DIGITAL DESIGN

ASSIGNMENT REPORT

ASSIGNMENT ID : 1

Student Name: 林洁芳

Student ID: 12011543

PART 1: DIGITAL DESIGN THEORY

Provide answers to the following questions:

1. What is the exact number of bytes in a system that contains (a) 16K bytes, (b) 32M bytes, and (c) 3.2G bytes?
   (a) 16K bytes = 16 * 1024 bytes = 16,384 bytes;
   (b) 32M bytes = 32 * 1024 * 1024 bytes = 33,554,432 bytes;
   (c) 3.2G bytes = 3.2 * 1024* 1024* 1024 = 3,435,973,837 bytes

2. What is the largest binary number that can be expressed with 12 bits? What are the equivalent decimal and hexadecimal numbers?
   (a) $111111111111_{(2)}$
   (b) $111111111111_{(2)}$ = $4095_{(10)}$ = 0xFFF

3. Convert the decimal number 248 to binary in two ways: (a) convert directly to binary; (b) convert first to hexadecimal and then from hexadecimal to binary. Which method is faster?
   (a) 248 = $11111000_{(2)}$
   (b) 248 = 0xF8 = 1111 $1000_{(2)}$
   The second method is faster.

4. Find the 9's and the 10's complement code of the following **decimal** numbers.
   a. 25273036
   b. 64322610
   (a) 9's complement : 74726963
   10's complement : 74726964
   (b) 9's complement : 35677389
   10's complement : 35677390

5. (a) Find the 16's complement of C6BF.
   (b) Convert C6BF to binary.
   (c) Find the 2's complement of the result in (b).
   (d) Convert the answer in (c) to hexadecimal, compare with the answer in (a).
   (a) 3941
   (b) 1100 0110 1011 1111
   (c) 0011 1001 0100 0001
   (d) 0011 1001 0100 $0001_{(2)}$ = 0x3941 (They are the same.)

6. Do the following conversion problems:
   (a) Convert decimal 23.5625 to binary.
   (b) Calculate the binary equivalent of 5/3 out to eight places. Then convert from binary to decimal. How close is the result to 5/3?
   (c) Convert the binary result in (b) into hexadecimal. Then convert the result to decimal. Is the answer the same? Explain why.
   (a) 10111.1001
   (b) 5/3 = 1.10101010 $_{(2)}$
   1.10101010 $_{(2)}$ = 1.6640625
   |1.6666666 - 1. 6640625| = 0.0026041
   0.0026041 close is the result to 5/3. (the error is 0.39%)
   (c) 0001.1010 1010 = 0x1.AA
   0x1.AA = 1.6640625
   The answers are the same.
   Because all base conversions have the same result when representing the same digit.

7. The state of a 12-bit register is 1001 0111 0101. What is its content if it represents
   (a) Three decimal digits in BCD?
   (b) Three decimal digits in the excess-3 code?
   (c) Three decimal digits in the 8,4,-2,-1 code?
   (d) Three decimal digits in the 6311 code?
   (e) A binary number?
   (a) 975
   (b) 642
   (c) 713
   (d) 754
   (e) 2421

8. We can perform logical operations on strings of bits by considering each pair of corresponding bits separately (called **bitwise operation**). Given two eight-bit strings A = 11011010 and B = 01001110, evaluate the eight-bit result after the following bitwise logical operations: (a) AND (b) OR (c) XOR (d) NOT A (e) NOT B (f) NAND (g) NOR. **(Please represent all the 7 results in hexadecimal)**
   (a) 0100 1010 = 0x4A
   (b) 1101 1110 = 0xDE
   (c) 1001 0100 = 0x94
   (d) 0010 0101 = 0x25

(e) 1011 0001 = 0xB1
(f) 1011 0101 = 0xB5
(g) 0010 0001 = 0x21

PART 2: DIGITAL DESIGN LAB

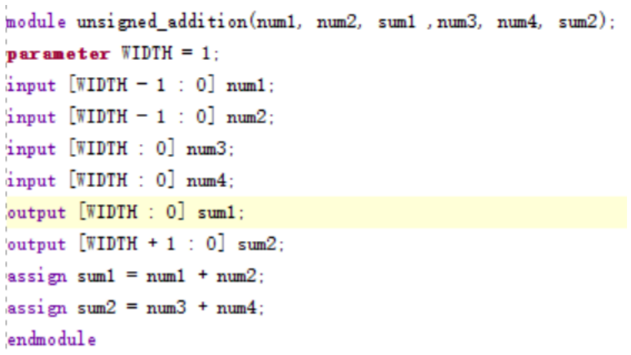
TASK1:

Create a project named as UnsignedAddition:

DESIGN

Verilog code:

```
module unsigned_addition (num1, num2, sum1, num3, num4, sum2);
        parameter WIDTH = 1;
        input [WIDTH - 1 : 0] num1;
        input [WIDTH - 1 : 0] num2;
        input [WIDTH : 0] num3;
        input [WIDTH : 0] num4;
        output [WIDTH : 0] sum1;
        output [WIDTH + 1 : 0] sum2;
        assign sum1 = num1 + num2;
        assign sum2 = num3 + num4;
endmodule
```



```
module unsigned_addition(num1, num2, sum1 ,num3, num4, sum2);
parameter WIDTH = 1;
input [WIDTH - 1 : 0] num1;
input [WIDTH - 1 : 0] num2;
input [WIDTH : 0] num3;
input [WIDTH : 0] num4;
output [WIDTH : 0] sum1;
output [WIDTH + 1 : 0] sum2;
assign sum1 = num1 + num2;
assign sum2 = num3 + num4;
endmodule
```

truth-table:

num1	0	0	1	1
num2	0	1	0	1
sum1	00	01	01	10

num3	num4	sum2
00	00	000

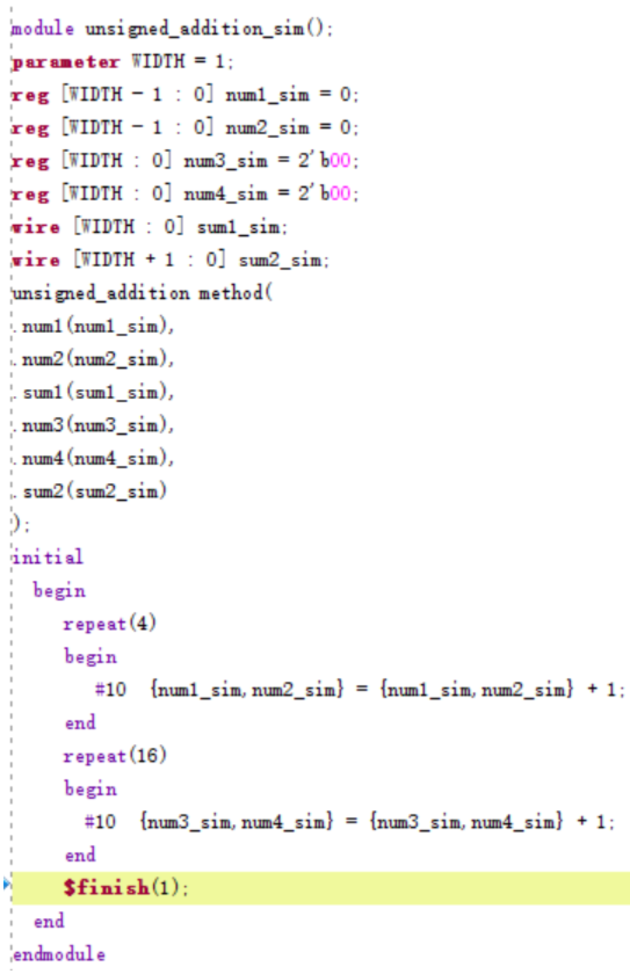
01	00	001
10	00	010
11	00	011
00	01	001
01	01	010
10	01	011
11	01	100
00	10	010
01	10	011
10	10	100
11	10	101
00	11	011
01	11	100
10	11	101
11	11	110

SIMULATION

Verilog code:

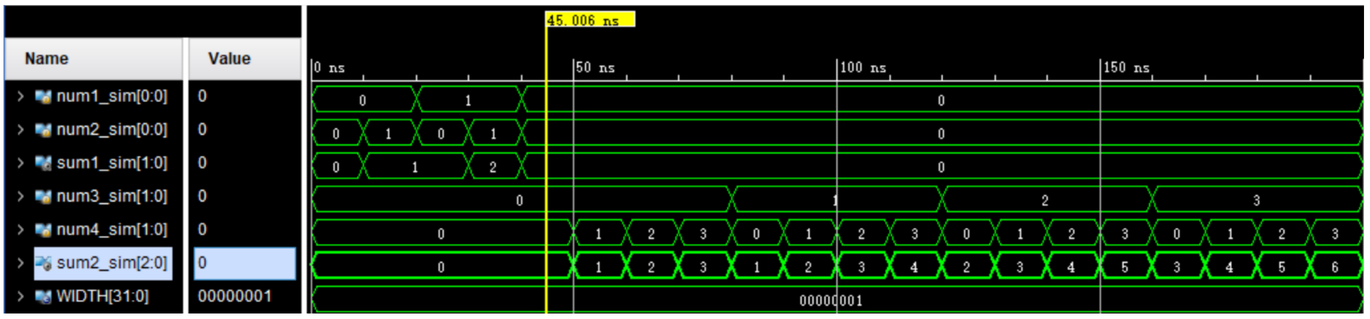
```
module unsigned_addition_sim();
parameter WIDTH = 1;
reg [WIDTH - 1 : 0] num1_sim = 0;
reg [WIDTH - 1 : 0] num2_sim = 0;
reg [WIDTH : 0] num3_sim = 2'b00;
reg [WIDTH : 0] num4_sim = 2'b00;
wire [WIDTH : 0] sum1_sim;
wire [WIDTH + 1 : 0] sum2_sim;
unsigned_addition method(
.num1(num1_sim),
.num2(num2_sim),
.sum1(sum1_sim),
.num3(num3_sim),
.num4(num4_sim),
.sum2(sum2_sim)
);
initial
  begin
    repeat(4)
    begin
      #10 {num1_sim,num2_sim} = {num1_sim,num2_sim} + 1;
    end
    repeat(16)
    begin
     #10 {num3_sim,num4_sim} = {num3_sim,num4_sim} + 1;
    end
```

```
            $finish(1);
        end
endmodule
```



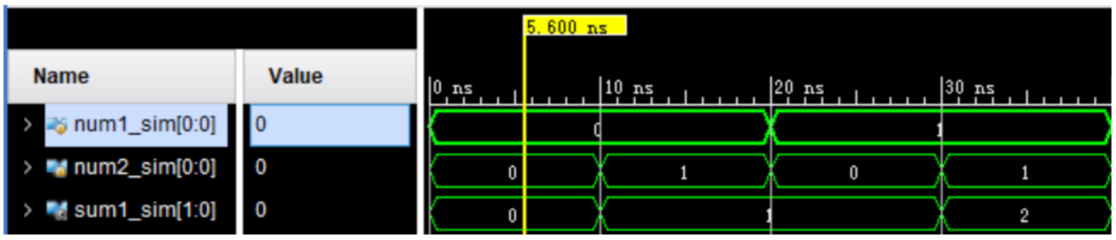
```
module unsigned_addition_sim();
parameter WIDTH = 1;
reg [WIDTH - 1 : 0] num1_sim = 0;
reg [WIDTH - 1 : 0] num2_sim = 0;
reg [WIDTH : 0] num3_sim = 2'b00;
reg [WIDTH : 0] num4_sim = 2'b00;
wire [WIDTH : 0] sum1_sim;
wire [WIDTH + 1 : 0] sum2_sim;
unsigned_addition method(
.num1(num1_sim),
.num2(num2_sim),
.sum1(sum1_sim),
.num3(num3_sim),
.num4(num4_sim),
.sum2(sum2_sim)
);
initial
  begin
     repeat(4)
     begin
        #10  {num1_sim,num2_sim} = {num1_sim,num2_sim} + 1;
     end
     repeat(16)
     begin
       #10  {num3_sim,num4_sim} = {num3_sim,num4_sim} + 1;
     end
     $finish(1);
  end
endmodule
```

simulation result:

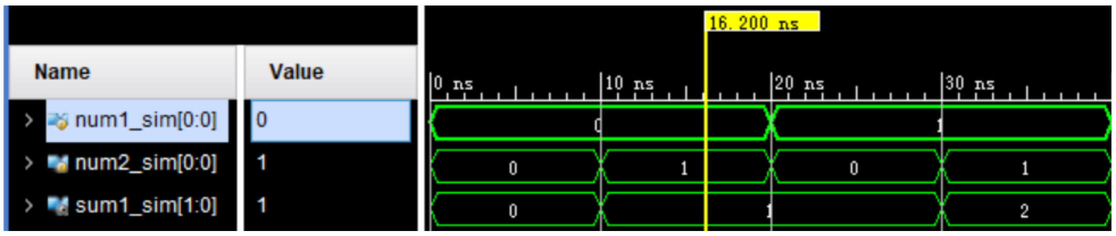


1 bit input :

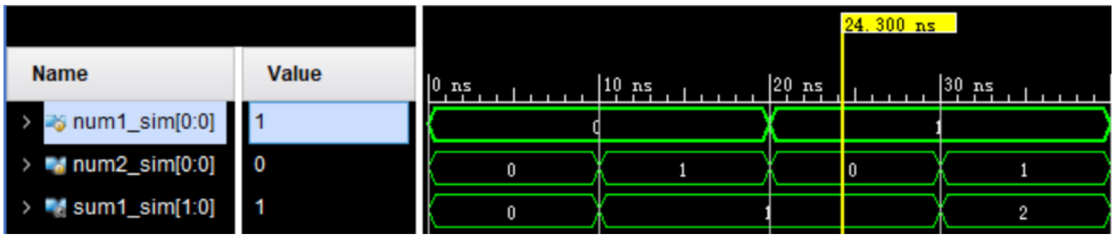
（1） num1 = 0； num2 = 0; then sum1 = 00; (binary to decimal: 0 + 0 = 0)



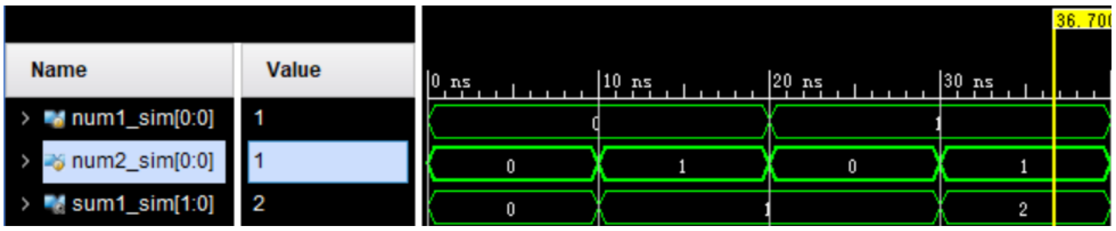
（2） num1 = 0； num2 = 1; then sum1 = 01; (binary to decimal: 0 + 1 = 1)



（3） num1 = 1； num2 = 0; then sum1 = 01; (binary to decimal: 1 + 0 = 1)

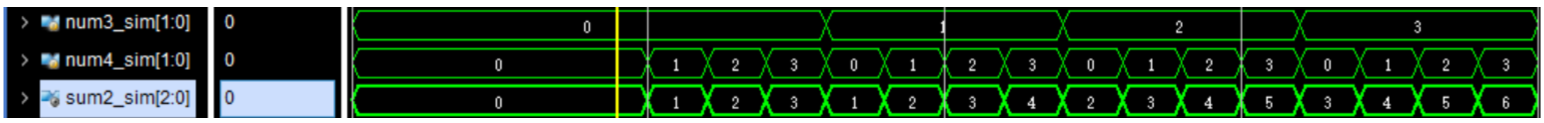


（4） num1 = 1； num2 = 1; then sum1 = 10; (binary to decimal: 1 + 1 = 2)



2 bits input :

The data from left to right is:



（1） num3 = 00； num4 = 00; then sum4 = 000;

(binary to decimal: 0 + 0 = 0)

（2） num3 = 00； num4 = 01; then sum4 = 001;

(binary to decimal: 0 + 1 = 1)

（3） num3 = 00； num4 = 10; then sum4 = 010;

(binary to decimal: 0 + 2 = 2)

（4） num3 = 00； num4 =11; then sum4 = 011;

(binary to decimal: 0 + 3 = 3)

(5) num3 = 01； num4 = 00; then sum4 = 001;

(binary to decimal: 1 + 0 = 1)

(6) num3 = 01； num4 = 01; then sum4 = 010;

(binary to decimal: 1 + 1 = 2)

(7) num3 = 01； num4 = 10; then sum4 = 011;

(binary to decimal: 1 + 2 = 3)

(8) num3 = 01； num4 = 11; then sum4 = 100;

(binary to decimal: 1 + 3 = 4)

(9) num3 = 10； num4 = 00; then sum4 = 010;

(binary to decimal: 2 + 0 = 2)

(10) num3 = 10； num4 = 01; then sum4 = 011;

(binary to decimal: 2 + 1 = 3)

(11) num3 = 10； num4 = 10; then sum4 = 100;

(binary to decimal: 2 + 2 = 4)

(12) num3 = 10； num4 = 11; then sum4 = 101;

(binary to decimal: 2 + 3 = 5)

(13) num3 = 11； num4 = 00; then sum4 = 00;

(binary to decimal: 3 + 0 = 3)

(14) num3 = 11； num4 = 01; then sum4 = 00;

(binary to decimal: 3 + 1 = 4)

(15) num3 = 11； num4 = 10; then sum4 = 00;

(binary to decimal: 3 + 2 = 5)

(16) num3 = 11； num4 = 11; then sum4 = 00;

(binary to decimal: 3 + 3 = 6)

THE DESCRIPTION OF OPERATION

problem and solution:

If the input numbers are 2 bits, the cases are so much to list line by line, so there is a good method to use “ repeat(n) ”;
If you use the word “repeat” , it can change each 10ns and can cover all the test cases;

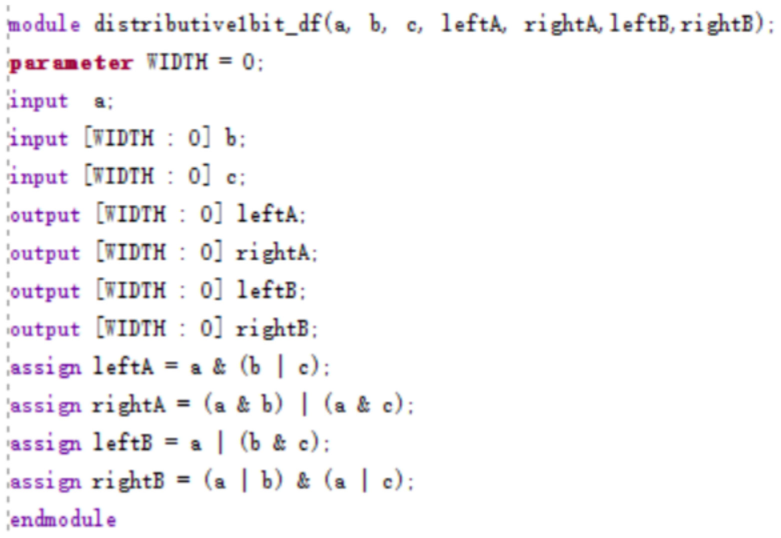
TASK2:

DESIGN

Verilog code:

(1) distributive1bit_df.v

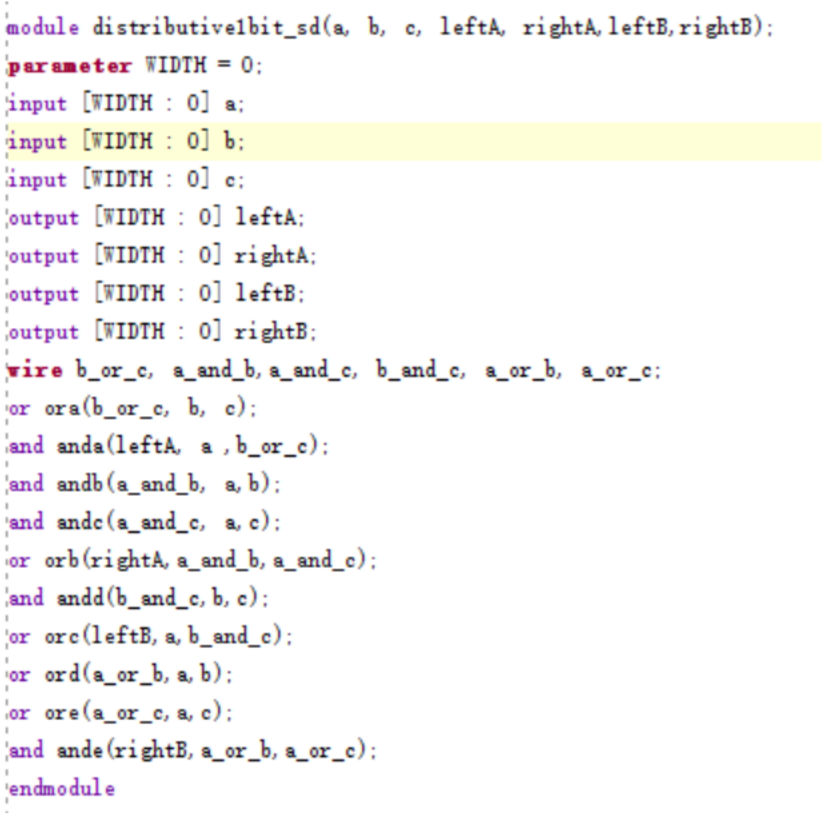
```
module distributive1bit_df(a, b, c, leftA, rightA,leftB,rightB);
parameter WIDTH = 0;
input  a;
input [WIDTH : 0] b;
input [WIDTH : 0] c;
output [WIDTH : 0] leftA;
output [WIDTH : 0] rightA;
output [WIDTH : 0] leftB;
output [WIDTH : 0] rightB;
assign leftA = a & (b | c);
assign rightA = (a & b) | (a & c);
assign leftB = a | (b & c);
assign rightB = (a | b) & (a | c);
endmodule
```



```
module distributive1bit_df(a, b, c, leftA, rightA,leftB,rightB);
parameter WIDTH = 0;
input  a;
input [WIDTH : 0] b;
input [WIDTH : 0] c;
output [WIDTH : 0] leftA;
output [WIDTH : 0] rightA;
output [WIDTH : 0] leftB;
output [WIDTH : 0] rightB;
assign leftA = a & (b | c);
assign rightA = (a & b) | (a & c);
assign leftB = a | (b & c);
assign rightB = (a | b) & (a | c);
endmodule
```

(2) distributive1bit_sd.v

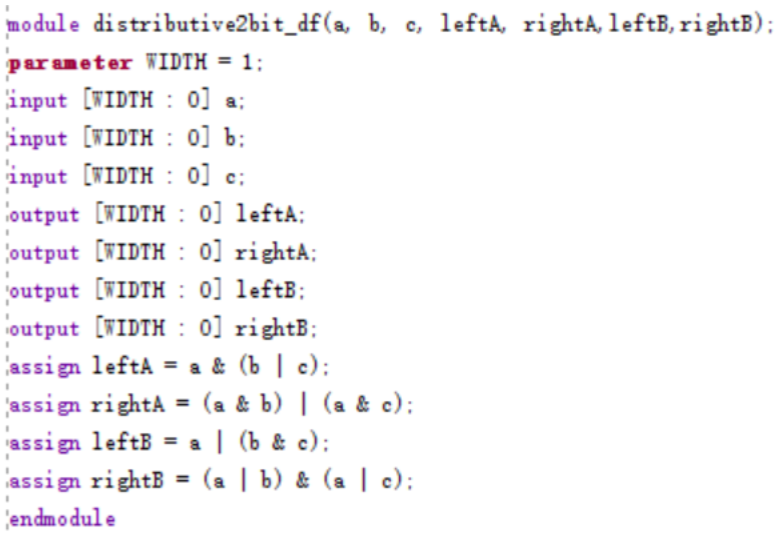
```
module distributive1bit_sd(a, b, c, leftA, rightA,leftB,rightB);
parameter WIDTH = 0;
input [WIDTH : 0] a;
input [WIDTH : 0] b;
input [WIDTH : 0] c;
output [WIDTH : 0] leftA;
output [WIDTH : 0] rightA;
output [WIDTH : 0] leftB;
output [WIDTH : 0] rightB;
wire b_or_c, a_and_b,a_and_c, b_and_c, a_or_b, a_or_c;
or ora(b_or_c, b, c);
and anda(leftA, a ,b_or_c);
and andb(a_and_b, a,b);
and andc(a_and_c, a,c);
or orb(rightA,a_and_b,a_and_c);
and andd(b_and_c,b,c);
or orc(leftB,a,b_and_c);
or ord(a_or_b,a,b);
or ore(a_or_c,a,c);
and ande(rightB,a_or_b,a_or_c);
endmodule
```



```
module distributive1bit_sd(a, b, c, leftA, rightA,leftB,rightB);
parameter WIDTH = 0;
input [WIDTH : 0] a;
input [WIDTH : 0] b;
input [WIDTH : 0] c;
output [WIDTH : 0] leftA;
output [WIDTH : 0] rightA;
output [WIDTH : 0] leftB;
output [WIDTH : 0] rightB;
wire b_or_c, a_and_b,a_and_c, b_and_c, a_or_b, a_or_c;
or ora(b_or_c, b, c);
and anda(leftA, a ,b_or_c);
and andb(a_and_b, a,b);
and andc(a_and_c, a,c);
or orb(rightA,a_and_b,a_and_c);
and andd(b_and_c,b,c);
or orc(leftB,a,b_and_c);
or ord(a_or_b,a,b);
or ore(a_or_c,a,c);
and ande(rightB,a_or_b,a_or_c);
endmodule
```

(3) distributive2bit_df.v

```
module distributive2bit_df(a, b, c, leftA,
rightA,leftB,rightB);
parameter WIDTH = 1;
input [WIDTH : 0] a;
input [WIDTH : 0] b;
input [WIDTH : 0] c;
output [WIDTH : 0] leftA;
output [WIDTH : 0] rightA;
output [WIDTH : 0] leftB;
output [WIDTH : 0] rightB;
assign leftA = a & (b | c);
assign rightA = (a & b) | (a & c);
assign leftB = a | (b & c);
assign rightB = (a | b) & (a | c);
endmodule
```

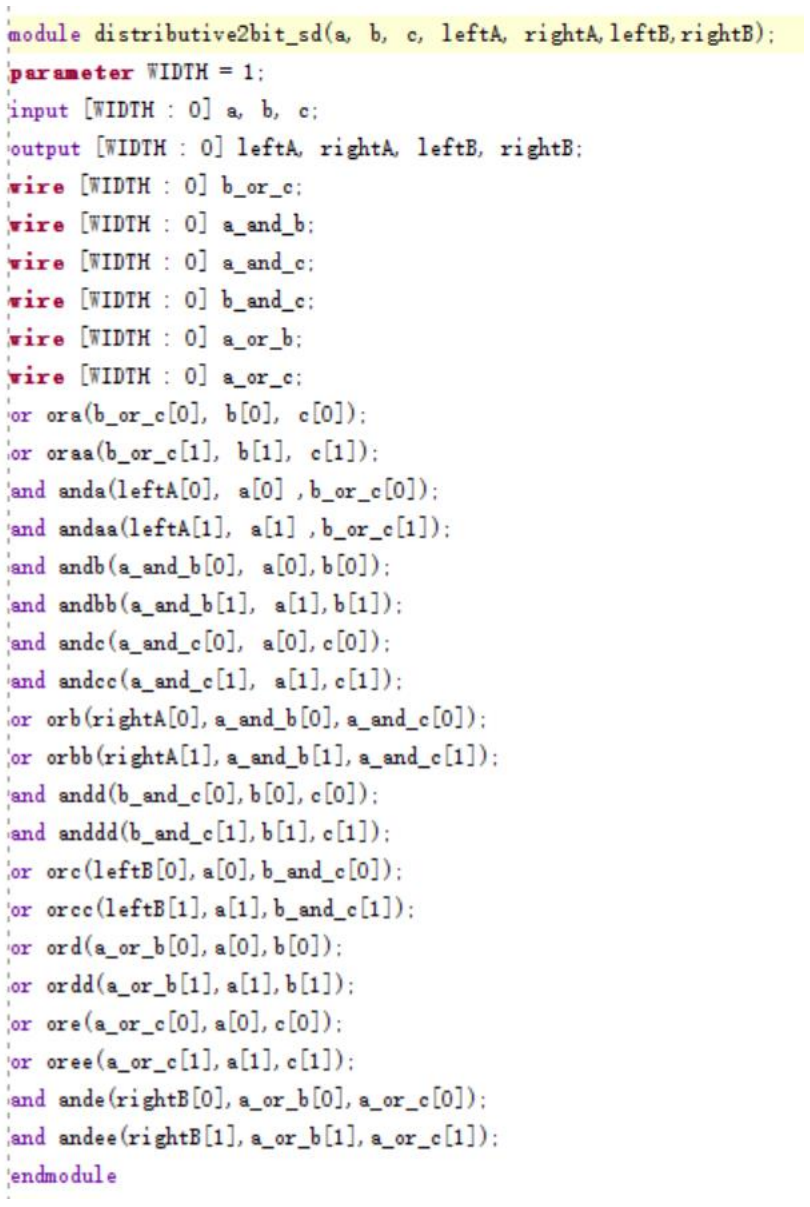


```
module distributive2bit_df(a, b, c, leftA, rightA,leftB,rightB);
parameter WIDTH = 1;
input [WIDTH : 0] a;
input [WIDTH : 0] b;
input [WIDTH : 0] c;
output [WIDTH : 0] leftA;
output [WIDTH : 0] rightA;
output [WIDTH : 0] leftB;
output [WIDTH : 0] rightB;
assign leftA = a & (b | c);
assign rightA = (a & b) | (a & c);
assign leftB = a | (b & c);
assign rightB = (a | b) & (a | c);
endmodule
```

（4） distributive2bit_sd.v

```
              module distributive2bit_sd(a, b, c, leftA,
rightA,leftB,rightB);
              parameter WIDTH = 1;
              input [WIDTH : 0] a;
              input [WIDTH : 0] b;
              input [WIDTH : 0] c;
              output [WIDTH : 0] leftA;
              output [WIDTH : 0] rightA;
              output [WIDTH : 0] leftB;
              output [WIDTH : 0] rightB;
              wire [WIDTH : 0] b_or_c;
              wire [WIDTH : 0] a_and_b;
              wire [WIDTH : 0] a_and_c;
              wire [WIDTH : 0] b_and_c;
              wire [WIDTH : 0] a_or_b;
              wire [WIDTH : 0] a_or_c;
              or ora(b_or_c[0], b[0], c[0]);
              or oraa(b_or_c[1], b[1], c[1]);
              and anda(leftA[0], a[0] ,b_or_c[0]);
              and andaa(leftA[1], a[1] ,b_or_c[1]);
              and andb(a_and_b[0], a[0],b[0]);
              and andbb(a_and_b[1], a[1],b[1]);
```

```
 and andc(a_and_c[0], a[0],c[0]);
 and andcc(a_and_c[1], a[1],c[1]);
 or orb(rightA[0],a_and_b[0],a_and_c[0]);
 or orbb(rightA[1],a_and_b[1],a_and_c[1]);
 and andd(b_and_c[0],b[0],c[0]);
 and anddd(b_and_c[1],b[1],c[1]);
 or orc(leftB[0],a[0],b_and_c[0]);
 or orcc(leftB[1],a[1],b_and_c[1]);
 or ord(a_or_b[0],a[0],b[0]);
 or ordd(a_or_b[1],a[1],b[1]);
 or ore(a_or_c[0],a[0],c[0]);
 or oree(a_or_c[1],a[1],c[1]);
 and ande(rightB[0],a_or_b[0],a_or_c[0]);
 and andee(rightB[1],a_or_b[1],a_or_c[1]);
endmodule
```



```
module distributive2bit_sd(a, b, c, leftA, rightA,leftB,rightB);
parameter WIDTH = 1;
input [WIDTH : 0] a, b, c;
output [WIDTH : 0] leftA, rightA, leftB, rightB;
wire [WIDTH : 0] b_or_c;
wire [WIDTH : 0] a_and_b;
wire [WIDTH : 0] a_and_c;
wire [WIDTH : 0] b_and_c;
wire [WIDTH : 0] a_or_b;
wire [WIDTH : 0] a_or_c;
or ora(b_or_c[0], b[0], c[0]);
or oraa(b_or_c[1], b[1], c[1]);
and anda(leftA[0], a[0] ,b_or_c[0]);
and andaa(leftA[1], a[1] ,b_or_c[1]);
and andb(a_and_b[0], a[0],b[0]);
and andbb(a_and_b[1], a[1],b[1]);
and andc(a_and_c[0], a[0],c[0]);
and andcc(a_and_c[1], a[1],c[1]);
or orb(rightA[0],a_and_b[0],a_and_c[0]);
or orbb(rightA[1],a_and_b[1],a_and_c[1]);
and andd(b_and_c[0],b[0],c[0]);
and anddd(b_and_c[1],b[1],c[1]);
or orc(leftB[0],a[0],b_and_c[0]);
or orcc(leftB[1],a[1],b_and_c[1]);
or ord(a_or_b[0],a[0],b[0]);
or ordd(a_or_b[1],a[1],b[1]);
or ore(a_or_c[0],a[0],c[0]);
or oree(a_or_c[1],a[1],c[1]);
and ande(rightB[0],a_or_b[0],a_or_c[0]);
and andee(rightB[1],a_or_b[1],a_or_c[1]);
endmodule
```

Truth-table:

1 bit:

a	0	1	0	0	1	1	0	1
b	0	0	1	0	1	0	1	1
c	0	0	0	1	0	1	1	1
a(c+b)	0	0	0	0	1	1	0	1
ab+ac	0	0	0	0	1	1	0	1
a+bc	0	1	0	0	1	1	1	1
(a+b)(a+c)	0	1	0	0	1	1	1	1

2 bits:

a	b	c	a(b+c)	ab+ac	a+bc	(a+b)(a+c)
00	00	00	00	00	00	00
00	00	01	00	00	00	00
00	00	10	00	00	00	00
00	00	11	00	00	00	00
00	01	00	00	00	00	00
00	01	01	00	00	01	01
00	01	10	00	00	00	00
00	01	11	00	00	01	01
00	10	00	00	00	00	00
00	10	01	00	00	00	00
00	10	10	00	00	10	10
00	10	11	00	00	10	10
00	11	00	00	00	00	00
00	11	01	00	00	01	01
00	11	10	00	00	10	10
00	11	11	00	00	11	11
01	00	00	00	00	01	01
01	00	01	01	01	01	01
01	00	10	00	00	01	01
01	00	11	01	01	01	01
01	01	00	01	01	01	01
01	01	01	01	01	01	01
01	01	10	01	01	01	01
01	01	11	01	01	01	01
01	10	00	00	00	01	01
01	10	01	01	01	01	01
01	10	10	00	00	11	11
01	10	11	01	01	11	11

01	11	00	01	01	01	01
01	11	01	01	01	01	01
01	11	10	01	01	11	11
01	11	11	01	01	11	11
10	00	00	00	00	10	10
10	00	01	00	00	10	10
10	00	10	10	10	10	10
10	00	11	10	10	10	10
10	01	00	00	00	10	10
10	01	01	00	00	11	11
10	01	10	10	10	10	10
10	01	11	10	10	11	11
10	10	00	10	10	10	10
10	10	01	10	10	10	10
10	10	10	10	10	10	10
10	10	11	10	10	10	10
10	11	00	10	10	10	10
10	11	01	10	10	11	11
10	11	10	10	10	10	10
10	11	11	10	10	11	11
11	00	00	00	00	11	11
11	00	01	01	01	11	11
11	00	10	10	10	11	11
11	00	11	11	11	11	11
11	01	00	01	01	11	11
11	01	01	01	01	11	11
11	01	10	11	11	11	11
11	01	11	11	11	11	11
11	10	00	10	10	11	11
11	10	01	11	11	11	11
11	10	10	10	10	11	11
11	10	11	11	11	11	11
11	11	00	11	11	11	11
11	11	01	11	11	11	11
11	11	10	11	11	11	11
11	11	11	11	11	11	11

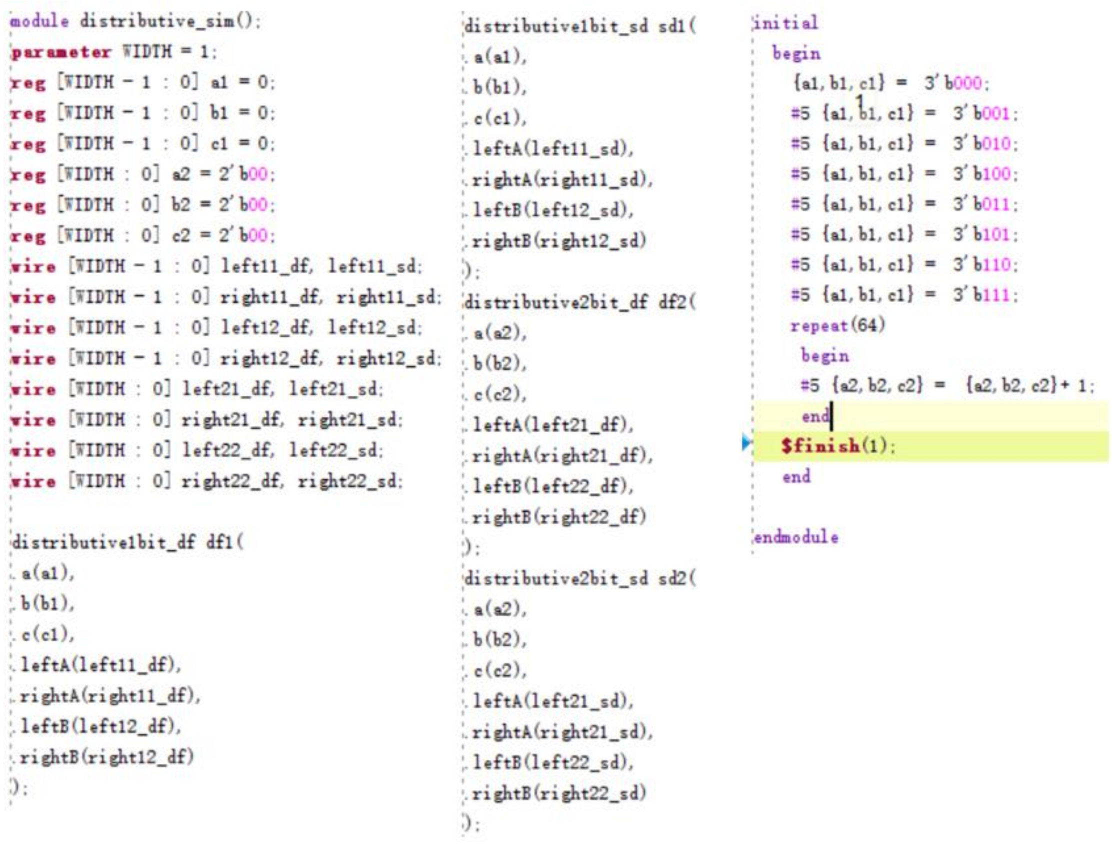
SIMULATION

Verilog code:

```
module distributive_sim();
```

```
parameter WIDTH = 1;
reg [WIDTH - 1 : 0] a1 = 0;
reg [WIDTH - 1 : 0] b1 = 0;
reg [WIDTH - 1 : 0] c1 = 0;
reg [WIDTH : 0] a2 = 2'b00;
reg [WIDTH : 0] b2 = 2'b00;
reg [WIDTH : 0] c2 = 2'b00;
wire [WIDTH - 1 : 0] left11_df, left11_sd;
wire [WIDTH - 1 : 0] right11_df, right11_sd;
wire [WIDTH - 1 : 0] left12_df, left12_sd;
wire [WIDTH - 1 : 0] right12_df, right12_sd;
wire [WIDTH : 0] left21_df, left21_sd;
wire [WIDTH : 0] right21_df, right21_sd;
wire [WIDTH : 0] left22_df, left22_sd;
wire [WIDTH : 0] right22_df, right22_sd;

distributive1bit_df df1(
.a(a1),
.b(b1),
.c(c1),
.leftA(left11_df),
.rightA(right11_df),
.leftB(left12_df),
.rightB(right12_df)
);
distributive1bit_sd sd1(
.a(a1),
.b(b1),
.c(c1),
.leftA(left11_sd),
.rightA(right11_sd),
.leftB(left12_sd),
.rightB(right12_sd)
);
distributive2bit_df df2(
.a(a2),
.b(b2),
.c(c2),
.leftA(left21_df),
.rightA(right21_df),
.leftB(left22_df),
.rightB(right22_df)
);
distributive2bit_sd sd2(
.a(a2),
.b(b2),
.c(c2),
.leftA(left21_sd),
.rightA(right21_sd),
.leftB(left22_sd),
.rightB(right22_sd)
);
initial
  begin
    {a1,b1,c1} =  3'b000;
    #5 {a1,b1,c1} =  3'b001;
    #5 {a1,b1,c1} =  3'b010;
    #5 {a1,b1,c1} =  3'b100;
    #5 {a1,b1,c1} =  3'b011;
    #5 {a1,b1,c1} =  3'b101;
    #5 {a1,b1,c1} =  3'b110;
    #5 {a1,b1,c1} =  3'b111;
    repeat(64)
     begin
     #5 {a2,b2,c2} =  {a2,b2,c2}+ 1;
     end
   $finish(1);
   end
 endmodule
```



```
module distributive_sim();
parameter WIDTH = 1;
reg [WIDTH - 1 : 0] a1 = 0;
reg [WIDTH - 1 : 0] b1 = 0;
reg [WIDTH - 1 : 0] c1 = 0;
reg [WIDTH : 0] a2 = 2'b00;
reg [WIDTH : 0] b2 = 2'b00;
reg [WIDTH : 0] c2 = 2'b00;
wire [WIDTH - 1 : 0] left11_df, left11_sd;
wire [WIDTH - 1 : 0] right11_df, right11_sd;
wire [WIDTH - 1 : 0] left12_df, left12_sd;
wire [WIDTH - 1 : 0] right12_df, right12_sd;
wire [WIDTH : 0] left21_df, left21_sd;
wire [WIDTH : 0] right21_df, right21_sd;
wire [WIDTH : 0] left22_df, left22_sd;
wire [WIDTH : 0] right22_df, right22_sd;

distributive1bit_df df1(
.a(a1),
.b(b1),
.c(c1),
.leftA(left11_df),
.rightA(right11_df),
.leftB(left12_df),
.rightB(right12_df)
);
```

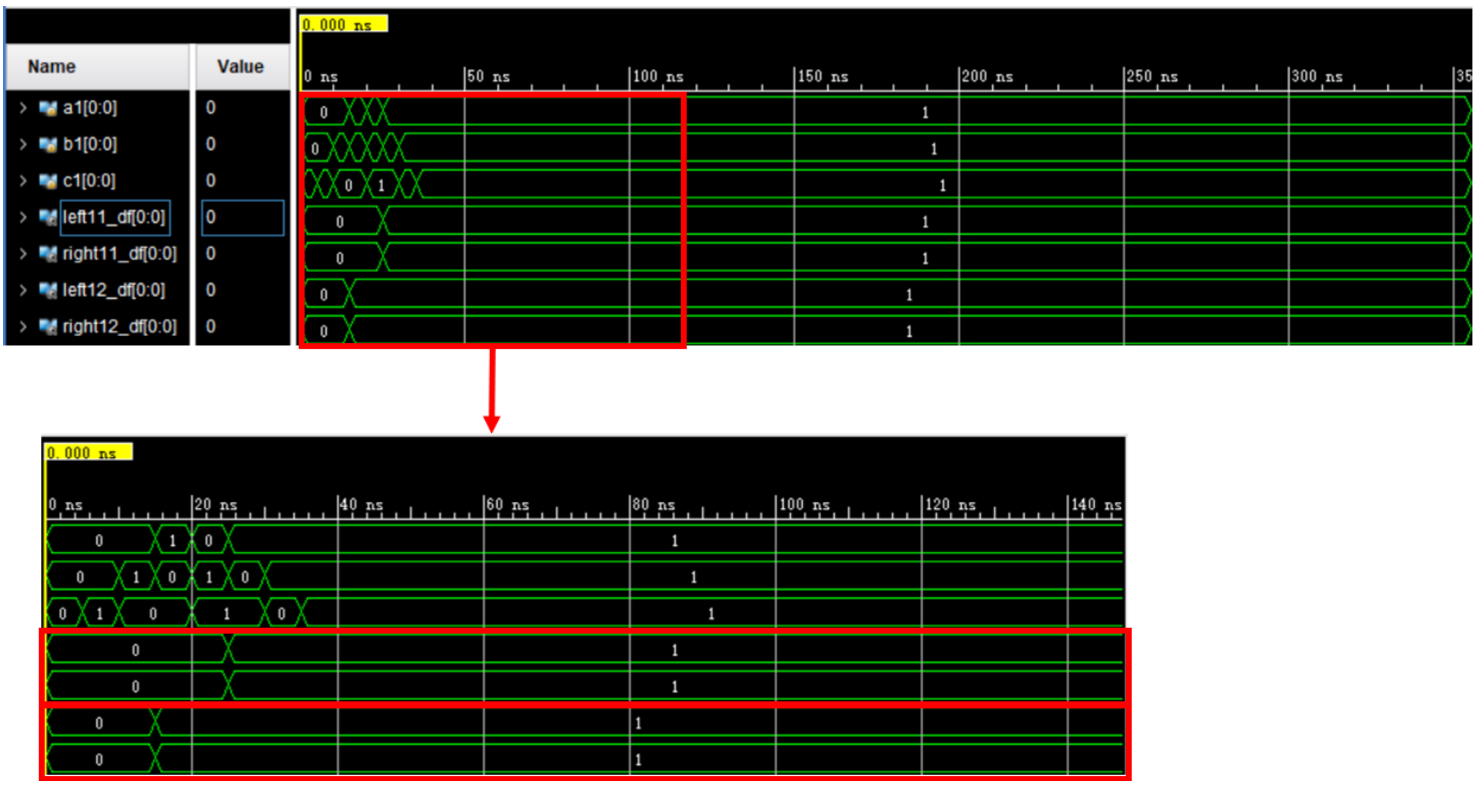
```
distributive1bit_sd sd1(
.a(a1),
.b(b1),
.c(c1),
.leftA(left11_sd),
.rightA(right11_sd),
.leftB(left12_sd),
.rightB(right12_sd)
);
distributive2bit_df df2(
.a(a2),
.b(b2),
.c(c2),
.leftA(left21_df),
.rightA(right21_df),
.leftB(left22_df),
.rightB(right22_df)
);
distributive2bit_sd sd2(
.a(a2),
.b(b2),
.c(c2),
.leftA(left21_sd),
.rightA(right21_sd),
.leftB(left22_sd),
.rightB(right22_sd)
);
```

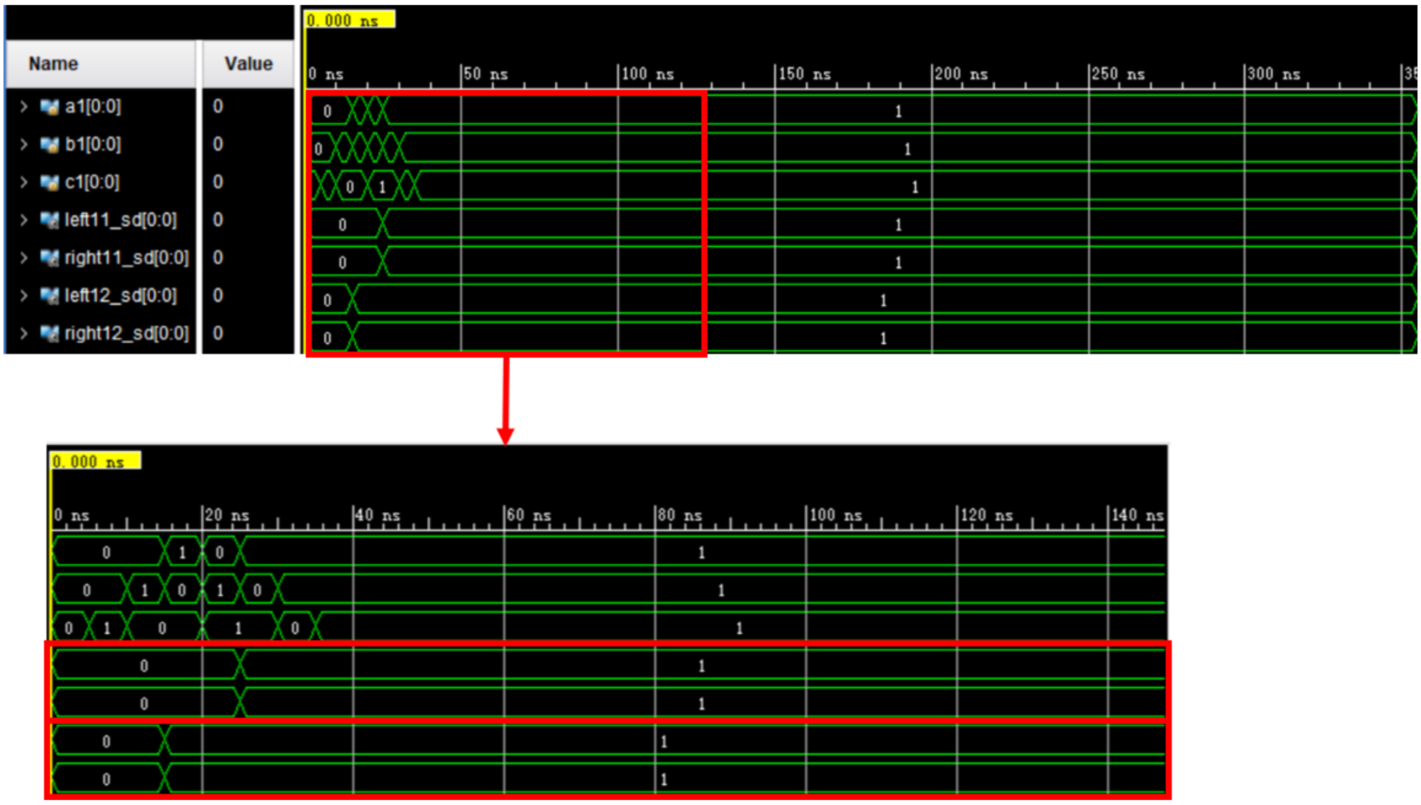
```
initial
  begin
    {a1, b1, c1} = 3'b000;
    #5 {a1, b1, c1} = 3'b001;
    #5 {a1, b1, c1} = 3'b010;
    #5 {a1, b1, c1} = 3'b100;
    #5 {a1, b1, c1} = 3'b011;
    #5 {a1, b1, c1} = 3'b101;
    #5 {a1, b1, c1} = 3'b110;
    #5 {a1, b1, c1} = 3'b111;
    repeat(64)
     begin
     #5 {a2, b2, c2} =  {a2, b2, c2}+ 1;
     end
   $finish(1);
  end

endmodule
```

simulation result:

The result of 1 bit:





a1, b1, c1 represent A, B, C in the formula:

A(B+C) = AB+AC ; A + BC = (A+B)(A+C);

Using data flow, left11_df means A(B+C) , right11_df means AB+AC, left12_df means A+BC, right12_df means (A+B)(A+C);

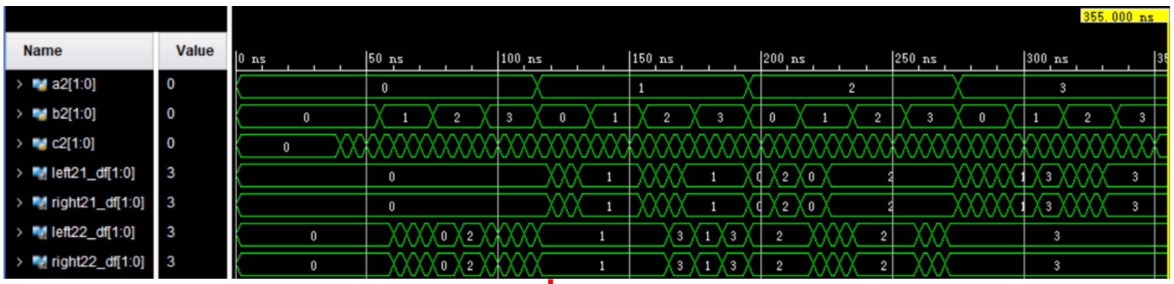
Using structure style, left11_sd means A(B+C) , right11_sd means AB+AC, left12_sd means A+BC, right12_sd means (A+B)(A+C);

we can know that 0 | 1 = 1, 1 | 0 = 1, 1 | 1 = 1, 0 | 0 = 0;

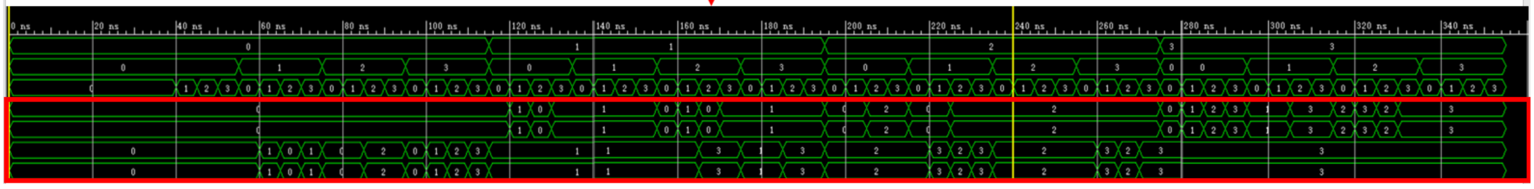
1 & 1 = 1, 0 & 1 = 0, 1 & 0 = 0, 0 & 0 = 0;

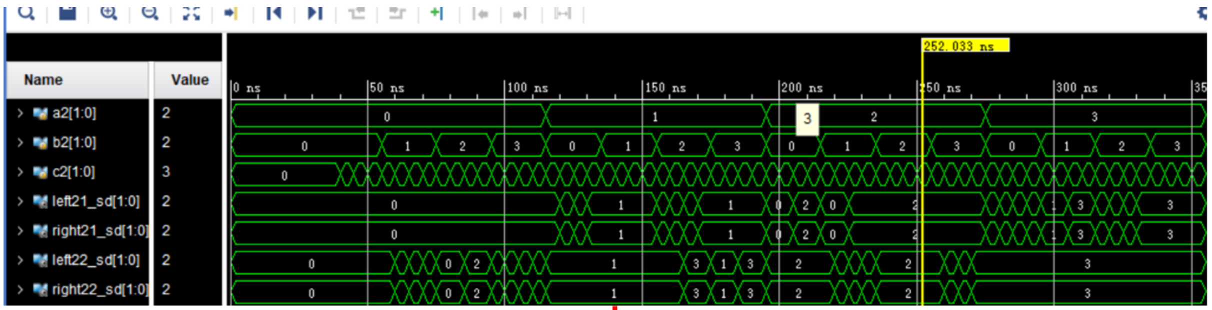
if left11_df = right11_df, left12_df = right12_df, left11_sd = right11_sd, left12_sd = right12_sd, then the formulas are proved.

The result of 2 bits:

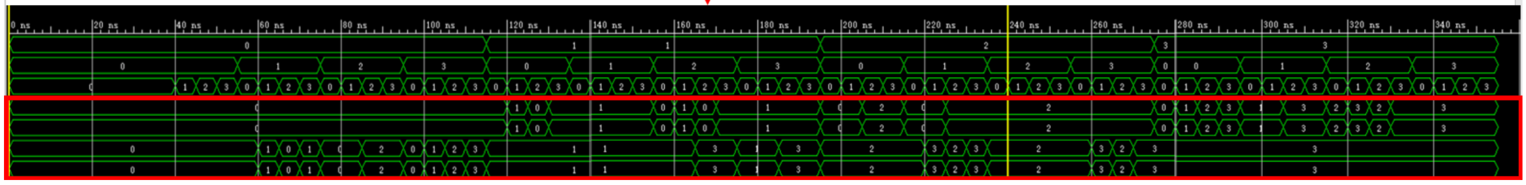


After zooming in:





After zooming in:



a2, b2, c2 represent A, B, C in the formula:

A(B+C) = AB+AC ; A + BC = (A+B)(A+C);

Using data flow, left21_df means A(B+C) , right21_df means AB+AC, left22_df means A+BC, right22_df means (A+B)(A+C);

Using structure style, left21_sd means A(B+C) , right21_sd means AB+AC, left22_sd means A+BC, right22_sd means (A+B)(A+C);

we can know that 0 | 1 = 1, 1 | 0 = 1, 1 | 1 = 1, 0 | 0 = 0;

1 & 1 = 1, 0 & 1 = 0, 1 & 0 = 0, 0 & 0 = 0;

If it has 2 bits, compute bit by bit;

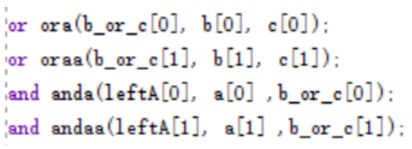
if left21_df = right21_df, left22_df = right22_df, left21_sd = right21_sd, left22_sd = right22_sd, then the formulas are proved, it means formulas can be used in multi bit-width.

THE DESCRIPTION OF OPERATION

problem and solution:

In the implementation process, when use structured style, if two or more bits are found, each bit must be carried out separately to do primitive gate calculate. Otherwise, an error will be reported.

The solution as follow: (like array to deal with one by one)



```
or ora(b_or_c[0], b[0], c[0]);
or oraa(b_or_c[1], b[1], c[1]);
and anda(leftA[0], a[0] ,b_or_c[0]);
and andaa(leftA[1], a[1] ,b_or_c[1]);
```