12.2 The $root top level

In SystemVerilog there is a top level called **`$root`**, which is the whole source text. This allows declarations outside any named modules or interfaces, unlike Verilog-2001.

SystemVerilog requires an elaboration phase. All modules and interfaces must be parsed before elaboration and the order of elaboration must be defined.

The source text can include the declaration and use of modules and interfaces. Modules can include the declaration and use of other modules and interfaces. Interfaces can include the declaration and use of other interfaces. A module or interface need not be declared before it is used in text order.

If there is no explicit top level instantiation, then all uninstantiated modules become implicitly instantiated within the top level. This is compatible with Verilog-2001.

The following paragraphs compare the $root top level and modules.

The $root top level:

— has a single occurrence

— can be distributed across any number of files

— localparam, variable, and net definitions are in a global name space and can be accessed throughout the hierarchy if they have been imported into the module where they are accessed

— task and function definitions are in a global name space and can be accessed throughout the hierarchy if they have been imported into the module where they are accessed

— constant (using keyword const) and type (using keyword typedef) declarations are in a global name space and can be accessed throughout the hierarchy without requiring an import

— named blocks must be accessed using $root.*block_name*

— may not contain initial or always procedures

— may contain procedural statements, which will be executed one time, as if in an initial procedure

Modules:

— can have any number of module definitions

— can have any number of module instances, which create new levels of hierarchy

— can be distributed across any number of files, and can be defined in any order

— variable and net definitions are in the module instance name space and are local to that scope

— task and function definitions are in the module instance name space and are local to that scope

— may contain any number of initial and always procedures

— may not contain procedural statements that are not within an initial procedure, always procedure, task, or function

12.2.1 Accessing the contents of $root

Statements and instantiations in $root can freely use the declarations made in $root and in the hierarchy below $root. Although all these declarations can be accessed from $root, some of them must be imported into the module in which they are accessed. Module, interface, type and constant declarations can be accessed without an import.

Because unintentional accessing of global data declarations, parameters, tasks and functions can be difficult to debug, maintain, and would be a common error for beginning Verilog users, they must be imported into the module before they are used. The import can be done before or after they used. Importing is done by supplying the name of the declaration.

In the following examples that show how a system might be initially modeled as functions in $root, but incrementally converted to modules. Note that the only place an import is required is when the task $root.left is called from module main.

(NOTE: The examples were taken from Simon's email 12-10-2001. I modified the middle one.)

Example xxx:

```
// usage of root without modules, just functions
typedef int myint;

function void main ();
  myint i,j,k;
  $display ("entering main...");
  left (k);
  right (i,j,k);
  $display ("ending... i=%0d, j=%0d, k=%0d", i,
j, k);
endfunction

function void left (output myint k);
  k = 34;
  $display ("entering left");
endfunction

function void right (output myint i, j, input
myint k);
  $display ("entering right");
  i = k/2;
  j = k+i;
endfunction
```

```
main();
```

Example xxx:

```
// usage of root with both modules and functions

typedef int myint;

module top ();
  myint i,j,k;
  import left;
  initial $display ("in top module...");
  left (k);
  right r (i,j,k);
  initial #100 $display ("ending... i=%0d, j=%0d,
k=%0d", i, j, k);
endmodule

function void left (output myint k);
  k = 34;
  $display ("entering left");
endfunction

module right (output myint i, j, input myint k);
  function void right (output myint i, j, input
myint k);
    $display ("entering right");
    i = k/2;
    j = k+i;
  endfunction
  always @(k) right (i, j, k);
endmodule

top main();
```

Example xxx:

```
// usage of root with modules

typedef integer myint;

module main ();
  myint i,j,k;
  initial $display ("starting in main...");
  left l (k);
  right r (i,j,k);
```

```
  initial #100 $display ("ending... i=%0d, j=%0d,
k=%0d", i, j, k);
endmodule

module left (output myint k);
  initial begin
    #10 k = 34;
    $display ("activating left");
  end
endmodule

module right (output myint i, j, input myint k);
  always @(k) begin
    $display ("activating right");
    i = k/2;
    j = k+i;
  end
endmodule

main m1();
```