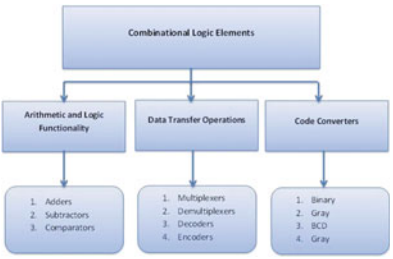
Chapter 2
Basic Logic Circuits and VHDL Description



"We cannot solve our problems with the same thinking we used when we created them." ----- Albert Einstein

Like a C or C++ programmer don't apply the logic. Design the combinational logic by using the HDL

Learn the VHDL constructs and imagine the synthesizable designs and RTL designs using VHDL!

Abstract This chapter describes the overview of various combinational logic elements. The chapter is organized in such a way that reader will be able to understand the concept of synthesizable RTL for the logic gates and small gate count combinational designs using synthesizable VHDL constructs. This chapter describes the basic logic gates, adders, gray-to-binary and binary-to-gray code converters. This chapter also covers the key practical concepts while designing by using the combinational logic elements.

Keywords RTL · Synthesis · AND · NOT · OR · NOR · XOR · XNOR · Tri-state · Bus · Truth table · Combinational · Code converter · Adder · Gray · Binary · ALU · Critical path · Arithmetic operations · Logic minimization · Nine-valued logic · De Morgan's theorem

The original version of this chapter was revised: The incorrect information have been corrected. The erratum to this chapter is available at 10.1007/978-981-10-3296-7_12

V. Taraate, *PLD Based Design with VHDL*,
DOI 10.1007/978-981-10-3296-7_2

2.1 Introduction to Combinational Logic

Combinational logic is implemented by the logic gates, and in the combinational logic, output is the function of present input. The goal of designer is always to implement the logic using minimum number of logic gates or logic cells. Minimization techniques are K-map, Boolean algebra, Shannon's expansion theorems, and hyperplanes.

The conventional design technique using the Boolean algebra can be used for better understanding of the design functionality. The familiarity of the De Morgan's theorem and logic minimization technique can play an important role while coding for the design functionality. The De Morgan's theorem states that

1. Bubbled OR is equal to NAND.
2. Bubbled AND is equal to NOR.

The NOR and NAND gates are universal logic elements and used to design the digital circuit functionality. NOR and NAND can be used as universal logic cells. Figure 2.1 gives more explanation about the De Morgan's Theorem.

The thought process of designer should be such that the design should have the optimal performance with lesser area density. The area minimization techniques play an important role in the design of combinational logic or functions. In the present scenario, designs are very complex; the design functionality is described using the hardware description language as VHDL or Verilog. The subsequent section focuses on the use of VHDL RTL to describe the combinational design. Figure 2.2 illustrates the different types of combinational logic elements. This chapter discusses the basic combinational logic elements used to design the logic circuits.

The complex combinational logic circuits are discussed in the subsequent chapters.

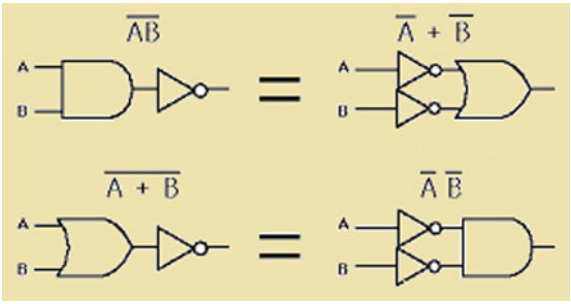


Fig. 2.1 De Morgan's theorems

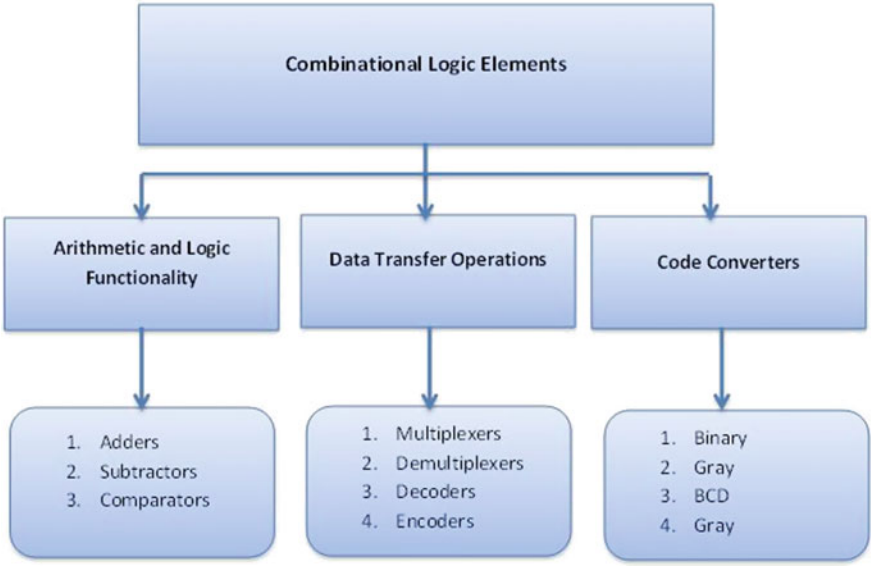


Fig. 2.2 Combinational logic elements

2.2 Logic Gates and Synthesizable RTL Using VHDL

This section discusses about the logic gates and the synthesizable VHDL RTL. In this section, the key VHDL constructs to describe the basic combinational logic gates are discussed.

To have a good understanding of VHDL, let us discuss on the key VHDL terminologies used to describe the combinational logic. Let us make our life simpler by understanding the logical operators as shown in Table 2.1.

Table 2.1 Logical operators

Logical operators	Operator description	VHDL description
NOT/not	Used as negation or to complement the input or signal	y_out <= NOT(a_in);
OR/or	To perform logical OR operation	y_out <= a_in OR b_in;
NOR/nor	To perform logical NOR operation	y_out <= a_in NOR b_in;
AND/and	To perform logical AND operation	y_out <= a_in AND b_in;
NAND/nand	To perform logical NAND operation	y_out <= a_in NAND b_in;
XOR/xor	To perform logical XOR operation	y_out <= a_in XOR b_in;
XNOR/xnor	To perform logical XNOR operation	y_out <= a_in XNOR b_in;

In the subsequent section, the logic design is described by using the concurrent process statement and if-then-else constructs.

2.2.1 NOT or Invert Logic

NOT logic complements the input. Not logic is also called as inverter or complement logic. Synthesizable RTL is shown in Example 2.1. The truth table of NOT logic is shown in Table 2.2.

```
-- invert or not logic using if-then-else

        library ieee;
        use ieee.std_logic_1164.all;

        entity not_logic_gate is
        port(  a_in:  in std_logic;
                 y_out: out std_logic
          );
        end not_logic_gate;

        architecture arch_not of not_logic_gate is
        begin

         process (a_in)
         begin
          if (a_in='0') then
                y_out<= '1';
                else
                y_out<= '0';
                 end if;
                end process;
end arch_not;
```

- For the input 'a_in' is equl to zero the output 'y_out' is equal to logical one
- For the input 'a_in' is equl to one the output 'y_out' is equal to logical zero

Example 2.1 Synthesizable VHDL code for NOT logic

Note Operator (<=) is used for the port or signal assignment. On the other hand, concurrent construct 'process' is used to infer both combinational and sequential logic by using the required VHDL constructs.

Table 2.2 Truth table for NOT logic

a_in	y_out
0	1
1	0

The use of the STD_LOGIC is nine-valued logic, and for the NOT function it is described below.

Input	U	X	0	1	Z	W	L	H	–
Output	U	X	1	0	X	X	1	0	X

Synthesis result for the NOT logic is shown in Fig. 2.3; input port of not logic gate is named as 'a_in' and output as 'y_out'.

The implementation of NOT using universal NAND and NOR logic gate is shown in Fig. 2.4.

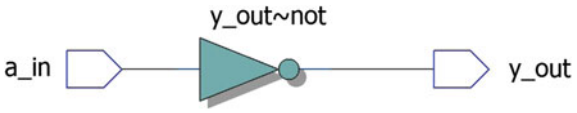


Fig. 2.3 Synthesis result for the NOT logic

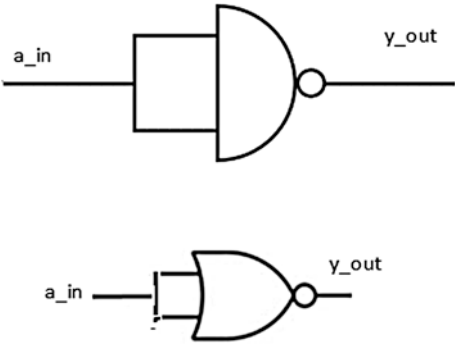
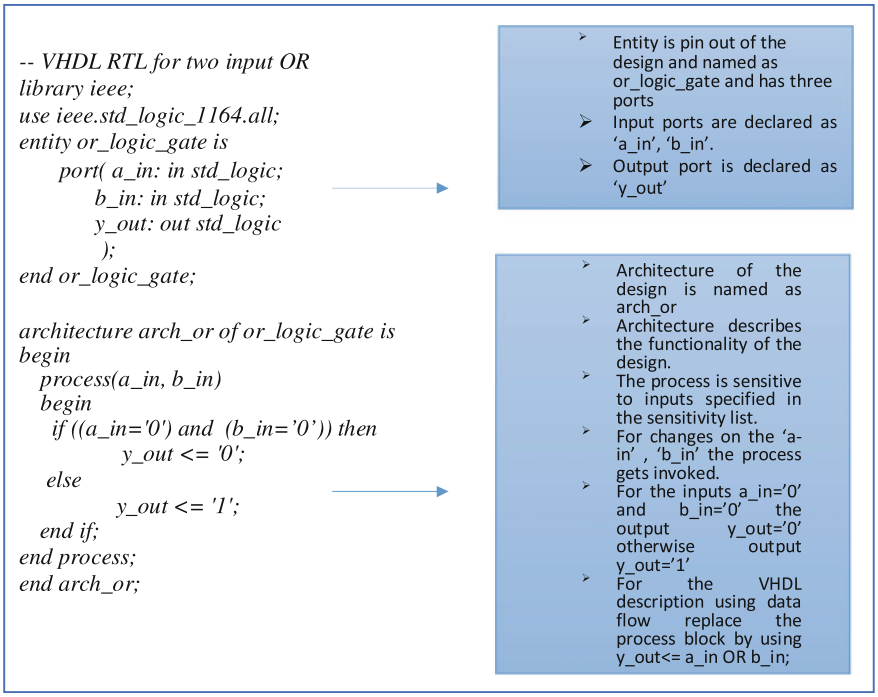


Fig. 2.4 NOT gate implementation using universal logic gates

2.2.2 Two-Input OR Logic

OR logic generates output as logical '1' when one of the inputs is logical '1'.

Synthesis result is shown in Example 2.2. The truth table of OR logic is shown in Table 2.3.



Example 2.2 Synthesizable VHDL code for two-input OR logic

Note While describing the design functionality, make sure that all the input ports are listed in the sensitivity list. Missing required signal from sensitivity list will create simulation and synthesis mismatch and will be discussed in the subsequent chapters.

Table 2.3 Truth table for two-input OR logic

a_in	b_in	y_out
0	0	0
0	1	1
1	0	1
1	1	1

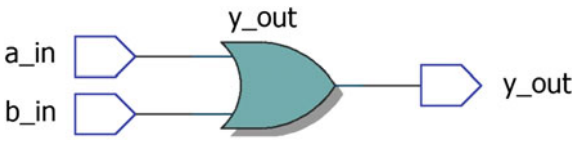


Fig. 2.5 Synthesis result for two-input OR logic

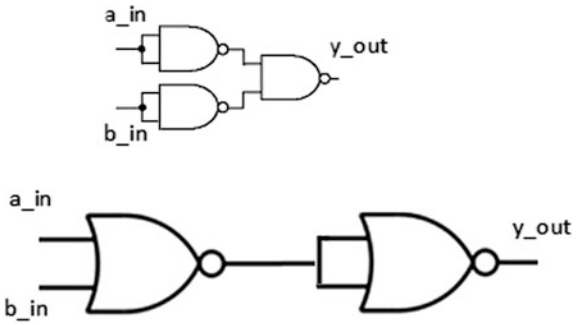


Fig. 2.6 OR gate implementation using universal gates

Synthesis result for the OR logic is shown in Fig. 2.5, input ports of OR logic gate are named as 'a_in' and 'b_in' and output as 'y_out'.

Using universal logic gates the implementation of OR gate is shown in Fig. 2.6.

2.2.3 Two-Input NOR Logic

NOR logic is opposite or complement of the OR logic. Synthesizable RTL is shown in Example 2.3. The truth table of NOR logic is shown in Table 2.4.

Synthesis result for the NOR logic is shown in Fig. 2.7; input ports of NOR logic gates are named as 'a_in' and 'b_in' and output as 'y_out'.

Two-input NOR gate implementation using universal logic gates is shown in Fig. 2.8.

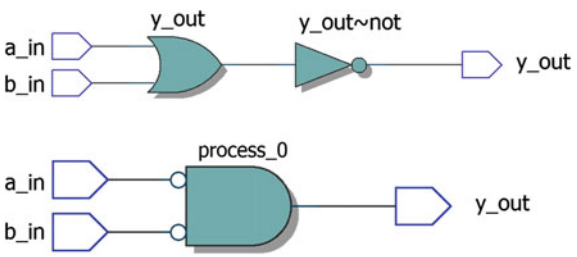


Fig. 2.7 Synthesized two-input NOR logic

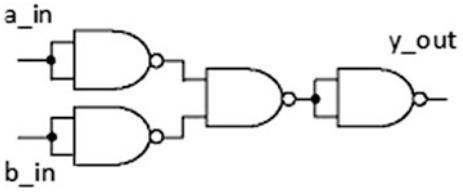


Fig. 2.8 NOR gate implementation using universal gates

```
-- VHDL RTL for two input NOR
library ieee;
use ieee.std_logic_1164.all;
entity nor_logic_gate is
      port ( a_in: in std_logic;
           b_in: in std_logic;
           y_out: out std_logic
            );
end  nor_logic_gate;
```

- Entity is pin out of the design and named as nor_logic_gate and has three ports
- Input ports are declared as 'a_in', 'b_in'.
- Output port is declared as 'y_out'

```
architecture arch_nor of nor_logic_gate is
begin
   process(a_in, b_in)
   begin
    if ((a_in='0') and (b_in='0')) then
              y_out <= '1';
    else
              y_out <= '0';
   end if;
end process;
end arch_nor;
```

- Architecture of the design is named as arch_nor
- Architecture describes the functionality of the design.
- The process is sensitive to inputs specified in the sensitivity list.
- For changes on the 'a-in' , 'b_in' the process gets invoked.
- For the inputs a_in='0' and b_in='0' the output y_out='1' otherwise output y_out='0'
- For the VHDL description using data flow replace the process block by using y_out<= a_in NOR b_in;

Example 2.3 Synthesizable VHDL code for NOR logic

Table 2.4 Truth table for two-input NOR logic

a_in	b_in	y_out
0	0	1
0	1	0
1	0	0
1	1	0

2.2.4 Two-Input AND Logic

AND logic generates an output as logical '1' when both the inputs 'a_in' and 'b_in' are logical '1'. Synthesizable RTL is shown in Example 2.4. The truth table of AND logic is shown in Table 2.5.

```
-- VHDL RTL for two input AND
library ieee;

use ieee.std_logic_1164.all;
entity and_logic_gate is
     port ( a_in: in std_logic;
            b_in: in std_logic;
            y_out: out std_logic
          );

end  and_logic_gate;


architecture arch_and  of and_logic_gate is
begin
   process(a_in, b_in)
   begin
    if ((a_in='1') and (b_in='1')) then
            y_out <= '1';
    else
           y_out <= '0';
   end if;
end process;
end arch_and;
```

- Entity is pin out of the design and named as and_logic_gate and has three ports
- Input ports are declared as 'a_in', 'b_in'.
- Output port is declared as 'y_out'

- Architecture of the design is named as arch_and
- Architecture describes the functionality of the design.
- The process is sensitive to inputs specified in the sensitivity list.
- For changes on the 'a-in' , 'b_in' the process gets invoked.
- For the inputs a_in='1' and b_in='1' the output y_out='1' otherwise output y_out='0'
- For the VHDL description using data flow replace the process block by using y_out<= a_in AND b_in;

Example 2.4 Synthesizable VHDL code for AND logic

Note AND gate is visualized as a series of two switches and used in programmable logic devices (PLD) as one of the elements to realize the required logic. Programmable AND plane can be created by using the AND logic gates with programmable inputs.

Table 2.5 Truth table for two-input AND logic

a_in	b_in	y_out
0	0	0
0	1	0
1	0	0
1	1	1

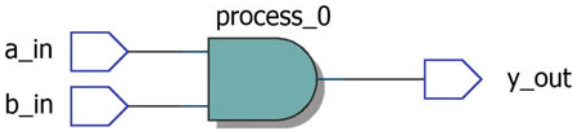


Fig. 2.9 Synthesis result for two-input AND logic

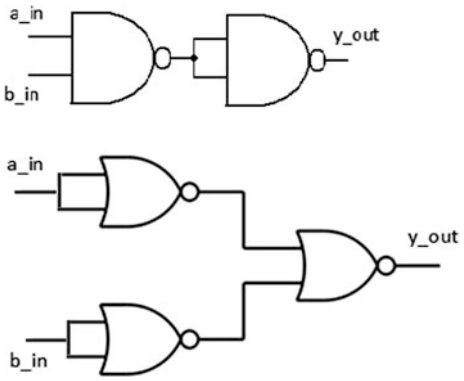


Fig. 2.10 AND gate implementation using universal gates

Synthesized two-input AND logic is shown in Fig. 2.9; input ports of AND logic gate are named as 'a_in' and 'b_in' and output as 'y_out'.

Two-input AND gate implementation using minimum number of universal gates is shown in Fig. 2.10.

2.2.5 Two-Input NAND Logic

NAND logic is opposite or complement of the AND logic. Synthesizable RTL is shown in Example 2.5. The truth table of NAND logic is shown in Table 2.6

Table 2.6 Truth table for two-input NAND logic

a_in	b_in	y_out
0	0	1
0	1	1
1	0	1
1	1	0

```
-- VHDL RTL for two input NAND
library ieee;

use ieee.std_logic_1164.all;
entity nand_logic_gate is
     port ( a_in: in std_logic;
            b_in: in std_logic;
            y_out: out std_logic
           );

end  nand_logic_gate;


architecture arch_nand  of nand_logic_gate is
begin
   process(a_in, b_in)
   begin
    if ((a_in='1') and (b_in='1')) then
            y_out <= '0';
    else
            y_out <= '1';
   end if;
end process;
end arch_nand;
```

- Entity is pin out of the design and named as nand_logic_gate and has three ports
- Input ports are declared as 'a_in', 'b_in'.
- Output port is declared as 'y_out'

- Architecture of the design is named as arch_nand
- Architecture describes the functionality of the design.
- The process is sensitive to inputs specified in the sensitivity list.
- For changes on the 'a-in' , 'b_in' the process gets invoked.
- For the inputs a_in='1' and b_in='1' the output y_out='0' otherwise output y_out='1'
- For the VHDL description using data flow replace the process block by using y_out<= a_in NAND b_in;

Example 2.5 Synthesizable VHDL RTL for two-input NAND logic

Note NAND logic is also treated as universal logic. By using NAND logic, all possible logic functions can be realized. NAND logic is used to implement the storage elements like latches or flip-flops and also to realize combinational functions.

Synthesis result for the NAND logic is shown in Fig. 2.11; input ports of NAND logic gate are named as 'a_in' and 'b_in' and output as 'y_out'.

As stated earlier, NAND is universal gate, and implementation of NAND using NOR is shown in Fig. 2.12.

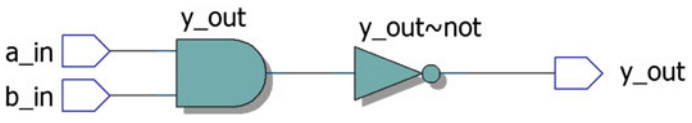


Fig. 2.11 Synthesis result for the result for the two-input NAND logic

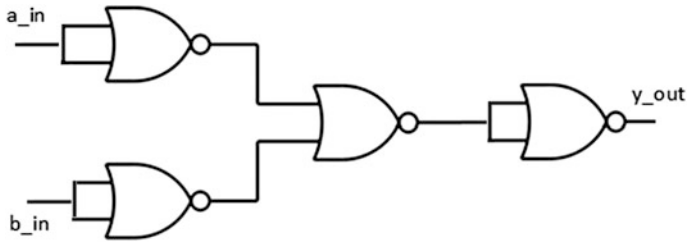


Fig. 2.12 Implementation of NAND using universal gates

2.2.6 *Two-Input XOR Logic*

Two-input XOR is called as exclusive OR logic and generates output as logical '1' when both inputs are not equal. Synthesizable RTL is shown in Example 2.6. The truth table of XOR logic is shown in Table 2.7.

```
-- VHDL RTL for two input XOR

library ieee;
use ieee.std_logic_1164.all;

entity xor_logic_gate is
port( a_in: in std_logic;
      b_in: in std_logic;
      y_out: out std_logic
);
end xor_logic_gate;

architecture arch_xor of xor_logic_gate is
begin

  process(a_in, b_in)
  begin

    if (a_in/=b_in) then
        y_out <= '1';
    else
        y_out <= '0';
  end if;

end process;

end arch_xor;
```

- Entity is the pin out of the design named as xor_logic_gate that has three ports.
- Input ports are declared as 'a_in' and 'b_in'
- Output port is declared as 'y_out'

- Architecture of the xor_logic_gate is named as arch_xor
- Architecture describes the relationship between the inputs and output.
- The procedural block 'Process' is always sensitive to the changes specified in the sensitivity list.
- For any changes in the inputs 'a_in' or b_in the process gets invoked.
- For the inputs, a_in is not equal to b_in output y_out = '1' otherwise output y_out = '0'
- For data flow model replace process block by using *y_out <= a_in xor b_in;*

Example 2.6 Synthesizable VHDL code for two-input XOR logic

Note XOR gate can be implemented by using two-input NAND gates. The number of two-input NAND gates required to implement two-input XOR gate are equal to 4. XOR gates are used to implement arithmetic operations like addition and subtraction. The implementation using minimum number of NAND gates is shown in Fig. 2.13.

Table 2.7 Truth table for two-input XOR logic

a_in	b_in	y_out
0	0	0
0	1	1
1	0	1
1	1	0

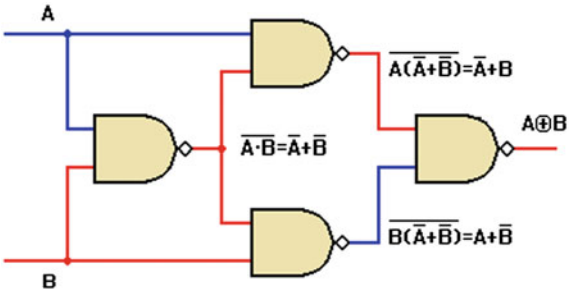


Fig. 2.13 XOR gate implementation using NAND

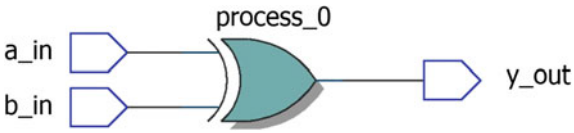


Fig. 2.14 Synthesis result for the two-input XOR logic

Synthesis result for the two-input XOR logic is shown in Fig. 2.14; input ports of XOR logic gate are named as 'a_in' and 'b_in' and output as 'y_out'.

If XOR cell or gate is not available in the library, then XOR logic is realized using AND-OR-Invert or by using minimum number of NAND gates.

XOR gate implementation using the universal gates is shown in Fig. 2.15.

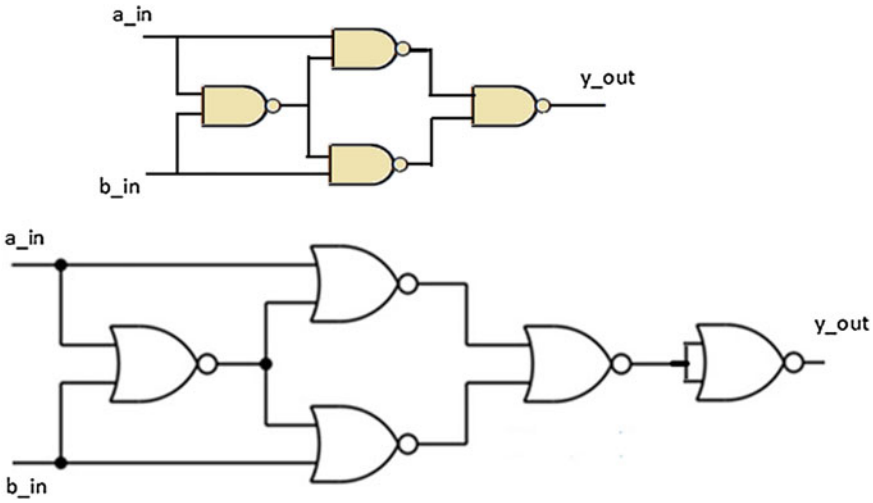


Fig. 2.15 XOR implementation using universal logic gates

2.2.7 Two-Input XNOR Logic

Two-input XNOR is called as exclusive NOR logic and generates output as logical '1' when two inputs are equal. XNOR is opposite or complement of XOR logic. Synthesizable RTL for XNOR is shown in Example 2.7. The truth table of XNOR logic is shown in Table 2.8.

Synthesis result for the XNOR logic is shown in Fig. 2.16; input ports of XNOR logic gate are named as 'a_in' and 'b_in' and output as 'y_out'.

If XNOR cell is not available in the library, then XNOR logic is realized by using Invert-AND-OR or by using minimum number of NAND or NOR gates.

The implementation of XNOR logic using universal gates is shown in Fig. 2.17.

In the practical scenario, the XOR and XNOR gates are used in the parity detection to detect for the even or odd parity. The subsequent chapter focuses on the complex designs and the synthesis. The even parity detector is shown in Fig. 2.18 and uses the XOR and XNOR gates to generate active high value at the output for even number of 1's in the input.

The odd parity checker to detect for odd number of 1's in the string is shown in Fig. 2.19. For odd number of 1's, it generates the active high output. As shown in figure, it uses three XOR gates.

```
-- VHDL RTL for two input XNOR

library ieee;
use ieee.std_logic_1164.all;

entity xnor_logic_gate is
 port( a_in: in std_logic;
       b_in: in std_logic;
       y_out: out std_logic
);
end xnor_logic_gate;

architecture arch_xnor of xnor_logic_gate is
begin

   process(a_in, b_in)
   begin

    if (a_in=b_in) then
          y_out <= '1';
    else
          y_out <= '0';
   end if;

end process;

end arch_xnor;
```

- Entity is the pin out of the design named as xnor_logic_gate that has three ports.
- Input ports are declared as 'a_in' and 'b_in'
- Output port is declared as 'y_out'

- Architecture of the xnor_logic_gate is named as arch_xnor
- Architecture describes the relationship between the inputs and output.
- The procedural block 'Process' is always sensitive to the changes specified in the sensitivity list.
- For any changes in the inputs 'a_in' or b_in the process gets invoked.
- For the inputs, a_in is equal to b_in output y_out= '1' otherwise output y_out = '0'
- For data flow model replace process block by using *y_out <= a_in xnor b_in;*

Example 2.7 Synthesizable VHDL code for XNOR logic

Table 2.8 Truth table for XNOR logic

a_in	b_in	y_out
0	0	1
0	1	0
1	0	0
1	1	1

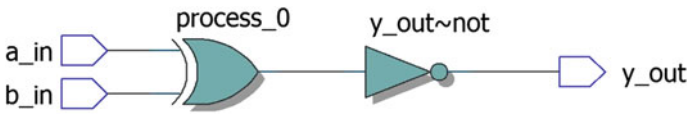


Fig. 2.16 Synthesis result for the XNOR logic

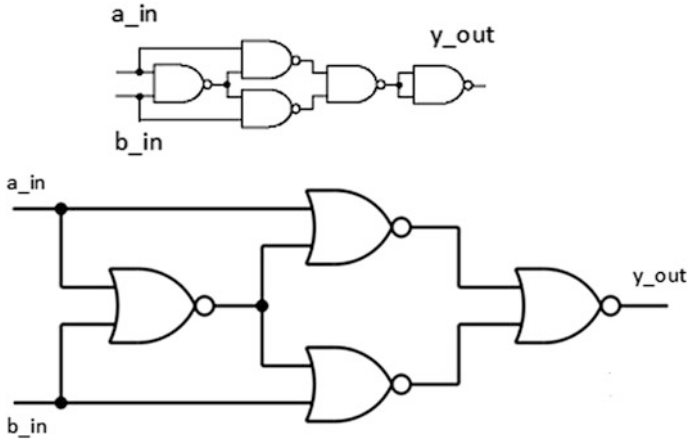


Fig. 2.17 XNOR implementation using universal logic gates

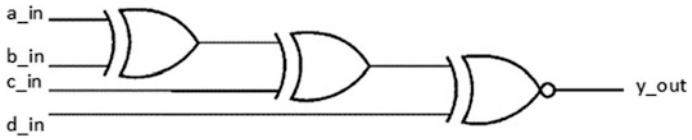


Fig. 2.18 Even-parity checker

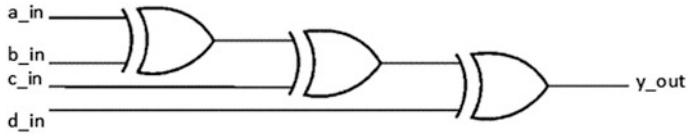


Fig. 2.19 Odd-parity checker

2.2.8 Tri-State Logic

Tri state has three logic states: logical '0', logical '1', and high impedance 'z'. Synthesizable RTL is shown in Example 2.8. The truth table of tri-state logic is shown in Table 2.9.

```
--Tri state logic

library ieee;
use ieee.std_logic_1164.all;
entity tri_state_bus is
port(  data_in:     in std_logic_vector(3 downto 0);
       enable:      in std_logic;
       data_out:    out std_logic_vector(3 downto 0)
);
end tri_state_bus;
```

- Std_logic_vector is used to generate the bus.
- Bus width is defined by using '3 downto 0' and it is 4 bit wide.

```
architecture arch_tri_state_bus of tri_state_bus is
begin
 process(data_in, enable)
 begin
       if (enable='1') then
         data_out <= data_in;
       else
         data_out <= "ZZZZ";
       end if;
 end process;

end arch_tri_state_bus;
```

- The tri state logic functionality is described by using the 'if then else construct'
- Tri state logic generates the output 'data_out' = 'data_in' for the 'enable' equal to one.
- For the 'enable' equal to zero the output of the tri-state logic is forced to be zero.

Example 2.8 Synthesizable VHDL code for tri-state bus logic

Note Avoid use of tri-state logic while developing the RTL. Tri state is difficult to test. Instead of tri-state logic, it is recommended to use multiplexers to develop the logic with enable.

Table 2.9 Truth table for tri-state logic

Enable	data_in	data_out
1	0000	0000
1	1111	1111
0	xxxx	zzzz

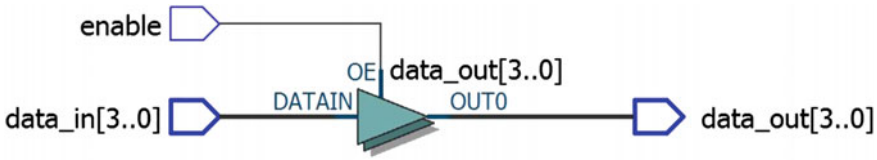


Fig. 2.20 Synthesis result for the tri-state logic

Synthesis result for the tri-state logic is shown in Fig. 2.20; input port of tri-state logic is named as 'data_in', enable input as 'enable', and output as 'data_out'.

2.3 Adder

Arithmetic operations like addition and subtraction play an important role in the efficient design of processor logic. Arithmetic and Logical Unit (ALU) of any processor is designed to perform the addition, subtraction, increment, and decrement operations. The arithmetic designs to be described by the RTL VHDL code to achieve the optimal area and to have less critical path. This section describes the important logic blocks to perform arithmetic operations with the synthesizable VHDL RTL description.

Adders are used to perform the binary addition of two binary numbers. Adders are used for signed or unsigned addition operations.

2.3.1 Half Adder

Half adder has two one-bit inputs 'a_in', 'b-in' and generates two one-bit outputs 'sum_out' and 'carry_out', where 'sum_out' is summation or addition output and 'carry_out' is carry output. Table 2.10 is the truth table for half adder, and RTL is described in Example 2.9.

Table 2.10 Truth table for half adder

a_in	b_in	sum_out	carry_out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

```
--half adder using the logical operators.

library ieee;

use ieee.std_logic_1164.all;

entity half_adder is
port( a_in: in std_logic;
      b_in: in std_logic;
      sum_out: out std_logic;
      carry_out: out std_logic
);
end half_adder;

architecture arch_half_adder of half_adder is
 begin

      sum_out <= a_in xor b_in;
      carry_out <= a_in and b_in;

end arch_half_adder;
```

- The half adder is described by using the XOR and AND logic gates.
- The XOR and AND are logic operators and generates the required 'sum_out' and 'carry_out' outputs.

Example 2.9 Synthesizable RTL code for half adder

Note Half adders are used as basic component to perform the addition. Full adder logic circuits are designed using the instantiation of half adders as components.

Synthesis result for the half adder is shown in Fig. 2.21; input ports of half adder are named as 'a_in' and 'b_in' and output as 'sum_out', 'carry_out'.

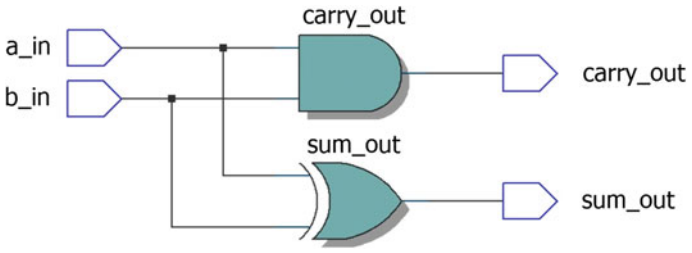


Fig. 2.21 Synthesis result for the half adder

2.3.2 Full Adder

Full adders are used to perform addition on three one-bit binary inputs.

Consider three, one-bit binary numbers named as 'a_in', 'b_in', 'c_in' and one-bit binary outputs as 'sum_out', 'carry_out'. Table 2.11 is the truth table for full adder and RTL is described in Example 2.10.

```
--full  adder using the logical operators.

library ieee;
use ieee.std_logic_1164.all;

entity full_ader is
port(  a_in: in std_logic;
       b_in: in std_logic;
       c_in : in std_logic;
       sum_out: out std_logic;
       carry_out: out std_logic
);
end full_adder;

architecture arch_full_adder of full_adder  is

signal wire1,wire2,wire3 : std_logic;

begin
        wire1 <= a_in xor b_in;
        wire2 <= a_in and b_in;
        sum_out <= c_in xor wire1;
        wire3 <= wire1 and c_in;
        carry_out <= wire2 or wire3;

end arch_full_adder;
```

- The full adder is described by using the XOR and AND logic gates.
- The XOR and AND are logic operators and generates the required 'sum_out' and 'carry_out' outputs.
- Signals are used as data objects to establish the required connectivity.

Example 2.10 Synthesizable VHDL code for full adder

Note Full adder consumes more area, so it is highly recommended to implement the adder logic using multiplexers. Subtraction can be performed by using 2's complement addition.

Table 2.11 Truth table for full adder

c_in	a_in	b_in	sum_out	carry_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

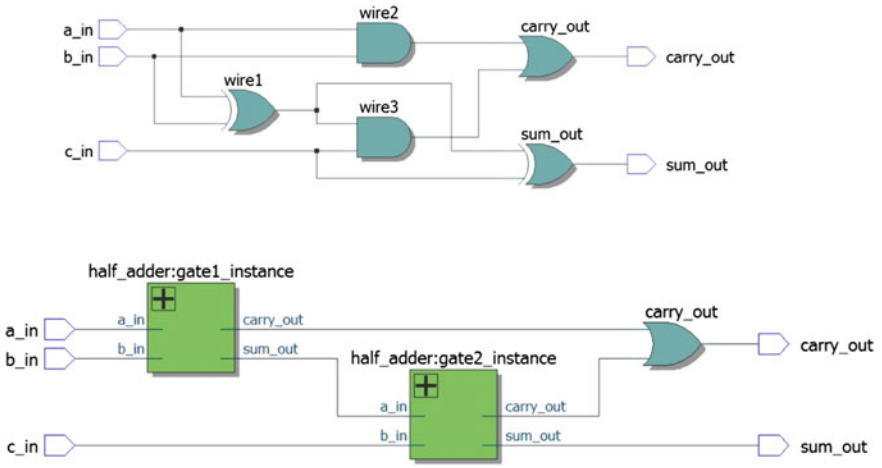


Fig. 2.22 Synthesized full adder

Synthesis result for the full adder is shown in Fig. 2.22; input ports of full adder are named as 'a_in', 'b_in', and 'c_in' and output as 'sum_out', 'carry_out'.

In the practical design scenarios, the multiplexers (MUX) can be used to implement the addition and subtraction operations. MUX is universal logic and discussed in the Chap. 4. The realization of the full adder is shown in the following figure. As shown in Fig. 2.23 by using 2:1 MUX, the logic is realized. The concept of using MUX to realize Boolean functions or logic gates is important to understand the PLD-based designs. Readers are encouraged to implement the logic of all the basic combinational elements using minimum number of 2:1 MUX.

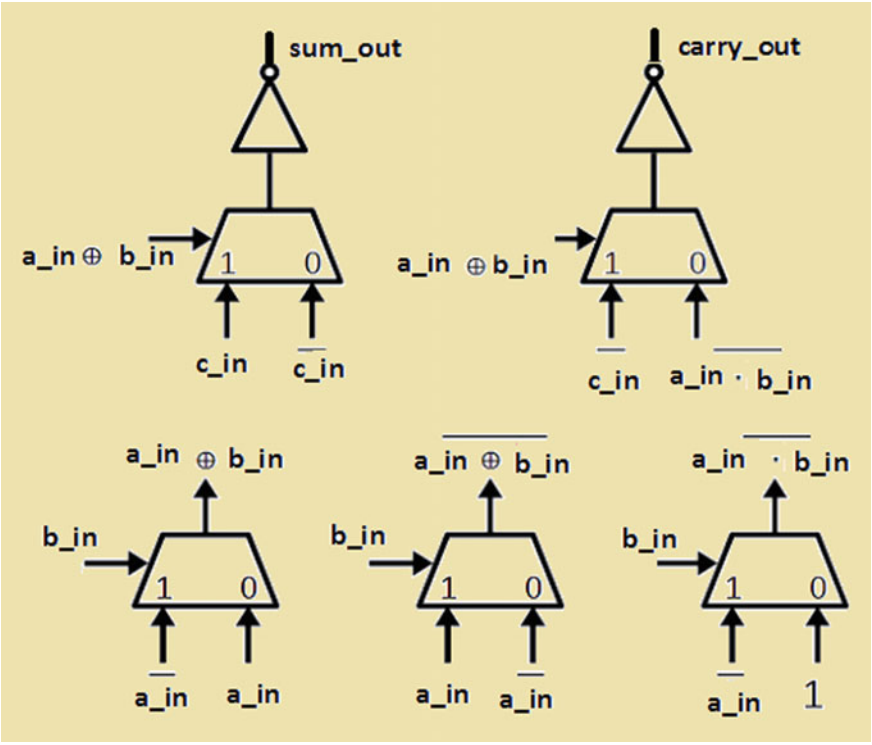


Fig. 2.23 Realization of full adder using MUX

2.4 Code Converters

This section deals with the commonly used code converters in the design. As name itself indicates, the code converters are used to convert the code from one number system to another number system. In the practical scenarios, binary-to-gray and gray-to-binary converters are used.

2.4.1 Binary-to-Gray Code Converter

Base of binary number system is 2, for any multibit binary number one or more than one bit changes at a time. In gray code, only one bit changes at a time. if we compare two successive gray codes.

The RTL description of 4-bit binary-to-gray code conversion is described in Example 2.11.

```
--Binary to Gray code converter
library ieee;
use ieee.std_logic_1164.all;
entity binary_to_gray is
port( data_in:      in std_logic_vector(3 downto 0);
      data_out: out std_logic_vector(3 downto 0)
  );
end binary_to_gray;
architecture arch_binary_to_gray of binary_to_gray is
begin
 data_out(3)<=data_in(3);
 data_out(2)<=data_in(3) xor data_in(2);
 data_out(1)<=data_in(2) xor data_in(1);
 data_out(0)<=data_in(1) xor data_in(0);
end arch_binary_to_gray;
```

- The 4 bit binary to gray converter is described by using the XOR logical operator
- 'data_in' is 4 bit binary input to the code converter
- 'data_out' is the 4 bit gray output from the code converter

Example 2.11 Synthesizable VHDL code for 4-bit binary-to-gray code converter

Note Gray codes are used in the multiple clock domain designs to transfer the control information from one of the clock domains to another clock domain.

Synthesis result for the binary to gray code converter is shown in Fig. 2.24.

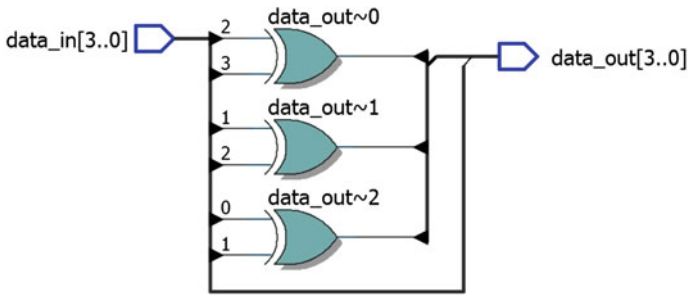


Fig. 2.24 Synthesis result for the 4-bit binary-to-gray converter

2.4.2 Gray-to-Binary Code Converter

Gray-to-binary code converter is reverse of that of binary-to-gray, and the RTL description of 4-bit gray-to-binary code conversion is described in Example 2.12.

```
-- Gray to binary code converter

library ieee;
use ieee.std_logic_1164.all;

entity gray_to_binary is
port( data_in:      in std_logic_vector(3 downto 0);
      data_out: inout std_logic_vector(3 downto 0)
  );
end gray_to_binary;

architecture arch_gray_to_binary of gray_to_binary is
begin

 data_out(3)<=data_in(3);
 data_out(2)<=data_out(3) xor data_in(2);
 data_out(1)<=data_out(2) xor data_in(1);
 data_out(0)<=data_out(1) xor data_in(0);

end arch_gray_to_binary;
```

- The 4 bit gray to binary converter is described by using the XOR logical operator
- 'data_in' is 4 bit Gray input to the code converter
- 'data_out' is the 4 bit Binary output from the code converter

Example 2.12 Synthesizable VHDL code for 4-bit gray-to-binary code converter

Note Gray codes are used in the gray counter implementation and also in the error correcting mechanism.

Synthesis result for the 4-bit gray to binary code converter is shown in Fig. 2.25.

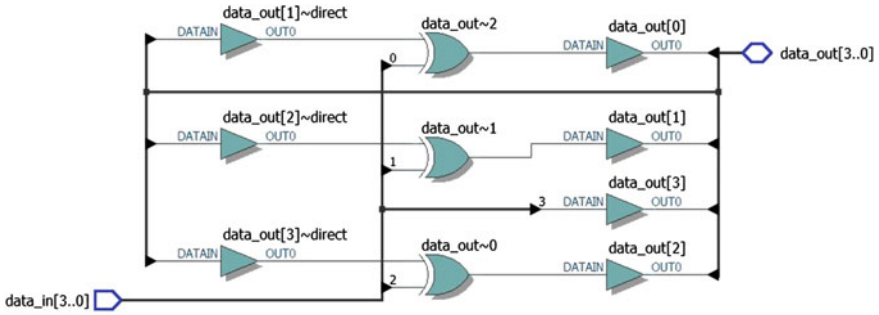


Fig. 2.25 Synthesis result for the 4-bit gray-to-binary converter

2.5 Summary

As discussed already in this chapter, following are important points need to be considered while implementing combinational logic design using VHDL.

1. Use minimum area by using least number of logic gates;
2. NAND and NOR are universal logic gates and used to implement any combinational or sequential logic;
3. Use all the required signals in the sensitivity to avoid simulation and synthesis mismatch;
4. Avoid the usage of tri-state logic and implement the logic required using multiplexers with proper enable circuit.
5. Use less number of adders in design. Adders can be implemented using multiplexers;
6. Subtraction can be implemented using 2's complement addition;
7. MUX can be used as universal logic to realize logic functions;
8. Parity can be checked by using the proper cascading of XOR and XNOR gates;
9. Gray codes are unique cyclic codes and can be used as error correcting codes.

http://www.springer.com/978-981-10-3294-3

PLD Based Design with VHDL
RTL Design, Synthesis and Implementation
Taraate, V.
2017, XXI, 423 p. 246 illus., Hardcover
ISBN: 978-981-10-3294-3