Digital Logic

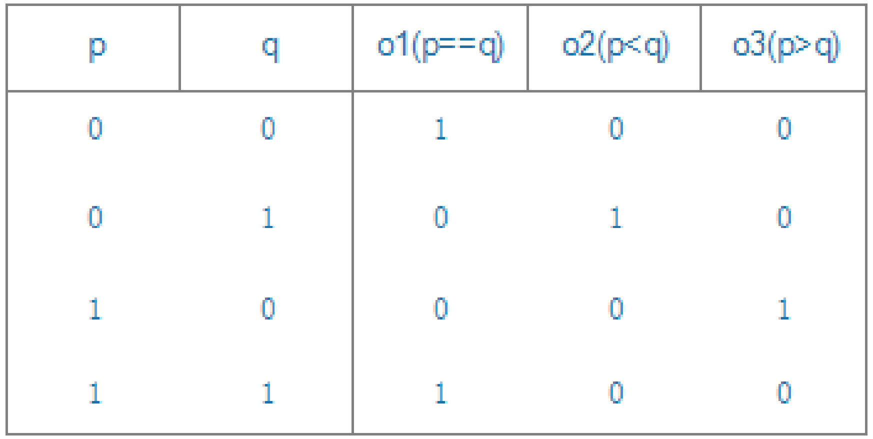
Lab5 Behavioral modeling in Verilog

Lab5

- 3 way of modeling
  - data-flow
  - structural-design
  - **Behavioral modeling**
- Verilog
  - initial VS **always**
  - **if else** VS conditional operator VS **case**
- Practices

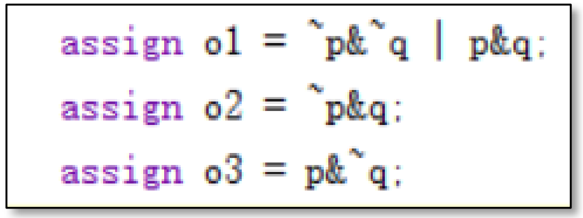
Magnitude comparator (1-bit) design

A magnitude comparator is a combinational circuit that compares two numbers p and q and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicate whether p = q, p > q, or p < q. (the bitwidth of both p and q are 1)



p	q	o1(p==q)	o2(p<q)	o3(p>q)
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
1	1	1	0	0

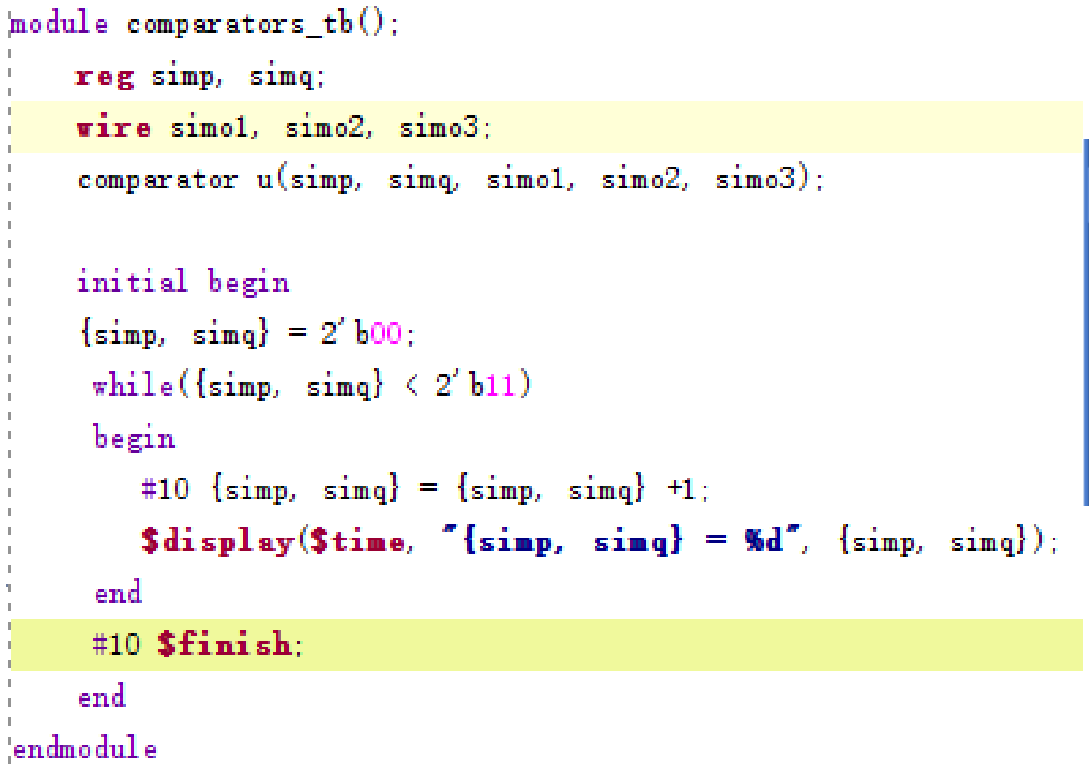
truth table for 1-bit comparator



```
assign o1 = ~p&~q | p&q;
assign o2 = ~p&q;
assign o3 = p&~q;
```

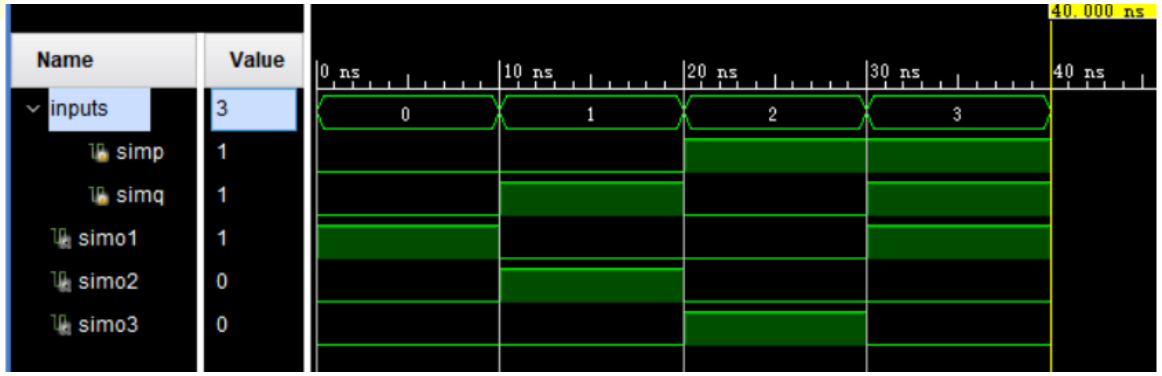
Magnitude comparator (1-bit) testbench & waveform

A magnitude comparator is a combinational circuit that compares two numbers p and q and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicate whether p = q, p > q, or p < q. (the bitwidth of both p and q are 1)



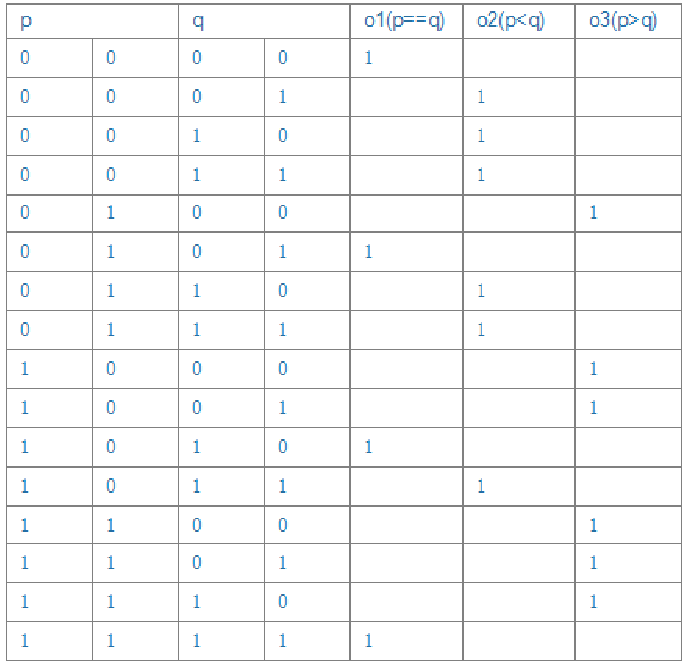
```
module comparators_tb();
    reg simp, simq;
    wire simo1, simo2, simo3;
    comparator u(simp, simq, simo1, simo2, simo3);

    initial begin
    {simp, simq} = 2'b00;
     while({simp, simq} < 2'b11)
     begin
        #10 {simp, simq} = {simp, simq} +1;
        $display($time, "{simp, simq} = %d", {simp, simq});
     end
     #10 $finish;
    end
endmodule
```



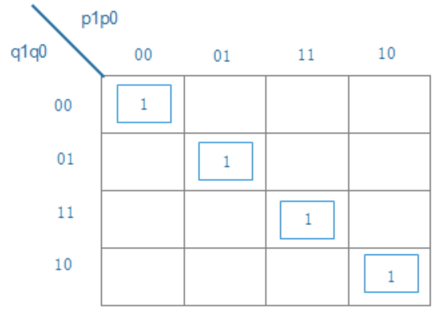
Magnitude comparator(2-bit) design

A magnitude comparator is a combinational circuit that compares two numbers p and q and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicate whether p = q, p > q, or p < q.
(the bitwidth of both p and q are 2)

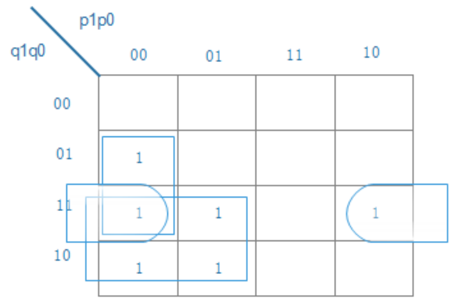


p		q		o1(p==q)	o2(p<q)	o3(p>q)
0	0	0	0	1		
0	0	0	1		1	
0	0	1	0		1	
0	0	1	1		1	
0	1	0	0			1
0	1	0	1	1		
0	1	1	0		1	
0	1	1	1		1	
1	0	0	0			1
1	0	0	1			1
1	0	1	0	1		
1	0	1	1		1	
1	1	0	0			1
1	1	0	1			1
1	1	1	0			1
1	1	1	1	1		

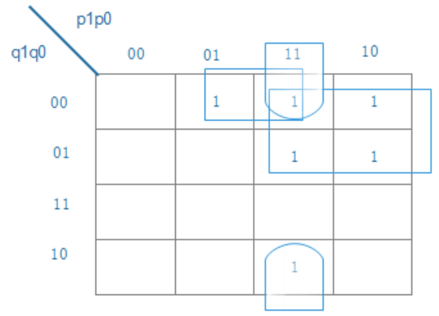
truth table for 2-bit comparator



o1 = p1'p0'q1'q0' + p1'p0q1'q0 + p1p0q1q0 + p1p0'q1q0'



o2=p1'q1+p1'p0'q0+p0'q1q0



o3=p1q1'+p0q1'q0'+p1p0q0'

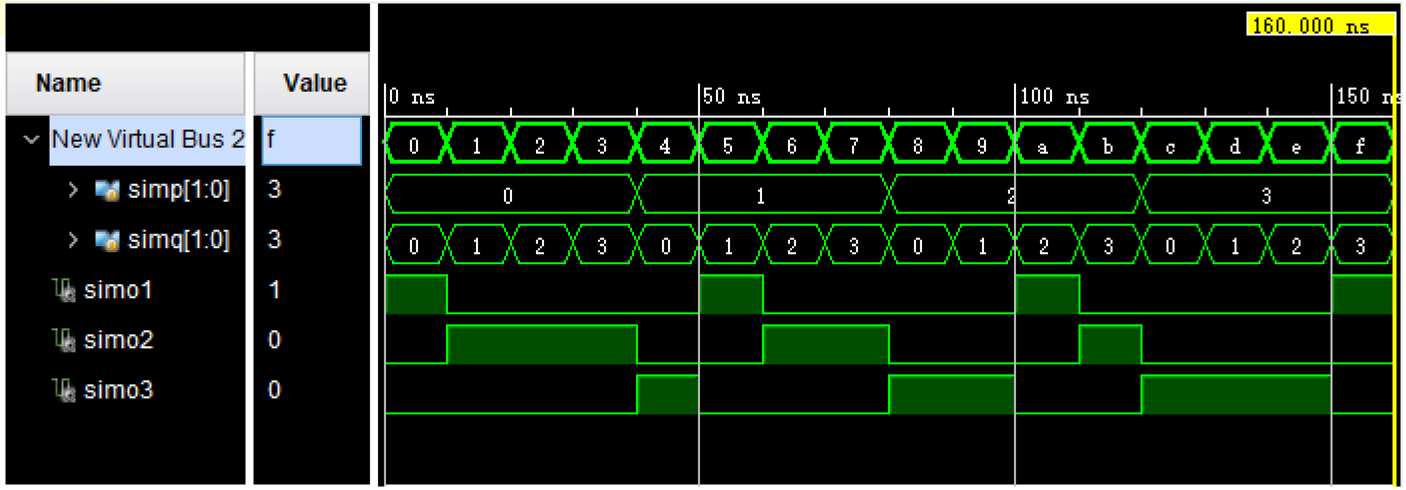
```
assign o1 = ~p[1]&~p[0]&~q[1]&~q[0] | ~p[1]&p[0]&~q[1]&q[0] | p[1]&p[0]&q[1]&q[0] | p[1]&~p[0]&q[1]&~q[0];
assign o2 = ~p[1]&q[1] | ~p[1]&~p[0]&q[0] | ~p[0]&q[1]&q[0];
assign o3 = p[1]&~q[1] | p[0]&~q[1]&~q[0] | p[1]&p[0]&~q[0];
```

Magnitude comparator(2-bit) testbench & waveform

A magnitude comparator is a combinational circuit that compares two numbers p and q and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicate whether p = q, p > q, or p < q. (the bitwidth of both p and q are 2)

```
module comparators_tb();
    reg[1:0] simp, simq;
    wire simo1, simo2, simo3;
    comparator u(simp, simq, simo1, simo2, simo3);

    initial begin
    {simp, simq} = 4'b0000;
     while({simp, simq} < 4'b1111)
     begin
        #10 {simp, simq} = {simp, simq} +1;
        $display($time, "{simp, simq} = %d", {simp, simq});
     end
     #10 $finish;
    end
endmodule
```



Name	Value
New Virtual Bus 2	f
simp[1:0]	3
simq[1:0]	3
simo1	1
simo2	0
simo3	0

How about other combination circuit?

Design Procedure

1. Specification: From the specifications, determine the inputs, outputs, and their symbols.
2. Formulation: Derive the truth table (functions) from the relationship between the inputs and outputs
3. Optimization: Derive the simplified Boolean functions for each output.
4. Logic diagram (optional): Draw a logic diagram for the resulting circuits using AND, OR, and inverters. (Or using required technology mapping)

- In lab practice, Step 4 can be obtained from RTL Analysis schematic.
- Besides, we should also do the simulation to verify the design.

Behavioral modeling(1)

Behavioral Models: Higher level of modeling where behavior of logic is modeled.

- An **always** block can include **a sensitivity list** in which any of these signals change will trigger the always block execution
  - **@(*)** , **@*** :
    - It is sensitive to changes in all input variables in the following statement block.
  - **@(signal1, signal2, …, signalx)** , **@(signal1 or signal2 or … or signalx)**:
    - It is only sensitive to changes of the singnals in the sensitivity list.

```
wire out1;
assign out1 = in1 & in2;            //data-flow
```

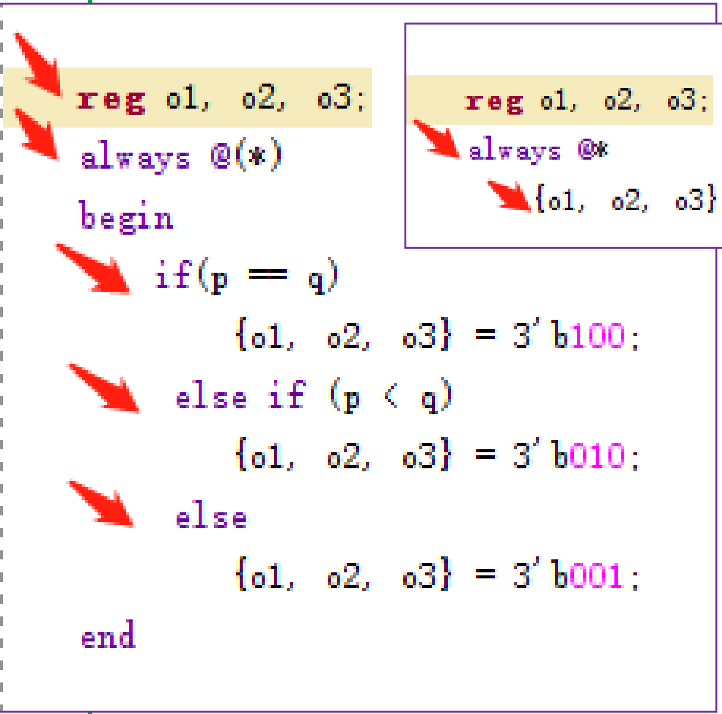
```
wire out1;
and uand1(out1, in1, in2);          //structure-design
```

```
reg out1;
always @(in1, in2)       //behavioral-models
      out1 = in1 & in2;
```

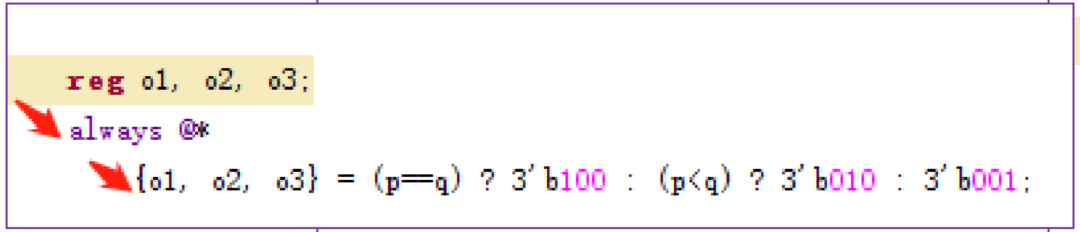
Behavioral modeling(2)

An **always** block can include **a sensitivity list** in which any of these signals change will trigger the always block execution.

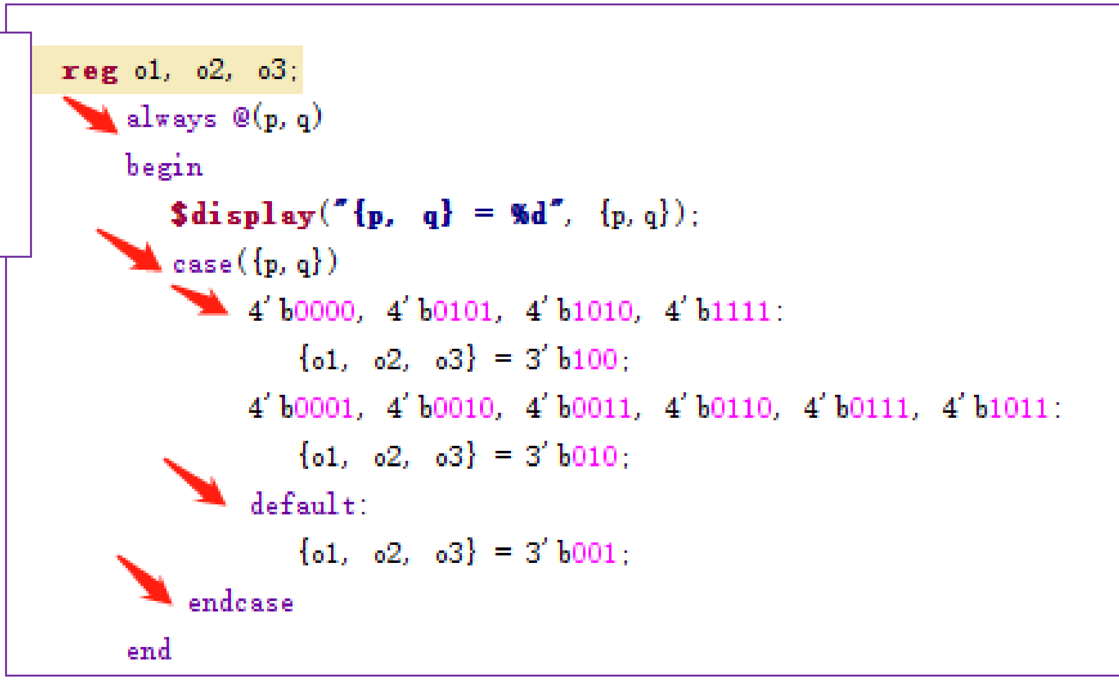
- The data type of the assigned object MUST be reg
- ‘**if else**’ and ‘**case**’ could ONLY be used as part of ‘initial’ or ‘always’
- ‘**if else**’ VS **conditional operator** VS ‘**case**’



```
reg o1, o2, o3;
always @(*)
begin
    if(p == q)
        {o1, o2, o3} = 3'b100;
     else if (p < q)
        {o1, o2, o3} = 3'b010;
     else
        {o1, o2, o3} = 3'b001;
end
```



```
reg o1, o2, o3;
always @*
    {o1, o2, o3} = (p==q) ? 3'b100 : (p<q) ? 3'b010 : 3'b001;
```



```
reg o1, o2, o3;
always @(p, q)
begin
   $display("{p, q} = %d", {p,q});
   case({p,q})
       4'b0000, 4'b0101, 4'b1010, 4'b1111:
           {o1, o2, o3} = 3'b100;
       4'b0001, 4'b0010, 4'b0011, 4'b0110, 4'b0111, 4'b1011:
           {o1, o2, o3} = 3'b010;
       default:
           {o1, o2, o3} = 3'b001;
   endcase
end
```

Behavioral modeling(3)

- initial VS always (ATTENTION!!!!)

- initial :
  - **“initial” is used ONLY in testbench;**
  - **statements in “initial” block execute ONLY once;**

- always:
  - **“always” could be used in both testbench and design module;**
  - **statements in “always” block execute repeatedly as long as the trigger condition(s) is(are) met;**

Practice1 (optional)

- Implement a 2-bit Magnitude Comparator using Behavioral Modeling
  - Using “if else” or “case”
  - Write the testbench in Verilog to verify the functionality of design

Practice2

- Implement a Rock-Paper-Scissors Comparator (Part1)
  - Define the input variables. In a game of rock-paper-scissors, there are two players(p1 and p2), each of whom chooses one of three options: rock, paper, or scissors. Therefore, the input variables for the comparator would be two binary numbers representing the choices of the two players.
  - Design the circuit to show if player1 wins(o1), player2 wins(o2) or there's a tie(o3). For example, if player 1 chooses paper and player 2 chooses scissors, the output should indicate that player 2 wins. You should design using:
    1. Dataflow design mode. For that, you need to fill up a truth table by listing all possible combinations of the input variables and their corresponding outputs
    2. Behavioral design mode. Use always block and else-if or case.
  - Build the testbench to test the circuit.
  - Add the constraint file and generate the bitstream and program the device to test the function, P1, P2 are switches, O1~O3 are LEDs
- Please backup your code so that you could reuse it for next lab

Practices(3) (optional)

Design a circuit that can find the 1’s complement and 2’s complement of a 3-bit input binary number. The output is 3 bits. Use an additional 1-bit input port called switch to change between the two types of complements: when the switch is 0, the output is the 1’s complement; when the switch is 1, the output is the 2’s complement.

- Write the corresponding truth table.
- Use behavioural modelling to do the design, “if else” or “case” is suggested.
- Write the testbench in Verilog to verify the functionality of design
- Design the constraint file and generate the bitstream and program the device to test the function

Tips: wire vs reg(1)

wire (net)	reg (register)
1) **Can't store info**, MUST be driven (such as continuous assignment) 2) The input and output port of module is wire by default. The input port **MUST be wire** 3) The type of left-hand side of assignment in initial or always block Can NOT be wire.	1) **Can sore info**, keep its value untile be changed. 2) The type of left-hand side of assignment in initial or always block **MUST be reg**. 3) The variable bind to output port Can NOT be reg.

Tips: reg vs wire(2)

complete the following table. If the data type is reg, tick the cell corresponding to reg. If the data type is wire, tick the cell corresponding to wire.

type	demo	wire	reg
input	module tx(input a, ...); ? module tx(input reg a, ...);		
output	module tx(output b, ...); ? module tx(output reg b,...);		

variable
be assigned in continuous mode
	wire x; ? reg x;
	assign x = a & b;		

variable
be assigned in procedure mode
	wire y; reg z; ? wire y,z; ? reg y,z; ? reg y; wire z;
	inital y= y+1; always @* z = a | b;		

variable used to bind input	wire inx; ? reg inx? not unot1(out, inx);		
variable used to bind output	wire outx; ? reg outx? not unot2(outx, in1);		

Tips: reg vs wire(3)

type	demo	wire	reg
input	module tx(**input a**, ...); ? module tx(input reg a, ...);	√	
output	module tx(**output b**, ...); ? module tx(**output reg b**,...);	√	√

variable
be assigned in continuous mode
	wire x; ? reg x;
	assign x = a & b;
		√	

variable
be assigned in procedure mode
	wire y; reg z; ? wire y,z; ? **reg y,z**; ? reg y; wire z;
	inital y= y+1; always @* z = a | b;
			√

variable
used to bind input
	wire inx; ? **reg inx;**
	not unot1(out, inx);
		√	√

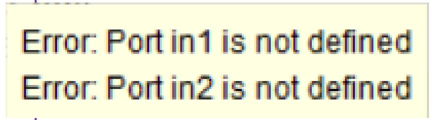
variable
used to bind output
	wire outx; ? reg outx;
	not unot2(**outx**, in1);
		√	

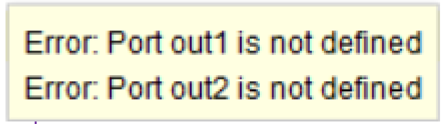
Summary: **ONLY in "inital" or "always" block, the type of assigned object MUST be "reg", otherwise its data type is "wire".**

TIPS: Port defination in Verilog

```
//style 1 : OK
module sub_wr( in1, in2, out1, out2 ) ;
input      in1, in2;
output     out1, out2;
endmodule
```

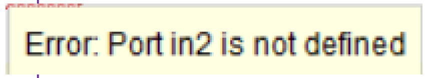
```
//style 3 : Error
module sub_wr( /*port list is empty*/ );
input      in1, in2;
output     out1, out2;
endmodule
```

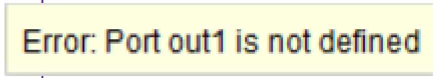

Error: Port in1 is not defined
Error: Port in2 is not defined


Error: Port out1 is not defined
Error: Port out2 is not defined

```
//style 2 : OK
module sub_wr(
  input   in1, in2,
  output  out1, out2 );
endmodule
```

```
//style 4: Error
module sub_wr( in1, out2 );
input      in1, in2;
output     out1, out2;
endmodule
```

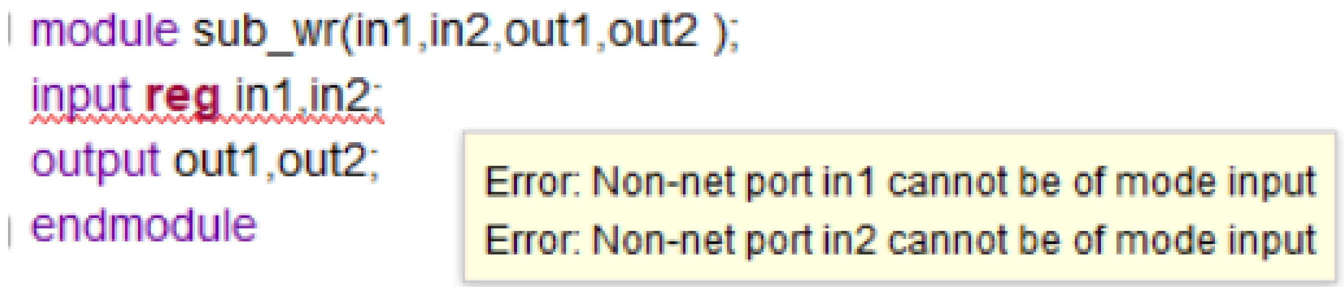

Error: Port in2 is not defined


Error: Port out1 is not defined

For the port defination, both style1 and style 2 are acceptable.

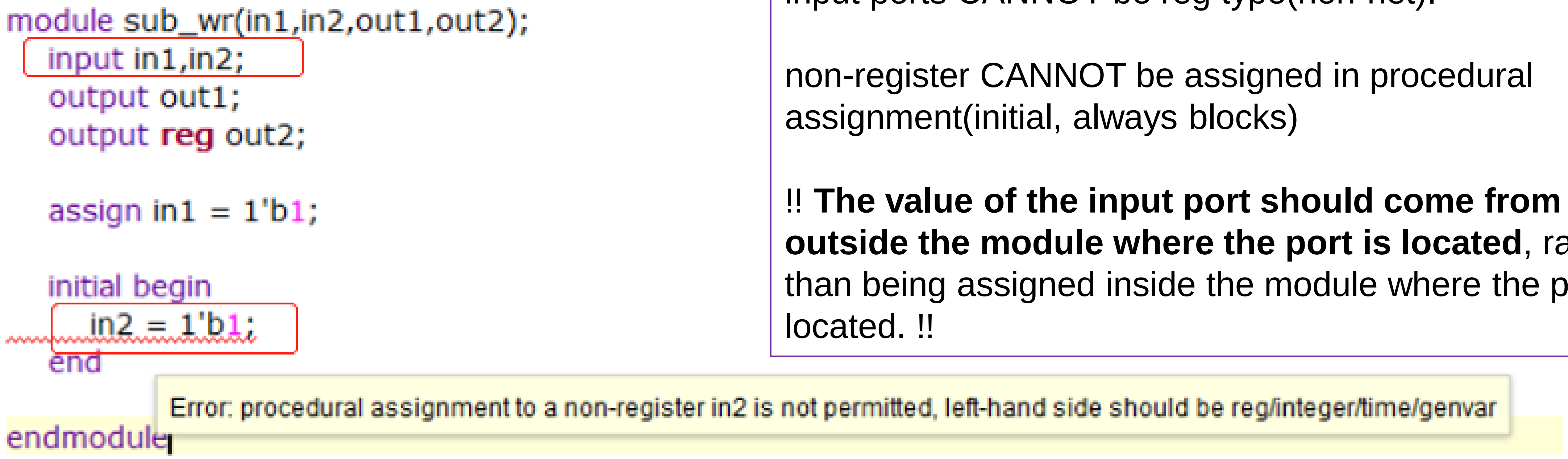
While the module has ports but the “port list” is empty, or while the ports in the “port list” are inconsistent with the actual port, it is a grammar error in Verilog.

TIPS： input port



```
module sub_wr(in1,in2,out1,out2 );
input reg in1,in2;
output out1,out2;
endmodule
```

```
Error: Non-net port in1 cannot be of mode input
Error: Non-net port in2 cannot be of mode input
```



```
module sub_wr(in1,in2,out1,out2);
   input in1,in2;
   output out1;
   output reg out2;

   assign in1 = 1'b1;

   initial begin
      in2 = 1'b1;
   end

endmodule
```

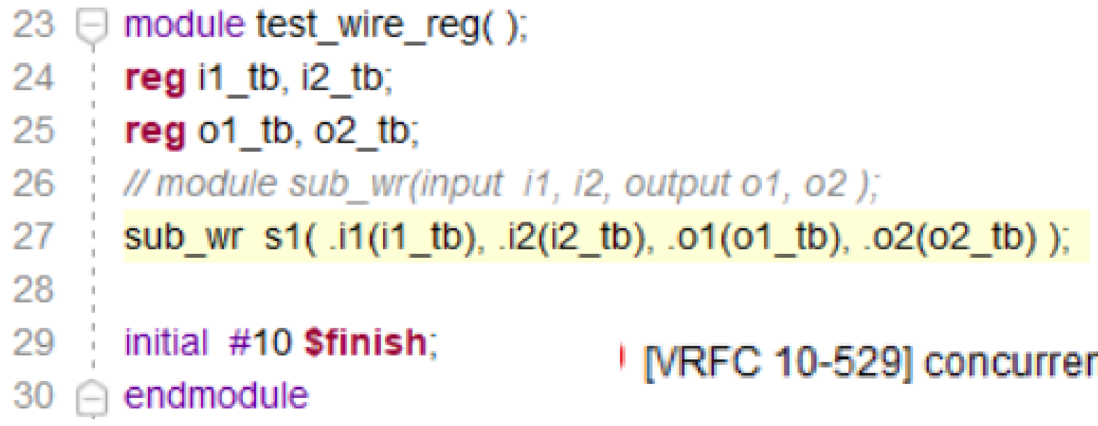
```
Error: procedural assignment to a non-register in2 is not permitted, left-hand side should be reg/integer/time/genvar
```

input ports CANNOT be reg type(non-net).

non-register CANNOT be assigned in procedural assignment(initial, always blocks)

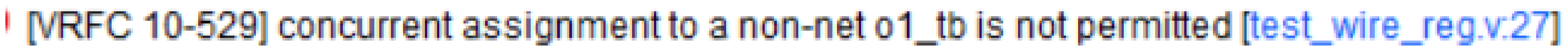
!! **The value of the input port should come from outside the module where the port is located**, rather than being assigned inside the module where the port is located. !!

Tips: output ports & the variable bind with output ports

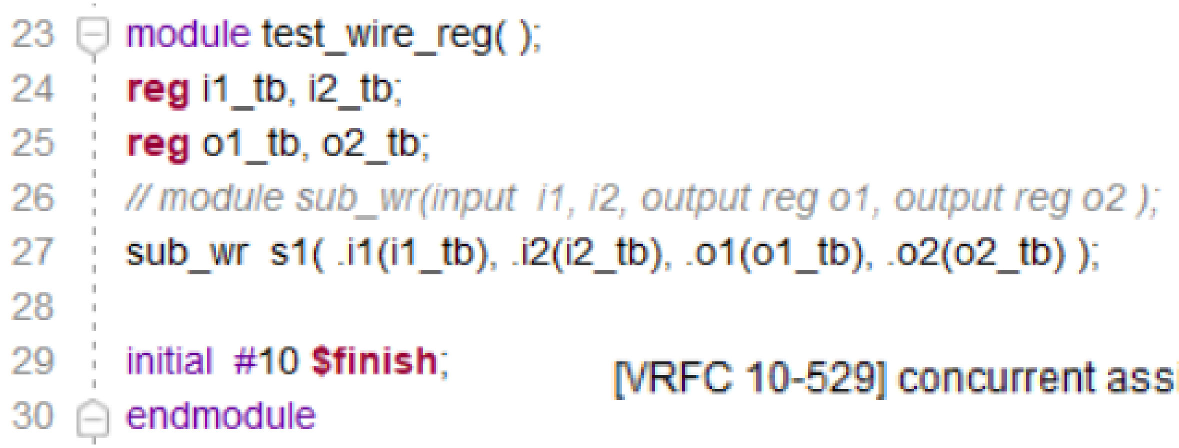


```
23 module test_wire_reg( );
24 reg i1_tb, i2_tb;
25 reg o1_tb, o2_tb;
26 // module sub_wr(input  i1, i2, output o1, o2 );
27 sub_wr  s1( .i1(i1_tb), .i2(i2_tb), .o1(o1_tb), .o2(o2_tb) );
28
29 initial  #10 $finish;
30 endmodule
```

Error:
output port is wire(default) while the **variable bind with it is reg**

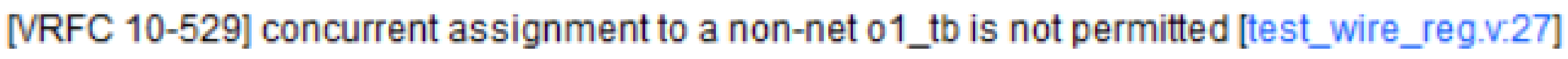


```
[VRFC 10-529] concurrent assignment to a non-net o1_tb is not permitted [test_wire_reg.v:27]
```



```
23 module test_wire_reg( );
24 reg i1_tb, i2_tb;
25 reg o1_tb, o2_tb;
26 // module sub_wr(input  i1, i2, output reg o1, output reg o2 );
27 sub_wr  s1( .i1(i1_tb), .i2(i2_tb), .o1(o1_tb), .o2(o2_tb) );
28
29 initial  #10 $finish;
30 endmodule
```

Error:
output port is reg while the **variable bind with it is reg**



```
[VRFC 10-529] concurrent assignment to a non-net o1_tb is not permitted [test_wire_reg.v:27]
```