

# Functions & Tasks

## Functions

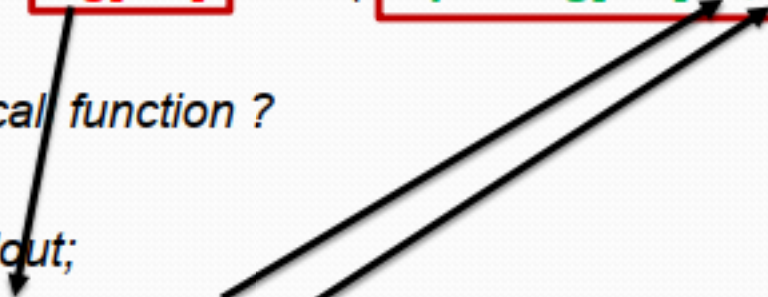
**function** **return\_type** **func\_name** (**arguments**);

**function** **reg[7:0]** **add** ( **input reg[7:0] a,b** );

How to call function ?

reg [7:0]out;

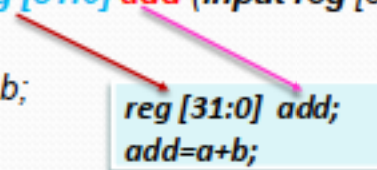
initial out = **add** (in1,in2);



## Functions

- Return value of a function is set by assigning a value to the **name of the function**.

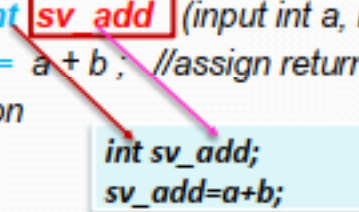
```
function reg [31:0] add (input reg [31:0] a,b);  
begin  
add = a + b;  
end  
endfunction
```



```
reg [31:0] add;  
add=a+b;
```

```
function int sv_add (input int a, b); //begin end not required  
return a + b ; //we can use return statement to return value  
endfunction
```

```
function int sv_add (input int a, b); //begin end not required  
sv_add = a + b ; //assign return value to function name  
endfunction
```



```
int sv_add;  
sv_add=a+b;
```

# Subroutines

- **Functions must execute without blocking simulation time.**
  - A function can have input, output, inout, and ref arguments.
  - A function shall not contain any time-controlled statements.
  - #,
  - ## (cycle delays) ,
  - @(posedge clk),
  - @(sig) ,
  - fork-join, fork-join\_any,
  - wait, wait\_order, or expect.
  - A function shall not call tasks regardless of whether those tasks contain time-controlling statements.
- Tasks can block simulation time during execution.

# Functions

- The primary purpose of a function is to return a value that is to be used in an expression.

```
function reg [3:0] add ( reg [3:0] a,b);  
return (a+b);  
endfunction
```

```
initial sum = add(5,5) + 55;
```

```
function bit max ( reg [3:0] a,b);  
return (a > b);  
endfunction
```

```
initial begin  
if ( max(10,5) )  
$display("Bigger number");  
else  
$display("Smaller number");  
end
```

```
function reg [3:0] add ( reg [3:0] a,b);  
return (a+b);  
endfunction
```

```
function reg [3:0] sub ( reg [3:0] a,b);  
return (a-b);  
endfunction
```

```
wire [3:0] out;  
assign out = sel ? add(5,5) : sub (10,5) ;
```

```
task add ( input reg [3:0] a,b, output reg [3:0] z);  
z = a+b;  
endtask
```

```
reg [3:0] temp,sum;  
initial begin  
add (5,5,temp);  
sum = temp + 55;  
end
```

## Function argument directions

- Default direction is input if no direction has been specified

function **logic** [15:0] **func** ( int x, int y );

Same as :

function **logic** [15:0] **func** (input int x, input int y);

formal arguments

initial out= func (a,b);

actual arguments

- Once a direction is given, subsequent **formals** default to the same direction
- function **logic** [15:0] **func** (int a, int b, output **logic** [15:0] u, v);

input int a,

input int b,

output **logic**[15:0] u,

output **logic**[15:0] v

## void functions

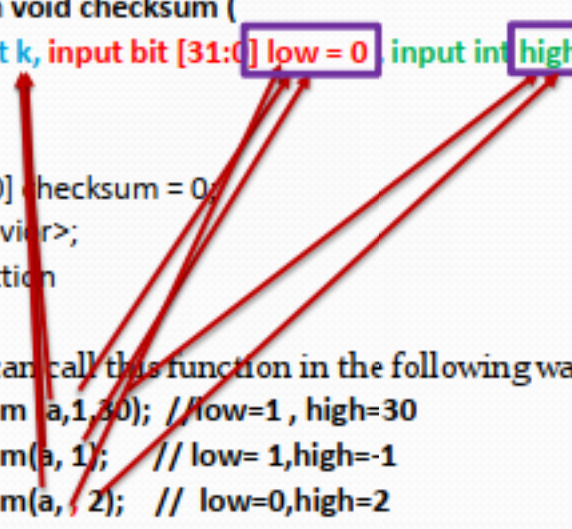
- Functions can be explicitly declared as a **void** type, indicating that there is no return value from the function.

```
function void print_state(...);  
$display("state = %s at time=%0t" , cur_state.name() , $time);  
endfunction  
  
initial print_state();
```

## Default Value for an Argument

- In SystemVerilog you can specify a default value that is used if you leave out an argument in the call.

```
function void checksum (  
input int k, input bit [31:0] low = 0 input int high = -1  
);
```



```
bit [31:0] checksum = 0;  
//<behavior>;  
endfunction
```

- You can call this function in the following ways

```
checksum(a,1,30); // low=1 , high=30  
checksum(a, 1);   // low= 1,high=-1  
checksum(a, 2);   // low=0,high=2  
checksum();        // Compile error: k has no default
```



## Passing Arguments by Name

- If you have a task or function with many arguments, some with default values, and you only want to set a few of those arguments, you can specify a subset by specifying the name of the routine argument with a port-like syntax

```
function void many (input int a=1, b=2, c=3, d=4);  
$display("a=%0d b=%0d c=%0d d=%0d", a, b, c, d);  
endfunction
```

```
initial begin  
  many(6, 7, 8, 9);  
  many();  
  many(.c(5));  
  many(, 6, .d(8));  
end
```

*a=6,b=7,c=8,d=9*

*a=1,b=2,c=3,d=4*

*a=1,b=2,c=5,d=4*

*a=1,b=6,c=3,d=8*

## Function with Enum return type

```
typedef enum {IDLE, WAIT, LOAD, STORE} states_t;
states_t p_state, n_state;

function enum {IDLE, WAIT, LOAD, STORE} states_t get_next (.....)
function reg a get_next (.....)
inp_state=LOAD
case (inp_state)
WAIT : get_next = LOAD;
LOAD : STORE = STORE;
STORE : get_next = WAIT;
default : get_next = inp_state; // default next state
endcase
endfunction
initial begin
p_state=LOAD;
n_state = get_next(p_state);
end
```

## tasks

```
task task_name (arguments);
```

```
task mytask (output int x, input logic y);
```

```
<statements>
```

```
endtask
```

- There is a default direction of **input** if no direction has been specified.
- Once a direction is given, subsequent formals default to the same direction

```
task print (a, b, output logic [15:0] u, v);
```

```
<statements>
```

```
endtask
```

```
input logic a;
```

```
input logic b;
```

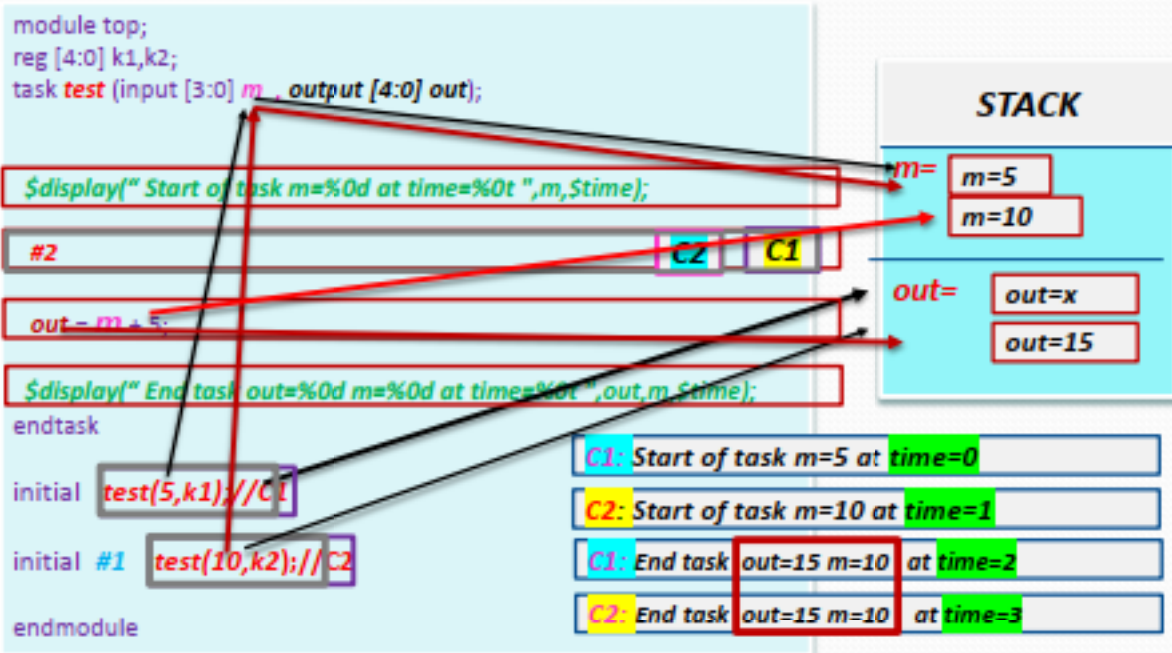
```
output logic [15:0] u;
```

```
output logic [15:0] v;
```

## Task with timing control

```
task wait_for_mem(input [31:0] addr, expect_data,output success);  
  logic [31:0] data;  
    @(posedge clock);  
    data=mem[addr];  
    success = (data == expect_data);  
endtask    //end_of_task  
  
bit result;  
initial begin  
  wait_for_mem(30,50,result);  
  If(result==1) $display("Pass : Data Matched ");  
  else $display("Fail : Data Mis match ");  
end
```

The diagram illustrates the flow of data between the task definition and its call. A red arrow originates from the `wait_for_mem` task definition and points to the `wait_for_mem` call in the `initial` block. Another red arrow originates from the `success` output of the task definition and points to the `result` variable in the `initial` block. A black arrow points from the `result` variable to the `If` statement, indicating the condition for the display message.



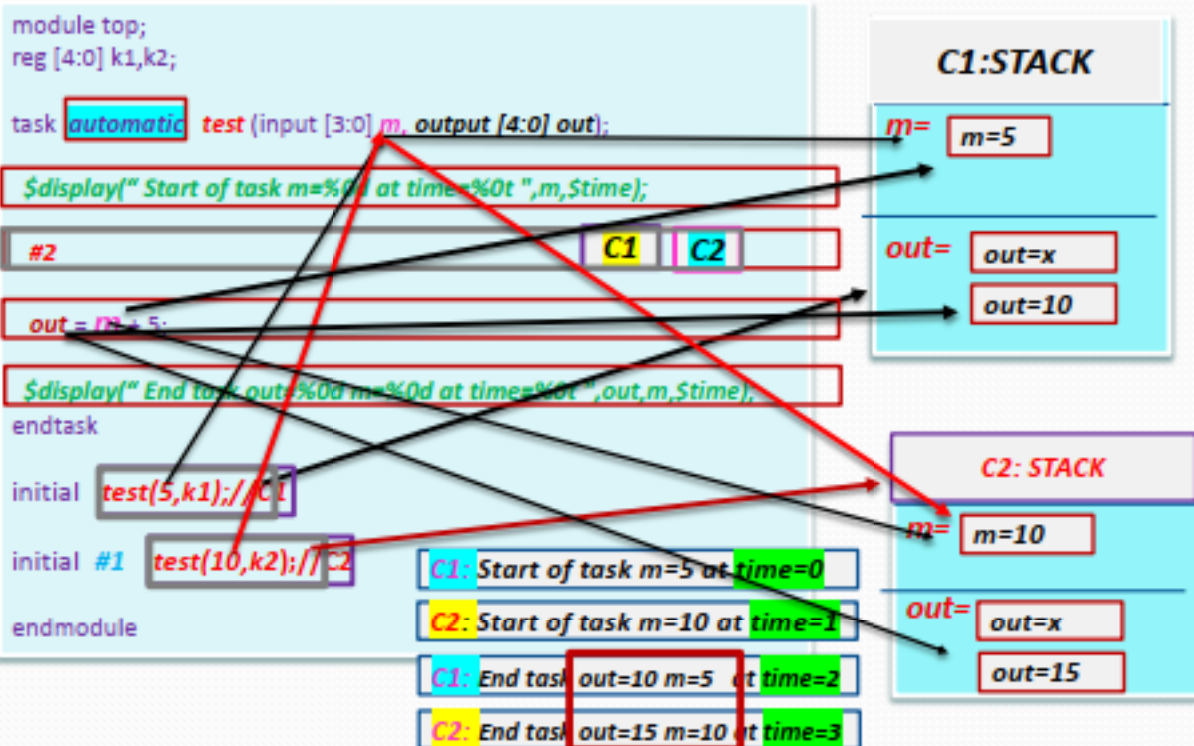
## Static task

```
task test (input [3:0] m , output [4:0] out);  
  $display(" Start of task m=%0d at time=%0t ",m,$time);  
  #2 out=m+1;  
  $display(" End of task   out=%0d at time=%0t ",out,$time);  
endtask
```

- What will happen if we call the task test from multiple concurrent block at the same time ?
- If you called task test a second time while the first one was still waiting, the second call would overwrite the arguments

## Use *automatic* storage

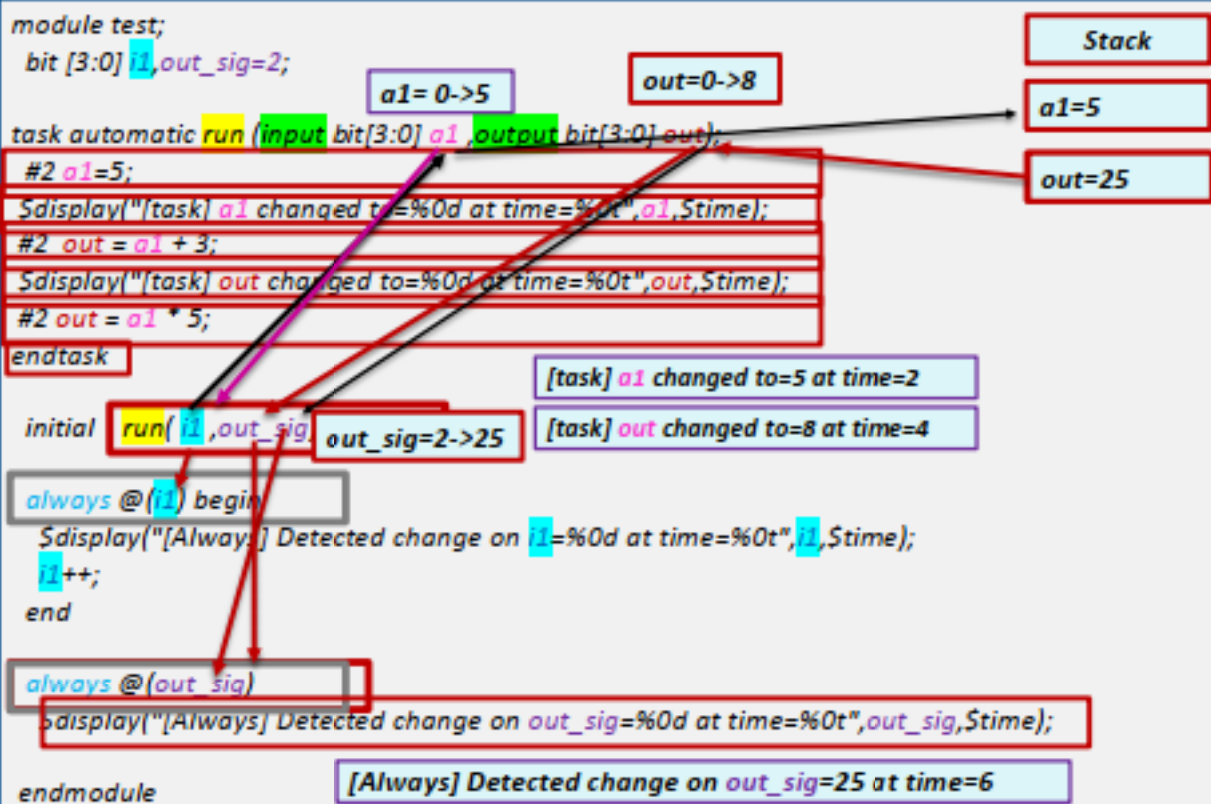
```
task automatic test (input [3:0] m,output [4:0] out);  
  $display(" Start of task m=%0d at time=%0t ",m,$time);  
  #2 out=m+5;  
  $display(" End of task  m=%0d out=%0d at time=%0t ",m,out,$time);  
endtask
```

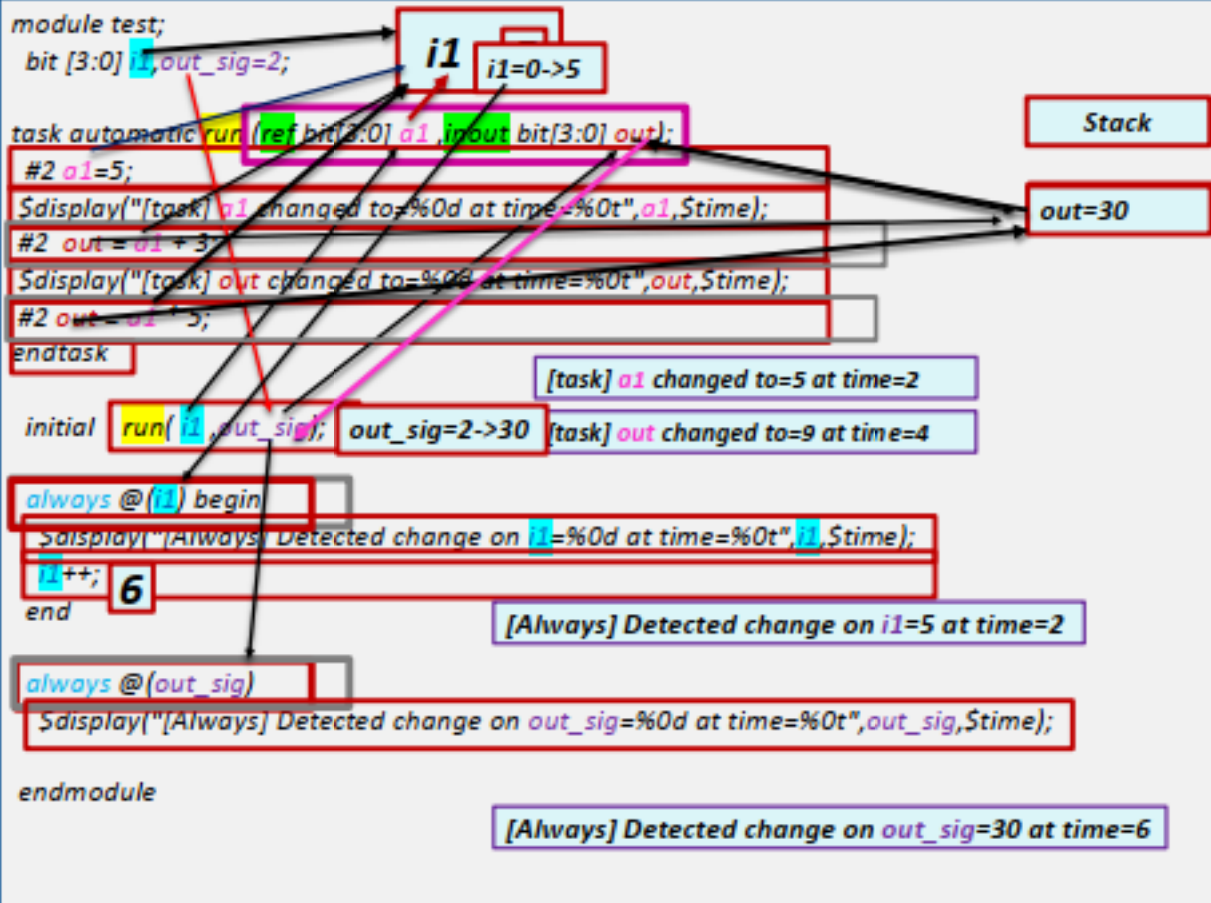




## **Static and automatic functions/tasks**

- *Functions/tasks defined within a module, interface, program, or package default to being static*
- *Functions/tasks can be defined to use automatic storage in the following two ways:*
  - *Explicitly declared using the optional automatic keyword as part of the function/task declaration.*
  - *Implicitly declared by defining the function/task within a module, interface, program, or package that is defined as automatic.*
  - *Functions/tasks defined within a class are always automatic*





### Use **ref direction** when passing arrays to methods

- Arguments passed by reference are not copied into the subroutine area, rather, a reference to the original argument is passed to the subroutine. The subroutine can then access the argument data via the reference.
- Arguments passed by reference shall be matched with **equivalent data types**.
- It shall be illegal to use argument passing by reference for subroutines with a **lifetime of static**.

```
module test;
  bit [7:0] arr1[50];
  bit [7:0] arr2[50];

  task automatic run (ref bit[7:0] ref_arg[50], inout bit[7:0] inout_arg[50]);
    bit [7:0] data;
    //content of task
    ref_arg[10] = 30; //write operation
    data = ref_arg[12]; //read operation
  endtask

  initial run(arr1, arr2);
endmodule
```

The diagram illustrates the concept of passing arrays by reference. Two blue arrows originate from the variable names **arr1** and **arr2** in the initial block. These arrows point to the **ref\_arg** parameter within the **run** task definition. A green rectangular box highlights the **ref** keyword and the **ref\_arg** parameter, indicating that the task receives references to the original arrays rather than copies of their data.

Use **"const ref direction"** when you want **read only operation** in methods

```
module test;
  bit [7:0] arr1[50];
  bit [7:0] arr2[50];

  task automatic run(const ref bit[7:0] ref_arg[50] , inout bit[7:0] inout_arg[50]);

    bit [7:0] data;
    //content of task
    //ref_arg[10] = 30;           //write operation not allowed
    data = ref_arg[12];          //read operation allowed
  endtask

  initial run(arr1, arr2);

endmodule
```

## ref/inout

- Arguments passed by reference are not copied into the subroutine area
- A *ref* argument is similar to an *inout* argument except that an *inout* argument is copied twice:
  - once from the actual into the argument when the subroutine is called and
  - once from the argument into the actual when the subroutine returns.
- To protect arguments passed by reference from being modified by a subroutine, the **const qualifier** can be used together with *ref* to indicate that the argument, although passed by reference, is a **read-only variable**.
- When the argument is passed by reference, both the caller and the subroutine share the same representation of the argument; therefore, any changes made to the argument, within either the caller or the subroutine, shall be visible to each other.



***Thank You***