Verilog HDL:
Functions & Tasks

Pravin Zode

Outline

- Functions
- Tasks
- Difference between Function and Task
- Synthesizability
-

Verilog Functions

- Functions are designed to process inputs and return a single value
- Cannot contain delays (#), event controls (@), or non-blocking assignments (<=)
- Execute in zero simulation time
- Useful for simple calculations
- Functions: Arithmetic operations like addition, multiplication

Verilog Function Syntax

```
1   // Style 1
2   function <return_type> <function_name> (<port_list>);
3     ...
4     return <value or expression>
5   endfunction
6
7   // Style 2
8   function <return_type> <function_name> ();
9     input <port_list>;
10    inout <port_list>;
11    output <port_list>;
12    ...
13    return <value or expression>
14  endfunction
```

Function Declaration

There are two ways to declare inputs to a function:

```
function [7:0] sum;
    input [7:0] a, b;
    begin
        sum = a + b;
    end
endfunction

function [7:0] sum (input [7:0] a, b);
    begin
        sum = a + b;
    end
endfunction
```

Function Call & Return

Function Return

```
1 | sum = a + b;
```

Function Call

```
1   reg [7:0] result;
2   reg [7:0] a, b;
3
4   initial begin
5       a = 4;
6       b = 5;
7       #10 result = sum (a, b);
8   end
```

Verilog Function

```
1   // Gray Code Converter - Using Function
2   module gray_function (
3       input  [3:0] bin,
4       output [3:0] gray
5   );
6       function [3:0] bin2gray;
7           input [3:0] b;
8           begin
9               bin2gray[3] = b[3];
10              bin2gray[2] = b[3] ^ b[2];
11              bin2gray[1] = b[2] ^ b[1];
12              bin2gray[0] = b[1] ^ b[0];
13          end
14      endfunction
15
16      assign gray = bin2gray(bin);
17  endmodule
```

Verilog Function

```
1   module function_example;
2
3     function compare(input int a, b);
4       if(a>b)
5         $display("a is greater than b");
6       else if(a<b)
7         $display("a is less than b");
8       else
9         $display("a is equal to b");
10      return 1; // Not mandatory to write
11    endfunction
12
13    initial begin
14      compare(10,10);
15      compare(5, 9);
16      compare(9, 5);
17    end
18  endmodule
```

Output

```
a is equal to b
a is less than b
a is greater than b
```

When Functions Are Synthesizable ?

- Functions are synthesizable when they use
  - bitwise, arithmetic, and logical operators (+ - ^ & | ~ << >>)
  - Conditional constructs (if-else, case, ?:)
  - Loops with fixed iteration bounds (e.g., for i=0 to 7)
  - Pure combinational logic (no timing controls, delays, or events)
  - Used in always @(*) or continuous assignments

When Functions Are NOT Synthesizable ?

- Functions are not synthesizable when they use
  - Unbounded / variable loops (iteration depends on runtime input)
  - Recursion (function calling itself)
  - Timing controls or delays (#, @, wait) inside functions
  - Dynamic memory / file I/O inside functions
  - Very large logic (e.g., factorial for general n) → hardware explosion

When to Use Functions ?

- Implementing reusable combinational logic (e.g., parity, Gray code, encoders)
- Improving readability by abstracting complex expressions
- Avoiding code duplication (same logic used in multiple places)
- Parameterizable small computations (bit manipulation, arithmetic helpers)
- Elaboration-time constant generation (computed once at compile time)

When to avoid Functions ?

- Tasks requiring delays, events, or timing control → use tasks instead
- Sequential behavior with multiple clock cycles
- Large iterative computations (e.g., factorial for general n)
- Operations that depend on runtime iteration counts

When Tasks Are Non-Synthesizable ?

- **Use of Delays (#)**Any # delay in a task prevents synthesis. Example: #10 output = input + 1;
- **Event Control** (@)Using @(posedge clk) or @(input_signal) inside a task → simulation only.
- **Wait Statements** : Tasks with wait(condition) cannot be synthesized.
- **File Operations :** Tasks that read/write files ($readmemb, $fwrite) are for testbenches only.
- **Infinite or Variable Loops** : Loops that depend on runtime inputs or are unbounded cannot synthesize.

Tasks are synthesizable only if they are purely combinational (no delays, events, or wait).

Verilog Tasks

Tasks

- A task is a reusable procedure or subroutine in Verilog
- Can have input, output, and inout ports
- Can contain loops, conditionals, and procedural statements
- Can include timing controls (#, @) for simulation
- Useful for multi-step operations or multiple outputs
- Synthesizable only if no delays or events are used.

Verilog Task Syntax

```
1   // Style 1
2   task [name];
3       input  [port_list];
4       inout  [port_list];
5       output [port_list];
6       begin
7           [statements]
8       end
9   endtask
10
11  // Style 2
12  task [name] (input [port_list], inout [port_list], output [port_list]);
13      begin
14          [statements]
15      end
16  endtask
17
18  // Empty port list
19  task [name] ();
20      begin
21          [statements]
22      end
23  endtask
```

Example : Verilog Task

```
1   module square_task(
2       input  [3:0] num,
3       output reg [7:0] result );
4
5       // Task definition
6       task compute_square;
7           input [3:0] x;
8           output [7:0] y;
9           begin
10              y = x * x;
11          end
12      endtask
13
14      // Call task in combinational block
15      always @(*) begin
16          compute_square(num, result);
17      end
18
19  endmodule
```

Output

```
time=0 num= 3 square=  9
time=10 num= 5 square= 25
time=20 num= 8 square= 64
```

Example : Verilog Task Reuse

```
1   module square_task_reuse(
2       input  [3:0] num1, num2, num3,
3       output reg [7:0] res1, res2, res3
4   );
5
6       // Task definition
7       task compute_square;
8           input [3:0] x;
9           output [7:0] y;
10          begin
11              y = x * x;
12          end
13      endtask
14
15      // Call task 3 times
16      always @(*) begin
17          compute_square(num1, res1);
18          compute_square(num2, res2);
19          compute_square(num3, res3);
20      end
21
22  endmodule
```

When to use Taks ?

- **Multiple Outputs Needed:**
  - Tasks allow input, output, and inout arguments.
  - Useful when a single computation produces more than one result.
- **Sequential or Procedural Operations** : Step by-step algorithms, stimulus generation, applying test vectors
- **Testbench Development** : include delays (#), event controls (@), and wait statements , verification, file I/O, and simulation control
- **Reusable Code Blocks :** repeated procedures to avoid code duplication.

When to avoid Taks ?

- **Single Output** Only If only one result is required, prefer a function (cleaner & concise).
- **Pure Combinational Expressions** For simple logic (e.g., parity, Gray code, max finder), functions or assign are better.
- **Inside Synthesizable RTL** with TimingTasks with #, @, or wait → not synthesizable
- **Overcomplication of Code** Using tasks for trivial logic makes the design harder to read and maintain.
- **Unnecessary Code Duplication** If the same logic fits neatly in a function, tasks add extra verbosity without benefit.

Difference Functions and Task

Function	Task
Function can enable another function but not another task	A task can enable other tasks and functions
Function always execute in 0 simulation time	Task may execute in non-zero simulation time
Functions must not contain any delay, event or timing control statements	Tasks may contain delay event or timing control statements
Functions must have atleast one input argument	Tasks may have zero or more arguments
Functions always return single value	Task do not return with a value
Functions cannot have output or inout arguments	Task can pass multiple values through output and inout arguments

Functions vs. Tasks — When to Use

Use Functions When:

- Logic is purely combinational
- Only one return value is needed
- Operation completes in a single simulation cycle
- Used for small reusable computations (e.g., parity, Gray conversion)

Use Tasks When:

- Need multiple outputs
- Timing control is required (@, #, wait)
- Sequential operations across simulation time
- More complex behavior (e.g., stimulus generation in testbenches, file I/O)

Summary

Tasks

- Can have input, output, and inout arguments
- Can include delays, event controls, and timing constructs
- Used for complex operations or multi-step procedures
- Invoked like a procedure call: my_task(a, b);

Functions

- Only have input arguments.
- Cannot contain delays or event controls.
- Must execute in one simulation time unit.
- Return a single value: result = my_function(x);

Functions → Best for RTL design (synthesizable combinational logic)
Tasks → Best for testbenches and simulation (non-synthesizable behavior)

Thank you !

Happy Learning