemplos.

Aunque la sintaxis de Verilog se parece un poco a la del lenguaje C, su semántica (es decir, "significado") se basa en la operación de hardware concurrente y es totalmente diferente de la ejecución secuencial de C. La sutileza de algunas construcciones del lenguaje y cierto comportamiento no-determinista de Verilog puede conducir a errores difíciles de detectar e introducir una discrepancia entre la simulación y la síntesis. La codificación de estos ejemplos sigue una filosofía de "mejor seguro que con errores". En lugar de escribir códigos cortos y rápidos, la atención se centra en el estilo y las construcciones que son claras y sintetizables y que pueden describir con precisión el hardware deseado.

2. Descripción general

input		**output**
$i0$	$i1$	eq
0	0	1
0	1	0
1	0	0
1	1	1

Tabla 1: Tabla de verdad del comparador de igualdad de 1-bit

Considere un comparador de igualdad de 1-bit con dos entradas, $i0$ e $i1$, y una salida, eq. La señal eq se afirma cuando $i0$ e $i1$ son iguales. La tabla de verdad de este circuito se muestra en la Tabla 1.

Suponga que queremos usar compuertas lógicas básicas, que incluyen `not`, `and`, `or` y `xor`, para implementar el circuito. Una forma de describir el circuito es usar un formato de suma de productos. La expresión lógica es

$$eq = i0 \cdot i1 + \bar{i0} \cdot \bar{i1} \tag{1}$$

Un posible código Verilog se muestra en el Listado 1. Examinamos las construcciones del lenguaje y declaraciones de este código en las siguientes secciones.

Listado 1: Implementación del comparador de 1-bit

```
module eq1
    // I/O ports
    (
    input wire i0, i1,
    output wire eq
    );

    // signal declaration
    wire p0, p1;

    // body
    // sum of two product terms
    assign eq = p0 | p1
    // product terms
    assign p0 = ~i0 & ~i1;
    assign p1 = i0 & i1;
endmodule
```

La mejor manera de entender un programa HDL (lenguaje de descripción de hardware) es pensar en términos de circuitos de hardware. Este programa consta de tres partes. La parte de los puertos I/O describe los puertos de entrada y salida de este circuito, que son $i0$, $i1$ y eq, respectivamente. La porción de declaración de señales especifica las señales de conexión internas, que son $p0$ y $p1$. La parte del cuerpo describe la organización interna del circuito. Hay tres asignaciones continuas en este código. Cada uno puede considerarse como una parte del circuito que realiza ciertas operaciones lógicas simples. Examinaremos las construcciones del lenguaje y las declaraciones de este código en la siguiente sección.

La representación gráfica de este programa se muestra en la Figura 1. Las tres asignaciones continuas constituyen las tres partes del circuito. Las conexiones entre estas partes se especifican implícitamente por los nombres de la señal y el puerto.

3. Elementos léxicos básicos y tipos de datos

3.1. Elementos léxicos

Identificador Un identificador da un nombre único a un objeto, como `eq1`, `i0` o `p0`. Se compone de letras, dígitos, el carácter de subrayado (_) y el signo peso ($). El signo $ generalmente se usa con una tarea o función del sistema.

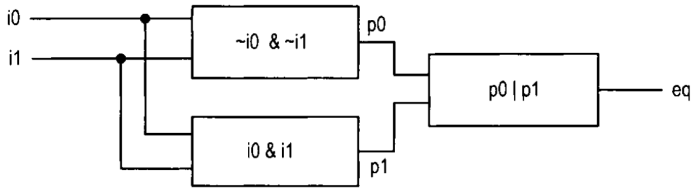

Figura 1: Representación gráfica del programa correspondiente al comparador de 1-bit.

El primer carácter de un identificador debe ser una letra o un guión bajo. Es una buena práctica dar a un objeto un nombre descriptivo. Por ejemplo, `mem-addr-en` es más significativo que `mae` para una señal de activación de dirección de memoria.

Verilog es un lenguaje que distingue entre mayúsculas y minúsculas. Por lo tanto, `data_bus`, `Data_bus` y `DATA_BUS` se refieren a tres objetos diferentes. Para evitar confusiones, debemos abstenernos de usar mayúsculas y minúsculas para diferenciar los identificadores.

Palabras reservadas Las palabras reservadas son identificadores predefinidos que se utilizan para describir las construcciones del lenguaje. El Listado 1 utiliza palabras clave tales como `module` y `wire`.

Espacio en blanco El espacio en blanco, que incluye los caracteres de espacio, tabulación y nueva línea, se usa para separar los identificadores y se puede usar libremente en el código Verilog. Podemos usar espacios en blanco adecuados para formatear el código y hacerlo más legible.

Comentarios Un comentario se utiliza para fines de documentación y será ignorado por el software. Verilog tiene dos formas de comentarios. Un comentario de una línea comienza con `//`, como en

```
// This is a comment.
```

Un comentario de múltiples líneas se encuentra ubicado entre `/*` y `*/`, como en

```
/* This is comment line 1.
   This is comment line 2.
   This is comment line 3. */
```

4. Tipo de datos

4.1. Sistema de cuatro valores

En la mayoría de los tipos de datos se utilizan cuatro valores básicos:

- 0: para "lógica 0"(o condición falsa)
- 1: para "lógica 1"(condición verdadera)
- z: para el estado de alta impedancia
- x: para un valor desconocido

El valor *z* corresponde a la salida de un buffer de tres estados. El valor *x* generalmente se usa en modelado y simulación y representa un valor que no es 0, 1 ni *z*, sino que representa un valor no inicializado de una entrada o un conflicto en una salida.

4.2. Grupos de tipos de datos

Verilog tiene dos grupos principales de tipos de datos: **net** y **variable**.

4.2.1. Grupo net

Los tipos de datos en el grupo **net** representan las conexiones físicas entre componentes de hardware. Se utilizan como las salidas de asignaciones continuas y como las señales de conexión entre diferentes módulos. El tipo de datos más utilizado en este el grupo es **wire**. Como su nombre lo indica, representa un *cable* de conexión.

El tipo de dato **wire** representa una señal de 1 bit como sigue:

```
wire p0, p1; // dos señales de 1 bit
```

Cuando una colección de señales se agrupa en un bus, se puede representar la misma utilizando un arreglo unidimensional (vector) como sigue:

```
wire [7:0] data1, data2; // 8-bits de datos
wire [31:0] addr; // dirección de 32-bits
wire [0:7] revers_data; // índice ascendiente debería evitarse
```

Mientras que el rango del índice puede ser descendente (como en `[7:0]`) o ascendente (como en `[0:7]`), se prefiere el primero ya que la posición más a la izquierda (es decir 7) corresponde al bit más significativo (MSB) de un número binario.

A veces se necesita un arreglo bidimensional para representar una memoria. Por ejemplo, una memoria de 32 por 4 (es decir, una memoria que tiene 32 palabras y cada palabra tiene 4 bits de ancho) puede ser representado como:

```
wire [3:0] mem1 [31:0]; // memoria de 32 por 4
```

Existen otros tipos de datos del grupo **net** que implican cierto comportamiento lógico o funcionalidad, tal como**wand** (para una conexión *and* cableada) y **supply0** (para la conexión a tierra del circuito). Nosotros no usaremos estos tipos de datos. Verilog-2001 también permite el tipo de datos **signed**.

4.2.2. Grupo variable

Los tipos de datos en el grupo de variables representan el almacenamiento abstracto en el modelado de comportamiento y se utilizan en los resultados de las asignaciones de procedimientos. Hay cinco tipos de datos en este grupo: `reg`, `integer`, `real`, `time` y `realtime`. El tipo de datos más utilizado en este grupo es `reg` y se puede sintetizar. El circuito inferido puede contener o no componentes de almacenamiento físico. Los últimos tres tipos de datos solo se pueden usar en modelado y simulación, y el uso del tipo de datos entero se analizará más adelante.

En Verilog-1995, el grupo de variables se conoce como *grupo de registros*. Dado que este término es el mismo para un registro de hardware físico (es decir, una colección de flip-flops), se cambia en la documentación de Verilog-2001 para evitar confusiones. Nosotros intentaremos utilizar el término *variable* para el tipo de datos y usamos el término *registro* para el circuito de un registro físico.

4.2.3. Representación de números

Una constante entera en Verilog puede representarse en varios formatos. Su forma general es:

```
[signo][tamaño]'[base][valor]
```

El término [base] especifica la base del número, la cual puede ser una de las siguientes:

- b o B: binario
- o u O: octal
- h o H: hexadecimal
- d o D: decimal

El término [valor] especifica el valor del número en su base correspondiente. Se puede incluir el caracter "_" para mayor claridad.

El término [tamaño] especifica el número de bits en un número. Este parámetro es opcional. Cuando se define el término [tamaño] el número se conoce como *sized number*, caso contrario se lo conoce como *unsized number*.

Cuando se define un número con tamaño (**sized number**) se especifica el número de bits de forma explícita. Si el tamaño del valor es más pequeño que el término [tamaño] especificado, los ceros se rellenan al frente para extender el número, excepto en varios casos especiales. El valor z o x se utilizan para rellenar si el MSB es z o x, y el MSB se rellena si se utiliza el tipo de datos **con signo** (*signed*). Algunos ejemplos de números con tamaño se muestran en la parte superior de la Tabla 2.

Un número sin tamaño (**unsized number**) omite el término [tamaño]. Su tamaño real depende en la computadora host, pero debe ser de al menos 32

bits. El término ’[base] también se puede omitir si el número está en formato decimal. Asumimos que se utilizan 32 bits en la máquina host. Varios ejemplos de de números sin tamaño se muestran en la parte inferior de la Tabla 2.

número	**valor almacenado**	**comentario**
5’b11010	11010	
5’b11_010	11010	_ ignorado
5’o32	11010	
5’h1a	11010	
5’d26	11010	
5’b0	00000	extensión del 0
5’b1	00001	extensión del 0
5’bz	zzzzz	extensión del z
5’bx	xxxxx	extensión del x
5’bx01	xxx01	extensión del x
-5’b00001	11111	complemento a 2 del 00001
’b11010	00000000000000000000000000011010	extensión a 32 bits
’hee	00000000000000000000000011101110	extensión a 32 bits
1	00000000000000000000000000000001	extensión a 32 bits
-1	11111111111111111111111111111111	extensión a 32 bits

Tabla 2: Tabla de verdad del comparador de igualdad de 1-bit

5. Esqueleto de un programa

Como su nombre lo indica, HDL se usa para describir hardware. Cuando desarrollamos o examinamos un código Verilog, es mucho más fácil de comprender si pensamos en términos de .organización de hardware.en lugar de pensar en un .algoritmo secuencial". La mayoría de los códigos de Verilog en este libro siguen el esqueleto básico que se muestra en el Listado 1. Consta de tres partes: declaración de puertos I/O, declaración de señales y cuerpo del módulo.

5.1. Declaración de puertos

La declaración del módulo y de los puertos del Listado 1 son:

```
module eq1
    (
    input wire i0, i1,
    output wire eq
    );
```

La declaración de I/O especifica los modos, tipos de dato y nombre de los puertos del módulo. La construcción genérica de un módulo es:

```
module [module_name]
    (
    [mode]  [data_type]  [port_names],
    [mode]  [data_type]  [port_names],
    ...
    [mode]  [data_type]  [port_names]
    );
```

El término `[mode]` puede ser **input**, **output** o **inout**, que representa un puerto de entrada, salida o bidireccional, respectivamente. Tenga en cuenta que no hay coma en la última declaración. El término `[data_type]` puede ser omitido si se define un elemento de tipo `wire`.

En Verilog-1995 los nombres de los puertos, modos y tipos de dato son declarados de forma separada. Por ejemplo, la misma declaración anterior se realizaría de la siguiente manera

```
module eq1 (i0, i1, e1);  // only port names in brackets
    // declare mode
    input i0, i1;
    output eq;
    // declare data type
    wire i0, i1;
    wire eq;
```

Este formato no será utilizado, sin embargo consideramos apropiado mostrar cómo se realizaba esta construcción por si encuentra material que utilice la sintaxis de Verilog-1995.

5.2. Cuerpo del programa

A diferencia de un programa en lenguaje C, donde las declaraciones se ejecutan secuencialmente, el cuerpo del programa de un módulo Verilog sintetizable se puede considerar como una colección de partes de un circuito. Estas partes se operan en paralelo y se ejecutan de forma simultánea. Hay varias maneras de describir una parte:

- Asignación continua
- Bloque “always”
- Instanciación de módulos

La primera forma de describir una parte del circuito es usando una *asignación continua*. Es útil para circuitos combinacionales simples. Su sintaxis es

```
assign [signal_name] = [expression];
```

Cada asignación continua puede pensarse como una parte del circuito. La señal del lado izquierdo es la salida y las señales utilizadas en la expresión del lado derecho son las entradas. La expresión describe la función de este circuito. Por ejemplo, considere la declaración

```
assign eq = p0 | p1;
```

Es un circuito que realiza la operación `or`. Cuando `p0` o `p1` cambia su valor, esta declaración se activa y se evalúa la expresión. El nuevo valor se asigna a `eq` después del retardo de propagación. Hay tres asignaciones continuas en el Listado 1 y corresponden a las tres partes del circuito que se muestran en la Figura 1. Dado que las asignaciones corresponden a las partes del circuito, el orden de estas sentencias no importa.

La segunda forma de describir una parte del circuito es usando un bloque *always*. Se utilizan asignaciones de procedimientos más abstractas dentro del bloque *always* y, por lo tanto, se pueden utilizar para describir operaciones de circuitos más complejas. El bloque *always* se analiza en la Sección 3.3.

La tercera forma de describir una parte de un circuito es mediante la creación de instancias de módulos. La instanciación crea una instancia de otro módulo y nos permite incorporar módulos prediseñados como subsistemas del módulo actual. La creación de instancias se analiza en la Sección 5.5.

5.3. Declaración de una señal

Esta parte especifica las señales internas y los parámetros utilizados en el módulo. Las señales internas se pueden considerar como los cables de interconexión entre las partes del circuito, como se muestra en la Figura 1

El Listado 1 declara dos señales internas

```
wire p0, p1;
```

Red implícita

En Verilog, no es necesario declarar explícitamente un identificador. Si se omite una declaración, se supone que es una red implícita. El tipo de datos predeterminado es cable. Podemos eliminar las declaraciones explícitas en el Listado 1 y el código simplificado se muestra en el Listado 2.

Listado 2: Implementación del comparador de 1-bit con redes implicitas

```
1  module eq1_implicit
2      (
3      input i0, i1,  // no data type declaration
4      output eq
5      );
6
7      // no internal signal declaration
8
9      // product terms must be placed in front
10     assign p0 = ~i0 & ~i1;  // implicit declaration
11     assign p1 = i0 & i1;  // implicit declaration
12     // sum of two product terms
13     assign eq = p0 | p1;
14
15 endmodule
```

Aunque el código es más compacto, puede introducir errores sutiles de identificadores mal escritos. Para mayor claridad y documentación, siempre debemos usar declaraciones explícitas.

5.4. Ejemplo

Podemos ampliar el comparador a entradas de 2 bits. Sean la entrada `a` y `b` y sea la salida *aeqb*. La señal *aeqb* se activa cuando ambos bits de *a* y *b* son iguales. El código se muestra en el Listado 3.

Listado 3: Implementación de un comparador de 2-bits

```
1  module eq2_sop
2      (
3      input wire[1:0] a, b,
4      output wire aeqb
5      );
6
7      // internal signal declaration
8      wire p0, p1, p2, p3;
9
10     // sum of product terms
11     assign aeqb = p0 | p1 | p2 | p3;
12
13     // product terms
14     assign p0 = (~a[1] & ~b[1]) & (~a[0] & ~b[0]);
15     assign p1 = (~a[1] & ~b[1]) & ( a[0] &  b[0]);
16     assign p2 = ( a[1] &  b[1]) & (~a[0] & ~b[0]);
17     assign p3 = ( a[1] &  b[1]) & ( a[0] &  b[0]);
18 endmodule
```

Los puertos *a* y *b* se declaran ahora como un arreglo de dos elementos. La derivación de la arquitectura del cuerpo es similar a la del comparador de 1-bit. Las señales $p0$, $p1$, $p2$ y $p3$ representan los resultados de los cuatro términos producto, y el resultado final *aeqb* es la expresión lógica en formato de suma de productos.

5.5. Descripción Estructural

Un sistema digital se compone frecuentemente de varios subsistemas más pequeños. Esto nos permite construir un sistema grande a partir de componentes más simples o prediseñados. Verilog proporciona un mecanismo conocido como instanciación de módulo (*module instantiation*) para realizar esta tarea. Este tipo de código se llama descripción estructural.

Una alternativa al diseño del comparador de 2-bits de la Sección 5.4 es utilizar el comparador de 1-bit previamente construidos como se muestra en la Figura 2, en el que se utilizan dos comparadores de 1 bit para comprobar los dos bits individuales y sus resultados se alimentan a una compuerta and. La señal

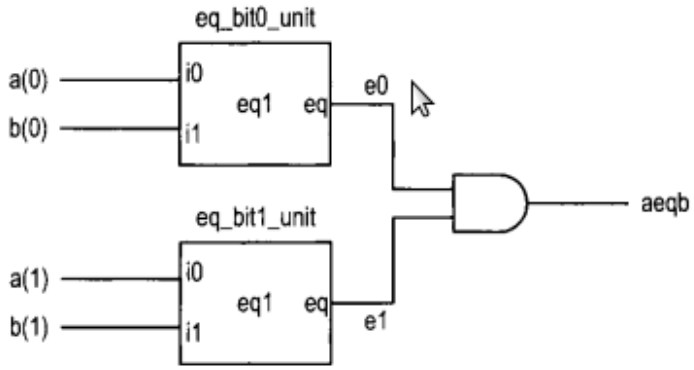

Figura 2: Representación gráfica del programa correspondiente al comparador de 2-bit.

aeqb se activa solo cuando ambos bits son iguales. El código correspondiente se muestra en el Listado 4.

Listado 4: Descripción estructural de un comparador de 2-bits

```
1  module eq2
2      (
3      input wire[1:0] a, b,
4      output wire aeqb
5      );
6
7      // internal signal declaration
8      wire e0, e1;
9
10     // body
11     // instantiation two 1-bit comparators
12     eq1 eq_bit0_unit (.i0(a[0]), .i1(b[0]), .eq(e0));
13     eq1 eq_bit1_unit (.eq(e1), .i0(a[1]), .i1(b[1]));
14
15     // a and b are equal if individual bits are equal
16     assign aeqb = e0 & e1;
17 endmodule
```

El código incluye dos sentencias de creación de instancias de módulos. La sintaxis simplificada del módulo instanciación es

```
[module_name] [instance_name]
(
 .[port_name]([signal_name]),
 .[port_name]([signal_name]),
 ...
 );
```

La primera parte del código especifica qué componente se utiliza. El término [module_name] indica el nombre del módulo y el término [instance_name] da una identificación única para una instancia. La segunda parte del código es la conexión de puertos, la cual indica las conexiones entre los puertos entradas/salidas de un módulo instanciado (módulo de nivel inferior) y las señalas externas usadas en el módulo actual (módulo de nivel superior). Esta forma de mapeo es conocida como conexión por nombre. El orden de los pares nombre-puerto y nombre-señal no tiene importancia.

En el Listado 4, la declaración de instanciación del primer componente es

```
eq1 eq_bit0_unit (.i0(a[0]), .i1(b[0]), .eq(e0));
```

El módulo eq1 es el definido en el Listado 1. La asignación de puertos refleja las conexiones que se muestran en la Figura 2. La declaración de instancias de componentes representa un circuito que tipo çaja negraçuya función se define en otro módulo.

Este ejemplo demuestra la estrecha relación entre un diagrama de bloques y el código. El código es esencialmente una descripción textual de un esquema.

Un esquema alternativo para asociar los puertos y las señales externas es la conexión por lista ordenada (a veces también conocida como conexión por posición). En este esquema, los nombres de los puertos del módulo de nivel inferior se omiten y las señales del módulo de nivel superior se enumeran en el mismo orden que en la declaración de puertos del módulo de nivel inferior. Con este esquema, las dos declaraciones de instancias de módulos del Listado 4 se pueden reescribir como

```
eq1 eq_bit0_unit (a[0], b[0], e0);
eq1 eq_bit1_unit (a[1], b[1], e1);
```

Aunque este esquema hace que el código sea más compacto, **es propenso a errores**, especialmente para un módulo con muchos puertos de entrada/salida. Por ejemplo, si modificamos el código del módulo de nivel inferior y cambia el orden de dos puertos en la declaración del puerto, todos los módulos instanciados deben ser corregido también. Si esto se hace accidentalmente durante la edición del código, el orden de puerto alterado puede dejarse sin detectar durante la síntesis y conduce a errores difíciles de encontrar. Siempre usaremos el esquema de conexión por nombre.

Verilog incluye un conjunto de primitivas predefinidos que se pueden instanciar como módulos. Estas primitivas corresponden a bloques de funciones simples a nivel de compuertas, como compuertas and, or y not. Por ejemplo, el circuito eql se puede implementar usando compuertas simples, como se muestra en la Figura 3. El código basado en primitivas se muestra en el Listado 5.

Listado 5: Implementación con primitivas de Verilog

```
1 module eq1_primitive
2     (
3     input wire i0, i1,
```

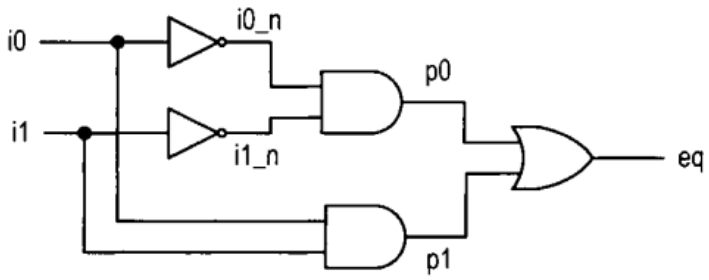

Figura 3: Diagrama de bajo nivel de un comparador de 1-bit.

```
    output wire eq
    );

    // internal signal declaration
    wire i0_n, i1_n, p0, p1;

    // primitive gate instantiations
    not unit1 (i0_n, i0);    // i0_n = not(i0);
    not unit2 (i1_n, 11);    // i1_n = not(i1);
    and unit3 (p0, i0_n, i1_n); // p0 = i0_n and i1_n;
    and unit4 (p1, i0, i1);     // p1 = i0 and i1;
    or unit5 (eq, p0, p1);      // eq = p0 or p1;
endmodule
```

Esta forma de código es muy tediosa y se puede reemplazar fácilmente con operaciones lógicas simples bit a bit. No usaremos las primitivas en este curso.

Además de las primitivas predefinidas, también podemos definir primitivas personalizadas, conocidas como primitivas definidos por el usuario (UDPs). Por ejemplo, podemos definir un circuito comparador de 1-bits en un UDP, como se muestra en el Listado 6.

Listado 6: Implementación con primitivas de Verilog

```
primitive eq1_udp(eq, i0, i1);
    output eq;
    input i0, i1;

    table
    // i0  i1 : eq
        0   0 : 1;
        0   1 : 0;
        1   0 : 0;
        1   1 : 1;
    endtable
endprimitive
```

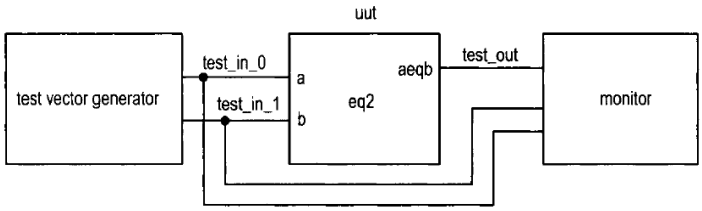

Figura 4: Testbench para el comparador de 2-bits

Un UPD es esencialmente una descripción de un circuito basada en tablas. La misma tabla también puede ser descrita por una sentencia *case*.

6. Testbench

Una vez que se desarrolla el código, se puede simular en una computadora para verificar la correcta operación del circuito y se puede sintetizar en un dispositivo físico. La simulación generalmente se realiza dentro del mismo entorno de trabajo de HDL.Para ello, creamos un programa especial conocido como *testbench*, el cual imitar un banco de pruebas de laboratorio físico. El esquema de un banco de pruebas de un comparador de 2 bits se muestra en la Figura 4. El bloque `uut` es la unidad bajo prueba (del inglés *unit under test*), el bloque **test vector generator** genera patrones de entrada de prueba, y el bloque **monitor** examina las respuestas de salida. En el Listado 7 se muestra un banco de pruebas simple para el comparador de 2 bits.

Listado 7: Testbench para un comparador de 2-bits

```
1  // The `timescale directive specifies that
2  // the simulation time unit is 1 ns and
3  // the simulation time step is 10 ps
4  `timescale 1 ns/10 ps
5
6  module eq2_testbench;
7      // signal declaration
8      reg [1:0] test_in0, test_in1;
9      wire test_out;
10
11     // instantiate the circuit under test
12     eq2_uut
13         (.a(test_in0), .b(test-in1), .aeqb(test_out));
14
15     // test vector generator
16     initial
17     begin
18         // test vector 1
```

```
19          test_in0 = 2'b00;
20          test_in1 = 2'b00;
21          # 200;
22          // test vector 2
23          test_in0 = 2'b01;
24          test_in1 = 2'b00;
25          # 200;
26          // test vector 3
27          test_in0 = Z'b01;
28          test_in1 = 2'b11;
29          # 200;
30          // test vector 4
31          test_in0 = Z'b10;
32          test_in1 = 2'b10;
33          # 200;
34          // test vector 5
35          test_in0 = 2'b10;
36          test_in1 = 2'b00;
37          # 200;
38          // test vector 6
39          test_in0 = 2'b11;
40          test_in1 = 2'b11;
41          # 200;
42          // test vector 7
43          test_in0 = 2'b11;
44          test_in1 = 2'b01;
45          # 200;
46          // stop simulation
47          $stop;
48      end
49  endmodule
```

El código consta de una instrucción de instanciación de módulo, que crea una instancia del comparador de 2 bits, y un bloque `initial`, que genera una secuencia de patrones de prueba. El bloque `initial` es una construcción Verilog especial, que se ejecuta una vez que comienza la simulación. Las declaraciones dentro de un bloque `initial` se ejecutan secuencialmente. Cada patrón de prueba es generado por tres declaraciones, como en

```
// test vector 2
test_in0 = 2'b01;
test_in1 = 2'b00;
#200
```

Las dos primeras declaraciones especifican los valores para las señales `test_in0` y `test_in1` y la tercera indica que los dos valores durarán 200 unidades de tiempo. La última instrucción, `$stop`, es una función del sistema Verilog que detiene la simulación y devuelve el control al software de simulación.

El código no tiene monitor. Podemos observar las formas de onda de entrada y salida en la pantalla de un simulador, que puede tratarse como un "analizador

lógico virtual".

Escribir código para un generador de vectores de prueba integral y un monitor requiere un conocimiento detallado de Verilog. Por ahora, esta lista puede servir como plantilla de banco de pruebas para otros circuitos combinacionales. Podemos sustituir la instancia `uut` y modificar los patrones de prueba de acuerdo con el nuevo circuito.

Tipo de operación	**Símbolo del operador**	**Descripción**	**Número de operandos**
Aritmético	+	suma	2
	-	resta	2
	*	multiplicación	2
	/	división	2
	%	módulo	2
	**	exponenciación	2
Desplazamiento	>>	desplazamiento lógico a la derecha	2
	<<	desplazamiento lógico a la izquierda	2
	>>>	desplazamiento aritmético a la derecha	2
	<<<	desplazamiento aritmético a la izquierda	2
Comparación	>	mayor que	2
	<	menor que	2
	>=	mayor o igual	2
	<=	menor o igual	2
Igualdad (*)	==	igualdad	2
	!=	desigualdad	2
	===	case equality	2
	!==	case inequality	2
Bit a bit	~	negación de 1 bit	1
	&	and de 1 bit	2
	|	or de 1 bit	2
	^	xor de 1 bit	2
Reducción	&	and de reducción	1
	|	or de reducción	1
	^	xor de reducción	1
Lógicos	!	negación lógica	1
	&&	and lógico	2
	||	or lógico	2
Concatenación	{ }	concatenación	cualquiera
	{ { } }	replicación	cualquiera
Condicional	? :	condicional	3

(*) == y != comparan igualdad/desigualdad lógica (comparan 1 y 0, otro estado da como resultado x
(*) === y !== comparan igualdad/desigualdad con lógica de 4 estados (comparan 1, 0, z y x)

Tabla 3: Operadores de Verilog

Operador	**Precedencia**
`! ~ + -` (unarios)	más alta
`**`	
`* / %`	
`+ -` (binarios)	
`>> << >>> <<<`	
`< <= > >=`	
`== != === !==`	
`&`	
`^`	
`|`	
`&&`	
`||`	
`? :`	la más baja

Tabla 4: Precedencia de los operadores