

A decorative graphic on the left side of the slide, consisting of a network of white lines and circles on a dark blue background. The lines are of varying thickness and connect various circular nodes, some of which are larger than others, creating a complex, circuit-like pattern that resembles a printed circuit board or a digital network.

DIGITAL DESIGN

LAB5 BEHAVIORAL MODELING IN VERILOG

2022 FALL TERM @ CSE . SUSTECH

LAB5

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

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

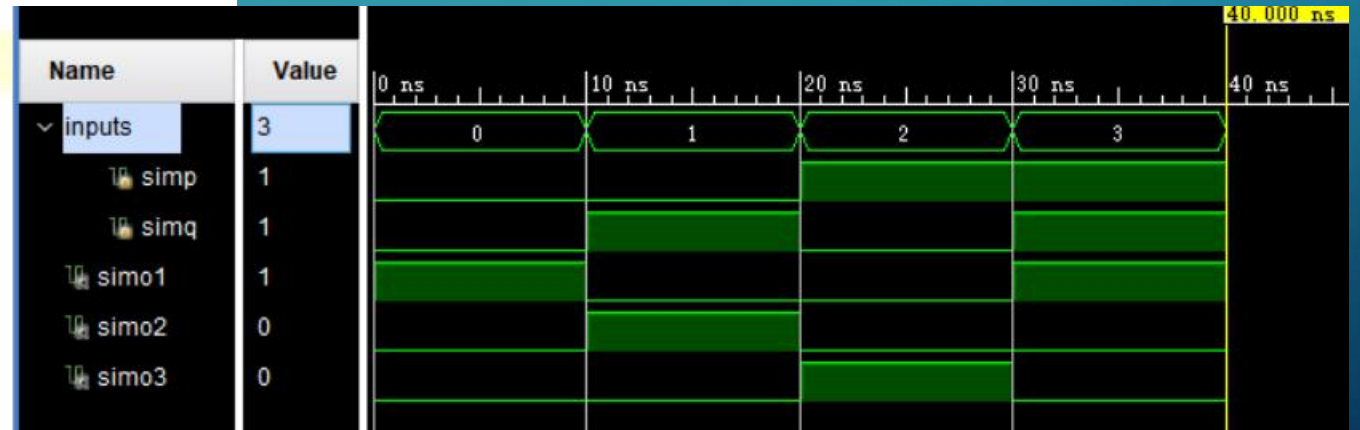
p	q	o1(p==q)	o2(p<q)	o3(p>q)
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
1	1	1	0	0

truth table for 1-bit comparator

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

MAGNITUDE COMPARATOR(1-BIT)

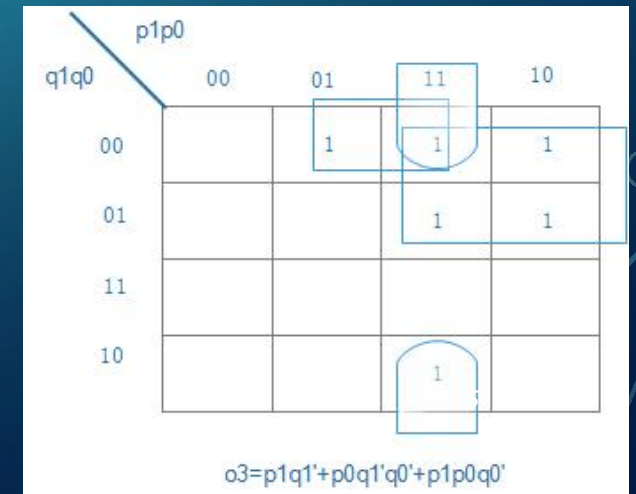
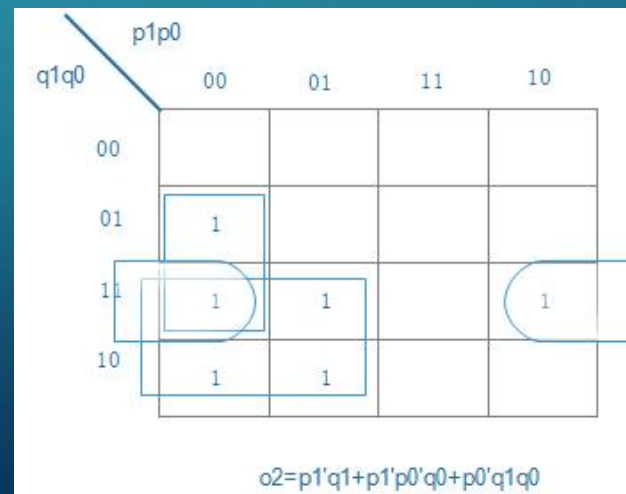
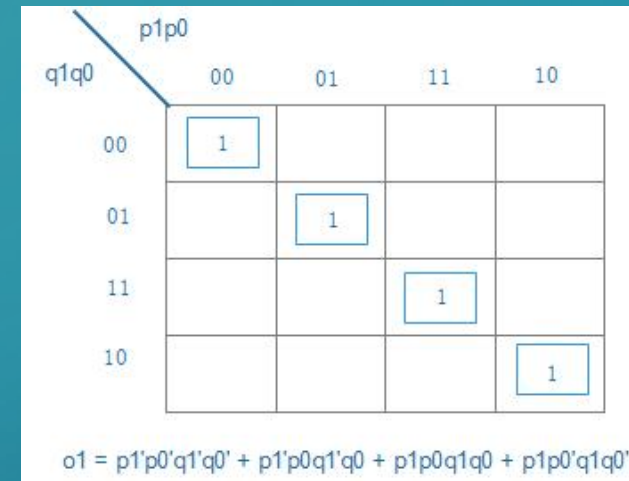
```
module comparators_tb();  
    reg simp, simq;  
    wire simo1, simo2, simo3;  
    comparator u(simp, simq, simo1, simo2, simo3);  
  
    initial begin  
        {simp, simq} = 2'b00;  
        while({simp, simq} < 2'b11)  
            begin  
                #10 {simp, simq} = {simp, simq} +1;  
                $display($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



MAGNITUDE COMPARATOR(2-BIT)

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

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

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

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

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

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

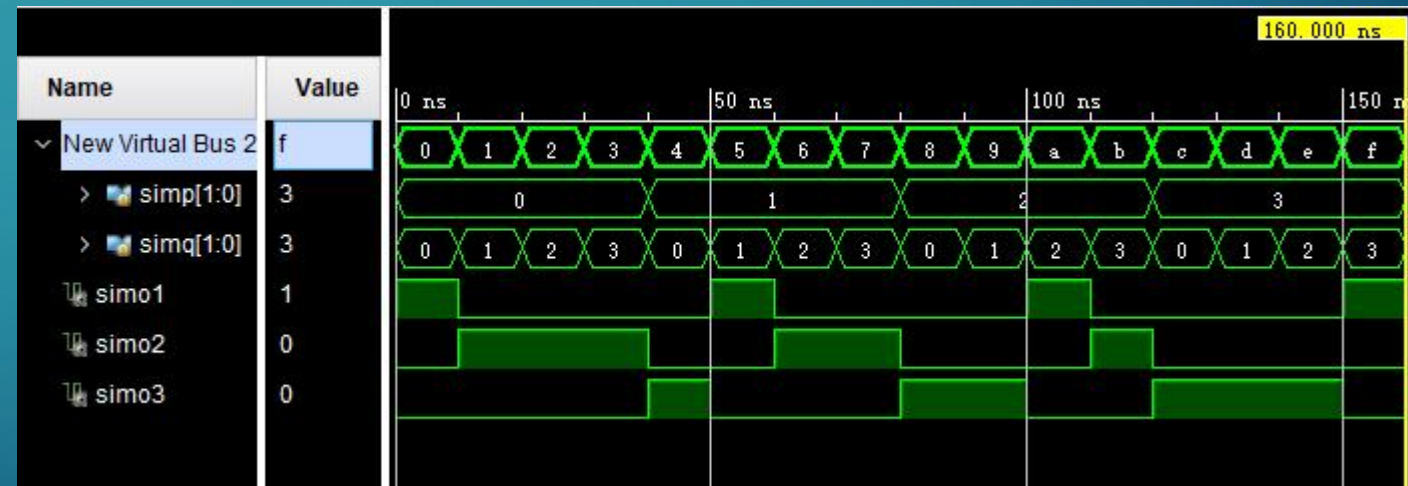
```

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
```



PRACTICE1

- Implement the Magnitude comparator(2-bits) by using **structure design**
 - Using Magnitude comparator(1-bit) and logic gates as components to build circuits.
 - Build the testbench to test the two circuit.
 - Do the simulation to test if the behavior of Magnitude comparator(2-bits) in data-flow design and the Magnitude comparator(2-bits) in structure-design are same?
 - How are about to implement the Magnitude comparator which compares larger bit-width inputs? What's the challenge?
 - Which design style will you prefer to use ?

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
用时间作敏感列表 以敏感信号的改变触发always 执行
- **@(*)**, **@*** : 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.

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

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

```
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**: 只会用在 test bench 里 只执行一次

- 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;

WIRE VS REG(1)

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

Error: Port in1 is not defined
Error: Non-net port in1 cannot be of mode input
Error: Port in2 is not defined
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
```

模块X的输入端口不能在X内部赋值

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

```
23 module test_wire_reg(  
24   );  
25   wire i1,i2;  
26   reg o1,o2;  
27   sub_wr s1(i1,i2,o1,o2);  
28 endmodule  
29  
30 module sub_wr(in1,in2,out1,out2);  
31   input in1,in2;  
32   output out1;  
33   output reg out2;  
34 endmodule
```

[Synth 8-685] variable 'o1' should not be used in output port connection [test_wire_reg.v:27]

WIRE VS REG(2)

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

REG VS WIRE(3)

Please 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; initial 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);		✓

REG VS WIRE(4)

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; initial 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 “initial” or “always” block, the type of assigned object **MUST** be “reg”, otherwise its data type is “wire”.

只有在 initial 和 always 里面被赋值才必须用 reg

PRACTICES(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

PRACTICES(3)

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 **behavioral modeling** 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