Chapter 1
Basic Building Blocks

Digital circuits are no longer defined by logical schemes but by Hardware Description Language programs [1]. The translation of this kind of definition to an actual implementation is realized by Electronic Automation Design tools (Chap. 5). All along this book the chosen language is VHDL. In this chapter the most useful constructions are presented. For all of the proposed examples, the complete source code is available at the Authors' web page.

1.1 Combinational Components

1.1.1 Boolean Equations

Among the predefined operations of any Hardware Description Language are the basic Boolean operations. Boolean functions can easily be defined. Obviously, all the classical logic gates can be defined. Some of them are considered in the following example.

Example 1.1
The following VHDL instructions define the logic gates NOT, AND2, OR2, NAND2, NOR2, XOR2, XNOR2, NAND3, NOR3, XOR3, XNOR3 (Fig. 1.1).

```
b <= NOT(a);
c <= a AND b;
c <= a OR b;
c <= a NAND b;
```

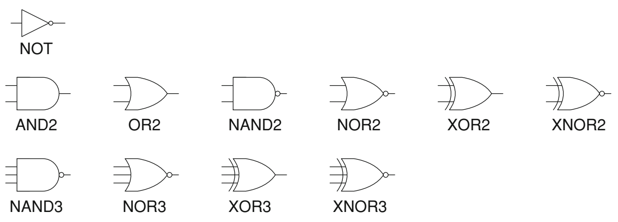


Fig. 1.1 Logic gates

```
c <= a NOR b;
c <= a XOR b;
c <= a XNOR b;
d <= NOT(a AND b AND c);
d <= NOT(a OR b OR c);
d <= a XOR b XOR c;
d <= NOT(a XOR b XOR c);
```

The same basic Boolean operations can be used to define sets of logic gates working in parallel. As an example, assume that signals $a = (a_{n-1}, a_{n-2},\ldots, a_0)$ and $b = (b_{n-1}, b_{n-2},\ldots, b_0)$ are n-bit vectors. The following assignation defines a set of n AND2 gates that compute $c = (a_{n-1}\cdot b_{n-1}, a_{n-2}\cdot b_{n-2},\ldots, a_0\cdot b_0)$:

```
c <= a AND b;
```

More complex Boolean functions can also be defined. It is worthwhile to indicate that within most Field Programmable Gate Arrays (FPGA) the basic combinational components are not 2-input logic gates but Look Up Tables (LUT) allowing the implementation of any Boolean function of a few numbers (4, 5, 6) of variables (Chap. 5). Hence, it makes sense to consider the possibility of defining small combinational components by the set of Boolean functions they implement.

Example 1.2
The following VHDL instruction defines an ANDORI gate (a 4-input 1-output component implementing the complement of $a\cdot b$ v $c\cdot d$, Fig. 1.2)

```
e <= NOT((a AND b) OR (c AND d));
```

and the two following instructions define a 1-bit full adder (a 3-input 2-output component implementing $a+b+c_{in}$ mod 2 and $a\cdot b$ v $a\cdot c_{in}$ v $b\cdot c_{in}$, Fig. 1.3).

```
s <= a XOR b XOR cin;
cout <= (a AND b) OR (a AND cin) OR (b AND cin);
```

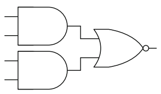


Fig. 1.2 ANDORI gate

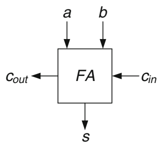


Fig. 1.3 One-bit full adder

1.1.2 Tables

Combinational components can also be defined by their truth tables, without the necessity to translate their definition to Boolean equations. As a first example, consider a 1-digit to 7-segment decoder, that is a 4-input 7-output combinational circuit.

Example 1.3
The behavior of a 4-to-7 decoder (Fig. 1.4) can be described by a conditional assignment instruction

```
WITH digit SELECT segments <=
  "1110111" WHEN "0000",
  "0010010" WHEN "0001",
  "1011101" WHEN "0010",
  "1011011" WHEN "0011",
  "0111010" WHEN "0100",
  "1101011" WHEN "0101",
  "1101111" WHEN "0110",
  "1010010" WHEN "0111",
  "1111111" WHEN "1000",
  "1111011" WHEN "1001",
  "1111110" WHEN "1010",
  "0101111" WHEN "1011",
  "1100101" WHEN "1100",
  "0011111" WHEN "1101",
  "1101101" WHEN "1110",
  "1101100" WHEN OTHERS;
```

The last choice of a WITH… SELECT construction must be WHEN OTHERS in order to avoid the inference of an additional latch, and it is the same for other multiple choice instructions such as CASE.

As mentioned above, small Look Up Tables are basic components of most FPGA families. The following example is a generic 4-input 1-output LUT.

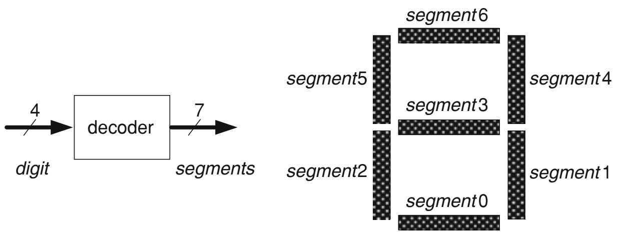
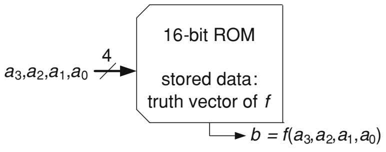


Fig. 1.4 Four-digit to seven-segment decoder

Fig. 1.5 Four-input Look Up Table



Example 1.4
The following entity defines a 4-input Boolean function whose truth vector is stored within a generic parameter (Fig. 1.5). Library declarations are omitted.

```
ENTITY lut4 IS
  GENERIC(truth_vector: STD_LOGIC_VECTOR(0 to 15));
PORT (
  a: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
  b: OUT STD_LOGIC
);
END lut4;

ARCHITECTURE behavior OF lut4 IS
BEGIN
  PROCESS(a)
    VARIABLE c: NATURAL;
  BEGIN
    c := CONV_INTEGER(a);
    b <= truth_vector(c);
  END PROCESS;
END behavior;
```

Then the following component instantiation defines a 4-input XOR function.

```
xor4: lut4
GENERIC MAP(truth_vector => "0110100110010110")
PORT MAP(a => a, b => b);
```

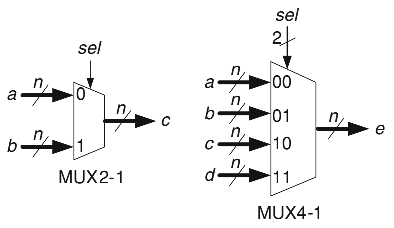


Fig. 1.6 Multiplexers

Comment 1.1
The VHDL model of the preceding Example 1.4 can be used for simulation purposes, independently of the chosen FPGA vendor. Nevertheless, in order to implement an actual circuit, the corresponding vendor's primitive component should be used instead (Chap. 5).

1.1.3 Controllable Connections

Multiplexers are the basic components for implementing controllable connections. Conditional assignments are used for defining them. Some of them are considered in the following example.

Example 1.5
The following conditional assignments define a 2-to-1 and a 4-to-1 multiplexer (Fig. 1.6). The signal types must be compatible with the conditional assignments: *a*, *b*, *c*, *d* and *e* are assumed to be *n*-bit vectors for some constant value *n*.

```
WITH sel SELECT c <= a WHEN '0', b WHEN OTHERS;

WITH sel SELECT e <=
  a WHEN "00",
  b WHEN "01",
  c WHEN "10",
  d WHEN OTHERS;
```

Demultiplexers, address decoders and tri-state buffers are other components frequently used for implementing controllable connections such as buses.

Example 1.6
The following equations define a 1-bit 1-to-2 demultiplexer (Fig. 1.7a)

```
b <= NOT(sel) AND a;
c <= sel AND a;
```

and the following conditional assignments a 1-to-4 demultiplexer (Fig. 1.7b)

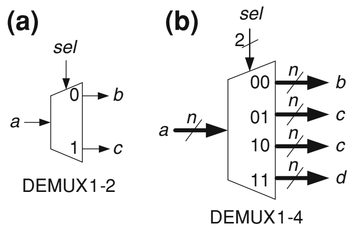


Fig. 1.7 Demultiplexers

```
WITH sel SELECT b <=
  a WHEN "00", (OTHERS => '0') WHEN OTHERS;
WITH sel SELECT c <=
  a WHEN "01", (OTHERS => '0') WHEN OTHERS;
WITH sel SELECT d <=
  a WHEN "10", (OTHERS => '0') WHEN OTHERS;
WITH sel SELECT e <=
  a WHEN "11", (OTHERS => '0') WHEN OTHERS;
```

In the second case the signal types must be compatible with the conditional assignments: *a*, *b*, *c*, *d* and *e* are assumed to be *n*-bit vectors for some constant value *n*.

An address decoder is a particular case of 1-bit demultiplexer whose input is 1.

Example 1.7
The following equations define a 3-input address decoder (Fig. 1.8).

```
rows(0) <=
  NOT(address(2)) AND NOT(address(1)) AND NOT(address(0));
rows(1) <=
  NOT(address(2)) AND NOT(address(1)) AND address(0);
rows(2) <=
  NOT(address(2)) AND address(1)      AND NOT(address(0));
rows(3) <=
  NOT(address(2)) AND address(1)      AND address(0);
rows(4) <=
  address(2)      AND NOT(address(1)) AND NOT(address(0));
rows(5) <=
  address(2)      AND NOT(address(1)) AND address(0);
rows(6) <=
  address(2)      AND address(1)      AND NOT(address(0));
rows(7) <=
  address(2)      AND address(1)      AND address(0);
```

Three-state buffers implement controllable switches.

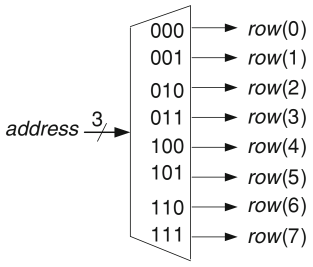


Fig. 1.8 Address decoder

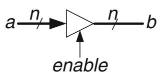


Fig. 1.9 Three-state buffer

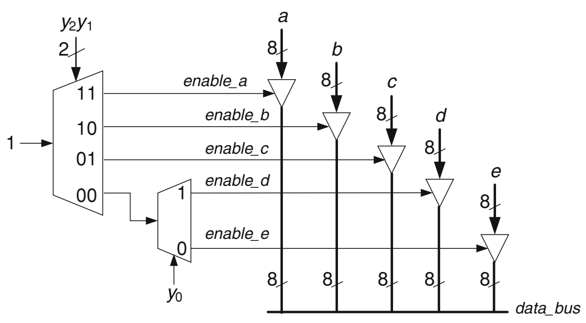


Fig. 1.10 Example of a data bus

Example 1.8
The following conditional assignment defines a tri-state buffer (Fig. 1.9): when the enable signal is equal to 1, output *b* is equal to input *a*, and when the enable signal is equal to 0, output *b* is disconnected (high impedance state). The signal types must be compatible with the conditional assignments: *a* and *b* are assumed to be *n*-bit vectors for some constant value *n*.

```
WITH enable SELECT b <=
  a WHEN '1',
  (OTHERS =>'Z') WHEN OTHERS;
```

An example of the use of demultiplexers and tri-state buffers, namely a data bus, is shown in Fig. 1.10.

Example 1.9
The circuit of Fig. 1.10, made up of demultiplexers and three-state buffers, can be described by Boolean equations (the multiplexers) and conditional assignments (the three-sate buffers).

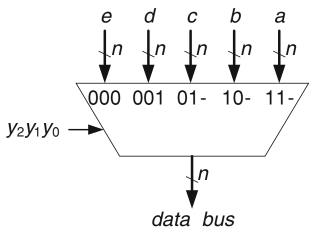


Fig. 1.11 Data bus, second version

```
enable_a <= y(2)      AND y(1);
enable_b <= y(2)      AND NOT(y(1));
enable_c <= NOT(y(2)) AND y(1);
enable_d <= NOT(y(2)) AND NOT(y(1))   AND y(0);
enable_e <= NOT(y(2)) AND NOT(y(1))   AND NOT(y(0));
WITH enable_a SELECT data_bus <=
  a WHEN '1', (OTHERS =>'Z') WHEN OTHERS;
WITH enable_b SELECT data_bus <=
  b WHEN '1', (OTHERS =>'Z') WHEN OTHERS;
WITH enable_c SELECT data_bus <=
  c WHEN '1', (OTHERS =>'Z') WHEN OTHERS;
WITH enable_d SELECT data_bus <=
  d WHEN '1', (OTHERS =>'Z') WHEN OTHERS;
WITH enable_e SELECT data_bus <=
  e WHEN '1', (OTHERS =>'Z') WHEN OTHERS;
```

Nevertheless, the same functionality can be implemented by an 8-bit 5-to-1 multiplexer (Fig. 1.11).

```
WITH y SELECT data_bus <=
  a WHEN "111" | "110",
  b WHEN "101" | "100",
  c WHEN "011" | "010",
  d WHEN "001",
  e WHEN OTHERS;
```

Generally, this second implementation is considered safer than the first one. In fact, tri-state buffers should not be used within the circuit core. They should only be used within I/O-port components (Sect. 1.4).

1.1.4 Arithmetic Circuits

Among the predefined operations of any Hardware Description Language there are also the basic arithmetic operations. The translation of this kind of description to

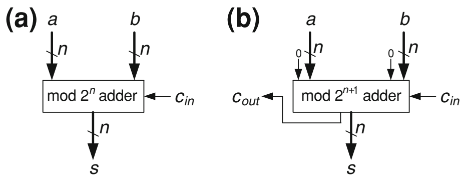


Fig. 1.12 *n*-bit adders

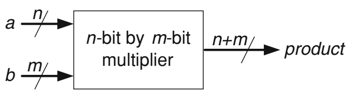


Fig. 1.13 *n*-bit by *m*-bit multiplier

actual implementations, using special purpose FPGA resources (carry logic, multiplier blocks), is realized by Electronic Automation Design tools (Chap. 5).

Example 1.10
The following arithmetic equation defines an adder mod 2^n, being *a*, *b* and *s* *n*-bit vectors and c_{IN} an input carry (Fig. 1.12a).

```
s <= a + b + c_in;
```

By adding a most significant bit 0 to one (or both) of the *n*-bit operands *a* and *b*, an *n*-bit adder with output carry c_{OUT} can be defined (Fig. 1.12b). The internal signal *sum* is an (*n*+1)-bit vector.

```
sum <= ('0'&a) + b + c_in;
s <= sum(n-1 DOWNTO 0);
c_out <= sum(n);
```

The following arithmetic equation defines an *n*-bit by *m*-bit multiplier where *a* is an *n*-bit vector, *b* an *m*-bit vector and *product* an (*n*+*m*)-bit vector (Fig. 1.13).

```
product <= a * b;
```

Comment 1.2
In most VHDL models available at the Authors' web page, the type *unsigned* has been used, so that bit-vectors can be treated as natural numbers. In some cases, it could be better to use the *signed* type, for example when bit-vectors are interpreted as 2's complement integers, and when magnitude comparisons or sign-bit extensions are performed.

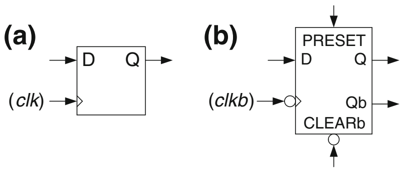


Fig. 1.14 D flip-flops

1.2 Sequential Components

1.2.1 Flip-Flops

The basic sequential component is the D-flip-flop. In fact, several types of D-flip-flop can be considered: positive edge triggered, negative edge triggered, with asynchronous input(s) and with complemented output.

Example 1.11
The following component is a D-flip-flop triggered by the positive edge of *clk* (Fig. 1.14a),

```
PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN q <= d; END IF;
END PROCESS;
```

while the following, which is controlled by the negative edge of *clkb*, has two asynchronous inputs *clearb* (active at low level) and *preset* (active at high level), having *clearb* priority, and has two complementary outputs *q* and *qb* (Fig. 1.14b).

```
PROCESS(clkb, preset, clearb)
BEGIN
  IF clearb = '0' THEN q <= '0'; qb <= '1';
  ELSIF preset = '1' THEN q <= '1'; qb <= '0';
  ELSIF clkb'EVENT AND clkb = '0' THEN q <= d; qb <= not(d);
  END IF;
END PROCESS;
```

Comment 1.3
The use of level-controlled, instead of edge-controlled, components is not recommendable. Nevertheless, if it were necessary, a D-latch could be modeled as follows:

```
IF en = '1' THEN q <= d; END IF;
```

where *en* is the enable signal and *d* the data input.

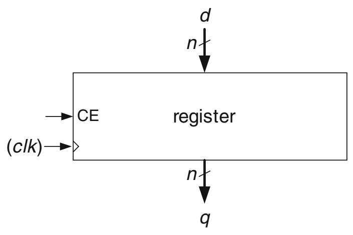


Fig. 1.15 Parallel register

1.2.2 Registers

Registers are sets of D-flip-flops controlled by the same synchronization and control signals, and connected according to some regular scheme (parallel, left or right shift, bidirectional shift).

Example 1.12
The following component is a parallel register with *ce* (*clock enable*) input, triggered by the positive edge of *clk* (Fig. 1.15).

```
PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF ce = '1' THEN q <= d; END IF;
  END IF;
END PROCESS;
```

As a second example (Fig. 1.16), the next component is a right shift register with parallel input (controlled by *load*) and serial input (controlled by *shift*).

```
PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF load = '1' THEN q <= parallel_in;
    ELSIF shift = '1' THEN
      q <= serial_in & q(n-1 downto 1);
    END IF;
  END IF;
END PROCESS;
```

1.2.3 Counters

A combination of registers and arithmetic operations permits the definition of counters.

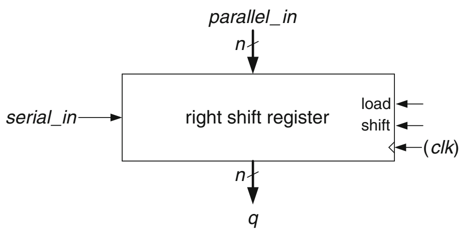


Fig. 1.16 Right shift register

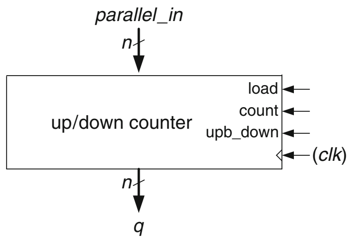


Fig. 1.17 Up/down counter

Example 1.13

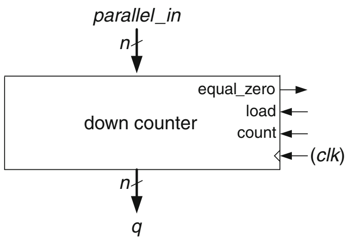
This defines an up/down counter (Fig. 1.17) with control signals *load* (input *parallel_in*), *count* (update the state of the counter) and *upb_down* (0: count up, 1: count down).

```
PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF load = '1' THEN q <= parallel_in;
    ELSIF count = '1' AND upb_down = '0' THEN
      q <= q + 1;
    ELSIF count = '1' AND upb_down = '1' THEN
      q <= q - 1;
    END IF;
  END IF;
END PROCESS;
```

The following component is a down counter (Fig. 1.18) with control signals *load* (input *parallel_in*) and *count* (update the state of the counter). An additional binary output *equal_zero* is raised when the state of the counter is *zero* (all 0's vector).

```
PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF load = '1' THEN q <= parallel_in;
    ELSIF count = '1' THEN q <= q - 1;
    END IF;
  END IF;
END PROCESS;
equal_zero <= '1' WHEN q = zero ELSE '0';
```

Fig. 1.18 Down counter



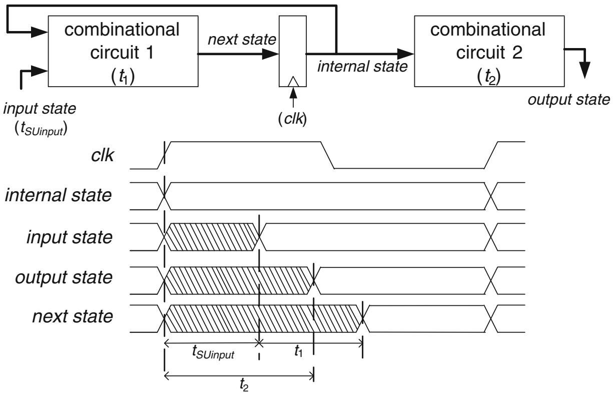


Fig. 1.19 Moore machine

1.2.4 Finite State Machines

Hardware Description Languages allow us to define finite state machines at an input/output behavioral level. The translation to an actual implementation including registers and combinational circuits—a classical problem of traditional switching theory—is realized by Electronic Automation Design tools (Chap. 5).

In a Moore machine, the output state only depends on the current internal state (Fig. 1.19) while in a Mealy machine the output state depends on both the input state and the current internal state (Fig. 1.20). Let $t_{SUinput}$ be the maximum set up time of the input state with respect to the positive clock edge, t_1 the maximum delay of the combinational block that computes the next internal state, and t_2 the maximum delay of the combinational block that computes the output state. Then, in the case of a Moore machine, the following conditions must hold

$$t_{SUinput} + t_1 < T_{CLK} \text{ and } t_2 < T_{CLK}, \tag{1.1}$$

and in the case of a Mealy machine

$$t_{SUinput} + t_1 < T_{CLK} \text{ and } t_{SUinput} + t_2 < T_{CLK}. \tag{1.2}$$

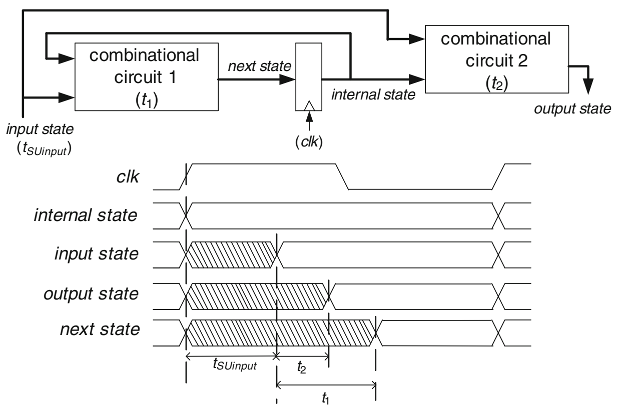
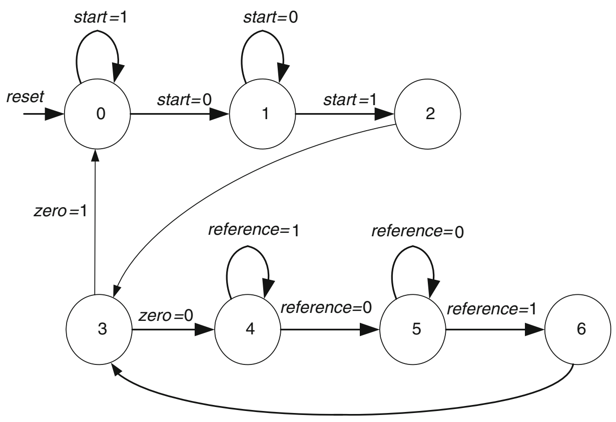


Fig. 1.20 Mealy machine

The set up and hold times of the register (Chap. 6) have not been taken into account.

Example 1.14
A Moore machine is shown in Fig. 1.21. It is the control unit of a programmable timer (Exercise 2.6.2). It has seven internal states, three binary input signals *start*, *zero* and *reference*, and two output signals *operation* (2 bits) and *done*. It can be described by the following processes.

```
next_state: PROCESS(reset, clk)
BEGIN
  IF reset = '1' THEN current_state <= 0;
  ELSIF clk'EVENT AND clk = '1' THEN
    CASE current_state IS
      WHEN 0 => IF start = '0'
                THEN current_state <= 1;
                END IF;
      WHEN 1 => IF start = '1'
                THEN current_state <= 2;
                END IF;
      WHEN 2 => current_state <= 3;
      WHEN 3 => IF zero = '1'
                THEN current_state <= 0;
                ELSE current_state <= 4;
                END IF;
      WHEN 4 => IF reference = '0'
                THEN current_state <= 5;
                END IF;
      WHEN 5 => IF reference = '1'
                THEN current_state <= 6;
```



state	*operation*	*done*
0	00	1
1	00	1
2	11	0
3	00	0
4	00	0
5	00	0
6	01	0

Fig. 1.21 An example of a Moore machine

```
                END IF;
      WHEN 6 => current_state <= 3;
    END CASE;
  END IF;
END PROCESS next_state;

output_state: PROCESS(current_state)
BEGIN
  CASE current_state IS
    WHEN 0 TO 1 => operation <= "00"; done <= '1';
    WHEN 2 =>      operation <= "11"; done <= '0';
    WHEN 3 TO 5 => operation <= "00"; done <= '0';
    WHEN 6 =>      operation <= "01"; done <= '0';
  END CASE;
END PROCESS output_state;
```

Example 1.15
Consider the Mealy machine of Table 1.1. It has four internal states, two binary inputs x_1 and x_0, and one binary output z. Assume that x_0 and x_1 are

Table 1.1 A Mealy machine: next state/z

	$X_1\ x_0$: 00	01	10	11
A	A/0	B/0	A/1	D/1
B	B/1	B/0	A/1	C/0
C	B/1	C/1	D/0	C/0
D	A/0	C/1	D/0	D/1

periodic, but out of phase, signals. Then the machine detects if x_0 changes before x_1 or if x_1 changes before x_0. In the first case the sequence of internal states is A B C D A B… and z = 0. In the second case the sequence is D C B A D C… and z = 1.

It can be described by the following processes.

```
input_state <= x1&x0;
next_state: PROCESS (reset, clk)
BEGIN
  IF reset = '1' THEN current_state <= A;
  ELSIF clk'EVENT AND clk = '1' THEN
    CASE current_state IS
      WHEN A => IF input_state = "01" THEN
                  current_state <= B;
                ELSIF input_state = "11" THEN
                  current_state  <= D;
                END IF;
      WHEN B => IF input_state = "10" THEN
                  current_state <= A;
                ELSIF input_state = "11" THEN
                  current_state  <= C;
                END IF;

      WHEN C => IF input_state = "00" THEN
                  current_state <= B;
                ELSIF input_state = "10" THEN
                  current_state  <= D;
                END IF;
      WHEN D => IF input_state = "00" THEN
                  current_state <= A;
                ELSIF input_state = "01" THEN
                  current_state  <= C;
                END IF;
    END CASE;
  END IF;
END PROCESS;
```

```
output_state: PROCESS (current_state, input_state)
BEGIN
  CASE current_state IS
    WHEN A => z <= x1;
    WHEN B => z <= not(x0);
    WHEN C => z <= not(x1);
    WHEN D => z <= x0;
  END CASE;
END PROCESS output_state;
```

1.3 Memory Blocks

With regards to memory blocks, a previous comment similar to Comment 1.1 must be outlined: VHDL models can be generated for simulation purposes; nevertheless, in order to implement an actual circuit, the corresponding vendor's primitive component should be used instead (Chap. 5).

Example 1.16
The following entity defines a Read Only Memory storing 2^n m-bit words. The stored data is defined by a generic parameter (Fig. 1.22). Library declarations are omitted.

```
ENTITY rom IS
  GENERIC(n, m: NATURAL; stored_data: STD_LOGIC_VECTOR);
PORT (
  address: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
  word: OUT STD_LOGIC_VECTOR(m-1 DOWNTO 0)
);
END rom;

ARCHITECTURE behavior OF rom IS
BEGIN
  PROCESS(address)
    VARIABLE c: NATURAL;
  BEGIN
    c := CONV_INTEGER(address)* m;
    word <= stored_data(c to c + m -1);
  END PROCESS;
END behavior;
```

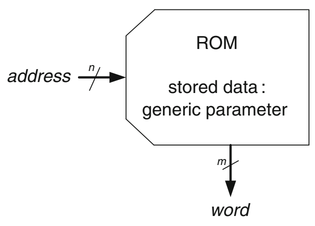


Fig. 1.22 Read only memory

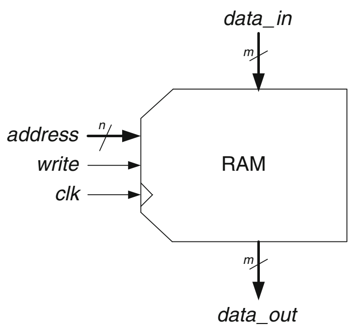


Fig. 1.23 Random access memory

Then the following component instantiation defines a ROM storing 16 4-bit words, namely

0100,1010,1011,1100,0111,1000,1001,0010,0001,1101,1101,1110,1111,0101,0100,0001

(from address 0000 to address 1111).

```
DUT: rom
GENERIC MAP(n => 4, m => 4, stored_data =>
X"4abc78921ddef541")
PORT MAP(address => address, word => word);
```

Example 1.17
The following entity defines a synchronous Random Access Memory storing 2^n m-bit words (Fig. 1.23). A *write* input enables the writing operation. Functionally, it is equivalent to a Register File made up of 2^n m-bit registers whose *clock enable* inputs are connected to *write*, plus an address decoder. Library declarations are omitted.

```
ENTITY ram IS
  GENERIC(n, m: NATURAL);
PORT (
  address: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
  data_in: IN STD_LOGIC_VECTOR(m-1 DOWNTO 0);
  clk, write: IN STD_LOGIC;
  data_out: OUT STD_LOGIC_VECTOR(m-1 DOWNTO 0)
);
END ram;

ARCHITECTURE behavior OF ram IS
  TYPE memory IS ARRAY (0 TO 2**n-1) of
    STD_LOGIC_VECTOR(m-1 DOWNTO 0);
  SIGNAL stored_data: memory;
BEGIN
  PROCESS(clk)
  BEGIN
    IF clk'EVENT AND CLK = '1' THEN
      IF write = '1' THEN

      stored_data(CONV_INTEGER(address)) <= data_in;
    END IF;
  END IF;
END PROCESS;
  data_out <= stored_data(CONV_INTEGER(address));
END behavior;
```

The following component instantiation defines a synchronous RAM storing 16 4-bit words.

```
DUT: ram
GENERIC MAP(n => 4, m => 4)
PORT MAP(address => address, data_in => data_in,
  clk => clk, write => write, data_out => data_out);
```

1.4 IO-Port Components

Once again, a previous comment similar to Comment 1.1 must be outlined: VHDL models can be generated for simulating input and output amplifiers; nevertheless, in order to implement an actual circuit, the corresponding vendor's I/O component should be used instead (Chap. 5).

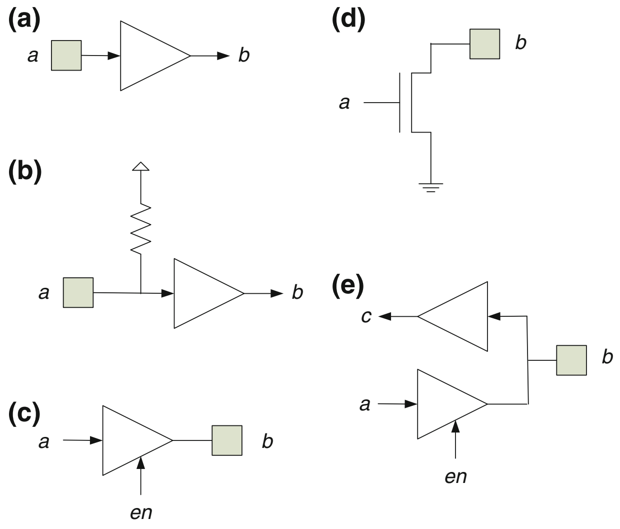


Fig. 1.24 I/O ports

In order to generate VHDL models of I/O amplifiers, it is convenient to understand the meaning of the STD_LOGIC type elements, that is

‘X’: forcing unknown, ‘0’ : forcing 0, ‘1’ : forcing 1,
‘W’: weak unknown, ‘L’ : weak 0, ‘H’ : weak 1,
‘Z’: high impedance

(“uninitialized” and “don’t care” states have no sense in the case of I/O amplifiers).

An amplifier generates low-impedance (forcing) signals when enabled and high impedance signals when disabled. Thus the possible outputs generated by an amplifier are ‘X’, ‘0’, ‘1’ and ‘Z’. In the following example several types of input, output and bidirectional amplifiers are defined.

Example 1.18

The following conditional assignment defines an input buffer (Fig. 1.24a):

```
WITH a SELECT b <=
'0' WHEN '0' | 'L',
'1' WHEN '1' | 'H',
'X' WHEN OTHERS;
```

An input buffer with a pull-up resistor can be defined as follows (Fig. 1.24b).

```
WITH a SELECT b <=
'0' WHEN '0' | 'L',
'1' WHEN '1' | 'H' | 'Z',
'X' WHEN OTHERS;
```

The definition of a tri-state output buffer (Fig. 1.24c) is the same as in example 1.8, that is

```
WITH en SELECT b <= a WHEN '1', 'Z' WHEN OTHERS;
```

An open-drain output can be defined as follows (Fig. 1.24d).

```
WITH a SELECT b <= '0' WHEN '1' | 'H', 'Z' WHEN OTHERS;
```

As an example of hierarchical description, a bidirectional I/O buffer can be defined by instantiating an input buffer and a tri-state output buffer (Fig. 1.24e).

```
an_input_buffer: input_buffer
PORT MAP(a => b, b => c);
an_output_buffer: tri_state_output
PORT MAP(a => a, en => en, b => b) ;
```

1.5 VHDL Models

The following complete VHDL models are available at the Authors' web page www.arithmetic-circuits.org:

logic_gates.vhd (Sects. 1.1.1, 1.1.2 and 1.1.3),
arithmetic_blocks.vhd (Sects. 1.1.4, 1.2.1 and 1.2.2),
sequential_components.vhd (Sects. 1.2.1, 1.2.2 and 1.2.3),
fnite_state_machines.vhd (Sect. 1.2.4),
memories.vhd (Sect. 1.3),
input_output.vhd (Sect. 1.4).

1.6 Exercises

1. Generate the VHDL model of a circuit that computes $y = a{\cdot}x$ where a is a bit, and x and y are n-bit vectors, so that $y = (a{\cdot}x_{n-1}, a{\cdot}x_{n-2},\ldots, a{\cdot}x_0)$.
2. Generate several models of a 1-bit full subtractor (Boolean equations, table, LUT instantiation).
3. Generate a generic model of an n-bit 8-to-1 multiplexer.

4. Generate a generic model of an *n*-input address decoder.
5. Design an *n*-bit magnitude comparator: given two *n*-bit naturals *a* and *b*, it generates a 1-bit output *gt* equal to 1 if $a \geq b$ and equal to 0 if $a < b$.
6. Design a 60-state up counter with *reset* and *count* control inputs.
7. Design a finite state machine with two binary inputs *x* and *y* and a binary output *z* defined as follows: if the input sequence is (x, t) = 00 01 11 10 00 01 11… then $z = 0$, and if the input sequence is (x, y) = 00 10 11 01 00 10 11… then $z = 1$.

Reference

1. Hamblen JO, Hall TS, Furman MD (2008) Rapid prototyping of digital systems. Springer, New York