

Chapter 4

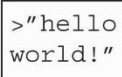


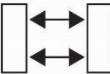
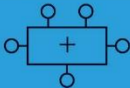
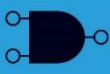
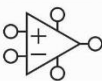


Digital Design and Computer Architecture: ARM[®] Edition

Sarah L. Harris and David Money Harris



Chapter 4 :: Topics

- Introduction
- Combinational Logic
- Structural Modeling
- Sequential Logic
- More Combinational Logic
- Finite State Machines
- Parameterized Modules
- Testbenches

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	



Introduction

- Hardware description language (HDL):
 - specifies logic function only
 - Computer-aided design (CAD) tool produces or *synthesizes* the optimized gates
- Most commercial designs built using HDLs
- Two leading HDLs:
 - **SystemVerilog**
 - developed in 1984 by Gateway Design Automation
 - IEEE standard (1364) in 1995
 - Extended in 2005 (IEEE STD 1800-2009)
 - **VHDL 2008**
 - Developed in 1981 by the Department of Defense
 - IEEE standard (1076) in 1987
 - Updated in 2008 (IEEE STD 1076-2008)



SystemVerilog

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 standard¹ in 1995. The language was extended in 2005 to streamline idiosyncrasies and to better support modeling and verification of systems. These extensions have been merged into a single language standard, which is now called SystemVerilog (IEEE STD 1800-2009). SystemVerilog file names normally end in `.sv`.

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 language was first envisioned for documentation but was quickly adopted for simulation and synthesis. The IEEE standardized it in 1987 and has updated the standard several times since. This chapter is based on the 2008 revision of the VHDL standard (IEEE STD 1076-2008), which streamlines the language in a variety of ways. At the time of this writing, not all of the VHDL 2008 features are supported by CAD tools; this chapter only uses those understood by Synplicity, Altera Quartus, and ModelSim. VHDL file names normally end in `.vhd`.

To use VHDL 2008 in ModelSim, you may need to set `VHDL93 = 2008` in the `modelsim.ini` configuration file.



HDL to Gates

• Simulation

- Inputs applied to circuit
- Outputs checked for correctness
- Millions of dollars saved by debugging in simulation instead of hardware
- Example: correcting a mistake in a cutting-edge integrated circuit costs more than a million dollars and takes several months. Intel's infamous FDIV (floating point division) bug in the Pentium processor forced the company to recall chips after they had shipped, at a total cost of \$475 million.
- Logic simulation is essential to test a system before it is built.

• Synthesis

- Transforms HDL code into a *netlist* describing the hardware (i.e., a list of gates and the wires connecting them)



HDL to Gates

- **Simulation**

- Inputs applied to circuit
- Outputs checked for correctness
- Millions of dollars saved by debugging in simulation instead of hardware

- **Synthesis**

- Transforms HDL code into a *netlist* describing the hardware (i.e., a list of gates and the wires connecting them)

IMPORTANT: When using an HDL, think of the **hardware** the HDL should produce

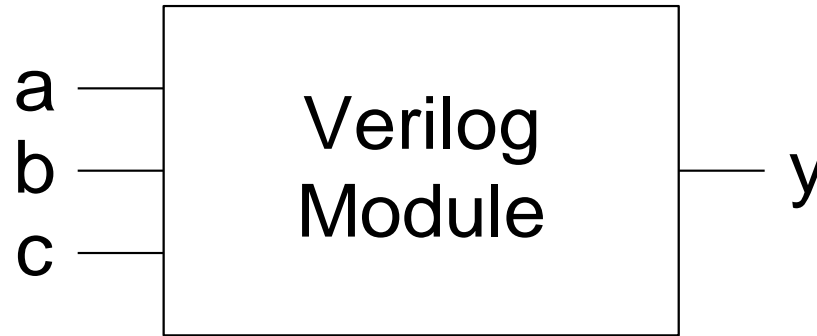


HDL

- In our experience, the best way to **learn an HDL is by example**. HDLs have specific ways of describing various classes of logic; these ways are called **idioms**.
- This chapter will teach you how to write the proper **HDL idioms** for each type of block and then how to put the blocks together to produce a working system.
- When you need to describe a particular kind of hardware, **look for a similar example** and adapt it to your purpose.
- We do not attempt to rigorously define all the syntax of the HDLs, because that is deathly boring and because it tends to encourage thinking of HDLs as programming languages, not **shorthand for hardware**.



SystemVerilog Modules



Two types of Modules:

- **Behavioral:** describe what a module does
- **Structural:** describe how it is built from simpler modules



Behavioral SystemVerilog

SystemVerilog:

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



Behavioral SystemVerilog

SystemVerilog:

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

- module/endmodule: required to begin/end module
- example: name of the module
- Operators:
 - ~: NOT
 - &: AND
 - |: OR



SystemVerilog

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

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

endmodule
```

A SystemVerilog module begins with the module name and a listing of the inputs and outputs. The `assign` statement describes combinational logic. `~` indicates NOT, `&` indicates AND, and `|` indicates OR.

`logic` signals such as the inputs and outputs are Boolean variables (0 or 1). They may also have floating and undefined values, as discussed in [Section 4.2.8](#).

The `logic` type was introduced in SystemVerilog. It supersedes the `reg` type, which was a perennial source of confusion in Verilog. `logic` should be used everywhere except on signals with multiple drivers. Signals with multiple drivers are called *nets* and will be explained in [Section 4.7](#).

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;
```

VHDL code has three parts: the library use clause, the entity declaration, and the architecture body. The library use clause will be discussed in [Section 4.7.2](#). The entity declaration lists the module name and its inputs and outputs. The architecture body defines what the module does.

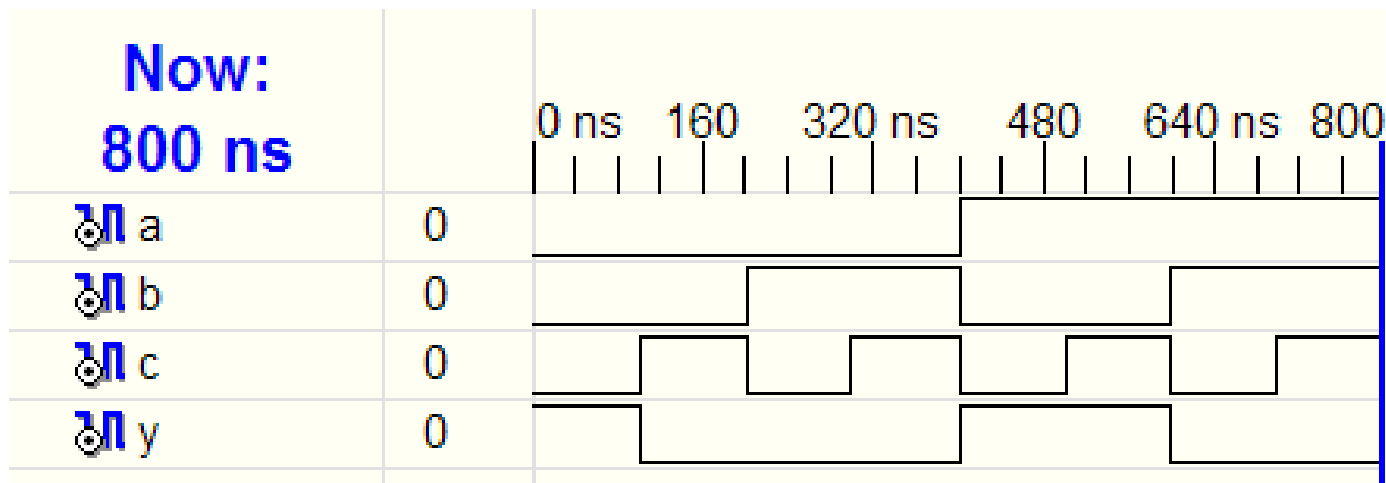
VHDL signals, such as inputs and outputs, must have a *type declaration*. Digital signals should be declared to be `STD_LOGIC` type. `STD_LOGIC` signals can have a value of '0' or '1', as well as floating and undefined values that will be described in [Section 4.2.8](#). The `STD_LOGIC` type is defined in the `IEEE.STD_LOGIC_1164` library, which is why the library must be used.

VHDL lacks a good default order of operations between AND and OR, so Boolean equations should be parenthesized.

HDL Simulation

SystemVerilog:

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

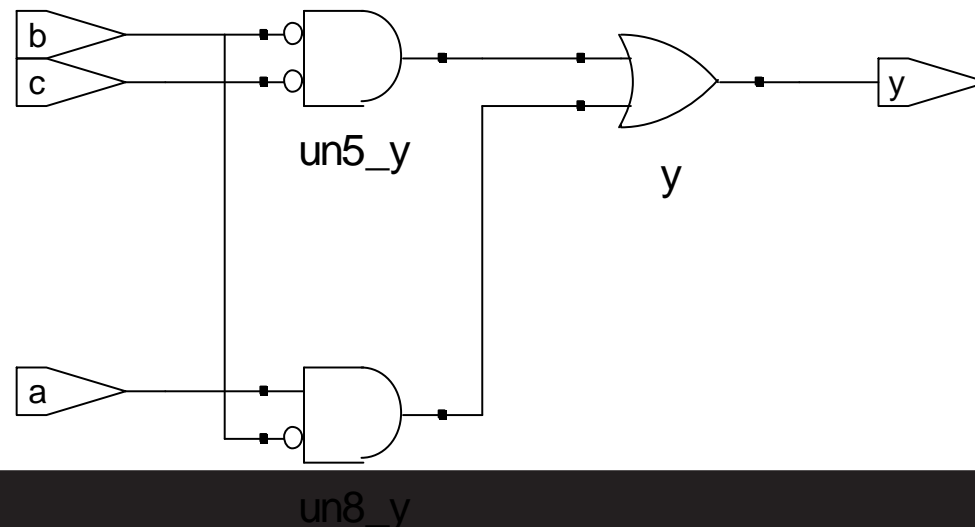


HDL Synthesis

SystemVerilog:

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

Synthesis:



SystemVerilog Syntax

- Case sensitive
 - **Example:** `reset` and `Reset` are not the same signal.
- No names that start with numbers
 - **Example:** `2mux` is an invalid name
- Whitespace ignored
- Comments:
 - `//` single line comment
 - `/*` multiline
comment `*/`



Structural Modeling - Hierarchy

```
module and3(input  logic a, b, c,  
            output logic y);  
    assign y = a & b & c;  
endmodule
```

```
module inv(input  logic a,  
           output logic y);  
    assign y = ~a;  
endmodule
```

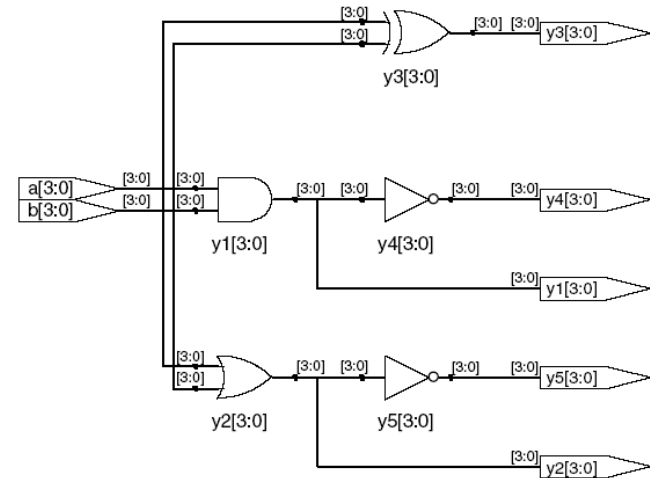
```
module nand3(input  logic a, b, c  
            output logic y);  
    logic n1;                // internal signal  
  
    and3 andgate(a, b, c, n1); // instance of and3  
    inv  inverter(n1, y);      // instance of inv  
endmodule
```



Bitwise Operators

Bitwise operators act on single-bit signals or on multi-bit busses.

```
module gates(input logic [3:0] a, b,  
            output logic [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
```



// single line comment

/*...*/ multiline comment



SystemVerilog

SystemVerilog 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.

SystemVerilog is case-sensitive. `y1` and `Y1` are different signals in SystemVerilog. However, it is confusing to use multiple signals that differ only in case.

VHDL

Comments beginning with `/*` continue, possibly across multiple lines, to the next `*/`. Comments beginning with `--` continue to the end of the 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.



SystemVerilog

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

/* five different two-input logic
   gates acting on 4-bit buses */
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 SystemVerilog *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 buses
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.

SystemVerilog

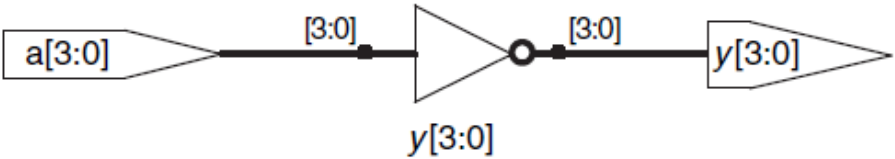
```
module inv(input logic [3:0] a,  
          output logic [3:0] y);  
  
    assign y=~a;  
endmodule
```

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;
```

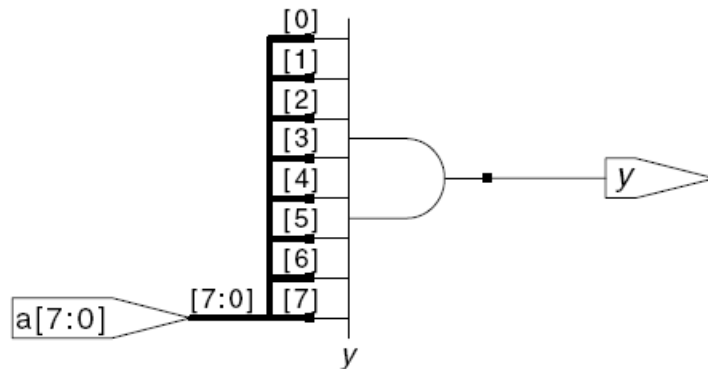
VHDL uses STD_LOGIC_VECTOR to indicate busses of STD_LOGIC. STD_LOGIC_VECTOR(3 downto 0) represents a 4-bit bus. The bits, from most significant to least significant, are a(3), a(2), a(1), and a(0). This is called *little-endian* order, because the least significant bit has the smallest bit number. We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



Reduction Operators

Reduction operators imply a multiple-input gate acting on a single bus.

```
module and8(input  logic [7:0] a,  
            output logic      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
```



HDL Example 4.4 EIGHT-INPUT AND

SystemVerilog

```
module and8(input  logic [7:0] a,
            output logic      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
```

VHDL

```
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;

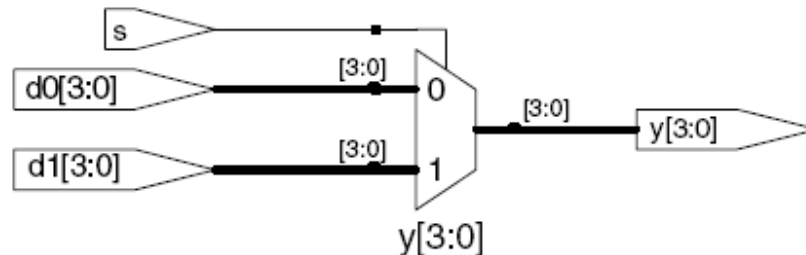
architecture synth of and8 is
begin
    y <= and a;
    -- and a is much easier to write than
    -- y <= a(7) and a(6) and a(5) and a(4) and
    --       a(3) and a(2) and a(1) and a(0);
end;
```



Conditional Assignment

Conditional assignments select the output from among alternatives based on an input called the condition.

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



? : is also called a *ternary operator* because it operates on 3 inputs: s, d1, and d0.



SystemVerilog

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 logic [3:0] d0, d1,
            input logic s,
            output logic [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

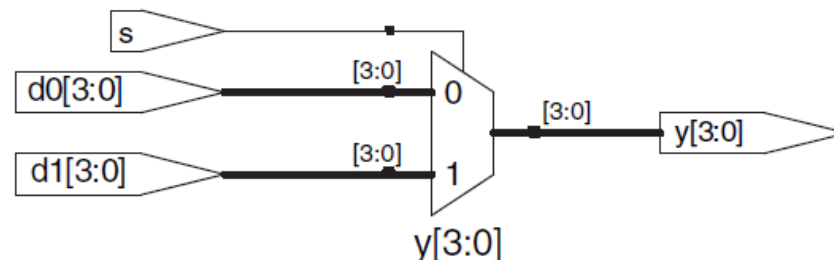
Conditional signal assignments perform different operations depending on some condition. They are especially useful for describing a multiplexer. For example, a 2:1 multiplexer can use conditional signal assignment to select one of two 4-bit inputs.

```
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 <= d1 when s else d0;
end;
```

The conditional signal assignment sets y to $d1$ if s is 1. Otherwise it sets y to $d0$. Note that prior to the 2008 revision of VHDL, one had to write `when s = '1'` rather than `when s`.



HDL Example 4.6 4:1 MULTIPLEXER

SystemVerilog

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

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

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

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 `switch/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 s select y <=
        d0 when "00",
        d1 when "01",
        d2 when "10",
        d3 when others;
end;
```

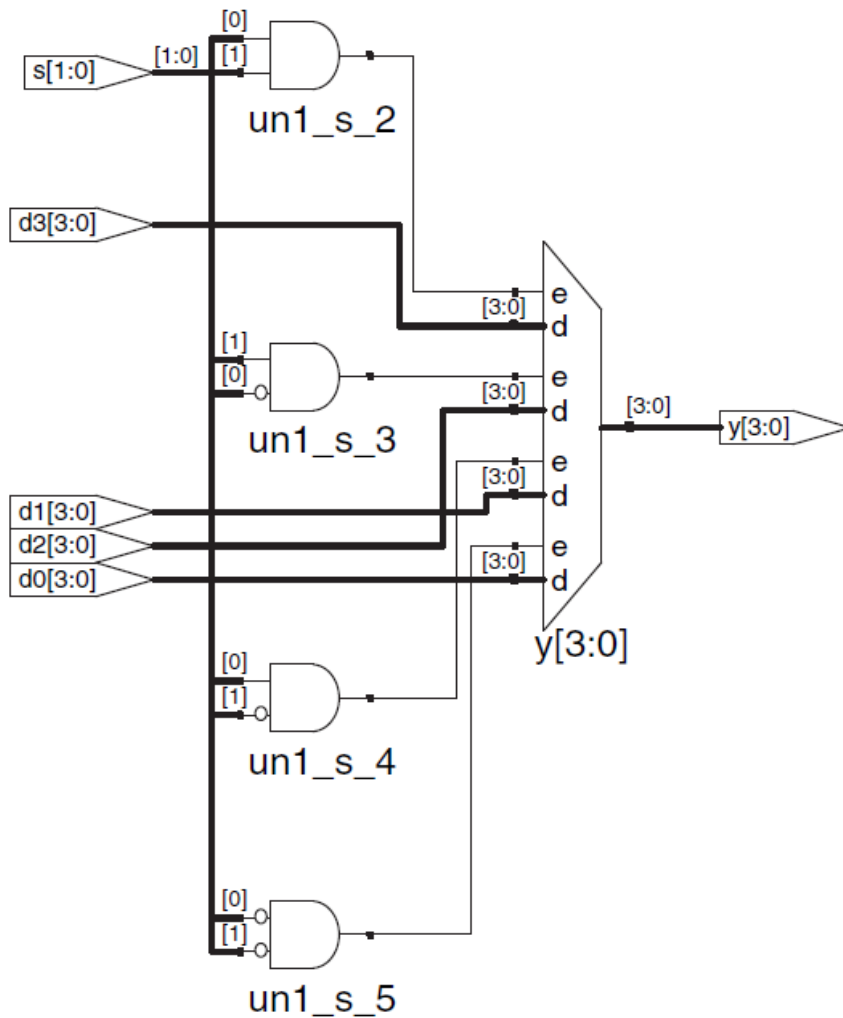



Figure 4.7 mux4 synthesized circuit



Internal Variables

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

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

intermediate signals, P and G ,

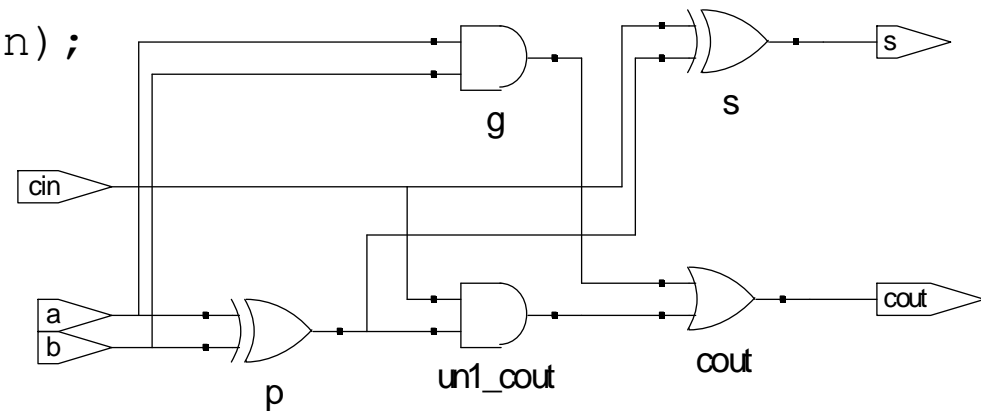
$$P = A \oplus B$$

$$G = AB$$

ite the full adder as follows:

$$S = P \oplus C_{in}$$

$$C_{out} = G + PC_{in}$$



HDL Example 4.7 FULL ADDER

SystemVerilog

In SystemVerilog, internal signals are usually declared as logic.

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

    logic 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 $p \leq a \text{ xor } b$;

```
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;
```



Precedence

Highest

\sim	NOT
$*$, $/$, $\%$	mult, div, mod
$+$, $-$	add, sub
\ll , \gg	shift
\lll , \ggg	arithmetic shift
$<$, \leq , $>$, \geq	comparison
$==$, $!=$	equal, not equal
$\&$, $\sim\&$	AND, NAND
\wedge , $\sim\wedge$	XOR, XNOR
$ $, $\sim $	OR, NOR
$?:$	ternary operator

Lowest



Table 4.1 SystemVerilog operator precedence

	Op	Meaning
H i g h e s t	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left/Right Shift
	<<<, >>>	Arithmetic Left/Right Shift
	<, <=, >, >=	Relative Comparison
L o w e s t	==, !=	Equality Comparison
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	, ~	OR, NOR
	?:	Conditional

The operator precedence for SystemVerilog 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;
```

Table 4.2 VHDL operator precedence

	Op	Meaning
H i g h e s t	not	NOT
	*, /, mod, rem	MUL, DIV, MOD, REM
	+, -	PLUS, MINUS
	rol, ror, srl, sll	Rotate, Shift logical
	<, <=, >, >=	Relative Comparison
	L o w e s t	=, /=
and, or, nand, nor, xor, xnor		Logical Operations

Multiplication has precedence over addition in VHDL, as you would expect. However, unlike SystemVerilog, 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

Format: N'Bvalue

N = number of bits, **B** = base

N'B is optional but recommended (default is decimal)

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00...0011
8'b11	8	binary	3	00000011
8'b1010_1011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011
42	Unsized	decimal	42	00...0101010



SystemVerilog

The format for declaring constants is $N'Bvalue$, where N is the size in bits, B is a letter indicating the base, and $value$ gives the value. For example, $9'h25$ indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. SystemVerilog supports 'b for binary, 'o for octal, 'd for decimal, and 'h for hexadecimal. If the base is omitted, it 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. An exception is that '0 and '1 are SystemVerilog idioms for filling a bus with all 0s and all 1s, respectively.

Table 4.3 SystemVerilog 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. The format for declaring `STD_LOGIC_VECTOR` constants is $NB"value$, where N is the size in bits, B is a letter indicating the base, and $value$ gives the value. For example, $9X"25"$ indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. VHDL 2008 supports B for binary, O for octal, D for decimal, and X for hexadecimal.

If the base is omitted, it defaults to binary. If the size is not given, the number is assumed to have a size matching the number of bits specified in the value. As of October 2011, Synplify Premier from Synopsys does not yet support specifying the size.

`others => '0'` and `others => '1'` are VHDL idioms to fill all of the bits with 0 and 1, respectively.

Table 4.4 VHDL numbers

Numbers	Bits	Base	Val	Stored
3B"101"	3	2	5	101
B"11"	2	2	3	11
8B"11"	8	2	3	00000011
8B"1010_1011"	8	2	171	10101011
3D"6"	3	10	6	110
6O"42"	6	8	34	100010
8X"AB"	8	16	171	10101011
"101"	3	2	5	101
B"101"	3	2	5	101
X"AB"	8	16	171	10101011

4.2.8 Z's and X's

HDLs use z to indicate a floating value, z is particularly useful for describing a tristate buffer, whose output floats when the enable is 0. Recall from Section 2.6.2 that a bus can be driven by several tristate buffers, exactly one of which should be enabled. [HDL Example 4.10](#) shows the idiom for a tristate buffer. If the buffer is enabled, the output is the same as the input. If the buffer is disabled, the output is assigned a floating value (z).

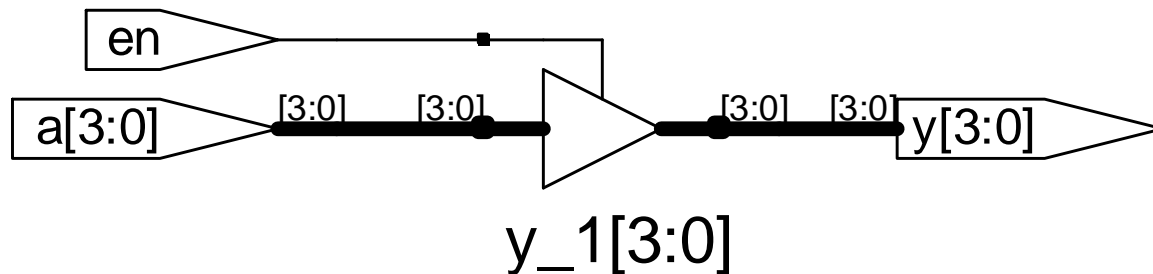
Similarly, HDLs use x to indicate an invalid logic level. If a bus is simultaneously driven to 0 and 1 by two enabled tristate buffers (or other gates), the result is x , indicating contention. If all the tristate buffers driving a bus are simultaneously OFF, the bus will float, indicated by z .



Z: Floating Output

SystemVerilog:

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



HDL Example 4.10 TRISTATE BUFFER

SystemVerilog

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

Notice that `y` is declared as `tri` rather than `logic`. `logic` signals can only have a single driver. Tristate busses can have multiple drivers, so they should be declared as a *net*. Two types of nets in SystemVerilog are called `tri` and `triereg`. Typically, exactly one driver on a net is active at a time, and the net takes on that value. If no driver is active, a `tri` floats (`z`), while a `triereg` retains the previous value. If no type is specified for an input or output, `tri` is assumed. Also note that a `tri` output from a module can be used as a `logic` input to another module. [Section 4.7](#) further discusses nets with multiple drivers.

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 <= a when en else "ZZZZ";  
end;
```

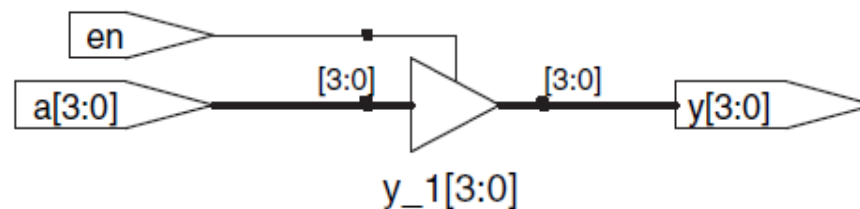


Figure 4.9 tristate synthesized circuit

HDL Example 4.11 TRUTH TABLES WITH UNDEFINED AND FLOATING INPUTS

SystemVerilog

SystemVerilog signal values are 0, 1, z, and x. SystemVerilog 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 SystemVerilog.

Table 4.5 SystemVerilog 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

Bit Manipulations: Example 1

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};
```

```
// if y is a 12-bit signal, the above statement produces:
```

```
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0
```

```
// underscores (_) are used for formatting only to make
```

```
// it easier to read. SystemVerilog ignores them.
```

Often it is necessary to operate on a subset of a bus or to concatenate (join together) signals to form busses. These operations are collectively known as bit swizzling. In [HDL Example 4.12](#), y is given the 9-bit value c2c1d0d0d0c0101 using **bit swizzling** operations

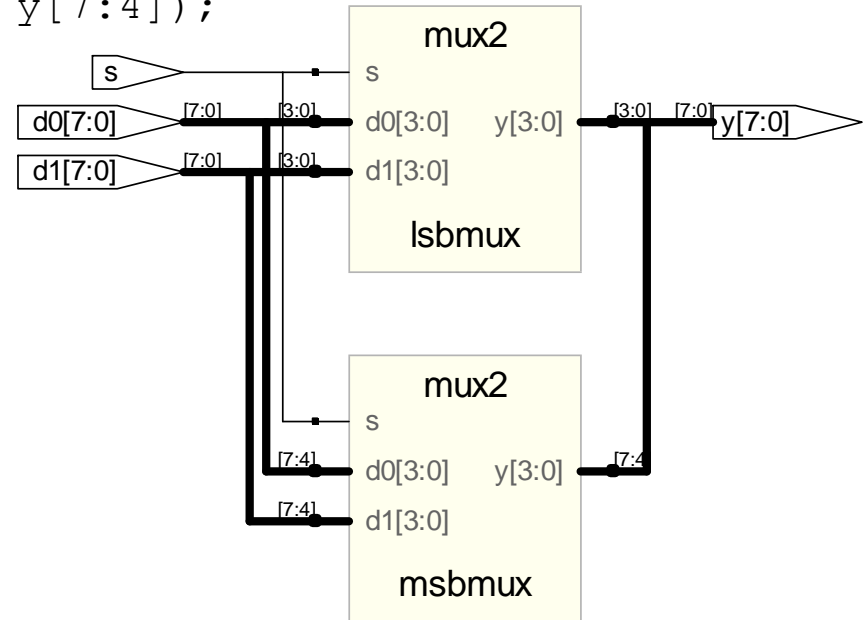


Bit Manipulations: Example 2

SystemVerilog:

```
module mux2_8(input  logic [7:0] d0, d1,
             input  logic      s,
             output logic [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
```



4.2.10 Delays

HDL statements may be associated with delays specified in arbitrary units. They are helpful during simulation to predict how fast a circuit will work (if you specify meaningful delays) and also for debugging purposes to understand cause and effect (deducing the source of a bad output is tricky if all signals change simultaneously in the simulation results). These delays are ignored during synthesis; the delay of a gate produced by the synthesizer depends on its t_{pd} and t_{cd} specifications, not on numbers in HDL code.

HDL Example 4.13 adds delays to the original function from HDL Example 4.1, $y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$. It assumes that inverters have a delay of 1 ns, three-input AND gates have a delay of 2 ns, and three-input OR gates have a delay of 4 ns. Figure 4.10 shows the simulation waveforms, with y lagging 7 ns after the inputs. Note that y is initially unknown at the beginning of the simulation.



HDL Example 4.13 LOGIC GATES WITH DELAYS

SystemVerilog

```
'timescale 1ns/1ps

module example(input  logic a, b, c,
              output logic y);

    logic 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
```

SystemVerilog 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) are used. In SystemVerilog, 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;

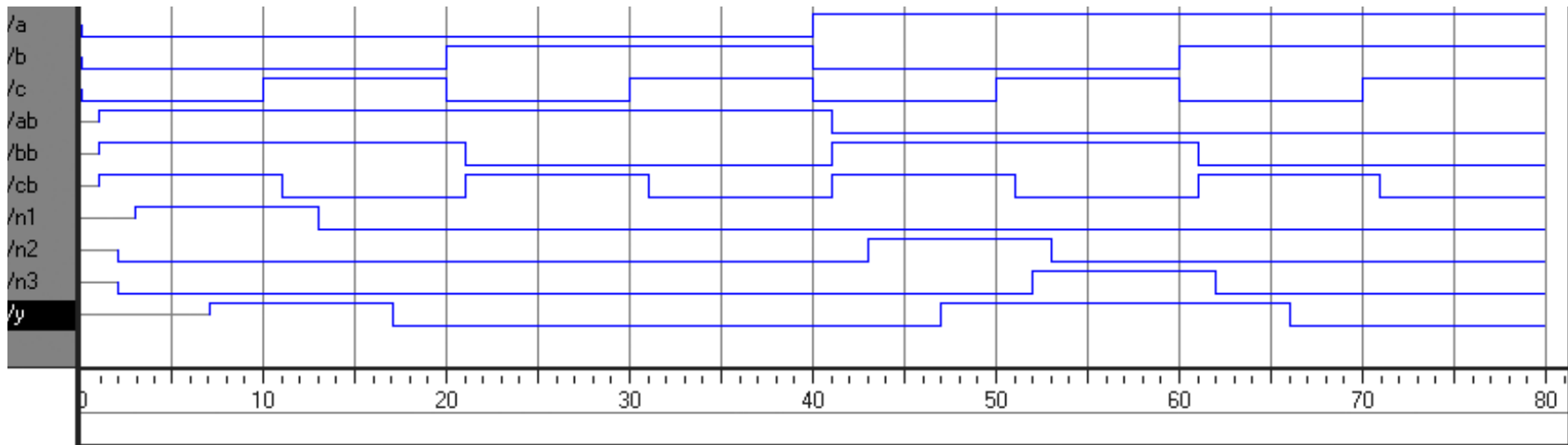
entity example is
    port(a, b, c: in  STD_LOGIC;
         y:      out STD_LOGIC);
end;

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.

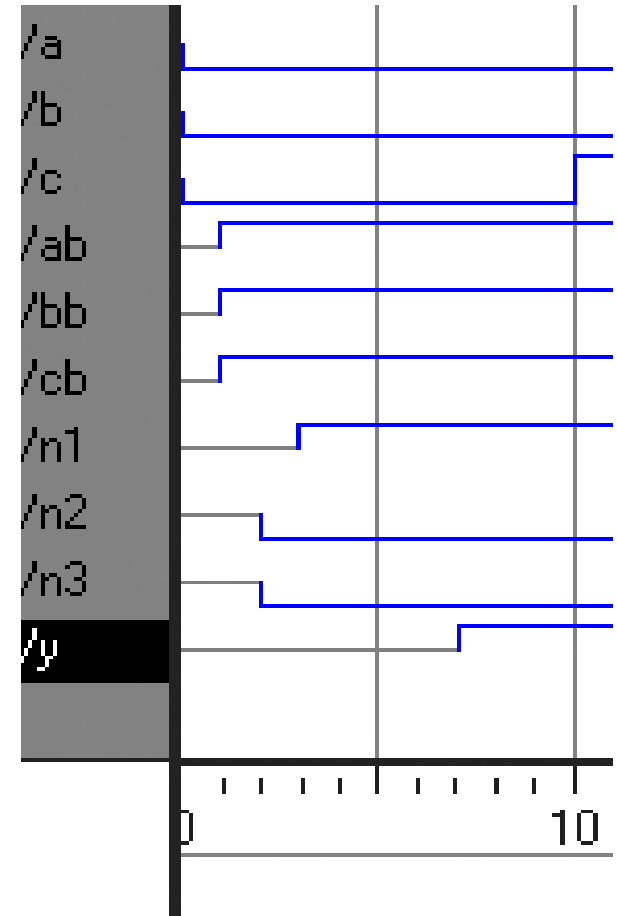
Delays

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



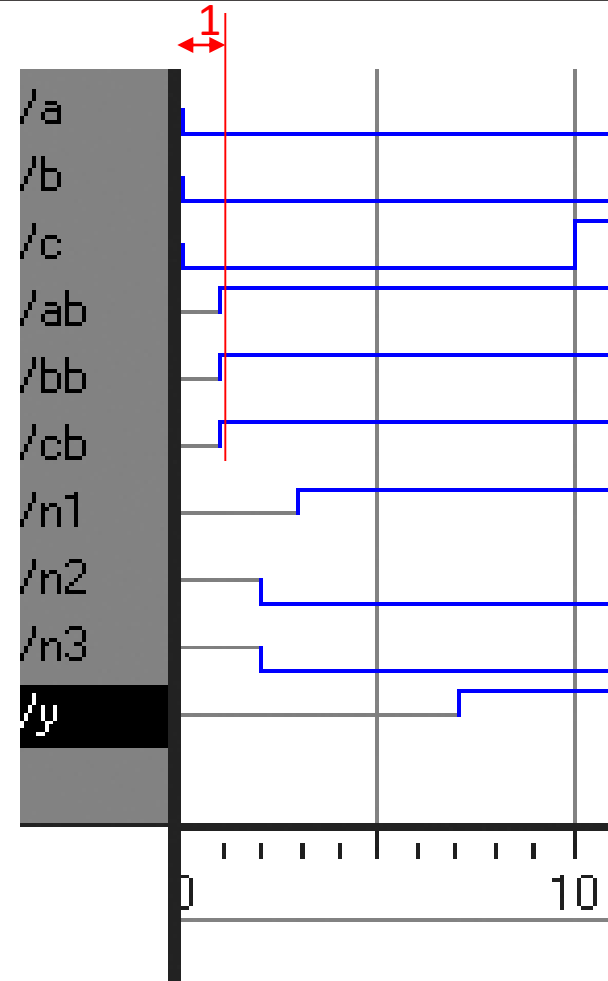
Delays

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



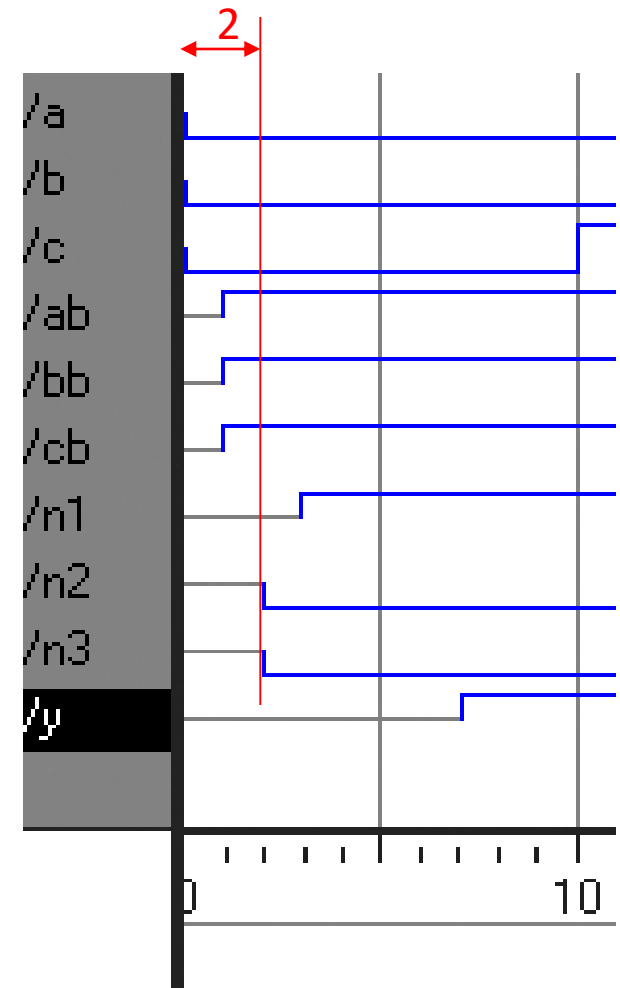
Delays

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



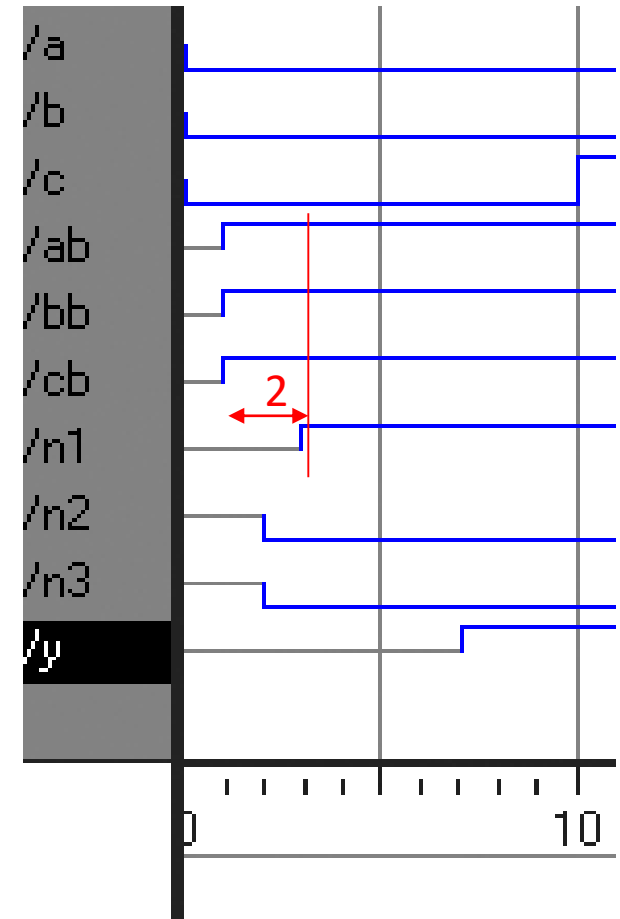
Delays

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



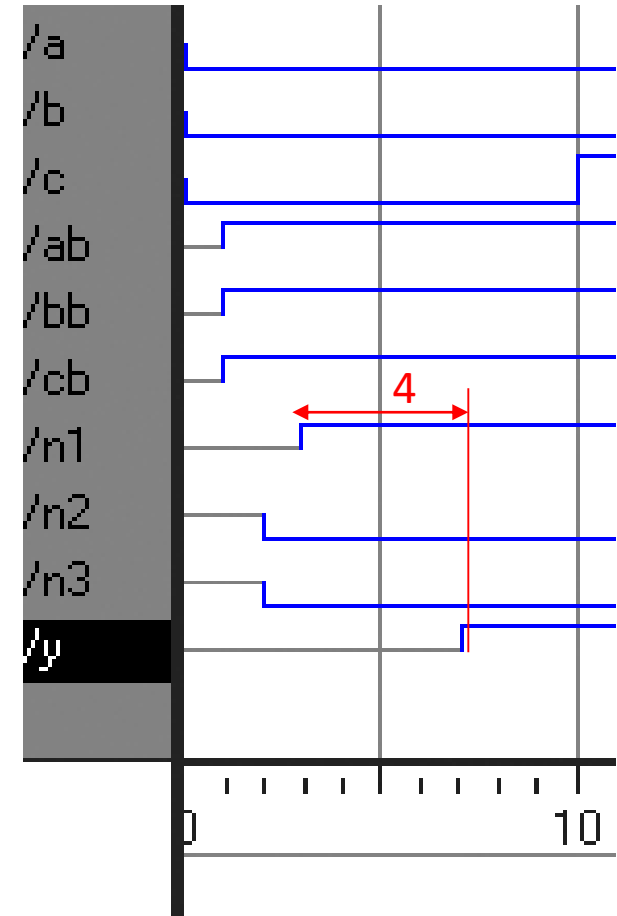
Delays

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



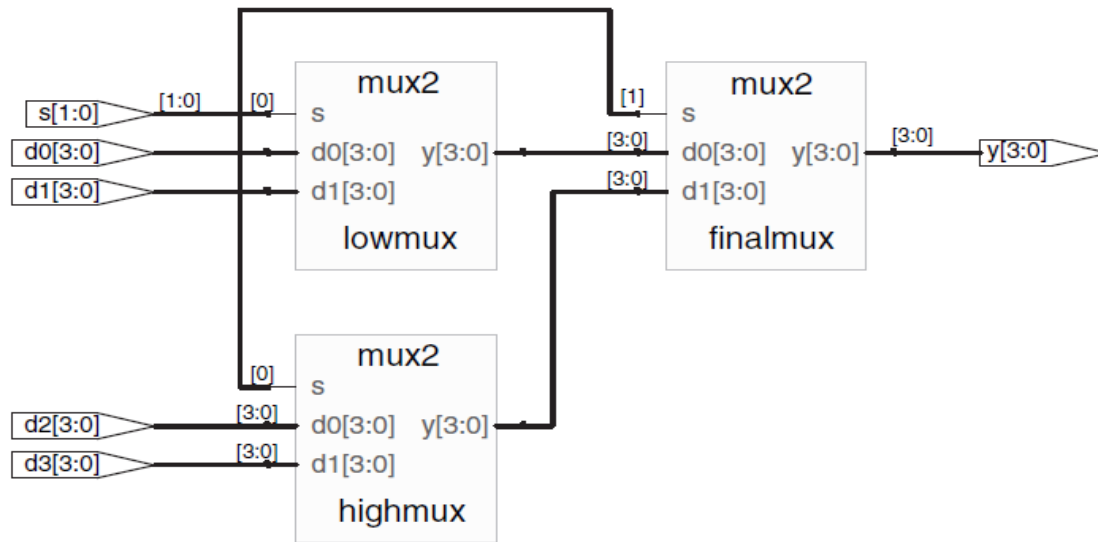
Delays

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



STRUCTURAL MODELING

- The previous section discussed **behavioral modeling**, describing a module in terms of the relationships between inputs and outputs.
- This section examines **structural modeling**, describing a module in terms of how it is composed of simpler modules.
- For example, HDL Example 4.14 shows how to assemble a 4:1 multiplexer from three 2:1 multiplexers



SystemVerilog

```

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

    logic [3:0] low, high;

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

```

The three mux2 instances are called `lowmux`, `highmux`, and `finalmux`. The mux2 module must be defined elsewhere in the SystemVerilog code — see [HDL Example 4.5](#), [4.15](#), or [4.34](#).

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.

HDL Example 4.15 uses structural modeling to construct a 2:1 multiplexer from a pair of tristate buffers. Building logic out of tristates is not recommended, however.

HDL Example 4.15 STRUCTURAL MODEL OF 2:1 MULTIPLEXER

SystemVerilog

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

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

In SystemVerilog, 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.

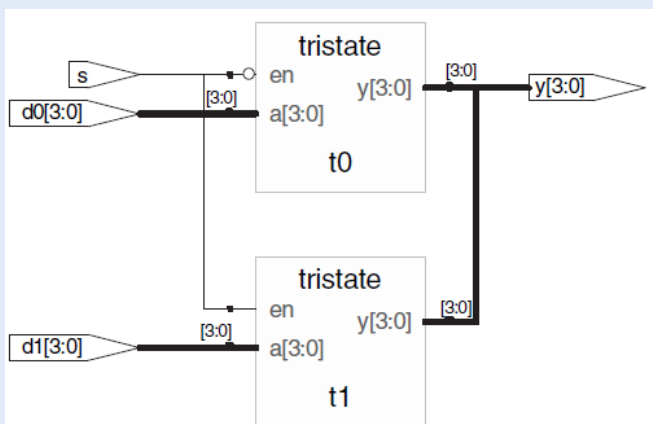


Figure 4.12 mux2 synthesized circuit

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.

- **HDL Example 4.16** shows how modules can access part of a bus. An 8-bit wide 2:1 multiplexer is built using two of the 4-bit 2:1 multiplexers already defined, operating on the low and high nibbles of the byte.
- In general, **complex systems are designed hierarchically**. The overall system is described structurally by instantiating its major components. Each of these components is described structurally from its building blocks, and so forth recursively until the pieces are simple enough to describe behaviorally. It is good style to avoid (or at least to minimize) mixing structural and behavioral descriptions within a single module.

HDL Example 4.16 ACCESSING PARTS OF BUSESSES

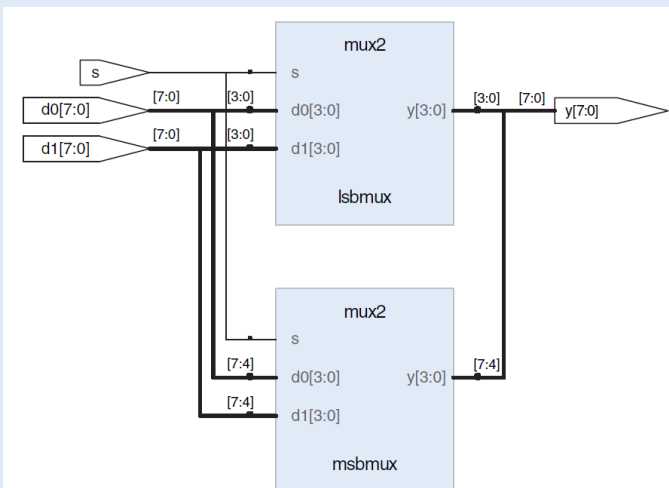
SystemVerilog

```

module mux2_8(input  logic [7:0] d0, d1,
             input  logic      s,
             output logic [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));
    msbmux: mux2
        port map(d0(7 downto 4), d1(7 downto 4),
                s, y(7 downto 4));
end;

```

Sequential Logic

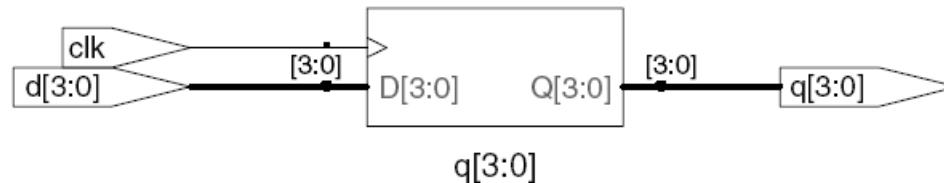
- SystemVerilog uses **idioms** to describe latches, flip-flops and FSMs
- HDL synthesizers recognize certain idioms and turn them into specific sequential circuits.
- Other coding styles may simulate correctly but produce incorrect hardware



D Flip-Flop

The vast majority of modern commercial systems are built with registers using positive edge-triggered D flip-flops. [HDL Example 4.17](#) shows the idiom for such flip-flops.

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



Always Statement

General Structure:

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

Whenever the event in `sensitivity list` occurs, `statement` is executed

In SystemVerilog `always` statements and VHDL process statements, signals **keep their old value until an event in the sensitivity list takes place** that explicitly causes them to change. Hence, such code, with appropriate sensitivity lists, can be used to describe sequential circuits with memory. For example, the flip-flop includes only `clk` in the sensitive list. It remembers its old value of `q` until the next rising edge of the `clk`, even if `d` changes in the interim.

In contrast, SystemVerilog continuous assignment statements (`assign`) and VHDL concurrent assignment statements (`<=>`) are reevaluated anytime any of the inputs on the right hand side changes. Therefore, such code necessarily describes combinational logic.



SystemVerilog

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

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

In general, a SystemVerilog `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`. Note that sensitivity lists are also referred to as stimulus lists.

`<=` 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.

As will be seen in subsequent sections, `always` statements can be used to imply flip-flops, latches, or combinational logic, depending on the sensitivity list and statement. Because of this flexibility, it is easy to produce the wrong hardware inadvertently. SystemVerilog introduces `always_ff`, `always_latch`, and `always_comb` to reduce the risk of common errors. `always_ff` behaves like `always` but is used exclusively to imply flip-flops and allows tools to produce a warning if anything else is implied.

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 rising_edge(clk) 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 checks if the change was a rising edge on `clk`. If so, 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 clk'event and clk = '1' then
        q <= d;
    end if;
end process;
```

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

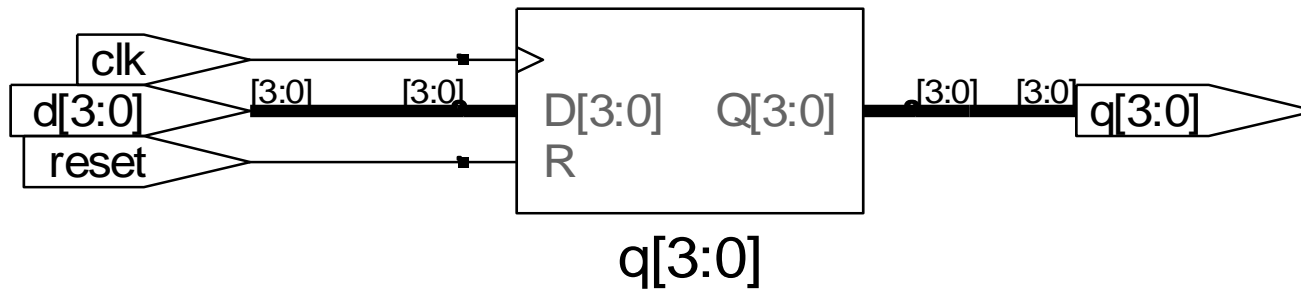
Resettable D Flip-Flop

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

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

endmodule
```

When simulation begins or power is first applied to a circuit, the output of a flop or register is unknown. This is indicated with x in SystemVerilog and u in VHDL. Generally, it is good practice to use resettable registers so that on powerup you can put your system in a known state. The reset may be either asynchronous or synchronous. Recall that asynchronous reset occurs immediately, whereas synchronous reset clears the output only on the next rising edge of the clock.



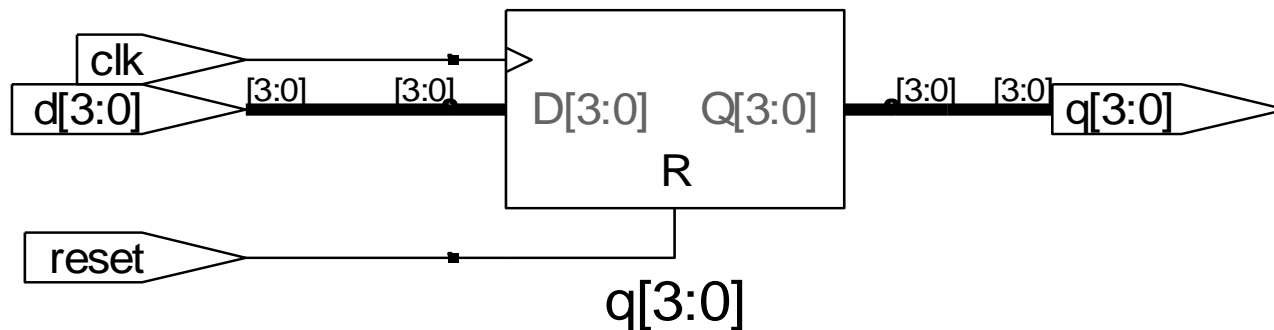
Resettable D Flip-Flop

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

```
    // asynchronous reset
```

```
    always_ff @(posedge clk, posedge reset)  
        if (reset) q <= 4'b0;  
        else      q <= d;
```

```
endmodule
```



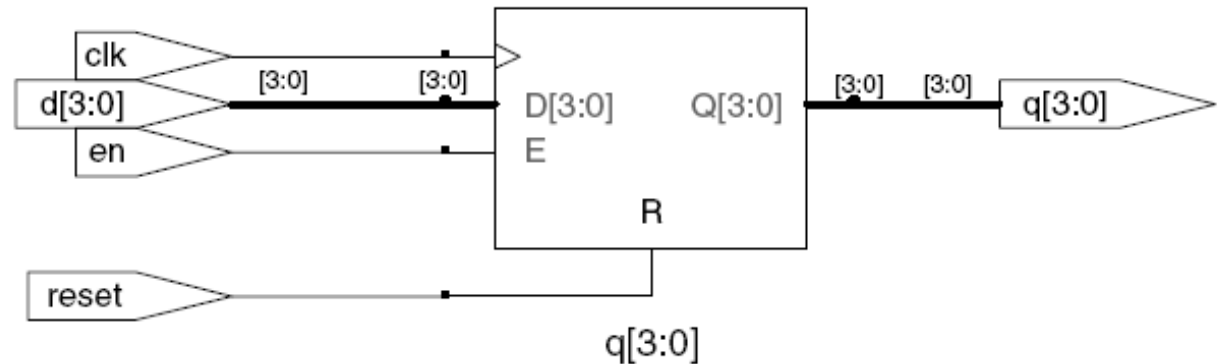
D Flip-Flop with Enable

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

```
// enable and asynchronous reset
```

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

```
endmodule
```



Enabled registers respond to the clock only when the enable is asserted.

[HDL Example 4.19](#) shows an asynchronously resettable enabled register that retains its old value if both reset and en are FALSE.



SystemVerilog

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

// asynchronous reset
always_ff@(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;

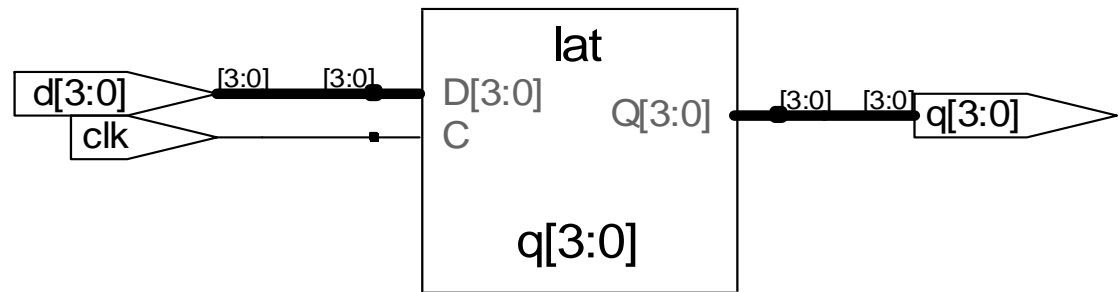
architecture asynchronous of flopenr is
-- asynchronous reset
begin
    process(clk, reset) begin
        if reset then
            q <= "0000";
        elsif rising_edge(clk) then
            if en then
                q <= d;
            end if;
        end if;
    end process;
end;
```



Latch

```
module latch(input logic clk,  
             input logic [3:0] d,  
             output logic [3:0] q);  
  
    always_latch  
        if (clk) q <= d;  
  
endmodule
```

Recall from Section 3.2.2 that a D latch is transparent when the clock is HIGH, allowing data to flow from input to output. The latch becomes opaque when the clock is LOW, retaining its old state. [HDL Example 4.21](#) shows the idiom for a D latch.



Warning: We don't use latches in this text. But you might write code that inadvertently implies a latch. Check synthesized hardware – if it has latches in it, there's an error.



HDL Example 4.21 D LATCH

SystemVerilog

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

    always_latch
        if (clk) q <= d;
endmodule
```

`always_latch` is equivalent to `always @(clk, d)` and is the preferred idiom for describing a latch in SystemVerilog. It evaluates any time `clk` or `d` changes. If `clk` is HIGH, `d` flows through to `q`, so this code describes a positive level sensitive latch. Otherwise, `q` keeps its old value. SystemVerilog can generate a warning if the `always_latch` block doesn't imply a latch.

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`.

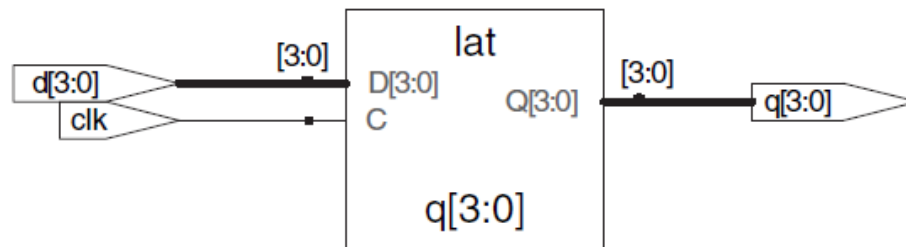


Figure 4.19 latch synthesized circuit

In [Section 4.2](#), we used assignment statements to describe combinational logic behaviorally. SystemVerilog always statements and VHDL process statements are used to describe sequential circuits, because they remember the old state when no new state is prescribed. However, always/process statements can also be used to describe combinational logic behaviorally if the sensitivity list is written to respond to changes in all of the inputs and the body prescribes the output value for every possible input combination. HDL Example 4.22 uses always/process statements to describe a bank of four inverters (see Figure 4.3 for the synthesized circuit).

HDLs support blocking and nonblocking assignments in an always/process statement. A group of blocking assignments are evaluated in the order in which they appear in the code, just as one would expect in a standard programming language. A group of nonblocking assignments are evaluated concurrently; all of the statements are evaluated before any of the signals on the left hand sides are updated.

SystemVerilog

In a SystemVerilog `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 blocking assignment and `<=` indicates a nonblocking assignment (also called a concurrent assignment). This is the first section where `:=` is introduced.

Nonblocking assignments are made to outputs and to signals. Blocking assignments are made to variables, which are declared in `process` statements (see [HDL Example 4.23](#)). `<=` can also appear outside `process` statements, where it is also evaluated concurrently.



HDL Example 4.22 INVERTER USING always/process

SystemVerilog

```
module inv(input logic [3:0] a,  
           output logic [3:0] y);  
  
    always_comb  
        y=~a;  
endmodule
```

`always_comb` reevaluates the statements inside the `always` statement any time any of the signals on the right hand side of `<=` or `=` in the `always` statement change. In this case, it is equivalent to `always @(a)`, but is better because it avoids mistakes if signals in the `always` statement are renamed or added. If the code inside the `always` block is not combinational logic, SystemVerilog will report a warning. `always_comb` is equivalent to `always @(*)`, but is preferred in SystemVerilog.

The `=` in the `always` statement is called a *blocking assignment*, in contrast to the `<=` nonblocking assignment. In SystemVerilog, 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](#).

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(all) begin  
        y <= not a;  
    end process;  
end;
```

`process(all)` reevaluates the statements inside the `process` any time any of the signals in the `process` change. It is equivalent to `process(a)` but is better because it avoids mistakes if signals in the `process` are renamed or added.

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

Other Behavioral Statements

- Statements that must be inside `always` statements:
 - `if / else`
 - `case, casez`



Combinational Logic using always

```
// combinational logic using an always statement
module gates(input logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
    always_comb // need begin/end because there is
    begin      // more than one statement in always
        y1 = a & b; // AND
        y2 = a | b; // OR
        y3 = a ^ b; // XOR
        y4 = ~(a & b); // NAND
        y5 = ~(a | b); // NOR
    end
endmodule
```

This hardware could be described with assign statements using fewer lines of code, so it's better to use assign statements in this case.



Combinational Logic using case

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);

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



Combinational Logic using case

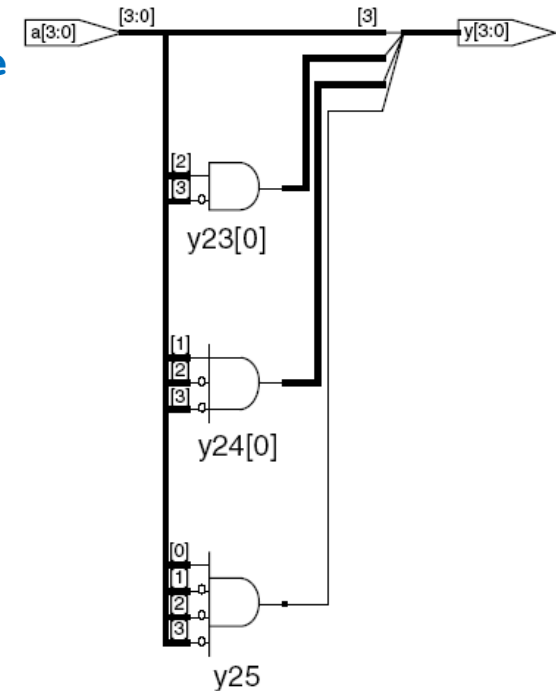
- case statement implies combinational logic **only if** all possible input combinations described
- Remember to use **default** statement



Combinational Logic using casez

```
module priority_casez(input logic [3:0] a,  
                    output logic [3:0] y);  
  
    always_comb  
        casez (a)  
            4'b1???: y = 4'b1000; // ?=don't care  
            4'b01??: y = 4'b0100;  
            4'b001?: y = 4'b0010;  
            4'b0001: y = 4'b0001;  
            default: y = 4'b0000;  
        endcase  
    endmodule
```

The casez statement acts like a case statement except that it also recognizes ? as don't care.



HDL Example 4.27 PRIORITY CIRCUIT USING DON'T CARES

SystemVerilog

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

The casez statement acts like a case statement except that it also recognizes ? as don't care.

VHDL

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

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

architecture dontcare of priority_casez is
begin
    process(all) begin
        case? a is
            when "1---" => y <= "1000";
            when "01--" => y <= "0100";
            when "001-" => y <= "0010";
            when "0001" => y <= "0001";
            when others => y <= "0000";
        end case?;
    end process;
end;
```

The case? statement acts like a case statement except that it also recognizes - as don't care.

Blocking and Nonblocking Assignments

- The guidelines on the next page explain when and how to use each type of assignment.
- If these guidelines are not followed, it is possible to write code that appears
 - to work in simulation
 - but synthesizes to incorrect hardware.

BLOCKING AND NONBLOCKING ASSIGNMENT GUIDELINES

SystemVerilog

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

```
always_ff @(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_comb` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful.

```
always_comb
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 rising_edge(clk) 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(all)` to model more complicated combinational logic where the `process` is helpful. Use blocking assignments for internal variables.

```
process(all)
    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;
```

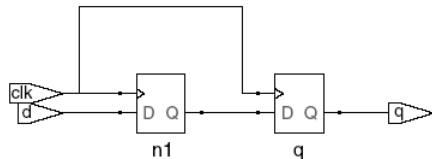
4. Do not make assignments to the same variable in more than one `process` or concurrent assignment statement.

Blocking vs. Nonblocking Assignment

- `<=` is **nonblocking** assignment
 - Occurs simultaneously with others
- `=` is **blocking** assignment
 - Occurs in order it appears in file

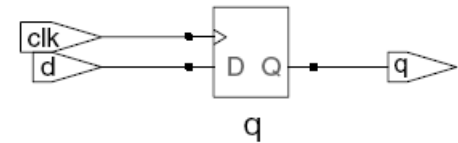
```
// Good synchronizer using
// nonblocking assignments
module syncgood(input logic clk,
                input logic d,
                output logic q);

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



```
// Bad synchronizer using
// blocking assignments
module syncbad(input logic clk,
               input logic d,
               output logic q);

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



Rules for Signal Assignment

- **Synchronous sequential logic:** use `always_ff @ (posedge clk)` and nonblocking assignments (`<=>`)

```
always_ff @ (posedge clk)
    q <= d; // nonblocking
```

- **Simple combinational logic:** use continuous assignments (`assign...`)

```
assign y = a & b;
```

- **More complicated combinational logic:** use `always_comb` and blocking assignments (`=`)
- Assign a signal in **only one** `always` statement or continuous assignment statement.

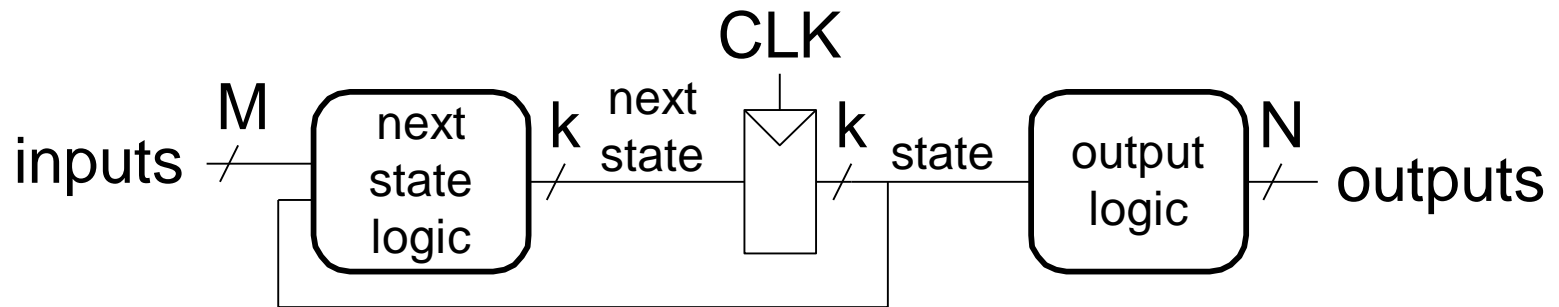


Finite State Machines (FSMs)

- Recall that a finite state machine (FSM) consists of a state register and two blocks of combinational logic to compute the next state and the output given the current state and the input. HDL descriptions of state machines are correspondingly divided into three parts to model the state register, the next state logic, and the output logic.

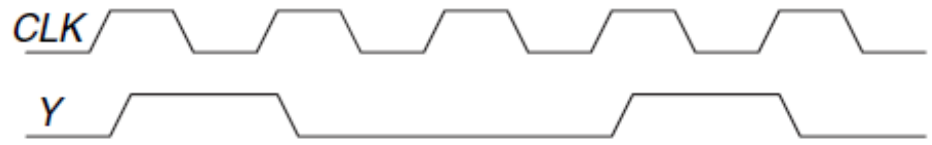
- Three blocks:**

- next state logic
- state register
- output logic



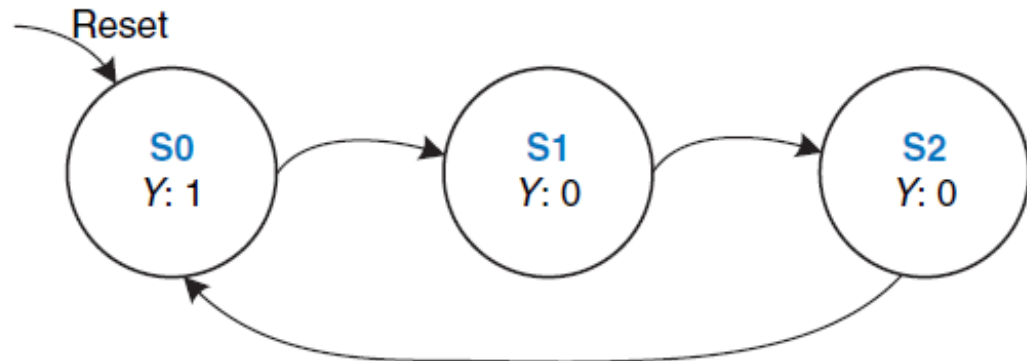
Divide-by-3 Counter

A divide-by-N counter has one output and no inputs. The output Y is HIGH for one clock cycle out of every N. In other words, the output divides the frequency of the clock by N. The waveform and state transition diagram for a divide-by-3 counter is shown in [Figure 3.28](#). Sketch circuit designs for such a counter using binary and one-hot state encodings.



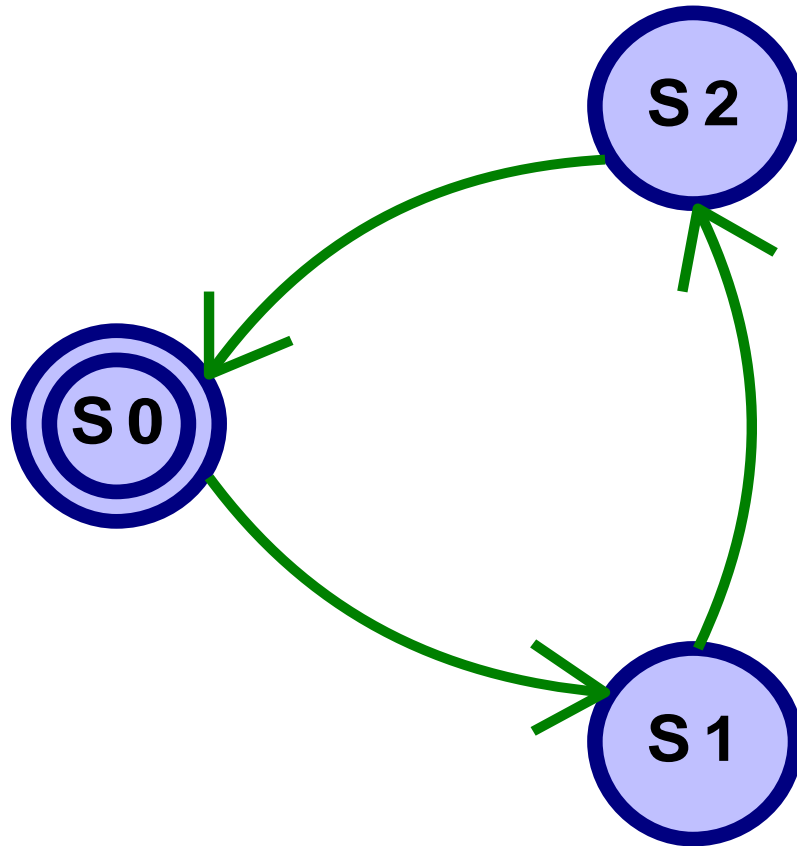
(a)

Figure 3.28 Divide-by-3 counter
(a) waveform and (b) state transition diagram



(b)

FSM Example: Divide by 3



The double circle indicates the reset state



Table 3.8 compares binary and one-hot encodings for the three states.

The binary encoding uses two bits of state. Using this encoding, the state transition table is shown in Table 3.9. Note that there are no inputs; the next state depends only on the current state. The output table is left as an exercise to the reader. The next-state and output equations are:

$$\begin{aligned} S'_1 &= \bar{S}_1 S_0 \\ S'_0 &= \bar{S}_1 \bar{S}_0 \end{aligned} \quad (3.4)$$

$$Y = \bar{S}_1 \bar{S}_0 \quad (3.5)$$

The one-hot encoding uses three bits of state. The state transition table for this encoding is shown in Table 3.10 and the output table is again left as an exercise to the reader. The next-state and output equations are as follows:

$$\begin{aligned} S'_2 &= S_1 \\ S'_1 &= S_0 \\ S'_0 &= S_2 \end{aligned} \quad (3.6)$$

$$Y = S_0 \quad (3.7)$$

Table 3.8 One-hot and binary encodings for divide-by-3 counter

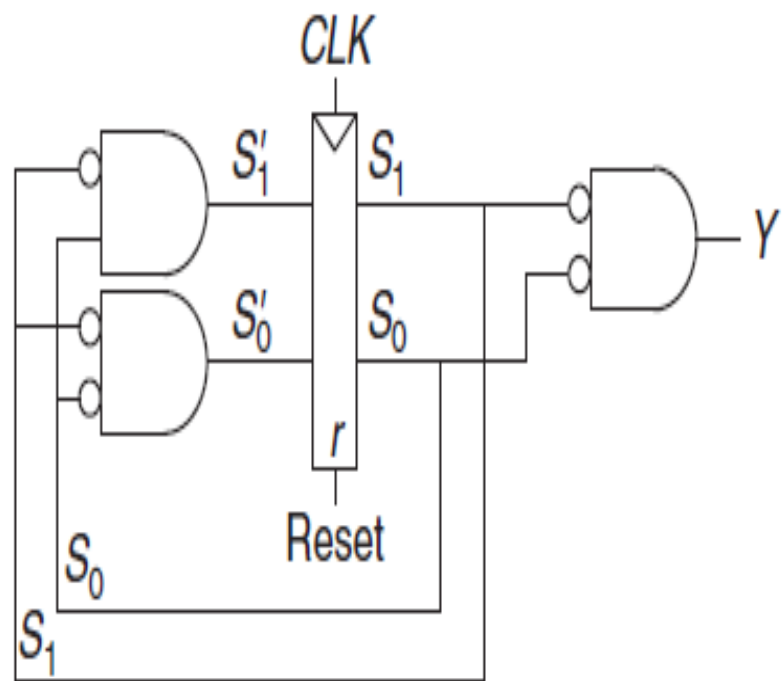
State	One-Hot Encoding			Binary Encoding	
	S_2	S_1	S_0	S_1	S_0
S0	0	0	1	0	0
S1	0	1	0	0	1
S2	1	0	0	1	0

Table 3.9 State transition table with binary encoding

Current State		Next State	
S_1	S_0	S'_1	S'_0
0	0	0	1
0	1	1	0
1	0	0	0

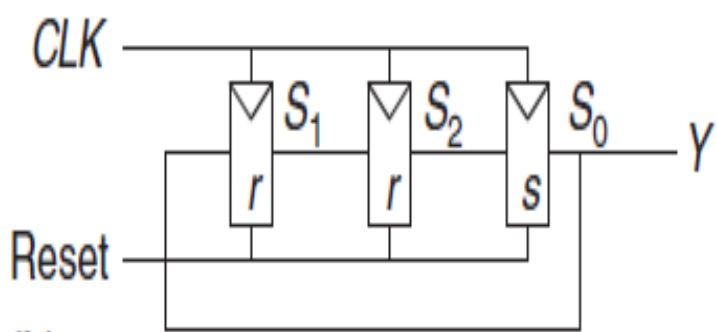
Table 3.10 State transition table with one-hot encoding

Current State			Next State		
S_2	S_1	S_0	S'_2	S'_1	S'_0
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1



next state logic state register output logic output

(a)



(b)

Figure 3.29 Divide-by-3 circuits for (a) binary and (b) one-hot encodings

FSM in SystemVerilog

```
module divideby3FSM (input  logic clk,
                    input  logic reset,
                    output logic q);

    typedef enum logic [1:0] {S0, S1, S2} statetype;
    statetype [1:0] state, nextstate;

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

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

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



SystemVerilog

```

module divideby3FSM(input  logic clk,
                   input  logic reset,
                   output logic y);
    typedef enum logic [1:0] {S0, S1, S2} statetype;
    statetype state, nextstate;

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

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

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

```

The typedef statement defines `statetype` to be a two-bit logic value with three possibilities: `S0`, `S1`, or `S2`. `state` and `nextstate` are `statetype` signals.

The enumerated encodings default to numerical order: `S0 = 00`, `S1 = 01`, and `S2 = 10`. The encodings can be explicitly set by the user; however, the synthesis tool views them as suggestions, not requirements. For example, the following snippet encodes the states as 3-bit one-hot values:

```

typedef enum logic [2:0] {S0=3'b001, S1=3'b010, S2=3'b100}
statetype;

```

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 then state <= S0;
        elsif rising_edge(clk) 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`.

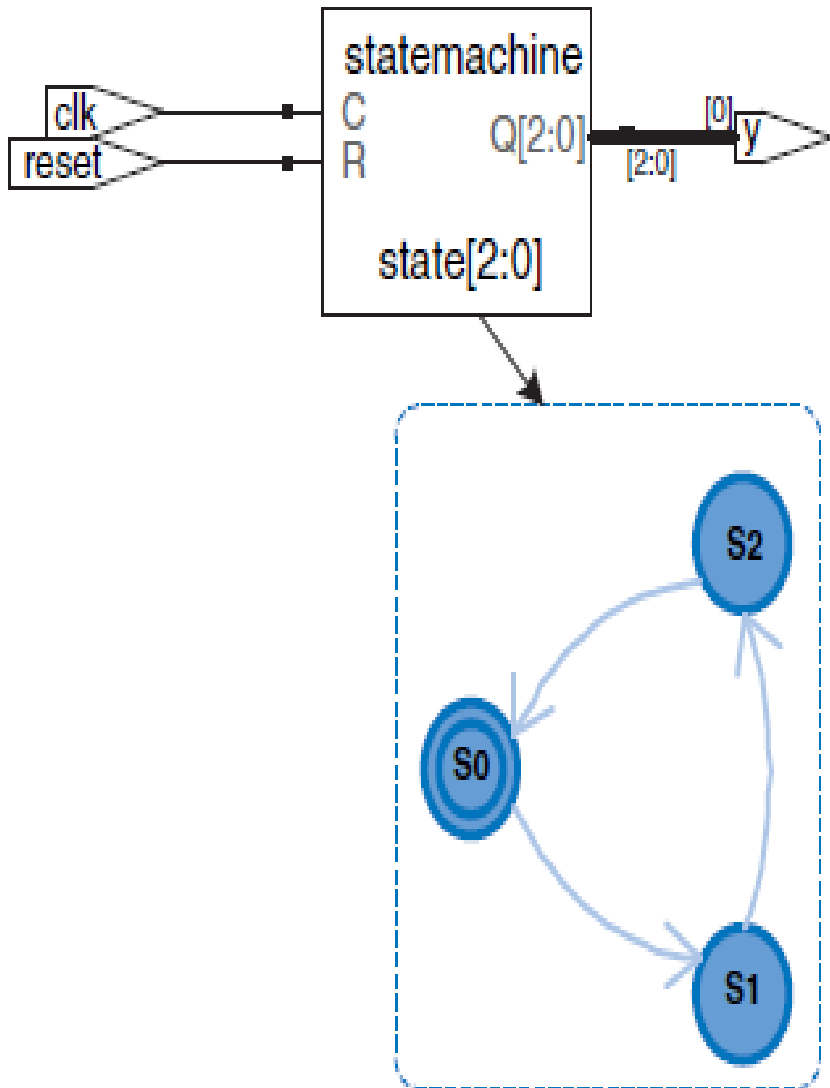


Figure 4.25 divideby3fsm synthesized circuit

Prior to SystemVerilog, Verilog primarily used two types: `reg` and `wire`. Despite its name, a `reg` signal might or might not be associated with a register. This was a great source of confusion for those learning the language. SystemVerilog introduced the `logic` type to eliminate the confusion; hence, this book emphasizes the `logic` type. This section explains the `reg` and `wire` types in more detail for those who need to read old Verilog code.

In Verilog, if a signal appears on the left hand side of `<=` or `=` in an `always` block, it must be declared as `reg`. Otherwise, it should be declared as `wire`. Hence, a `reg` signal might be the output of a flip-flop, a latch, or combinational logic, depending on the sensitivity list and statement of an `always` block.

Input and output ports default to the `wire` type unless their type is explicitly defined as `reg`. The following example shows how a flip-flop is described in conventional Verilog. Note that `clk` and `d` default to `wire`, while `q` is explicitly defined as `reg` because it appears on the left hand side of `<=` in the `always` block.

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

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

SystemVerilog introduces the `logic` type. `logic` is a synonym for `reg` and avoids misleading users about whether it is actually a flip-flop. More-

Parameterized Modules

- So far all of our modules have had fixed-width inputs and outputs.
- For example, we had to define separate modules for 4- and 8-bit wide 2:1 multiplexers.
- HDLs permit variable bit widths using parameterized modules.
- HDL Example 4.34 declares a parameterized 2:1 multiplexer with a default width of 8, then uses it to create 8- and 12-bit 4:1 multiplexers.



Parameterized Modules

2:1 mux:

```
module mux2
  #(parameter width = 8) // name and default value
  (input logic [width-1:0] d0, d1,
   input logic s,
   output logic [width-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

Instance with 8-bit bus width (uses default):

```
mux2 myMux(d0, d1, s, out);
```

Instance with 12-bit bus width:

```
mux2 #(12) lowmux(d0, d1, s, out);
```

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 logic [11:0] d0, d1, d2, d3,
               input logic [1:0] s,
               output logic [11:0] y);
  logic [11:0] low, hi;
  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], 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 <= d1 when s else d0;
end;
```

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

Testbenches

- A testbench is an HDL module that is used to test another module, called the device under test (DUT).
- The testbench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced.
- The input and desired output patterns are called test vectors.
- It instantiates the DUT, then applies the inputs. Blocking assignments and delays are used to apply the inputs in the appropriate order.
- The user must view the results of the simulation and verify by inspection that the correct outputs are produced.
- Testbenches are simulated the same as other HDL modules. However, they are not synthesizable.



Testbenches Types

- Types:
 - Simple
 - Self-checking
 - Self-checking with testvectors



Testbench Example

- Write SystemVerilog code to implement the following function in hardware:

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

- Name the module `sillyfunction`



Testbench Example

- Write SystemVerilog code to implement the following function in hardware:

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

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



Simple Testbench

```
module testbench1();
  logic a, b, c;
  logic y;
  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // apply inputs one at a time and wait 10 time units
  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
```



Simple Testbenches

- A simple testbench instantiates the DUT, then applies the inputs.
- Blocking assignments and delays are used to apply the inputs in the appropriate order.
- **The user must view the results** of the simulation and verify by inspection that the correct outputs are produced.



SystemVerilog

```

module testbench1();
  logic a, b, c, 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.

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 Testbenches

- Checking for correct outputs is tedious and error-prone. Moreover, determining the correct outputs is much easier when the design is fresh in your mind; if you make minor changes and need to retest weeks later, determining the correct outputs becomes a hassle.
- A much better approach is to write a self-checking testbench.



Self-checking Testbench

```
module testbench2();
  logic a, b, c;
  logic y;
  sillyfunction dut(a, b, c, y); // instantiate dut
  initial begin // apply inputs, check results one at a
    time
      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
  end
endmodule
```

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



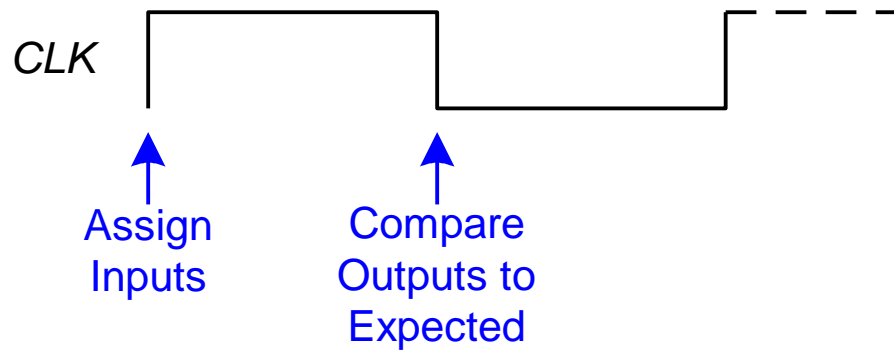
Testbench with Testvectors

- Writing code for each test vector also becomes tedious, especially for modules that require a large number of vectors. An even better approach is to place the test vectors in a separate file.
- The testbench simply reads the test vectors from the file, applies the input test vector to the DUT, waits, checks that the output values from the DUT match the output vector, and repeats until reaching the end of the test vectors file.
- Testvector file: inputs and expected outputs
- Testbench:
 1. Generate clock for assigning inputs, reading outputs
 2. Read testvectors file into array
 3. Assign inputs, expected outputs
 4. Compare outputs with expected outputs and report errors



Testbench with Testvectors

- Testbench clock:
 - assign inputs (on rising edge)
 - compare outputs with expected outputs (on falling edge).



- Testbench clock also used as clock for synchronous sequential circuits



Testvectors File

- **File:** `example.tv` (a text file containing the inputs and expected output written in binary)
- contains vectors of `abc_y` expected

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



1. Generate Clock

```
module testbench3();  
    logic        clk, reset;  
    logic        a, b, c, yexpected;  
    logic        y;  
    logic [31:0] vectornum, errors;    // bookkeeping variables  
    logic [3:0]  testvectors[10000:0]; // array of testvectors  
  
    // instantiate device under test  
    sillyfunction dut(a, b, c, y);  
  
    // generate clock  
    always        // no sensitivity list, so it always executes  
        begin  
            clk = 1; #5; clk = 0; #5;  
        end
```



2. Read Testvectors into Array

```
// 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
```

```
// Note: $readmemb reads testvector files written in
// hexadecimal
```



3. Assign Inputs & Expected Outputs

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



4. Compare with Expected Outputs

```
// 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
```

```
// Note: to print in hexadecimal, use %h. For example,
```

```
//      $display("Error: inputs = %h", {a, b, c});
```



4. Compare with Expected Outputs

```
// increment array index and read next testvector
    vectornum = vectornum + 1;
    if (testvectors[vectornum] === 4'bx) begin
        $display("%d tests completed with %d errors",
            vectornum, errors);
    $finish;
    end
end
endmodule
```

// **===** and **!==** can compare values that are 1, 0, x, or z.



HDL Example 4.39 TESTBENCH WITH TEST VECTOR FILE

SystemVerilog

```
module testbench3();
  logic      clk, reset;
  logic      a, b, c, y, yexpected;
  logic [31:0] vectornum, errors;
  logic [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 // check result
      $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
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.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 y_expected: STD_LOGIC;
  signal clk, reset: STD_LOGIC;
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, pulse reset
  process begin
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;

  -- run tests
  process is
    file tv: text;
    variable L: line;
    variable vector_in: std_logic_vector(2 downto 0);
    variable dummy: character;
    variable vector_out: std_logic;
    variable vectornum: integer := 0;
    variable errors: integer := 0;
  begin
    FILE_OPEN(tv, "example.tv", READ_MODE);
    while not endfile(tv) loop

      -- change vectors on rising edge
      wait until rising_edge(clk);

      -- read the next line of testvectors and split into pieces
      readline(tv, L);
      read(L, vector_in);
      read(L, dummy); -- skip over underscore
```

`$readmemb` reads a file of binary numbers into the `testvectors` array. `$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 (`a`, `b`, and `c`) and the expected output (`yexpected`) based on the four bits in the current test vector.

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. `$display` is a system task to print in the simulator window. For example, `$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 SystemVerilog module supports up to 10,001 test vectors, it will terminate the simulation after executing the eight vectors in the file.

```
read(L, vector_out);
(a, b, c) <= vector_in(2 downto 0) after 1 ns;
y_expected <= vector_out after 1 ns;

-- check results on falling edge
wait until falling_edge(clk);

if y /= y_expected then
    report "Error: y = " & std_logic'image(y);
    errors := errors+1;
end if;

vectornum := vectornum+1;
end loop;

-- summarize results at end of simulation
if (errors=0) then
    report "NO ERRORS -- " &
        integer'image(vectornum) &
        " tests completed successfully."
        severity failure;
else
    report integer'image(vectornum) &
        " tests completed, errors = " &
        integer'image(errors)
        severity failure;
end if;
end process;
end;
```

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

Summary

- Hardware description languages (HDLs) are **extremely important tools** for modern digital designers. Once you have learned SystemVerilog or VHDL, you will be able to specify digital systems much faster than if you had to draw the complete schematics.
- The debug cycle is also often much faster, because modifications require code changes instead of tedious schematic rewiring. However, the debug cycle can be much longer using HDLs if you don't have a good idea of the hardware your code implies.
- HDLs are used for both simulation and synthesis. Logic simulation is a powerful way to test a system on a computer before it is turned into hardware. Simulators let you check the values of signals inside your system that might be impossible to measure on a physical piece of hardware.
- Logic synthesis converts the HDL code into digital logic circuits.
- The most important thing to remember when you are writing HDL code is that you are describing real hardware, not writing a computer program.
- The most common beginner's mistake is to write HDL code without thinking about the hardware you intend to produce.

