Capítulo 8

1. Reseña histórica de los lenguajes de descripción de hardware.
2. Introducción a Verilog HDL
    1. Módulos
    2. Simulación y Síntesis
3. Convenciones de Léxico en Verilog
4. Tipos de Datos en Verilog
5. Modelado por Flujo de Datos: RTL
6. Modelado Estructural
    1. Descripción por Modelado Estructural
    2. Diseño Jerárquico
    3. Formas de descripción en Modelado Estructural
    4. Instanciación por descripción posicional: el sumador completo
    5. Instanciación por descripción nombrada: el sumador completo
    6. Descripción estructural del multiplexor
7. Diseño secuencial en Verilog
    a. Sentencia Always y If-Then-Else
    b. Asignaciones de bloqueo y de no bloqueo
    c. Flip-Flops, Registros y Latches
    d. Sentencia Case
8. Descripción de Máquinas de Estado Sincrónicas
9. System tasks, system function
10. Testbench en Verilog
11. Herramientas de compilación y simulación
    a. Compilador iverilog
    b. Simulación con resultados en terminal: vvp runtime engine
    c. Archivos de resultados: system tasks dumpfile y dumpvars
    d. Visor de formas de onda gtkwave

Reseña:

Un lenguaje de descripción de hardware (HDL) es un lenguaje de programación especializado que se utiliza para describir la estructura y el comportamiento de los circuitos electrónicos y, más comúnmente, los circuitos lógicos digitales.

Un lenguaje de descripción de hardware permite una descripción formal y precisa de un circuito electrónico que permite el análisis y la simulación automatizados de un circuito electrónico. También permite la síntesis de una descripción HDL en una lista de conexiones (una especificación de los componentes electrónicos físicos y cómo están conectados entre sí), que luego se pueden colocar y enrutar para producir el conjunto de máscaras que se utilizan para crear un circuito integrado.

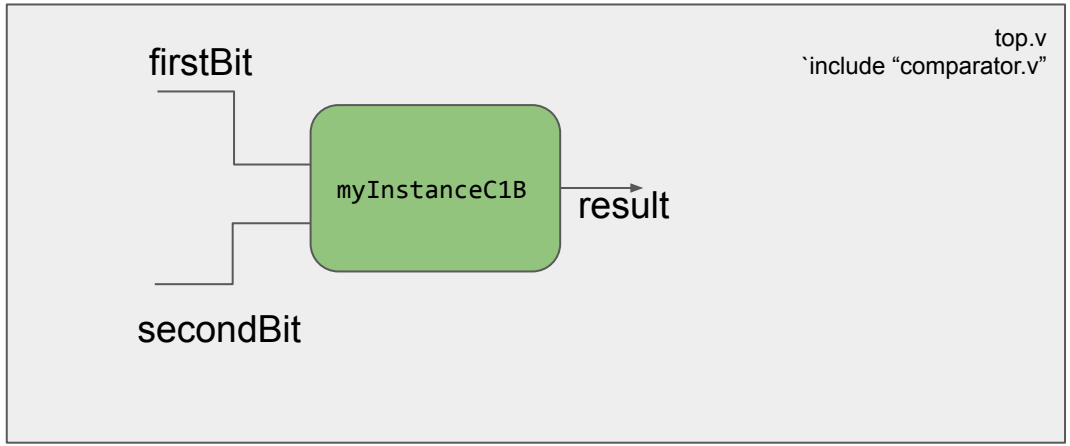
Los primeros lenguajes de descripción de hardware aparecieron a finales de la década de 1960, pareciendo lenguajes más tradicionales. El primero que tuvo un efecto duradero fue descrito en 1971 en el texto Computer Structures de C. Gordon Bell y Allen Newell. Este texto introdujo el concepto de nivel de transferencia de registro (RTL), utilizado por primera vez en el lenguaje ISP para describir el comportamiento del PDP-8 de Digital Equipment Corporation (DEC).

VHDL y Verilog (System Verilog) son ahora HDL dominantes en la industria electrónica, mientras que los HDL más antiguos y menos capaces desaparecieron gradualmente del uso. Sin embargo, VHDL y Verilog comparten muchas de las mismas limitaciones, como no ser adecuados para la simulación de circuitos analógicos o de señal mixta . Los HDL especializados (como Confluence) se introdujeron con el objetivo explícito de corregir limitaciones específicas de Verilog y VHDL, aunque ninguno tuvo la intención de reemplazarlos.

A lo largo de los años, se ha invertido mucho esfuerzo en mejorar las HDL. La última iteración de Verilog, formalmente conocida como IEEE 1800-2005 SystemVerilog, presenta muchas características nuevas (clases, variables aleatorias y propiedades) para abordar la creciente necesidad de una mejor aleatorización, jerarquía de diseño y reutilización del banco de pruebas.

```
module comparator1Bit(
    input wire x, y,
    output wire eq
);

  wire s0, s1;

  assign s0 = ~x & ~y;
  assign s1 = x & y;
  assign eq = s0 | s1;

endmodule
```

```
integer cuenta;
wire firstBit;
wire secondBit;
wire result;

comparator1Bit myInstanceC1B(
  .x(firstBit),
  .y(secondBit),
  .eq(result)
)
```



Reglas léxicas en verilog:
Verilog es un lenguaje que distingue entre mayúsculas y minúsculas, es decir, las letras mayúsculas y minúsculas tienen significados diferentes. Además, Verilog es un lenguaje de formato libre (es decir, los espacios se pueden agregar libremente), pero se recomienda no abusar de esta práctica para escribir los códigos, ya que es claro y legible. Por último, en Verilog, '//' se usa para comentarios; Además, los comentarios de varias líneas se pueden escribir entre / * y * /.

Tipos de datos:
Existen dos tipos principales, las "net" que son las utilizadas para *alambrar* el diseño, además existen las "variables" que representa el almacenamiento de valores en el diseño.

Los *wires* son un tipo de net, los *integer* y los *reg* un tipo de variable.

Valores lógicos:

Valor lógico	Descripción
0	Valor lógico '0' o condición falsa
1	Valor lógico '1' o condición verdadera
z	Estado de alta impedancia (usado para buffers triestado)
x	Condición "No importa" o "Valor desconocido"

Representación de números en Verilog

Binario:

```
reg [1:0] a = 2'b01; //  number =  1; size = 2 bit;
reg [2:0] a = -3'b1; // unsigned number= -1 (in 2's complement form); size = 3 bit;
```

Decimal:

```
reg [3:0] a = 3'd1;  // number = 1; size =3 bit;
reg [3:0] a = -3'd1; // unsigned number = -1 (in 2's complement form); size =3 bit;
reg [3:0] a = 1; // unsigned number = 1; size = 4 bit;
reg [3:0] a = -1; // unsigned number = -1; size = 4 bit in 2's complement form;
```

Con signo:

```
integer a = 1; // signed number = 1; size = 32 bit;
integer a = -1; // signed number = -1; size = 32 bit in 2's complement form;
// Para representaciones hexadecimales y octales, se utiliza "h" y "o" en lugar de
"b" en formato binario.
```

Números con signo:

Por defecto, los tipos de datos 'reg' y 'wire' son números sin signo, mientras que 'integer' es un número con signo. Los números con signo se puede definir para 'reg' y 'wire' usando palabras clave 'signed', es decir, 'reg signed' y 'wire signed' respectivamente, como se muestra en la tabla de la derecha.

Número	Valor
reg [1:0] a = 2'b01;	01
reg [7:0] a = 2'b0001_1111;	00011111
reg [2:0] a = -3'b1;	111
reg [3:0] a = 4'd1;	0001
reg [3:0] a = -4'd1;	1111
reg [5:0] a = 6'o12;	001_010
reg [5:0] b = 6'h1f;	0001_1111
reg [3:0] a = 1;	0001
reg [3:0] a = -1;	1111
reg signed [3:0] a = 1;	0001
reg signed [3:0] a = -1;	1111
integer a = 1;	0000_0000_..._0001
integer a = -1;	1111_1111_..._1111
reg [4:0] a = 5'bx	xxxxx
reg [4:0] a = 5'bz	zzzzz
reg [4:0] a = 5'bx01	xxx01

Operadores en Verilog

Tipo	Símbolo	Nombre
Arithmetic	+	add
	-	subtract
	*	multiply
	/	divide
	%	modulus (remainder)
	**	power
Bitwise	~	not
	|	or
	&	and
	^	xor
	~& or &~	nand
Relational	>	greater than
	<	less than
	>=	greater than or equal
	<=	less than or equal
	==	equal
	!=	not equal
Logical	!	negation
	||	logical OR
	&&	logical AND
Shift operator	>>	right shift
	<<	left shift
	>>>	right shift with MSB shifted to right
	<<<	same as <<
Concatenation	{ }	Concatenation
	{ { } }	Replication
Conditional	? :	conditional
Sign-format	$unsigned()	signed to unsigned conversion
	$signed()	unsigned to signed conversion

Operadores de concatenación y replicación:
La operación de concatenación '{}' se usa para combinar arreglos más pequeños para crear un arreglo grande como se muestra a continuación:

```
wire[1:0] a = 2b'01;
wire[2:0] b = 3b'001;
wire[3:0] c ;
  assign c = {a, b} // c = 01001 is created
  using a and b;
```

El operador de replicación se usa para repetir ciertos bits como se muestra a continuación,

```
  assign c = { 2{a}, 1'b0 } // c = 01010
  i.e. a is repeated two times i.e. 01-01
```

Operadores de desplazamiento:
Si a = 1011_0011,
a >> 3 = 0001_0110, Es decir, desplaza 3 bits a la derecha y **llena el MSB con ceros**.
a << 3 = 1001_1000, Es decir, desplaza 3 bits a la izquierda y **llena el LSB con ceros**.
a >>> 3 = 1111_0110, Es decir, desplaza 3 bits a la derecha y **complete el MSB con el bit de signo**, es decir, el MSB original.
a <<< 3 = 1111-0110 Es decir, lo mismo que << 3.

Parámetros: Los parámetros son utilizados para definir valores en un diseño que eventualmente podría ser modificado y solamente editando el valor del parámetro, la funcionalidad seguirá intacta.

localparam: La palabra clave 'localparam' se usa para definir las constantes en verilog.

```
module constantEx(
    input wire [3:0] a, b,
    output wire [3:0] z
);

localparam N = 3, M = 2; //localparam

wire [N:0] x;
wire [2**N:0] y;

// use x and y here
assign z = a & b;

endmodule
```

parameter y defparam: Podemos definir el parámetro en el módulo, que se puede modificar durante la instanciación del componente.

```
module parameterEx
#(parameter N = 2, M = 3  //parameter
)(
    input wire [N-1:0] a, b,
    output reg [N-1:0] z
);
always @(a,b)
begin
    if (a==b) z = 1;
    else z = 0;
end
  endmodule
-------------------------------------------------------------
// parameterInstantEx.v
module parameterInstantEx(
    input wire [4:0] a, b,
    output wire [4:0] z);
parameterEx #(.N(5)) compare4bit ( .a(a), .b(b), .z(z));
parameterEx compare4bit_2 ( .a(a), .b(b), .z(z));
defparam compare4bit_2.N = 5;//'defparam' to set the value of parameter
endmodule
```

Asignaciones de procedimiento: Son las maneras utilizadas para que el valor de un dato sea modificado.

Bloque "always": Todas las declaraciones dentro del bloque always se ejecutan secuencialmente. Además, si el módulo contiene más de un bloque always, entonces todos los bloques always se ejecutan en paralelo, es decir, los bloques always son los bloques concurrentes.

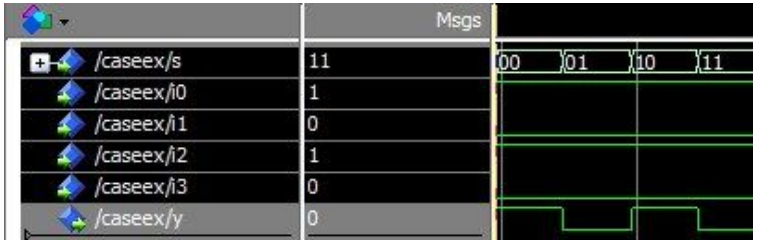
Asignaciones bloqueantes y no bloqueantes: Hay dos tipos de asignaciones que se pueden usar dentro del bloque always, asignaciones de bloqueo y no bloqueo. El signo "=" se utiliza para asignaciones bloqueantes; mientras que "<=" se usa para asignaciones no bloqueantes.

```
// blockAssignment.v
module blockAssignment(
    input wire x, y,
    output reg z, w
);
always @(x,y)
  begin
    z = x;
    w = z & y;
  end
endmodule
```

```
// nonblockAssignment.v
module nonblockAssignment(
    input wire x, y,
    output reg z, w
);
always @(x,y)
  begin
    z <= x;
    w <= z & y;
  end
endmodule
```

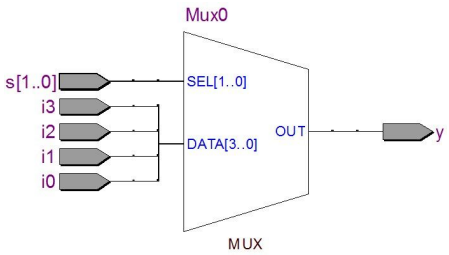
Declaración if-else

```
module Mux(
    input wire[1:0] s,
    input wire i0, i1, i2, i3,
    output reg y
);

always @(s)
  begin
    if (s==2'b00)
        begin //begin-end is required for more than one statements
            y = i0;
            // more statements
        end
    else if (s==2'b01) y = i1;
    else if (s==2'b10) y = i2;
    else if (s==2'b11) y = i3;
    else y = y; // no change
  end
endmodule
```



Declaración case

```
module caseEx(
    input wire[1:0] s,
    input wire i0, i1, i2, i3,
    output reg y
);

always @(s)
  begin
    case (s)
        0 : y = i0;
        2'b01 : y = i1;
        2 : y = i2;
        3 : y = i3;
        default : y = 1'bx;
    endcase
  end
endmodule
```



FF-D

```
module BasicDFF(
    input wire clk, reset,
    input wire d,
    output reg q
);

always @(posedge clk, posedge reset)
  begin
    if (reset ==  1) q = 0;
    else q = d;
  end
endmodule
```

FF-D con Enable

```
module BasicDFF_enable(
    input wire clk, reset, en,
    input wire d,
    output reg q
);

always @(posedge clk, posedge reset)
  begin
    if (reset ==  1)  q <= 0;
    else if (en == 1) q <= d;
  end
endmodule
```

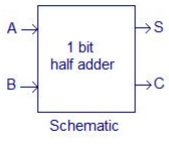
Banco de pruebas: Dado que los bancos de pruebas se utilizan solo con fines de simulación (no para síntesis), por lo tanto, se puede utilizar una gama completa de construcciones de Verilog, por ejemplo, las palabras clave 'para', 'mostrar' y 'monitor', etc., se pueden utilizar para escribir bancos de pruebas.

Circuitos combinacionales:

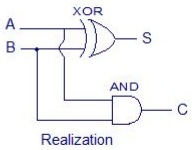
```
module half_adder
(
    input wire a, b,
    output wire sum, carry
);

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

endmodule
```

Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Truth table



Schematic



Realization

```
// half_adder_tb.v
`timescale 1 ns/10 ps  // time-unit = 1 ns, precision = 10 ps
module half_adder_tb;
    reg a, b;
    wire sum, carry;
    // duration for each bit = 20 * timescale = 20 * 1 ns  = 20ns
    localparam period = 20;

    half_adder UUT (.a(a), .b(b), .sum(sum), .carry(carry));

    initial // initial block executes only once
        begin
            $dumpfile ("half_adder_tb.vcd");
            $dumpvars(1, half_adder_tb);
            // values for a and b
            a = 0;
            b = 0;
            #period; // wait for period
            a = 0;
            b = 1;
            #period;
            a = 1;
            b = 0;
            #period;
            a = 1;
            b = 1;
            #period;
        end
endmodule
```

```
// half_adder_procedural_tb.v
`timescale 1 ns/10 ps  // time-unit = 1 ns, precision = 10 ps
module half_adder_procedural_tb;
    reg a, b;
    wire sum, carry;
    // duration for each bit = 20 * timescale = 20 * 1 ns  = 20ns
    localparam period = 20;

    half_adder UUT (.a(a), .b(b), .sum(sum), .carry(carry));

reg clk;
// note that sensitive list is omitted in always block
// therefore always-block run forever
always
begin
    clk = 1'b1;
    #20; // high for 20 * timescale = 20 ns

    clk = 1'b0;
    #20; // low for 20 * timescale = 20 ns
end

initial begin
    $dumpfile("half_adder_procedural_tb.vcd"); // Change filename as
appropriate.
    $dumpvars(1, half_adder_procedural_tb);
end
```

```
always @(posedge clk)
begin
    // values for a and b
    a = 0;
    b = 0;
    #period; // wait for period
    // display message if output not matched
    if(sum != 0 || carry != 0)
        $display("test failed for input combination 00");
    a = 0;
    b = 1;
    #period; // wait for period
    if(sum != 1 || carry != 0)
        $display("test failed for input combination 01");
    a = 1;
    b = 0;
    #period; // wait for period
    if(sum != 1 || carry != 0)
        $display("test failed for input combination 10");
    a = 1;
    b = 1;
    #period; // wait for period
    if(sum != 0 || carry != 1)
        $display("test failed for input combination 11");
    a = 0;
    b = 1;
    #period; // wait for period
    if(sum != 1 || carry != 1)
        $display("test failed for input combination 01");
    $stop;   // end of simulation
end
  endmodule
```