SystemVerilog Procedural Blocks, Tasks & Functions

Introduction

With **Day 1**, the conceptual foundations of modern verification were established. With **Day 2**, we learned how SystemVerilog models **data and execution semantics**.

Day 3 marks the transition from data modeling to behavioral abstraction.

In real verification environments, engineers do not write long monolithic blocks of code. Instead, behavior is **decomposed, reused, and parameterized** using:

- Procedural blocks
- Tasks
- Functions

This phase explains **how SystemVerilog executes behavior over time**, how logic is reused safely, and why certain constructs (like ref) must be handled carefully.

This phase is critical before moving into:

- Interfaces
- Clocking blocks
- Testbench architectures

Scope of Day 3

Day 3 focuses on **procedural abstraction and execution control**, covering:

- task vs function
- void functions
- Argument passing mechanisms
- Automatic vs static variables
- Time literals and delays

1. Procedural Blocks in SystemVerilog

Procedural blocks define when and how code executes.

1.1 initial Block

- Executes once, starting at simulation time 0
- Commonly used in testbenches

Example:

```
initial begin
  $display("Executed once at time %0t", $time);
end
```

1.2 always Block

- Executes repeatedly
- Legacy construct
- Does not express intent clearly

Example:

```
always #5 clk = ~clk;
```

1.3 always_comb

- Used for combinational logic
- Automatic sensitivity list
- Prevents incomplete assignments

Example:

```
always_comb begin
  y = a & b;
end
```

Preferred over always @(*)

1.4 always_ff

- Used for sequential logic
- Enforces flip-flop semantics
- Detects illegal usage

Example:

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

Required for clean RTL and verification models.

System Verilog Code example showing how actually Verilog and system Verilog differing.

4 bit adder code using Verilog:

```
module registered_adder (
  input        clk,
  input        rst,
  input  [3:0] a,
  input  [3:0] b,
  output reg [4:0] sum
);

  // Internal combinational signal
  reg [4:0] sum_comb;

  // Combinational logic
  always @(*) begin
    sum_comb = a + b;
  end

  // Sequential logic (registered output)
  always @(posedge clk) begin
    if (rst)
      sum <= 5'd0;
    else
      sum <= sum_comb;
  end
endmodule
```

4 bit adder code using System Verilog:

```
module registered_adder (
  input  logic        clk,
  input  logic        rst,
  input  logic [3:0]  a,
  input  logic [3:0]  b,
  output logic [4:0]  sum
);

  logic [4:0] sum_comb;

  // Combinational logic
  always_comb begin
    sum_comb = a + b;
```

```
  end

  // Sequential logic (registered output)
  always_ff @(posedge clk) begin
    if (rst)
      sum <= 5'd0;
    else
      sum <= sum_comb;
  end

endmodule
```

Comparison of the 2 codes:

Feature	Verilog Implementation	SystemVerilog Implementation	Why SystemVerilog Is Better
Signal declaration	reg, separate from wire	logic	logic removes reg/wire confusion
Output type	output reg [4:0] sum	output logic [4:0] sum	Unified data type, safer
Combinational block	always @(*)	always_comb	Auto sensitivity + latch protection
Sensitivity handling	Tool-dependent	Language-enforced	Prevents missing signals
Sequential block	always @(posedge clk)	always_ff @(posedge clk)	Enforces flip-flop behavior
Assignment safety	Blocking allowed by mistake	Blocking disallowed	Prevents RTL bugs
Multiple drivers	Allowed silently	Compile-time error	Early bug detection
Intent clarity	Generic always	Intent-specific blocks	Self-documenting code
Debug effort	Higher	Lower	Errors caught early
Industry usage	Legacy support	Industry standard	Modern RTL methodology

2.1 Functions

Functions are used to compute and return a value.

Key Rules

- Must return a value
- Cannot consume time
- Used for pure computation

Example:

```
function int add(int a, int b);
  return a + b;
endfunction
```

2.2 Tasks

Tasks are used to model **procedures that may consume time**.

Key Rules

- Do not return a value (directly)
- Can include delays
- Used for sequences and operations

Example:

```
task drive_signal(output logic sig);
  sig = 1'b1;
  #10;
  sig = 1'b0;
endtask
```

2.3 Task vs Function Summary

Feature	**Function**	**Task**
Returns value	Yes	No
Time control	No	Yes
Delay allowed	No	Yes
Typical use	Computation	Behavior

3. Void Functions

SystemVerilog allows functions that **do not return a value** using void.

Example:

```
function void log_msg(string msg);
  $display("LOG: %s", msg);
Endfunction
```

Why void functions exist

- Clean logging
- Debug helpers
- Configuration routines

Avoids misusing tasks when no timing is involved.

4. Argument Passing Mechanisms

SystemVerilog provides **four ways** to pass arguments.

4.1 input (Default)

- Passed by value
- Read-only inside the task/function

Example:

```
function int square(input int x);
  return x * x;
endfunction
```

4.2 output

- Value assigned inside procedure
- Returned to caller

Example:

```
task set_flag(output logic f);
  f = 1'b1;
endtask
```

4.3 ref (Pass-by-Reference)

- Direct access to caller variable
- No local copy

Example:

```
task increment(ref int x);
  x = x + 1;
endtask
```

Danger of ref

- Can modify data unexpectedly
- Makes debugging harder
- Breaks encapsulation

Use ref only when performance or behavior demands it.

4.4 const ref

- Read-only reference
- No copy, but safe

Example:

```
function int sum(const ref int a[], int size);
  int total = 0;
  foreach (a[i]) total += a[i];
  return total;
endfunction
```

Preferred over ref for arrays and large objects.

5. Automatic vs Static Variables

5.1 Automatic Variables

- New instance created on each call
- Default for functions

Example:

```
function int counter();
  int count = 0;
  count++;
  return count;
endfunction
```

Each call returns the same value.

5.2 Static Variables

- Shared across calls
- Retains value

Example:

```
function int static_counter();
  static int count = 0;
  count++;
  return count;
endfunction
```

5.3 Comparison Summary

Feature	Automatic	Static
Lifetime	Per call	Entire simulation
Retains value	No	Yes
Thread safe	Yes	No

6. Time Literals and Delays

SystemVerilog supports **explicit time units** for clarity.

```
#10ns;
#5ps;
```

Advantages

- Improved readability
- Reduced timing ambiguity
- Cleaner waveform analysis

Preferred over raw numeric delays.

System Verilog Code Examples:

1. Task vs Function

```
module task_vs_function;

  function int add(int a, int b);
    $display("Function executed at %0t", $time);
    return a + b;
  endfunction

  task delay_task();
    #5;
    $display("Task executed at %0t", $time);
  endtask

  initial begin
    $display("Sum = %0d", add(3,4));
    delay_task();
  end

endmodule
```

Output :

```
# Function executed at 0
# Sum = 7
# Task executed at 5
```

2. ref

```
module ref_example;

  task modify(ref int x);
    x = x + 10;
  endtask

  initial begin
    int a = 5;
    modify(a);
    $display("a = %0d", a);
  end

endmodule
```

Output :   a = 15.

Note : If you are facing any compile issue, use automatic keyword in task.

For example, in the above code task modify (ref int x); is replaced with task automatic modify(ref int x);

Key Takeaways – Day 3

- Tasks model **behavior over time**
- Functions model **pure computation**
- void functions improve structure
- ref enables performance but risks correctness
- const ref is safer for large data
- Automatic vs static affects reentrancy
- Time literals improve clarity and safety

What comes Next ?

Interfaces, Clocking Blocks