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 ©.

```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.std_logic_unsigned.all;
4 use IEEE.std_logic_arith.all;
5 use IEEE.numeric_std.all;
```

Listing 2.1 Libraries

E. Tlelo-Cuautle et al., *Engineering Applications of FPGAs*,
DOI 10.1007/978-3-319-34115-6_2

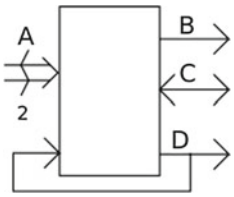


Fig. 2.1 Black box

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.

```
entity black_bok 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.

```
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].

```
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

```
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):

$$\text{if A} = \text{B then C} = 0$$

$$\text{if A} \neq \text{B then C} = 1$$

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity XOR_G is
5    port(
6    A : in  std_logic;
7    B : in  std_logic;
8    C : out std_logic
9    );
10 end XOR_G;
11
12 architecture behavioral of XOR_G is
13 begin
14   process(A,B)
15   begin
16     if A = B then
17       C <= '0';
18     else
19       C <= '1';
20     end if;
21   end process;
22 end architecture behavioral;
```

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):

Fig. 2.2 RTL XOR

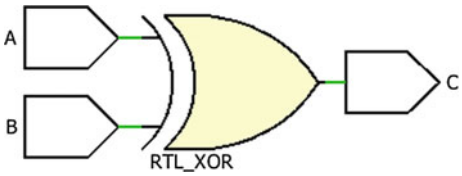


Table 2.1 XOR true table

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

$$\text{if A } = 1 \text{ and B} = 1 \text{ then C} = 1$$

$$\text{other case } \text{C} = 0$$

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity AND_G is
5    port(
6    A : in  std_logic;
7    B : in  std_logic;
8    C : out std_logic
9    );
10 end AND_G;
11
12 architecture behavioral of AND_G is
13 begin
14   process(A,B)
15   begin
16     if A = '1' and B = '1' then
17       C <= '1';
18     else
19       C <= '0';
20     end if;
21   end process;
22 end architecture behavioral;
```

Listing 2.8 AND gate behavioral description

Fig. 2.3 RTL AND

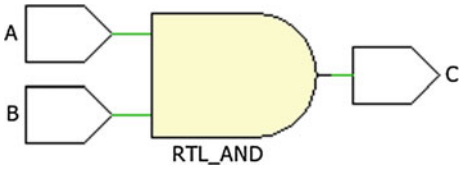


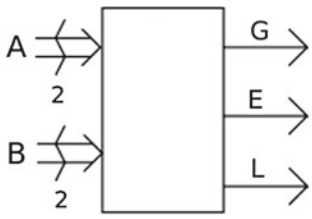
Table 2.2 AND true table

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.3 2-bit comparator true table

A	B	G	E	L
00	00	0	1	0
00	01	0	0	1
00	10	0	0	1
00	11	0	0	1
01	00	1	0	0
01	01	0	1	0
01	10	0	0	1
01	11	0	0	1
10	00	1	0	0
10	01	1	0	0
10	10	0	1	0
10	11	0	0	1
11	00	1	0	0
11	01	1	0	0
11	10	1	0	0
11	11	0	1	0

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:

$$\text{if A} = 1 \text{ and B} = 1 \text{ then C} = 1$$

$$\text{other case } \text{C} = 0$$

```
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):

$$\text{if A } = 1 \text{ and B} = 1 \text{then C } = 0$$

Table 2.4 NAND true table

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Fig. 2.5 RTL NAND

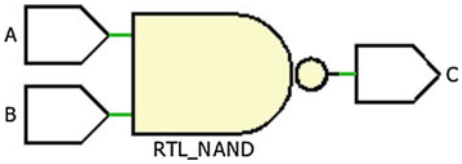


Table 2.5 OR true table

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

$$\text{other case } C = 1$$

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity NAND_G is
5    port(
6    A : in  std_logic;
7    B : in  std_logic;
8    C : out std_logic
9    );
10 end NAND_G;
11
12 architecture Data_flow of NAND_G is
13 begin
14
15   C <= '0' when (A = '1' and B = '1') else '1';
16
17 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):

$$\text{if A } = 0 \text{ and B } = 0 \text{ then C } = 0$$

$$\text{other case C } = 1$$

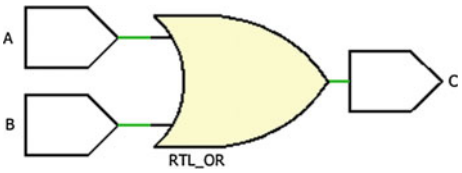


Fig. 2.6 RTL OR

```
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

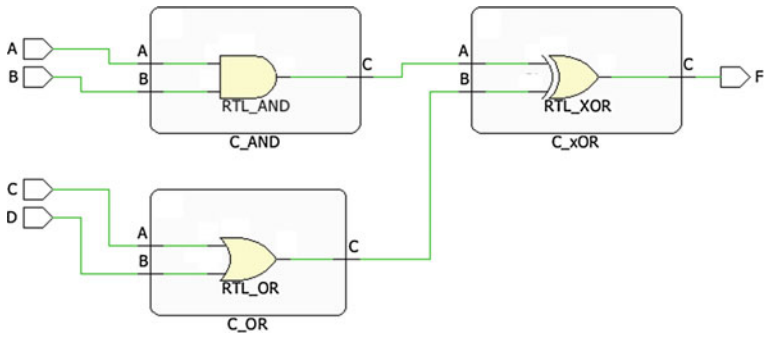


Fig. 2.7 RTL EXAMPLE

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.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity example is
5      port (
6      A : in  std_logic;
7      B : in  std_logic;
8      C : in  std_logic;
9      D : in  std_logic;
10     F : out std_logic
11         );
12 end example;
13
14 architecture structural of example is
15 component AND_G
16     port(
17     A : in   std_logic;
18     B : in   std_logic;
19     C : out  std_logic
20     );
21 end component;
22 component OR_G
23     port(
```

```
    A : in    std_logic;
    B : in    std_logic;
    C : out   std_logic
    );
end component;
component XOR_G
    port(
    A : in    std_logic;
    B : in    std_logic;
    C : out   std_logic
    );
end component;

signal SI0, SI1, SI2, SI3, SI4 : std_logic;

begin
    DUT1  : AND_G  port map(A,B,SI0);
    DUT2  : XOR_G  port map(SI0,SI1,F);
    DUT3  : OR_G   port map(C,D,SI1);

end structural;
```

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 that 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).

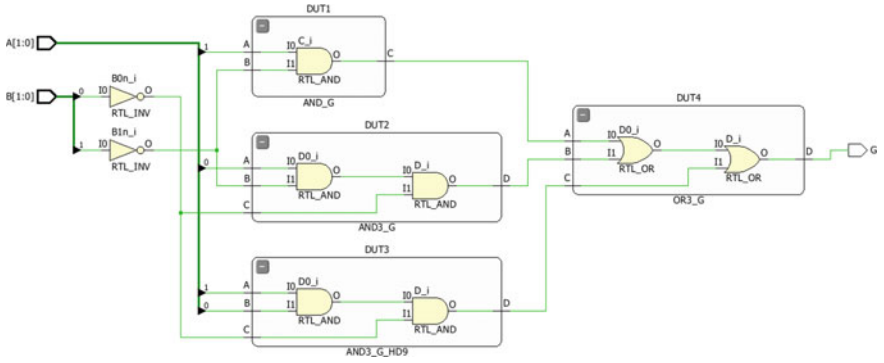


Fig. 2.8 RTL signal G

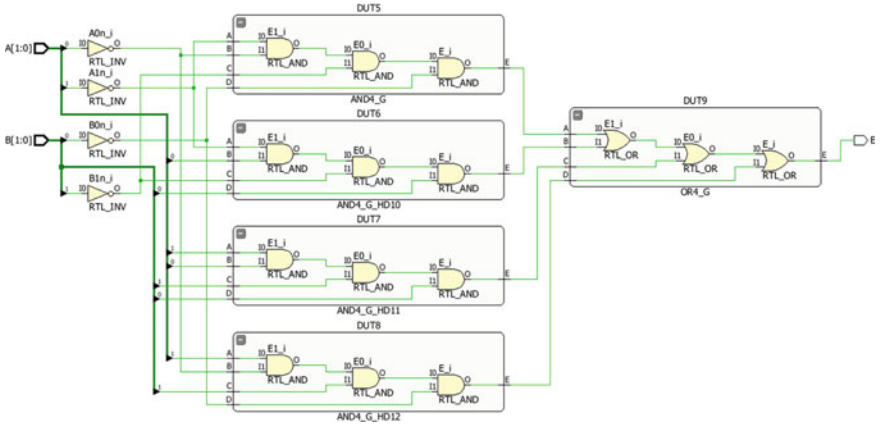


Fig. 2.9 RTL signal E

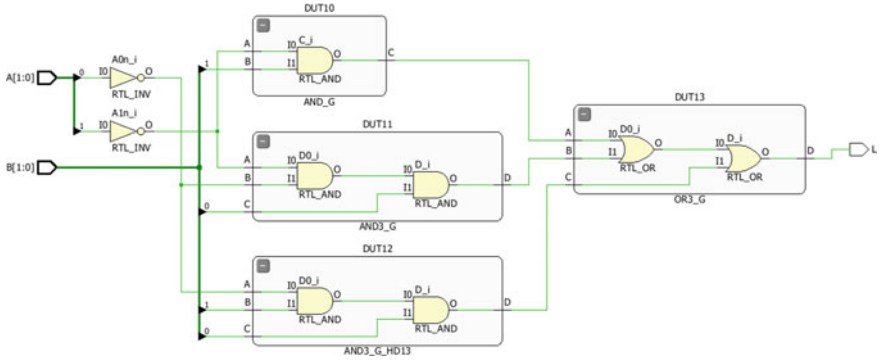


Fig. 2.10 RTL signal L

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity comparator_2bits is
5    port (
6    A : in  std_logic_vector(1 downto 0 );
7    B : in  std_logic_vector(1 downto 0 );
8    G : out std_logic;     -- A > B
9    E : out std_logic;     -- A = B
10   L : out std_logic   -- L < B
11     );
12 end comparator_2bits;
13
14 architecture structural of comparator_2bits is
15
16 component AND_G
17   port(
```

```
18    A : in  std_logic;
19    B : in  std_logic;
20    C : out std_logic
21    );
22  end component;
23
24  component OR3_G
25    port(
26    A : in  std_logic;
27    B : in  std_logic;
28    C : in  std_logic;
29    D : out std_logic
30    );
31  end component;
32
33  component AND4_G
34    port(
35    A : in  std_logic;
36    B : in  std_logic;
37    C : in  std_logic;
38    D : in  std_logic;
39    E : out std_logic
40    );
41  end component;
42
43  component AND3_G
44    port(
45    A : in  std_logic;
46    B : in  std_logic;
47    C : in  std_logic;
48    D : out std_logic
49    );
50  end component;
51
52  component OR4_G
53    port(
54    A : in  std_logic;
55    B : in  std_logic;
56    C : in  std_logic;
57    D : in  std_logic;
58    E : out std_logic
59    );
60  end component;
61
62  signal A1n,A0n, B0n,B1n : std_logic;
63  signal S1,S2,S3,S4,S5,S6,S7,S8,S9,S10 : std_logic;
64
65  begin
66    B0n <= not B(0);
67    B1n <= not B(1);
68    A0n <= not A(0);
69    A1n <= not A(1);
70
```

```
71   ------- G ----------
72   DUT1: AND_G  port map(A(1),B1n,S1);
73   DUT2: AND3_G port map(A(0),B1n,B0n,S2);
74   DUT3: AND3_G port map(A(1),A(0),B0n,S3);
75   DUT4: OR3_G  port map(S1,S2,S3,G);
76
77   ------- E ----------
78   DUT5: AND4_G port map(A1n,A0n,B1n,B0n,S4);
79   DUT6: AND4_G port map(A1n,A(0),B1n,B(0),S5);
80   DUT7: AND4_G port map(A(1),A(0),B(1),B(0),S6);
81   DUT8: AND4_G port map(A(1),A0n,B(1),B0n,S7);
82   DUT9: OR4_G  port map(S4,S5,S6,S7,E);
83
84   ------- L ----------
85   DUT10: AND_G  port map(A1n,B(1),S8);
86   DUT11: AND3_G port map(A1n,A0n,B(0),S9);
87   DUT12: AND3_G port map(A0n,B(1),B(0),S10);
88   DUT13: OR3_G  port map(S8,S9,S10,L);
89
90 end structural;
```

Listing 2.14 2-bit Comparator structural description

2.4 Modules Description Examples

In this section, a descriptions and simulation of some common circuits are given. For example, the blocks: multiplexor, 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 usign Active-HDL is presented in Fig. 2.12

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity mux2_1 is
5    port (
6    I0 : in  std_logic;
7    I1 : in  std_logic;
8    S  : in  std_logic;
9    Y  : out std_logic
10     );
11 end mux2_1;
12
13 architecture data_flow of mux2_1 is
```

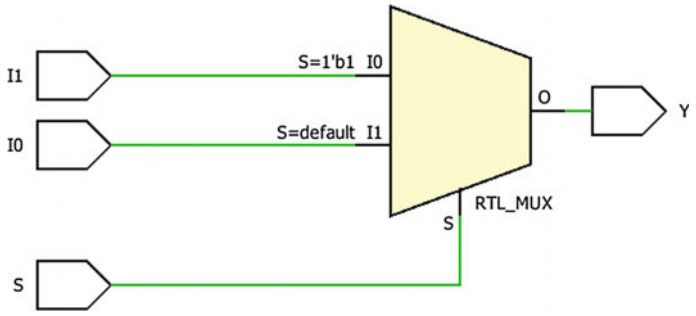


Fig. 2.11 Mux2_1

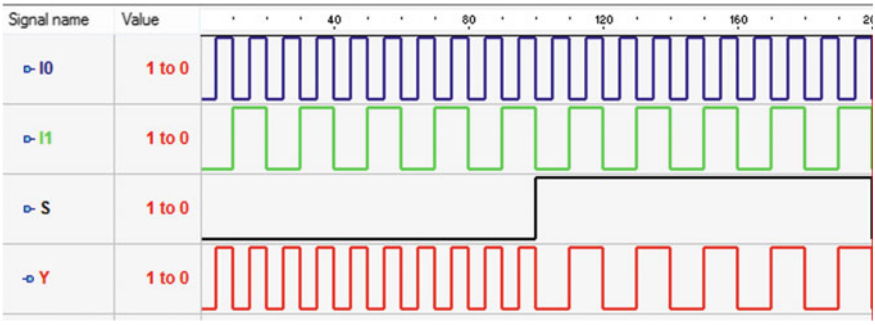


Fig. 2.12 Simulation of Mux2_1

```
14 begin
15   Y <= I0 when S = '0' else I1;
16 end data_flow;
```

Listing 2.15 Mux2_1 description

Figure 2.13 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.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux4_1_n is
5   generic(n : integer := 4);
6   Port ( I0 : in  STD_LOGIC_VECTOR (n-1 downto 0);
7          I1 : in  STD_LOGIC_VECTOR (n-1 downto 0);
```

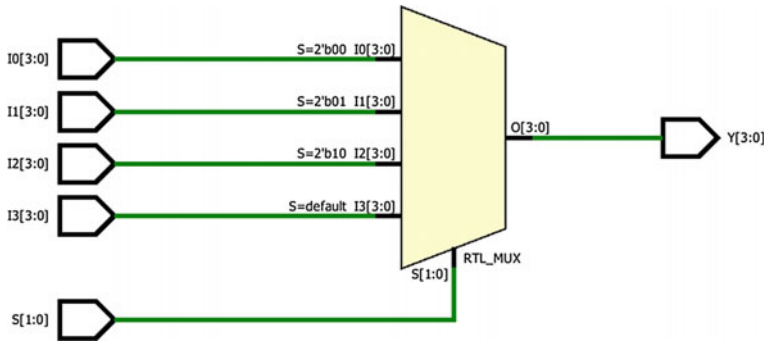


Fig. 2.13 Mux4_1_n

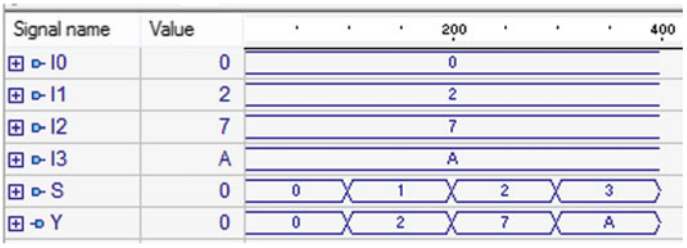


Fig. 2.14 Simulation of Mux4_1_n

```
        I2 : in  STD_LOGIC_VECTOR (n-1 downto 0);
        I3 : in  STD_LOGIC_VECTOR (n-1 downto 0);
        S  : in  STD_LOGIC_VECTOR (1   downto 0);
        Y  : out STD_LOGIC_VECTOR (n-1 downto 0)
        );
end mux4_1_n;

architecture data_flow of mux4_1_n is

begin
   with S select
      Y <= I0 when "00",
           I1 when "01",
           I2 when "10",
           I3 when others;

end data_flow;
```

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"

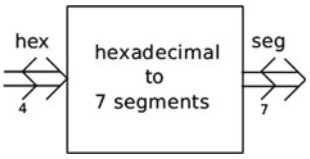


Fig. 2.15 Hexadecimal to 7 segments decoder

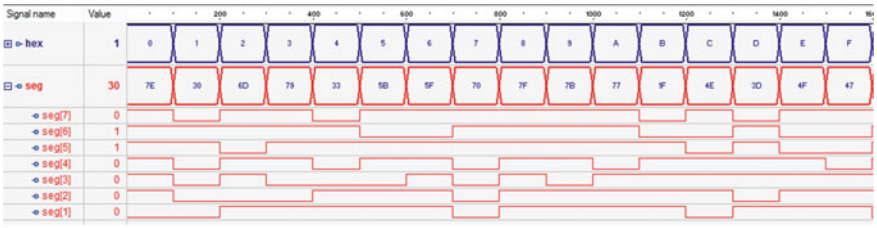


Fig. 2.16 Simulation hexadecimal to 7 segments decoder

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";
```

```
30        when x"E" => Seg <= "1001111";
31        when others => Seg <= "1000111"; -- F
32      end case;
33    end process;
34
35  end behavioral;
```

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.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
```

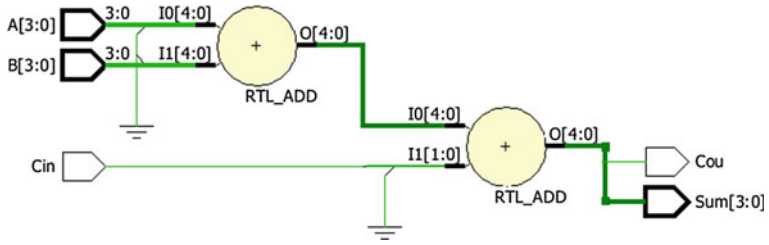


Fig. 2.17 RTL adder

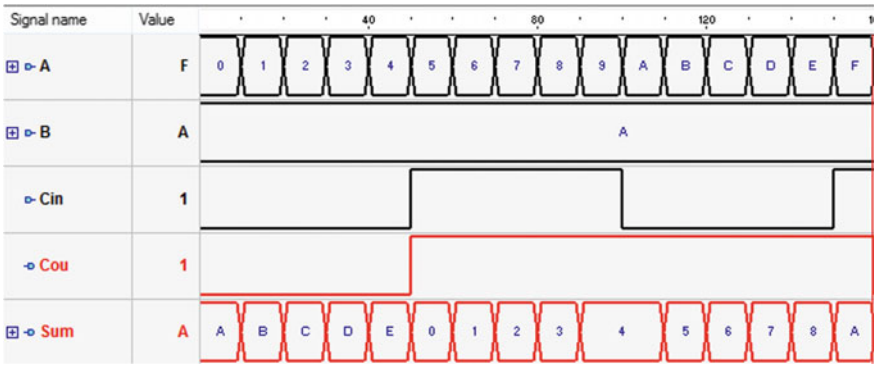


Fig. 2.18 Simulation adder

```
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);
begin

  -- 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
    );
```

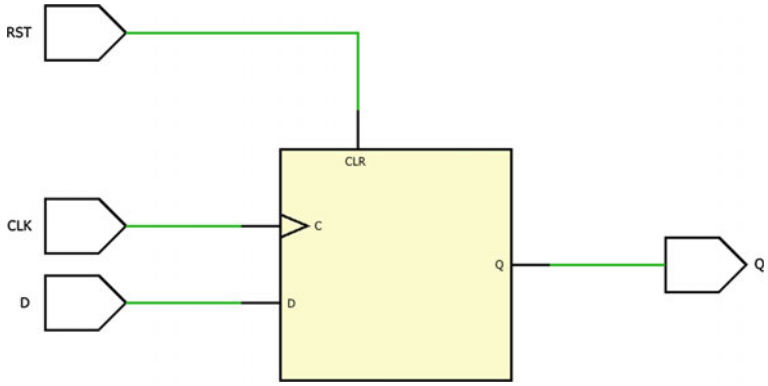


Fig. 2.19 RTL Flip_Flop D

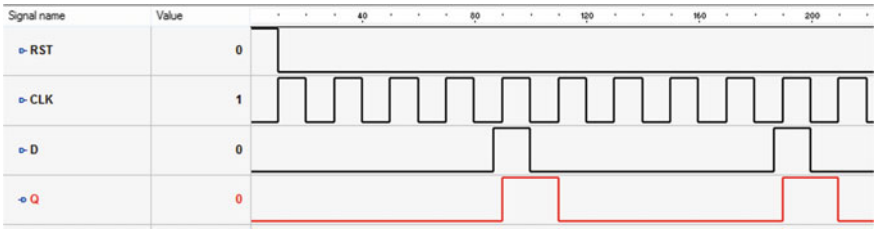


Fig. 2.20 Simulation Flip_Flop D

```
11 end flip_flop_d;
12
13 architecture behavioral of flip_flop_d is
14 begin
15   process(RST,CLK)
16   begin
17     if RST = '1' then
18       Q <= '0';
19     elsif rising_edge(CLK) then
20       Q <= D;
21     end if;
22   end process;
23
24 end behavioral;
```

Listing 2.19 Flip_Flop D description

Figure 2.21 shows a RTLRTL 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

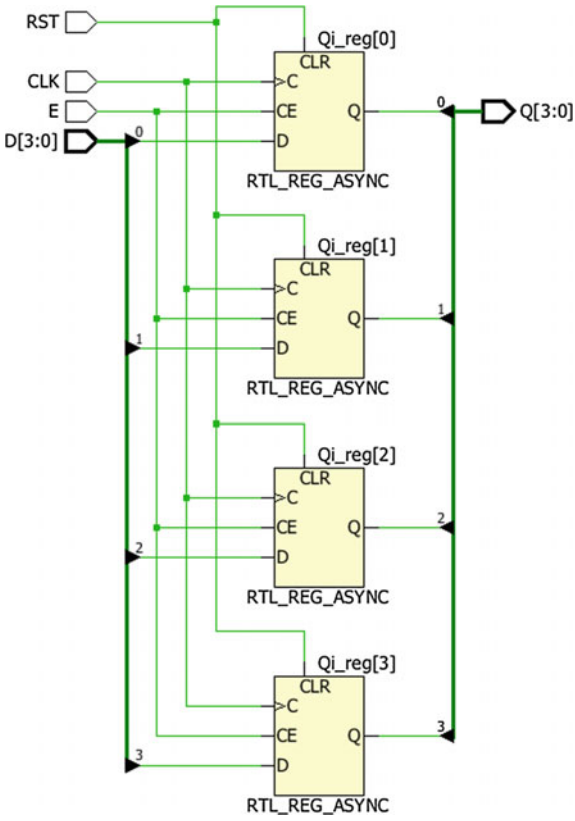


Fig. 2.21 RTL parallel–parallel enable register

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.

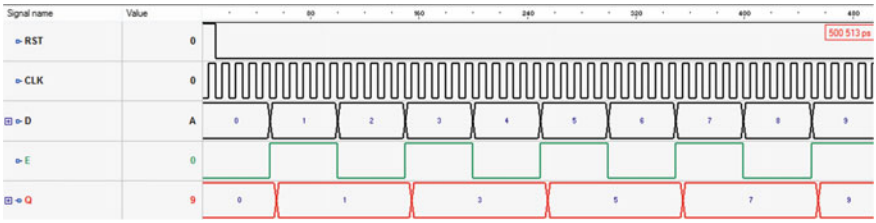


Fig. 2.22 Simulation parallel–parallel enable register

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity register_epp is
5    generic(n : integer := 4);
6    port(
7    RST : in  std_logic;
8    CLK : in  std_logic;
9    D   : in  std_logic_vector(n-1 downto 0);
10   E   : in  std_logic;
11   Q   : out std_logic_vector(n-1 downto 0)
12   );
13 end register_epp;
14
15 architecture behavioral of register_epp is
16 signal Qi : std_logic_vector(n-1 downto 0);
17 begin
18   Q <= Qi;
19
20   process(RST,CLK)
21   begin
22     if RST = '1' then
23       Qi <= (others => '0');
24     elsif rising_edge(CLK) then
25       if E = '1' then
26         Qi <= D;
27       else
28         Qi <= Qi;
29       end if;
30     end if;
31   end process;
32
33 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).

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

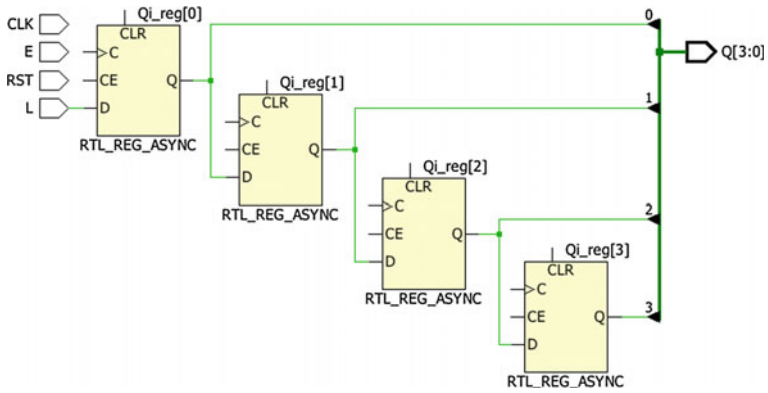


Fig. 2.23 RTL *left*-shift register parallel output

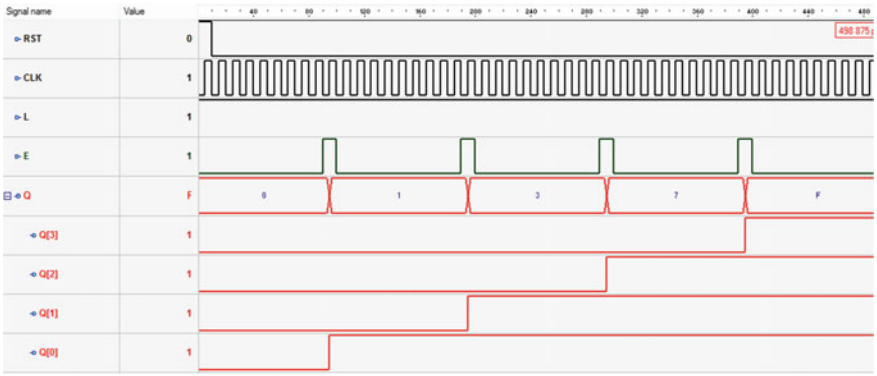


Fig. 2.24 Simulation *left*-shift register parallel output

```
3
4  entity reg_shift is
5    generic(n : integer := 4);
6    port (
7    RST : in  std_logic;
8    CLK : in  std_logic;
9    L   : in  std_logic;
10   E   : in  std_logic;
11   Q   : out std_logic_vector(n-1 downto 0)
12     );
13 end reg_shift;
14
15 architecture behavioral of reg_shift is
16 signal Qi : std_logic_vector(n-1 downto 0);
17 begin
18   Q <= Qi;
19
```

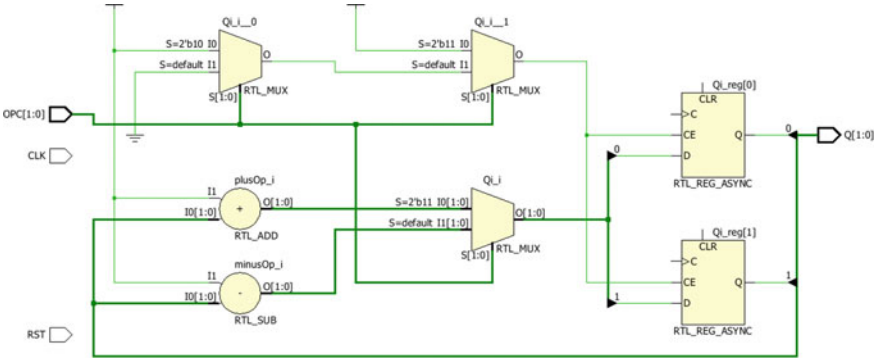


Fig. 2.25 RTL of the ascending/descending enable counter

```
20   process(RST,CLK)
21   begin
22     if RST = '1' then
23       Qi <= (others => '0');
24     elsif rising_edge(CLK) then
25       if E = '1' then
26         Qi <= Qi(n-2 downto 0) & L;
27       else
28         Qi <= Qi;
29       end if;
30     end if;
31   end process;
32
33 end behavioral;
```

Listing 2.21 Left-Shift register parallel output description

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.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity counter is
```

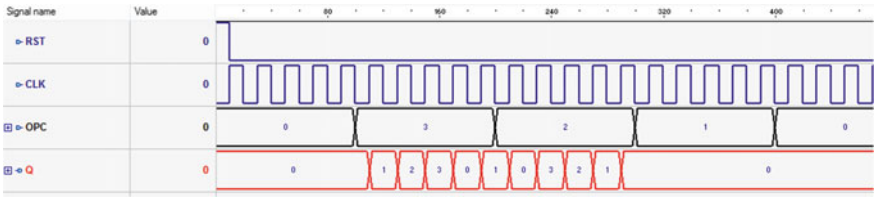


Fig. 2.26 Simulation of the ascending/descending enable counter

```
  port (
  RST : in  std_logic;
  CLK : in  std_logic;
  OPC : in  std_logic_vector(1 downto 0);
  Q   : out std_logic_vector(1 downto 0)
    );
end counter;

architecture behavioral of counter is
signal Qi : unsigned(1 downto 0);
begin
  Q <= std_logic_vector(Qi);

  process(RST,CLK)
  begin
    if RST = '1' then
      Qi <= (others => '0');
    elsif rising_edge(CLK) then
      if OPC = "11" then
        Qi <= Qi + 1;
      elsif OPC = "10" then
        Qi <= Qi - 1;
      else
        Qi <= Qi;
      end if;
    end if;
  end process;

end behavioral;
```

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.

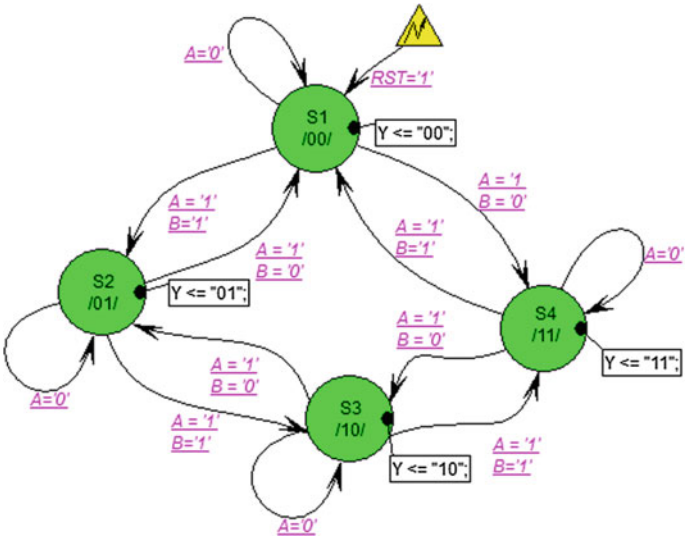


Fig. 2.27 Finite state machine

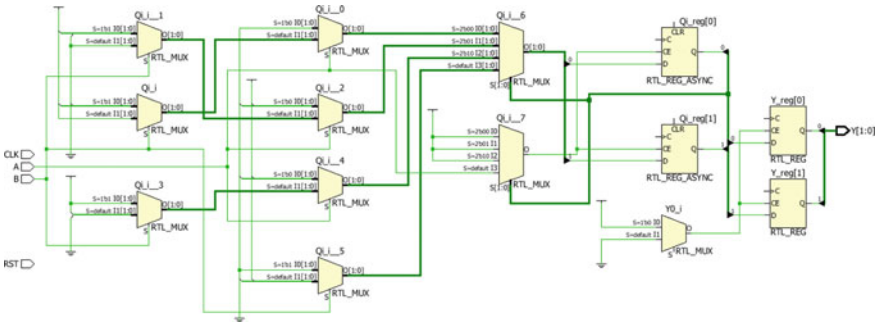


Fig. 2.28 RTL of the finite state machine

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.

```
1  library ieee;
2  use IEEE.std_logic_1164.all;
3
4  entity fsm is
5    port (
6    RST : in  std_logic;
7    CLK : in  std_logic;
8    A   : in  std_logic;
9    B   : in  std_logic;
10   Y   : out std_logic_vector(1 downto 0)
```

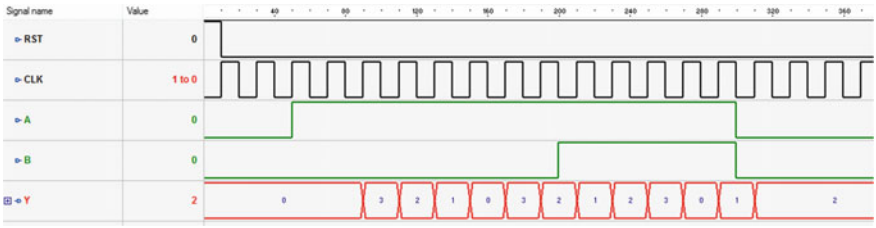


Fig. 2.29 Simulation of the FSM

```
    );
end fsm;

architecture behavioral of fsm is
signal Qi : std_logic_vector(1 downto 0);
begin
  Y <= Qi;
  process(RST,CLK)
  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 =>
```

```
52          if A = '0' then
53            Qi <= "11";
54          elsif B = '1' then
55            Qi <= "00";
56          else
57            Qi <= "10";
58          end if;
59        end case;
60      end if;
61    end process;
62
63  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.

http://www.springer.com/978-3-319-34113-2

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