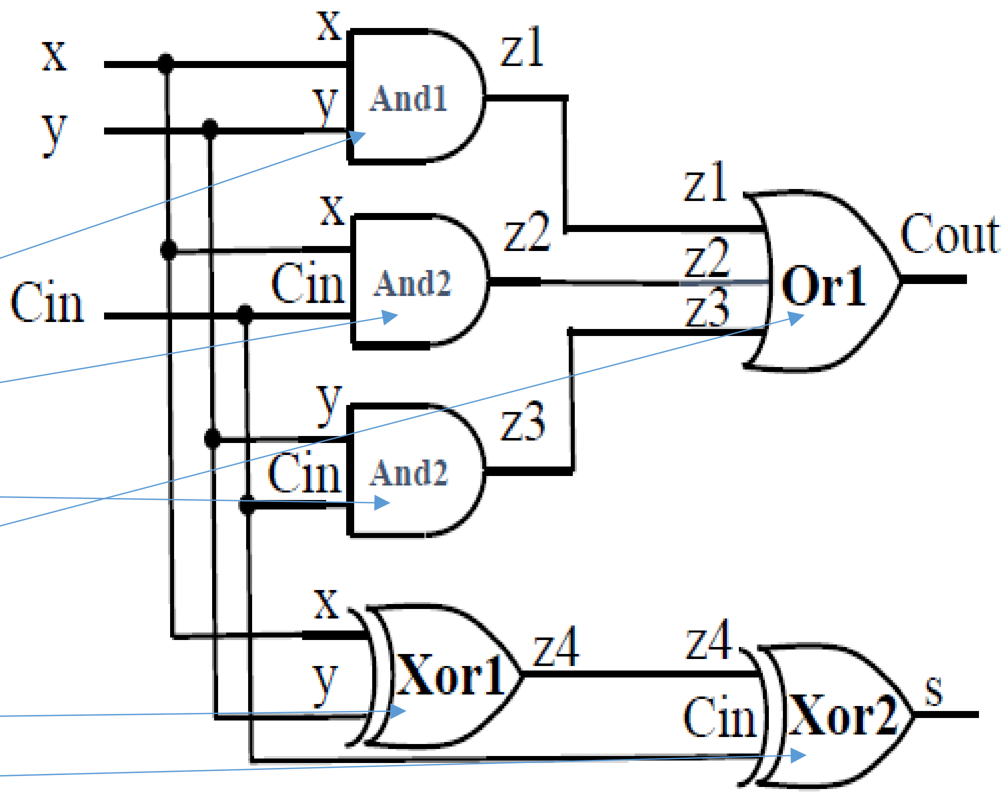
Gate Instantiations (Note Ch.2 p.15)

Pre-defined modules for logic gates

```
module fulladd(Cin,x,y,s,Cout);
      input Cin,x,y;
      output s,Cout;

      wire zl,z2,z3,z4;

      and And1(zl,x,y);
      and And2(z2,x,Cin);
      and And3(z3,y,Cin);

      or Or1(Cout,zl,z2,z3);

      xor Xorl(z4,x,y);
      xor Xor2(s,z4,Cin);
endmodule
```

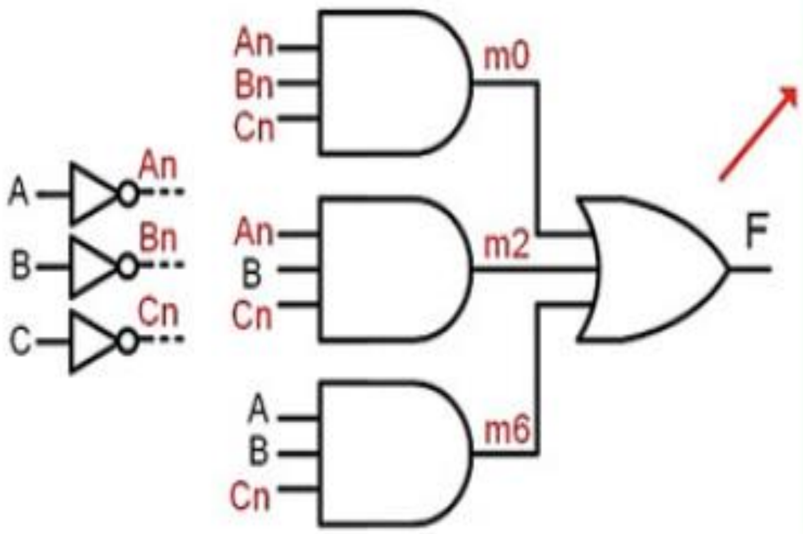
The first one is output while the rest are inputs



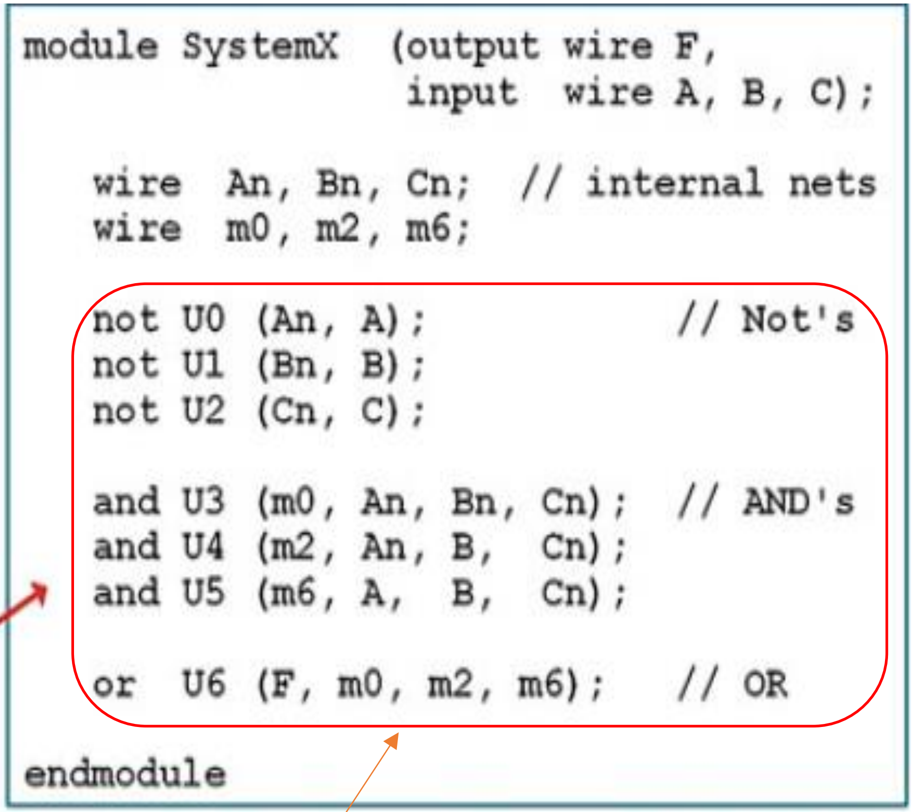
The instance name is optional.

The order of execution statements is not important unless inside an **always** block.

(Another example from: “Quick Start Guide to Verilog” p.54)



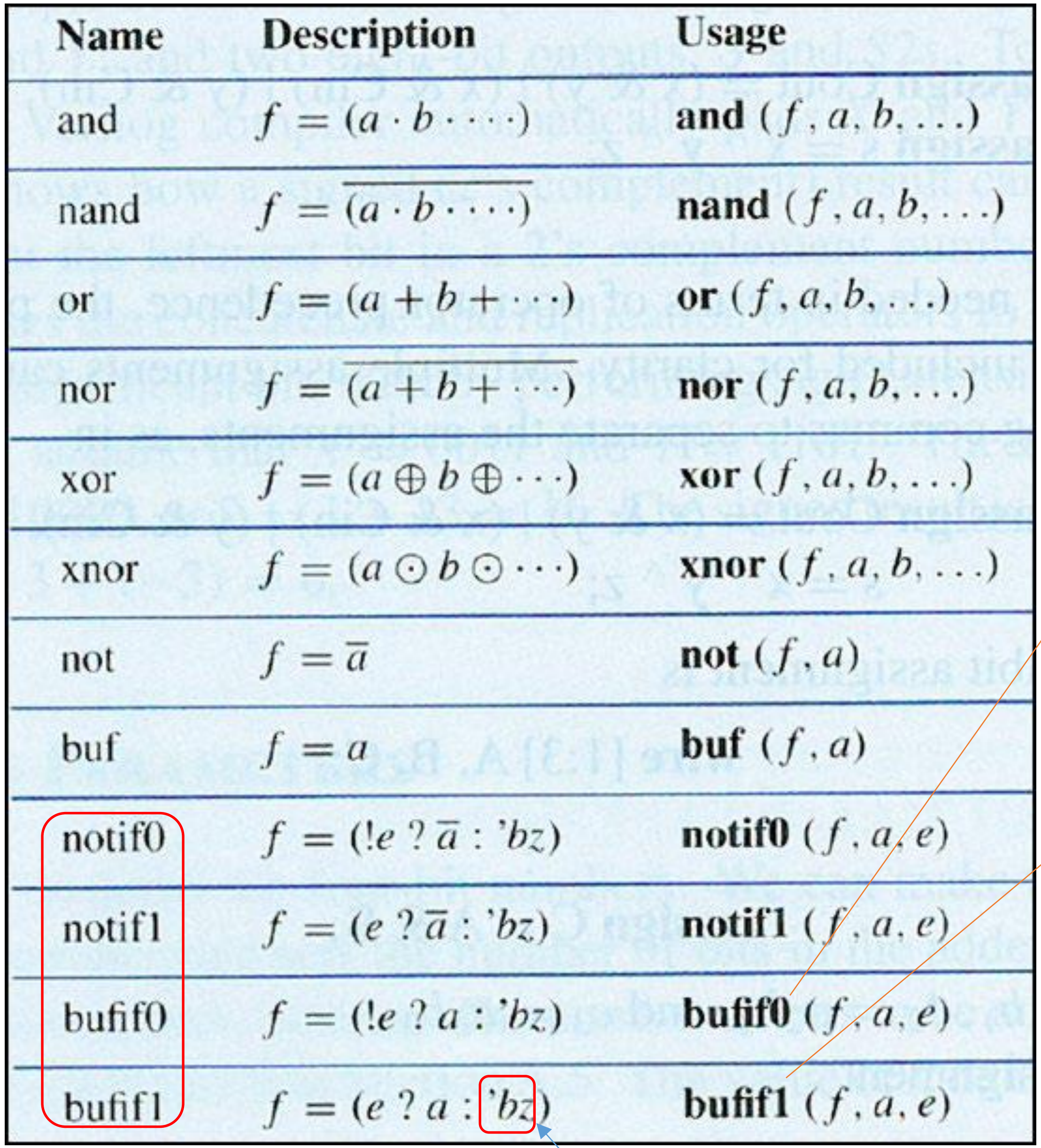
The output is always listed first in the port mapping when using gate level primites.



```
module SystemX  (output wire F,
                 input  wire A, B, C);

   wire  An, Bn, Cn;  // internal nets
   wire  m0, m2, m6;

   not U0  (An, A);             // Not's
   not U1  (Bn, B);
   not U2  (Cn, C);

   and U3  (m0, An, Bn, Cn);   // AND's
   and U4  (m2, An, B,  Cn);
   and U5  (m6, A,  B,  Cn);

   or  U6  (F, m0, m2, m6);    // OR

endmodule
```

Note these statements are concurrent and their order is not important (since they are not enclosed in any **always** / **initial** block).

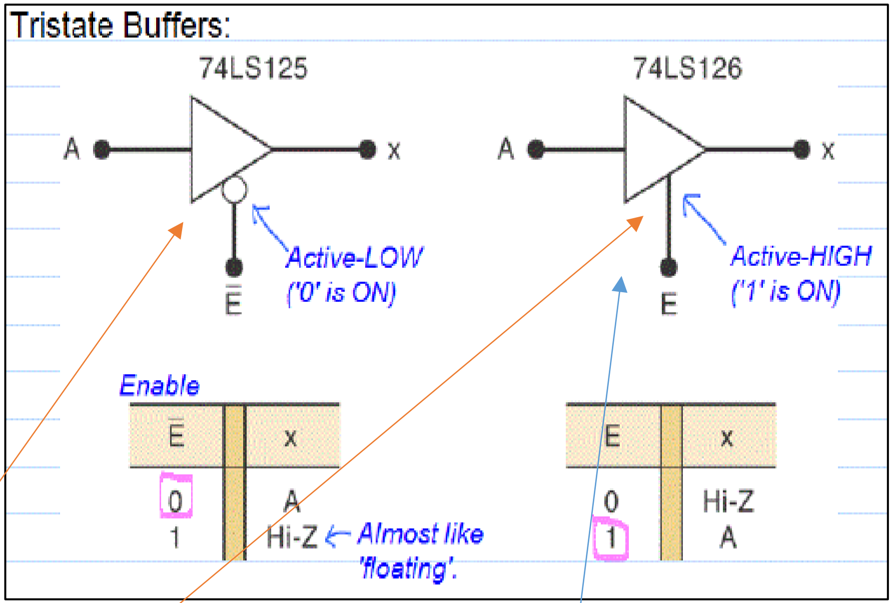
Note Ch.2 p.16



Name	Description	Usage
and	$f = (a \cdot b \cdot \cdots)$	**and** $(f, a, b, \ldots)$
nand	$f = \overline{(a \cdot b \cdot \cdots)}$	**nand** $(f, a, b, \ldots)$
or	$f = (a + b + \cdots)$	**or** $(f, a, b, \ldots)$
nor	$f = \overline{(a + b + \cdots)}$	**nor** $(f, a, b, \ldots)$
xor	$f = (a \oplus b \oplus \cdots)$	**xor** $(f, a, b, \ldots)$
xnor	$f = (a \odot b \odot \cdots)$	**xnor** $(f, a, b, \ldots)$
not	$f = \overline{a}$	**not** (f, a)
buf	$f = a$	**buf** (f, a)
notif0	$f = (!e \; ? \; \overline{a} : \text{'bz})$	**notif0** (f, a, e)
notif1	$f = (e \; ? \; \overline{a} : \text{'bz})$	**notif1** (f, a, e)
bufif0	$f = (!e \; ? \; a : \text{'bz})$	**bufif0** (f, a, e)
bufif1	$f = (e \; ? \; a : \boxed{\text{'bz}})$	**bufif1** (f, a, e)

Hi-Z of any bits

From DE2:



$\overline{E}$	x
$\boxed{0}$	A
1	Hi-Z

E	x
0	Hi-Z
$\boxed{1}$	A

Tristate Inverters

Tristate Buffers, e.g.: **bufif1**(x, A, E);

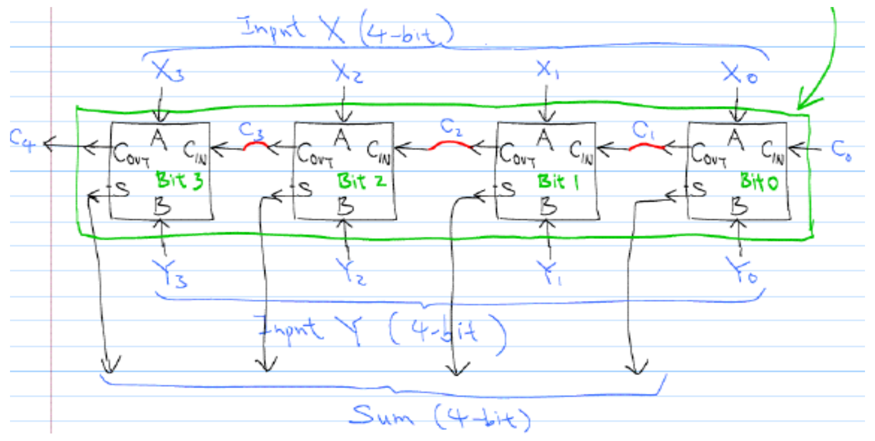
Sub-circuit (Note Ch.2 p.17)

```
module adder4(carryin,X,Y,S, carryout);
        input carryin; input [3:0] X, Y; output [3:0] S; output carryout; wire [3:1] C;

        fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
        fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
        fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
        fulladd stage3 (.Cout(carryout), .s(S[3]), .y(Y[3]), .x(X[3]), .Cin(C[3]));
endmodule
```

Positional port mapping: Following the order of ports as defined in **fulladd**

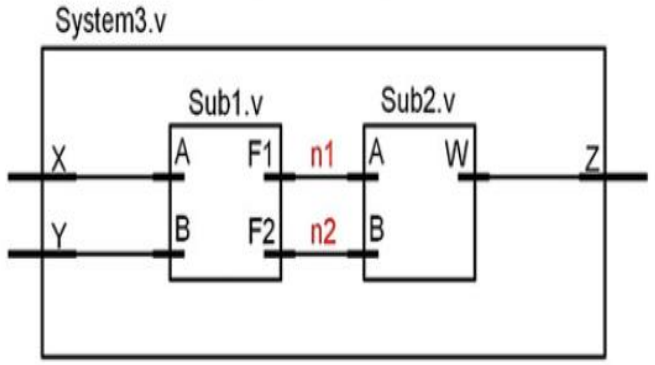
Explicit port mapping

Module defined on p.15: **module** fulladd(Cin,x,y,s,Cout);



(Another example from: “Quick Start Guide to Verilog” p.52-53)

Verilog Structural Design using Explicit Port Mapping



```
module Sub1 (output wire F1, F2,
             input  wire A,  B);

  // behavior here...

endmodule
```

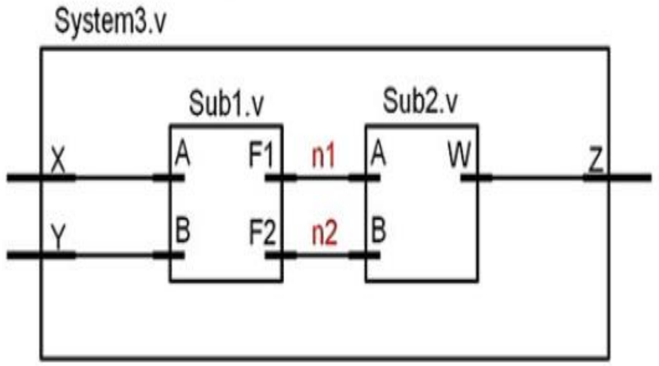
```
module Sub2 (output wire W,
             input  wire A,  B);

  // behavior here...

endmodule
```

```
module System3 (output wire Z,
                input  wire X, Y);

   wire n1, n2;

   Sub1 U0 (.F1(n1), .F2(n2), .A(X),  .B(Y));
   Sub2 U1 (.W(Z),    .A(n1),  .B(n2));

endmodule
```

The lower-level port name is explicitly listed (preceded by a period).

The signal being connected to the lower-level port is listed inside of parenthesis.

Verilog Structural Design using Positional Port Mapping



```
module Sub1 (output wire F1, F2,
             input  wire A,  B);

  // behavior here...

endmodule
```

```
module Sub2 (output wire W,
             input  wire A,  B);

  // behavior here...

endmodule
```

```
module System3 (output wire Z,
                input  wire X, Y);

   wire n1, n2;

   Sub1 U0 (n1, n2, X, Y);
   Sub2 U1 (Z, n1, n2);

endmodule
```

The signals to be connected to the lower-level module must be listed in the same order as they were defined in the lower-level system.

Verilog for Combinational Circuits (Ch.2, p.18)

- Simple combinational circuits - use continuous assignments (**assign**).
- Complicated circuits - use blocking assignments (**=**) in **always** block.

```
module mux2to1(w0,w1,s,f);
    input w0, w1, s;
    output f;
    assign f = s ? w1 : w0;
endmodule
```

If s == true then f = w1; else f= w0;

```
module mux2to1(w0,w1,s,f);
    input w0, w1, s; output reg f;
    always @ (w0, w1, s)
        f = s ? w1 : w0;
endmodule
```

Can be replaced by: **if** (s==1) f = w1; **else** f= w0; (Not allowed in **assign**.)

s	w1	w0	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

s = 0 (false)

f = w0

s = 1 (true)

f = w1

(Ch.2, p.19)

```
module Ex4(W,Y,z);
        input [3:0] W; output reg [1:0] Y; output reg z;

        integer k;

        always @(W)
        begin
                Y=2'bx;
                z=0;

                for(k=0; k<4; k=k+1)
                        if(W[k])
                        begin
                                Y=k;
                                Z=1;
                        end
        end
endmodule
```

Statements in the for-loop may over-write them later.

$\underbrace{k=0;\ k<4;\ k=k+1}$ k: 0 to 3

For example: when k=0

If true (i.e. 1)

Must be type **reg** when being assigned in **always** or **initial** block.

The order of statements inside **always** or **initial** block is important.

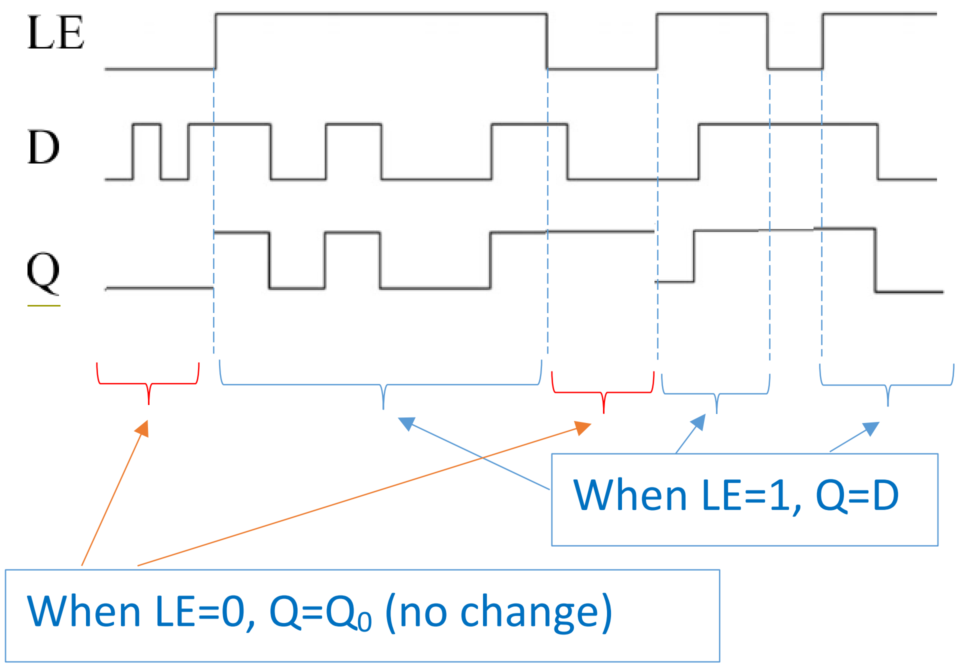
w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

For example, let say w==4'b0111:
When k==0: w[0]==1 → Y=0, Z=1.
When k==1: w[1]==1 → Y=1, Z=1.
When k==2: w[2]==1 → Y=2, Z=1.
When k==3: w[3]==0
Finally, Y=2, Z=1.

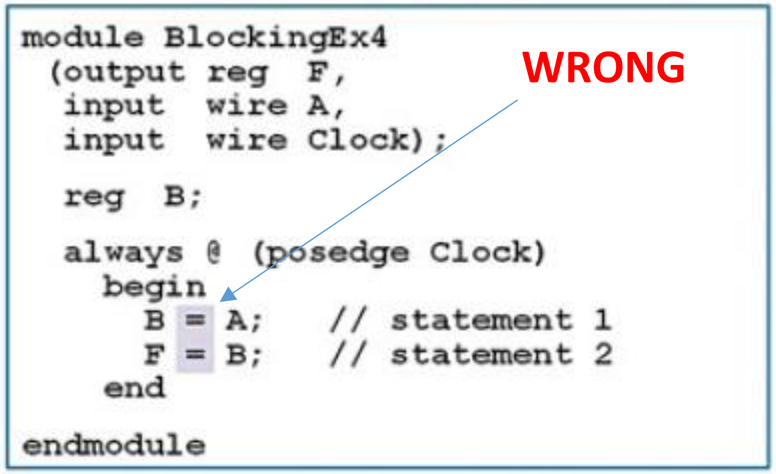
Verilog for Sequential Circuits (Ch.2, p.19)

- Sequential circuits - use non-blocking assignments (**<=**) in **always** block.

```
module  D_latch(D, LE, Q);
input D, LE;
output reg Q;

always @(D or LE)
begin           Same as: (D, LE)
if (LE) Q <= D;
end
endmodule
```

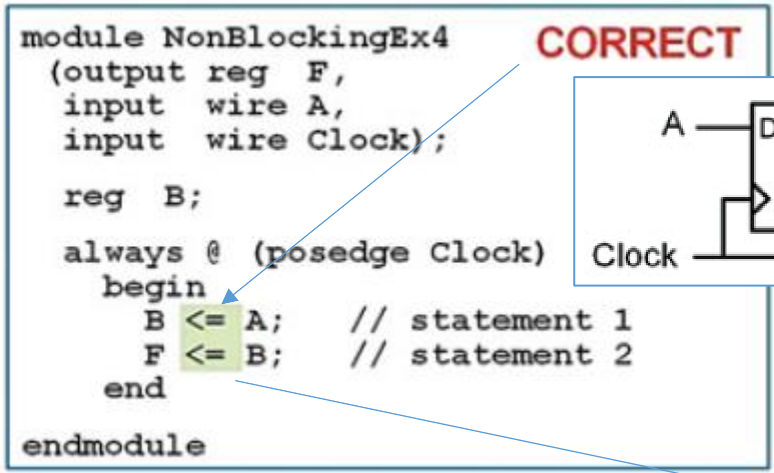


(Another example from: “Quick Start Guide to Verilog” p.72)



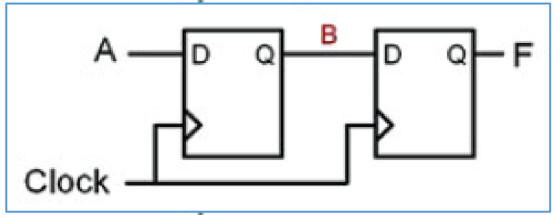
```
module BlockingEx4
  (output reg  F,
   input  wire A,
   input  wire Clock);

   reg  B;

   always @ (posedge Clock)
     begin
       B = A;    // statement 1
       F = B;    // statement 2
     end

endmodule
```

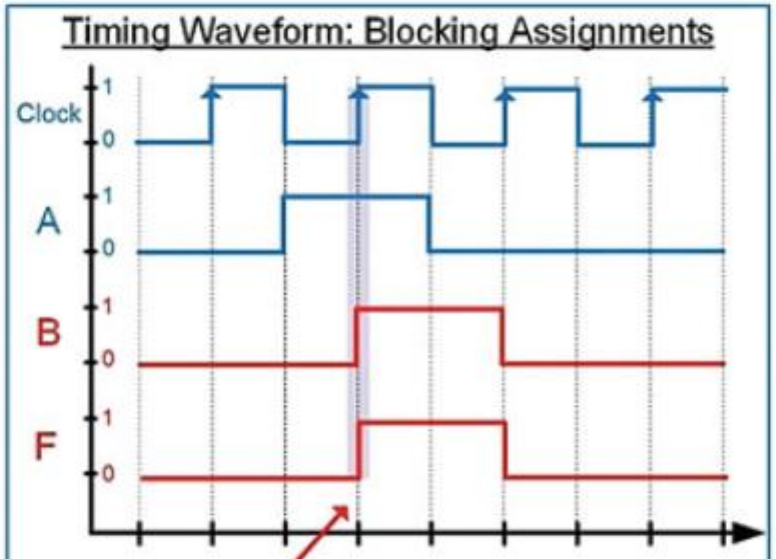
WRONG



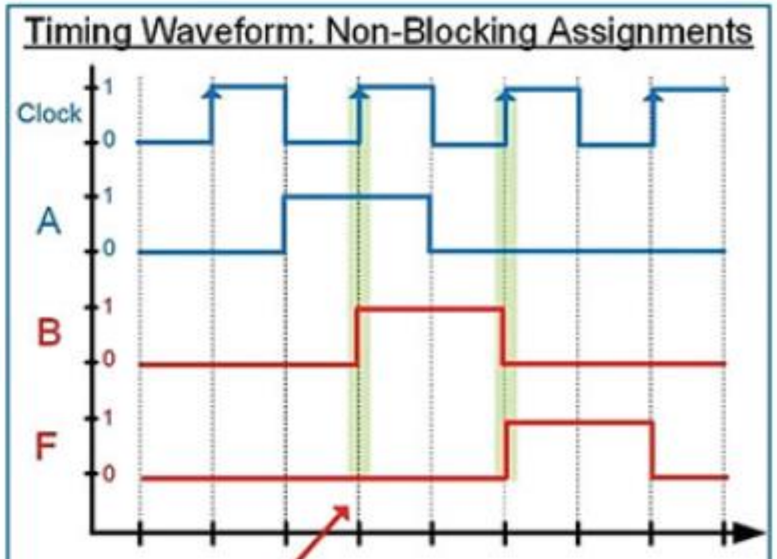
```
module NonBlockingEx4
  (output reg  F,
   input  wire A,
   input  wire Clock);

   reg  B;

   always @ (posedge Clock)
     begin
       B <= A;    // statement 1
       F <= B;    // statement 2
     end

endmodule
```

CORRECT





Since blocking assignments take place immediately, F = B = A.



Since non-blocking assignments take place at the end of the block, it takes two edges for the output F to receive the input A.

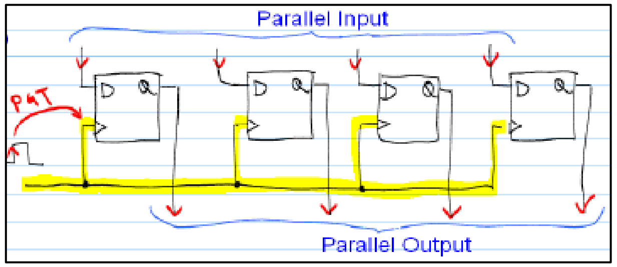
It is like…

begin
  B_temp = A;
  F_temp = B;

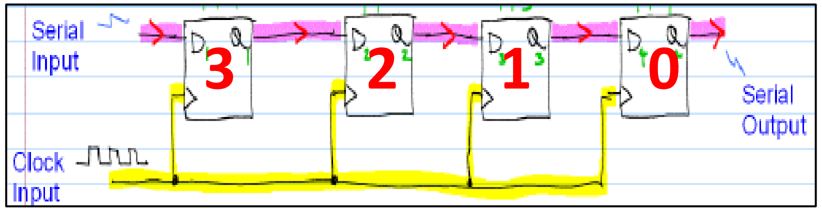
//At the end:
  B = B_temp;
  F = F_temp;
end

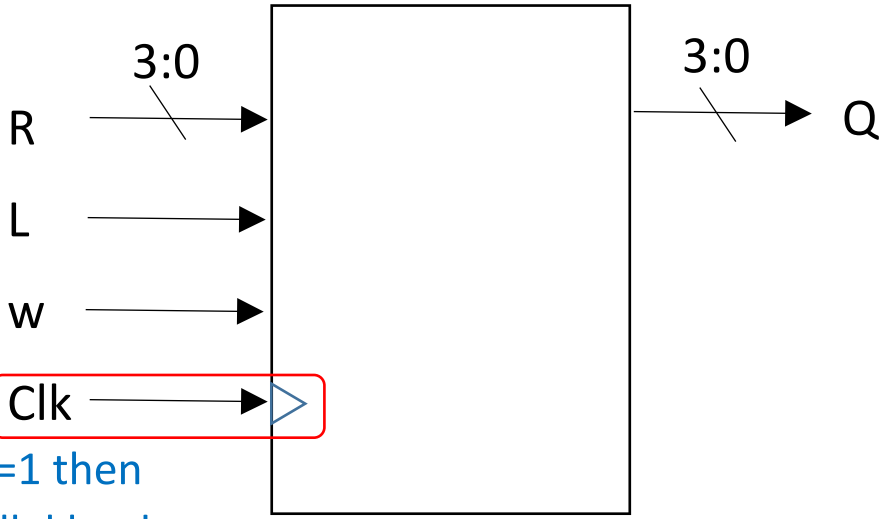
Ch.2, p.20

When L==1
Parallel load:



When
L==0
Shift:





```
module Ex6 (R, L, w, Clk, Q);
        input [ 3:0] R;
        input L, w, Clk;
        output reg [3:0] Q;
        integer k;
        always@(posedge Clk)
                if (L) Q <= R;
                else
                begin
                        for(k=0; k<3; k=k+1)
                                Q[k] <= Q[k+1];
                        Q[3] <= w;
                end
endmodule
```

If L==1 then
Parallel load

k: 0 to 2

Use unblocking assignment (<=) in always blocks for sequential circuits.

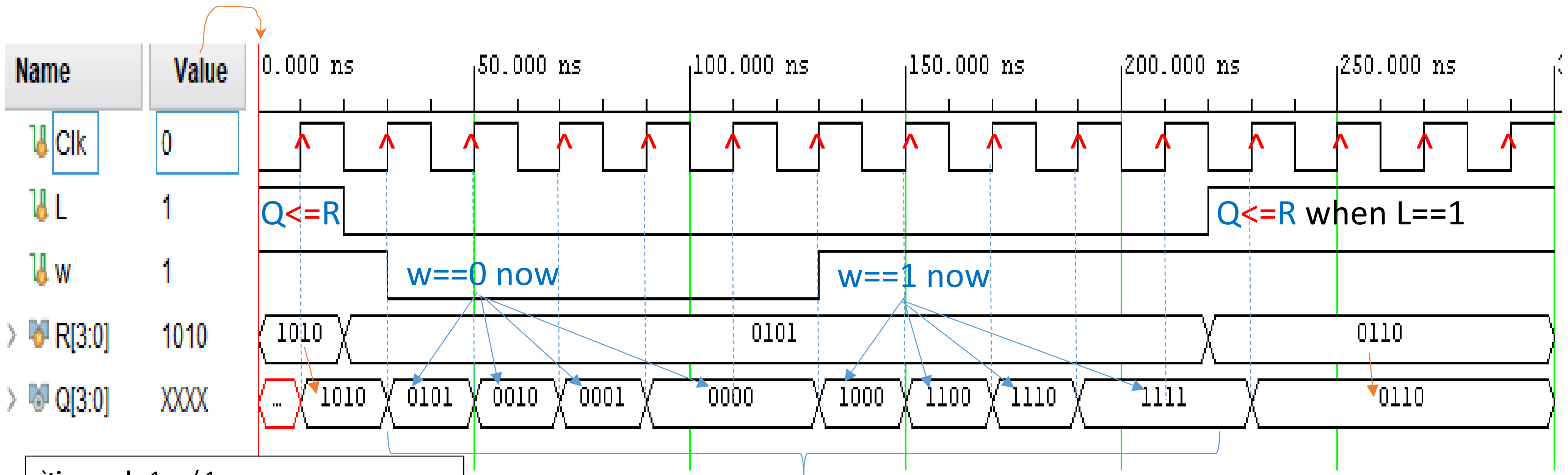
For example, let say, Q==4'b1101 initially:

When k==0: Q[0]<=Q[1] → Q[0]<=0.
When k==1: Q[1]<=Q[2] → Q[1]<=1.
When k==2: Q[2]<=Q[3] → Q[2]<=1.

Q[3]<=w; (w is either 0 or 1.)

Finally, Q becomes w110.
(Q has been shifted right.)

Fig. 2-18 (Ch.2, p.20)



Q will shift right while L==0

When L==0:
Q[0] <= Q[1]
Q[1] <= Q[2]
Q[2] <= Q[3]
Q[3] <= w

```
`timescale 1ns / 1ps

module Ex6_tb();
   reg Clk=0, L=0, w=0; reg [3:0] R;
   wire [3:0] Q;
   Ex6 dut (R, L, w, Clk, Q);

   always #10 Clk = ~Clk;

   initial
   begin
   L=1; w=1; R=10;
   #20 L=0; R=5;
   #10 w=0;
   #100 w=1;
   #90 L=1; R=6;
   end
endmodule
```

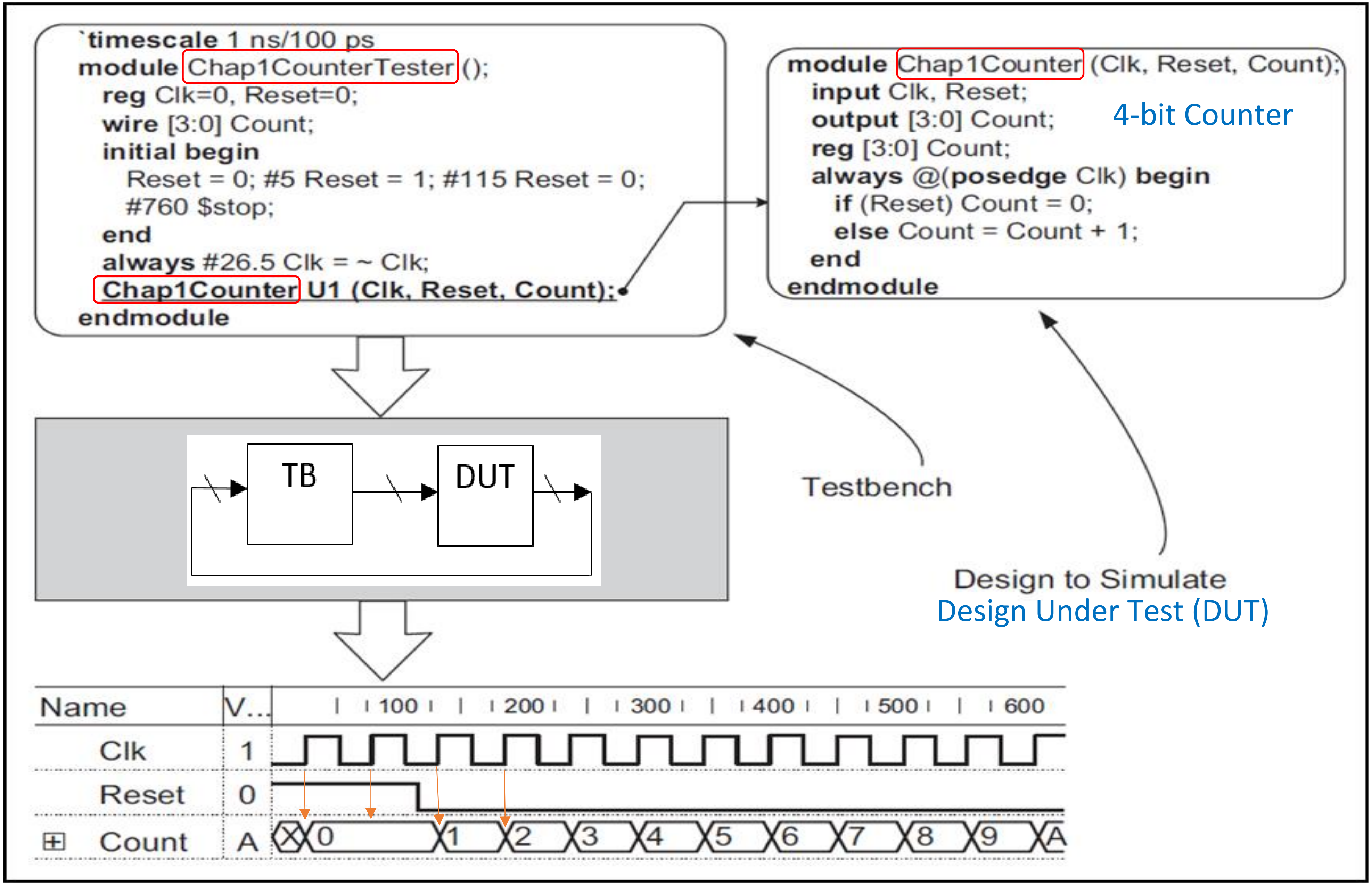
Generating clock pulses

Test sequence

The test-bench module generating the above signals for testing Ex6.

(See Fig. 2.20 on p.23 for another method.)

Simulation (Notes p. 21-23) – see also notes for Lab 1



```
`timescale 1 ns/100 ps
module Chap1CounterTester();
  reg Clk=0, Reset=0;
  wire [3:0] Count;
  initial begin
    Reset = 0; #5 Reset = 1; #115 Reset = 0;
    #760 $stop;
  end
  always #26.5 Clk = ~ Clk;
  Chap1Counter U1 (Clk, Reset, Count);
endmodule
```

```
module Chap1Counter (Clk, Reset, Count);
  input Clk, Reset;
  output [3:0] Count;
  reg [3:0] Count;
  always @(posedge Clk) begin
    if (Reset) Count = 0;
    else Count = Count + 1;
  end
endmodule
```

4-bit Counter

Testbench

Design to Simulate
Design Under Test (DUT)

Figure 20 - Verilog testbench for Ex6 in Figure 18

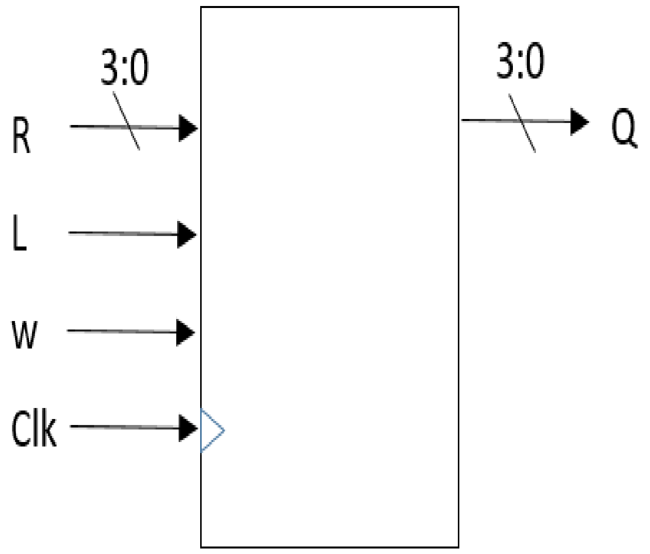
```
`timescale 1ns / 1ps // simulator time unit = 1 ns and precision = 1 ps
module Ex6_tb(); // module name Ex6_tb, input and output is optional here
  wire [3:0] Q; // output in design become input in test-bench
  reg [3:0] R; // input in design become output in test-bench
  reg L;
  reg w;
  reg Clk=0; // Clk is initialized with logic '0'
  initial // initial block is not synthesizeable, meant for simulation only
  begin
     L=1; // Initially L= '1'
    #20 L=0; // 20 ns later L= '0'
    #200 L=1; // another 200 ns later L= '1'
  end
  initial
  begin
     w=1; // Initially w = '1'
    #30 w=0; // 30 ns later w= '0'
    #100 w=1; // another 100 ns later w= '1'
  end
  initial
  begin
    R=4'b1010; // Initially R = "1010"
    #20 R=4'b0011; // 20 ns later R = "0011"
    #200 R=4'b0110; // another 200 ns R = "0110"
  end
  always #10 Clk=~Clk; // Clock pulse train of period = 10 + 10 ns = 20 ns

  Ex6 dut(R,L,w,Clk,Q); //an instance of Ex6 is added and called dut
endmodule
```

initial block for L

initial block for w

initial block for R

always block for clock pulses at Clk



This method uses separate blocks for the test signals. (Easier to control timing.)