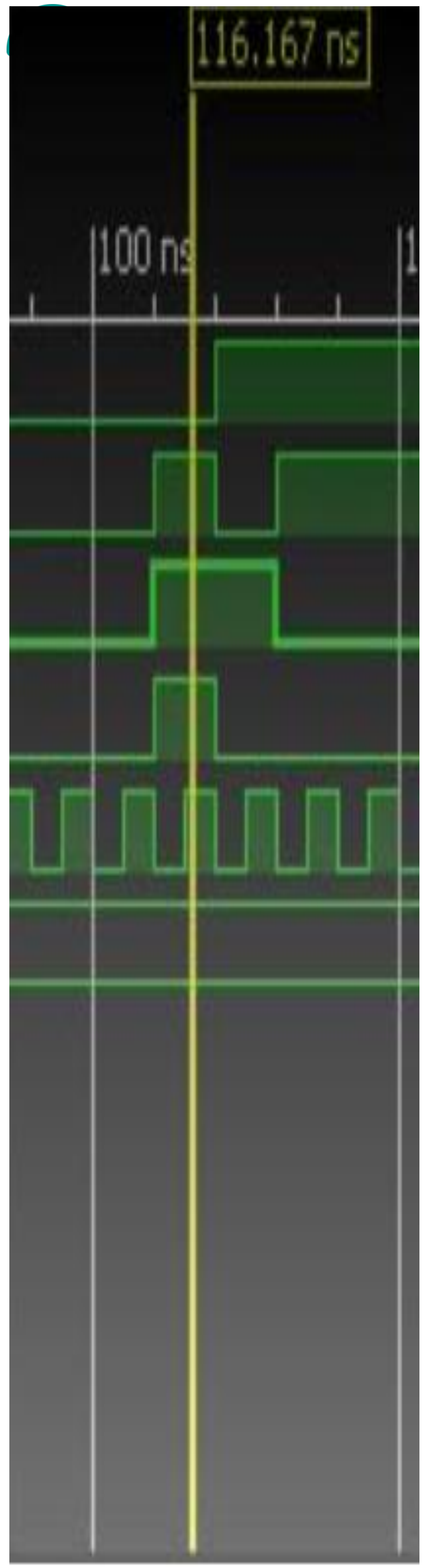
# Verilog III

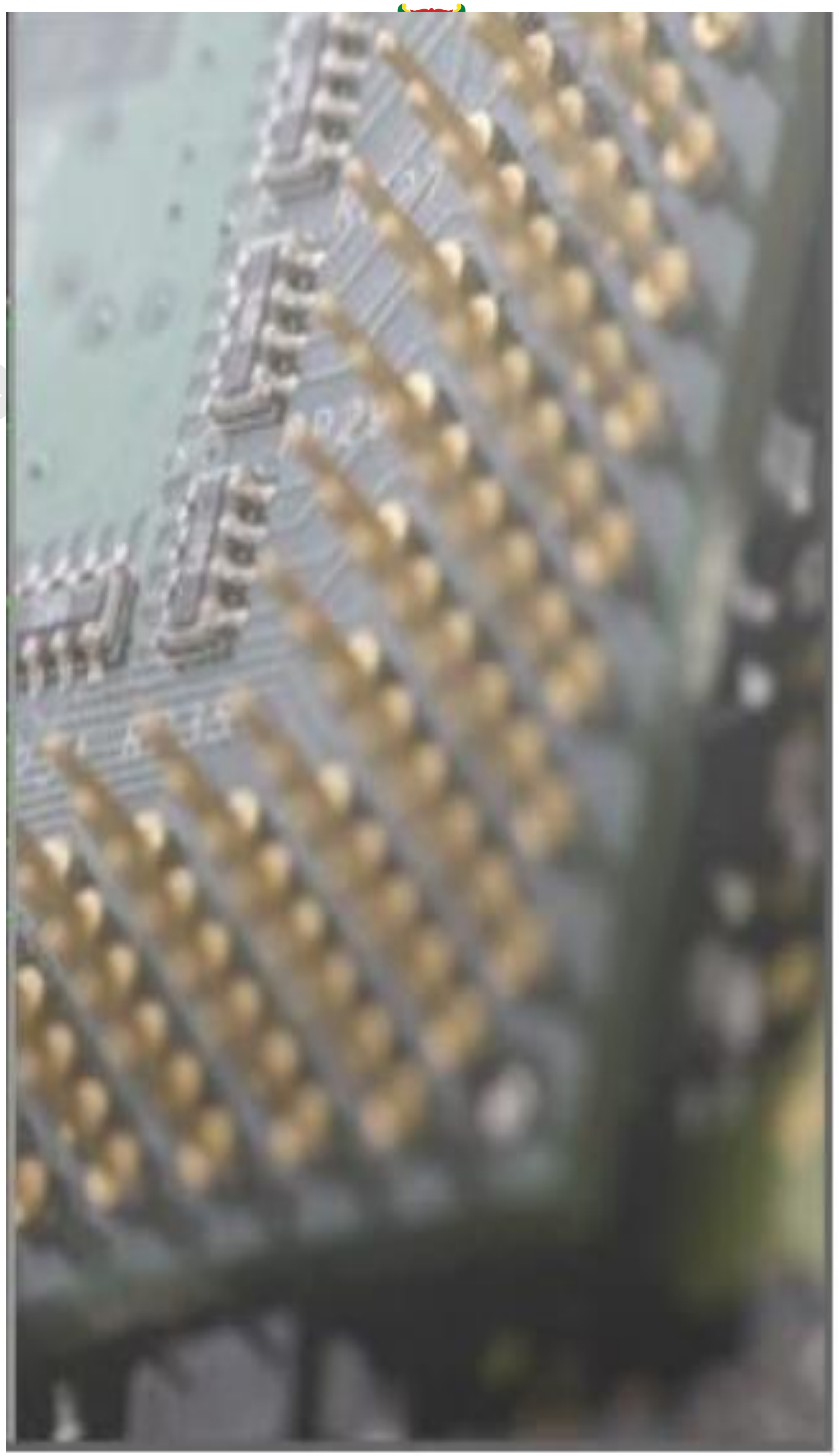
MC. Martin González Pérez

# Table of contents

- Verilog processes and events
- Blocking and Non-blocking assignments
- Simulation cycle
- Exercises
- Modeling synthesizable logic

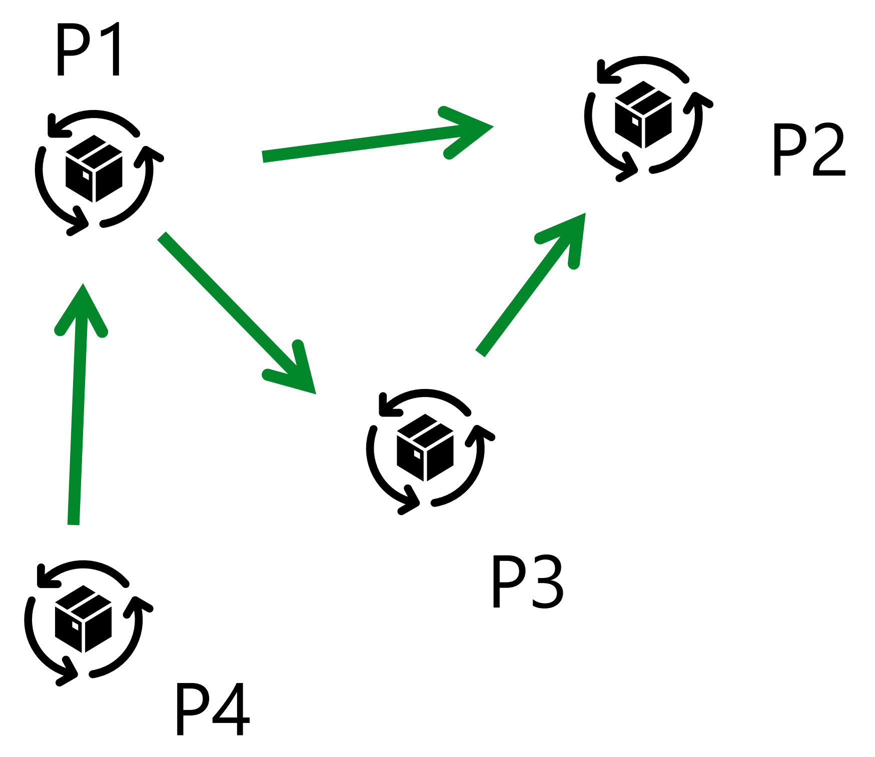


# Verilog processes and events



# Processes and events

Circuit simulations are based on processes and events. Processes react to input events and, in turn, generate output events. Some examples of processes in Verilog are **continuous assignments** and the procedural blocks as **initial** and **always**.



# Continuous assignment

**Continuous assignment**: Continuous assignment is its own process; the simulator automatically update the driven value when any of the input changes.

**Example**

```
assign sum = a + b;
```

**a** and **b** transitions are the input events and **sum** is the output event.

## Initial

**Initial**: Initial blocks are executed at the start of the simulation (time=0), and once their execution is complete, the process terminates and is not called again.

**Example**

```
initial begin
    a = 0;
    b = 0;
    #10;
    a = 2;
    b = 1;
end
```

$\downarrow$ Execute once

**initial block is not synthesizable !!**

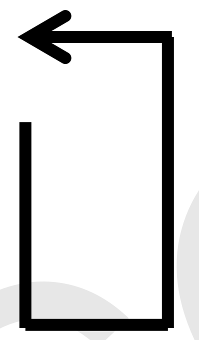
Initial blocks doesn’t have input events they are executed at start of sequentially, statements (processes) inside are executed sequentially, and left-hand operator can be seen as output events.

## Always

**Always**: Always construct is executed at the start of the simulation, and once the execution is finished, it executes again. Always blocks use a control event to block its execution and avoid simulation hanging. A control event starts with @(event expression), the event expression is also called sensitive list.

**Example**

```
always @(a or b)
begin
    eq = a==b;
end
```



Transitions of signals **a** and **b** are the events that control the always block execution. Statements inside are executed sequentially and left-hand operator can be seen as output events.

**Sensitive list examples**

@(a or b)
@(a)
@(*)
@(posedge clk)
@(posedge clk or negedge arstn)

# Interaction between processes

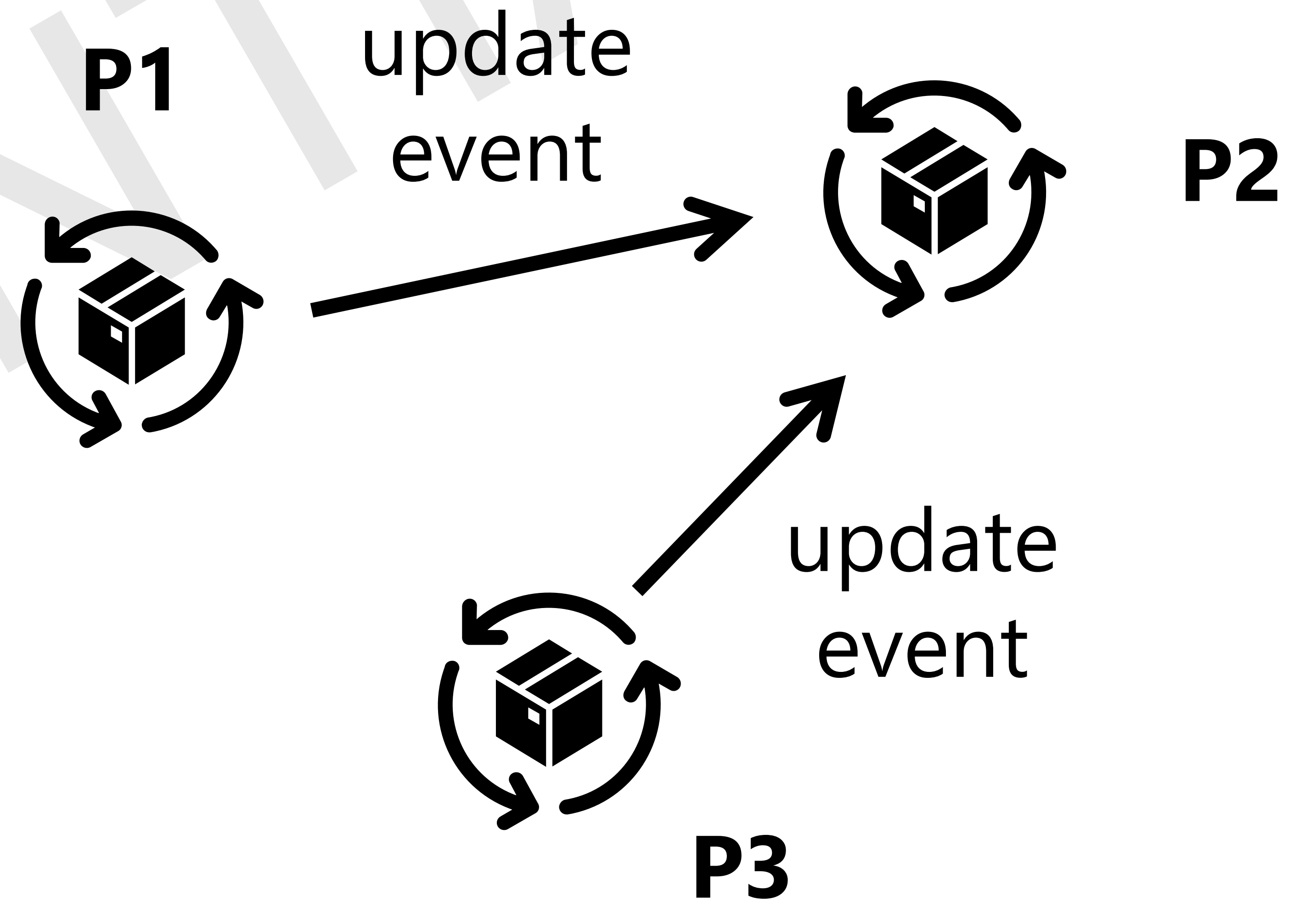
Designs usually contains several modules, and a module typically contains many procedural blocks.

- Procedural block executes its statements sequentially like a conventional programming language.
- Every procedural block is executed concurrently, like hardware.
- Blocks communicate each others using events, nets and variables.

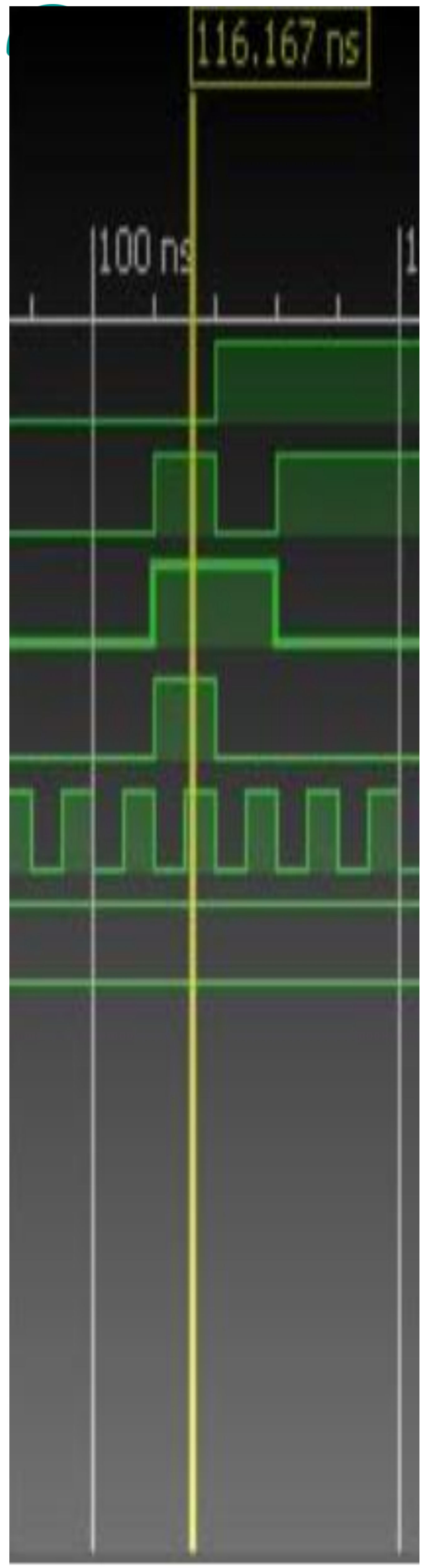
These features make it possible to model hardware.

# Interaction between processes

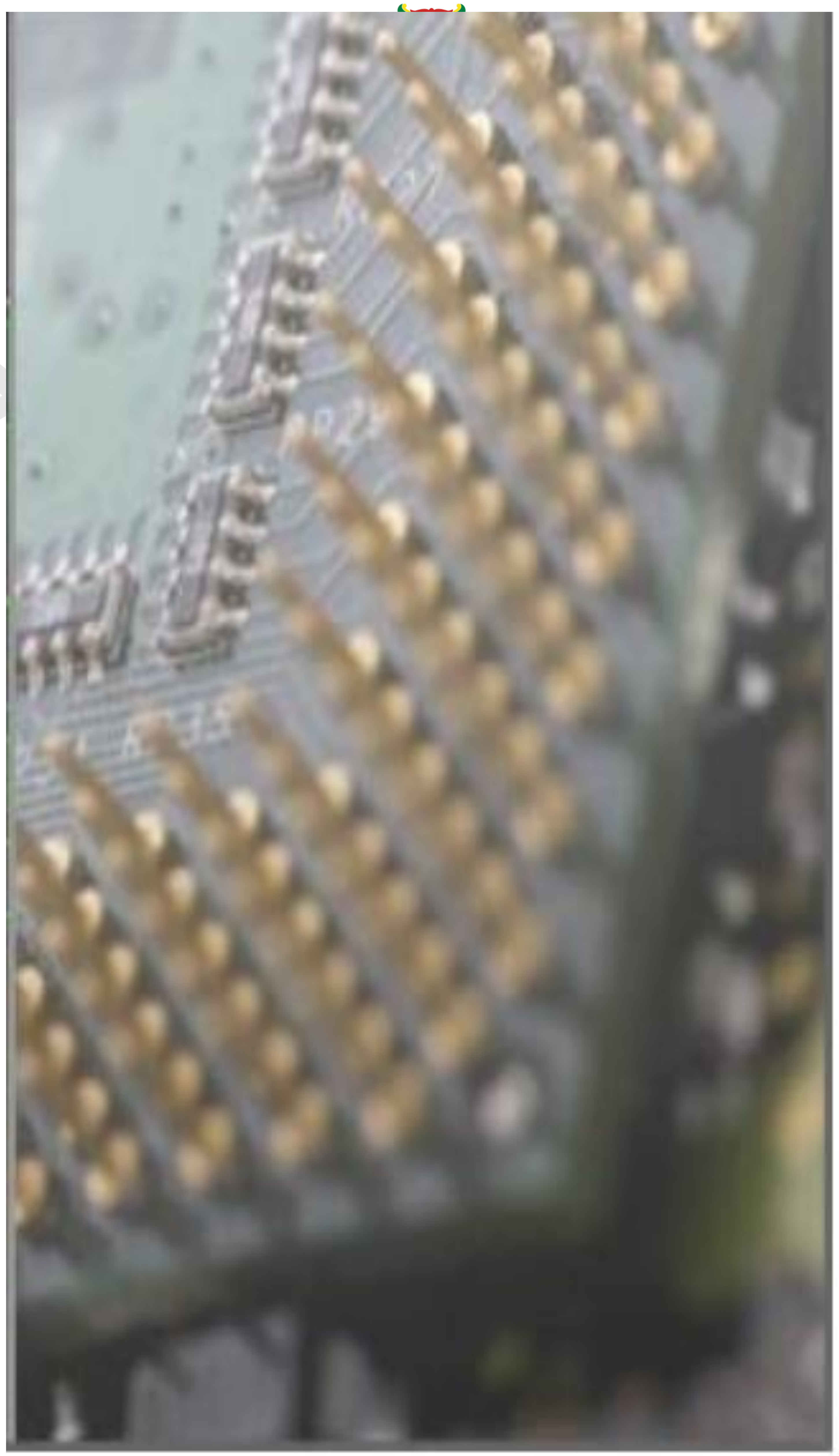
```
always@(a or b) begin: P1
    x = a ^ b;
end

always@(x or y) begin: P2
    r = x | y;
end

always@(c or d) begin: P3
    y = c & d;
end
```



Even if P1 is above in the code, this doesn’t mean that P1 will be executed first by the simulator, Verilog ensures that procedural blocks are evaluated concurrently.

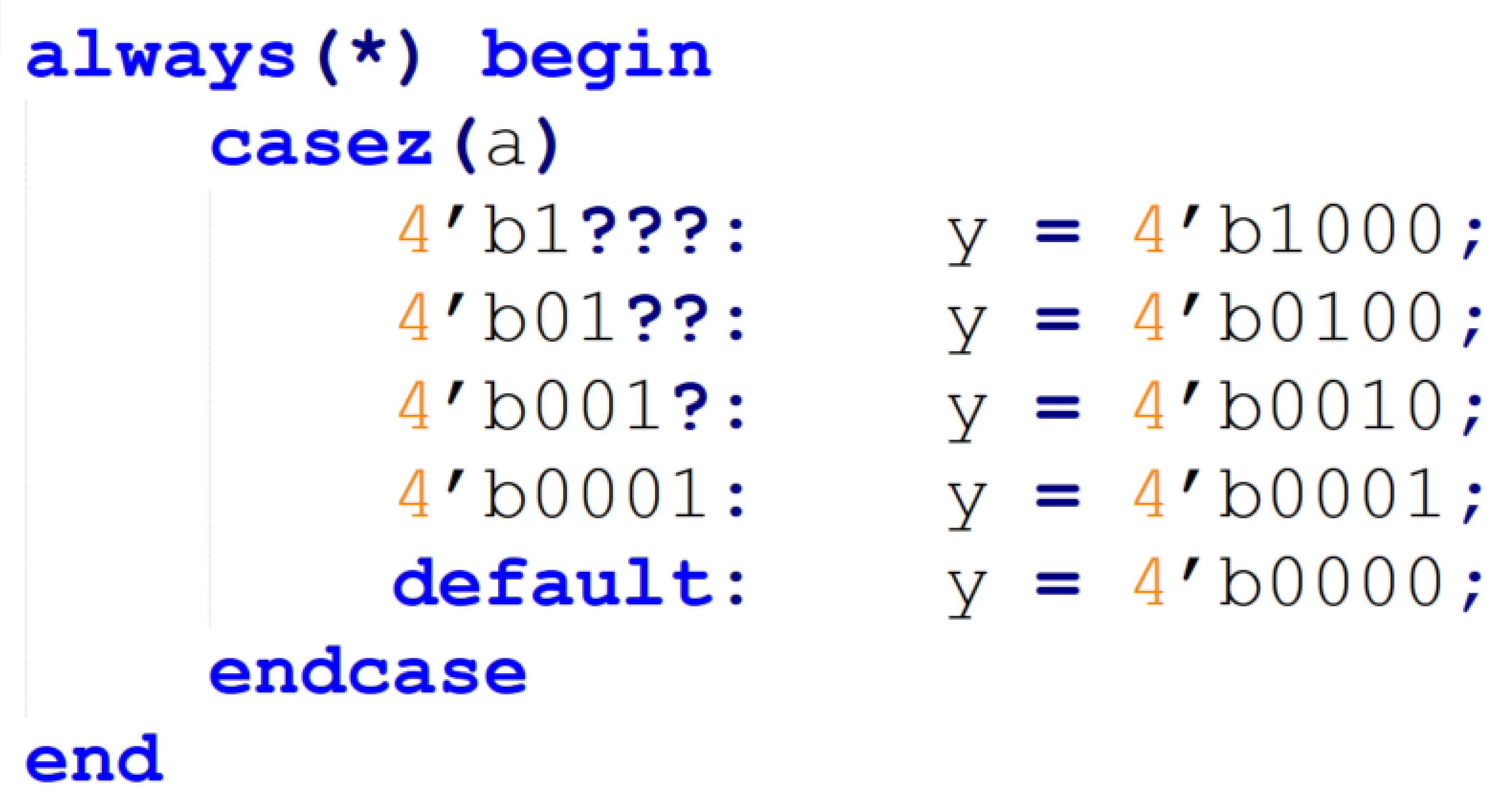


## Blocking and nonblocking assignments



## Blocking assignment

**Blocking assignments** use the “=“ operator. They are called "blocking" because **the execution of the following statement is blocked until the current assignment is completed**. This means that operations are performed in a **sequential order**, and each line of code "waits" for the previous line to complete before continuing.



```
always(*) begin
    casez(a)
        4’b1???:    y = 4’b1000;
        4’b01??:    y = 4’b0100;
        4’b001?:    y = 4’b0010;
        4’b0001:    y = 4’b0001;
        default:    y = 4’b0000;
    endcase
end
```

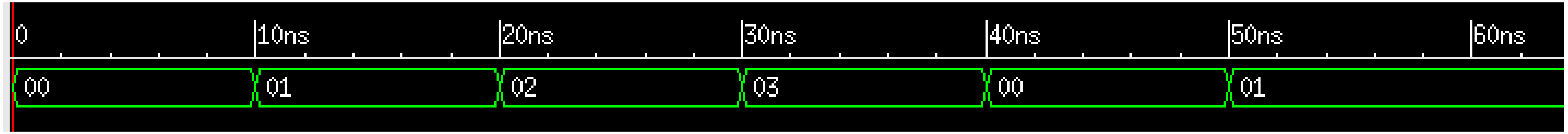
## Blocking assignment

```
Initial
begin
    cnt = 8'h00;
    for (int i=0; i<=4; i=i+1)
        begin
            #10;
            cnt = cnt + 1;
            if (cnt == 4)
                cnt = 0;
        end
end
```

What would be the maximum value of **cnt**?

## Blocking assignment

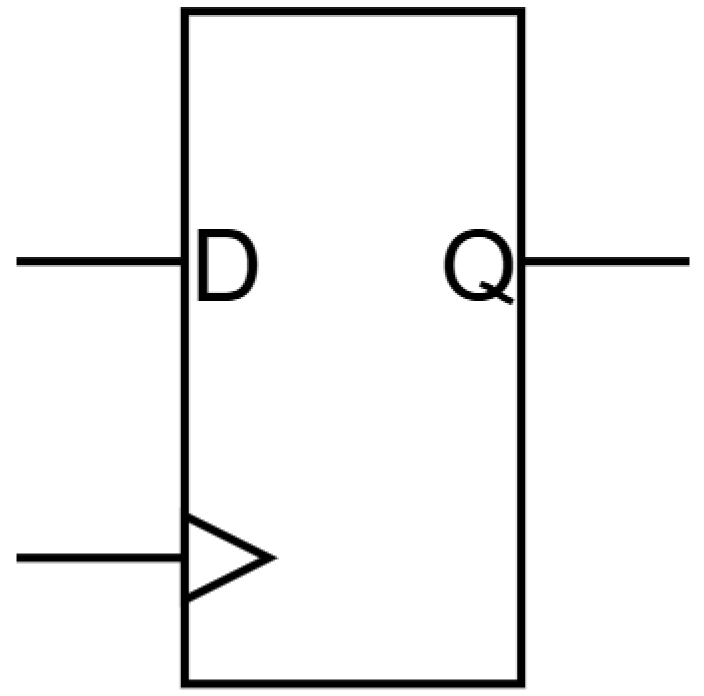
```
Initial
begin
    cnt = 8'h00;
    for (int i=0; i<=4; i=i+1)
        begin
            #10;
            cnt = cnt + 1;
            if (cnt == 4)
                cnt = 0;
        end
end
```



## Nonblocking assignment

**Nonblocking assignments** use the **<=** operator. They are called "nonblocking" because **the left hand operator is not updated immediately and don’t block the execution of the next statement.** The simulator compute the value of the left hand operator and schedules the variable update for a point in the simulation where all currently active blocks have executed up to the point where they are **all blocked**. This ensures that any other active block that reads the variable **reads the old value and not the new value.**

```
always @(posedge clk)
     q<=d;
```



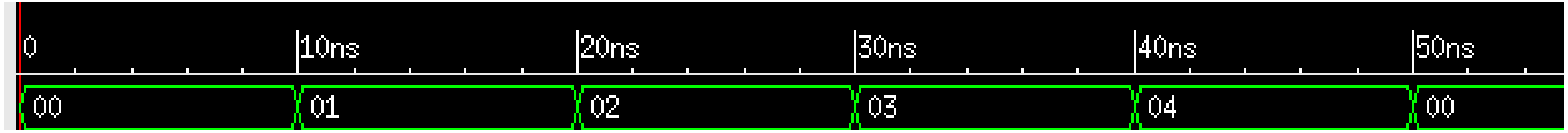
# Nonblocking assignment

```
Initial
begin
    cnt = 8'h00;
    for (int i=0; i<=4; i=i+1)
        begin
            #10;
            cnt <= cnt + 1;
            if (cnt == 4)
                cnt <= 0;
        end
end
```

What would be the maximum value of **cnt**?

## Nonblocking assignment

```
Initial
begin
    cnt = 8’h00;
    for (int i=0; i<=4; i=i+1)
        begin
            #10;
            cnt <= cnt + 1;
            if (cnt == 4)
                cnt <= 0;
        end
end
```



## Guidelines

1-Use **always @(posedge clk)** and nonblocking assignments (<=) to model synchronous sequential logic.

```
always @(posedge clk)
    q<=d;
```

2-Use continuous assignments to model simple combinational logic.

```
assign y = s ? b : a;
```

## Guidelines

3-Use **always(*)** and blocking assignments to model complex combinational logic where statements are helpful.

```
always(*) begin
    casez(a)
        4'b1???:    y = 4'b1000;
        4'b01??:    y = 4'b0100;
        4'b001?:    y = 4'b0010;
        4'b0001:    y = 4'b0001;
        default:    y = 4'b0000;
    endcase
end
```

4-Do not make assignments to the same signal in more that one always statement or continuous assignment (avoid multiple drivers).

## Guidelines

Not following the previous guidelines can cause a code that appears to work in simulation, but synthesized circuit can be incorrect.

The following slides are to explain the rationale behind the previous guidelines:

# Understanding blocking and Nonblocking

**Code 1**

Using blocking assignment, statement 1 is evaluated, and **m** is updated immediately. Then, statement 2 is evaluated, and the value of **n** is updated using the newly updated value of **m**.

m = 0 =0^0

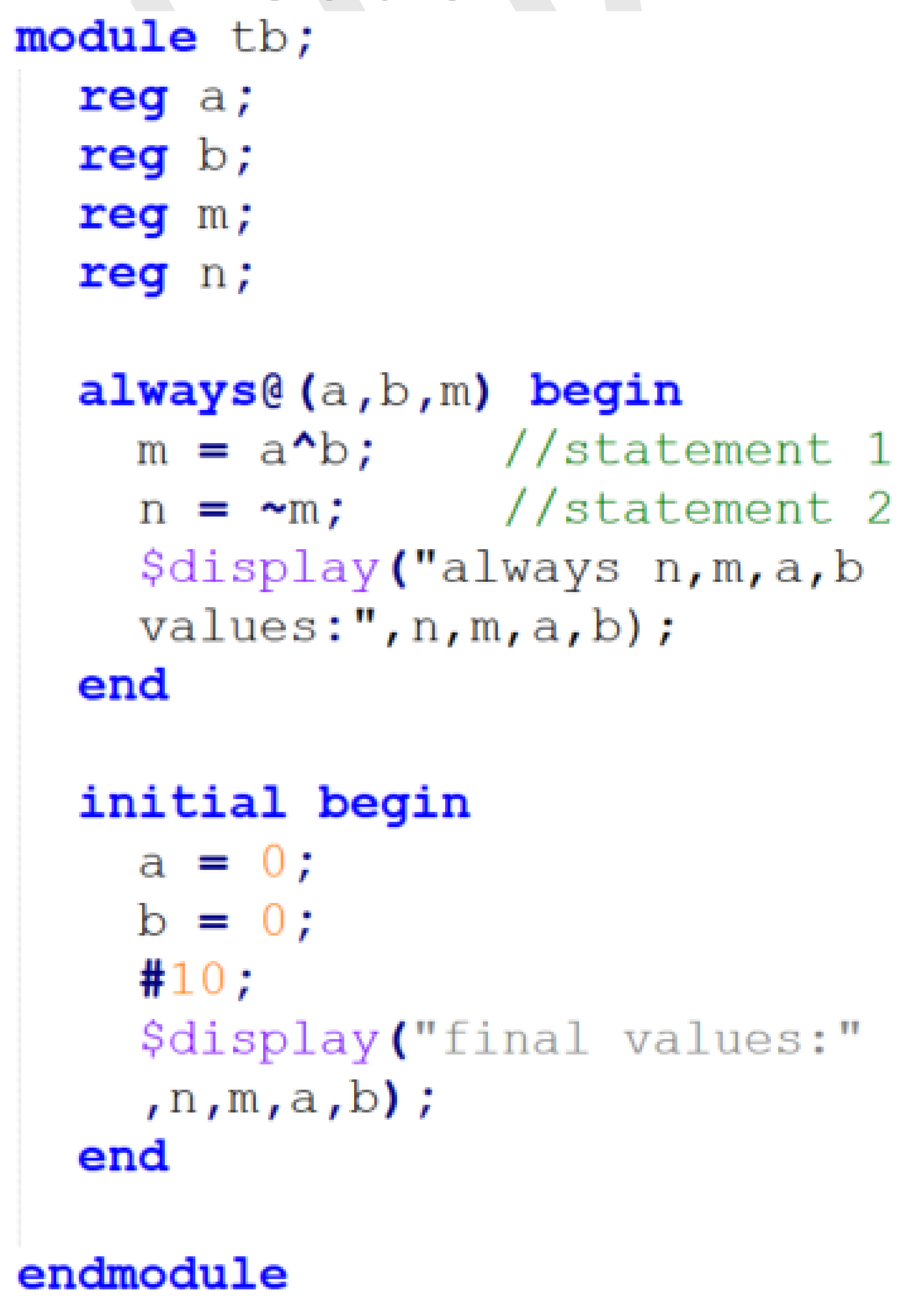
n = 1 = ~0

**Code 2**

Using non-blocking assignment, statements 1 and 2 are evaluated but **m** and **n** are not updated immediately. At the end of the always block execution, both **m** and **n** are updated. When **m** changes, the always block is executed a second time, updating the value of **n**. Since n is not in the sensitivity list, the always block is not executed a third time.

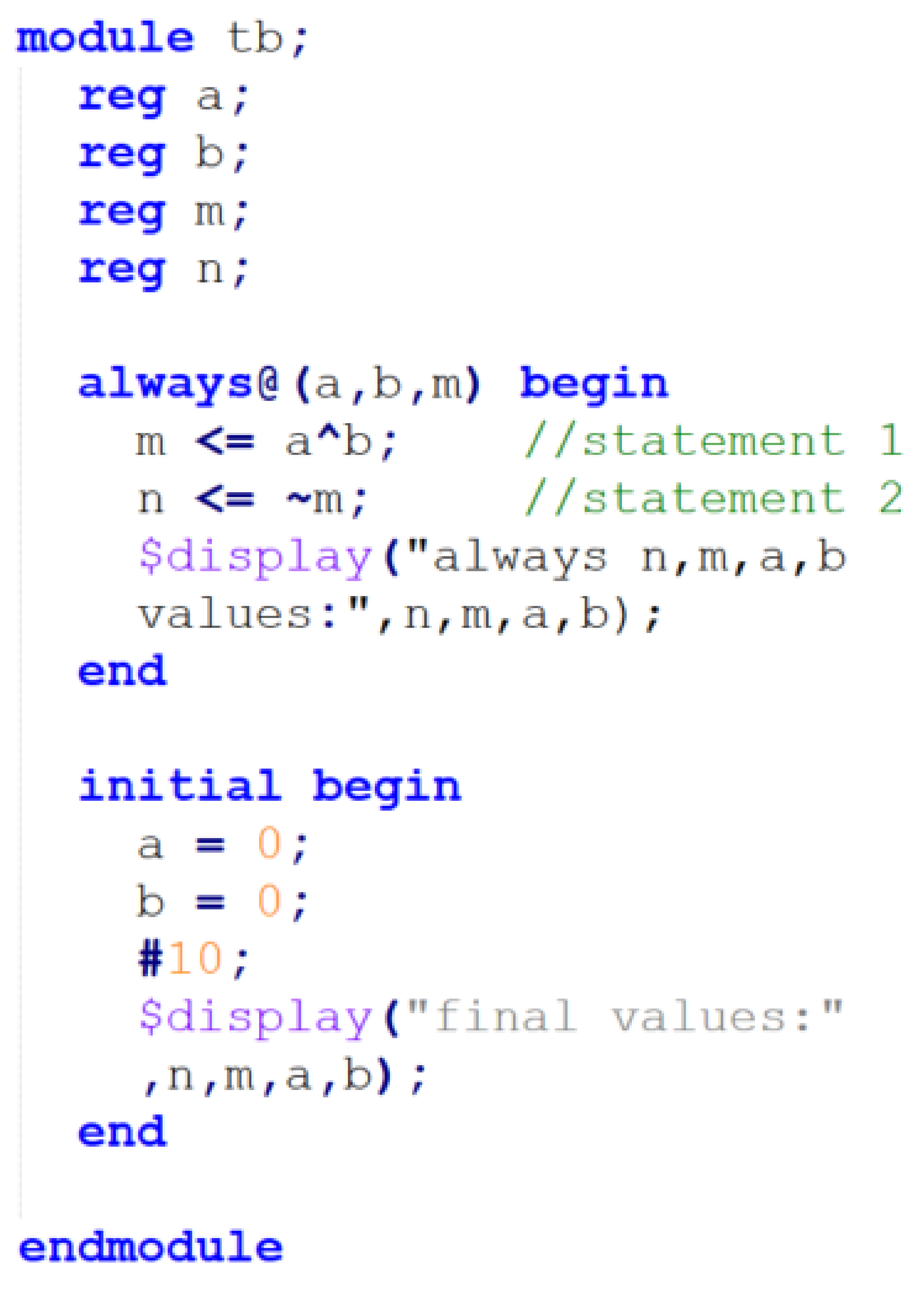
```
m = x            n = x          //initial values
m = 0 = 0^0      n = x = ~x     //first execution
m = 0 = 0^0      n = 1 = ~0     //second execution
```

**Code 1**



```
module tb;
  reg a;
  reg b;
  reg m;
  reg n;

  always@(a,b,m) begin
    m = a^b;     //statement 1
    n = ~m;      //statement 2
    $display("always n,m,a,b
    values:",n,m,a,b);
  end

  initial begin
    a = 0;
    b = 0;
    #10;
    $display("final values:"
    ,n,m,a,b);
  end

endmodule
```

**Code 2**



```
module tb;
  reg a;
  reg b;
  reg m;
  reg n;

  always@(a,b,m) begin
    m <= a^b;     //statement 1
    n <= ~m;      //statement 2
    $display("always n,m,a,b
    values:",n,m,a,b);
  end

  initial begin
    a = 0;
    b = 0;
    #10;
    $display("final values:"
    ,n,m,a,b);
  end

endmodule
```

## Understanding blocking and Nonblocking

The result of both codes is the same, but when using blocking assignment, the always block is executed only once. In contrast, when using non-blocking assignment, the always block is executed twice.
Execute always block multiple times make the simulation slower.

**Remember, when modeling combinational logic, the correct approach is to use blocking assignments.**

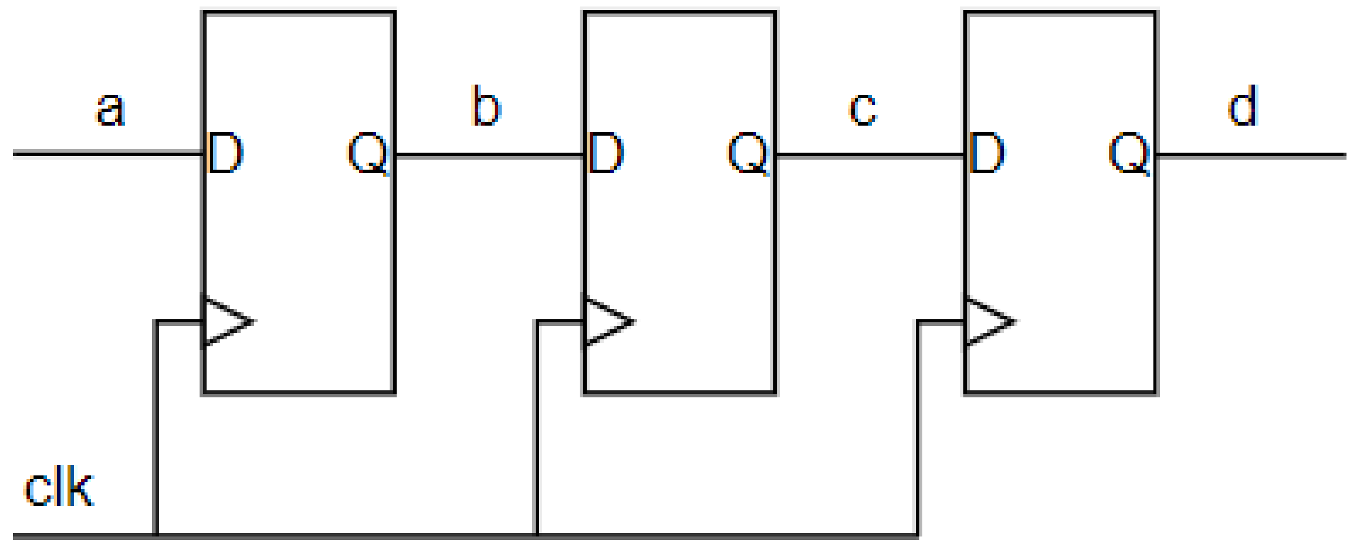
## Assign order "blocking assignment"

**Code 1**

```
always @(posedge clk)
begin
    b=a;
    c=b;
    d=c;
end
```

**Code 2**

```
always @(posedge clk)
begin
    d=c;
    c=b;
    b=a;
end
```



**Which code models the behavior of the shown circuit?**

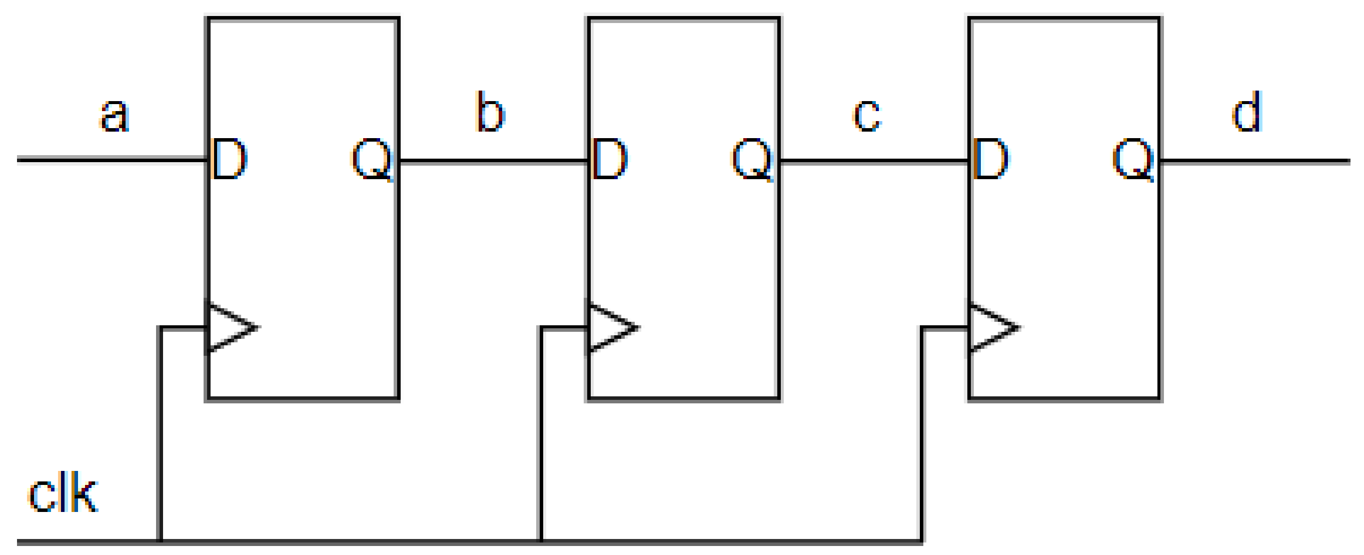
## Assign order “blocking assignment”

**Code 1**

```
always @(posedge clk)
begin
    b=a;
    c=b;
    d=c;
end
```

**Code 2**

```
always @(posedge clk)
begin
    d=c;
    c=b;
    b=a;
end
```



**Which code models the behavior of the shown circuit?**

**Code 2: The order affects functionality**

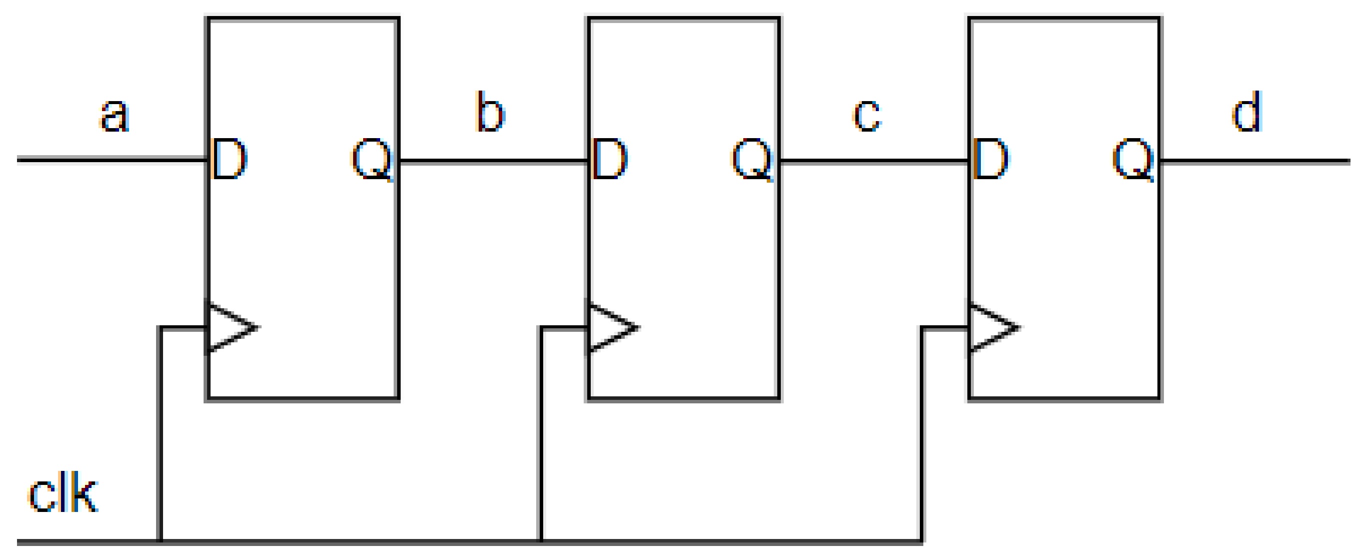
# Assign order "nonblocking assignment"

**Code 1**

```
always @(posedge clk)
begin
      b<=a;
      c<=b;
      d<=c;
end
```

**Code 2**

```
always @(posedge clk)
begin
      d<=c;
      c<=b;
      b<=a;
end
```



**Which code models the behavior of the shown circuit?**

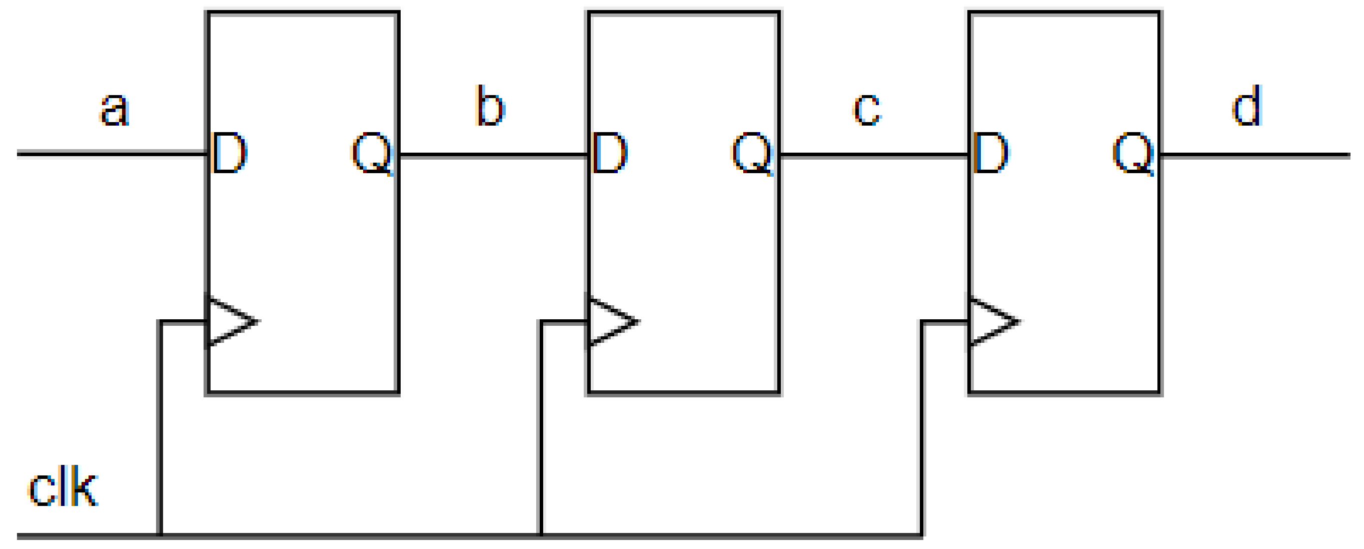
# Assign order "nonblocking assignment"

**Code 1**

```
always @(posedge clk)
begin
    b<=a;
    c<=b;
    d<=c;
end
```

**Code 2**

```
always @(posedge clk)
begin
    d<=c;
    c<=b;
    b<=a;
end
```



**Which code models the behavior of the shown circuit?**

**Both codes: Order does not affect functionality**

# Assignment types

**Code 1**

```
always @(a,b,c)
begin
      m=a+1;
      n=b+2;
      s=m+n;
      m=c+1;
      q=m+n;
end
```

**Code 2**

```
always @(a,b,c)
begin
      m<=a+1;
      n<=b+2;
      s<=m+n;
      m<=c+1;
      q<=m+n;
end
```

What values will **s** and **q** take in terms of **a**, **b** and **c**?

# Assignment types

**Code 1**

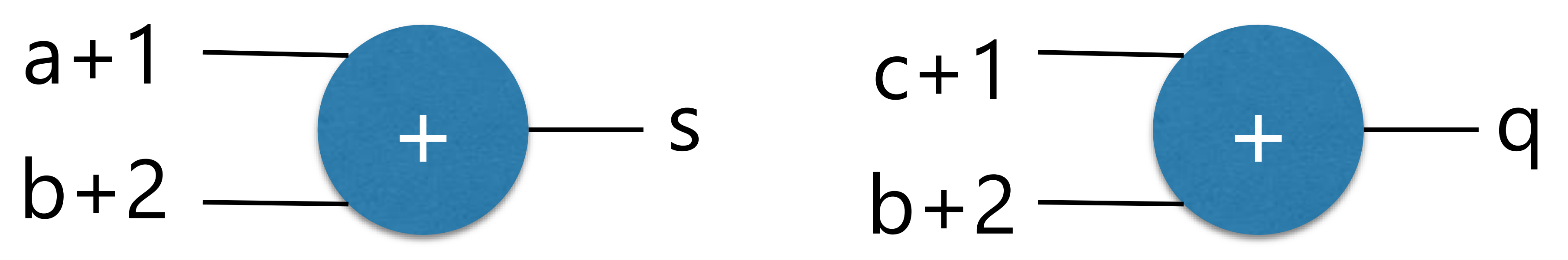
```
always @(a,b,c)
begin
      m=a+1;
      n=b+2;
      s=m+n;
      m=c+1;
      q=m+n;
end
```

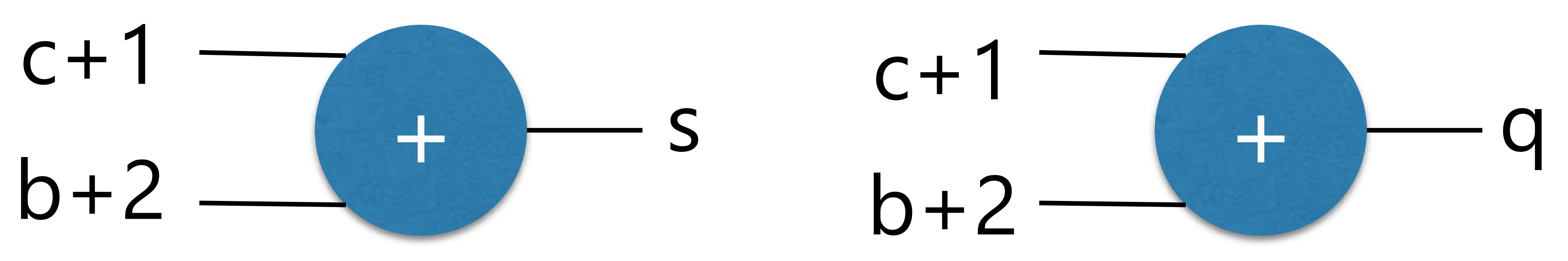
**m** and **n** are temporary variables.

**Code 2**

```
always @(a,b,c)
begin
      m<=a+1;
      n<=b+2;
      s<=m+n;
      m<=c+1;
      q<=m+n;
end
```

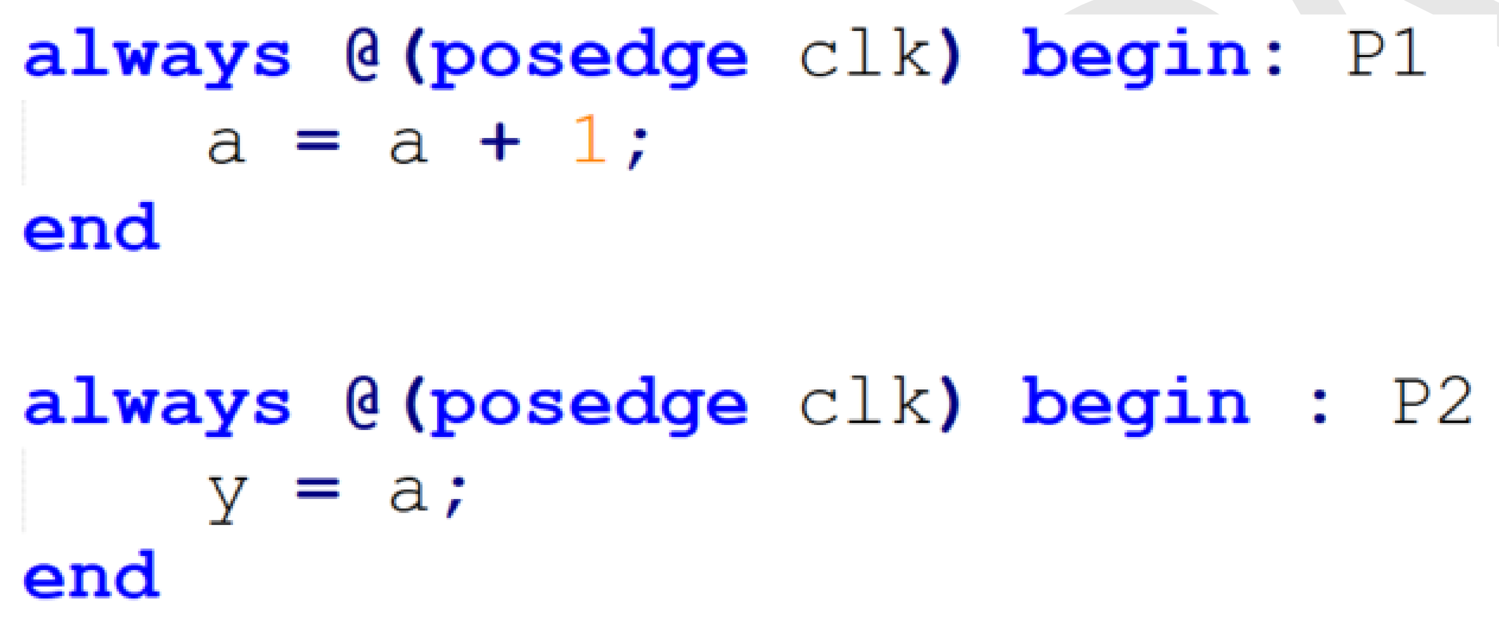
**m** and **n** aren't temporary variables.





## Synchronizing multiple procedural blocks

**Blocking assignments** can lead to **race conditions** when the **same event triggers multiple procedures.** This happens because the procedural blocks are **executed in a non-deterministic order.** Since variables evaluated using blocking assignments are updated immediately, a procedural block may read a value that may or may not have already been updated.



```
always @(posedge clk) begin: P1
    a = a + 1;
end

always @(posedge clk) begin : P2
    y = a;
end
```

If P1 is executed first y+ = a+1 or y = a+

If P2 is executed first y+ = a

**(+) symbol indicate next state.**

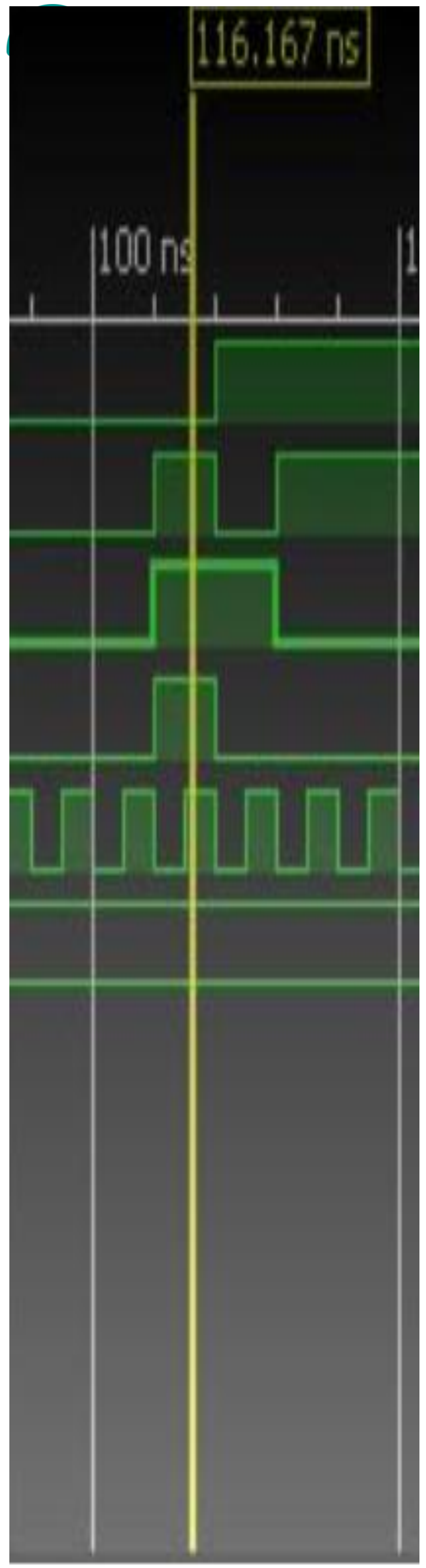
# Synchronizing multiple procedural blocks

In this example, both procedural blocks are triggered on the positive clock edge. Since nonblocking assignments are used, the variables are updated simultaneously using the old values, regardless of the order in which the procedural blocks execute.

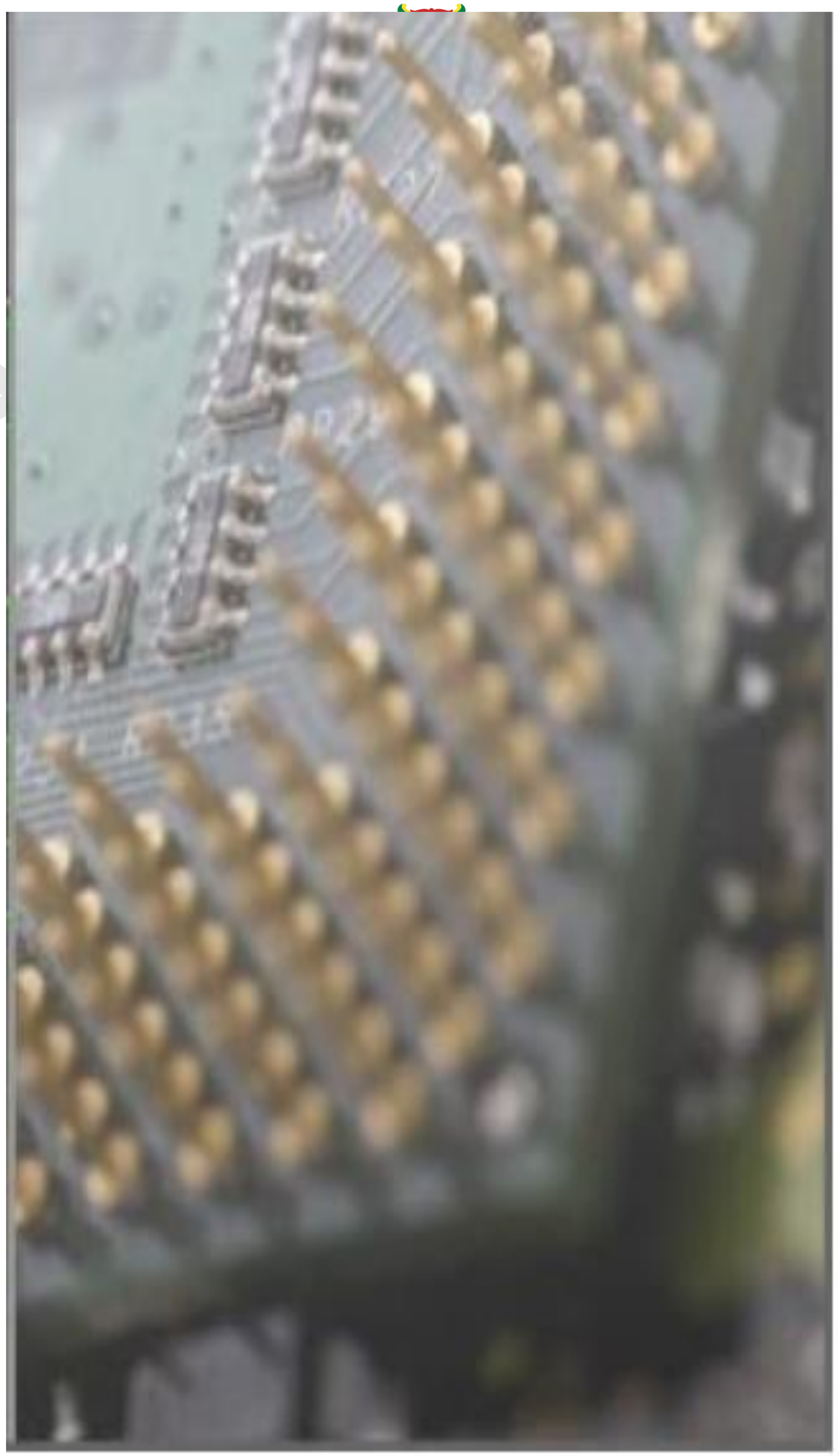
```
always @(posedge clk) begin: P1
    a <= a + 1;
end

always @(posedge clk) begin : P2
    y <= a;
end
```

P1 evaluate a+ = a+1
P2 evaluate y+ = a

**(+) symbol indicate next state.**

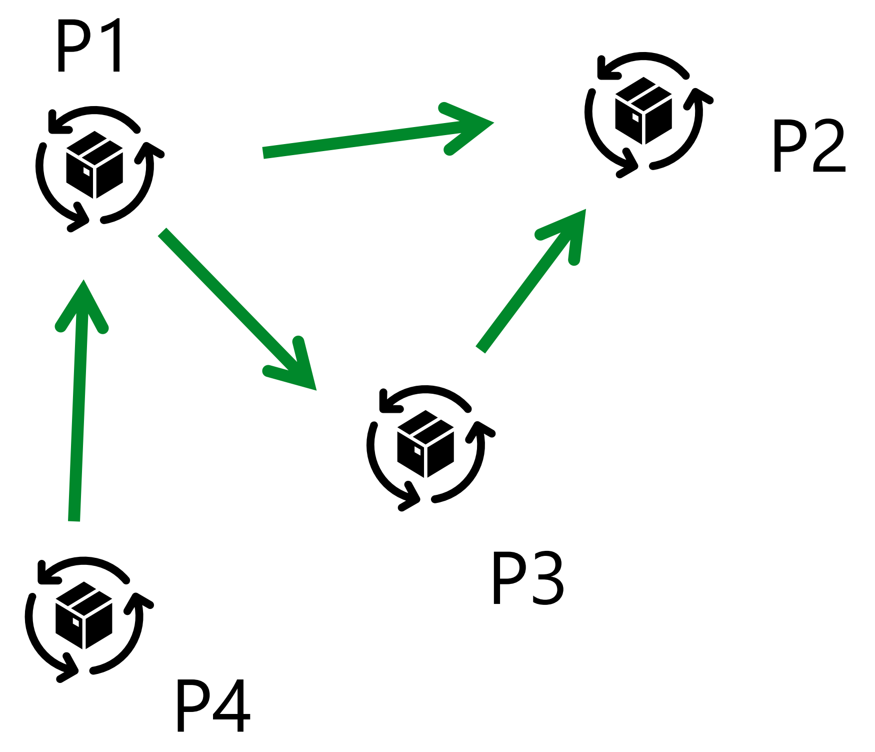


## Simulation cycle



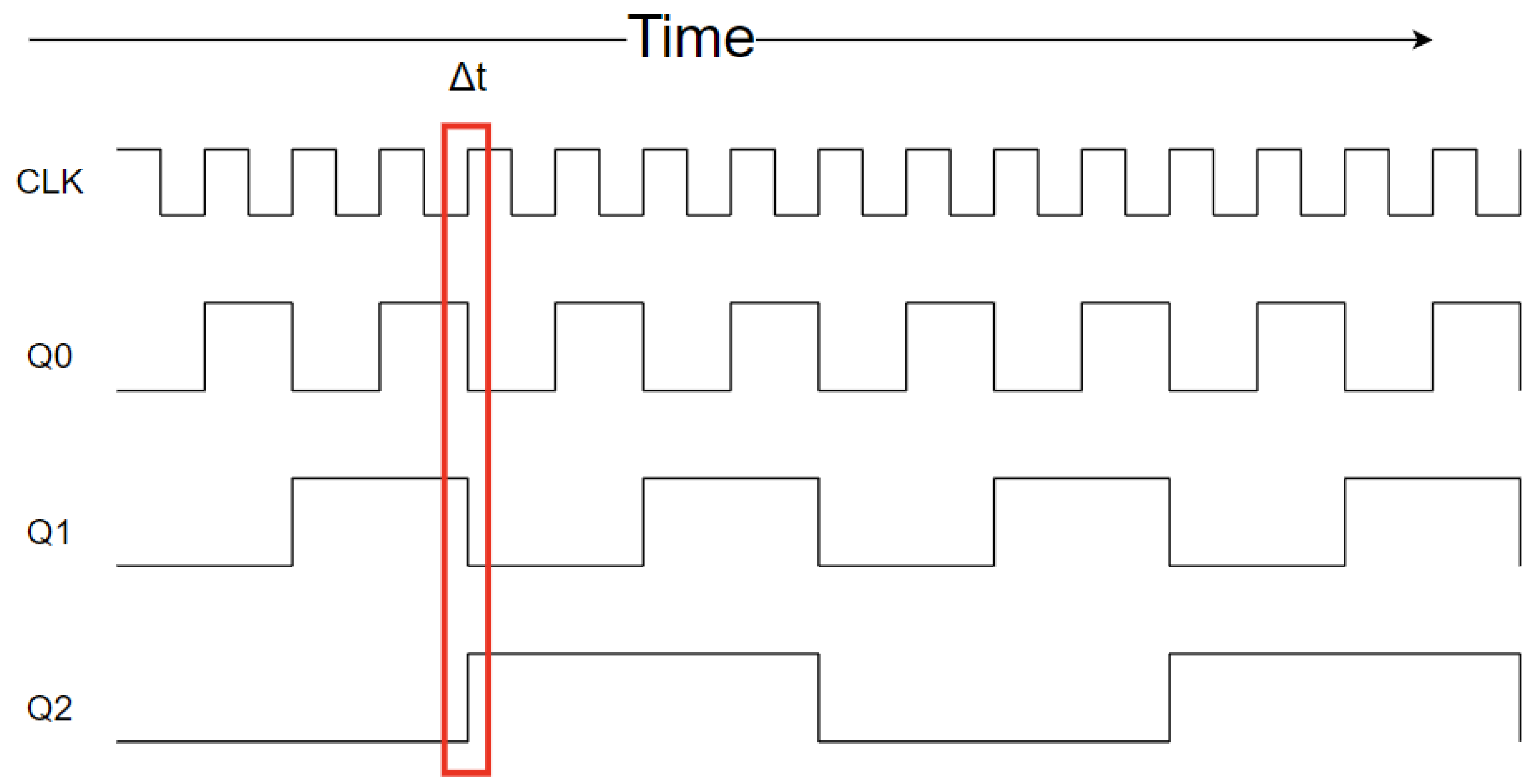
# Verilog simulation

Verilog simulations are based on processes and events. This approach allows the simulation to capture how signals **change over time** and how different **blocks of code react to these changes**.



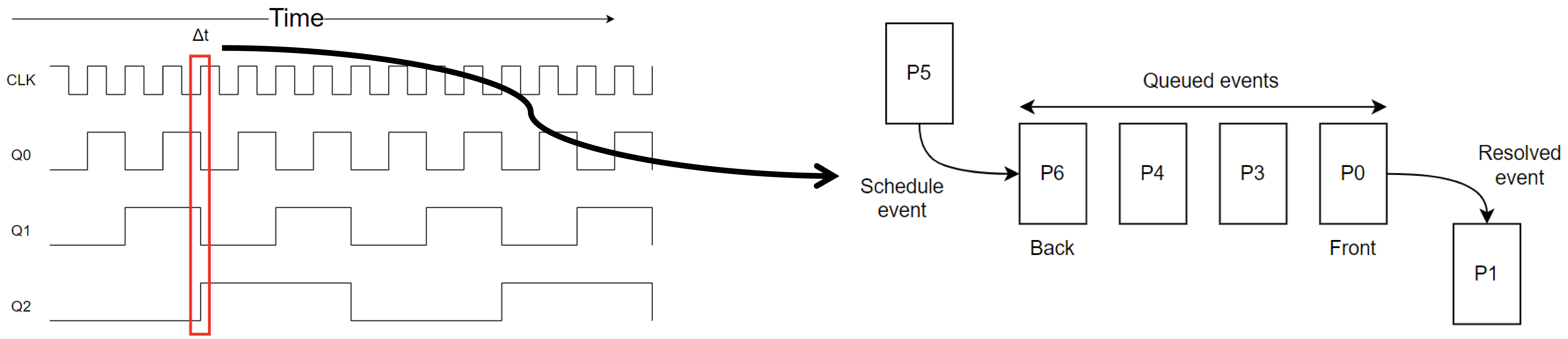
## Simulation time

Simulation time represents the temporal progression in the simulation (e.g. 10 nanoseconds). Delta cycles are **infinitesimal increments** of time that **allow resolving events occurring at the same logical time** without advancing the actual simulation time.
A **time slot** is a specific point in simulation time where events occur.

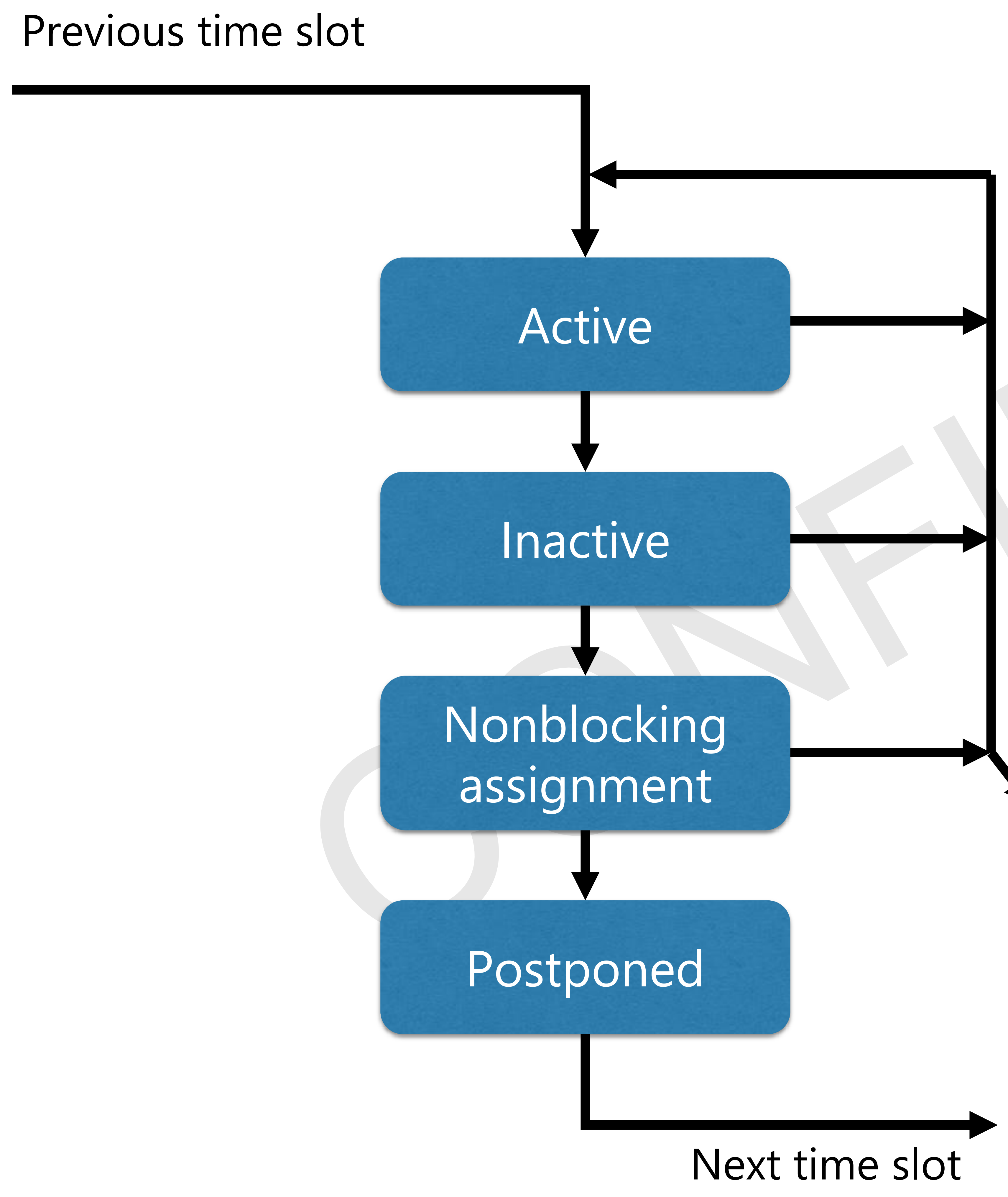


# Scheduling events

Verilog organizes events into different queues, which are processed in a specific order during each **time slot**.



## Time slot



Update and evaluate blocking assignments.
Evaluate right hand statements (RHS) of nonblocking assignments.
Continuous assignments.
**$display** command.

Wait for resolve statements (event queues) //#0 delay.

Update left hand statements (LHS) of nonblocking assignments.

NonBlocking Assignment (NBA) updates can trigger processes and return to active region, this is called **delta cycle.**

Final values **$monitor, $strobe** commands.

Cinvestav Guadalajara

Nayarit NUESTRA LEALTAD Y COMPROMISO | COCYTEN

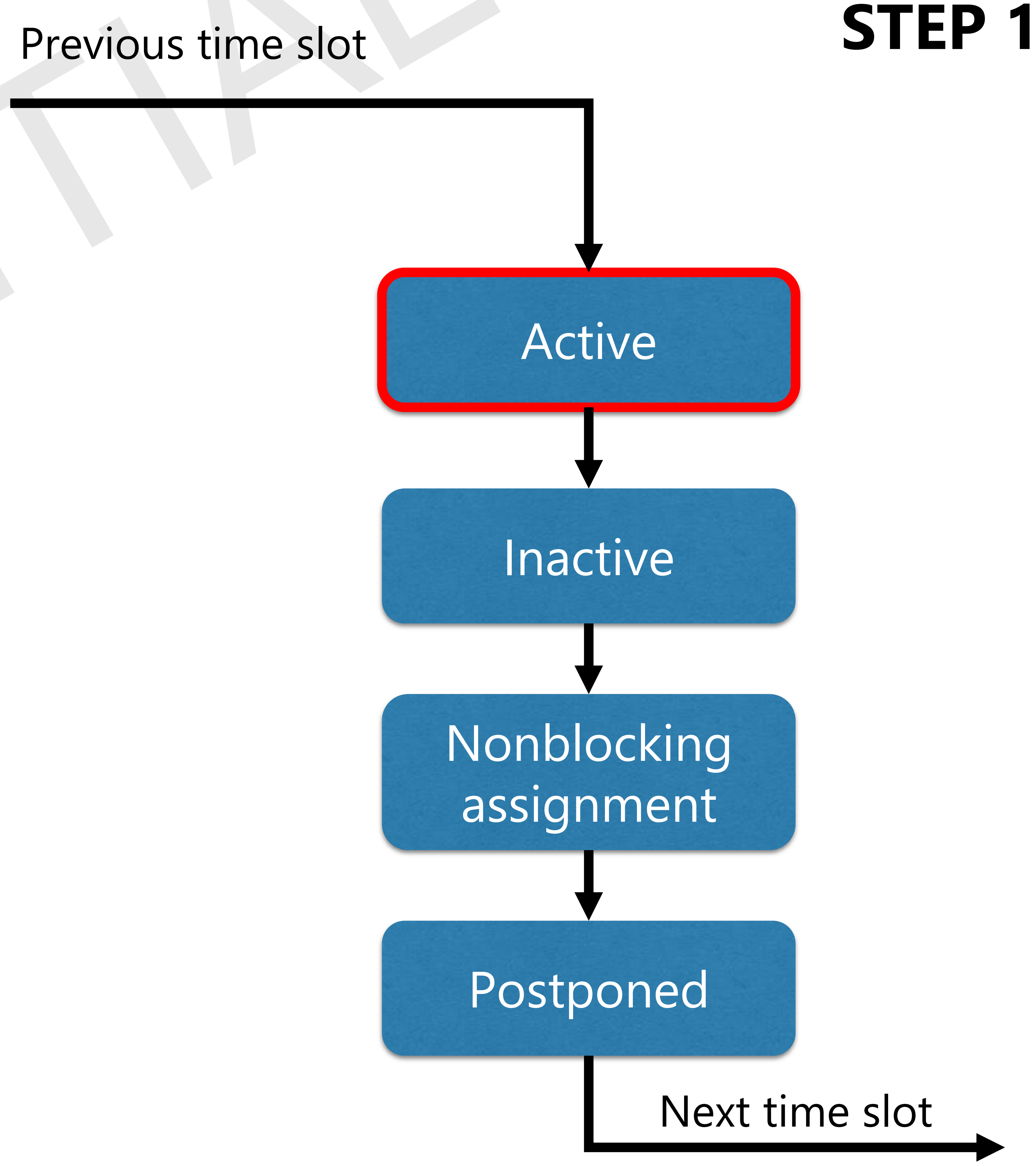
## Time slot

```
module tb;
reg a=0, b=0, c=0, d=0;

  initial
    begin: P0
      #10;
      a = 1;
      #10;
      $display("P0--a: %b, d: %b, c: %b",a,d, c);
    end

  always @(a or b)
    begin: P1
    c <= a^b;
      $display("P1--a: %b, d: %b, c: %b",a,d, c);
    end

  always @(a or c)
    begin: P2
      d <= c & a;
      $display("P2--a: %b, d: %b, c: %b",a, d, c);
    end
endmodule
```

delay

| | |
|---|---|
| a | 0 |
| b | 0 |
| c | 0 |
| d | 0 |

**STEP 1**



## Time slot

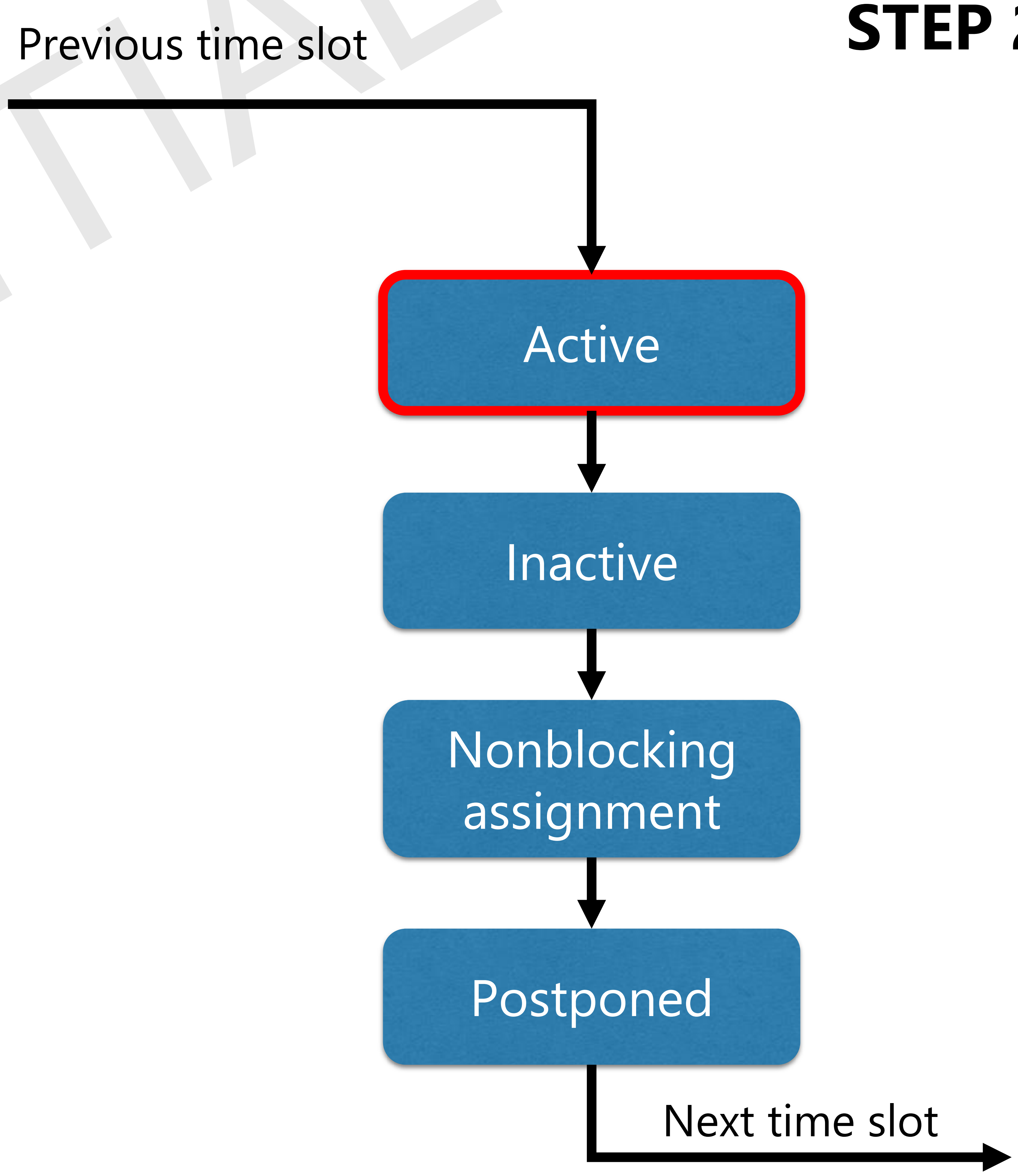
**STEP 2**

```
module tb;
reg a=0, b=0, c=0, d=0;

  initial
    begin: P0
      #10;
      a = 1;
      #10;
      $display("P0--a: %b, d: %b, c: %b",a,d, c);
    end

  always @(a or b)
    begin: P1
    c <= a^b;
      $display("P1--a: %b, d: %b, c: %b",a,d, c);
    end

  always @(a or c)
    begin: P2
      d <= c & a;
      $display("P2--a: %b, d: %b, c: %b",a, d, c);
    end
endmodule
```

set a=1

change in **a** trigger **P1** and **P2.**

| | |
|---|---|
| a | 1 |
| b | 0 |
| c | 0 |
| d | 0 |



## Time slot

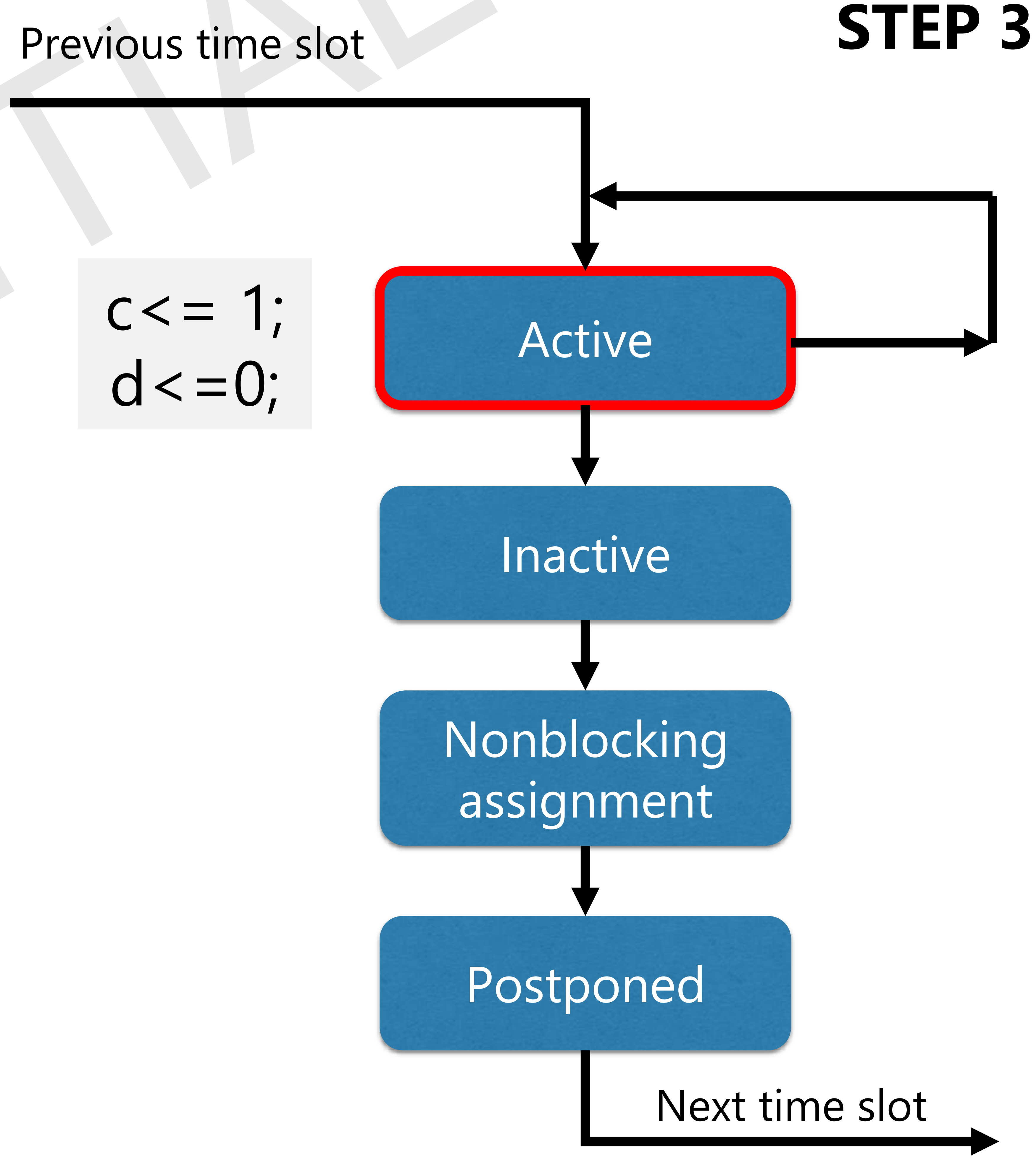
```
module tb;
reg a=0, b=0, c=0, d=0;

  initial
    begin: P0
      #10;
      a = 1;
      #10;
      $display("P0--a: %b, d: %b, c: %b",a,d, c);
    end

  always @(a or b)
    begin: P1
    c <= a^b;
      $display("P1--a: %b, d: %b, c: %b",a,d, c);
    end

  always @(a or c)
    begin: P2
      d <= c & a;
      $display("P2--a: %b, d: %b, c: %b",a, d, c);
    end
endmodule
```

Go back to active region and evaluate RHS of **P1** and **P2.**

| | |
|---|---|
| a | 1 |
| b | 0 |
| c | 0 |
| d | 0 |

**STEP 3**

c<= 1;
d<=0;



# Time slot

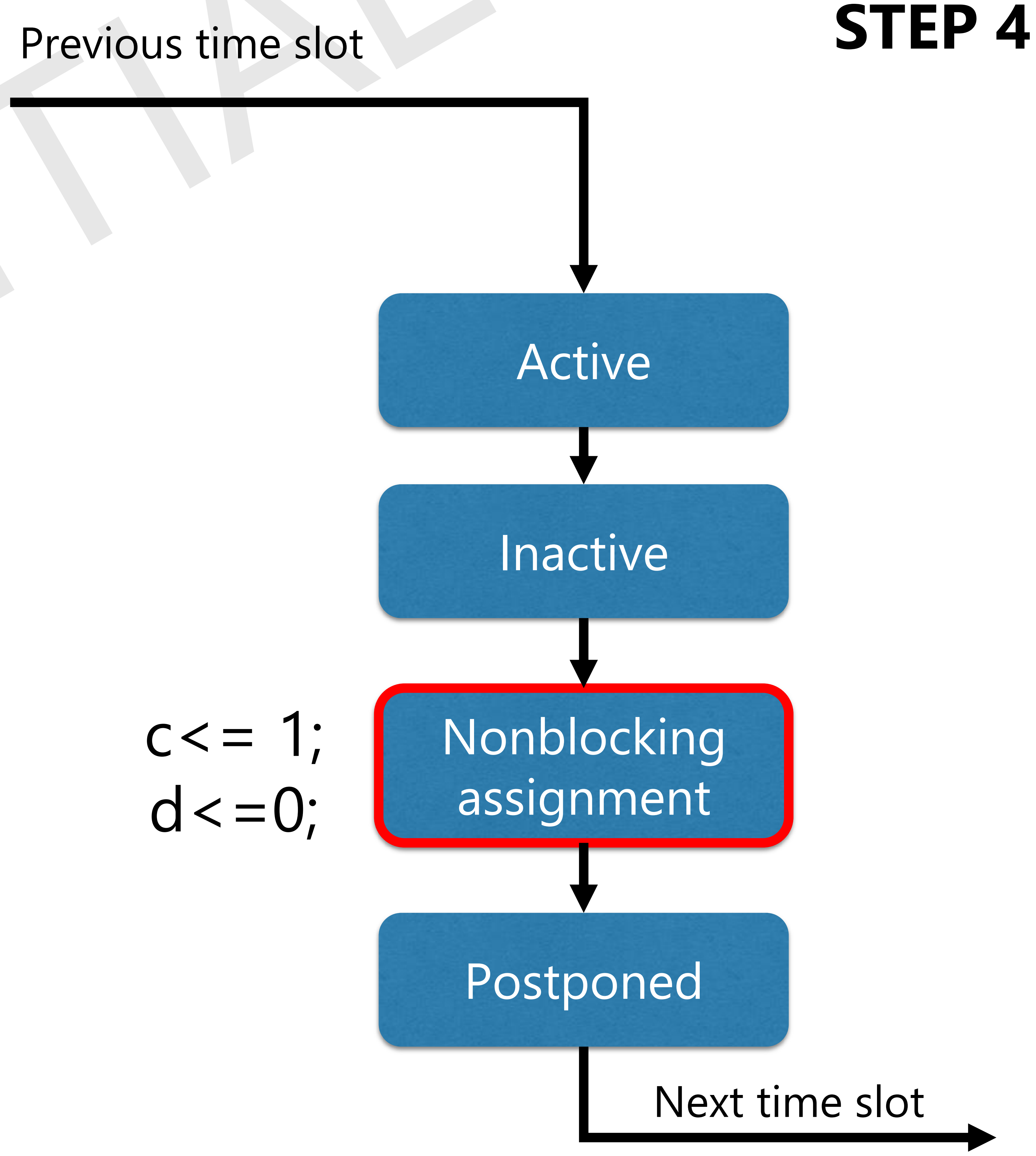
**STEP 4**

```
module tb;
reg a=0, b=0, c=0, d=0;

  initial
    begin: P0
      #10;
      a = 1;
      #10;
      $display("P0--a: %b, d: %b, c: %b",a,d, c);
    end

  always @(a or b)
    begin: P1
    c <= a^b;
      $display("P1--a: %b, d: %b, c: %b",a,d, c);
    end

  always @(a or c)
    begin: P2
      d <= c & a;
      $display("P2--a: %b, d: %b, c: %b",a, d, c);
    end
endmodule
```

Go to NBA region and update **c** and **d.**

Change in c trigger **P2.**

| | |
|---|---|
| a | 1 |
| b | 0 |
| c | 1 |
| d | 0 |



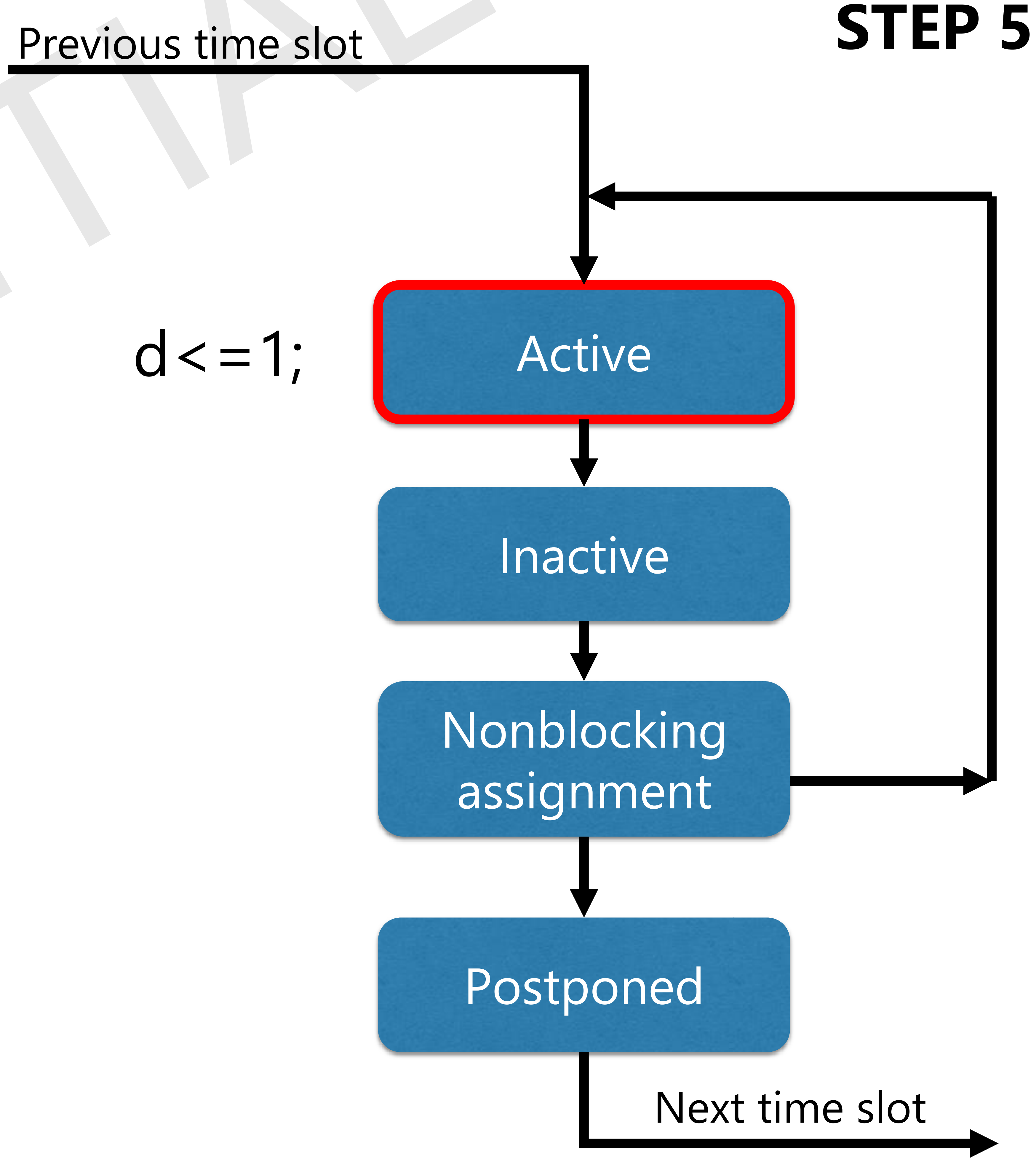
## Time slot

```
module tb;
reg a=0, b=0, c=0, d=0;

  initial
    begin: P0
      #10;
      a = 1;
      #10;
      $display("P0--a: %b, d: %b, c: %b",a,d, c);
    end

  always @(a or b)
    begin: P1
    c <= a^b;
      $display("P1--a: %b, d: %b, c:
    end

  always @(a or c)
    begin: P2
      d <= c & a;
      $display("P2--a: %b, d: %b, c: %b",a, d, c);
    end
endmodule
```

Go back to active region and evaluate RHS in **P2.**

|   |   |
|---|---|
| a | 1 |
| b | 0 |
| c | 1 |
| d | 0 |

**STEP 5**



# Time slot

**STEP 6**

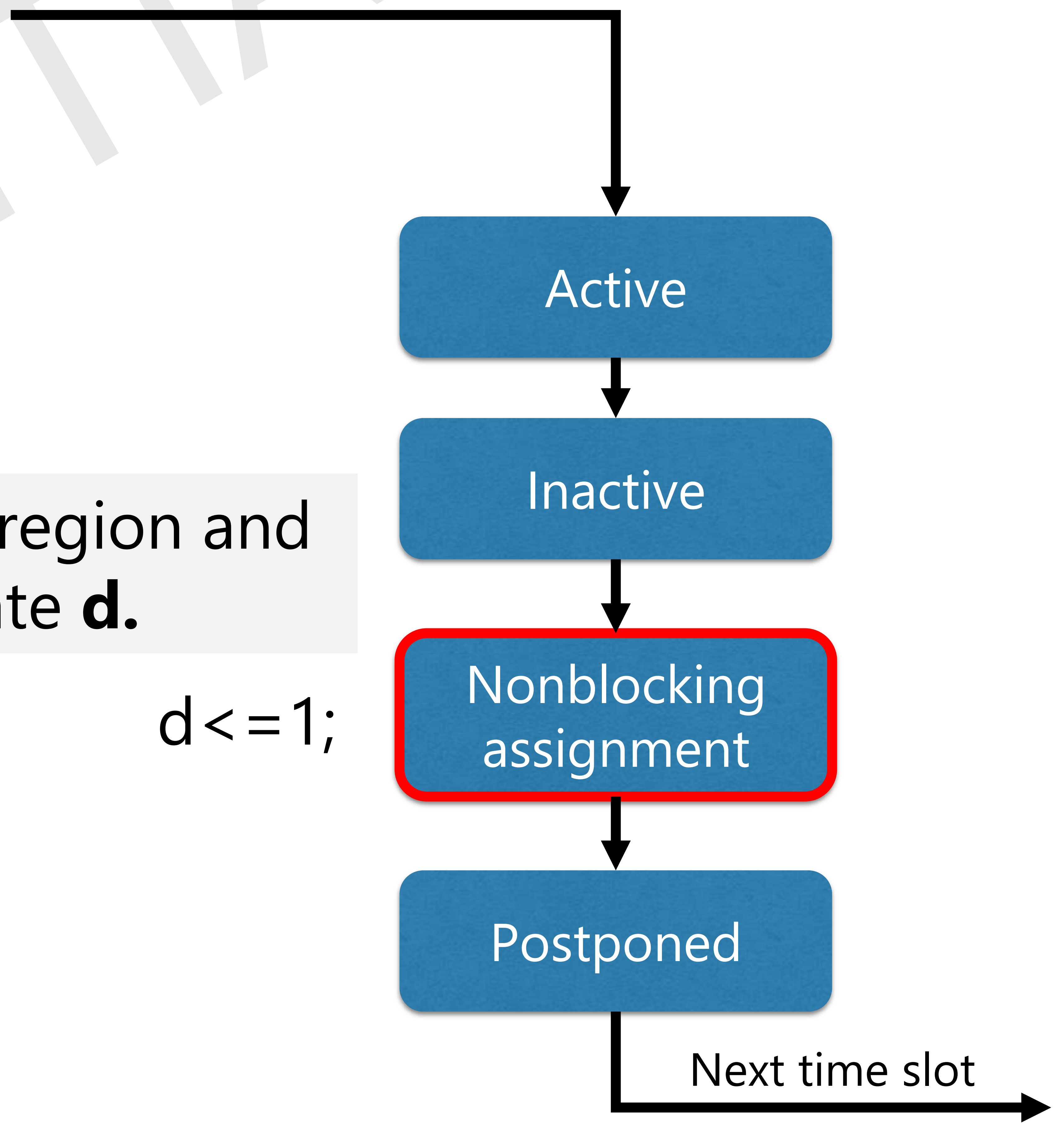
```
module tb;
reg a=0, b=0, c=0, d=0;

  initial
    begin: P0
      #10;
      a = 1;
      #10;
      $display("P0--a: %b, d: %b, c: %b",a,d, c);
    end

  always @(a or b)
    begin: P1
    c <= a^b;
      $display("P1--a: %b, d: %b, c: %b",a,d, c);
    end

  always @(a or c)
    begin: P2
      d <= c & a;
      $display("P2--a: %b, d: %b, c: %b",a, d, c);
    end
endmodule
```

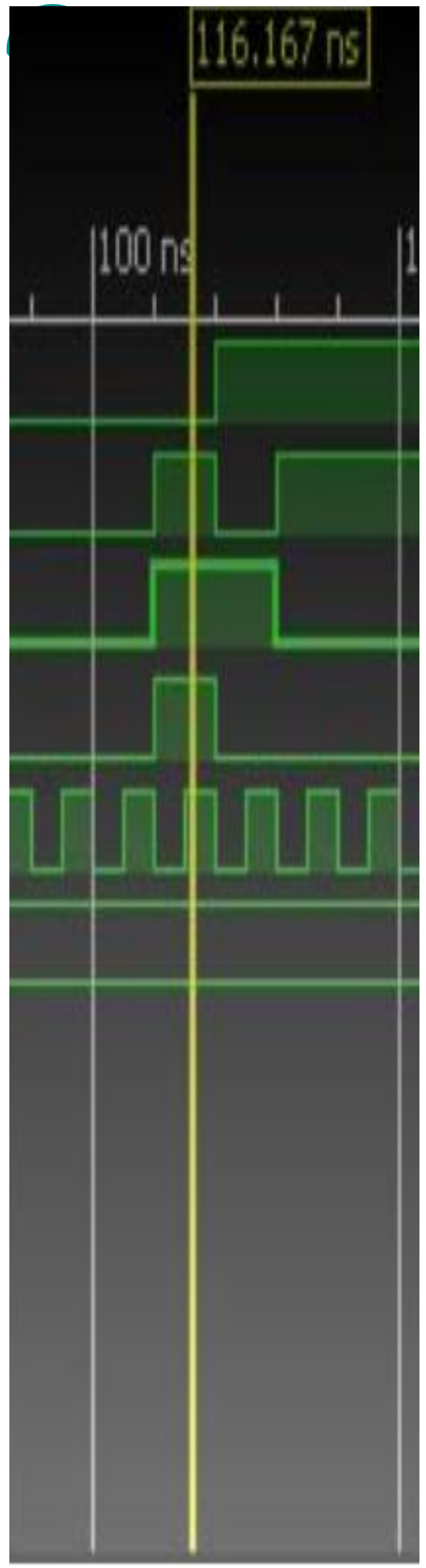
There are no changes in variables that trigger procedural blocks.

Go to NBA region and evaluate **d.**

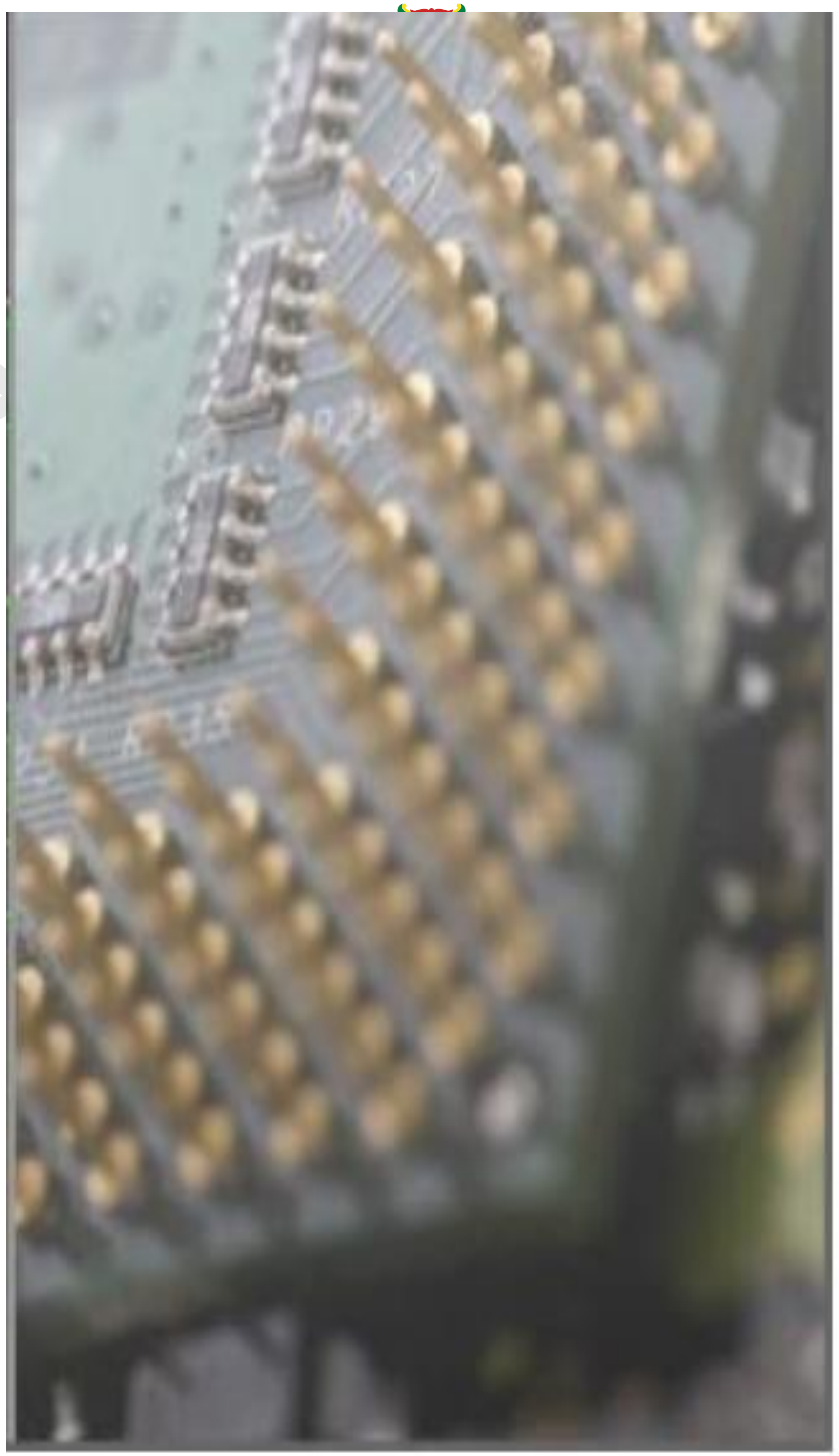
d<=1;

| | |
|---|---|
| a | 1 |
| b | 0 |
| c | 1 |
| d | 1 |





## Exercises



## Exercise

How can we model a clock?

## Exercise

How can we model a clock?

```
parameter HALF_PERIOD=10;

reg clk;

initial
begin
    clk = 0;
end

always #HALF_PERIOD clk = ~clk;
```

```
parameter HALF_PERIOD=10;

reg clk = 0;

always #HALF_PERIOD clk = ~clk;
```

```
parameter HALF_PERIOD=10;

reg clk;

initial
begin
    clk = 0;
    forever
    begin
        #HALF_PERIOD clk = ~clk;
    end
end
```

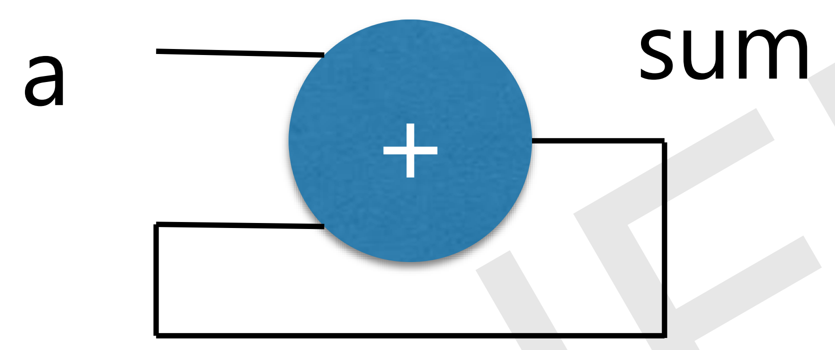
## Exercise

What is the behavior of the following code?

```
module tb;
  reg [7:0]sum;
  reg [7:0]a;

  initial begin
    $display("start");
    sum = 3;
    a = 1;
    #10;
  end

  always @(*)
    begin
      sum = sum+a;
      $display("sum update sum: %0d",sum);
    end

endmodule
```

## Exercise

What is the behavior of the following code?

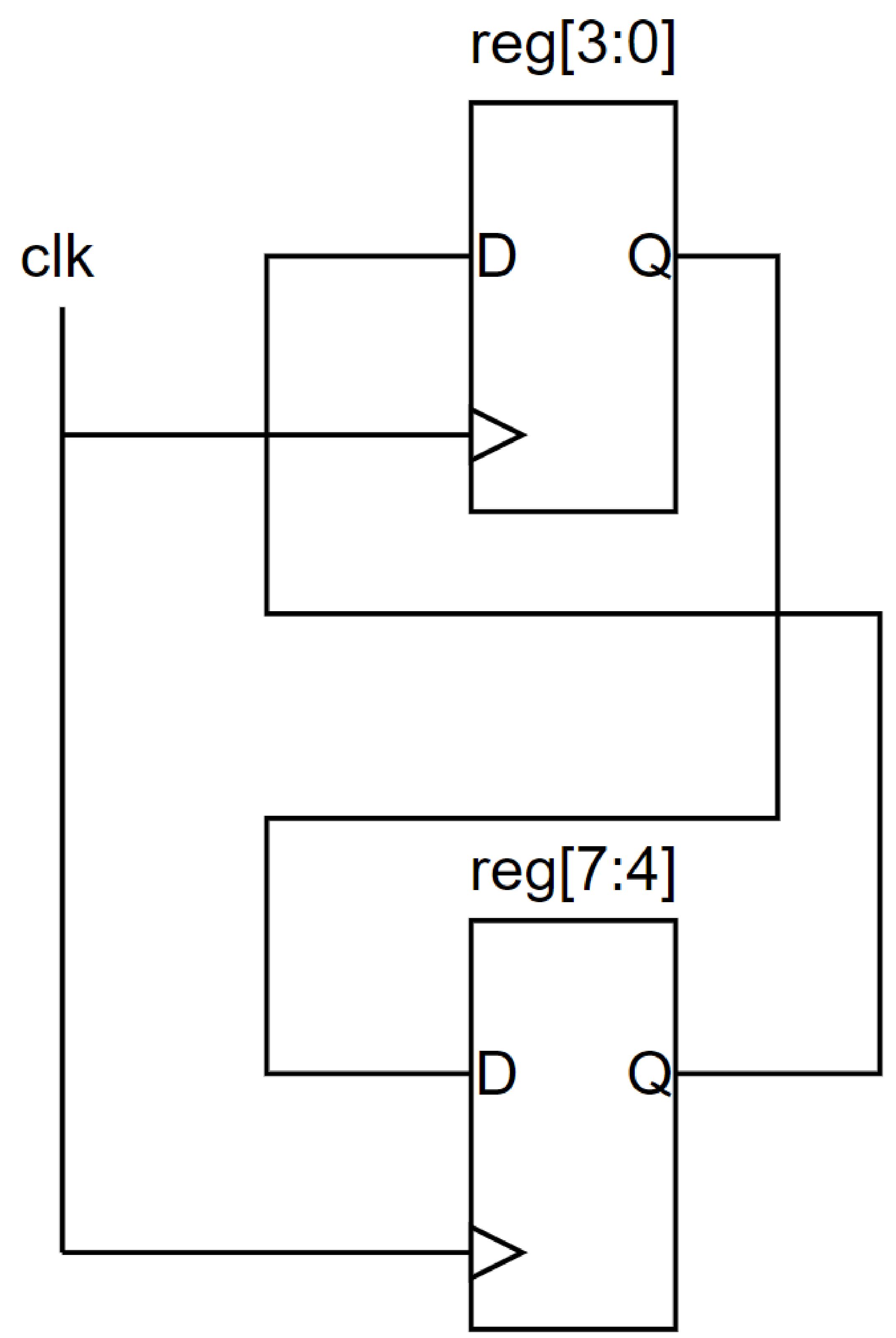


```
module tb;
  reg [7:0]sum;
  reg [7:0]a;

  initial begin
    $display("start");
    sum = 3;
    a = 1;
    #10;
  end

  always @(*)
    begin
      sum = sum+a;
      $display("sum update sum: %0d",sum);
    end

endmodule
```

Combinational loops could hang the simulation and should be avoided.
Remember that feedback circuits must be sequential.

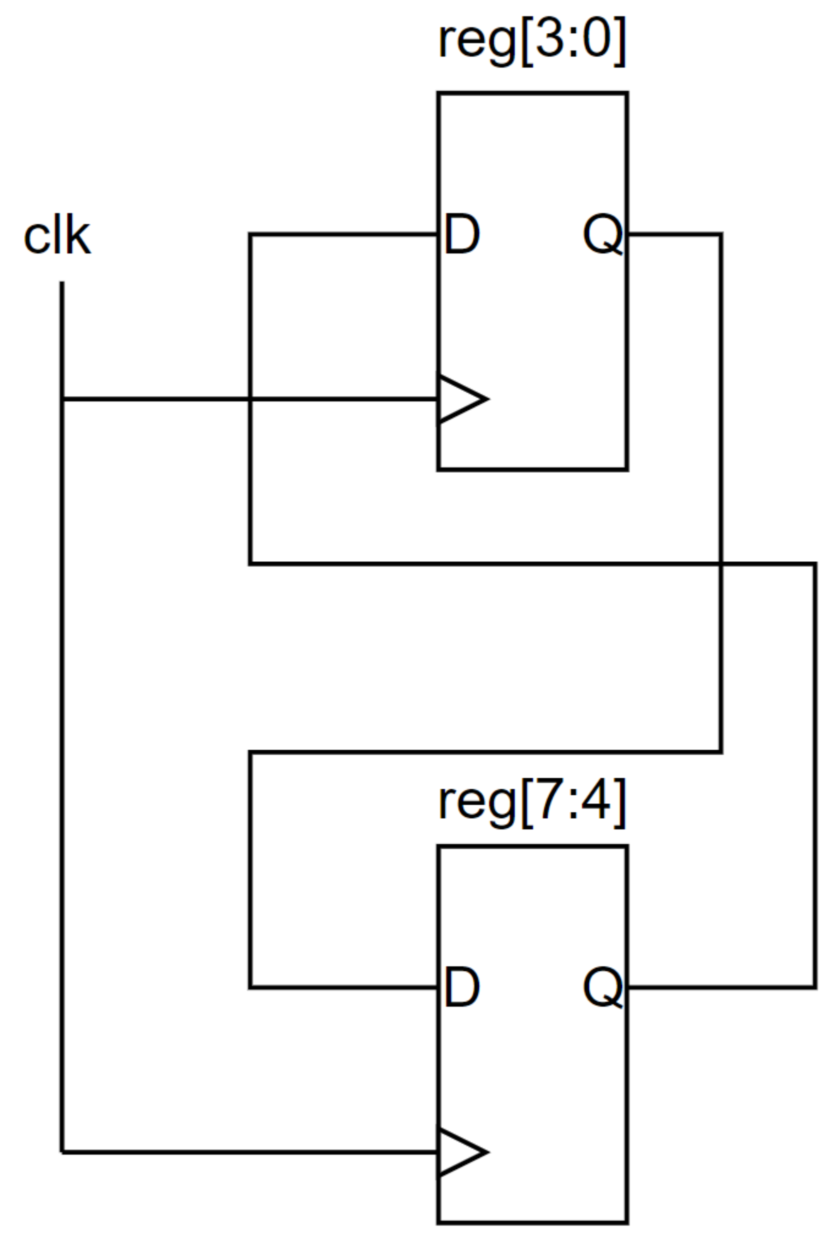
## Exercise

How can we swap the nibbles of a byte?



## Exercise

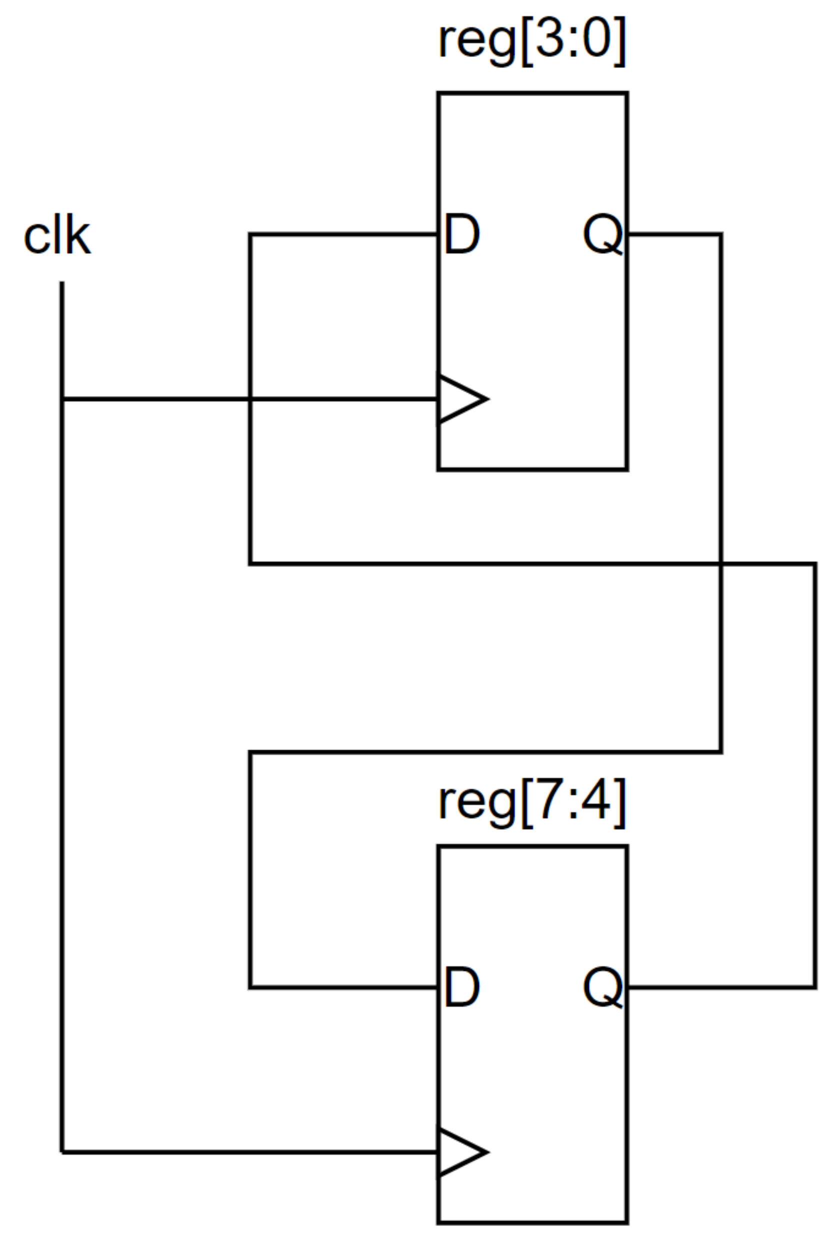
How can we swap the nibbles of a byte?



```
module tb;
  reg [7:0] r_byte;
  reg clk=0;

  always #5 clk = ~clk;

  initial begin
    r_byte = 8'h0f;
    repeat (4) begin
        @(posedge clk);
        #1;
        $display("swaped byte %b",r_byte);
    end
    $finish;
  end

  always @(posedge clk)
    begin
      r_byte[7:4] <= r_byte[3:0];
      r_byte[3:0] <= r_byte[7:4];
    end
endmodule
```

# Exercise

How can we swap the nibbles of a byte?
Can we model this using blocking assignments?



```
module tb;
  reg [7:0] r_byte;
  reg clk=0;

  always #5 clk = ~clk;

  initial begin
    r_byte = 8'h0f;
    repeat (4) begin
        @(posedge clk);
        #1;
        $display("swaped byte %b",r_byte);
    end
    $finish;
  end

  always @(posedge clk)
    begin
      r_byte[7:4] <= r_byte[3:0];
      r_byte[3:0] <= r_byte[7:4];
    end
endmodule
```

## Exercise

How can we swap the nibbles of a byte?
Can we model this using blocking assignments?

- It is possible to model the behavior of this circuit using blocking assignment with an auxiliary variable, as we would in software, but this is incorrect. We must always keep in mind that Verilog is used to model electrical circuits, so techniques like the one shown in the code, which might be valid in software, are not correct for describing hardware.

```
module tb;
  reg [7:0] r_byte;
  reg [7:0] aux_reg;
  reg clk=0;

  always #5 clk = ~clk;

  initial begin
    r_byte = 8'h0f;
    repeat (4) begin
        @(posedge clk);
        #1;
        $display("swaped byte %b",r_byte);
    end
    $finish;
  end

  always @(posedge clk)
    begin
      aux_reg[7:4] = r_byte[3:0];
      aux_reg[3:0] = r_byte[7:4];
      r_byte = aux_reg;
    end
endmodule
```

**Remember, not following guidelines could result in incorrect hardware.**

## Exercise

Is correct mixing blocking and nonblocking assignment in a procedural block?

```
always @(posedge clk)
begin : P0
    reg[7:0] m;
    m = a + b;
    s <= m -1;
end
```

Cinvestav Guadalajara

Nayarit NUESTRA LEALTAD Y COMPROMISO | COCYTEN

## Exercise

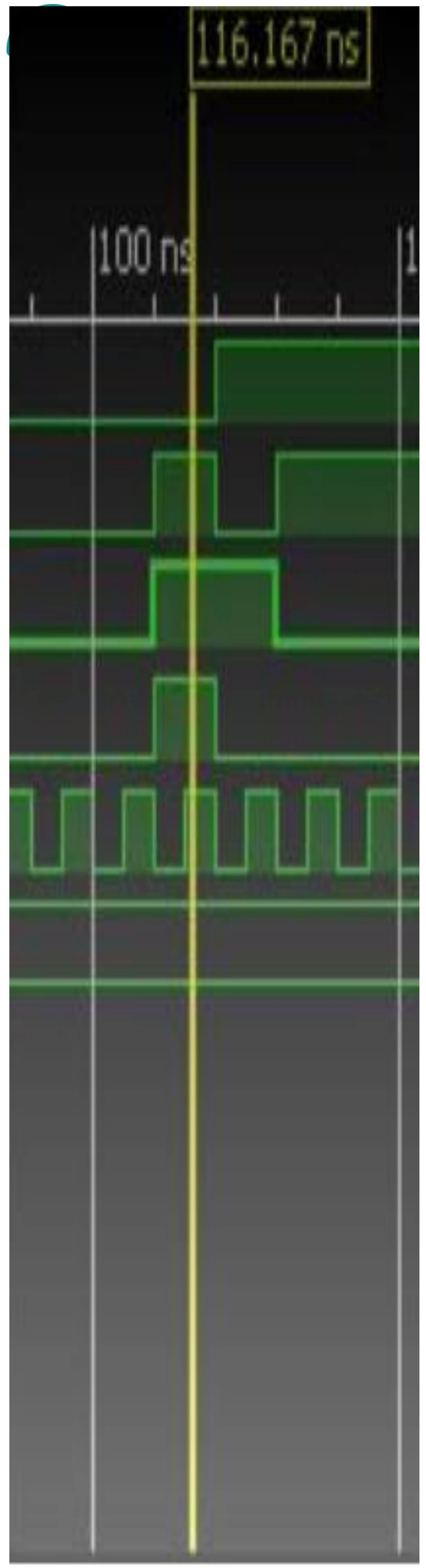
Is correct mixing blocking and nonblocking in a procedural block?

```
always @(posedge clk)
begin : P0
    reg[7:0] m;
    m = a + b;
    s <= m -1;
end
```

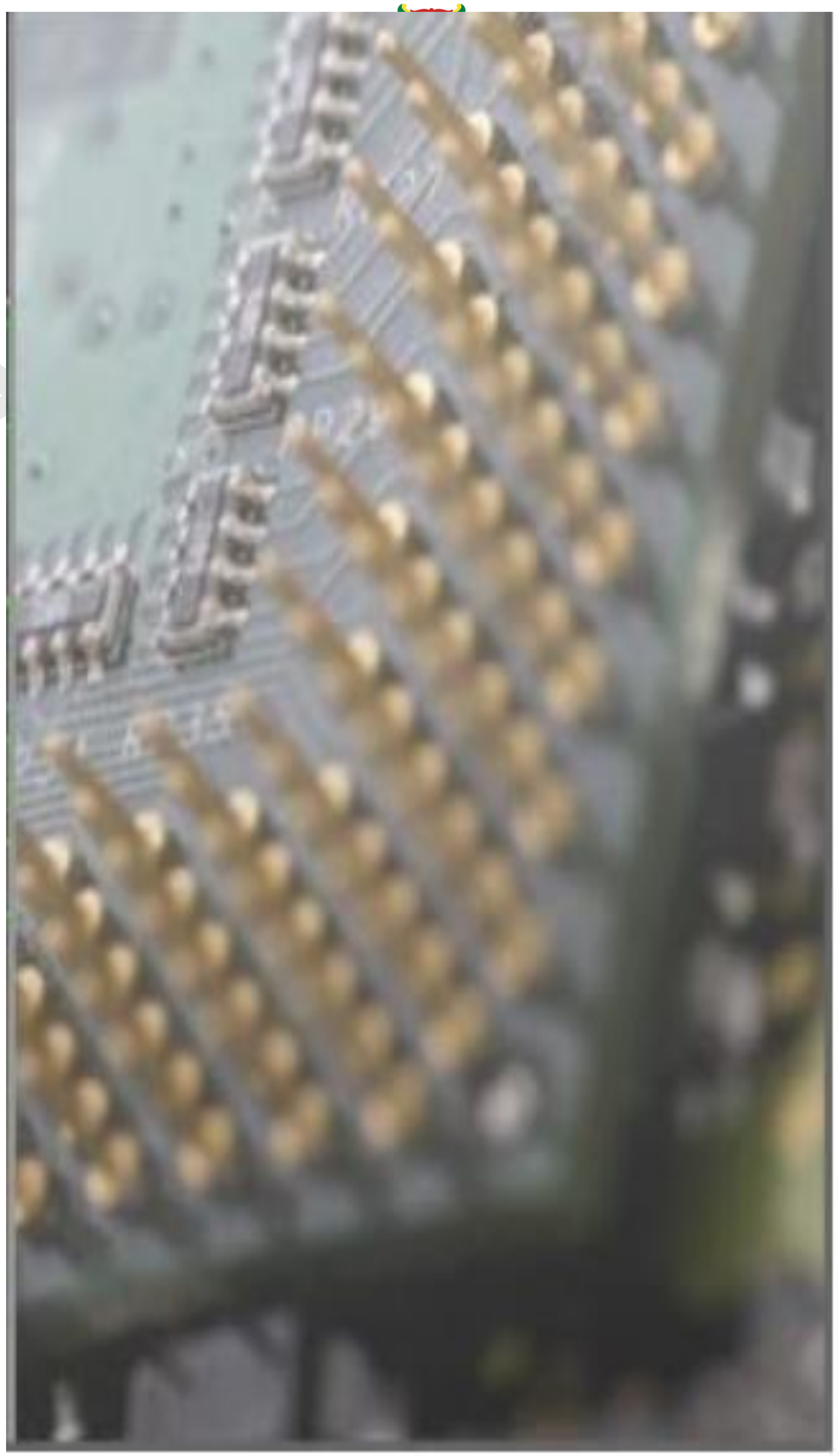
**In order to declare local variables, it is necessary to name the block.**

We can use temporary variables to simplify assignments in sequential blocks. However, these variables should be strictly temporary and should not be used in another block.

**It is highly recommended to perform simplifications in combinational always blocks to avoid mistakes.**

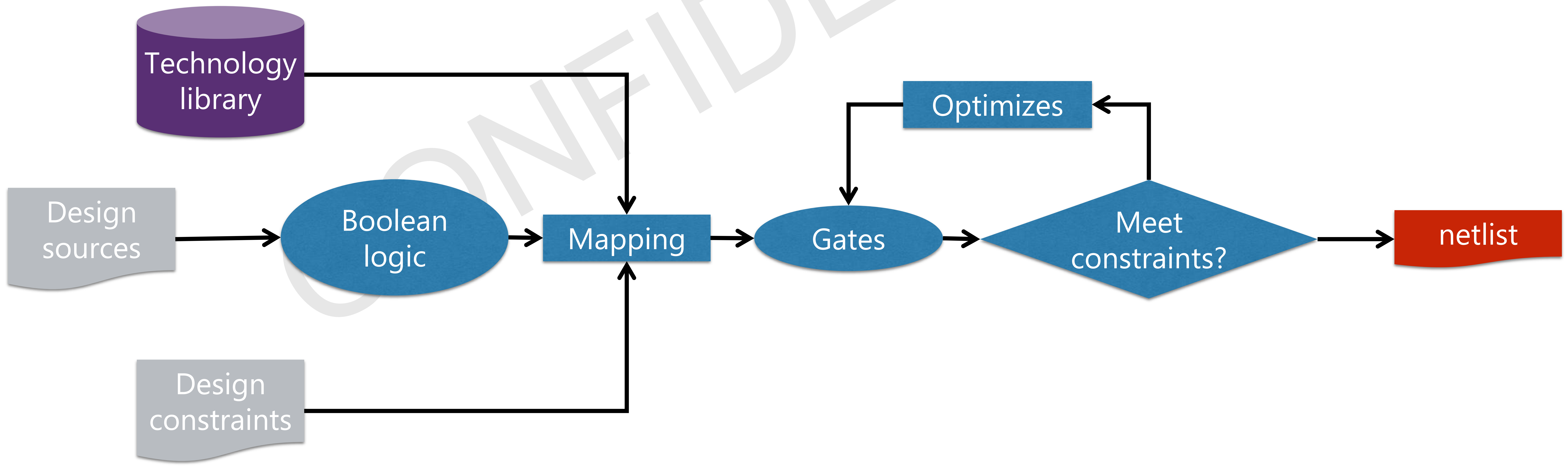


## Modeling Synthesizable logic



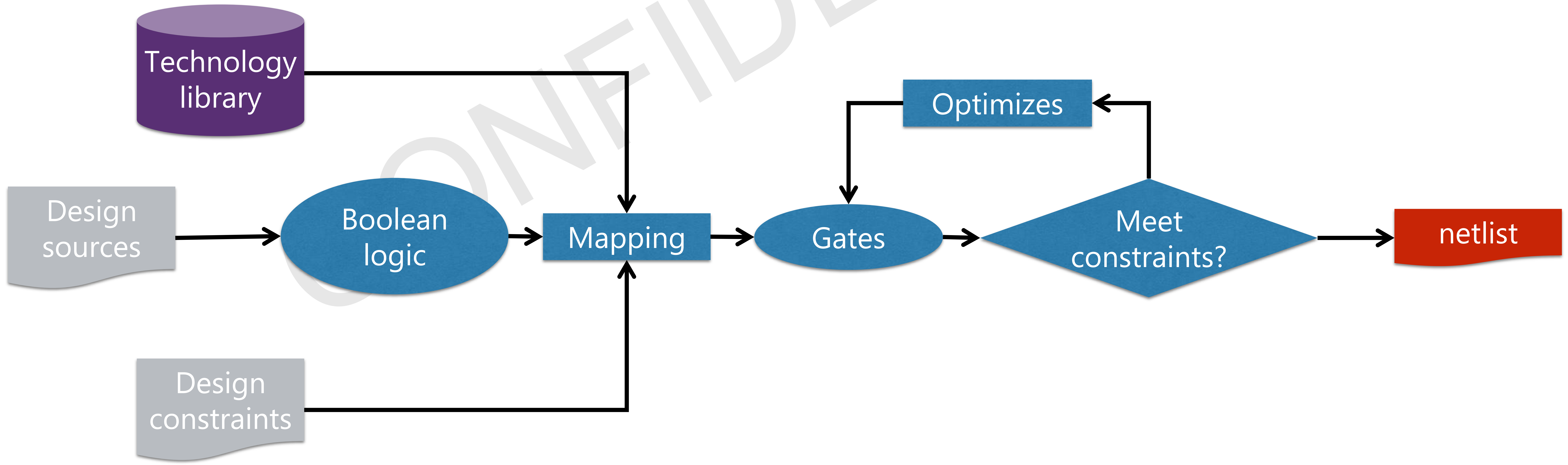
# What is logic synthesis?

Synthesis transform RTL into an optimized netlist of predefined cells.
We provide **design source**, **design constraints** and **technology library** to the tool, and it infers logic from the technology library and optimizes the circuit to meet constraints.



# Code matters !!

Synthesis tools don't read your mind. They transforms your RTL logic into a Boolean logic quite literally and then make optimizations. A good code can help to the process of optimization to obtain good results. Providing explicit descriptions of behavior can lead to more reliable hardware. **Remember, while you write RTL, what you are designing is hardware.**



# Code for synthesis

There are several guidelines for describing synthesizable RTL. The following are some recommendations for correctly describing hardware for both combinational and sequential logic.

# Combinational logic

Combinational logic is modeled as a process that is evaluated continuously whenever an input changes.

```
wire w = expression;

assign w = expression;

always * begin
    rega = expression;
end
```

## Combinational logic: sensitive list

"When using procedural blocks to describe combinational logic, it is important to include all inputs in the sensitivity list. The easiest way to achieve this is by using @*.

```
always @(a,b,sel) begin
    y = a;
    if (sel)
        y = b;
end
```

```
always @* begin
    y = a;
    if (sel)
        y = b;
end
```

## Combinational logic: Incomplete assignment

If an execution path in a procedure block is incomplete and one output is not updated, then that output value must be retained. In this case the synthesis tolls will infer a latch to model the behavior.

```verilog
//latch
always @* begin
    if (sel)
        y = b;
end
```

```verilog
//latch
always @* begin
    case (sel)
        1:y = b;
    endcase
end
```

# Combinational logic: Incomplete assignment

If an execution path in a procedure block is incomplete and one output is not updated, then that output value must be retained. In this case the synthesis tolls will infer a latch to model the behavior.

```
//latch
always @* begin
    if (sel)
        y = b;
end
```

```
//latch
always @* begin
    case (sel)
        1:y = b;
    endcase
end
```

Avoid latches is easy!!! Just complete the path or provide a default value.

```
//default value
always @* begin
    y = 1'b0;
    if (sel)
        y = b;
end
```

```
//complete path
always @* begin
    if (sel)
        y = b;
    else
        y = 1'b0;
end
```

```
//complete path
always @* begin
    case (sel)
        0:y = 1'b0;
        1:y = b;
    endcase
end
```

```
//default value
always @* begin
    y = 1'b0;
    case (sel)
        1:y = b;
    endcase
end
```

## Sequential logic

For modeling sequential logic, we use always procedural blocks and continuous assignments. Modeling sequential logic has some basic rules:

- Use only edge events in the sensitive list.
- Use non-blocking assignment for non temporary variables (flop-flops).

```
//counter
always @(posedge clk)
begin
    if (count = 9)
        count <= 4'd0;
    else
        count <= count + 1;
end
```

How do we initialize the counter?

# Sequential logic: reset

We can model a reset adding an if statement.

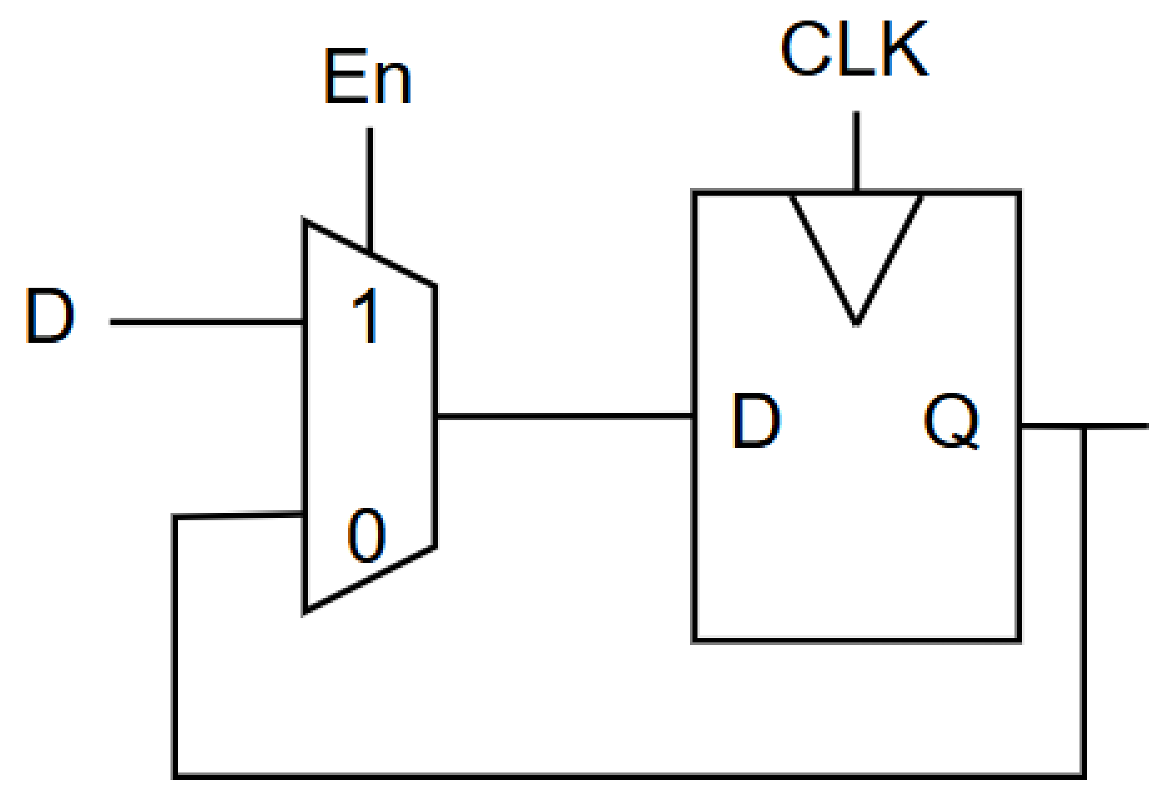
```
//synchronous
always @(posedge clk)
begin
    if (!rst)
        count <= 4'd0;
    else begin
        if (count = 9)
            count <= 4'd0;
        else
            count <= count + 1;
    end
end
```

```
//Asynchronous
always @(posedge clk or negedge rst)
begin
    if (!rst)
        count <= 4'd0;
    else begin
        if (count = 9)
            count <= 4'd0;
        else
            count <= count + 1;
    end
end
```

# Sequential logic: Incomplete assignments

Incomplete assignments in sequential blocks does not infers a latch, ifers a flip-flop, since the block only react to edge of clk.

```
always @(posedge clk)
begin
    if (en)
        q <= d;
end
```


En
CLK
D
1
0
D
Q