Verilog vs. VHDL

From Chap. 4 of

“digital design and computer architecture” by D. Harris, 2007

outlines

- combinational logic gate
- reduction operator
- conditional assignment
- operator precedence
- VHDL types
- structural modeling
- register
- blocking/non-blocking assignment
- priority circuits
- finite state machine

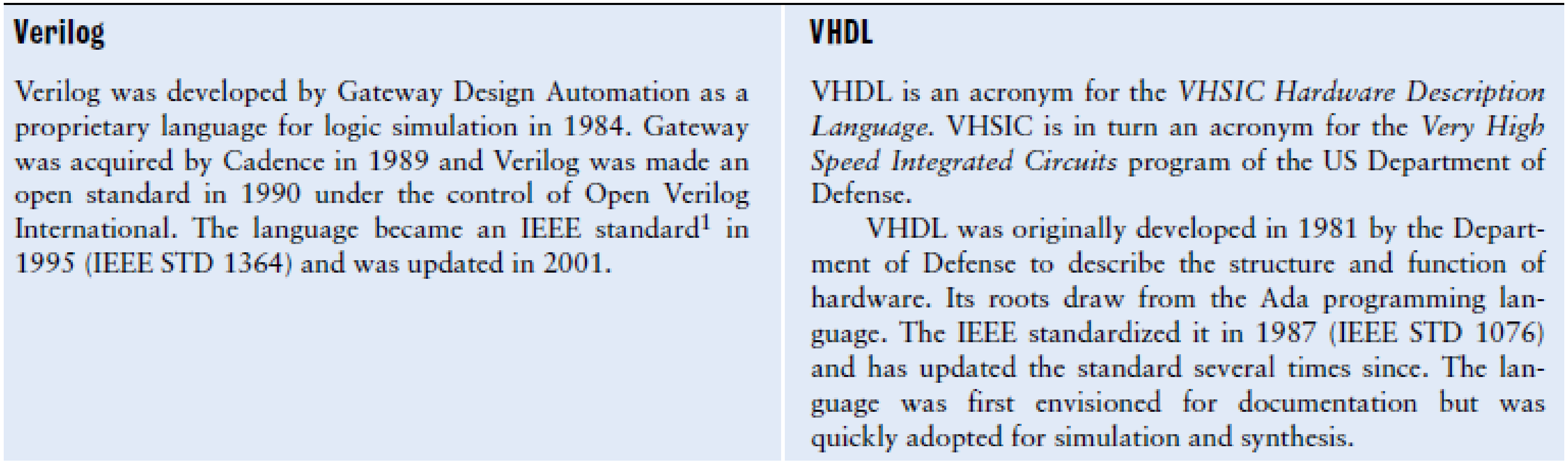
History

Verilog (IEEE std. 1364)

- Become IEEE standard in 1995
- major updated in 2001

VHDL (IEEE std. 1076)

- Become IEEE standard in 1987
- Updated several times (major update in 1993, 2008)



Verilog

Verilog was developed by Gateway Design Automation as a proprietary language for logic simulation in 1984. Gateway was acquired by Cadence in 1989 and Verilog was made an open standard in 1990 under the control of Open Verilog International. The language became an IEEE standard[1] in 1995 (IEEE STD 1364) and was updated in 2001.

VHDL

VHDL is an acronym for the *VHSIC Hardware Description Language*. VHSIC is in turn an acronym for the *Very High Speed Integrated Circuits* program of the US Department of Defense.

VHDL was originally developed in 1981 by the Department of Defense to describe the structure and function of hardware. Its roots draw from the Ada programming language. The IEEE standardized it in 1987 (IEEE STD 1076) and has updated the standard several times since. The language was first envisioned for documentation but was quickly adopted for simulation and synthesis.

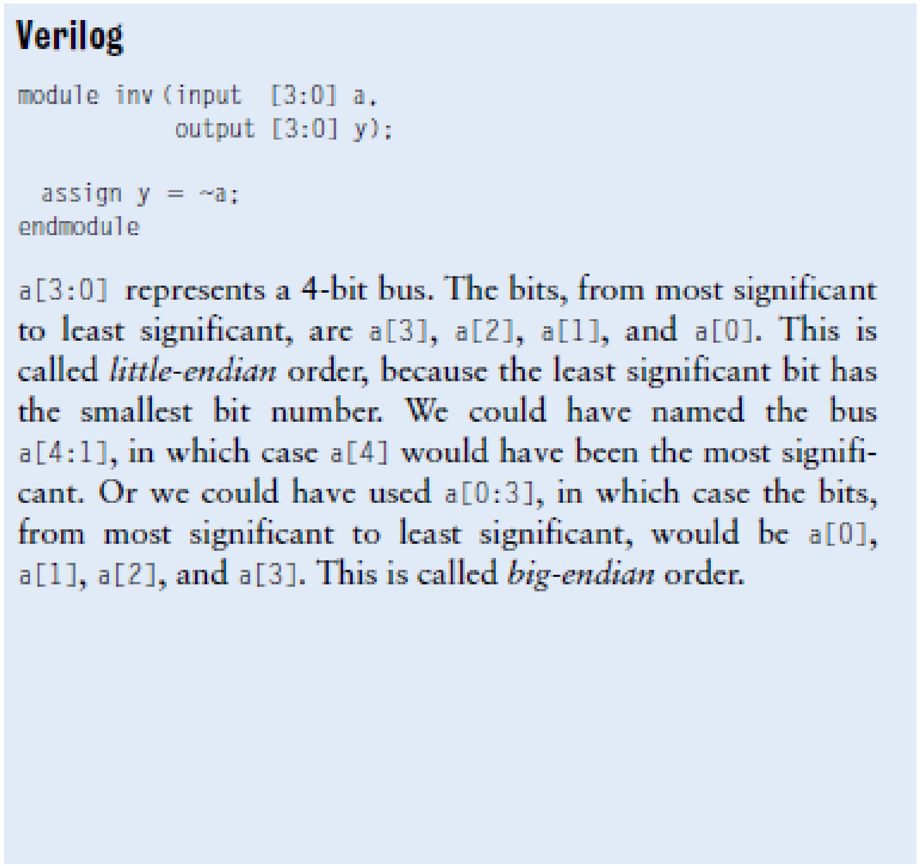
4-bit inverter

input [3:0] a;

assign y = ~a; // dataflow

a: in std_logic_vector (3 downto 0);

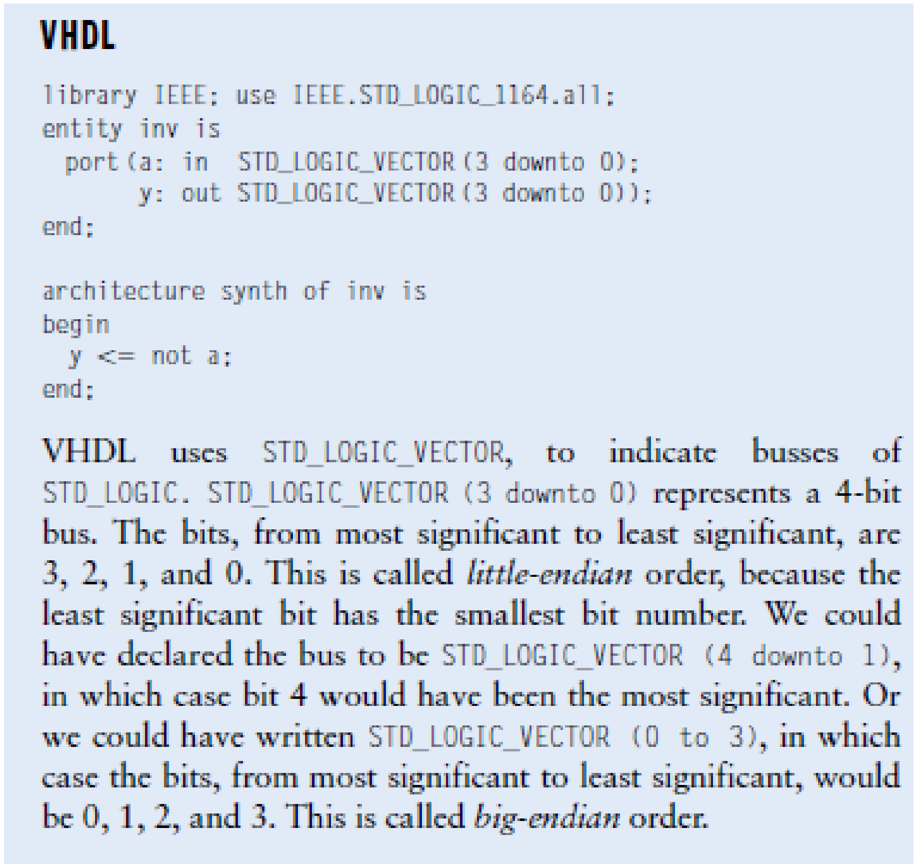
y <= not a; -- concurrent codes



Verilog

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

  assign y = ~a;
endmodule
```

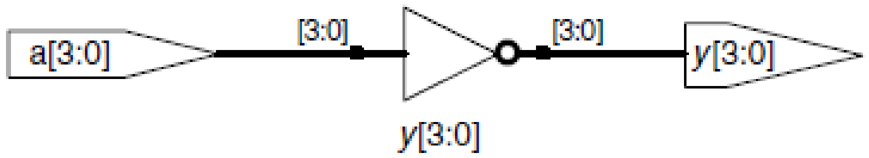
a[3:0] represents a 4-bit bus. The bits, from most significant to least significant, are a[3], a[2], a[1], and a[0]. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have named the bus a[4:1], in which case a[4] would have been the most significant.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



VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

VHDL uses STD_LOGIC_VECTOR, to indicate busses of STD_LOGIC. STD_LOGIC_VECTOR (3 downto 0) represents a 4-bit bus. The bits, from most significant to least significant, are 3, 2, 1, and 0. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have declared the bus to be STD_LOGIC_VECTOR (4 downto 1), in which case bit 4 would have been the most significant. Or we could have written STD_LOGIC_VECTOR (0 to 3), in which case the bits, from most significant to least significant, would be 0, 1, 2, and 3. This is called *big-endian* order.

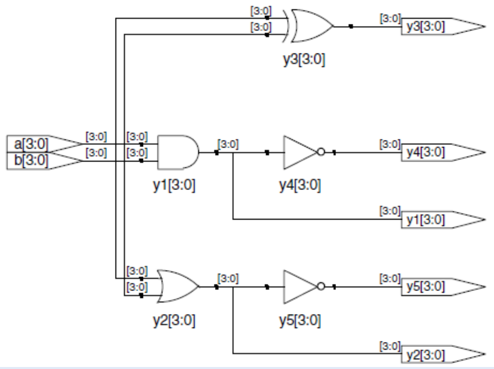


combinational logic gate (using logical operators)

```verilog
// Verilog logic gates
// continuous assignment
// operators: &, |, ^, ~

assign y1 = a & b;
assign y2 = a | b;
assign y3 = a ^ b;
assign y4 = ~(a & b);
assign y5 = ~(a | b);
```

```vhdl
-- VHDL logic gates
-- concurrent statements
-- operators: and, or, xor, not

y1 <= a and b;
y2 <= a or b;
y3 <= a xor b;
y4 <= a nand b;
y5 <= a nor b;
```

CL logic gate using logical operators



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.

VHDL

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

entity gates is
  port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
       y1, y2, y3, y4,
       y5:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
  -- Five different two-input logic gates
  --acting on 4 bit busses
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

not, xor, and or are examples of VHDL *operators*, whereas a, b, and y1 are *operands*. A combination of operators and operands, such as a and b, or a nor b, is called an *expression*. A complete command such as y4 <= a nand b; is called a *statement*.

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

comments

Verilog

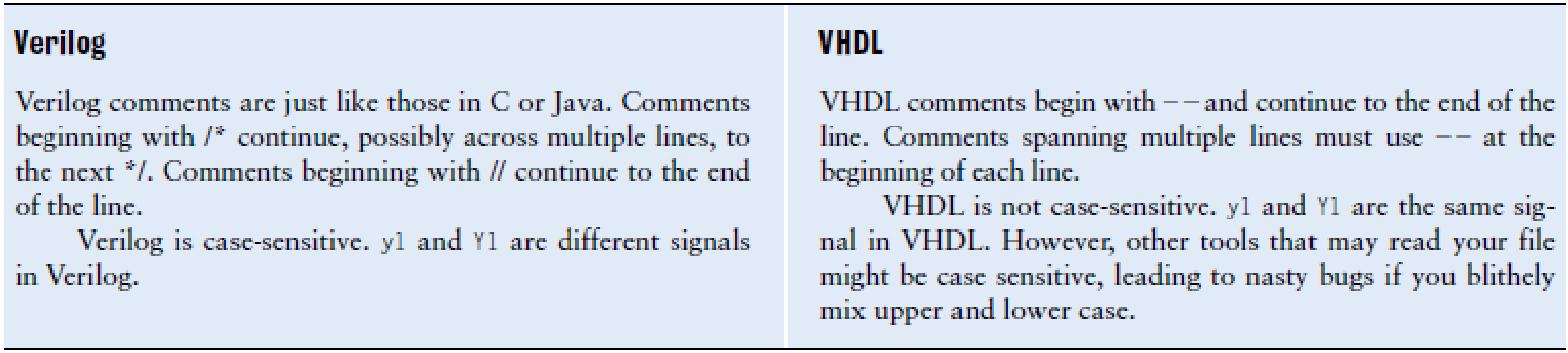
// single line comment
/* multiple line comments */

case sensitive (y1 != Y1)

VHDL

-- single line comment

case insensitive (y1 == Y1)



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.

VHDL

VHDL comments begin with -- and continue to the end of the line. Comments spanning multiple lines must use -- at the beginning of each line.

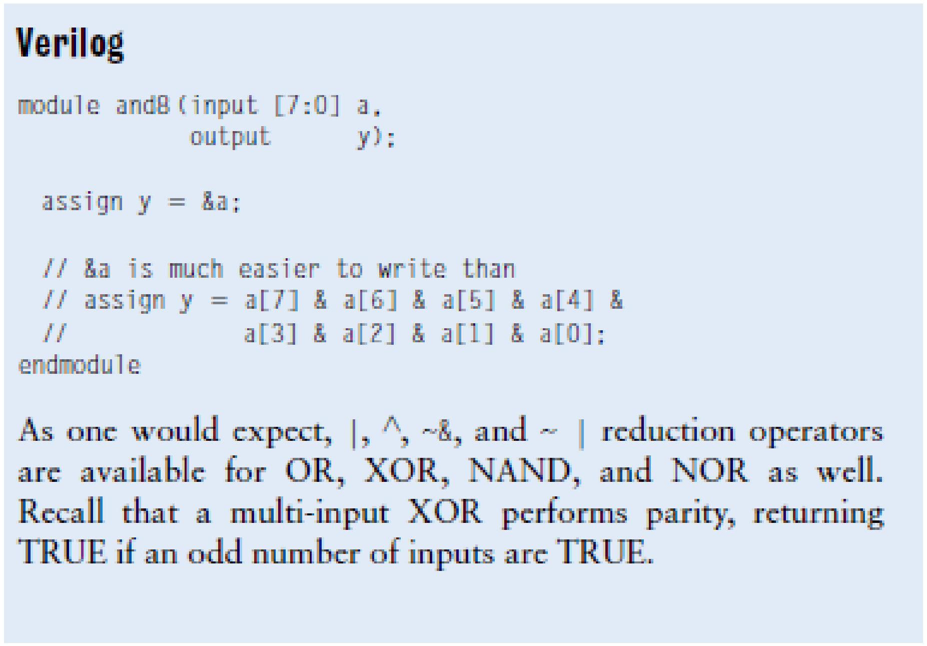
VHDL is not case-sensitive. `y1` and `Y1` are the same signal in VHDL. However, other tools that may read your file might be case sensitive, leading to nasty bugs if you blithely mix upper and lower case.

reduction operator

wire [7:0] a;

assign y = **&**a;

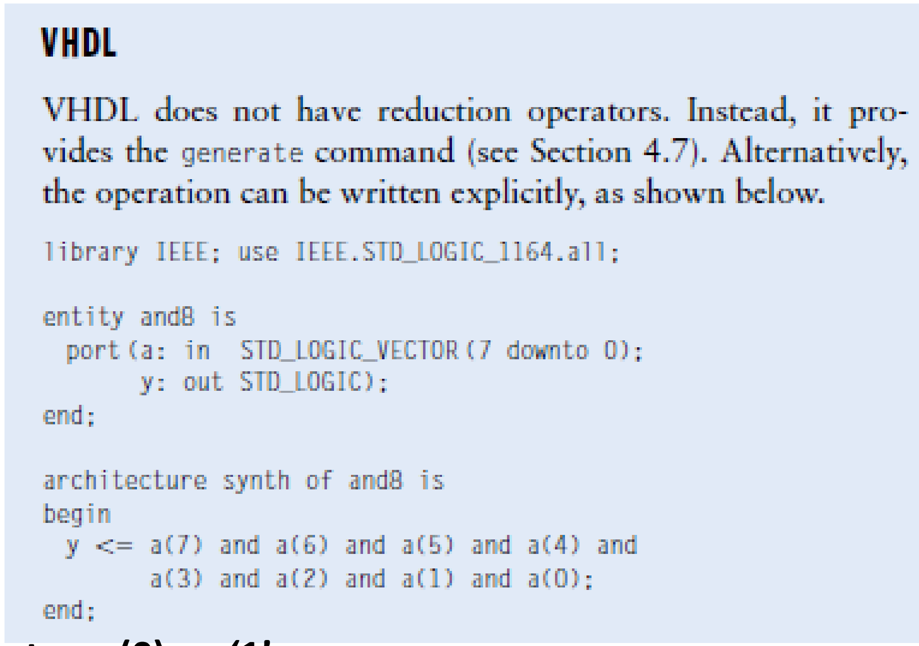
signal a : std_logic_vector (7 **down to** 0)
y <= a(7) **and** a(6) **and** a(5) **and** a(4) **and** a(3) **and** a(2) **and** a(1) **and** a(0)
-- could also use VHDL generate loop



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.

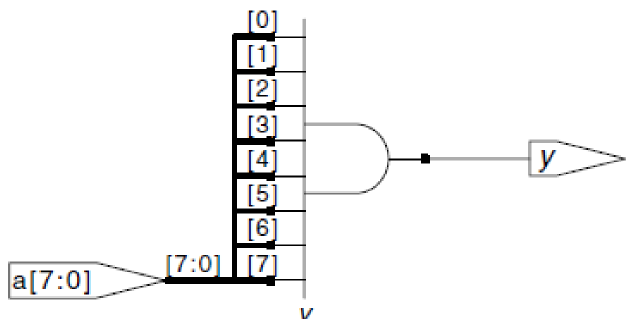


VHDL

VHDL does not have reduction operators. Instead, it provides the generate command (see Section 4.7). Alternatively, the operation can be written explicitly, as shown below.

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

entity and8 is
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of and8 is
begin
  y <= a(7) and a(6) and a(5) and a(4) and
       a(3) and a(2) and a(1) and a(0);
end;
```

temp(0) <= ‘1’;
for i in 0 to a’length -2 generate
  temp(i+1) <= temp(i) and a(i);
end generate;--
y <= temp(a’length-1);
-- or use variable temp := ‘`1’ inide process



VHDL reduction logical operation for large bit-width vectors

```
…
process (all) begin
variable tmp: std_logic_vector (127 downto 0);
integer i;
tmp : = x(0);
for i in 1 to 127 loop
tmp := tmp and x(i);
end loop;
end process;

out <= tmp;
…
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
…
SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL c: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
…
v <= a + b;       -- illegal (arithmetic operations are not OK.)
w <= a AND b;  -- legal (logical operations are OK for STD_LOGIC_VECTOR)
```

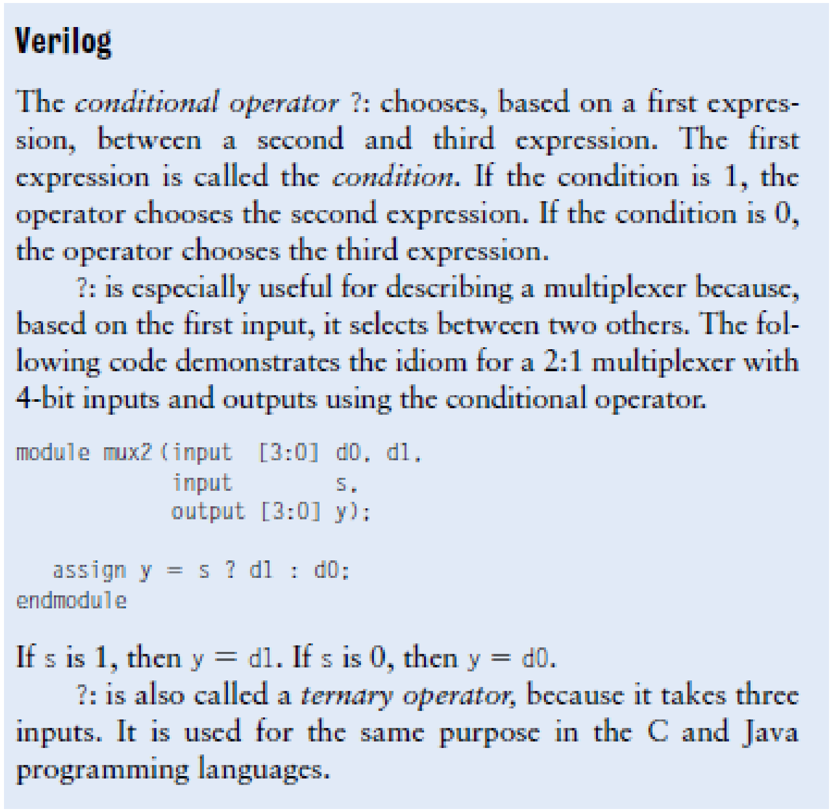
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all; -- package defining signed and unsigned data types
…
SIGNAL a: IN SIGNED (7 DOWNTO 0);
SIGNAL b: IN SIGNED (7 DOWNTO 0);
SIGNAL c: OUT SIGNED (7 DOWNTO 0);
…
v <= a + b;       -- legal (arithmetic operations are OK for SIGNED, UNSIGNED )
w <= a AND b;  -- illegal (logic operations are not OK for signed/unsigned)
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all; -- or USE ieee.std_logic_signed.all
-- allow unsigned (or signed) arithmetic operations on std_logic_vector data types
…
SIGNAL a: IN     STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN     STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL c: OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
…
v <= a + b; -- arithmetic operation OK, std_logic_vector viewed asunsigned
w <= a AND b; -- OK
```

conditional assignment

assign y = s **?** d1 **:** d0;
// Verilog ternary operator

y <= d0 **when** s = ‘0’ **else**
d1;



Verilog

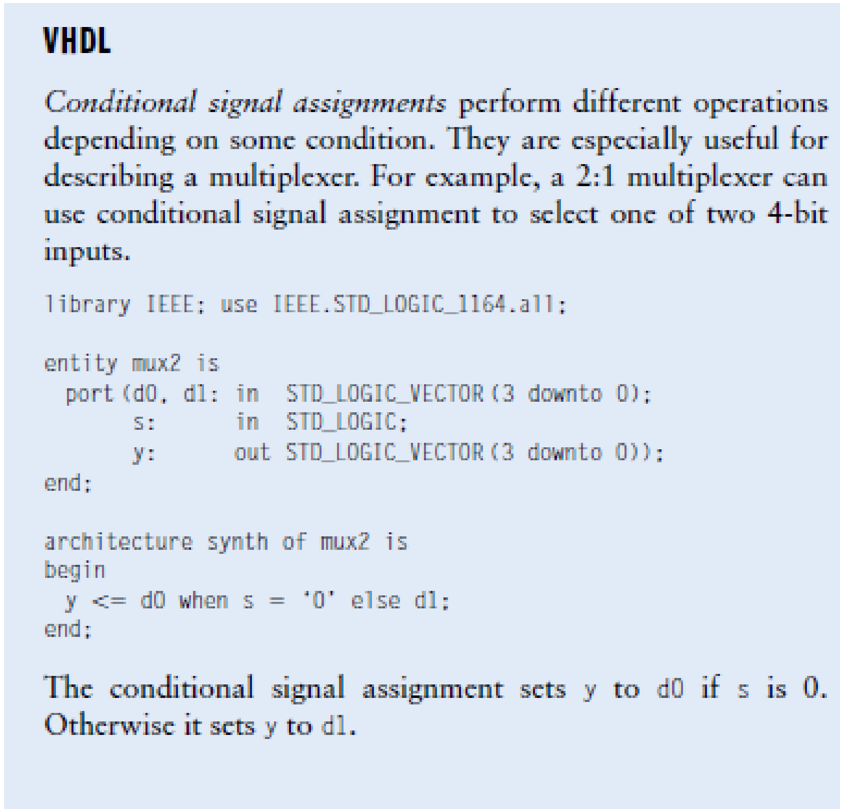
The *conditional operator* ?: chooses, based on a first expression, between a second and third expression. The first expression is called the *condition*. If the condition is 1, the operator chooses the second expression. If the condition is 0, the operator chooses the third expression.

?: 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.

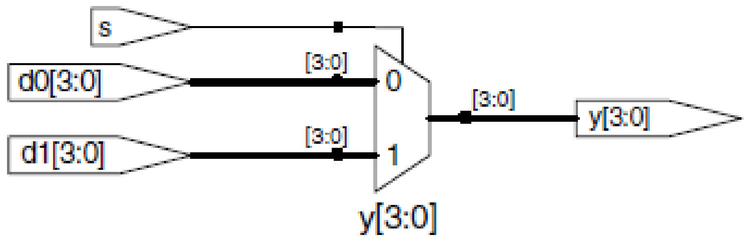


VHDL

Conditional signal assignments perform different operations depending on some condition. They are especially useful for describing a multiplexer. For example, a 2:1 multiplexer can use conditional signal assignment to select one of two 4-bit inputs.

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

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d0 when s = '0' else d1;
end;
```

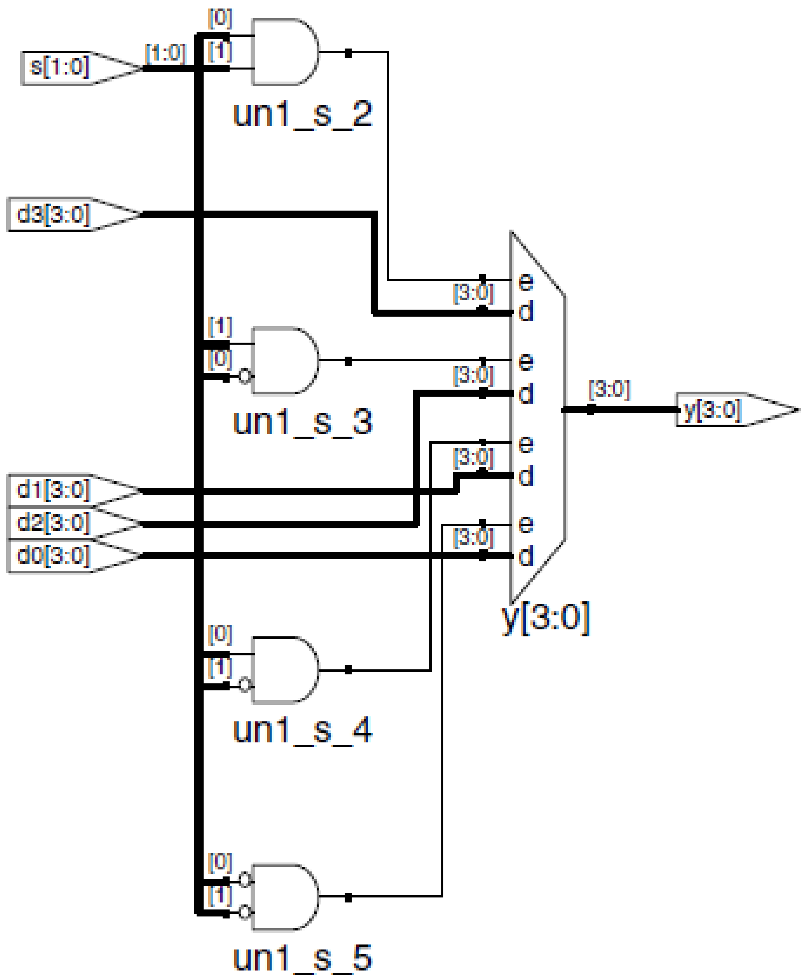
The conditional signal assignment sets y to d0 if s is 0. Otherwise it sets y to d1.



4:1 multiplexer (MUX4), using continuous assignment/concurrent code

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

```
y <= d0 when s = "00" else
     d1 when s = "01" else
     d2 when s = "10" else
     d3;
```



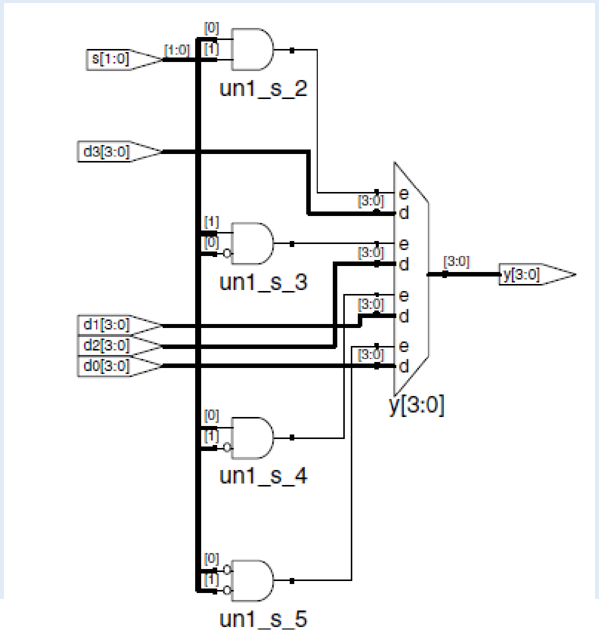
4:1 multiplexer

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].



VHDL

A 4:1 multiplexer can select one of four inputs using multiple else clauses in the conditional signal assignment.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth1 of mux4 is
begin
  y <= d0 when s = "00" else
       d1 when s = "01" else
       d2 when s = "10" else
       d3;
end;
```

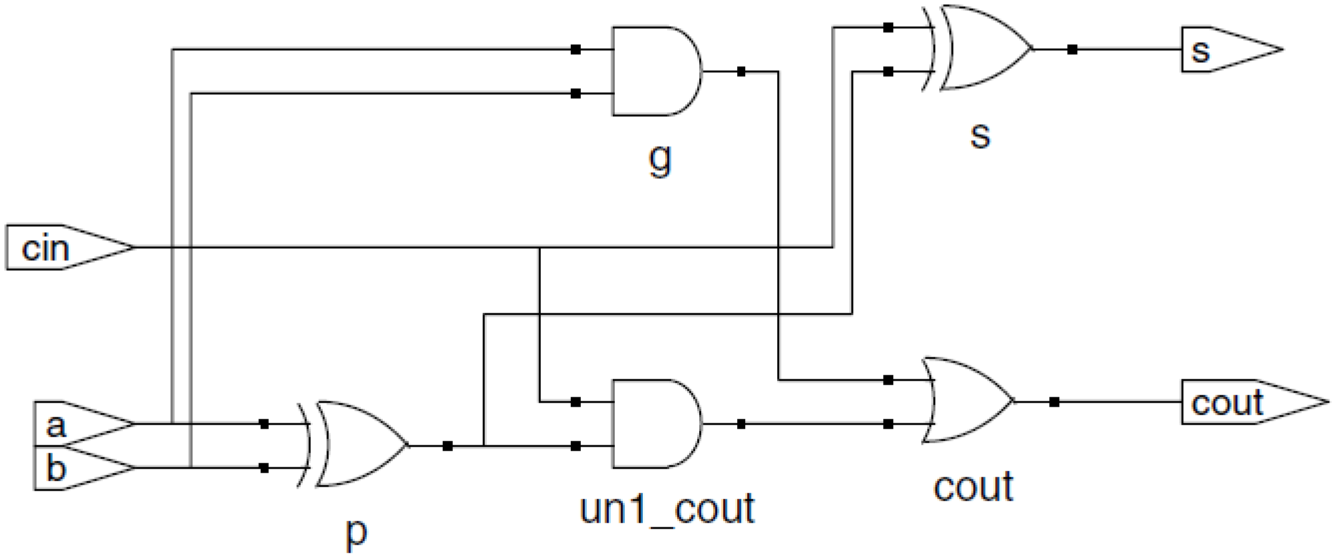
VHDL also supports *selected signal assignment statements* to provide a shorthand when selecting from one of several possibilities. This is analogous to using a case statement in place of multiple if/else statements in some programming languages. The 4:1 multiplexer can be rewritten with selected signal assignment as follows:

```
architecture synth2 of mux4 is
begin
  with a select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
end;
```

full adder (using logical operators)

```verilog
assign p = a ^ b;
assign g = a & b;
assign s = p ^ cin;
assign cout = g | (p & cin);
```

```vhdl
p <= a xor b;
g <= a and b;
s <= p xor cin;
cout <= g or (p and cin);
```



full adder

Verilog

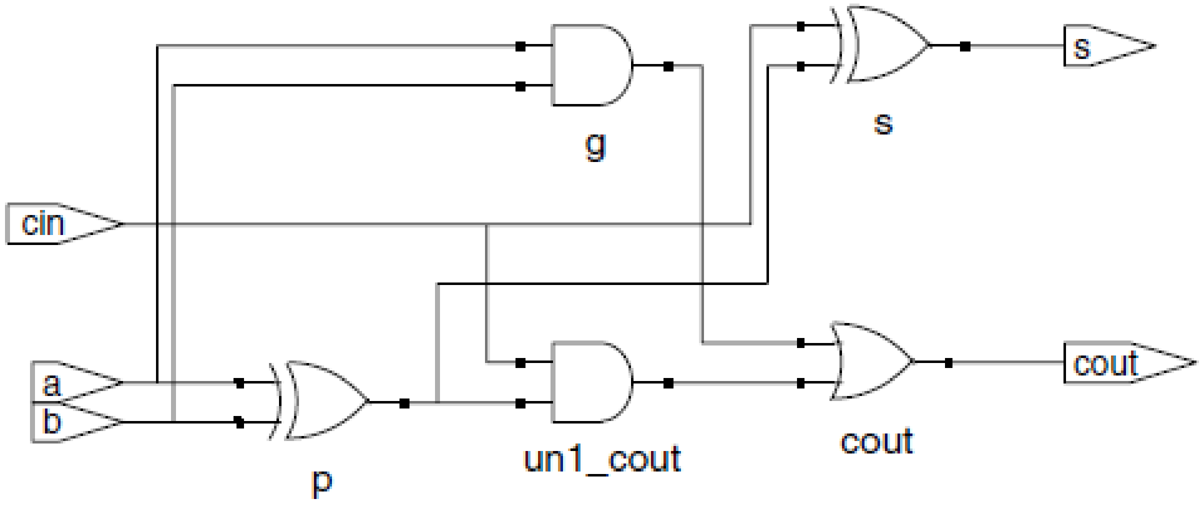
In Verilog, *wires* are used to represent internal variables whose values are defined by `assign` statements such as `assign p = a ^ b;` Wires technically have to be declared only for multibit busses, but it is good practice to include them for all internal variables; their declaration could have been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

VHDL

In VHDL, *signals* are used to represent internal variables whose values are defined by *concurrent signal assignment statements* such as `p <= a xor b;`

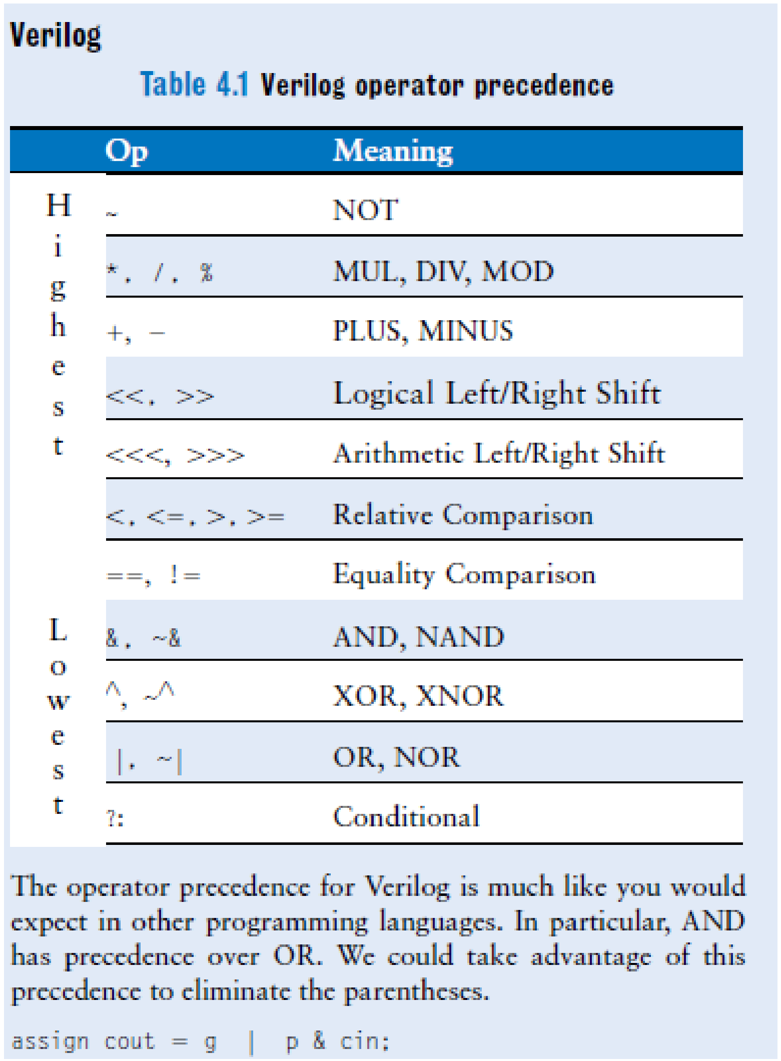
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;

  s <= p xor cin;
  cout <= g or (p and cin);
end;
```



operator precedence

```
assign cout = g | p & cin ;
// is equivalent to
assign  cout = g | (p & cin) ;
```

```
cout <= g or p and cin;
-- is equivalent to
cout <= (g or p) and cin;
```



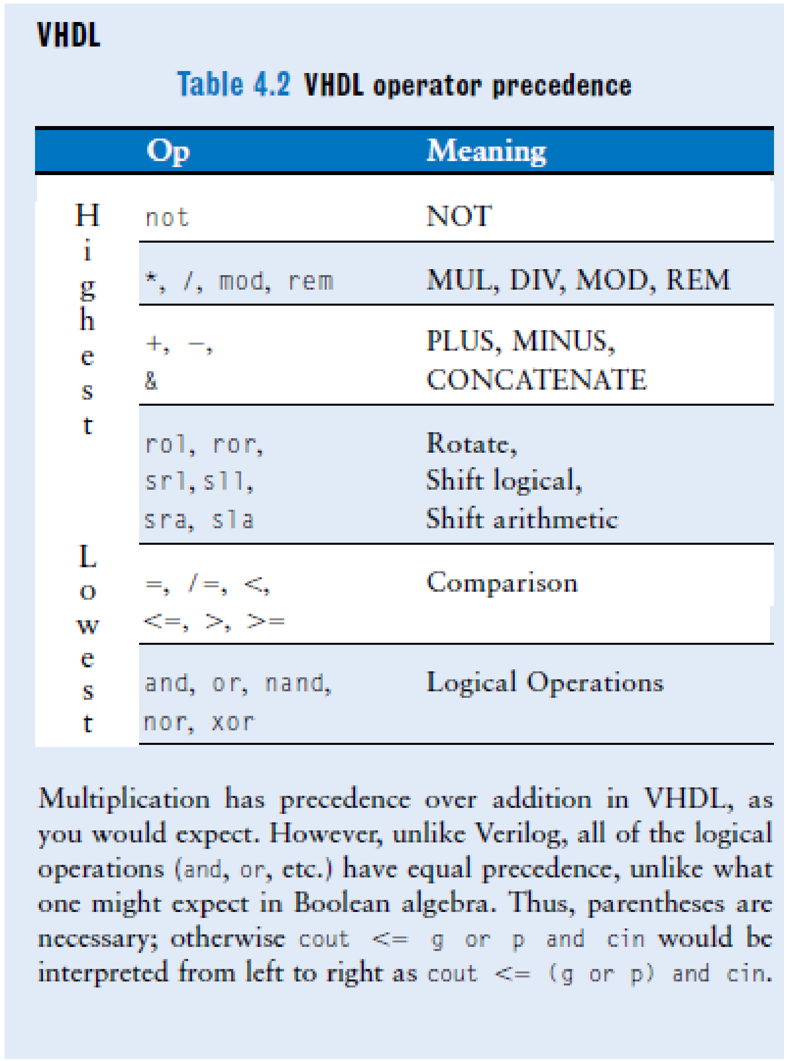
Verilog

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;
```



VHDL

Table 4.2 **VHDL operator precedence**

	Op	Meaning
Highest	not	NOT
	*, /, mod, rem	MUL, DIV, MOD, REM
	+, −, &	PLUS, MINUS, CONCATENATE
	rol, ror, srl, sll, sra, sla	Rotate, Shift logical, Shift arithmetic
	=, /=, <, <=, >, >=	Comparison
Lowest	and, or, nand, nor, xor	Logical Operations

Multiplication has precedence over addition in VHDL, as you would expect. However, unlike Verilog, all of the logical operations (and, or, etc.) have equal precedence, unlike what one might expect in Boolean algebra. Thus, parentheses are necessary; otherwise `cout <= g or p and cin` would be interpreted from left to right as `cout <= (g or p) and cin`.

numbers

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

VHDL

In VHDL, STD_LOGIC numbers are written in binary and enclosed in single quotes: '0' and '1' indicate logic 0 and 1.

STD_LOGIC_VECTOR numbers are written in binary or hexadecimal and enclosed in double quotation marks. The base is binary by default and can be explicitly defined with the prefix `X` for hexadecimal or `B` for binary.

Table 4.4 VHDL numbers

Numbers	Bits	Base	Val	Stored
"101"	3	2	5	101
B"101"	3	2	5	101
X"AB"	8	16	161	10101011

tri-state buffer (using continuous assignment/concurrent code)

// Verilog
assign y = en ? a : 4**’b**z;

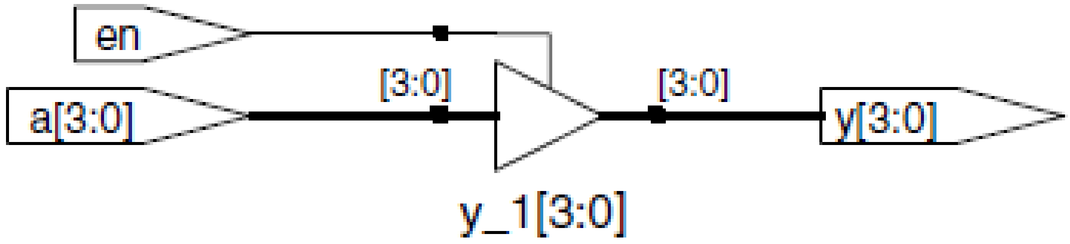
-- VHDL
y <= **“**zzzz**”** **when** en = ‘0’ **else** a;
-- y <= (others => ‘z’) wjen en = ‘0’
 --else a;

Verilog

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

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

VHDL

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

entity tristate is
  port(a:   in  STD_LOGIC_VECTOR(3 downto 0);
       en:  in  STD_LOGIC;
       y:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of tristate is
begin
  y <= "ZZZZ" when en = '0' else a;
end;
```



truth table

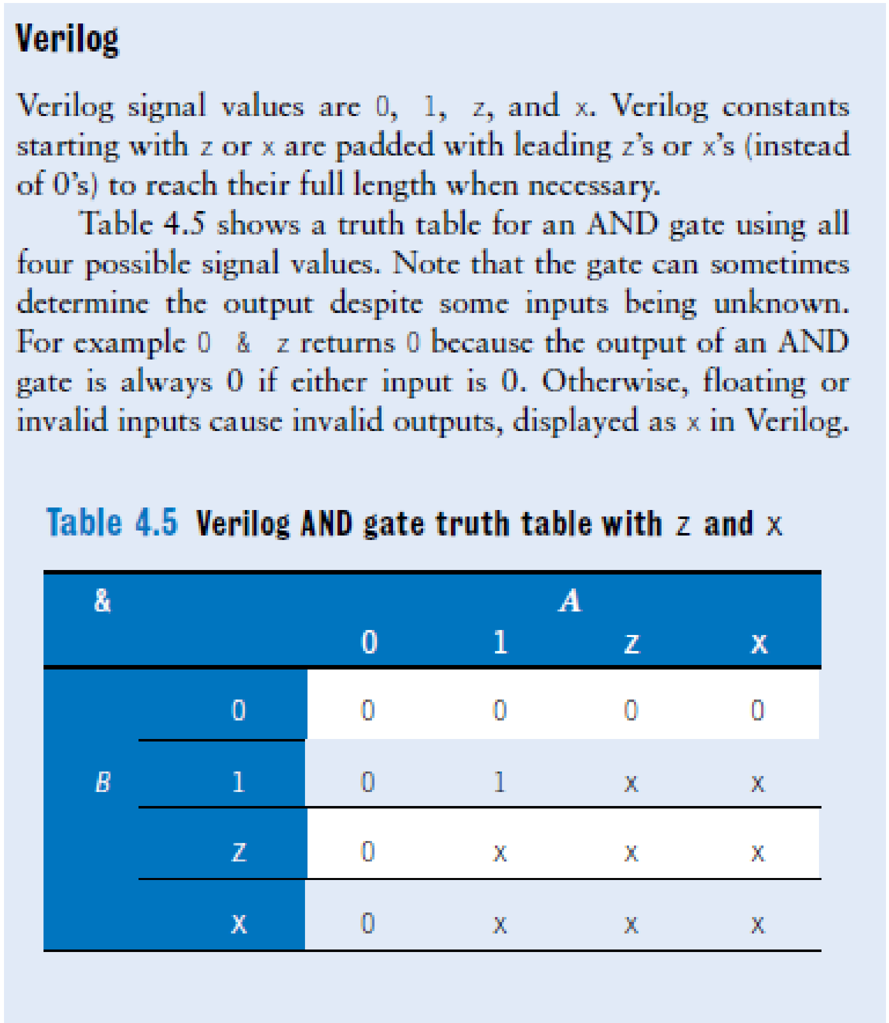
// Verilog

x is unknown

-- VHDL

'x' is unknown

'u' is un-initialized



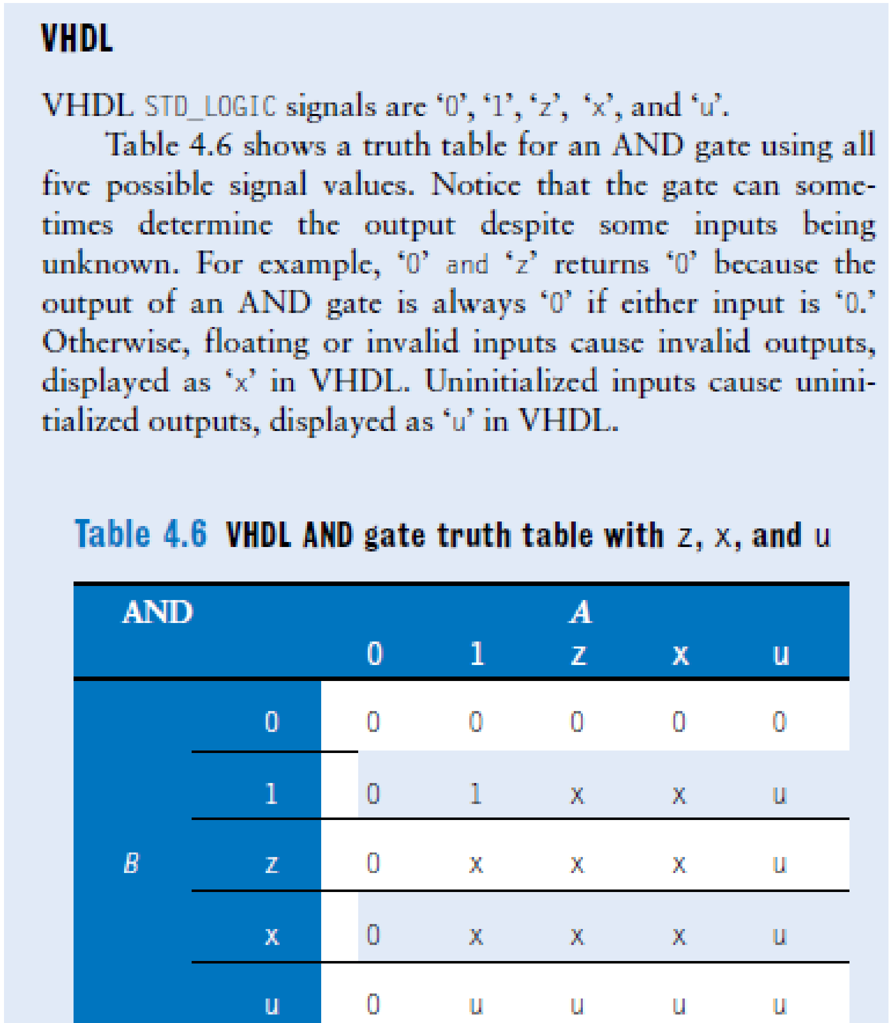
Verilog

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



VHDL

VHDL STD_LOGIC signals are '0', '1', 'z', 'x', and 'u'.

Table 4.6 shows a truth table for an AND gate using all five possible signal values. Notice that the gate can sometimes determine the output despite some inputs being unknown. For example, '0' and '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 VHDL. Uninitialized inputs cause uninitialized outputs, displayed as 'u' in VHDL.

Table 4.6 VHDL AND gate truth table with z, x, and u

AND		A				
		0	1	z	x	u
B	0	0	0	0	0	0
	1	0	1	x	x	u
	z	0	x	x	x	u
	x	0	x	x	x	u
	u	0	u	u	u	u

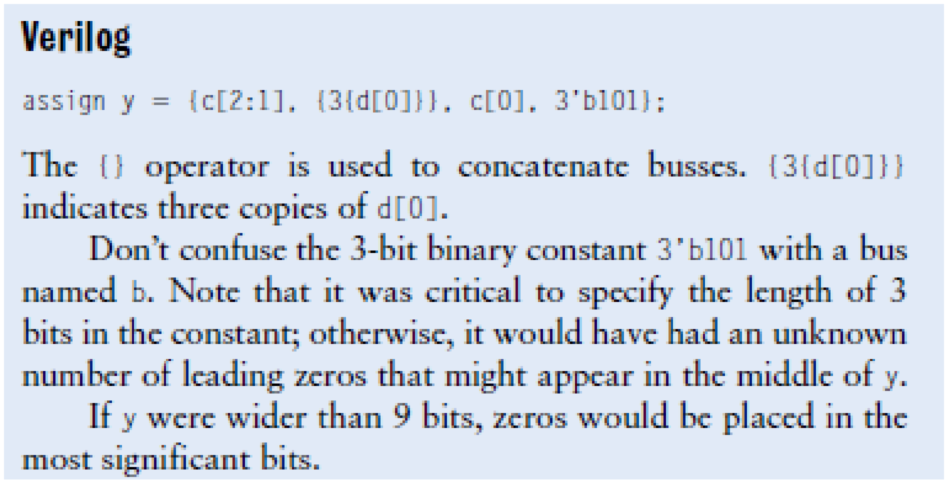
concatenation

Verilog

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

VHDL

```
y <=
c(2 downto 1) & d(0) & d(0) & d(0)
    & c(0) & "101";
-- y <= ( c(2 downto 0), d(0), d(0),
    d(0), c(0), "101" );
```



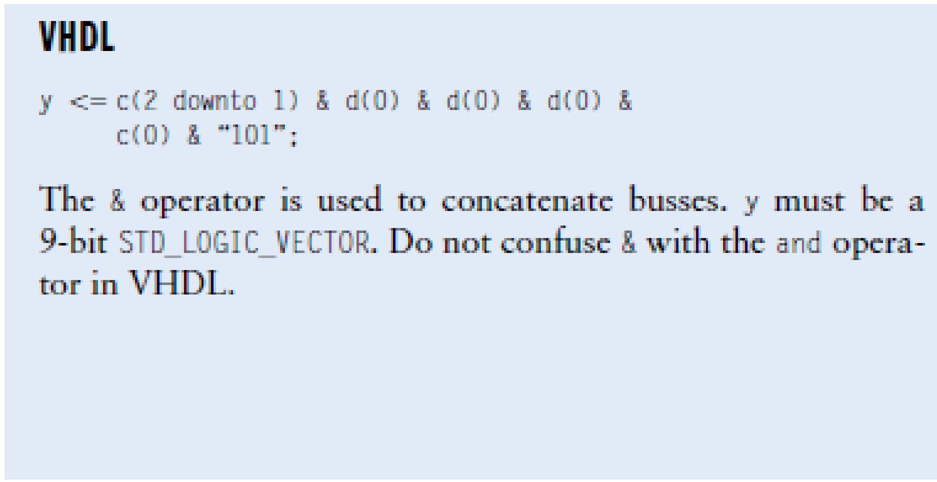
Verilog

```
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.



VHDL

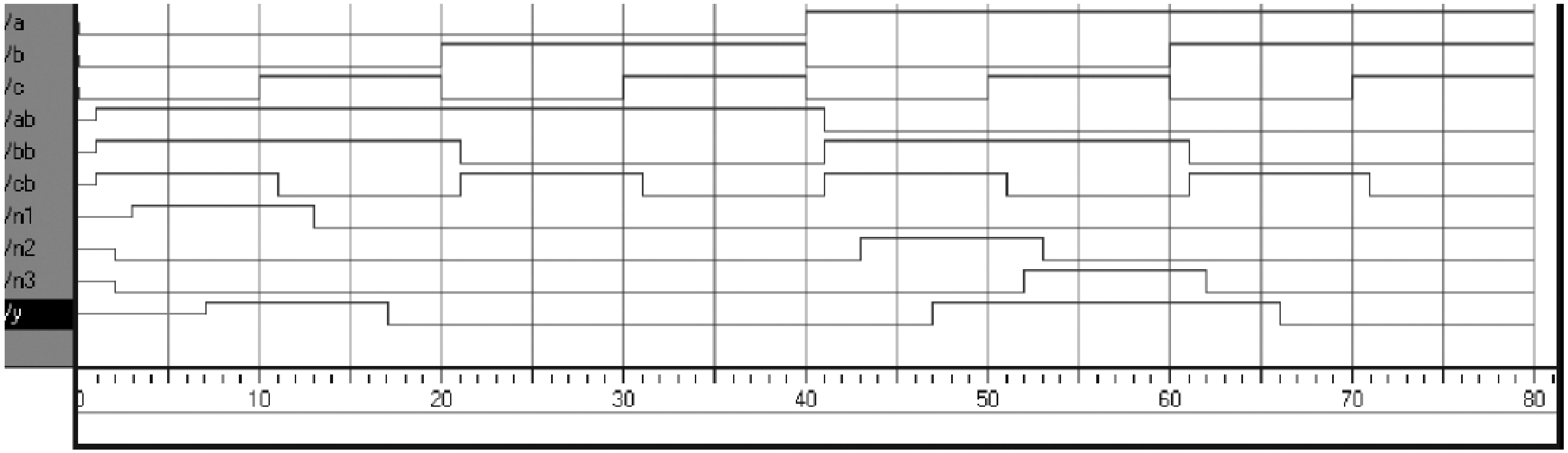
```
y <= c(2 downto 1) & d(0) & d(0) & d(0) &
     c(0) & "101";
```

The & operator is used to concatenate busses. y must be a 9-bit STD_LOGIC_VECTOR. Do not confuse & with the and operator in VHDL.

delay

```
`timescale unit/precision
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;
```

```
ab <= not a after 1ns;
bb <= not b after 1ns
cb <= not  c after 1ns;
n1 <= ab and bb and cb after 2ns;
n2 <= a and b and cb after 2ns;
n3 <= a and bb and c after 2ns;
y <= n1 or n2 or n3 after 4ns;
```



delay

Verilog

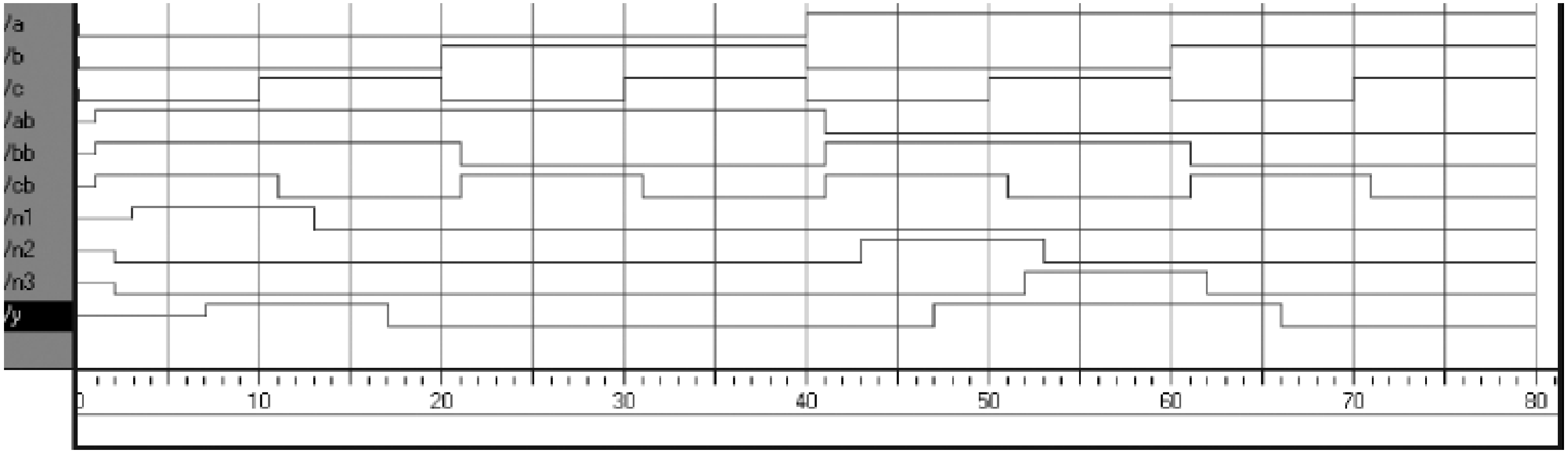
```
`timescale 1ns/1ps

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.

VHDL

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

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

architecture synth of example is
signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
  ab <= not a after 1 ns;
  bb <= not b after 1 ns;
  cb <= not c after 1 ns;
  n1 <= ab and bb and cb after 2 ns;
  n2 <= a and bb and cb after 2 ns;
  n3 <= a and bb and c after 2 ns;
  y <= n1 or n2 or n3 after 4 ns;
end;
```

In VHDL, the `after` clause is used to indicate delay. The units, in this case, are specified as nanoseconds.



Type in VHDL

```
-- illegal VHDL 93, but legal in VHDL 2008
-- Boolean true IS NOT  std_logic ‘1’
y <= d1 when sel else
      d0;
q <= (state = S2);


-- corrected VHDL
y <= d1 when (sel = ‘1’) else
      d0;
q <= 1 when (state = S2) else
       0;
```

VHDL "BUFFER" port

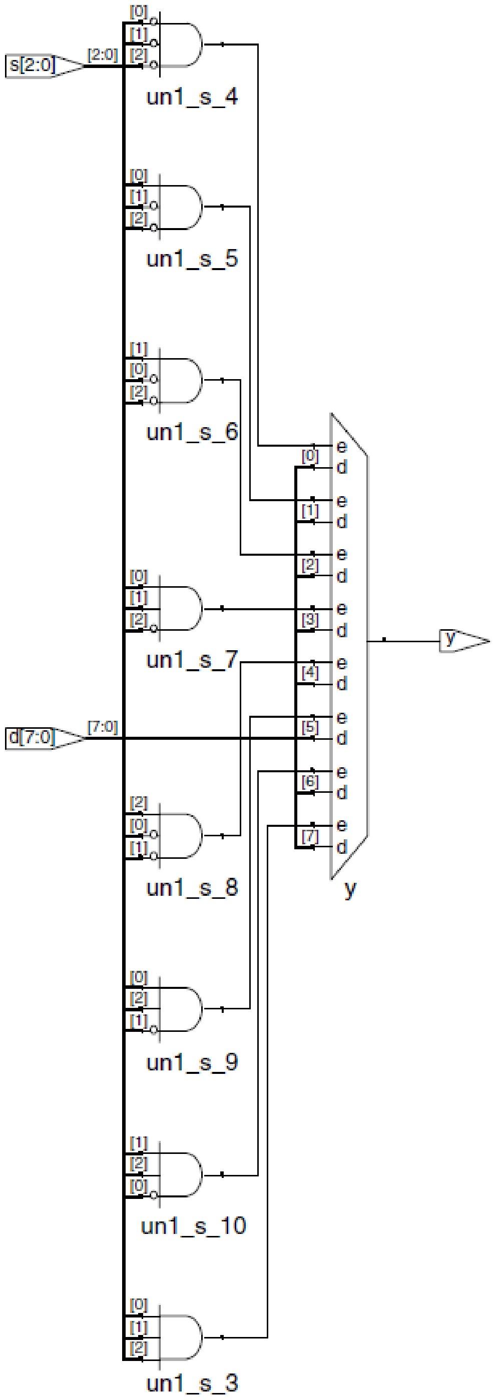
- unlike **INOUT** which is bidirectional port, **BUFFER** is output port, but also used internally

```
library ieee;
use ieee.std_logic_1164.all;

-- illegal port for v
entity and23 is
  port (a, b, c: IN std_logic;
            v, w: OUT std_logic);
end;

architecture of synth of and23 is
begin
  v <= a and b;
  w <= v and c;
end
```

```
library ieee;
use ieee.std_logic_1164.all;

-- legal port for v
entity and23 is
  port (a, b, c: IN std_logic;
              v : BUFFER std_logic;
               w: OUT std_logic);
end;

architecture of synth of and23 is
begin
  v <= a and b;    -- v is an output port
  w <= v and c;    -- v is also used internally
end
```

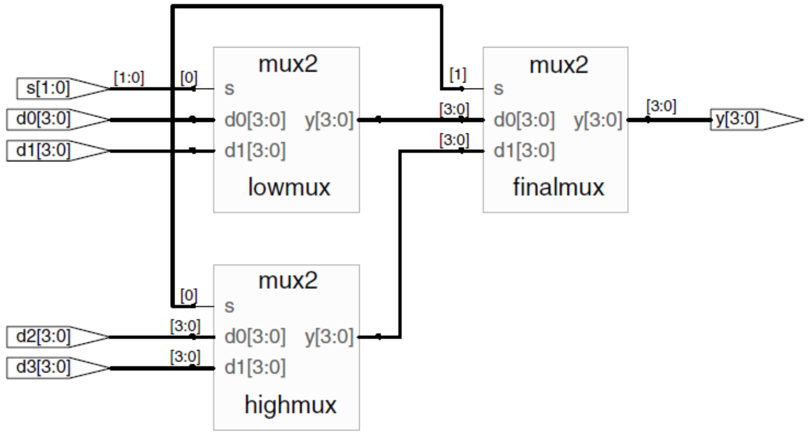
MUX8 with type conversion

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity mux8 is
port ( d: in std_logic_vector(7 downto 0);
        s: in std_logic_vector (2 downto 0);
        y: out std_logic);
end;

architecture synth of mux8 is
begin
  y<= d( conv_integer(s) );
end
```



structural modeling (MUX4 based on MUX2)

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



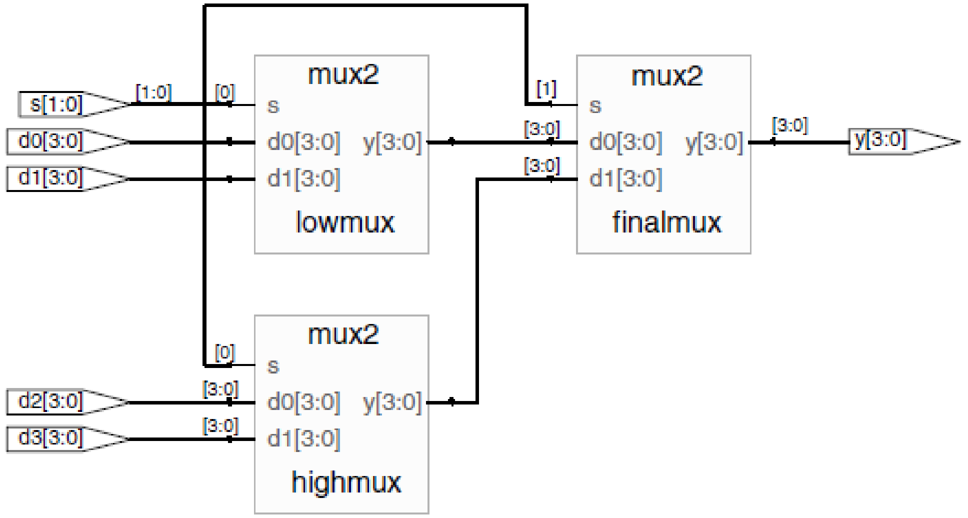
```
component mux2
port (d0, d1: in std_logic_vector(3 downto 0);
            s : in std_logic;
            y : out std_logic_vector (3 downto 0));
end component;

…
lowmux : mux2 port map (d0, d1, s(0), low);
//lowmux: mux2 port map(d0=>d0, d1=>d1, s=>s(0), y=>low);
highmux: mux2 port map (d2, d3, s(0), high);
//highmux: mux2 port map (d0=>d2, d1=>d3, s=>s(0),
y=>high);
finalmux: mux2 port map (low, high, s[1], y);
// finalmux: mux2 port map (d0=>low, d1=>high, s=>s(1),
y=>y);
…
```

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.

structural model of MUX4



VHDL

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

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux4 is
  component mux2
    port(d0,
         d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
  lowmux:   mux2 port map(d0, d1, s(0), low);
  highmux:  mux2 port map(d2, d3, s(0), high);
  finalmux: mux2 port map(low, high, s(1), y);
end;
```

The architecture must first declare the `mux2` ports using the `component` declaration statement. This allows VHDL tools to check that the component you wish to use has the same ports as the entity that was declared somewhere else in another entity statement, preventing errors caused by changing the entity but not the instance. However, component declaration makes VHDL code rather cumbersome.

Note that this architecture of `mux4` was named `struct`, whereas architectures of modules with behavioral descriptions from Section 4.2 were named `synth`. VHDL allows multiple architectures (implementations) for the same entity; the architectures are distinguished by name. The names themselves have no significance to the CAD tools, but `struct` and `synth` are common. Synthesizable VHDL code generally contains only one architecture for each entity, so we will not discuss the VHDL syntax to configure which architecture is used when multiple architectures are defined.

structural model of MUX2 based on tri-state buffers

Verilog

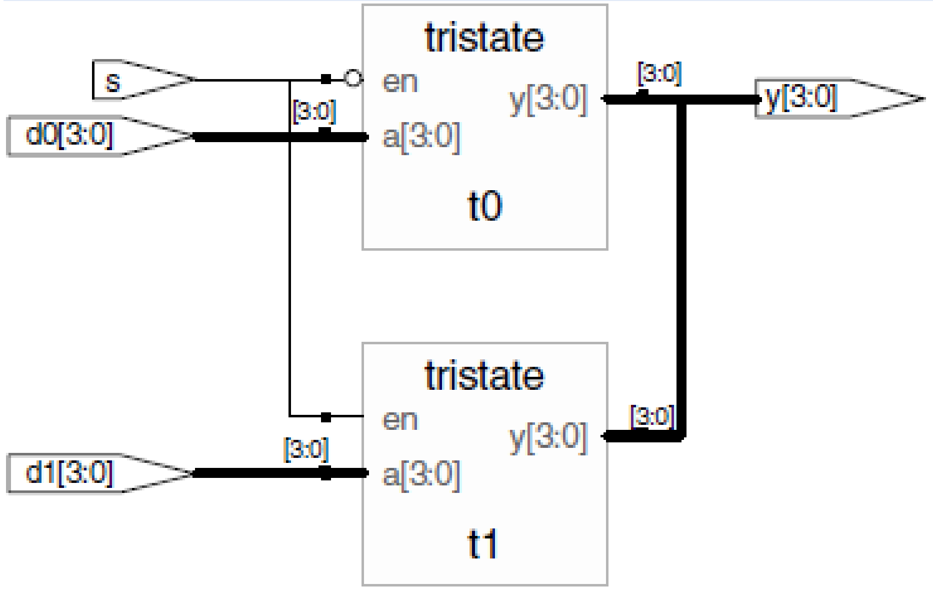
```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

  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.

VHDL

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

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux2 is
  component tristate
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal sbar: STD_LOGIC;
begin
  sbar <= not s;
  t0: tristate port map(d0, sbar, y);
  t1: tristate port map(d1, s, y);
end;
```

In VHDL, expressions such as `not s` are not permitted in the port map for an instance. Thus, `sbar` must be defined as a separate signal.



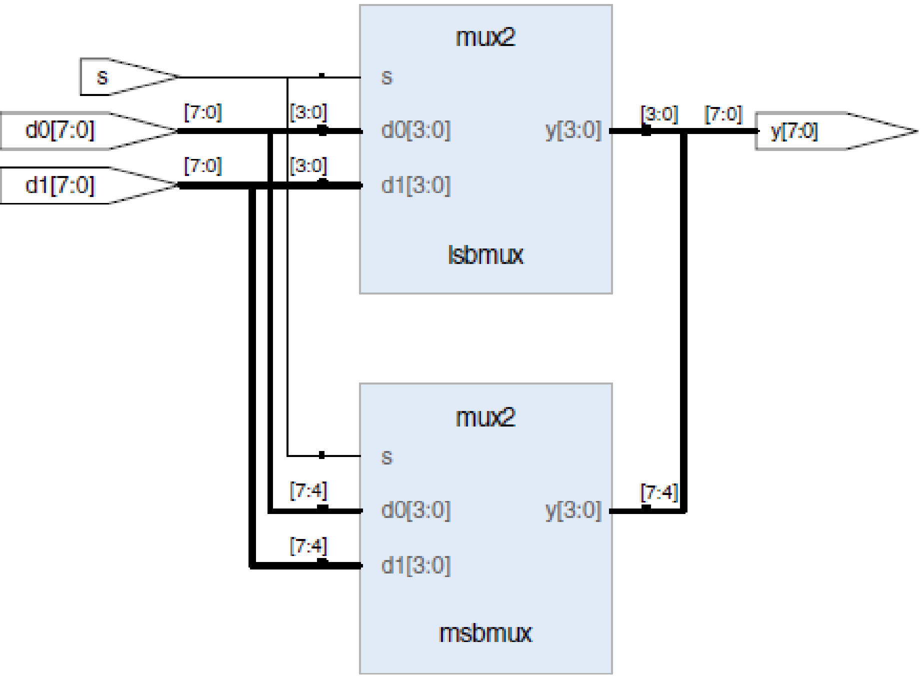
accessing parts of buses

Verilog

```
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
```

VHDL

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

entity mux2_8 is
  port(d0, d1: in  STD_LOGIC_VECTOR(7 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux2_8 is
  component mux2
    port(d0, d1: in   STD_LOGIC_VECTOR(3
                               downto 0);
         s:      in   STD_LOGIC;
         y:      out  STD_LOGIC_VECTOR(3 downto 0));
  end component;
begin

  lsbmux:  mux2
    port map(d0(3 downto 0), d1(3 downto 0),
             s, y(3 downto 0));
  msbhmux: mux2
    port map(d0(7 downto 4), d1(7 downto 4),
             s, y(7 downto 4));
end;
```



data types (vectors)

- Vectors: multiple bit widths (default is scalar)

```
wire [7:0] bus;        // 8-bit signal, bus[7] is the MSB, bus[0] is the LSB
wire [31:0] word;        // 32-bit word
reg [255:0] data;      // 256-bit word register, data[255] is MSB
reg [0:40] v_addr;  // v_addr[0] is MSB
```

- Vector Part Select

```
word[7:0]    // the least significant byte of the 32-bit vector word
v_addr[0:1]  // two most significant bits of the vector v_addr
```

- variable vector part select

  [<starting_bit> **+:** width**]** : part-select *increments*

  [<starting_bit> **-:** width**]** : part-select *decrements*

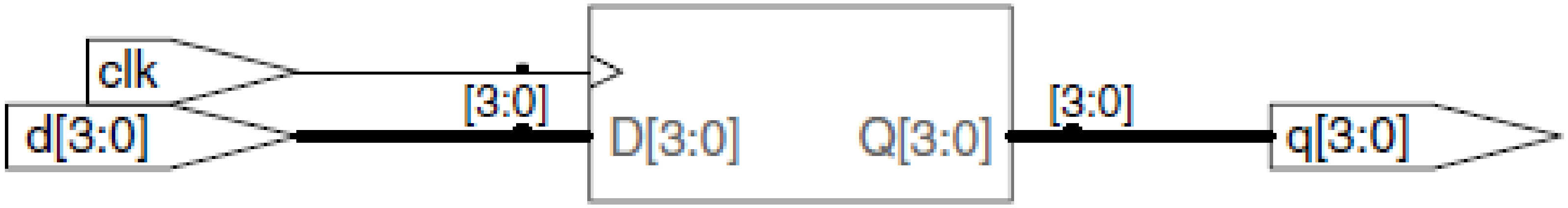
  starting bit can be varied, but the width must be constant

```
eg1.: reg [255:0] data1;
      reg [0:255] data2;
      reg [7:0] byte;

      byte=data1[31-:8]  // data1[31:24]
      byte=data1[24+:8] // data1[31:24]
      byte=data2[31-:8]  // data2[24:31]
      byte=data2[24+:8] // data2[24:31]

eg2.  reg [255:0] data1;
      for (j=0; j<=31; j=j+1) byte = data1[(j*8)+:8] ;
      // sequence is [7:0], [15:8], …, [255:248]
```

register

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

```
process (clk) begin
if (clk'event and clk='1' ) then
-- if ( rising_edge (clk) ) then
  q <= d;
end if;
end process;
```



register

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`.

<= 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.

VHDL

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

entity flop is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of flop is
begin
  process(clk) begin
    if clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end;
```

A VHDL `process` is written in the form

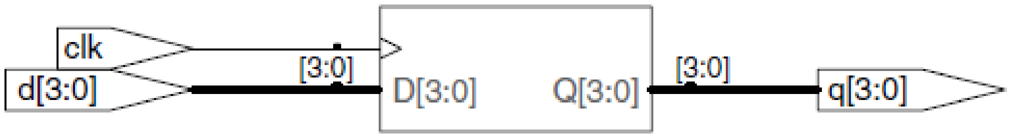
```
process(sensitivity list) begin
  statement;
end process;
```

The statement is executed when any of the variables in the `sensitivity list` change. In this example, the `if` statement is executed when `clk` changes, indicated by `clk'event`. If the change is a rising edge (`clk = '1'` after the event), then `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`.

An alternative VHDL idiom for a flip-flop is

```
process(clk) begin
  if RISING_EDGE(clk) then
    q <= d;
  end if;
end process;
```

`RISING_EDGE(clk)` is synonymous with `clk'event and clk = 1`.



resettable register

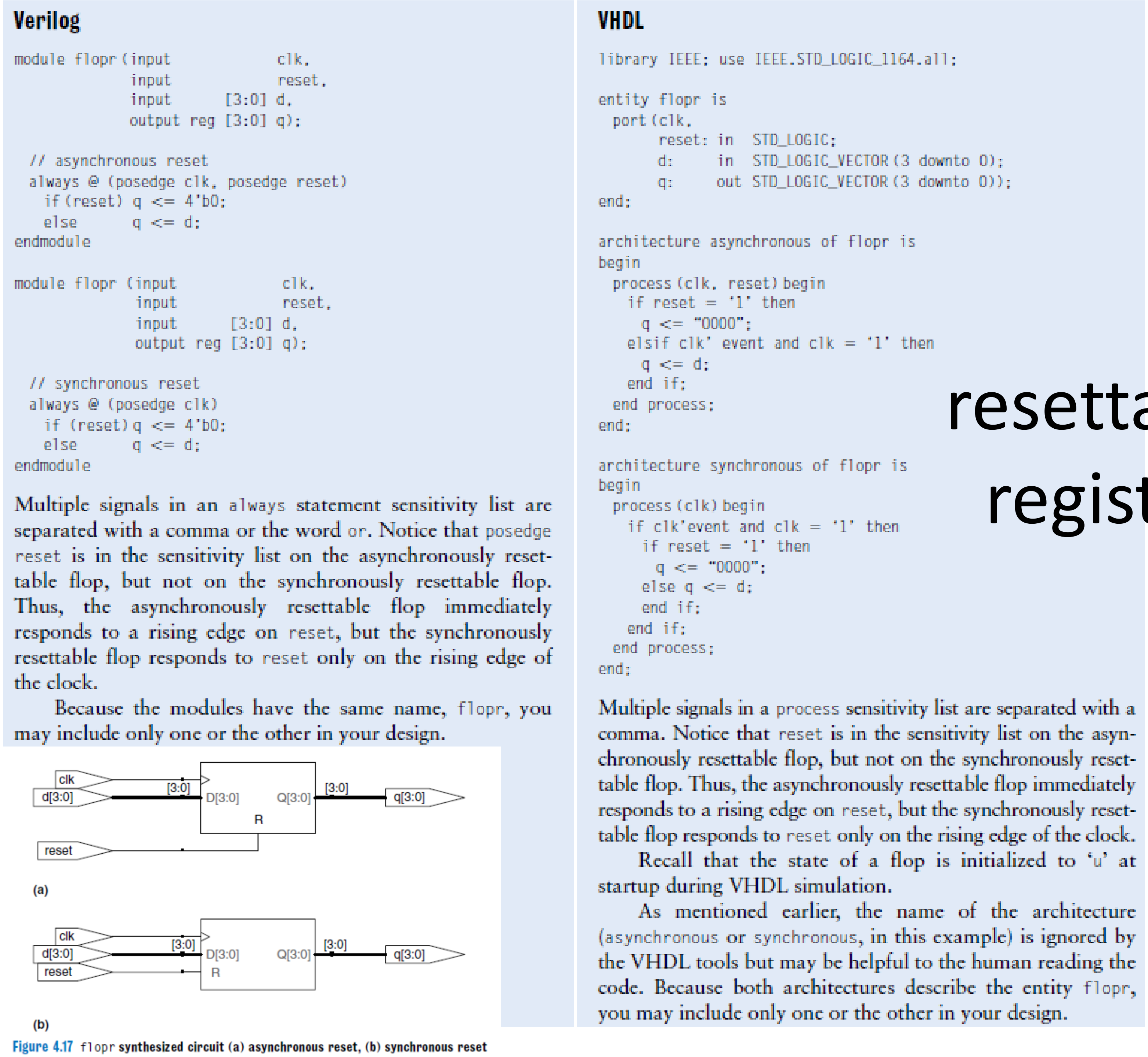
```verilog
// asynchronous reset
always @ (posedge clk, posedge reset)
  if (reset) q <= 4’b0;
  else q <= d;
```

```vhdl
-- asynchronous reset
process (clk, reset)
begin
      if (reset = ‘1’) then
        q <= “0000”;
      elsif (clk’event and clk=‘1’ ) then
        q <= d;
      end if;
end process;
```

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

```vhdl
-- synchronous reset
process (clk)
begin
      if (clk’event and clk=‘1’ ) then
        if (reset = ‘1’) then q <= “0000”;
        else q <= d;
      end if;
end process;
```

resettable register



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.

VHDL

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

entity flopr is
  port(clk,
       reset: in  STD_LOGIC;
       d:     in  STD_LOGIC_VECTOR(3 downto 0);
       q:     out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset = '1' then
      q <= "0000";
    elsif clk' event and clk = '1' then
      q <= d;
    end if;
  end process;
end;

architecture synchronous of flopr is
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then
        q <= "0000";
      else q <= d;
      end if;
    end if;
  end process;
end;
```

Multiple signals in a `process` sensitivity list are separated with a comma. Notice that `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.

Recall that the state of a flop is initialized to 'u' at startup during VHDL simulation.

As mentioned earlier, the name of the architecture (`asynchronous` or `synchronous`, in this example) is ignored by the VHDL tools but may be helpful to the human reading the code. Because both architectures describe the entity `flopr`, you may include only one or the other in your design.

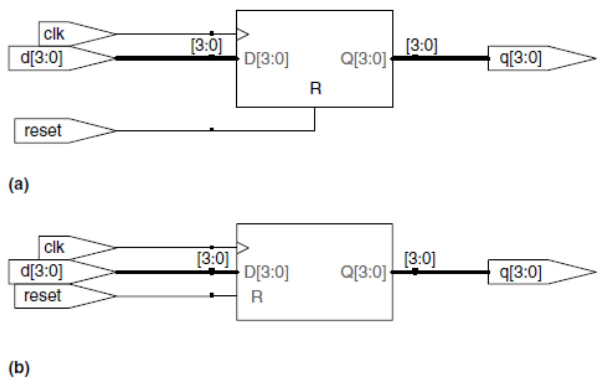
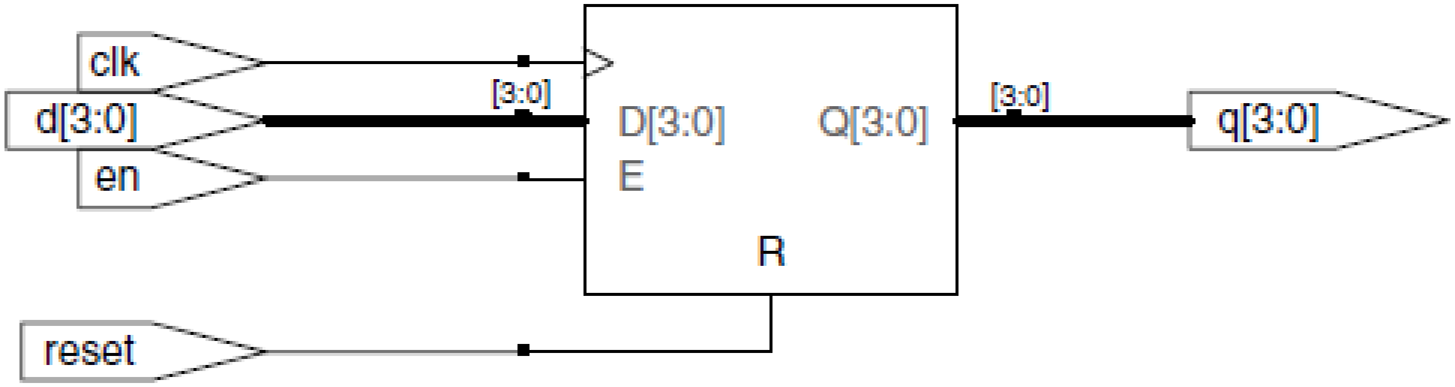


Figure 4.17 flopr synthesized circuit (a) asynchronous reset, (b) synchronous reset

enabled register

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

```
-- asynchronous reset
process (clk, reset)
begin
    if (reset = '1') then
      q <= "0000";
    elsif (clk'event and clk='1' ) then
      if (en = '1') then q <= d; end if;
    end if;
end process;
```



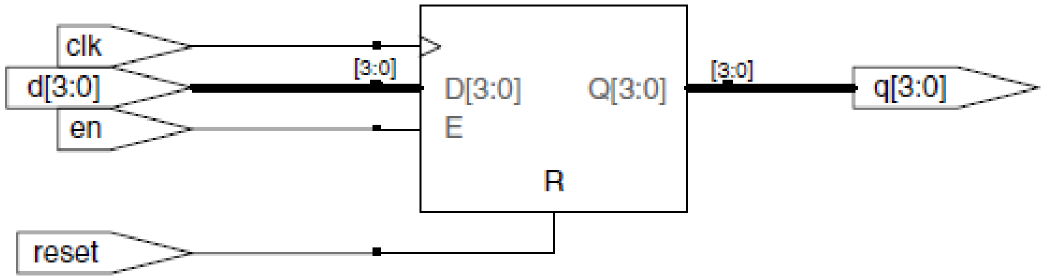
enabled register

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
```

VHDL

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

entity flopenr is
  port (clk,
        reset,
        en: in  STD_LOGIC;
        d:  in  STD_LOGIC_VECTOR(3 downto 0);
        q:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopenr is
-- asynchronous reset
begin
  process (clk, reset) begin
    if reset = '1' then
      q <= "0000";
    elsif clk'event and clk = '1' then
      if en = '1' then
        q <= d;
      end if;
    end if;
  end process;
end;
```



2-bit shift register

Verilog

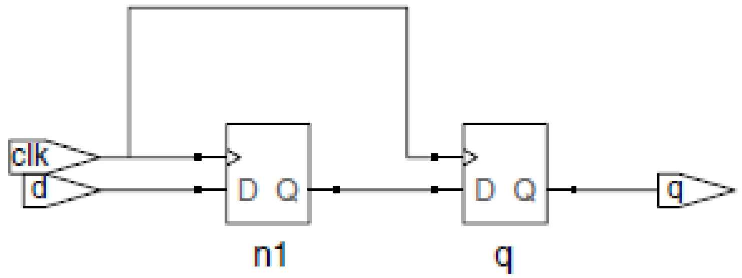
```
module sync(input       clk,
            input       d,
            output reg q);

  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.

VHDL

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

entity sync is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture good of sync is
  signal n1: STD_LOGIC;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      n1 <= d;
      q <= n1;
    end if;
  end process;
end;
```

n1 must be declared as a signal because it is an internal signal used in the module.



quzz: long shift registers

- use one line of code

```
{internal[1:1023], out} <= {d, internal[1:1023]};
```

- use for loop

```
internal[1] <= d;
for (i=2; i<=1023; i= i+1)   internal[i] <= internal [i-1];
out <= internal[1023];
```

- use pointers pointing to the registers to be written and to be read, and clock-gating other registers

D-latch

Verilog

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

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

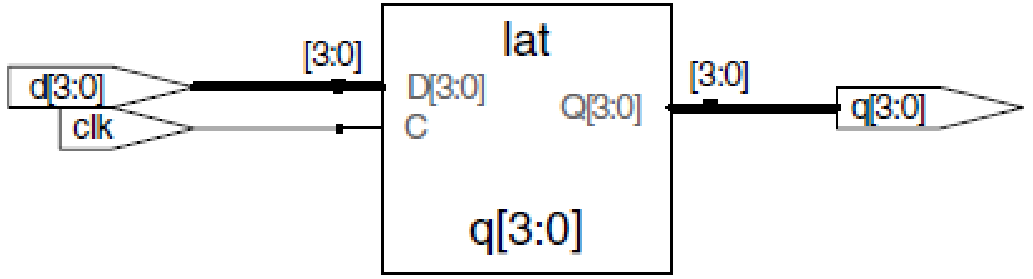
The sensitivity list contains both clk and d, so the always statement evaluates any time clk or d changes. If clk is HIGH, d flows through to q.

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.

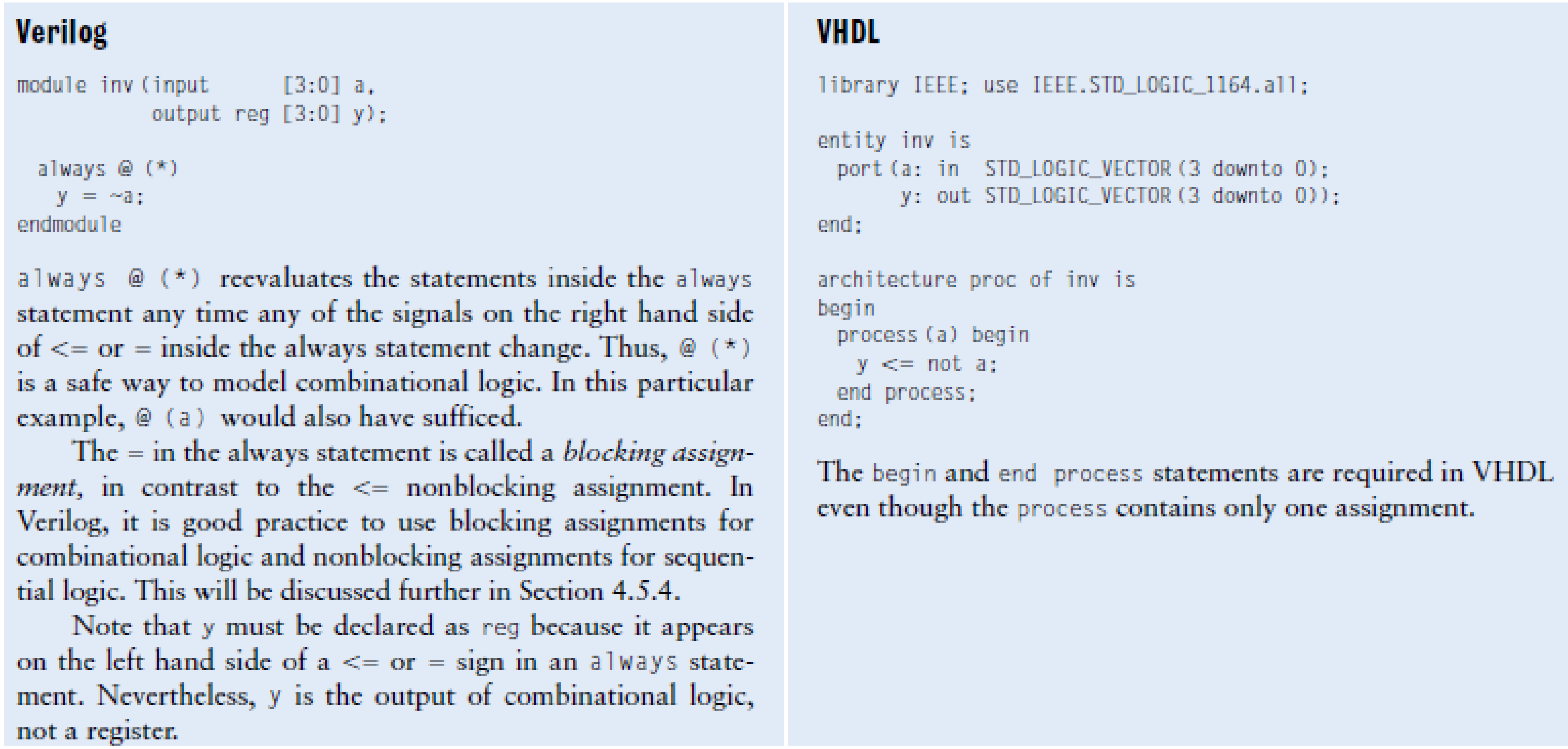
VHDL

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

entity latch is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of latch is
begin
  process(clk, d) begin
    if clk = '1' then q <= d;
    end if;
  end process;
end;
```

The sensitivity list contains both clk and d, so the process evaluates anytime clk or d changes. If clk is HIGH, d flows through to q.



inverter using 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.

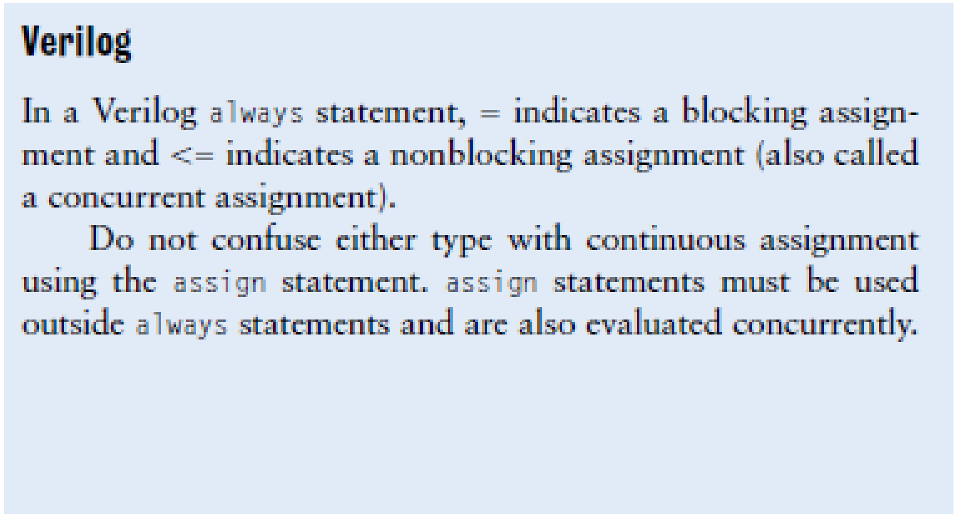
VHDL

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

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture proc of inv is
begin
  process(a) begin
    y <= not a;
  end process;
end;
```

The `begin` and `end process` statements are required in VHDL even though the `process` contains only one assignment.

blocking/non-blocking assignment (Verilog) signal/variable assignment (in VHDL)

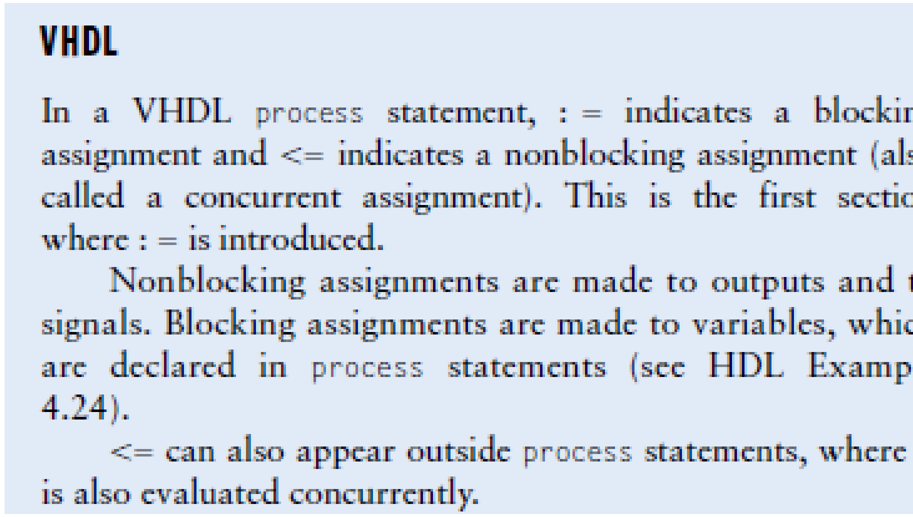
- for simple combinational logic (CL)
  - use use continuous assignment (concurrent code)
- for more complex CL
  - use blocking assignment in sequential codes
- for sequential logic
  - use non-blocking assignment



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.



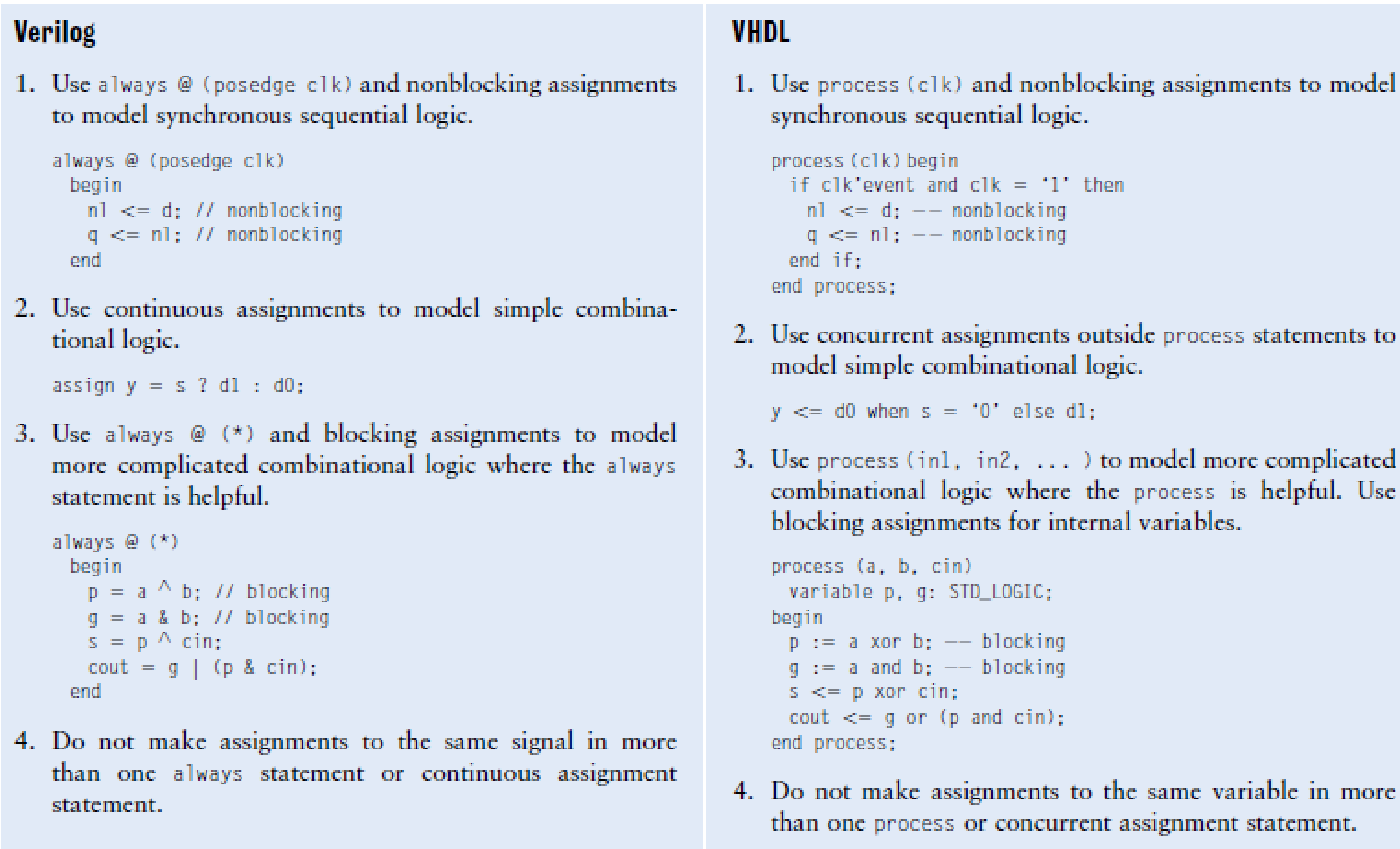
VHDL

In a VHDL `process` statement, : = indicates a blockir assignment and <= indicates a nonblocking assignment (als called a concurrent assignment). This is the first sectic where : = is introduced.

Nonblocking assignments are made to outputs and t signals. Blocking assignments are made to variables, whic are declared in `process` statements (see HDL Examp 4.24).

<= can also appear outside `process` statements, where is also evaluated concurrently.

blocking/non-blocking assignment (Verilog) signal/variable assignment (in VHDL)



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.

VHDL

1. Use `process (clk)` and nonblocking assignments to model synchronous sequential logic.

```
process (clk) begin
  if clk'event and clk = '1' then
    n1 <= d; -- nonblocking
    q <= n1; -- nonblocking
  end if;
end process;
```

2. Use concurrent assignments outside `process` statements to model simple combinational logic.

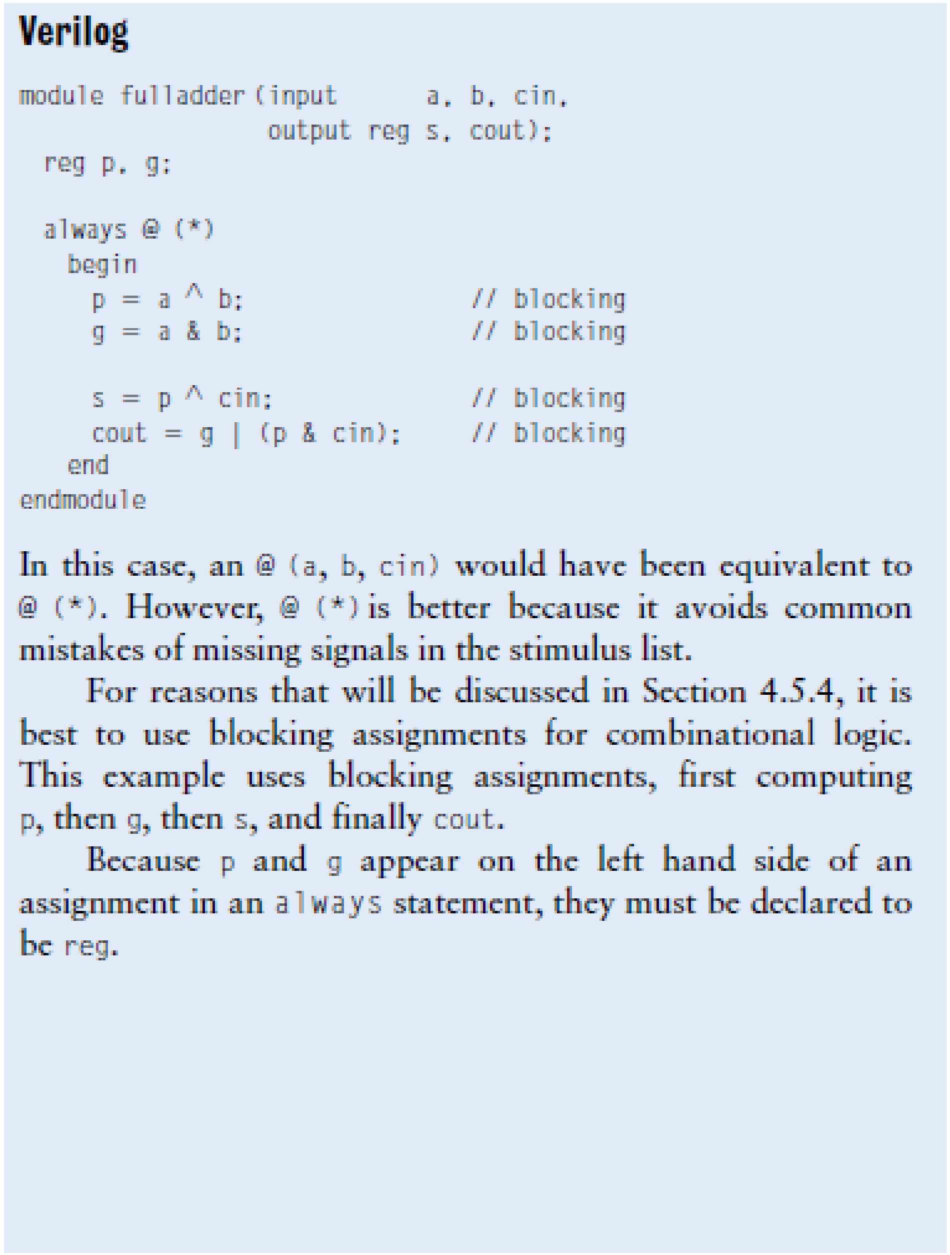
```
y <= d0 when s = '0' else d1;
```

3. Use `process (in1, in2, ... )` to model more complicated combinational logic where the `process` is helpful. Use blocking assignments for internal variables.

```
process (a, b, cin)
  variable p, g: STD_LOGIC;
begin
  p := a xor b; -- blocking
  g := a and b; -- blocking
  s <= p xor cin;
  cout <= g or (p and cin);
end process;
```

4. Do not make assignments to the same variable in more than one `process` or concurrent assignment statement.

full adder using always/process



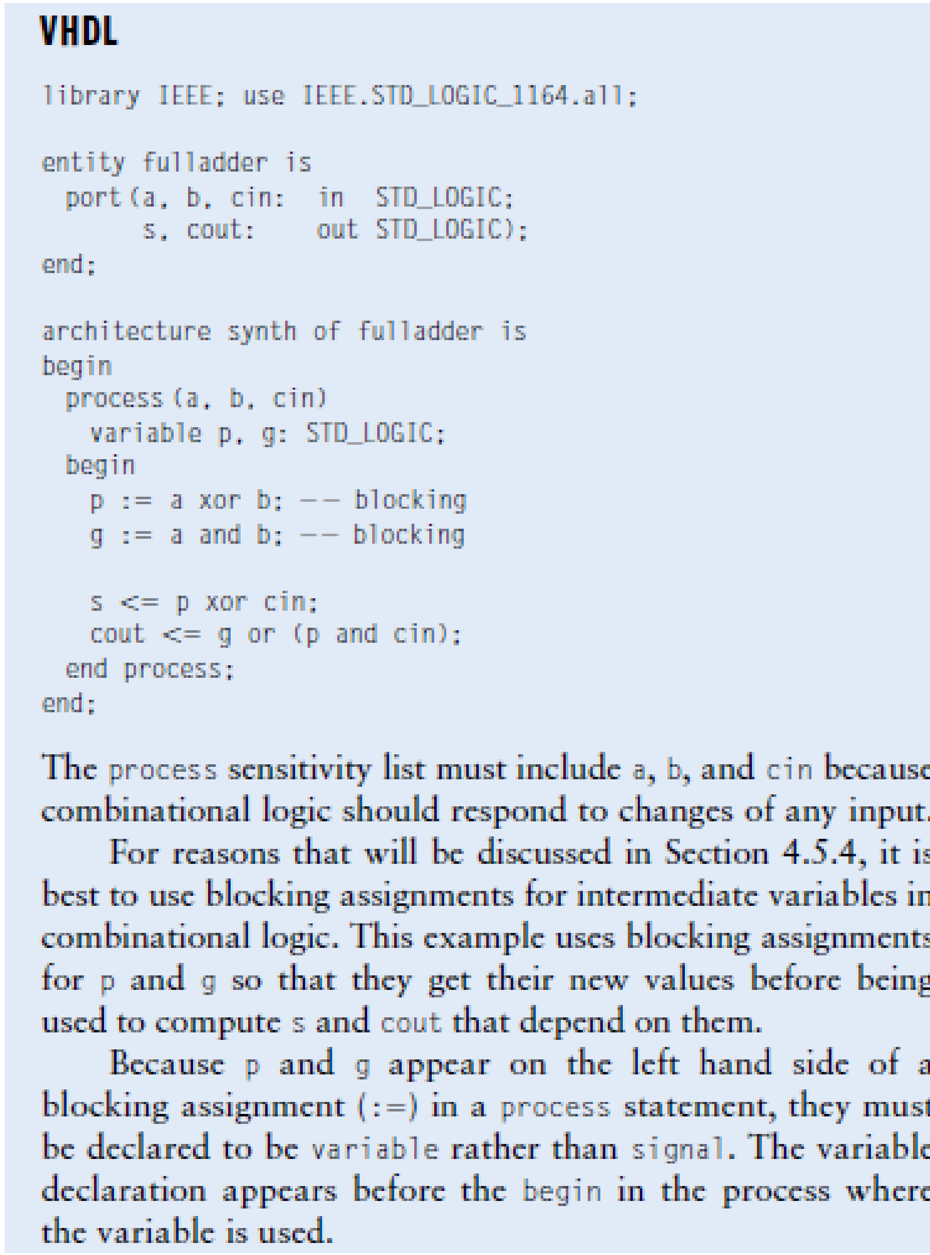
Verilog

```
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`.



VHDL

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

entity fulladder is
  port(a, b, cin:  in  STD_LOGIC;
       s, cout:    out STD_LOGIC);
end;

architecture synth of fulladder is
begin
  process(a, b, cin)
    variable p, g: STD_LOGIC;
  begin
    p := a xor b; -- blocking
    g := a and b; -- blocking

    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

The `process` sensitivity list must include `a`, `b`, and `cin` becaus
combinational logic should respond to changes of any input.

For reasons that will be discussed in Section 4.5.4, it i
best to use blocking assignments for intermediate variables i
combinational logic. This example uses blocking assignment
for `p` and `g` so that they get their new values before bein
used to compute `s` and `cout` that depend on them.

Because `p` and `g` appear on the left hand side of a
blocking assignment (`:=`) in a `process` statement, they mus
be declared to be `variable` rather than `signal`. The variabl
declaration appears before the `begin` in the process wher
the variable is used.

full adder using non-blocking assignment

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.

VHDL

```
-- nonblocking assignments (not recommended)
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port (a, b, cin: in  STD_LOGIC;
        s, cout:   out STD_LOGIC);
end;

architecture nonblocking of fulladder is
  signal p, g: STD_LOGIC;
begin
  process (a, b, cin, p, g) begin
    p <= a xor b; -- nonblocking
    g <= a and b; -- nonblocking

    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

Because p and g appear on the left hand side of a nonblocking assignment in a process statement, they must be declared to be signal rather than variable. The signal declaration appears before the begin in the architecture, not the process.

Verilog

If the sensitivity list of the always statement in HDL Example 4.29 were written as always @ (a, b, cin) rather than always @ (*), then the statement would not reevaluate when p or g changes. In the previous example, s would be incorrectly left at 0, not 1.

VHDL

If the sensitivity list of the process statement in HDL Example 4.29 were written as process (a, b, cin) rather than process (a, b, cin, p, g), then the statement would not reevaluate when p or g changes. In the previous example, s would be incorrectly left at 0, not 1.

full adder
using Verilog blocking and non-blocking assignments

```
// not good coding
// always block might be evaluated twice
module FA (input a, b, cin,  output reg s,
cout);
reg p, g;
always @  (*)
begin
  p <= a ^ b;
  g <= a & b;

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

```
// good coding
// always block be evaluated only once
module FA (input a, b, cin,  output reg s,
cout);
reg p, g;
always @  (*)
begin
  p = a ^ b;
  g = a & b;

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

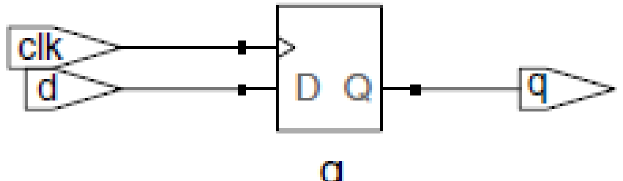
“bad” synchronizer with blocking assignment

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
```

VHDL

```
-- Bad implementation using blocking assignment

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity syncbad is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture bad of syncbad is
begin
  process(clk)
    variable n1: STD_LOGIC;
  begin
    if clk'event and clk = '1' then
      n1 := d; -- blocking
      q <= n1;
    end if;
  end process;
end;
```



case (7-segment display decoder)

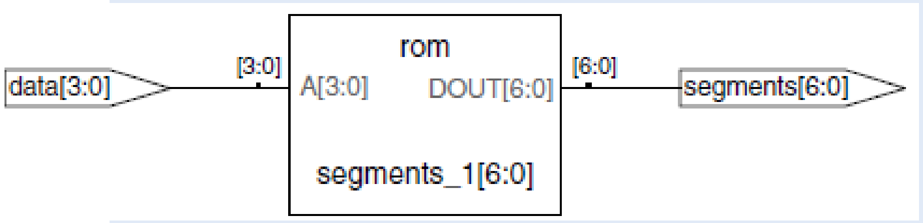
Verilog

```
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.



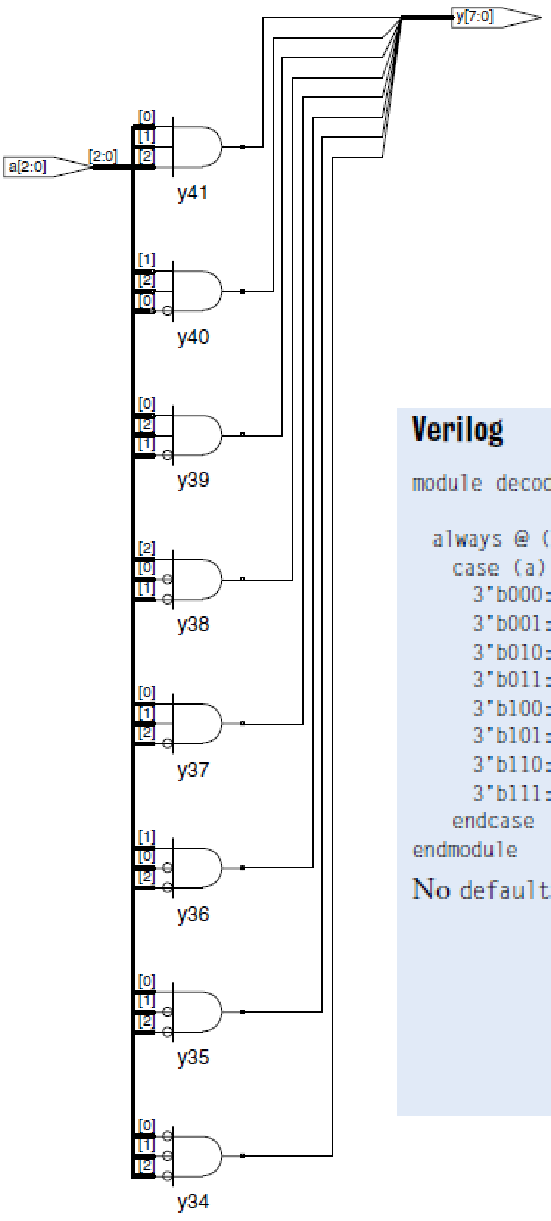
VHDL

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

entity seven_seg_decoder is
  port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_seg_decoder is
begin
  process(data) begin
    case data is
      --                              abcdefg
      when X"0" => segments <= "1111110";
      when X"1" => segments <= "0110000";
      when X"2" => segments <= "1101101";
      when X"3" => segments <= "1111001";
      when X"4" => segments <= "0110011";
      when X"5" => segments <= "1011011";
      when X"6" => segments <= "1011111";
      when X"7" => segments <= "1110000";
      when X"8" => segments <= "1111111";
      when X"9" => segments <= "1111011";
      when others => segments <= "0000000";
    end case;
  end process;
end;
```

The case statement checks the value of data. When data is 0, the statement performs the action after the =>, setting segments to 1111110. The case statement similarly checks other data values up to 9 (note the use of X for hexadecimal numbers). The others clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

Unlike Verilog, VHDL supports selected signal assignment statements (see HDL Example 4.6), which are much like case statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic.

decoder using case



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.

VHDL

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

entity decoder3_8 is
  port(a: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of decoder3_8 is
begin
  process(a) begin
    case a is
      when "000" => y <= "00000001";
      when "001" => y <= "00000010";
      when "010" => y <= "00000100";
      when "011" => y <= "00001000";
      when "100" => y <= "00010000";
      when "101" => y <= "00100000";
      when "110" => y <= "01000000";
      when "111" => y <= "10000000";
    end case;
  end process;
end;
```

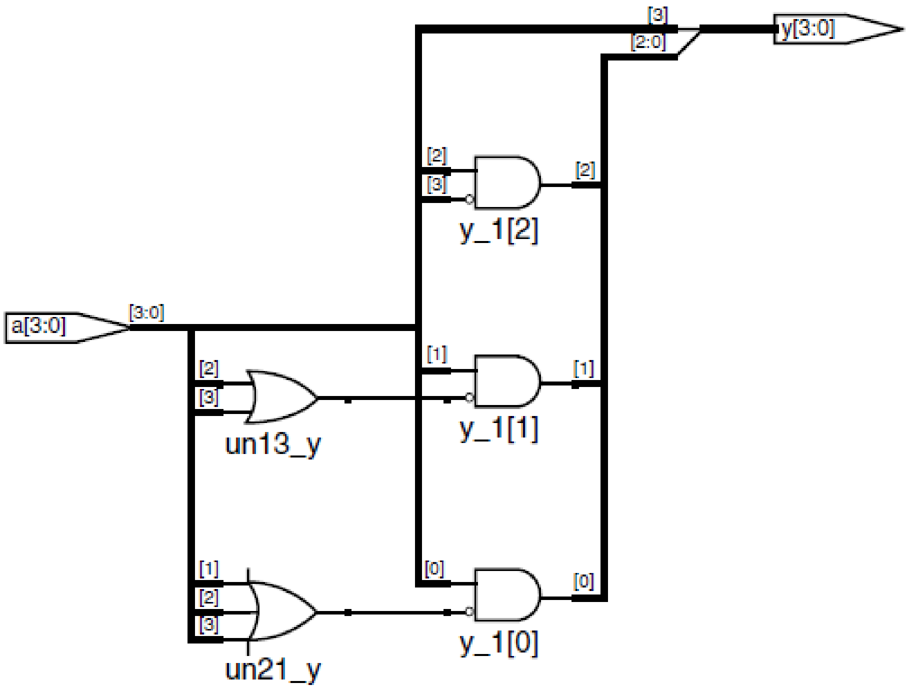
No `others` clause is needed because all cases are covered.

if statement (*priority* circuit)

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.



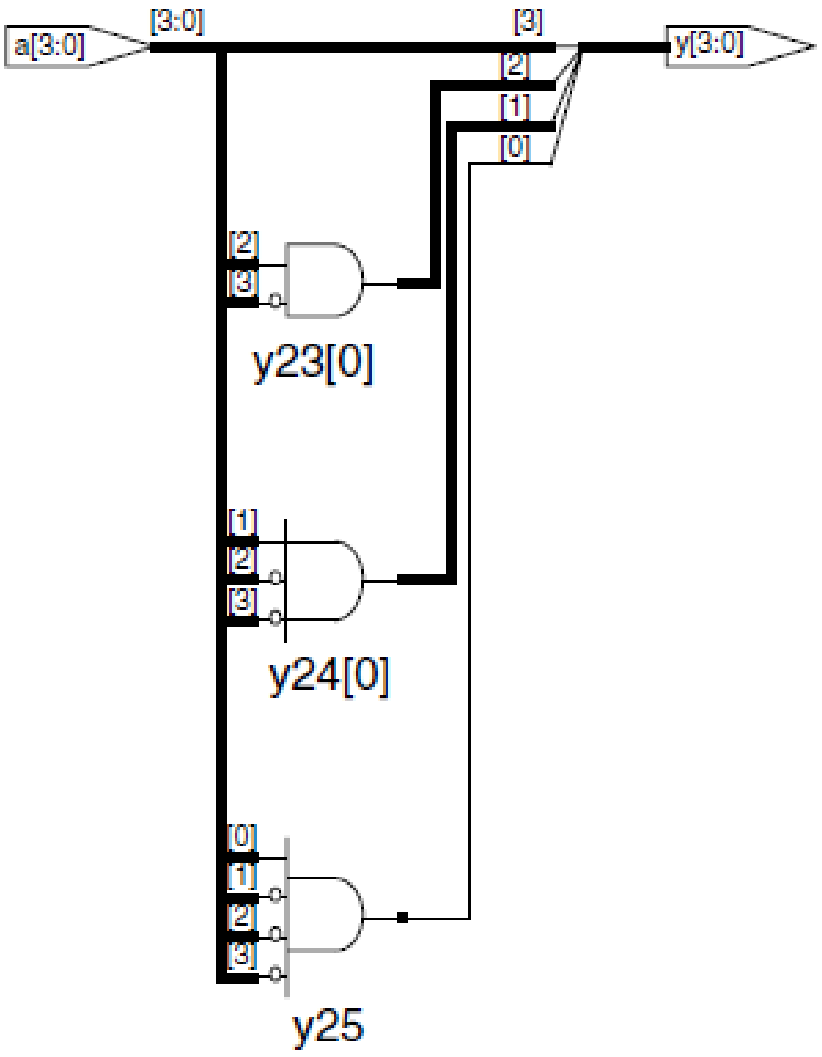
VHDL

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

entity priority is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of priority is
begin
  process(a) begin
    if    a(3) = '1' then y <= "1000";
    elsif a(2) = '1' then y <= "0100";
    elsif a(1) = '1' then y <= "0010";
    elsif a(0) = '1' then y <= "0001";
    else                  y <= "0000";
    end if;
  end process;
end;
```

Unlike Verilog, VHDL supports conditional signal assignment statements (see HDL Example 4.6), which are much like `if` statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic. (Figure follows on next page.)

priority circuit using **casez** in Verilog

```
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
```



FSM (divide-by-3)

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
```

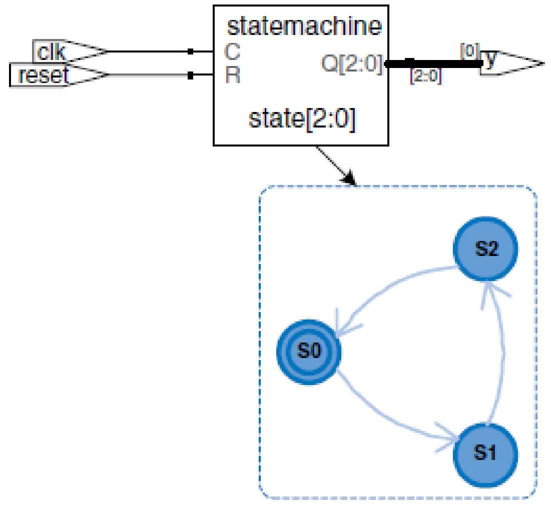
The `parameter` statement is used to define constants within a module. Naming the states with parameters is not required, but it makes changing state encodings much easier and makes the code more readable.

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`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity divideby3FSM is
  port(clk, reset: in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process (clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  nextstate <= S1 when state = S0 else
               S2 when state = S1 else
               S0;

  -- output logic
  y <= '1' when state = S0 else '0';
end;
```



This example defines a new *enumeration* data type, `statetype`, with three possibilities: `S0`, `S1`, and `S2`. `state` and `nextstate` are `statetype` signals. By using an enumeration instead of choosing the state encoding, VHDL frees the synthesizer to explore various state encodings to choose the best one.

The output, `y`, is 1 when the `state` is `S0`. The inequality-comparison uses `/=`. To produce an output of 1 when the state is anything but `S0`, change the comparison to `state /= S0`.

Verilog

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

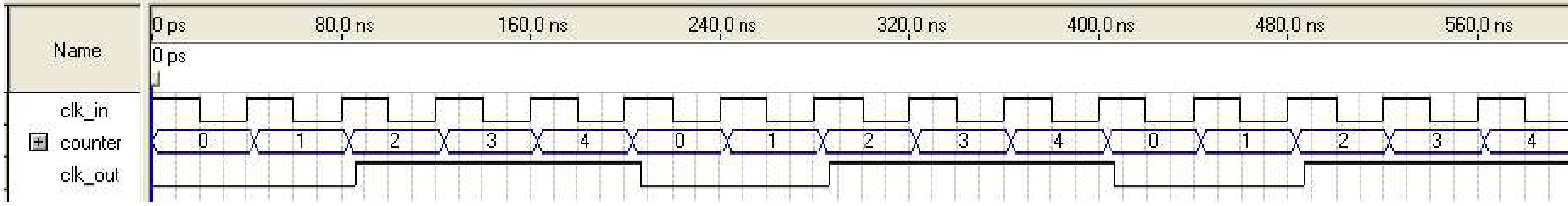
VHDL

```
-- output logic
y <= '1' when (state = S0 or state = S1) else '0';
```

frequency divider with VARIABLE

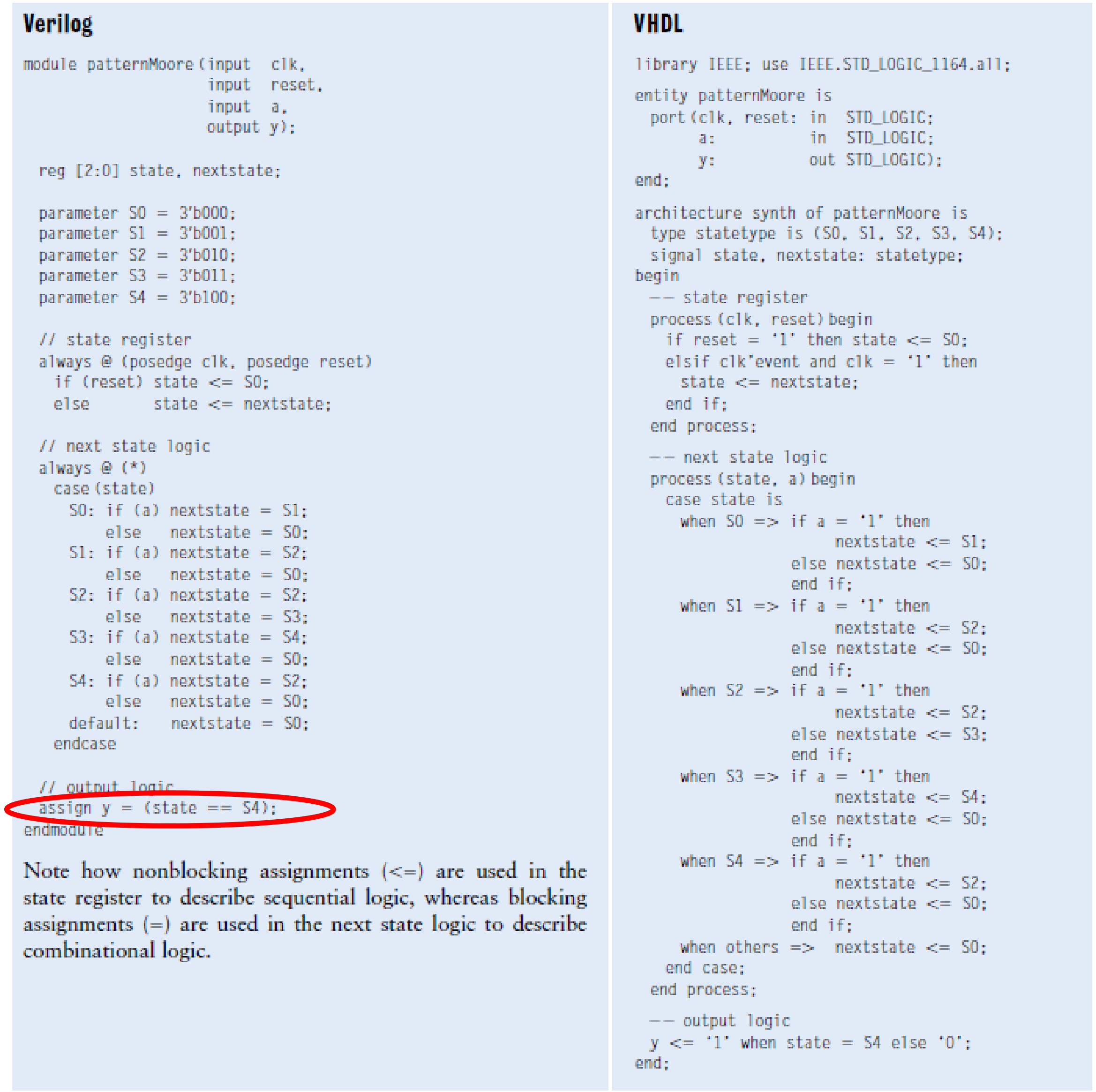
```
-------------------------------------------------
ENTITY clock_divider IS
   GENERIC (M: NATURAL := 5;
   PORT (clk_in: IN BIT;
         clk_out: OUT BIT);
END ENTITY;
-------------------------------------------------
ARCHITECTURE circuit OF clock_divider IS
BEGIN
   PROCESS (clk_in)
      VARIABLE counter: NATURAL RANGE 0 TO M;
   BEGIN
      IF clk_in'EVENT AND clk_in='1' THEN
         counter := counter + 1;
         IF counter=M/2 THEN
            clk_out <= '1';
         ELSIF counter=M THEN
            clk_out <= '0';
            counter := 0;
         END IF;
      END IF;
   END PROCESS;
END ARCHITECTURE;
```

Simulation results for *M*=5:



pattern recognizer (Moore FSM)

- searching for the pattern of “**1101**” (from left to right)
- if input sequence is “111011010” (from left to right), => output sequence is “000010010”
- In Moore FSM, output depends only on state
  - need 5 states in this example



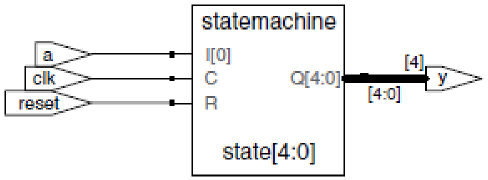
Verilog

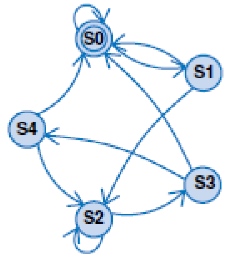
```
module patternMoore(input  clk,
                    input  reset,
                    input  a,
                    output y);

  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
```

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.

VHDL

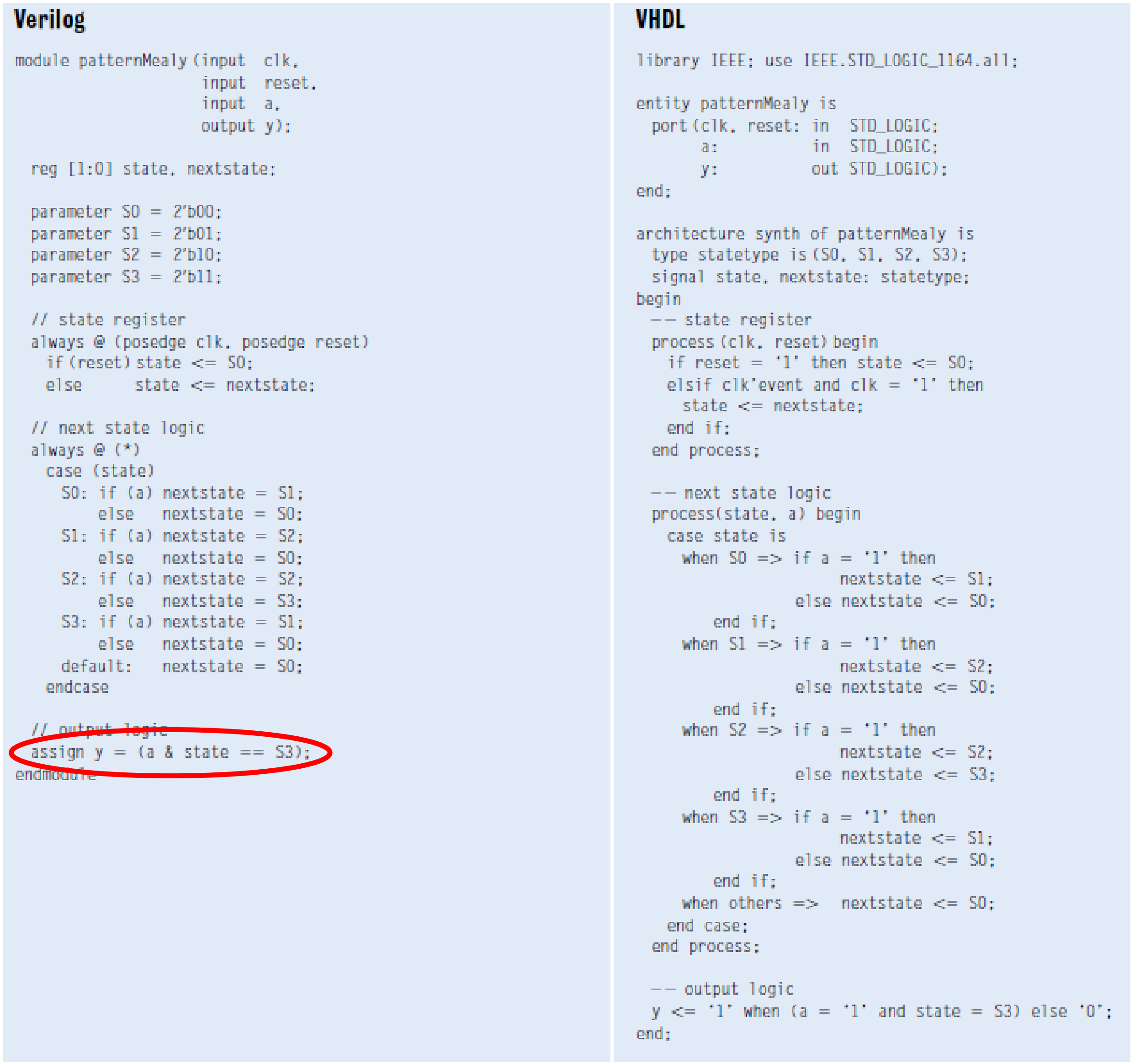
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMoore is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMoore is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(state, a) begin
    case state is
      when S0 => if a = '1' then
                    nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if a = '1' then
                    nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if a = '1' then
                    nextstate <= S2;
                 else nextstate <= S3;
                 end if;
      when S3 => if a = '1' then
                    nextstate <= S4;
                 else nextstate <= S0;
                 end if;
      when S4 => if a = '1' then
                    nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when others => nextstate <= S0;
    end case;
  end process;

  -- output logic
  y <= '1' when state = S4 else '0';
end;
```





pattern recognizer (Mealy FSM)

- In Mealy FSM, output depends on both input and states
  - need only 4 states in this example

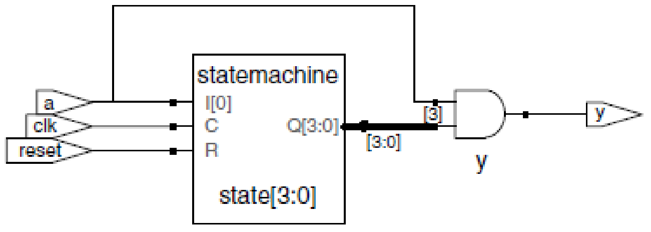


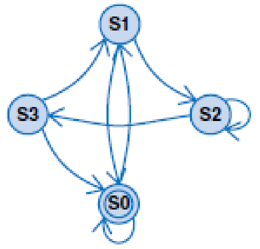
Verilog

```
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
```

VHDL

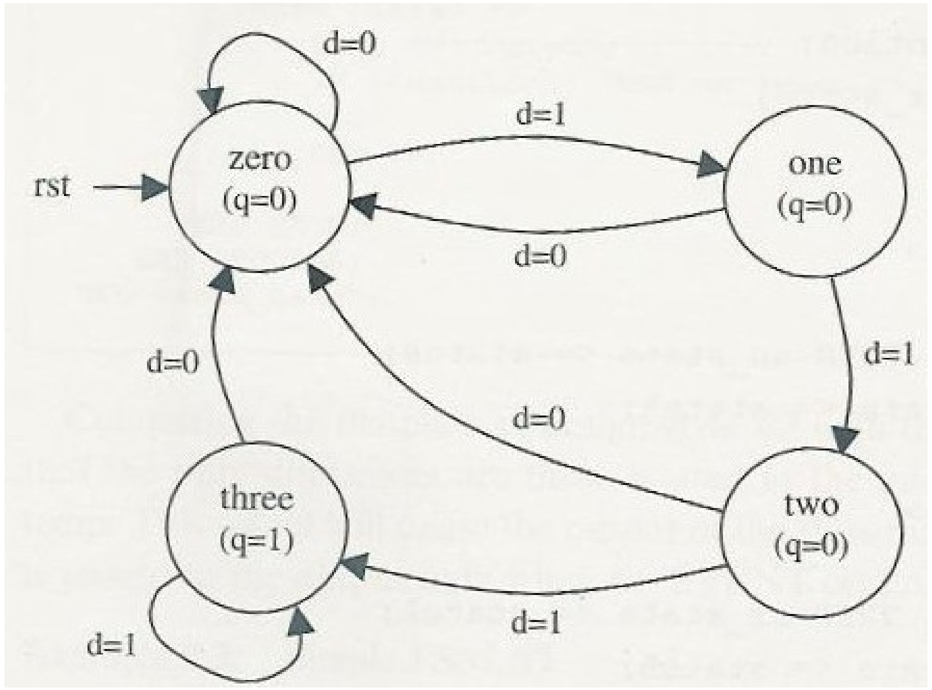
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMealy is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMealy is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(state, a) begin
    case state is
      when S0 => if a = '1' then
                      nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if a = '1' then
                      nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if a = '1' then
                      nextstate <= S2;
                 else nextstate <= S3;
                 end if;
      when S3 => if a = '1' then
                      nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when others =>  nextstate <= S0;
    end case;
  end process;

  -- output logic
  y <= '1' when (a = '1' and state = S3) else '0';
end;
```



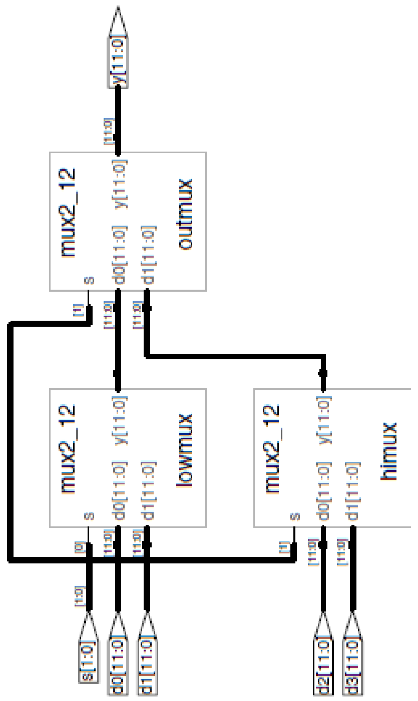


“111” string detector

```
-- detect a string sequence of “111”
ENTITY string_detector IS PORT (d, clk, rst: IN BIT; q: OUT BIT); END string_detector;
-------------------------------------------
ARCHITECTURE my_arch OF string_detector IS
TYPE state IS (zero, one, two, three);
SIGNAL pr_state, nx_state: state;
BEGIN
----- Lower section: --------------------
PROCESS (rst, clk) BEGIN
IF (rst='1') THEN
        pr_state <= zero;
ELSIF (clk'EVENT AND clk='1') THEN
        pr_state <= nx_state;
END IF;
END PROCESS;
---------- Upper section: ---------------
PROCESS (d, pr_state) BEGIN
CASE pr_state IS
WHEN zero => q <= ‘0’;IF (d=‘1’) THEN nx_state <= one; ELSE nx_state <= zero; END IF;
WHEN one => q <= ‘0’; IF (d=‘1’) THEN nx_state <= two; ELSE nx_state <= zero; END IF;
WHEN two => q <= ‘0’; IF (d=‘1’) THEN nx_state <= three; ELSE nx_state <= zero; END IF;
WHEN three => q <= ‘1’; IF (d=’0’) THEN nx_state <= zero; ELSE nx_state <= three; END IF;
END CASE;
END PROCESS;
END my_arch;
```



parameterized modules



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.

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.

VHDL

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

entity mux2 is
  generic(width: integer := 8);
  port(d0,
       d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:  in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d0 when s = '0' else d1;
end;
```

The `generic` statement includes a default value (8) of `width`. The value is an integer.

```
entity mux4_8 is
  port(d0, d1, d2,
       d3: in  STD_LOGIC_VECTOR(7 downto 0);
       s:  in  STD_LOGIC_VECTOR(1 downto 0);
       y:  out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux4_8 is
  component mux2
    generic(width: integer);
  port(d0,
       d1: in  STD_LOGIC_VECTOR(width-1 downto 0) ;
       s:  in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal low, hi: STD_LOGIC_VECTOR(7 downto 0);
begin
  lowmux: mux2 port map(d0, d1, s(0), low);
  himux:  mux2 port map(d2, d3, s(0), hi);
  outmux: mux2 port map(low, hi, s(1), y);
end;
```

The 8-bit 4:1 multiplexer, `mux4_8`, instantiates three 2:1 multiplexers using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `generic map`, as shown below.

```
lowmux: mux2 generic map(12)
             port map(d0, d1, s(0), low);
himux:  mux2 generic map(12)
             port map(d2, d3, s(0), hi);
outmux: mux2 generic map(12)
             port map(low, hi, s(1), y);
```

parameterized decoder

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.

VHDL

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

entity decoder is
  generic (N: integer := 3);
  port (a: in  STD_LOGIC_VECTOR (N-1 downto 0);
        y: out STD_LOGIC_VECTOR (2**N-1 downto 0));
end;

architecture synth of decoder is
begin
  process (a)
    variable tmp: STD_LOGIC_VECTOR (2**N-1 downto 0);
  begin
    tmp := CONV_STD_LOGIC_VECTOR(0, 2**N);
    tmp (CONV_INTEGER(a)) := '1';
    y <= tmp;
  end process;
end;
```

`2**N` indicates 2^N.

`CONV_STD_LOGIC_VECTOR(0, 2**N)` produces a `STD_LOGIC_VECTOR` of length 2^N containing all 0's. It requires the `STD_LOGIC_ARITH` library. The function is useful in other parameterized functions, such as resettable flip-flops that need to be able to produce constants with a parameterized number of bits. The bit index in VHDL must be an integer, so the `CONV_INTEGER` function is used to convert a from a `STD_LOGIC_VECTOR` to an integer.

N-bit AND using generate loop

Verilog

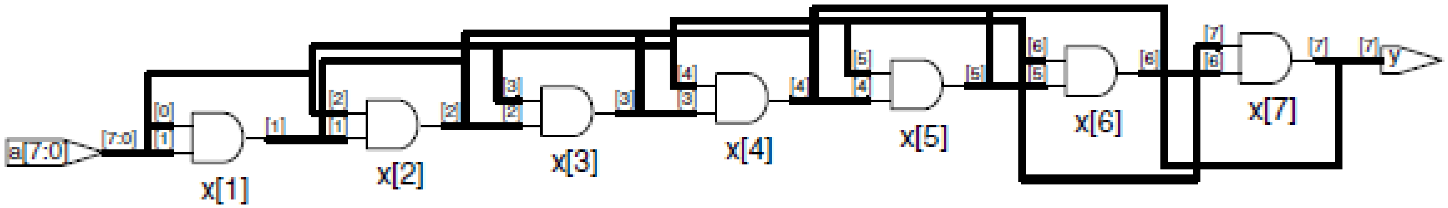
```
module andN
  #(parameter width = 8)
   (input [width-1:0] a,
    output             y);

  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!

VHDL

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

entity andN is
  generic (width: integer := 8);
  port (a: in  STD_LOGIC_VECTOR (width-1 downto 0);
        y: out STD_LOGIC);
end;

architecture synth of andN is
  signal i: integer;
  signal x: STD_LOGIC_VECTOR (width-1 downto 1);
begin
  AllBits: for i in 1 to width-1 generate
    LowBit: if i = 1 generate
      A1: x(1) <= a(0) and a(1);
    end generate;
    OtherBits: if i /= 1 generate
      Ai: x(i) <= a(i) and x(i-1);
    end generate;
  end generate;
  y <= x(width-1);
end;
```



testbench

Verilog

```
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.

VHDL

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

entity testbench1 is -- no inputs or outputs
end;

architecture sim of testbench1 is
  component sillyfunction
    port (a, b, c: in  STD_LOGIC;
          y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map (a, b, c, y);

  -- apply inputs one at a time
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    wait; -- wait forever
  end process;
end;
```

The process statement first applies the input pattern 000 and waits for 10 ns. It then applies 001 and waits 10 more ns, and so forth until all eight possible inputs have been applied.

At the end, the process waits indefinitely; otherwise, the process would begin again, repeatedly applying the pattern of test vectors.

self-checking testbench

Verilog

```
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.

VHDL

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

entity testbench2 is -- no inputs or outputs
end;

architecture sim of testbench2 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  -- checking results
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
     assert y = '1' report "000 failed.";
    c <= '1';                     wait for 10 ns;
     assert y = '0' report "001 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
     assert y = '0' report "010 failed.";
    c <= '1';                     wait for 10 ns;
     assert y = '0' report "011 failed.";
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
     assert y = '1' report "100 failed.";
    c <= '1';                     wait for 10 ns;
     assert y = '1' report "101 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
     assert y = '0' report "110 failed.";
    c <= '1';                     wait for 10 ns;
     assert y = '0' report "111 failed.";
    wait; -- wait forever
  end process;
end;
```

The `assert` statement checks a condition and prints the message given in the `report` clause if the condition is not satisfied. `assert` is meaningful only in simulation, not in synthesis.

testbench with test vector file (1/2)

Verilog

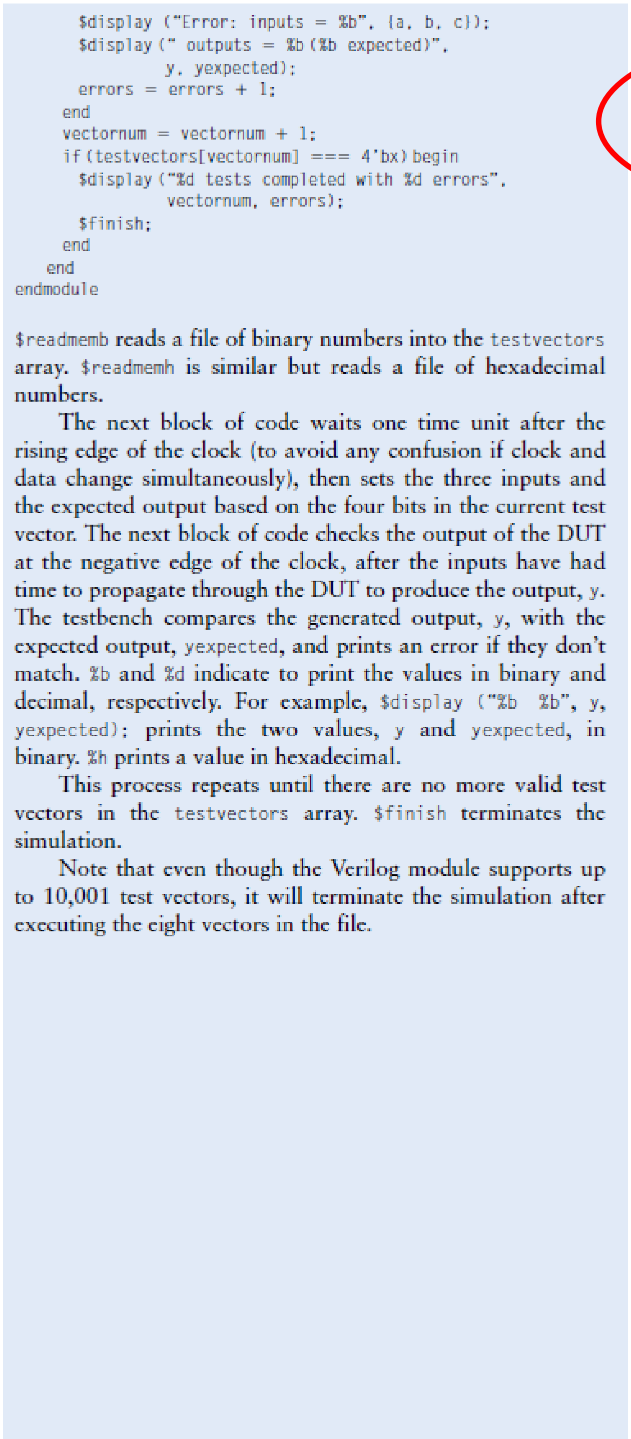
```
module testbench3();
  reg         clk, reset;
  reg         a, b, c, yexpected;
  wire        y;
  reg [31:0] vectornum, errors;
  reg [3:0]  testvectors[10000:0];

  // 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
```

VHDL

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

entity testbench3 is -- no inputs or outputs
end;

architecture sim of testbench3 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
            y:    out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
  signal clk, reset: STD_LOGIC;
  signal yexpected:  STD_LOGIC;
  constant MEMSIZE: integer := 10000;
  type tvarray is array(MEMSIZE downto 0) of
    STD_LOGIC_VECTOR(3 downto 0);
  signal testvectors: tvarray;
  shared variable vectornum, errors: integer;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- at start of test, load vectors
  -- and pulse reset
  process is
    file tv: TEXT;
    variable i, j: integer;
    variable L: line;
    variable ch: character;
```

testbench with test vector file (2/2)

- refer to user guides for more system tasks on reading data from files



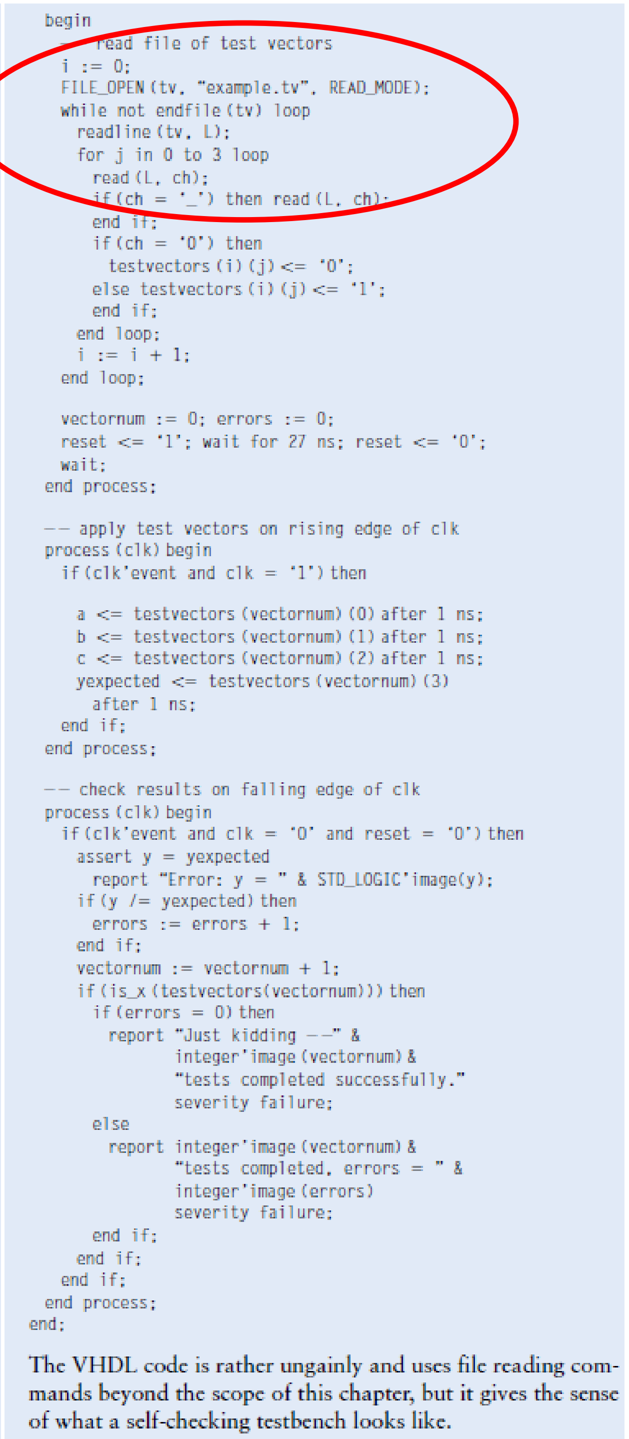
```
        $display ("Error: inputs = %b", {a, b, c});
        $display (" outputs = %b (%b expected)",
                  y, yexpected);
        errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
        $display ("%d tests completed with %d errors",
                  vectornum, errors);
        $finish;
      end
    end
endmodule
```

$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.



```
  begin
    -- read file of test vectors
    i := 0;
    FILE_OPEN (tv, "example.tv", READ_MODE);
    while not endfile (tv) loop
      readline (tv, L);
      for j in 0 to 3 loop
        read (L, ch);
        if (ch = '_') then read (L, ch);
        end if;
        if (ch = '0') then
          testvectors (i) (j) <= '0';
        else testvectors (i) (j) <= '1';
        end if;
      end loop;
      i := i + 1;
    end loop;

    vectornum := 0; errors := 0;
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;

  -- apply test vectors on rising edge of clk
  process (clk) begin
    if (clk'event and clk = '1') then

      a <= testvectors (vectornum) (0) after 1 ns;
      b <= testvectors (vectornum) (1) after 1 ns;
      c <= testvectors (vectornum) (2) after 1 ns;
      yexpected <= testvectors (vectornum) (3)
        after 1 ns;
    end if;
  end process;

  -- check results on falling edge of clk
  process (clk) begin
    if (clk'event and clk = '0' and reset = '0') then
      assert y = yexpected
        report "Error: y = " & STD_LOGIC'image(y);
      if (y /= yexpected) then
        errors := errors + 1;
      end if;
      vectornum := vectornum + 1;
      if (is_x (testvectors(vectornum))) then
        if (errors = 0) then
          report "Just kidding --" &
                 integer'image (vectornum) &
                 "tests completed successfully."
                 severity failure;
        else
          report integer'image (vectornum) &
                 "tests completed, errors = " &
                 integer'image (errors)
                 severity failure;
        end if;
      end if;
    end if;
  end process;
end;
```

The VHDL code is rather ungainly and uses file reading commands beyond the scope of this chapter, but it gives the sense of what a self-checking testbench looks like.