Fundación FULGOR

Diseño Digital Avanzado

Unidad 1 - Introducción a Verilog como HDL

Dr. Ariel L. Pola
apola@fundacionfulgor.org.ar
August 21, 2022

Tabla de Contenidos

Contenidos Temáticos

Contenidos Temáticos

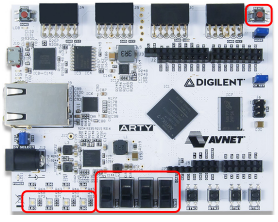
Unidad 1
Introducción a Verilog como lenguaje de diseño de Hardware

- Introducción.
- Historia.
- Síntesis Lógica.
- Módulos.
- Partición del diseño y diseño jerárquico.
- Valores lógicos y tipos de datos.
- Los cuatro niveles de abstracción (switch, gate, dataflow y behavioral).
- Tareas y funciones.
- Aritmética signada.
- Verificación en diseño de hardware.
- Cobertura de código.

Introducción

Diseñar Mientras Aprendemos

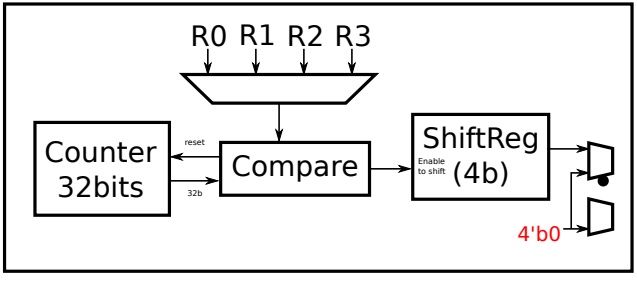
- El objetivo es aprender el lenguaje de descripción de hardware Verilog mientras diseñamos un módulo que nos permita controlar los leds de la FPGA.
- Los leds RGB se prenderán uno a la vez durante un tiempo definido por unas constantes.
- Después de ese tiempo se prende el siguiente led y se apaga el anterior.
- Dos switches permiten la selección del tiempo de encendido, un switch permite habilitar el comportamiento y el último switch cambia a otro color los leds.
- Se utiliza el reset del sistema para inicializar todas las variables.



Cómo es el esquemático?.Qué módulos necesitamos?

Diagrama en Bloques

- Contador de 32b
- Comparador
- Multiplexores
- Shift register



Historia

- Verilog fue inventado por Phil Moorby en 1985 mientras trabajaba en *Automated Integrated Design Systems*, más tarde renombrada *Gateway Design Automation*.
- El objetivo de Verilog era ser un lenguaje de modelado de hardware.
- VHDL es otro lenguaje que se utiliza para el diseño de hardware.
- Con la llegada de herramientas de síntesis por parte de *Synopsys* en 1987, Verilog y VHDL empezaron a cambiar todo el paradigma y el espectro de la metodología de diseño de hardware.
- En pocos años, los lenguajes HDL se convirtieron en la elección favorita para el diseño de hardware.

Historia

- *Gateway Design Automation* fue comprada por *Cadence Design Systems* en 1990, quien ahora tiene todos los derechos sobre los simuladores lógicos de Verilog y Verilog-XL hechos por *Gateway*.
- En 1995, con el incremento en el éxito de VHDL, *Cadence* decidió hacer el lenguaje abierto y disponible para estandarización y transfirió Verilog al dominio público a través de *Open Verilog International*, actualmente conocida como *Accellera*.
- La norma Verilog está todavía en evolución.
- Se están agregando más y más características y sintaxis que, por un lado, proporcionan un mayor nivel de abstracción, y por otro lado ayudan al diseñador a verificar eficientemente un diseño RTL.
- Las últimas normas son Verilog 2005 y SystemVerilog 2005, la última adición de más características para el modelado y verificación.

Qué es Verilog?

- Verilog es un lenguaje de descripción de hardware con similitudes a C, pero **no es un lenguaje de programación de software**.
- Cada línea de código Verilog significa que uno o más componentes de hardware se incluyen en el diseño.
- Es un lenguaje muy rico que permite varios niveles de abstracción, como el diseño, modelado, simulación e implementación.
- No todo el lenguaje es sintetizable por una herramienta de síntesis.
- La herramienta de síntesis traduce Verilog en un diseño a nivel de compuertas.
- La herramienta de síntesis comprende sólo un subconjunto de Verilog llamado **RTL**.
- El nivel de transferencia de registro *(Register Transfer Level - RTL)* significa la colocación de registros en el diseño y el flujo de datos entre los registros.
- Los avances en la tecnología están permitiendo a los diseñadores desarrollar diseños más complejos, lo que representa un auténtico desafío para los ingenieros de prueba y verificación.
- La prueba de un diseño complejo requiere creatividad e ingenio.

Usando Verilog HDL

Síntesis Lógica

- El código escrito en RTL se sintetiza para la implementación de nivel de compuertas.
- El proceso de síntesis toma el RTL y lo traduce en un *netlist* de nivel de compuerta optimizado.
- Para la síntesis lógica, el usuario especifica restricciones de diseño y la tecnología de destino en forma de una biblioteca de celdas estándar *(standar cells)*.
- La biblioteca contiene compuertas lógicas básicas estándar, como AND y OR, o macro-celdas como sumadores, multiplicadores, flip-flops, multiplexores, etc.
- La herramienta convierte completamente el diseño descrito en el lenguaje de descripción de hardware RTL en un diseño que contiene celdas estándar.
- Para mapear de forma óptima la descripción de alto nivel en hardware real, la herramienta realiza varios pasos.
  - Convierte primero la descripción RTL en lógica booleana no optimizada.
  - Realiza varias transformaciones para optimizar la lógica sujeto a restricciones de usuario (optimización independiente de la tecnología de destino).
  - La herramienta mapea la lógica optimizada a celdas estándar específicas de la tecnología.

Módulos

- Un código Verilog tiene un módulo de nivel superior, que puede instanciar muchos otros módulos.
- El módulo es el componente básico de Verilog.
- Cada módulo contiene instrucciones e instanciación de módulos de nivel inferior.
- En el diseño de RTL este módulo, una vez sintetizado, infiere la lógica digital.
- El diseñador diseña un diseño de hardware como módulos jerárquicamente interconectados de nivel inferior que forman módulos de nivel superior.
- En el siguiente nivel de jerarquía, los módulos así construidos están conectados además para diseñar incluso módulos de alto nivel (múltiples capas de módulos).
- En el nivel superior, el diseñador también puede concebir la funcionalidad de una aplicación en términos de módulos interconectados.
- Los módulos individuales también pueden ser sintetizados de forma incremental para facilitar la síntesis de diseños grandes.

Usando Verilog HDL

Módulos

- Los módulos son declarados e instanciados como clases en C ++, pero las declaraciones de módulo no se pueden anidar.
- Las instancias de módulos de bajo nivel están interconectadas y los módulos tienen puertos para estas interconexiones.

Ejemplo

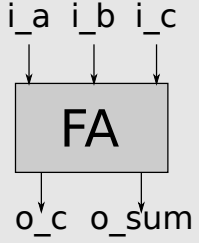
- Modulo

```
1 module fulladder(<port declaration>);
2 ...
3 endmodule
```

- Descripción

```
1 module fulladder(
2     output o_sum,
3     output o_c,
4     input  i_a,
5     input  i_b,
6     input  i_c);
7     assign {o_c,o_sum}=i_a+i_b+i_c;
8 endmodule
```

- Esquema



Usando Verilog HDL

Puertos

- Los puertos se utilizan para interconectar los módulos instanciados.
- **Verilog 95** permite la declaración de los puestos en la definición del módulo y luego se listan en cualquier orden.
- **Verilog 2001** incorpora el listado de los puertos dentro de la definición del módulo *(ANSI style)*.

Declaración de Puertos

- Verilog 95

```
1  module fulladder(
2      o_sum, o_c, i_a, i_b, i_c);
3
4     output o_sum;
5     output o_c;
6     input  i_a;
7     input  i_b;
8     input  i_c;
9
10    assign {o_c,o_sum}=i_a+i_b+i_c;
11 endmodule
```

- Verilog 2001

```
1  module fulladder(
2     output o_sum,
3     output o_c,
4     input  i_a,
5     input  i_b,
6     input  i_c);
7     assign {o_c,o_sum}=i_a+i_b+i_c;
8  endmodule
```

Puertos

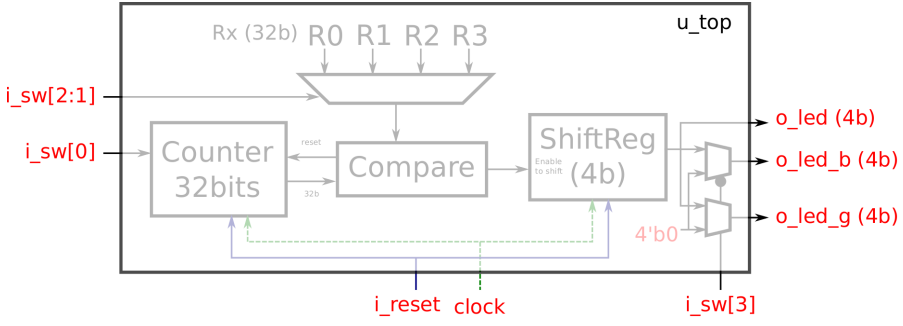
- El tamaño de los puertos se definen como [MSB:LSB] o [LSB:MSB].

Declaración de Puertos

```
1 module adder(
2     output [4:0] o_sum,
3     input  [3:0] i_a,
4     input  [3:0] i_b);
5     assign o_sum = i_a + i_b;
6 endmodule
```

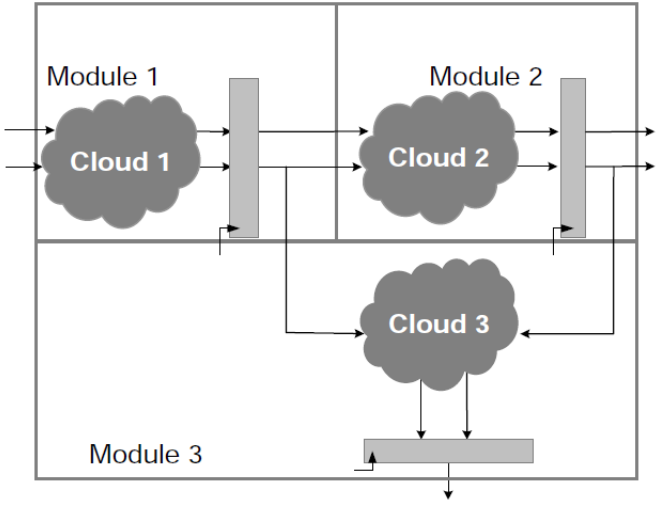
Módulos

- Qué puertos necesitamos?
- Escribir el módulo Top Level con sus puertos.



Lineamientos de Diseño RTL

- La partición de un diseño digital en una serie de módulos es importante.
- Un módulo no debe ser demasiado pequeño ni demasiado grande.
- Siempre que sea posible, **el diseño debe dividirse** de manera que los límites del módulo terminen en salidas de registros. Esto hará que sea más fácil sintetizar el módulo de nivel superior o síntesis jerárquica en cualquier nivel con limitaciones de tiempo.
- El diseñador también debe asegurarse de que **ninguna combinación de nubes cruce los límites del módulo**. Esto da a la herramienta de síntesis más facilidades para generar lógica optimizada.


Module 1
Module 2
Cloud 1
Cloud 2
Cloud 3
Module 3

Lineamientos para el Flujo de Diseño de Nivel de Sistema

- Un diseñador de sistemas primero captura los requisitos y especificaciones (R & S) del sistema de tiempo real bajo análisis.
- La implementación del algoritmo en software o hardware necesita realizar cálculos en los datos de entrada y producir datos de salida a las velocidades especificadas.
  - *Por ejemplo, en un sistema de comunicación digital, el requisito puede describirse en términos de velocidades de datos y el estándar de comunicación que modula estos datos para la transmisión.*
- El desarrollo de algoritmos es uno de los pasos más críticos en el diseño de sistemas.
- Los algoritmos se desarrollan utilizando herramientas como MATLAB, Simulink o C/C++, o en cualquier lenguaje de alto nivel.
- Funcionalmente, cumplir con R & S es una consideración importante cuando el diseñador selecciona un algoritmo de varias opciones.
- Aunque la satisfacción de los requisitos funcionales es la consideración principal, el desarrollador debe tener en cuenta la implementación final del algoritmo en una plataforma integrada que consiste en ASICs, FPGAs y DSPs.

Lineamientos para el Flujo de Diseño de Nivel de Sistema

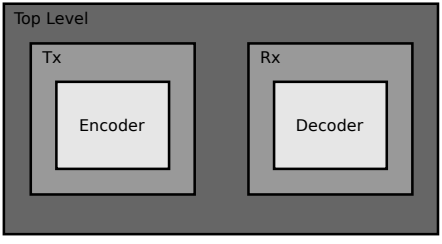
- **Para facilitar la partición del diseño en una plataforma híbrida incorporada, es importante que un diseñador de sistemas defina todos los componentes del diseño, especificando claramente el flujo de datos entre ellos.**
- El programa debe ser definido como sucederá en el sistema actual.
  - *Por ejemplo, en sistemas de procesamiento de señales en tiempo real, los datos se procesan bloque por bloque. En esta forma, se adquiere un buffer de datos de entrada y se pasa al primer componente del sistema. El componente procesa este buffer de datos y pasa la salida al componente siguiente en orden de ejecución.*
- Adherirse a estas directrices facilitará la tarea de partición en HW y SW, co-diseño y co-verificación.

Lineamientos para el Flujo de Diseño de Nivel de Sistema

- **Es importante que los diseñadores en las fases subsiguientes en el ciclo de diseño se adhieran a los mismos componentes y nombres de variables en la medida de lo posible.**
- Esto facilita enormemente ir y venir en el ciclo de diseño mientras el diseñador está realizando mejoras y verificando su funcionalidad y rendimiento.

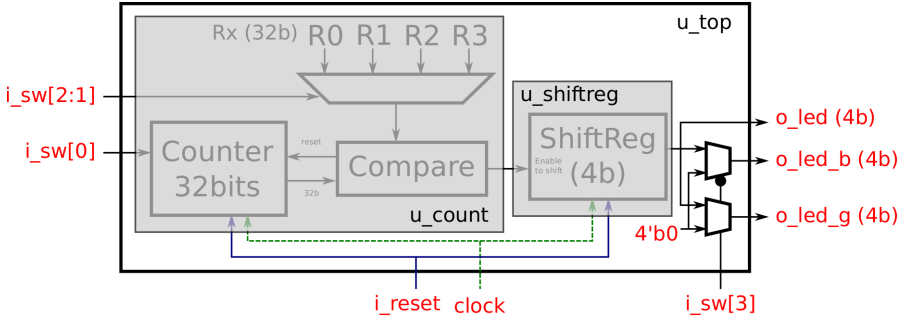
Jerarquías

- Verilog funciona de manera eficiente con un concepto de modelado jerárquico.
- El código Verilog contiene un módulo de nivel superior y puedo o no tener más módulos instanciados.
- El módulo de nivel superior *(top level)* no se instancia en ningún lugar.
- Pueden existir varias instancias de un módulo de nivel inferior.
- Verilog es un HDL y, a diferencia de otros lenguajes de programación, una vez sintetizado cada instanciación infiere una copia física del hardware con sus propias compuertas lógicas, registros y cables.



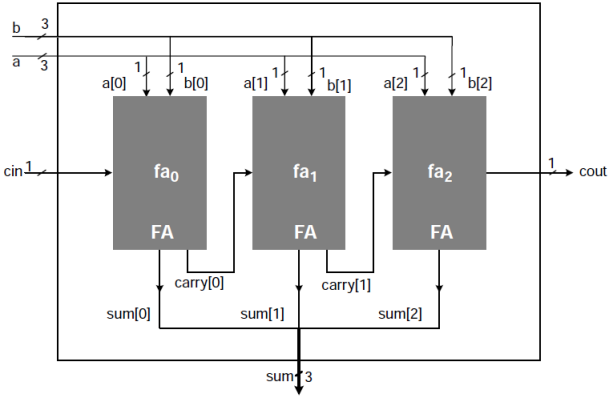
Jerarquías

- Qué jerarquías podemos encontrar?
- Qué módulos podemos agrupar? Por qué?



Instancias

- Instanciar significa incluir un módulo dentro de otro.
- La instancia consiste en:
  - Nombre del módulo **(fulladder)**
  - Nombre de la instancia **(u_fulladder)**
  - Declaración de puertos
- El orden en la declaración de los puertos es importante dependiendo el estilo que utilicemos.
  - Verilog 95 solo permite declarar los puertos según el orden que fueron definidos en la declaración del módulo.
  - Verilog 2001 incorpora la lista de puertos en la instancia sin tener en cuenta el orden que fueron definidos en la declaración del módulo.



Ejemplo de instancias de sumadores FA en un Sumador Ripple Carry (Ripple Carry Adder)

Instancias de FA en RCA

- Verilog 95

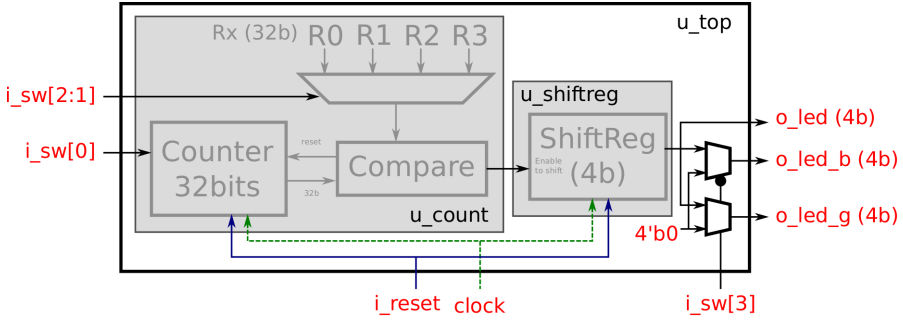
```
1  module rca(o_sum, o_c,
2             i_a, i_b, i_c);
3    output [2:0]          o_sum;
4    output                o_c;
5    input [2:0]           i_a, i_b;
6    input                 i_c;
7
8    wire [1:0]            carry;
9    // module instantiation
10   fulladder
11     u_fulladder_0(o_sum[0],
12                   carry[0],
13                   i_a[0], i_b[0],
14                   i_c);
15   fulladder
16     u_fulladder_1(o_sum[1],carry[1],
17                   i_a[1], i_b[1],
18                   carry[0]);
19   fulladder
20     u_fulladder_2(o_sum[2], o_c,
21                   i_a[2], i_b[2],
22                   carry[1]);
23 endmodule
```

- Verilog 2001

```
1  module rca(output [2:0] o_sum,
2             output       o_c,
3             input [2:0]  i_a, i_b,
4             input        i_c);
5    wire [1:0]            carry;
6    // module instantiation
7    fulladder
8      u_fulladder_0(.o_sum(o_sum[0]),
9                    .o_c(carry[0]),
10                   .i_a(i_a[0]),
11                   .i_b(i_b[0]),
12                   .i_c(i_c));
13   fulladder
14     u_fulladder_1(.o_sum(o_sum[1]),
15                   .o_c(carry[1]),
16                   .i_a(i_a[1]),
17                   .i_b(i_b[1]),
18                   .i_c(carry[0]));
19   fulladder
20     u_fuladder_2(.o_sum(o_sum[2]),
21                  .o_c(o_c),
22                  .i_a(i_a[2]),
23                  .i_b(i_b[2]),
24                  .i_c(carry[1]));
25 endmodule
```

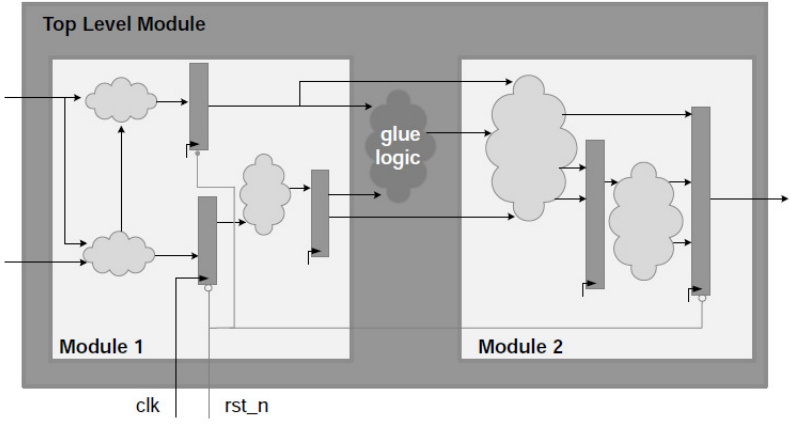
Instancias

- Instanciar los módulos del diseño.
- Solamente definir los puertos.



Lineamientos de Síntesis

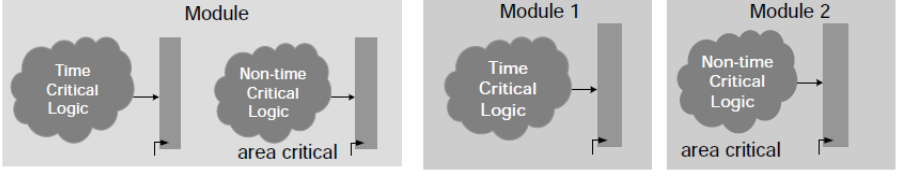
- **Lógica de encolado *(Glue Logic)***
  - Mientras que el diseñador está dividiendo el diseño de forma jerárquica en una serie de módulos, debe evitar la lógica de encolado que conecta dos módulos [**?**].
  - Esto puede suceder después de corregir un desajuste de interfaz o agregar alguna funcionalidad que falta al depurar el diseño.
  - Cualquier lógica de este tipo debería formar parte de la lógica combinacional de uno de los módulos constituyentes.
  - **La lógica de encolado puede causar problemas en síntesis ya que los módulos individuales pueden satisfacer restricciones de tiempo, mientras que el módulo de nivel superior puede que no.**



Ejemplo de lógica de encolado(Glue Logic)

Lineamientos de Síntesis

- **Módulos con objetivos comunes de diseño**
  - El diseñador debe evitar colocar en el mismo módulo una lógica crítica y no crítica en tiempo.
  - El módulo con lógica de tiempo crítico debe ser sintetizado para mejorar la sincronización, mientras que el módulo con lógica no crítica en el tiempo se optimiza para el mejor área.
  - Ponerlos en el mismo módulo producirá un diseño sub-óptimo.
  - La lógica debe ser dividida y colocada en dos módulos separados.



Descripción

- Un *bit* en Verilog puede tomar uno de los cuatro posibles valores mostrados en la tabla.
- Es importante recordar que no hay valores desconocidos en un circuito real, y **X** en simulación significa solamente que el simulador de Verilog no determina un valor en **0** o **1** de un puerto.

Posibles Valores de un Bit	
0	Cero, Lógico Bajo, Falso o Masa
1	Uno, Lógico Alto o Potencia
X	Desconocido
Z	Alta Impedancia, Desconectado o Puerto Tri-State

Nets

- Son conexiones físicas entre componentes.
- Tipos de Nets:
  - *wire, tri. wor, trior, wand, triand, tri0, tri1, supply0, supply1, trireg.*
- En RTL Verilog el tipo de dato mas utilizado es *wire*.
- Una variable *wire* representa uno o múltiples bits.
- Aplicación:
  - Conecta puertos entre instancias.
  - Salida de operaciones lógicas.
- En la síntesis, las variables tipo *wire* infieren físicamente un cable.

Asignación Continua

```
1 module operation
2   (output [4:0] o_a , input [1:0]  i_s1, input [1:0]  i_s2);
3    wire [3:0]   dot;
4    assign dot  = i_s1 * i_s2;
5    assign o_a  = dot  + i_s1;
6 endmodule // operation
```

Register

- Las variables *reg* se usan para el almacenamiento implícito. Es decir que, al asignarle un determinado valor a una variable *reg*, a menos que se modifique dicha variable, conservará el valor previamente asignado.
- **Una variable *reg* es inferida como registro o dispositivo de almacenamiento cuando tiene asociado una señal de reloj (clock).**
- Es importante notar que una variable de tipo *reg* no implica necesariamente un registro un hardware, sino que puede inferir un cable físico una vez sintetizada.
- Otros tipos de datos *register* son *integer, time* y *real*.

Asignación de Proceso

```
1 module operation
2   (output reg [4:0] o_a, input [1:0] i_s1, input [1:0] i_s2, input clock);
3    reg [3:0]   dot;
4    always@(*) dot  = i_s1 * i_s2;
5    always@(posedge clock) o_a  <= dot  + i_s1;
6 endmodule // operation
```

Declaración

- **Tipo de dato:** Especifica el tipo de dato *wire/reg*.
- **Signed/Unsigned:** Definición de variable signada o no signada. Por defecto las variables son no signadas.
- **Range:**
  - Define el número de bits o ancho de palabra de la variable.
  - En el caso de definición de memorias o arreglos la segunda definición determina el número de filas.
  - Sino se definen el rango el valor por defecto es un bit.
- **Name:** Nombre de la variable.

```
1 wire/reg <signed/unsigned> [<range or width_word>] <name> [<range or len_memory>];
```

Ejemplo

WIRE

```
1  // 1-bit no signado
2  wire x1;
3
4  // 1-bit signado
5  wire signed x2;
6
7  // 8-bits no signado
8  wire [7:0] bus1;
9
10 // 8-bits signado
11 wire signed [7:0] bus2;
12
13 // 8-bits by 1024-row no signado
14 wire [7:0] array1 [1023:0];
15
16 // 8-bits by 1024-row signado
17 wire signed [7:0] array2 [1023:0];
```

REG

```
1  // 1-bit no signado
2  reg x1_d;
3
4  // 1-bit signado
5  reg x2_d;
6
7  // 8-bits no signado
8  reg [7:0] bus1_d;
9
10 // 8-bits signado
11 reg signed [7:0] bus2_d;
12
13 // 8-bits by 1024-row no signado
14 reg [7:0] memory1 [1023:0];
15
16 // 8-bits by 1024-row signado
17 reg signed [7:0] memory2 [1023:0];
```

Constantes

- Al igual que las variables, una constante en Verilog puede ser de cualquier tamaño.
- Se puede escribir en formato decimal (d), binario (b), octal (o) o hexadecimal (h). Decimal es el formato predeterminado.
- Una constante se representa como $< width >' < type >< value >$:
  - **Width:** Tamaño de la constante.
  - **Type:** Tipo de formato.
  - **Value:** Valor representado.
- Los números decimales sin especificación de tamaño y formato se tratan como enteros con signo.
- Los números decimales con tamaño y formato especificados se tratan como enteros sin signo.

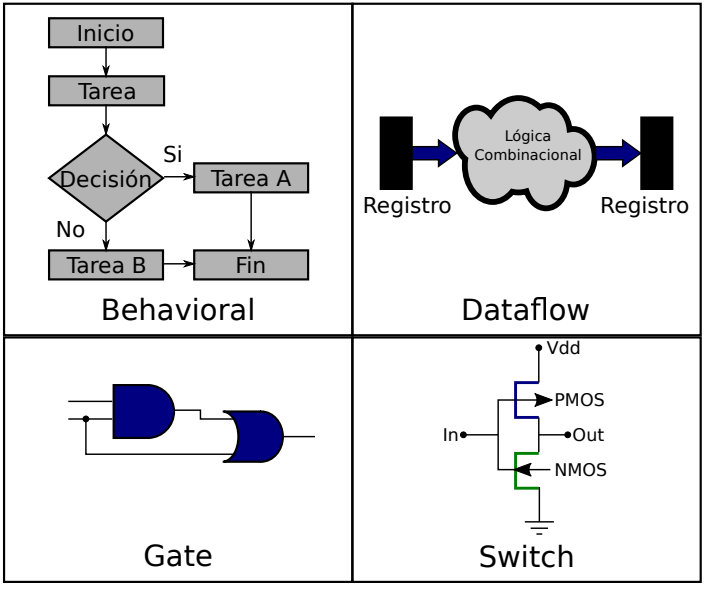
Ejemplos de formato de constantes

Ejemplo de formato		
Entero	**Almacenado como**	**Descripción**
1	0000_0000_0000_0000_0000_0000_0000_0001	32 bits Entero Signado
8'hAA	1010_1010	8 bits Hexadecimal
6'b10_0011	10_0011	6 bits Binario
4'o17	1111	4 bits Octal
'hFF	0000_0000_0000_0000_0000_0000_1111_1111	32 bits Hexadecimal
6'hCA	00_1010	Valor truncado
6'hA	00_1010	Rellena con ceros

Los Cuatro Niveles de Abstracción

Los Cuatro Niveles de Abstracción

Clasificación



Nivel Switch o Transistor

- Es el nivel más bajo de abstracción.
- Este nivel se utiliza para construir compuertas, aunque su uso se está volviendo cada vez mas escaso, ya que las herramientas CAD proporcionan una mejor manera de diseñar y modelar compuertas en el nivel del transistor.

Nivel Gate o Estructural

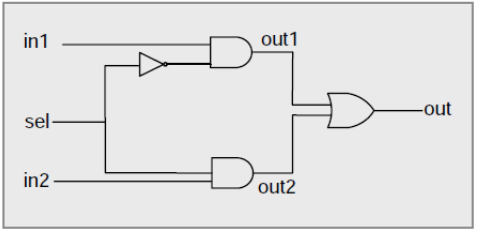
- El modelado a nivel de compuerta (gate) está en un bajo nivel de abstracción y no se usa para el diseño de codificación en RTL.
- El interés en este nivel surge del hecho de que las herramientas de síntesis compilan código de alto nivel y generan código a nivel de compuerta.
- Este código se puede simular utilizando el estímulo desarrollado para el código de nivel RTL.
- La simulación a nivel de puerta es muy lenta en comparación con el código de nivel RTL original.
- Se suele realizar una ejecución selectiva del código para unos cuantos casos de prueba para obtener confianza en el código sintetizado.
- La simulación se puede realizar con información de temporización en un archivo de retardo estándar *(Standard Delay File - SDF)*.
- El SDF se genera para simulaciones pre-layout o post-layout, empleando las herramientas de síntesis o *place and route*.

Nivel Gate o Estructural

- El diseñador puede ejecutar la simulación utilizando el netlist de nivel de compuertas y el SDF.
- Las herramientas proporcionan reportes de tiempo *(report timing)* que facilitan al diseñador la identificación de caminos *(path)* que no cumplen con las restricciones de tiempo.
- El código a nivel compuerta se construye a partir de primitivas Verilog (nand, nor, and, or, xor, buf, not).
- El modelado a este nivel requiere describir el circuito usando compuertas lógicas.
- Los retrasos también se pueden modelar en este nivel.

Los Cuatro Niveles de Abstracción

Nivel Gate o Estructural



(a)

```
module mux (out, in1, in2, sel);
   output out;
   input in1, in2, sel;
   wire out1, out2, sel_n;
   and #5 a1(out1, in1, sel_n);
   and #5 a2(out2, in2, sel);
   or #5 o1(out, out1, out2);
   not n1(sel_n, sel);
endmodule
```

(b)

Ejemplo de implementación a nivel de compuertas.

Nivel Dataflow o RTL

- Es un nivel de abstracción superior al nivel de compuertas.
- Expresiones, operandos y operadores son utilizados con frecuencia en este nivel.
- Notar que utilizar cualquier operador incluye hardware al desarrollo final.
- En el caso de las operaciones aritméticas, dependiendo el dispositivo para el cual se diseña (FPGAs o VLSI), se infiere diferentes componentes. Por ejemplo, en las FPGAs se instancias DSP, los cuales son IP dedicados que optimizan las operaciones aritméticas. A diferencia de implementaciones VLSI que se instancian sumadores CSA.

Los Cuatro Niveles de Abstracción

Nivel Dataflow o RTL

Tipo	Operadores							
Arithmetic	+	−	=	*	/	%	**	
Binary Bitwise	∼	&	∼ &	|	∼ |	^	∼^	^∼
Unary Reduction	&	∼ &	|	∼ |	^	∼^	+	−
Logical	!	&&	||	==	===	!=	!==	
Rational	<	>	<=	>=				
Logical Shift	>>	<<						
Arithmetic Shift	>>>	<<<						
Conditional	? :							
Concatenation	{}							
Replication	{{}}							

Ejemplo

```
1  // Suma
2  assign c = a + b;
3
4  // Condicional
5  assign out = sel ? a : b;
6
7  if(a ==1'b1)
8    out = 1'b1;
9  else
10   out = 1'b0;
11
12 // Repeticion
13 A = 2'b01;
14 B = {4{A}}; // B = 8'b01010101
```

```
1  // Logical Shift
2  A = 6'b101111;
3  B = A >> 2; // B = 6'b001011
4
5  // Arithmetic Shift
6  A = 6'b101111;
7  B = A >>> 2; // B = 6'b111011
8
9  A = 6'b101111;
10 B = A <<< 2; // B = 6'b111110
11
12 // Reduction
13 A = 4'b1011;
14 assign out = &A; // out = 1'b0
```

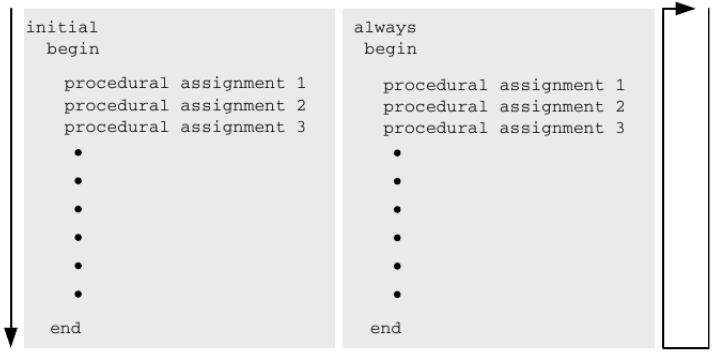
Nivel Comportamental

- El nivel de comportamiento es el nivel más alto de abstracción en Verilog.
- Este nivel proporciona construcciones de lenguaje de alto nivel como *for, while, repeat, if-else* y *case*.
- Existen herramientas de síntesis de nivel de comportamiento que toman un modelo de comportamiento completo y lo sintetizan, pero la lógica generada con estas herramientas no suele ser óptima.
- Estas herramientas no se utilizan en aquellos diseños donde el área, la potencia y la velocidad son consideraciones importantes.
- Verilog restringe todas las declaraciones de comportamiento para ser encerrado en un bloque de procedimiento *(Procedural Block)*.
- En un bloque de procedimiento todas las variables en el lado izquierdo de las sentencias deben declararse como del tipo **reg**, mientras que los operandos del lado derecho en las expresiones pueden ser del tipo **reg** o **wire**.
- Hay dos tipos de bloqueo de procedimiento, *always* e *initial*

Bloques procedurales *Always* e *Initial*

- Un bloque de procedimiento contiene una o varias instrucciones por bloque.
- Una instrucción de asignación utilizada en un bloque de procedimiento se denomina asignación de procedimiento.
- **Initial**
  - Se ejecuta sólo una vez, comenzando en $t = 0$ tiempo de simulación.
  - No es un bloque sintetizable, solamente se utiliza para estímulos en simulación.
  - Generalmente hay mas de un bloque de estímulos, todos ellos se ejecutan concurrentemente.
- **Always**
  - Siempre se ejecuta continuamente partiendo en $t = 0$ y repetidamente después.
  - Es sintetizable.

Los Cuatro Niveles de Abstracción
Nivel Comportamental



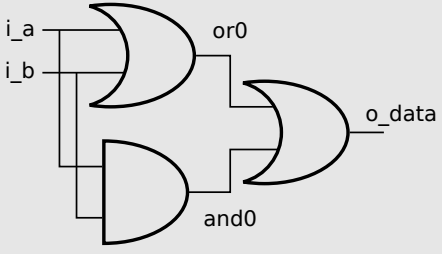
Ejemplo de bloques Always e Initial.

Asignación bloqueante (=)

- Una asignación bloqueante es una asignación regular dentro de un bloque de procedimiento.
- Estas asignaciones se llaman bloqueante porque cada asignación bloquea la ejecución de las asignaciones posteriores en la secuencia.
- En el código RTL Verilog, estas asignaciones se utilizan para modelar la lógica combinacional.
- Para que el código RTL deduzca la lógica combinatoria, las asignaciones de procedimiento bloqueantes se colocan en un mismo bloque *always*.
- Los bloques contienen una **lista de sensibilidad**. Sólo se ejecuta un bloque cuando cambia una de las variables de la lista de sensibilidad.
- El método clásico de listado de sensibilidad es escribir todas las entradas al bloque entre paréntesis, donde cada entrada está separada por una etiqueta “or”. A partir de Verilog-2001 se admiten listas de sensibilidad separadas por comas. También admite escribir un “*” para la lista de sensibilidad.

Asignación Bloqueante

```
1
2  module bloqueante
3    (output reg o_data,
4     input i_a,i_b);
5
6     reg or0, and0;
7
8     always@(*) begin
9         or0      = i_a | i_b;
10        and0     = i_a & i_b;
11        o_data   = or0 | and0;
12    end
13 endmodule
```

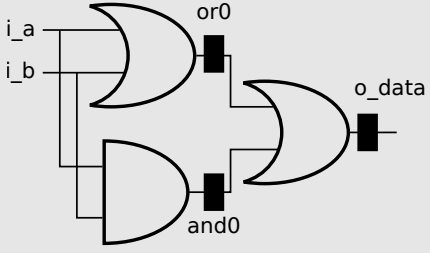


Asignación no-bloqueante ($<=$)

- Las asignaciones de procedimientos no bloqueantes no bloquean otras declaraciones en el bloque y estas sentencias se ejecutan en paralelo.
- El simulador ejecuta esta funcionalidad asignando la salida de estas sentencias a variables temporales y al final de la ejecución del bloque estas variables temporales se asignan a variables reales.
- Las variables del lado izquierdo son del tipo *reg*.
- Se utiliza para inferir lógica sincrónica.
- Todas las asignaciones se realizan en paralelo.

Asignación no-Bloqueante

```
1
2  module nobloqueante
3    (output reg o_data,
4     input i_a,i_b, clock);
5
6     reg or0, and0;
7
8     always@(posedge clock) begin
9        or0      <= i_a | i_b;
10       and0     <= i_a & i_b;
11       o_data   <= or0 | and0;
12    end
13 endmodule
```



Bloqueante vs no-Bloqueante

```
1  reg [2:0]x;
2  reg [2:0]y;
3
4  initial
5  begin
6      x = 3'b0;
7      y = 3'b0;
8  end
9
10 always @(posedge clock)
11 begin
12     x[0] = 1'b1;
13     x[1] = x[0];
14     x[2] = x[1];
15 end
16
17 always @(posedge clock)
18 end
19     y[0] <= 1'b1;
20     y[1] <= y[0];
21     y[2] <= y[1];
22 end
```

- ¿Qué pasa con x e y al producirse un flanco de clock?
- ¿Cuál tiene más sentido práctico?

Bloqueante vs no-Bloqueante

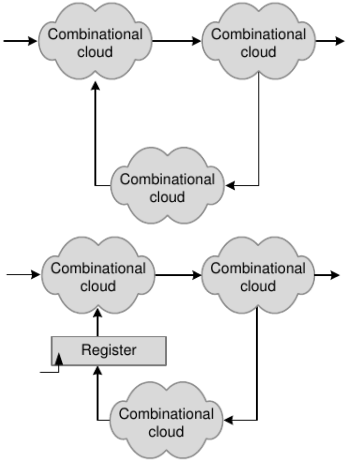
Es importante tener en cuenta que, cuando se codifica en RTL, la asignación procedural no bloqueante ($<=$) debe usarse sólo para modelar lógica secuencial y la asignación procedural bloqueante ($=$) debe usarse sólo para modelar lógica combinacional. No deben mezclarse por ningún motivo ambos tipos de asignación en un mismo bloque procedural.

Control de Tiempo

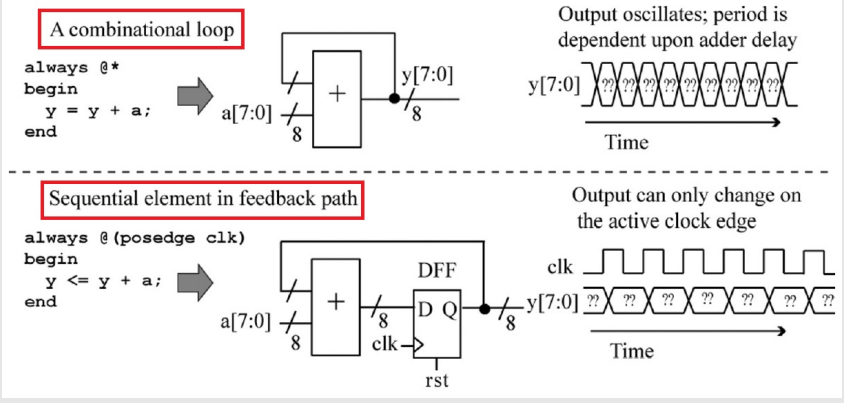
- Verilog proporciona diferentes construcciones de tiempo para modelar el tiempo y los retardos en el diseño. El simulador de Verilog trabaja internamente sin unidades de tiempo para simular la lógica.
- El tiempo simulado en cualquier instancia del diseño se puede acceder utilizando la variable “$time”.
- El diseñador puede insertar retardos en el código colocando **#**<**numero**> (por ej. #10). Al encontrar esta sentencia, la simulación detiene la ejecución de la sentencia hasta que hayan pasado <numero> de unidades de tiempo. El control se libera de esa sentencia o bloque para que otros procesos puedan ejecutarse.
- No se utiliza para síntesis.
- Otra directiva de control de tiempo importante en Verilog es “@”. Esta directiva modela el control basado en eventos. Detiene la ejecución de la sentencia hasta que sucede el evento. Esta construcción de control de tiempo se usa para modelar lógica combinacional y secuencial en bloques procedurales.
- En simulación se puede asignar una unidad de tiempo representativa a los retardos mediante el *timescale* que se define como *'timescale* < *time_unit* > / < *time_precision* >. La unidad de tiempo definirá la unidad de los retardos en simulación y la unidad de precisión definirá cómo se redondean antes de ser usados en la simulación. Ej: *'timescale 1ns/100ps*

Sistema Realimentado (Feedback)

- Cualquier circuito realimentado que sea solamente combinacional no será sintetizado ni simulado correctamente por la herramienta. Se los suele llamar *loops combinacionales* y deben ser evitados.
- Un sistema realimentado, como por ejemplo un acumulador, debe incluir registros en algún punto del camino realimentado.
- Importante: Utilizar una señal de inicialización (reset) en estos registros. Esto es porque cualquier valor previo almacenado en los registros corromperá todos los cálculos futuros. Desde el punto de vista de simulación, se asume una x en los registros hasta su inicialización, la cual puede propagarse por el circuito realimentado si no se aplica una señal de reset adecuadamente.
- El reset puede ser síncrono o asíncrono y, en ambos casos, activo por alto o bajo.



Ejemplo: Sistema Realimentado (Feedback)



Ejemplo: reset síncrono y asíncrono

```
1  // Registro con reset asincrono
2  // activo por bajo
3  always @ (posedge clk or negedge rst_n)
4    begin
5      if(!rst_n)
6        acc_reg <= 16'b0;
7      else
8        acc_reg <= data+acc_reg;
9    end
10
11 // Registro con reset asincrono
12 // activo por alto
13 always @ (posedge clk or posedge rst)
14   begin
15     if(rst)
16       acc_reg <= 16'b0;
17     else
18       acc_reg <= data+acc_reg;
19   end
```

```
1  // Registro con reset sincrono
2  // activo por bajo
3  always @ (posedge clk)
4    begin
5      if(!rst_n)
6        acc_reg <= 16'b0;
7      else
8        acc_reg <= data+acc_reg;
9    end
10
11 // Registro con reset asincrono
12 // activo por alto
13 always @ (posedge clk)
14   begin
15     if(rst)
16       acc_reg <= 16'b0;
17     else
18       acc_reg <= data+acc_reg;
19   end
```

Generación de Clock y Reset

- Normalmente estas señales provienen de un oscilador de cristal fuera del chip o FPGA (clock) y un pulsador (reset).
- El Clock se distribuye o encamina utilizando árboles de reloj (clock trees). Se trata de rutas especialmente diseñadas que llevan los relojes a los registros causando un mínimo sesgo *(skew)* a estas señales especiales.
- En la FPGA las fuentes de clock externas se deben asociar a uno de los pines dedicados para este propósito. En Xilinx, la asociación de pines se define en un archivo de **constraints** con la extensión *XDC (Xilinx Design Constraints)*. Esta extensión se adoptó en Vivado reemplazando al archivo *UCF (User Constraint File)* usado en ISE.
- En la simulación, estas señales son generadas como estímulos para poder llevar a cabo la verificación del código RTL, como se muestra en el siguiente ejemplo.

Ejemplo

```
1 initial begin // Inicializacion de variables
2    clk      = 0; // clock se inicia con 0
3    #5 rst_n = 0; // reset en bajo
4    #2 rst_n = 1; // reset en alto
5 end
6 always #10 clk = (~clk); // generacion de clock
```

Sentencia CASE

- La sentencia *case* evalúa una expresión y en función de su valor ejecuta la sentencia o grupo de sentencias agrupadas en el primer caso en que coincida.
- En caso de no cubrir todos los posibles valores de la expresión a avaluar, es necesario el uso de un caso por defecto (*default*).
- La sentencia case reconoce tanto al valor *z* como *x* como valores lógicos. Se puede usar el símbolo *?* para representar un valor indiferente (“don’t care”).
- Existen dos variantes de *case*:
  - *casez*: Considera los valores en *z* como indiferentes (“don’t care”).
  - *casex*: Considera los valores en *x* y *z* como indiferentes (“don’t care”).

Sentencia CASE -Ejemplos

```
1  module mux4_1(in1,in2,in3,in4,sel,out);
2     input  [1 :0] sel;
3     input  [15:0] in1, in2, in3, in3;
4     output [15:0] out;
5     reg    [15:0] out;
6     always @(*) begin
7        case (sel)
8          2'b00: out = in1;
9          2'b01: out = in2;
10         2'b10: out = in3;
11         2'b11: out = in4;
12         default: out = 16'bx;
13       endcase
14    end
15 endmodule
```

```
1  always @(op_code)
2    begin
3       casez (op_code)
4          4'b1???: alu_inst(op_code);
5          4'b01??: mem_rd(op_code);
6          4'b001?: mem_wr(op_code);
7       endcase
8    end
```

Sentencia IF

- Evalúa la expresión y realiza una tarea o asignación en caso de ser True o False.
- Soporta múltiples condiciones.

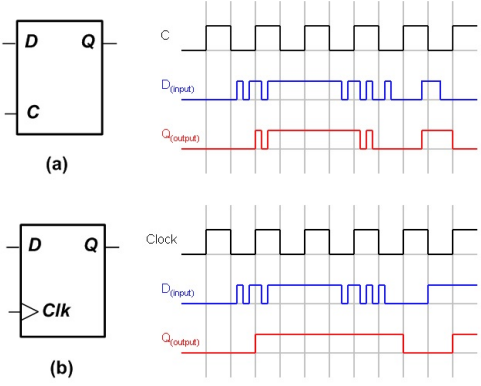
Ejemplo

```
1 if (brach_flag)
2   PC = brach_addr;
3 else
4   PC = next_addr;
```

```
1 always @(op_code) begin
2     if (op_code == 2'b00)
3       cntr_sgn = 4'b1011;
4     else if(op_code == 2'b01)
5       cntr_sgn = 4'b1110;
6     else
7       cntr_sgn = 4'b0000;
8 end
```

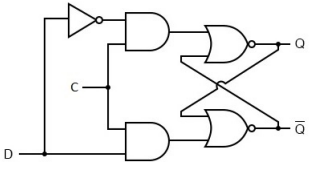
Evitar Latches en el Diseño

- Un diseñador debe evitar cualquier sintaxis de RTL que infiera latches.
- Un latch es un dispositivo de almacenamiento que almacena un valor sin el uso de un reloj.
- Su implementación suele ser específica de la tecnología y suelen generar problemas de routeo y timing, o derivar en loops combinacionales, por lo que deben evitarse en diseños síncronos.
- Para evitarlos, el diseñador debe cumplir con ciertas pautas de codificación al definir lógica combinacional en bloques procedurales.
  - En sentencias ***if***, se debe siempre incluir la instrucción *else* y para cada condición se deben asignar valores a todas las variables, o bien se deben asignar valores predeterminados a todas las variables fuera del bloque condicional.
  - En sentencias ***case***, se deben especificar todas las condiciones posibles de la variable de selección, y para cada condición se deben asignar valores a todas las variables. De no ser así, se debe incluir la condición *default* y asignar valores predeterminados a todas las variables fuera del bloque condicional.



D-Latch (a) y D-Flip-Flop (b)

C	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0



D-Latch lógica interna

Latch inferido - Caso 1

```
1  input [1:0] sel;
2  reg   [1:0] out_a,out_b;
3  always @(*) begin
4     if (sel == 2'b00) begin
5        out_a = 2'b01;
6        out_b = 2'b10;
7     end
8     else
9       out_a = 2'b01;
10 end
```

```
1  input [1:0] sel;
2  reg   [1:0] out_a,out_b;
3  always @(*) begin
4     out_a = 2'b00;
5     out_b = 2'b00;
6     if (sel=2'b00) begin
7        out_a = 2'b01;
8        out_b = 2'b10;
9     end
10    else
11      out_a = 2'b01;
12 end
```

Latch inferido - Caso 2

```
1  always @(*) begin
2     out_a = 2'b00;
3     out_b = 2'b00;
4     if (sel==2'b00) begin
5        out_a = 2'b01;
6        out_b = 2'b10;
7     end
8     else if (sel == 2'b01)
9       out_a = 2'b01;
10 end
```

```
1  always @(*) begin
2     out_a = 2'b00;
3     out_b = 2'b00;
4     if (sel==2'b00) begin
5        out_a = 2'b01;
6        out_b = 2'b10;
7     end
8     else if (sel == 2'b01)
9       out_a = 2'b01;
10    else
11      out_a = 2'b00;
12 end
```

Loops

- Se utilizan para ejecutar un bloque de sentencias múltiples veces.
  - for: Es sintetizable y el más comúnmente usado en RTL.
  - while: Es sintetizable pero no se recomienda su uso en RTL.
  - repeat: Es sintetizable pero no se recomienda su uso en RTL.
  - forever: No es sintetizable, sólo se usa en simulaciones.

Ejemplos

```
integer ii;
reg [7:0] op;

always @(posedge i_clock) begin
  if (load_en) begin
    op <= load_val;
  end
  else begin
    for(ii=0; ii<7; ii=ii+1) begin
      op[ii+1] <= op[ii];
    end
    op[0] <= op[7];
  end
end
```

```
reg clock;

initial begin
  clock = 1'b0;
  repeat (10)
    #5 clock = !clock;
  $display("Simulation Completed");
end
```

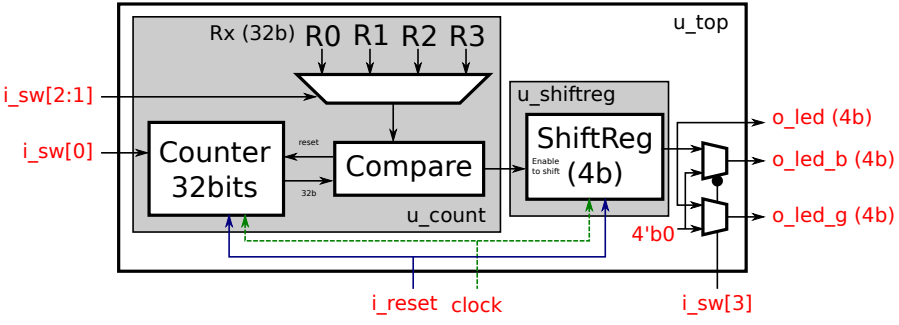
Loops - Ejemplos

```
1  integer ii;
2  reg [7:0] data[15:0];
3
4  initial begin
5    ii=0;
6    data[ii] = ii*ii;
7
8    while (data[ii] < 100) begin
9      $display("Time %2d: data at Index %1d is %2d", $time, ii
         , data[ii]);
10     ii = ii + 1;
11     data[ii] = ii*ii;
12     #10;
13   end
14 end
```

```
1  reg clock;
2
3  initial begin
4    clock = 0;
5    forever begin
6      #5 clock = !clock;
7    end
8  end
9
10 initial begin
11   $monitor ("Time = %d, clock = %b", $time, clock);
12   #100 $finish;
13 end
```

Modelar el comportamiento

- Describir cada uno de los sub-sistemas



Control de Simulaciones

- Verilog provee varias funciones para el control de las simulaciones.
  - *$finish*: Sale de la simulación.
  - *$stop*: Detiene la simulación, que luego puede ser retomada desde el punto de suspensión.
  - *$display*: Imprime una salida.
  - *$monitor*: Es similar a *$display* pero está activo todo el tiempo. Solo puede ejecutarse un *$monitor* a la vez en toda la simulación.
  - *$fmonitor,$fdisplay*: Escriben los valores en archivos, que deben ser abiertos previamente con la función *$fopen*.
  - *$readmemb,$readmemh*: Permiten la carga de datos desde archivos en formato binario o hexadecimal a memorias.

Otros

- **Directivas del pre-procesador**:
  - ***‘define***: Define una macro, que consiste en la sustitución de una o varias lineas de texto por un nombre. Se suele usar para definir constantes.

```
1 ‘define COEFF 6’b001101
2
3 reg [5:0] term;
4
5 initial begin
6   term = ‘COEFF;
7 end
```

  - ***‘ifdef - ‘else - ‘endif***: Condicional pre-compilación.
  - ***‘include***: Permite insertar el contenido de un archivo fuente en otro archivo durante la compilación en el lugar donde se define el comando. La compilación continúa como si el contenido del archivo fuente incluido apareciera en lugar del comando *‘include*. Se suele usar para incluir definiciones y tareas de uso común, evitando código repetido dentro de los módulos. Ejemplo: *‘include “utiles.v”*
- **Comentarios**: Al igual que C se utilizan // y /* */

Parámetros (parameter)

- Los parámetros son constantes que son locales a un módulo.
- A un parámetro se le asigna un valor por defecto en el módulo y para cada instancia de este módulo se le puede asignar un valor diferente.
- Los parámetros son muy útiles para mejorar la reutilización de los módulos desarrollados.
- Un módulo se llama **parametrizado** *(parametered)* si está escrito de tal manera que el mismo módulo se pueda instanciar para diferentes anchos de puertos de entrada y salida.

Parámetros locales (localparam)

- Es una constante local al módulo.
- No pueden cambiar su valor en una instancia.

Los Cuatro Niveles de Abstracción
Nivel Comportamental

Declaración

```
module adder(o_sum,i_a,i_b,i_c);
    parameter  DATA    = 4;
    localparam OUTPUT = DATA + 1;

    output [OUTPUT -1 : 0] o_sum;
    input  [DATA   -1 : 0] i_a, i_b;
    input                  i_c;

    assign o_sum = i_a+i_b+i_c;
endmodule
```

```
module adder
  #(parameter DATA    = 4,
    parameter OUTPUT = DATA + 1)
    (output [OUTPUT -1 : 0] o_sum,
     input  [DATA   -1 : 0] i_a, i_b,
     input                  i_c);

   assign o_sum = i_a+i_b+i_c;
endmodule
```

Instancia

```
module stimulus(o_sum,in1,in2,i_c);
    parameter DATA    = 8;
    localparam OUTPUT = DATA + 1;

    output [OUTPUT -1 : 0] o_sum;
    input  [DATA   -1 : 0] in1, in2;
    input                  i_c;

    adder #(DATA)
    u_adder(o_sum,
            in1  ,
            in2  ,
            i_c);
endmodule
```

```
module stimulus
  #(parameter DATA    = 8,
    parameter OUTPUT = DATA + 1)
   (output [OUTPUT -1 : 0] o_sum,
    input  [DATA   -1 : 0] in1, in2,
    input                  i_c);

   adder #(.DATA(DATA),
           .OUTPUT(OUTPUT))
   u_adder(.o_sum(o_sum),
           .in1(in1),
           .in2(in2),
           .i_c(i_c));
endmodule
```

Generate

- Un bloque *generate* se puede usar para crear múltiples instancias de módulos y código, o instanciar modulos y definir código condicionalmente.
- Se definen dentro de un módulo, usando las palabras clave *generate* y *endgenerate*.
- Pueden contener instancias de módulos, declaraciones de asignación continua y bloques procedurales.
- No pueden contener declaraciones de puertos ni de parámetros.
- Pueden ser anidados.
- Tipos:
  - Generate loop (for)
  - Generate condicional (if o case)

Generate for loop

- Permite instanciar un módulo o replicar asignaciones N veces.
- La variable del loop se debe definir usando el tipo *genvar*.

Ejemplo

```
1  module adder
2    #(parameter N=4)
3      (  input [N-1:0] a, b,
4         output [N-1:0] sum, cout);
5
6    genvar i;
7
8    generate
9      for (i = 0; i < N; i = i + 1)
10     begin : ha_instances //nombrarlos es una buena practica
11           ha u_ha (.a(a[i]), .b(b[i]), .sum(sum[i]), .cout(cout[i]));
12     end
13   endgenerate
14 endmodule
15
16 // Half-adder
17 module ha ( input a, b, output  sum, cout);
18   assign sum  = a ^ b;
19   assign cout = a & b;
20 endmodule
```

Generate condicional

- Generate if: Permite hacer uso de la sentencia *if - else if - else* para instanciar módulos o definir bloques de código condicionalmente.
- Generate case: Permite hacer uso de la sentencia *case* para instanciar módulos o definir bloques de código condicionalmente.

Ejemplo

```
1  module operation
2      #( parameter OP_SEL = 0)
3       ( input a, b,
4         output [1:0] out);
5
6    generate
7      if (OP_SEL)
8        addition u_add (.a(a), .b(b), .out(out));
9      else
10       substraction u_sub (.a(a), .b(b), .out(out));
11   endgenerate
12
13 endmodule
```

```
1  module operation
2      #( parameter OP_SEL = 0)
3       ( input a, b,
4         output [1:0] out);
5
6    generate
7      case (OP_SEL)
8        0 : addition u_add (.a(a), .b(b), .out(out));
9        1 : substraction u_sub (.a(a), .b(b), .out(out));
10     endcase
11   endgenerate
12
13 endmodule
```

Tarea

- Se puede utilizar para codificar una funcionalidad que se repite varias veces en un módulo.
- Tiene entradas, salidas y puede tener sus variables locales.
- Todas las variables definidas en el módulo también son accesibles en la tarea.
- Debe definirse en el mismo módulo utilizando palabras claves *task* y *endtask*. Se puede llamar desde varios módulos siempre y cuando esté definida en un archivo aparte que se incluye utilizando la palabra clave *'include* dentro de cada módulo.
- Se llaman desde bloques *initial* o *always* o desde otras tareas en un módulo.
- Pueden contener declaraciones de tiempo *(#,@)*.
- Pueden usar y/o asignar valores a cualquier señal declarada como global.
- Importa el orden de los puertos de entrada y salida al momento de llamar una tarea.
- Pueden utilizarse para modelar lógica combinacional o secuencial.

Ejemplo

```
1  module RCA
2    (output reg [3:0] o_sum,
3     output reg       o_c,
4     input      [3:0] i_a, i_b,
5     input            i_c);
6
7     reg [4:0]        carry;
8     integer          i;
9
10    task FA(output reg o_sum, o_carry,
11            input      in1, in2, i_carry);
12
13       {o_carry,o_sum} = in1 + in2 + i_carry;
14    endtask // FA
15
16    always @(*) begin
17       carry[0]=c_in;
18       for(i=0; i<4; i=i+1) begin
19          FA(i_a[i], i_b[i], carry[i], o_sum[i], carry[i+1]);
20       end
21       o_c = carry[4];
22    end
23 endmodule
```

Función

- Similar a una tarea, ya que también implementa código que se puede llamar varias veces dentro de un módulo.
- Se define en el módulo usando las palabras clave *function* y *endfunction*.
- Sólo puede calcular una salida, debiendo tener al menos una entrada.
- La salida debe asignarse a una variable implícita que lleve el nombre y el alcance de la función.
- No puede utilizar construcciones de tiempo *(#,@)*.
- Puede ser llamada desde un bloque de procedimiento o una declaración de asignación continua.
- Puede llamarse desde otras funciones y tareas, mientras que una función no puede llamar a una tarea.
- Sólo pueden utilizarse para modelar lógica combinacional (no soportan asignaciones no bloqueantes).

Ejemplo

```
1  module MUX4to1(output  out, input [3:0] in, input [1:0] sel);
2     wire                out1, out2;
3     function MUX2to1;
4        input            in1, in2;
5        input            select;
6        assign MUX2to1 = (select) ? in2:in1;
7     endfunction // MUX2to1
8     assign out1 = MUX2to1(in[0], in[1], sel[0]);
9     assign out2 = MUX2to1(in[2], in[3], sel[0]);
10    assign out  = MUX2to1(out1, out2, sel[1]);
11 endmodule
12 module testFunction;
13    reg [3:0] IN;
14    reg [1:0] SEL;
15    wire      OUT;
16    MUX4to1 mux(IN, SEL, OUT);
17    initial begin
18       IN = 1;    SEL = 0;
19       #5 IN = 7; SEL = 0;
20       #5 IN = 2; SEL = 1;
21       #5 IN = 4; SEL = 2;
22       #5 IN = 8; SEL = 3;
23    end
24    initial
25      $monitor($time, " %b %b %b\n", IN, SEL, OUT);
26 endmodule
```

Aritmética Signada

- Las variables *reg* y *wire* modifican usando la palabra clave *signed*.
- Habilita las operaciones aritméticas signadas que consideran la extensión de signo.
- Se castea una variable a signada o no signada usando $*signed* o $*unsigned*.
- $>>>$ considera la expansión de signo.

Ejemplo

```
1  // Asignaciones
2  reg signed      [63 :0] data;
3  wire signed     [7  :0] vector;
4  input signed    [31 :0] a;
5  function signed [128:0] alu;
6
7  // Desplazamiento Signado
8  reg [63:0]        data;
9  always @(*) begin
10    out = ($signed(data))>>> 2;
11 end
```

Verificación en Diseño de Hardware

Verificación en Diseño de Hardware

Introducción

- Verilog está especialmente diseñado para el modelado de hardware y carece de características que facilitan la verificación de complejos diseños digitales.
- Herramientas automáticas de diseño electrónico *Electronic Design Automation - EDA* facilitan al diseñador la verificación del diseño empleando diferentes metodologías.
- Es importante señalar que la verificación debe realizarse de manera que el código desarrollado para la verificación sea reutilizable.
- Tres términos que generan confusión:
  - **Validación:** Las especificaciones (a menudo un modelo C) satisfacen los requisitos del mercado o del cliente.
  - **Verificación:** La implementación (RTL, netlist) coincide con las especificaciones.
  - **Prueba (testing):** El dispositivo fabricado coincide con la implementación.

Verificación en Diseño de Hardware

Enfoques para Probar un Diseño Digital

- **Prueba Black-Box**
  - Es una prueba contra especificaciones cuando la estructura interna del sistema es desconocida.
  - Se aplica un conjunto de entradas y se comprueban las salidas en función de la especificación o de la salida esperada, sin tener en cuenta los detalles internos del sistema.
  - El diseño a probar se denomina normalmente el **dispositivo bajo prueba** *(Device Under Test - DUT)*.
- **Prueba White-Box**
  - Contrasta con las especificaciones mientras se hace uso de la estructura interna conocida del sistema.
  - Permite al desarrollador localizar un error para una rápida corrección.
  - Por lo general, este tipo de pruebas es realizado por el desarrollador del módulo.

Niveles de Prueba en el Ciclo de Desarrollo

- **Prueba a nivel de módulo y componente**
  - Un componente es una combinación de módulos.
  - Se emplea la técnica *White-Box* y es realizada por el desarrollador del módulo.
- **Prueba de integración**
  - Los módulos implementados se instancian como componentes de un módulo y se utilizan pruebas *Black-Box* y *White-Box*.
  - La interacción entre ellos se verifica utilizando casos de prueba *(test cases)*.
- **Prueba a nivel de sistema**
  - Se realiza después de integrar todos los componentes del sistema, para comprobar el cumplimiento de las especificaciones.
  - Incluye verificación funcional, evaluación del rendimiento y pruebas de *stress*.
- **Pruebas de regresión**
  - Son un subconjunto de vectores de prueba que el diseñador debe ejecutar después de cualquier corrección de errores o modificación significativa en un diseño ya probado.

Métodos para generar casos de prueba

- **Generación exhaustiva de vectores de prueba:** Muy utilizado para diseños pequeños o pruebas a nivel de módulos, ya que la generación de vectores en ocasiones demando mucho tiempo de procesamiento.
- **Pruebas aleatorias:** Suele utilizarse para diseños grandes, en donde las entradas se generan aleatoriamente a partir de un gran conjunto de valores posibles. En muchos casos, se consideran los casos extremos *worst and best cases*.
- **Pruebas basadas en restricciones (constraint):** Funciona con pruebas aleatorias, por lo que la aleatoriedad se ve obligada a trabajar en un rango definido.
- **Pruebas para localizar una falla:** Se emplea un conjunto de secuencias de entrada como estrategia de prueba para identificar fallos, teniendo en cuenta patrones que pueden generar errores en el diseño.
- **Comprobador de modelos (model checkers):** El diseñador puede hacer uso de modelos para comprobar diseños que implementan protocolos estándar (por ejemplo, interfaces).

Cobertura de Código (coverage)

- Las herramientas de cobertura de código proporcionan la capacidad de evaluar la calidad de la verificación.
- Pueden proporcionar información sobre qué partes del código han sido probadas, así como qué estados y argumentos de una máquina de estado finito han sido probados.
- Una cobertura de código del 100% es muy difícil lograr, por lo cual se buscan casos funcionales que permitan arribar a ese valor.
- La cobertura considera expresiones y *toggle* de compuertas.