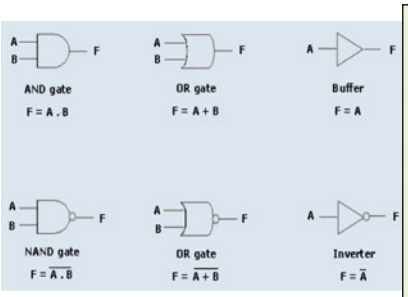
Chapter 2
Combinational Logic Design (Part I)



An efficient RTL design engineer always works on the optimal design constraints and uses minimum number of logic gates. This chapter describes about the combinational Logic design and synthesizable Verilog RTL. Also deals with the practical and real life scenarios, useful while implementing combinational designs.

Abstract This chapter describes the use of Verilog HDL to code the combinational logic design and covers the small gate count designs. The chapter is organized in such a way that it can give the practical synthesizable Verilog HDL understanding with key practical scenarios and applications. The synthesizable Verilog HDL is described for the required functionality and the synthesized logic is explained for practical understanding. This chapter is useful to build the practical expertise to code the combinational designs using synthesizable Verilog constructs.

Keywords Logic gates · NOT · AND · NAND · OR · NOR · EXOR · EXNOR · Buffer · Adder · Subtractor · Gray · Binary · Code-conversion · Blocking assignment · Continuous assignment · Procedural lock · Always · Tri state · Two's compliment

2.1 Introduction to Combinational Logic

Combinational logic is implemented by using the logic gates and in the combinational logic, output is the function of present input. The goal of a 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 hyper planes. The thought process of a designer should be such that; the

V. Taraate, *Digital Logic Design Using Verilog*,
DOI 10.1007/978-81-322-2791-5_2

design should have the optimal performance with lesser area density. The area minimization techniques have 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 Verilog. The subsequent section focuses on the use of Verilog RTL to describe the combinational design.

2.2 Logic Gates and Synthesizable RTL

This section discusses about the logic gates and the synthesizable Verilog RTL.

2.2.1 NOT or Invert Logic

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

Synthesized NOT logic is shown in the Fig. 2.1, input port of NOT logic gate is named as 'a_in' and output as 'y_out.'

2.2.2 Two-Input OR Logic

OR logic generates output as logical '1' when one of the input is logical '1.' Synthesizable RTL is shown in the Example 2.2. The truth table of OR logic is shown in the Table 2.2.

Synthesized OR logic is shown in the Fig. 2.2, input ports of OR logic gate are named as 'a_in,' 'b_in,' and output as 'y_out'.

2.2.3 Two-Input NOR Logic

NOR logic is the opposite or complement of the OR logic. Synthesizable RTL is shown in the Example 2.3. The truth table of NOR logic is shown in the Table 2.3. NOR is universal logic gate, Bubbled AND is NOR and it is DeMorgans's theorem.

Synthesized NOR logic is shown in the Fig. 2.3, input ports of NOR logic gates are named as 'a_in,' 'b_in,' and output as 'y_out.'

```
// Verilog RTL code for Not or Invert Logic

module not_logic (a_in,y_out);

input a_in;

output  y_out;

// Functionality of design

assign y_out = ~a_in;

endmodule

// Verilog RTL code for Not or Invert Logic using procedural block

module not_logic (a_in,y_out);

input a_in;

output  y_out;

reg y_out;

// Functionality of design

always@(a_in)

y_out = ~a_in;

endmodule
```

assign is used for continuous assignment to describe the combinational logic.

Continuous assignments are used to assign values to nets. Here net type is wire.

'always' block is procedural block and executes when there is an event on 'a_in' input.

An output 'y_out' is updated with the not of 'a_in' input. As 'y_out' is used inside the procedural block it is declared as 'reg'.

Example 2.1 Synthesizable Verilog code for NOT logic

Table 2.1 Truth table for NOT logic

a_in	y_out
0	1
1	0

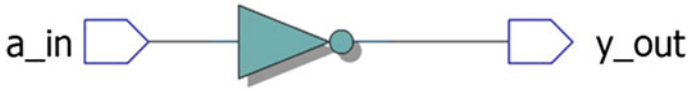


Fig. 2.1 Synthesized NOT logic

```
//verilog RTL code for the OR Logic

module or_logic (a_in, b_in, y_out);

input a_in;

input b_in;

output y_out;

reg y_out;

always@ (a_in or b_in)

begin

        if (a_in==0 && b_in==0)

            y_out = 1'b0;

        else

            y_out = 1'b1;

end

endmodule
```

Procedural block executes when there is event on either 'a_in' or 'b_in'.

Output is assigned to logic '0' when 'a_in', 'b_in' both are zero.

Output y_out is assigned to logical '1' when one of the input 'a_in' or 'b_in' is either logical one.

OR logic can be expressed using

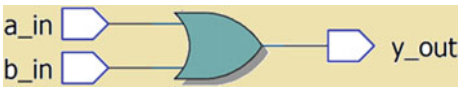
assign y_out = a_in | b_in;
where y_out is wire.

Example 2.2 Synthesizable Verilog 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 signals from sensitivity list will create simulation and synthesis mismatch and will be discussed in Chap. 3

Table 2.2 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. 2.2 Synthesized two-input OR logic



```
//verilog RTL code for the NOR Logic
module nor_logic (a_in, b_in, y_out);
input a_in;
input b_in;
output y_out;
reg y_out;
always@ (a_in or b_in)
begin
        if (a_in==0 && b_in==0)
          y_out = 1'b1;
        else
         y_out = 1'b0;
end
endmodule
```

Procedural block executes when there is event on either 'a_in' or 'b_in'.

Output is assigned to logic '1' when 'a_in', 'b_in' both are zero.

Output y_out is assigned to logical '0' when one of the input 'a_in' or 'b_in' is either logical one.

NOR logic can be expressed using :

assign y_out =~ (a_in | b_in) ; where y_out is wire.

Example 2.3 Synthesizable Verilog code for NOR logic

Table 2.3 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

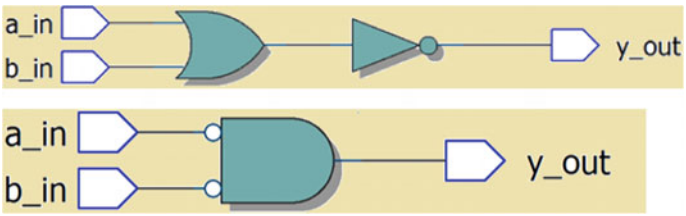


Fig. 2.3 Synthesized two-input NOR logic

2.2.4 Two-Input AND Logic

AND logic generates an output as logical '1' when both the inputs 'a,' 'b,' are logical '1.' Synthesizable RTL is shown in the Example 2.4. The truth table of AND logic is shown in the Table 2.4.

```
//verilog RTL code for the AND  Logic

module and_logic (a_in, b_in, y_out);

input a_in;

input b_in;

output y_out;

reg y_out;

 always@ (a_in or b_in)

begin

        if (a_in==1 && b_in==1)

            y_out = 1'b1;

       else

           y_out = 1'b0;

end

endmodule
```

Procedural block executes when there is event on either 'a_in' or 'b_in'.

Output is assigned to logic '1' when 'a_in', 'b_in' both are one.

Output y_out is assigned to logical '0' when one of the input 'a_in' or 'b_in' is either logical zero.

AND logic can be expressed using :

assign y_out =(a_in & b_in) ; where y_out is wire.

Example 2.4 Synthesizable Verilog code for two-input AND logic. *Note* AND gate is visualized as a series of two switches and used in programmable logic devices (PLD) as one of the element to realize the required logic. Programmable AND plane can be created using the AND logic gates as primary elements having feature as programmable inputs

Table 2.4 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

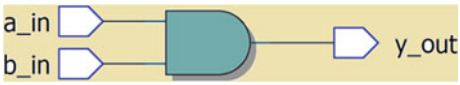


Fig. 2.4 Synthesized two-input AND logic

Synthesized two-input AND logic is shown in the Fig. 2.4, input ports of AND logic gate are named as 'a_in,' 'b_in,' and output as 'y_out.'

2.2.5 *Two-Input NAND Logic*

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

```
//verilog RTL code for the NAND  Logic

module nand_logic (a_in, b_in, y_out);

input a_in;

input b_in;

output y_out;

reg y_out;

always@ (a_in or b_in)

begin

        if (a_in==1 && b_in==1)

          y_out = 1'b0;

        else

          y_out = 1'b1;

end

endmodule
```

Procedural block executes when there is event on either 'a_in' or 'b_in'.

Output is assigned to logic '0' when 'a_in', 'b_in' both are one.

Output y_out is assigned to logical '1' when one of the input 'a_in' or 'b_in' is either logical zero.

NAND logic can be expressed using :

assign y_out =~(a_in & b_in) ; where y_out is wire.

Example 2.5 Synthesized Verilog RTL for two-input NAND Logic. *Note* NAND logic is also treated as universal logic. 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. According to DeMorgan's theorem the bubbled OR is equivalent to NAND

Table 2.5 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

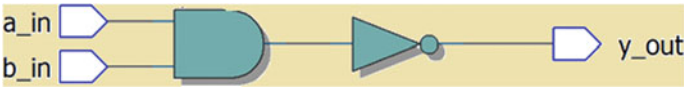


Fig. 2.5 Synthesized two-input NAND logic

Synthesized NAND logic is shown in the Fig. 2.5, input ports of NAND logic gate is named as 'a_in,' 'b_in,' and output as 'y_out.'

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 the Example 2.6. The truth table of XOR logic is shown in the Table 2.6.

Synthesized two-input XOR logic is shown in the Fig. 2.6; input ports of XOR logic gate are named as 'a_in,' '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 using minimum number of NAND gates.

2.2.7 Two-Input XNOR Logic

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

Synthesized XNOR logic is shown in the Fig. 2.7, input port s of XNOR logic gate are named as 'a_in,' 'b_in,' and output as 'y_out'.

If XNOR cell is not available in the library then XNOR logic is realized using AND-OR-Invert or using minimum number of NAND or NOR gates. Minimum five two input NAND gates are required to realize the 2 input XNOR gate.

```
//verilog RTL code for the XOR Logic

module xor_logic (a_in, b_in, y_out);

input a_in;

input b_in;

output y_out;

reg y_out;

always@ (a_in or b_in)

begin

        if (a_in!=b_in)

          y_out = 1'b1;

        else

          y_out = 1'b0;

end

endmodule
```

Procedural block executes when there is event on either 'a_in' or 'b_in'.

Output is assigned to logic '0' when 'a_in', 'b_in' both are having same logic value.

Output y_out is assigned to logical '1' when both of the inputs 'a_in' or 'b_in' are having different logic values.

XOR logic can be expressed using :

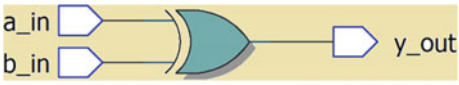
assign y_out =(a_in ^ b_in) ; where y_out is wire.

Example 2.6 Synthesizable Verilog code for two-input XOR logic. *Note* XOR gate can be implemented 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 such as addition and subtraction

Table 2.6 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. 2.6 Synthesized two-input XOR logic



```
//verilog RTL code for the XNOR Logic
module xnor_logic (a_in, b_in, y_out);
input a_in;
input b_in;
output y_out;
reg y_out;
always@ (a_in or b_in)
begin
        if (a_in!=b_in)
          y_out = 1'b0;
        else
         y_out = 1'b1;
end
endmodule
```

Procedural block executes when there is event on either 'a_in' or 'b_in'.

Output is assigned to logic '1' when 'a_in', 'b_in' both are having same logic value.

Output y_out is assigned to logical '0' when both of the inputs 'a_in' or 'b_in' are having different logic values.

XNOR logic can be expressed using :

assign y_out =~ (a_in ^ b_in) ; where y_out is wire.

Example 2.7 Synthesizable Verilog code for XNOR logic

Table 2.7 Truth table for XNOR logic

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

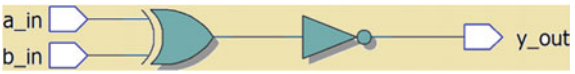


Fig. 2.7 Synthesized XNOR logic

2.2.8 Tri-state Logic

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

Synthesized tri-state logic is shown in the Fig. 2.8, input port of tri-state NOT logic is named as 'data_in,' enable input as 'enable' and output as 'data_out.'

```
// Verilog RTL code for tri-state logic

module tri_sate_logic (data_in,enable, data_out);

input [3:0] data_in;

input enable;

output [3:0]  data_out;

reg [3:0] data_out;

// Functionality of design

always@(data_in, enable)

if (enable)

data_out = data_in;

else

data_out= 4'bzzzz;

endmodule
```

'always' block is procedural block and executes when there is an event on 'data_in' input or enable.

An output 'data_out' is updated with the current value of 'data_in' for 'enable=1'. For 'enable=0' an output of tri-state logic is high impedance.

Example 2.8 Synthesizable Verilog code for tri-state 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 2.8 Truth table for tri-state logic

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

Fig. 2.8 Synthesized tri-state NOT logic

2.3 Arithmetic Circuits

Arithmetic operations such as addition and subtraction has an important role in the efficient design of processor logic. Arithmetic logic unit (ALU) of any processor can be designed to perform the addition, subtraction, increment, decrement operations. The arithmetic designs are described by the RTL Verilog code to achieve the optimal area and less critical path. This section describes the important logic blocks to perform arithmetic operations with the equivalent Verilog RTL description.

2.3.1 Adder

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

2.3.1.1 Half Adder

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

Table 2.9 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

```
//Verilog RTL code for half adder

module half_adder ( a_in, b_in, sum_out,carry_out);

input a_in;

input b_in;

output sum_out;

output carry_out;

wire sum_out;

wire carry_out;

assign sum_out = a_in ^ b_in;

assign carry_out = a_in & b_in;

endmodule
```

Output sum_out is assigned as XOR of 'a_in', 'b_in'. XOR is summation operation.

Output carry_out is assigned as AND of 'a_in', 'b_in'. Carry logic is AND operation.

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

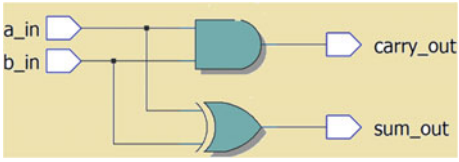


Fig. 2.9 Synthesized half adder

Synthesized half adder is shown in the Fig. 2.9, input ports of half adder are named as 'a_in,' 'b_in,' and output as 'sum_out,' 'carry_out.'

2.3.1.2 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 2.10 is the truth table for full adder and RTL is described in the Example 2.10.

Table 2.10 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

```
//Verilog RTL code for one bit full adder

module full_adder ( a_in, b_in,c_in, sum_out,carry_out);

input a_in;

input b_in;

input c_in;

output sum_out;

output carry_out;

wire sum_out;

wire carry_out;

assign {carry_out,sum_out} = a_in + b_in +c_in;

endmodule
```

'assign' is continuous assignment statement and used to assign the one bit sum output 'sum_out' and carry output 'carry_out' after performing addition operation.

Example 2.10 Synthesizable Verilog code for full adder. *Note* Full adder consumes more area so it is highly recommended to implement the adder logic using multiplexers

Synthesized full adder is shown in the Fig. 2.10, input ports of full adder are named as 'a_in,' 'b_in,' 'c_in' and output as 'sum_out' 'carry_out.'

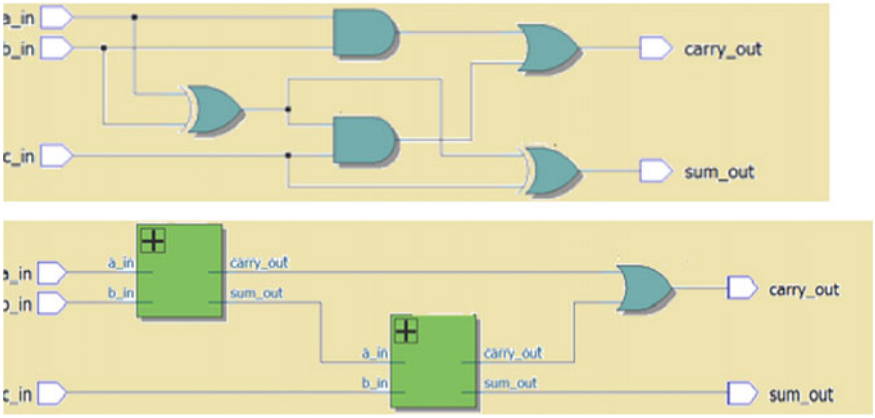


Fig. 2.10 Synthesized full adder

2.3.2 Subtractor

Subtractors are used to perform the binary subtraction of two binary numbers. This section describes about the half and full subtractors.

2.3.2.1 Half Subtractor

Half subtractor has two, one-bit inputs 'a,' 'b' and generates two one-bit outputs 'd,' 'bor'. Where 'd' is difference output and 'bor' is borrow output. Table 2.11 is the truth table for half subtractor and RTL is described in the Example 2.11.

Synthesized half subtractor is shown in the Fig. 2.11, input ports of half adder are named as 'a,' 'b,' and output as 'd,' 'bor.'

2.3.2.2 Full Subtractor

Full subtractors are used to perform subtraction of three, one-bit binary inputs. Consider three, one-bit numbers named as 'a,' 'b,' 'c' and one-bit binary outputs as 'd,' 'bor.' Table 2.12 is the truth table description for full subtractor and RTL is described in the Example 2.12 and Fig. 2.12.

Table 2.11 Truth table for half subtractor

a	b	d	bor
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

```
// Verilog RTL code for half subtractor
module half_subtractor  (a, b, d, bor) ;
input   a;
input   b;
output  d;
output  bor;
wire    d;
 wire   bor;
assign  d= a  ^  b;
assign  bor = (~a  &  b ) ;
endmodule
```

Output'd' is assigned as XOR of 'a', 'b'. XOR is used to generate output.

Output 'bor' is assigned as complement of 'a' AND 'b'

Example 2.11 Synthesizable Verilog code for half subtractor. *Note* Half subtractors are used as basic component to perform the binary subtractions. Full subtractor logic circuits are designed using the instantiation of half subtractors as components

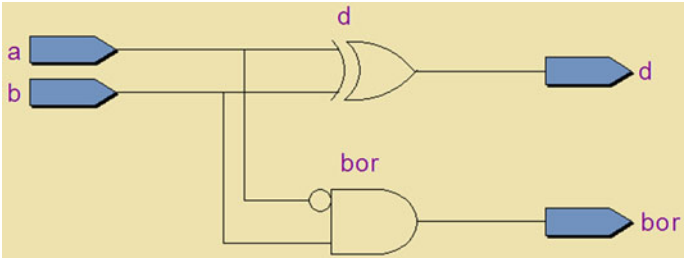


Fig. 2.11 Synthesized half subtractor

Table 2.12 Truth table for full subtractor

c	a	b	d	bor
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

```
module full_subtractor (a, b, c, d, bor) ;
input a;
input b;
input c;
output d;
output bor;
reg [1:0] temp;
always @ (a or b or c)
begin
temp = a - b - c;
end
assign d = temp[0];
assign bor = temp[1];
endmodule
```

'temp' is two bit temporary variable used to hold the data. 'temp' is declared as reg as it is used in the always block.

'In the expression subtraction of 'a', 'b', 'c' is assigned to 'temp' using binary arithmetic subtraction ('-') operator.

Example 2.12 Synthesizable Verilog code for full subtractor. *Note* It is recommended to use the full adder to perform the subtraction operation. Subtraction is performed using two's complement addition

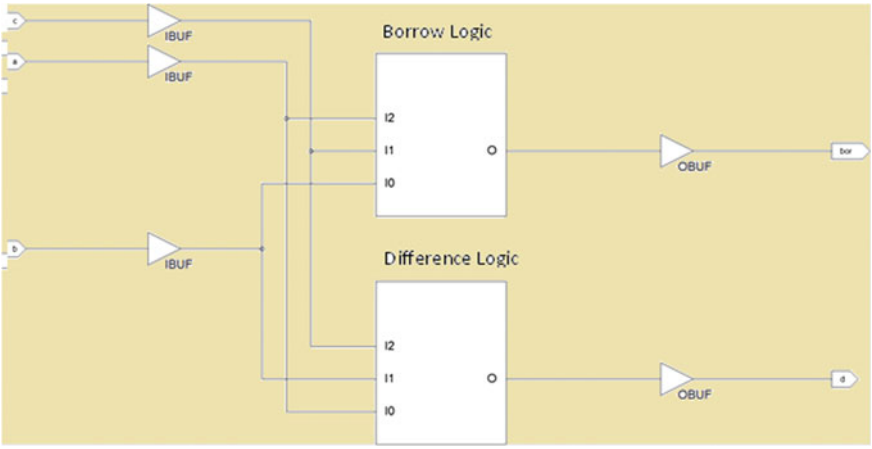


Fig. 2.12 Synthesized full subtractor

Synthesized full subtractor is shown in the Fig. 2.12 input ports of full subtractor are named as 'a,' 'b,' 'c' and output as 'd,' 'bor.'

2.3.3 Multi-bit Adders and Subtractors

Multi-bit adders and subtractors are used in the design of arithmetic units for the processors. The logic density depends upon the number of input bits of adder or subtractor.

2.3.3.1 Four-Bit Full Adder

Many practical designs use multi-bit adders and subtractors. It is the industrial practice to use basic component as full adder to perform the addition operation. For example, if designer needs to implement the four-bit design logic of an adder, then four full adders are required. As shown in the Example 2.13, addition is performed on two, four-bit binary numbers 'A,' 'B.' The final result is four-bit addition and output at 'S.' Carry input is Ci and carry output is Co.

Synthesized four-bit adder is shown in the Fig. 2.13, input ports of four-bit adder are named as 'A,' 'B,' 'Ci,' and output as 'S,' 'Co.'

```
// Verilog RTL code for 4 bit full adder
module adder_4bit (A,B,Ci,S,Co) ;
parameter data_size =4;
input [data_size-1:0] A;
input [data_size-1:0] B;
input Ci;
output [data_size-1:0 ]S;
output Co;
// Functionality of design
assign {Co, S} = A + B + Ci;
endmodule
```

parameter is used to define the data_size according to the given design specifications.

assign is used for assigning the carry output 'Co' and 4-bit sum output 'S' after addition.

Example 2.13 Synthesizable Verilog code for four-bit adder. *Note* Four-bit addition operation uses four full adders. Depending on signed or unsigned addition requirements the Verilog code can be modified

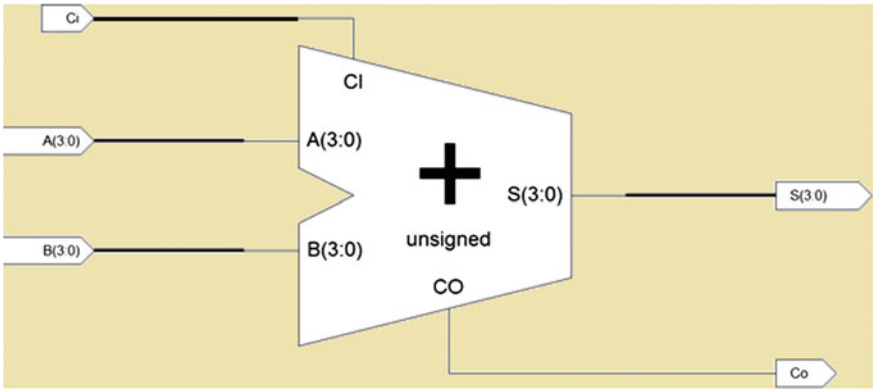


Fig. 2.13 Synthesized four-bit adder

2.3.3.2 Four-Bit Adder and Subtractor

Design of addition and subtraction can be accomplished using the adders only. Subtraction can be performed using two's complement addition. For example consider the scenario shown in the Table 2.13.

Synthesized four-bit adder/subtractor is shown in the Fig. 2.14, for Example 2.14, input ports of four-bit adder/subtractor are named as 'A,' 'B,' 'Ci,' and output as 'S,' 'Co.' When control input SUB is equal to logic '0' then it performs the addition and for control input SUB is equal to logic '1' it performs the subtraction which is 2's complement addition.

Table 2.13 Operational table for adder subtractor

Operation	Description	Expression
Addition	Unsigned addition of A, B	A + B + 0
Subtraction	Unsigned subtraction of A, B	A – B = A + ∼B + 1

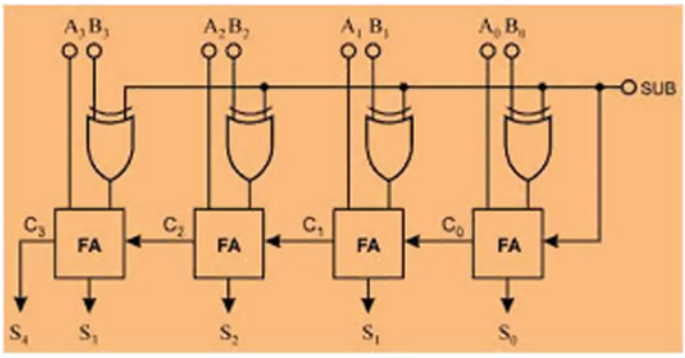


Fig. 2.14 Synthesized four-bit adder/subtractor

```
// Verilog RTL code for 4 bit full adder subtractor
module adder_subtractor_4bit (A,B,Ci,S,Co) ;
parameter data_size =4;
input [data_size-1:0] A;
input [data_size-1:0] B;
input Ci;
output [data_size-1:0 ] S;
output Co;
reg [data_size-1:0 ] S;
reg Co;
// Functionality of design
always @ ( A or B or Ci)
if (Ci)
 {Co,S}= A – B;
else
{Co,S} =A + B;
endmodule
```

parameter is used to define the data_size according for the given design specifications.

For Ci='1' S= A-B else

S=A+B

Example 2.14 Synthesizable Verilog code for four-bit adder and subtractor. *Note* Consider SUB control,input as Ci and S4 as Co in the synthesized logic. Here, the resource used is binary full adder to perform both the additions and subtractions. Subtraction operation is performed using adders only. Resource sharing and resource utilization are to be discussed in the Chap. 3

2.3.4 Comparators and Parity Detectors

In most of the practical scenarios; comparators are used to compare the equality of two binary numbers. Parity detectors are used to compute the even or odd parity for the given binary number. It becomes very essential for the design engineer to have the better understanding of this.

2.3.4.1 Binary Comparators

These are used to compare the two binary numbers. As discussed earlier Verilog supports four value logic and they are logical '0,' logical '1,' don't care 'x' and high impedance 'z.' Verilog supports logical equality operator (==) and inequality

Table 2.14 Operational table for comparator

Condition	Description	Verilog expression
A==B	Assign output as XOR of A, B	A ^ B
A!=B	Assign output as AND of A, B	A & B

operator (!=), and these are used to describe the comparison of two numbers. These operators are used in the Verilog Synthesizable RTL code.

For example consider the operational Table 2.14. As shown in the table; when A, B both are equal then output 'Y' is assigned to XOR of 'A,' 'B' and for unequal case output 'Y' is assigned to AND of 'A,' 'B' (Example 2.15).

Synthesized equivalent block representation is shown in the Fig. 2.15 (Example 2.15).

```
// Verilog RTL code for 1-bit comparator
module comparator (A,B,Y) ;
input A;
input B;
output Y;
reg Y;
// Functionality of design
always @ ( A or B)
begin
   if ( A==B)
     Y = A ^ B;
  else
     Y = A & B;
end
endmodule
```

Equality operator '==' is used for comparison. For 'x' or 'z' inputs the result of logical comparison is always false. If 'A' or 'B' becomes either 'x' or 'z' then the result will be A & B as

Example 2.15 Synthesizable Verilog code for 1-bit comparator. *Note* Logical equality and inequality operators are used in the synthesizable RTL code and for any of the operands are 'x' or 'z' comparison is false

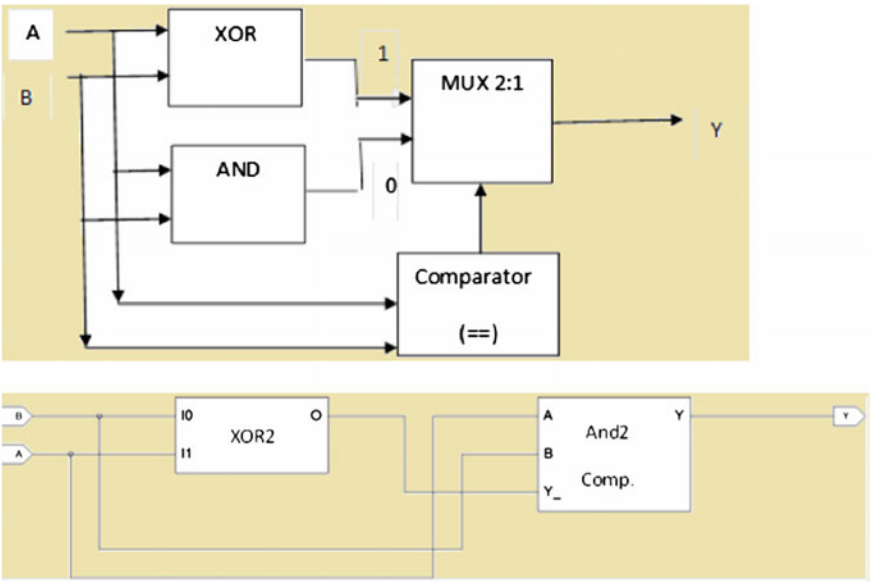


Fig. 2.15 Synthesized equality comparator

2.3.4.2 Parity Detector

Parity detectors are used to detect the even or odd parity for the binary number string. For even number of 1's, the output required is logical '0' and for odd number of 1's the output required is logical '1,' then the RTL Verilog can be described as shown in the Example 2.16.

```
// Verilog RTL code ffor parity detector
module parity_logic   (A, P) ;
input  [3:0] A;
output P;
wire P;
// Functionality of design
assign  P = ( A[3] ^ A[2] ^ A[1] ^ A[0] );
endmodule
```

XOR (^) operator is used to detect the parity . The assign statement assigns the output 'P' to logical '1' for odd number of 1's and for even number of 1's ouput is equal to logical '0'

Example 2.16 Synthesizable Verilog code for parity detector. *Note* Parity detectors are used in many of the DSP applications and an integral module for encryption engines

Table 2.15 Operational table for parity detector

Condition	Description
Odd 1's	Assign output as logical 1
Even 1's	Assign output as logical zero

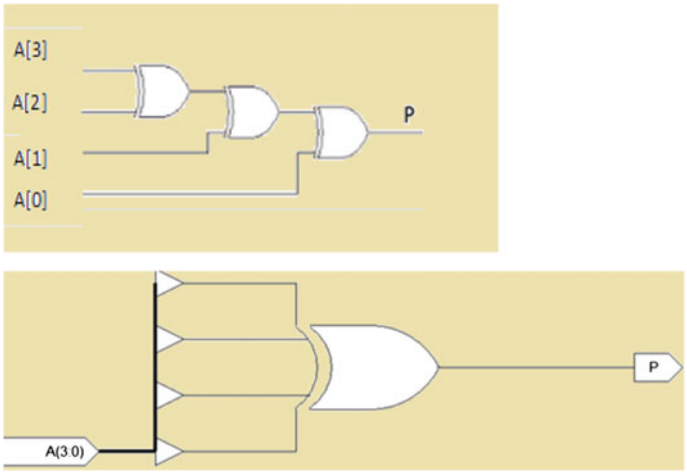


Fig. 2.16 Synthesized parity detector

The operational table for the parity detector is shown below in Table 2.15. For odd number of 1's the output is logical '1' and for even number of 1's output is assigned as logical '0'.

Synthesized equivalent block representation is shown in the Fig. 2.16.

2.3.5 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.

2.3.5.1 Binary to Gray Code Converter

Base of binary number system is 2, for any multi-bit binary number one or more than one bit changes at a time. In gray code, only one bit changes at a time.

The RTL description of four-bit binary to gray code conversion is described in Example 2.17.

```
// Verilog RTL code for 4-bit binary to Gray converterr

module binary_to_gray   (B, G) ;

input  [3:0] B;

output [3:0] G;

// Functionality of design

assign  G[3] = B[3];

assign  G[2] = B[3] ^ B[2];

assign  G[1] = B[2] ^ B[1];

assign G[0] = B[1] ^ B[0];

endmodule
```

Example 2.17 Synthesizable Verilog code for four-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 domain to another clock domain

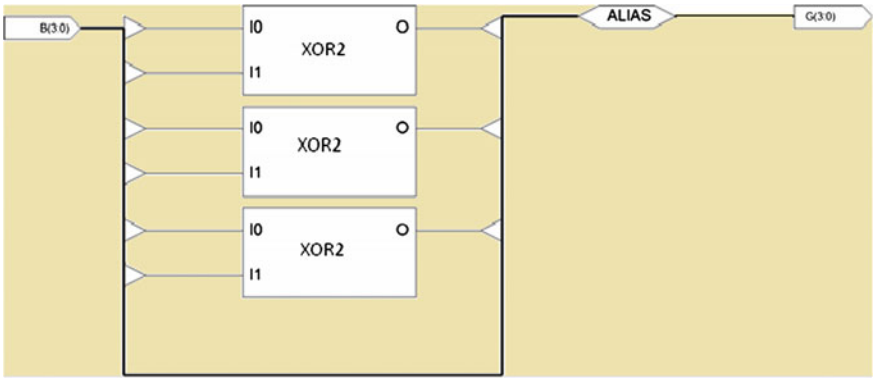


Fig. 2.17 Synthesized four-bit binary to gray converter

Synthesized equivalent block representation is shown in Fig. 2.17.

2.3.5.2 Gray to Binary Code Converter

Gray to binary code converter is opposite of that of binary to gray and the RTL description of four-bit gray to binary code conversion described in Example 2.18.

Synthesized equivalent block representation is shown the Fig. 2.18.

```
// Verilog RTL code for 4-bit gray to binary  converter
module  gray_to_binary    (G, B) ;
input  [3:0] G;
output [3:0] B;
// Functionality of design
assign  B[3] = G[3];
assign  B[2] = G[3] ^ G[2];
assign  B[1] = G[3] ^ G[2] ^ G[1];
assign B[0] = G[3] ^ G[2] ^ G[1] ^ G[0];
endmodule
```

Gray to Binary code uses the XOR operator to realize the equivalent logic.

Example 2.18 Synthesizable Verilog code for four-bit gray to binary code converter. *Note* Gray codes are used in the gray counter implementation and also in the error correcting mechanism

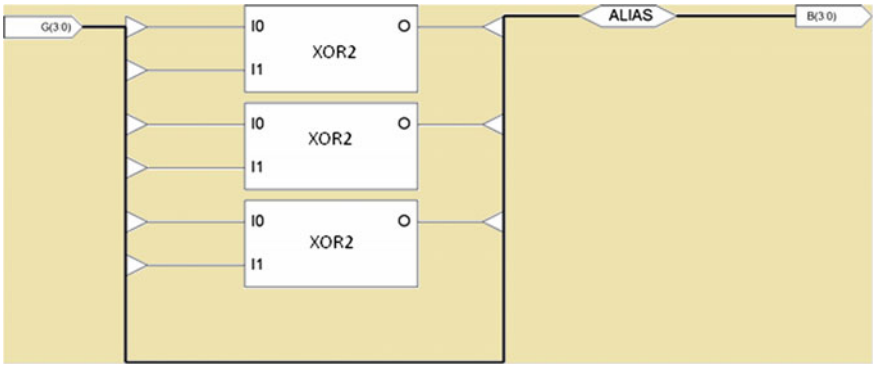


Fig. 2.18 Synthesized four-bit gray to binary converter

2.4 Summary

As discussed already in this chapter; following are the important points need to be considered while implementing combinational logic RTL.

1. Use minimum area by sharing the arithmetic resources.
2. Use all the required signals in the sensitivity list to avoid simulation and synthesis mismatch.
3. Avoid use of tri-state logic and implement the logic required using multiplexers with proper enable circuit.

4. Verilog supports four value logic and they are logical ‘0,’ logical ‘1,’ don’t care ‘x,’ high impedance ‘z.’

   - Use less number of adders in design. Adders can be implemented using multiplexers.
   - NAND and NOR are universal logic gates and used to implement any combinational or sequential logic.

http://www.springer.com/978-81-322-2789-2

Digital Logic Design Using Verilog
Coding and RTL Synthesis
Taraate, V.
2016, XXIII, 416 p. 267 illus., 226 illus. in color.,
Hardcover
ISBN: 978-81-322-2789-2