DIGITAL DESIGN

LAB3 GATES, BITWISE OPERATION & PRIMITIVE IN VERILOG, STRUCTURED DESIGN

2022 SUMMER TERM

LAB3

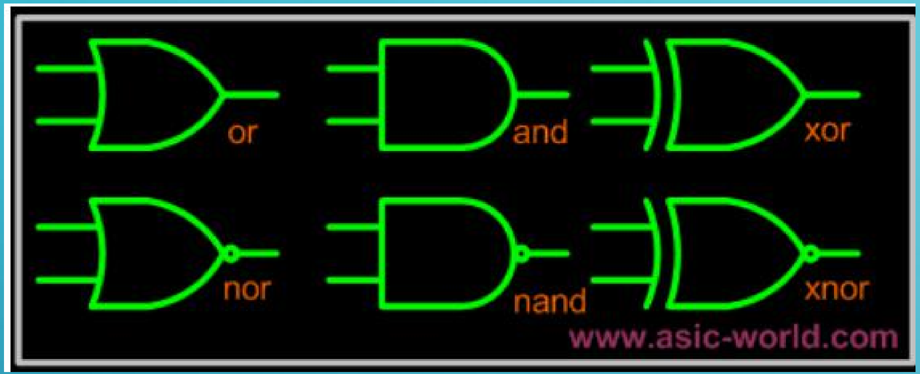
- Gates and Primitives in verilog
- bitwise and logic operations in verilog
- *Standard form vs SOP*(sum of minterms) vs POS(product of maxterms)
- Two ways to do the design
  - 1. data flow
  - 2. structured design
- Two kinds of Schematic
  - Schematic in ‘RTL Analysis’
  - Schematic of Synthesize

PRIMITIVE

- Verilog has built in primitives like gates, transmission gates, and switches. These are rarely used in design (RTL Coding), but are used in post synthesis world for modeling the ASIC/FPGA cells; these cells are then used for gate level simulation, or what is called as SDF simulation. Also the output netlist format from the synthesis tool, which is imported into the place and route tool, is also in Verilog gate level primitives.
- Note : RTL engineers still may use gate level primitives or ASIC library cells in RTL when using IO CELLS, Cross domain synch cells.

http://www.asic-world.com/verilog/index.html

PRIMITIVE GATE



Gate	Description
and	N-input AND gate
nand	N-input NAND gate
or	N-input OR gate
nor	N-input NOR gate
xor	N-input XOR gate
xnor	N-input XNOR gate

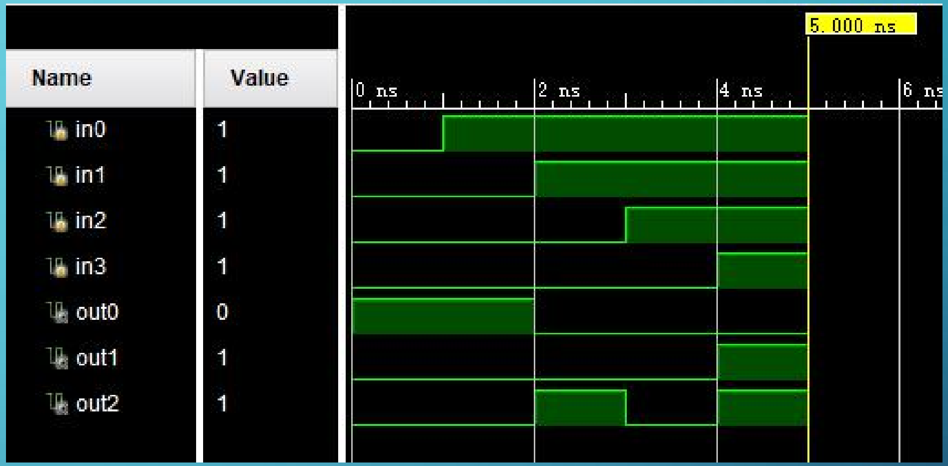
- The gates have one scalar output and multiple scalar inputs. The 1st terminal in the list of gate terminals is an output and the other terminals are inputs.
- Declaration Format
  - <gate> [<driving power><delay>] <uut1>(port list)[,<uut2>(port list), ..., <uutn>(port list)];
  - nand #10 nd1(a, data, clock, clear);
  - and and1(out1, in1, in2),

    and2(out2, in3, in4);

http://www.asic-world.com/verilog/index.html

PRIMITIVE GATE

```
module gates();
reg in0, in1, in2, in3;
wire out0, out1, out2;

not U1(out0, in1);
and U2(out1, in0, in1, in2, in3);
xor U3(out2, in1, in2, in3);

initial begin
    $monitor(
    "in0=%b, in1=%b, in2=%b, in3=%b; out0=%b, out1=%b, out2=%b",
    in0, in1, in2, in3, out0, out1, out2);

    in0 = 0;
    in1 = 0;
    in2 = 0;
    in3 = 0;

    #1 in0 = 1;
    #1 in1 = 1;
    #1 in2 = 1;
    #1 in3 = 1;

    #1 $finish;
end
endmodule
```



Tcl Console × Messages Log

```
# run 1000ns
in0=0, in1=0, in2=0, in3=0; out0=1, out1=0, out2=0
in0=1, in1=0, in2=0, in3=0; out0=1, out1=0, out2=0
in0=1, in1=1, in2=0, in3=0; out0=0, out1=0, out2=1
in0=1, in1=1, in2=1, in3=0; out0=0, out1=0, out2=0
in0=1, in1=1, in2=1, in3=1; out0=0, out1=1, out2=1
$finish called at time : 5 ns : File "D:/wangqing/digital_logic_2021/
INFO: [USF-XSim-96] XSim completed. Design snapshot 'gates_behav' loa
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:08 . Me
```

http://www.asic-world.com/verilog/index.html

BITWISE AND LOGICAL OPERATIONS IN VERILOG

- Logic Values
  - Four-valued logic (The IEEE 1364 standard)

    0, 1, Z (high impedance), and X (unknown logic value).

Logic Value	Description
0	zero, low, false
1	one, high, true
z or **Z**	high impedance, floating
x or **X**	unknown, uninitialized, contention

http://www.asic-world.com/verilog/index.html

BITWISE AND LOGICAL OPERATIONS IN VERILOG

- Operators
  - Arithmetic Operators, Relational Operators, Equality Operators, Logical Operators, Bit-wise Operators, Reduction Operators, Shift Operatorsm, Concatenation Operator, Replication Operator, Conditional Operators.
- Bitwise Operators and Logical Operators
- Operator Precedence

<table>
<tr><th>Operator</th><th>Symbols</th></tr>
<tr><td>Unary, Multiply, Divide, Modulus</td><td>!, ~, *, /, %</td></tr>
<tr><td>Add, Subtract, Shift</td><td>+, - , <<, >></td></tr>
<tr><td>Relation, Equality</td><td><,>,<=,>=,==,!=,===,!==</td></tr>
<tr><td>Reduction</td><td>&, !&,^,^~,|,~|</td></tr>
<tr><td>Logic</td><td>&&, ||</td></tr>
<tr><td>Conditional</td><td>? :</td></tr>
</table>

http://www.asic-world.com/verilog/index.html

BITWISE AND LOGICAL OPERATIONS IN VERILOG

- Bitwise Operators

Operator type	Operator symbols	Operation performed
Bitwise	~	Bitwise NOT (1's complement)
	&	Bitwise AND
	|	Bitwise OR
	^	Bitwise XOR
	~^ or ^~	Bitwise XNOR

- Bitwise operators perform a bit wise operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand.
- When operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

http://www.asic-world.com/verilog/index.html

BITWISE AND LOGICAL OPERATIONS IN VERILOG

- Bitwise Operators

```
initial begin
  // Bit Wise Negation
  $display (" ~4'b0001           = %b", (~4'b0001));
  $display (" ~4'bx001           = %b", (~4'bx001));
  $display (" ~4'bz001           = %b", (~4'bz001));
  // Bit Wise AND
  $display (" 4'b0001 &  4'b1001 = %b", (4'b0001 &  4'b1001));
  $display (" 4'b1001 &  4'bx001 = %b", (4'b1001 &  4'bx001));
  $display (" 4'b1001 &  4'bz001 = %b", (4'b1001 &  4'bz001));
  // Bit Wise OR
  $display (" 4'b0001 |  4'b1001 = %b", (4'b0001 |  4'b1001));
  $display (" 4'b0001 |  4'bx001 = %b", (4'b0001 |  4'bx001));
  $display (" 4'b0001 |  4'bz001 = %b", (4'b0001 |  4'bz001));
  // Bit Wise XOR
  $display (" 4'b0001 ^  4'b1001 = %b", (4'b0001 ^  4'b1001));
  $display (" 4'b0001 ^  4'bx001 = %b", (4'b0001 ^  4'bx001));
  $display (" 4'b0001 ^  4'bz001 = %b", (4'b0001 ^  4'bz001));
  // Bit Wise XNOR
  $display (" 4'b0001 ~^ 4'b1001 = %b", (4'b0001 ~^ 4'b1001));
  $display (" 4'b0001 ~^ 4'bx001 = %b", (4'b0001 ~^ 4'bx001));
  $display (" 4'b0001 ~^ 4'bz001 = %b", (4'b0001 ~^ 4'bz001));
  #10 $finish;
end
```

```
~4'b0001           = 1110
~4'bx001           = x110
~4'bz001           = x110
4'b0001 &  4'b1001 = 0001
4'b1001 &  4'bx001 = x001
4'b1001 &  4'bz001 = x001
4'b0001 |  4'b1001 = 1001
4'b0001 |  4'bx001 = x001
4'b0001 |  4'bz001 = x001
4'b0001 ^  4'b1001 = 1000
4'b0001 ^  4'bx001 = x000
4'b0001 ^  4'bz001 = x000
4'b0001 ~^ 4'b1001 = 0111
4'b0001 ~^ 4'bx001 = x111
4'b0001 ~^ 4'bz001 = x111
```

http://www.asic-world.com/verilog/index.html

BITWISE AND LOGICAL OPERATIONS IN VERILOG

- Logical Operators
  - Expressions connected by && and || are evaluated from left to right
  - Evaluation stops as soon as the result is known
  - The result is a scalar value:
    - 0 if the relation is false
    - 1 if the relation is true
    - x if any of the operands has x (unknown) bits

Operator type	Operator symbols	Operation performed
Logical	!	NOT
	&&	AND
	||	OR

http://www.asic-world.com/verilog/index.html

BITWISE AND LOGICAL OPERATIONS IN VERILOG

- Logical Operators

```
initial begin
  // Logical AND
  $display ("1'b1 && 1'b1 = %b", (1'b1 && 1'b1));
  $display ("1'b1 && 1'b0 = %b", (1'b1 && 1'b0));
  $display ("1'b1 && 1'bx = %b", (1'b1 && 1'bx));
  // Logical OR
  $display ("1'b1 || 1'b0 = %b", (1'b1 || 1'b0));
  $display ("1'b0 || 1'b0 = %b", (1'b0 || 1'b0));
  $display ("1'b0 || 1'bx = %b", (1'b0 || 1'bx));
  // Logical Negation
  $display ("! 1'b1       = %b", (!  1'b1));
  $display ("! 1'b0       = %b", (!  1'b0));
  #10 $finish;
end
```

```
1'b1 && 1'b1 = 1
1'b1 && 1'b0 = 0
1'b1 && 1'bx = x
1'b1 || 1'b0 = 1
1'b0 || 1'b0 = 0
1'b0 || 1'bx = x
! 1'b1       = 0
! 1'b0       = 1
```

http://www.asic-world.com/verilog/index.html

BITWISE AND LOGICAL OPERATIONS IN VERILOG

- Bitwise Operators and Logical Operators
- Example
  - assign q = a & b & c ;
  - assign z = x | y;
  - assign t = a & b | ~c;
- Priority
  - ~ ! > & > ^ ~^ ^~ > | > && > ||

http://www.asic-world.com/verilog/index.html

STANDARD FORM VS SOP VS POS

- F(A,B,C)=A+B’C
- F(A,B,C)=AB(C’+C)+AB’(C’+C)+B’C(A’+A)=A’B’C+AB’C‘+AB’C+ABC’+ABC

  =m1+m4+m5+m6+m7=∑(1,4,5,6,7)
- F(A,B,C)”=(F(A,B,C)’)’=(m0+m2+m3)’=M0.M2.M3 = ∏(0,2,3)

A	*B*	*C*	*F*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

```
module Lab5_demo1(
    input a,
    input b,
    input c,
    output z1,
    output z2,
    output z3
    );

    assign z1 = a | ((~b) & c);
    assign z2 = (a&b&c) | (a&b&(~c)) | (a&(~b)&c) | (a&(~b)&(~c)) | ((~a)&(~b)&c);//m7+m6+m5+m4+m1
    assign z3 = (a|b|c) & (a|(~b)|c) & (a|(~b)|(~c));//M0 • M2 • M3
endmodule
```

TWO WAYS TO DO THE DESIGN

- Data flow design: using “assign “ as *continuous assignment* , to transfer the data from input ports through variables to the output ports .

- Structured design: using verilog to define the relationship between modules. It is based on the module/IP.

DATA FLOW DESIGN

Demo: a) x | (x&y) b) x & (x | y)

```
module Lab3_dataflow(
    input x,
    input y,
    output q1,
    output q2,
    output q3
    );

    assign q1 = x;
    assign q2 = x | (x & y);
    assign q3 = x & (x | y);
endmodule
```

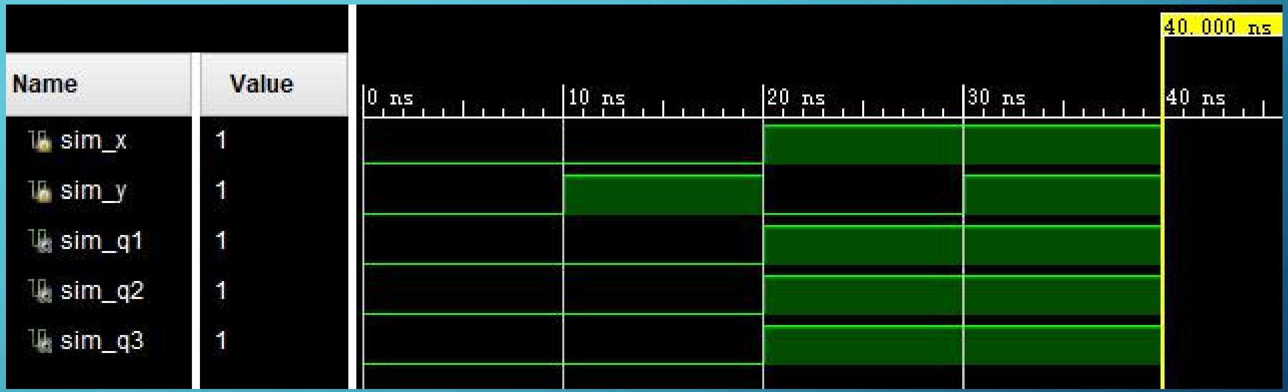
```
module Lab3_dataflow_sim();

    reg sim_x, sim_y;
    wire sim_q1, sim_q2, sim_q3;

    Lab3_dataflow df_sim1(
    .x(sim_x),
    .y(sim_y),
    .q1(sim_q1),
    .q2(sim_q2),
    .q3(sim_q3)
    );

    initial begin
        #10 sim_x = 0; sim_y = 0;
        #10 sim_x = 0; sim_y = 1;
        #10 sim_x = 1; sim_y = 0;
        #10 sim_x = 1; sim_y = 1;
        #10 $finish;
    end
endmodule
```

STRUCTURE DESIGN

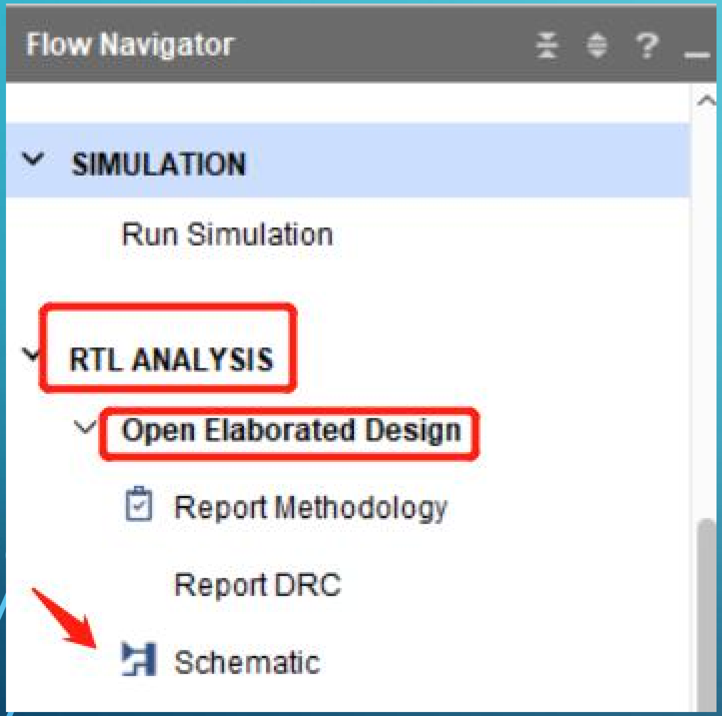
Demo: a) x | (x&y) b) x & (x | y)

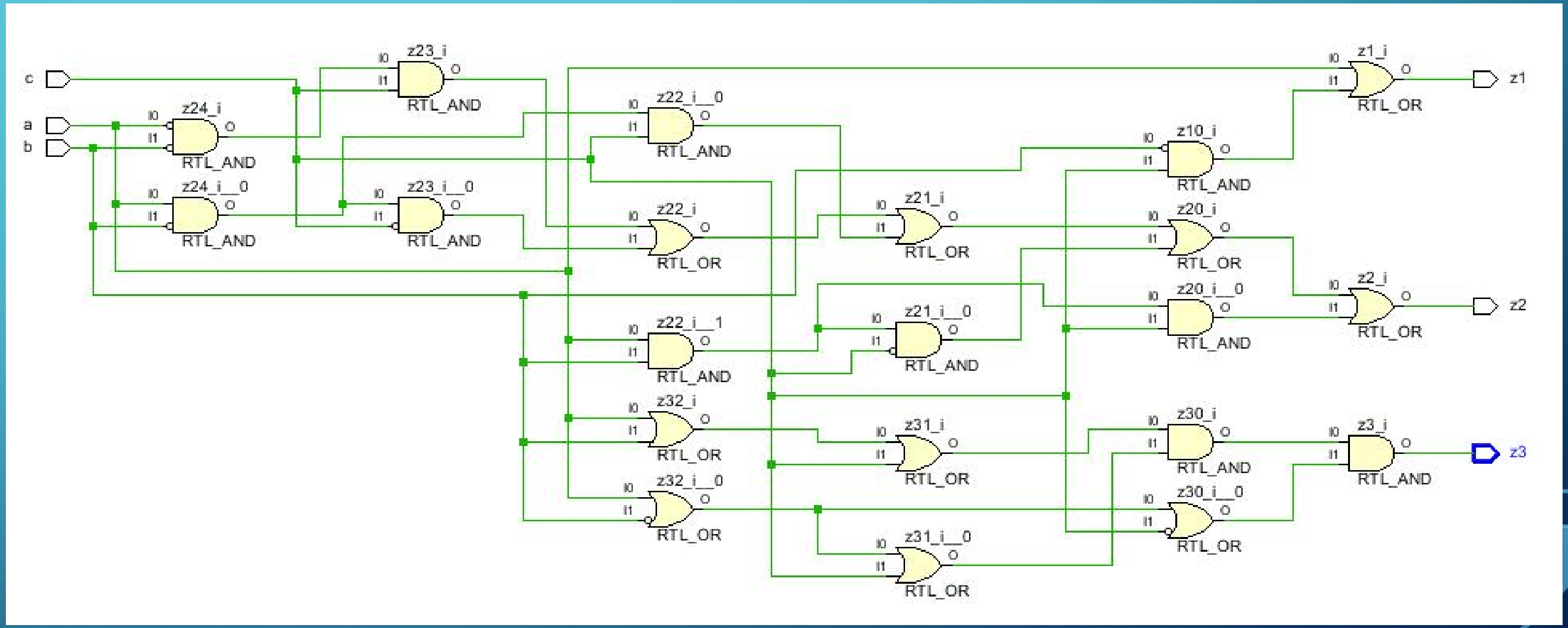
```
module Lab3_struct_design(
    input x,
    input y,
    output q1,
    output q2,
    output q3
    );

    wire o_xandy, o_xory;

    assign q1 = x;

    and and1(o_xandy, x, y);
    or or1(o_xory, x, y);

    or or2(q2, x, o_xandy);
    and and2(q3, x, o_xory);
endmodule
```

```
module Lab3_struct_design_sim();

    reg sim_x, sim_y;
    wire sim_q1, sim_q2, sim_q3;

    Lab3_struct_design uut(
        sim_x,
        sim_y,
        sim_q1,
        sim_q2,
        sim_q3
    );

    initial begin
        {sim_x, sim_y} = 2'b00;
        #10 {sim_x, sim_y} = 2'b01;
        #10 {sim_x, sim_y} = 2'b10;
        #10 {sim_x, sim_y} = 2'b11;
        #10 $finish;
    end
endmodule
```

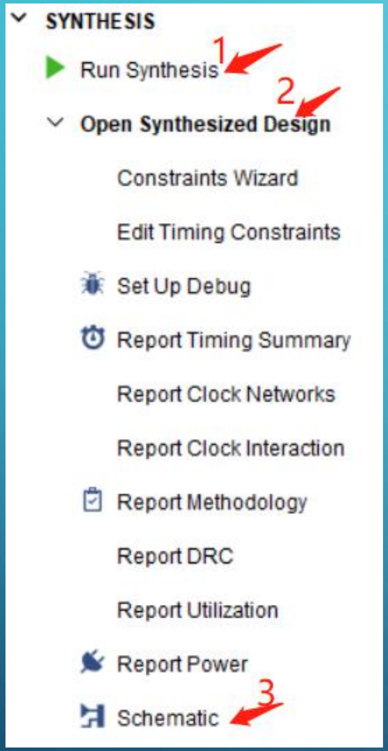


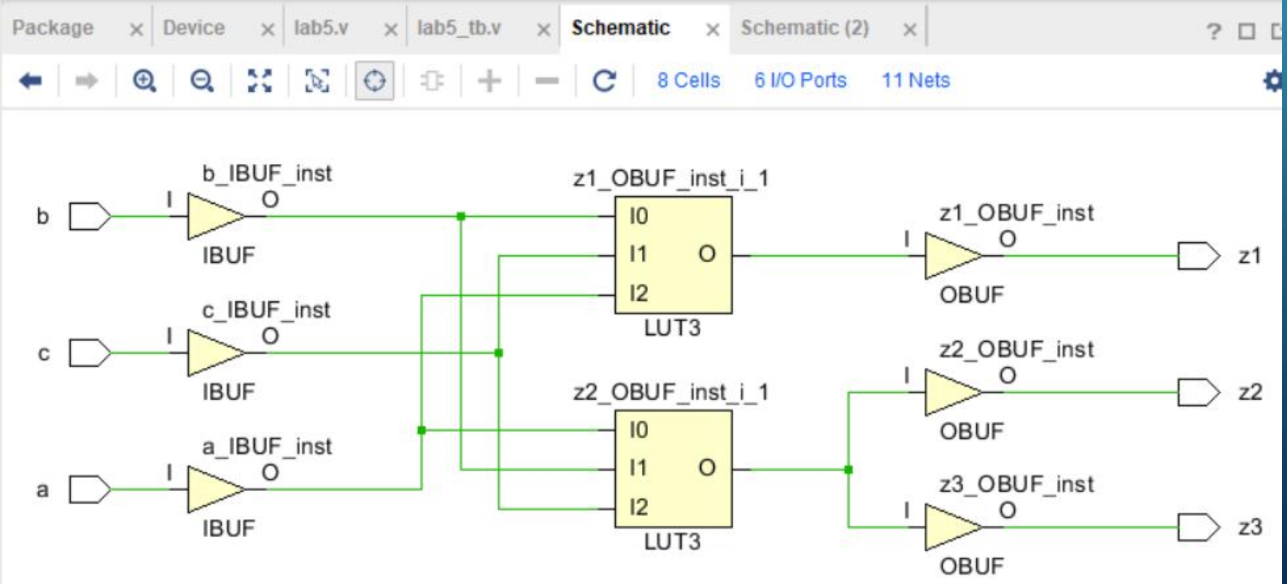
SCHEMATIC IN 'RTL ANALYSIS'





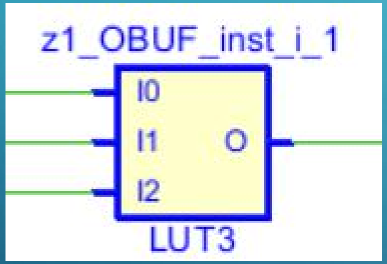
SCHEMATIC OF SYNTHESIZE(1)

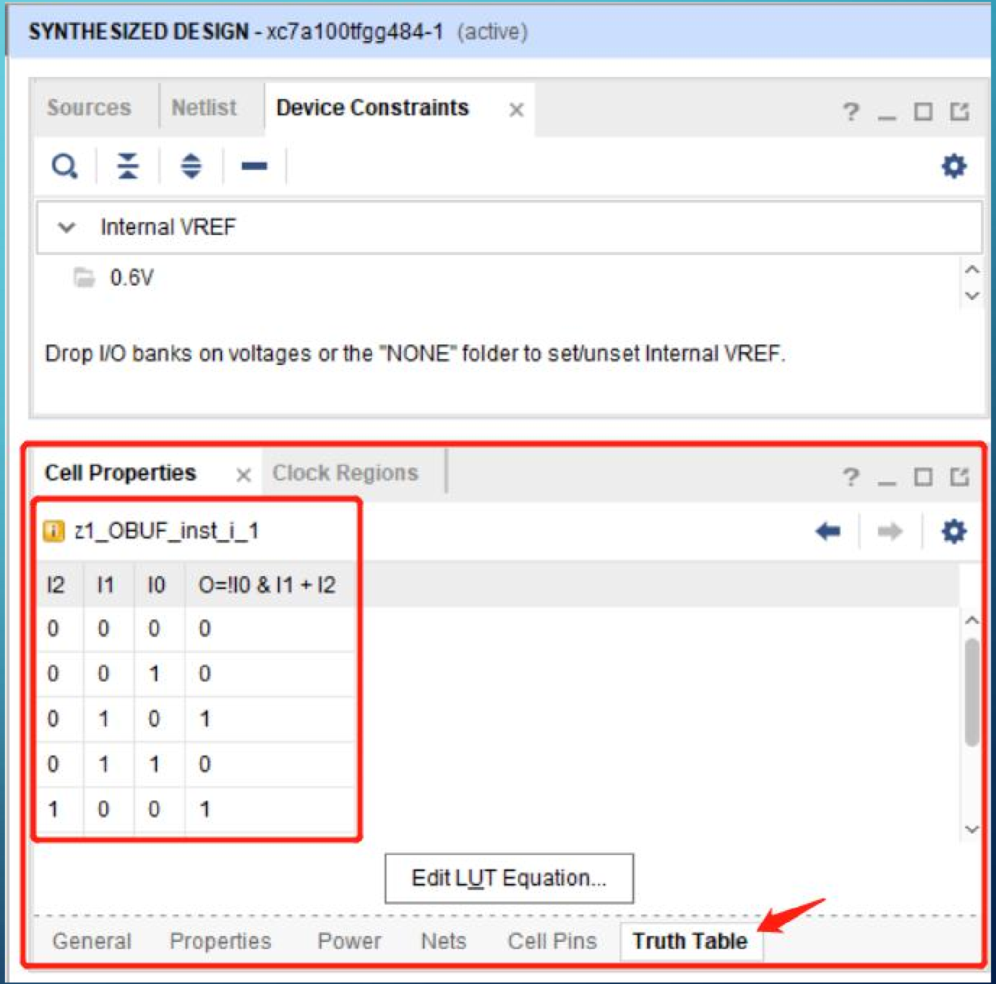




SCHEMATIC OF SYNTHESIZE(2)

- Double click the LUT in schematic window
- In the Cell Properties window , choose “Truth Table” , you can find the truth table of the cell





I2	I1	I0	O=!I0 & I1 + I2
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1

PRACTICES 1

- 1. Do the design (both x and y are 1 bit width) using two ways respectively:

  I) structure design with primitive gates

  II) data flow

  a) ~(x | y) b) ~x & ~y c) ~(x & y) d) ~x | ~y

- 2. create testbench, do simulation to verify function of the design
- 3. generate bitstream file, and test on the FPGA board

PRACTICES 2

- 4. think about : if the bitwise of x and y is more than 1 bit
  - 1) can the primitive gates be used in question 1
  - 2) will the result of bit-width calculation be the same as that on 1 bit width input