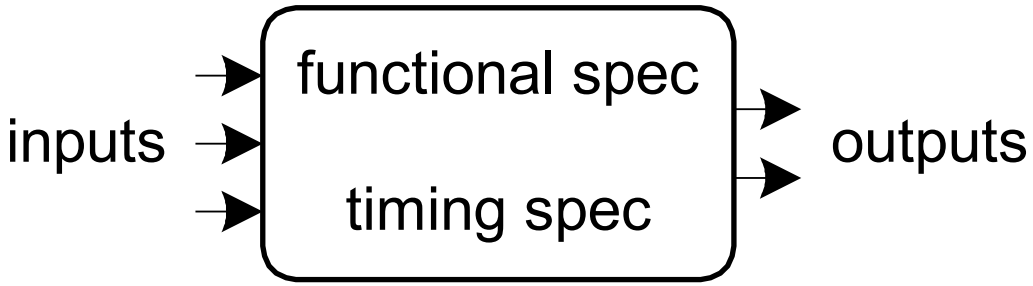
Digital System Design

Hajar Falahati

hfalahati@ipm.ir
hfalahati@ce.sharif.edu

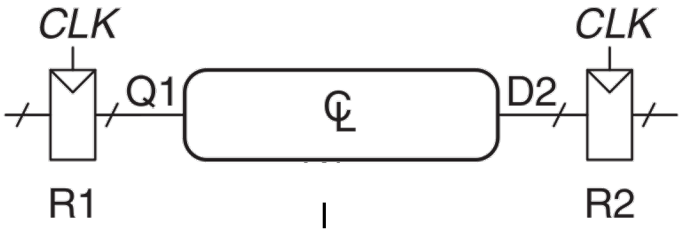
Specifications

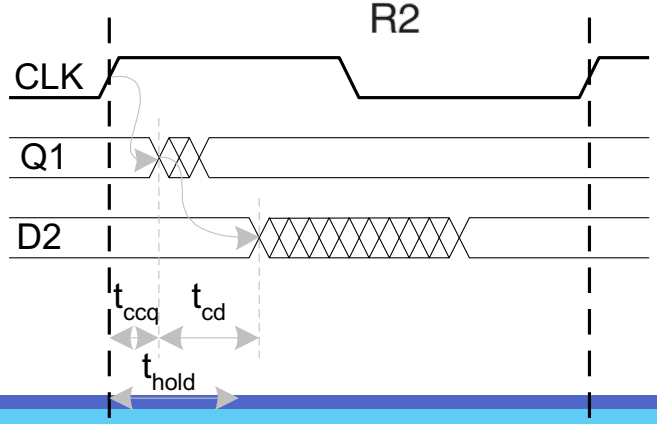
- A logic circuit is composed of:
  - Inputs
  - Outputs
- *Functional specification* (describes relationship between inputs and outputs)
- *Timing specification* (describes the delay between inputs changing and outputs responding)

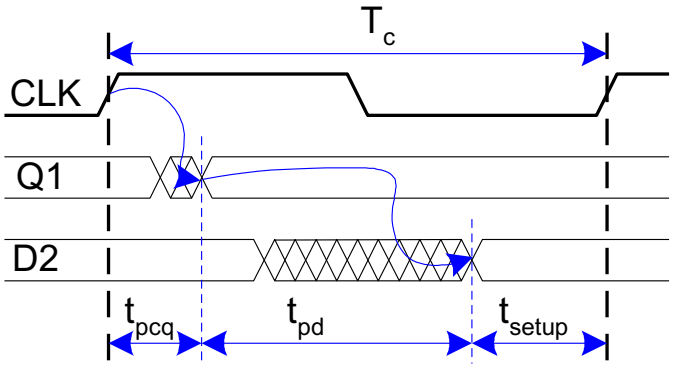


Sequential Timing Summary

t_{ccq} / t_{pcq}	clock-to-q delay (contamination/propagation)
t_{cd} / t_{pd}	combinational logic delay (contamination/propagation)
t_{setup}	time that FF inputs must be stable before next clock edge
t_{hold}	time that FF inputs must be stable after a clock edge
T_c	clock period







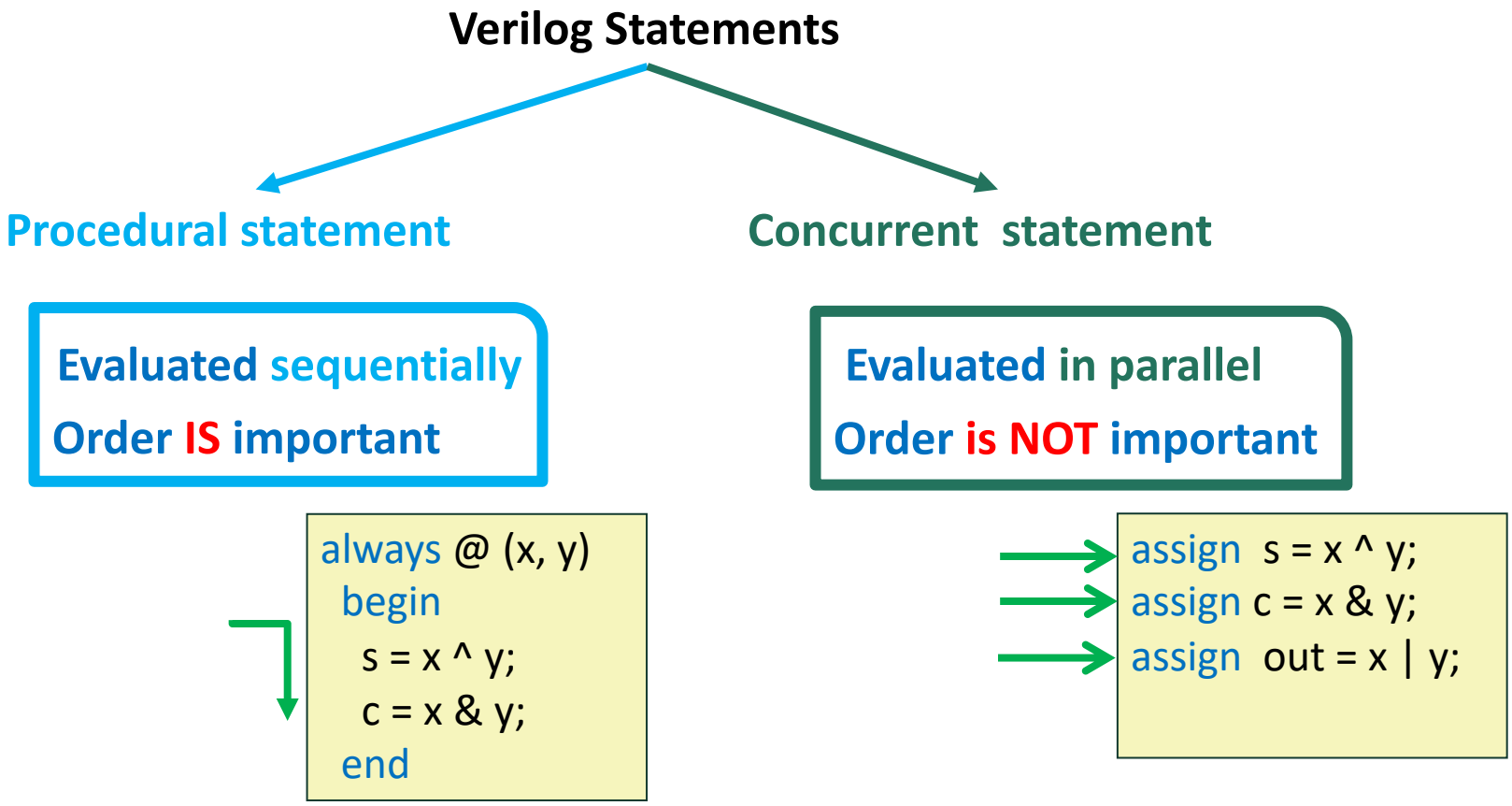
Outline

- Dataflow

- Gate-level

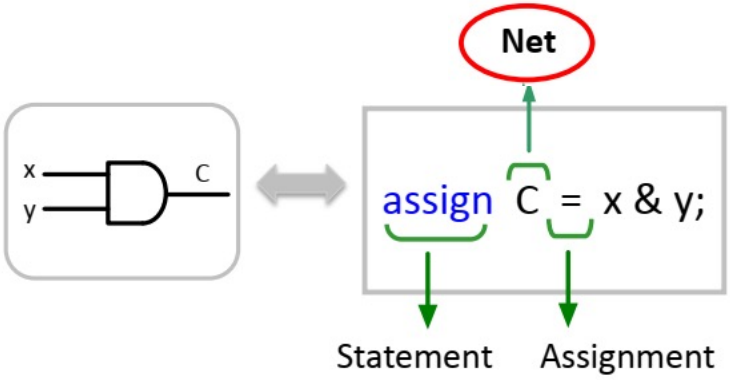
Concurrent Statement

Recall: Verilog Statement



Concurrent Statement

- Evaluated in parallel
- Each statement describes part of the circuit, thus, it is concurrent
- Realized as connection or wire in the design
- Format




```
wire   [1:3] A, B, C;
assign C = A&B;
```

Equivalent

```
assign C[1] = A[1]&B[1];
assign C[2] = A[2]&B[2];
assign C[3] = A[3]&B[3];
```

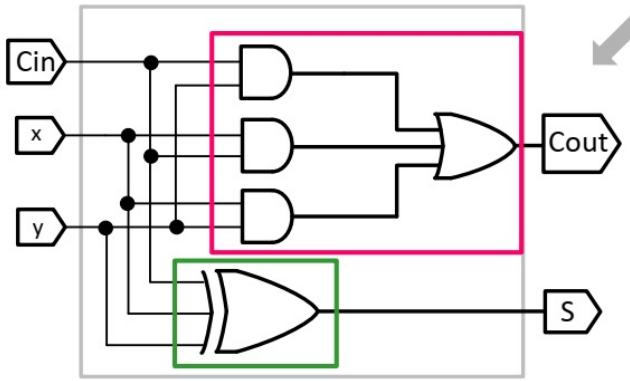
- assign used only for nets (to be synthesizable)

Sample: Full Adder

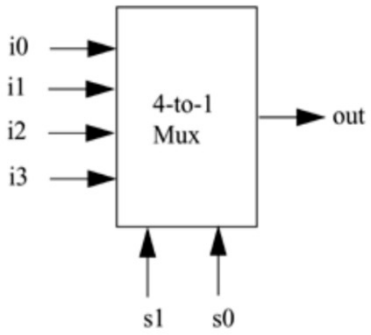
x	y	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

```
module Adder (Cin, x, y, S, Cout)
   input  x, y, Cin;
   output  S, Cout;
   wire S, Cout;

   assign S = x ^ y ^ Cin;

   assign Cout = (x & y)|(x & Cin)|(y & Cin);

endmodule
```

```
module Adder (Cin, x, y, S, Cout)
   input  x, y, Cin;
   output  S, Cout;
   wire S, Cout;


   assign {Cout, S} = x + y + Cin;

endmodule
```

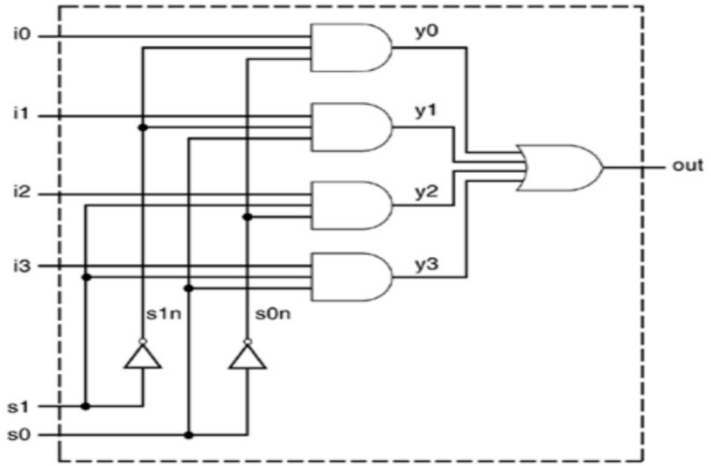


Sample: 4-1 MUX

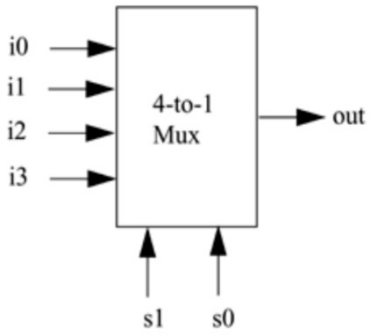


s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

```
module mux4_1(out,i0,i1,i2, i3,s0,s1);
 input   i0,i1,i2,i3,s0,s1;
 output  out;
 wire    s1n,s0n,y0,y1,y2,y3;
assign out =(~s1 & ~s0 & i0) |
            (~s1 &  s0 & i1) |
             (s1  & ~s0 & i2) |
            (s1  &  s0 & i3);
endmodule
```

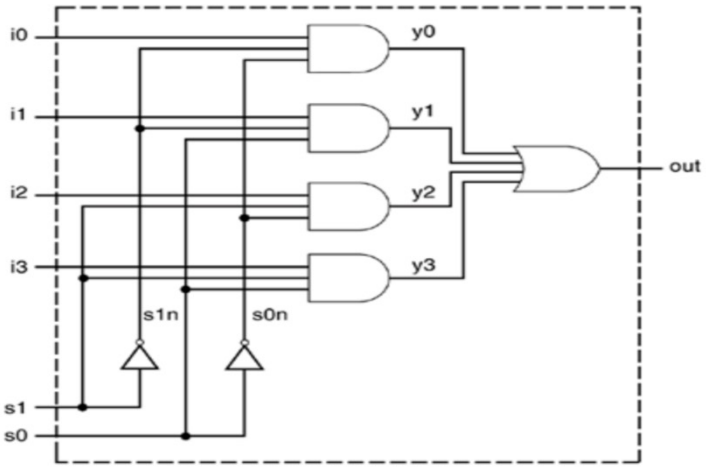


Sample: 4-1 MUX



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

```
module mux4_1(out,i0,i1,i2,i3,s0,s1);

 input   i0,i1,i2,i3,s0,s1;
 output  out;

 assign out = s1? (s0 ? i3:i2) : (s0 ?
i1:i0);

endmodule
```



Dataflow: Delay

- Delay can be used with continuous assignments by using the “#” sign
  - 2 time unit of delay on wire S
  - 5 time units of delay for AND gate
  - Any change in x or y reflects on S after ? time unit delay

```
wire  #2  S;
assign  #5  S = x&y;
```

Dataflow: Delay

- Delay can be used with continuous assignments by using the “#” sign
  - 2 time unit of delay on wire S
  - 5 time units of delay for AND gate
  - Any change in x or y reflects on S after 7 time unit delay

```
wire  #2  S;
assign  #5  S = x&y;
```

Delays at Dataflow Level

- Regular assignment delay

```
assign  #10  out = in1 & in2;
```

- Implicit continuous assignment delay

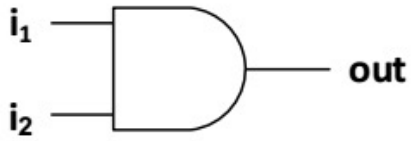
```
wire #10 out = in1 & in2;

 //same as
wire    out;
assign  #10 out = in1 & in2;
```

- Net declaration delay

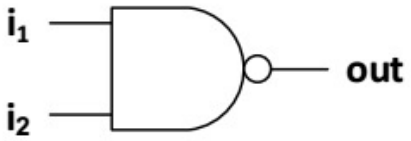
```
wire #10 out ;

assign  out = in1 & in2;
```

Gate

Primitive Gates: and, nand (cont'd)



and	i_1 = 0	1	x	z
i_2 = 0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

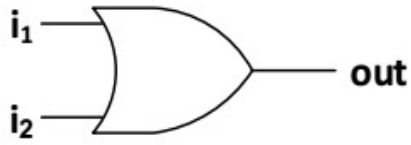


nand	i_1 = 0	1	x	z
i_2 = 0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

and g0 (out, i_1, i_2)

nand g1 (out, i_1, i_2)

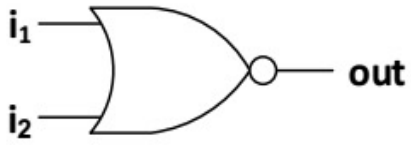
Primitive Gates: or, nor (cont'd)



or	i_1 = 0	i_1 = 1	i_1 = x	i_1 = z
i_2 = 0	0	1	x	x
i_2 = 1	1	1	1	1
i_2 = x	x	1	x	x
i_2 = z	x	1	x	x

or g0 (out, i_1, i_2)

or (out, i_1, i_2)

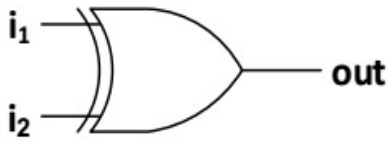


nor	i_1 = 0	i_1 = 1	i_1 = x	i_1 = z
i_2 = 0	1	0	x	x
i_2 = 1	0	0	0	0
i_2 = x	x	0	x	x
i_2 = z	x	0	x	x

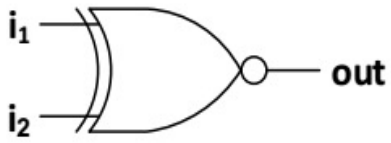
nor g1 (out, i_1, i_2)

nor (out, i_1, i_2)

Primitive Gates: xor, xnor (cont'd)



xor	i_1: 0	1	x	z
i_2: 0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

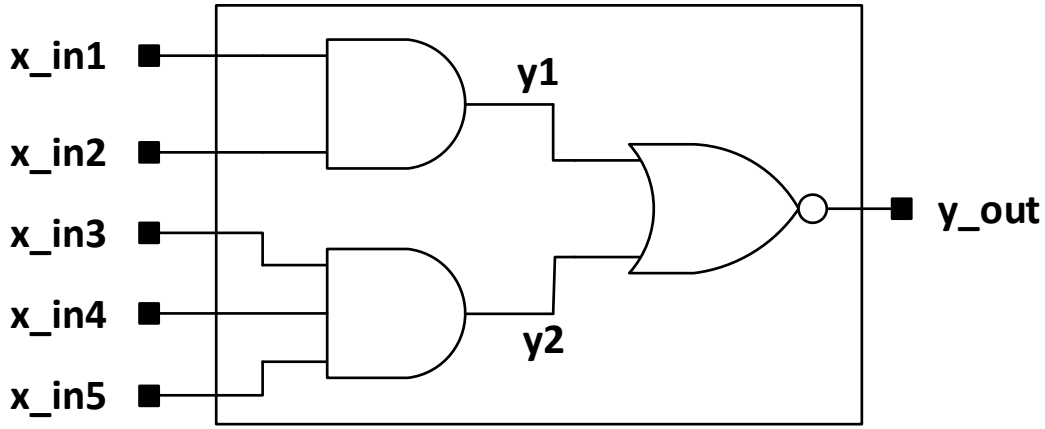


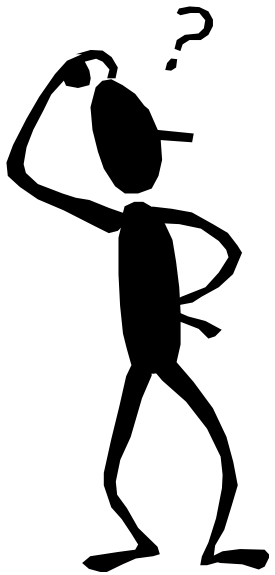
xnor	i_1: 0	1	x	z
i_2: 0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

xor g0 (out, i_1, i_2)

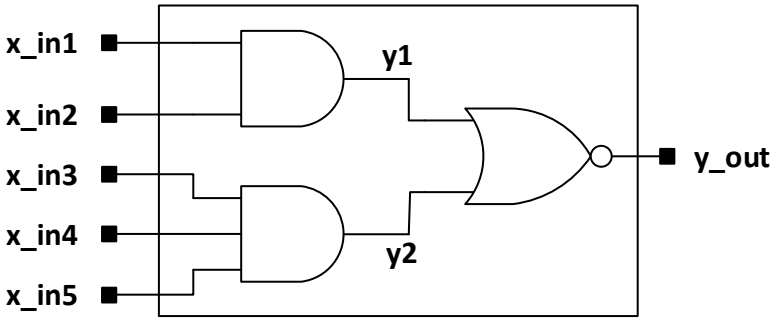
xnor g1 (out, i_1, i_2)

Gate-Level Design Sample ?



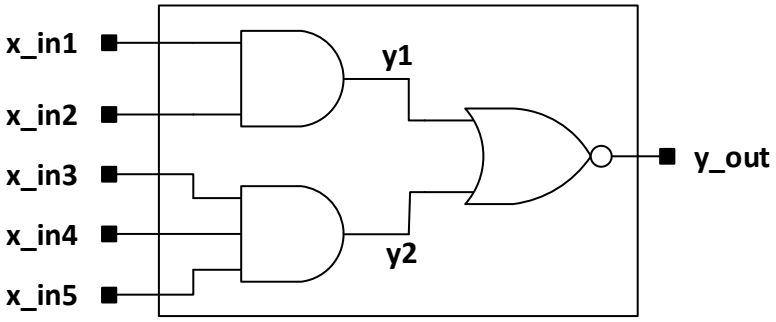


Sample: Step1



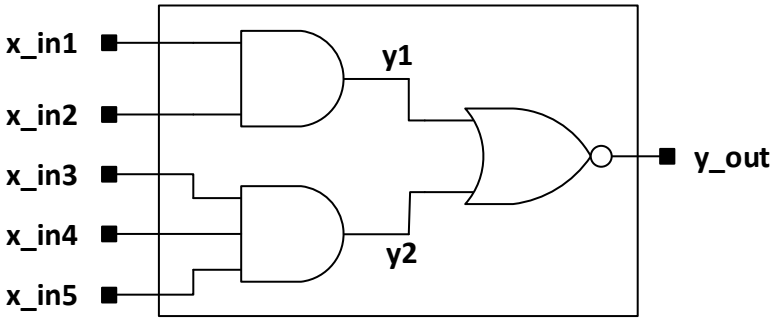

```
module gateLevelSample(y_out,x_in1,x_in2,x_in3,x_in4,x_in5);
endmodule
```

Sample: Step2



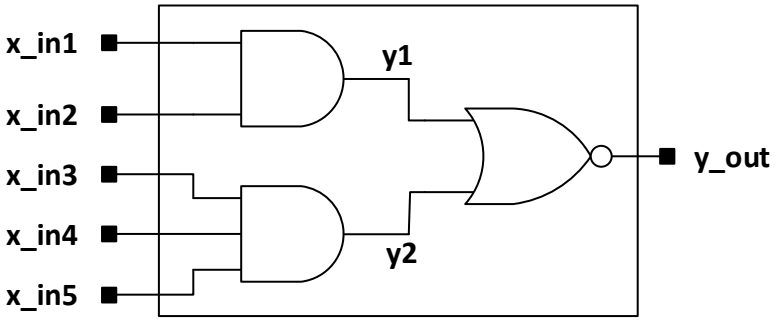

```
module gateLevelSample(y_out,x_in1,x_in2,x_in3,x_in4,x_in5);
       output y_out;
       input  x_in1 , x_in2 , x_in3 , x_in4 , x_in5;

endmodule
```

Sample: Step3



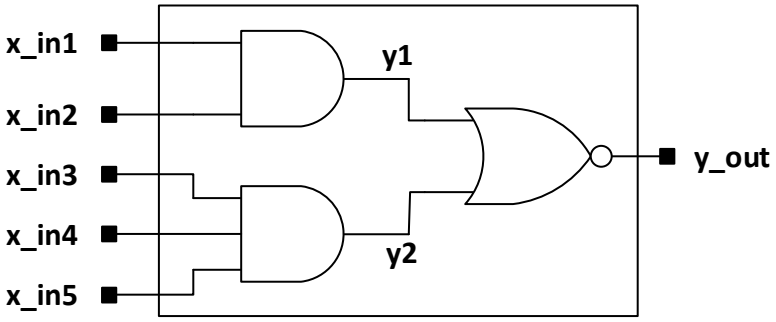

```
module gateLevelSample(y_out,x_in1,x_in2,x_in3,x_in4,x_in5);
       output y_out;
       input  x_in1 , x_in2 , x_in3 , x_in4 , x_in5;

       wire   y1 , y2;

endmodule
```

Sample: Step4



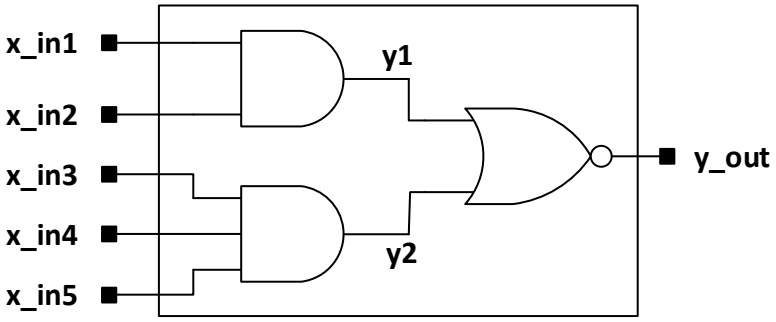

```
module gateLevelSample(y_out,x_in1,x_in2,x_in3,x_in4,x_in5);
       output y_out;
       input  x_in1 , x_in2 , x_in3 , x_in4 , x_in5;

       wire   y1 , y2;

       and    (y1 , x_in1 , x_in2);

endmodule
```

Sample: Step5



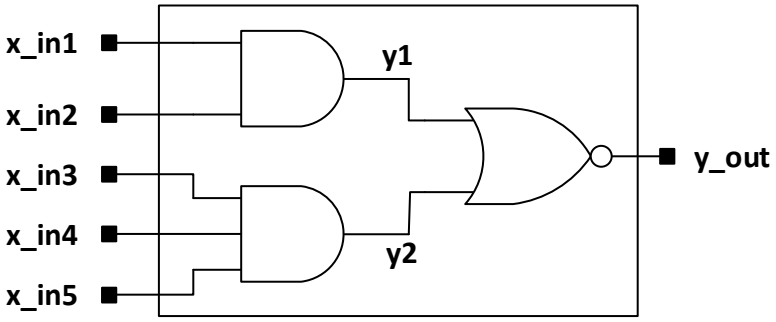

```
module gateLevelSample(y_out,x_in1,x_in2,x_in3,x_in4,x_in5);
       output y_out;
       input  x_in1 , x_in2 , x_in3 , x_in4 , x_in5;

       wire   y1 , y2;

       and    (y1 , x_in1 , x_in2);
       and    (y2 , x_in3 , x_in4 , x_in5);
endmodule
```

Sample: Step6




```
module gateLevelSample(y_out,x_in1,x_in2,x_in3,x_in4,x_in5);
       output y_out;
       input  x_in1 , x_in2 , x_in3 , x_in4 , x_in5;

       wire   y1 , y2;

       and    (y1 , x_in1 , x_in2);
       and    (y2 , x_in3 , x_in4 , x_in5);
       nor    (y_out , y1 , y2);
endmodule
```

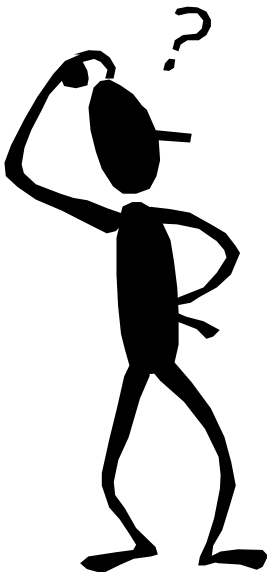
Sample: Is this code correct?




```
module gateLevelSample(y_out,x_in1,x_in2,x_in3,x_in4,x_in5);
       output y_out;
       input  x_in1 , x_in2 , x_in3 , x_in4 , x_in5;

       wire   y1 , y2;

       nor    (y_out , y1 , y2);
       and    (y1 , x_in1 , x_in2);
       and    (y2 , x_in3 , x_in4 , x_in5);
endmodule
```

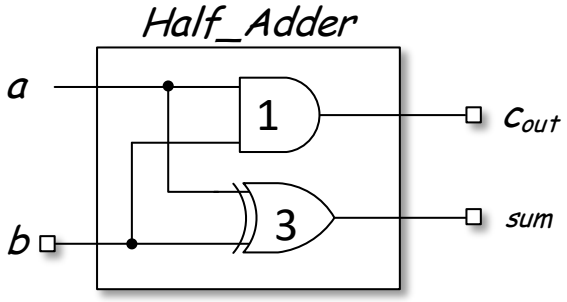
Design a 1-bit Full Adder in Gate Level

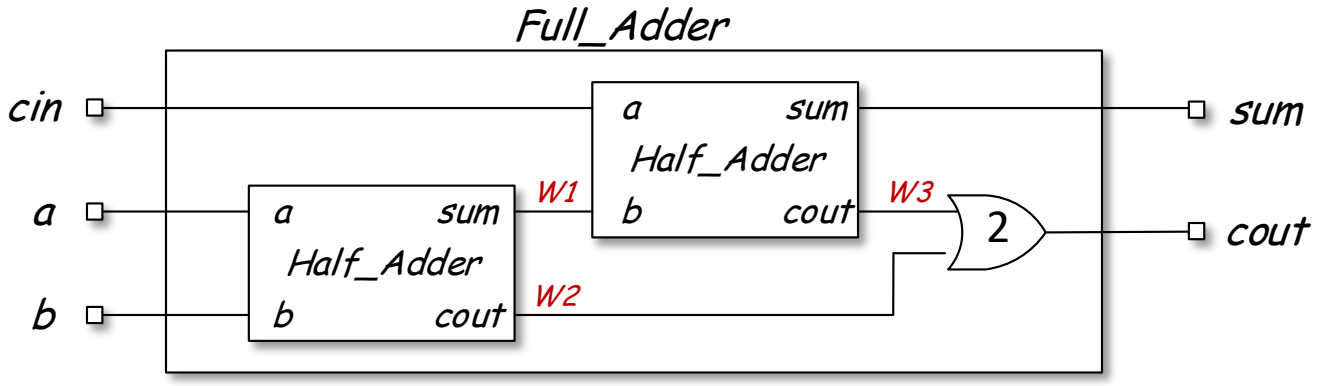
- Design a 1-bit half adder
- Design a 1-bit full adder
- **xor gate**
  - delay: 3
- **and gate**
  - delay: 1
- **or gate**
  - delay: 2



- `**timescale** 1ns/100ps

FA: Gate-level Description





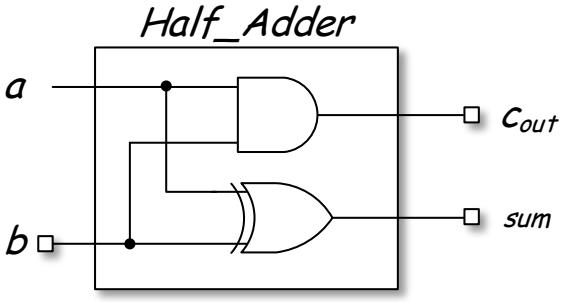
FA: Design Block

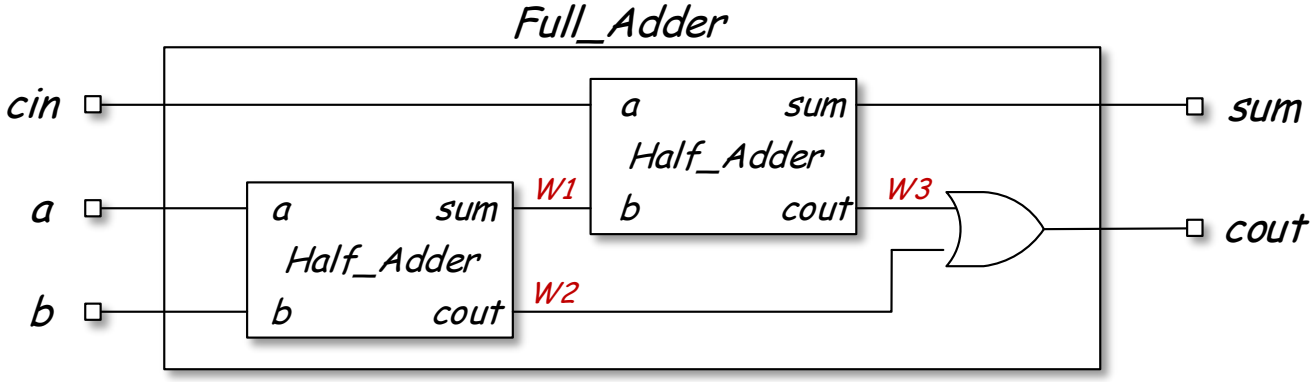
```
module Half_Adder(sum , cout , a , b);
        input      a , b;
        output   sum , cout;

        xor #3(sum , a , b);
        and #1(cout , a , b);
endmodule
```

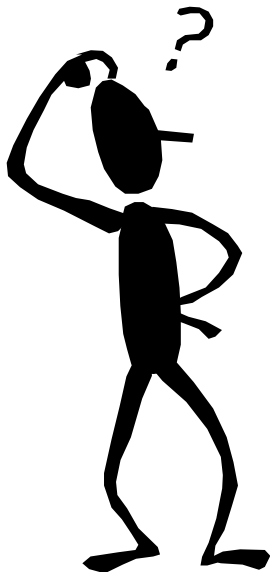
- xor gate
  - delay: 3
- and gate
  - delay: 1
- or gate
  - delay: 2
- `timescale 1ns/100ps

```
module Full_Adder(sum , cout , a , b , c_in);
        input        a , b, C_in;
        output  sum , cout;
        wire    w1, w2, w3;

        Half_Adder M1(w1 , w2, a , b);
        Half_Adder M1(sum , w3, c_in , w1);
        or #2 (cout , w2 , w3);
endmodule
```

FA: Delay analysis

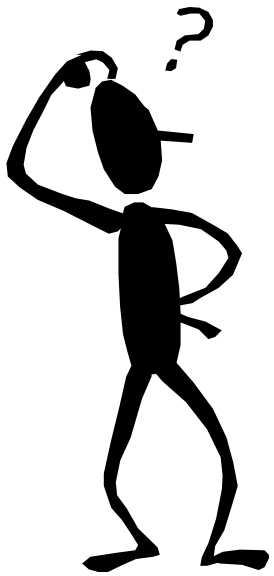






Design a 3-input XOR

- **nand gate**
  - Tplh: 2, 4, 6
  - Tphl: 3, 5, 7
- **not gate**
  - Tplh: 1, 3, 5
  - Tphl: 2, 4, 6
- `**`timescale**` 1ns/100ps



3-input XOR: Truth Table

a	b	c	y	
0	0	0	0	
0	0	1	1	~a~bc
0	1	0	1	~ab~c
0	1	1	0	
1	0	0	1	a~b~c
1	0	1	0	
1	1	0	0	
1	1	1	1	abc

~a~bc + ~ab~c + a~b~c + abc

3-input XOR: Functionality

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

~a~bc + ~ab~c + a~b~c + abc

~a (~bc + b~c) + a (~b~c + bc)

~a (b xor c) + a (b xnor c)

3-input XOR: Nand-Not Expression

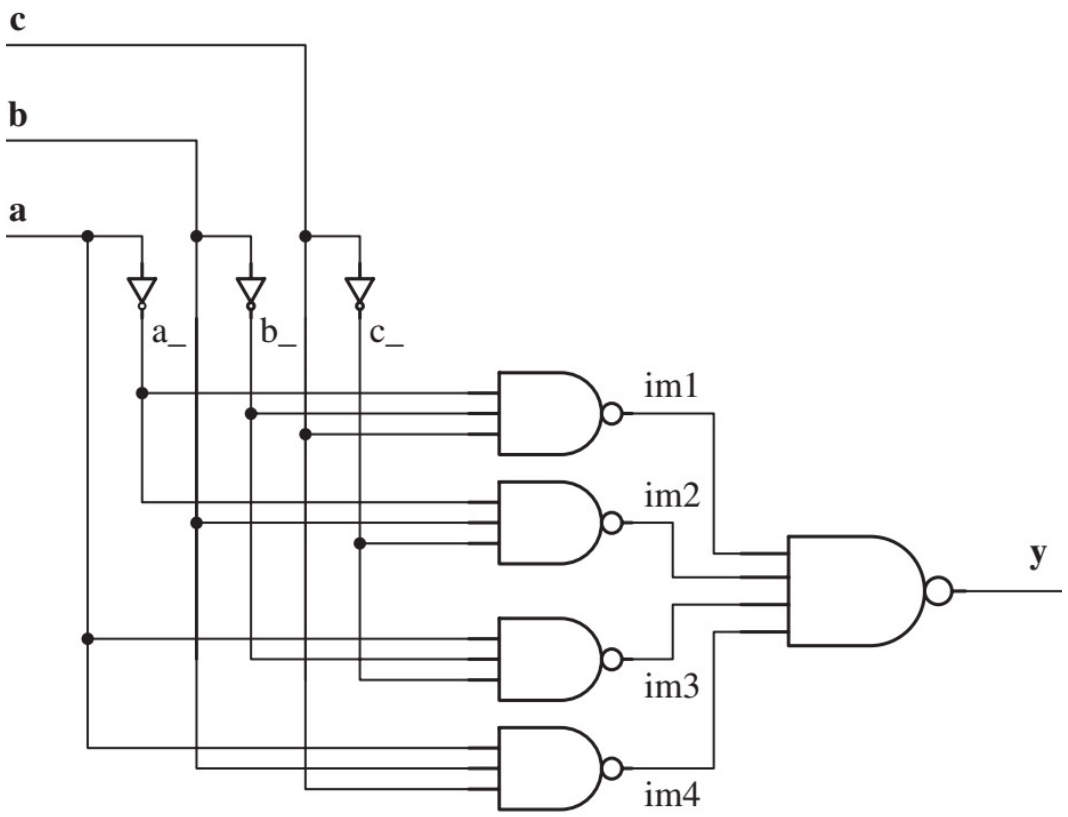
a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

~a~bc + ~ab~c + a~b~c + abc

A + B === ~(~A . ~ B)

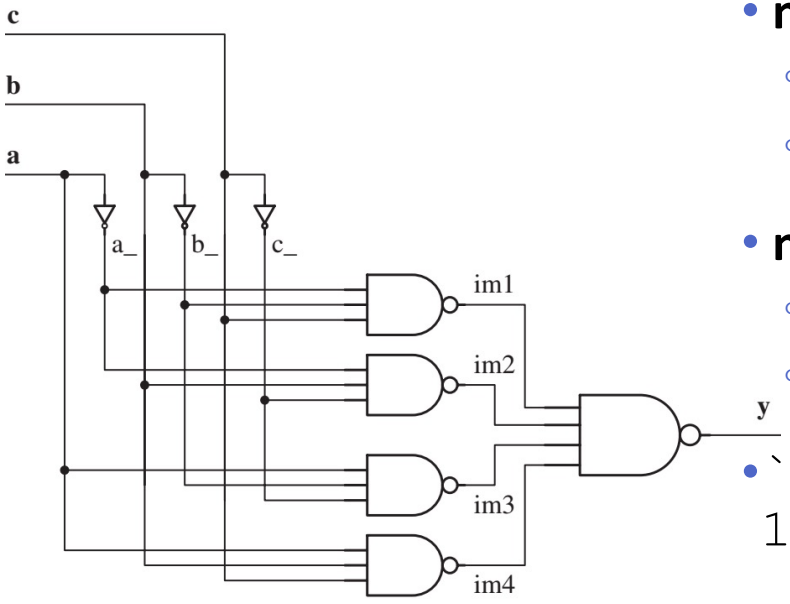
~(~(~a~bc) . ~(~ab~c) .~(a~b~c) . ~(abc))

3-input XOR: Gate-level Description



3-input XOR: Design Block

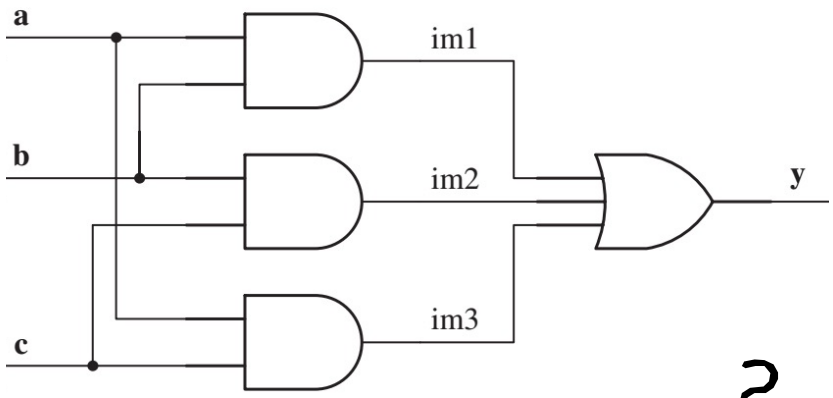
```
`timescale 1ns/100ps
module xor3(y, a, b, c);
 input      a, b, c;
 output  y;
 wire    im1, im2,im3, im4;
 wire    a_, b_, c_;

not #(1:3:5, 2:4:6)
    (a_, a),(b_, b),(c_, c);
nand #(2:4:6, 3:5:7)
(im1, a_, b_, c),(im2, a_, b, c_);
(im3, a, b_, c_),(im4, a, b, c);

 nand #(2:4:6, 3:5:7) (y, im1, im3, im3, im4);
endmodule
```

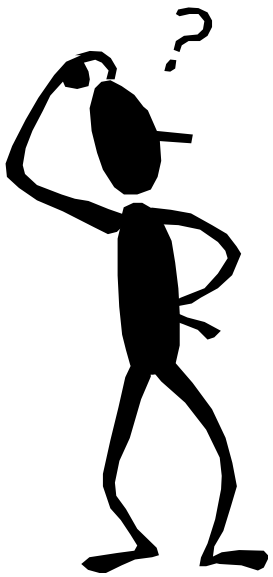


- **nand gate**
  - Tplh: 2, 4, 6
  - Tphl: 3, 5, 7
- **not gate**
  - Tplh: 1, 3, 5
  - Tphl: 2, 4, 6
- `timescale 1ns/100ps

Sample 4

- **and gate**
  - Tplh: 2
  - Tphl: 4
- **or gate**
  - Tplh: 3
  - Tphl: 5
- `**`timescale**` 1ns/100ps

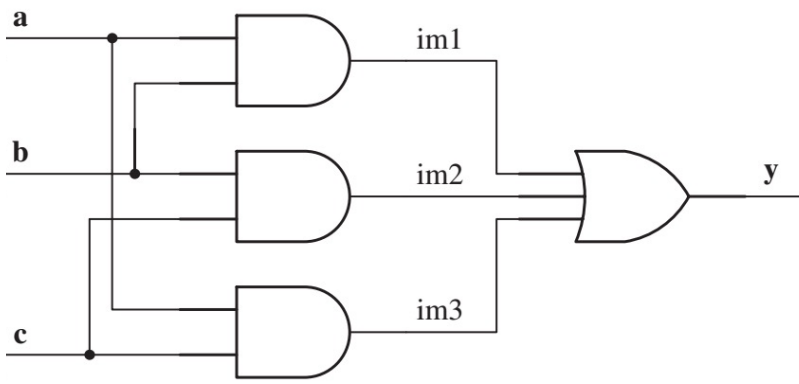




Sample 4: Design Block

```
`timescale 1ns/100ps

module sample2(y, a, b, c);
  input         a, b, c;
  output  out;
  wire    im1, im2, im3;

  and #(2,4)
      (im1, a, b),
      (im2, b, c),
      (im3, a, c);
   or #(3,5)(y, im1, im2, im3);

endmodule
```

- **and gate**
  - Tplh: 2
  - Tphl: 4
- **or gate**
  - Tplh: 3
  - Tphl: 5
- `timescale 1ns/100ps



Multiple instantiations

Array of gate instances

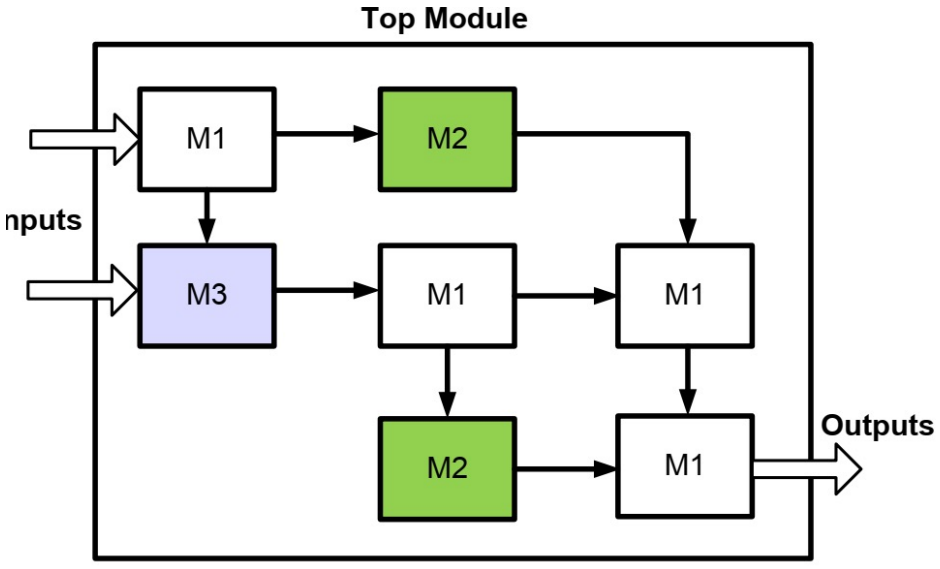
```
wire [7:0] OUT, IN1, IN2;
nand n0(OUT[0], IN1[0],IN2[0]);
nand n1(OUT[1], IN1[1],IN2[1]);
nand n2(OUT[2], IN1[2],IN2[2]);
nand n3(OUT[3], IN1[3],IN2[3]);
nand n4(OUT[4], IN1[4],IN2[4]);
nand n5(OUT[5], IN1[5],IN2[5]);
nand n6(OUT[6], IN1[6],IN2[6]);
nand n7(OUT[7], IN1[7],IN2[7]);
```

```
wire [7:0] OUT, IN1, IN2;
nand  n[7:0] (OUT, IN1,IN2);
```

Modeling

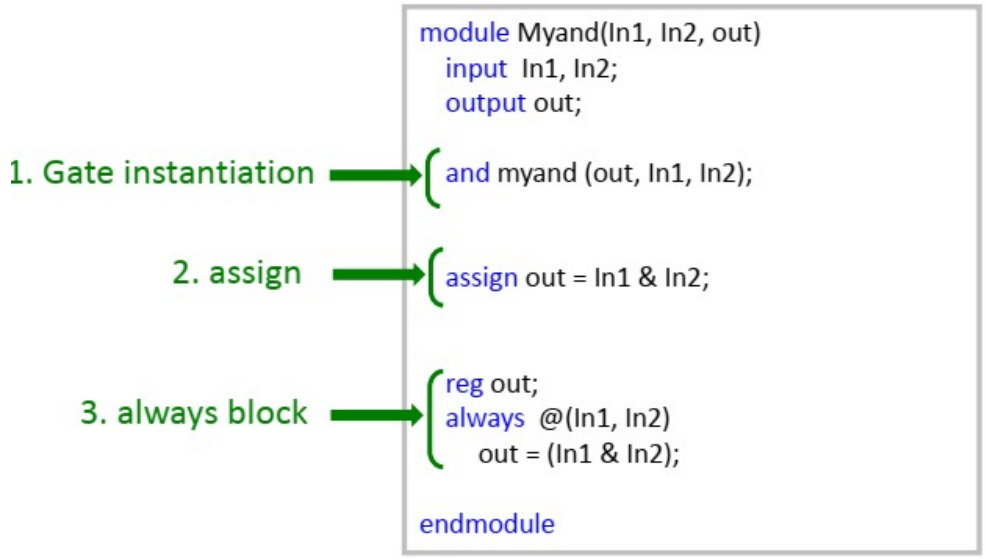
Using sub-circuits

- A design can use multiple submodules or a module multiple times
- Using a module in another is called “instantiation”
- Top-level module: the module that has not been instantiated
- To use a module inside another, it should be explicitly instantiated



Using modules

- There are some built-in primitive logic gates in Verilog that can be instantiated
  - Built-in primitives means there is no need to define a module for these gates
  - and, or, nor, ….



Thank You