

Peter the Great St. Petersburg Polytechnic University  
Institute of Computer Science and Cybersecurity  
Graduate School of Computer Technologies and Information Systems

**Lecture: Common Mistakes in SystemVerilog**

Subject: Automation of discrete device design (in English)

Completed by student of group 5130901/10101 \_\_\_\_\_ Nepomnyaschij M.T.  
(signature)

Lecturer \_\_\_\_\_ Antonov A.P.  
(signature)

Saint Petersburg

2023

## Table of contents

SystemVerilog Unveiled: Navigating Complexities and Crafting Robust Code.....	4
Introduction to SystemVerilog Complexity .....	4
Part 1.....	4
Wire Assignments and the Evolution of Logic Types .....	4
Testing code and project compiling.....	5
1.    Lecture code .....	5
2.    Given code.....	6
Part 2.....	7
Variable Initialization and Behavioral Nuances .....	7
Testing code and project compiling.....	9
1.    Lecture code .....	9
2.    Given code 1 .....	11
3.    Given code 2.....	12
Part 3.....	14
Global variables.....	14
Testing code and project compiling.....	15
1.    Lecture code .....	15
2.    Given code.....	15
Part 4.....	17
Enumerated types .....	17
Testing code and project compiling.....	18
1.    Lecture code .....	18
2.    Given code.....	18
Part 6.....	20
Equality Operators: Unveiling Double Equals vs. Triple Equals .....	20
Testing code and project compiling.....	21
1.    Lecture code .....	21
2.    Given code.....	22

## List of illustrations

Figure 1 – Wire and Logic types .....	4
Figure 2 - test.sv code listing .....	5
Figure 3 - Result of test.sv compilation .....	5
Figure 4 – sv_mist1.sv code compilation .....	6
Figure 5 – Result of sv_mist1.sv compilation .....	6
Figure 6 – Variable initialization demonstration .....	7
Figure 7 – Static and automatic differences .....	8
Figure 8 – Task calls .....	8
Figure 9 – test2.sv (static) code listing .....	9
Figure 10 – Result of test2.sv (static) compilation .....	9
Figure 11 – test2.sv (automatic) code corrections .....	10
Figure 12 – Result of test2.sv (automatic) compilation .....	10
Figure 13 – sv_mist2_1.sv code listing .....	11
Figure 14 – Result of sv_mist2_1.sv compilation .....	11
Figure 15 – sv_mist2_2.sv code listing .....	12
Figure 16 – Result of sv_mist2_2.sv compilation (1) .....	12
Figure 17 – Result of sv_mist2_2.sv compilation (2) .....	13
Figure 18 – Global variables .....	14
Figure 19 – assignVar.sv code listing .....	15
Figure 20 – Result of assignVar.sv compilation .....	15
Figure 21 – sv_mist3.sv code listing .....	16
Figure 22 – Result of sv_mist3.sv compilation .....	16
Figure 23 – Enumerated types .....	17
Figure 24 – Examples of printing values .....	17
Figure 25 – enumt.sv code listing .....	18
Figure 26 – Result of enum.sv compilation .....	18
Figure 27 – sv_mist4.sv code listing .....	18
Figure 28 – Result of sv_mist4.sv compilation .....	19
Figure 29 – Types of equality operators .....	20
Figure 30 – Vector comparisons .....	20
Figure 31 – equal2.sv code listing .....	21
Figure 32 – Result of equal2.sv compilation .....	21
Figure 33 – sv_mist6.sv code listing .....	22
Figure 34 – Result of sv_mist6.sv compilation .....	22

# SystemVerilog Unveiled: Navigating Complexities and Crafting Robust Code

## Introduction to SystemVerilog Complexity

The lecture initiates a meticulous exploration of SystemVerilog, emphasizing the examination of inherent complexities. The instructional approach focuses on dissecting wire assignments, variable types, and nuanced intricacies surrounding task implementation. The overarching objective is to facilitate a profound comprehension of these complexities, enabling the establishment of best practices and proficiency in SystemVerilog programming.

## Part 1

### Wire Assignments and the Evolution of Logic Types

To set the stage, we delved into the historical context of Verilog, where the roles of wire and reg were distinct and well-defined. Wires were the conduits for connecting component instances, while regs served as local variable storage. With the evolution to SystemVerilog, the term reg has been replaced by logic, ushering in a new era. However, this transition introduces a layer of complexity, as logic not only serves as an equivalent to reg but also encapsulates a data type with four states (1, 0, x, z).

Unlike traditional Verilog, SystemVerilog allows for point-to-point connections using the logic type, thereby reducing the reliance on wires in specific scenarios. This alteration becomes particularly noteworthy in synthesizable designs, where the need for wires diminishes, especially when tri-state connections are absent.

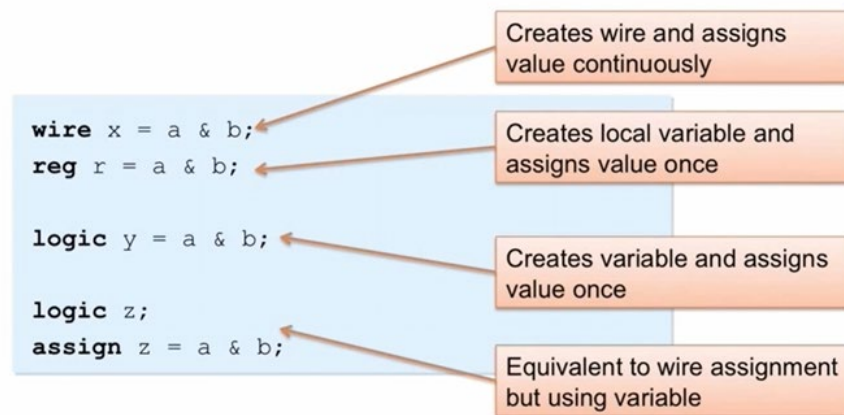


Figure 1 – Wire and Logic types

The image demonstrates different ways to assign values to variables in programming:

- Continuous Assignment (wire): Values update automatically when input variables change (like wires in a circuit).
- Procedural Assignment (reg): Values are assigned within code blocks and only change when explicitly updated.

## Testing code and project compiling

### 1. Lecture code

Automatic variables emerge as a solution, ensuring unique instances for each task invocation, thereby mitigating inadvertent interactions. The simultaneous invocation of tasks from multiple processes is explored, shedding light on how variables behave within diverse scoping contexts.

```
Lecture 3 - Common Mistakes - test.sv

1  module test;
2
3      initial begin
4          $dumpfile("dump.vcd");
5          $dumpvars(0, test);
6          #100;
7          $finish;
8      end
9
10     logic a = 0;
11     logic b = 0;
12
13     always begin
14         a = #5 !a;
15         b = #10 !b;
16     end
17
18     wire x = a & b;
19     reg r = a & b;
20
21     logic y = a & b;
22
23     logic z;
24     assign z = a & b;
25
26 endmodule : test
27
```

Figure 2 - test.sv code listing

By running the code, we can get the wave on the picture below:

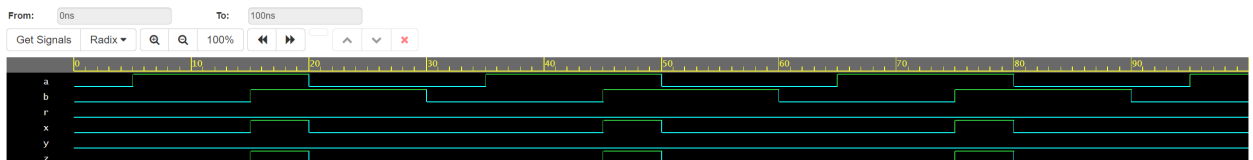


Figure 3 - Result of test.sv compilation

This picture shows that whenever a and b are 1 at the same time the wire value (“x” on the wave) is also 1. The signal “z” acts the same which shows the identical operation of assign and wire logic.

If you want the signal to be just initialized of course you can just assign it like this but if you really do want it to act like the old wire assignments worked in the past you need to create the variable but then assign to the variable, as well.

## 2. Given code

```
lecture 3 - Common Mistakes - sv_mist1.sv

1  `timescale 1ns/1ns
2
3  module sv_mist1();
4
5      // Declare signals a and b
6      logic a;
7      logic b;
8
9      // Combinational logic using wire
10     wire w = a & b;
11
12     // Combinational logic using reg
13     reg r = a & b;
14
15     // SystemVerilog logic type, behaves like reg
16     logic y = a & b;
17
18     // SystemVerilog logic type with assign statement, behaves like wire
19     logic z;
20     assign z = a & b;
21
22     // Embedded test scenario
23     initial begin
24         a = '0;
25         b = '0;
26         // Toggle signal 'a' every 5 time units
27         #5 a = !a;
28         // Toggle signal 'b' every 10 time units
29         #10 b = !b;
30     end
31
32     initial begin
33         $dumpfile("dump.vcd");
34         $dumpvars(0, sv_mist1);
35         #100;
36         $finish;
37     end
38
39 endmodule
40
```

Figure 4 – sv\_mist1.sv code compilation

By compiling the code we'll get the wave which is presented on the picture below:

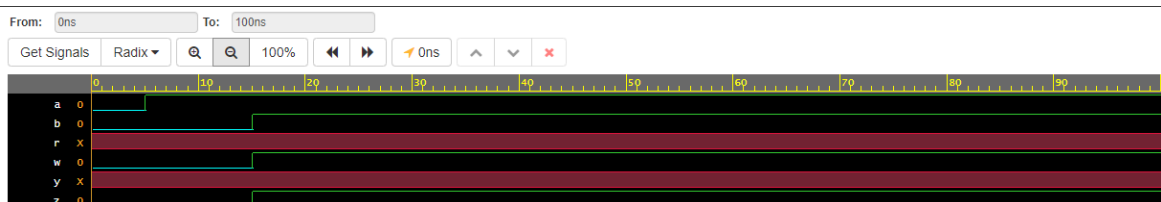


Figure 5 – Result of sv\_mist1.sv compilation

As we can see, the given code is working correctly. So, when both a and b = 1 => w and z have value 1.

## Part 2

### Variable Initialization and Behavioral Nuances



Figure 6 – Variable initialization demonstration

1. Static Variable Initialization (default):
  - This variable is initialized once at compile time.
  - It retains its value across multiple invocations of the block.
  - Useful for maintaining state across function calls or procedural blocks.
2. Automatic Variable Initialization:
  - This variable is initialized each time it's encountered during execution.
  - It's suitable for temporary local storage within a block.
3. No Qualifier (Implicit Static):
  - It is treated as static, but not automatically initialized.
  - The value of `count` is assigned later in the code.

In summary, “static” ensures persistence across invocations, while “automatic” initializes per execution cycle. The third case is static but lacks automatic initialization.

In other words: Static tasks are scrutinized for their potential to create shared variables across multiple invocations, leading to unforeseen interactions. Automatic tasks emerge as a remedy, fostering task-specific variable instances.

```

task static ST;    // default
    int i;        // static
    automatic int j;
    ...
endtask

```

```

task automatic AT;
    int i;        // automatic
    static int j;
    ...
endtask

```

Figure 7 – Static and automatic differences

If you have a static task which again is the default, then any variables in the task itself are also going to be static so "i" in this case is going to be a static variable.

If you call task ST multiple times from multiple always blocks or multiple initial blocks every one of those invocations of ST, they share the same variable "i" so if one invocation updates "i" that updated value will be seen by the other one when it runs.

So that can be sort of non-intuitive trying to reuse a task repeatedly to do some common operation. If you don't want that behavior to happen, you can set a particular variable to be automatic and that means that every invocation of ST in this case will have its own copy of variable "j". okay so that's not shared among the different calls. If you have an automatic task that means that all the variables are automatic by default, so in this case the variable "i" in every invocation of that AT in this case will be unique – they'll have their own copy. But now "j" since it's static will be shared among all the AT implications.

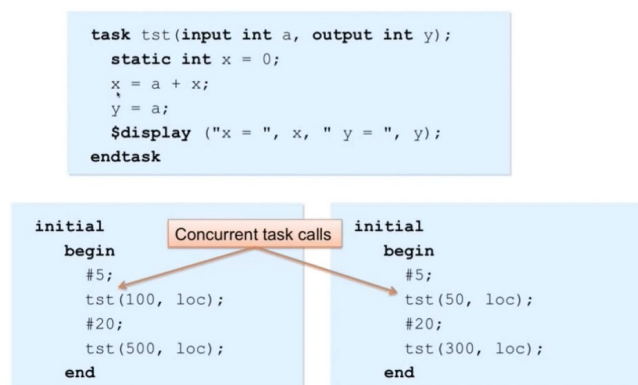


Figure 8 – Task calls

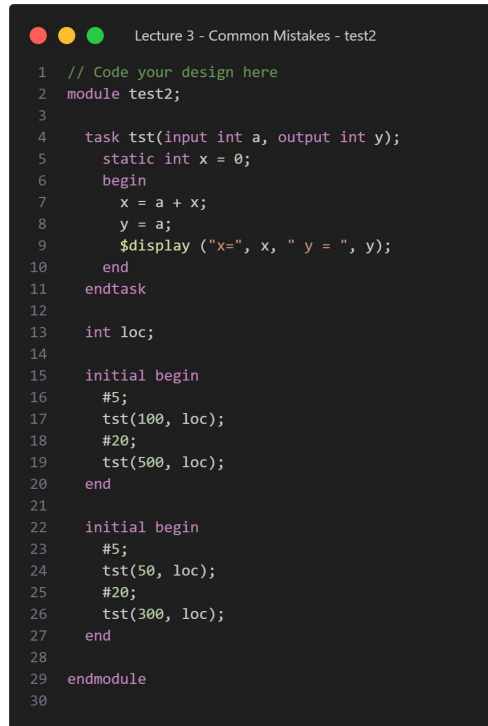
The idea is that we're going to have two concurrent initial blocks, so both are running concurrently in the simulator and then we're going to be calling the test task from each of them and they're going to be at the same time. So, the question is which one is going to be invoked first. That's really tool dependent. The point is that now these two tasks are actually called at time five, but we don't know the order and then that we wait 20 and we'll call them again... This is an example of calling the same task from multiple processes. An initial block serves as a process, and in this demonstration, we observe how a variable is shared between them, particularly when they are static versus automatic.



## Testing code and project compiling

### 1. Lecture code

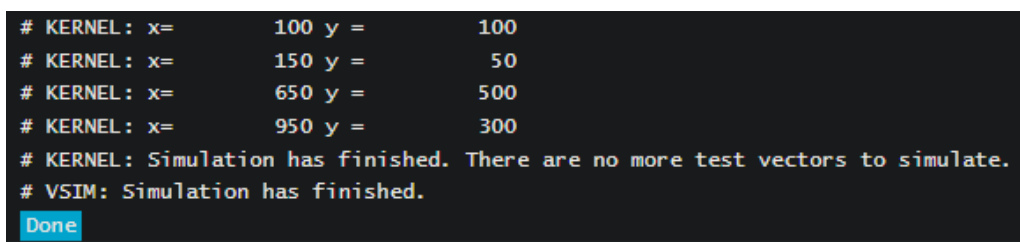
So, we will start by selecting the static variable case. In this case, we have a task named "test" with inputs a and output y, where x is declared as static and initialized to zero. The logic within the task involves updating x and y. Two initial blocks are set up to run this task concurrently.



```
1 // Code your design here
2 module test2;
3
4     task tst(input int a, output int y);
5         static int x = 0;
6         begin
7             x = a + x;
8             y = a;
9             $display ("x=", x, " y = ", y);
10        end
11    endtask
12
13    int loc;
14
15    initial begin
16        #5;
17        tst(100, loc);
18        #20;
19        tst(500, loc);
20    end
21
22    initial begin
23        #5;
24        tst(50, loc);
25        #20;
26        tst(300, loc);
27    end
28
29 endmodule
30
```

Figure 9 – test2.v (static) code listing

Upon execution, with x being static and shared between the two calls, the results demonstrate that the first instance completes with x reaching a hundred, and y also equals a hundred. Subsequently, the other instance runs, producing its outcome.



```
# KERNEL: x=      100 y =      100
# KERNEL: x=      150 y =       50
# KERNEL: x=      650 y =      500
# KERNEL: x=      950 y =      300
# KERNEL: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done
```

Figure 10 – Result of test2.v (static) compilation

So, it ran the first instance with a value of one hundred, and then it executed the second one with 50. You can observe that x retains the value from the initial invocation because it's already at a hundred. Consequently, when we add the 50, it becomes 150. After a 20-unit delay, it appears that, once again, it ran the first initial block. This time, the value of x is 500. Adding this to the previous 150 gives us 650. Subsequently, it executed the second initial block, where another 300 was added.

This demonstration illustrates that the value of x is shared among all the different calls. Consequently, when it gets updated, the successive calls utilize this updated value.

Now let's change static to automatic:

```

5  automatic int x = 0;
6  begin
7      x = a + x;
8      y = a;
9      $display ("x=", x, " y = ", y);
10 end

```

Figure 11 – test2.sv (automatic) code corrections

The result of compilation is shown below:

```

# KERNEL: x=      100 y =      100
# KERNEL: x=       50 y =       50
# KERNEL: x=     500 y =     500
# KERNEL: x=      300 y =      300
# KERNEL: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done

```

Figure 12 – Result of test2.sv (automatic) compilation

It seems that the first call ran first, and as a result, x becomes a hundred. However, in the subsequent call, when it runs at the bottom, it begins with x being zero. This discrepancy arises because it has its own copy of the variable x, designated as automatic. As a result, the x value does not increment in this case since it is not shared. The key takeaway here is that if you desire variables to be shared across different tasks, although this may not be a common scenario, you can declare them as static. On the other hand, if you wish for each task to possess its own unique value, then declaring them as automatic is the appropriate choice.

## 2. Given code 1

```
lecture 3 - Common Mistakes - sv_mist2_1.sv

1 `timescale 1ns/1ns
2 module sv_mist2_1();//Example test2_1 from lecture
3   byte x,y, y_S, y_A;
4   always_comb begin
5     static byte cnt_S = 8'd1;
6     $write ("Static   \ttime[%0t]   \tx=
7     \cnt_S+= x;
8     ",y,$timecnt_S;cnt_S);
9     $display ("\ty_S=
10    \ty_S);
11   always_comb begin
12     automatic byte cnt_A = 8'd1;
13     $write ("Automatic \ttime[%0t]   \ty=
14     \cnt_A+= y;
15     ",y,$timecnt_A;cnt_A);
16     $display ("\ty_A=
17     \ty_A);
18   //embedded test
19   initial begin
20     #0 $display ("");
21
22     #1 y=8'd1;
23     #0 x=8'd1;
24     #0 $display ("");
25
26     #1 y=8'd2;
27     #0 x=8'd2;
28     #0 $display ("");
29
30     #1 y=8'd3;
31     #0 x=8'd3;
32     #0 $display ("");
33
34     #1 $stop;
35   end
36 endmodule
37
```

Figure 13 – sv\_mist2\_1.sv code listing

This code defines a module with two `always_comb` blocks demonstrating the usage of static and automatic variables. The static variable (`cnt_S`) retains its value across multiple calls, while the automatic variable (`cnt_A`) is re-initialized in each call. The code also includes an initial block with a test sequence for variables `x` and `y`, displaying the values and cumulative counts of `cnt_S` and `cnt_A` at each step, and stopping after three iterations.

```
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: Static      time[0]      x=0      cnt_S=1 y_S=1
# KERNEL: Automatic   time[0]      y=0      cnt_A=1 y_A=1
# KERNEL:
# KERNEL: Automatic   time[1]      y=1      cnt_A=1 y_A=2
# KERNEL: Static      time[1]      x=1      cnt_S=1 y_S=2
# KERNEL:
# KERNEL: Automatic   time[2]      y=2      cnt_A=1 y_A=3
# KERNEL: Static      time[2]      x=2      cnt_S=2 y_S=4
# KERNEL:
# KERNEL: Automatic   time[3]      y=3      cnt_A=1 y_A=4
# KERNEL: Static      time[3]      x=3      cnt_S=4 y_S=7
# KERNEL:
# RUNTIME: Info: RUNTIME_0070 design.sv (34): $stop called.
# KERNEL: Time: 4 ns, Iteration: 0, Instance: /sv_mist2_1, Process: @INITIAL#19_20.
# KERNEL: Stopped at time 4 ns + 0.
# VSIM: Simulation has finished.
Done
```

Figure 14 – Result of sv\_mist2\_1.sv compilation

This code is like the test2 code from the lecture, so we can see again that it works well.

### 3. Given code 2

```
lecture 3 - Common Mistakes - sv_mist2_2.sv

1  `timescale 1ns/1ns
2  module sv_mist2_2();
3      byte y_S, y_A;
4      bit CLK;
5      always_ff @(posedge CLK) begin
6          static byte cnt_S = '0;
7          cnt_S += 8'd1;
8          y_S = cnt_S;
9      end
10     always_ff @(posedge CLK) begin
11         automatic byte cnt_A = '0;
12         cnt_A += 8'd1;
13         y_A = cnt_A;
14     end
15     //This is an embedded test
16     always
17         #1 CLK = ~CLK;
18     always @(posedge CLK) begin
19         byte cnt_t;
20         cnt_t += 8'd1;
21         $display ("CLK posedge number[
22 ]"#0c$display ("\ty_A=
23 ",#0$display ("\ty_S=
24 ",#0$display ("");
25     end
26     initial begin
27         repeat (7) @(negedge CLK);
28         $stop;
29     end
30     //
31 endmodule
32
```

Figure 15 – sv\_mist2\_2.sv code listing

This SystemVerilog code above presents a module with two `always_ff` blocks illustrating the use of static and automatic variables in a clocked context. The static variable (`cnt_S`) retains its value across clock cycles, while the automatic variable (`cnt_A`) is re-initialized in each clock cycle. The code includes an embedded test with a clock signal (`CLK`), displaying the values of `y_A` and `y_S` after each positive clock edge. The simulation stops after 7 clock cycles.

The result of compiling is presented on the picture below:

```
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: CLK posedge number[1]
# KERNEL:      y_A=1
# KERNEL:      y_S=1
# KERNEL:
# KERNEL: CLK posedge number[2]
# KERNEL:      y_A=1
# KERNEL:      y_S=2
# KERNEL:
# KERNEL: CLK posedge number[3]
# KERNEL:      y_A=1
# KERNEL:      y_S=3
# KERNEL:
# KERNEL: CLK posedge number[4]
# KERNEL:      y_A=1
# KERNEL:      y_S=4
# KERNEL:
# KERNEL: CLK posedge number[5]
# KERNEL:      y_A=1
# KERNEL:      y_S=5
# KERNEL:
# KERNEL: CLK posedge number[6]
# KERNEL:      y_A=1
# KERNEL:      y_S=6
```

Figure 16 – Result of sv\_mist2\_2.sv compilation (1)

```
# KERNEL:
# KERNEL: CLK posedge number[7]
# KERNEL:     y_A=1
# KERNEL:     y_S=7
# KERNEL:
# RUNTIME: Info: RUNTIME_0070 design.sv (28): $stop called.
# KERNEL: Time: 14 ns, Iteration: 0, Instance: /sv_mist2_2, Process: @INITIAL#26_3@.
# KERNEL: Stopped at time 14 ns + 0.
# VSIM: Simulation has finished.
Done
```

Figure 17 – Result of sv\_mist2\_2.sv compilation (2)

Simulation runs for 7 clock cycles, showing the behavior of static and automatic variables in a clocked context. The embedded test displays cumulative counts of y\_A and y\_S after each positive clock edge. As we can see, all tests passed and the code works well.

## Part 3

### Global variables

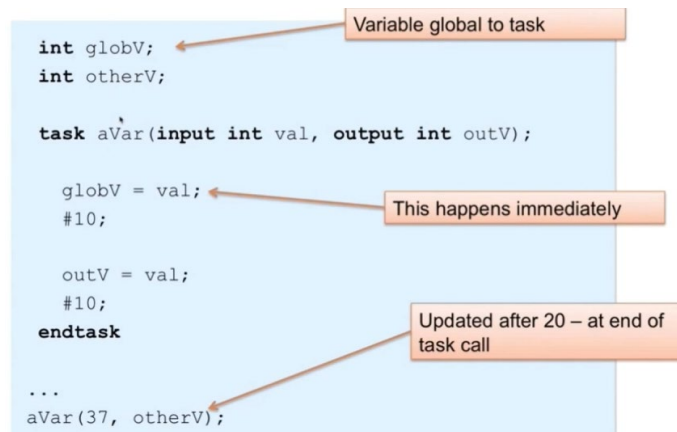


Figure 18 – Global variables

According to the picture above:

- Task `aVar` has an input `val` and an output `outV`.
- Two variables, `globV` and `otherV` (declared globally), are accessible within the task's scope.
- When calling `aVar`, `globV` immediately receives the value of `val`.
- However, `outV` updates after a 10-unit delay, affecting `otherV` only at the task's end.
- This timing discrepancy can be confusing: `outV` doesn't instantly impact `otherV`.

Here we have our code again. We've declared a global variable and another variable. Our task, which takes an input `val` and produces an output `outV`, includes assignments with a 10-unit delay. Additionally, an initial block that calls `aVar` with the decimal value 37 was created.

Subsequently, we make another call to `aVar` with the value 73. In essence, we are running two calls in this scenario. When we execute this code, we can observe the outcomes.

## Testing code and project compiling

### 1. Lecture code

```
lecture 3 - Common Mistakes - assignVar.sv

1 // Code your design here
2 module assignVar;
3
4   int globV;
5   int otherV;
6
7   task aVar(input int val, output int outV);
8     begin
9       globV = val;
10      #10;
11      outV = val;
12      #10;
13    end
14  endtask
15
16  initial begin
17    aVar(37, otherV);
18    #50;
19    aVar(73, otherV);
20  end
21
22  initial begin
23    $dumpfile("dump.vcd");
24    $dumpvars;
25    #200 $finish;
26  end
27
28 endmodule : assignVar
29
```

Figure 19 – assignVar.sv code listing

The result of compilation is the wave below:

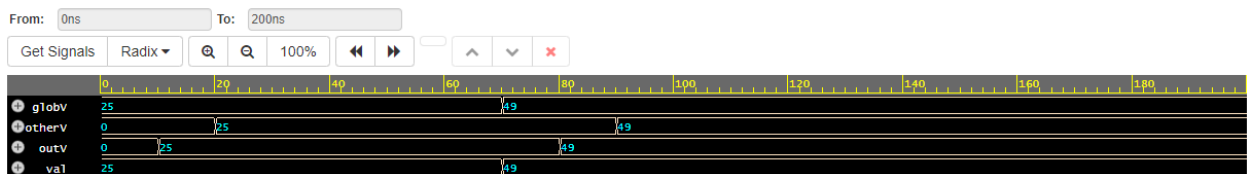


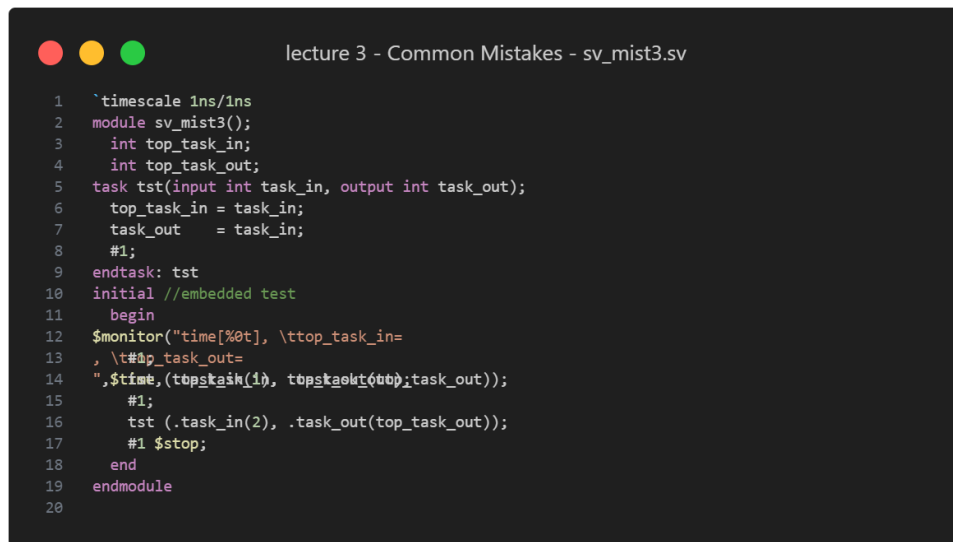
Figure 20 – Result of assignVar.sv compilation

So, we can observe that at time 0, we obtain the value 25. Indeed, if we inspect the moment when this task is called, specifically on 16<sup>th</sup> stroke without any pound delay, it occurs at time 0. Since there is no delay, the assignment to the global variable "globV" happens as the first action in the task, setting it immediately to the value 37, represented as 25 in hexadecimal.

However, a crucial point arises: when does the assignment to "outV" occur? Theoretically, it should happen after 10 units, as per the code. Yet, it waits another 10 units before concluding the task. This temporal distinction is vital to comprehend, as the assignment to "outV" occurs 10 units after the initial 10-unit delay. Indeed, "outV," the internal variable, is set after a delay of 10 units. However, the return value doesn't occur until 25 units. This pattern repeats when we alter the value to 73, which is equivalent to 49 in hexadecimal.

### 2. Given code

As a result, the assignment to global variables within a task happens immediately upon task execution. However, the update of output values or variables designated as output parameters only occurs when the task concludes, not immediately after their assignment.



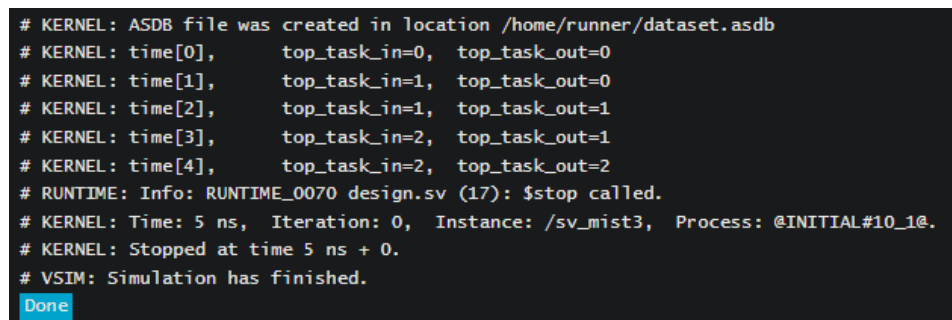
```

1  `timescale 1ns/1ns
2  module sv_mist3();
3      int top_task_in;
4      int top_task_out;
5  task tst(input int task_in, output int task_out);
6      top_task_in = task_in;
7      task_out    = task_in;
8      #1;
9  endtask: tst
10 initial //embedded test
11 begin
12     $monitor("time[%0t], \ttop_task_in=
13     , \ttop_task_out=
14     ", $time, (top_task_in(1), top_task_out(1));
15     #1;
16     tst (.task_in(2), .task_out(top_task_out));
17     #1 $stop;
18 end
19 endmodule
20

```

Figure 21 – sv\_mist3.v code listing

This SystemVerilog code defines a module (sv\_mist3) with a task (tst) that takes an input parameter (task\_in) and outputs a result (task\_out). The initial block serves as an embedded test, calling the task twice with different input values and monitoring the top\_task\_in and top\_task\_out variables. The \$monitor statement displays the time, top\_task\_in, and top\_task\_out during simulation. The simulation stops after the second task execution. Note that the output of the task (task\_out) is assigned to top\_task\_out, and this assignment occurs after a delay of 1 time unit within the task.



```

# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: time[0],      top_task_in=0,  top_task_out=0
# KERNEL: time[1],      top_task_in=1,  top_task_out=0
# KERNEL: time[2],      top_task_in=1,  top_task_out=1
# KERNEL: time[3],      top_task_in=2,  top_task_out=1
# KERNEL: time[4],      top_task_in=2,  top_task_out=2
# RUNTIME: Info: RUNTIME_0070 design.sv (17): $stop called.
# KERNEL: Time: 5 ns, Iteration: 0, Instance: /sv_mist3, Process: @INITIAL#10_1@.
# KERNEL: Stopped at time 5 ns + 0.
# VSIM: Simulation has finished.

```

Figure 22 – Result of sv\_mist3.v compilation

The output on the picture above displays the time, top\_task\_in and top\_task\_out during simulation. The embedded test calls the tst task twice with different inputs (1 and 2). There is a 1-time unit delay after each task invocation, impacting the assignment of top\_task\_out. top\_task\_in is assigned the value of task\_in from the task. top\_task\_out is assigned the value of task\_out from the task after the delay. The simulation stops after the second task execution.



## Part 4

### Enumerated types

There's a variable called "state" with values "idle," "starting," and "stopped," forming an enumerated type. Enumerated types are useful for enhancing code readability, especially in state machines or control systems. The recommended practice is to use a typedef to declare the enumerated type, making it easier to declare variables with clear names like "state\_variable." This approach simplifies code comprehension and management. If you were to print out an enumerated type, you would see the defined values.

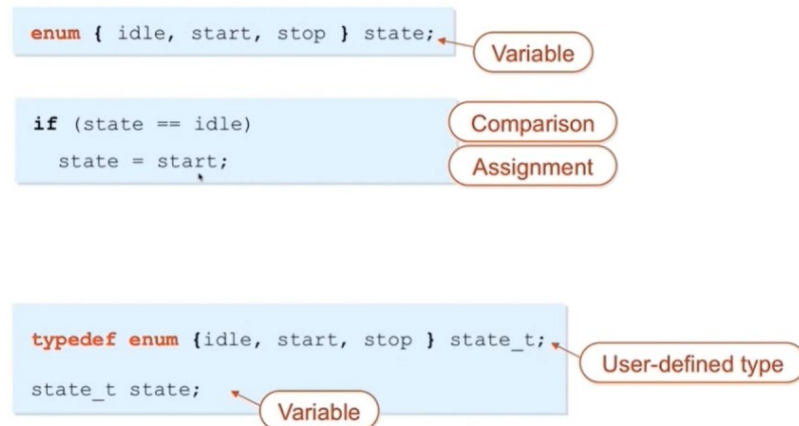


Figure 23 – Enumerated types

When using enumerated types in waveforms, you can visualize values like "idle," "start," and "stop," aiding in debugging state machines. However, when printing these values, the state names aren't automatically displayed. To achieve this, you need to implement a specific trick or method.

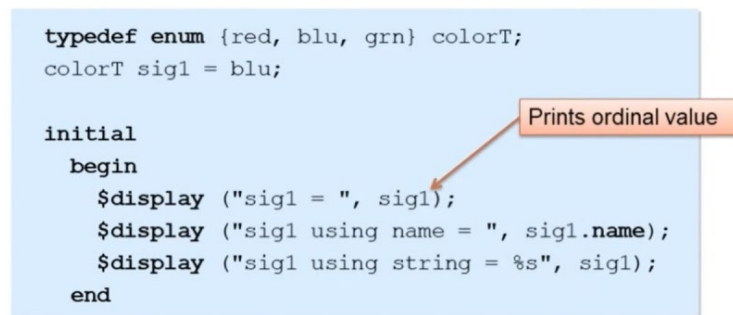


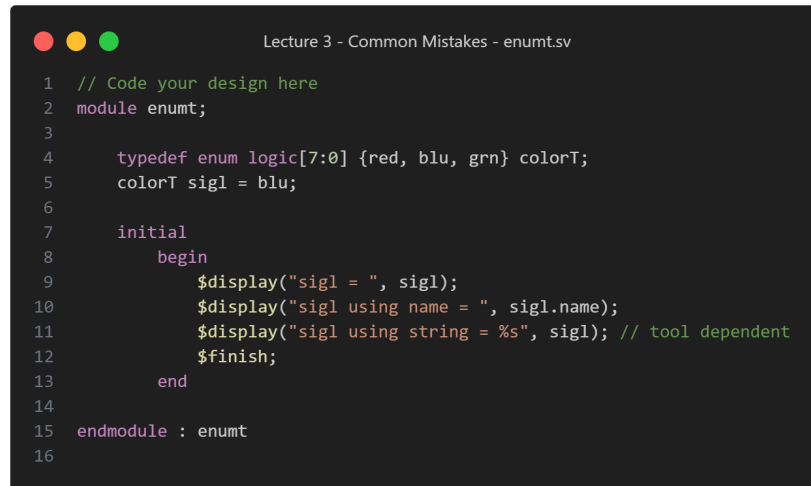
Figure 24 – Examples of printing values

We have a signal defined as a color type with red, green, and blue. We declare a signal named "sig1" of this color type and initialize it to be blue. This is a legal enumerated type, utilizing a typedef. However, when trying to print it using a dollar display, an issue arises. It displays the ordinal value (0 for red, 1 for blue, and 2 for green) instead of the desired red, green, or blue. To print the actual enumerated type of value, you can use "sig1.name" since "sig1" is of the enumerated type. This converts the value into the enumerated type, allowing you to see it printed as red, green, or blue, as intended. While some tools offer string formatting for this purpose, using the enumerated type of function is the recommended and standard approach.

## Testing code and project compiling

### 1. Lecture code

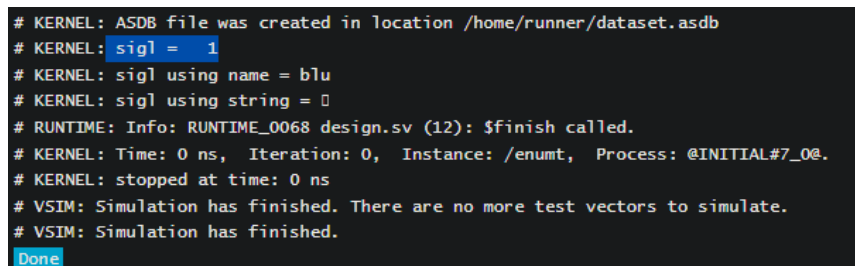
Here's an example:



```
1 // Code your design here
2 module enumt;
3
4     typedef enum logic[7:0] {red, blu, grn} colorT;
5     colorT sigl = blu;
6
7     initial
8     begin
9         $display("sigl = ", sigl);
10        $display("sigl using name = ", sigl.name);
11        $display("sigl using string = %s", sigl); // tool dependent
12    $finish;
13    end
14
15 endmodule : enumt
16
```

Figure 25 – enumt.sv code listing

We have a type defined as red, green, blue. In this case, we have a signal of that type initialized to blue. When we run this, the first printout shows that signal\_one is equal to 1 because the ordinal value of blue is 1 (0 for red, 1 for blue, and 2 for green).

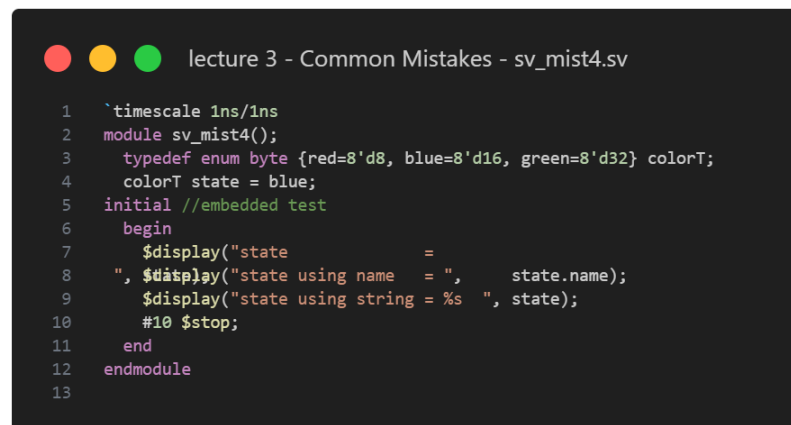


```
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: sigl = 1
# KERNEL: sigl using name = blu
# KERNEL: sigl using string = 0
# RUNTIME: Info: RUNTIME_0068 design.sv (12): $finish called.
# KERNEL: Time: 0 ns, Iteration: 0, Instance: /enumt, Process: @INITIAL#7_0@.
# KERNEL: stopped at time: 0 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done
```

Figure 26 – Result of enum.sv compilation

To display the actual color name, we use "sigma.name," and now we correctly get "blue." The method of formatting as a string may depend on the tool, but following the standard and using the name attribute is the recommended approach.

### 2. Given code



```
1 `timescale 1ns/1ns
2 module sv_mist4();
3     typedef enum byte {red=8'd8, blue=8'd16, green=8'd32} colorT;
4     colorT state = blue;
5     initial //embedded test
6     begin
7         $display("state = ");
8         $display("state using name = ", state.name);
9         $display("state using string = %s", state);
10        #10 $stop;
11    end
12 endmodule
13
```

Figure 27 – sv\_mist4.sv code listing

The module defines an enumerated type colorT with values red, blue, and green, each assigned a byte value. The variable state is initialized to blue. In an embedded test, information about state is displayed using decimal, name, and string formats.

```
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: state = 16
# KERNEL: state using name = blue
# KERNEL: state using string = []
# RUNTIME: Info: RUNTIME_0070 design.sv (10): $stop called.
# KERNEL: Time: 10 ns, Iteration: 0, Instance: /sv_mist4, Process: @INITIAL#5_0@.
# KERNEL: Stopped at time 10 ns + 0.
# VSIM: Simulation has finished.
Done
```

Figure 28 – Result of sv\_mist4.sv compilation

This code is like the lecture's enumt.sv code, so it's clear that it works well.

The output displays information about the state variable in different formats during simulation: decimal value, name attribute, and string representation. The code demonstrates the use of an enumerated type (colorT) with symbolic names ('red', 'blue', 'green'). The simulation stops after a delay of 10-time units.

## Part 6

### Equality Operators: Unveiling Double Equals vs. Triple Equals

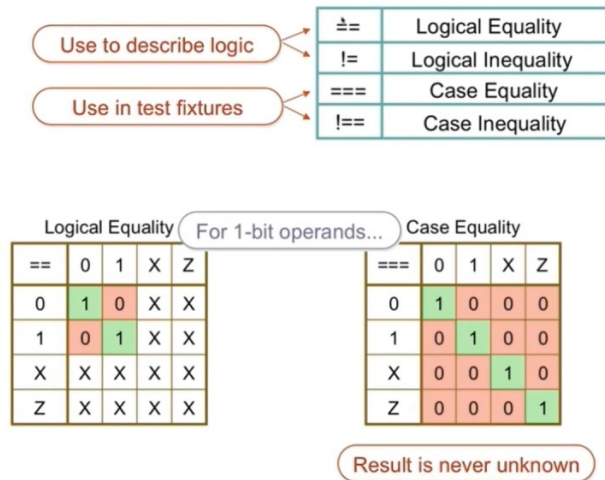


Figure 29 – Types of equality operators

The next mistake is the equality operators in SystemVerilog. There are double equals ( $==$ ), exclamation point equals ( $!=$ ), and triple equals ( $===$ ), along with their not equal counterparts. The double equals checks for zeros and ones, resulting in an unknown value for cases like "0x" or "x0." This is acceptable for logical purposes but not ideal for checking design outputs in a test bench. For precise matching, especially in output checking, triple equals and exclamation point double equal are recommended. They ensure an exact match, and the result is never unknown.

The issue with unknown results is illustrated in vector comparisons:

<code>4'b0011 == 4'b0110</code>	<code>1'b0</code>
<code>4'b0011 == 4'b0XX1</code>	<code>1'bX</code>
<code>4'b0110 == 4'b0XX1</code>	<code>1'b0</code>
<code>4'b0XX1 == 4'b0XX1</code>	<code>1'bX</code>

<code>4'b0011 === 4'b0110</code>	<code>1'b0</code>
<code>4'b0011 === 4'b0XX1</code>	<code>1'b0</code>
<code>4'b0110 === 4'b0XX1</code>	<code>1'b0</code>
<code>4'b0XX1 === 4'b0XX1</code>	<code>1'b1</code>

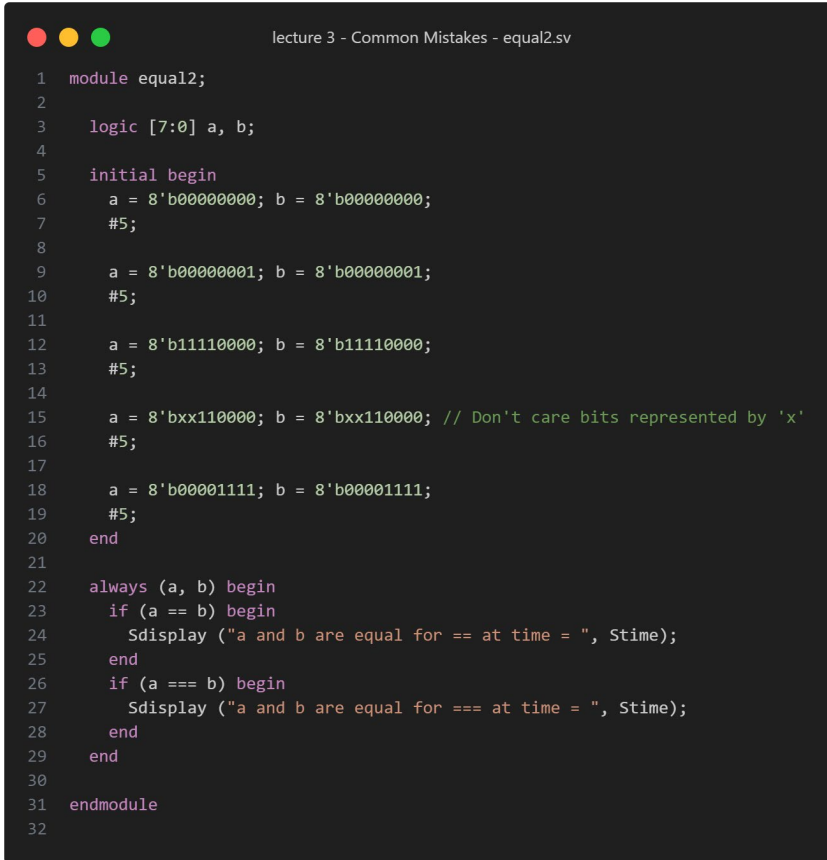
<code>if (output != expected)</code> <code>\$display("Error");</code>	Not executed if output is unknown
<code>if (output !== expected)</code> <code>\$display("Error");</code>	Executed if output is unknown

Figure 30 – Vector comparisons

For instance, with "0011" and "0xx1," the double equals might yield an unknown result, potentially masking errors in output checking. Using exclamation point double equal ensures failure when comparing designs with unexpected 'x' values.

## Testing code and project compiling

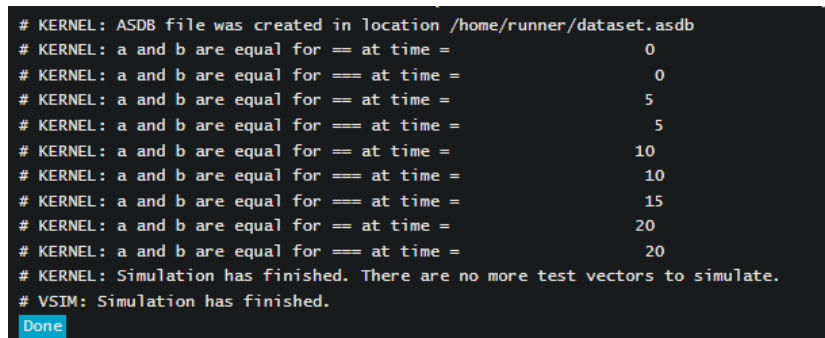
### 1. Lecture code



```
lecture 3 - Common Mistakes - equal2.sv
1  module equal2;
2
3      logic [7:0] a, b;
4
5      initial begin
6          a = 8'b00000000; b = 8'b00000000;
7          #5;
8
9          a = 8'b00000001; b = 8'b00000001;
10         #5;
11
12         a = 8'b11110000; b = 8'b11110000;
13         #5;
14
15         a = 8'bxx110000; b = 8'bxx110000; // Don't care bits represented by 'x'
16         #5;
17
18         a = 8'b00001111; b = 8'b00001111;
19         #5;
20     end
21
22     always (a, b) begin
23         if (a == b) begin
24             $display ("a and b are equal for == at time = ", $time);
25         end
26         if (a === b) begin
27             $display ("a and b are equal for === at time = ", $time);
28         end
29     end
30
31 endmodule
32
```

Figure 31 – equal2.sv code listing

In the provided example on the picture above, the comparison with triple equals provides accurate results, whereas double equals may fail to trigger the display due to unknown results. This demonstrates the importance of choosing the appropriate equality operator for precise and reliable output checking in SystemVerilog.



```
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: a and b are equal for == at time = 0
# KERNEL: a and b are equal for === at time = 0
# KERNEL: a and b are equal for == at time = 5
# KERNEL: a and b are equal for === at time = 5
# KERNEL: a and b are equal for == at time = 10
# KERNEL: a and b are equal for === at time = 10
# KERNEL: a and b are equal for === at time = 15
# KERNEL: a and b are equal for == at time = 20
# KERNEL: a and b are equal for === at time = 20
# KERNEL: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done
```

Figure 32 – Result of equal2.sv compilation

As a result, in vector comparisons, when using triple equals (===), it accurately compares even if there are 'x's present. However, with double equals (==), if 'x's are involved, the result becomes unknown, and the comparison doesn't trigger the if statement for further action, potentially leading to overlooked errors in output checking. Choosing the appropriate equality operator is crucial for precise and reliable results in SystemVerilog.

## 2. Given code

```
lecture 3 - Common Mistakes - sv_mist6.v

1 `timescale 1ns/1ns
2 module sv_mist6();
3     logic [7:0] a = '0, b='0;
4     always_comb begin
5         $display ("%t", $time);
6         if (a == b) $display ("Double Equal for a(%b) == b(%b) is TRUE ", a, b);
7         else $display ("Double Equal for a(%b) == b(%b) is FALSE ", a, b);
8         if (a === b) $display ("Tripple Equal for a(%b) === b(%b) is TRUE ", a, b);
9         else $display ("Tripple Equal for a(%b) === b(%b) is FALSE ", a, b);
10    end
11    always_comb begin
12        if (a != b) $display ("          a(%b) != b(%b) is TRUE ", a, b);
13        if (a !== b) $display ("          a(%b) !== b(%b) is TRUE ", a, b);
14    end
15    initial begin//embedded test
16        #1;
17        a = '0; b = '1;
18        #1;
19        a = 8'bxx110000; b = 8'bxx110000;
20        #1;
21        a = 8'bxx110000; b = 8'b11110000;
22        #1;
23        $stop;
24    end
25 endmodule
26
```

Figure 33 – sv\_mist6.v code listing

This code is like equal2.v file from the lecture one. After its compilation, we get the result presented on the picture below:

```
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL:
# KERNEL: time[0]
# KERNEL: Double Equal for a(00000000) == b(00000000) is TRUE
# KERNEL: Tripple Equal for a(00000000) === b(00000000) is TRUE
# KERNEL:
# KERNEL: time[1]
# KERNEL: Double Equal for a(00000000) == b(11111111) is FALSE
# KERNEL: Tripple Equal for a(00000000) === b(11111111) is FALSE
# KERNEL:          a(00000000) != b(11111111) is TRUE
# KERNEL:          a(00000000) !== b(11111111) is TRUE
# KERNEL:
# KERNEL: time[2]
# KERNEL: Double Equal for a(xx110000) == b(xx110000) is FALSE
# KERNEL: Tripple Equal for a(xx110000) === b(xx110000) is TRUE
# KERNEL:
# KERNEL: time[3]
# KERNEL: Double Equal for a(xx110000) == b(11110000) is FALSE
# KERNEL: Tripple Equal for a(xx110000) === b(11110000) is FALSE
# KERNEL:          a(xx110000) != b(11110000) is TRUE
# RUNTIME: Info: RUNTIME_0070 design.v (23): $stop called.
# KERNEL: Time: 4 ns, Iteration: 0, Instance: /sv_mist6, Process: @INITIAL#15_2@.
# KERNEL: Stopped at time 4 ns + 0.
# VSIM: Simulation has finished.
Done
```

Figure 34 – Result of sv\_mist6.v compilation

The output shows the results of comparisons between logic vectors `a` and `b` using double equals (`==`) and triple equals (`===`) operators, as well as inequality using `!=` and `!==` operators. The code illustrates the differences in behavior between these operators in SystemVerilog. The embedded test sets different values for `a` and `b`, showcasing the outcomes of the comparisons. The simulation stops after the embedded test.