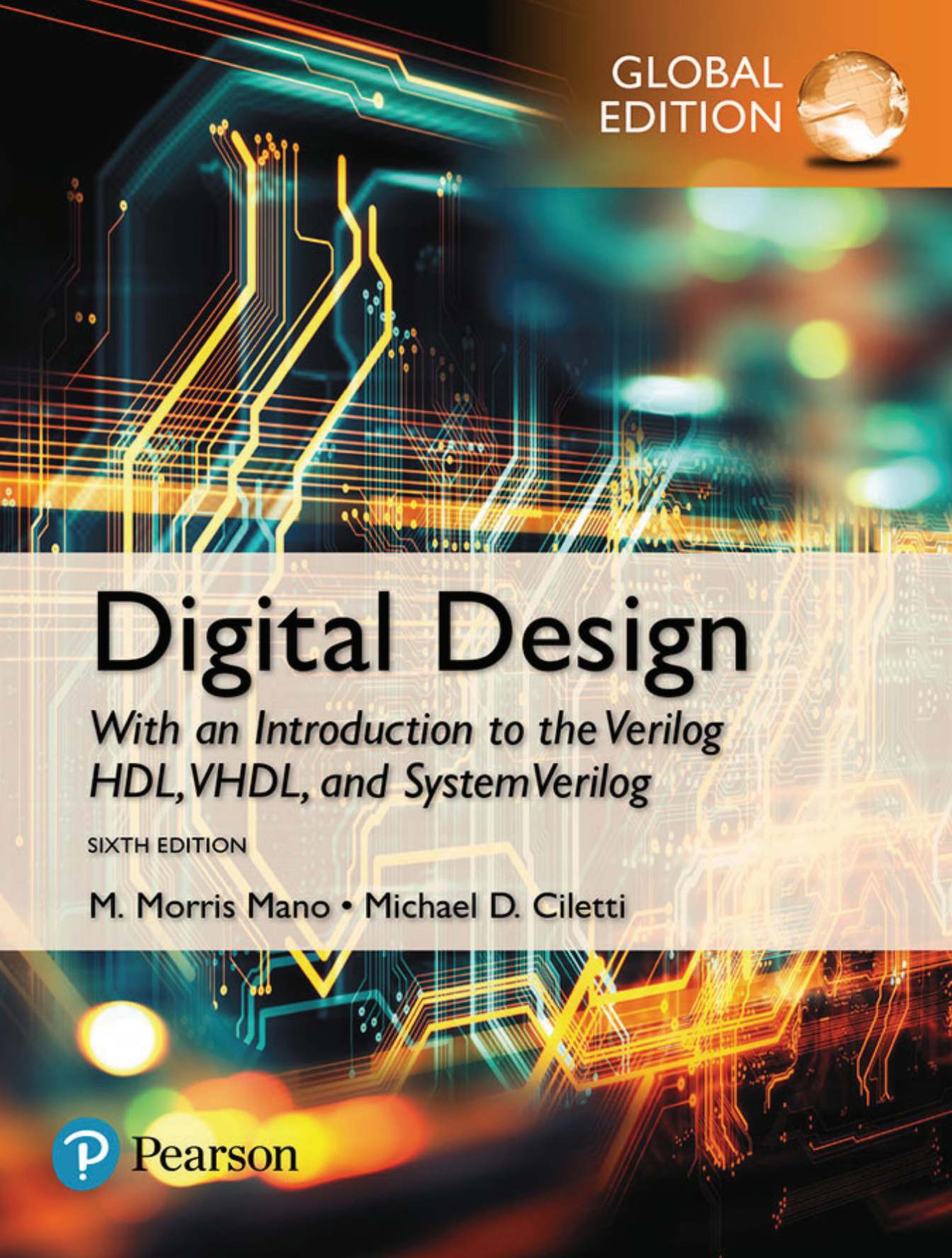
GLOBAL
EDITION
Digital Design
With an Introduction to the Verilog
HDL, VHDL, and SystemVerilog
SIXTH EDITION
M. Morris Mano • Michael D. Ciletti
Pearson

Digital Design

With an Introduction to the Verilog HDL,
VHDL, and SystemVerilog

design, the four-bit adder is defined first, and then full adders are defined and interconnected. In a bottom-up design, the half adder is defined, then the full adder is constructed; the four-bit adder is built by instantiating and interconnecting the full-adders.[10]

HDL Example 4.2 (Hierarchical Modeling—Eight-Bit Adder)

Verilog

At the bottom of the design hierarchy shown in Fig. 4.33 a half adder is composed of primitive gates. At the next level of the hierarchy, a full adder is formed by instantiating and connecting a pair of half adders. The third module describes the eight-bit adder by instantiating and linking together two four-bit adders. This example illustrates optional Verilog 2001, 2005 syntax, which eliminates extra typing of identifiers declaring the mode (e.g., **output**), type (**reg**), and declaration of a vector range (e.g., [3: 0]) of a port. The first version of the standard (1995) uses separate statements for these declarations; the revised standard includes the declarations within the port.

```
module Add_half (input a, b, output c_out, sum),
  xor G1(sum, a, b);          // Gate instance names are optional
  and G2(c_out, a, b);
endmodule

module Add_full (input a, b, c_in, output c_out, sum);   // see Fig. 4.8
  wire w1, w2, w3;          // w1 is c_out; w2 is sum
  Add_half M1 (a, b, w1, w2);
  Add_half M0 (w2, c_in, w3, sum);
  or (c_out, w1, w3);
endmodule

module Add_rca_4 (input [3:0] a, b, input c_in output c_out, output [3:0] sum);
  wire c_in1, c_in3, c_in4;        // Intermediate carries
  Add_full M0 (a[0], b[0], c_in, c_in1, sum[0]);
  Add_full M1 (a[1], b[1], c_in1, c_in2, sum[1]);
  Add_full M2 (a[2], b[2], c_in2, c_in3, sum[2]);
  Add_full M3 (a[3], b[3], c_in3, c_out, sum[3]);
endmodule

module Add_rca_8 (input [7:0] a, b, input c_in, output c_out, output [7:0] sum,)
  wire c_in4;
  Add_rca_4 M0 (a[3:0], b[3:0], c_in, c_in4, sum[3:0]);
  Add_rca_4 M1 (a[7:4], b[7:4], c_in4, c_out, sum[7:4]);
endmodule
```

[10] Note that the first character of an identifier cannot be a number, but can be an underscore. Thus, the eight-bit adder could be named *_8bit_adder*. An alternative name that is meaningful and does not present the possibility of neglecting the leading underscore character is *Add_rca_*8.

Verilog modules can be instantiated within other modules, but module declarations cannot be nested; that is, a module declaration cannot be inserted into the text between the **module** and **endmodule** keywords of another module. Also, instance names (e.g., M0) must be specified when a module is instantiated within another module.

VHDL

A VHDL hierarchical model of *Add_rca_8_vhdl*, an 8-bit adder, constructs components for the logic gates in Fig. 4.34, and uses them in the half adders and full adders. Once *Add_full_vhdl* and *Add_half_vhdl* are written they can be used to create *Add_rca_4_vhdl* and *Add_rca_8_vhdl.*

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

-- Model for 2-input AND component
entity and2_gate is
  port (A, B: in Std_Logic; C: out Std_Logic);
end and2_gate;

architecture Boolean_Equation of and2_gate is
begin
  C <= A and B;       -- Logic operator
end Boolean_Equation;

-- Model for 2-input OR component
entity or2_gate is
  port (A, B: in Std_Logic; C: out Std_Logic);
end or2_gate;

architecture Boolean_Equation of or2_gate is
begin
  C <= A or B;   -- Logic operator
end Boolean_Equation;

-- Model for exclusive-or component
entity xor_2_gate is
  port (A, B: in Std_Logic; C: out Std_Logic);
end xor_2_gate;

architecture Boolean_Equation of xor_2_gate is
begin
  C <= A xor B;
end Boolean_Equation;
```

The components *and2_gate* and *xor2_gate* are then used in models for *Add_half_vhdl* and *Add_full_vhdl.*

```
entity Add_half_vhdl is
  port (a, b: in std_logic; c_out, sum: out std_logic);
end Add_half
```

```
architecture Structure of Add_half is
  component and2_gate          -- Identify component being used
  port (a, b: in std_logic; c: out std_logic);   -- Identify port of the component
end component;

component xor2_gate   -- Component declaration
  port (a, b: in std_logic; c: out std_logic);
and component;

begin      -- Instantiate components and connect ports
  G1: xor2_gate      port map (a, b, sum);
  G2: and2_gate      port map (a, b, c_out,);
end Structure;

entity Add_full_vhdl is
  port (a, b, c_in: in std_logic; c_out, sum: out std_logic);
end Add_full_vhdl

architecture Structure of Add_full_vhdl is
  component or2_gate
    port (a, b: in std_logic; c: out std_logic);
  end component;
  component Add_half_vhdl
     port (a, b: in std_logic; c_out, sum: out std_logic);
  end component;
  signal w1, w2, w3: std_logic;
  begin
    M0: Add_half_vhdl port map (b, c_in, c_out, sum);
    M1 Add_half port map (a, b, w1, w2);
    G1 or2_gate port map (w1, w3, c_out);
end Structure;

entity Add_rca_4_vhdl is
  port (A, B: in bit_vector (3 downto 0); c_in: in Std_Logic;
               c_out: out Std_Logic; sum: out bit_vector (3 downto 0);
end Add_rca_4_vhdl;

architecture Structure of Add_rca_4_vhdl is
    component Add_full_rca_vhdl
      port (a, b: in Std_Logic_Vector (3 downto 0); c_in: in Std_Logic; c_out: out Std_
      Logic; sum: out Std_Logic_Vector (3 downto 0);
    end component;
    signal c_in1, c_in2, c_in3;
begin
    M0: Add_full_vhdl port map (a(0), b(0), c_in, c_in1, sum(0));
    M1: Add_full_vhdl port map (a(1), b(1), c_in1, c_in2, sum(1));
    M2: Add_full_vhdl port map (a(2), b(2), c_in2, c_in3, sum(2));
    M3: Add_full_vhdl port map (a(3), b(3), c_in3, c_out, sum(3));
end Structure;
```

```
entity Add_rca_8_vhdl is
port (a, b: in Std_Logic_Vector (7 downto 0); c_in: in Std_Logic;
        c_out: out Std_Logic, sum: Std_Logic_Vector (7 downto 0));
end Add_rca_8_vhdl;

architecture Structure of Add_rca_8_vhdl is
 component Add_rca_4_vhdl;
  port (a, b: in Std_Logic_Vector (3 downto 0); c_in: in Std_Logic;
          c_out: out Std_Logic; sum: Std_Logic_Vector (3 downto 0));
 end component;
 signal c_in4        -- Connects 4-bit adders
 M0 Add_rca_4_vhdl port map (a(3 downto 0), b(3 downto 0), c_in, c_in4,
 sum(3 downto 0 ));
 M1 Add_rca_4_vhdl port map (a(7 downto 4), b(7 downto 4), c_in4, c_out,
 sum(7 downto 4 ));
end Structure
```

The code for *Add_rca_8* illustrates how gate-level design in VHDL becomes bulky with declarations of components. Hierarchical design can be made simple if component declarations exploit dataflow models at the lower levels of the hierarchy. For example, a half adder can be designed and used as a component in the design of a full-adder.

```
entity half_adder_vhdl is
  port (S, C: out Std_Logic; x, y: in Std_Logic);
end half_adder_vhdl;

architecture Dataflow of half_adder_vhdl is
  S <= x xor y;
  C <= x and y;
end Dataflow;

entity full_adder_vhdl is
  port (S, C: out Std_Logic; x, y, z: in Std_Logic);
end half_adder_vhdl

architecture Structural of full_adder_vhdl is
  signal S1, C1, C2: Std_Logic;
  component half_adder_vhdl port (S, C: out Std_Logic; x, y, z: in Std_Logic);
begin
  HA1: half_adder_vhdl port map (S => S1, C => C1, x => x, y => y);
  HA2: half_adder_vhdl port map (S => S, C => C2, x => S1, y => z);
  C <= C2 or C1;
end Structural;

entity ripple_carry_4_bit_adder_vhdl is
  port (Sum: out Std_Logic_Vector (3 downto 0); C4: out Std_Logic; A, B: in
    Std_Logic_Vector (3 downto 0); C0: in Std_Logic);
end ripple_carry_4_bit_adder_vhdl;
```

```
architecture Structural of ripple_carry_4_bit_adder_vhdl is
  component full_adder_vhdl port Sum: out Std_Logic_Vector (3 downto 0); C4: out
    Std_Logic; A, B: in Std_Logic_Vector (3 downto 0); C0: in Std_Logic);
    signal C1, C2, C3: Std_Logic;

begin
 FA0: full_adder_vhdl port map (S => Sum(0), C => C1, x => A(0), y = B(0), z => C0);
 FA1: full_adder_vhdl port map (S => Sum(1), C => C2, x => A(1), y = B(1), z => C1);
 FA2: full_adder_vhdl port map (S => Sum(2), C => C3, x => A(2), y = B(2), z => C2);
 FA3: full_adder_vhdl port map (S => Sum(3), C => C4, x => A(3), y = B(3), z => C3);
end ripple_carry_4_bit_adder_vhdl;
```

HDL Models of Three-State Gates

A three-state gate has a data signal input, a data signal output, and a control input. The control input determines whether the gate is in its normal operating state or in its high-impedance state.

Verilog (Predefined Buffers and Inverters) Verilog has four types of predefined three-state gates, as shown in Fig. 4.35. The **bufif1** gate behaves like a normal buffer if *control* = 1. The output goes to a high-impedance state **z** when *control* = 0. The **bufif0** gate behaves in a similar fashion, except that the high-impedance state occurs when *control* = 1. The two **notif** gates operate in a similar manner, but the output is the complement of the input when the gate is not in a high-impedance state. The gates are instantiated with the statement

gate name (*output*, *input*, *control*);

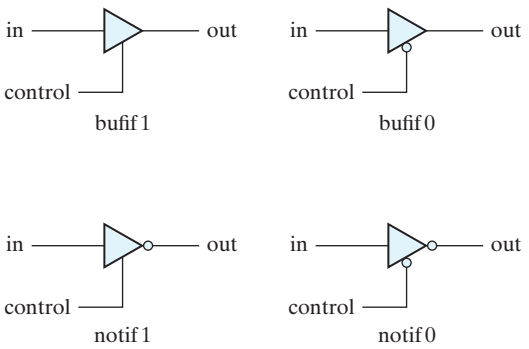

FIGURE 4.35
Three-state gates

The gate name can be that of any 1 of the 4 three-state gates. In simulation, the output can result in 0, 1, **x**, or **z**. Two examples of gate instantiation are

```
bufif1  (OUT, A, control);
notif0  (Y, B, enable);
```

In the first example, *OUT* has the same value as *A* when *control* = 1. *OUT* goes to **z** when *control* = 0. In the second example, output Y = **z** when *enable* = 1 and output $Y = B'$ when *enable* = 0.

The outputs of three-state gates can be connected together to form a common output line. To explicitly identify such a connection, Verilog uses the net-type keyword **tri** (for tristate) to indicate that the identifier has multiple drivers. As an example, consider the two-to-one-line multiplexer with three-state gates shown in Fig. 4.36.

The description must use a **tri** data type for the output, because *m_out* has two drivers:

```
// Mux with three-state output
module mux_tri (m_out, A, B, select);
  output  m_out;
  input   A, B, select;
  tri     m_out;

  bufif1 (m_out, A, select);
  bufif0 (m_out, B, select);
endmodule
```

The outputs of the three-state buffers are identical (*m_out*). In order to show that they have a common connection, it is necessary to declare *m_out* with the keyword **tri**.

Keywords **wire** and **tri** are examples of a set of data types called *nets*, which represent connections between hardware elements. In simulation, their value is determined by a continuous assignment statement or by the device whose output they represent. The word *net* is not a keyword, but represents a class of data types, such as **wire, wor, wand, tri, triand, trior, supply1,** and **supply0**. The **wire** declaration is used most frequently. In fact, if an identifier is used, but not declared, the language specifies that it will be interpreted (by default), for example, as a **wire**. The net **wor** models the hardware implementation of the wired-OR configuration (emitter-coupled logic). The **wand** models the

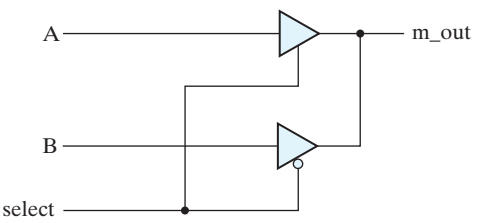

FIGURE 4.36
Two-to-one-line multiplexer with three-state buffers