DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Sistemas Digitales

Lenguajes de descripción de hardware - HDL

Primer Cuatrimestre 2024

Sistemas Digitales
DC - UBA

Introducción

Clase de hoy

Hoy vamos a ver:

- Introducción a los lenguajes de descripción de hardware (HDL).

Clase de hoy

Hoy vamos a ver:

- Introducción a los lenguajes de descripción de hardware (`HDL`).
- Uso en **lógica combinatoria**.

Clase de hoy

Hoy vamos a ver:

- Introducción a los lenguajes de descripción de hardware (`HDL`).
- Uso en **lógica combinatoria**.
- Uso en **lógica secuencial**.

Clase de hoy

Hoy vamos a ver:

- Introducción a los lenguajes de descripción de hardware (`HDL`).
- Uso en **lógica combinatoria**.
- Uso en **lógica secuencial**.
- Simulación y testing.

Introducción

Hasta ahora vimos como modelar hardware a partir de una vista esquemática (ej. Logisim), en base a la composición de compuertas lógicas o componentes encapsulados de forma discreta y explícita. Pero a la hora de diseñar e implementar hardware industrial y a gran escala, es necesario contar con **herramientas que permitan describir la estructura y comportamiento de nuestro diseño de forma sencilla, sintética y escalable**. En particular utilizando una gramática simple y que encapsule nuestras necesidades.

Para esto vamos a hacer uso de los **lenguajes de descripción de hardware** (HDL por sus siglas en inglés). Existen dos lenguajes dominantes actualmente: `VHDL` y `System Verilog`. En la materia vamos a utilizar `System Verilog` pero los principios y experiencia con los que nos vamos a familiarizar facilitan aprender un segundo HDL.

Módulos

Un módulo es un bloque de hardware con un nombre, una lista de entradas y de salidas. Se pueden describir de dos maneras:

Módulos

Un módulo es un bloque de hardware con un nombre, una lista de entradas y de salidas. Se pueden describir de dos maneras:

- **Comportamental:** Describiendo como se modifican las señales de salida en base a la entrada y el estado del componente.

Módulos

Un módulo es un bloque de hardware con un nombre, una lista de entradas y de salidas. Se pueden describir de dos maneras:

- **Comportamental:** Describiendo como se modifican las señales de salida en base a la entrada y el estado del componente.
- **Estructural:** Describiendo la composición de diversos componentes y cómo se vinculan sus entradas y salidas entre sí.

Módulos

Un módulo es un bloque de hardware con un nombre, una lista de entradas y de salidas. Se pueden describir de dos maneras:

- **Comportamental:** Describiendo como se modifican las señales de salida en base a la entrada y el estado del componente.
- **Estructural:** Describiendo la composición de diversos componentes y cómo se vinculan sus entradas y salidas entre sí.

Una compuerta `AND`, un multiplexor o un sumador son ejemplos de módulos.

Módulos

A continuación vemos un ejemplo de un módulo en `System Verilog` que asigna a su única salida una suma de productos sobre las señales `a, b, c`, $y = \neg a \wedge \neg b \wedge \neg c \vee a \wedge \neg b \wedge \neg c \vee a \wedge \neg b \wedge c$ o, en notación aritmética, $y = \overline{abc} + a\overline{bc} + a\overline{b}c$, aquí las entradas se indican como `input logic`, las salidas como `output logic` y `module foo` indica el comienzo del módulo llamado `foo`:

System Verilog

```
module foo(input logic a, b, c, output logic y);
 assign y =
  ~a & ~b & ~c |
  a & ~b & ~c |
  a & ~b & c;
endmodule
```

Simulación y síntesis

El código HDL es la fuente, principalmente, de dos procesos:

Simulación y síntesis

El código HDL es la fuente, principalmente, de dos procesos:

- **Simulación:** En el que se simula el comportamiento del módulo al suministrar una serie de valores a sus señales de entrada y validar que las señales de salida sean las que se esperan.

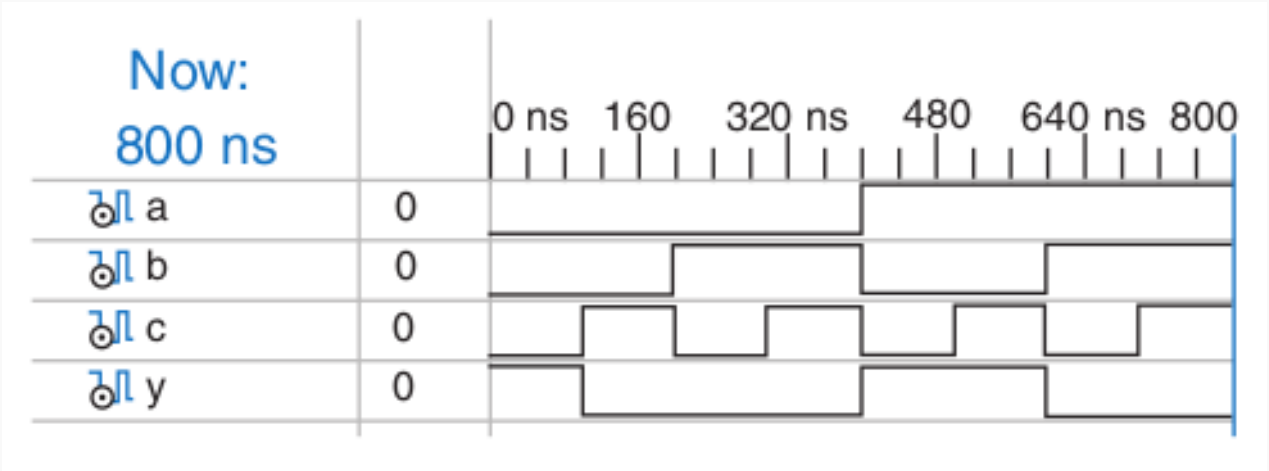
Simulación y síntesis

El código HDL es la fuente, principalmente, de dos procesos:

- **Simulación:** En el que se simula el comportamiento del módulo al suministrar una serie de valores a sus señales de entrada y validar que las señales de salida sean las que se esperan.
- **Síntesis:** En el que se traduce la descripción en HDL a un conjunto de compuertas lógicas, de acuerdo al soporte (electrónica) que el componente vaya a tener.

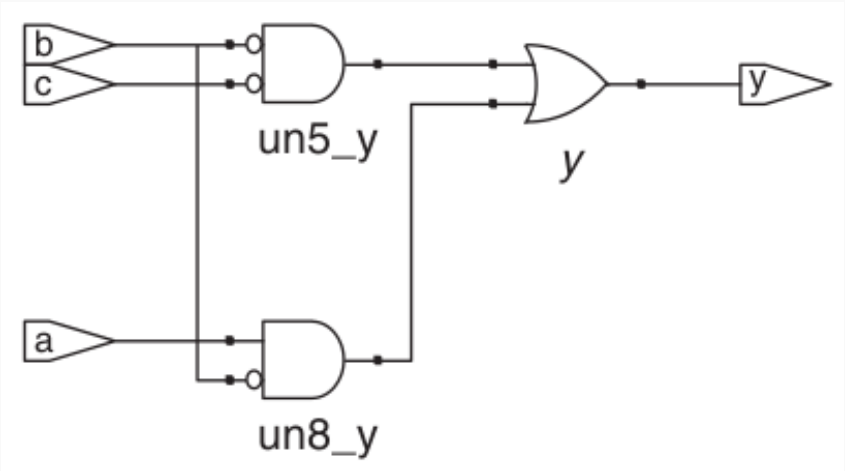
Simulación

Aquí vemos las formas de onda de las entradas y las salidas de nuestro módulo `foo` al ser simulado.



Síntesis

Aquí vemos la traducción de nuestro módulo `foo` en componentes discretos al ser sintetizado.



Lógica combinatoria

Operadores binarios

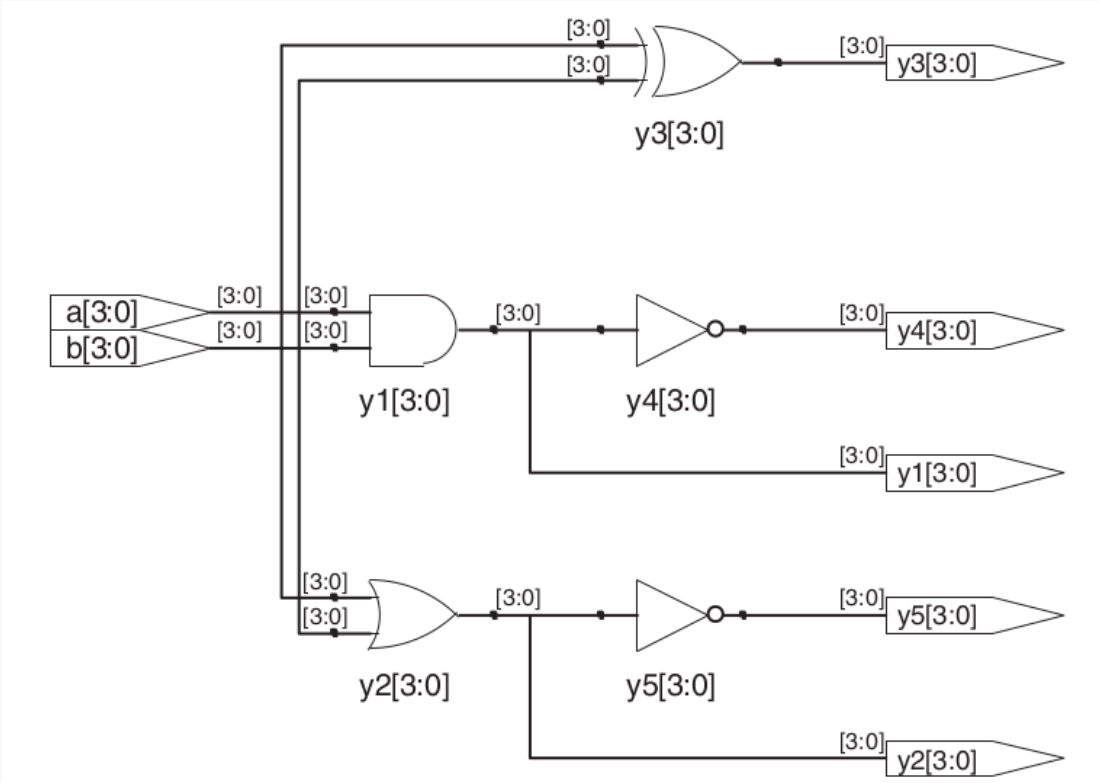
Veamos ejemplos aplicación de operadores a nivel bit (`bitwise`):

System Verilog

```
module compuertas(input logic [3:0] a, b,
 output logic [3:0] y1, y2,y3, y4, y5);

/* compuertas logicas con dos entradas de 4 bits */
 assign y1 = a & b;// AND
 assign y2 = a | b;// OR
 assign y3 = a ^ b;// XOR
 assign y4 = ~(a & b); // NAND
 assign y5 = ~(a | b); // NOR
endmodule
```

El circuito sintetizado para el módulo `compuertas`.



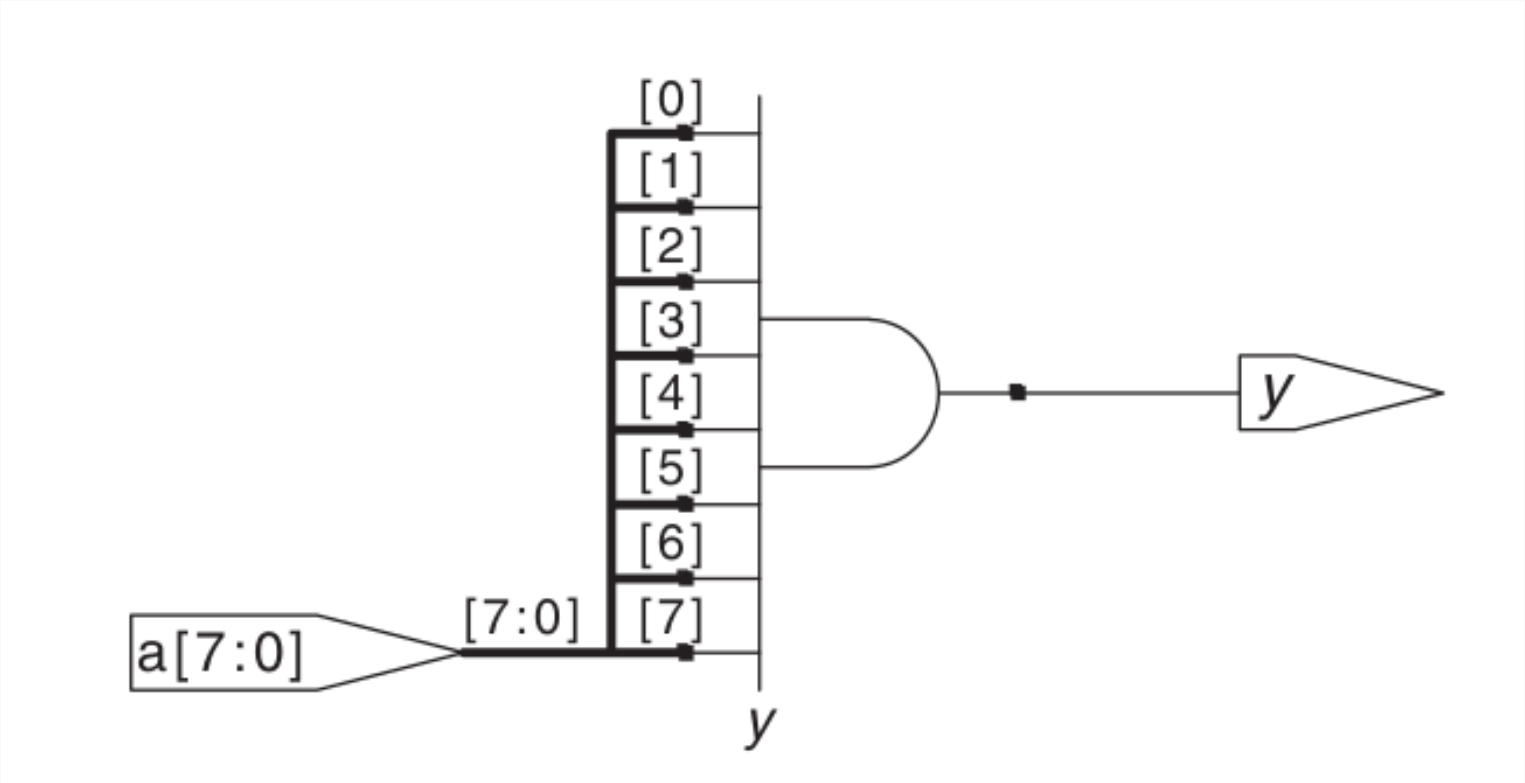
Operadores de reducción

Los operadores de reducción permiten colapsar los bits de una señal de tamaño variable aplicando una operación lógica:

System Verilog

```
module reduccion(input logic[7:0] a, output logic y);
 assign y = &a;
// &a es compacto y equivale a escribir
// assign y = a[7] & a[6] & a[5] & a[4] &
//  a[3] & a[2] & a[1] & a[0];
endmodule
```

El circuito sintetizado para el módulo `reduccion`.



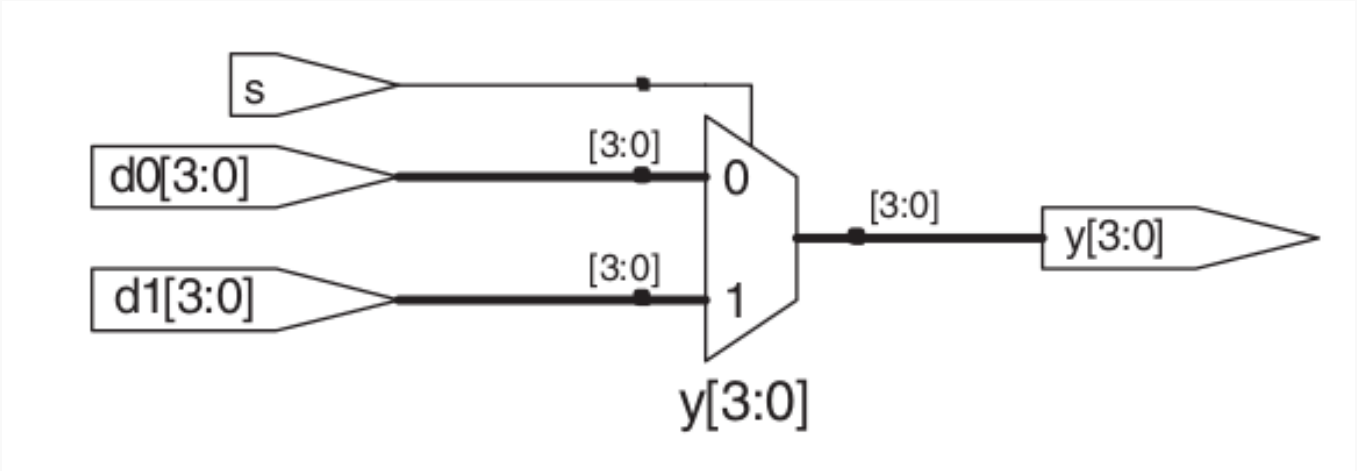
Asignación condicional

La asignación condicional va inferir un multiplexor de dos entradas, su sintaxis es similar a la del operador ternario de C:

System Verilog

```
module mux2(input logic [3:0] d0, d1, input logic s,
 output logic [3:0] y);
 assign y = s ? d1 : d0;
endmodule
```

El circuito sintetizado para el módulo `mux2`.



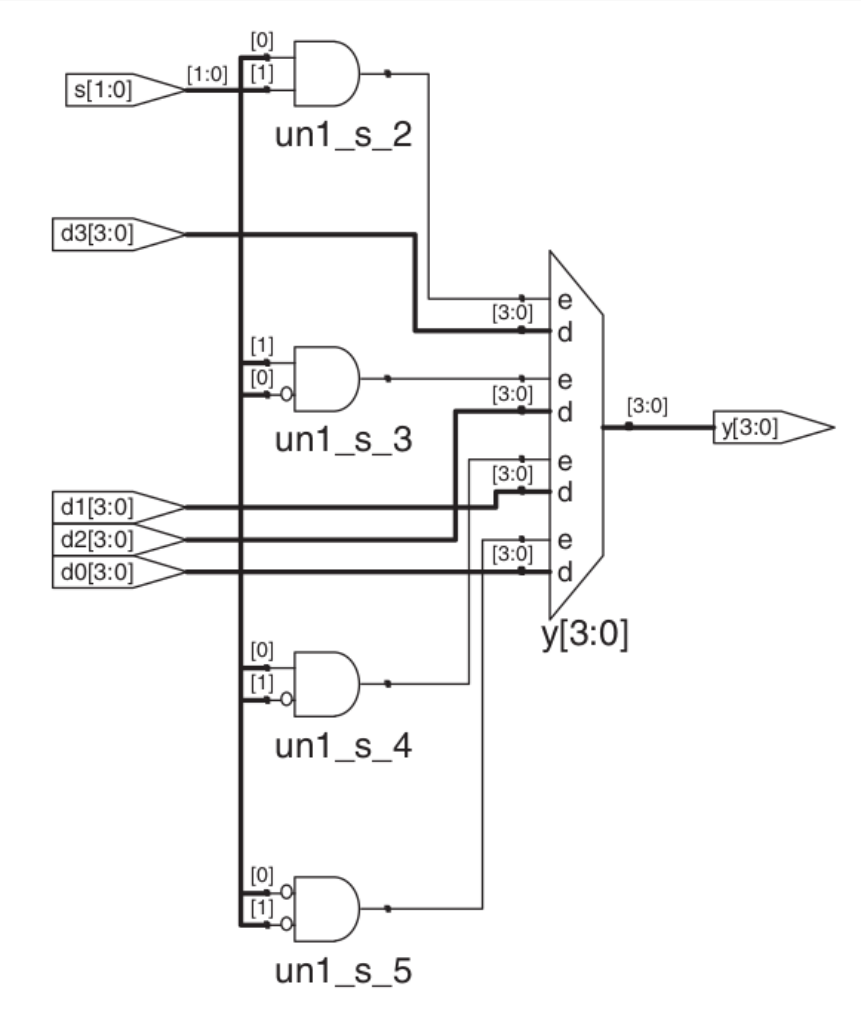
Asignación condicional compuesta

La asignación condicional compuesta va inferir un multiplexor de cuatro entradas:

System Verilog

```
module mux4(input
logic [3:0] d0, d1, d2, d3, input logic[1:0] s,
 output logic [3:0] y);
assign y = s[1] ? (s[0] ? d3 : d2)
 : (s[0] ? d1 : d0);
endmodule
```

El circuito sintetizado para el módulo `mux4` (salida seleccionada por `hot-enable` en lugar de selector).



Variables internas

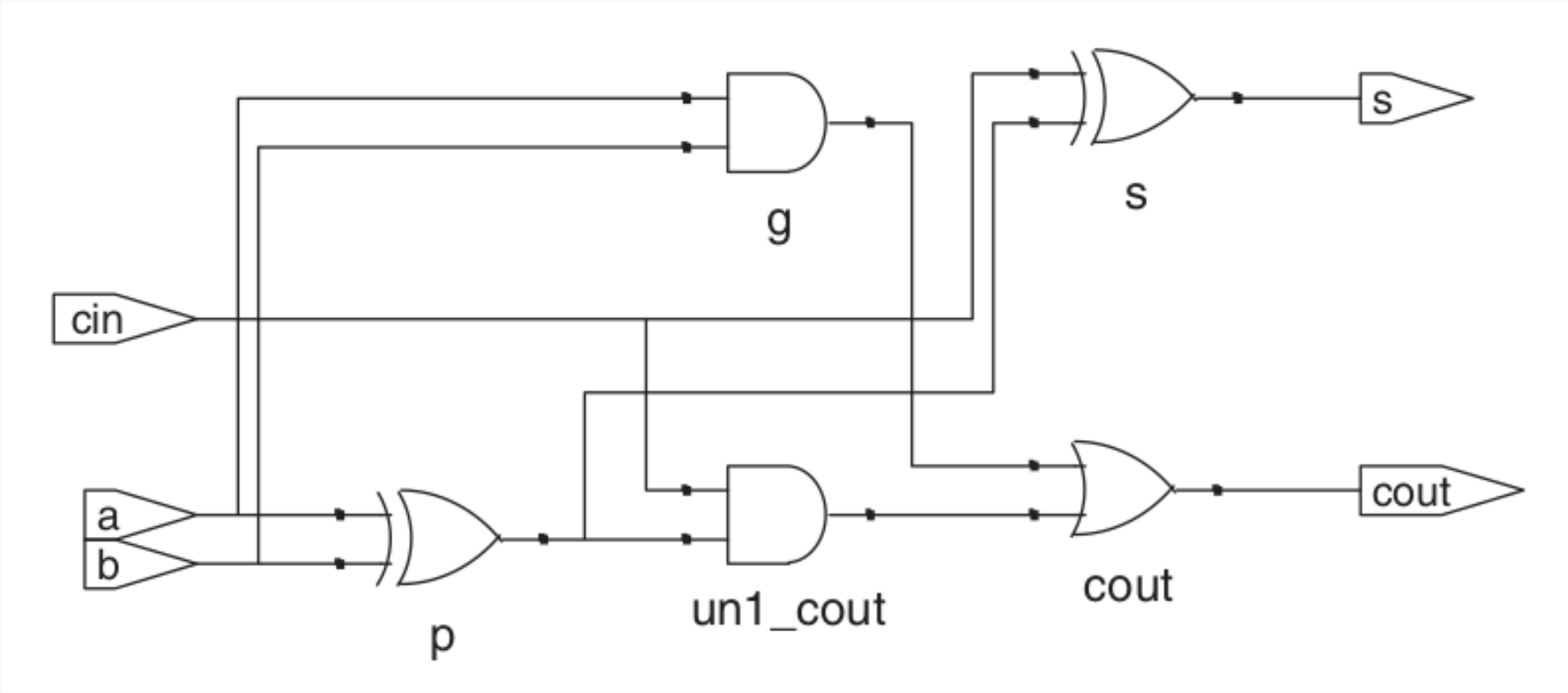
Con frecuencia resulta necesario declarar variables internas para indicar el comportamiento de nuestros circuitos:

System Verilog

```
module fulladder(input logic a, b, cin
 output logic s, cout);
 logic p, g; //variables internas
 assign p = a ^ b;
 assign g = a & b;
 assign s = p ^ cin;
 assign cout = p | (p & cin);
endmodule
```

Síntesis

El circuito sintetizado muestra las relaciones entre las operaciones, señales de entrada, salida y variables internas.



Precedencia de operaciones

Op	Meaning
~	NOT
*, /, %	MUL, DIV, MOD
+, -	PLUS, MINUS
<<, >>	Logical Left/Right Shift
<<<, >>>	Arithmetic Left/Right Shift
<, <=, >, >=	Relative Comparison
==, !=	Equality Comparison
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
?:	Conditional

Declarando valores númericos

Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000 ... 0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00 ... 0101010

Valores de una señal

Un buffer de tres estados asigna el valor de la entrada a la salida si su señal de control `enable` está activa, en caso contrario la salida flota, esto quiere decir que queda en un estado de alta resistencia (impedancia), lo que en el sentido práctico equivale a que la compuerta se haya desconectado. En `System Verilog` un valor de `z` indica que una señal se encuentra en alta impedancia.

Valor desconocido

Hay situaciones en las que el valor de una señal no puede definirse y se considera como desconocida o de valor `x`. Esto se puede deber a:

Valor desconocido

Hay situaciones en las que el valor de una señal no puede definirse y se considera como desconocida o de valor `x`. Esto se puede deber a:

- **Valor no inicializado:** La simulación no puede definir el valor de una señal debido a que no se la ha inicializado.

Valor desconocido

Hay situaciones en las que el valor de una señal no puede definirse y se considera como desconocida o de valor `x`. Esto se puede deber a:

- **Valor no inicializado:** La simulación no puede definir el valor de una señal debido a que no se la ha inicializado.
- **Contención:** Si dos buffers de tres estados están conectados a un bus y sus valores difieren (uno alto y otro bajo), se indicará que la señal es `x`.

Valor desconocido

Hay situaciones en las que el valor de una señal no puede definirse y se considera como desconocida o de valor `x`. Esto se puede deber a:

- **Valor no inicializado:** La simulación no puede definir el valor de una señal debido a que no se la ha inicializado.
- **Contención:** Si dos buffers de tres estados están conectados a un bus y sus valores difieren (uno alto y otro bajo), se indicará que la señal es `x`.

La ocurrencia de una señal con valor `x` suele indicar la existencia de un error en el diseño.

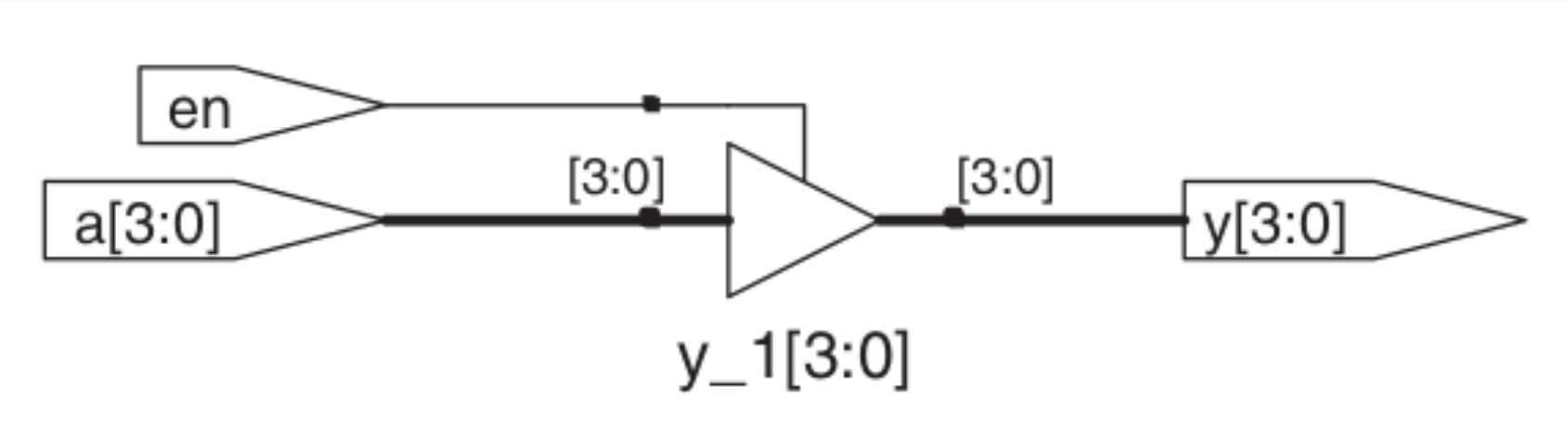
Variables internas

Un buffer de tres estados se declara con un condicional entre un valor de entrda y un estado de alta impedancia (z), nótese que la salida está declarada como `tri` ya que su valor puede depender de más de una fuente a la vez (buses):

System Verilog

```
module buffer3(input logic[3:0] a,
 input logic en,
 output tri[3:0] y);
 assign y = en ? a : 4'bz;
endmodule
```

El circuito sintetizado a partir de `buffer3`.



Manipulación de bits

La sintaxis de `System Verilog` provee varias herramientas para manipular bits y componer así nuevos valores, el operador {} permite concatenar bits, el operador [] permite acceder a una lista específica de bits, veamos el siguiente ejemplo:

System Verilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

Manipulación de bits

La sintaxis de `System Verilog` provee varias herramientas para manipular bits y componer así nuevos valores, el operador {} permite concatenar bits, el operador [] permite acceder a una lista específica de bits, veamos el siguiente ejemplo:

System Verilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

Aquí `c[2:1]` accede al segundo y tercer bit de la entrada `c`, `{3{d[0]}}` concatena tres copias del bit `d[0]` (el menos significativo de `d`) y `3'b101` agrega una constante de 3 bits a la composición, nótese que debemos definir el tamaño de la constante binaria en las composiciones.

Manipulación de bits

System Verilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

Manipulación de bits

System Verilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

Esto equivale a $y = c_2 c_1 d_0 d_0 d_0 c_0 101$.

Modelado estructural

Modelado estructural

Hasta este punto los diseños que se presentaron siguieron un enfoque de **modelado comportamental**, donde se especificó el comportamiento de las salidas en función de los valores de entrada y cómo operar en términos aritméticos y lógicos con ellos. Ahora vamos emplear el enfoque **estructural**, donde se realizará una composición de módulos previamente declarados para conseguir un componente más complejo.

Composición de multiplexores

El siguiente diseño implementa un multiplexor de 4 entradas componiendo 3 multiplexores de 2 entradas:

System Verilog

```
module mux4(input logic [3:0] d0, d1, d2, d3,
 input logic [1:0] s, output logic [3:0] y);
 logic [3:0] low, high;
 mux2 lowmux(d0, d1, s[0], low);
 mux2 highmux(d2, d3, s[0], high);
 mux2 finalmux(low, high, s[1], y);
endmodule
```

Composición de multiplexores

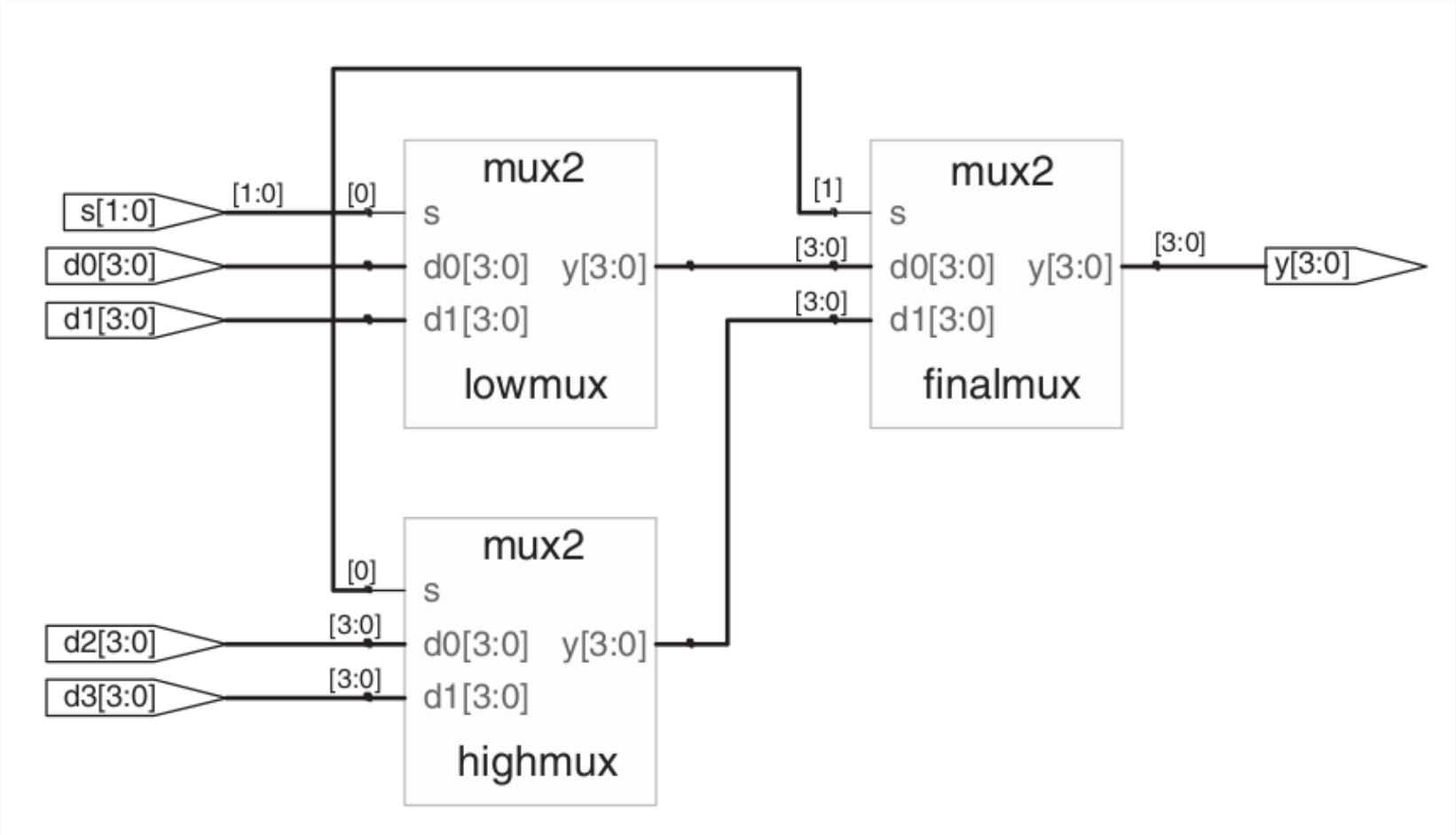
El siguiente diseño implementa un multiplexor de 4 entradas componiendo 3 multiplexores de 2 entradas:

System Verilog

```
module mux4(input logic [3:0] d0, d1, d2, d3,
 input logic [1:0] s, output logic [3:0] y);
 logic [3:0] low, high;
 mux2 lowmux(d0, d1, s[0], low);
 mux2 highmux(d2, d3, s[0], high);
 mux2 finalmux(low, high, s[1], y);
endmodule
```

Se hace uso en cascada de tres instancias del módulo `mux2`, la forma de instanciar un módulo por composición es con el nombre del mismo (`mux2`) seguido por el nombre de la instancia (`lowmux`) y finalmente los nombres de las señales que se corresponden con

El circuito sintetizado a partir de `mux4`.



Composición de multiplexores

El siguiente diseño implementa un multiplexor de 2 entradas componiendo 2 buffers tri-estado de 4 entradas:

System Verilog

```
module mux2(input logic [3:0] d0, d1,
 input logic s, output tri[3:0] y);
 tristate t0(d0, ~s, y);
 tristate t1(d1, s, y);
endmodule
```

Composición de multiplexores

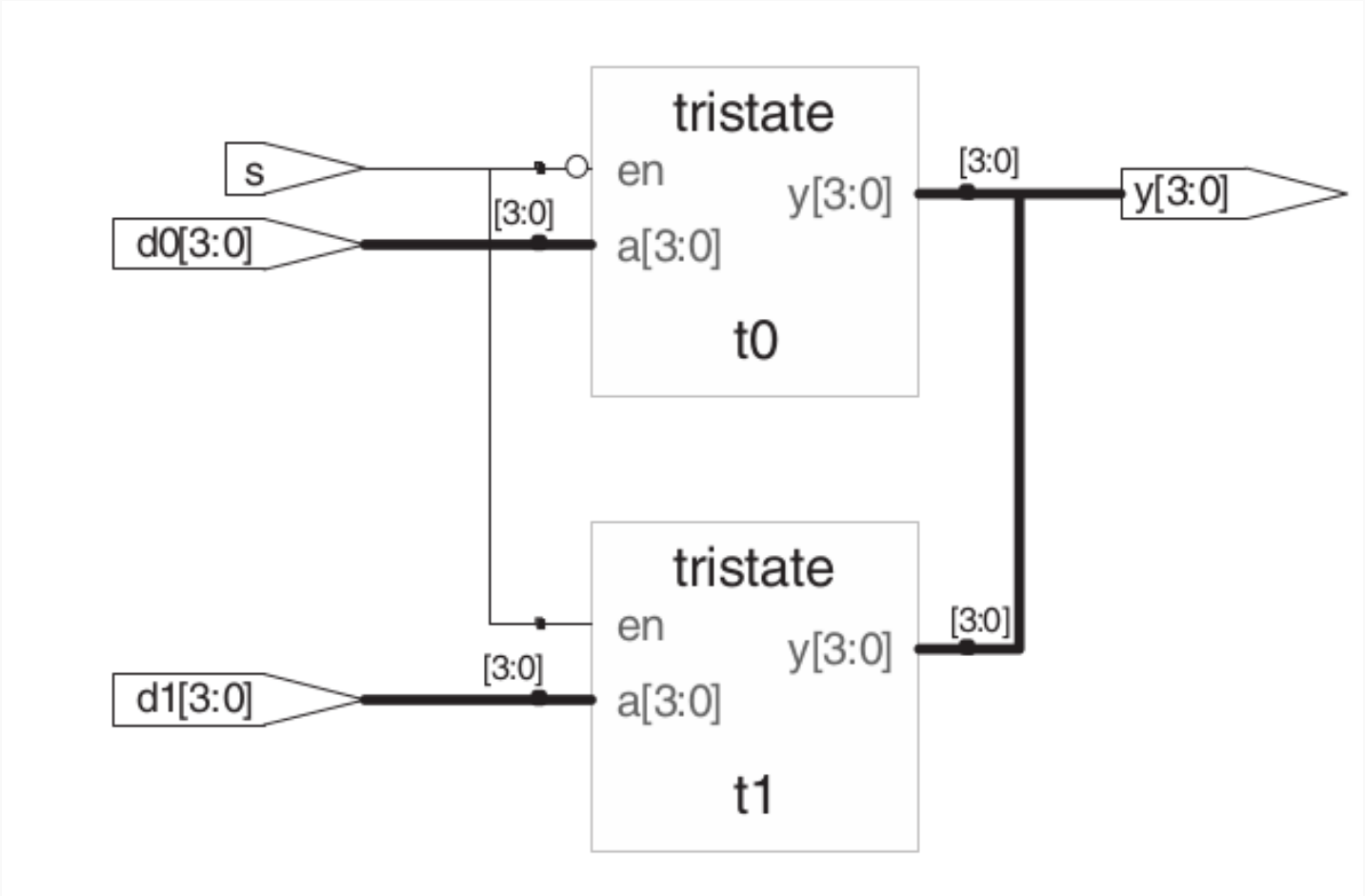
El siguiente diseño implementa un multiplexor de 2 entradas componiendo 2 buffers tri-estado de 4 entradas:

System Verilog

```
module mux2(input logic [3:0] d0, d1,
 input logic s, output tri[3:0] y);
 tristate t0(d0, ~s, y);
 tristate t1(d1, s, y);
endmodule
```

Utilizando el enfoque estructural, la implementación de un mismo comportamiento se puede conseguir como composición de distinto tipo de componentes (buffers vs. multiplexores).

El circuito sintetizado a partir de `mux4`.



Diseño jerárquico

En el modelado estructural, es habitual definir los componentes de forma jerárquica, comenzando por los componentes principales (`top level`) para luego ir describiendo la estructura de cada uno de ellos y de sus propios componentes a su vez.

Lógica secuencial

Bloques always

En los lenguajes de especificación de hardware los circuitos secuenciales suelen describirse con componentes de memoria y se emplean bloques `always` para indicar frente a qué evento se debe actualizar el estado de cuáles componentes. Estos bloques vienen acompañados de una lista de eventos (`sensitivity list`) frente a los que cobran efecto.

Registros

El siguiente diseño implementa un flip flop de 4 bits:

System Verilog

```
module flop(input logic clk,
 input logic[3:0] d, output logic[3:0] q);
 always_ff @(posedge clk)
  q <= d;
endmodule
```

Registros

El siguiente diseño implementa un flip flop de 4 bits:

System Verilog

```
module flop(input logic clk,
 input logic[3:0] d, output logic[3:0] q);
 always_ff @(posedge clk)
  q <= d;
endmodule
```

El bloque `always_ff` va a sintetizar flip flops, nótese que la asignación no se hace con un `assign` sino con un operador de asignación no bloqueante (<=). Veremos en breve los distintos tipos de bloques `always` y la diferencia entre asignacions bloqueantes y no bloqueantes.

Registros

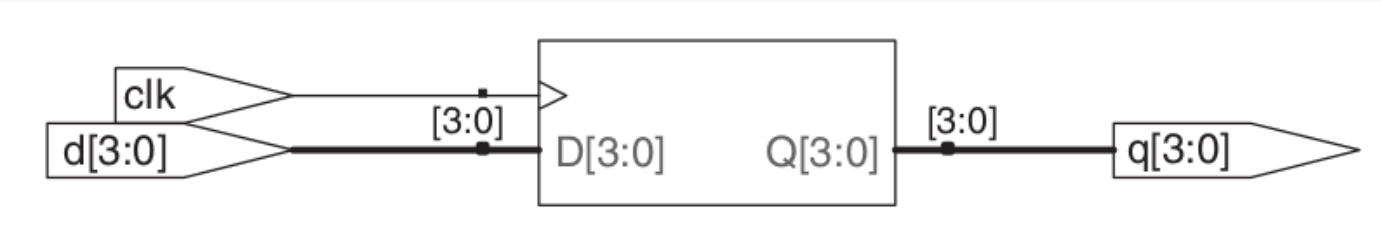
El siguiente diseño implementa un flip flop de 4 bits:

System Verilog

```
module flop(input logic clk,
 input logic[3:0] d, output logic[3:0] q);
 always_ff @(posedge clk)
  q <= d;
endmodule
```

En este caso, la asignación no bloqueante de `d` en `q` se hace cuando ocurre el evento `posedge clk`, lo que significa que se actualiza el estado en el flanco ascendente de la señal de reloj.

El circuito sintetizado a partir de `flop`.



Registros con reset

Al introducir una señal de `reset`, la misma puede ser sincrónica (cobra efecto frente al evento del flanco ascendente de reloj) o asincrónica (cobra efecto en cuanto cambia de valor la señal).

Reset sincrónico/asincrónico

System Verilog

```
module flopr(input logic clk, input logic reset,
 input logic[3:0] d, output logic[3:0] q);
 //reset asincronico
 always_ff@(posedge clk, posedge reset)
  if(reset) q <= 4'b0;
  else q <= d;
endmodule;

module flopr(input logic clk, input logic reset,
input logic[3:0] d, output logic[3:0] q);
 //reset sincronico
 always_ff@(posedge clk)
  if(reset) q <= 4'b0;
  else q <= d;
endmodule;
```

Registros con enable

System Verilog

```
module flopenr(input logic clk, input logic reset,
 input logic en,
 input logic[3:0] d, output logic[3:0] q);
 //reset asincronico
 always_ff@(posedge clk, posedge reset)
  if(reset) q <= 4'b0;
  else if(en) q <= d;
endmodule;
```

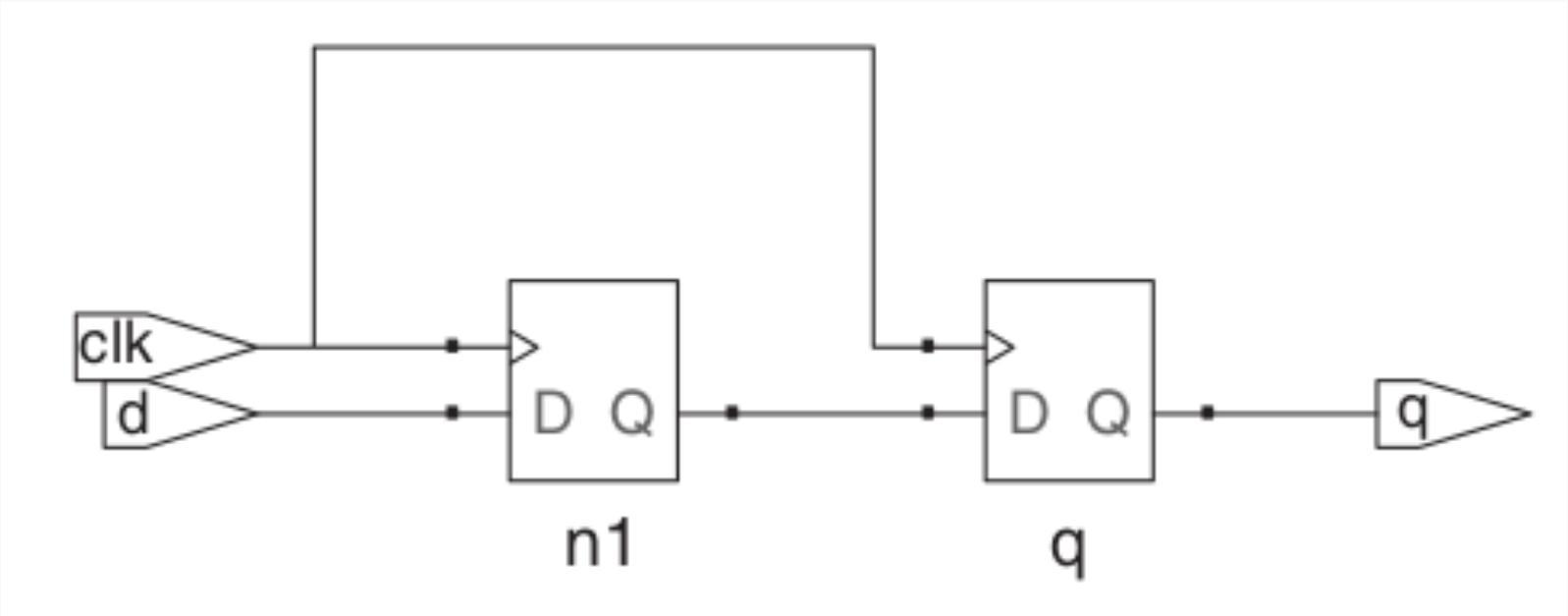
Sincronizador

Si el bloque `always` tiene más de una asignación se las debe agrupar en un par `begin end`.

System Verilog

```
module sync(input logic clk,
 input logic d,
 output logic q);
 //reset asincronico
 always_ff@(posedge clk)
  begin
   n1 <= d;
   q <= n1;
  end
endmodule;
```

El circuito sintetizado a partir de `sync`.



Lógica combinatoria 2

Always comb

System Verilog

```
module fulladder(input logic a, b, cin,
 output logic s, cout);
 logic p, g;
 always_comb
 begin
  p = a ^ b;
  g = a & b;
  s = p ^ cin;
  cout = g | (p & cin);
 end
endmodule
```

Always comb

System Verilog

```
module fulladder(input logic a, b, cin,
 output logic s, cout);
 logic p, g;
 always_comb
 begin
  p = a ^ b;
  g = a & b;
  s = p ^ cin;
  cout = g | (p & cin);
 end
endmodule
```

Los bloques `always comb` se evalúan siempre que una de las señales a la derecha de una asignación cambien de valor.

Always comb

System Verilog

```
always_comb
 begin
  p = a ^ b;
  g = a & b;
  s = p ^ cin;
  cout = g | (p & cin);
 end
```

Always comb

System Verilog

```
always_comb
 begin
  p = a ^ b;
  g = a & b;
  s = p ^ cin;
  cout = g | (p & cin);
 end
```

Nótese el uso de asignaciones bloqueantes (= en lugar de no bloqueantes (<=), esto obliga al simulador (no tiene efecto en la síntesis) a respetar secuencialemnte el orden de actualización de los valores a izquierda a medida que calcula el próximo estado, por ejemplo p se actualizará antes que s, que depende de la señal anterior.

Case


```
module sevenseg(input logic[3:0] data,
 output logic [6:0] segments);
 always_comb
  case(data)//abc_defg
   0: segments = 7'b111_1110;
   1: segments = 7'b011_0000;
   2: segments = 7'b110_1101;
   3: segments = 7'b111_1001;
   4: segments = 7'b011_0011;
   5: segments = 7'b101_1011;
   6: segments = 7'b101_1111;
   7: segments = 7'b111_0000;
   8: segments = 7'b111_1111;
   9: segments = 7'b111_0011;
   default: segments = 7'b000_0000;
  endcase
endmodule
```

Case


```
module sevenseg(input logic[3:0] data,
 output logic [6:0] segments);
 always_comb
  case(data)//abc_defg
   0: segments = 7'b111_1110;
   1: segments = 7'b011_0000;
   2: segments = 7'b110_1101;
   3: segments = 7'b111_1001;
   4: segments = 7'b011_0011;
   5: segments = 7'b101_1011;
   6: segments = 7'b101_1111;
   7: segments = 7'b111_0000;
   8: segments = 7'b111_1111;
   9: segments = 7'b111_0011;
   default: segments = 7'b000_0000;
  endcase
endmodule
```