Capítulo IV
Lenguajes de Descripción de Hardware (HDL)

Cristián Tejos

Segundo Semestre 2017

Basado en apuntes de Marcelo Guarini
y el libro
Digital Design and Computer Architecture (1st edition),
D. M. Harris & S. L. Harris, Elsevier 2007

Introducción

- En los años 90, se vio que los ingenieros eran mucho más productivos si trabajaban a un nivel de abstracción mayor, especificando sólo la función lógica y dejando que herramientas de CAD produjeran el circuito de compuertas optimizado.

- La especificación se realiza utilizando un lenguaje de descripción de hardware (HDL en inglés).

- Los dos lenguajes más populares son Verilog y VHDL.

- Construidos sobre el mismo principio pero con diferente sintaxis.

- Nuestro estudio se concentrará en Verilog.

Introducción

- Como ejemplo para ver la diferente forma de sintaxis entre Verilog y VHDL consideremos la descripción de un circuito que realiza la función

$$y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$$

Verilog

```
module sillyfunction (input a, b, c,
                      output y);

  assign y = ~a & ~b & ~c |
             a & ~b & ~c |
             a & ~b &  c;

endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sillyfunction is
  port (a, b, c: in  STD_LOGIC;
        y:       out STD_LOGIC);
end;

architecture synth of sillyfunction is
begin
  y <= ((not a) and (not b) and (not c)) or
       (a and (not b) and (not c)) or
       (a and (not b) and c);
end;
```

Introducción

- Origen del Verilog

Fue desarrollado por Gateway Design Automation como un lenguaje propio para simulación lógica en 1984.

Gateway fue adquirida por Cadence en 1989 y Verilog fue convertido en un estándar abierto en 1990 bajo el control de Open Verilog International

El lenguaje se transformó en un estándar IEEE en 1995 (IEEE STD 1364) y en el 2001 fue actualizado.

Introducción

Simulación y Síntesis

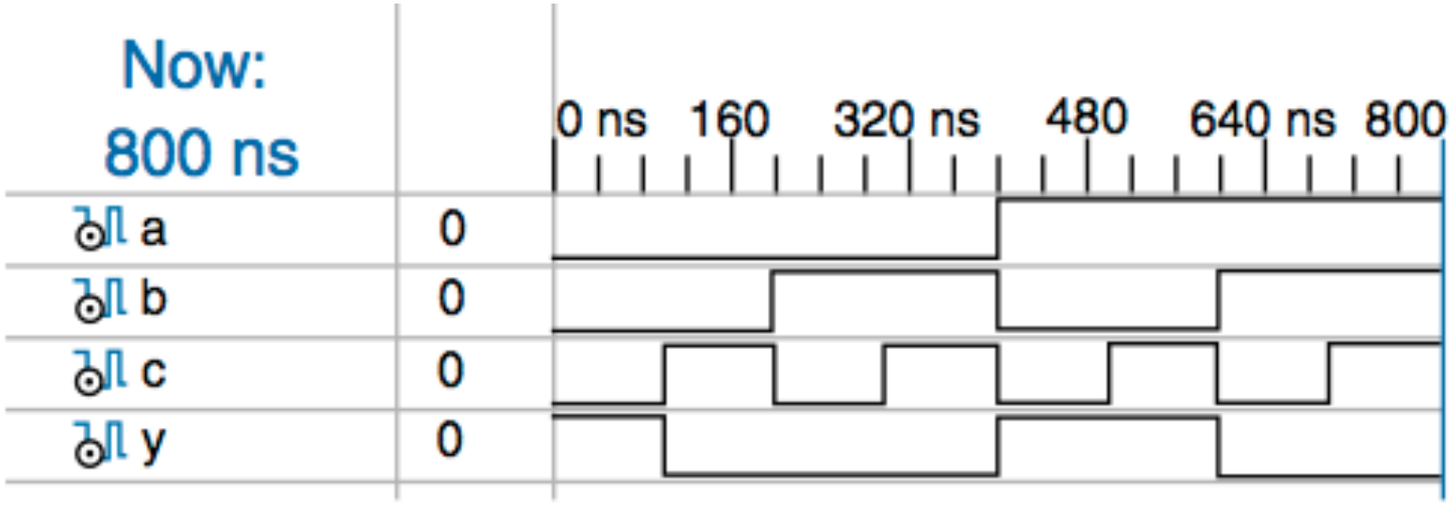
- Los dos mayores propósitos de los lenguajes HDL son la simulación de circuitos y la síntesis.
- Simulación: comprobar el funcionamiento de circuitos antes de producirlos.
- Síntesis: transformar del código HDL en una "*netlist*" que describe el hardware (compuertas lógicas y los alambres que las interconectan).

Introducción

Ejemplo de Simulación y Síntesis del circuito `sillyfunction`

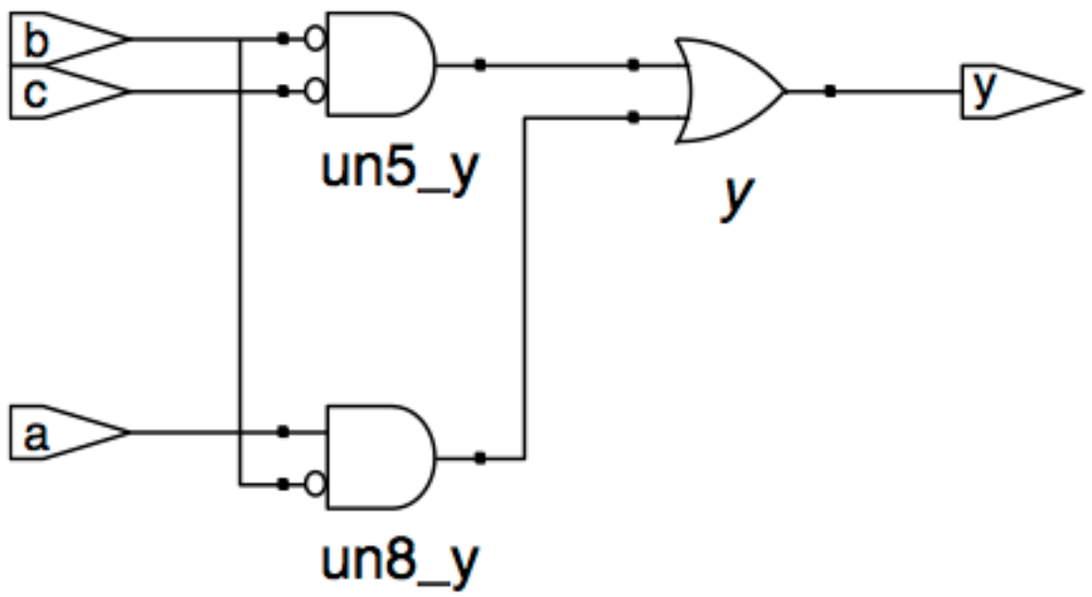
Simulación

Realizada con el simulador ISE de Xilinx



Síntesis

Realizada con Synplify Pro de Synplicity



Introducción

Simulación

- Es importante eliminar los *buggs* de diseño de un sistema digital, especialmente cuando los clientes pagan y cuando hay vidas que dependen de la operación correcta del chip.
- Descubrir fallas en un chip en el laboratorio es prácticamente imposible ya que sólo hay acceso a las señales en los *pines* de entrada y de salida.
- Corregir errores después de construir un chip, es extremadamente costoso.
- El problema en la división en la unidad de punto flotante de INTEL les costó US $475.000.000.

Introducción

One of the most common mistakes for beginners is to think of HDL as a computer program rather than as a shorthand for describing digital hardware. If you don't know approximately what hardware your HDL should synthesize into, you probably won't like what you get. You might create far more hardware than is necessary, or you might write code that simulates correctly but cannot be implemented in hardware. Instead, think of your system in terms of blocks of combinational logic, registers, and finite state machines. Sketch these blocks on paper and show how they are connected before you start writing code.

Modelación de Comportamiento (*Behavioral Modelling*)

- En esta sección describiremos cómo escribir modelos de comportamiento de circuitos combinacionales utilizando VERILOG

Lógica Combinacional

(Behavioral modelling)

- Operaciones bit a bit (bitwise operators)

Ejemplo:
INVERSORES

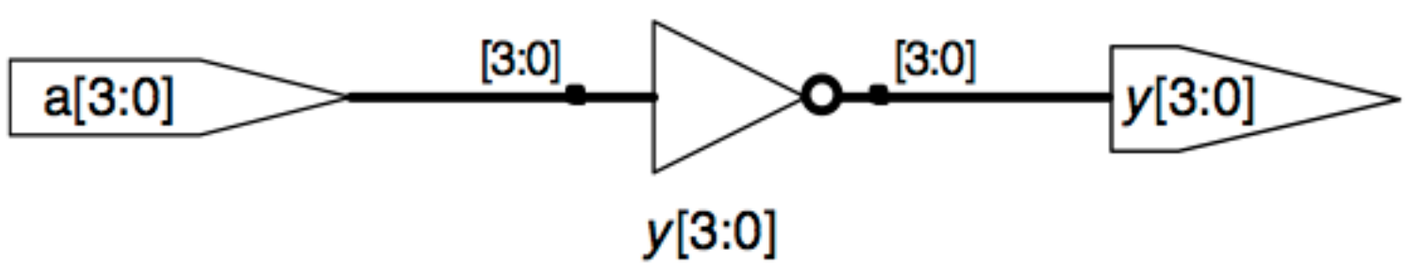
Verilog

```
module inv (input  [3:0] a,
            output [3:0] y);

  assign y = ~a;
endmodule
```

`a[3:0]` represents a 4-bit bus. The bits, from most significant to least significant, are `a[3]`, `a[2]`, `a[1]`, and `a[0]`. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have named the bus `a[4:1]`, in which case `a[4]` would have been the most significant. Or we could have used `a[0:3]`, in which case the bits, from most significant to least significant, would be `a[0]`, `a[1]`, `a[2]`, and `a[3]`. This is called *big-endian* order.

Circuito `inv` sintetizado



Lógica Combinacional

(Behavioral modelling)

- Operaciones bit a bit (bitwise operators)

Ejemplo: COMPUERTAS LÓGICAS

Verilog

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

  /* Five different two-input logic
     gates acting on 4 bit busses */
  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
```

~, ^, and | are examples of Verilog *operators*, whereas `a`, `b`, and `y1` are *operands*. A combination of operators and operands, such as `a & b`, or `~(a | b)`, is called an *expression*. A complete command such as `assign y4 = ~(a & b);` is called a *statement*.

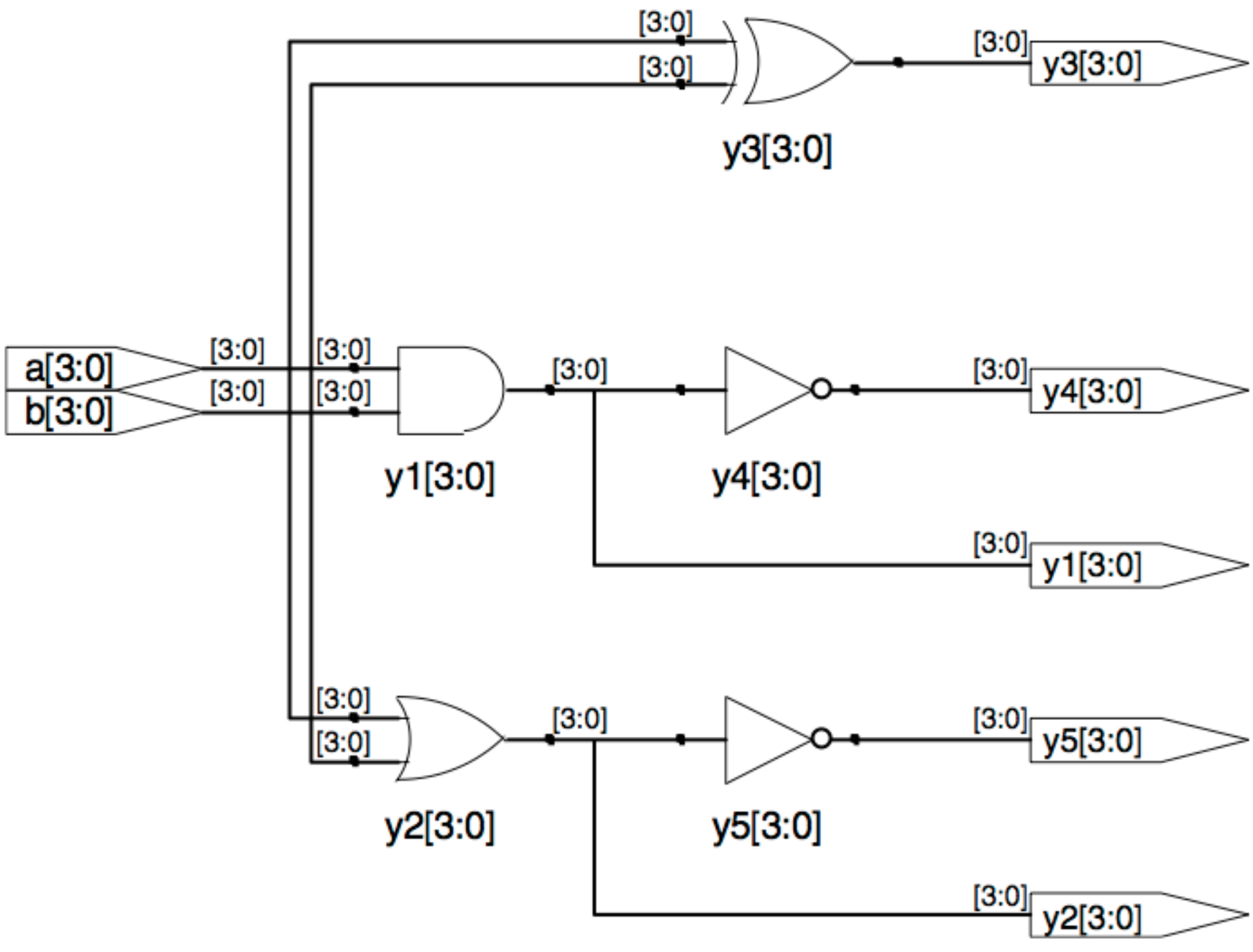
`assign out = in1 op in2;` is called a *continuous assignment statement*. Continuous assignment statements end with a semicolon. Anytime the inputs on the right side of the = in a continuous assignment statement change, the output on the left side is recomputed. Thus, continuous assignment statements describe combinational logic.

Lógica Combinacional

(Behavioral modelling)

- Operaciones bit a bit (bitwise operators)

Circuito `gates` sintetizado



Lógica Combinacional

(Behavioral modelling)

- Comentarios y espacios en blanco

VERILOG no es sensible a los espacios en blanco, tabs y line breaks, sin embargo, una buena indentación y uso de líneas en blanco ayudan a leer diseños complejos

Verilog

Verilog comments are just like those in C or Java. Comments beginning with /* continue, possibly across multiple lines, to the next */. Comments beginning with // continue to the end of the line.

Verilog is case-sensitive. `y1` and `Y1` are different signals in Verilog.

Lógica Combinacional

(Behavioral modelling)

- Operadores de reducción

Ejemplo:
Compuerta AND de
8 entradas

Verilog

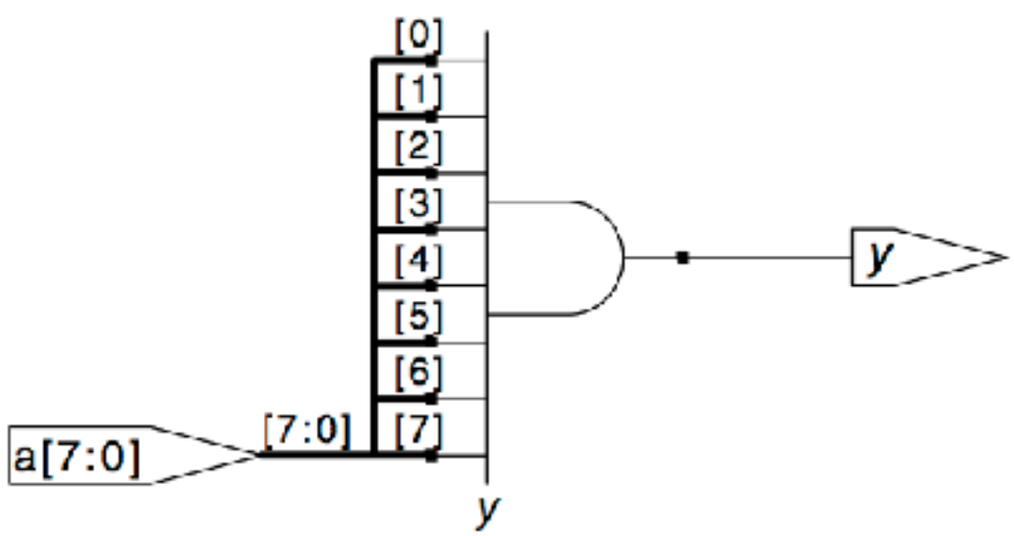
```
module and8 (input [7:0] a,
             output      y);

  assign y = &a;

  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

As one would expect, |, ^, ~&, and ~ | reduction operators are available for OR, XOR, NAND, and NOR as well. Recall that a multi-input XOR performs parity, returning TRUE if an odd number of inputs are TRUE.

Circuito `and8`
sintetizado



Lógica Combinacional

(Behavioral modelling)

- Asignación condicional

Ejemplo:
Multiplexor 2:1

Verilog

The *conditional operator* ?: chooses, based on a first expression, between a second and third expression. The first expression is called the *condition*. If the condition is 1, the operator chooses the second expression. If the condition is 0, the operator chooses the third expression.

?: is especially useful for describing a multiplexer because, based on the first input, it selects between two others. The following code demonstrates the idiom for a 2:1 multiplexer with 4-bit inputs and outputs using the conditional operator.

```
module mux2 (input  [3:0] d0, d1,
             input        s,
             output [3:0] y);

   assign y = s ? d1 : d0;
endmodule
```

If `s` is 1, then `y` = `d1`. If `s` is 0, then `y` = `d0`.

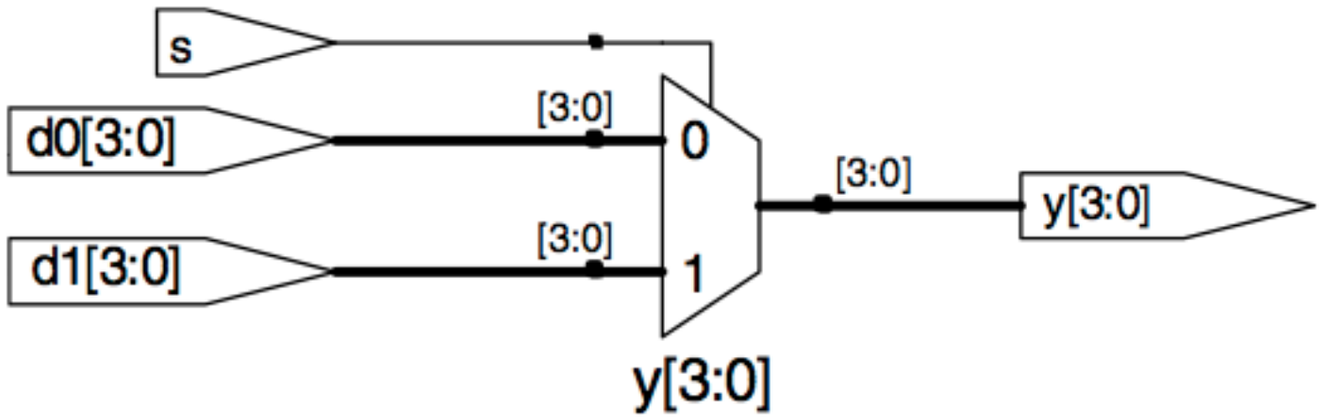
?: is also called a *ternary operator*, because it takes three inputs. It is used for the same purpose in the C and Java programming languages.

Lógica Combinacional

(Behavioral modelling)

- Asignación condicional

Circuito `mux2` sintetizado



Lógica Combinacional

(Behavioral modelling)

- Asignación condicional

Ejemplo:
Multiplexor 4:1

Verilog

A 4:1 multiplexer can select one of four inputs using nested conditional operators.

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

  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```

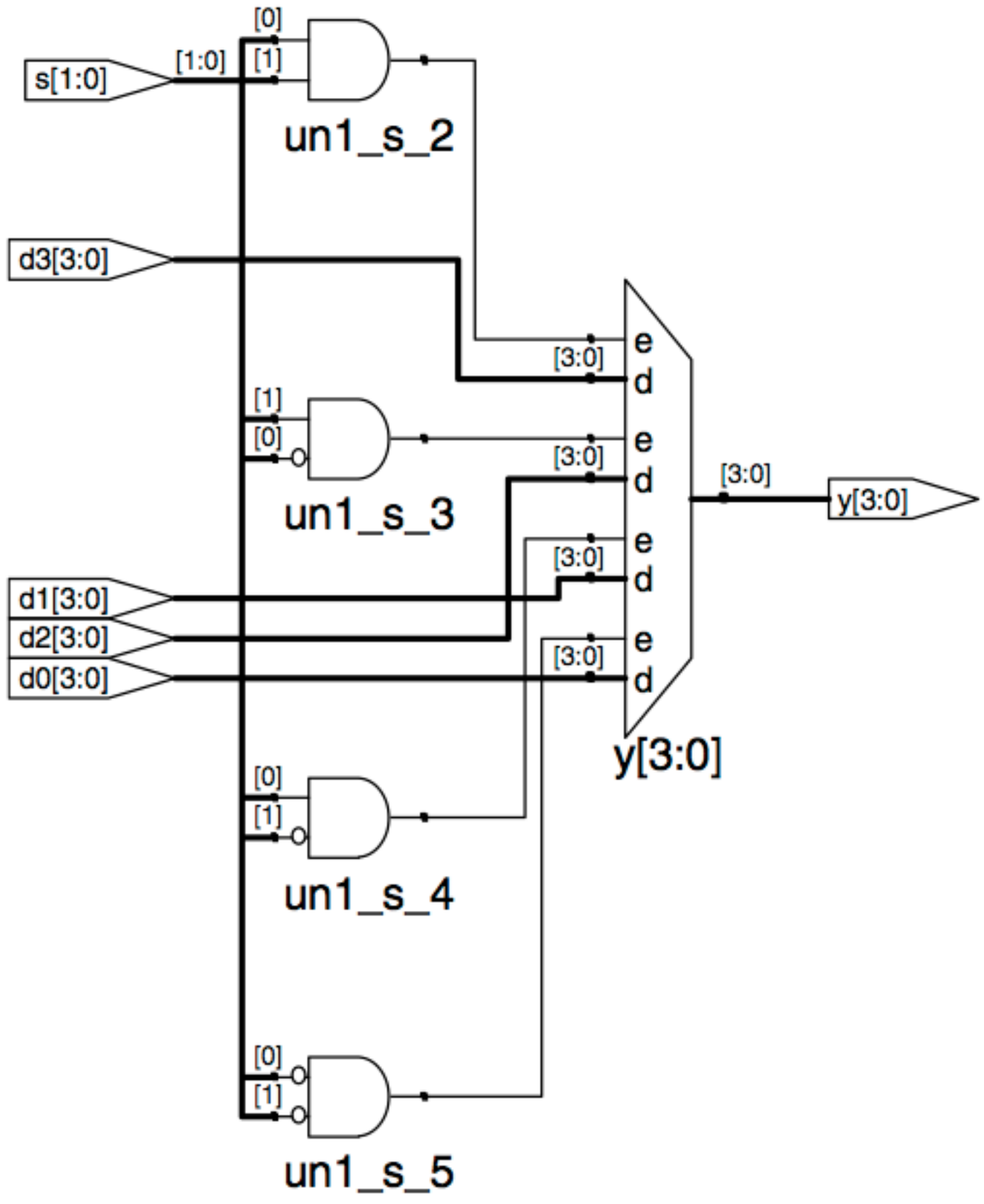
If `s[1]` is 1, then the multiplexer chooses the first expression, `(s[0] ? d3 : d2)`. This expression in turn chooses either `d3` or `d2` based on `s[0]` (`y = d3` if `s[0]` is 1 and `d2` if `s[0]` is 0). If `s[1]` is 0, then the multiplexer similarly chooses the second expression, which gives either `d1` or `d0` based on `s[0]`.

Lógica Combinacional

(Behavioral modelling)

- Asignación condicional

Circuito `mux4` sintetizado



Lógica Combinacional

(Behavioral modelling)

- Variables internas

Ejemplo:
FULL ADDER

Verilog

In Verilog, *wires* are used to represent internal variables whose values are defined by `assign` statements such as `assign p = a ^ b;` Wires technically have to be declared only for multibit busses, but it is good practice to include them for all internal variables; their declaration could have been omitted in this example.

```
module fulladder(input  a, b, cin,
                 output s, cout);

  wire p, g;

  assign p = a ^ b;
  assign g = a & b;

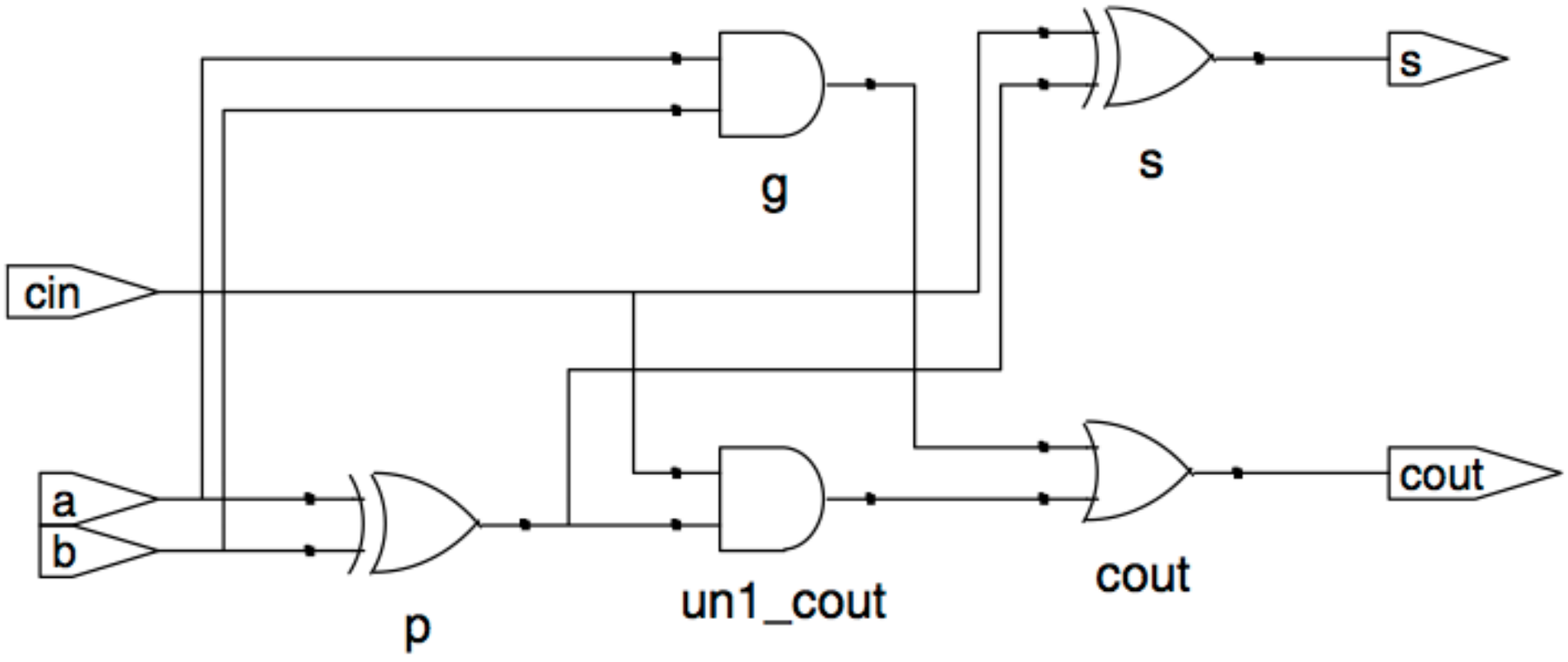
  assign s = p ^ cin;
  assign cout = g  |  (p & cin);
endmodule
```

Lógica Combinacional

(Behavioral modelling)

- Variables Internas

Circuito
`fulladder`
sintetizado



Lógica Combinacional

(Behavioral modelling)

- Precedencia de operadores

TABLA DE PRECEDENCIA

Table 4.1 Verilog operator precedence

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

The operator precedence for Verilog is much like you would expect in other programming languages. In particular, AND has precedence over OR. We could take advantage of this precedence to eliminate the parentheses.

```
assign cout = g  |  p & cin;
```

Lógica Combinacional

(Behavioral modelling)

- Números

NÚMEROS EN VERILOG

Verilog

Verilog numbers can specify their base and size (the number of bits used to represent them). The format for declaring constants is `N'Bvalue`, where `N` is the size in bits, `B` is the base, and `value` gives the value. For example `9'h25` indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. Verilog supports `'b` for binary (base 2), `'o` for octal (base 8), `'d` for decimal (base 10), and `'h` for hexadecimal (base 16). If the base is omitted, the base defaults to decimal.

If the size is not given, the number is assumed to have as many bits as the expression in which it is being used. Zeros are automatically padded on the front of the number to bring it up to full size. For example, if `w` is a 6-bit bus, `assign w = 'b11` gives `w` the value 000011. It is better practice to explicitly give the size.

Table 4.3 Verilog numbers

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

Lógica Combinacional

(Behavioral modelling)

- Z's y X's

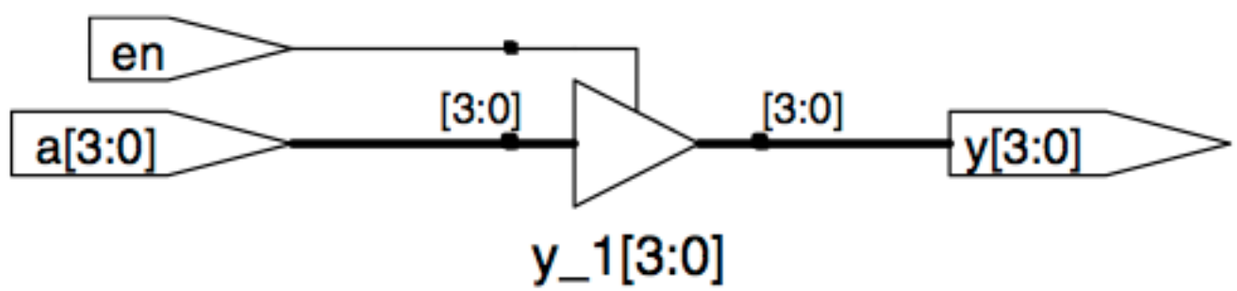
Ejemplo:
BUFFER con control de tercer estado

Verilog

```
module tristate(input  [3:0] a,
                input        en,
                output [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

Circuito `tristate` sintetizado



Lógica Combinacional

(Behavioral modelling)

- Z's y X's

En VERILOG x se utiliza para indicar un nivel lógico inválido. Si una línea de un *bus* es simultáneamente llevada a 0 y a 1 por dos *buffers tristate* habilitados el resultado es x, indicando contención. Si todos los *buffers tristate* que alimentan una línea de un *bus* están deshabilitados el resultado es z.

Al comienzo de una simulación los estados de los *flip-flops* se inicializan en un estado desconocido, x en el caso de VERILOG.

Lógica Combinacional

(Behavioral modelling)

- Z's y X's

Ejemplo:
Tabla de la función AND en VERILOG

Verilog

Verilog signal values are `0`, `1`, `z`, and `x`. Verilog constants starting with `z` or `x` are padded with leading z's or x's (instead of 0's) to reach their full length when necessary.

Table 4.5 shows a truth table for an AND gate using all four possible signal values. Note that the gate can sometimes determine the output despite some inputs being unknown. For example `0 & z` returns `0` because the output of an AND gate is always 0 if either input is 0. Otherwise, floating or invalid inputs cause invalid outputs, displayed as `x` in Verilog.

Table 4.5 Verilog AND gate truth table with z and x

&		*A*			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

Lógica Combinacional

(Behavioral modelling)

- Concatenación de bits

Verilog

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

The {} operator is used to concatenate busses. {3{d[0]}} indicates three copies of d[0].

Don't confuse the 3-bit binary constant 3'b101 with a bus named b. Note that it was critical to specify the length of 3 bits in the constant; otherwise, it would have had an unknown number of leading zeros that might appear in the middle of y.

If y were wider than 9 bits, zeros would be placed in the most significant bits.

Lógic

- Retardos

Ejemplo:
COMPUERTAS LÓGICAS CON RETARDO

Verilog

```
‘timescale 1ns/1ps

module example(input  a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~ {a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

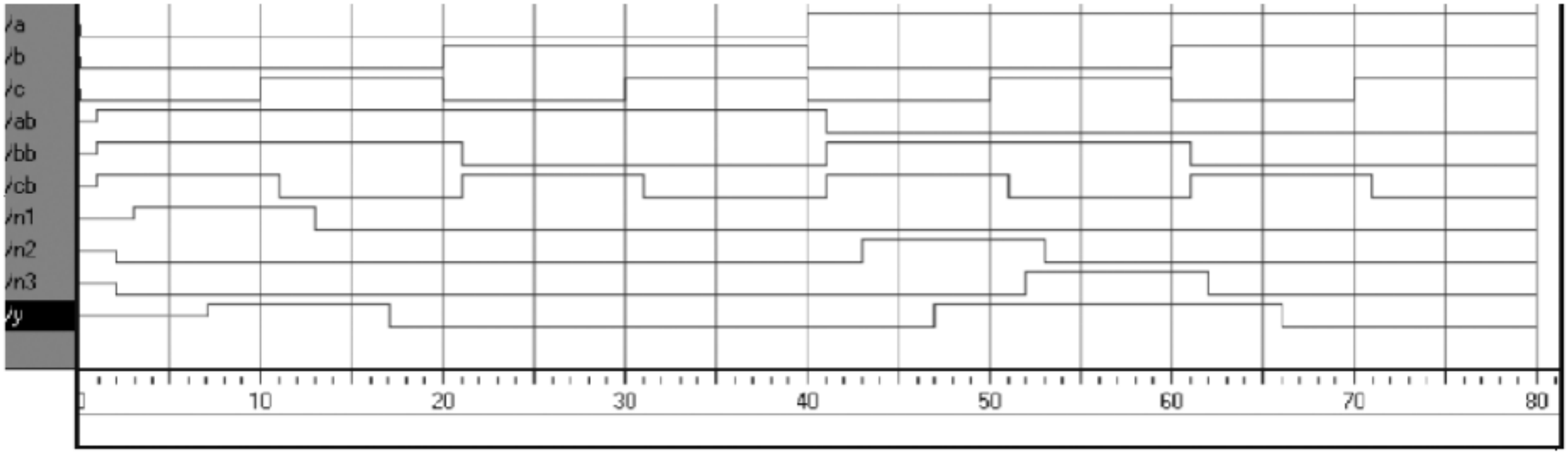
Verilog files can include a timescale directive that indicates the value of each time unit. The statement is of the form `‘timescale unit/precision`. In this file, each unit is 1 ns, and the simulation has 1 ps precision. If no timescale directive is given in the file, a default unit and precision (usually 1 ns for both) is used. In Verilog, a # symbol is used to indicate the number of units of delay. It can be placed in `assign` statements, as well as non-blocking (<=) and blocking (=) assignments, which will be discussed in Section 4.5.4.

Lógica Combinacional

(Behavioral modelling)

- Retardos

Simulación de
módulo `example`



Modelación Estructural
(*Structural Modelling*)

- En la sección que acabamos de ver utilizamos modelación de comportamiento, describiendo módulos en términos de las relaciones entre entradas y salidas.
- En esta sección examinaremos la modelación estructural, describiendo los módulos en términos de la composición de módulos más simples.

Lógica Combinacional

(Structural modelling)

Ejemplo: MODELO ESTRUCTURAL DE UN MULTIPLEXOR 4:1

Verilog

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

  wire [3:0] low, high;

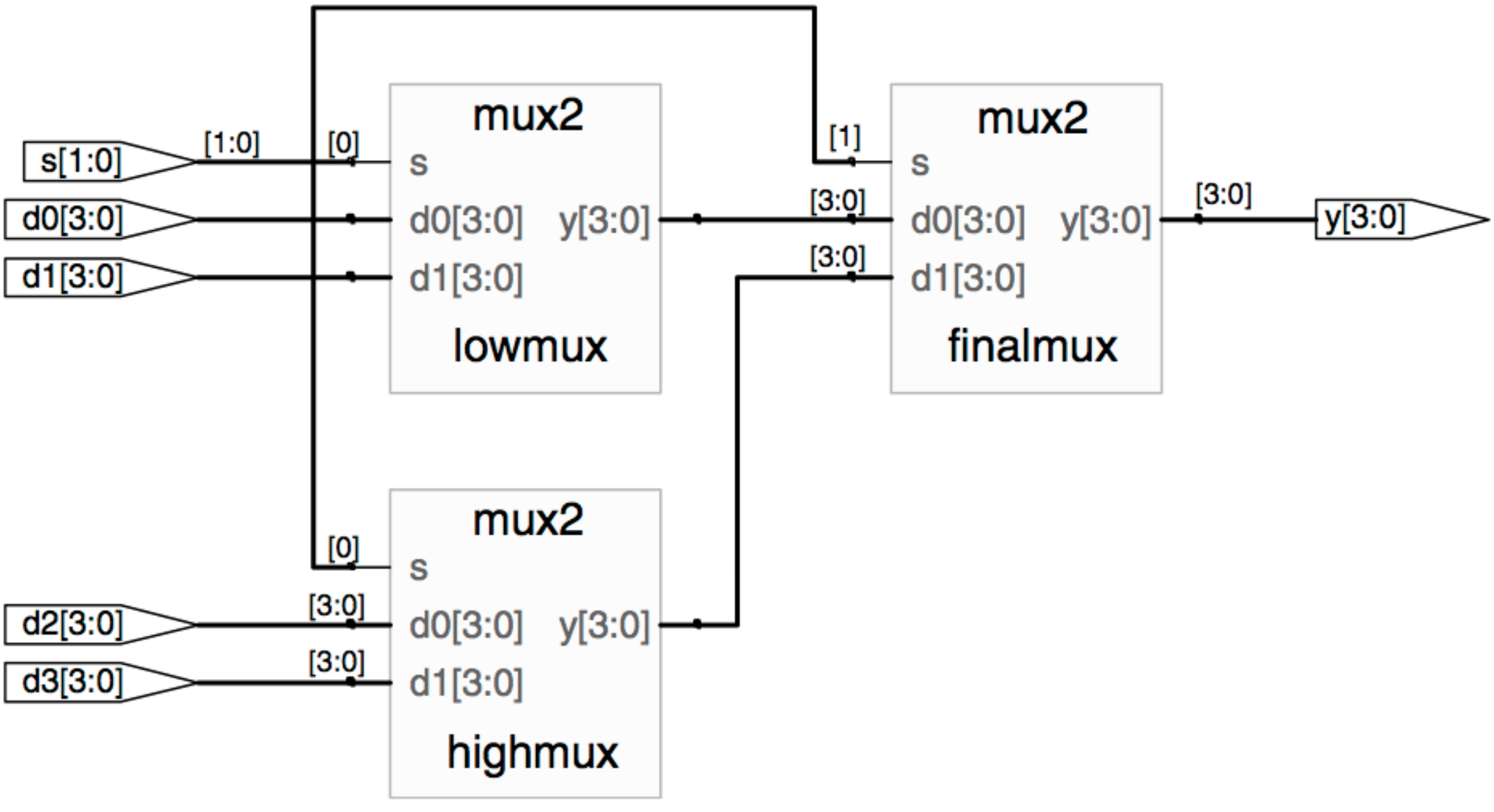
  mux2 lowmux (d0, d1, s[0], low);
  mux2 highmux (d2, d3, s[0], high);
  mux2 finalmux (low, high, s[1], y);
endmodule
```

The three `mux2` instances are called `lowmux`, `highmux`, and `finalmux`. The `mux2` module must be defined elsewhere in the Verilog code.

Lógica Combinacional

(Structural modelling)

Circuito `mux4` sintetizado



Lógica Combinacional

(Structural modelling)

- A cada copia del multiplexor de 2:1 se le llama instancia.
- A las múltiples instancias de un mismo módulo se les distingue por un único nombre. En el ejemplo que vimos: `lowmux`, `highmux` y `finalmux`.
- El ejemplo es un caso de regularidad, en el cual el multiplexor de 2:1 es reutilizado muchas veces.

Lógica Combinacional

(Structural modelling)

Ejemplo: MODELO ESTRUCTURAL DE UN MULTIPLEXOR 2:1

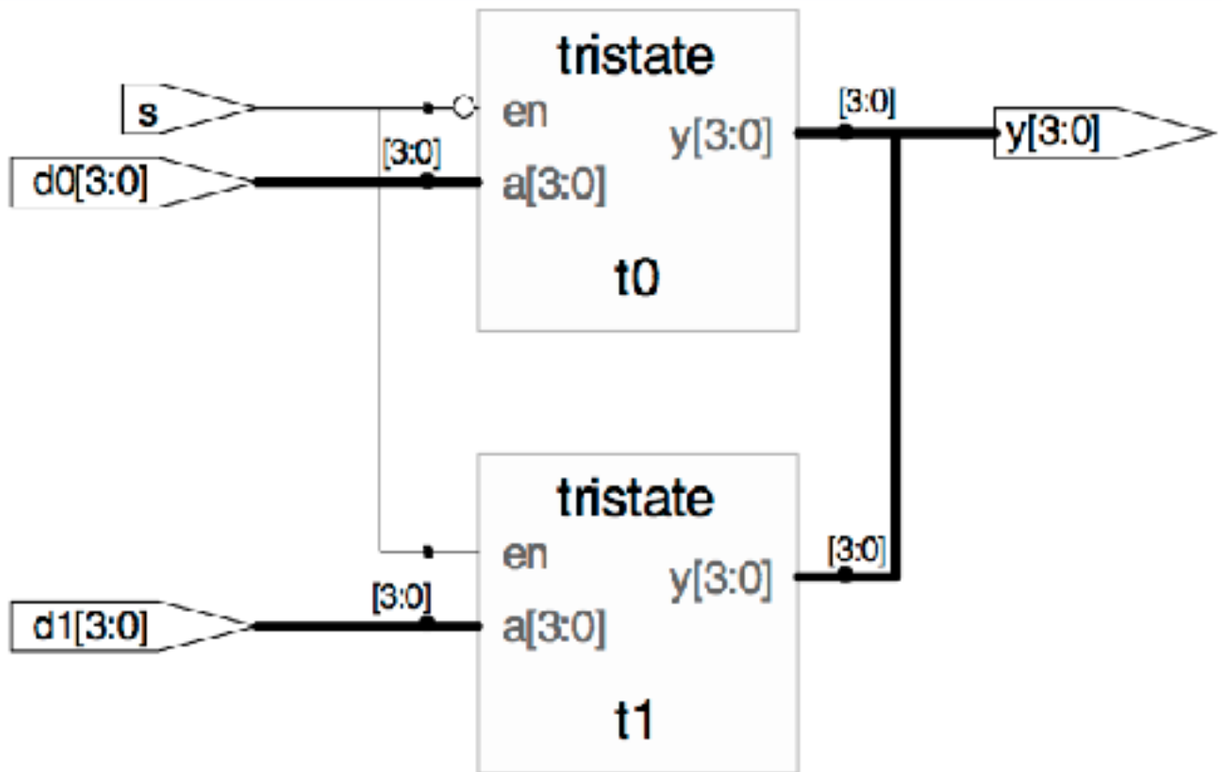
Verilog

```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

In Verilog, expressions such as ~s are permitted in the port list for an instance. Arbitrarily complicated expressions are legal but discouraged because they make the code difficult to read.

Circuito `mux2` sintetizado



Lógica Combinacional

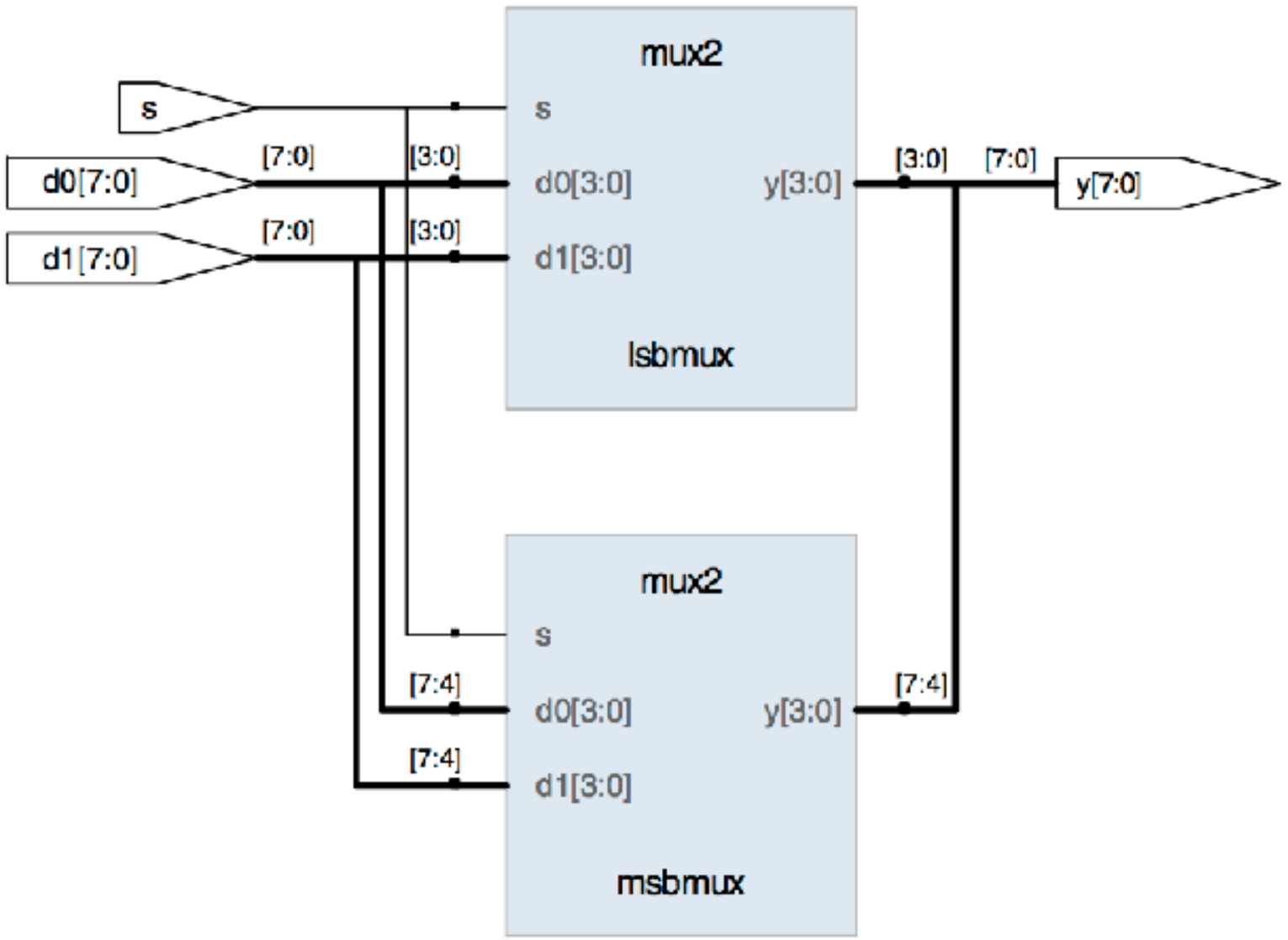
(Structural modelling)

Ejemplo:
Acceso a parte de un BUS. Mux 2:1 de 8 bits de ancho utilizando dos de los multiplexores de 2:1 de cuatro bits que vimos, operando en los 4 bits superiores y 4 bits inferiores del byte.

```
module mux2_8(input  [7:0] d0, d1,
              input        s,
              output [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

Circuito `mux2_8` sintetizado



Lógica Secuencial

- Los sintetizadores HDL reconocen ciertos términos y los convierten en circuitos secuenciales específicos.
- Otros estilos de código pueden simular correctamente pero sintetizar en circuitos con errores evidentes y en algunos casos errores sutiles que en general suelen ser difíciles de identificar.
- Esta sección presenta las formas apropiadas para describir registros y latches.

Lógica Secuencial

Registros

- La gran mayoría de los sistemas comerciales modernos son construidos utilizando *flip-flops* D gatillados por flanco positivo.
- En VERILOG la declaración `always` significa mantener el antiguo valor hasta que ocurra un evento en la lista de sensibilidad.
- Tal código, con una lista de sensibilidades apropiadas puede utilizarse para describir circuitos secuenciales con memoria. (Ej.: el *flip-flop* requiere sólo `clk` en la lista de sensibilidades)
- En contraste, la asignación continua `assign` es reevaluada cada vez que cualquiera de las entradas (al lado derecho del

Lógica Secuencial

REGISTROS

Ejemplo:
REGISTRO CON *FLIP-FLOPs* D

Verilog

```
module flop (input              clk,
             input       [3:0] d,
             output reg [3:0] q);

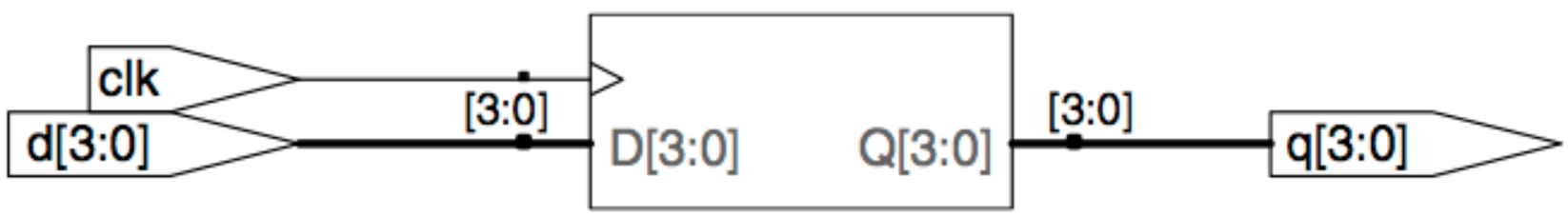
  always @ (posedge clk)
    q <= d;
endmodule
```

A Verilog `always` statement is written in the form

```
always @ (sensitivity list)
  statement;
```

The `statement` is executed only when the event specified in the `sensitivity list` occurs. In this example, the statement is `q <= d` (pronounced "q gets d"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`.

Circuito `flop` sintetizado



Lógica Secuencial

<= is called a *nonblocking assignment*. Think of it as a regular = sign for now; we'll return to the more subtle points in Section 4.5.4. Note that <= is used instead of `assign` inside an `always` statement.

All signals on the left hand side of <= or = in an `always` statement must be declared as `reg`. In this example, `q` is both an output and a `reg`, so it is declared as `output reg [3:0] q`. Declaring a signal as `reg` does not mean the signal is actually the output of a register! All it means is that the signal appears on the left hand side of an assignment in an `always` statement. We will see later examples of `always` statements describing combinational logic in which the output is declared `reg` but does not come from a flip-flop.

Lógica Secuencial

REGISTROS

Ejemplo:
REGISTRO
CON RESET

Verilog

```
module flopr (input            clk,
              input            reset,
              input      [3:0] d,
              output reg [3:0] q);

  // asynchronous reset
  always @ (posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule

module flopr (input            clk,
              input            reset,
              input      [3:0] d,
              output reg [3:0] q);

  // synchronous reset
  always @ (posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule
```

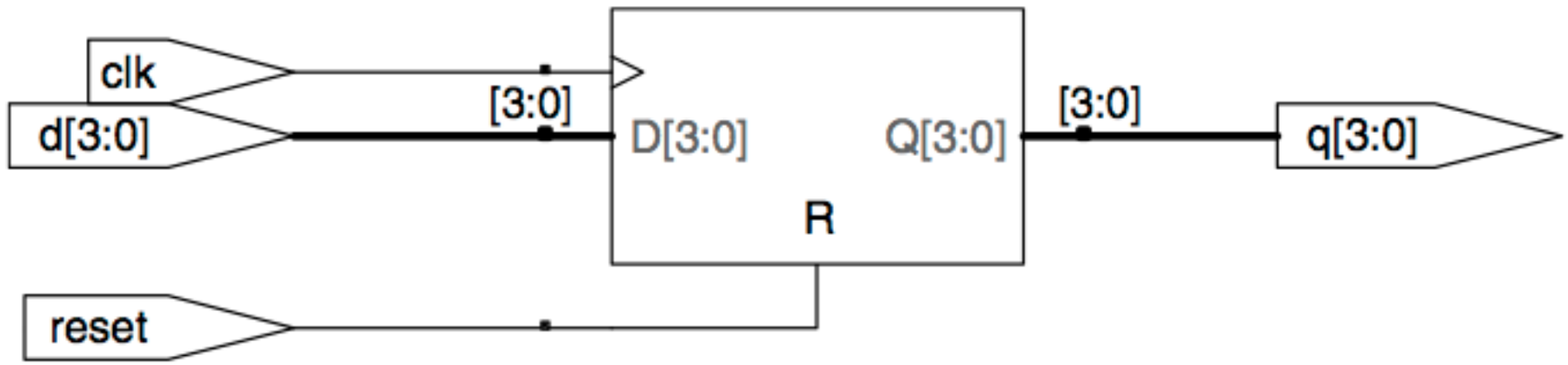
Multiple signals in an `always` statement sensitivity list are separated with a comma or the word `or`. Notice that `posedge reset` is in the sensitivity list on the asynchronously resettable flop, but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

Because the modules have the same name, `flopr`, you may include only one or the other in your design.

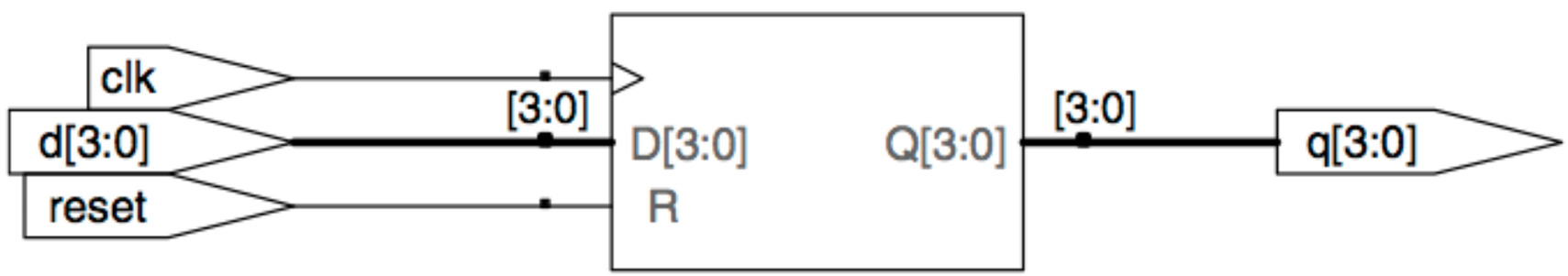
Lógica Secuencial

REGISTROS

Circuito `flopr` (con reset asíncrono) sintetizado



Circuito `flopr` (con reset síncrono) sintetizado



Lógica Secuencial

REGISTROS

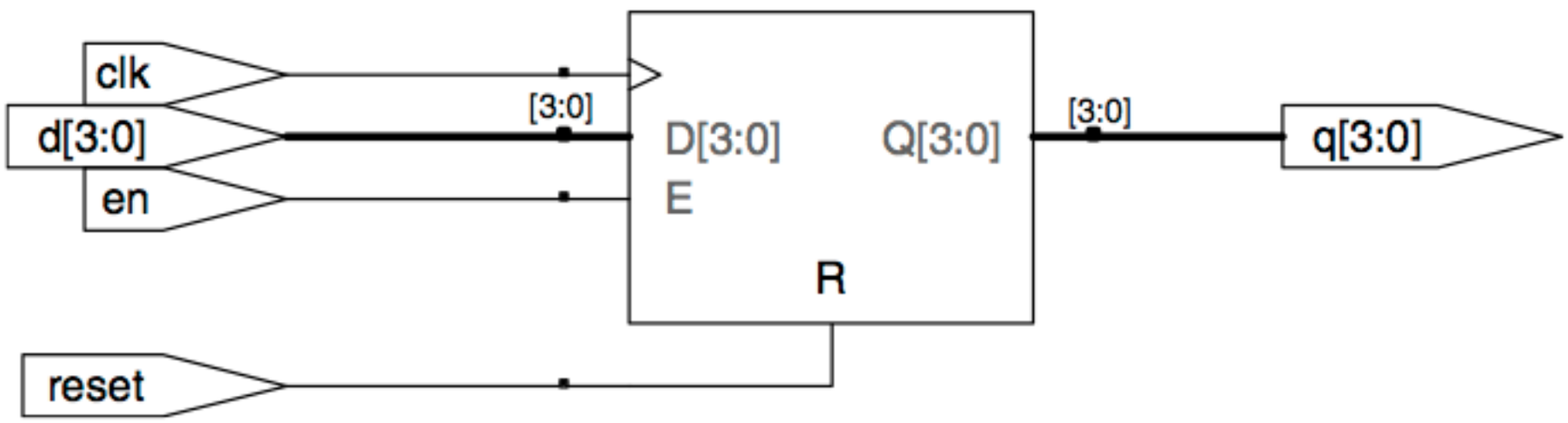
Ejemplo:
Registros con habilitación.

Verilog

```
module flopenr (input            clk,
                input            reset,
                input            en,
                input      [3:0] d,
                output reg [3:0] q);

  // asynchronous reset
  always @ (posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else if (en) q <= d;
endmodule
```

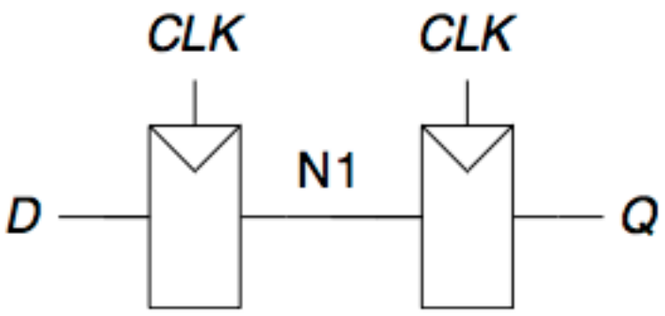
Circuito `flopenr` sintetizado



Lógica Secuencial

REGISTROS MÚLTIPLES

Ejemplo:
circuito sincronizador



Verilog

```
module sync(input      clk,
            input      d,
            output reg q);

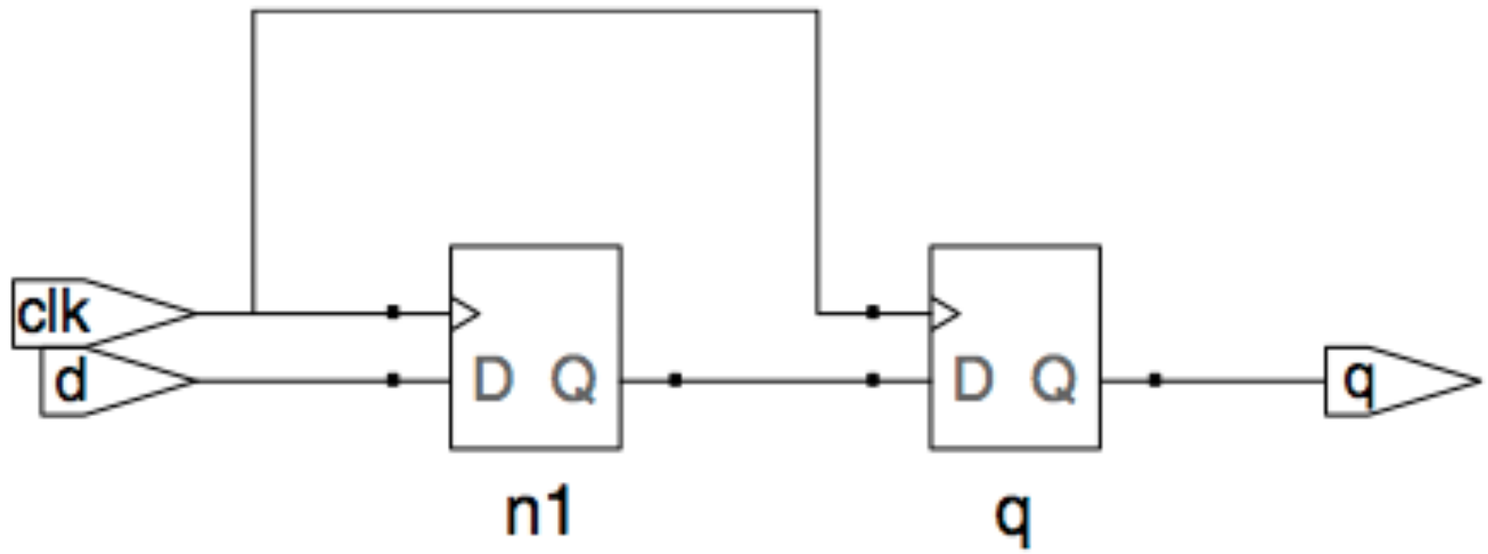
  reg n1;

  always @ (posedge clk)
    begin
      n1 <= d;
      q <= n1;
    end
endmodule
```

n1 must be declared as a reg because it is an internal signal used on the left hand side of <= in an always statement. Also notice that the begin/end construct is necessary because multiple statements appear in the always statement. This is analogous to { } in C or Java. The begin/end was not needed in the flopr example because if/else counts as a single statement.

Lógica Secuencial

Circuito `sync` sintetizado



Lógica Secuencial

LATCHES

- Recordemos que un *latch* tipo D es "transparente" cuando el *clock* es HIGH, permitiendo que los datos fluyan desde la entrada hasta la salida.
- El *latch* se hace "opaco" cuando el *clock* es LOW, reteniendo el último estado.
- No todas las herramientas de síntesis soportan bien los *latches*.
- A menos que Ud. sepa bien como su sintetizador soporta los *latches*, y si tiene una muy buena razón para utilizarlos, trate de evitarlos.

Lógica Secuencial

LATCHES

Ejemplo:
Latch D.

Verilog

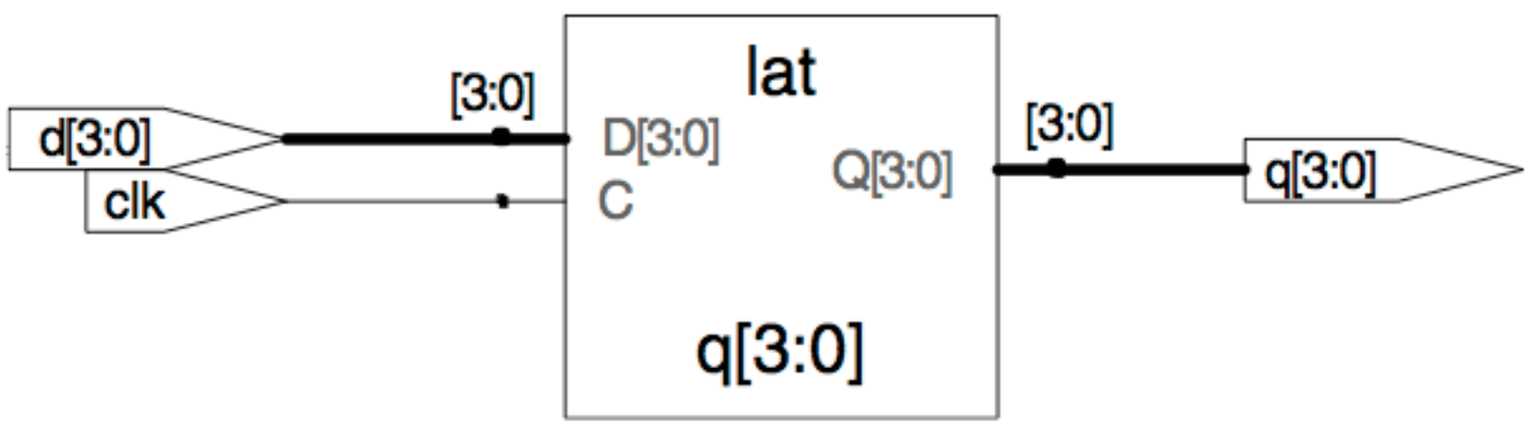
```
module latch(input             clk,
             input      [3:0] d,
             output reg [3:0] q);

  always @ (clk, d)
    if (clk) q <= d;
endmodule
```

The sensitivity list contains both `clk` and `d`, so the `always` statement evaluates any time `clk` or `d` changes. If `clk` is HIGH, `d` flows through to `q`.

`q` must be declared to be a `reg` because it appears on the left hand side of <= in an `always` statement. This does not always mean that `q` is the output of a register.

Circuito `latch` sintetizado



Un poco más de Lógica Combinacional

Ejemplo:
INVERSOR
UTILIZANDO
`always/process`.

Verilog

```
module inv (input      [3:0] a,
            output reg [3:0] y);

  always @ (*)
    y = ~a;
endmodule
```

`always @ (*)` reevaluates the statements inside the `always` statement any time any of the signals on the right hand side of <= or = inside the always statement change. Thus, `@ (*)` is a safe way to model combinational logic. In this particular example, `@ (a)` would also have sufficed.

The = in the always statement is called a *blocking assignment*, in contrast to the <= nonblocking assignment. In Verilog, it is good practice to use blocking assignments for combinational logic and nonblocking assignments for sequential logic. This will be discussed further in Section 4.5.4.

Note that `y` must be declared as `reg` because it appears on the left hand side of a <= or = sign in an `always` statement. Nevertheless, `y` is the output of combinational logic, not a register.

Un poco más de Lógica Combinacional

Verilog

In a Verilog `always` statement, = indicates a blocking assignment and <= indicates a nonblocking assignment (also called a concurrent assignment).

Do not confuse either type with continuous assignment using the `assign` statement. `assign` statements must be used outside `always` statements and are also evaluated concurrently.

Un poco más de Lógica Combinacional

Ejemplo:
FULL ADDER
UTILIZANDO
`always/process`.

Verilog

```
module fulladder(input      a, b, cin,
                 output reg s, cout);
  reg p, g;

  always @ (*)
    begin
      p = a ^ b;                // blocking
      g = a & b;                // blocking

      s = p ^ cin;              // blocking
      cout = g | (p & cin);     // blocking
    end
endmodule
```

In this case, an `@(a, b, cin)` would have been equivalent to `@(*)`. However, `@(*)` is better because it avoids common mistakes of missing signals in the stimulus list.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for combinational logic. This example uses blocking assignments, first computing `p`, then `g`, then `s`, and finally `cout`.

Because `p` and `g` appear on the left hand side of an assignment in an `always` statement, they must be declared to be `reg`.

Un poco más de Lógica Combinacional

DECLARACIÓN CASE

Ejemplo:
DECODIFICADOR
BCD A 7 SEGMENTOS

Verilog

```
module sevenseg (input      [3:0] data,
                 output reg [6:0] segments);
  always @ (*)
    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_1011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```

The `case` statement checks the value of `data`. When `data` is 0, the statement performs the action after the colon, setting `segments` to 1111110. The `case` statement similarly checks other `data` values up to 9 (note the use of the default base, base 10).

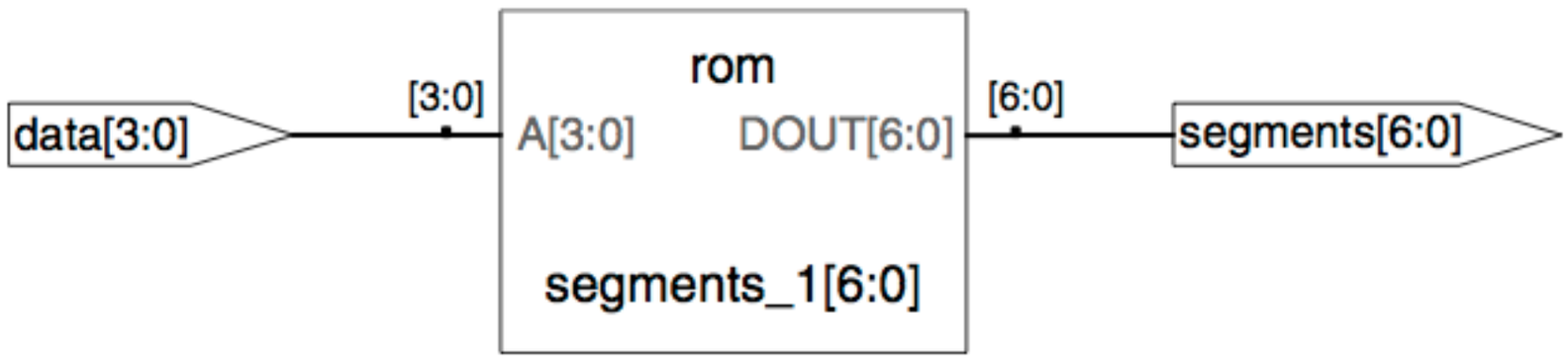
The `default` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

In Verilog, `case` statements must appear inside `always` statements.

Un poco más de Lógica Combinacional

DECLARACIÓN CASE

Circuito `sevenseg` sintetizado



Un poco más de Lógica Combinacional

DECLARACIÓN CASE

Ejemplo:
DECODIFICADOR 3:8

Verilog

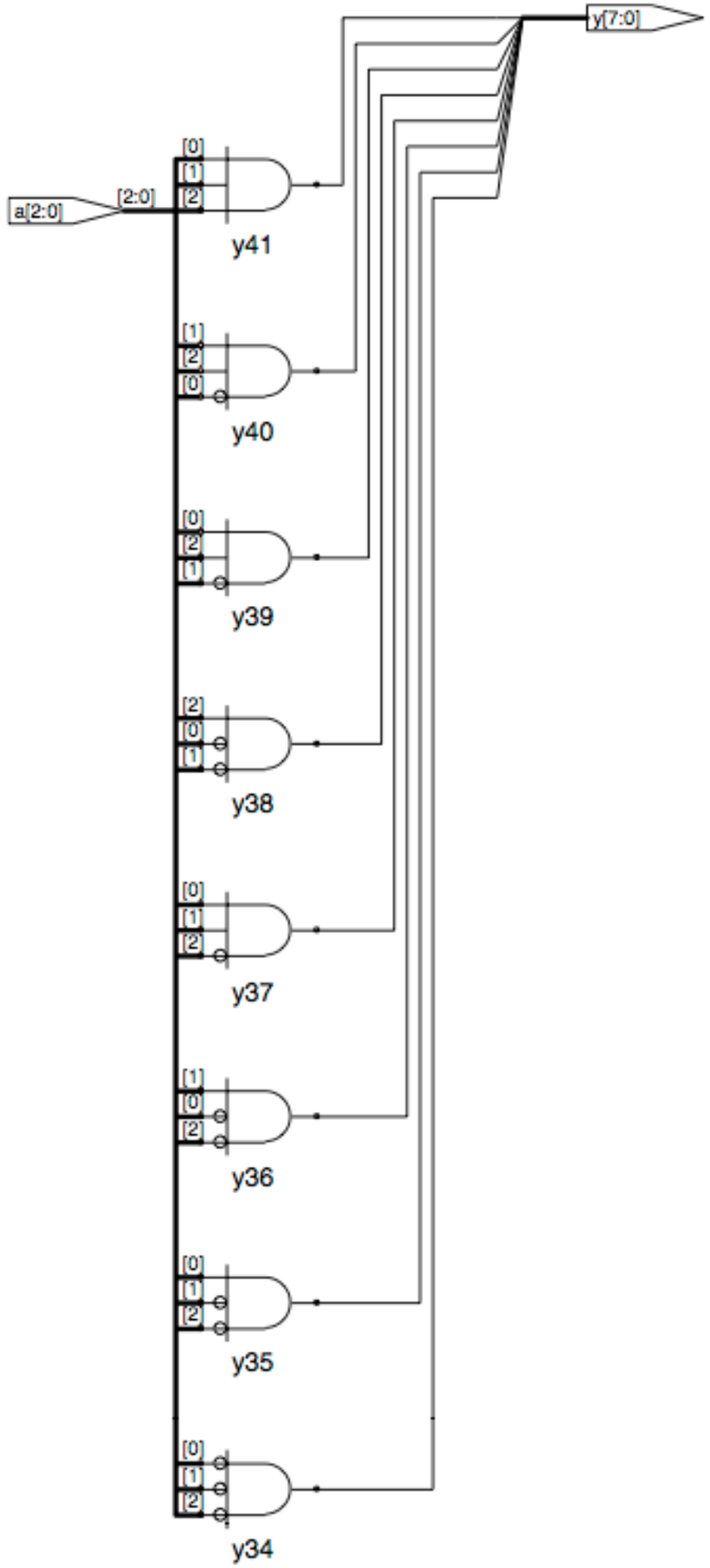
```
module decoder3_8 (input      [2:0] a,
                   output reg [7:0] y);
  always @ (*)
    case (a)
      3'b000: y = 8'b00000001;
      3'b001: y = 8'b00000010;
      3'b010: y = 8'b00000100;
      3'b011: y = 8'b00001000;
      3'b100: y = 8'b00010000;
      3'b101: y = 8'b00100000;
      3'b110: y = 8'b01000000;
      3'b111: y = 8'b10000000;
    endcase
endmodule
```

No `default` statement is needed because all cases are covered.

Un poco más de Lógica Combinacional

DECLARACIÓN CASE

Circuito
`decoder3_8`
sintetizado



Un poco más de Lógica Combinacional

DECLARACIÓN If

Ejemplo:
CIRCUITO DE PRIORIDAD

Verilog

```
module priority (input      [3:0] a,
                 output reg [3:0] y);

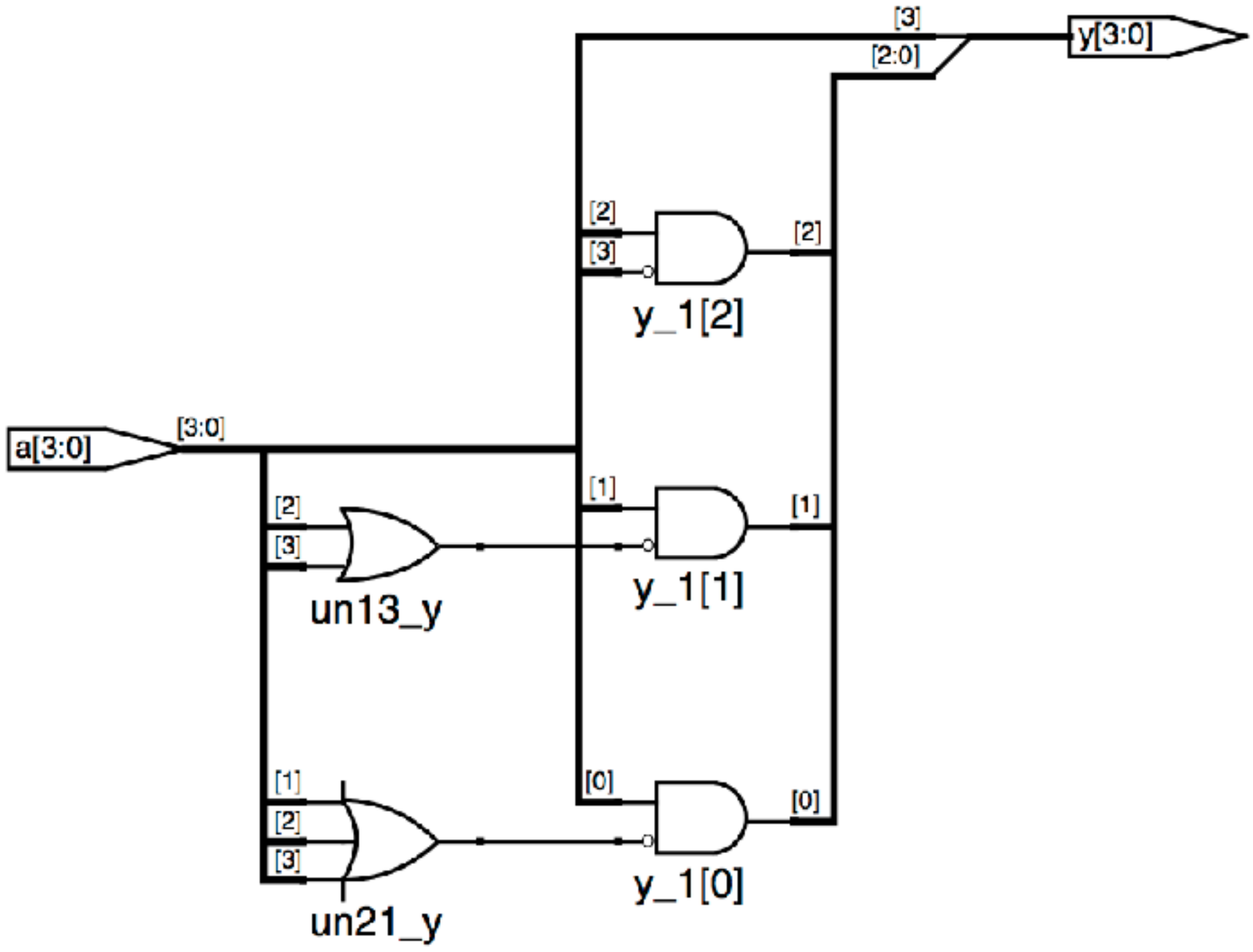
  always @ (*)
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000;
endmodule
```

In Verilog, `if` statements must appear inside of `always` statements.

Un poco más de Lógica Combinacional

DECLARACIÓN If

Circuito `priority` sintetizado



Un poco más de Lógica Combinacional

DECLARACIÓN casez

- Verilog provee `casez` para describir tablas de verdad con *don´t cares* (indicados como `?` en la declaración `casez`).

Un poco más de Lógica Combinacional

DECLARACIÓN casez

Ejemplo:
CIRCUITO DE PRIORIDAD

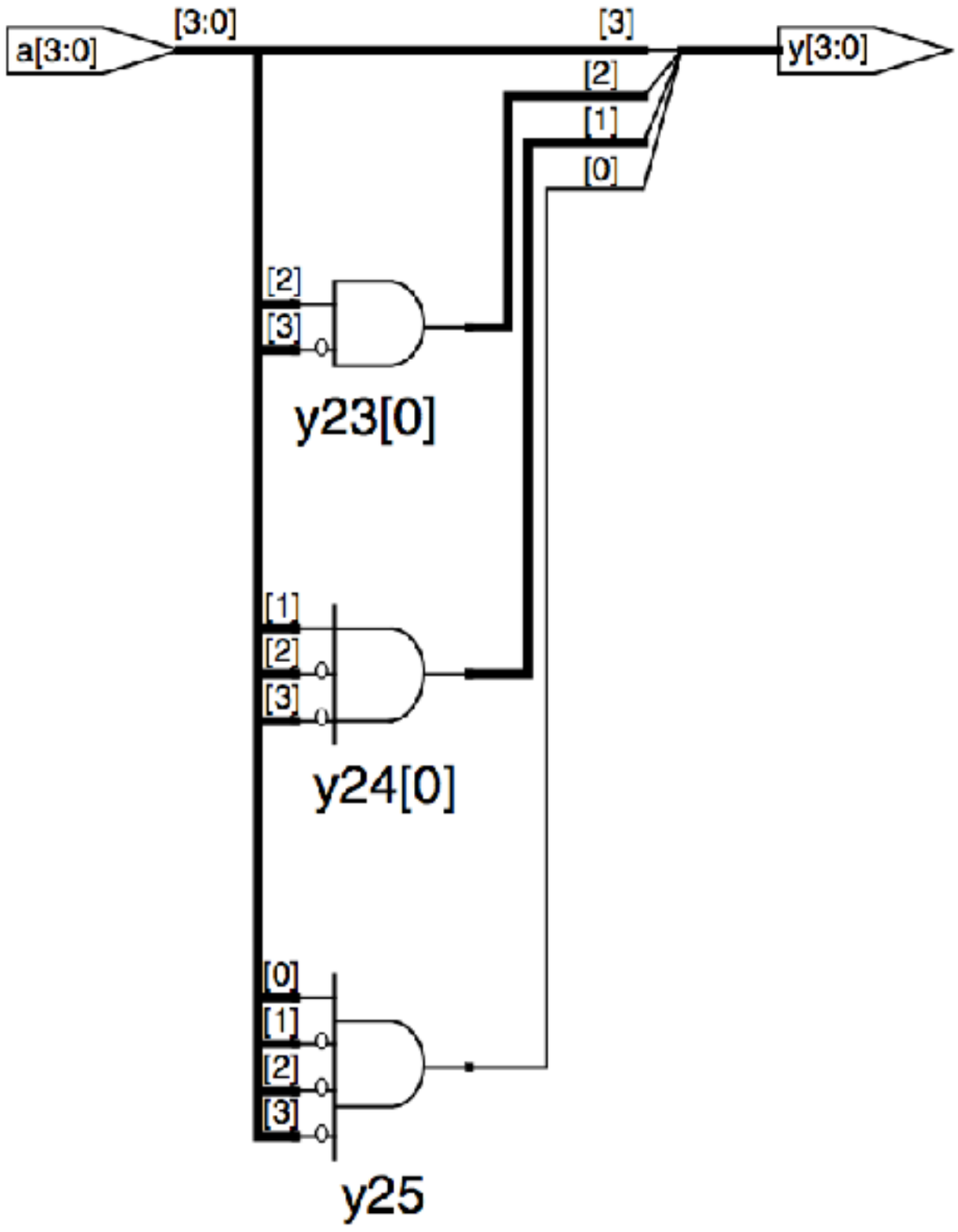
```
module priority_casez(input      [3:0] a,
                      output reg [3:0] y);

  always @ (*)
    casez (a)
      4'b1???: y = 4'b1000;
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
  endcase
endmodule
```

Un poco más de Lógica Combinacional

DECLARACIÓN casez

Circuito `priority` sintetizado



Asignaciones *Blocking* y *Nonblocking*

- La guía de la próxima transparencia explica cómo utilizar ambos tipos de asignaciones (*nonblocking y blocking assignments*).
- Si esta guía no se sigue, es posible escribir código Verilog que aparentemente funciona en las simulaciones pero que sintetiza hardware incorrecto.

Asignaciones *Blocking* y *Nonblocking*

Verilog

1. Use `always @ (posedge clk)` and nonblocking assignments to model synchronous sequential logic.

```
always @ (posedge clk)
  begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
  end
```

2. Use continuous assignments to model simple combinational logic.

```
assign y = s ? d1 : d0;
```

3. Use `always @ (*)` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful.

```
always @ (*)
  begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
  end
```

4. Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.

Asignaciones *Blocking* y *Nonblocking*

LÓGICA COMBINACIONAL

- El *full adder* que vimos está modelado correctamente utilizando asignaciones *blocking*.
- supongamos que originalmente *a*, *b* y *cin* son 0, entonces *p*, *g*, *s* y *cout* son también 0.
- En determinado momento *a* cambia a 1, gatillando la declaración *always*. Las cuatro asignaciones *blocking* se ejecutan en el orden que se muestra. Notar que *p* y *g* toman sus nuevos valores antes que *s* y *cout* sean evaluados debido a la asignación *blocking*.

Esto es importante ya que queremos calcular *s* y *cout* con los nuevos valores de *p* y de *g*.

1. $p \leftarrow 1 \oplus 0 = 1$
2. $g \leftarrow 1 \bullet 0 = 0$
3. $s \leftarrow 1 \oplus 0 = 1$
4. $cout \leftarrow 0 + 1 \bullet 0 = 0$

Asignaciones *Blocking* y *Nonblocking*

LÓGICA COMBINACIONAL

Verilog

```
module fulladder(input      a, b, cin,
                 output reg s, cout);
  reg p, g;

  always @ (*)
    begin
      p = a ^ b;              // blocking
      g = a & b;              // blocking

      s = p ^ cin;            // blocking
      cout = g | (p & cin);   // blocking
    end
endmodule
```

In this case, an `@(a, b, cin)` would have been equivalent to `@(*)`. However, `@(*)` is better because it avoids common mistakes of missing signals in the stimulus list.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for combinational logic. This example uses blocking assignments, first computing `p`, then `g`, then `s`, and finally `cout`.

Because `p` and `g` appear on the left hand side of an assignment in an `always` statement, they must be declared to be `reg`.

Asignaciones *Blocking* y *Nonblocking*

LÓGICA COMBINACIONAL

- Implementación del *full adder* utilizando asignaciones *nonblocking*.

Verilog

```
// nonblocking assignments (not recommended)
module fulladder (input      a, b, cin,
                  output reg s, cout);
  reg p, g;

  always @ (*)
    begin
      p <= a ^ b; // nonblocking
      g <= a & b; // nonblocking

      s <= p ^ cin;
      cout <= g | (p & cin);
    end
endmodule
```

Because `p` and `g` appear on the left hand side of an assignment in an `always` statement, they must be declared to be `reg`.

Asignaciones *Blocking* y *Nonblocking*

LÓGICA COMBINACIONAL

- En este caso las cuatro asignaciones *nonblocking* se ejecutan concurrentemente

$$p \leftarrow 1 \oplus 0 = 1 \quad g \leftarrow 1 \bullet 0 = 0 \quad s \leftarrow 0 \oplus 0 = 0 \quad cout \leftarrow 0 + 0 \bullet 0 = 0$$

- *s* se calcula concurrentemente con *p*, por lo tanto usa el valor anterior de *p*.
- Sin embargo *p* cambia de 0 a 1. Este cambio gatilla una segunda evaluación del *always* como sigue

$$p \leftarrow 1 \oplus 0 = 1 \quad g \leftarrow 1 \bullet 0 = 0 \quad s \leftarrow 1 \oplus 0 = 1 \quad cout \leftarrow 0 + 1 \bullet 0 = 0$$

- Esta vez *p* es 1 y *s* cambia en forma correcta a 1.

Asignaciones *Blocking* y *Nonblocking*

LÓGICA COMBINACIONAL

- La asignación *nonblocking* eventualemnte evalúa en forma correcta, pero el *always* debe evaluarse dos veces.
- Esto hace que la simulación sea más lenta, sin embargo sintetiza el mismo hardware.
- Otra desventaja de las asignaciones *nonblocking* para modelar lógica combinacional es que se obtiene un resultado incorrecto si no se incluyen las variables intermedias en la lista de sensibilidades.
- Peor aún, algunas herramientas sintetizan el hardware correcto cuando una lista de sensibilidades fallada causa una simulación incorrecta. Esto genera una inconsistencia entre los resultados de la simulación y lo que hace el hardware.

Asignaciones *Blocking* y *Nonblocking*

LÓGICA SECUENCIAL

- El sicronizador que vimos se modela correctamente utilizando asignaciones *nonblocking*.
- Supongamos que inicialmente `d=0`, `n1=1` y `q=0`.
  En el flanco de subida del clock, las dos siguientes asignaciones ocurren concurrentemente

$$n1 \leftarrow d = 0 \quad q \leftarrow n1 = 1$$

- Intentemos describir el mismo sincronizador utilizando asignaciones *blocking*.

Asignaciones *Blocking* y *Nonblocking*

LÓGICA SECUENCIAL

Verilog

```
// Bad implementation using blocking assignments

module syncbad (input      clk,
                input      d,
                output reg q);

  reg n1;

  always @ (posedge clk)
    begin
       n1 = d; // blocking
       q = n1; // blocking
    end
endmodule
```

Asignaciones *Blocking* y *Nonblocking*

LÓGICA SECUENCIAL

- En el flanco de subida del clock, `d` es copiado a `n1`. Luego, este nuevo valor de `n1` es copiado a `q`, haciendo que `d` aparezca equivocadamente en `n1` y en `q`.

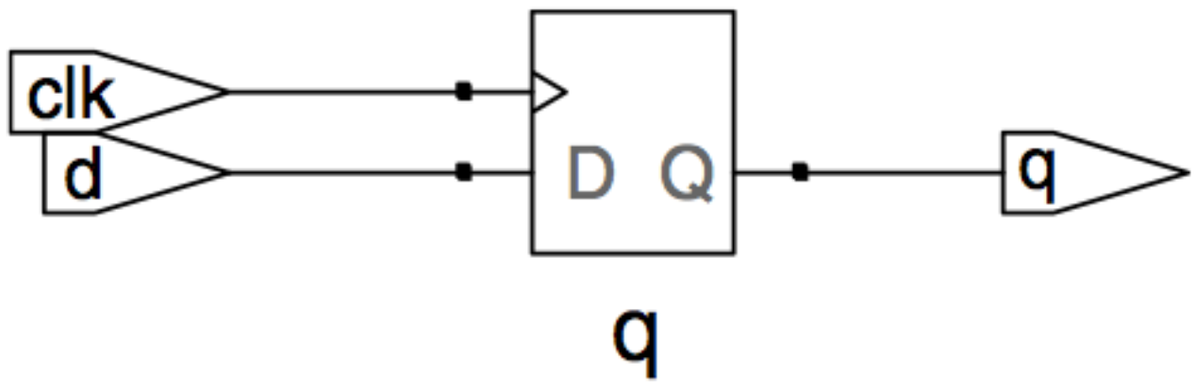
- Las asignaciones ocurren una después de la otra, por lo tanto

1. `n1` $\leftarrow$ `d` $= 0$
2. `q` $\leftarrow$ `n1` $= 0$

Asignaciones *Blocking* y *Nonblocking*

LÓGICA SECUENCIAL

- Debido a que `n1` es invisible al mundo externo, y no tiene influencia en el valor de q, el sintetizador optimiza equivocadamente, como muestra la figura.



- La moraleja es utilizar exclusivamente asignaciones *nonblocking* en declaraciones *always* cuando se modela lógica secuencial.

Máquinas de Estado Finitas

- Una máquina de estados finita (FSM) consiste de un registro de estados y dos bloques combinacionales para generar el próximo estado y la salida, dados el estado actual y la entrada.
- Para modelar una FSM con Verilog, debemos dividir el problema en tres partes, modelar: el registro de estados, la lógica para el próximo estado y la lógica de salida.

Máquinas de Estado Finitas

Verilog

```
module divideby3FSM (input  clk,
                     input  reset,
                     output y);

  reg [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;

  // state register
  always @ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always @ (*)
    case (state)
      S0: nextstate = S1;
      S1: nextstate = S2;
      S2: nextstate = S0;
      default: nextstate = S0;
    endcase

  // output logic
  assign y = (state == S0);
endmodule
```

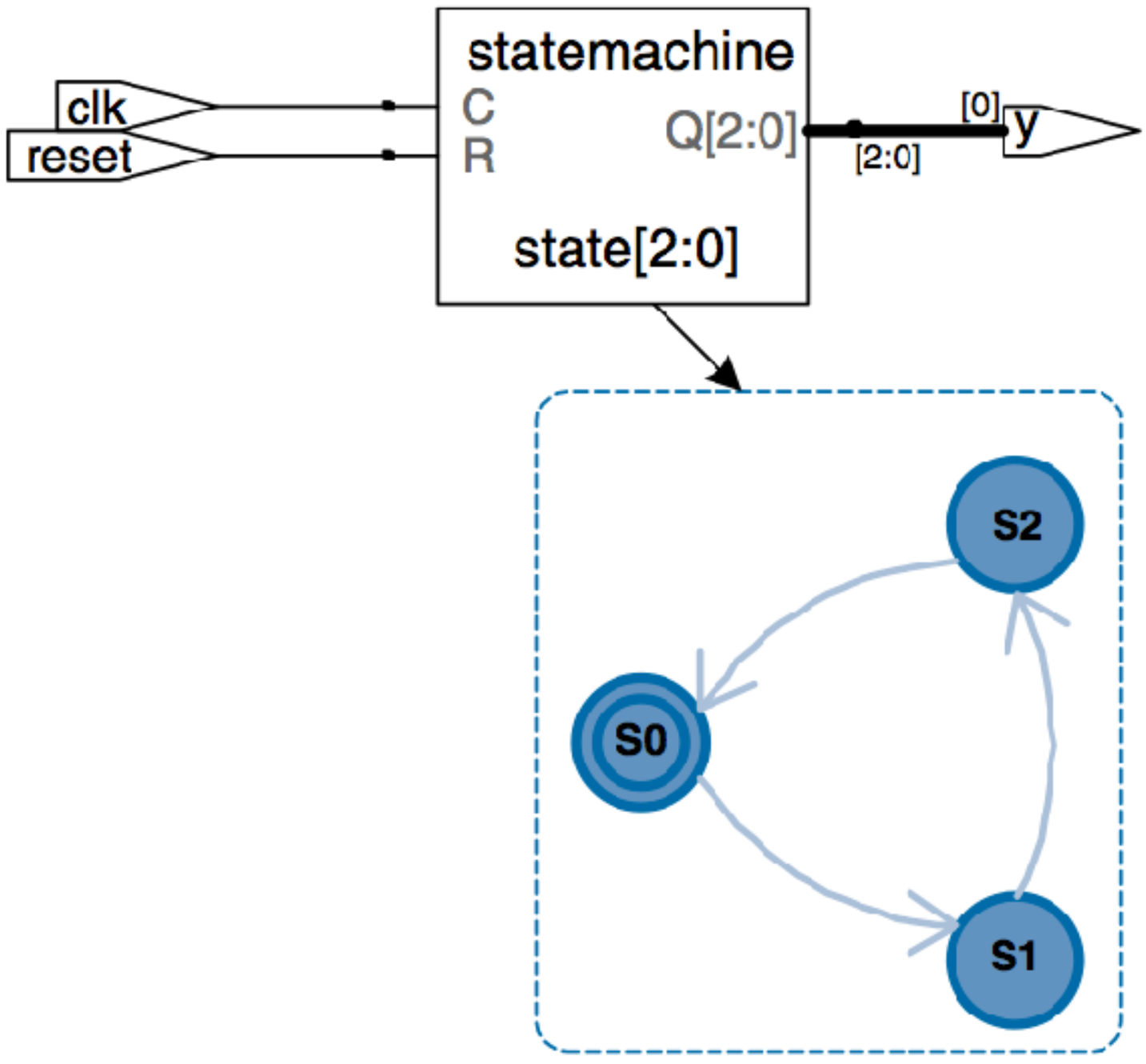
The `parameter` statement is used to define constants within a module. Naming the states with parameters is not required, but it makes changing state encodings much easier and makes the code more readable.

Notice how a `case` statement is used to define the state transition table. Because the next state logic should be combinational, a `default` is necessary even though the state `2'b11` should never arise.

The output, `y`, is 1 when the state is `S0`. The *equality comparison* `a == b` evaluates to 1 if `a` equals `b` and 0 otherwise. The *inequality comparison* `a != b` does the inverse, evaluating to 1 if `a` does not equal `b`.

Máquinas de Estado Finitas

Circuito sintetizado



Máquinas de Estado Finitas

- Si por algún motivo hubiésemos querido que y fuera `1` tanto en `S0` como en `S1`, la descripción sería:

```
// Output Logic
assign y = (state == S0  |  state == S1);
```

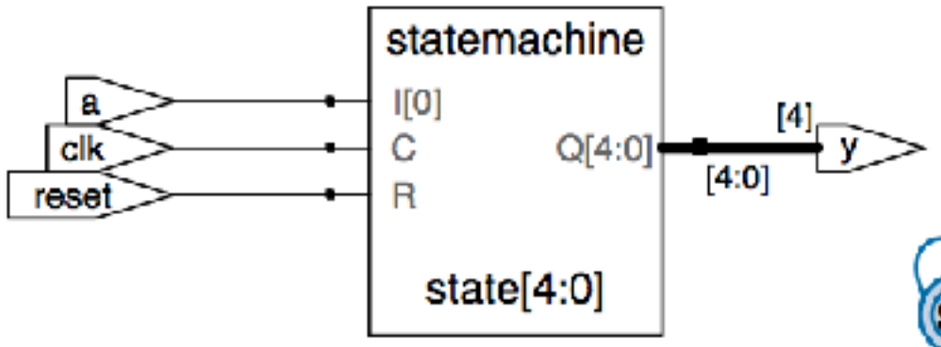
- El siguiente ejemplo muestra un reconocedor de patrones de bits implementado utilizando una máquina de Moore y una máquina de Mealy.

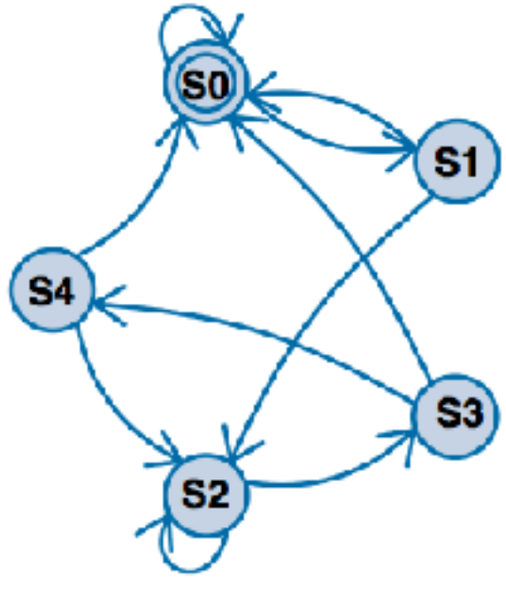
Máquinas de Estado Finitas

RECONOCEDOR DE SECUENCIA

Note how nonblocking assignments (<=) are used in the state register to describe sequential logic, whereas blocking assignments (=) are used in the next state logic to describe combinational logic.

Circuito sintetizado






```
module patternMoore(input  clk,
                    input  reset,
                    input  a,
                    output y);

  reg [2:0] state, nextstate;

  parameter S0 = 3'b000;
  parameter S1 = 3'b001;
  parameter S2 = 3'b010;
  parameter S3 = 3'b011;
  parameter S4 = 3'b100;

  // state register
  always @ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

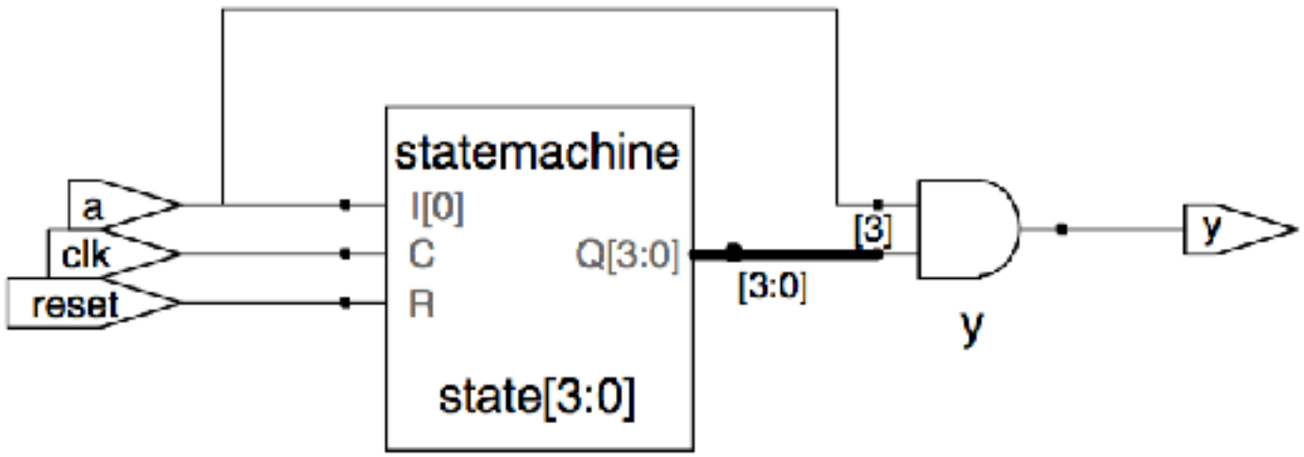
  // next state logic
  always @ (*)
    case (state)
      S0: if (a) nextstate = S1;
          else   nextstate = S0;
      S1: if (a) nextstate = S2;
          else   nextstate = S0;
      S2: if (a) nextstate = S2;
          else   nextstate = S3;
      S3: if (a) nextstate = S4;
          else   nextstate = S0;
      S4: if (a) nextstate = S2;
          else   nextstate = S0;
      default:   nextstate = S0;
    endcase

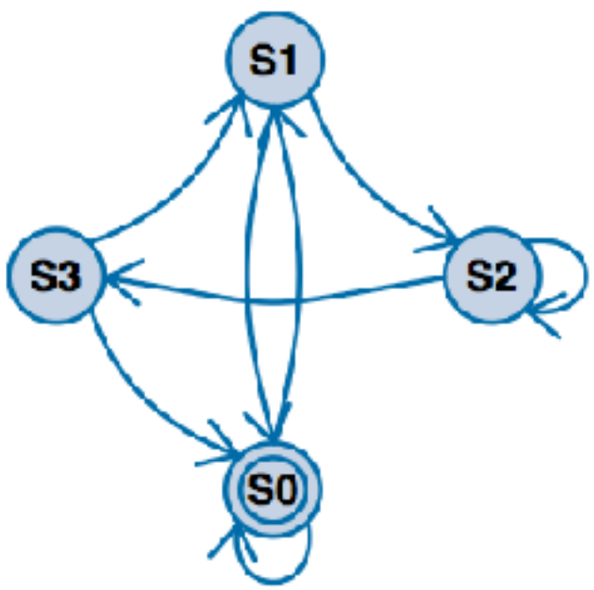
  // output logic
  assign y = (state == S4);
endmodule
```

Máquinas de Estado Finitas

RECONOCEDOR DE SECUENCIA

Circuito sintetizado






```
module patternMealy(input  clk,
                    input  reset,
                    input  a,
                    output y);

  reg [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;

  // state register
  always @ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always @ (*)
    case (state)
      S0: if (a) nextstate = S1;
          else   nextstate = S0;
      S1: if (a) nextstate = S2;
          else   nextstate = S0;
      S2: if (a) nextstate = S2;
          else   nextstate = S3;
      S3: if (a) nextstate = S1;
          else   nextstate = S0;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (a & state == S3);
endmodule
```

Módulos Parametrizados

- Hasta ahora todos los módulos han sido de un ancho de bits fijo.
- Por ejemplo, hemos desarrollado módulos separados para 4 multiplexores de 2 bits y para 8 multiplexores de 2 bits.
- Verilog permite anchos de bit variables utilizando módulos parametrizados.

Módulos Parametrizados

Multiplexores parametrizados de N bits

Verilog

```
module mux2
  #(parameter width = 8)
    (input  [width-1:0] d0, d1,
     input              s,
     output [width-1:0] y);

    assign y = s ? d1 : d0;
endmodule
```

Verilog allows a `#(parameter ...)` statement before the inputs and outputs to define parameters. The `parameter` statement includes a default value (8) of the parameter, `width`. The number of bits in the inputs and outputs can depend on this parameter.

```
module mux4_8(input  [7:0] d0, d1, d2, d3,
              input  [1:0] s,
              output [7:0] y);

  wire [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[1], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

The 8-bit 4:1 multiplexer instantiates three 2:1 multiplexers using their default widths.

Módulos Parametrizados

Multiplexores
parametrizados de N bits

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `#()` before the instance name, as shown below.

```
module mux4_12 (input  [11:0] d0, d1, d2, d3,
                input  [1:0]  s,
                output [11:0] y);

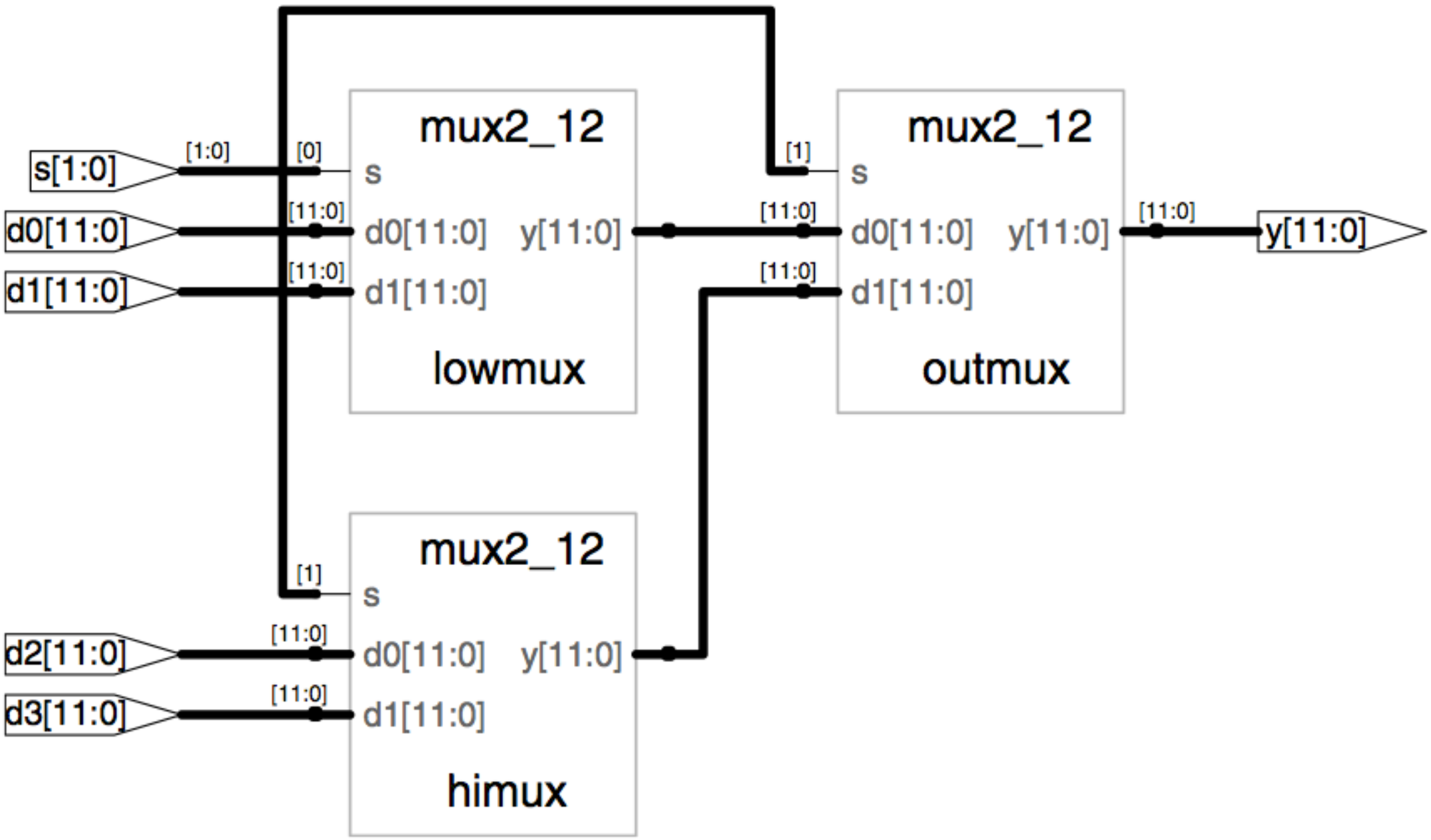
  wire [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[1], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

Do not confuse the use of the # sign indicating delays with the use of #(. . .) in defining and overriding parameters.

Módulos Parametrizados

`mux4_12` sintetizado



Módulos Parametrizados

Decodificador
parametrizado de N : 2^N

Verilog

```
module decoder #(parameter   N = 3)
                (input       [N-1:0]    a,
                 output reg  [2**N-1:0] y);

  always @ (*)
    begin
      y = 0;
      y[a] = 1;
    end
endmodule
```

2**N indicates 2^N.

Módulos Parametrizados

Compuerta AND parametrizada de *N* bits

Verilog

```
module andN
  #(parameter width = 8)
    (input [width-1:0] a,
     output            y);

  genvar i;
  wire [width-1:1] x;

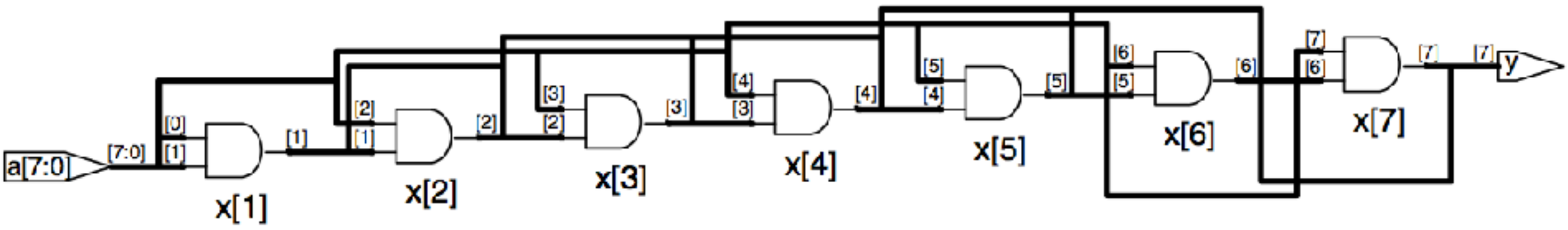
  generate
    for (i=1; i<width; i=i+1) begin:forloop
      if (i == 1)
        assign x[1] = a[0] & a[1];
      else
        assign x[i] = a[i] & x[i-1];
    end
  endgenerate
  assign y = x[width-1];
endmodule
```

The `for` statement loops through `i` = 1, 2, . . . , `width-1` to produce many consecutive AND gates. The `begin` in a `generate for` loop must be followed by a : and an arbitrary label (`forloop`, in this case).

Of course, writing `assign y = &a` would be much easier!

Módulos Parametrizados

Compuerta AND
parametrizada, sintetizada
de 8 bits



Testbenches

- Un *testbench* es un módulo Verilog utilizado para probar otro módulo, normalmente llamado *unit under test* (uut) o *device under test* (dut).
- Un *testbench* contiene declaraciones que aplican entradas a la uut e, idealmente, comprueba que las salidas correspondientes sean las correctas.

Testbenches

Testbench para la unidad `sillyfunction`

$$y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$$

```
module testbench1 ();
  reg a, b, c;
  wire y;

  // instantiate device under test
  sillyfunction dut (a, b, c, y);

  // apply inputs one at a time
    initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
    a = 1; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
  end
endmodule
```

The `initial` statement executes the statements in its body at the start of simulation. In this case, it first applies the input pattern 000 and waits for 10 time units. It then applies 001 and waits 10 more units, and so forth until all eight possible inputs have been applied. `initial` statements should be used only in testbenches for simulation, not in modules intended to be synthesized into actual hardware. Hardware has no way of magically executing a sequence of special steps when it is first turned on.

Like signals in `always` statements, signals in `initial` statements must be declared to be `reg`.

Testbenches

- Definir una "instancia" del módulo "dut".
- Utilizar asignaciones *blocking* y retardos para generar las entradas en el orden apropiado.
- El usuario revisa las salidas en el módulo de simulación para comprobar que sean las correctas.
- Los *testbenches* no son sintetizables.
- Comprobar las salidas es tedioso y puede llevar a error. Cuando el diseño está fresco en la mente puede ser fácil, pero si se introduce algún cambio y se retoma algunas semanas después puede ser una muy complejo.
- Una mejor forma es construir un *testbench* con auto chequeo.

Testbenches

Testbench con auto chequeo para la unidad `sillyfunction`

```
module testbench2 ();
  reg a, b, c;
  wire y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  // checking results
  initial begin
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1;                #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0;         #10;
    if (y !== 0) $display("010 failed.");
    c = 1;                #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1;                #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0;         #10;
    if (y !== 0) $display("110 failed.");
    c = 1;                #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

This module checks `y` against expectations after each input test vector is applied. In Verilog, comparison using == or != is effective between signals that do not take on the values of `x` and `z`. Testbenches use the === and !== operators for comparisons of equality and inequality, respectively, because these operators work correctly with operands that could be `x` or `z`. It uses the `$display` *system task* to print a message on the simulator console if an error occurs. `$display` is meaningful only in simulation, not synthesis.

Testbenches

- Escribir código para cada vector de prueba también resulta tedioso, especialmente para circuitos que requieren un gran número de vectores.
- Una mejor forma consiste en poner los vectores de prueba en un archivo separado.
- En este caso el *testbench* lee los vectores del archivo, aplica un vector al dut, espera, comprueba que la salida corresponda al vector de salida, repitiendo esta operación hasta alcanzar el final del archivo.

Testbenches

Ejemplo:

- El *testbench* genera un clock utilizando always sin una lista de estímulos, de tal forma que es continuamente reevaluado.
- Al comienzo de la simulación lee los vectores desde el archivo example.tv y pulsa reset por dos ciclos.
- El contenido de example.tv es:

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

Testbenches

Testbench con auto chequeo y archivo de vectores para la unidad `sillyfunction`

```
module testbench3();
  reg        clk, reset;
  reg        a, b, c, yexpected;
  wire       y;
  reg [31:0] vectornum, errors;
  reg [3:0]  testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end

// apply test vectors on rising edge of clk
always @(posedge clk)
  begin
    #1; {a, b, c, yexpected} =
          testvectors[vectornum];
  end

// check results on falling edge of clk
always @(negedge clk)
  if(~reset) begin // skip during reset
    if(y !== yexpected) begin
      $display("Error: inputs = %b", {a, b, c});
      $display(" outputs = %b (%b expected)",
               y, yexpected);
      errors = errors + 1;
    end
    vectornum = vectornum + 1;
    if(testvectors[vectornum] === 4'bx) begin
      $display("%d tests completed with %d errors",
               vectornum, errors);
      $finish;
    end
  end
endmodule
```

Testbenches

`$readmemb` reads a file of binary numbers into the `testvectors` array. `$readmemh` is similar but reads a file of hexadecimal numbers.

The next block of code waits one time unit after the rising edge of the clock (to avoid any confusion if clock and data change simultaneously), then sets the three inputs and the expected output based on the four bits in the current test vector. The next block of code checks the output of the DUT at the negative edge of the clock, after the inputs have had time to propagate through the DUT to produce the output, `y`. The testbench compares the generated output, `y`, with the expected output, `yexpected`, and prints an error if they don't match. `%b` and `%d` indicate to print the values in binary and decimal, respectively. For example, `$display ("%b %b", y, yexpected)`: prints the two values, `y` and `yexpected`, in binary. `%h` prints a value in hexadecimal.

This process repeats until there are no more valid test vectors in the `testvectors` array. `$finish` terminates the simulation.

Note that even though the Verilog module supports up to 10,001 test vectors, it will terminate the simulation after executing the eight vectors in the file.

Testbenches

- En cada flanco de subida un nuevo vector es aplicado y la salida es comprobada en el flanco de bajada.
- Este clock (y reset) también sería provisto si la dut fuera secuencial.
- Los errores son reportados a medida que ocurren.
- Al final de la simulación, el *testbench* imprime el número total de vectores aplicados y el número de errores detectados.