Chapter 2
VHDL

2.1 A Brief History of VHDL

VHDL is the acronym of Very High-Speed Integrated Circuit Hardware Description Language, and it was developed around 1980 at the request of the U.S. Department of Defense. At the beginning, the main goal of VHDL was the electric circuit simulation; however, tools for synthesis and implementation in hardware based on VHDL behavior or structure description files were developed later. With the increasing use of VHDL, the need for standardized was generated. In 1986, the Institute of Electrical and Electronics Engineers (IEEE) standardized the first hardware description language, VHDL, through the 1076 and 1164 standards. VHDL is technology/vendor independent, then VHDL codes are portable and reusable.

2.2 VHDL Structure

VHDL is a structured language. Each description of a file has three main blocks:

- Libraries
- Entity
- Architecture

Listing 2.1 shows the main standard libraries for logic and arithmetic descriptions. “Unsigned” and “arith” libraries were developed by Synopsys Inc., they may be under ©.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
use IEEE.numeric_std.all;
```

Listing 2.1 Libraries
Entity can be seen as a black box as shown by Fig. 2.1, where the inputs and outputs must be defined here (see listing 2.2). For example, Fig. 2.1 has four ports: signal A is type in, signal B is type out, signal C is type in/out, and signal D is type buffer.

- **in.** Input signal to the entity. Unidirectional
- **out.** Output signal to the entity. Unidirectional
- **in/out.** Input–output signal to the entity. Bidirectional
- **buffer.** Allows internal feedbacks inside the entity. The declared port behavior is as an output.

The data type for each port must be defined. Some of the most used in VHDL are:

- **Bit.** The only values that port allows are 0 or 1.
- **Boolean.** Take the values true or false.
- **Integer.** This type cover all integer values.
- **std_logic.** This data type allows nine values
  - U Unitialized
  - X Unknown
  - 0 Low
  - 1 High
  - Z High impedance
  - W Weak unknown
  - L Weak low
  - H Weak high
  - '*' Don’t care
- **bit_vector.** A vector of bits.
- **std_logic_vector.** A vector of bits of type std_logic.

```
1 entity name_of_entity is
2 port(
3 port_name: port_mode signal_type;
4 port_name: port_mode signal_type;
5 ......
6 );
7 end [entity] [name_of_entity];
```

Listing 2.2 Entity declaration
Listing 2.3 shows the entity description for the black box of Fig. 2.1.

```vhdl
entity black_box is
  port(
    A: in std_logic_vector(1 downto 0);
    B: out std_logic;
    C: inout std_logic;
    D: buffer std_logic
  );
end black_box;
```

**Listing 2.3** Entity black box

Architecture contains a description of how the circuit should function, from which the actual circuit is inferred. A syntax for an architecture description is shown in listing 2.4.

```vhdl
architecture architecture_name of entity_name is
  [architecture_declarative_part]
begin
  [architecture_statements_part]
end [architecture] [architecture_name];
```

**Listing 2.4** Architecture syntax

Listing 2.5 shows an example of an architecture description for an AND gate. A complete description of the AND gate including libraries and entity is shown in listing 2.6. You may check the next related books [13, 14, 15, 16].

```vhdl
architecture example of AND_G is
begin
  C <= A AND B;
  -- This is a comment
  -- C is an output
  -- A, B are inputs
end architecture example;
```

**Listing 2.5** Architecture of AND gate

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

entity AND_G is
  port(
    A: in std_logic;
    B: in std_logic;
    C: out std_logic
  );
end AND_G;

architecture example of AND_G is
begin
  C <= A AND B;
end architecture example;
```

**Listing 2.6** AND gate description
2.3 Levels of Abstraction

VHDL allows different styles for architecture description, they can be classified as:

- Behavioral description
- Structural description
- Data flow description

2.3.1 Behavioral Description

Behavioral description reflects the system function, how the system works without taking care about the elements that compose it. It is just a relation between inputs and outputs. A process structure is present in a combinational description. For example, listing 2.7 shows a behavioral description for a XOR gate. For this example it is considered that (Fig. 2.2 and Table 2.1):

if A = B then C = 0

if A ≠ B then C = 1

Listing 2.7 XOR gate behavioral description

Another example is shown in listing 2.8. It shows the behavioral description for the AND gate considering that (Fig. 2.3 and Table 2.2):
2.3 Levels of Abstraction

Table 2.1 XOR true table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

if $A = 1$ and $B = 1$ then $C = 1$

other case $C = 0$

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

entity AND_G is
  port(
    A : in   std_logic;
    B : in   std_logic;
    C : out std_logic
  );
end AND_G;

architecture behavioral of AND_G is
begin
  process(A,B)
  begin
    if A = '1' and B = '1' then
      C <= '1';
    else
      C <= '0';
    end if;
  end process;
end architecture behavioral;
```

Listing 2.8 AND gate behavioral description

Fig. 2.2 RTL XOR

Fig. 2.3 RTL AND
Table 2.2  AND true table

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.3  2-bit comparator true table

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 2.4  2-bit comparator

Listing 2.9 shows the behavioral description of a 2-bit comparator (Table 2.3). Figure 2.4 shows the inputs and outputs of the 2-bit comparator. For the behavioral description it is considered that:

if \( A = 1 \) and \( B = 1 \) then \( C = 1 \)

other case \( C = 0 \)
library IEEE;
use IEEE.std_logic_1164.all;

entity comparator_2bits is
port(
A : in std_logic_vector(1 downto 0);
B : in std_logic_vector(1 downto 0);
G : out std_logic;
E : out std_logic;
L : out std_logic
);
end comparator_2bits;

architecture behavioral of comparator_2bits is
begin
combinational: process (A,B)
begin
if A>B then
G := '1';
else
G := '0';
end if;
if A=B then
E := '1'
else
E := '0';
end if;
if A<B then
L := '1';
else
L := '0';
end if;
end process combinational;
end architecture behavioral;

Listing 2.9 2-bit comparator behavioral description

2.3.2 Data Flow Description

Data flow description designates the way how data can be transferred from one signal to another without using sequential statements. The data flow descriptions are concurrent; these kinds of descriptions allow to define the flow that data take from one module to another. An example of data flow description is shown in listing 2.10 (Table 2.4, Fig. 2.5):

if A = 1 and B = 1 then C = 0
Table 2.4 NAND true table

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 2.5 RTL NAND

Table 2.5 OR true table

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

other case $C = 1$

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

entity NAND_G is
  port(
    A : in std_logic;
    B : in std_logic;
    C : out std_logic
  );
end NAND_G;

architecture Data_flow of NAND_G is
begin
  C <= '0' when (A = '1' and B = '1') else '1';
end architecture Data_flow;
```

Listing 2.10 NAND gate data flow description

Another example of data flow description is shown in listing 2.11. In this case, the data flow description for the OR gate considers that (Table 2.5, Fig. 2.6):

if $A = 0$ and $B = 0$ then $C = 0$

other case $C = 1$
2.3 Levels of Abstraction

Fig. 2.6 RTL OR

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

entity OR_G is
  port(
    A : in  std_logic;
    B : in  std_logic;
    C : out std_logic
  );
end OR_G;

architecture Data_flow of OR_G is
begin
  C <= '0' when (A = '0' and B = '0') else '1';
end architecture Data_flow;
```

Listing 2.11 OR gate data flow description

Listing 2.9 shows the data flow description of a 2-bit comparator. Figure 2.4 shows the inputs and output of the 2-bit comparator and Table 2.3 its True Table.

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

entity comparator_2bits is
  port(
    A : in std_logic_vector(1 downto 0);
    B : in std_logic_vector(1 downto 0);
    G : out std_logic;
    E : out std_logic;
    L : out std_logic
  );
end comparator_2bits;

architecture data_flow of comparator_2bits is
begin
  G <= '1' when A > B else '0';
  E <= '1' when A = B else '0';
  L <= '1' when A < B else '0';
end architecture data_flow;
```

Listing 2.12 2-bit comparator data flow description
2.3.3 Structural Description

Structural description is based on established logic models (gates, adders, counters, etc.), which are called as components and they are interconnected in a netlist. Structural description has a hierarchy, it is necessary to reduce the design in small modules (components). These components will be called into another module of more hierarchy. This reduction allows a practical analysis of small modules and it is a simple form to describe.

Figure 2.7 shows an example of structural description, in this example are used the AND, OR, XOR gates described above. Entity “example” is the top level design. Listing 2.13 shows the structural description for the example.

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

entity example is
  port (A: in std_logic; B: in std_logic; C: in std_logic; D: in std_logic; F: out std_logic);
end example;

architecture structural of example is
component AND_G
  port (A: in std_logic; B: in std_logic; C: out std_logic);
end component;
component OR_G
  port (A: in std_logic; B: in std_logic; C: out std_logic);
end component;
```
2.3 Levels of Abstraction

Listing 2.13 Structural Description Example

Listing 2.14 is the structural description of the 2-bit comparator, its RTL was divided into three sections. The first one corresponds to G signal, when A is greater than B, the RTL is shown in Fig. 2.8. The second one shows the RTL for E signal, when A is equal to B, in this case Fig. 2.9 shows its RTL. Finally, the RTL for signal L is shown in Fig. 2.10, when A is lower than B. This example for the structural description of a 2-bit comparator, shows different levels of abstraction, beginning with gates, their interconnections into a more complex gates (for example the OR4_G is an OR with four inputs), the description of a logic function (G, E, L) and finally a combinational circuit (comparator).

Fig. 2.8 RTL signal G
library ieee;
use ieee.std_logic_1164.all;

entity comparator_2bits is
port ( A : in std_logic_vector(1 downto 0);
   B : in std_logic_vector(1 downto 0);
   G : out std_logic; — A > B
   E : out std_logic; — A = B
   L : out std_logic — L < B
);
end comparator_2bits;

architecture structural of comparator_2bits is
component AND_G
   port(
component OR3_G
   port(
      A : in std_logic;
      B : in std_logic;
      C : in std_logic;
      D : out std_logic
   );
end component;

component AND4_G
   port(
      A : in std_logic;
      B : in std_logic;
      C : in std_logic;
      D : in std_logic;
      E : out std_logic
   );
end component;

component AND3_G
   port(
      A : in std_logic;
      B : in std_logic;
      C : in std_logic;
      D : out std_logic
   );
end component;

component OR4_G
   port(
      A : in std_logic;
      B : in std_logic;
      C : in std_logic;
      D : in std_logic;
      E : out std_logic
   );
end component;

signal A1n, A0n, B0n, B1n : std_logic;
signal S1, S2, S3, S4, S5, S6, S7, S8, S9, S10 : std_logic;

begin
   B0n <= not B(0);
   B1n <= not B(1);
   A0n <= not A(0);
   A1n <= not A(1);
2.4 Modules Description Examples

In this section, a description and simulation of some common circuits are given. For example, the blocks: multiplexer, adder, decoder, flip_flop, registers, and counters.

2.4.1 Combinational Circuits

Some gates were described above, so the first example is a simple multiplexer 2 to 1. Mux2_1 RTL is shown in Fig. 2.11 and its description in listing 2.15. Its simulation using Active-HDL is presented in Fig. 2.12.
2.4 Modules Description Examples

Figure 2.11 presents a multiplexor 4 to 1, but in this case each input is a vector of “n-bit” except the input S which has 2-bit width and it does not depend on the generic n. To declare n the keyword generic is used as is shown in listing 2.16, in line 5. n is the integer type and its default value is four. By using the generic keyword, the value of the vector width can be modified when the multiplexer is used as a component. The description used the with/select structure, for the last case (“11”) the keyword others is applied, others included all the combination described for std_logic signals (see Sect. 2.2). The simulation of the mux4_1_n is shown in Fig. 2.14.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux4_1_n is
  generic(n : integer := 4);
  Port ( I0 : in STD_LOGIC_VECTOR (n−1 downto 0);
       I1 : in STD_LOGIC_VECTOR (n−1 downto 0);
   Y <= I0 when S = '0' else I1;
end data_flow;
```

Listing 2.15  Mux2_1 description

```vhdl
begin
  Y <= I0 when S = '0' else I1;
end data_flow;
```

Listing 2.15  Mux2_1 description
Listing 2.16  Mux4_1_n description

An example of hexadecimal to 7 segments decoder is shown below. Figure 2.15 shows one input vector of 4 bits and one output vector of 7 bits, for this example the description is behavioral. Simulation for hexadecimal to 7 segments decoder is presented in Fig. 2.16. Please check that segment “a” corresponds to bit seg(7), “b”
to seg(6), “c” to seg(5), “d” to seg(4), “e” to seg(3), “f” to seg(2), and “g” to seg(1), for this description the signal seg does not have a bit seg(0) declared.

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

entity hex_7seg is
  port ( 
    hex : in std_logic_vector(3 downto 0);
    seg : out std_logic_vector(7 downto 1) 
  );
end hex_7seg;

architecture behavioral of hex_7seg is
begin
  process(hex)
  begin
    case hex is 
    -- abcdefg
    when x"0" => Seg <= "1111110";
    when x"1" => Seg <= "0110000";
    when x"2" => Seg <= "1101101";
    when x"3" => Seg <= "1111001";
    when x"4" => Seg <= "0110011";
    when x"5" => Seg <= "1011011";
    when x"6" => Seg <= "1011111";
    when x"7" => Seg <= "1110000";
    when x"8" => Seg <= "1111111";
    when x"9" => Seg <= "1111011";
    when x"A" => Seg <= "1110111";
    when x"B" => Seg <= "0011111";
    when x"C" => Seg <= "1001110";
    when x"D" => Seg <= "0111101";
  end case;
end process;
```

Fig. 2.15 Hexadecimal to 7 segments decoder

Fig. 2.16 Simulation hexadecimal to 7 segments decoder
Listing 2.17  Hexadecimal to 7 segments decoder description

Figure 2.17 shows an RTL for a complete adder of 4-bit. The adder has three inputs, two vectors of 4 bits (A and B) and one signal of 1 bit (Cin), and two outputs, one signal of one bit (Cou) and one vector of 4 bits (Sum). A and B are the numbers to be added, Cin is the input carry, Cou is the output carry and Sum is the result of the sum. Listing 2.18 shows a generic adder description, it can be seen in line 6 a generic integer and its default value set to 4. Three internal signals of unsigned type are used for data conversion and to store the internal sum (C, Ai, Bi). These conversions are shown in lines 22 and 23 and the sum in line 26. Finally, the result is converted to std_logic type in lines 29 and 30. The adder simulation is shown in Fig. 2.18.

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

Fig. 2.17  RTL adder

Fig. 2.18  Simulation adder
entity adder_n is
generic(n : integer := 4);
port (
    A : in std_logic_vector(n-1 downto 0);
    B : in std_logic_vector(n-1 downto 0);
    Cin : in std_logic;
    Sum : out std_logic_vector(n-1 downto 0);
    Cou : out std_logic
);
end adder_n;

architecture behavioral of adder_n is
signal C : unsigned(n downto 0);
signal Ai,Bi : unsigned(n-1 downto 0);
beginn
    −− data conversion to unsigned
    Ai <= unsigned(A);
    Bi <= unsigned(B);
    −− adder
    C <= ('0' & Ai) + ('0' & Bi) + ('0' & Cin);
    −− data conversion to std_logic
    Sum <= std_logic_vector(C(n-1 downto 0));
    Cou <= std_logic(C(n));
end behavioral;

Listing 2.18  Adder description

2.4.2 Sequential Circuits

A basic element in sequential logic is the flip_flop D, its RTL view is shown in Fig. 2.19. The inputs for the flip_flop are: asynchronous reset (RST), clock (CLK), and datum (D), the only output signal is Q. The simulation of the flip_flop is presented in Fig. 2.20, here one can see how Q takes the value of D when the clock transition is positive and holds this value until a new clock transition is presented. The behavioral description of the flip_flop is presented in listing 2.19

library ieee;
use ieee.std_logic_1164.all;

entity flip_flop_d is
    port (  
        RST : in std_logic;
        CLK : in std_logic;
        D : in std_logic;
        Q : out std_logic
    );
end;
Figure 2.21 shows a RTL RTL of a parallel-parallel enable register of four bits, each bit is stored in a flip_flop. Its inputs are: asynchronous reset (RST), clock (CLK), enable (E), and data (D), the output signal is a vector Q. The simulation of the register can be seen in Fig. 2.22. In the simulation is noted the register behavior, apart from the clock, enable signal must be activated to load D, until E is active, the output Q
2.4 Modules Description Examples

Fig. 2.21 RTL parallel–parallel enable register

takes the value of D (in each positive clock transition), when E is not active Q holds the last data.

Listing 2.20 is the description of the register, with a generic width. In this example, it was necessary an internal signal Qi. Line 18 assigns Qi to the output Q. The enable is described in lines 25–29, when \( E = 1 \) the register load the data D, when \( E = 0 \) it holds the previous values.

Fig. 2.22 Simulation parallel–parallel enable register
library ieee;
use ieee.std_logic_1164.all;

entity register_epp is
    generic(n : integer := 4);
    port(
        RST : in std_logic;
        CLK : in std_logic;
        D : in std_logic_vector(n-1 downto 0);
        E : in std_logic;
        Q : out std_logic_vector(n-1 downto 0)
    );
end register_epp;

architecture behavioral of register_epp is
    signal Qi : std_logic_vector(n-1 downto 0);
begin
    Q <= Qi;

    process(RST,CLK)
    begin
        if RST = '1' then
            Qi <= (others => '0');
        elsif rising_edge(CLK) then
            if E = '1' then
                Qi <= D;
            else
                Qi <= Qi;
            end if;
        end if;
    end process;
end behavioral;

Listing 2.20 Parallel–parallel enable register description

The next example is a left-shift register with a parallel output, the RTL view is shown in Fig. 2.23, to simplify the RTL, common signals (asynchronous reset RST, clock CLK, and enable E) were removed. One can see how the data flow from flip_flop Qi(0) to Q(1) . . . and so on, and the output signal takes the value in a parallel way.

Simulation of the left-shift register with parallel output is shown in Fig. 2.24. L was fixed with a value of one. The register loads this value when the enable (E) is equal to 1 and the clock (CLK) is in a positive transition. Previous values are moved to the left, after four active enable cycles the register is fully load of ones. When the enable is E = '0' the register holds its present value.

Listing 2.21 shows the behavioral description for the left-shift register with parallel output. Line 18 shows the output parallel assignation. Lines 25–29 show the enable and shift functions. In line 26 one can see that signal L is concatenated to vector Qi, due to this one bit of the vector Qi must be removed, in this case the MSB (n–1).
entity reg_shift is
  generic (n : integer := 4);
  port (  
    RST : in std_logic;
    CLK : in std_logic;
    L : in std_logic;
    E : in std_logic;
    Q : out std_logic_vector(n-1 downto 0)
  );
end reg_shift;

architecture behavioral of reg_shift is
signal Q_i : std_logic_vector(n-1 downto 0);
begin
  Q <= Q_i;
Other common sequential circuit is the counter. Figure 2.25 shows a RTL view of a counter ascending/descending with enable module 4. The inputs signals are: clock (CLK), asynchronous reset (RST), and operation counter (OPC). The output is the signal vector Q of 2-bit.

Ascending/descending enable counter simulation is shown in Fig. 2.26. When OPC is “00” or “01” the present value of the counter is holding. When OPC = “11” the value is increased in one each clock cycle and when OPC = “10” the value is decreased in one each clock cycle.

The description of the ascending/descending enable counter is shown in listing 2.22. The behavior of input signal OPC is described from line 24–30. In this example Qi was defined of type unsigned. Line 17 shows the output assigned, due to Qi is the type unsigned a signal conversion must be done using std_logic_vector.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity counter is
process (RST,CLK)
begin
  if RST = '1' then
    Qi <= (others => '0');
  elsif rising_edge(CLK) then
    if E = '1' then
      Qi <= Qi(n-2 downto 0) & L;
    else
      Qi <= Qi;
    end if;
  end if;
end process;
end behavioral;
```

Fig. 2.25  RTL of the ascending/descending enable counter
2.4 Modules Description Examples

Fig. 2.26 Simulation of the ascending/descending enable counter

Listing 2.22 Counter ascending/descending enable description

The last example is a finite state machine (FSM). The inputs of the FSM are: asynchronous reset (RST), clock (CLK), enable (A), and enable (B). The output is a vector of 2 bits (Y). The FSM is shown in Fig. 2.27. The FMS has four states, when the reset is active the FSM goes to state 1. For each state if signal A = ‘0’ then the FSM stays is the actual state, if A = ‘1’ and B = ‘1’ the FSM goes to the next state, for A = ’1’ and B = ‘0’ the FSM returns to the previous state. In state one (S1) the output is Y = “00”, Y = “01” for S2, Y = “10” for S3 and Y = “11” for S4. This is a Moore Machine, then, the output depends on the actual state.
RTL view of the FSM is shown in Fig. 2.28. Simulation of the FSM is presented in Fig. 2.29, here one can see that the FSM has a behavior as the previous counter example, signal A and B represent the signal OPC in the counter.

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

entity fsm is
  port ( 
    RST : in  std_logic;  
    CLK : in  std_logic;  
    A   : in  std_logic;  
    B   : in  std_logic;  
    Y   : out std_logic_vector(1 downto 0)
  );
end entity fsm;
```

---

**Fig. 2.27** Finite state machine

**Fig. 2.28** RTL of the finite state machine
architecture behavioral of fsm is
begin
    process(CLK, RST)
        begin
            if RST = '1' then
                Qi <= "00";
            elsif rising_edge(CLK) then
                case Qi is
                    when "00" =>
                        if A = '0' then
                            Qi <= "00";
                        elsif B = '1' then
                            Qi <= "01";
                        else
                            Qi <= "11";
                        end if;
                    when "01" =>
                        if A = '0' then
                            Qi <= "01";
                        elsif B = '1' then
                            Qi <= "10";
                        else
                            Qi <= "00";
                        end if;
                    when "10" =>
                        if A = '0' then
                            Qi <= "10";
                        elsif B = '1' then
                            Qi <= "11";
                        else
                            Qi <= "01";
                        end if;
                    when others =>
                        Qi <= "10";
                end case;
            end if;
        end process;
end fsm;
if A = '0' then
  Qi <= "11";
elsif B = '1' then
  Qi <= "00";
else
  Qi <= "10";
end if;
end case;
end if;
end process;
end behavioral;

Listing 2.23  Finite state machine behavioral description

The behavioral description of the FSM is presented in listing 2.23. To describe the FSM a case structure is used, for each state one case is used. The output is assigned in line 17.
Engineering Applications of FPGAs
Chaotic Systems, Artificial Neural Networks, Random
Number Generators, and Secure Communication
Systems
Tlelo-Cuautle, E.; Rangel-Magdaleno, J.; de la Fraga, L.G.
2016, XVI, 222 p. 204 illus., 130 illus. in color.,
Hardcover
ISBN: 978-3-319-34113-2