VERILOG FUNCTIONS AND TASKS

- A function or task is a group of statements that performs some specific action. Both of them can be called at various points to perform a certain operation.
- Thus to execute the same code several times, there are functions in a programming language (also known as a subroutine). In verilog, there are two types of subroutines, functions and tasks.

What are subroutines?

- A routine defines the execution flow of a code. A subroutine is some code that is executed only when called by the main programme. Once the execution of the subroutine is complete, the pointer moves back to the main programme.

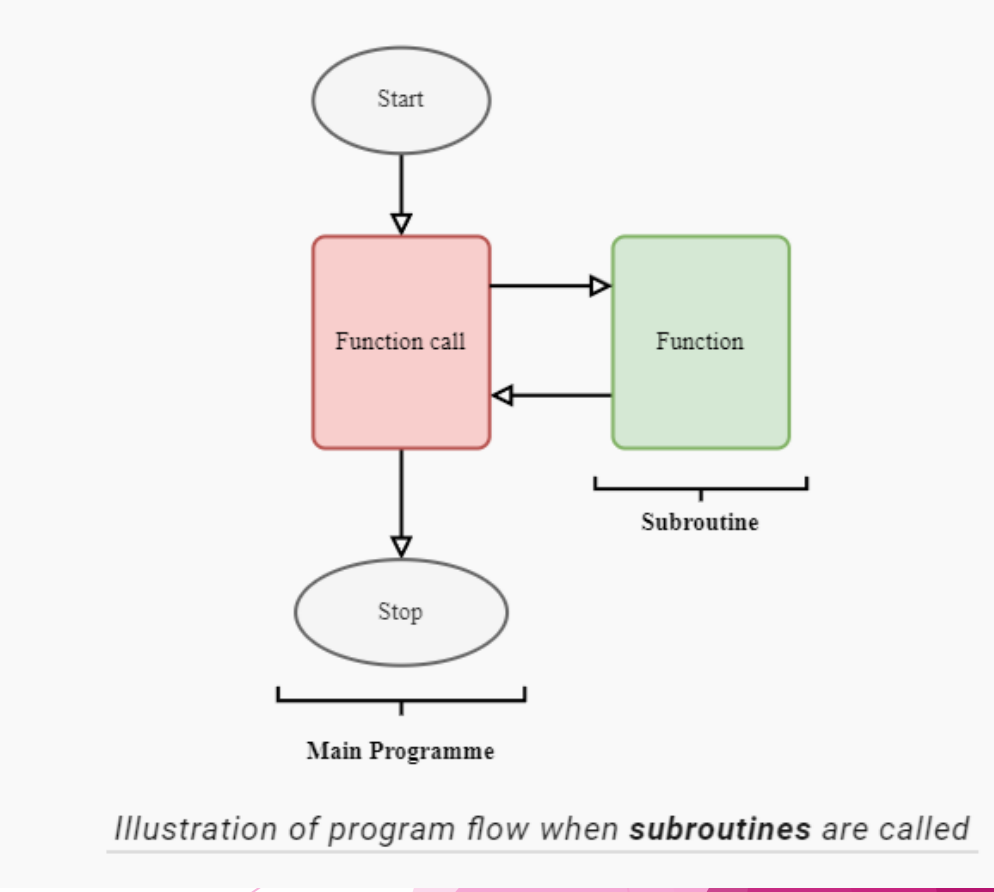

*Illustration of program flow when **subroutines** are called*

HOW TO WRITE A FUNCTION

Syntax

```
function [automatic] [return_type] name ([arguments]); // Function Declaration
    begin
    // function definition goes here
    end
endfunction
```

- To write a function, first, we need to declare it. This step is known as a functional declaration.
- After declaration, we write the function's body, which is nothing but the function's functionality and is known as a function definition.
- we cannot define a function before declaring it.

PURPOSE OF FUNCTIONS

- The purpose of a function is to return a value that is to be used in an expression.
- A function definition always starts with the ***function*** keyword followed by the return type, name, and a port list enclosed in parentheses. And it ends with the ***endfunction*** keyword.
- Functions should have at least one input declaration and a statement that assigns a value to the register with the same name as the function.
- The return type can be ***void*** if the function does not return anything.
- Functions can only be declared inside a module declaration and can be called through always blocks, continuous assignments, or other functions.
- Functions describe combinational logic. Functions are an excellent way to reuse procedural code.
- Functions cannot contain any time-controlled statements, and they cannot enable tasks. Functions can return only one value.

FUNCTION DECLARATION

- Function declaration consists of 3 parts:
  - *Function name*
  - *Automatic*
  - *Return type*
  - *Arguments*

- A function is declared using a function keyword, and the function definition ends with the endfunction keyword. The function definition goes inside begin end block

WAYS TO DEFINE FUNCTIONS:

```
/ Style 1
unction <return_type> <function_name> (input <port_list>,inout <port_list>, output <port_list>);
...
return <value or expression>
ndfunction

/ Style 2
unction <return_type> <function_name> ();
 input <port_list>;
 inout <port_list>;
 output <port_list>;
 ...
 return <value or expression>
ndfunction
```

```
1  function [7:0] sum;
2      input [7:0] a, b;
3      begin
4          sum = a + b;
5      end
6  endfunction
7
8  function [7:0] sum (input [7:0] a, b);
9      begin
10         sum = a + b;
11     end
12 endfunction
```

FUNCTION EXAMPLE

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

→ Function Return Value

Function Return Value

The function definition will implicitly create an internal variable of the same name as that of the function.

Hence it is illegal to declare another variable of the same name inside the scope of the function.

The return value is initialized by assigning the function result to the internal variable.

CALLING A FUNCTION

- A function call is an operand with an expression. A function call must specify in its terminal list all the input parameters.
- **Example**

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

VERILOG FUNCTIONS RULES

Rules for Using Functions in Verilog:
➢ Verilog functions can have one or more input arguments
➢ Functions can only return one value
➢ Functions can not use time consuming constructs such as posedge, wait or delays (#)
➢ We can't call tasks from within a function
➢ We can call other functions from within a function
➢ Non-blocking assignment can't be used within a function
➢ Local variables can be declared and used inside of the function
➢ We can access and modify global variables from inside a verilog function
➢ If we don't specify a return type, the function will return a single bit

```
1  // Using inline declaration of the inputs
2  function integer addition (input integer in_a, in_b);
3    // Return the sum of the two inputs
4    addition = in_a + in_b;
5  endfunction
6
7  // Declaring the inputs in the function body
8  function integer addition;
9    input integer in_a;
10   input integer in_b;
11   begin
12     // Return the sum of the two inputs
13     addition = in_a + in_b;
14   end
15 endfunction
```

In the example below, in_a would map to the a argument and in_b would map to b.

```
1  // Calling a verilog function
2  func_out = addition(a, b);
```

AUTOMATIC FUNCTIONS IN VERILOG

- We can also use the verilog automatic keyword to declare a function as reentrant.
- When we declare a function as reentrant, the variables and arguments within the function are dynamically allocated. In contrast, normal functions use static allocation for internal variables and arguments.
- Functions which use the automatic keyword allocate memory whenever the function is called. The memory is then deallocated once the function has finished with it.
- As a result of this, our simulation software can execute multiple instances of an automatic function.
- We can use the automatic keyword to write recursive functions in verilog. This means we can create functions which call themselves to perform a calculation.

```
function automatic integer factorial (input integer a);
  begin
    if (a > 1) begin
      factorial = a * factorial(a - 1);
    end
    else begin
      factorial = 1;
    end
  end
endfunction
```

```
module m;

 function void this_is_static;
   int count = 1;
   $display("this_is_static - count= %d", count++);
 endfunction

 function static void this_is_also_static;
   int count = 1;
   $display("this_is_also_static - count= %d", count++);
 endfunction

 function automatic void this_isnt_static;
   int count = 1;
   $display("this_isnt_static - count= %d", count++);
 endfunction

 function automatic void this_isnt_static_either;
    static int count = 1;
    $display("this_isnt_static_either, but count is - count=
%d", count++);
  endfunction

  initial
    begin
      repeat(3)
        this_is_static;
      repeat(3)
        this_is_also_static;
      repeat(3)
        this_isnt_static;
      repeat(3)
        this_isnt_static_either;
    end

endmodule
```

Verilog Task

- We use verilog tasks to write small sections of code that we can reuse throughout our design.
- Unlike functions, we can use time consuming constructs such as wait, posedge or delays (#) within a task. As a result of this, we can use both blocking and non-blocking assignment in verilog tasks.
- Verilog tasks can also have any number of inputs and can also generate any number of outputs. This is in contrast to functions which can only return a single value.
- These features mean tasks are best used to implement simple pieces of code which are repeated several times in our design.
- We can also create global tasks which are shared by all modules in a given file. To do this we simply write the code for the task outside of the module declarations in the file.

Syntax:

```
// Style 1
task <task_name> (input <port_list>, inout
<port_list>, output <port_list>);
 ...
endtask

// Style 2
task <task_name> ();
  input <port_list>;
  inout <port_list>;
  output <port_list>;
  ...
endtask
```

```
module task_example;

  task compare(input int a, b, output done);
    if(a>b)
      $display("a is greater than b");
    else if(a<b)
      $display("a is less than b");
    else
      $display("a is equal to b");

    #10;
    done = 1;
  endtask

  initial begin
    bit done;
    compare(10,10, done);
    if(done) $display("comparison completed at time = %0t", $time);
    compare(5,9, done);
    if(done) $display("comparison completed at time = %0t", $time);
    compare(9,5, done);
    if(done) $display("comparison completed at time = %0t", $time);
  end
endmodule
```

Output:

```
a is equal to b
comparison completed at time = 10
a is less than b
comparison completed at time = 20
a is greater than b
comparison completed at time = 30
```

```
module task_demo;
    reg [3:0] op;
    // Task without any arguments.
    task print();
        begin
            $display("At [%0t] printing from task", $time);
            $display("At [%0t], output from task = %b", $time, op);
        end
    endtask

    // Task with 2 ip arguments and 1 output argument. As task calculates
    // sum thus it needs to return a value. Function cannot
    // be used as we want to give some delay also.
    task sum(input reg[3:0] a, b, output reg [3:0] sum);
        begin
            $display("At [%0t], Calculating sum...", $time);
            #10;
            sum = a + b;
        end
    endtask

    initial begin
        // As there is no return type, thus tasks are not
        // assigned to any variable. Rather, the variable in which
        // output needs to be assigned is passed as an argument.
        sum(4'b1010, 4'b0101, op);
        print();
    end
endmodule
```

Output

```
# At [0], Calculating sum...
# At [10] printing from task
# At [10], output from task = 1111
```

AUTOMATIC TASKS IN VERILOG

- We can also use the automatic keyword with verilog tasks in order to make them reentrant. using the automatic keyword means that our simulation tool uses dynamic memory allocation.
- tasks use static memory allocation by default
- In contrast, tasks which use the automatic keyword allocate memory whenever the task is called. The memory is then freed once the task has finished with it.

The code below shows how we write a static task to implement this example.

```
1  // Task which performs the increment
2  task increment(input integer incr);
3    integer i = 1;
4    i = i + incr;
5    $display("Result of increment = %0d", i);
6  endtask
7
8  // Run the task three times
9  initial begin
10   increment(1);
11   increment(2);
12   increment(3);
13 end
```

What is its output???

```
// Task which performs the increment
task increment(input integer incr);
  integer i = 1;
  i = i + incr;
  $display("Result of increment = %0d", i);
endtask

// Run the task three times
initial begin
  increment(1);
  increment(2);
  increment(3);
end
```

Running this code in the icarus verilog simulation tool results in the following output:

```
Result of increment = 2
Result of increment = 4
Result of increment = 7
```

The code snippet below shows the same task except that this time we use the automatic keyword.

```
1  // Automatic task which performs the increment
2  task automatic increment(input integer incr);
3    integer i = 1;
4    i = i + incr;
5    $display("Result of increment = %0d", i);
6  endtask
7
8  // Run the task three times
9  initial begin
10   increment(1);
11   increment(2);
12   increment(3);
13 end
```

What is its output???

```
1  // Automatic task which performs the increment
2  task automatic increment(input integer incr);
3    integer i = 1;
4    i = i + incr;
5    $display("Result of increment = %0d", i);
6  endtask
7
8  // Run the task three times
9  initial begin
10   increment(1);
11   increment(2);
12   increment(3);
13 end
```

Running this code in the icarus verilog simulation tool results in the following output:

```
1  Result of increment = 2
2  Result of increment = 3
3  Result of increment = 4
```

Task	Function
Can have zero or more than one argument.	It needs to have at least one argument.
A task can have delay statements inside it.	A function cannot have a delay statement. It should return a value at the same time step.
A Task Does not have a return type. However, output arguments help return value.	A function does have a return type.
A task can return more than one values as there can be any number of output arguments.	A function can return only one value at a time
A task can call another function or a task from its body.	A function can only call another function from its body. A task cannot be called as it can consume time, and function is not allowed to consume time.