Sharif University of Technology
Department of Computer Engineering

Digital System Design
Behavioral Modeling

Siavash Bayat-Sarmadi

Outline

- Structured Procedures
  - initial
  - always
- Procedural Assignments
  - Blocking
  - Non-blocking
- Timing Controls
- Conditional Statements
- Multiway Branching

Outline (Cont'd.)

- Loops
- Sequential Blocks
- Parallel Blocks
- Generate Blocks
- Examples

Behavioral Modeling

- Architectural evaluation at an algorithmic level
- Designers do not think in terms of logic gates or data flow
- Behavior of the circuit
- Resembles C programming
- A great amount of flexibility for designer

Structured Procedures

- always and initial
- Concurrent to each other
- Only in behavioral modeling
- Cannot be nested

Initial Statement

- Starts at time 0
- Executes exactly once during a simulation
- When more than one statement inside
  - Statements grouped with begin and end
- Multiple initial blocks
  - Start to execute concurrently at time 0

Initial Statement (Cont'd.)

```
module stimulus;
    reg x,y, a,b, m;

    initial
        m = 1'b0;

    initial
    begin
        #5 a = 1'b1;
        #25 b = 1'b0;
    end

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

    initial
        #50 $finish;

endmodule
```

Initial Statement (Cont'd.)

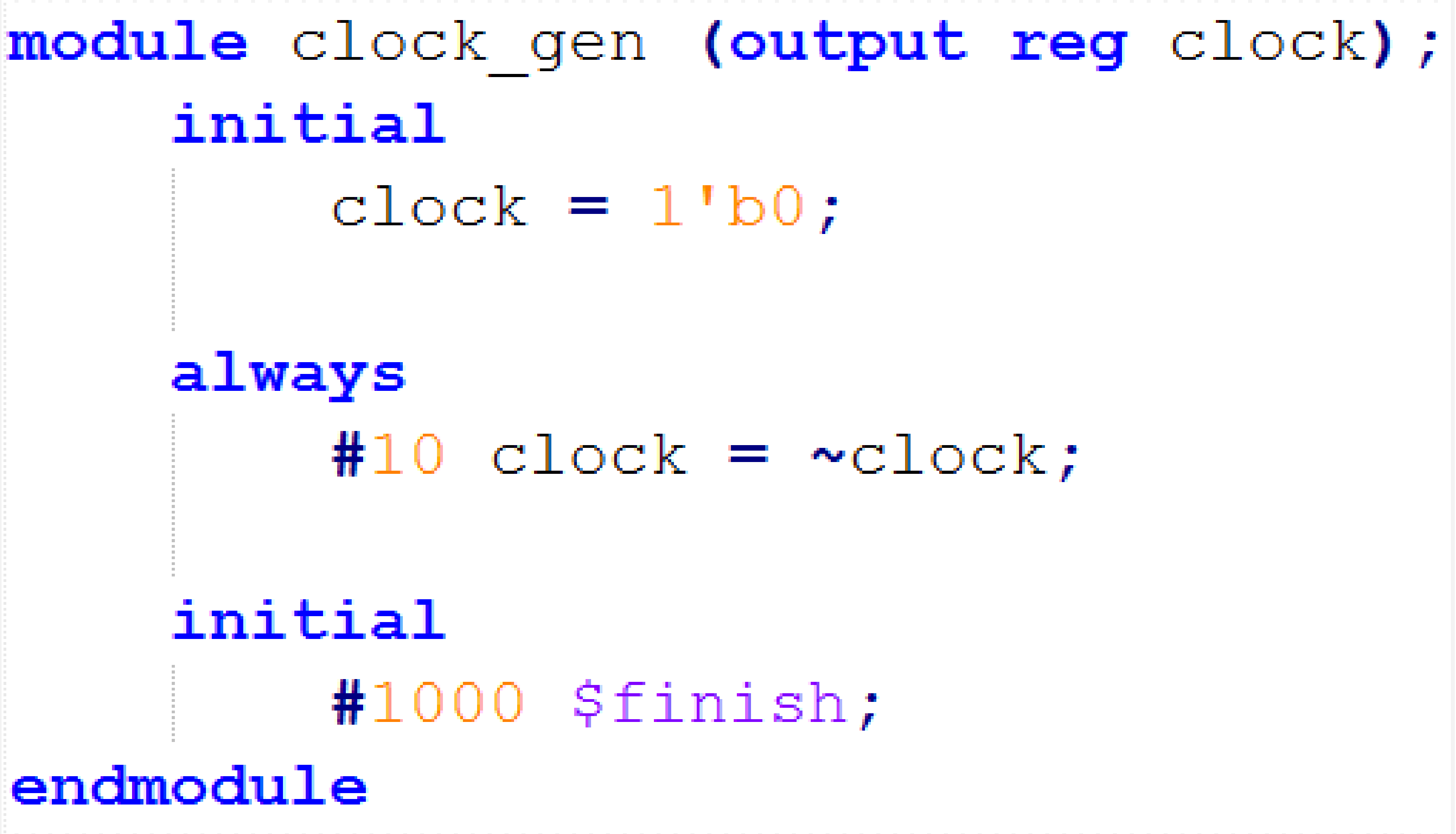
- Execution sequence

time	statement executed
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	$finish;

Always Statement

- Starts at time 0
- Executes continuously in a looping fashion
- Used to model a block of activity that is repeated continuously in a digital circuit

Example



```
module clock_gen (output reg clock);
    initial
        clock = 1'b0;

    always
        #10 clock = ~clock;

    initial
        #1000 $finish;
endmodule
```

Example (Cont'd.)

- Active from time 0 till the circuit is powered off
- Clock period
  - 20
- initial statement
  - Initialization of clock
  - simulation halt

Procedural Assignments

- Update values of variables
  - Rather than continuously drive a net
  - Compared with
    - Continuous assignment in dataflow modeling
- Types
  - Blocking =
  - Nonblocking <=
    - Mixing blocking and nonblocking in the same always blocks
      - Not recommended

Rules

- LHS
  - reg
  - integer
  - real
  - time
  - Bit/part select of these variables
  - Concatenation of these variables
- RHS
  - Any data type

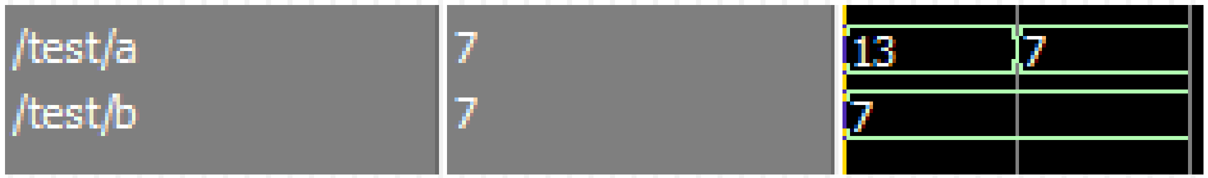
Rules (Cont'd.)

- When RHS has more bits than LHS
  - More significant bits truncated
- When LHS has more bits than RHS
  - More significant bits zero-filled

Blocking Assignments

- You're already familiar with
- Executed in the order they are specified

```
initial
begin
      a = 13;
      b = 7;
      #10
      a = b;
      b = a;
      #10
      $stop;
end
```

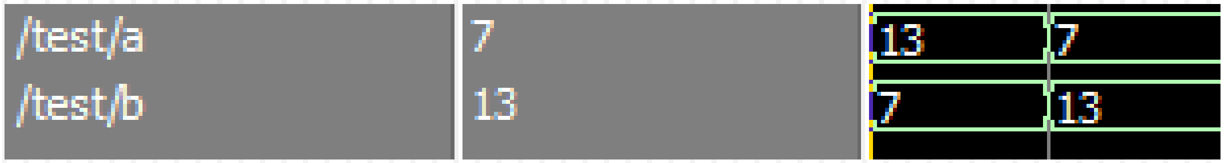


```
/test/a    7    13    7
/test/b    7    7
```

Nonblocking Assignments

- Allow scheduling of statements without blocking execution

```
initial
begin
      a <= 13;
      b <= 7;
      #10
      a <= b;
      b <= a;
      #10
      $stop;
end
```



Nonblocking Assignments (Cont'd.)

1. Evaluate RHS
   - Results stored
2. Schedule assignment
3. Advance time
   - Simulator does not wait for the nonblocking statement to complete execution
4. Assign the stored value

Operators

- Same as dataflow
- Example
  - a = b & c & d;
  - out <= in1 + in2;
  - out <= sel ? a : b;

Timing Controls

- Specify the simulation time at which procedural statements will execute
- Timing control:
  - Delay-based
  - Event-based
  - Level-sensitive

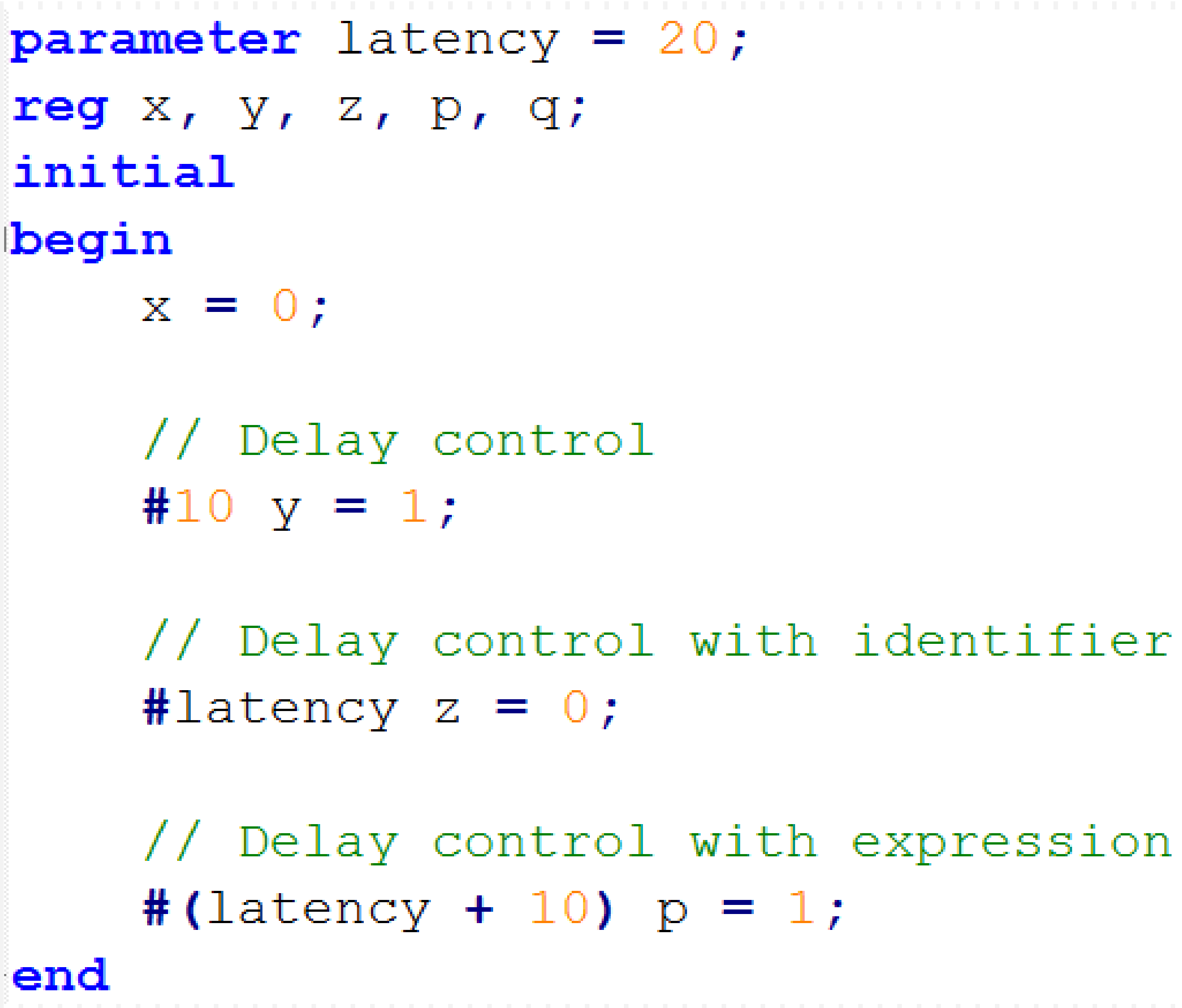
Delay-Based Timing Control

- The delay between
  - Encountering
  - Execution
- Symbol: #
- Types
  - Regular
  - Intra-assignment
  - Zero delay

Regular delay control

- Already familiar with
- Non-zero delay to the left of a procedural assignment

Example



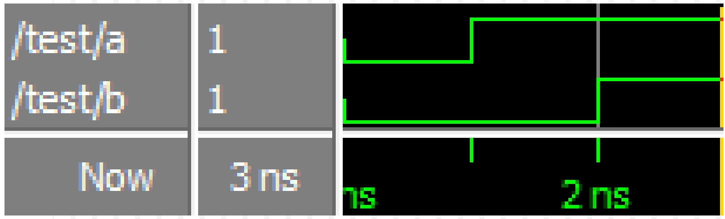
```
parameter latency = 20;
reg x, y, z, p, q;
initial
begin
    x = 0;

    // Delay control
    #10 y = 1;

    // Delay control with identifier
    #latency z = 0;

    // Delay control with expression
    #(latency + 10) p = 1;
end
```

Intra-Assignment delays

Regular delays	Intra-Assignment delays
Delay to the left the statement	Delay to the right of the '=' operator
Defer RHS evaluation	Evaluate RHS just now
Defer the assignment	Defer the assignment of the computed value

Example 1

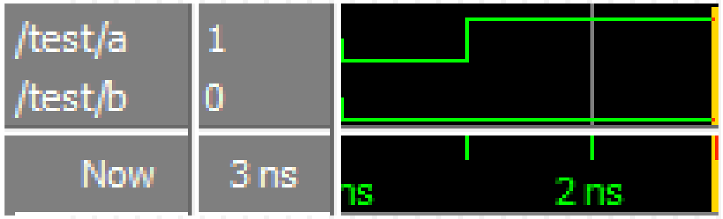
```
module testbench;
    reg a,b;

    initial
    begin
        a=1'b0;
        b=1'b0;
        #1 a=1'b1;
    end

    initial
    begin
        #2 b = a;
        $display($time,"b=%b",b);
    end
endmodule
```



2 b=1

Example 2

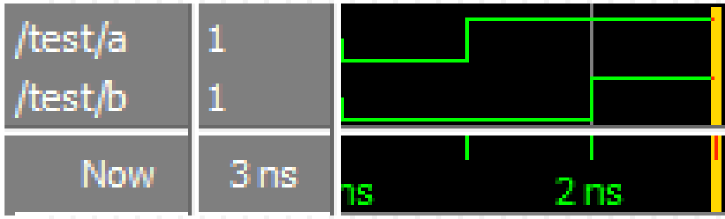
```
module testbench;
    reg a,b,c;

    initial
    begin
        a=1'b0;
        b=1'b0;
        #1 a=1'b1;
    end

    initial
    begin
        b = #2 a;
        $display($time,"b=%b",b);
    end
endmodule
```



2 b=0

Example 3

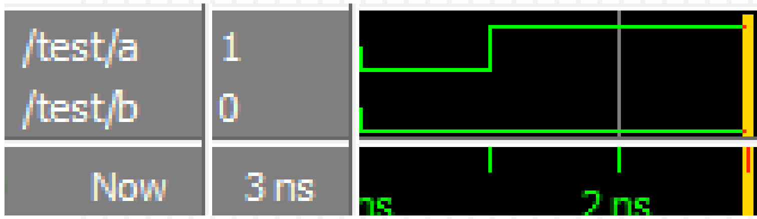
```
module testbench;
    reg a,b,c;

    initial
    begin
        a=1'b0;
        b=1'b0;
        #1 a=1'b1;
    end

    initial
    begin
        #2 b <= a;
        $display($time,"b=%b",b);
    end
endmodule
```



2 b=0

Example 4

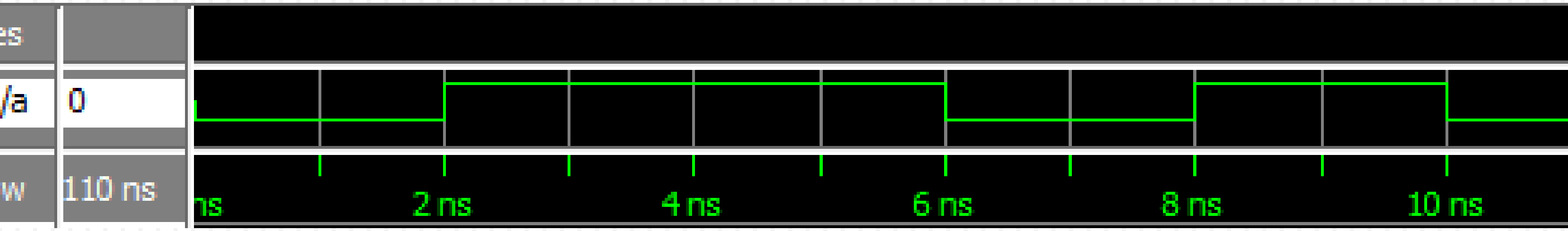
```
module testbench;
    reg a,b,c;

    initial
    begin
        a=1'b0;
        b=1'b0;
        #1 a=1'b1;
    end

    initial
    begin
        b <= #2 a;
        $display($time,"b=%b",b);
    end
endmodule
```



0 b=0

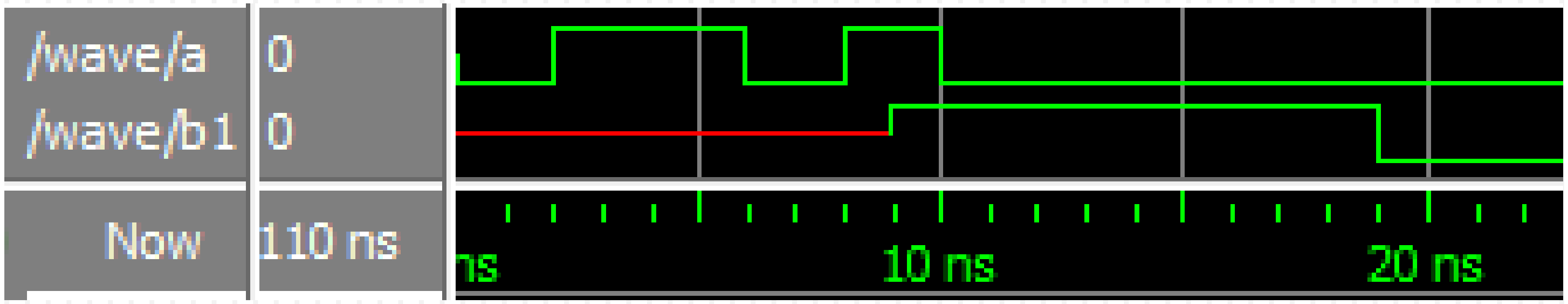
Example 5

```
reg a, b1, b2, b3, b4;
initial begin
    a = 1'b0;
    #2 a = 1'b1;
    #4 a = 1'b0;
    #2 a = 1'b1;
    #2 a = 1'b0;
    #100 $stop;
end
```



Example 5 (Cont.)

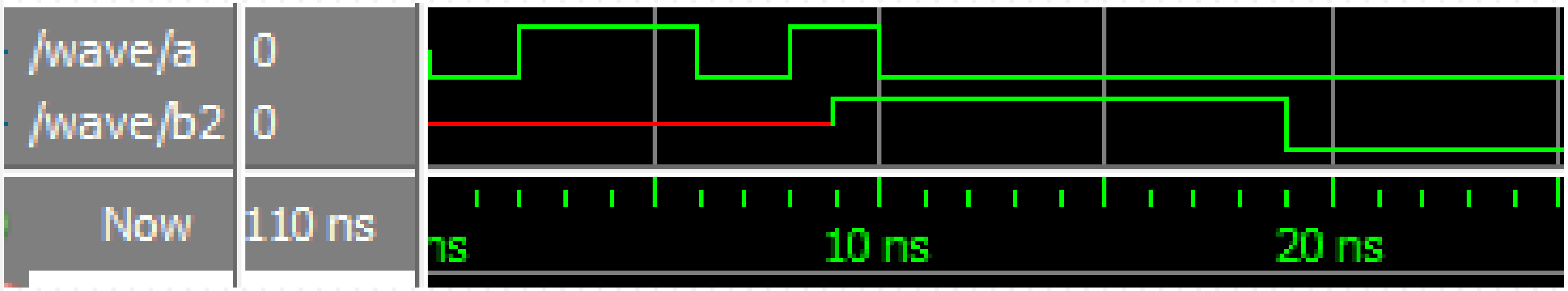
```
always @ (a)
      #9 b1 = a;
```



/wave/a	0
/wave/b1	0
Now	110 ns

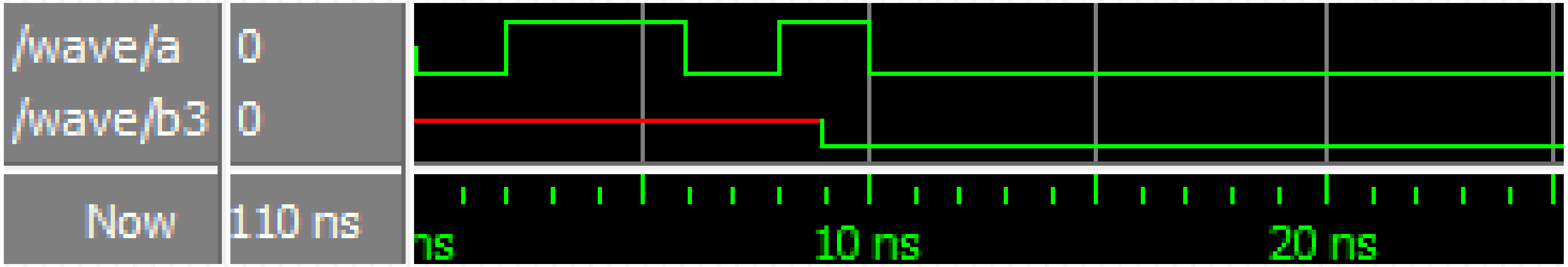
Example 5 (Cont.)

```
always @ (a)
        #9 b2 <= a;
```



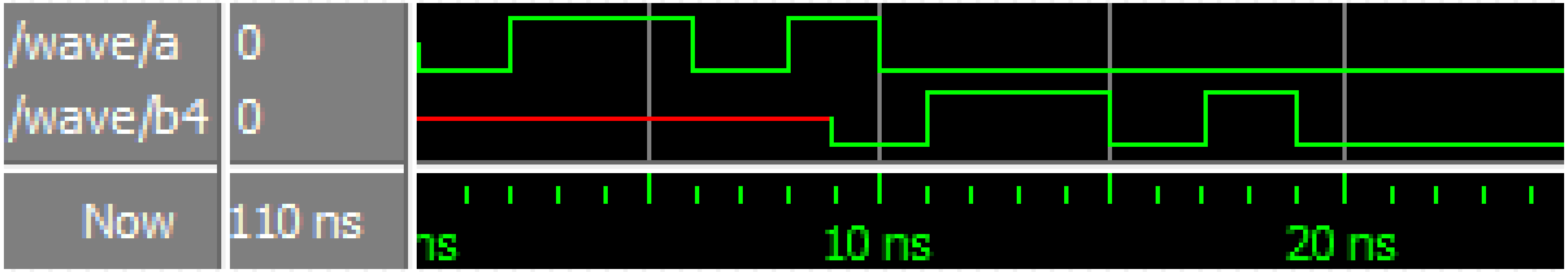
Example 5 (Cont.)

```
always @ (a)
        b3 = #9 a;
```



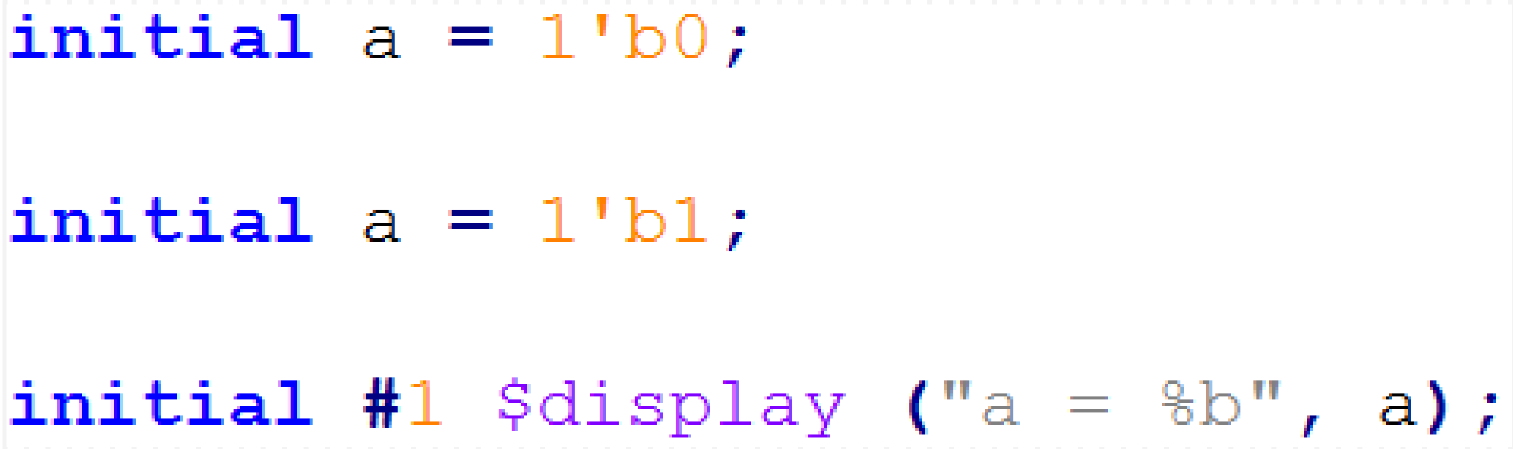
Example 5 (Cont.)

```
always @ (a)
        b4 <= #9 a;
```



Zero delay control

- The order of execution of statements in different *always*/*initial* blocks
  - Nondeterministic
    - Eg.



```
initial a = 1'b0;

initial a = 1'b1;

initial #1 $display ("a = %b", a);
```

Zero delay control (Cont'd.)

- The order of execution of statements in different *always*/*initial* blocks
  - Nondeterministic
- The statements after #0 are executed last
  - Eg.

```
initial #0 a = 1'b0;

initial a = 1'b1;

initial #1 $display ("a = %b", a);
```

Zero delay control (Cont'd.)

- The order of execution of statements in different *always*/*initial* blocks
  - Nondeterministic
- The statements after #0 are executed last
- Eliminates race conditions
- Multiple zero delay statements
  - the order between them is nondeterministic

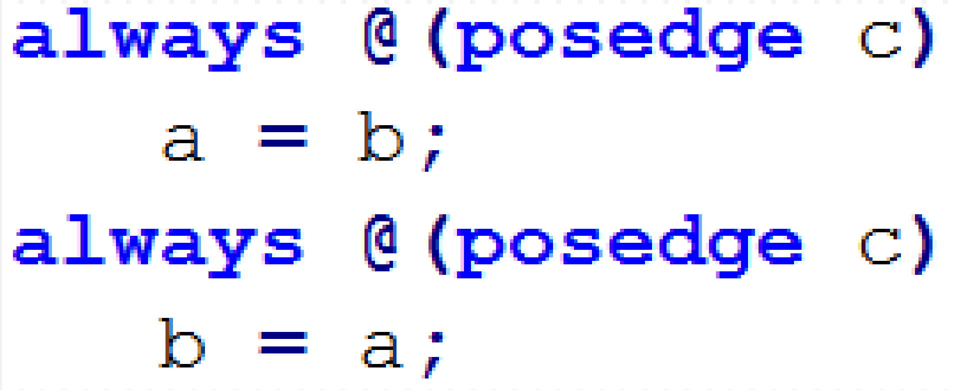
Event-Based Timing Control

- Regular event control
- Event OR Control
- Named event control
- Level-sensitive timing control

Regular event control

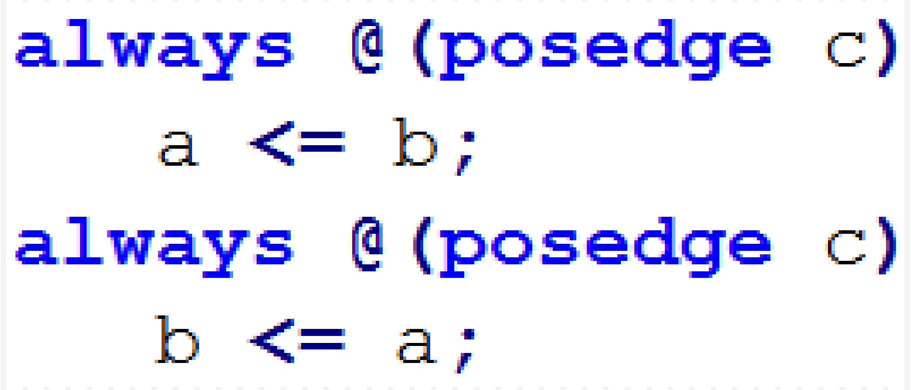
- Statements can be executed on
  - Changes in signal value
  - Positive edge
    - @ (posedge clock) q = d;
  - Negative edge
    - @ (negedge clock) q = d;
  - Both
    - q = @ (clock) d;

Example 1



```
always @(posedge c)
    a = b;
always @(posedge c)
    b = a;
```

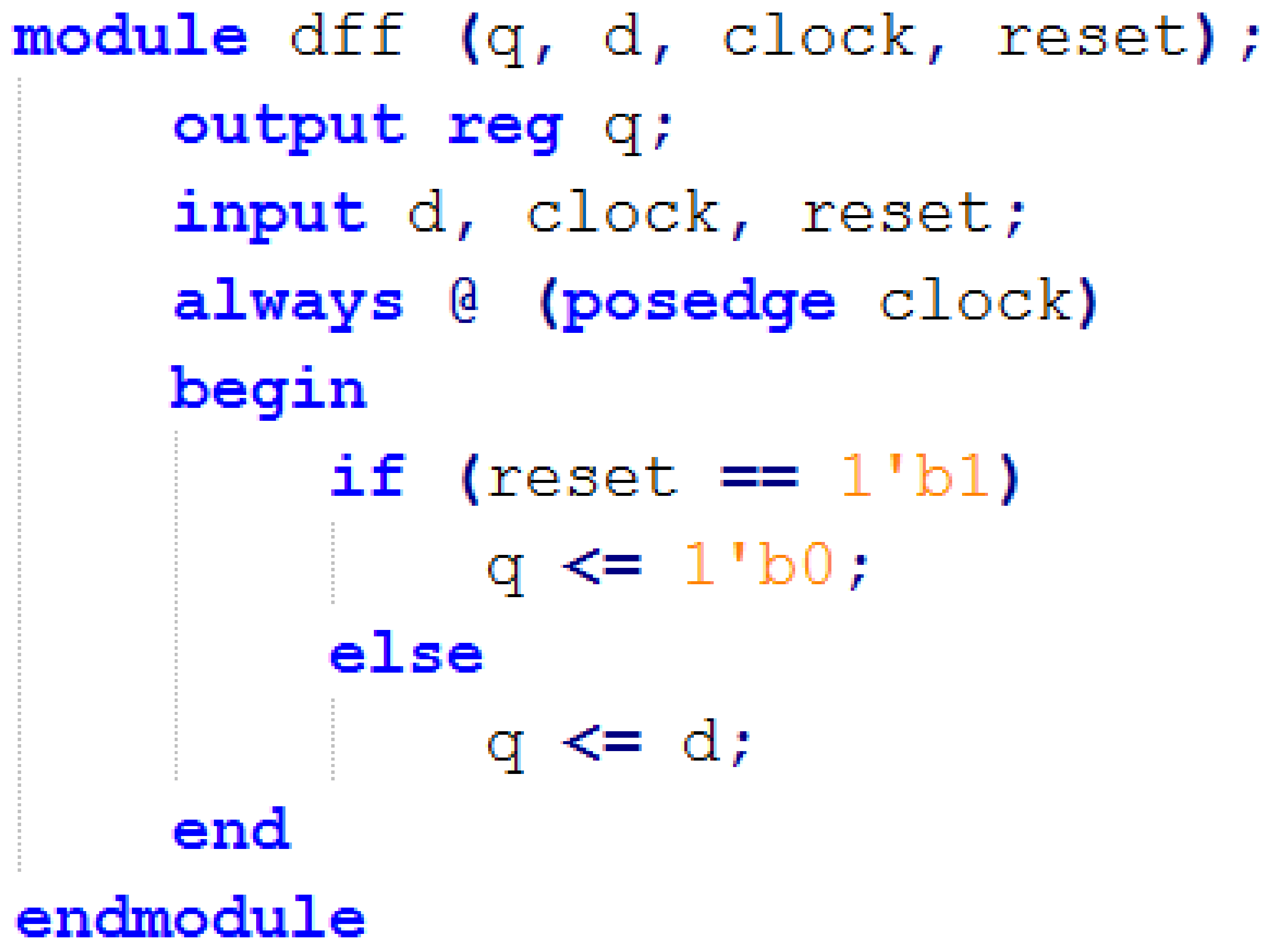
//Race condition



```
always @(posedge c)
    a <= b;
always @(posedge c)
    b <= a;
```

//Swap operation

Example 2



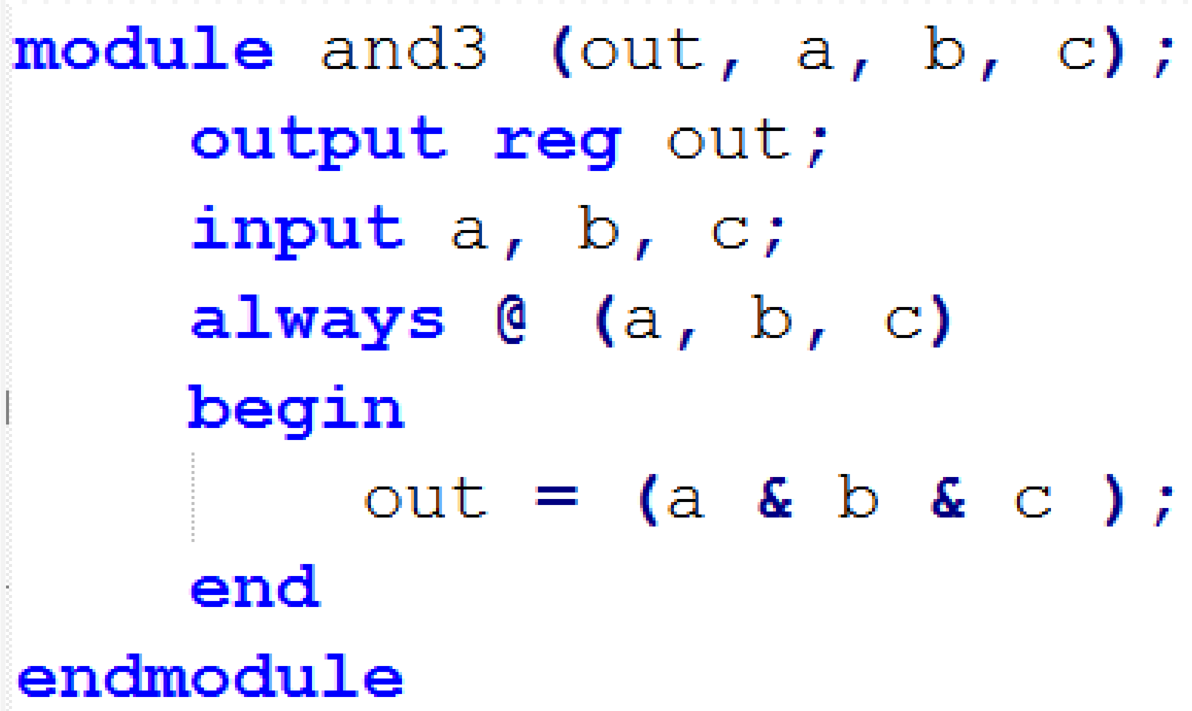
```
module dff (q, d, clock, reset);
    output reg q;
    input d, clock, reset;
    always @ (posedge clock)
    begin
        if (reset == 1'b1)
            q <= 1'b0;
        else
            q <= d;
    end
endmodule
```

Event OR Control

- OR of events
- Example
  - @ (posedge clock or negedge reset)
  - @ (posedge clock , posedge reset)
- The list is called the 'sensitivity list'

Event OR Control (Cont'd.)

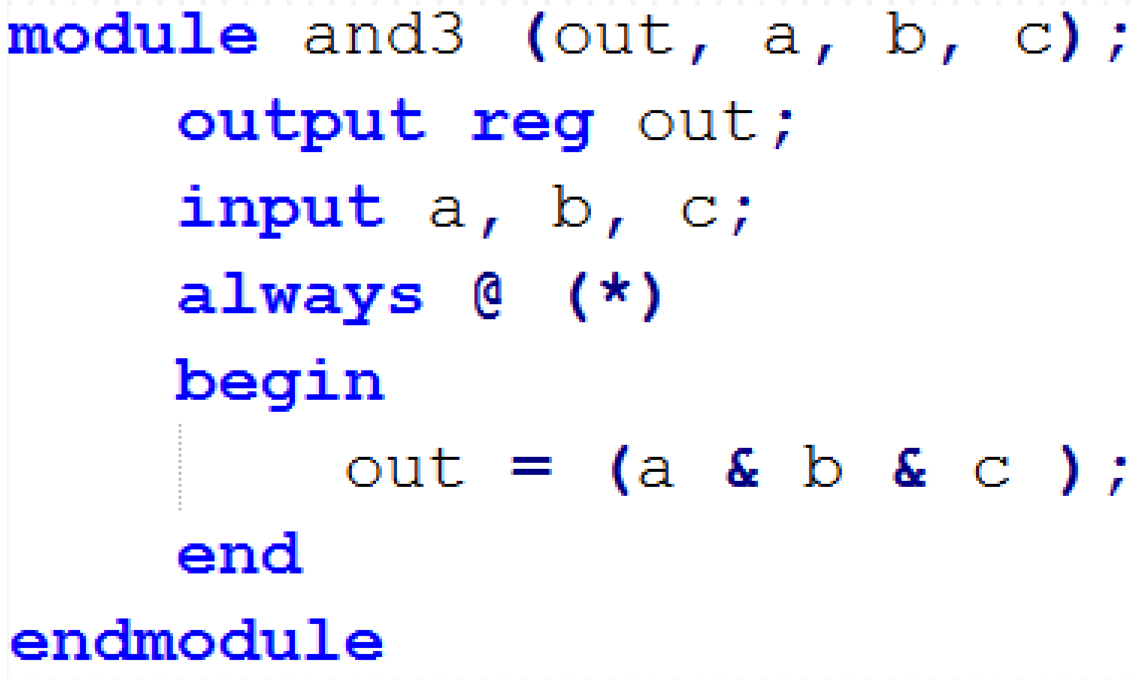
- To declare a combination logic block
  - @ (all RHS variables)
- Example



```
module and3 (out, a, b, c);
    output reg out;
    input a, b, c;
    always @ (a, b, c)
    begin
        out = (a & b & c );
    end
endmodule
```

Event OR Control (Cont'd.)

- @(*)
  - sensitive to a change on any signal that may be read by the statement group

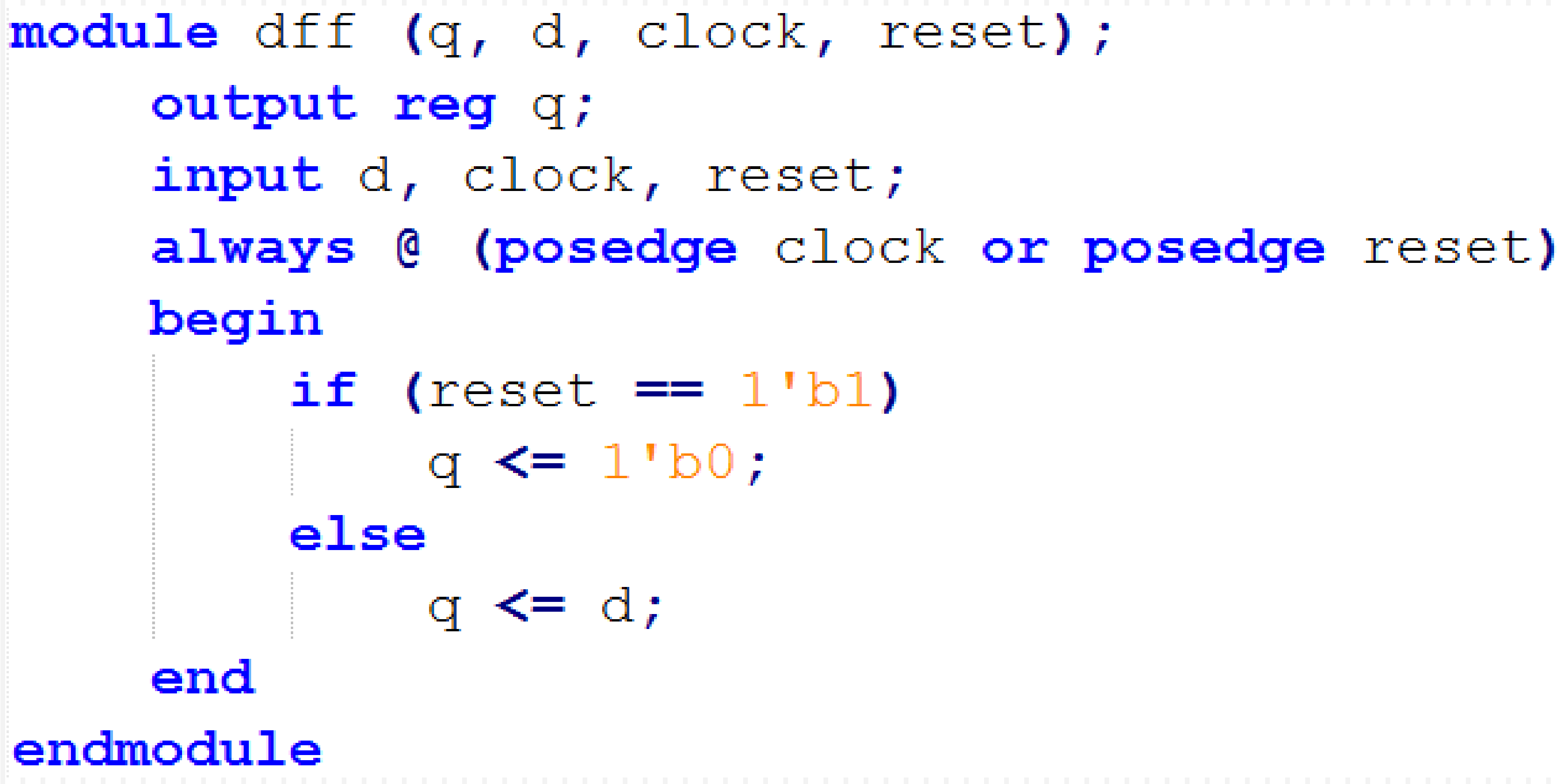


```
module and3 (out, a, b, c);
    output reg out;
    input a, b, c;
    always @ (*)
    begin
        out = (a & b & c );
    end
endmodule
```

Event OR Control (Cont'd.)

- To declare a positive-edge triggered sequential logic block
  - @ (posedge clock)
  - With async. reset
    - Active low
      - @ (posedge clock or negedge reset)
    - Active high
      - @ (posedge clock or posedge reset)

Example



```
module dff (q, d, clock, reset);
    output reg q;
    input d, clock, reset;
    always @ (posedge clock or posedge reset)
    begin
        if (reset == 1'b1)
            q <= 1'b0;
        else
            q <= d;
    end
endmodule
```

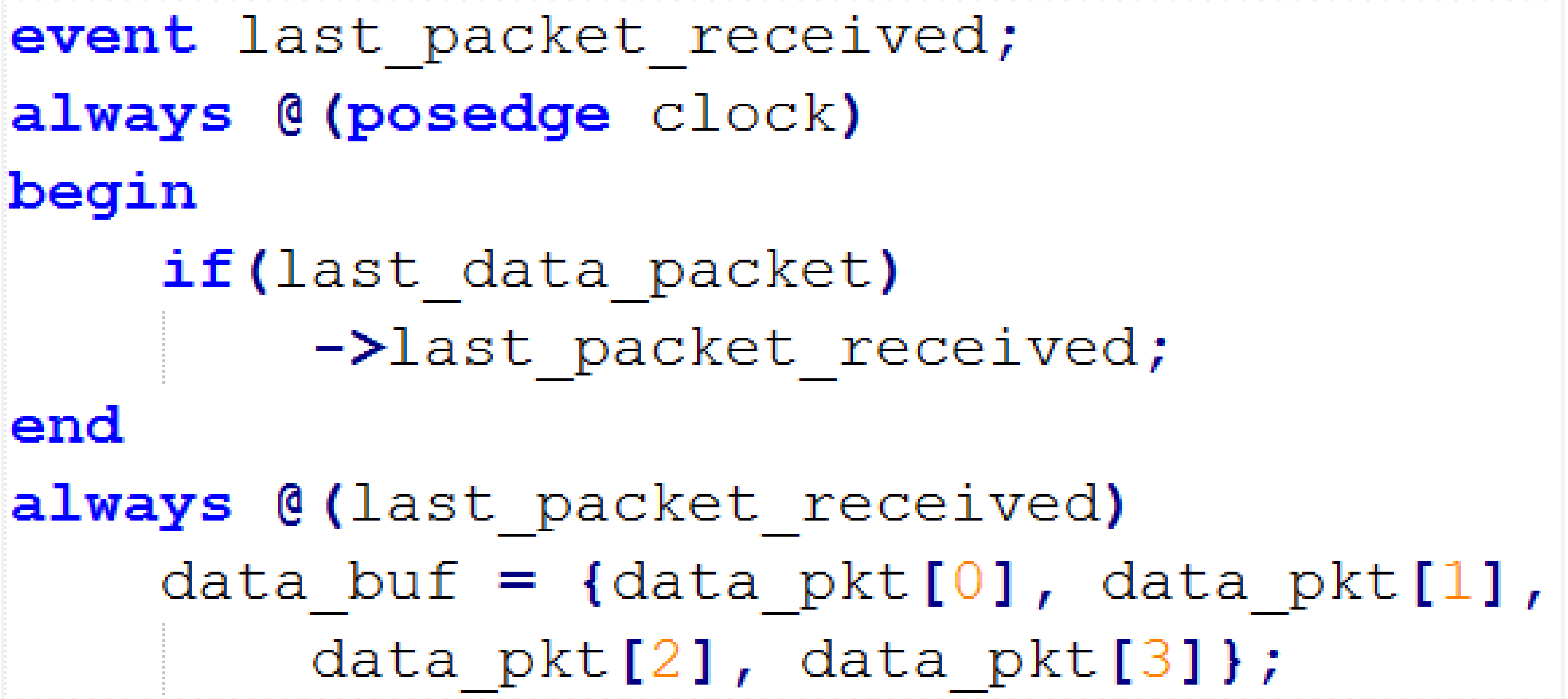
if else to be covered a bit later

Named event control

- Declare
  - event last_packet_received;
- Trigger
  - -> last_packet_received;
- recognize the occurrence
  - @ (last_packet_received)

Example

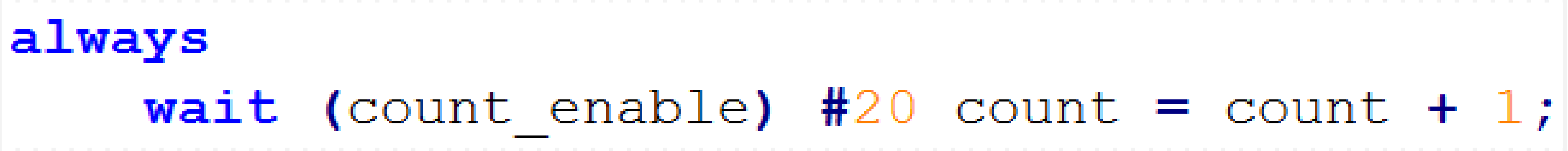
- Storing data after receiving last packet



```
event last_packet_received;
always @(posedge clock)
begin
    if(last_data_packet)
        ->last_packet_received;
end
always @(last_packet_received)
    data_buf = {data_pkt[0], data_pkt[1],
        data_pkt[2], data_pkt[3]};
```

Level-Sensitive Timing Control

- The ability to wait for a certain condition
- Keyword: wait
- Example



```
always
    wait (count_enable) #20 count = count + 1;
```

Level-Sensitive Timing Control (Cont'd.)

- Example:

```
always
    wait (count_enable) #20 count = count + 1;
```

- the value of *count_enable is monitored continuously*
  - *count_enable = 0* $\longrightarrow$ *the statement is not entered*
  - *count_enable = 1* $\longrightarrow$ *count = count + 1 is executed after 20 time units*
  - *count_enable stays at 1* $\longrightarrow$ *count will be incremented every 20 time units*

Conditional Statements

- Keywords: if and else
- Syntax: similar to C
  - Example:

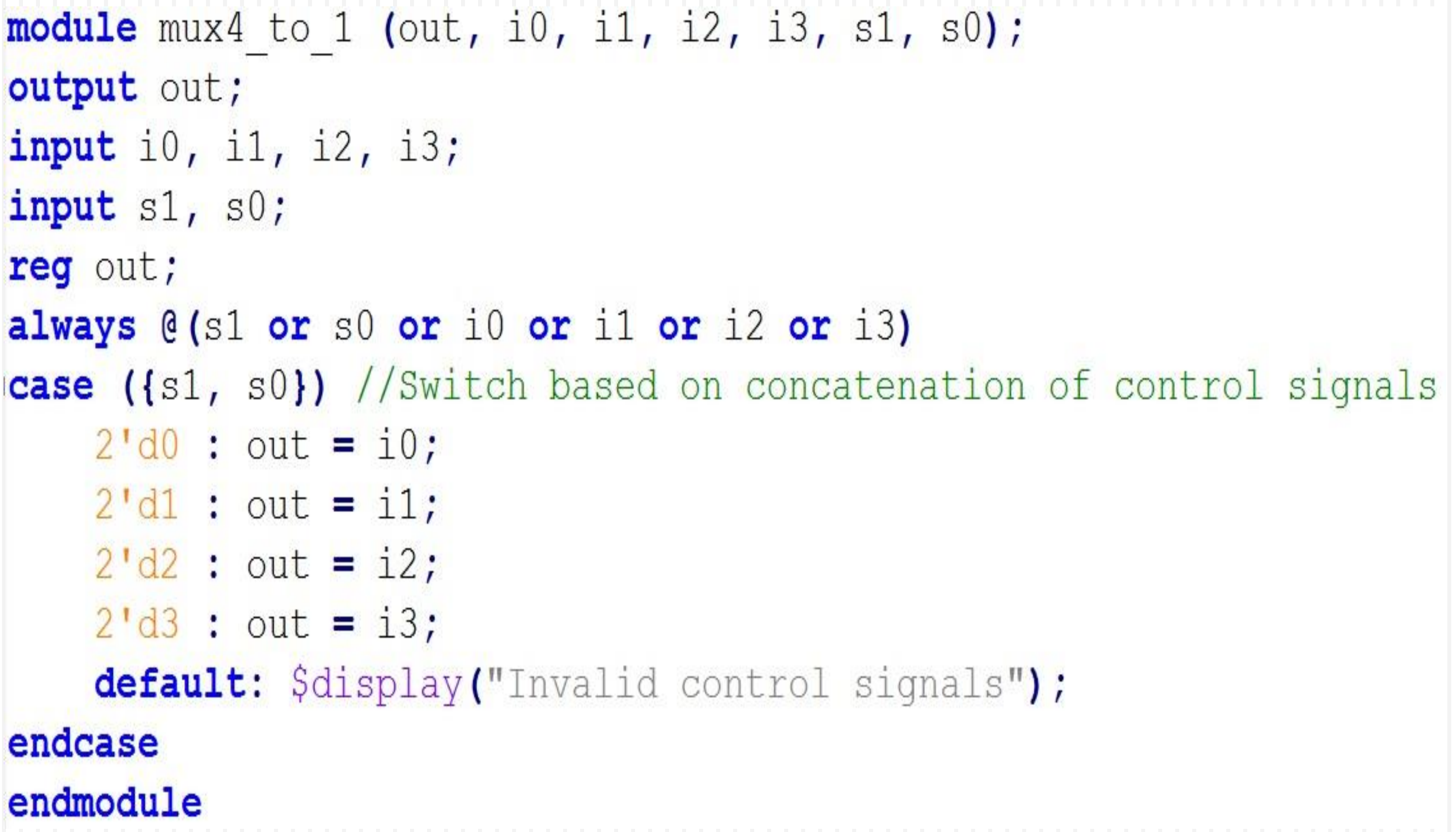
```
 if (<expression1>) true_statement1 ;
else if (<expression2>) true_statement2 ;
else if (<expression3>) true_statement3 ;
else default_statement ;
```

- *<expression>*
  - True (1 or a non-zero value)
  - False (zero) or ambiguous (x)

Multiway Branching

- Case Statement
  - Keywords: case, endcase, and default
  - Syntax: similar to C
- The case statements can be nested
- Act like a many-to-one multiplexer

Multiway Branching (Cont'd.)

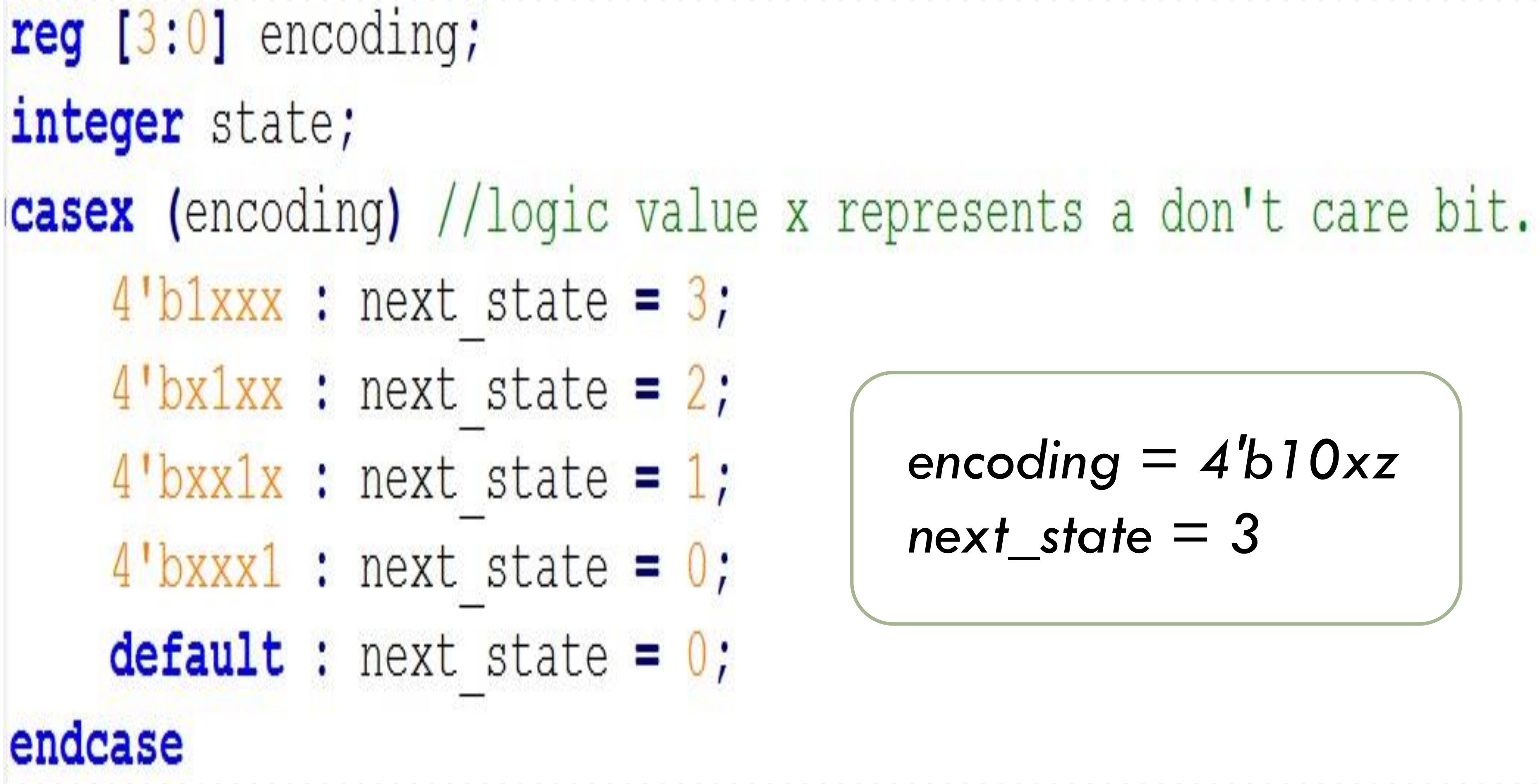


```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0}) //Switch based on concatenation of control signals
    2'd0 : out = i0;
    2'd1 : out = i1;
    2'd2 : out = i2;
    2'd3 : out = i3;
    default: $display("Invalid control signals");
endcase
endmodule
```

Multiway Branching (Cont'd.)

- Keywords: casex and casez
- Casez: treats all z values as don't cares
  - z can also be represented by ?
- Casex: treats all x and z values as don't cares
- The use of casex and casez allows comparison of only non-x or -z positions

Multiway Branching (Cont'd.)



```
reg [3:0] encoding;
integer state;
casex (encoding) //logic value x represents a don't care bit.
     4'b1xxx : next_state = 3;
     4'bx1xx : next_state = 2;
     4'bxx1x : next_state = 1;
     4'bxxx1 : next_state = 0;
     default : next_state = 0;
endcase
```

encoding = 4'b10xz

next_state = 3

Loops

- Loops:
  - *While*
  - *For*
  - *Repeat*
  - *Forever*
- Syntax: similar to C
- All looping statements can appear only inside an initial or always block

While Loop

- Loop executes until the while-expression is not true
- Example
  - Find the first bit with a value 1 in flag (vector variable)

```
'define TRUE 1'b1';
'define FALSE 1'b0;
reg [15:0] flag;
integer i; //integer to keep count
reg continue;

initial
begin
   flag = 16'b 0010_0000_0000_0000;
   i = 0;
   continue = 'TRUE;

   while((i < 16) && continue )
   begin
      if (flag[i])
      begin
         $display("TRUE bit at element number %d", i);
         continue = 'FALSE;
      end
      i = i + 1;
   end
end
```

For Loop

- An initial condition
- A check to see if the terminating condition is true
- A procedural assignment to change value of the control variable
- Example
  - Initialize array elements

```
'define MAX_STATES 32

//Integer array state with elements 0:31
integer state [0: 'MAX_STATES-1];
integer i;

initial
begin
    //initialize all even locations with 0
    for(i = 0; i < 32; i = i + 2)
        state[i] = 0;

    //initialize all odd locations with 1
    for(i = 1; i < 32; i = i + 2)
        state[i] = 1;
end
```

Repeat Loop

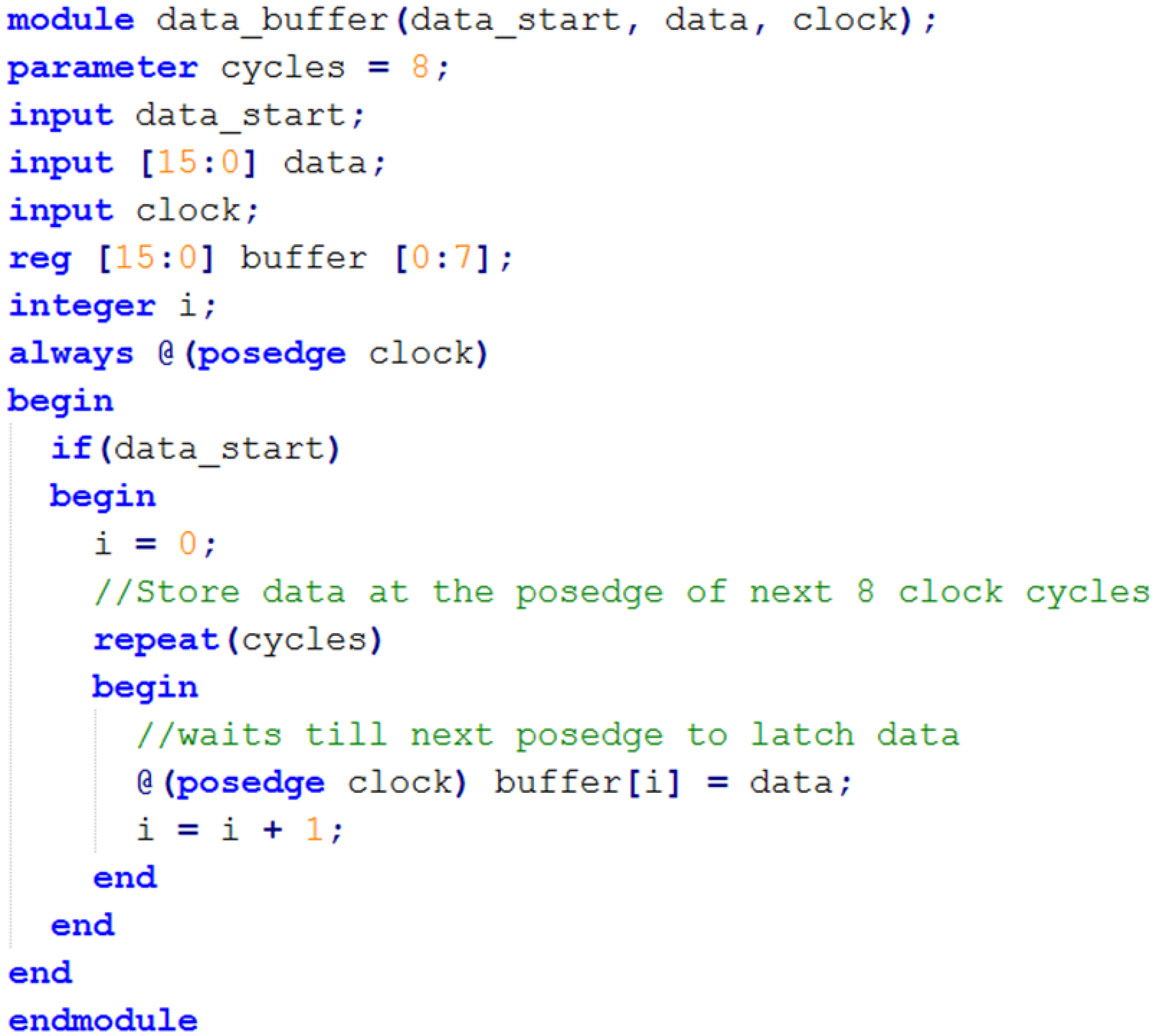
- Executes the loop a fixed number of times
  - Constant
  - Variable or signal value
    - Evaluated when the loop starts
    - Changes during loop execution ignored
- Example 1
  - Increment and display count from 0 to 127

Repeat Loop (Cont’d.)

```
integer count;
initial
begin
    count = 0;
    repeat(128) //a number, a variable or a signal value
    begin
        $display("Count = %d", count);
        count = count + 1;
    end
end
```

Repeat Loop (Cont'd.)

- Example 2
  - Data buffer module example
    - After it receives a data_start signal
    - Reads data for next 8 cycles



```
module data_buffer(data_start, data, clock);
parameter cycles = 8;
input data_start;
input [15:0] data;
input clock;
reg [15:0] buffer [0:7];
integer i;
always @(posedge clock)
begin
  if(data_start)
  begin
    i = 0;
    //Store data at the posedge of next 8 clock cycles
    repeat(cycles)
    begin
      //waits till next posedge to latch data
      @(posedge clock) buffer[i] = data;
      i = i + 1;
    end
  end
end
endmodule
```

Forever loop

- Does not contain any expression and executes forever until the $finish task is encountered
- Forever loop equivalent to *while (1)*
- Example:
  - Clock generation:

    ```
    forever #10 clock = ~clock;
    ```
  - Synchronize two register values:

    ```
    forever @(posedge clock) x <= y;
    ```

Sequential and Parallel Blocks

- Block types:
  - *Sequential blocks*
  - *Parallel blocks*
- Features of blocks:
  - *Named blocks*
  - *Disabling named blocks*
  - *Nested blocks*

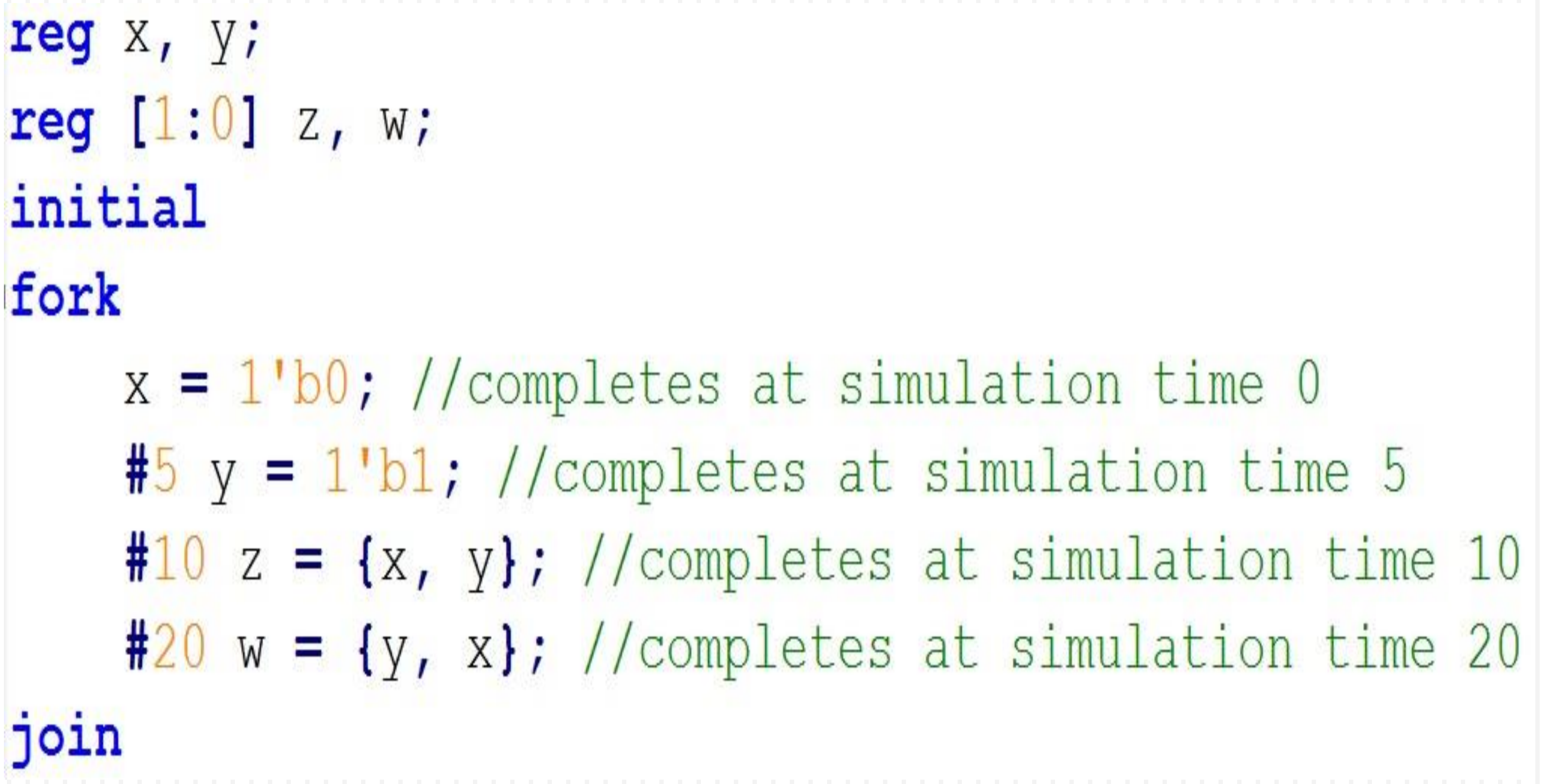
Sequential and Parallel Blocks (Cont'd.)

- Sequential blocks:
  - Keywords: begin and end
  - A statement is executed only after its preceding statement completes execution
- Parallel blocks:
  - Keywords: fork and join

Parallel blocks

- Statements are executed concurrently
- Ordering of statements is controlled by the delay or event control
- If delay or event control is specified, it is relative to the time the block was entered

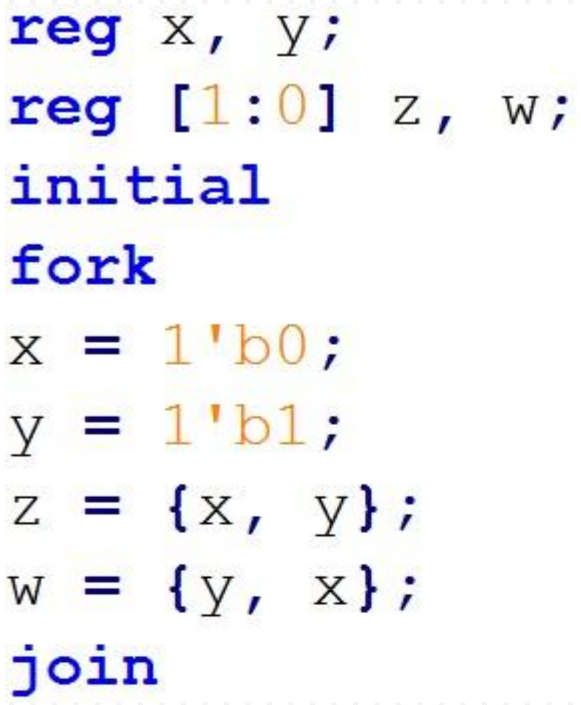
Parallel blocks (Cont'd.)



```
reg x, y;
reg [1:0] z, w;
initial
fork
    x = 1'b0; //completes at simulation time 0
    #5 y = 1'b1; //completes at simulation time 5
    #10 z = {x, y}; //completes at simulation time 10
    #20 w = {y, x}; //completes at simulation time 20
join
```

Parallel blocks (Cont’d.)

- Race conditions:



```
reg x, y;
reg [1:0] z, w;
initial
fork
x = 1'b0;
y = 1'b1;
z = {x, y};
w = {y, x};
join
```

- What is the difference between this and non-blocking assignment?

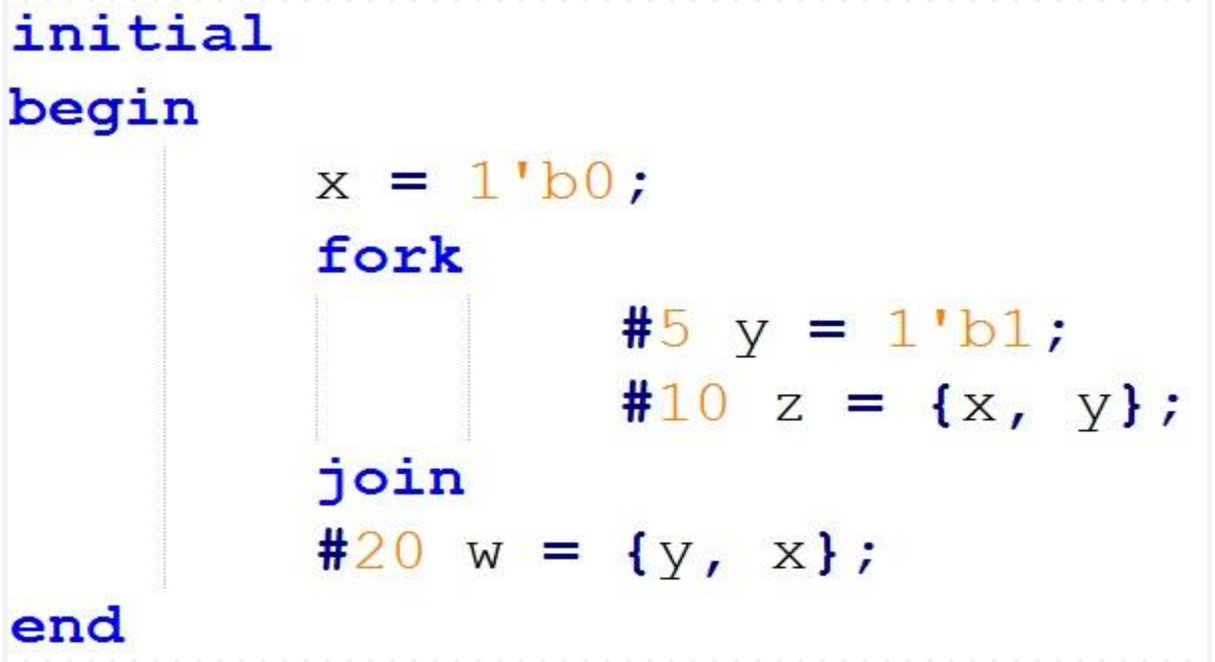
- All statements start at simulation time 0
- Two of the possible outcomes
  - If *x = 1'b0 and y = 1'b1 execute first*
    - *z* = *1* , w = 2
  - If *x = 1'b0 and y = 1'b1 execute last*
    - *z = w = 2'bxx*

Parallel blocks (Cont'd.)

- All statements in the fork-join block are executed at once
- CPUs running simulations can execute only one statement at a time
- Fork: splitting a single flow into independent flows
- Join: joining the independent flows back into a single flow

Special Features of Blocks

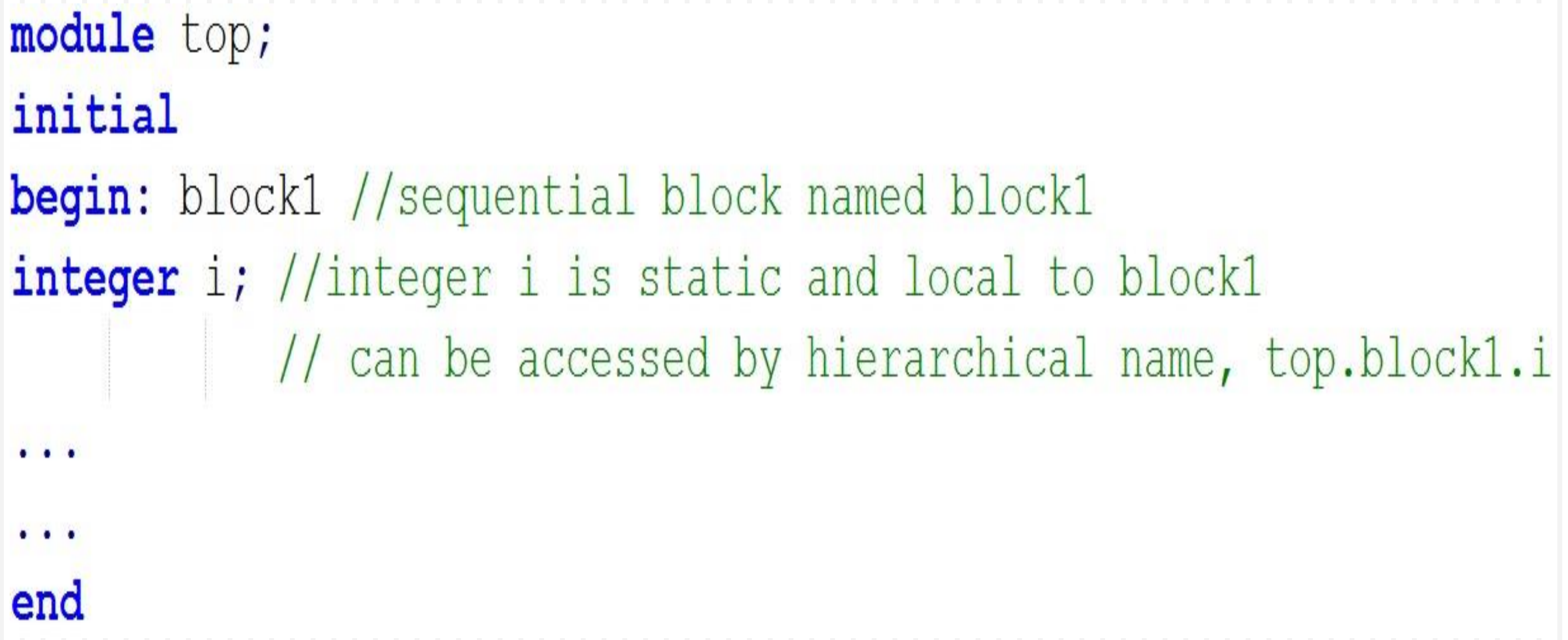
- Nested blocks
  - Sequential and parallel blocks can be mixed
  - Example:



```
initial
begin
        x = 1'b0;
        fork
                #5 y = 1'b1;
                #10 z = {x, y};
        join
        #20 w = {y, x};
end
```

Special Features of Blocks (Cont'd.)

- Named blocks
  - Example



```
module top;
initial
begin: block1 //sequential block named block1
integer i; //integer i is static and local to block1
           // can be accessed by hierarchical name, top.block1.i
...
...
end
```

Special Features of Blocks (Cont'd.)

- Disabling named blocks:

  Disabling a block causes the execution control to be passed to the statement immediately succeeding the block

  - Keyword: disable
  - Used to get out of loops
  - Handle error conditions

Special Features of Blocks (Cont'd.)

- Break statement in C vs. keyword disable
  - A break statement can break the current loop only
  - The keyword disable allows disabling of any named block in the design

```
reg [15:0] flag;
integer i; //integer to keep count
initial
begin
  flag = 16'b 0010_0000_0000_0000;
  i = 0;
  begin: block1 //The main block inside while is named block1
  while(i < 16)
   begin
      if (flag[i])
      begin
          $display("Encountered a TRUE bit at element number %d", i);
          disable block1; //disable block1 because you found true bit.
      end
      i = i + 1;
   end
  end
end
```

Generate Blocks

- Dynamic generation
  - At elaboration time
  - Before the simulation begins
- Keywords
  - generate
  - endgenerate

Generate Blocks (Cont'd.)

- What can be instantiated
  - Variable declarations
  - Modules
  - User defined primitives
  - Gate primitives
  - Continuous assignments
  - initial
  - always

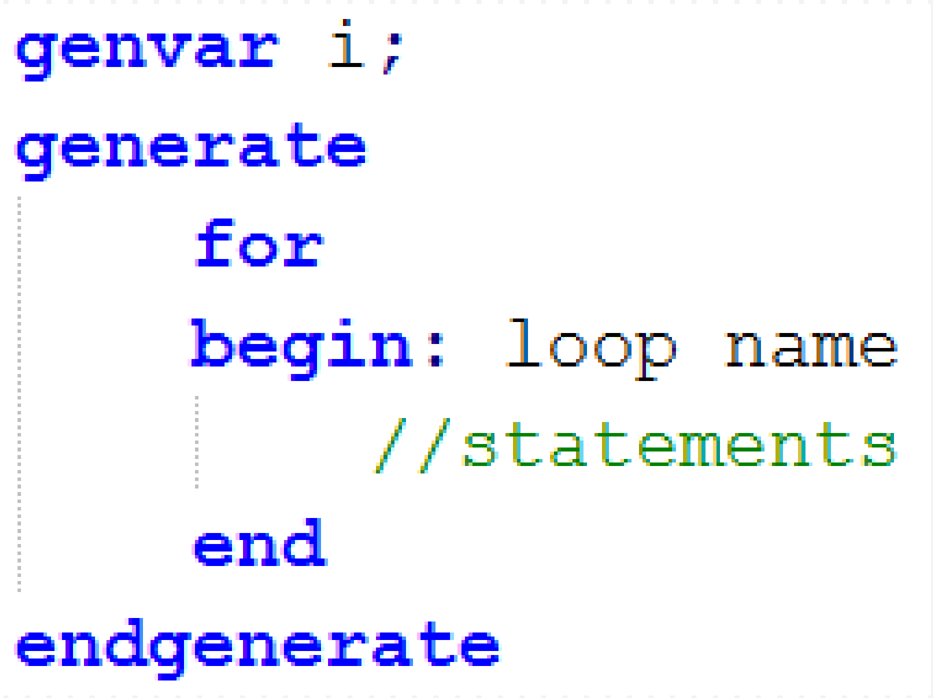
Generate Blocks (Cont'd.)

- Allowed in the generate scope
  - Variable declaration
  - Parameter redefinition
  - tasks and functions
    - Covered later
- Not permitted in a generate statement
  - parameters, local parameters declarations
  - input, output, inout declarations
  - Specify blocks

Generate Blocks (Cont'd.)

- Particularly convenient when
  - The same operation or module instance is repeated
    - generate loop
  - Certain code is conditionally included based on parameter definitions
    - generate conditional
    - generate case

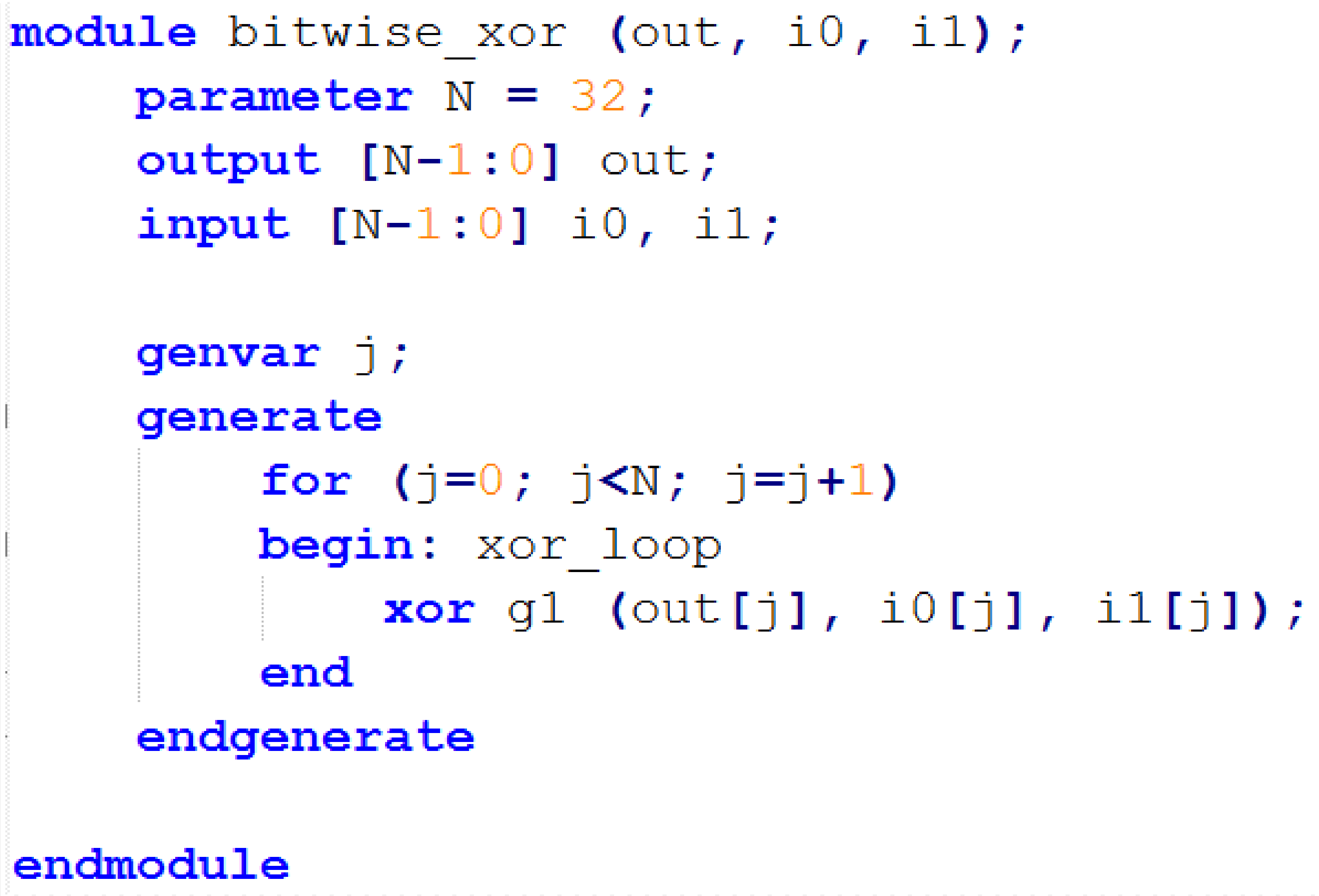
Generate Loop



```
genvar i;
generate
      for
      begin: loop name
            //statements
      end
endgenerate
```

- genvar can be defined only in a generate loop
- Generate loops can be nested

Example 1

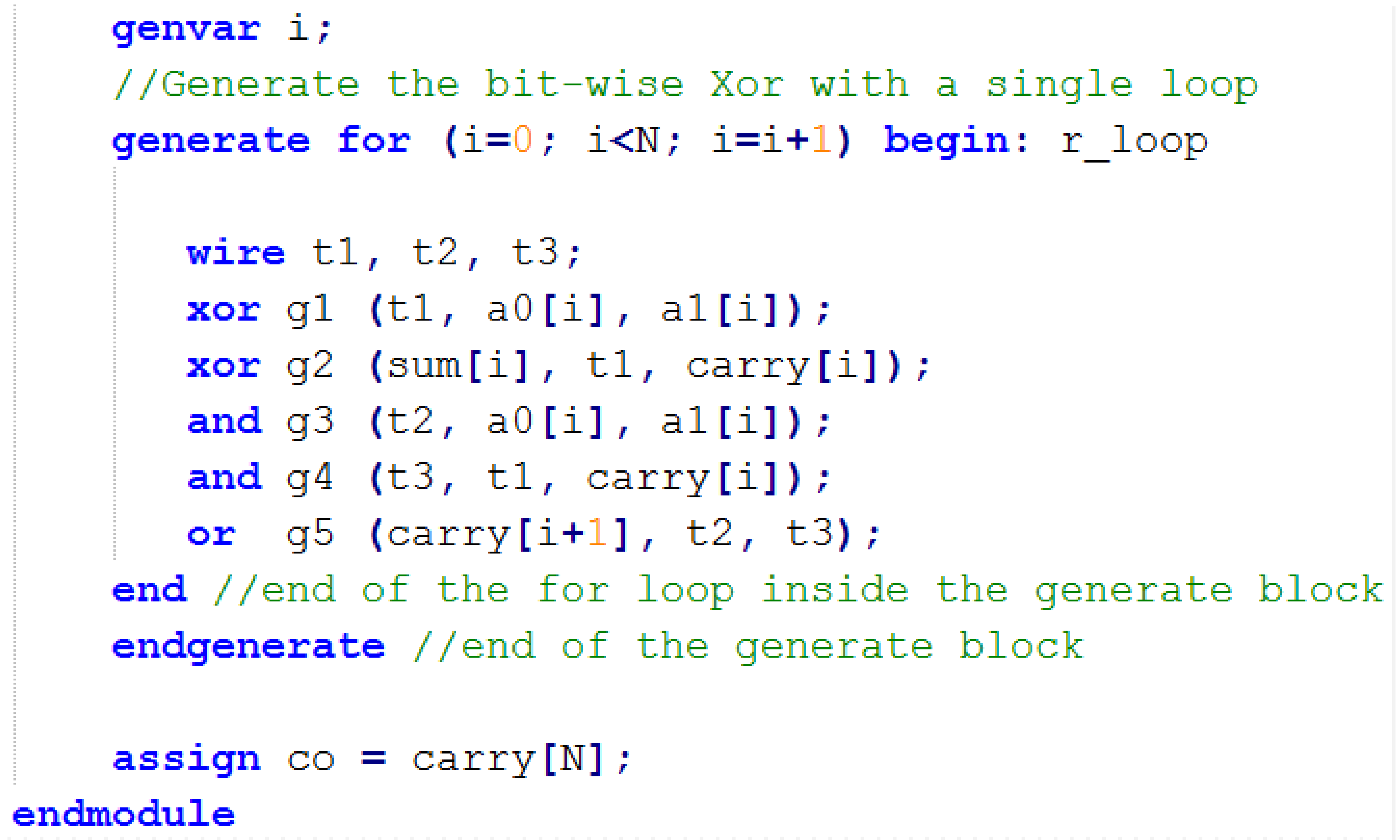


```
module bitwise_xor (out, i0, i1);
    parameter N = 32;
    output [N-1:0] out;
    input [N-1:0] i0, i1;

    genvar j;
    generate
        for (j=0; j<N; j=j+1)
        begin: xor_loop
            xor g1 (out[j], i0[j], i1[j]);
        end
    endgenerate

endmodule
```

Example 2

```
module ripple_adder(co, sum, a0, a1, ci);
parameter N = 4; // 4-bit bus by default

// Port declarations
output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;

//Local wire declaration
wire [N  :0] carry;

//Assign 0th bit of carry equal to carry input
assign carry[0] = ci;
```

Example 2 (Cont'd.)



```
  genvar i;
  //Generate the bit-wise Xor with a single loop
  generate for (i=0; i<N; i=i+1) begin: r_loop

     wire t1, t2, t3;
     xor g1 (t1, a0[i], a1[i]);
     xor g2 (sum[i], t1, carry[i]);
     and g3 (t2, a0[i], a1[i]);
     and g4 (t3, t1, carry[i]);
     or  g5 (carry[i+1], t2, t3);
  end //end of the for loop inside the generate block
  endgenerate //end of the generate block

  assign co = carry[N];
endmodule
```

Example 2 (Cont'd.)

- genvar i
  - A temporary loop variable
  - Used only in the evaluation of the generate block
  - Does not exist during the simulation
    - generate loops are unrolled before simulation

Example 2 Hierarchical instance names

Xor	And	Or
r_loop[0].g1	r_loop[0].g3	r_loop[0].g5
r_loop[1].g1	r_loop[1].g3	r_loop[1].g5
r_loop[2].g1	r_loop[2].g3	r_loop[2].g5
r_loop[3].g1	r_loop[3].g3	r_loop[3].g5
r_loop[0].g2	r_loop[0].g4	
r_loop[1].g2	r_loop[1].g4	
r_loop[2].g2	r_loop[2].g4	
r_loop[3].g2	r_loop[3].g4	

Example 2 Hierarchical instance names (Cont'd.)

- Generated instances are connected with the following generated nets

r_loop[0].t1 r_loop[1].t1 r_loop[2].t1 r_loop[3].t1	r_loop[0].t2 r_loop[1].t2 r_loop[2].t2 r_loop[3].t2	r_loop[0].t3 r_loop[1].t3 r_loop[2].t3 r_loop[3].t3

Generate Conditional

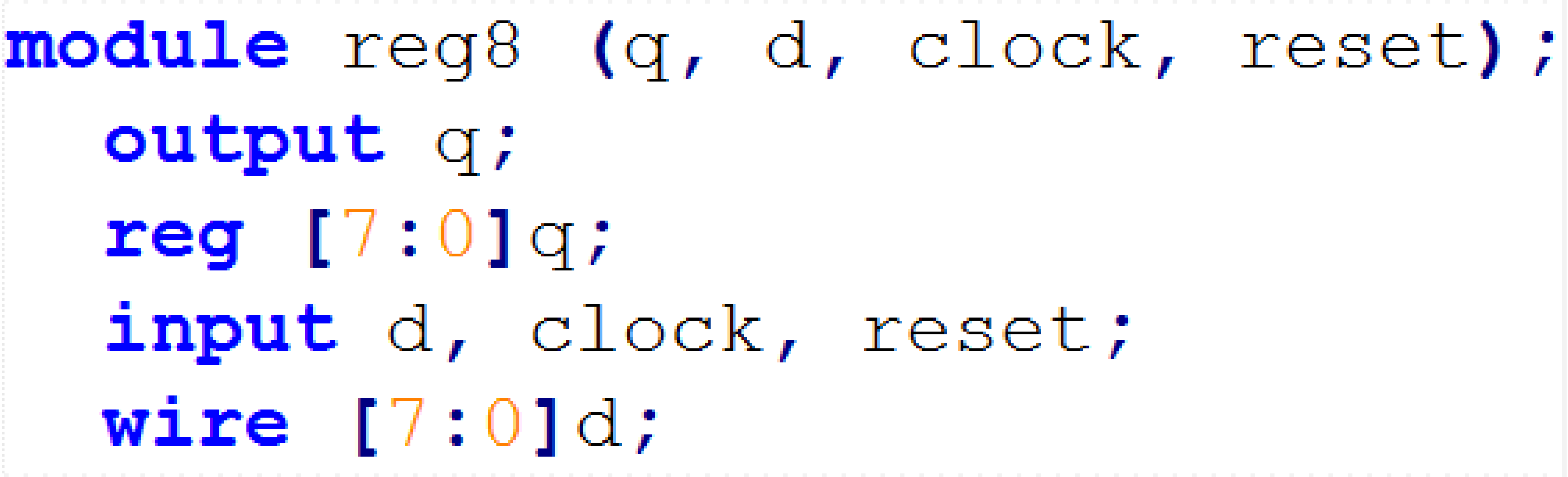
```
module multiplier (product, a0, a1);
    parameter a0_width = 8;
    parameter a1_width = 8;
    localparam product_width = a0_width + a1_width;

    output [product_width -1:0] product;
    input [a0_width-1:0] a0;
    input [a1_width-1:0] a1;

    generate
        if (a0_width <8) || (a1_width < 8)
            cla_multiplier #(a0_width, a1_width)
                m0 (product, a0, a1);
        else
            tree_multiplier #(a0_width, a1_width)
                m0 (product, a0, a1);
    endgenerate
endmodule
```

Generate Case

```
module adder(co, sum, a0, a1, ci);
    parameter N = 4;
    output [N-1:0] sum;
    output co;
    input [N-1:0] a0, a1;
    input ci;
    generate
        case (N)
            1: adder_1bit adder1(c0, sum, a0, a1, ci);
            2: adder_2bit adder2(c0, sum, a0, a1, ci);
            default: adder_cla #(N)
                adder3(c0, sum, a0, a1, ci);
        endcase
    endgenerate
endmodule
```

Example

- 8-bit Register
  - always in generate block



```
module reg8 (q, d, clock, reset);
   output q;
   reg [7:0]q;
   input d, clock, reset;
   wire [7:0]d;
```

Example (Cont'd.)

```
   genvar i;
   generate
      for (i = 0; i < 7; i = i + 1)
      begin
            always @(posedge clock, posedge reset)
            begin
               if (reset == 1'b1)
                  q[i] <= 1'b0;
               else
                  q[i] <= d[i];
            end
      end
   endgenerate
endmodule
```