: Verilog Functions

- A key feature of VLSI circuits is the repeated use of similar structures.
- Verilog provides functions and tasks to easily express repeated invocation of the same functionality or the repeated use of structures.
- A Verilog function is similar to a Verilog task, which will be described later. They have small differences: a function cannot drive more than one output, nor can a function contain delays.

- The functions can also be defined in separate files, and the compile directive **`include** will be used to include the function in the file
- The functions should be executed in zero time delay, which means that the functions cannot include timing delay information. The functions can have any number of inputs but can return only a single output.
- A function returns a value by assigning the value to the function name. The function's return value can be a single bit or multiple bits. The variables declared within the function are local variables, but the functions can also use global variables when no local variables are used. When local variables are used, basically output is assigned only at the end of function execution. The functions can call other functions but cannot call tasks. Functions are frequently used to do type conversions.
- A function executes a sequential algorithm and returns a single value to the calling program.

When the following function is called, it returns a value equal to the input rotated one position to the right

```
function [7:0] rotate_right;
  input [7:0] reg_address;
  begin
    rotate_right = reg_address >> 1;
  end
endfunction
```

A function call can be used anywhere that an expression can be used. For example, if $A = 10010101$, the statement

```
B <= rotate_right(A);
```

would set B equal to 11001010 and leave A unchanged.

The general form of a function declaration is

```
function <range or type> function-name
  input [declarations]
<declarations>     // reg, parameter, integer, etc.
  begin
  sequential statements
  end
endfunction
```

The general form of a function call is

```
function_name(input-argument-list)
```

The number and type of parameters on the `input-argument-list` must match the input [declaration] in the function declaration. The parameters are treated as input values and cannot be changed during the execution of the function.

The general form of a function call is

```
function_name(input-argument-list)
```

The number and type of parameters on the `input-argument-list` must match the input [declaration] in the function declaration. The parameters are treated as input values and cannot be changed during the execution of the function.

If parity circuits are used in several parts in a system, one could call the function each time it is desired. The function can be called as follows:

```
module function_test(Z);
  output reg [4:0] Z;
  reg [3:0] INP;
  initial
  begin
    INP = 4'b0101;
    Z = parity(INP);
  end

endmodule
```

Illustrates a function using a **for** loop. The loop index *i* will be initialized to 0 when the **for** loop is entered, and the sequential statements will be executed. Execution will be repeated for *i* = 1, *i* =2,

and *i* = 3; then the loop will terminate.

```
// This function adds two 4-bit vectors and a carry.
// Illustrates function creation and use of loop.
// It returns a 5-bit sum.

function [4:0] add4;
  input [3:0] A;
  input [3:0] B;
  input       cin;

  reg   [4:0] sum;
  reg         cout;

  begin
    integer i;
    for (i=0; i<=3; i=i+1)
    begin
      cout = (A[i] & B[i]) | (A[i] & cin) | (B[i] & cin);
      sum[i] = A[i] ^ B[i] ^ cin;
      cin = cout;
  end

   sum[4] = cout;
   add4 = sum;
 end
endfunction
```

A Function to Compute Squares and Its Call

```
module test_squares (CLK);

    input CLK;
    reg[3:0] FN;
    reg[7:0] answer;

    function [7:0] squares;
        input[3:0] Number;

        begin
            squares = Number * Number;
        end
    endfunction

    initial
    begin

        FN = 4'b0011;
    end

    always @(posedge CLK)
    begin
        answer = squares(FN);
    end
```

Verilog Tasks

- Tasks facilitate decomposition of Verilog code into modules. Unlike functions, which return only a single value implicitly with the function name, tasks can return any number of values. The form of a task declaration is

task task_name

input [declarations]

output [declarations]

<declarations> // reg, parameter, integer, etc.

begin

sequential statements

end task_name;

- The formal-parameter-list specifies the inputs and outputs to the task and their types. A task call is a sequential or concurrent statement of the form

task_name(actual-parameter-list);

- Unlike functions, a task can be executed in non-zero simulation time. Tasks may contain delay, event, or timing control statements. Tasks can have zero or more arguments of type input, output, or inout. Tasks do not return with a value, but they can pass multiple values through **output** and **inout** arguments.
- As an example, we will write a task *Addvec*, which will add two *4*-bit data and a carry and return a *4*-bit sum and a carry. We will use a task call of the form

```
Addvec(A, B, Cin, Sum, Cout);
```

where *A*, *B*, and *Sum* are *4*-bit data and *Cin* and *Cout* are 1-bit data.

- **Functions:**
- At least one input arguments, but no output or inout arguments Returns a single value by assigning the value to the function name Can call other functions, but cannot call tasks
- Cannot embed delays, wait statements or any time-controlled statement Executes in zero time
- Can be recursive
- Cannot contain non-blocking assignment or procedural continuous assignments

```
// This task adds two 4-bit data and a carry and
// returns an n-bit sum and a carry. Add1 and Add2 are assumed
// to be of the same length and dimensioned 3 downto 0.

    task Addvec;
      input [3:0] Add1;
      input [3:0] Add2;
      input Cin;
      output [3:0] sum;
      output cout;

      reg C;

      begin
         C = Cin;
         integer i;
         for(i = 0; i <= 4; i = i + 1)
            begin
               sum[i] = Add1[i] ^ Add2[i] ^ C ;
               C = (Add1[i] & Add2[i]) | (Add1[i] & C) | (Add2[i] & C);
            end
         cout = C ;
      end
```

```
        integer i;
        for(i = 0; i <= 4; i = i + 1)
            begin
                sum[i] = Add1[i] ^ Add2[i] ^ C ;
                C = (Add1[i] & Add2[i]) | (Add1[i] & C) | (Add2[i] & C);
            end
        cout = C ;
    end
endtask
```

same as the one used in the *add4* function. *C* must be a variable, since the new value of *C* is needed each time through the loop; hence it is declared as **reg**. After *4* times through the loop, all 4 bits of *sum* have been computed. It is desirable not to mix blocking and non-blocking statements in tasks.

The tasks can be defined in the module that the functions will be used in. The tasks can also be defined in separate files, and the compile directive '**include should be used (')** (described in Section 8.12) will be used to include the task in the file. The variables declared within the task are local to the task. When no local variables are used, global variables will be used for input and output. When only local variables are used within the task, the variables from the last execution within the task will be passed to the caller.

Functions:
At least one input arguments, but no output or inout arguments

through the loop, all 4 bits of *sum* have been computed. It is desirable not to mix blocking and non-blocking statements in tasks.

The tasks can be defined in the module that the functions will be used in. The tasks can also be defined in separate files, and the compile directive '**include should be used (')** (described in Section 8.12) will be used to include the task in the file. The variables declared within the task are local to the task. When no local variables are used, global variables will be used for input and output. When only local variables are used within the task, the variables from the last execution within the task will be passed to the caller.

Functions:
At least one input arguments, but no output or inout arguments
Returns a single value by assigning the value to the function name
Can call other functions, but cannot call tasks
Cannot embed delays, wait statements or any time-controlled statement
Executes in zero time
Can be recursive
Cannot contain non-blocking assignment or procedural continuous assignments