DIGITAL DESIGN

ASSIGNMENT REPORT

ASSIGNMENT ID : 1

Student Name: 何泽安（He Zean）

Student ID: 12011323

PART 1: DIGITAL DESIGN THEORY

Q. 1

(a) 16 kilo bytes = $16*2^{10}$ bytes = 16384 bytes

(b) 32 mega bytes = $32*2^{20}$ bytes = 33554432 bytes

(c) 3.2 giga bytes = $3.2*2^{30}$ bytes ≈ 3435973837 bytes

Q. 2

Bin: $(1111\ 1111\ 1111)_2$

Dec: $1*2^{11}+1*2^{10}+...+1*2^1+1*2^0 = (4095)_{10}$

Hex: $(FFF)_{16}$ (since $(1111)_2 = 1*2^3+1*2^2+1*2^1+1*2^0 = (15)_{10} = (F)_{16}$)

Q. 3

(a)

$$248 \div 2 = 124 \cdots 0$$

$$124 \div 2 = 62 \cdots 0$$

$$62 \div 2 = 31 \cdots 0$$

$$31 \div 2 = 15 \cdots 1$$

$$15 \div 2 = 7 \cdots 1$$

$$7 \div 2 = 3 \cdots 1$$

$$3 \div 2 = 1 \cdots 1$$

$$1 \div 2 = 0 \cdots 1$$

$$\text{Therefore, } (248)_{10} = (11111000)_2$$

(b)

$$248 \div 16 = 15 \cdots 8$$

$$15 \div 16 = 0 \cdots F$$

$$\text{Therefore, } (248)_{10} = (F8)_{16} = (1111\ 1000)_2$$

Obviously, (b) is faster.

Q. 4

(a)

9's complement: $(10^8 - 1) - 25273036 = 74726963$

10's complement: $10^8 - 25273036 = 74726964$

(b)

9's complement: $(10^8 - 1) - 64322610 = 35677389$

10's complement: $10^8 - 64322610 = 35677390$

Q. 5

(a) FFFF - C6BF + 1 = 3941

(b) $(C6BF)_{16} = (1100\ 0110\ 1011\ 1111)_2$

(c) $(1111\ 1111\ 1111\ 1111)_2 - (1100\ 0110\ 1011\ 1111)_2 + (1)_2 = (11\ 1001\ 0100\ 0001)_2$

(d) $(0011\ 1001\ 0100\ 0001)_2 = (3941)_{16}$, same as the answer in (a).

Q. 6

(a)

$$23 \div 2 = 11 \cdots 1$$

$$11 \div 2 = 5 \cdots 1$$

$$5 \div 2 = 2 \cdots 1$$

$$2 \div 2 = 1 \cdots 0$$

$$1 \div 2 = 0 \cdots 1$$

Therefore, $(23)_{10} = (10111)_2$

$$.5625 * 2 = 1.125$$

$$.125 * 2 = 0.25$$

$$.25 * 2 = 0.5$$

$$.5 * 2 = 1.0$$

Therefore, $(.5625)_{10} = (.1001)_2$

Therefore, $(23.5625)_{10} = (10111.1001)_2$

(b)

$$5/3 \approx 1.6666666667$$

$$.6666666667 * 2 = 1.3333333334$$

$$.3333333334 * 2 = 0.6666666668$$

$$.6666666668 * 2 = 1.3333333336$$

$$.3333333336 * 2 = 0.6666666672$$

$$.6666666672 * 2 = 1.3333333344$$

$$.3333333344 * 2 = 0.6666666688$$

$.6666666688 * 2 = 1.3333333376$

$.3333333376 * 2 = 0.6666666752$

Thus, $(5/3)_{10} \approx (1.10101010)_2$

$(1.10101010)_2 = 1*2^0+1*2^{-1}+0*2^{-2}+1*2^{-3}+0*2^{-4}+1*2^{-5}+0*2^{-6}+1*2^{-7}+0*2^{-8} = \mathbf{(1.6640625)_{10}}$

It loses about 0.156%'s precision.

(c)

$(0001.\ 1010\ 1010)_2 = (1.AA)_{16}$

$(1.AA)_{16} = 1*16^0 + 10*16^{-1} + 10*16^{-2} = (1.6640625)_{10}$

Here we can group 4 binary numbers and convert them into one hexadecimal number:

$(abcd)_2 = (e)_{16}$

Where a, b, c and d are either 1 or 0; e is a hex number (0 - E)

$(abcd)_2 = a*2^3+b*2^2+c*2^1+d*2^0$, $0 <= (abcd)_2 <= 15$

then for any abcd, there exists an e, s.t. $(e)_{16} = (abcd)_2$

Thus, converting a binary number into hexadecimal doesn't lose precision.

Q. 7

(a) $(1001\ 0111\ 0101)_{BCD} = (975)_{10}$

(b) $(1001\ 0111\ 0101)_{XS3} = (642)_{10}$

(c) $(1001\ 0111\ 0101)_{84\text{-}2\text{-}1} = (713)_{10}$

(d) $(1001\ 0111\ 0101)_{6311} = (754)_{10}$

(e) $(100101110101)_2 = (2421)_{10}$

Q. 8

(a) 11011010 & 01001110 = $(01001010)_2$ = $(4A)_{16}$

(b) 11011010 | 01001110 = $(11011110)_2$ = $(DE)_{16}$

(c) 11011010 ˆ 01001110 = $(10010100)_2$ = $(94)_{16}$

(d) ~ 11011010 = $(00100101)_2$ = $(25)_{16}$

(e) ~ 01001110 = $(10110001)_2$ = $(B1)_{16}$

(f) ~ (11011010 & 01001110) =~ 01001010 = $(10110101)_2$ = $(B5)_{16}$

(g) ~ (11011010 | 01001110) =~ 11011110 = $(00100001)_2$ = $(21)_{16}$

PART 2: DIGITAL DESIGN LAB (TASK1)

DESIGN

- *Data flow*

```
`timescale 1ns/1ps

module UnsignedAdditionDF
#(parameter WIDTH = 1) (
   input [WIDTH-1:0] a, b,
   output [WIDTH:0] sum
);
   assign sum = a + b;
endmodule
```

- *Structured style*

```
`timescale 1ns/1ps

module UnsignedAdditionSD
#(parameter WIDTH = 1) (
```

```
    input [WIDTH-1:0] a, b,
    output [WIDTH:0] sum
);
    genvar i;
    wire [WIDTH:0] carry;
    assign carry[0] = 0;
    assign sum[WIDTH] = carry[WIDTH];
    generate
        for (i = 0; i < WIDTH; i++) begin
            fullAdder bitAddition(a[i], b[i], carry[i],
                                  sum[i], carry[i+1]);
        end
    endgenerate
endmodule

module fullAdder (
    input a, b cin,
    output sum, cout
);
    wire t1, t2, t3;

    xor(sum, a, b, cin);
    and(t1, a, b);
    and(t2, a, cin);
    and(t3, b, cin);
    or(cout, t1, t2, t3);
endmodule
```

- *Truth-table (1 bit)*

a (1 bit)	***b (1 bit)***	***sum (2 bit, dec)***	*sum[1]*	*sum[0]*
0	***0***	***0***	*0*	*0*
0	***1***	***1***	*0*	*1*
1	***0***	***1***	*0*	*1*
1	***1***	***2***	*1*	*0*

- *Truth-table (Full adder)*

a	*b*	*Carry (in)*	*sum*	*Carry (out)*
0	*0*	*0*	*0*	*0*
0	*1*	*0*	*1*	*0*
1	*0*	*0*	*1*	*0*
1	*1*	*0*	*0*	*1*
0	*0*	*1*	*1*	*0*
0	*1*	*1*	*0*	*1*
1	*0*	*1*	*0*	*1*
1	*1*	*1*	*1*	*1*

- *Truth-table (2 bit)*

a (2bit dec)	*b (2bit dec)*	*sum (3bit dec)*	*a[1]*	*a[0]*	*b[1]*	*b[0]*	*sum[2]*	*sum[1]*	*sum[0]*
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
0	*1*	*1*	*0*	*0*	*0*	*1*	*0*	*0*	*1*
0	*2*	*2*	*0*	*0*	*1*	*0*	*0*	*1*	*0*

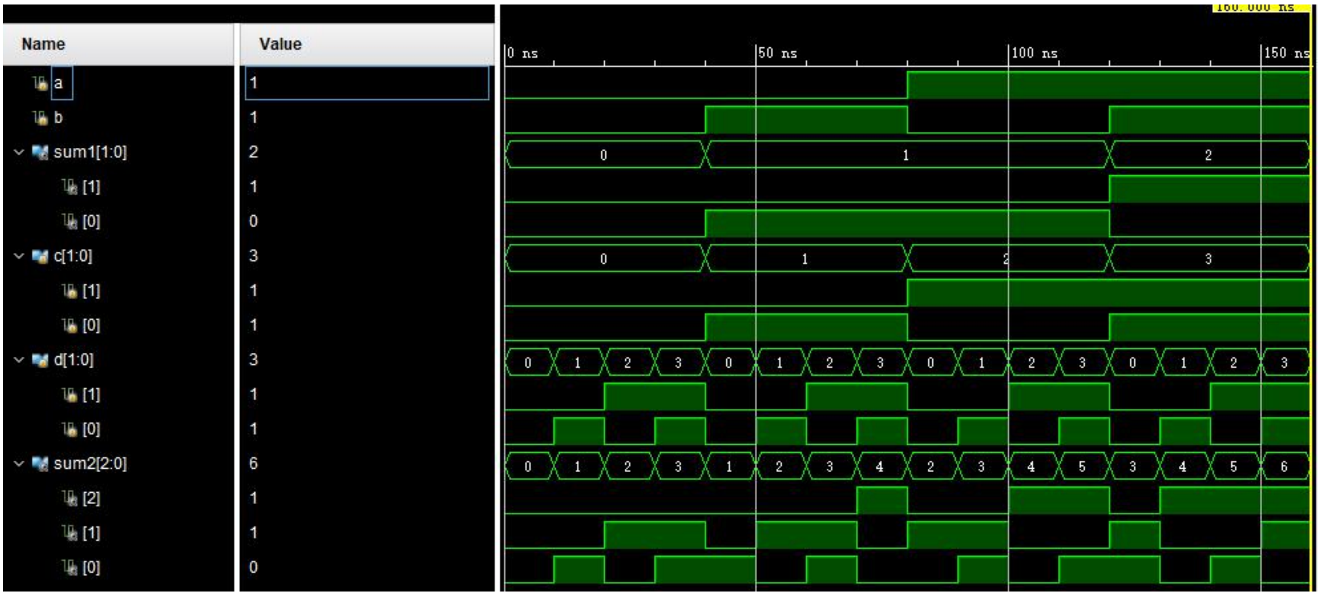
0	*3*	*3*	*0*	*0*	*1*	*1*	*0*	*1*	*1*
1	*0*	*1*	*0*	*1*	*0*	*0*	*0*	*0*	*1*
1	*1*	*2*	*0*	*1*	*0*	*1*	*0*	*1*	*0*
1	*2*	*3*	*0*	*1*	*1*	*0*	*0*	*1*	*1*
1	*3*	*4*	*0*	*1*	*1*	*1*	*1*	*0*	*0*
2	*0*	*2*	*1*	*0*	*0*	*0*	*0*	*1*	*0*
2	*1*	*3*	*1*	*0*	*0*	*1*	*0*	*1*	*1*
2	*2*	*4*	*1*	*0*	*1*	*0*	*1*	*0*	*0*
2	*3*	*5*	*1*	*0*	*1*	*1*	*1*	*0*	*1*
3	*0*	*3*	*1*	*1*	*0*	*0*	*0*	*1*	*1*
3	*1*	*4*	*1*	*1*	*0*	*1*	*1*	*0*	*0*
3	*2*	*5*	*1*	*1*	*1*	*0*	*1*	*0*	*1*
3	*3*	*6*	*1*	*1*	*1*	*1*	*1*	*1*	*0*

SIMULATION

- *Testbench*

```
`timescale 1ns/1ps

module UnsignedAdditionSim ();
// for testing 1-bit adder
   reg a, b;
   reg [1:0] sum1;
   UnsignedAdditionDF#(1) adder1bit(.a(a), .b(b), .sum(sum1));

// for testing 2-bit adder
   reg [1:0] c, d;
   reg [2:0] sum2;
   UnsignedAdditionDF#(2) adder2bit(.a(c), .b(d), .sum(sum2));

   initial begin
      {a,b,c,d} = 0; // init, RHS is actually 6'b000000
      #160 $finish;
   end

   initial begin
      while({a,b} < 2'b11)
      #10 {a,b} = {a,b} + 1;
   end

   initial begin
      while({c,d} < 4'b1111)
      #10 {c,d} = {c,d} + 1;
   end
endmodule
```

The above is for <data flow>, to test <structured design>, just change "UnsignedAdditionDF" into "UnsignedAdditionSD".



The waveforms of both types of design are the same.

I merged the test cases of both 1bit and 2bit into one simulation: since it takes 160ns to check the 16 possible cases for 2bits', I set the period of 1bit's as 40ns.

a+b=sim1 checks the 1bit's module. As the fig shows, from 0-40ns, a=0,b=0,sum1=0; 40-80ns: a=0,b=1,sum1=1; 80-120ns: a=1,b=0,c=1; 120-160ns: a=1,b=1,c=2_{10}(=10_2)

c+d=sim2 checks the 2bits' module. 0-10ns: 0+0=0; 10-20ns: 0+1=1; 20-30ns: 0+2=2; 30-40ns: 0+3=3; 40-50ns: 1+0=1; 50-60ns: 1+1=2; 60-70ns: 1+2=3; 70-80ns: 1+3=4; 80-90ns: 2+0=2; 90-100ns: 2+1=3; 100-110ns: 2+2=4; 110-120ns: 2+3=5; 120-130ns: 3+0=3; 130-140ns: 3+1=4; 140-150ns: 3+2=5; 150-160ns: 3+3=6.

THE DESCRIPTION OF OPERATION

- How to design the gate level of a full adder?

  Drawing a K-map

- How to make the instance of module exactly has the necessary full adders?

  Using the ***generator syntax*** (see Fundamentals of Digital Logic with Verilog Design 3e, pp.83)

- At first I found it's difficult to handle the varibles to change their values, that's complex, then I thought about that I can write two initial parts in my sim

- Why don't we support LaTeX for our report???

PART 2: DIGITAL DESIGN LAB (TASK2)

DESIGN

- *Verilog design while using data flow*

```
// distributive1bit_df.v
`timescale 1ns/1ps

module Distributive1bit_df (
   input a, b, c,
   output alhs, arhs, aissame,
   output blhs, brhs, bissame
);
   assign alhs = a & (b | c);
   assign arhs = a & b | a & c;
   assign aissame = ~(alhs ^ arhs);

   assign blhs = a | b & c;
   assign brhs = (a | b) & (a | c);
   assign bissame = ~(blhs ^ brhs);
endmodule
```

```
// distributive2bit_df.v
`timescale 1ns/1ps

module Distributive2bit_df (
   input [1:0] a, b, c,
   output [1:0] alhs, arhs,
   output [1:0] blhs, brhs,
   output [1:0] aissame, bissame
);
   assign alhs = a & (b | c);
   assign arhs = a & b | a & c;
   assign aissame = ~(alhs ^ arhs);

   assign blhs = a | b & c;
   assign brhs = (a | b) & (a | c);
```

```
    assign bissame = ~(blhs ^ brhs);
endmodule
```

- *Verilog design while using structured design*

```
// distributive1bit_sd.v
`timescale 1ns/1ps

module Distributive1bit_sd (
    input a, b, c,
    output alhs, arhs, aissame,
    output blhs, brhs, bissame
);
    // we omit the declations of wires here
    or(aobc, b, c);
    and(alhs, a, aobc);

    and(anab, a, b);
    and(anac, a, c);
    or(arhs, anab, anac);

    xor(axlr, alhs, arhs);
    not(aissame, axlr);

    and(bnbc, b, c);
    or(blhs, a, bnbc);

    or(boab, a, b);
    or(boac, a, c);
    and(brhs, boab, boac);

    xor(bxlr, blhs, brhs);
    not(bissame, bxlr);
endmodule
```

```
// distributive2bit_sd.v
`timescale 1ns/1ps

module Distributive2bit_sd (
    input [1:0] a, b, c,
    output [1:0] alhs, arhs,
    output [1:0] blhs, brhs,
```

```
    output [1:0] aissame, bissame
);
    // distributive <a>
    wire [1:0] aobc, anab, anac;
    // [0]
    or(aobc[0], b[0], c[0]);
    and(alhs[0], a[0], aobc[0]);

    and(anab[0], a[0], b[0]);
    and(anac[0], a[0], c[0]);
    or(arhs[0], anab[0], anac[0]);

    // [1]
    or(aobc[1], b[1], c[1]);
    and(alhs[1], a[1], aobc[1]);

    and(anab[1], a[1], b[1]);
    and(anac[1], a[1], c[1]);
    or(arhs[1], anab[1], anac[1]);

    // check if lhs==rhs
    xor(axlr0, alhs[0], arhs[0]);
    not(aissame[0], axlr0);
    xor(axlr1, alhs[1], arhs[1]);
    not(aissame[1], axlr1);

    // distributive <b>
    wire [1:0] bnbc, boab, boac;
    and(bnbc[0], b[0], c[0]);
    or(blhs[0], a[0], bnbc[0]);

    or(boab[0], a[0], b[0]);
    or(boac[0], a[0], c[0]);
    and(brhs[0], boab[0], boac[0]);

    and(bnbc[1], b[1], c[1]);
    or(blhs[1], a[1], bnbc[1]);
    or(boab[1], a[1], b[1]);
    or(boac[1], a[1], c[1]);
    and(brhs[1], boab[1], boac[1]);

    xor(bxlr0, blhs[0], brhs[0]);
    not(bissame[0], bxlr0);
```

```
    xor(bxlr1, blhs[1], brhs[1]);
    not(bissame[1], bxlr1);
endmodule
```

- *Truth-table*

1-bit

A	*B*	*C*	*A(B+C)* *a_lhs*	*AB+AC* *a_rhs*	*A+BC* *b_lhs*	*(A+B)(A+C)* *b_rhs*
0	*0*	*0*	*0*	*0*	*0*	*0*
0	*0*	*1*	*0*	*0*	*0*	*0*
0	*1*	*0*	*0*	*0*	*0*	*0*
0	*1*	*1*	*0*	*0*	*1*	*1*
1	*0*	*0*	*0*	*0*	*1*	*1*
1	*0*	*1*	*1*	*1*	*1*	*1*
1	*1*	*0*	*1*	*1*	*1*	*1*
1	*1*	*1*	*1*	*1*	*1*	*1*

2-bit

A	*B*	*C*	*A(B+C)* *a_lhs*	*AB+AC* *a_rhs*	*A+BC* *b_lhs*	*(A+B)(A+C)* *b_rhs*

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

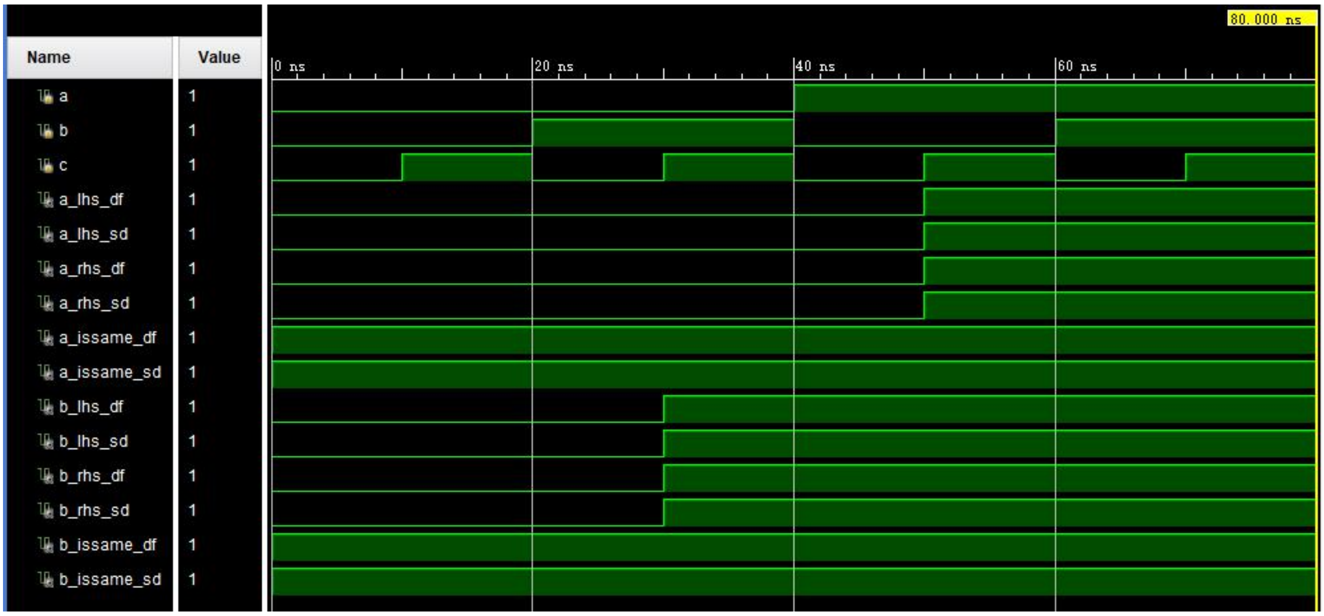
10	*11*	*01*	*10*	*10*	*11*	*11*
10	*11*	*10*	*10*	*10*	*10*	*10*
10	*11*	*11*	*10*	*10*	*11*	*11*
11	*00*	*00*	*00*	*00*	*11*	*11*
11	*00*	*01*	*01*	*01*	*11*	*11*
11	*00*	*10*	*10*	*10*	*11*	*11*
11	*00*	*11*	*11*	*11*	*11*	*11*
11	*01*	*00*	*01*	*01*	*11*	*11*
11	*01*	*01*	*01*	*01*	*11*	*11*
11	*01*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

1—BIT

```
// distributive1bit_sim.v
`timescale 1ns/1ps

module Distributive1bit_sim ();
   reg a, b, c;
   wire a_lhs_df, a_lhs_sd, a_rhs_df, a_rhs_sd;
   wire a_issame_df, a_issame_sd;

   wire b_lhs_df, b_lhs_sd, b_rhs_df, b_rhs_sd;
   wire b_issame_df, b_issame_sd;

   Distributive1bit_df d1df(a, b, c, a_lhs_df, a_rhs_df,
        a_issame_df, b_lhs_df, b_rhs_df, b_issame_df);
   Distributive1bit_sd d1sd(a, b, c, a_lhs_sd, a_rhs_sd,
        a_issame_sd, b_lhs_sd, b_rhs_sd, b_issame_sd);

   initial begin
      {a,b,c} = 0;
      while ({a,b,c} < 3'b111)
         #10 {a,b,c} = {a,b,c} + 1;
      #10 $finish;
   end
endmodule
```



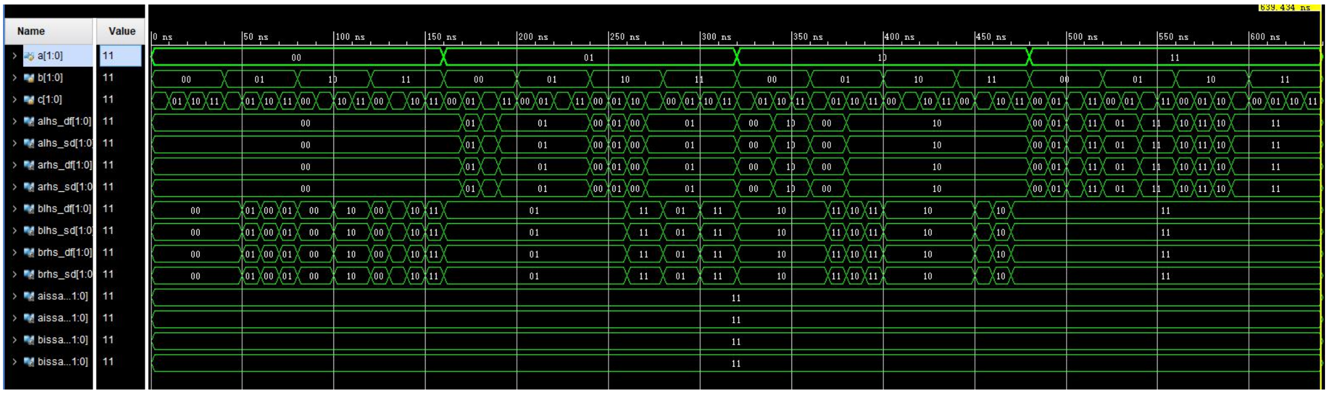
This is the waveform for testing 1bit distribution: data flow and structured design share the same three input, a, b and c. *About the qustion (a)*: Form A(B+C) is marked as LHS (dataflow design: a_lhs_df; structed design: a_lhs_sd) and form AB+AC is marked as RHS (dataflow design: a_rhs_df; structed design: a_rhs_sd). *The naming strategy of question (b) is similar.*

Also, to check if LHS is equivalent to RHS easier, variables *x_issame_y* are defined, since ~(a^b) is equivalent to (a==b). Here we can observe that these 4 variables are always **true (1)**, that means in any possible 4 cases, we have LHS=RHS, thus they are equivalent, thus the distributive laws (a and b) are checked.

2—BIT

```
// distributive2bit_sim.v
`timescale 1ns/1ps

module Distributive2bit_sim ();
   reg [1:0] a, b, c;
   wire [1:0] alhs_df, alhs_sd, arhs_df, arhs_sd;
   wire [1:0] blhs_df, blhs_sd, brhs_df, brhs_sd;
   wire [1:0] aissame_df, aissame_sd;
   wire [1:0] bissame_df, bissame_sd;

   Distributive2bit_df d2df(a, b, c, alhs_df, arhs_df,
         blhs_df, brhs_df, aissame_df, bissame_df);
   Distributive2bit_sd d2sd(a, b, c, alhs_sd, arhs_sd,
```

```
        blhs_sd, brhs_sd, aissame_sd, bissame_sd);

   initial begin
      {a,b,c} = 0;
      while ({a,b,c} < 6'b111111)
         #10 {a,b,c} = {a,b,c} + 1;
      #10 $finish;
   end
endmodule
```



The above waveform shows the 64 cases of 2-bit distributive module. Please note that the values marked on the figure are all binary, the first bit and the second bit are calculated separately (some are too small that didn't marked in the figure, however they had been checked right). Also, the last four variables [1:0] *aissame_df, aissame_sd, bissame_df, bissame_sd* are always ***11*** as shown, aka, the answer of these 2 bits are always be the same (and correct).

THE DESCRIPTION OF OPERATION

- *At first the sim waveform was incorrect, that the output comes like [Z0] or [Z1], then I checked my design file (structured design) and found I fogot to build a gate to connect the output and other variables.*