Tasks and Functions

► A designer is frequently required to implement the same functionality at many places in a behavioral design.

► Verilog provides tasks and functions to break up large behavioral designs into smaller pieces.
Tasks

► Must be used if any one of the following conditions is true:

1. There are delay, timing, or event control constructs in the procedure
2. Zero or more than one output arguments
3. No input arguments
Functions

Must be used if all conditions are true:

1. No delay, timing, or event control constructs
2. Single return value
3. At least one input
4. No output arguments
## Difference

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>Task</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A function can enable another function but not another task</td>
<td>1. A task can enable other tasks and functions</td>
</tr>
<tr>
<td>2. Functions always execute in 0 simulation time</td>
<td>2. Tasks may execute in non-zero simulation time.</td>
</tr>
<tr>
<td>3. Functions must not contain any delay, event or timing control statements.</td>
<td>3. Tasks may contain delay, event, or timing control statements.</td>
</tr>
</tbody>
</table>
## Difference (continue…)  

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>Task</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Functions must have at least one input argument. They can have more than one input.</td>
<td>4. Tasks may have zero or more arguments of type input, output or inout.</td>
</tr>
<tr>
<td>5. Functions always return a single value. They cannot have output or inout arguments.</td>
<td>5. Tasks do not return with a value but can pass multiple values through output and inout arguments.</td>
</tr>
</tbody>
</table>
Task declaration

//task declaration syntax
<task>
  ::= task <name_of_task>;
  <tf_declaration>+ 
  <statement_or_null>
  endtask 
<name_of_task>
  ::= <IDENTIFIER>
<tf_declaration>
  ::= <parameter_declaration>
  | | = <input_declaration>
  | | = <output_declaration>
  | | = <inout_declaration>
  | | = <reg_declaration>
  | | = <time_declaration>
  | | = <integer_declaration>
  | | = <real_declaration>
  | | = <event_declaration>
Task invocation

//task invocation syntax

<task_enable>
::= <name_of_task>;
||= <name_of_task>
(<expression><, <expression>>*)


Example 8-1  Input and Output Arguments in Tasks

//Define a module called operation which contains the task bitwise_oper
module operation;

parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;

initial
    $monitor( "%0d AB_AND = %b, AB_OR = %b, AB_XOR = %b, A = %b, B = %b", $time, AB_AND, AB_OR, AB_XOR, A, B);

initial
begin
    #1
    A = 16'b1111_0000_1010_0111;
    B = 16'b1010_0101_1000_1100;
end

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

Continued.........
Example 8-1: Input and Output Arguments in Tasks Continue..

```vhdl
//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
```
Example 8-2  Direct Operation on reg Variables

//Define a module which contains the task asymmetric_sequence
module sequence;

reg clock;

initial
begin
    $gr_waves("clock", clock);
    #1000 $stop;
end

initial
ininit_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
dendtask

Continued......
Example 8-2  Direct Operation on reg Variables Continue..

//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
Function Declaration

//function declaration syntax
<function>
 ::= function <range_or_type> ? <name_of_function>;  
     <tf_declaration>+  
     <statement>  
     endfunction

<range_or_type>
 ::= <range>  
    | I = <INTEGER>  
    | R = <REAL>

<name_of_function>
 ::= <IDENTIFIER>

<tf_declaration>
 ::= <parameter_declaration>  
    | I = <input_declaration>  
    | I = <reg_declaration>  
    | I = <time_declaration>  
    | I = <integer_declaration>  
    | R = <real_declaration>
Example 8-3: Parity Calculation

// Define a module which contains the function calc-parity
module parity;

reg [31:0] addr;
reg parity;

initial
begin
    addr = 32'h3456_789a;
    #10 addr = 32'hc4c6_78ff;
    #10 addr = 32'hff56_ff9a;
    #10 addr = 32'h3faa_aaaa;
end

// Compute new parity whenever address value changes
always @(addr)
begin
    parity = calc_parity(addr); // First invocation of calc-parity
    $display("Parity calculated = %b", calc_parity(addr));
    // Second invocation of calc-parity
end

Continued......
Example 8-3: Parity Calculation Continue...

//define the parity calculation function
function calc_parity;

input [31:0] address;

begin
    //set the output value appropriately. Use the implicit
    //internal register calc_parity.
    calc_parity = ^address;
    //Return the ex-or of all address
    bits.
end

endfunction

endmodule
Example 8-4: Left/Right Shifter

// Define a module which 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;

initial
begin
$monitor("%0d left= %h right = %h addr = %h", $time, left_addr, right_addr, addr);
    #1 addr = 32'h3456_789a;
    #10 addr = 32'hc4c6_78ff;
    #10 addr = 32'hff56_ff9a;
    #10 addr = 32'h3faa_aaaa;
end

// 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
Continued……
Example 8-4: Left/Right Shifter Cont

//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