DIGITAL DESIGN

LAB5 BEHAVIORAL MODELING

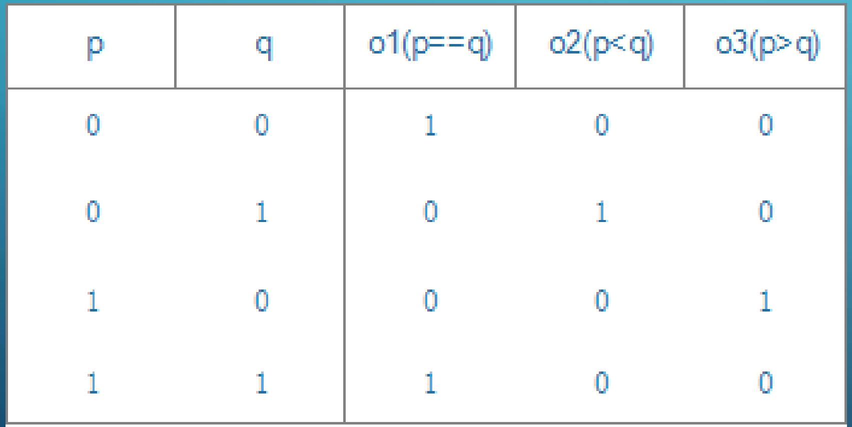
2022 SUMMER TERM

LAB5

- Behavioral modeling
- Verilog
  - Structure modeling VS Behavioral modeling
  - initial VS always
  - if else VS conditional operator VS case
  - wire VS reg
  - Non-blocking assignments VS blocking assignments
- Practice

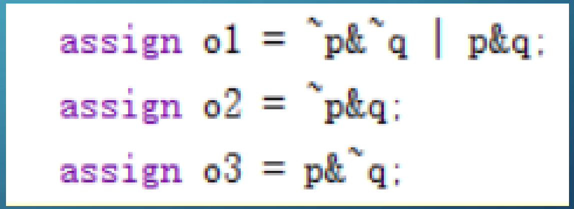
MAGNITUDE COMPARATOR (1-BIT)

- 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 $p < q$.
- Use dataflow modeling



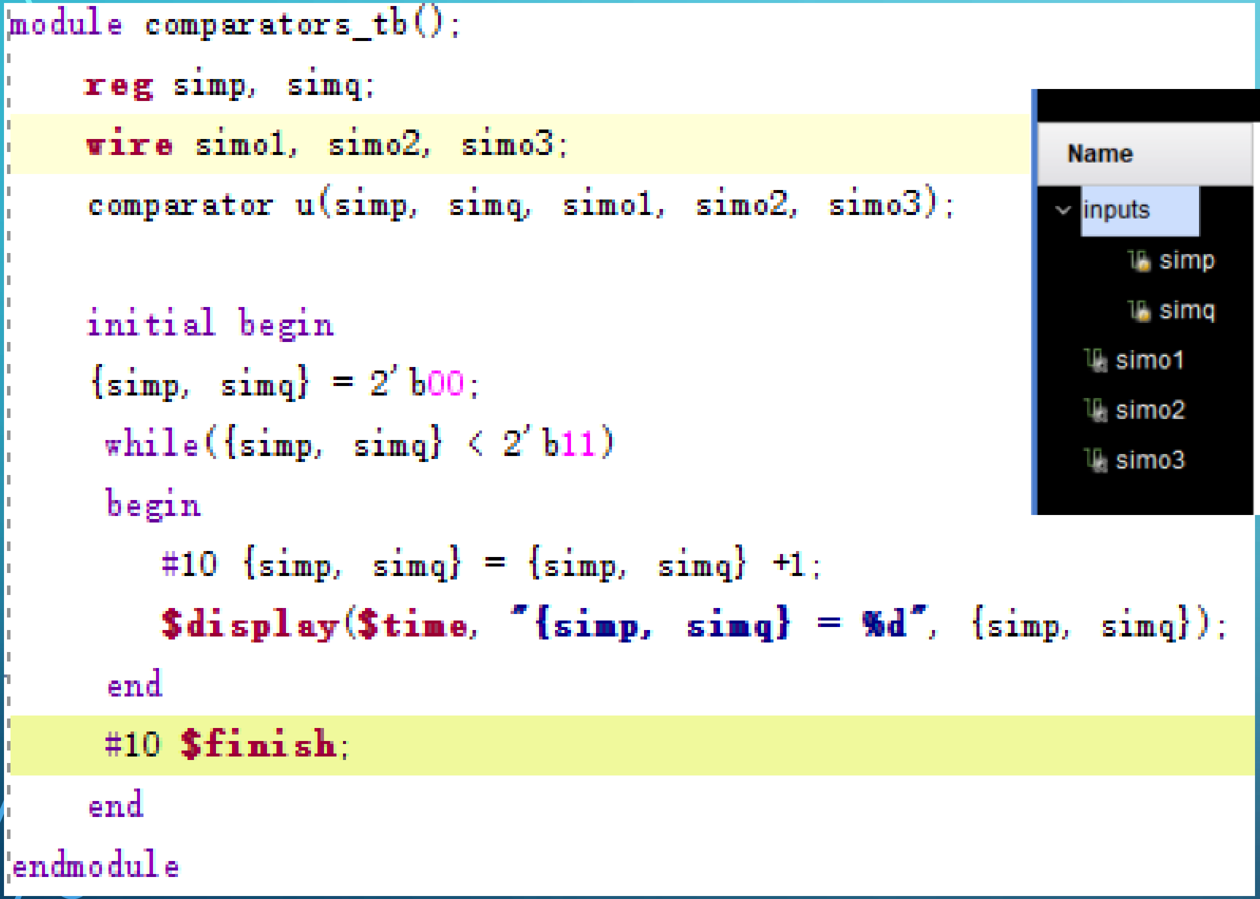
p	q	o1(p==q)	o2(p<q)	o3(p>q)
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
1	1	1	0	0

truth table for 1-bit comparator

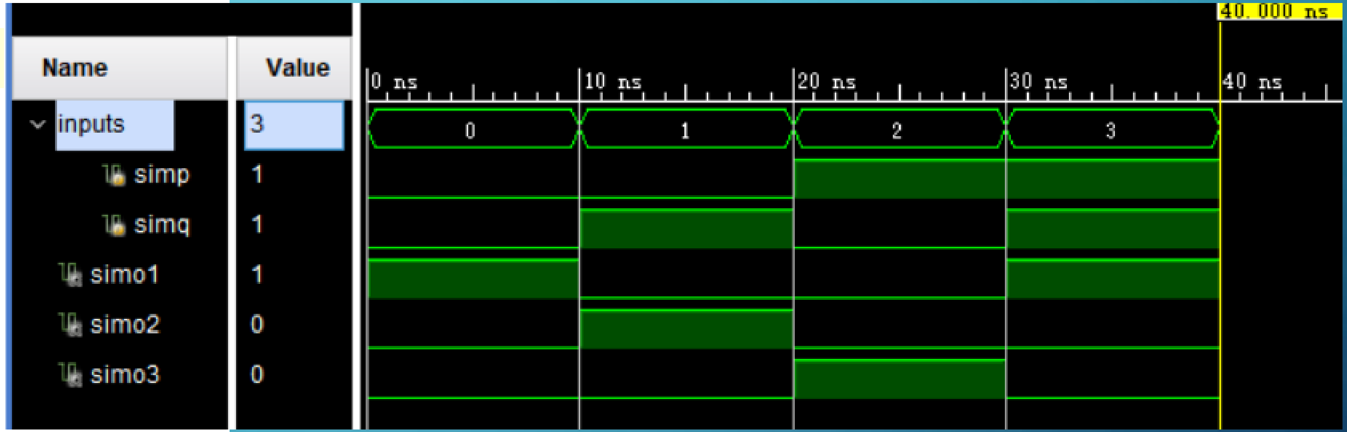


```
assign o1 = ~p&~q | p&q;
assign o2 = ~p&q;
assign o3 = p&~q;
```

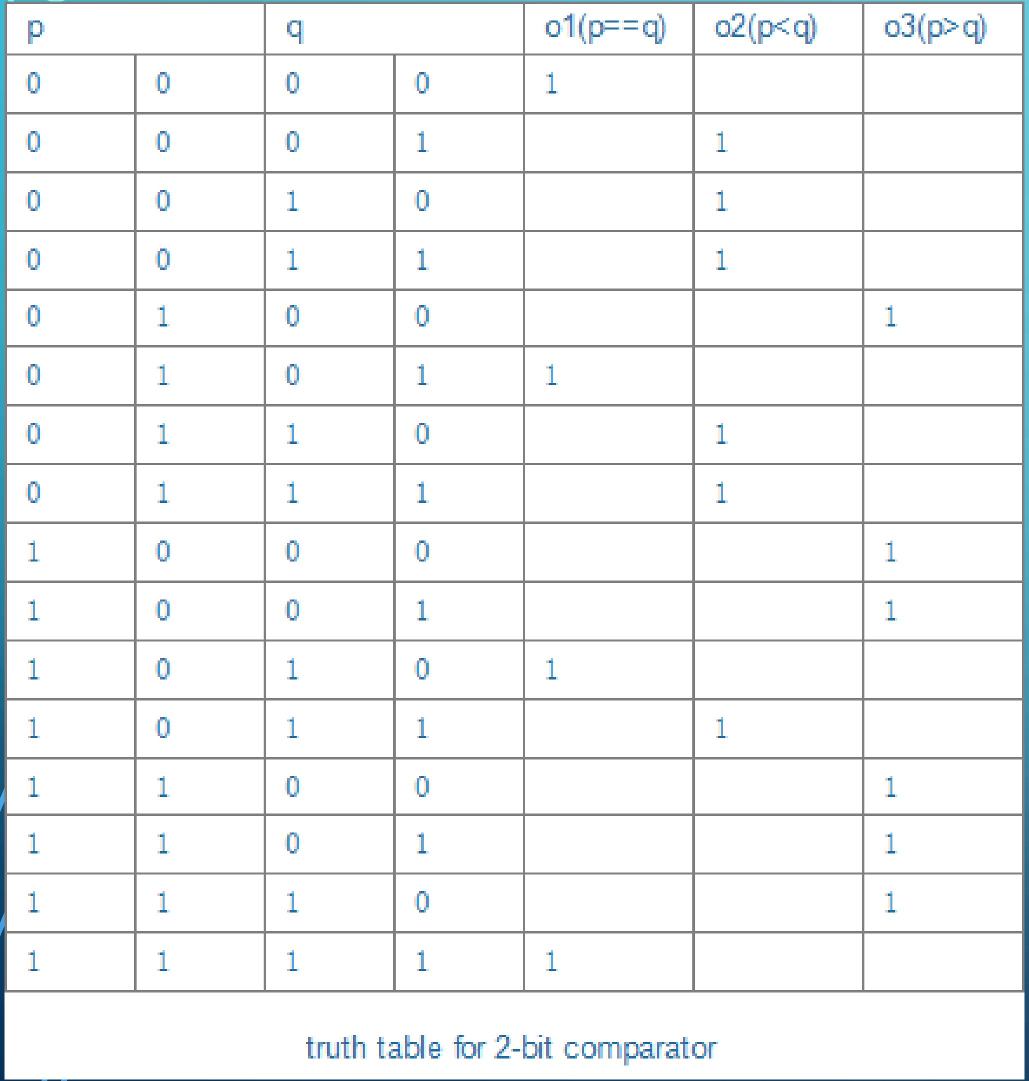
MAGNITUDE COMPARATOR(1-BIT)



```
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
```

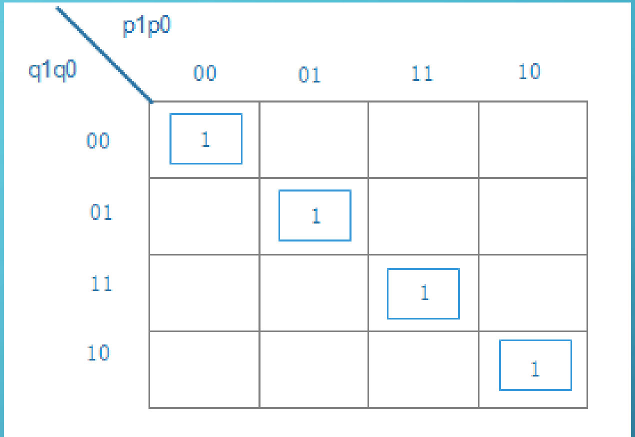


MAGNITUDE COMPARATOR(2-BIT)



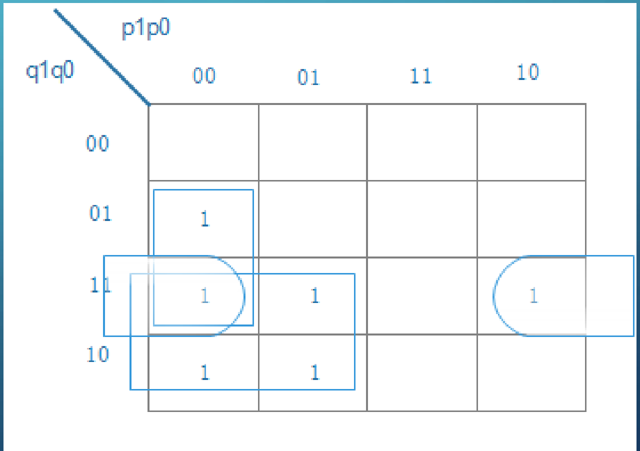
p		q		o1(p==q)	o2(p<q)	o3(p>q)
0	0	0	0	1		
0	0	0	1		1	
0	0	1	0		1	
0	0	1	1		1	
0	1	0	0			1
0	1	0	1	1		
0	1	1	0		1	
0	1	1	1		1	
1	0	0	0			1
1	0	0	1			1
1	0	1	0	1		
1	0	1	1		1	
1	1	0	0			1
1	1	0	1			1
1	1	1	0			1
1	1	1	1	1		

truth table for 2-bit comparator



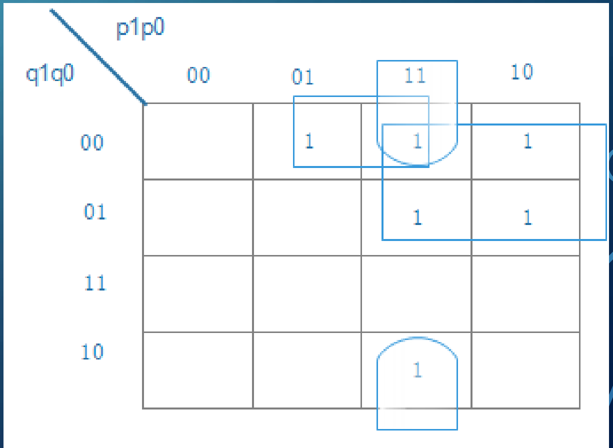
q1q0 \ p1p0	00	01	11	10
00	1			
01		1		
11			1	
10				1

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



q1q0 \ p1p0	00	01	11	10
00				
01	1			
11	1	1		1
10	1	1		

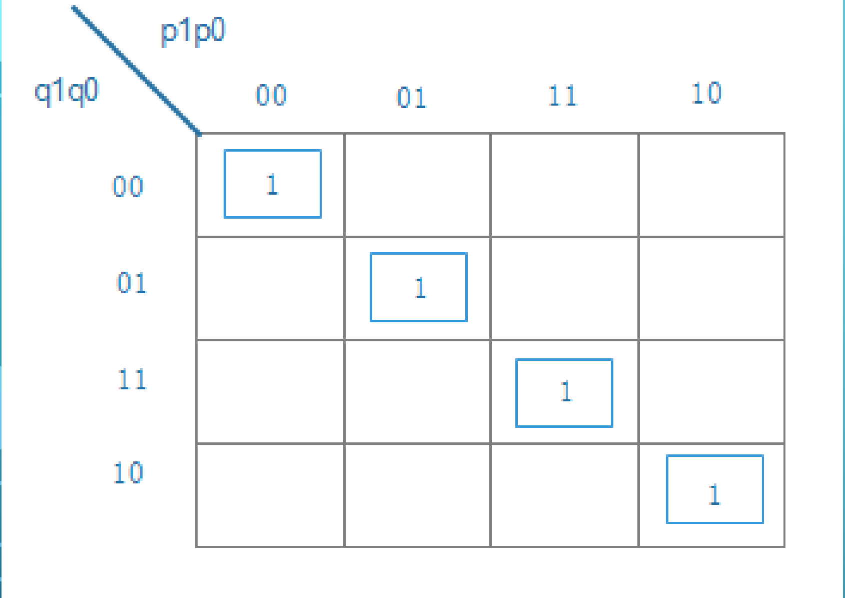
o2=p1'q1+p1'p0'q0+p0'q1q0



q1q0 \ p1p0	00	01	11	10
00		1	1	1
01			1	1
11				
10			1	

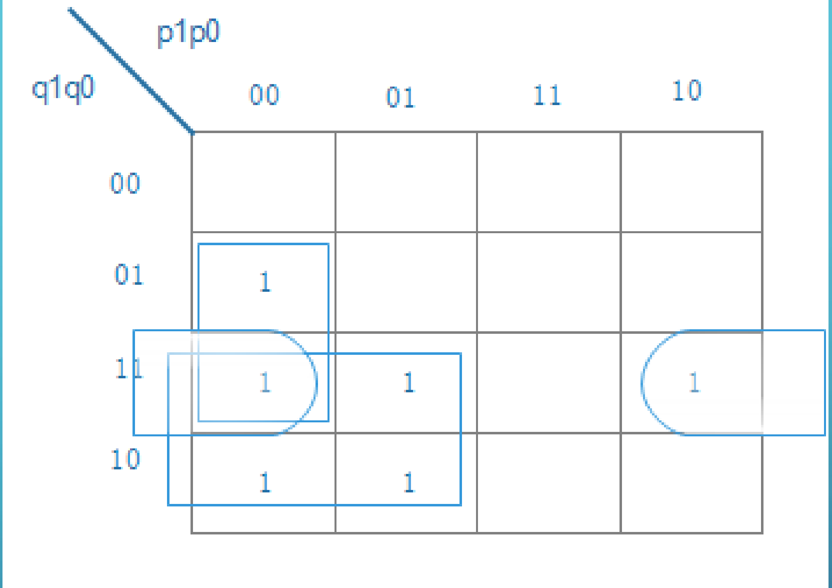
o3=p1q1'+p0q1'q0'+p1p0q0'

MAGNITUDE COMPARATOR(2-BIT)



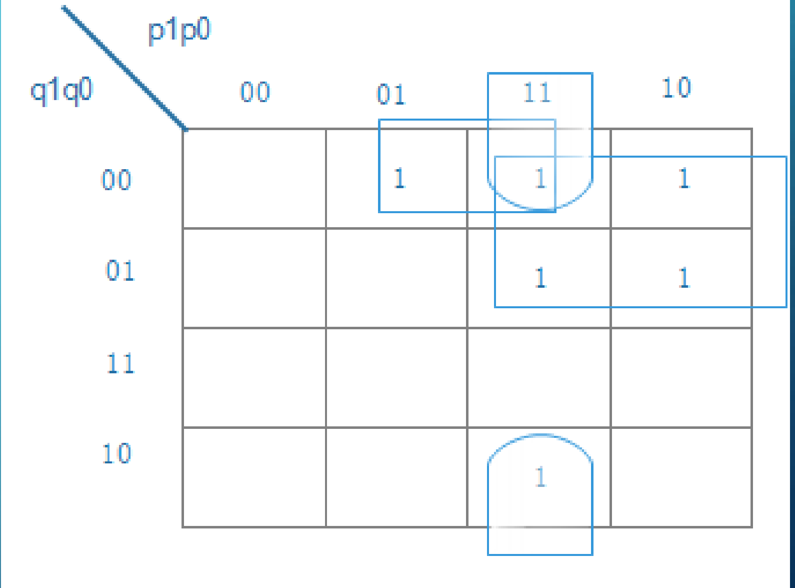
q1q0 \ p1p0	00	01	11	10
00	1			
01		1		
11			1	
10				1

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



q1q0 \ p1p0	00	01	11	10
00				
01	1			
11	1	1		1
10	1	1		

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

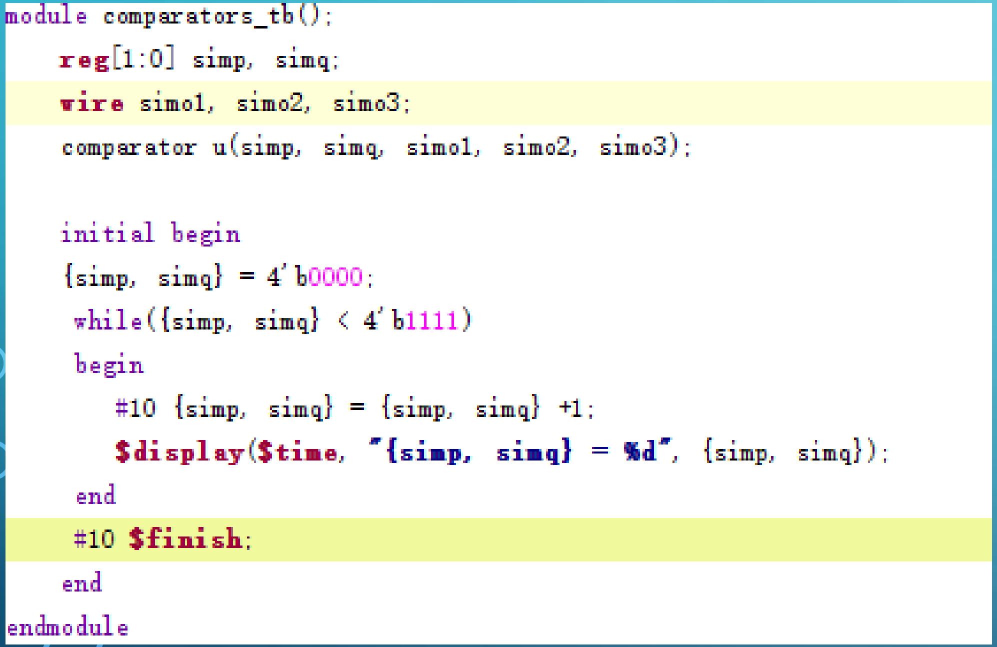


q1q0 \ p1p0	00	01	11	10
00		1	1	1
01			1	1
11				
10			1	

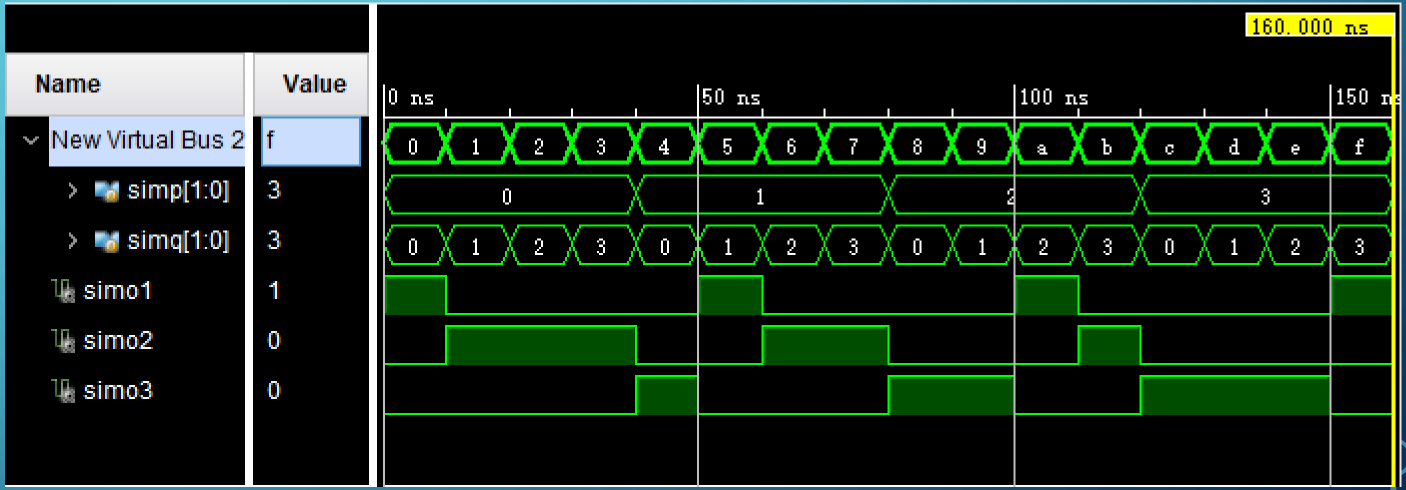
o3=p1q1'+p0q1'q0'+p1p0q0'

```
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)



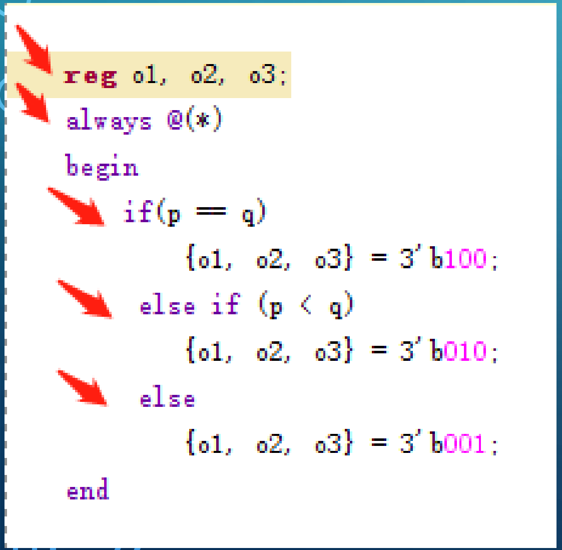
```
module comparators_tb();
    reg[1:0] simp, simq;
    wire simo1, simo2, simo3;
    comparator u(simp, simq, simo1, simo2, simo3);

    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
```



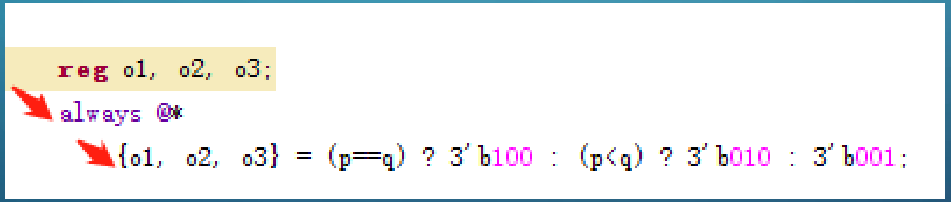
Name	Value
New Virtual Bus 2	f
simp[1:0]	3
simq[1:0]	3
simo1	1
simo2	0
simo3	0

BEHAVIORAL MODELING(1)

- **initial** VS **always** : statements in initial block execute only once; statements in always block execute repeatedly once the trigger condition is satisfied.
- 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 singnales in the sensitivity list.
- **‘if else’** VS **conditional operator** VS **‘case’**

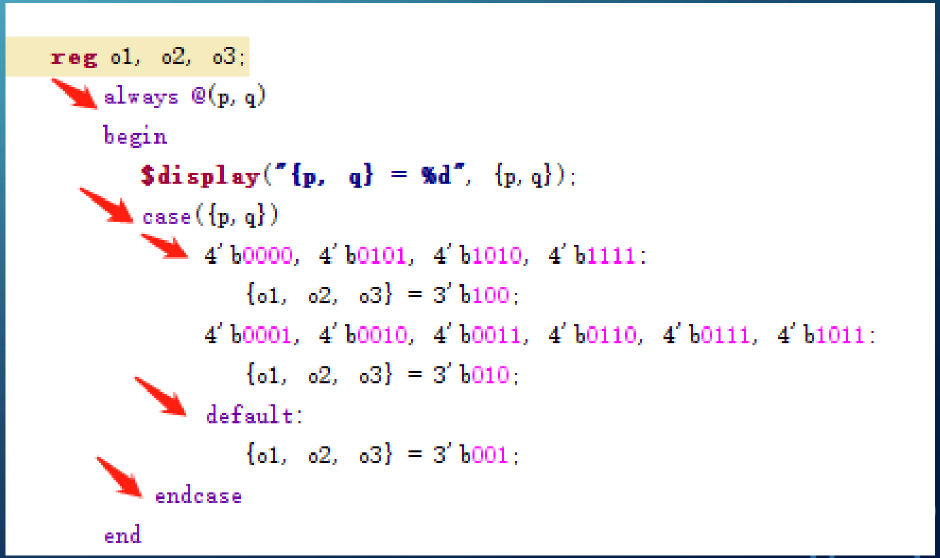


```
reg o1, o2, o3;
always @(*)
begin
    if(p == q)
        {o1, o2, o3} = 3'b100;
    else if (p < q)
        {o1, o2, o3} = 3'b010;
    else
        {o1, o2, o3} = 3'b001;
end
```



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

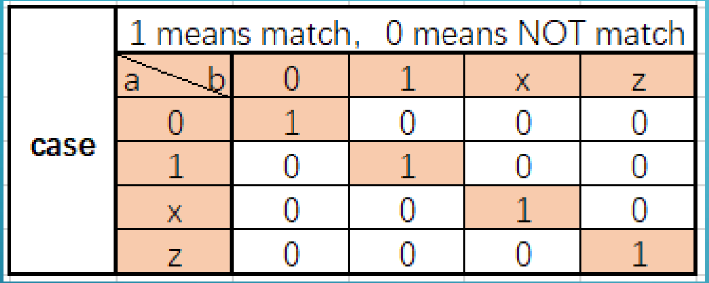
(Condition) ? A : B



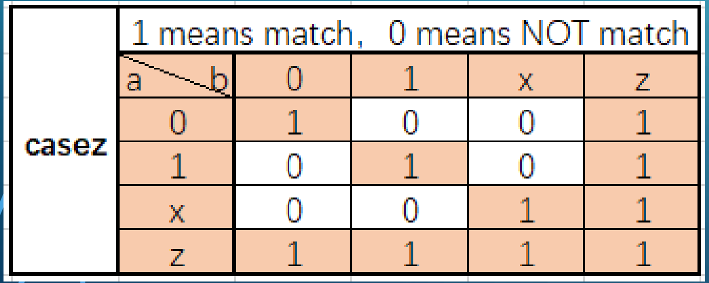
```
reg o1, o2, o3;
always @(p, q)
begin
    $display("{p, q} = %d", {p,q});
    case({p, q})
        4'b0000, 4'b0101, 4'b1010, 4'b1111:
            {o1, o2, o3} = 3'b100;
        4'b0001, 4'b0010, 4'b0011, 4'b0110, 4'b0111, 4'b1011:
            {o1, o2, o3} = 3'b010;
        default:
            {o1,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

BEHAVIORAL MODELING(2)

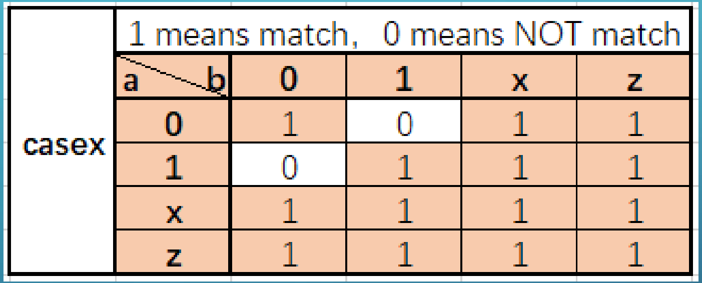
- case VS casez VS casex
  - For example : using case, casez, casex to match 'a' and 'b'



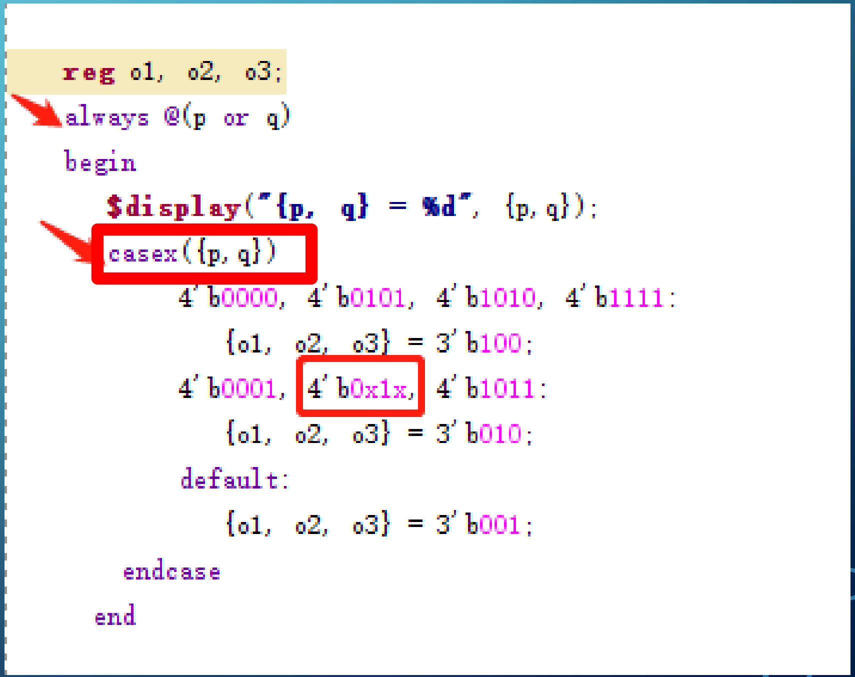
case	1 means match, 0 means NOT match				
	a \ b	0	1	x	z
	0	1	0	0	0
	1	0	1	0	0
	x	0	0	1	0
	z	0	0	0	1



casez	1 means match, 0 means NOT match				
	a \ b	0	1	x	z
	0	1	0	0	1
	1	0	1	0	1
	x	0	0	1	1
	z	1	1	1	1



casex	1 means match, 0 means NOT match				
	a \ b	0	1	x	z
	0	1	0	1	1
	1	0	1	1	1
	x	1	1	1	1
	z	1	1	1	1



```
reg o1, o2, o3;
always @(p or q)
begin
   $display("{p, q} = %d", {p,q});
  casex({p,q})
       4'b0000, 4'b0101, 4'b1010, 4'b1111:
           {o1, o2, o3} = 3'b100;
       4'b0001, 4'b0x1x, 4'b1011:
           {o1, o2, o3} = 3'b010;
       default:
           {o1, o2, o3} = 3'b001;
    endcase
  end
```

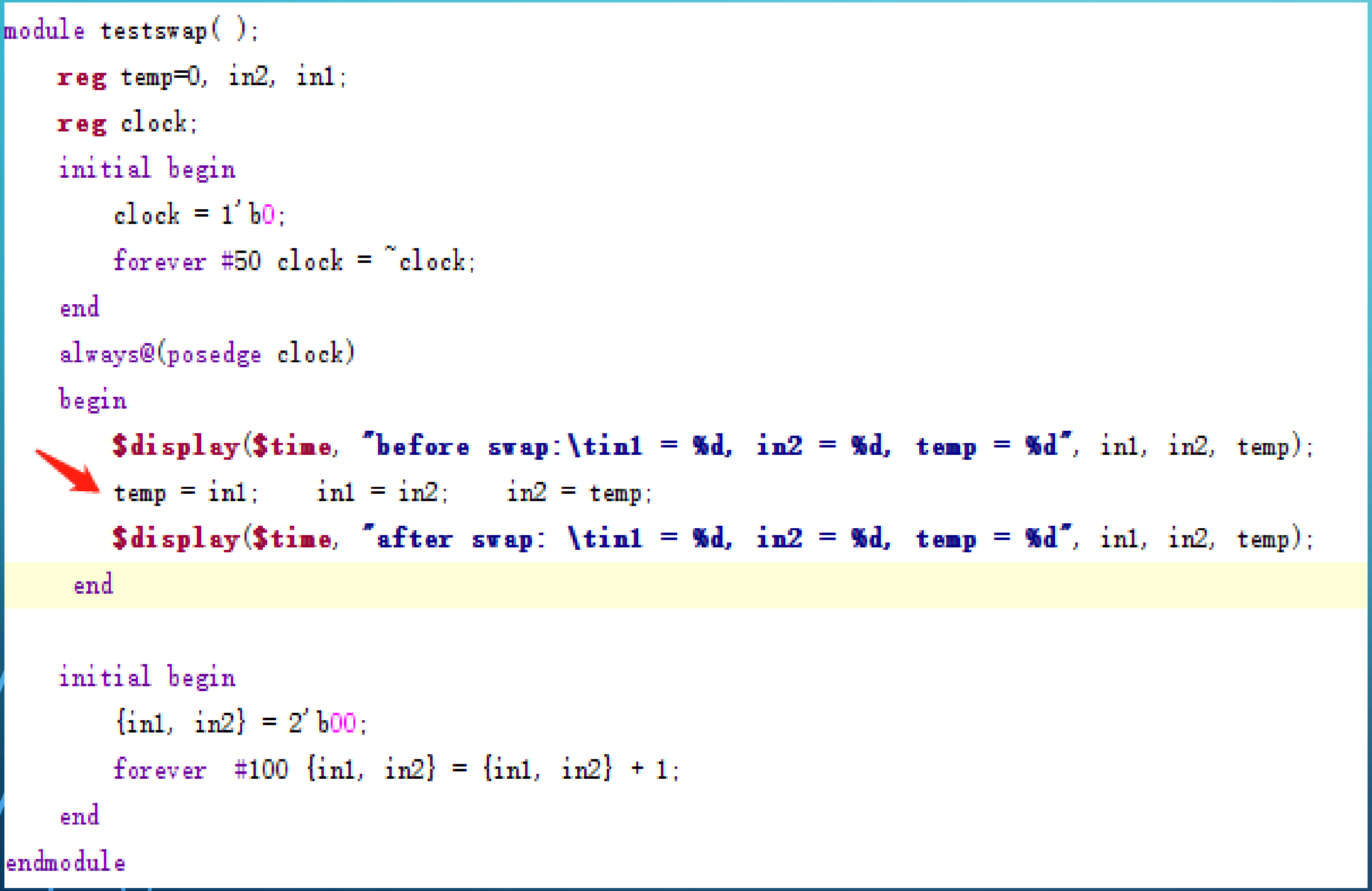
BEHAVIORAL MODELING(3)

Non-blocking assignment VS Blocking assignment

- The ‘**=**’ token represents a token represents a **blocking procedural assignment**
- The ‘**<=**’ token represents a token represents a **non-blocking blocking assignment**
- A combinational logic **always block should use** Blocking assignments(“=“).
- A sequential logic **always block should use** Non-blocking assignments(“<=“).

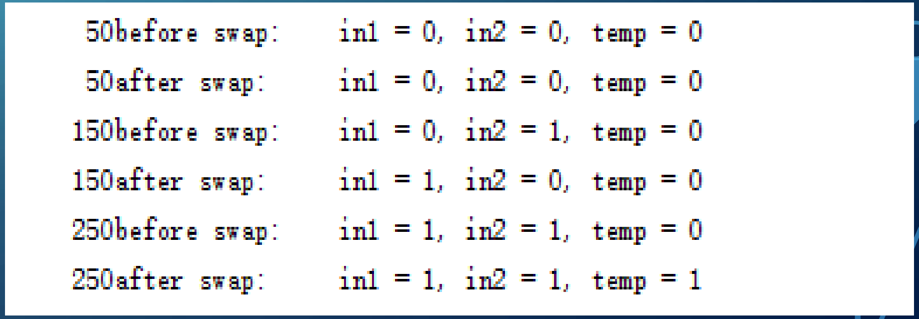
NOTE: **DO NOT** mixing different assignment in the same always block **!!!**

BEHAVIORAL MODELING(5)



```
module testswap( );
   reg temp=0, in2, in1;
   reg clock;
   initial begin
       clock = 1'b0;
       forever #50 clock = ~clock;
   end
   always@(posedge clock)
   begin
       $display($time, "before swap:\tin1 = %d, in2 = %d, temp = %d", in1, in2, temp);
       temp = in1;     in1 = in2;     in2 = temp;
       $display($time, "after swap: \tin1 = %d, in2 = %d, temp = %d", in1, in2, temp);
    end

   initial begin
       {in1, in2} = 2'b00;
       forever  #100 {in1, in2} = {in1, in2} + 1;
   end
endmodule
```

- The function of this module is swapping the value of in1 and in2. here we realize it with blocking assignment



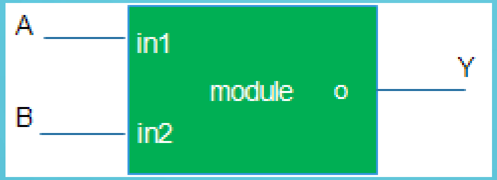
```
  50before swap:    in1 = 0, in2 = 0, temp = 0
  50after swap:     in1 = 0, in2 = 0, temp = 0
 150before swap:    in1 = 0, in2 = 1, temp = 0
 150after swap:     in1 = 1, in2 = 0, temp = 0
 250before swap:    in1 = 1, in2 = 1, temp = 0
 250after swap:     in1 = 1, in2 = 1, temp = 1
```

WIRE VS REG(1)

- There are two data types in Verilog: wire and register.
  - wire is a kind of net, which is equivalent to physical connection.
  - wire is used to connects two points, and thus does not have any driving strength
  - wire data types can be used for connecting the output port to the actual driver
  - a wire can be assigned a value by a continuous assign statement, which is used for designing **combinational** logic
  - default data type is wire: this means that if you declare a port or variable without specifying reg or wire, it will be a 1-bit wide wire.
  - reg is a kind of register, which is equivalent to memory cell.
  - reg can store value and drive strength. Reg can be used for modeling both combinational and sequential logic.
  - reg data type can be driven from initial and always block.
  - The LHS of a behavioral block(initial , always) should be declared *as* **reg**

http://www.asic-world.com/tidbits/wire_reg.html

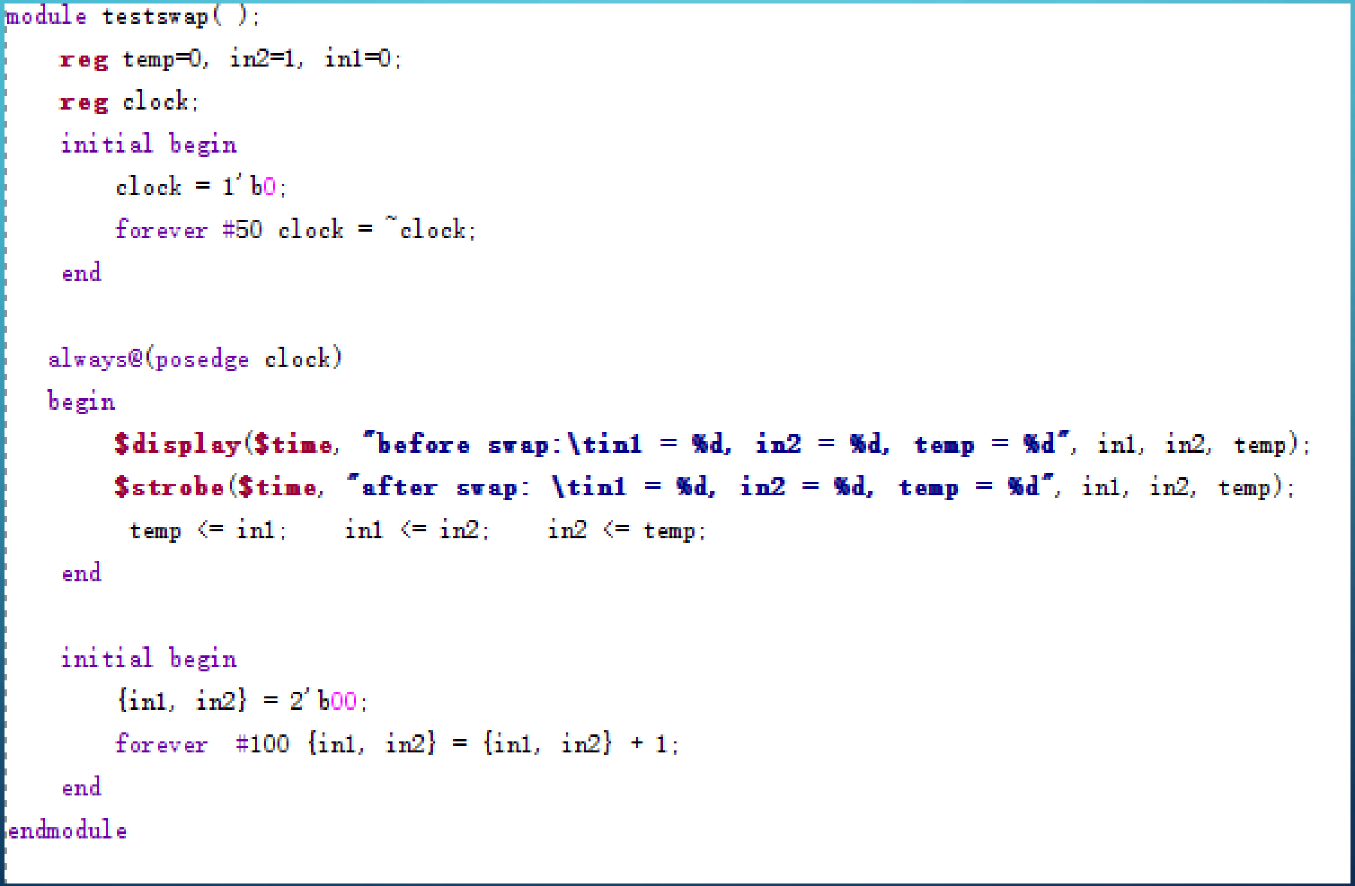
WIRE VS REG(2)



- **Design Source**
  - **input port** could be driven by both wire/register, but it could only be declared as wire
  - **output port** can be declared as wire or register, but it can only drive wire
  - bidirectional port can only be declared as wire
- **Simulation Source**
  - variables bined to input port could be **reg**, and be driven by codes
  - variables bined to output port could be **wire**

PRACTICE 1

Run the following code, try to find why the value of in1 and in2 has not been swapped
Try to revise the code and make it work.



```
module testswap( );
   reg temp=0, in2=1, in1=0;
   reg clock;
   initial begin
      clock = 1'b0;
      forever #50 clock = ~clock;
   end

  always@(posedge clock)
  begin
      $display($time, "before swap:\tin1 = %d, in2 = %d, temp = %d", in1, in2, temp);
      $strobe($time, "after swap: \tin1 = %d, in2 = %d, temp = %d", in1, in2, temp);
       temp <= in1;    in1 <= in2;    in2 <= temp;
   end

   initial begin
      {in1, in2} = 2'b00;
      forever  #100 {in1, in2} = {in1, in2} + 1;
   end
endmodule
```

PRACTICE 2

16 students with sid from 0 to 15 need to be grouped into 4 groups: divide sid by 4, if the remainder is 0, they belongs to group0, if remainder is 1, they belongs to group1, if remainder is 2, they belongs to group2, if remainder is 3, they belongs to group3.

Take the sid as input, the group id as output:

- Write down the corresponding truth table.
- Use K-map simplify the function, implement the circuit using data flow design.
- Use behavioral modeling to do the design, "if else" or "case" is suggested.
- Write the testbench in Verilog to verify the function of design
- Design the constraint file and generate the bitstream and program the device to test the function