Chapter 6

Chapter 6. Dataflow Modelling

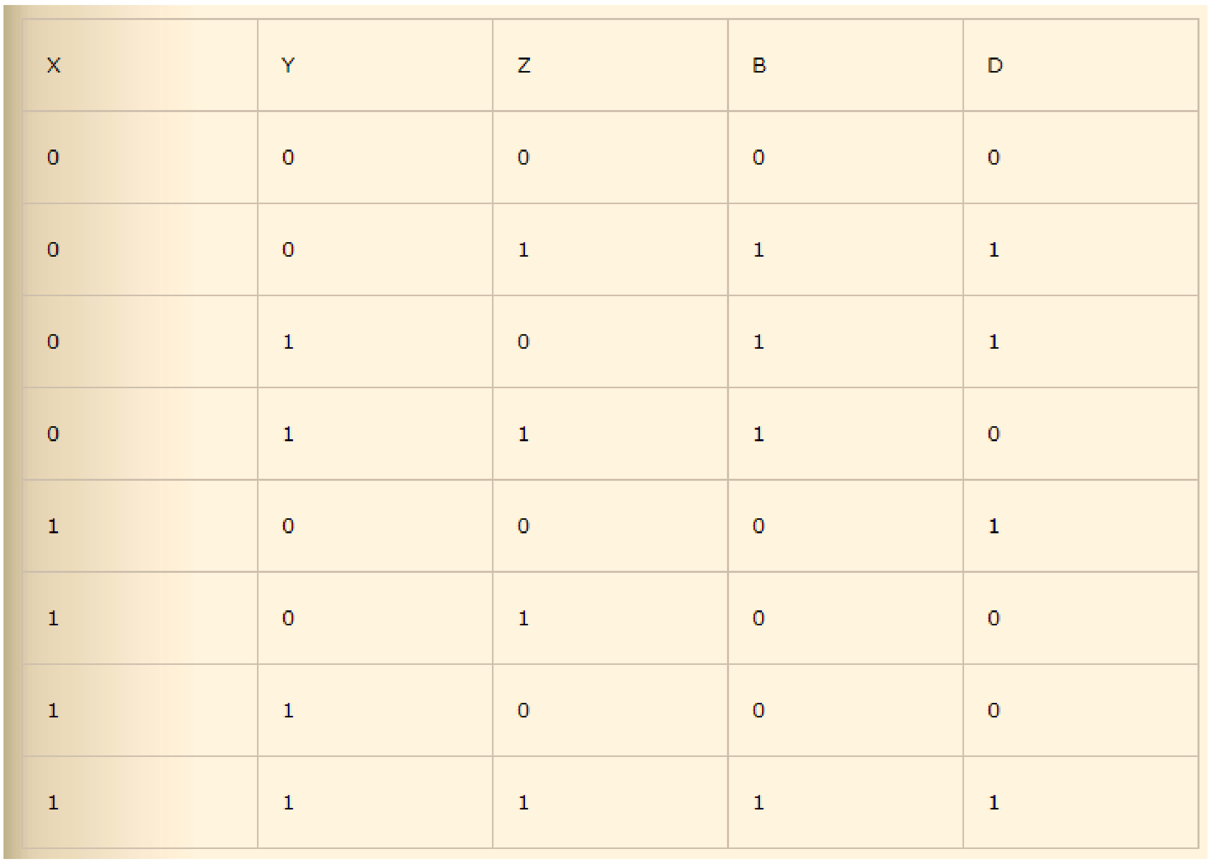
6.7 Exercises

1. A full subtractor has three 1-bit inputs x, y, and z(previous borrow) and two 1-bit outputs D(difference) and B(borrow). The logic equations for D and B are as follows:

D=x'.y'.z + x'.y.z' + x.y'.z' + x.y.z

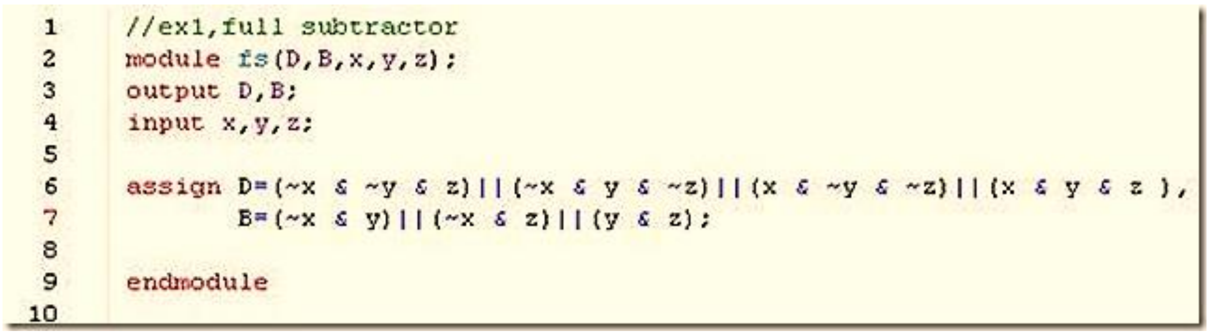
B=x'.y + x'.z + y.z

Write the full Verilog description for the subtractor module, including I/O ports (Remember that + in logic equations corresponds to a logical or operator (||) in dataflow). Instantiate the subtractor inside a stimulus block and test all eight possible combinations of x, y, and z given in the following truth table.

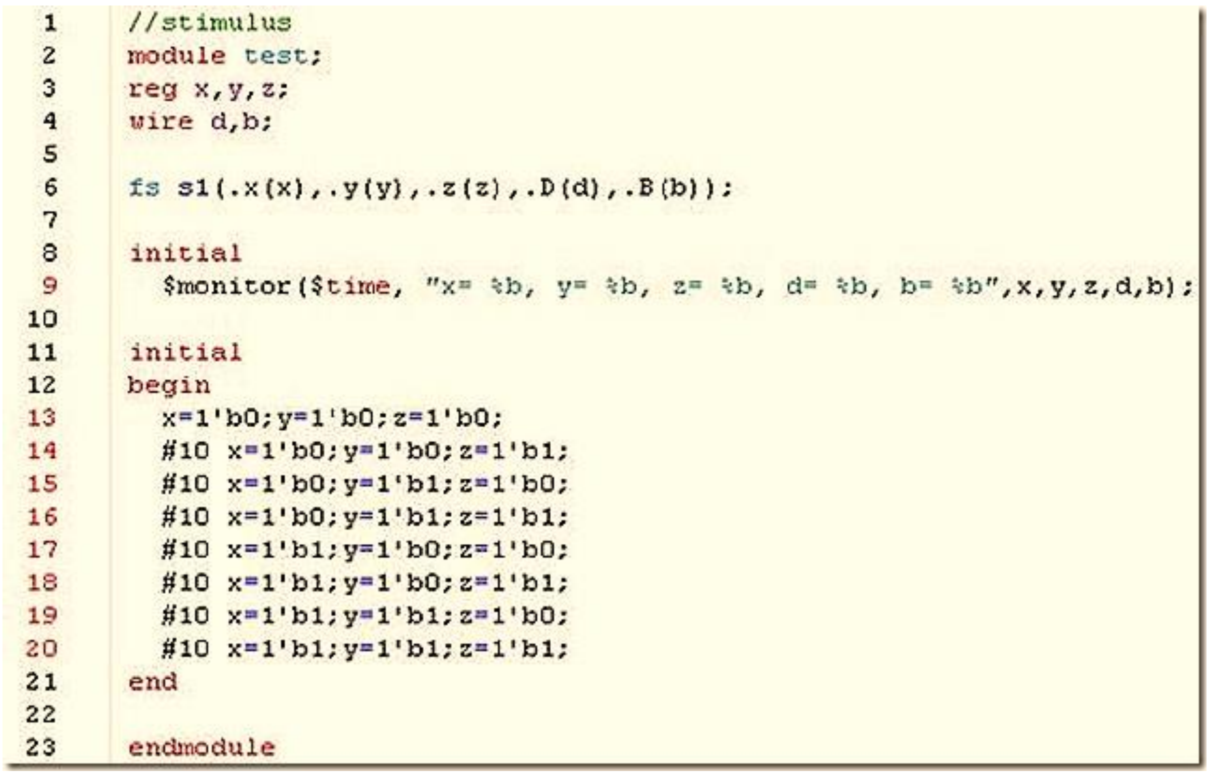


X	Y	Z	B	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

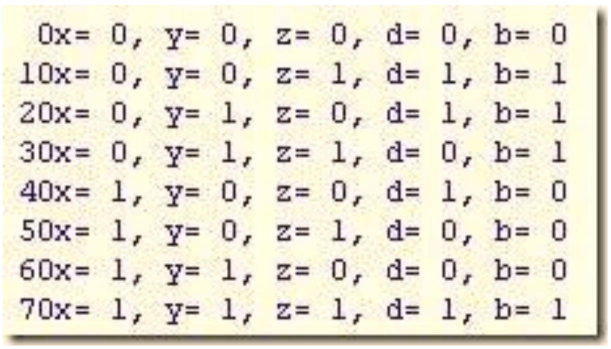
My answer:



```verilog
//ex1,full subtractor
module fs(D,B,x,y,z);
output D,B;
input x,y,z;

assign D=(~x & ~y & z)||(~x & y & ~z)||(x & ~y & ~z)||(x & y & z ),
       B=(~x & y)||(~x & z)||(y & z);

endmodule
```



```verilog
//stimulus
module test;
reg x,y,z;
wire d,b;

fs s1(.x(x),.y(y),.z(z),.D(d),.B(b));

initial
  $monitor($time, "x= %b, y= %b, z= %b, d= %b, b= %b",x,y,z,d,b);

initial
begin
  x=1'b0;y=1'b0;z=1'b0;
  #10 x=1'b0;y=1'b0;z=1'b1;
  #10 x=1'b0;y=1'b1;z=1'b0;
  #10 x=1'b0;y=1'b1;z=1'b1;
  #10 x=1'b1;y=1'b0;z=1'b0;
  #10 x=1'b1;y=1'b0;z=1'b1;
  #10 x=1'b1;y=1'b1;z=1'b0;
  #10 x=1'b1;y=1'b1;z=1'b1;
end

endmodule
```



```
 0x= 0, y= 0, z= 0, d= 0, b= 0
10x= 0, y= 0, z= 1, d= 1, b= 1
20x= 0, y= 1, z= 0, d= 1, b= 1
30x= 0, y= 1, z= 1, d= 0, b= 1
40x= 1, y= 0, z= 0, d= 1, b= 0
50x= 1, y= 0, z= 1, d= 0, b= 0
60x= 1, y= 1, z= 0, d= 0, b= 0
70x= 1, y= 1, z= 1, d= 1, b= 1
```

2. A magnitude comparator checks if one number is greater than or equal to or less than another number. A 4-bit magnitude comparator takes two 4-bit numbers, A and B, as input. We write the bits in A and B as follows. The leftmost bit is the most significant bit.

A=A(3)A(2)A(1)A(0)

B=B(3)B(2)B(1)B(0)

The magnitude can be compared by comparing the numbers bit by bit, starting with the most significant bit. If any bit mismatches, the number with bit 0 is the lower number. To realize this functionality in logic equations, let us define an intermediate variable. Notice that the function below is an xnor function.

x(i)=A(i)B(i)+A(i)’B(i)’

The three outputs of the magnitude comparator are A_gt_B, A_lt_B, A_eq_B. They are define with the following logic equations:

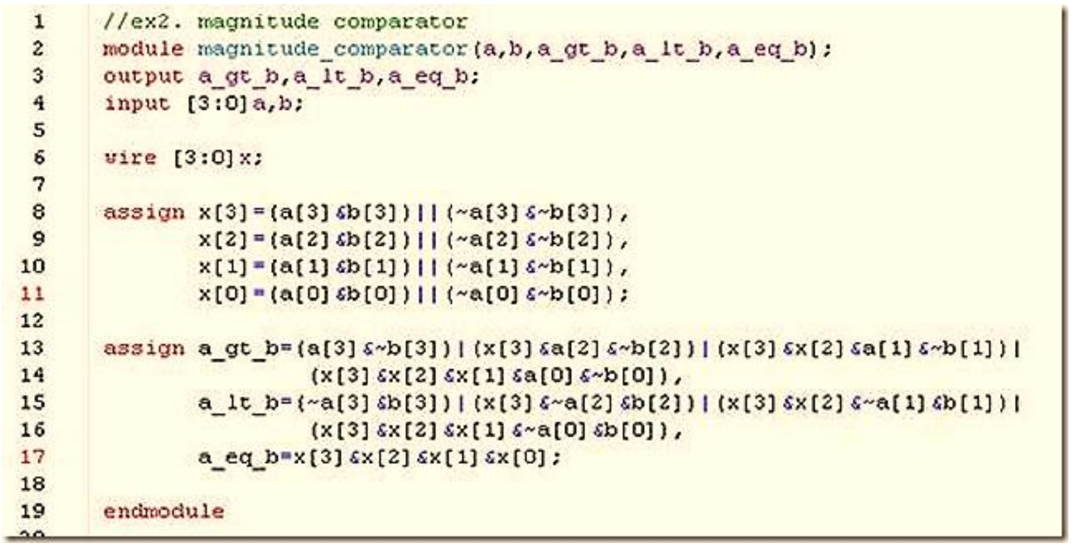
A_gt_B=A(3)B(3)’ + x(3).A(2).B(2)' + x(3).x(2).A(1).B(1)' + x(3).x(2).x(1).A(0).B(0)'

A_lt_B = A(3)'.B(3) + x(3).A(2)'.B(2) + x(3).x(2).A(1)'.B(1) + x(3).x(2).x(1).A(0)'.B(0)

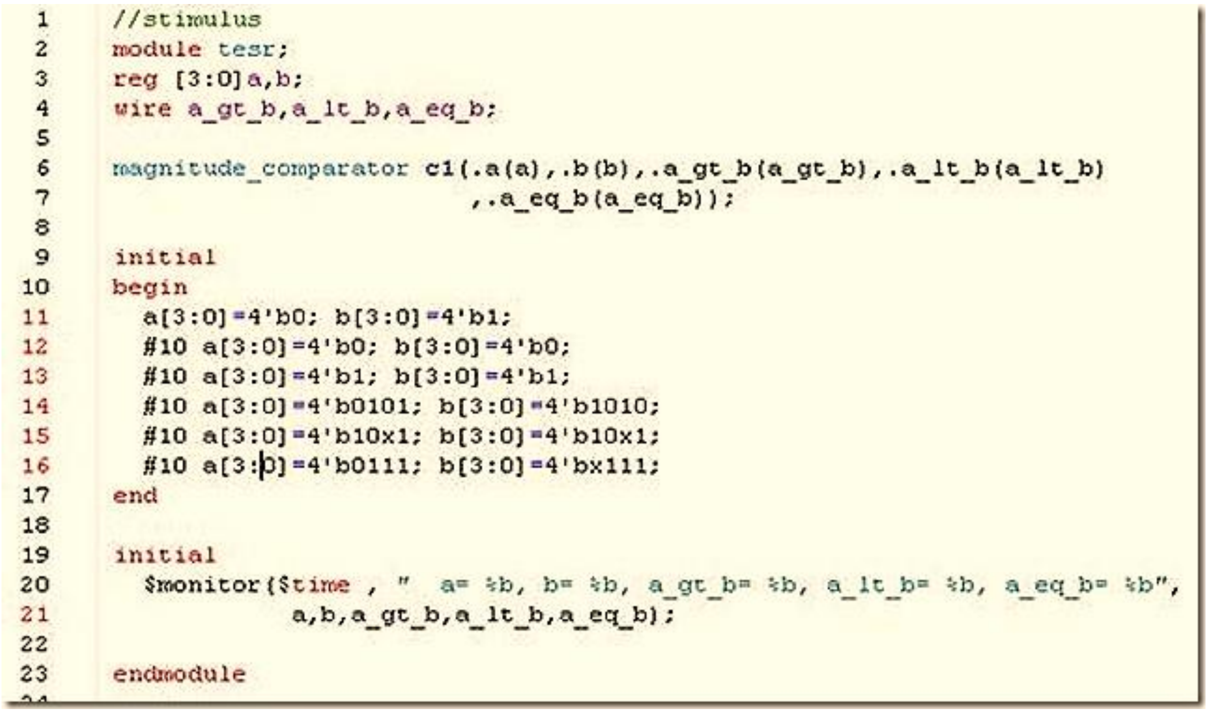
A_eq_B = x(3).x(2).x(1).x(0)

Write the Verilog description of the module magnitude_comparator. Instantiate the magnitude comparator inside the stimulus module and try out a few combinations of A and B.

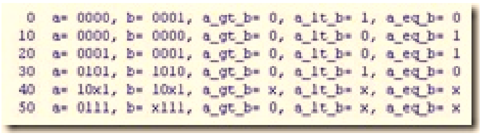
My answer:



```
//ex2. magnitude comparator
module magnitude_comparator(a,b,a_gt_b,a_lt_b,a_eq_b);
output a_gt_b,a_lt_b,a_eq_b;
input [3:0]a,b;

wire [3:0]x;

assign x[3]=(a[3]&b[3])||(~a[3]&~b[3]),
       x[2]=(a[2]&b[2])||(~a[2]&~b[2]),
       x[1]=(a[1]&b[1])||(~a[1]&~b[1]),
       x[0]=(a[0]&b[0])||(~a[0]&~b[0]);

assign a_gt_b=(a[3]&~b[3])|(x[3]&a[2]&~b[2])|(x[3]&x[2]&a[1]&~b[1])|
              (x[3]&x[2]&x[1]&a[0]&~b[0]),
       a_lt_b=(~a[3]&b[3])|(x[3]&~a[2]&b[2])|(x[3]&x[2]&~a[1]&b[1])|
              (x[3]&x[2]&x[1]&~a[0]&b[0]),
       a_eq_b=x[3]&x[2]&x[1]&x[0];

endmodule
```



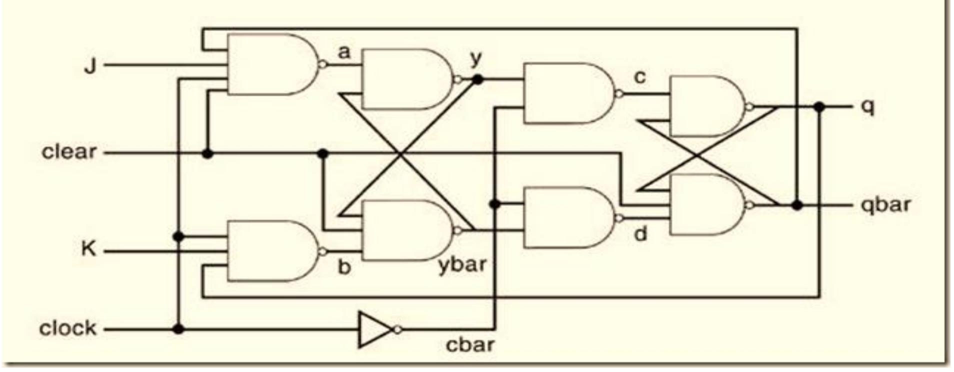
```
//stimulus
module tesr;
reg [3:0]a,b;
wire a_gt_b,a_lt_b,a_eq_b;

magnitude_comparator c1(.a(a),.b(b),.a_gt_b(a_gt_b),.a_lt_b(a_lt_b)
                        ,.a_eq_b(a_eq_b));

initial
begin
  a[3:0]=4'b0; b[3:0]=4'b1;
  #10 a[3:0]=4'b0; b[3:0]=4'b0;
  #10 a[3:0]=4'b1; b[3:0]=4'b1;
  #10 a[3:0]=4'b0101; b[3:0]=4'b1010;
  #10 a[3:0]=4'b10x1; b[3:0]=4'b10x1;
  #10 a[3:0]=4'b0111; b[3:0]=4'bx111;
end

initial
  $monitor($time , "  a= %b, b= %b, a_gt_b= %b, a_lt_b= %b, a_eq_b= %b",
           a,b,a_gt_b,a_lt_b,a_eq_b);

endmodule
```



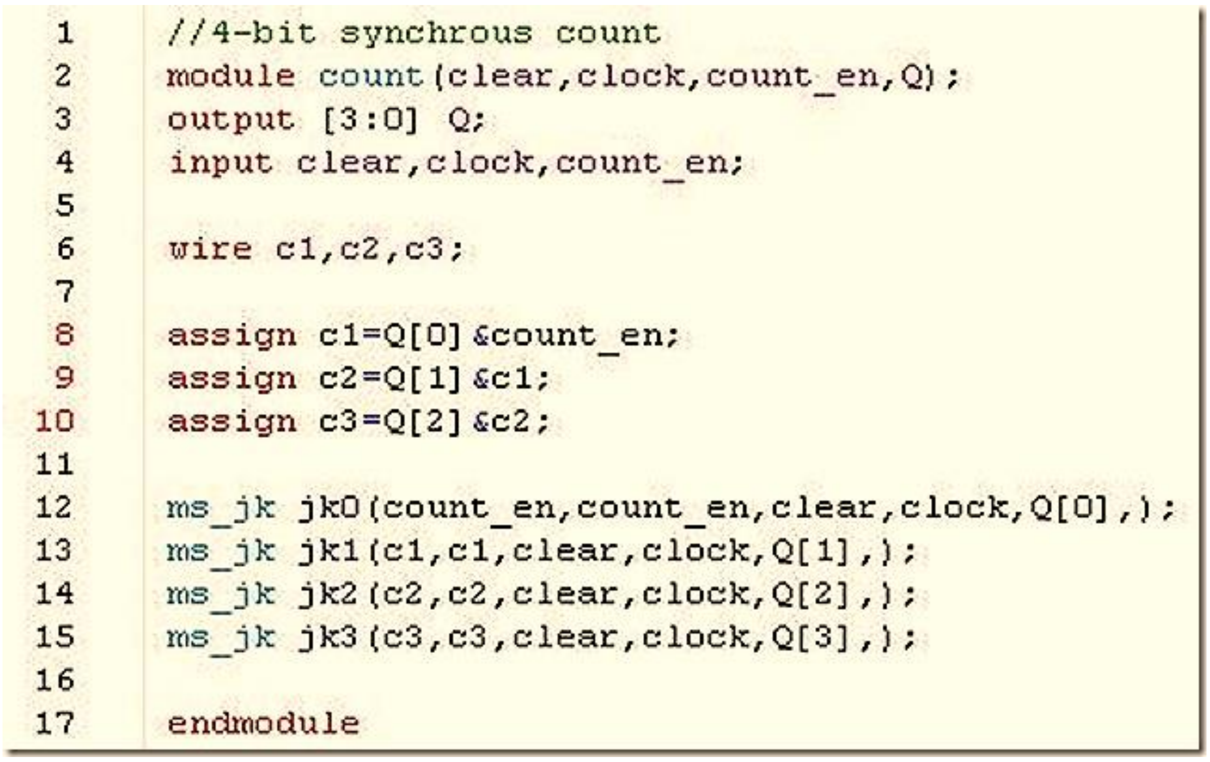
```
 0  a= 0000, b= 0001, a_gt_b= 0, a_lt_b= 1, a_eq_b= 0
10  a= 0000, b= 0000, a_gt_b= 0, a_lt_b= 0, a_eq_b= 1
20  a= 0001, b= 0001, a_gt_b= 0, a_lt_b= 0, a_eq_b= 1
30  a= 0101, b= 1010, a_gt_b= 0, a_lt_b= 1, a_eq_b= 0
40  a= 10x1, b= 10x1, a_gt_b= x, a_lt_b= x, a_eq_b= x
50  a= 0111, b= x111, a_gt_b= 0, a_lt_b= x, a_eq_b= x
```

3. A synchronous counter can be designed by using master-slave JK flipflops. Design a 4-bit synchronous counter. Circuit diagrams for the synchronous counter and the JK flipflop are given below. The clear signal is active low. Data gets latched on the positive edge of clock, and the output of the flipflop appears on the negative edge of clock. Counting is disabled when count_enable signal is low. Write the dataflow description for the synchronous counter. Write a stimulus file that exercises clear and count_enable. Display the output count Q[3:0].

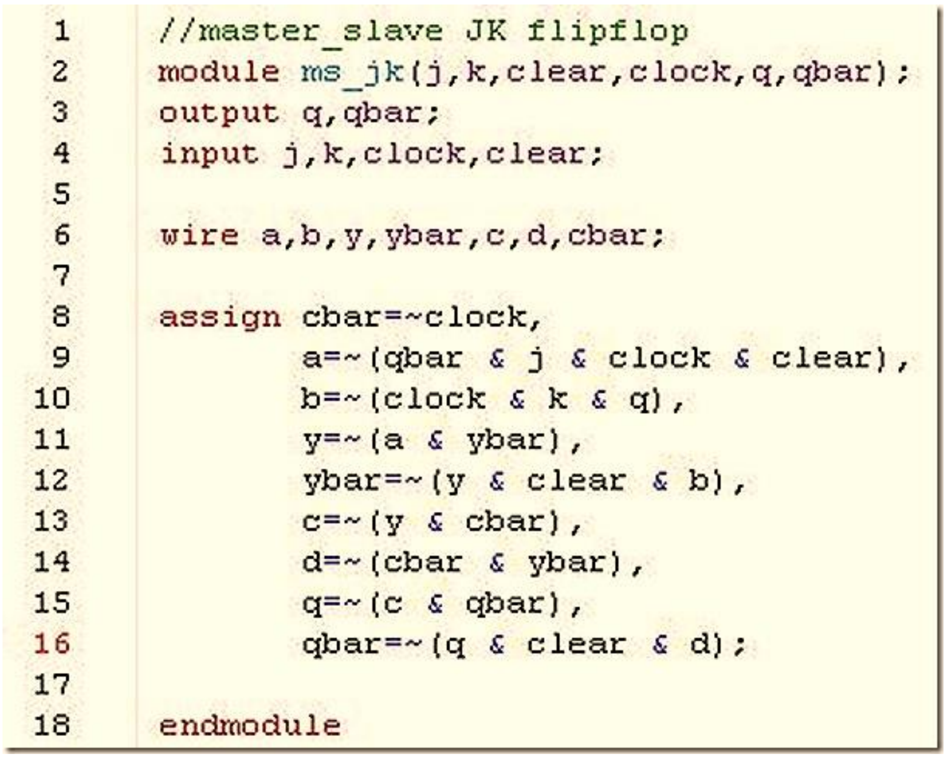
Figure 6-5. Master-Slave JK-flipflop



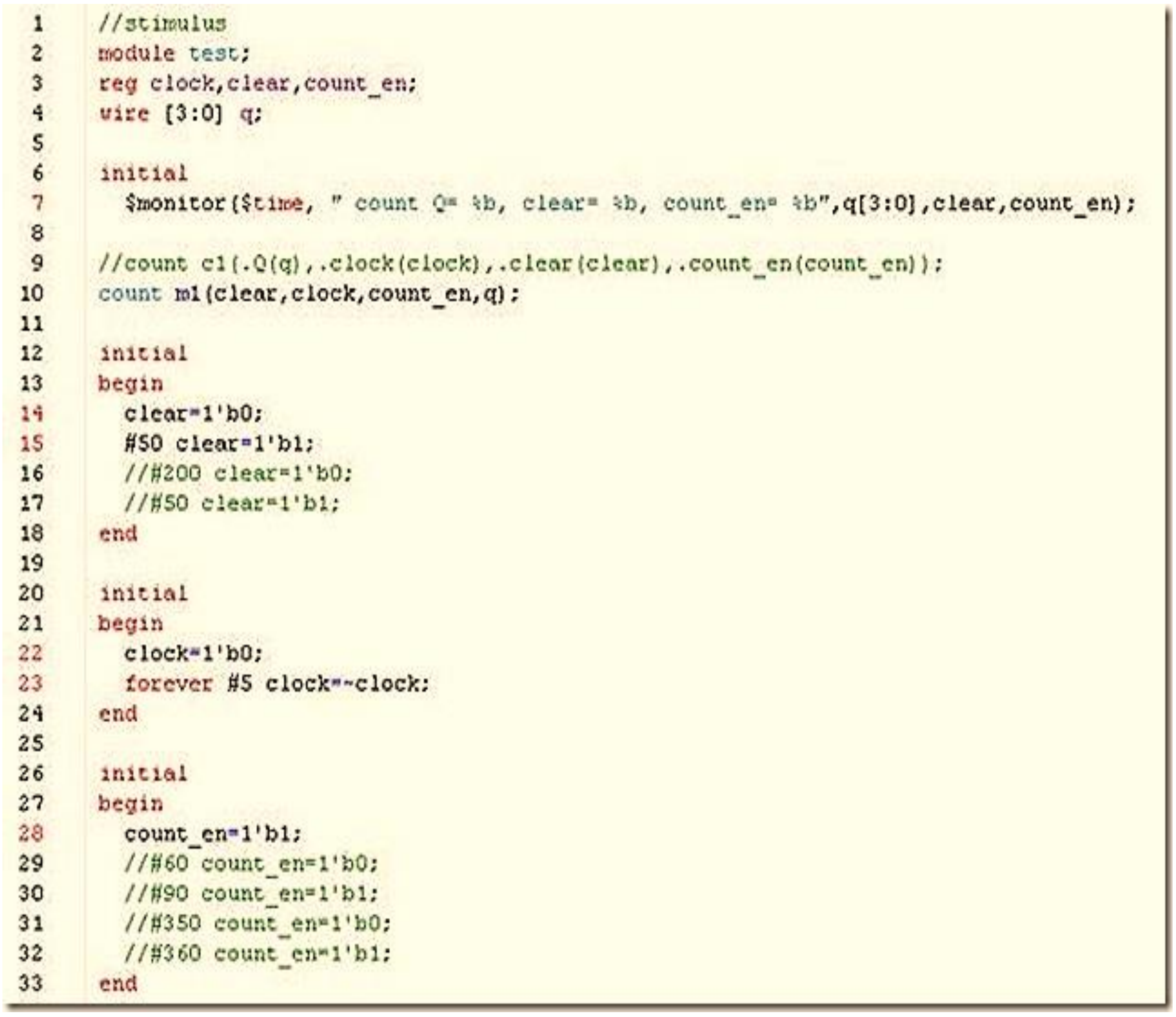
My answer:



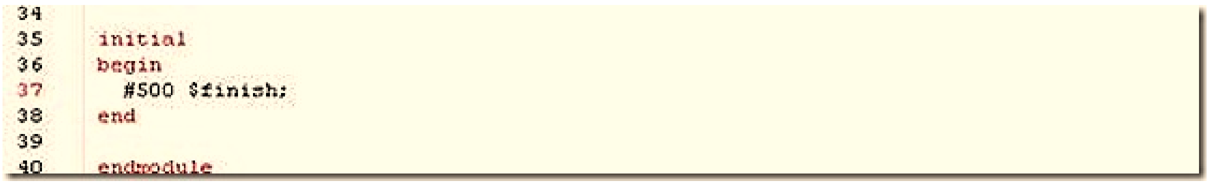
```
//4-bit synchrous count
module count(clear,clock,count_en,Q);
output [3:0] Q;
input clear,clock,count_en;

wire c1,c2,c3;

assign c1=Q[0]&count_en;
assign c2=Q[1]&c1;
assign c3=Q[2]&c2;

ms_jk jk0(count_en,count_en,clear,clock,Q[0],);
ms_jk jk1(c1,c1,clear,clock,Q[1],);
ms_jk jk2(c2,c2,clear,clock,Q[2],);
ms_jk jk3(c3,c3,clear,clock,Q[3],);

endmodule
```



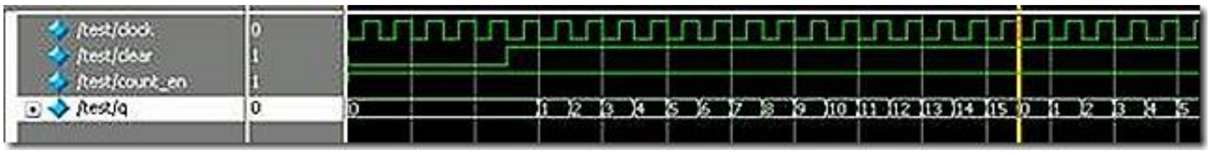
```
//master_slave JK flipflop
module ms_jk(j,k,clear,clock,q,qbar);
output q,qbar;
input j,k,clock,clear;

wire a,b,y,ybar,c,d,cbar;

assign cbar=~clock,
       a=~(qbar & j & clock & clear),
       b=~(clock & k & q),
       y=~(a & ybar),
       ybar=~(y & clear & b),
       c=~(y & cbar),
       d=~(cbar & ybar),
       q=~(c & qbar),
       qbar=~(q & clear & d);

endmodule
```



```
//stimulus
module test;
reg clock,clear,count_en;
wire [3:0] q;

initial
  $monitor($time, " count Q= %b, clear= %b, count_en= %b",q[3:0],clear,count_en);

//count c1(.Q(q),.clock(clock),.clear(clear),.count_en(count_en));
count m1(clear,clock,count_en,q);

initial
begin
  clear=1'b0;
  #50 clear=1'b1;
  //#200 clear=1'b0;
  //#50 clear=1'b1;
end

initial
begin
  clock=1'b0;
  forever #5 clock=~clock;
end

initial
begin
  count_en=1'b1;
  //#60 count_en=1'b0;
  //#90 count_en=1'b1;
  //#350 count_en=1'b0;
  //#360 count_en=1'b1;
end
```



```

initial
begin
  #500 $finish;
end

endmodule
```


/test/clock 0
/test/clear 1
/test/count_en 1
/test/q 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5