TASKS vs FUNCTIONS - COMPLETE SYSTEMVERILOG GUIDE

PART 1: BASIC DEFINITIONS

FUNCTION

- A subroutine that RETURNS A VALUE. In Verilog it cannot be void but SystemVerilog supports void functions
- Executes in ZERO simulation time
- Cannot contain timing controls (#, @, wait)
- Used for COMPUTATIONS

TASK

- A subroutine will not return value(s) explicitly as functions but using ports we can achieve the return
- Can consume simulation time
- Can contain timing controls (#, @, wait)
- Can have multiple outputs
- Used for PROCEDURES/SEQUENCES

QUICK COMPARISON TABLE

Feature	Function	Task
Return Value	Exactly one (If there is no explicit return mentioned then implicit return will be done via function name)	Multiple or None (strictly no explicit return but similar behavior can be achieved via output ports)
Arguments	Only input (SV: input, output, inout, ref)	input, output, and inout, ref
Timing Control	None allowed (Immediate)	Allowed (#, @, wait)
Call Method	Used in an expression: `a = f(b)`	Used as a statement: `t(a, b)`
Assignments	Only Blocking (=)	Both = and <= allowed
Can Call...	Other functions only	Tasks and functions

IMPORTANT NOTES

Note 1: Return Keyword - Verilog vs SystemVerilog

In classic Verilog (Verilog-95 and Verilog-2001), there is no `return` keyword.

Instead, Verilog uses **implicit return** - you return a value by assigning it to a variable that has the same name as the function itself.

How "Return" Works in Verilog

When you declare a function, the compiler automatically creates a hidden internal register with the function's name. To return a value, you must assign your result to that name before the function ends.

The return Keyword in SystemVerilog

If you use SystemVerilog (extension .sv), the `return` keyword is available. It allows you to exit a function early and return a value immediately, much like in C or Java.

Feature	Verilog (Standard)	SystemVerilog
`return` Keyword	No	Yes
How to return	Assign to function name	Use `return` or assign to name
Early Exit	Not possible (must reach endfunction)	Possible using `return`
Void Functions	Not supported	Supported (`function void`)

Note 2: Function Definition - Common Mistakes

Example Code:

```
module test (input a, output b);

    function buffer (input ain);
        buffer = ain;
    endfunction

    assign b = buffer(a);

endmodule
```

Important Details:

Detail 1: Since `buffer` is simply the return variable, don't write `buffer ain` (assuming return ain).

❌ **WRONG:** `buffer ain;`

✅ **CORRECT:** `buffer = ain;`

You must assign the value to the function name.

Detail 2: The Problem with `assign` inside a function

In Verilog, the `assign` keyword is used for **Continuous Assignments** at the module level (outside of procedural blocks `always` and `initial`).

❌ **You CANNOT use the keyword `assign` inside a function**

✅ Inside a function, you must use a **Blocking Procedural Assignment**, which is just the `=` sign by itself.

Detail 3: Width Specification

By default, if you don't specify a bit-width for the function or its input, Verilog assumes 1-bit.

If you want to handle a 4-bit bus:

```
function [3:0] buffer (input [3:0] ain);
    buffer = ain;
endfunction
```

Note 3: Function Arguments - Must Be Declared

Common Mistake:

```
module test (input a, output b);

    function buffer (a);  // ✕ WRONG!
        buffer = a;
    endfunction

    assign b = buffer(a);

endmodule
```

The Problem:

In your code `function buffer (a)`, you have listed the name `a`, but you have not told Verilog what it is (its direction and type).

Even though the module has an input named `a`, the function cannot see it directly—you must define it specifically inside the function.

Functions do not "inherit" the ports of the module. You must explicitly declare the input.

The Solution:

✕ **Wrong:** `function buffer (a)`

✅ **Right (ANSI style):** `function buffer (input a);`

✅ **Right (Old Style):**

```
function buffer;
    input a;
    begin
        buffer = a;
    end
endfunction
```

Corrected Code (Modern ANSI Style):

```
module test (input a, output b);

    // You MUST specify 'input' inside the parentheses
    function buffer (input ain);
        buffer = ain;
    endfunction
```

```
    assign b = buffer(a);

endmodule
```

Best Practice:

Give different names (like `ain` inside the function) to keep your code readable and avoid "shadowing" bugs.

Note 4: Function/Task Location

In Verilog, the function definition should be **inside the module**. You cannot directly define the function outside the module. If you want to do so, other techniques need to be incorporated. Similarly for SystemVerilog (techniques will be discussed later).

PART 2: BASIC EXAMPLES

```
module basic_examples;

    // FUNCTION EXAMPLE - Returns a value
    function int add(int a, int b);
        return a + b;  // Returns result
    endfunction

    // TASK EXAMPLE - Can have delays
    task display_message(string msg);
        #10;  // Can have delay
        $display("[Time=%0t] %s", $time, msg);
    endtask

    initial begin
        int result;

        $display("\n========== FUNCTION vs TASK BASICS ==========\n");

        // Using FUNCTION
        result = add(5, 3);  // Can use in expressions
        $display("Function result: %0d", result);

        // Using TASK
        display_message("Hello from task");  // Consumes time
        $display("After task at time %0t", $time);
    end

endmodule
```

PART 3: KEY DIFFERENCES (Demonstrated)

```
module key_differences;

    // ======================================
    // DIFFERENCE 1: Return Value
    // ======================================
```

```
// Function MUST return a value in Verilog
// (In SystemVerilog it can be void)
function int multiply(int a, int b);
    return a * b;  // Must return
endfunction

// Task doesn't return value directly (uses output args)
task multiply_task(int a, int b, output int result);
    result = a * b;  // Uses output argument
endtask

// ========================================
// DIFFERENCE 2: Timing Controls
// ========================================

// Function CANNOT have delays
function int func_no_delay(int x);
    // #10;  ERROR! Functions can't have delays
    return x * 2;
endfunction

// Task CAN have delays
task task_with_delay(int x, output int result);
    #10;  // OK in tasks
    result = x * 2;
endtask

// ========================================
// DIFFERENCE 3: Multiple Outputs
// ========================================

// Function returns ONE value only
function int func_one_output(int a, int b);
    return a + b;  // Only one return value
endfunction

// Task can have MULTIPLE outputs
task task_multiple_outputs(int a, int b,
                           output int sum,
                           output int diff,
                           output int prod);
    sum = a + b;
    diff = a - b;
    prod = a * b;
endtask

// ========================================
// DIFFERENCE 4: Usage in Expressions
// ========================================

initial begin
    int x, y, sum, diff, prod;

    $display("\n========== KEY DIFFERENCES ==========\n");

    // Function can be used in expressions
    x = multiply(3, 4) + 10;  // Can use in expression
    $display("Function in expression: x = %0d", x);

    // Task CANNOT be used in expressions
    multiply_task(3, 4, y);   // But separate statement
    // x = multiply_task(3,4) + 10;  ERROR!
    $display("Task result: y = %0d", y);
```

```
        // Multiple outputs from task
        task_multiple_outputs(10, 3, sum, diff, prod);
        $display("Task multiple outputs: sum=%0d, diff=%0d, prod=%0d",
                 sum, diff, prod);
    end

endmodule
```

PART 4: VERILOG vs SYSTEMVERILOG DIFFERENCES

```
module verilog_vs_systemverilog;

    // ========================================
    // ENHANCEMENT 1: Void Functions (SV only)
    // ========================================

    // SV: Can have void functions (don't return value)
    function void print_header(string title);
        $display("==================");
        $display("%s", title);
        $display("==================");
    endfunction

    // Verilog: Would need to use task for this

    // ========================================
    // ENHANCEMENT 2: Default Arguments (SV only)
    // ========================================

    // SV: Can have default argument values
    function int power(int base, int exp = 2);  // Default exp = 2
        int result = 1;
        repeat(exp) result = result * base;
        return result;
    endfunction

    // Verilog: No default arguments

    // ========================================
    // ENHANCEMENT 3: Pass by Reference (SV only)
    // ========================================
  basics of referencing the variables :
```

To make this simple, let's compare **Pass by Value** (the default) and **Pass by Reference** using a real-world analogy: **Editing a Document.**

---

1. Pass by Value (The "Photocopy" Method)

When you pass by value, it is like making a photocopy of your homework and giving it to a friend. If your friend scribbles on the photocopy, **your original paper stays clean.**

Example:

Code snippet

function void increment_val(int a); // Standard "Pass by Value"
    a = a + 1;

```
endfunction

// In your testbench:
int x = 10;
increment_val(x);
$display(x); // Result: 10 (The original 'x' did not change!)
```

---

2. Pass by Reference (The "Google Doc" Method)

When you pass by reference (ref), it is like sharing a **Google Doc link** with your friend. You both are looking at and editing the **exact same file**. If they change a sentence, you see that change immediately in the original document.

Example:

Code snippet

```
// Note the 'ref' and 'automatic' keywords
function automatic void increment_ref(ref int a);
    a = a + 1;
endfunction

// In your testbench:
int x = 10;
increment_ref(x);
$display(x); // Result: 11 (The original 'x' WAS modified!)
```

Key Differences in this Example

Feature	increment_val(x)	increment_ref(x)
What is passed?	A copy of the number 10.	The "address" of the variable x.
Memory	Two separate variables exist.	Only one variable exists in memory.
Side Effect	Safe; x cannot be touched.	Powerful; the function can change x.

Why use ref? (The "Big Array" Case)

Imagine you have an array containing **1,000 integers**.

- **By Value:** SystemVerilog has to copy all 1,000 numbers into the function. This is slow and eats up your computer's RAM.
- **By Reference:** SystemVerilog just passes **one single pointer** (the address). It's like saying, "The data is over there in Box #500." It is nearly instant.

Important Syntax Rules to Remember:

1. **automatic:** You must use the keyword automatic in your function/task header if you use ref.
2. **No Constants:** You cannot do increment_ref(10); because "10" is a constant and doesn't have a memory address you can "reference." You must pass a variable like x.

BASICS :
SystemVerilog Task and Function Argument Passing

1. Argument Passing Overview

SystemVerilog provides several means for passing arguments to functions and tasks, allowing for flexible data manipulation and memory efficiency:

* Argument pass by value
* Argument pass by reference
* Argument pass by name
* Argument pass by position
* Default argument values

2. Argument Pass by Value

In argument pass by value, the mechanism works by copying each argument into the subroutine area. If any changes are made to arguments within the subroutine, those changes will **not** be visible outside the subroutine. This is the default behavior in Verilog and SystemVerilog.

Example: Pass by Value
In this example, variables `x` and `y` are passed as arguments to the function `sum`. Changes to the argument `x` within the function are not visible outside.

```
systemverilog
module argument_passing;
  int x, y, z;

  // Function to add two integer numbers
  function int sum(int x, y);
    x = x + y;
    return x + y;
  endfunction

  initial begin
    x = 20;
    y = 30;
    z = sum(x, y);

$display("----------------------------------------------------------------------");
    $display("\tValue of x = %0d", x); // Output: 20
    $display("\tValue of y = %0d", y); // Output: 30
    $display("\tValue of z = %0d", z); // Output: 80

$display("----------------------------------------------------------------------");
  end
endmodule
```

3. Argument Pass by Reference

In pass by reference, a reference (pointer) to the original argument is passed to the subroutine. As the argument within the subroutine is pointing to the original memory location, any changes to the argument within the subroutine **will** be visible outside. To indicate pass by reference, the argument declaration is preceded by the keyword `ref`.

Note: Subroutines using `ref` must be declared as `automatic`.

Example: Pass by Reference

In this example, changes to the argument `x` within the function are visible outside the function because it is passed by reference.

```
systemverilog
module argument_passing;
  int x, y, z;

  // Function using pass by reference
  function automatic int sum(ref int x, input int y);
    x = x + y;
    return x + y;
  endfunction

  initial begin
    x = 20;
    y = 30;
    z = sum(x, y);

$display("----------------------------------------------------------------------");
    $display("\tValue of x = %0d", x); // Output: 50
    $display("\tValue of y = %0d", y); // Output: 30
    $display("\tValue of z = %0d", z); // Output: 80

$display("----------------------------------------------------------------------");
  end
endmodule
```

4. Argument Pass by Reference with the `const` Keyword

Any modifications to the argument value in a pass by reference can be avoided by using the `const` keyword before `ref`. Any attempt to change the argument value inside the subroutine will lead to a compilation error. This allows you to pass large data structures efficiently without the risk of accidental modification.

Example: Const Reference Error

```
systemverilog
module argument_passing;
  int x, y, z;

  function automatic int sum(const ref int x, input int y);
    x = x + y; // ERROR: x is declared as const and cannot be modified
    return x + y;
  endfunction
endmodule
```

5. Default Argument Values

A default value can be specified for the arguments of a subroutine. In the subroutine call, arguments with a default value can be omitted. If a value is passed to an argument that has a default, the new value overrides the default.

Example: Default Values

```
systemverilog
module argument_passing;
  int q;

  function int sum(int x=5, y=10, z=20);
    return x + y + z;
```

```
  endfunction

  initial begin
    q = sum( , , 10); // x and y use defaults (5, 10), z is overridden to 10

$display("----------------------------------------------------------------------");
    $display("\tValue of q = %0d", q); // Output: 25

$display("----------------------------------------------------------------------");
  end
endmodule
```

6. Argument Pass by Name

In argument pass by name, arguments can be passed in any order by specifying the name of the subroutine argument using a period `.` prefix. This is useful for long argument lists to ensure clarity.

Example: Pass by Name

```
systemverilog
module argument_passing;
  int x;
  string y;

  function void display(int x, string y);
    $display("\tValue of x = %0d, y = %0s", x, y);
  endfunction

  initial begin
    // Arguments are passed out of order using their names
    display(.y("Hello World"), .x(2016));
  end
endmodule
```

```
    // SV: ref argument (pass by reference)
    function void swap(ref int a, ref int b);
        int temp = a;
        a = b;
        b = temp;
    endfunction

    // Verilog: Only pass by value for functions

    // ========================================
    // ENHANCEMENT 4: Return Statement Anywhere (SV only)
    // ========================================

    // SV: Can return from anywhere
    function int find_max(int arr[]);
        if (arr.size() == 0) return -1;  // Early return

        int max = arr[0];
        foreach(arr[i])
            if (arr[i] > max) max = arr[i];
        return max;
    endfunction

    // Verilog: Must assign to function name
```

```
    // ========================================
    // ENHANCEMENT 5: Automatic Functions (SV only)
    // ========================================

    // SV: Automatic storage (reentrant/recursive)
    function automatic int factorial(int n);
        if (n <= 1) return 1;
        return n * factorial(n-1);  // Recursive
    endfunction

    // Verilog: Functions are static by default, not reentrant

    initial begin
        int a = 5, b = 10, arr[] = '{3, 7, 2, 9, 1};

        $display("\n========== SYSTEMVERILOG ENHANCEMENTS ==========\n");

        // Void function
        print_header("SV Features Demo");

        // Default arguments
        $display("power(3) = %0d", power(3));      // Uses default exp=2
        $display("power(3,3) = %0d", power(3, 3)); // Explicit exp=3

        // Pass by reference
        $display("\nBefore swap: a=%0d, b=%0d", a, b);
        swap(a, b);
        $display("After swap:  a=%0d, b=%0d", a, b);

        // Early return
        $display("\nMax value: %0d", find_max(arr));

        // Recursive function
        $display("Factorial(5) = %0d", factorial(5));
    end

endmodule
```

PART 5: FUNCTION ARGUMENT TYPES

```
module function_arguments;

    // ======================================
    // INPUT Arguments (default)
    // ======================================

    function int add_input(input int a, input int b);
        return a + b;
    endfunction

    // "input" is default, so this is same as above:
    function int add_default(int a, int b);
        return a + b;
    endfunction

    // ======================================
    // OUTPUT Arguments (SV Functions only - Verilog tasks only)
    // ======================================

    function void calculate(int a, int b,
                            output int sum,
                            output int product);
        sum = a + b;
        product = a * b;
    endfunction

    // ======================================
    // INOUT Arguments (bidirectional)
    // ======================================

    function void increment(inout int value);
        value = value + 1;
    endfunction

    // ======================================
    // REF Arguments (Pass by reference - SV only)
    // ======================================

    function void double_it(ref int value);
        value = value * 2;  // Modifies original variable
    endfunction

    // Difference: ref vs inout
    // ref   = Direct reference, no copying (efficient, SV only)
    // inout = Copied in and out (less efficient, both Verilog & SV)

    initial begin
        int x = 10, y = 5, sum, prod;

        $display("\n========== FUNCTION ARGUMENTS ==========\n");

        // Input arguments
        $display("add_input(3, 4) = %0d", add_input(3, 4));

        // Output arguments
        calculate(10, 5, sum, prod);
        $display("calculate(10,5): sum=%0d, prod=%0d", sum, prod);
```

```
        // Inout arguments
        $display("Before increment: x=%0d", x);
        increment(x);
        $display("After increment:  x=%0d", x);

        // Ref arguments
        $display("Before double: y=%0d", y);
        double_it(y);
        $display("After double:  y=%0d", y);
    end

endmodule
```

PART 6: TASK ARGUMENT TYPES

```
module task_arguments;

    // ======================================
    // INPUT Arguments
    // ======================================

    task task_input(input int a, input int b);
        $display("Task received: a=%0d, b=%0d", a, b);
    endtask

    // ======================================
    // OUTPUT Arguments (common in tasks)
    // ======================================

    task task_output(int a, int b, output int result);
        #10;  // Tasks can have delays
        result = a + b;
    endtask

    // ======================================
    // INOUT Arguments
    // ======================================

    task task_inout(inout int value);
        #5;
        value = value * 10;
    endtask

    // ======================================
    // REF Arguments
    // ======================================

    task task_ref(ref int value);
        #5;
        value = value + 100;
    endtask

    initial begin
        int x = 5, result;

        $display("\n========== TASK ARGUMENTS ==========\n");

        task_input(10, 20);

        task_output(5, 3, result);
```

```
        $display("Task output result: %0d", result);

        $display("Before task_inout: x=%0d", x);
        task_inout(x);
        $display("After task_inout:  x=%0d", x);

        task_ref(x);
        $display("After task_ref:    x=%0d", x);
    end

endmodule
```

PART 7: AUTOMATIC vs STATIC

```
module automatic_vs_static;

    // ========================================
    // STATIC (Default in Verilog, optional in SV)
    // ========================================

    function int static_counter();
        int count = 0;  // Static storage - persists between calls means the
value will be incremented for every function calls
        count++;
        return count;
    endfunction

    // ========================================
    // AUTOMATIC (Reentrant/Recursive - SV only)
    // ========================================

    function automatic int auto_counter();
        int count = 0;  // Automatic storage - new each call
        count++;
        return count;
    endfunction

    // Recursive example (requires automatic)
    function automatic int fibonacci(int n);
        if (n <= 1) return n;
        return fibonacci(n-1) + fibonacci(n-2);
    endfunction

    initial begin
        $display("\n========== AUTOMATIC vs STATIC ==========\n");

        // Static storage
        $display("Static calls:");
        $display("  Call 1: %0d", static_counter());
        $display("  Call 2: %0d", static_counter());
        $display("  Call 3: %0d", static_counter());
        $display("  → Count persists between calls");

        // Automatic storage
        $display("\nAutomatic calls:");
        $display("  Call 1: %0d", auto_counter());
        $display("  Call 2: %0d", auto_counter());
        $display("  Call 3: %0d", auto_counter());
        $display("  → Count resets each call");
```

```
        // Recursion
        $display("\nRecursive fibonacci(7) = %0d", fibonacci(7));
    end

endmodule
```

PART 8: CLASS METHODS (Functions & Tasks)

```
class Calculator;
    int last_result;

    // ========================================
    // Regular Function
    // ========================================

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

    // ========================================
    // Virtual Function (for polymorphism)
    // ========================================

    virtual function int multiply(int a, int b);
        last_result = a * b;
        return last_result;
    endfunction

    // ========================================
    // Static Function (belongs to class, not instance)
    // ========================================

    static function int get_version();
        return 1;  // Can't access instance variables
    endfunction

    // ========================================
    // Task with Delay
    // ========================================

    task delayed_compute(int a, int b, output int result);
        #10;
        result = a + b;
        last_result = result;
    endtask

    // ========================================
    // Extern Declaration (body defined outside class)
    // ========================================

    extern function int subtract(int a, int b);
endclass

// Extern function body
function int Calculator::subtract(int a, int b);
    last_result = a - b;
    return last_result;
endfunction
```

```
module class_methods_demo;
    initial begin
        Calculator calc;
        int result;

        $display("\n========== CLASS METHODS ==========\n");

        calc = new();

        // Regular function
        $display("add(5, 3) = %0d", calc.add(5, 3));

        // Virtual function
        $display("multiply(4, 5) = %0d", calc.multiply(4, 5));

        // Static function (no object needed)
        $display("Version: %0d", Calculator::get_version());

        // Task with delay
        calc.delayed_compute(10, 20, result);
        $display("Delayed result: %0d", result);

        // Extern function
        $display("subtract(10, 3) = %0d", calc.subtract(10, 3));
    end
endmodule
```

PART 9: ADVANCED FEATURES

```
module advanced_features;

    // ========================================
    // Variable Number of Arguments (SV only)
    // ========================================

    function int sum_all(int values[]);
        int total = 0;
        foreach(values[i]) total += values[i];
        return total;
    endfunction

    // ========================================
    // Return Arrays (SV only)
    // ========================================

    function int[] create_sequence(int n);
        int seq[] = new[n];
        for(int i=0; i<n; i++) seq[i] = i;
        return seq;
    endfunction

    // ========================================
    // Function with Constraints
    // ========================================

    function int constrained_random(int min, int max);
        return $urandom_range(max, min);
    endfunction

    // ========================================
```

```
    // Task with Fork-Join
    // ======================================

    task parallel_operations();
        fork
            begin
                #10;
                $display("[%0t] Task 1 done", $time);
            end
            begin
                #20;
                $display("[%0t] Task 2 done", $time);
            end
        join
    endtask

    initial begin
        int arr[] = '{1, 2, 3, 4, 5};
        int seq[];

        $display("\n========== ADVANCED FEATURES ==========\n");

        // Array argument
        $display("Sum of array: %0d", sum_all(arr));

        // Return array
        seq = create_sequence(5);
        $display("Sequence: %p", seq);

        // Random with constraints
        $display("Random[1-10]: %0d", constrained_random(1, 10));

        // Parallel task
        parallel_operations();
    end

endmodule
```

COMPREHENSIVE COMPARISON TABLES

FUNCTION vs TASK COMPARISON

Feature	Function	Task
Return Value	✅ MUST return value (or void in SV)	❌ No direct return (uses output args)
Timing Controls (#, @, wait)	❌ Cannot have (executes in 0 time)	✅ Can have #, @, wait
Simulation Time	✅ Zero time	⚠ Can consume time
Multiple Outputs	❌ One return only (SV: can use output)	✅ Multiple via output
Used in Expression	✅ Yes (`x = func()`)	❌ No
Call Other Tasks	❌ Cannot call tasks	✅ Can call tasks/funcs
Fork-Join	❌ Not allowed	✅ Allowed
Typical Use	Calculations, Computations	Sequences/Procedures, Time-consuming ops

Feature	Function	Task
Arguments	input, output (SV), inout, ref (SV)	input, output, inout, ref (SV)

VERILOG vs SYSTEMVERILOG

Feature	Verilog	SystemVerilog
Void Functions	❌ Not supported	✅ Supported
Default Arguments	❌ Not supported	✅ Supported
Return Statement	⚠ Limited	✅ Anywhere
Pass by Reference	❌ Not in functions	✅ ref keyword
Automatic Functions	❌ Static only	✅ automatic keyword
Function Outputs	❌ Not supported	✅ Supported
Class Methods	❌ No classes	✅ Full OOP support
Virtual Functions	❌ Not supported	✅ For polymorphism
Static Methods	❌ No classes	✅ Supported

WHEN TO USE WHAT

USE FUNCTION WHEN:

- ✅ Need to return a value
- ✅ Pure computation (no delays)
- ✅ Want to use in expressions
- ✅ Need to call from constraints
- ✅ Implementing mathematical operations
- ✅ Building testbench utilities

USE TASK WHEN:

- ✅ Need timing controls (delays)
- ✅ Need multiple outputs
- ✅ Implementing protocols/sequences
- ✅ Driver/monitor operations
- ✅ Test sequences
- ✅ Complex verification procedures

ARGUMENT TYPE COMPARISON

Argument Type	Function Support	Task Support	Behavior
input	✅ Yes (default)	✅ Yes (default)	Pass by value, read-only
output	✅ Yes (SV only)	✅ Yes	Pass by value out
inout	✅ Yes	✅ Yes	Pass by value in and out
ref	✅ Yes (SV only)	✅ Yes (SV only)	Pass by reference (efficient)

ref vs inout

- **ref** = Direct reference, no copying (efficient, SV only)
- **inout** = Copied in and out (less efficient, both Verilog & SV)

KEY TAKEAWAYS

1. **Functions** are for calculations that return a value with zero simulation time
2. **Tasks** are for procedures that may take time and can have multiple outputs
3. **SystemVerilog** adds many enhancements over Verilog (void functions, default args, ref, automatic, etc.)
4. **Use functions** when you need results in expressions
5. **Use tasks** when you need timing or complex sequences
6. In **Verilog**, return by assigning to function name; in **SV**, use `return` keyword
7. **Always declare** function arguments explicitly - they don't inherit from module ports