

# Verilog

# **Verilog Assignments and Blocks: always**

# Verilog “always” statement

- ❑ always block is a **procedural** assignment block in Verilog
- ❑ The code inside always block is executed **sequentially**
- ❑ **All always** blocks are **parallel** and start at time point 0
- ❑ always block is **event-dependent**, it executes at some event

always @ (*sensitivity\_list*)  
statement

always @ (*sensitivity\_list*) begin  
statements  
end

Some event

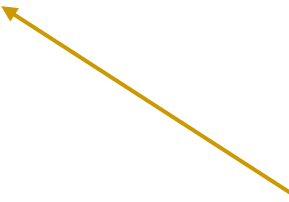
In case of multiple  
statements “begin” / “end”  
construct is needed

# Verilog “always” statement: sensitivity list

- ❑ Sensitivity list is the expression that defines when will the always block be executed, it's a list of signals
- ❑ It contains the signals(one, or multiple) whose change will trigger the execution of the statements inside the always block

```
always @ (a or b) begin  
    statements  
end
```

```
always @ (*) begin  
    statements  
end
```



The “statements” will execute whenever a or b change their values

# Verilog “always” statement: sensitivity list

---

- ❑ Sensitivity list is the expression that defines when will the always block be executed, it's a list of signals
- ❑ It contains the signals(one, or multiple) whose change will trigger the execution of the statements inside the always block

```
always @ (a or b) begin  
    statements  
end
```

```
always @ (*) begin  
    statements  
end
```

The “statements” will execute whenever any signal changes its value

# Verilog “always” statement: operation

- ❑ always can be used to model **combinational** and **sequential** logic
  - A **flip-flop** gets triggered whenever there is an active edge of the provided **clock**
  - Any **combinational** logic activates whenever there is an **input signal change**
- ❑ The blocks of code work concurrently, the connection/shared variables/nets/regs determine the flow of the data
- ❑ always block is a **continuous process** which is getting triggered by an event and performs the steps in it sequentially

`always @ (negedge clk)`  
statement

Whenever there is a negative edge on the clk signal, the statement is executed

# Verilog “always” statement: combinational logic

---

```
module test (  
  input in1, in2, in3, in4,  
  output out  
);  
  wire a, b;  
  assign a = in1 & in2;  
  assign b = in3 ^ in4;  
  assign out = a | b;  
endmodule
```

```
module test (  
  input in1, in2, in3, in4,  
  output reg out  
);  
  reg a, b;  
  always @ (in1, in2, in3, in4) begin  
    a = in1 & in2;  
    b = in3 ^ in4;  
    out = a | b;  
  
  end  
endmodule
```

# Verilog “always” statement: combinational logic

```
module test (  
  input in1, in2, in3, in4,  
  output out  
);  
  wire a, b;  
  assign a = in1 & in2;  
  assign b = in3 ^ in4;  
  assign out = a | b;  
endmodule
```

```
module test (  
  input in1, in2, in3, in4,  
  output reg out  
);  
  reg a, b;  
  always @ (in1, in2, in3, in4) begin  
    a = in1 & in2;  
    b = in3 ^ in4;  
    out = a | b;  
  end  
endmodule
```

All signals used in a **procedural block** that have a value assigned to them **should be of a reg type**



# Verilog “always” statement: combinational logic

```
module test (  
  input in1, in2, in3, in4,  
  output out  
);  
  wire a, b;  
  assign a = in1 & in2;  
  assign b = in3 ^ in4;  
  assign out = a | b;  
endmodule
```

```
module test (  
  input in1, in2, in3, in4,  
  output out  
);  
  reg a, b, out_reg;  
  always @ (in1, in2, in3, in4) begin  
    a = in1 & in2;  
    b = in3 ^ in4;  
    out_reg = a | b;  
  
  end  
  assign out = out_reg;  
endmodule
```

All signals used in a **procedural block** that have a value assigned to them **should be of a reg type**

# Verilog “always” statement: combinational logic

```
module test (  
  input in1, in2, in3, in4,  
  output out  
);  
  wire a, b;  
  assign a = in1 & in2;  
  assign b = in3 ^ in4;  
  assign out = a | b;  
endmodule
```

```
module test (  
  input in1, in2, in3, in4,  
  output out  
);  
  reg a, b, out_reg;  
  always @ (in1, in2, in3, in4) begin  
    a = in1 & in2;  
    b = in3 ^ in4;  
    out_reg = a | b;  
  
  end  
  assign out = out_reg;  
endmodule
```

All signals used in a **procedural block** that have a value assigned to them **should be of a reg type**

# Verilog “always” statement: sequential logic

---

```
module test (  
    input clk, rst, d,  
    output q  
);  
    reg q;  
    always @ (posedge clk)  
        if (!rst)  
            q <= 1'b0;  
        else  
            q <= d;  
endmodule
```

```
module test (  
    input clk, rst, d,  
    output q  
);  
    reg q;  
    always @ (posedge clk, rst)  
        if (!rst)  
            q <= 1'b0;  
        else  
            q <= d;  
endmodule
```

# Verilog “always” statement: sequential logic

- ❑ Verilog description of a positive edge triggered D flip-flop, with synchronous (left) and asynchronous (right) reset

```
module test (  
    input clk, rst, d,  
    output q  
);  
    reg q;  
    always @ (posedge clk)  
        if (!rst)  
            q <= 1'b0;  
        else  
            q <= d;  
endmodule
```

```
module test (  
    input clk, rst, d,  
    output q  
);  
    reg q;  
    always @ (posedge clk, rst)  
        if (!rst)  
            q <= 1'b0;  
        else  
            q <= d;  
endmodule
```

# Verilog “always” statement: sequential logic

- ❑ Verilog description of a positive edge triggered D flip-flop, with synchronous (left) and asynchronous (right) reset

```
module test (  
    input clk, rst, d,  
    output q  
);  
    reg q;  
    always @ (posedge clk)  
        if (!rst)  
            q <= 1'b0;  
        else  
            q <= d;  
endmodule
```

```
module test (  
    input clk, rst, d,  
    output q  
);  
    reg q;  
    always @ (posedge clk, rst)  
        if (!rst)  
            q <= 1'b0;  
        else  
            q <= d;  
endmodule
```

# Verilog “always” statement: no sensitivity list

---

- ❑ The sensitivity list represents a sort of **timing sense** for the execution of the always block, remember **always** repetition is a continuous process
  - Sequential
- ❑ It's possible to drop the sensitivity list description

**always** clk = ~clk;

**always** #5 clk = ~clk;

# Verilog “always” statement: no sensitivity list

- ❑ The sensitivity list represents a sort of **timing sense** for the execution of the always block, remember **always** repetition is a continuous process
  - Sequential
- ❑ It's possible to drop the sensitivity list description

**always** clk = ~clk;

This code inverts the clk signal at every 0 time-unit increments  
With this kind of expression, **the always block will hang as an infinite loop. No sense of timing.**

**always** #5 clk = ~clk;

# Verilog “always” statement: no sensitivity list

- ❑ The sensitivity list represents a sort of **timing sense** for the execution of the always block, remember **always** repetition is a continuous process
  - Sequential
- ❑ It's possible to drop the sensitivity list description

**always** clk = ~clk;

Bringing back the sense of timing without the sensitivity list.

This code inverts the clk signal at every 0 time-unit increments  
With this kind of expression, **the always block will hang as an infinite loop. No sense of timing.**

**always** #5 clk = ~clk;

A 5 time-unit delay



# Verilog Blocks: initial

# Verilog “initial” statement

- ❑ **initial** block is a procedural assignment block in Verilog
- ❑ The code inside of the initial block is executed sequentially
- ❑ The code inside of the initial block is **not synthesizable**
- ❑ initial statements are used in **simulations** and are used for **initialization** of the design ports

**initial**

statement

**initial begin**

statements

**end**

In case of multiple statements “begin” / “end” construct is needed

# Verilog “initial” statement: operation

---

- ❑ The code in initial block **doesn't** “become” a hardware circuit
- ❑ All initial blocks are **parallel** and start at time point 0
- ❑ initial block is **executed only once**

```
module test;  
    reg a, b;  
  
    initial begin  
        a = 1'b0;  
        b = 1'b1;  
    end  
endmodule
```

# Verilog “initial” statement: operation

- ❑ The code in initial block **doesn't** “become” a hardware circuit
- ❑ All initial blocks are **parallel** and start at time point 0
- ❑ initial block is **executed only once**

```
module test;
```

```
  reg a, b;
```

initial is a procedural block,  
hence a & b are of **reg** type

```
  initial begin
```

```
    a = 1'b0;
```

```
    b = 1'b1;
```

Are assigned their values at 0  
time point, i.e., the start of the  
simulation

```
  end
```

```
endmodule
```

# Verilog “initial” statement: delays

---

- ❑ The delay increments the time by specified units: **#number**

```
module test;  
    reg a, b, c, d;  
  
    initial begin  
        a = 1'b0;  
        b = 1'b1;  
    end  
    initial begin  
        c = 1'b1;  
        #5 d = 1'b0;  
    end  
endmodule
```

# Verilog “initial” statement: delays

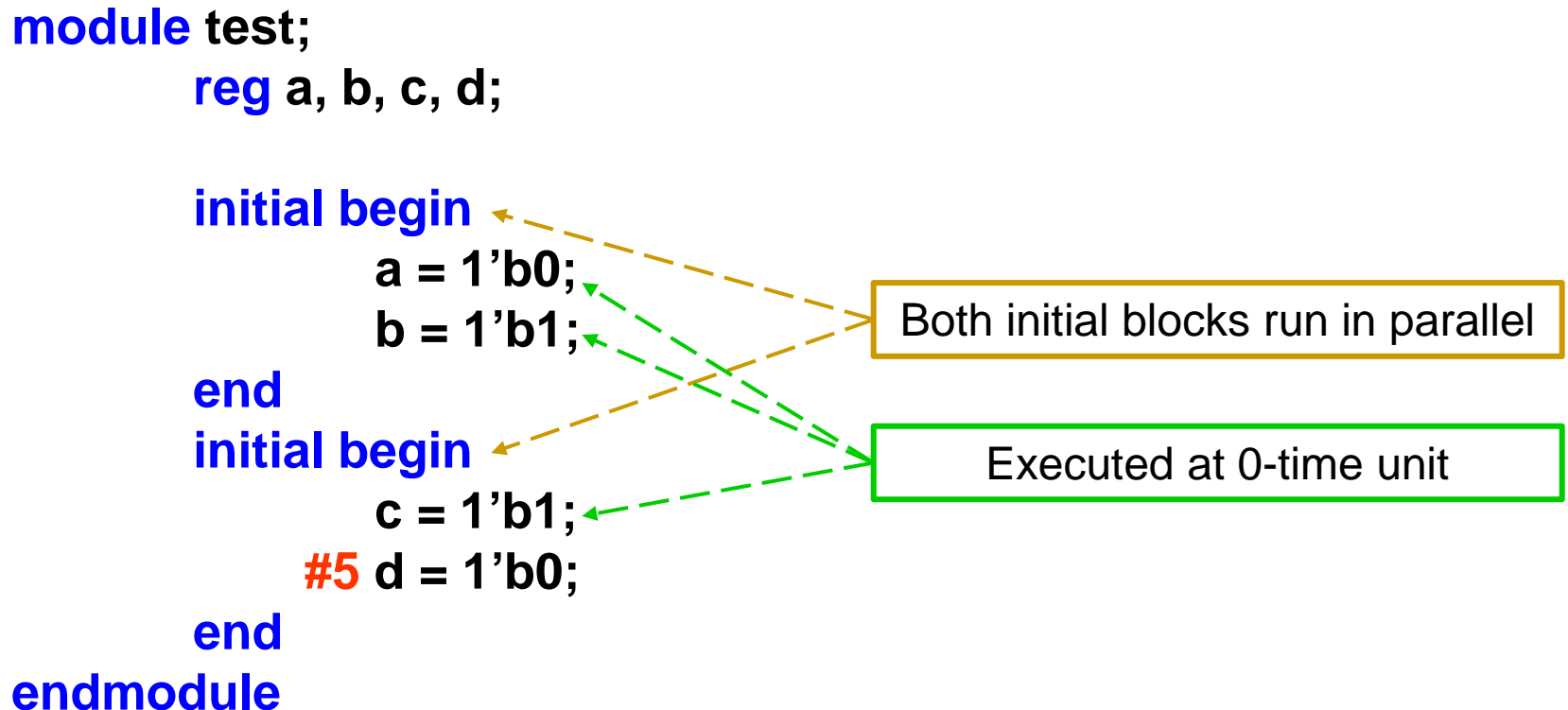
- ❑ The delay increments the time by specified units: **#number**

```
module test;  
    reg a, b, c, d;  
  
    initial begin  
        a = 1'b0;  
        b = 1'b1;  
    end  
    initial begin  
        c = 1'b1;  
        #5 d = 1'b0;  
    end  
endmodule
```

Both initial blocks run in parallel

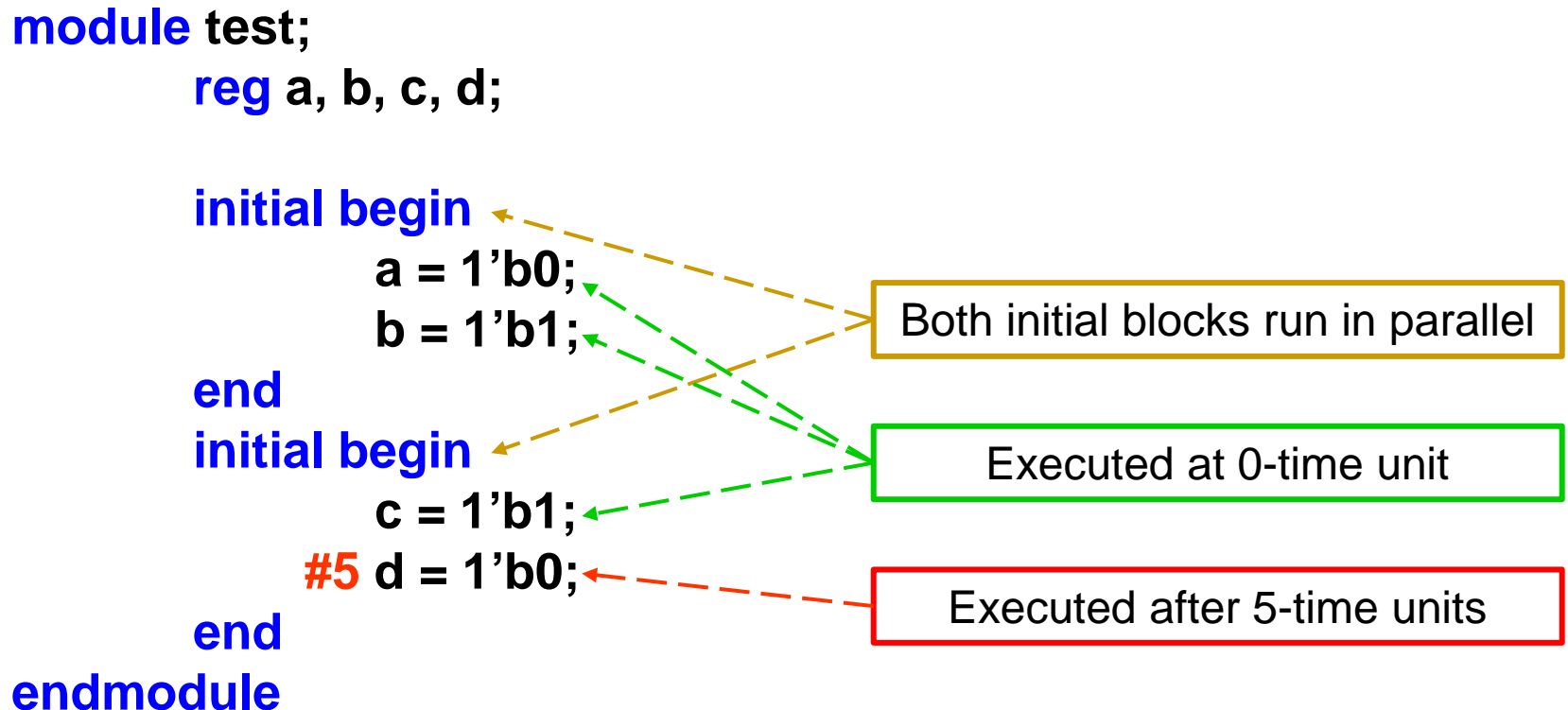
# Verilog “initial” statement: delays

- ❑ The delay increments the time by specified units: **#number**



# Verilog “initial” statement: delays

- ❑ The delay increments the time by specified units: **#number**





# Verilog “initial” statement: **\$finish** task

- There is a Verilog system task, which is used to finish the simulation: **\$finish**

```
module test;
    reg a, b, c, d;

    initial begin
        a = 1'b0;
        #15;
        b = 1'b1;
        #5 $finish;
    end
endmodule
```

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

    initial begin
        #10 a = 1'b0;
        #15 b = 1'b1;
    end
    initial begin
        c = 1'b1;
        #25 $finish;
    end
endmodule
```

---

# **Verilog Assignments and Blocks: summary**

---

# Verilog assign, always and initial statements

---

## ❑ assign

- Is a continuous statement, used to assign values to wires
- Is used to model combinational logic
- Is a one-line statement

## ❑ always

- Is a procedural block, used to assign values to regs
- Can be used to model combinational and sequential logic

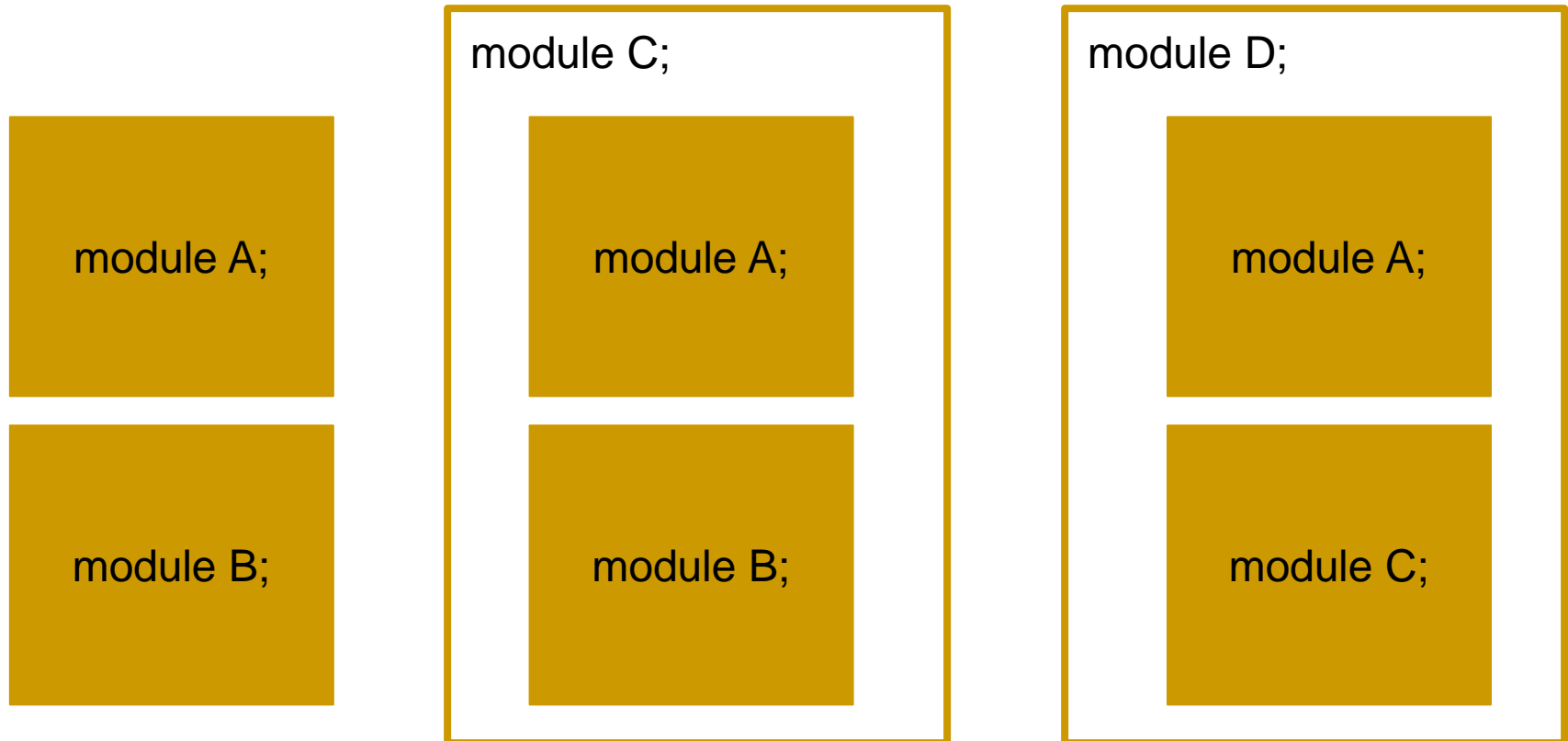
## ❑ initial

- Is a procedural block, used to initialize values or ports of the design
- Is not synthesized and doesn't "become" a hardware circuit

# Verilog Module Instantiation

# Verilog module instantiation

- ❑ To make complex modules we can use other modules to build bigger ones



# Verilog module instantiation (2)

```
module my_and(input a, b,  
              output out);  
    assign out = a & b;  
endmodule
```

```
module circuit(input in1, in2, in3,  
               output y);  
    wire and_out;  
  
    my_and and0(in1, in2, and_out);  
  
    assign y = and_out | in3;  
endmodule
```

module my\_and



module circuit

and0

module my\_and

# Verilog module instantiation (3)

```
module my_and(input a, b,  
              output out);  
    assign out = a & b;  
endmodule
```

A module called “my\_and”

```
module circuit(input in1, in2, in3,  
               output y);  
    wire and_out;  
    my_and and0(in1, in2, and_out);  
    assign y = and_out | in3;  
endmodule
```

The instance of “my\_and”

A module called “circuit”

```
my_and and0(in1, in2, and_out);
```

The name of  
the module

Name of the  
instance

Connections inside the parent module

# Verilog module instantiation (4)

```
module my_and(input a, b,  
              output out);  
    assign out = a & b;  
endmodule
```

```
module circuit(input in1, in2, in3,  
               output y);  
    wire and_out;  
    my_and and0(.a(in1), .b(in2), y(and_out));  
    assign y = and_out | in3;  
endmodule
```

The order of port declaration has to be known

my\_and and0(in1, in2, and\_out);

.out(and\_out)

The ports are connected via names:

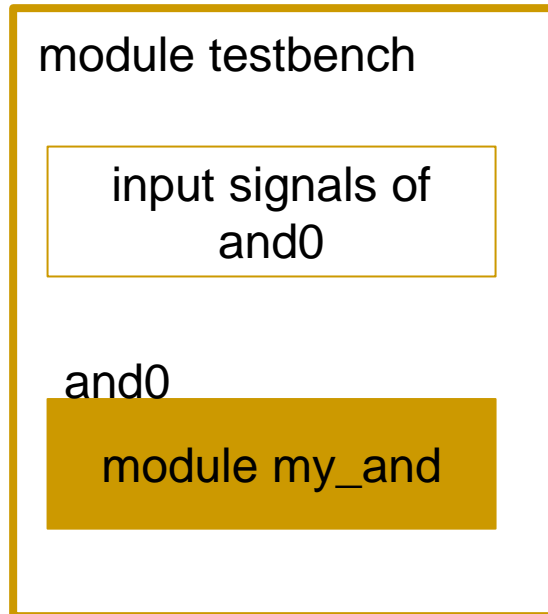
Signal **a** of **my\_and** is connected to **in1** from **circuit** module



# Verilog Testbench

# Verilog Testbench

- ❑ Testbenches are used in Verilog code simulation
- ❑ Testbench is a module which “calls” the design/unit under test(DUT/UUT)
- ❑ With the help of testbenches we simulate input signals of the design

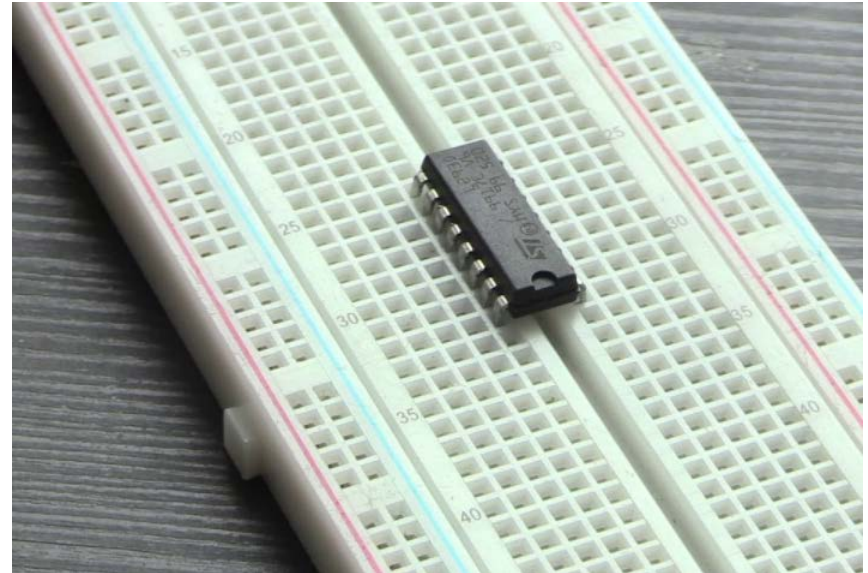


# Verilog Testbench (2)

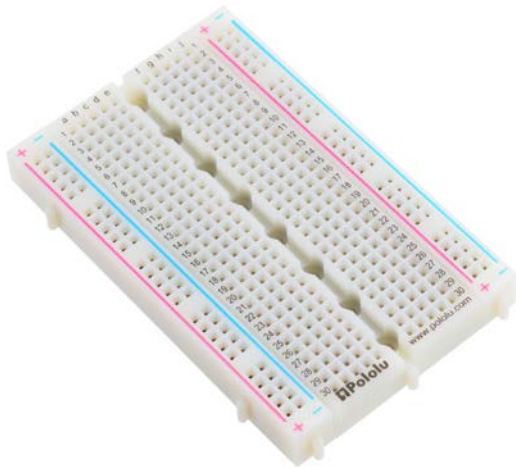
Our module under test or UUT



The instantiated module in testbench



Our testbench

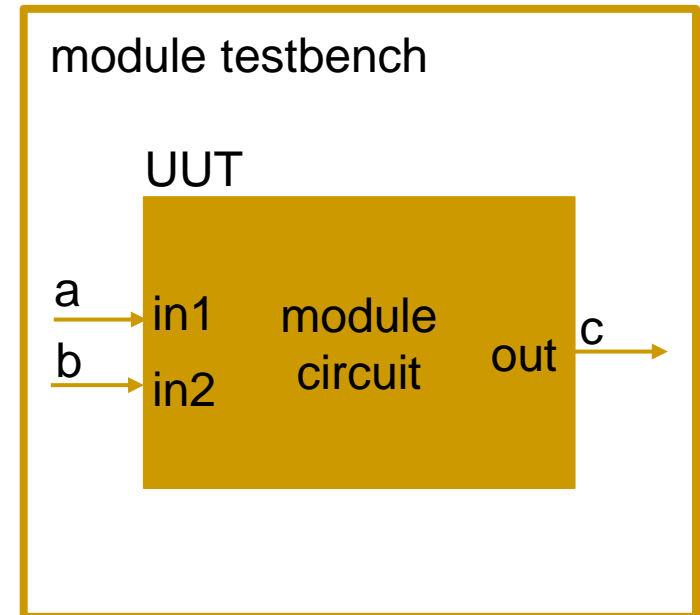
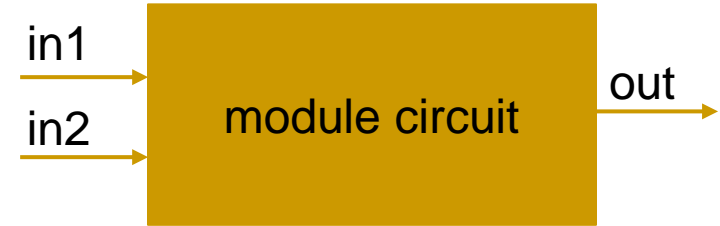


Now the testbench is ready for signal application

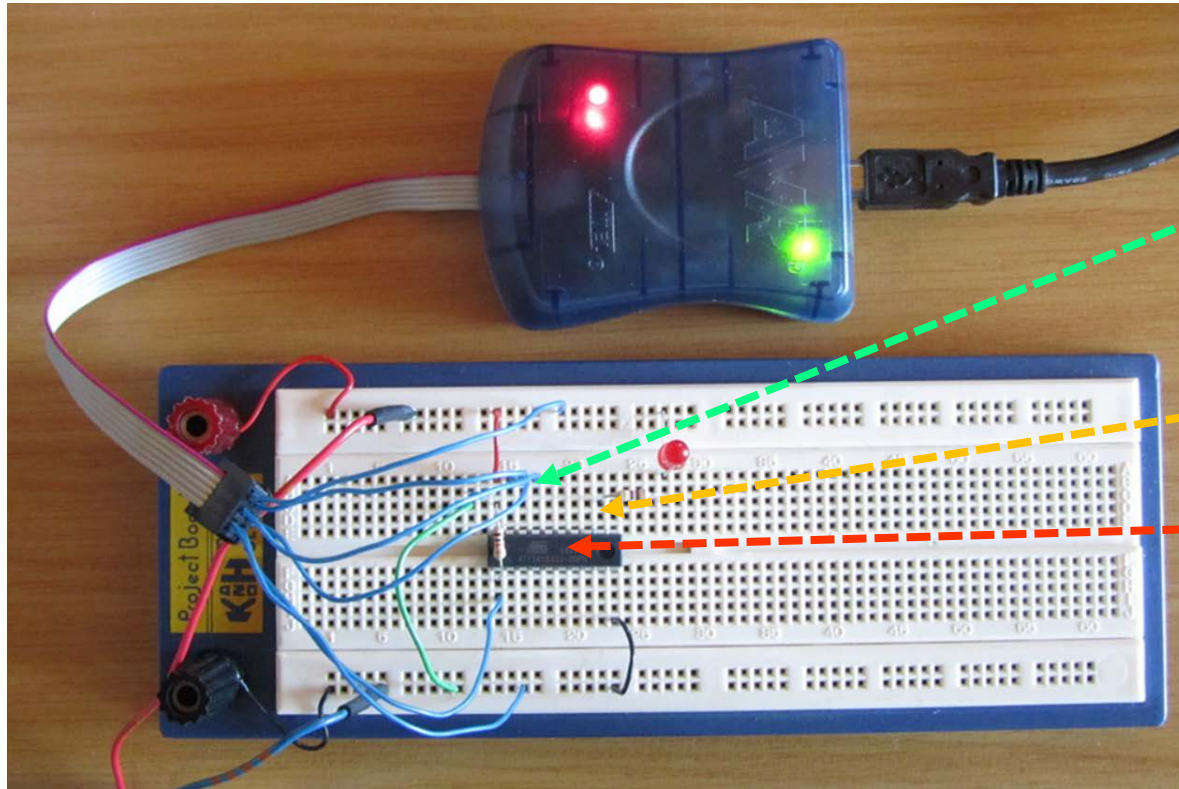
# Verilog Testbench: syntax

The general syntax:

```
module testbench;  
    wire c;  
    reg a, b;  
  
    circuit UUT(.in1(a), .in2(b), .out(c));  
  
    initial begin  
        a = 1'b1;  
        b = 1'b0;  
        #10 a = 1'b0;  
        #15 b = 1'b1;  
        #30 $finish;  
    end  
endmodule
```



# Verilog Testbench Analogy



Input signals

Output signal

The design

---

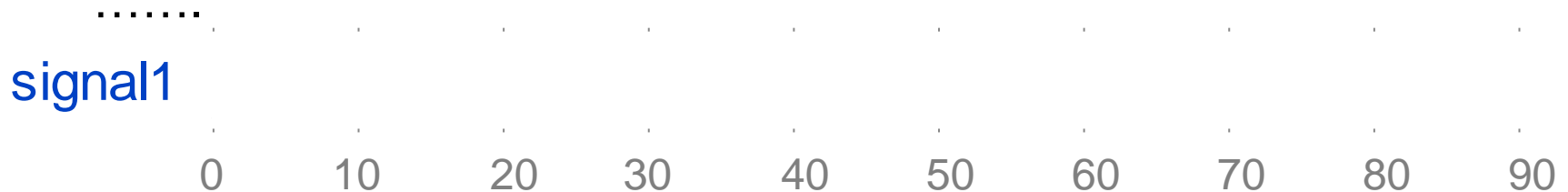
# **Blocking and non-blocking assignments**

---

# Verilog Blocking statements

- ❑ Blocking statements are performed via “=” sign
- ❑ Blocking statements are executed sequentially in procedural blocks, one after another

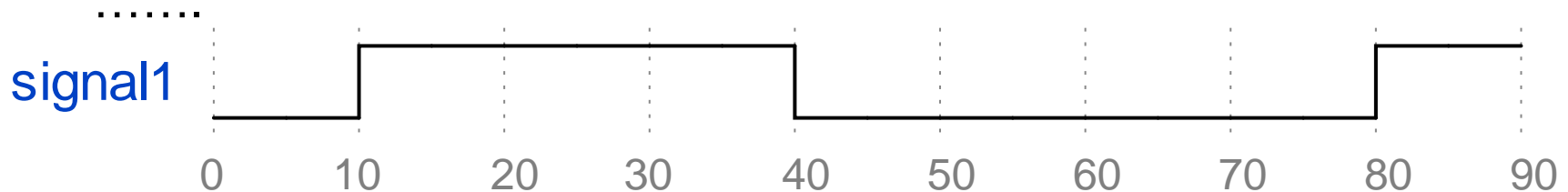
```
module block();  
    reg signal1;  
    initial begin  
        signal1 = 0;  
        #10 signal1 = 1'b1;  
        #30 signal1 = 1'b0;  
        #40 signal1 = 1'b1;  
    end
```



# Verilog Blocking statements

- ❑ Blocking statements are performed via “=” sign
- ❑ Blocking statements are executed sequentially in procedural blocks, one after another

```
module block();  
    reg signal1;  
    initial begin  
        signal1 = 0;  
        #10 signal1 = 1'b1;  
        #30 signal1 = 1'b0;  
        #40 signal1 = 1'b1;  
    end  
endmodule
```





# Verilog Non-Blocking statements

- ❑ Blocking statements are performed via “<=” sign
- ❑ Non-blocking assignments are executed without blocking the rest of the statements

```
module non_block();  
    reg signal2;  
    initial begin  
        signal2 <= 0;  
        #10 signal2 <= 1'b1;  
        #30 signal2 <= 1'b0;  
        #40 signal2 <= 1'b1;  
        .....
```

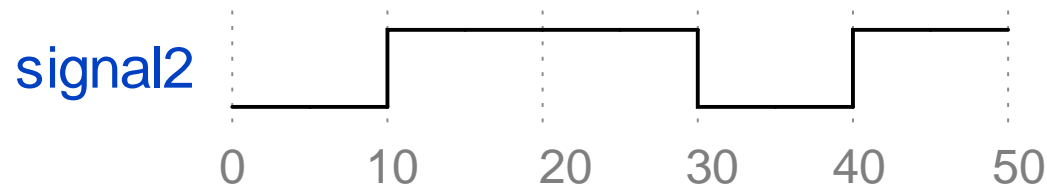
signal2

0 10 20 30 40 50

# Verilog Non-Blocking statements

- ❑ Blocking statements are performed via “<=” sign
- ❑ Non-blocking assignments are executed without blocking the rest of the statements

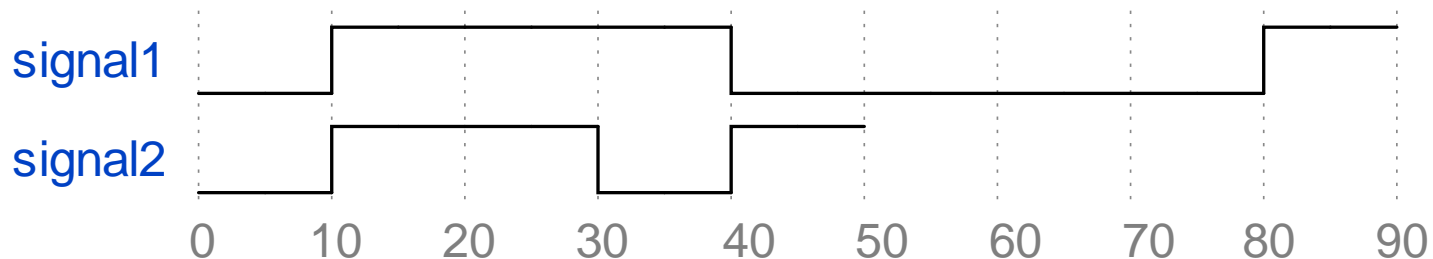
```
module non_block();  
    reg signal2;  
    initial begin  
        signal2 <= 0;  
        #10 signal2 <= 1'b1;  
        #30 signal2 <= 1'b0;  
        #40 signal2 <= 1'b1;  
        .....  
    end  
endmodule
```



# Verilog Blocking and non-blocking assignments: comparison

```
module block();  
  reg signal1;  
  initial begin  
    signal1 = 0;  
    #10 signal1 = 1'b1;  
    #30 signal1 = 1'b0;  
    #40 signal1 = 1'b1;  
  end  
  .....  
endmodule
```

```
module non_block();  
  reg signal2;  
  initial begin  
    signal2 <= 0;  
    #10 signal2 <= 1'b1;  
    #30 signal2 <= 1'b0;  
    #40 signal2 <= 1'b1;  
  end  
  .....  
endmodule
```



# Verilog Blocking and non-blocking assignments: guidelines

---

- ❑ Use **blocking** assignments in always blocks that are written to generate **combinational logic**
- ❑ Use **nonblocking** assignments in always blocks that are written to generate **sequential logic**
- ❑ When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.
- ❑ **Do not mix** blocking and nonblocking assignments in the same always block.

# **Verilog control flow, statements: if-else**

# if-else Statement

---

Keyword: **if-else**

```
if (<expression>) true_statement;
```

```
if (<expression>) true_statement;  
else           false_statement;
```

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

**if** (expression)  $\iff$  **if**(expression!=0)

# Conditional Operator Implementation

---

- ❑ Ternary operator ? :
- ❑ Nested conditional operators

**var = condition ? True : False**  
**var = 1 ? 2'b10 : 2'b11**

```
module multiplexer4_1 (  
    input  in0, in1, in2, in3, a0, a1,  
    output y  
);  
    assign y = a0 ? (a1 ? in3 : in1) : (a1 ? in2 : in0);  
endmodule
```

# if-else Statement, if Statement

---

```
module mux2_1(  
    input in0, in1, s,  
    output reg y  
);  
    always @(in0, in1, s)  
        if (s == 1)  
            y = in1;  
        else  
            y = in0;  
endmodule
```

```
module mux2_1(  
    input in0, in1, s,  
    output reg y  
);  
    always @(in0, in1, s) begin  
        y = in0;  
        if (s == 1) y = in1;  
    end  
endmodule
```

Both descriptions correspond to 2:1 multiplexer.



# Equality Operators

---

Expression	Description	Possible Logical Value	
<code>a == b</code>	a equal to b, result unknown if x or z in a or b	0,1,x	Supported by synthesis
<code>a != b</code>	a not equal to b, result unknown if x or z in a or b	0,1,x	Supported by synthesis
<code>a === b</code>	a equal to b, including x and z	0,1	Not supported by synthesis
<code>a !== b</code>	a not equal to b, including x and z	0,1	Not supported by synthesis

# if-else: Examples

---

//Type 1 statements

```
if(!lock) buffer = data;
```

```
if(enable) out = in;
```

//Type 2 statements

```
if (counter < MAX_number) begin
```

```
    sum = sum + data;
```

```
    counter = counter + 1;
```

```
end
```

```
else
```

```
    $display("End of operation ");
```

//Type 3 statements

//Execute statements based on ALU control signal.

```
reg[1:0] alu_control;
```

```
...
```

```
if (alu_control == 0) y = x + z;
```

```
else if (alu_control == 1) y = x - z;
```

```
else $display("Invalid ALU control signal");
```

# **Verilog control flow, statements: case**

# Case Statement in Verilog

---

```
module ... (...);  
    declarations;  
    always @ (sensitivity list)  
        case (case switch list)  
            case0: case0 statement;  
            case1: begin  
                case1 statements;  
            end  
            case2: case2 statement;  
            ....  
            caseN: caseN statement;  
            default: $display ("Please check the bits");  
        endcase  
    endmodule
```

# Case Statement

---

- ❑ The keywords `case`, `endcase`, and `default` are used in the `case` statement.

```
case (selection-expression)
  alternative1: statement1;
  alternative2: statement2;
  alternative3: statement3;
  ...
  default: default_statement;
endcase
```

- ❑ Case statement may yield a faster synthesized circuit, based on multiplexers, and is more readable.

# Case Statement: Example

---

```
reg[1:0] op_code;  
...  
case (op_code)  
    2'b00: y = x + z;  
    2'b01: y = x - z;  
    2'b10: y = x * z;  
    default: $display("Invalid op_code");  
endcase
```

- ❑ In a non full case, the synthesizer will infer a **latch** to retain the previous outputs for any cases that are not covered.
- ❑ The best coding practice is to use **full case statements only**.
- ❑ The good coding practice is to use **default** statement except full case (especially for simulation).

# Casex, Casez

- `casez` treats all `z` values in the case alternatives or the case expression as `don't cares`. `z` can also be represented by ?
- `casex` treats all `x` and `z` values in the case item or the case expression as `don't cares`.
- Example

```
reg [3:0] encoding;  
integer state;  
case (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
```

Input encoding = 4'b010z would cause next\_state = 2 to be executed.

# Loops: for, while



# Looping Statements: For

---

- ❑ Looping statements in Verilog: **while**, **for**, **repeat** and **forever**
  - All looping statements can appear **only inside an initial or always** blocks.
- ❑ Syntax of **for** statement:  

```
for (index=first_expr; logical_expression; index=second_expr)  
    begin procedural-statements end
```
- 1. At the beginning of each iteration, it checks the logical expression.
- 2. If the value is false, the **for** loop stops execution.
- 3. For synthesis, the **first\_expr** must be **constant**. **Why?**
- 4. The **second\_exp**: incrementing or decrementing.

# for Statement: Example

## One's counter

```
module counter
#(
    parameter n=32,
    localparam logn=$clog2(n)
)(
    input [n-1:0] A,
    output reg [logn:0] count
);
    integer i;
    always @(A)
    begin
        count=0;
        for (i=0; i<n; i=i+1)
            count=count + A[i];
    end
endmodule
```

## Another variant

```
module counter
#(
    parameter n=32,
    localparam logn=$clog2(n)
)(
    input [n-1:0] A,
    output reg [logn:0] count
);
    reg [n-1:0] ra;
    always @(A)
    begin
        count=0;
        for (ra=A; ra !=0; ra=ra>>1)
            count=count + ra[0];
    end
endmodule
```

# Looping Statements: While

- ❑ Syntax of **while** statement:  
**while** (logical\_expression) procedural\_statement
- ❑ The **while** loop executes until the logical expression is not true.

```
module counter(A, count);  
    parameter n=32;  
    localparam logn=$clog2(n);  
    input [n - 1:0] A;  
    output reg [logn:0] count;  
    reg [n - 1:0] ra;  
    always @(A)  
        begin  
            while (expression)  
                begin  
                    end  
            end  
        end  
endmodule
```

# Loops: repeat, forever

# Looping Statements: repeat

---

- ❑ Syntax of **repeat** statement:

**repeat** (integer\_expression) procedural statement

- ❑ **repeat** executes the loop a fixed number of times.

```
module repeat;  
    reg r_Clock = 1'b0;  
    initial begin  
        repeat (10)  
            #5 r_Clock = ~r_Clock;  
        $display("Simulation Complete");  
    end  
endmodule
```

# repeat Statement: Example

```
module counter(A, count); // MSB 100000 LSB
    parameter n = 32;
    parameter logn = 5;
    input [n-1:0] A;
    output reg [logn:0] count;
    reg [n-1:0] ra;

    always @(A)
        begin
            count = 0;
            ra = A;
            repeat (n)
                begin
                    .....
                end
            end
        end
    endmodule
```

# Looping Statements: forever

---

- ❑ Syntax of **forever** statement:  
    **forever** procedural statements
- ❑ **forever** executes the loop until the simulation finishes.

```
module forever;  
    reg f_Clock = 1'b0;  
    initial begin  
        forever  
            #15 f_Clock = ~f_Clock;  
    end  
endmodule
```

# Verilog Timescale

- ❑ Verilog simulation depends on time
- ❑ Verilog simulator needs to know what is the time unit
- ❑ The ``timescale` compiler directive is used to provide the timescale

``timescale <time unit>/<time precision>`

``timescale 1ns/1ps`

The time unit #1 is 1ns

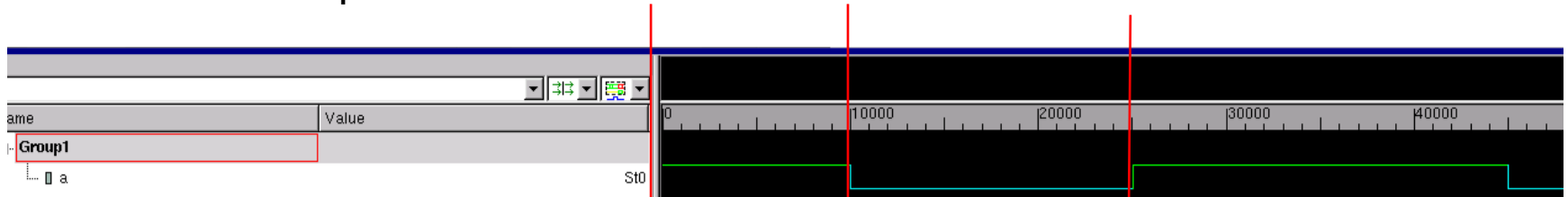
The precision is 1ps

#10 is 10ns

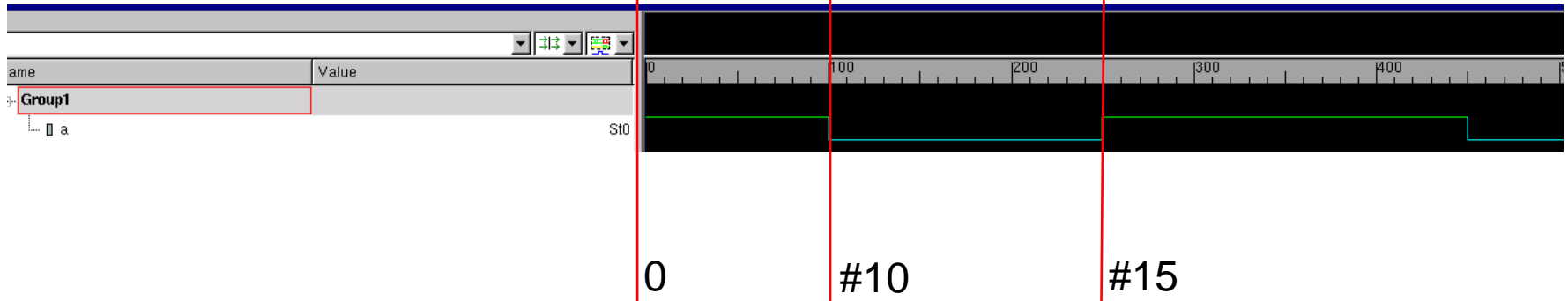


# Verilog Timescale & delays

``timescale 1ns/1ps`

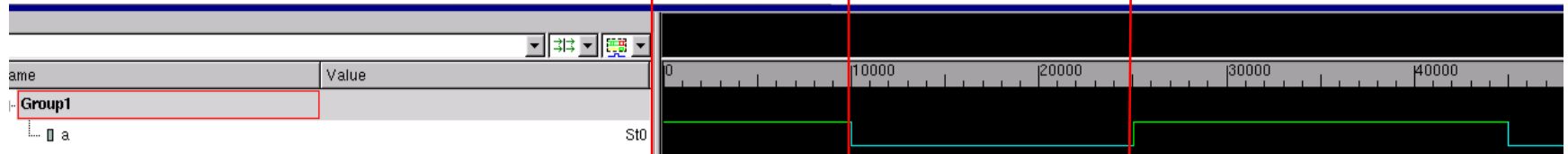


``timescale 10ns/1ns`

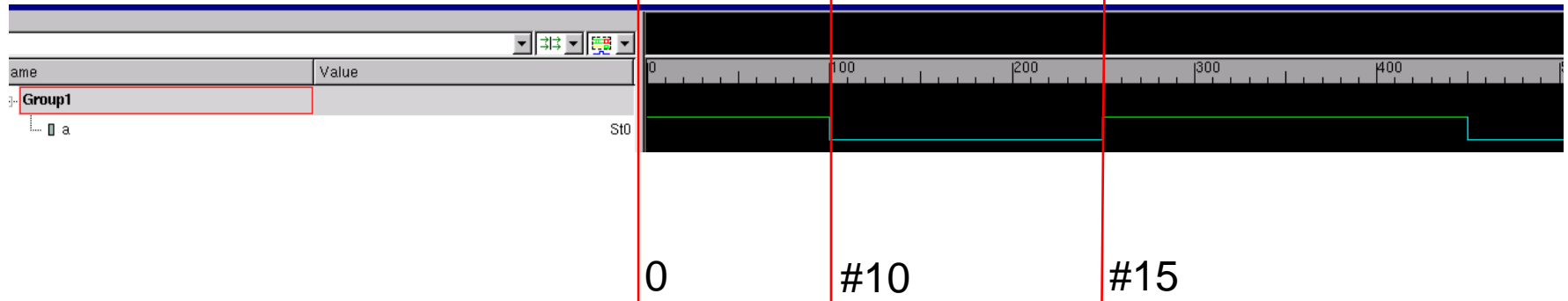


# Verilog Timescale & delays

``timescale 1ns/1ps`



``timescale 10ns/1ns`



# Verilog Tasks

# Verilog tasks

---

**Tasks are subroutines** that can be called anytime in the module they are defined.

- ❑ Tasks can include time delays.
- ❑ Tasks can have any number of inputs and outputs.
- ❑ If a variable is declared within the task, it is local to the task and can't be used outside the task.
- ❑ Tasks can drive global variables if no local variables are declared.
- ❑ Tasks are called with statements and cannot be used in an expression.

# Verilog tasks: syntax

- ❑ A task doesn't need to have arguments in port list
  - there are different types of task declaration:

The task name is like a module name

`task [name];`  
`input [port_list];`  
`inout [port_list];`  
`output [port_list];`  
`begin`  
`[statements]`  
`end`  
`endtask`

`task [name] (in/out [port_list]);`  
`begin`  
`[statements]`  
`end`  
`endtask`

`task [name] ();`  
`begin`  
`[statements]`  
`end`  
`endtask`

Calling the task:

.....  
`[name] ([arguments]);`  
.....

Correspond to the  
ports of the task

# Verilog tasks: example

```
module task_calling(adc_a, adc_b, adc_a_conv, adc_b_conv);  
    input [7:0] adc_a, adc_b;  
    output [7:0] adc_a_conv, adc_b_conv;  
    reg [7:0] adc_a_conv, adc_b_conv;  
    task convert;  
        input [7:0] adc_in;  
        output [7:0] out;  
        begin  
            out = ( 9/5 ) * (adc_in + 32)  
        end  
    endtask  
    always @ (adc_a)  
        begin  
            convert(adc_a, adc_a_conv);  
        end  
endmodule
```

The declaration of the task

This is how we call the task

# Verilog Functions

# Verilog functions

---

**Functions** are like tasks, with some differences

- ❑ Functions cannot drive more than one output
- ❑ Functions cannot include timing delays, like posedge, negedge, simulation delay
  - → functions implement **combinational** logic
- ❑ Functions can have any number of inputs but **only one output**
- ❑ The variables declared within the function are local to that function
- ❑ The order of declaration within the function are considered and have to be the same as the caller
- ❑ Functions can use and modify global variables, when no local variables are used
- ❑ Functions can call other functions but cannot call tasks



# Verilog functions: syntax

The function name is like a module name

`function` name;  
    [statements]  
`endfunction`

`function` name (input/output/inout [port\_list]);  
    [statements]  
`endfunction`

Calling the function:

```
.....  
        var = [name] ([arguments]);  
.....
```

**!** The function definition will implicitly create an internal variable of the same name as the function

→ it is illegal to declare another variable of the same name inside the scope of the function

# Verilog functions: example

```
module function_calling(a, b, c, d, e, f);  
    input a, b, c, d, e ;  
    output f;
```

```
    function myfunction;  
        input a, b, c, d;  
        begin  
            myfunction = ((a + b) + (c - d));  
        end  
    endfunction
```

```
    assign f = (myfunction (a, b, c, d)) ? e :0;  
endmodule
```

→ The declaration of the function

→ This is how we call the function

# Verilog block statements

# Verilog block statements

---

There are ways to group a set of statements together if the statements are syntactically or “logically” equivalent, to a single statement.

These are known as **block statements**. There are two types of block statements:

- ❑ Sequential
- ❑ Parallel

# Verilog block statements: sequential

---

- Sequential statements are wrapped in **begin-end** keywords and all of them are going to be executed one after another.

.....

**initial begin**

    #10 data = 8'hfe;

    #20 data = 8'h11;

**end**

.....

# Verilog block statements: sequential

---

- Sequential statements are wrapped in **begin-end** keywords and all of them are going to be executed one after another.

```
.....  
initial begin  
    #10 data = 8'hfe;  
    #20 data = 8'h11;  
end  
.....
```

↓

```
initial begin  
    |  
    #10 data = 8'hfe;  
    #20 data = 8'h11;  
    ↓  
end
```

# Verilog block statements: parallel

---

- ❑ Parallel blocks can execute statements concurrently, delay control can be used for timing

.....

initial begin

    #10 data = 8'hfe;

    fork

        #20 data = 8'h11;

        #10 data = 8'h00;

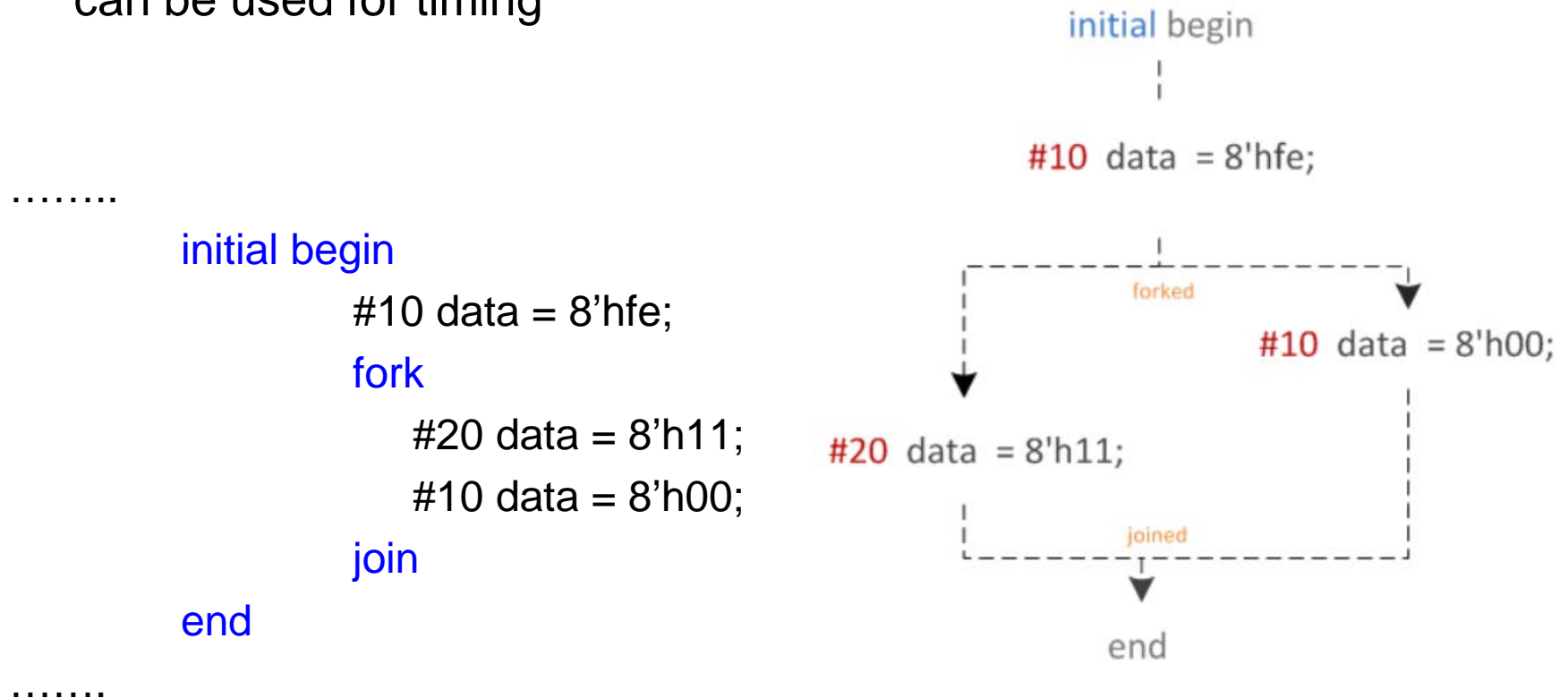
    join

end

.....

# Verilog block statements: parallel

- Parallel blocks can execute statements concurrently, delay control can be used for timing





# Verilog block statements: parallel

---

- Parallel blocks can execute statements concurrently, delay control can be used for timing

.....

```
initial begin
    #10 data = 8'hfe;
    fork
        #10 data = 8'h11;
        begin
            #20 data = 8'h00;
            #30 data = 8'haa;
        end
    join
end
```

.....

# Verilog block statements: parallel

- Parallel blocks can execute statements concurrently, delay control can be used for timing

.....

**initial begin**

**#10 data = 8'hfe;**

**fork**

**#10 data = 8'h11;**

**begin**

**#20 data = 8'h00;**

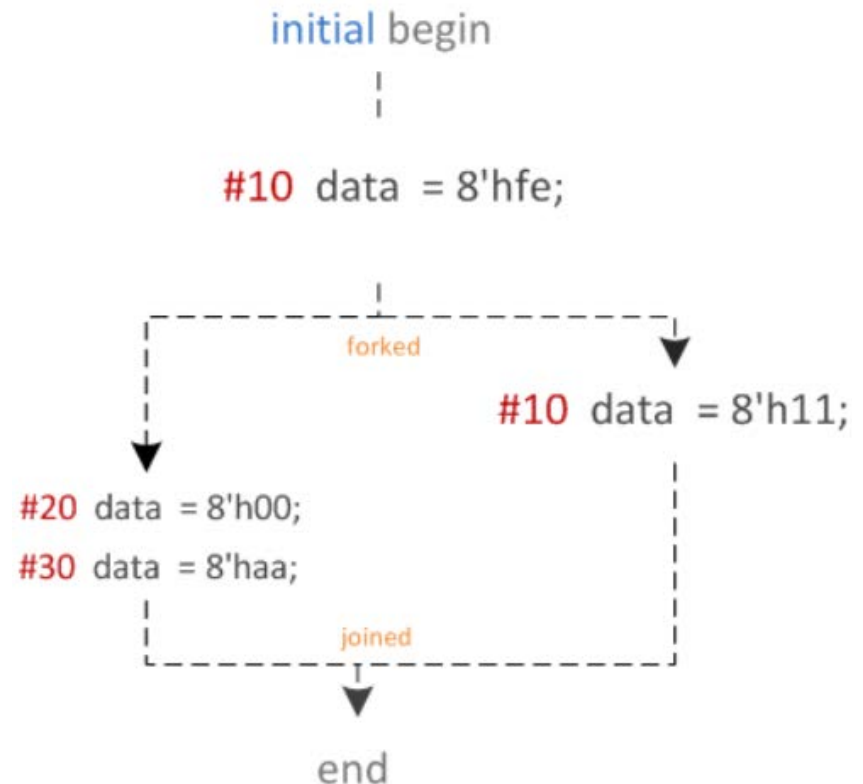
**#30 data = 8'haa;**

**end**

**join**

**end**

.....



# Verilog generate

# Verilog generate block

---

- ❑ Verilog generate statement is **a powerful construct for writing configurable, synthesizable RTL**
  - create multiple instantiations of modules and code, or conditionally instantiate blocks of code

There are two types of Verilog generate:

- ❑ Generate loop
  - Allows a block of code to be instantiated multiple times
- ❑ Conditional generate
  - Allows to select one block of code between multiple blocks
- ❑ All the generate expressions have to be deterministic: constant expressions (parameters)

# Verilog generate loop

---

- ❑ Generate loop
  - Allows a block of code to be instantiated multiple times
- ❑ The syntax for generate loop is similar to for loop
  - The loop index must be declared with **genvar** keyword before it's used

# Verilog generate loop: example(1)

```
module gray2bin
  #(parameter SIZE = 8)
  (
    input [SIZE-1:0] gray,
    output [SIZE-1:0] bin
  );

  genvar gi;
  // generate and endgenerate is optional
  generate
    for (gi = 0; gi < SIZE-1; gi = gi + 1) begin : genbit
      assign bin[gi] = ^gray[SIZE - 1 : gi];
    end
    assign bin[SIZE-1] = gray[SIZE-1];
  endgenerate
endmodule
```

Module parameter

Name of generate

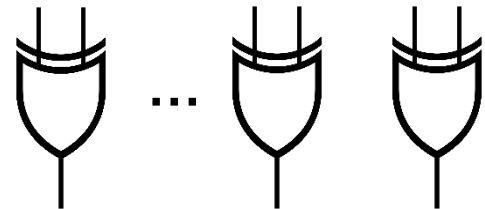
# Verilog generate loop: example(1)

```
module gray2bin
  #(parameter SIZE = 8)
  (
    input [SIZE-1:0] gray,
    output [SIZE-1:0] bin
  );

  genvar gi;
  // generate and endgenerate is optional
  generate
    for (gi = 0; gi < SIZE-1; gi = gi + 1) begin : genbit
      assign bin[gi] = ^gray[SIZE - 1 : gi];
    end
    assign bin[SIZE-1] = gray[SIZE-1];
  endgenerate
endmodule
```

Module parameter

Name of generate



# Verilog generate loop: example(2-1)

```
module my_design
  #(parameter N=4)
  (  input [N-1:0] a, b,
    output [N-1:0] sum, cout);

  // Declare a temporary loop variable to be used during
  // generation and it won't be available during simulation
  genvar i;

  // Generate for loop to instantiate N times
  generate
    for (i = 0; i < N; i = i + 1) begin : bbb
      ha u0 (a[i], b[i], sum[i], cout[i]);
    end
  endgenerate
endmodule
```

Module “ha” is a half-adder,  
supposedly defined in a separate file  
or before my\_design



# Verilog generate loop: example(2-2)

```
module tb;  
  parameter N = 2;  
  reg [N-1:0] a, b;  
  wire [N-1:0] sum, cout;
```

```
  // Instantiate top level design with N=2 so that it will have 2  
  // separate instances of half adders and both are given two separate  
  // inputs  
  my_design #(N(N)) md( .a(a), .b(b), .sum(sum), .cout(cout));
```

```
initial begin  
  a <= 0;  
  b <= 0;  
  #10 a <= 'h2;  
    b <= 'h3;  
  #20 b <= 'h4;  
  #10 a <= 'h5;  
end  
endmodule
```

`#(.N(N)) md(.a(a), .b(b), .sum(sum), .cout(cout));`

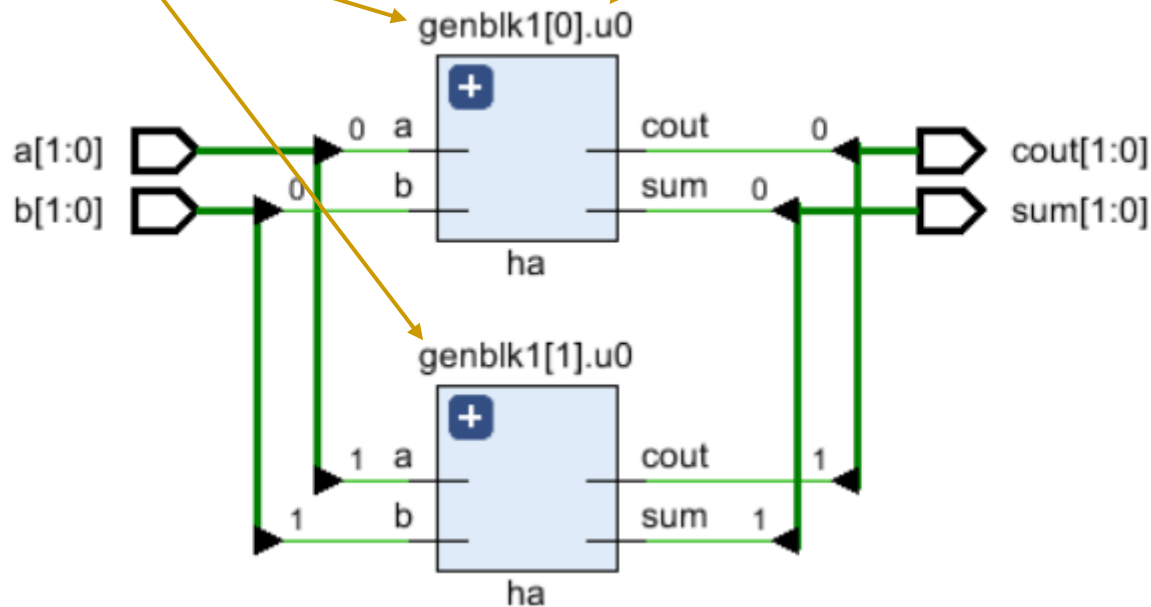
Parameter

Module instantiation itself

# Verilog generate loop: example(2-3)

Name of the generate block

Name of the instance from  
“my\_design” module



2 instances, because parameter = 2

# Verilog variable scope

# Variable scopes

---

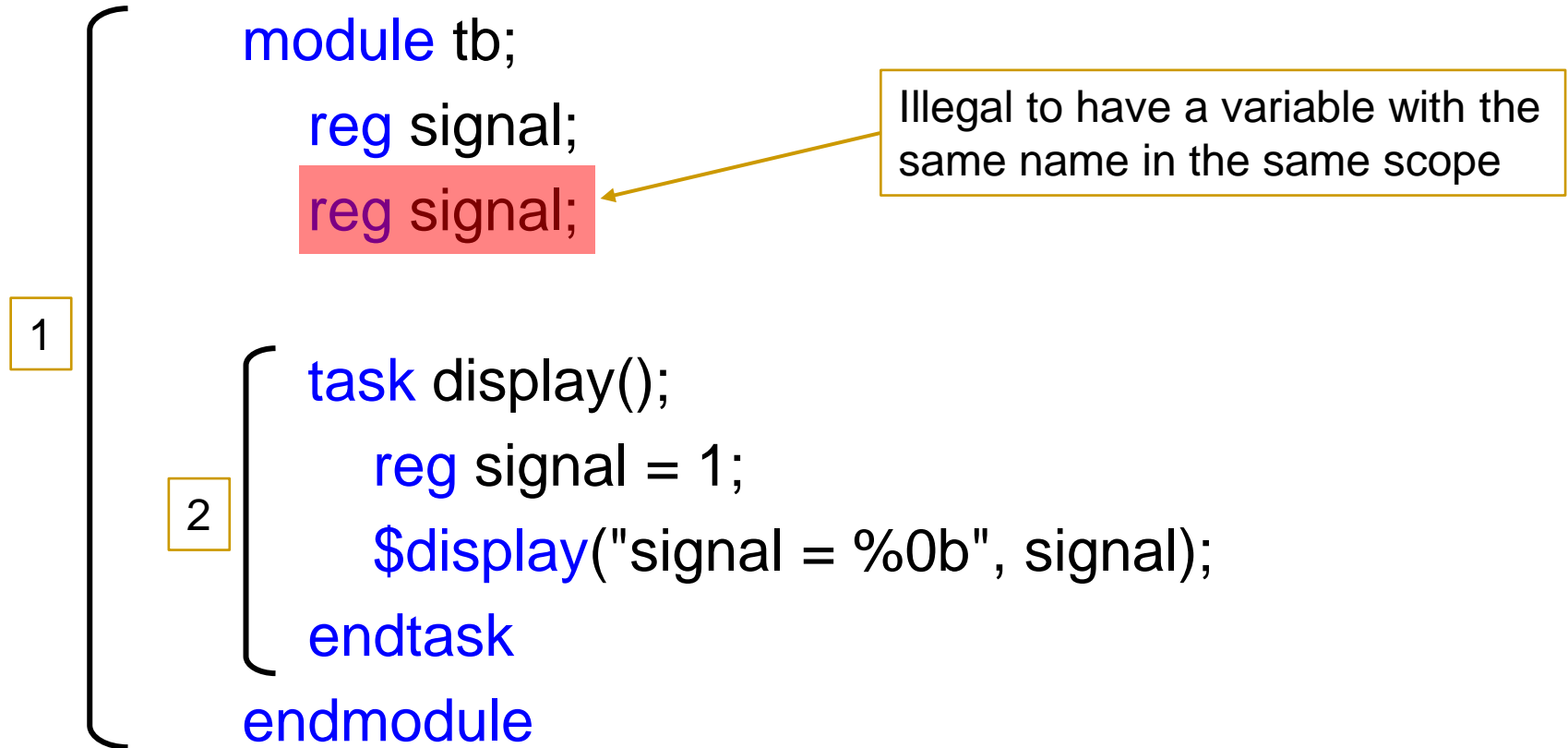
- ❑ The scope defines the visibility of certain pieces of code to variables and functions/methods
- ❑ The scope defines a namespace where the variables and other objects have their respective names, and the collisions are avoided between two or more namespaces

# Variable scopes: example

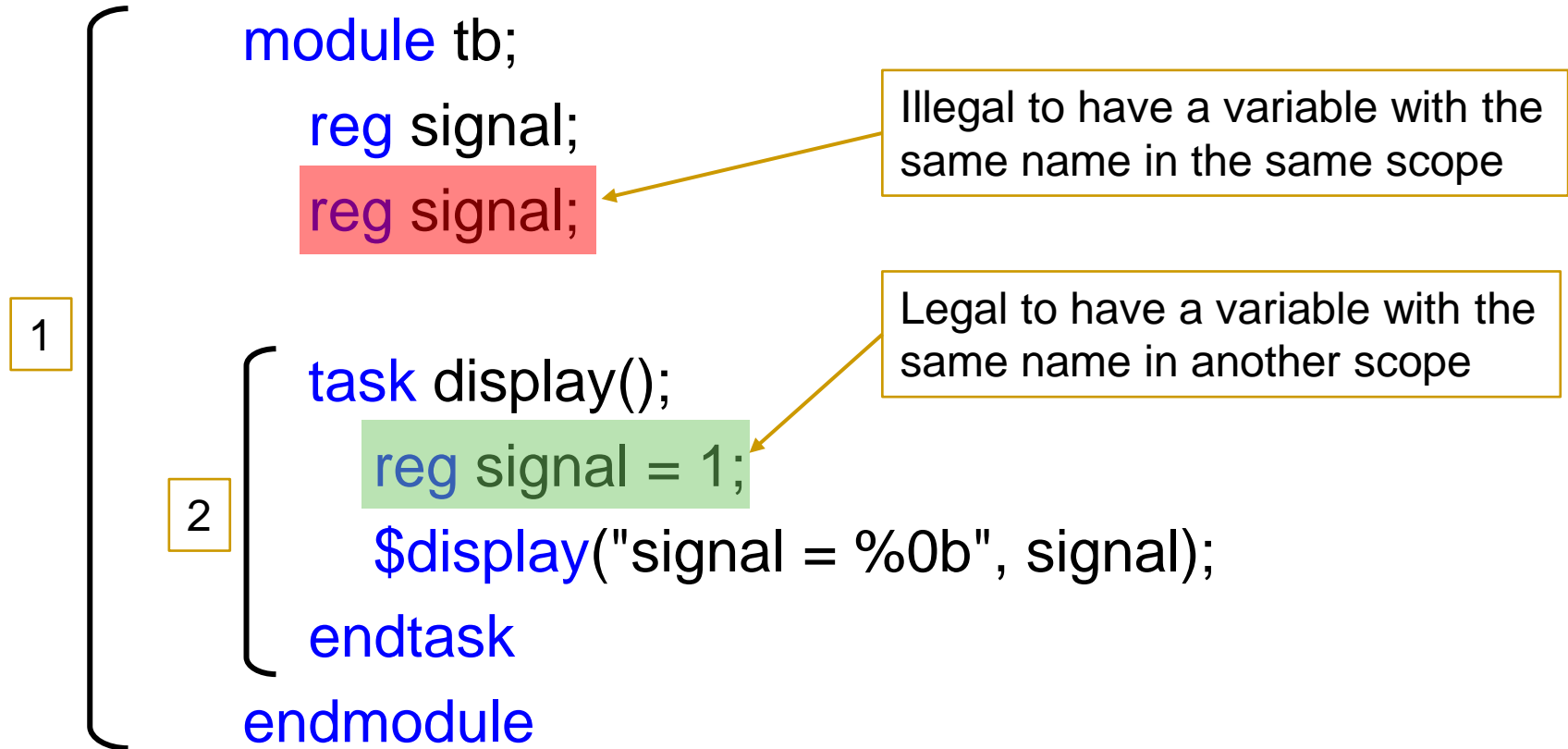
---

```
module tb;  
    reg signal;  
    reg signal;  
  
    task display();  
        reg signal = 1;  
        $display("signal = %0b", signal);  
    endtask  
endmodule
```

# Variable scopes: example



# Variable scopes: example



# Display tasks



# Display tasks

---

- ❑ Display system tasks are mainly used to display informational and **debug** messages to track the flow of **simulation** and to debug faster
- ❑ **\$write**(<params/vars/strs>)
- ❑ **\$display**(<params/vars/strs>)
- ❑ **\$monitor**(<arguments>)

# Display tasks: \$write

- ❑ `$write` prints out the arguments in its parenthesis in the order they appear
  - It doesn't append a newline character at the end

```
module test;  
    reg [7:0] a;  
    initial begin  
        a = 8'b1;  
        $write("The value of the reg is %0b", a);  
    end  
endmodule
```

A format specifier



Result: The value of the reg is 11111111

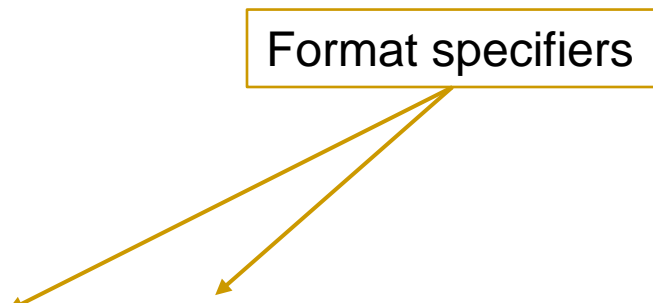
# Display tasks: \$display

- ❑ **\$display** prints out the arguments in its parenthesis in the order they appear
  - it appends a newline character at the end

```
module test;  
    reg [7:0] a;  
    initial begin  
        a = 8'b1;  
        #5 a = 8'b10110011;  
        $display("Display time=%0t, a=%0b", $time, a);  
    end  
endmodule
```

Format specifiers

Result: Display time=5, a=10110011



# Display tasks: \$monitor

- ❑ **\$monitor** prints out the arguments in its parenthesis in the order they appear whenever one of the signals changes
  - It runs as a separate thread and monitors for any signal change

**module** test;

**reg** [7:0] a;

**initial begin**

        a = 8'b1;

**\$monitor**("The value of the reg a is %0b", a);

        #10 a = 8'b11110000;

        #20 a = 8'b10100101;

**end**

**endmodule**

Result: The value of the reg a is 11111111  
          The value of the reg a is 11110000  
          The value of the reg a is 10100101

# Verilog File I/O

# Verilog File I/O

---

- ❑ There are different system tasks and functions that help to open/close, edit, read and write values from/to files in Verilog code.

```
module tb;
```


```
// Declare a variable to store the file handler
```

```
integer fd;
```

```
initial begin
```

```
    fd = $fopen("my_file.txt", "w");
```

Opening a file with “write” permission



```
// Close the file handle pointed to by "fd"
```

```
$fclose(fd);
```

```
end
```

```
endmodule
```

# Verilog File I/O: arguments

Argument	Description
"r" or "rb"	Open for reading
"w" or "wb"	Create a new file for writing. If the file exists, truncate it to zero length and overwrite it
"a" or "ab"	If file exists, append (open for writing at EOF), else create a new file
"r+", "r+b" or "rb+"	Open for both reading and writing
"w+", "w+b" or "wb+"	Truncate or create for update
"a+", "a+b", or "ab+"	Append, or create new file for update at EOF

# Verilog File I/O: functions

Function	Description
<code>\$fdisplay</code>	Similar to <code>\$display</code> , writes out to file instead
<code>\$fwrite</code>	Similar to <code>\$write</code> , writes out to file instead
<code>\$fmonitor</code>	Similar to <code>\$monitor</code> , writes out to file instead

These functions write everything in decimal by default.

There are also the options for printing out hex, oct and binary values:

- `$fdisplayh`
- `$fdisplayo`
- `$fdisplayb`



# Verilog File I/O: writing

---

```
module tb;
```


```
// Declare a variable to store the file handler
```

```
integer fd;
```

```
initial begin
```

```
    fd = $fopen("my_file.txt", "w");
```

Opening a file with “write” permission



```
    $fdisplay(fd, "This line is gonna be printed into my_file");
```

```
// Close the file handle pointed to by "fd"
```

```
    $fclose(fd);
```


```
end
```

```
endmodule
```

# Verilog File I/O: reading

```
module tb;
  reg[8*45:1] str;
  integer fd;
  initial begin
    fd = $fopen("my_file.txt", "r");
    // Keep reading lines until EOF is found
    while (! $feof(fd)) begin
      // Get current line into the variable 'str'
      $fgets(str, fd);
      $display("%0s", str);
    end
    $fclose(fd);
  end
endmodule
```

Opening a file with "read" permission



End Of File



# Verilog Compiler Directives

# Verilog Compiler directives

---

Compiler directives:

- ❑ Direct the compiler to treat the code in a specific way
- ❑ Can be used to control the compilation process
- ❑ Have specific syntax and keywords:  
``<keyword> <flag/macro/empty>`
- ❑ Compiler directives may appear anywhere in the source description, but it **is recommended** that they appear outside a module declaration
- ❑ A directive is effective from the point at which it is declared to the point at which another directive overrides it

# Verilog Compiler directives: include

---

## ❑ Syntax:

``include <filename>`

## ❑ Description:

- The ``include` compiler directive lets you insert the entire contents of a source file into another file during Verilog compilation
- The compilation proceeds as though the contents of the included source file appear in place of the ``include` command
- You can use the ``include` compiler directive to include global or commonly-used definitions and tasks without encapsulating repeated code within module boundaries
- You can use the ``include` compiler directive to include a Verilog file with modules defined in it

# Verilog Compiler directives: define

---

## ❑ Syntax:

``define <macro>`

## ❑ Description:

- This compiler directive is used for defining text macros
- These are normally defined in Verilog file "name.vh", where name can be the module that you are coding
- Since `define is compiler directive, it can be used across multiple files

# Verilog Compiler directives: ifdef, else

---

## ❑ Syntax:

``ifdef <macro> <opt. `else> `endif`

## ❑ Description:

- Optionally includes lines of source code during compilation
- The ``ifdef` directive checks that a macro has been defined, and if so, compiles the code that follows. If the macro has not been defined, the compiler compiles the code (if any) following the **optional** ``else` directive
- You can control what code is compiled by choosing whether to define the text macro, either with ``define`
- The ``endif` directive marks the end of the conditional code

# Verilog Compiler directives: undef

---

## ❑ Syntax:

``undef <macro>`

## ❑ Description:

- The ``undef` compiler directive lets you remove definitions of text macros created by the ``define` compiler directive
- You can use the ``undef` compiler directive to undefine a text macro that you use in more than one file



# Verilog Compiler directives: example

---

```
`include modules.v
`define DEFINED

module test;
    initial begin
        `ifdef DEFINED
            $display("DEFINED is defined");
        `else
            $display("DEFINED is not defined");
        $finish;
    end
endmodule
```