Functions in SystemVerilog

SystemVerilog functions are fundamental constructs for encapsulating reusable blocks of procedural code, primarily used for combinatorial logic and calculations. They offer a structured and modular approach to both hardware design and verification. While sharing conceptual similarities with C programming language functions, SystemVerilog functions possess specific characteristics tailored to the context of Hardware Description Languages (HDLs).

Core Principles and Syntax

The fundamental syntax for defining a function in SystemVerilog is indeed quite similar to that in C, promoting intuitive code organization and readability.

General Syntax Structure:

```
function [return_type] function_name ([input_or_ref_arguments]);
   // Local variable declarations (optional)
   // Procedural statements (e.g., assignments, if-else, case)
   // return value; (mandatory if a non-void return_type is specified)
endfunction
```

- **function / endfunction**: These keywords explicitly define the beginning and end of a function block.
- **[return_type] (Optional)**: Specifies the data type of the value that the function will return. If this is omitted, the function implicitly returns a 1-bit logic value. For functions that do not inherently return a value (similar to void in C), the return_type can be omitted, or the return value can be assigned directly to the function_name itself within the function body.
- **function_name**: The unique identifier for the function.
- **[input_or_ref_arguments] (Optional)**: A comma-separated list of arguments that the function accepts. These arguments allow data to be passed into the function.
    - **input (Default)**: Arguments are passed by value. A local copy of the argument's value is created, and any modifications within the function do not affect the original variable outside the function's scope. This is the most common and safest method, particularly for synthesizable design.
    -
    - Code:-

```
function int add_values(input int val_a, input int int val_b);
   // val_a and val_b are local copies
   return val_a + val_b;
endfunction
```

- **ref (Reference)**: Arguments are passed by reference, meaning the function operates directly on the original variable. Modifications inside the function will affect the variable outside. While useful in specific verification scenarios (e.g., modifying state in a scoreboard), ref arguments should be used judiciously in synthesizable code due to their implications for hardware realization.
- Code:-

```
function void increment_counter(ref int count);
   count = count + 1; // Modifies the original 'count' variable
endfunction
```

Calling Functions

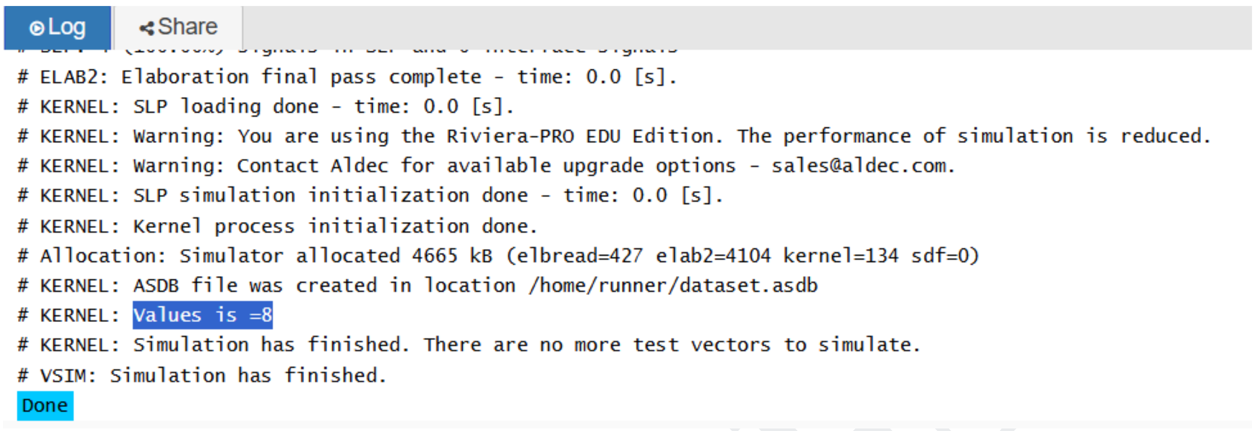
Functions in SystemVerilog are invoked in a manner identical to C functions: by their name followed by parentheses containing the actual arguments (if any). The returned value can be assigned to a variable.

Here's a practical example demonstrating an addition function within a simple testbench:

```
module tb;
  // This function takes two 5-bit inputs and returns their sum as a 6-bit value
  function bit[5:0] add(input bit [4:0] a,b);
    return a+b; // Performs the addition
  endfunction

  bit [5:0] hiii; // Declare a 6-bit variable to store the result

  initial begin
    // Call the 'add' function with 5'b00010 (2 decimal) and 5'b00110 (6 decimal)
    hiii = add(5'b00010, 5'b00110);
    // Display the resulting value in decimal format
    $display("Values is =%0d", hiii); // Expected output: Values is =8
  end
endmodule
```

This example clearly shows how the add function is defined with a specified return type (bit[5:0]) and input arguments (bit [4:0]). It then demonstrates how this function is called from within an initial block, just as you would call a function in C, and its result is used.

Output:-



```
# ELAB2: Elaboration final pass complete - time: 0.0 [s].
# KERNEL: SLP loading done - time: 0.0 [s].
# KERNEL: Warning: You are using the Riviera-PRO EDU Edition. The performance of simulation is reduced.
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 4665 kB (elbread=427 elab2=4104 kernel=134 sdf=0)
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: Values is =8
# KERNEL: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done
```

Here's another example that demonstrates a multiplication function and conditional output based on the result:

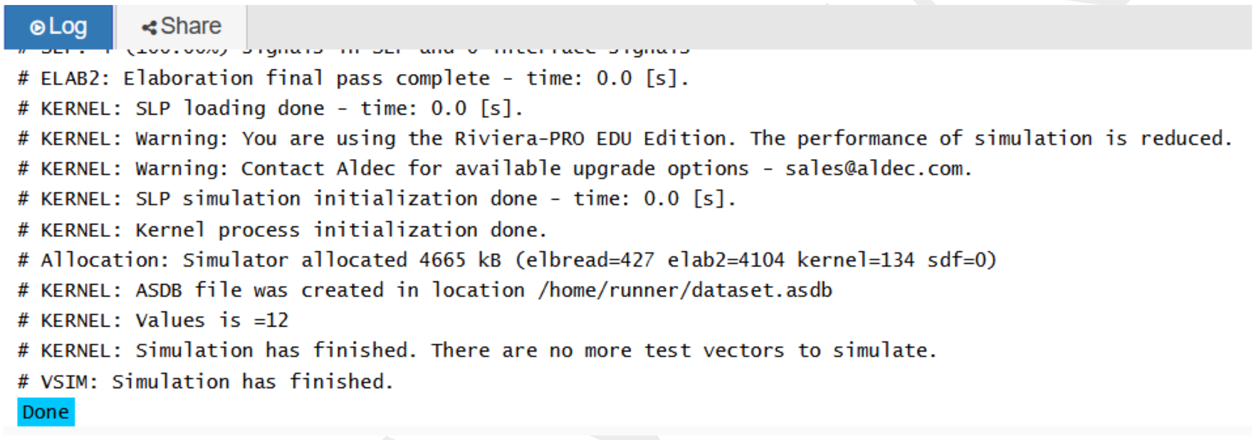
This SystemVerilog code defines a testbench module tb. Inside this module, a function named add (though it performs multiplication) is declared. This function takes two 5-bit inputs, a and b, and returns their product as a 12-bit value.

An initial block then executes a sequence of operations:

1. A 12-bit variable hiii is declared.
2. The add function is called with 5'b00010 (decimal 2) and 5'b00110 (decimal 6). The result of their multiplication (2 * 6 = 12) is assigned to hiii.
3. An if-else statement checks if the value of hiii is equal to 12.
   - If true, it displays "Values is =12".
   - If false, it displays "Values not is =X" (where X is the actual value of hiii).

```
module tb;
  // This function takes two 5-bit inputs and returns their product as a 12-bit
value
  function bit[11:0] add(input bit [4:0] a,b);
    return a*b; // Performs the multiplication
  endfunction

  bit [11:0] hiii; // Declare a 12-bit variable to store the result
  initial begin
    // Call the 'add' function with 5'b00010 (2 decimal) and 5'b00110 (6 decimal)
    hiii = add(5'b00010, 5'b00110);

    // Check the result and display accordingly
    if(hiii == 12)
      $display("Values is =%0d", hiii); // Expected output: Values is =12
    else
      $display("Values not is =%0d", hiii);
  end
endmodule
```

Output:-



```
# ELAB2: Elaboration final pass complete - time: 0.0 [s].
# KERNEL: SLP loading done - time: 0.0 [s].
# KERNEL: Warning: You are using the Riviera-PRO EDU Edition. The performance of simulation is reduced.
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 4665 kB (elbread=427 elab2=4104 kernel=134 sdf=0)
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: Values is =12
# KERNEL: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
```

And a more comprehensive example showcasing different argument types and function calls:

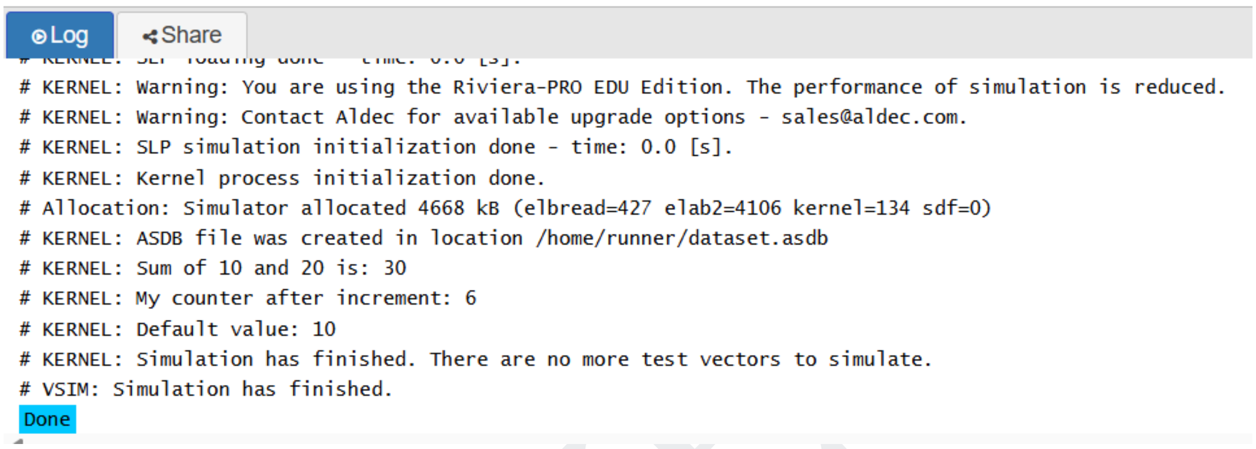
```
module example_module;

  int data_a = 10;

  int data_b = 20;

  int sum_result;

  int my_counter = 5;

  int fixed_val; // Declare outside initial block



  initial begin

    // Function with return value
    sum_result = add_values(data_a, data_b);
    $display("Sum of %0d and %0d is: %0d", data_a, data_b, sum_result);

    // Function with ref and automatic
    increment_counter(my_counter);
    $display("My counter after increment: %0d", my_counter);

    // Function without arguments
    fixed_val = get_default_value();
    $display("Default value: %0d", fixed_val);
end

// Function returning int
function int add_values(input int a, input int b);
    return a + b;
endfunction

// Function using 'ref' must be 'automatic'
function automatic void increment_counter(ref int count);
    count = count + 1;
endfunction

// Simple function without arguments
function int get_default_value();
```

return 10;

endfunction

Endmodule

Output:-



```
# KERNEL: SLP loading done - time: 0.0 [s].
# KERNEL: Warning: You are using the Riviera-PRO EDU Edition. The performance of simulation is reduced.
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 4668 kB (elbread=427 elab2=4106 kernel=134 sdf=0)
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: Sum of 10 and 20 is: 30
# KERNEL: My counter after increment: 6
# KERNEL: Default value: 10
# KERNEL: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done
```

Key Characteristics and Strict Limitations

1. **Combinatorial Execution (Zero Time):** This is the most critical characteristic. SystemVerilog functions execute in **zero simulation time**. They are designed purely for combinatorial logic, where outputs are a direct and immediate consequence of inputs. This property makes them suitable for description of hardware elements that have no sequential behavior.
2. **No Timing Constructs Allowed**: Directly stemming from their combinatorial nature, functions **MUST NOT** contain any timing control statements. This includes:
   - **Delays (#)**: e.g., #10, #(delay_val)
   - **Event Controls (@)**: e.g., @(posedge clk), @(reset)
   - **wait statements**: e.g., wait (condition)
   - **fork-join blocks**: These imply parallel execution and potential timing interaction.

   Any attempt to include these constructs within a function will result in a compilation error, as they violate the fundamental principle of zero-time execution. This restriction is crucial for ensuring that functions can be accurately synthesized into pure combinational hardware.
3. **Use of $display and Other System Tasks**: As you noted, SystemVerilog functions can freely use system tasks like $display, $monitor, $error, $fatal, $info, etc. These are invaluable for debugging, status reporting, and

general simulation control. Their execution does not introduce simulation time.

4. **Single Return Value**: Functions are primarily designed to compute and return a single value. While modifications to ref arguments are possible, the main purpose is to produce a result that can be used in an expression.
5. **Automatic Storage (Default)**: By default, all variables declared within a function (including its arguments) have *automatic storage*. This means memory is allocated for them upon entry into the function and deallocated upon exit. Each call to the function gets its own fresh set of variables. This contrasts with tasks, which default to static storage (variables persist across calls unless explicitly declared automatic).

Applications

SystemVerilog functions are indispensable in various aspects of digital design and verification:

- **Synthesizable Design**: Implementing logic that maps directly to combinatorial gates, such as:
    - Arithmetic units (adders, multipliers)
    - Data encoders/decoders
    - Parity generators/checkers
    - Multiplexers
- **Verification Environments**: For utility and procedural tasks within testbenches, including:
    - Calculating expected values in scoreboards.
    - Implementing constraint functions for constrained random verification.
    - Processing and formatting data for display or logging.
    - Determining coverage hit points.

Conclusion

SystemVerilog functions provide a robust and efficient way to encapsulate combinatorial logic and calculations. Their C-like syntax fosters familiarity and ease of use, making them an accessible construct for many engineers. However, understanding their strict zero-time execution model and the absolute prohibition of timing constructs is paramount. Adhering to these principles ensures that functions are correctly interpreted by synthesis tools for hardware implementation and accurately modeled in simulation for verification purposes.