

TASKS AND FUNCTIONS



Introduction

- A designer is frequently required to implement the same functionality at many places in a behavioral design.
- This means that the commonly used parts should be abstracted into routines and the routines must be invoked instead of repeating the code.
- Most programming languages provide procedures or subroutines to accomplish this. Verilog provides tasks and functions to break up large behavioral designs into smaller pieces.
- Tasks and functions allow the designer to abstract Verilog code that is used at many places in the design.
- Tasks have input, output, and inout arguments; functions have input arguments. Thus, values can be passed into and out from tasks and functions.

Differences between Tasks and Functions

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.

- ❑ Both tasks and functions must be defined in a module and are local to the module.
- ❑ Tasks are used for common Verilog code that contains delays, timing, event constructs, or multiple output arguments.
- ❑ Functions are used when common Verilog code is purely combinational, executes in zero simulation time, and provides exactly one output.
- ❑ Functions are typically used for conversions and commonly used calculations.
- ❑ Tasks can have input, output, and inout arguments; functions can have input arguments. In addition, they can have local variables, registers, time variables, integers, real, or events.
- ❑ Tasks or functions cannot have wires.
- ❑ Tasks and functions contain behavioral statements only.
- ❑ Tasks and functions do not contain always or initial statements but are called from always blocks, initial blocks, or other tasks and functions.

TASKS

- A task is like a procedure, it provides the ability to execute common pieces of code from several different places in a description. This common piece of code is written as a task **(using a task definition)** so that it can be called **(by a task call)** from different places in a design description.
- A task can contain timing controls, that is, delays, and it can call other tasks and functions as well
- Tasks are declared with the keywords **task and endtask.**
- Tasks must be used if any one of the following conditions is true for the procedure:
 - i. There are delay, timing, or event control constructs in the procedure.
 - ii. The procedure has zero or more than one output arguments.
 - iii. The procedure has no input arguments

Task Declaration and Invocation

- A task is defined using a task definition. It is of the form:

```
task task_id;  
    [ declarations ]  
    procedural_statement  
endtask
```

- A task can have zero, one, or more arguments.
- Values are passed to and from a task through arguments.
- In addition to input arguments (receive values for a task), a task can have output arguments (return values from a task) and inout arguments as well.
- A task definition is written within a module declaration

Example 1: A task definition:

Reverse the bits

```
module Has_Task;
  parameter MAXBITS = 8;

  task Reverse_Bits;
    input [MAXBITS - 1 : 0] Din;
    output [MAXBITS - 1 : 0] Dout;
    integer K;

    begin
      for (K = 0; K < MAXBITS; K = K + 1)
        Dout [MAXBITS - K] = Din [K];
      end
    endtask
  . . .
endmodule
```

- The inputs and outputs of a task are declared at the beginning of the task.
- The order of these inputs and outputs specify the order to be used in a task call.



Example 2: Rotate left



```

task Rotate_Left;
  inout [1:16] In_Arr;
  input [0:3] Start_Bit, Stop_Bit, Rotate_By;
  reg Fill_Value;
  integer Mac1, Mac3;

begin
  for (Mac3 = 1; Mac3 <= Rotate_By; Mac3 = Mac3 + 1)
  begin
    Fill_Value = In_Arr [Stop_Bit];
    for (Mac1 = Stop_Bit; Mac1 >= Start_Bit + 1;
      Mac1 = Mac1 - 1)
      In_Arr [Mac1] = In_Arr [Mac1 - 1];

    In_Arr [Start_Bit] = Fill_Value;
  end
end
endtask

```

- 
- ❑ Fill_Value is a local register that is directly visible only within the task.
 - ❑ The first argument in this task is the inout array, In_Arr, followed by the three inputs, Start_Bit, Stop_Bit and Rotate_By.
 - ❑ In addition to the task arguments, a task can reference any variable defined in the module in which the task is declared.

Task Calling

- A task is called (or enabled, as it is said in Verilog HDL) by a task enable statement that specifies the argument values passed to the task and the variables that receive the results.
- A task enable statement is a procedural statement and can thus appear within an always or an initial statement. ‘
- It is of the form:
task_id [(expr1 , expr2 , . . . , exprN)] ;
- The list of arguments must match the order of input, output and inout declarations in the task definition. In addition, arguments are passed by value, not by reference.
- A task can be called more than once concurrently with each call having its own control.

- The biggest point to be careful is that a variable declared within a task is static, that is, it never disappears or gets re-initialized. Thus one task call might modify a local variable whose value may be read by another task call


- Here is an example of a task call for the task Reverse_Bits

```
// Register declaration:
```

```
reg [MAXBITS - 1 : 0] Reg_X, New_Reg;
```

```
Reverse_Bits (Reg_X, New_Reg);           // Calling task.
```

- The value of Reg_X is passed as the input value, that is, to Din. The output of the task Dout is returned back to New_Reg.
- Note that because a task can contain timing controls, a task may return a value later in time than when it was called. **The output and inout arguments in a task call must be registers** because a task enable statement is a procedural statement. In the above example, New_Reg must be declared as a register.



Example 3: A task that references a variable that is not passed in through its argument list. Even though referencing global variables is considered bad programming style, it is sometimes useful as shown in the following example.

```
module Global_Var;
```

```
  reg [0:7] RamQ [0:63];
```

```
  integer Index;
```

```
  reg CheckBit;
```

```
  task GetParity;
```

```
    input Address;
```

```
    output ParityBit;
```

```
    ParityBit = ^ RamQ [Address];
```

```
  endtask
```

```
  initial
```

```
    for (Index = 0; Index <= 63; Index = Index + 1) begin
```

```
      GetParity (Index, CheckBit);
```

```
      $display ("Parity bit of memory word %d is %b.",  
                Index, CheckBit);
```

```
    end
```

```
  endmodule
```

- *The address of the memory RamQ is passed as an argument and the memory is referenced directly within the task.*

Eg: GENERATE WAVEFORM (X)

```
module TaskWait;  
    reg NoClock;  
  
    task GenerateWaveform;  
        output ClockQ;  
        begin  
            ClockQ = 1;  
            #2 ClockQ = 0;  
            #2 ClockQ = 1;  
            #2 ClockQ = 0;  
        end  
    endtask  
  
    initial  
        GenerateWaveform (NoClock);  
endmodule
```

- A task can have delays and or it can wait for certain events to occur.
- However, an assignment to an output argument is not passed to the calling argument until the task exits.
- The assignments to ClockQ do not appear on NoClock, that is, no waveform appears on NoClock; only the final assignment to ClockQ, which is 0, appears on NoClock after the task returns.
- One way to avoid this problem is to make ClockQ as a global register, that is, declare it outside the task.

FUNCTIONS

- ❑ Functions, similar to tasks, also provide the capability to execute common code from different places in a module.
- ❑ The difference from a task is that a function can return only one value, it cannot contain any delays (must execute in zero time) and it cannot call any other task.
- ❑ In addition, a function must have at least one input. No output or inout declarations are allowed in a function.
- ❑ A function may call other functions. .

Function Definition

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

```
function [ range ] function_id;  
    input_declaration  
    other_declarations  
    procedural_statement  
endfunction
```


- An input to the function is declared using the input declaration.
- If no range is specified with the function definition, then a 1-bit return value is assumed.

e.g.: Reverse bits using function

```
module Function_Example;
    parameter MAXBITS = 8;

    function [MAXBITS - 1 : 0] Reverse_Bits;
        input [MAXBITS - 1 : 0] Din;
        integer K;
        begin
            for (K = 0; K < MAXBITS; K = K + 1)
                Reverse_Bits [MAXBITS - K - 1] = Din [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, Din.*
- *K is a local integer.*

- 
- The function definition implicitly declares a register internal to the function, with the same name and range as the function.
 - A function returns a value by assigning a value to this register explicitly in the function definition.
 - An assignment to this register must therefore be present within a function definition.

e.g. 2: Parity Function

```
function Parity;  
  input [0:31] Set;  
  reg [0:3] Ret;  
  integer J;  
  begin  
    Ret = 0;  
  
    for (J = 0; J <= 31; J = J + 1)  
      if (Set[J] == 1)  
        Ret = Ret + 1;  
  
    Parity = Ret % 2;  
  end  
endfunction
```

- In this function, *Parity* is the name of the function.
- Since no size has been specified, the function returns a 1-bit value.
- *Ret* and *J* are local registers.
- Note that the last procedural assignment assigns a value to the register which returns the value from the function (a register with the same name as function is implicitly declared within the function)

Function Call

- A function call is part of an expression. It is of the form

func_id (expr1 , expr2 , . . . , exprN)

- Here is an example of a function call

```
reg [MAXBITS - 1 : 0] New_Reg, Reg_X;    // Reg declaration
```


```
New_Reg = Reverse_Bits(Reg_X);  
// Function call in right-hand side expression.
```

- Similar to a task, all local registers declared within a function definition are static, that is, local registers within a function retain their values between multiple invocations of the function

Summary



- Tasks and functions used in behavior Verilog modeling.
- ❖ Tasks and functions are used to define common Verilog functionality that is used at many places in the design. Tasks and functions help to make a module definition more readable by breaking it up into manageable subunits. Tasks and functions serve the same purpose in Verilog as subroutines do in C.
- ❖ Tasks can take any number of input, inout, or output arguments. Delay, event, or timing control constructs are permitted in tasks. Tasks can enable other tasks or functions.
- ❖ Re-entrant tasks defined with the keyword `automatic` allow each task call to operate in an independent space. Therefore, re-entrant tasks work correctly even with concurrent tasks calls.

- 
- ❖ Functions are used when exactly one return value is required and at least one input argument is specified. Delay, event, or timing control constructs are not permitted in functions. Functions can invoke other functions but cannot invoke other tasks.
 - ❖ A register with name as the function name is declared implicitly when a function is declared. The return value of the function is passed back in this register.
 - ❖ Recursive functions defined with the keyword automatic allow each function call to operate in an independent space. Therefore, recursive or concurrent calls to such functions will work correctly.
 - ❖ Tasks and functions are included in a design hierarchy and can be addressed by hierarchical name referencing



THANK YOU