Mastering SystemVerilog Argument Passing

Discover the three key techniques - Pass By Value, Pass By Reference, and Pass By Constant Reference - and how they impact your code's behavior, memory usage, and performance.

Pass By Value

Protect your data with an independent copy.

1

Pass By Reference

Unlock direct access, but tread carefully.

2

Pass By Constant Reference

Combine the best of both worlds.

3

Pass By Value

1 Independent Local Copy

Pass by value creates a separate copy of the data for the function to work with, leaving the original untouched.

2 Data Integrity

The original data is protected from unintended modification by the called function.

3 Potential Overhead

Creating a local copy can consume memory and impact performance, especially with large data types.

Pass By Value: Practical Examples

The following examples illustrate the fundamental behavior of Pass By Value, emphasizing how the original variables remain unaffected by operations performed on their copies within tasks.

Example 1: Basic Arithmetic Operation with Input and Output

```
module tb;
  bit [3:0] a, b, y; // 'a' and 'b' here are separate from the task's 'a' and 'b'
  initial begin
    $dumpfile("tb.vcd");
    $dumpvars(0, tb);
  end

  // Pass By Value Task: 'input' specifier explicitly defines pass by value.
  // 'output' means 'y' will receive the result from the task.
  task add(input a, input b, output y);
    y = a + b; // Operates on local copies of a, b, and assigns to the output 'y'
    $display("Time: %0t | Inside task: A = %0d, B = %0d, Y = %0d", $time, a, b, y);
  endtask

  initial begin
    // Values 3 and 4 are passed by value to the 'add' task.
    add(3, 4, y);
    // 'a' and 'b' in this scope were never assigned, so they retain their default value (0 for bit)
    $display("Time: %0t | Outside task: value of a = %0d and b = %0d, Y = %0d", $time, a, b, y);
  end
endmodule
```

Output:

Time: 0 | Inside task: A = 3, B = 4, Y = 7

Time: 0 | Outside task: value of a = 0 and b = 0, Y = 7

As observed, the variables **a** and **b** in the top-level module remain at their default uninitialized values (0 for bit type), despite the task **add** receiving 3 and 4 as inputs. This clearly demonstrates that the task operates on copies.

Example 2: Swapping Values (No Effect on Original)

```
module tb;
  task swap ( input bit [1:0] a, [1:0] b);
    bit [1:0] temp;
    temp = a;
    a = b;
    b = temp;
    $display("Time: %0t | Inside swap: Value of a : %0d and b : %0d", $time, a, b);
  endtask

  initial begin
    $dumpfile("tb.vcd");
    $dumpvars(0, tb);
    $display("Time: %0t | Before swap call: Passing 1 and 2 by value.", $time);
    swap(1, 2);
    $display("Time: %0t | After swap call: Original values (if variables were passed) would be
unaffected.", $time);
  end
endmodule
```

Output:

Time: 0 | Before swap call: Passing 1 and 2 by value.
Time: 0 | Inside swap: Value of a : 2 and b : 1
Time: 0 | After swap call: Original values (if variables were passed) would be unaffected.

Even though **a** and **b** are swapped inside the **swap** task, these changes are confined to the local copies. If actual variables (e.g., **int x = 1, y = 2;** then **swap(x, y);**) were passed instead of literals, **x** and **y** would retain their initial values after the task call, unequivocally proving the "no modification to original" characteristic of pass by value.

Pass By Reference

1 Direct Access to Original Data

With pass by reference, you're working with the actual original data, not a copy. Any changes you make directly affect the original.

2 Explicit Reference Keyword

Use the **ref** keyword to explicitly pass arguments by reference. This allows functions to serve as true "setters" or "updaters" for external data.

3 Automatic Functions

Declare functions as **automatic** to ensure local variables and arguments are allocated on the stack, supporting re-entrancy and preventing conflicts.

Pass By Reference: Examples with Scalar and Array Types

These examples demonstrate the direct impact of pass by reference on original variables, highlighting its utility for modification and efficiency for arrays.

Example 3: Modifying Integers Directly

```
module tb;
  bit [3:0] dummy_a, dummy_b; // Used for VCD dumping, not directly involved in 'add' task
  initial begin
    $dumpfile("tb.vcd");
    $dumpvars(0, tb);
  end

  // Automatic task with ref arguments: modifications to 'a' and 'b' affect the original variables.
  task automatic add(ref int a, ref int b);
    $display("Time: %0t | Inside task (before modification): a = %0d, b = %0d", $time, a, b);
    a = a + 5; // Directly modifies the original 'x'
    b = b + 5; // Directly modifies the original 'y'
    $display("Time: %0t | Inside task (after modification): a = %0d, b = %0d", $time, a, b);
  endtask

  initial begin
    int x = 3;
    int y = 4;
    $display("Time: %0t | Before task call: x = %0d and y = %0d", $time, x, y);
    add(x, y); // x and y are passed by reference
    $display("Time: %0t | After task call: value of x = %0d and y = %0d", $time, x, y);
  end
endmodule
```

Output:
Time: 0 | Before task call: x = 3 and y = 4
Time: 0 | Inside task (before modification): a = 3, b = 4
Time: 0 | Inside task (after modification): a = 8, b = 9
Time: 0 | After task call: value of x = 8 and y = 9

This output clearly shows that the changes made to **a** and **b** inside the **add** task persist in **x** and **y** outside the task, confirming direct modification of the original variables.

Pass By Reference with Arrays

Passing arrays by reference is a common and highly efficient practice in SystemVerilog, especially for large arrays, as it completely bypasses the memory and time overhead of copying the entire array. This is invaluable in verification environments where large data structures are frequently manipulated.

Example 4: Initializing a Small Array Using ref

```
module tb;
  bit[3:0] array[16];
  function automatic void array_in(ref bit[3:0] a[16]);
    for(int i=0; i <= 15; i++) begin
      a[i]=i;
    end
    $display("Time: %0t | Inside function: Array after initialization (first two): a[0]=%0d, a[1]=%0d",
$time, a[0], a[1]);
  endfunction

  initial begin
    $dumpfile("waveform.vcd");
    $dumpvars(0, tb);
    $display("Time: %0t | Before array_in call: array[0]=%0d (initial value)", $time, array[0]);
    array_in(array);
    $display("Time: %0t | After array_in call: Original array updated. Displaying full array:", $time);
    for(int i=0; i < array.size(); i++) begin
      $display("Time: %0t | array[%0d]=%0d", $time, i, array[i]);
    end
  end
endmodule
```

Output:
Time: 0 | Before array_in call: array[0]=0 (initial value)
Time: 0 | Inside function: Array after initialization (first two): a[0]=0, a[1]=1
Time: 0 | After array_in call: Original array updated. Displaying full array:
Time: 0 | array[0]=0
Time: 0 | array[1]=1
...
Time: 0 | array[15]=15

Pass By Reference: Large Array Initialization and Efficiency

The true efficiency benefits of Pass By Reference become evident when working with larger data structures. The following example demonstrates initializing a larger array, underscoring how **ref** avoids significant copy overhead. Note the critical correction in loop bounds to prevent out-of-bounds access, a common pitfall in array manipulation.

Example 5: Initializing a Larger Array Using ref

```
module tb;
  bit[10:0] array[32]; // Declare a larger unpacked array of 32 elements (indices 0-31)
  function automatic void array_in(ref bit[10:0] a[32]);
    for(int i=0; i < a.size(); i++) begin // CORRECTED LOOP BOUND
      a[i]=(i*8); // Directly modifies the original 'array'
    end
    $display("Time: %0t | Inside function: Array after initialization (first two): a[0]=%0d, a[1]=%0d",
$time, a[0], a[1]);
  endfunction

  initial begin
    $dumpfile("waveform.vcd");
    $dumpvars(0, tb);
    $display("Time: %0t | Before array_in call: array[0]=%0d (initial value)", $time, array[0]);
    array_in(array); // Pass the 'array' by reference
    $display("Time: %0t | After array_in call: Original array updated. Displaying full array:", $time);
    for(int i=0; i < array.size(); i++) begin // CORRECTED LOOP BOUND
      $display("Time: %0t | array[%0d]=%0d", $time, i, array[i]);
    end
  end
endmodule
```

Output:

Time: 0 | Before array_in call: array[0]=0 (initial value)
Time: 0 | Inside function: Array after initialization (first two): a[0]=0, a[1]=8
Time: 0 | After array_in call: Original array updated. Displaying full array:
Time: 0 | array[0]=0
Time: 0 | array[1]=8
...
Time: 0 | array[31]=248

This example vividly illustrates how a function, by taking a reference, can efficiently populate or modify a large array declared in a different scope without incurring the overhead of copying all 32 elements. This performance gain is particularly critical in large-scale SystemVerilog verification environments where extensive data manipulation is common.

Pass By Constant Reference

const ref combines the efficiency of pass by reference with the data integrity of pass by value. Functions receive a direct reference to the original variable, but are forbidden from modifying it.

This is useful for reading or processing large data without altering it, like calculating a checksum or searching an array. It avoids the overhead of copying data while ensuring the original remains protected.

Syntax for Pass By Constant Reference

```
function void display_array(const ref int arr[]);
  // arr[0] = 99; // ERROR: This line would cause a compilation error
  // ... read operations on arr are allowed
endfunction
```

Example: Using const ref with an Array

```
module ConstRefArrayExample;
  int data_array[3] = '{10, 20, 30};

  function void process_array_const(const ref int arr[]);
    $display("Time: %0t | Inside function (const ref): Values are %p", $time, arr);
    // arr[0] = 99; // ERROR: This line would cause a compilation error
  endfunction

  initial begin
    $dumpfile("waveform.vcd");
    $dumpvars(0, ConstRefArrayExample);
    $display("Time: %0t | Before function call: %p", $time, data_array);
    process_array_const(data_array);
    $display("Time: %0t | After function call: %p", $time, data_array);
  end
endmodule
```

Output:

Time: 0 | Before function call: '{10, 20, 30}
Time: 0 | Inside function (const ref): Values are '{10, 20, 30}
Time: 0 | After function call: '{10, 20, 30}

The output confirms that the **data_array** remains unchanged, demonstrating the read-only nature enforced by **const ref**.