Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks 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 at least one **input** argument. They can have more than one **input**.	Tasks may have zero or more arguments of type **input, output**, or **inout**.
Functions always return a single value. They cannot have **output** or **inout** arguments.	Tasks do not return with a value, but can pass multiple values through **output** and **inout** arguments.

Input and Output Arguments in Tasks

```
//Define a module called operation that contains the task bitwise_oper
module operation;
…
…
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;

always @(A or B) //whenever A or B changes in value
begin
        //invoke the task bitwise_oper. provide 2 input arguments A, B
        //Expect 3 output arguments AB_AND, AB_OR, AB_XOR
       //The arguments must be specified in the same order as they
       //appear in the task declaration.
        bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);
end
…
…
//define task bitwise_oper
task bitwise_oper;
output [15:0] ab_and, ab_or, ab_xor; //outputs from the task
input [15:0] a, b; //inputs to the task
begin
        #delay ab_and = a & b;
        ab_or = a | b;
        ab_xor = a ^ b;
end
endtask
…
endmodule
```

```
module task_wait;
  reg no_clock;

  task generate_waveform;
    output qclock;
    begin
      qclock = 1;
      #2 qclock = 0;
      #2 qclock = 1;
      #2 qclock = 0;
    end
  endtask

  initial
    generate_waveform (clk_ssp);
endmodule
```

The assignments to `qclock` do not appear on `clk_ssp`, that is, no waveform appears on `clk_ssp`; only the final assignment to `qclock`, which is 0, appears on `clk_ssp` after the task returns. One way to avoid this problem is to make `qclock` as a global variable, that is, declare it outside the task.

```
//Define a module that contains the task asymmetric_sequence
module sequence;
…
reg clock;
…
initial
        init_sequence; //Invoke the task init_sequence
…
always
begin
        asymmetric_sequence; //Invoke the task asymmetric_sequence
end
…
…
//Initialization sequence
task init_sequence;
begin
        clock = 1'b0;
end
endtask

//define task to generate asymmetric sequence
//operate directly on the clock defined in the module.
task asymmetric_sequence;
begin
        #12 clock = 1'b0;
        #5 clock = 1'b1;
        #3 clock = 1'b0;
        #10 clock = 1'b1;
end
endtask
…
…
endmodule
```

declaration

task called

task template

task template

Re-entrant (Automatic) Tasks

```
// Module that contains an automatic (re-entrant) task
// Only a small portion of the module that contains the task definition
// is shown in this example. There are two clocks.
// clk2 runs at twice the frequency of clk and is synchronous
// with clk.
module top;
reg [15:0] cd_xor, ef_xor; //variables in module top
reg [15:0] c, d, e, f; //variables in module top
-
task automatic bitwise_xor;
output [15:0] ab_xor; //output from the task
input [15:0] a, b; //inputs to the task
begin
    #delay ab_and = a & b;
    ab_or = a | b;
    ab_xor = a ^ b;
end
endtask
...
-
// These two always blocks will call the bitwise_xor task
// concurrently at each positive edge of clk. However, since
// the task is re-entrant, these concurrent calls will work correctly.
always @(posedge clk)
    bitwise_xor(ef_xor, e, f);
-
always @(posedge clk2) // twice the frequency as the previous block
    bitwise_xor(cd_xor, c, d);
-
-
endmodule
```

tion, a function must have at least one input. No output or inout declarations are allowed in a function. A function may call other functions.

10.2.1 Function Definition

A function definition can appear anywhere in a module declaration. It is of the form:

```
function [automatic] [signed]
    [range_or_type] function_id;
  input_declaration
  other_declarations
  procedural_statement
endfunction
```

An input to the function is declared using the input declaration. If no range or type is specified with the function definition, then a 1-bit return value is assumed. The return type can be one of **real**, **integer**, **time** or **realtime**. The return value of a function can be declared to be a signed value by using the keyword **signed**. Here is an example of a function.

```
module function_example;
  parameter MAXBITS = 8;

  function [MAXBITS-1:0] reverse_bits;
    input [MAXBITS-1:0] data_in;
    integer k;
    begin
      for (k = 0; k < MAXBITS; k = k + 1)
        reverse_bits[MAXBITS - k - 1] = data_in[k];
    end
  endfunction
  . . .
endmodule
```

The name of the function is `reverse_bits`. The function returns a vector of size `MAXBITS`. The function has one input, `data_in`. `k` is a local integer.

Left/Right Shifter

```
//Define a module that contains the function shift
module shifter;
...
//Left/right shifter
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1
reg [31:0] addr, left_addr, right_addr;
reg control;

//Compute the right- and left-shifted values whenever
//a new address value appears
always @(addr)
begin
    //call the function defined below to do left and right shift.
     left_addr = shift(addr, `LEFT_SHIFT);
     right_addr = shift(addr, `RIGHT_SHIFT);
end
...
...
//define shift function. The output is a 32-bit value.
function [31:0] shift;
input [31:0] address;
input control;
begin
     //set the output value appropriately based on a control signal.
     shift = (control == `LEFT_SHIFT) ?(address << 1) : (address >> 1);

end
endfunction
...
...
endmodule
```