University of California, Irvine

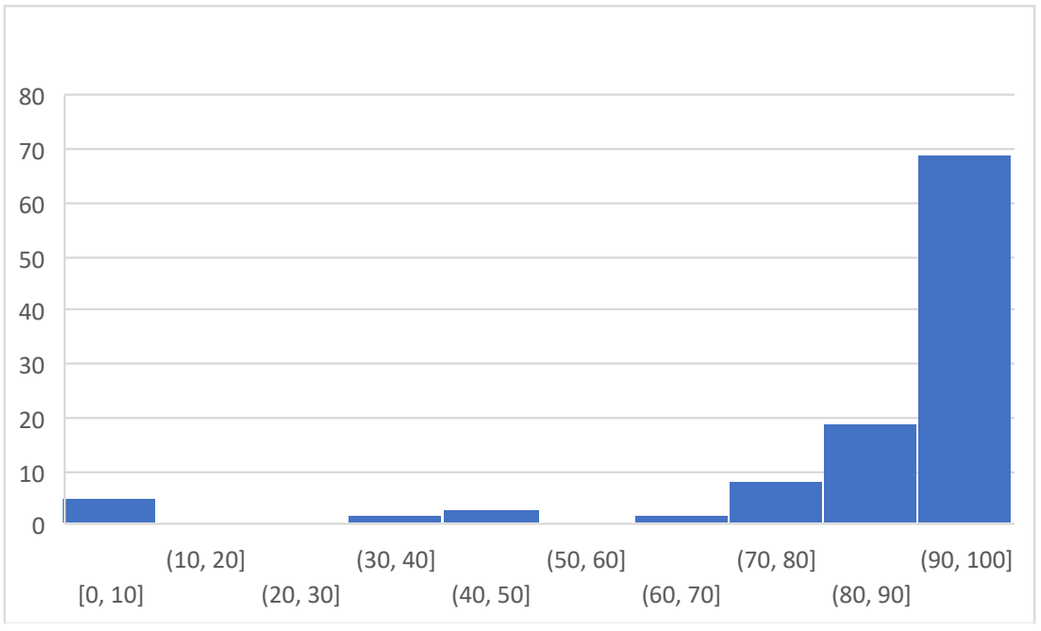
# EECS 31L: INTRODUCTION TO DIGITAL DESIGN LAB
# LECTURE 4

**Salma Elmalaki**
**salma.elmalaki@uci.edu**

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

# LECTURE 4: LOGISTICS

- **All questions should be posted on canvas**
- Lab 2 deadline is Monday 2/1/2021 at 11:00 pm on canvas
  - late submission between 11:0pm to midnight (50% deduction)
- Lab 1 is graded
  - Mean = 87.5




- You have one week after the grades are released for reconsideration

# LECTURE 4: OVERVIEW

- Behavioral Description
- Sequential statements
  - if, case, for, while, …
  - Storage elements
  - DFF
  - JK FF
  - Binary counters
- Blocking vs. non-blocking assignment

# LECTURE 4: OVERVIEW

- Behavioral Description
- Sequential statements
  - if, case, for, while, …
  - Storage elements
  - DFF
  - JK FF
  - Binary counters
- Blocking vs. non-blocking assignment

# BEHAVIORAL DESCRIPTION

- How output behaves in response to input changes
- Details of logic diagram not needed
- Useful for
  - Combinational circuits modeling
  - Sequential circuits modeling
- Useful for complex designs
  - Arithmetic units
- Major behavioral description
  - always/initial
  - process(VHDL)

# BEHAVIORAL DESCRIPTION

- How output behaves in response to input changes
- Details of logic diagram not needed
- Useful for
  - Combinational circuits modeling
  - Sequential circuits modeling
- Useful for complex designs
  - Arithmetic units
- Major behavioral description
  - always/initial
  - process(VHDL)

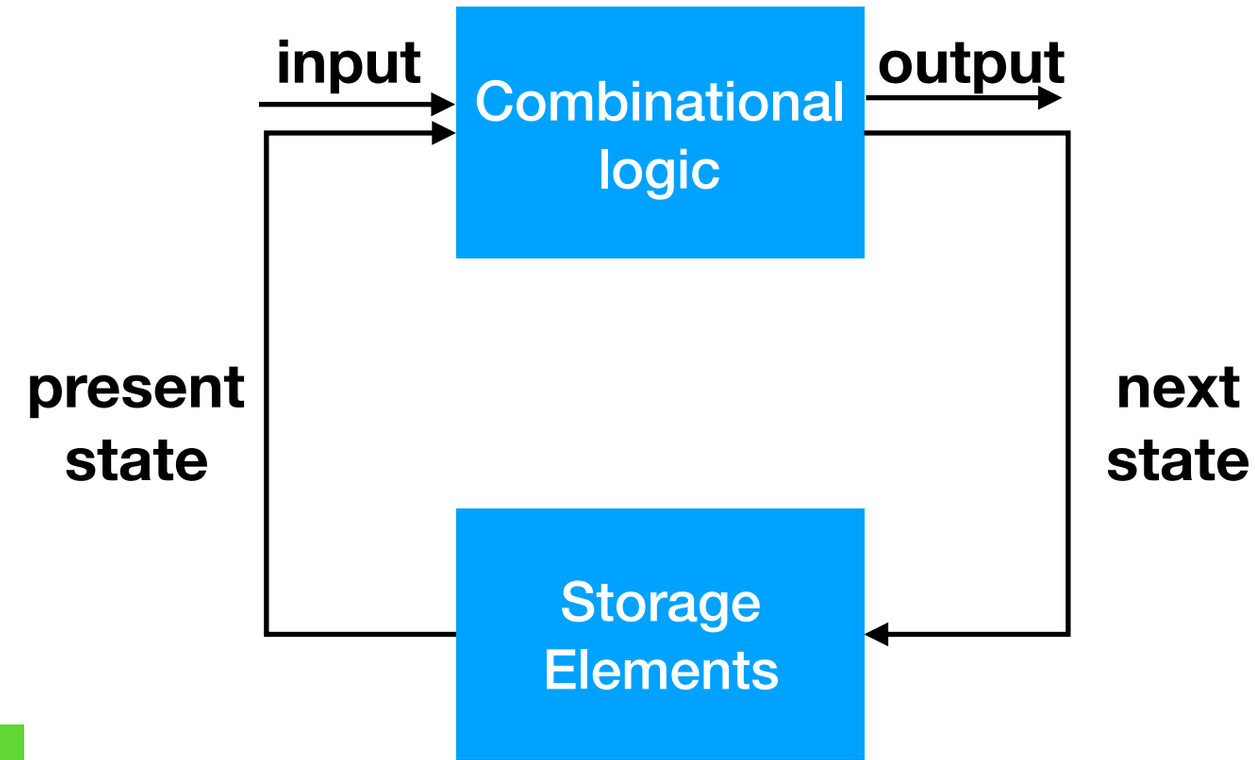
# COMBINATIONAL VS. SEQUENTIAL CIRCUITS

- Output depends on input


- **Decoder/Encoder**
- **Multiplixer/Demultiplexer**
- **Adder**
- **Address Decoder**

In principle, the system requires no memory and can be implemented using conventional logic gates

- Output depends on input and past state




- **Latches**
- **Flip-flops**
- **Shift registers**
- **Counters**
- **CPU controller**

# BEHAVIORAL DESCRIPTION

- Basic statements of behavioral description
  - always/initial blocks
- Sequential statements
  - If
  - Case
  - Wait
  - Loop
- Blocking vs. non-blocking assignments

# BEHAVIORAL DESCRIPTION

- Basic statements of behavioral description
  - always/initial blocks
- Sequential statements
  - If
  - Case
  - Wait
  - Loop
- Blocking vs. non-blocking assignments

# ALWAYS

```
always @(sensitivity_list)
  begin
    //sequential_statements;
  end
```

- Syntax: always @ (*)
  - Sensitive to all signals inside always
- A variable should never be assigned a value in more than one always block
- Any signal assigned inside always should be
  - reg
  - Integrer
  - wire cannot be assigned inside always
- Execute **repeatedly** until the simulation is terminated

# INITIAL

```
initial
  begin
    //sequential_statements;
  end
```

- Similar to always format
- No sensitivity list needed
- Executed **only once** at the **beginning** of simulation
- Usually used
  - To initialize variables
  - Inside test benched
  - Not synthesizable

# BEHAVIORAL DESCRIPTION

- Basic statements of behavioral description
  - always/initial blocks
- Sequential statements
  - If
  - Case
  - Wait
  - Loop
- Blocking vs. non-blocking assignments

# IF STATEMENTS

- **If** statement is used to choose which statement should be executed depending on the conditional expression

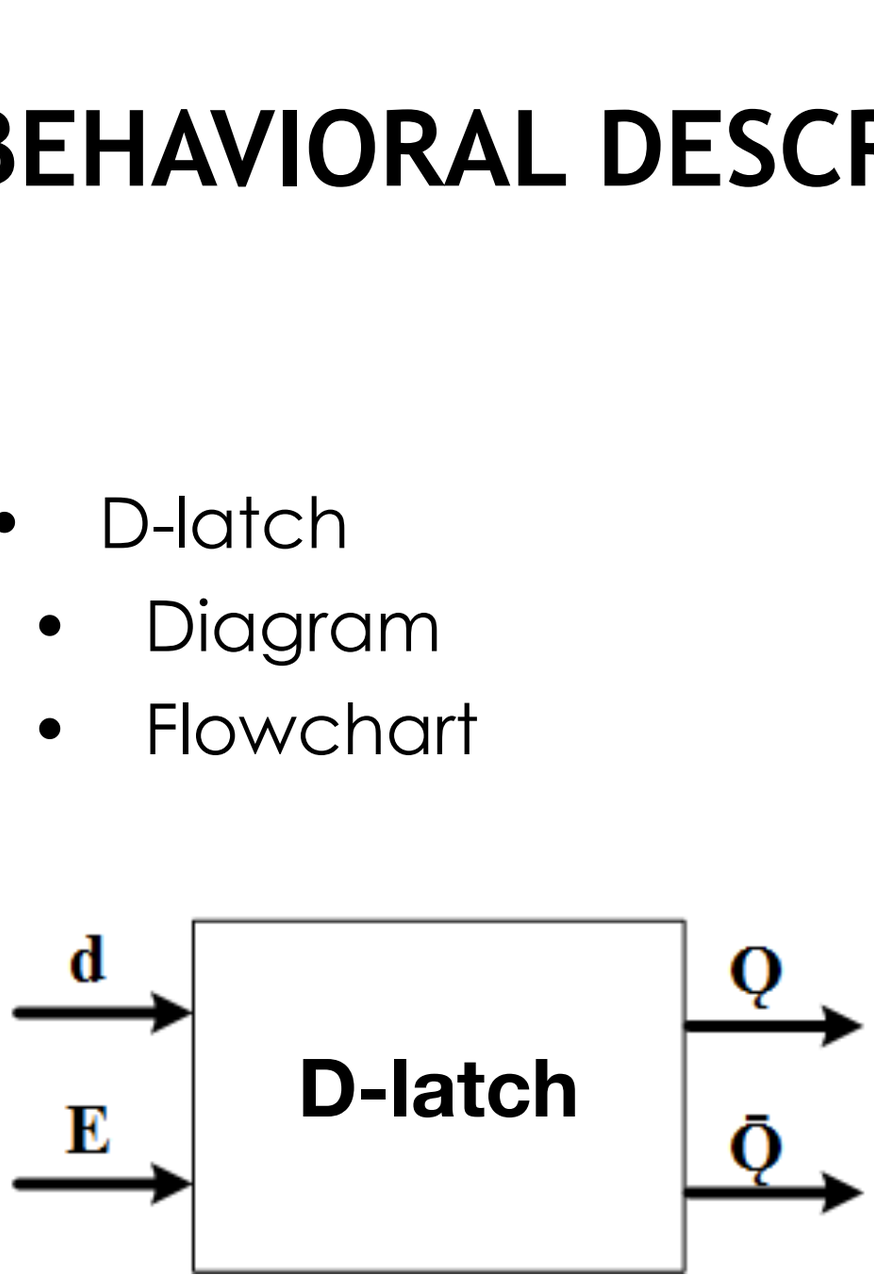
```
if (conditional expression) begin
   statement1;
   statement2;
end else if (conditional expression)
  statement3;
else
  statement4;
```

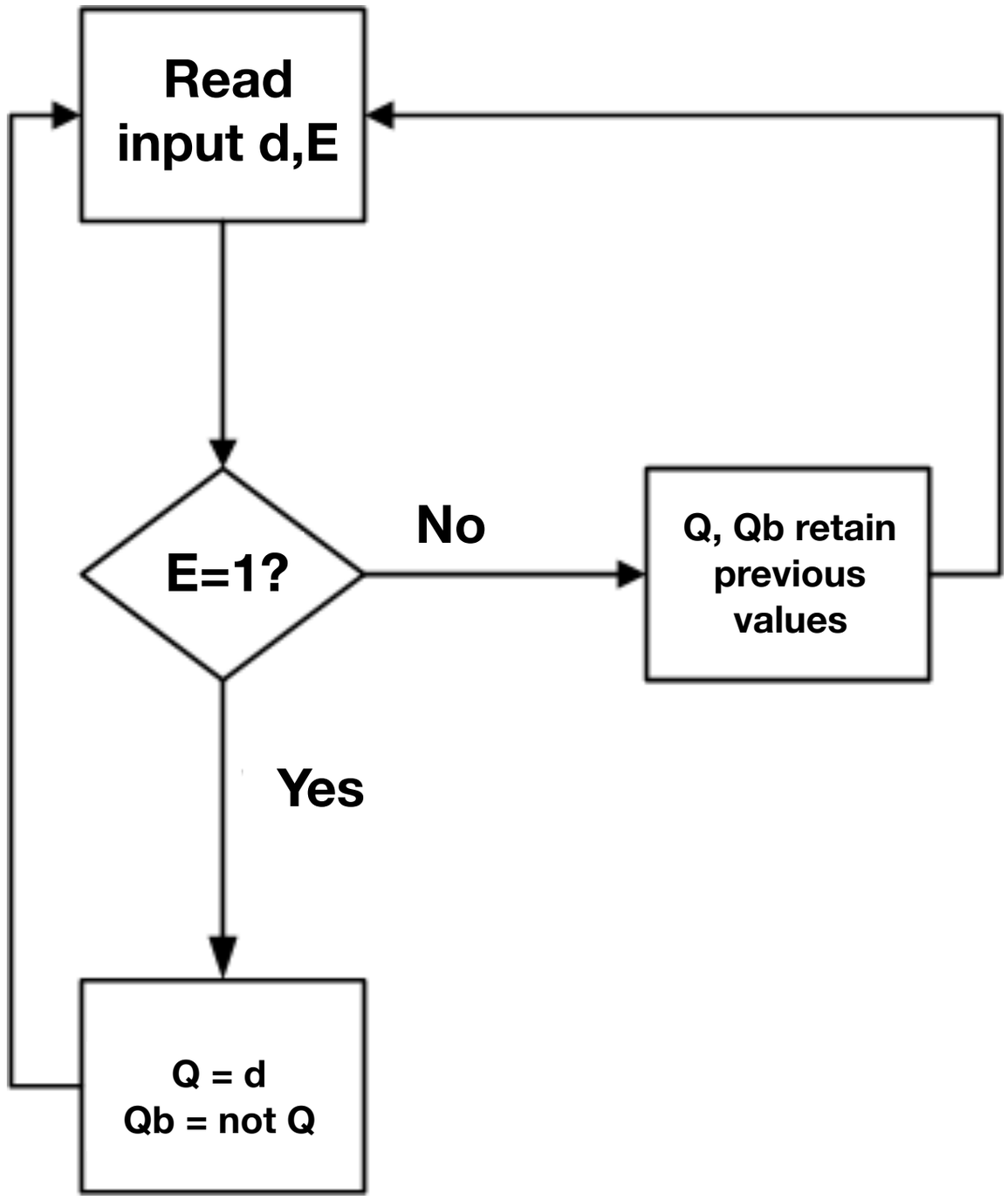
If only one statement no need for begin/end

```
if (x==a && y==b)
   output = 1'b0;
else if (x==a && y==c)
   output = 1'b1;
else
  output = 1'bZ;
```

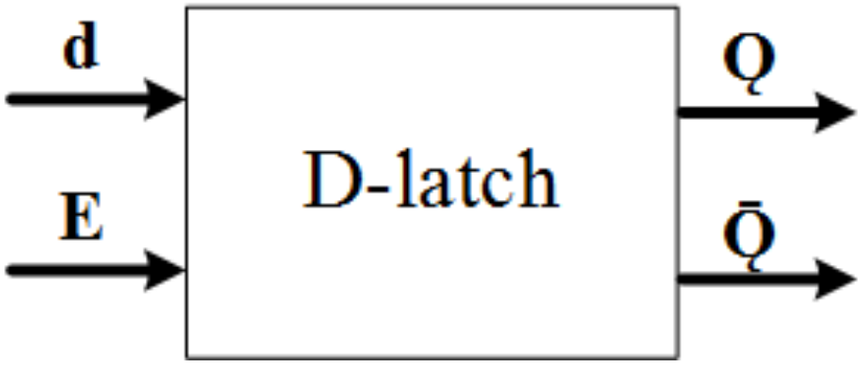
# BEHAVIORAL DESCRIPTION EXAMPLE: D-LATCH

- D-latch
  - Diagram
  - Flowchart

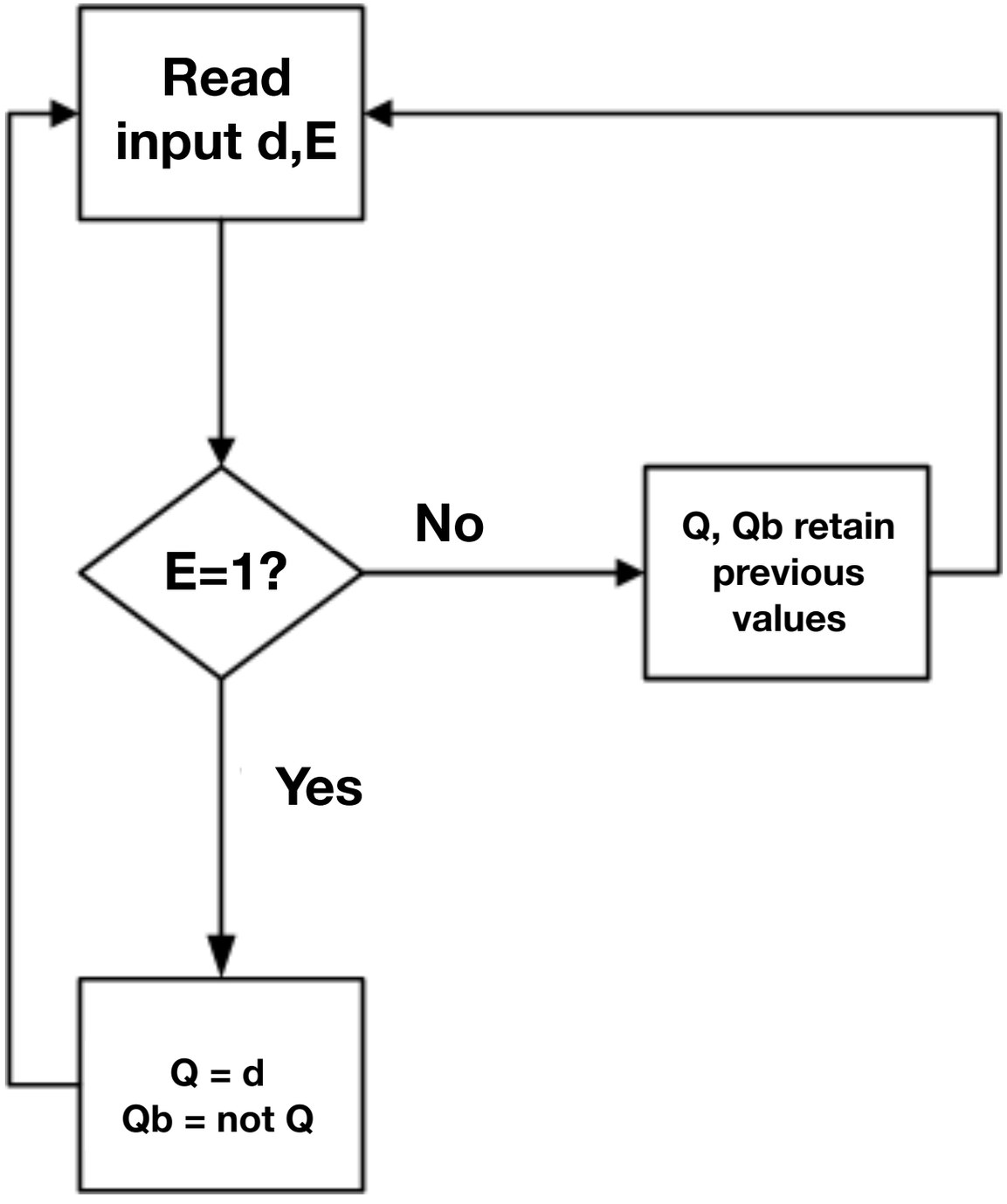


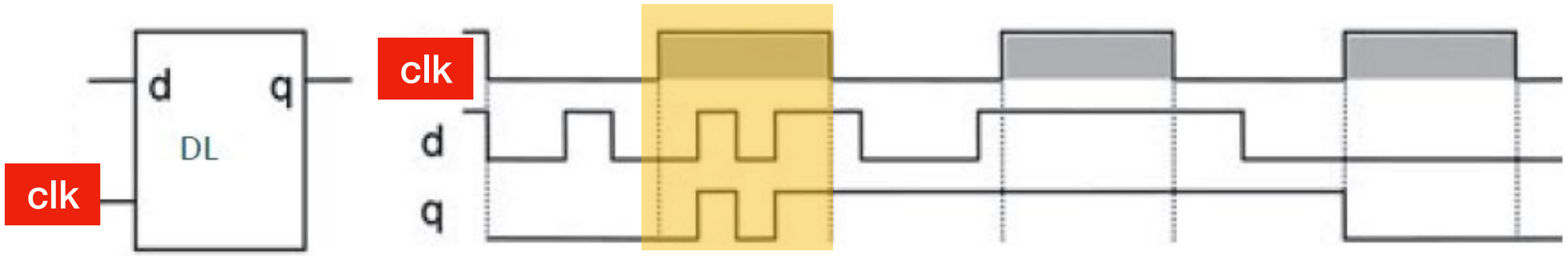
# BEHAVIORAL DESCRIPTION EXAMPLE: D-LATCH




```
module D_latch (d,e,q,q_bar);
input d,e;
output reg q,q_bar;

always @(d,e)
begin
  if (e==1)
  begin
    q=d;
    q_bar = ~q;
  end
end
endmodule
```



# BEHAVIORAL DESCRIPTION EXAMPLE: D-LATCH




```
module D_latch (d,e,q,q_bar);
input d,e;
output reg q,q_bar;

always @(d,e)
begin
  if (e==1)
  begin
    q=d;
    q_bar = ~q;
  end
end
endmodule
```

- Ports are defined by default as *wire*
- reg type before signal names are necessary

otherwise COMPILATION ERROR

# SEQUENTIAL STATEMENTS

- Sequential statements includes:
  - if
    - D-Latch example
  - case
  - wait
  - loop

# CASE STATEMENTS

- **Case** statement:
  - The case statement selects for execution one of several alternative sequences of statements the alternative is chosen based on the value of associated expression.
  - Must cover all possible options
- Simple but rigid!
- It uses the verify operator === to do bitwise comparison
- Everything must be explicitly coded

```
case (control-expression)
 value : begin statement1; end
 value : begin statement2; end
 default:begin default statement; end
 endcase
```

```
 case (sel)
  2'b00 : begin y = I1; z=I1&I2; end
  2'b01 : y = I2;
  2'b10 : y = I3;
  default: y = I4;
 endcase
```

Single statement does not need begin/end

# CASE, CASEX, CASEZ STATEMENTS

- **Which statements will be executed if a == 1'b0?**

```
case ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

```
casez ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

```
casex ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

- The **casez** statement treats high-impedance (z) values as don't-care values.
- The **casex** statement treats high-impedance (z) and unknown (x) values as don't care values.
- If any of the bits in the case expression or case item expression is a don't-care value then that bit position will be ignored.

# CASE, CASEX, CASEZ STATEMENTS

- **Which statements will be executed if a == 1'b1?**

```
case ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

```
casez ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

```
casex ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

- The **casez** statement treats high-impedance (z) values as don't-care values.
- The **casex** statement treats high-impedance (z) and unknown (x) values as don't care values.
- If any of the bits in the case expression or case item expression is a don't-care value then that bit position will be ignored.

# CASE, CASEX, CASEZ STATEMENTS

- **Which statements will be executed if a == 1'bz?**

```
case ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

```
casez ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

```
casex ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

- The **casez** statement treats high-impedance (z) values as don't-care values.
- The **casex** statement treats high-impedance (z) and unknown (x) values as don't care values.
- If any of the bits in the case expression or case item expression is a don't-care value then that bit position will be ignored.

# CASE, CASEX, CASEZ STATEMENTS

- **Which statements will be executed if a == 1'bx?**

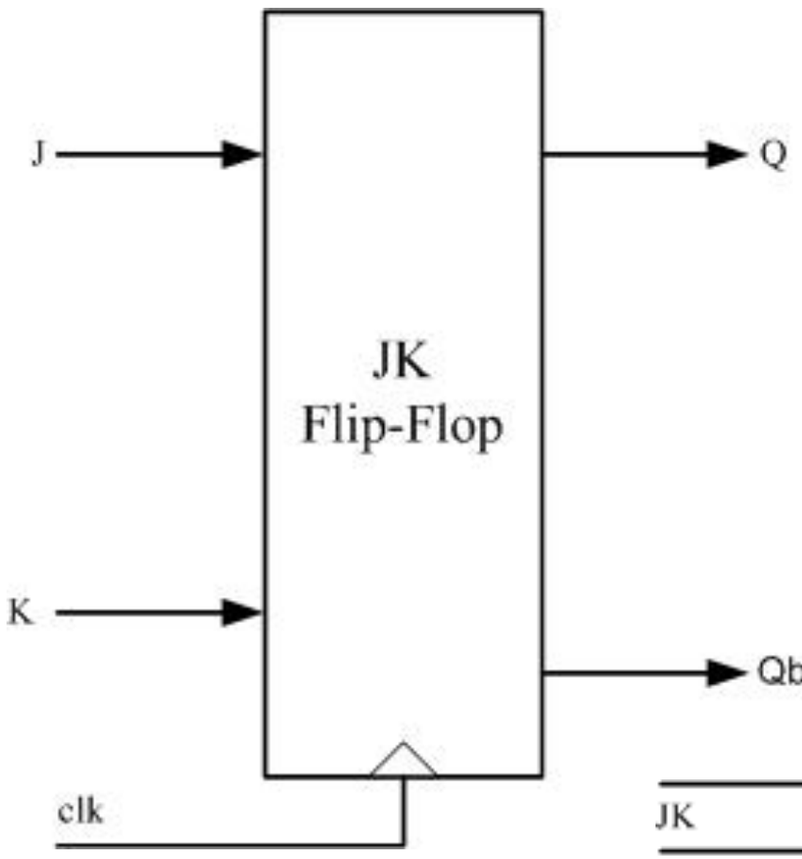
```
case ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

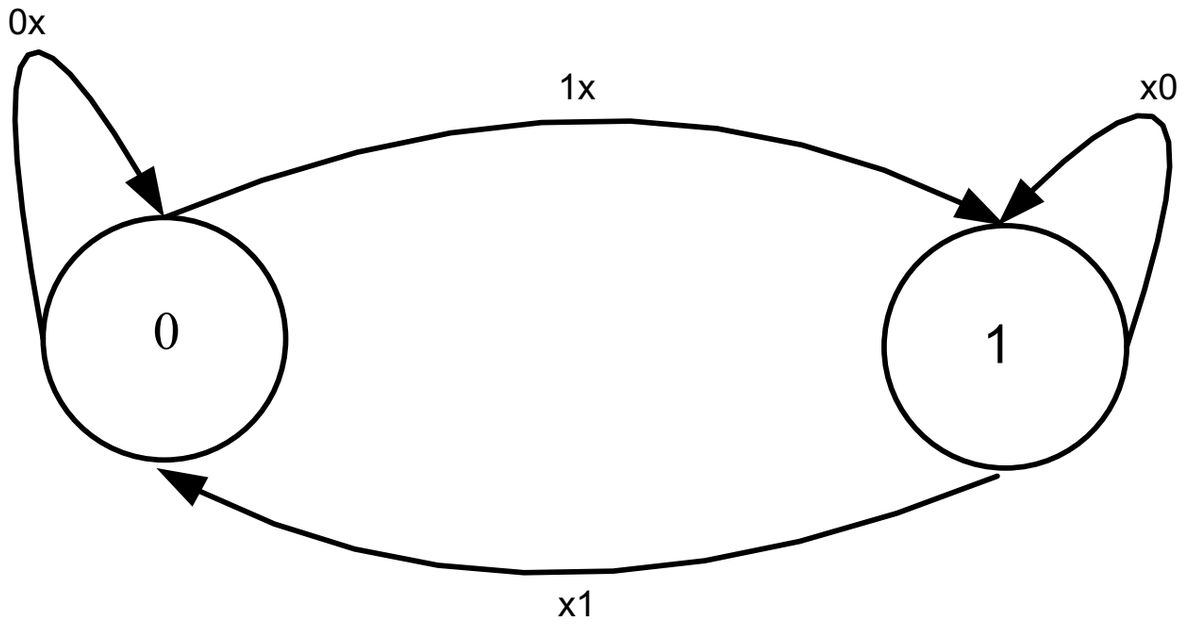
```
casez ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

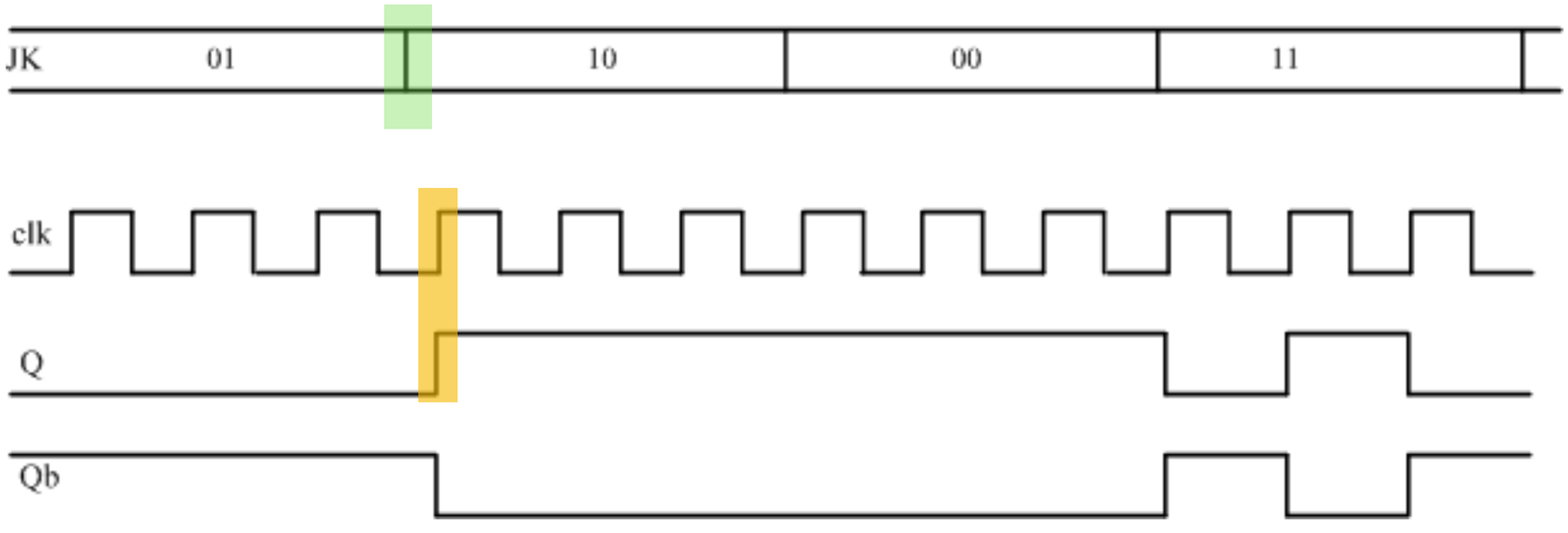
```
casex ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

- The **casez** statement treats high-impedance (z) values as don't-care values.
- The **casex** statement treats high-impedance (z) and unknown (x) values as don't care values.
- If any of the bits in the case expression or case item expression is a don't-care value then that bit position will be ignored.

# EXAMPLE ON CASE: POSITIVE EDGE TRIGGERED J-K FF



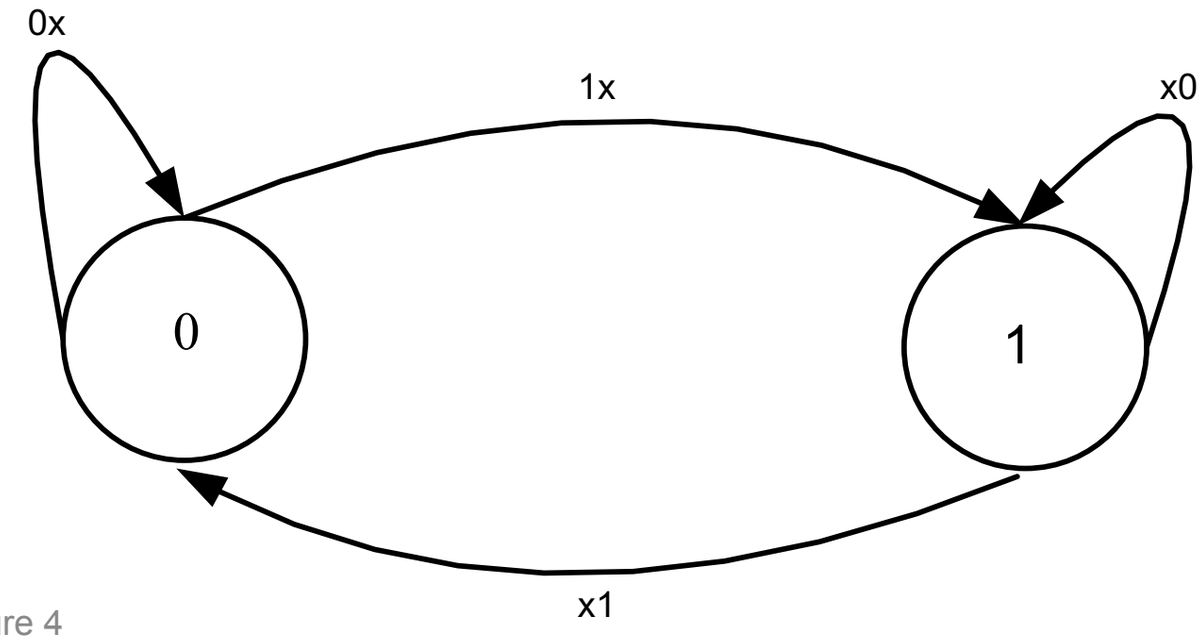



# EXAMPLE ON CASE: POSITIVE EDGE TRIGGERED J-K FF

```
module JK_FF (JK,clk,q,q_bar);
input [1:0] JK;
input clk;
output reg q,q_bar;

always @(posedge clk)
begin
  case (JK)
     2'b00 : q=q;
     2'b01 : q=1b0;
     2'b10 : q=1'b1;
     2'b11 : q=~q;
  endcsae
q_bar = ~q;
end
endmodule
```

posedge is a keyword meaning at the rising edge of the clock

Remember the parenthesis around the case condition



# SEQUENTIAL STATEMENTS

- Sequential statements includes:
  - if
    - D-Latch example
  - case
    - JK-FF
  - wait
  - loop

# Break 5 minutes

# SEQUENTIAL STATEMENTS

- Sequential statements includes:
  - if
    - D-Latch example
  - case
    - JK-FF
  - wait
  - loop

# WAIT STATEMENT

- Wait for a time period
- Can be used to generate clocks with different time periods

```
wait (condition) #(<optional_delay>) statement
```

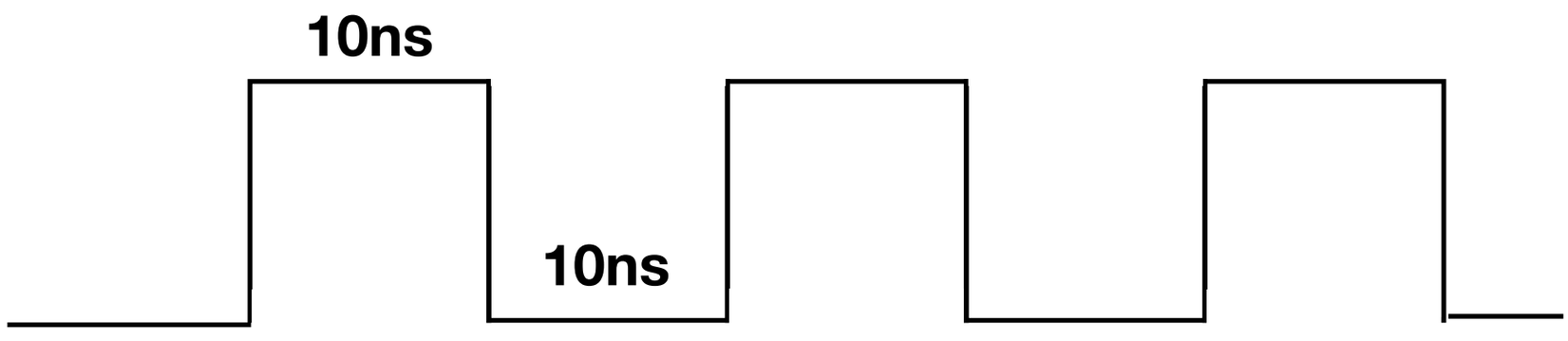
**wait (i>10&&j<5) #10 a=b;**

```
#delay
```

**# 10;**

# EXAMPLE ON WAIT: CLOCK GENERATION

```
module wait_statement (a,b,c)
output reg a,b,c;

initial
    begin
        a=0;
    end
//initial is done
always
    begin
    #10 ;
            a = ~ a;
    end

endmodule
```




**Time period = 20 ns**

# SEQUENTIAL STATEMENTS

- Sequential statements includes:
  - if
    - D-Latch example
  - case
    - JK-FF
  - wait
    - Clock generation
  - loop
    - for
    - while
    - repeat
    - forever

# LOOP STATEMENTS

- For loop

```
for(counter_low;counter_up;counter_update)
  begin
     sequential_statements;
  end
```

```
for (i=0;i<=2;i=i+1)
begin
 if (temp[i] == 1'b1)
   begin
      result = result + 2**i;
   end
end
statement1;
statement2;
```

Don't forget to declare the index i

What does this loop do?

# LOOP STATEMENTS

- While loop

```
while(condition)
  begin
     sequential_statements;
  end
```

```
while (i>1)
  begin
     i= i/2;
     i= j+1;
  end
```

Don’t forget to declare the index i

# LOOP STATEMENTS

- **Repeat** loop
  - No condition is allowed

```
repeat(fixed_number)
  begin
     sequential_statements;
  end
```

- **Forever** loop
  - No condition
  - Loop endlessly
  - Use for clock generation

```
initial
  begin
     clk=1'b0;
  forever #20clk=~clk;
  end
```

# LOOP SUMMARY

Loop statements provide a means of modeling blocks of behavioral statements.

- **forever**

  continuously repeats the statement that follows it. Therefore, it should be used with timing controls (otherwise it hangs the simulation).

- **repeat**

  executes a given statement a fixed number of times. The number of executions is set by the expression, which follows the repeat keyword. If the expression evaluates to unknown, high-impedance, or a zero value, then no statement will be executed.

- **while**

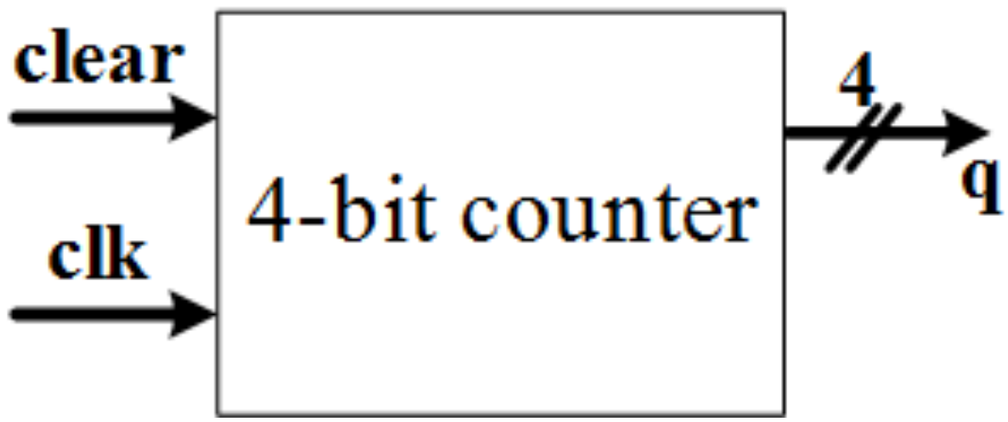
  executes a given statement until the expression is true. If a while statement starts with a false  value, then no statement will be executed.

- **for**

  executes a given statement until the expression is true. At the initial step, the first assignment will be executed. At the second step, the expression will be evaluated. If the expression evaluates to an unknown, high-impedance, or zero value, then the for statement will be terminated. Otherwise, the statement and second assignment will be executed. After that, the second step is repeated.

# EXAMPLE: 4-BIT COUNTER WITH SYNCHRONOUS CLEAR

- Implement a counter with 4 bits
  - At rising edge of clk:
    - If clear is 1; reset the counter
    - Otherwise the counter starts from zero and counts at each rising clk edge



# EXAMPLE: 4-BIT COUNTER WITH SYNCHRONOUS CLEAR

```
module count_4(clr,clk,q);
input clk, clr;
output reg [3:0] q;
integer i,j, result;
initial
begin
  q=4'b0000; //initialize count
end
always @ (posedge clk)
begin
  if (clr==0)
  begin
      result=0;
      // change binary to integer
      for (i=0;i<4;i=i+1)
      begin
       if (q[i]==1)
          result=result +2**i;
      end
      result =result+1;
      for (j=0; j<4; j=j+1)
      begin
        if (result % 2==1)
          q[j]= 1;
        else
          q[j]= 0;
        result=result/2;
      end
    end
  else
     q=4'b0000;
end
endmodule
```

# MORE EXAMPLES IN THE BOOK

- 4-bit counter with synchronous hold.
- Shift registers
- Calculating factorial

# BEHAVIORAL DESCRIPTION

- Basic statements of behavioral description
  - always/initial blocks
- Sequential statements
  - If
    - D-latch FF
  - Case
    - JK FF
  - Wait
    - clock generation
  - loop (for, while, repeat, forever)
    - 4-bit synchronous clear counter

- Blocking vs. non-blocking assignments

# BLOCKING VS. NON-BLOCKING ASSIGNMENT

**Inside sequential block (like always or initial)**

- **blocking**
  - Use **=** operator
  - The result of this assignment can be seen in the subsequent statements
  - Statements are executed sequentially instead of concurrently
- **non-blocking**
  - Use **<=** operator
  - The result of this assignment cannot be seen in subsequent statements
  - Statements are executed concurrently instead of sequentially
  - The order of non-blocking assignments is not important as long as the destinations of the assignments are different signals

# EXAMPLE; BLOCKING VS. NON-BLOCKING ASSIGNMENT

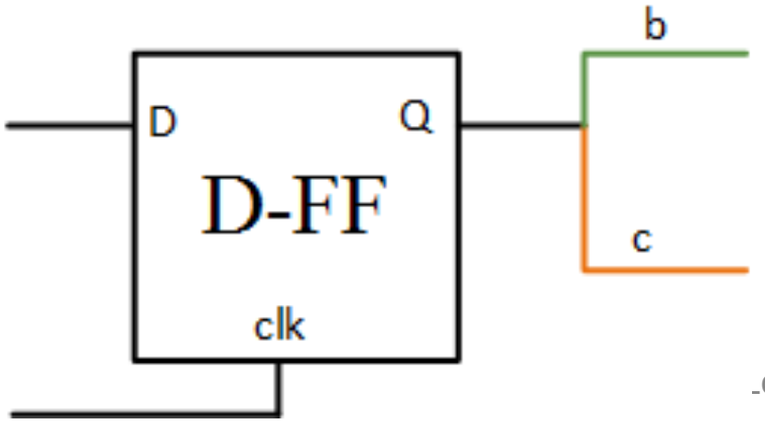
```
module block_nonblock();
reg a, b, c, d , e, f ;

 // Blocking assignments
initial begin
 #10 a = 1'b1;
 #20 b = 1'b0;
 #40 c = 1'b1;
end

/* The simulator assigns
1 to a at time 10, 0 to
b at 30, and 1 to c at
70
*/
```

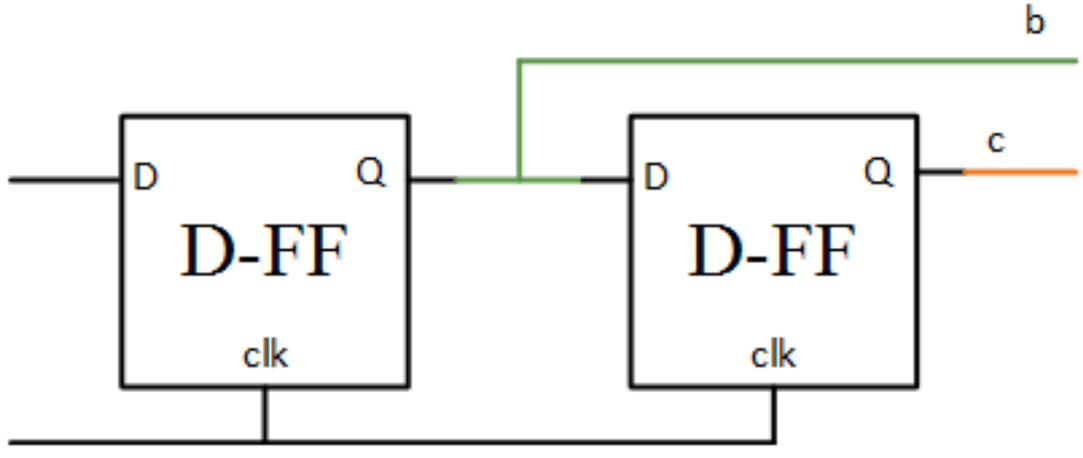
```
// Non-blocking
assignments
initial begin
 #10  d <=  1'b1;
 #20  e <=  1'b0;
 #40  f <=  1'b1;
end
endmodule

/* The simulator
assigns 1 to d at time
10, 0 to e at 20, and 1
to f at 40 */
```

# EXAMPLE; BLOCKING VS. NON-BLOCKING ASSIGNMENT

```
module blocking (clk,a,c);
  input clk;
  input a;
  output c;
  wire clk,a;
  reg c,b;
always @ (posedge clk )
 begin
   b = a;
   c = b;
 end
endmodule
```

```
module non-blocking (clk,a,c);
  input clk;
  input a;
  output c;
  wire clk,a;
  reg c,b;
always @ (posedge clk )
 begin
   b <= a;
   c <= b;
 end
endmodule
```



# LECTURE 4: SUMMARY

- Behavioral Description
- Sequential statements
  - if, case, for, while, …
  - Storage elements
  - DFF
  - JK FF
  - Binary counters
- Blocking vs. non-blocking assignment

# QUESTIONS??