

2.7.2 Generated identifiers

Generated identifiers are created by generate loops (see [12.1.3.2](#)); and are a special case of identifiers in that they can be used in hierarchical names (see [12.4](#)). A generated identifier is the named generate block identifier terminated with a `{[digit(s)]}` string. This identifier is used as a node name in hierarchical names (see [12.4](#)).

3.5 Implicit declarations

The syntax shown in [3.2](#) shall be used to declare nets and variables explicitly. In the absence of an explicit declaration, an implicit net of default net type shall be assumed in the following circumstances:

- If an identifier is used in a port expression declaration, then an implicit net of default net type shall be assumed, with the vector width of the port expression declaration. See [12.3.3](#) for a discussion of port expression declarations.
- If an identifier is used in the terminal list of a primitive instance or a module instance, and that identifier has not been explicitly declared previously in one of the declaration statements of the instantiating module, then an implicit scalar net of default net type shall be assumed.
- If an identifier appears on the left-hand side of a continuous assignment statement, and that identifier has not been declared previously, then an implicit scalar net of default net type shall be assumed. See [6.1.2](#) for a discussion of continuous assignment statements.

See [19.2](#) for a discussion of control of the type for implicitly declared nets with the ``default_nettype` compiler directive.

3.10.1 Net arrays

Arrays of nets can be used to connect ports of generated instances. Each element of the array can be used in the same fashion as a scalar or vector net.

3.12 Name spaces

In Verilog HDL, there are seven name spaces; two are global and five are local. The global name spaces are *definitions* and *text macros*. The *definitions name space* unifies all the **module** (see [12.1](#)), **macromodule** (see [12.1](#)), and **primitive** (see [8.1](#)) definitions. Once a name is used to define a module, macromodule, or primitive, the name shall not be used again to declare another module, macromodule, or primitive.

The *text macro name space* is global. Since text macro names are introduced and used with a leading ` character, they remain unambiguous with any other name space (see [19.3](#)). The text macro names are defined in the linear order of appearance in the set of input files that make up the description of the design unit. Subsequent definitions of the same name override the previous definitions for the balance of the input files.

There are five local name spaces: *block*, *module*, *port*, *specify block*, and *attribute*. Once a name is defined within the *block*, *module*, *port*, or *specify block* name space, it shall not be defined again in that space (with the same or a different type). As described in [2.8](#), it is legal to redefine names within the *attribute* name space.

The *block name space* is introduced by the named block (see [9.8](#)), function (see [10.3](#)), and task (see [10.2](#)) constructs. It unifies the definitions of the named blocks, functions, tasks, parameters, named events and the variable type of declaration (see [3.2.2](#)). The variable type of declaration includes the **reg**, **integer**, **time**, **real**, and **realtime** declarations.

The *module name space* is introduced by the **module**, **macromodule**, and **primitive** constructs. It unifies the definition of functions, tasks, named blocks, instance names, parameters, named events, net type of declaration, and variable type of declaration. The net type of declaration includes **wire**, **wor**, **wand**, **tri**, **trior**, **triand**, **tri0**, **tri1**, **trireg**, **supply0**, and **supply1** (see [3.7](#)).

The *port name space* is introduced by the **module**, **macromodule**, **primitive**, **function**, and **task** constructs. It provides a means of structurally defining connections between two objects that are in two different name spaces. The connection can be unidirectional (either **input** or **output**) or bidirectional (**inout**). The port name space overlaps the module and the block name spaces. Essentially, the port name space specifies the type of connection between names in different name spaces. The port type of declarations include **input**, **output**, and **inout** (see [12.3](#)). A port name introduced in the port name space may be reintroduced in the module name space by declaring a variable or a wire with the same name as the port name.

The *specify block name space* is introduced by the **specify** construct (see [14.2](#)).

The *attribute name space* is enclosed by the (* and *) constructs attached to a language element (see [2.8](#)). An attribute name can be defined and used only in the attribute name space. Any other type of name cannot be defined in this name space.

10.3.5 Use of constant functions

Constant function calls are used to support the building of complex calculations of values at elaboration time (see [12.1.3](#)). A *constant function call* shall be a function invocation of a *constant function* local to the calling module where the arguments to the function are *constant expressions*. *Constant functions* are a subset of normal Verilog functions that shall meet the following constraints:

- They shall contain no hierarchical references.
- Any function invoked within a *constant function* shall be a *constant function* local to the current module.
- All system tasks within a constant function shall be ignored.
- All system functions within a constant function shall be illegal.
- All parameter values used within the function shall be defined before the use of the invoking *constant function call* (i.e. any parameter use in the evaluation of a *constant function call* constitutes a use of that parameter at the site of the original *constant function call*).
- All identifiers which are not parameters or functions shall be declared locally to the current function.
- If they use any parameter value that is affected directly or indirectly by a **defparam** statement (see [12.2.1](#)), the result is undefined. This can produce an error or the constant function can return an indeterminate value.
- **They shall not be declared inside a generate scope.**
- They shall not themselves use constant functions in any context requiring a constant expression.

Constant function calls are evaluated at elaboration time. Their execution has no effect on the initial values of the variables used either at simulation time or among multiple invocations of a function at elaboration time. In each of these cases, the variables are initialized as they would be for normal simulation.

12.1 Modules

```

module_declaration ::= (From Annex A - A.1.3)
    { attribute_instance } module_keyword module_identifier [ module_parameter_port_list ]
    list_of_ports ; { module_item }
    endmodule
    | { attribute_instance } module_keyword module_identifier [ module_parameter_port_list ]
    [ list_of_port_declarations ] ; { non_port_module_item }
    endmodule

module_keyword ::= module | macromodule

module_parameter_port_list ::= (From Annex A - A.1.4)
    # ( parameter_declaration { , parameter_declaration } )

list_of_ports ::= ( port { , port } )

list_of_port_declarations ::= ( port_declaration { , port_declaration } ) | ( )

port ::= [ port_expression ] | . port_identifier ( [ port_expression ] )

port_expression ::= port_reference | { port_reference { , port_reference } }

port_reference ::= port_identifier [ [ constant_range_expression ] ]

port_declaration ::= { attribute_instance } inout_declaration
    | { attribute_instance } input_declaration
    | { attribute_instance } output_declaration

module_item ::= (From Annex A - A.1.5)
    port_declaration ;
    | non_port_module_item

module_or_generate_item ::= { attribute_instance } module_or_generate_item_declaration
    | { attribute_instance } parameter_override
    | { attribute_instance } continuous_assign
    | { attribute_instance } gate_instantiation
    | { attribute_instance } udp_instantiation
    | { attribute_instance } module_instantiation
    | { attribute_instance } initial_construct
    | { attribute_instance } always_construct

module_or_generate_item_declaration ::=
    net_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration
    | time_declaration
    | realtime_declaration
    | event_declaration
    | genvar_declaration
    | task_declaration
    | function_declaration

non_port_module_item ::=
    module_or_generate_item
    | generated_instantiation
    | specify_block
    | { attribute_instance } local_parameter_declaration ;
    | { attribute_instance } parameter_declaration ;
    | { attribute_instance } specparam_declaration

parameter_override ::= defparam list_of_defparam_assignments ;

```

Syntax 12-1—Syntax for module

12.1.1 Top-level modules

Top-level modules are modules that are included in the source text but are not instantiated, as described in [12.1.2](#).

12.1.3 Generated instantiation

After a Verilog design has been parsed, but before simulation begins, the design must have the modules being instantiated linked to the modules being defined, the parameters propagated among the various modules, and hierarchical references resolved. This phase in understanding a Verilog description is termed *elaboration*.

Generate instantiations are resolved during elaboration because that is when the parameters associated with a module become defined, hence, allowing the definition of the generated statements and declarations. Genvars are only defined during the evaluation of the generate instantiations and do not exist during simulation of a design.

Generate statements facilitate the creation of parameterized models. When used with constant functions (see [10.3.5](#)), parameters can be used to constrain other parameter(s) or localparam(s) in a generated design.

All generate instantiations are coded within a module scope and require the keywords **generate - endgenerate**.

Generate statements allow control over the declaration of variables, functions and tasks, as well as control over instantiations. Generated instantiations are one or more: modules, user defined primitives, Verilog gate primitives, continuous assignments, initial blocks and always blocks. Generated declarations and instantiations can be conditionally instantiated into a design. Generated variable declarations and instantiations can be multiply instantiated into a design. Generated instances have unique identifier names and can be referenced hierarchically as described in [12.4](#).

To support the interconnection between structural elements and/or procedural blocks, generate statements permit the following Verilog data types to be declared within the generate scope: **net**, **reg**, **integer**, **real**, **time**, **realtime**, and **event**. Generated data types have unique identifier names and can be referenced hierarchically as described in [12.4](#).

Parameters may be redefined using *defparam statements* (see [12.2.1](#)) or *module instance parameter value assignments* (see [12.2.2](#)) within the generate scope. However, a *defparam statement* within the generate scope or within a hierarchy instantiated within the generate scope shall only modify the value of a parameter declared within the generate scope or within a hierarchy instantiated within the generate scope.

Task and function declarations shall also be permitted within the generate scope. Generated tasks and functions shall have unique identifier names and may be referenced hierarchically as described in [12.4](#).

Module declarations and module items that shall not be permitted in a generate statement include: parameters, local parameters, input declarations, output declarations, inout declarations and specify blocks.

Connections to generated module instances are handled the same way as they are handled with normal module instances as described in [12.1.2](#).

Generated statements are created using one of the following three methods: generate-loop, generate-conditional, or generate-case.

The syntax for generate instantiations is given in [Syntax 12-2](#).

```

module_item ::= (From Annex A - A.1.5)
    port_declaration ;
    | non_port_module_item
module_or_generate_item ::=
    { attribute_instance } module_or_generate_item_declaration
    | { attribute_instance } parameter_override
    | { attribute_instance } continuous_assign
    | { attribute_instance } gate_instantiation
    | { attribute_instance } udp_instantiation
    | { attribute_instance } module_instantiation
    | { attribute_instance } initial_construct
    | { attribute_instance } always_construct
module_or_generate_item_declaration ::=
    net_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration
    | time_declaration
    | realtime_declaration
    | event_declaration
    | genvar_declaration
    | task_declaration
    | function_declaration
generated_instantiation ::= (From Annex A -A.4.2)
    generate { generate_item } endgenerate
generate_item_or_null ::=
    generate_item | ;
generate_item ::=
    generate_conditional_statement
    | generate_case_statement
    | generate_loop_statement
    | generate_block
    | module_or_generate_item
generate_conditional_statement ::=
    if ( constant_expression ) generate_item_or_null [ else generate_item_or_null ]
generate_case_statement ::= case ( constant_expression )
    genvar_case_item { genvar_case_item } endcase
genvar_case_item ::= constant_expression { , constant_expression } :
    generate_item_or_null | default [ : ] generate_item_or_null
generate_loop_statement ::=
    for ( genvar_assignment ; constant_expression ; genvar_assignment )
        begin : generate_block_identifier { generate_item } end
genvar_assignment ::=
    genvar_identifier = constant_expression
generate_block ::=
    begin [ : generate_block_identifier ] { generate_item } end

```

Syntax 12-2—Syntax for generate blocks

12.1.3.1 genvar - generate statement index variable

An index variable that shall only be declared for use in generate statements shall be declared as a *genvar* and is referred to as a *genvar* in the rest of this section.

The syntax for generate statement index variable declarations is given in [Syntax 12-3](#).

```
genvar_declaration ::= (From Annex A - A.2.1.3)  
    genvar list_of_genvar_identifiers ;  
list_of_genvar_identifiers ::= (From Annex A - A.2.3)  
    genvar_identifier { , genvar_identifier }
```

Syntax 12-3—Syntax for generate statement index variable declaration

A *genvar* shall be declared within the module where the *genvar* is used. A *genvar* can be declared either inside or outside of a generate scope. A *genvar* is an integer that is local to and shall only be used within a generate loop that uses it as an index variable. If any bit of the *genvar* ever is set to an X or Z or if the *genvar* is set to a negative value, this shall be an error.

Genvars are only defined during the evaluation of the generate blocks (see [12.1.3](#)), and do not exist during simulation of a Verilog design.

The value of a *genvar* shall only be defined by a generate loop. Two generate loops using the same *genvar* as an index variable shall not be nested. The value of a *genvar* can be referenced in any context where the value of a parameter could be referenced.

12.1.3.2 generate-loop

A generate-loop permits one or more variable declarations, modules, user defined primitives, gate primitives, continuous assignments, initial blocks and always blocks to be instantiated multiple times using a for-loop. The index loop variable used in a generate for-loop shall be declared as a *genvar*. Both *genvar* assignments in the for-loop shall assign to the same *genvar*, which is the loop index variable. The first *genvar* assignment in the for-loop shall not reference the loop index variable on the right-hand side.

Examples:

Example 1—A parameterized gray-code to binary-code converter module using a loop to generate continuous assignments

```
module gray2bin1 (bin, gray);  
  parameter SIZE = 8;      // this module is parameterizable  
  output [SIZE-1:0] bin;  
  input  [SIZE-1:0] gray;  
  
  genvar i;  
  
  generate for (i=0; i<SIZE; i=i+1) begin:bit  
    assign bin[i] = ^gray[SIZE-1:i];  
  end endgenerate  
endmodule
```

Example 2—The same gray-code to binary-code converter module in example 1 is built using a loop to generate always blocks

```
module gray2bin2 (bin, gray);  
  parameter SIZE = 8;      // this module is parameterizable  
  output [SIZE-1:0] bin;  
  input  [SIZE-1:0] gray;  
  reg    [SIZE-1:0] bin;  
  
  genvar i;  
  
  generate for (i=0; i<SIZE; i=i+1) begin:bit  
    always @(gray[SIZE-1:i]) // fixed part-select  
      bin[i] = ^gray[SIZE-1:i];  
  end endgenerate  
endmodule
```

The models in examples 3 and 4 are parameterized modules of ripple adders using a loop to generate Verilog gate primitives. Example 3 uses a two-dimensional net declaration outside of the generate loop to make the connections between the gate primitives while example 4 makes the net declaration inside of the generate loop to generate the wires needed to connect the gate primitives for each iteration of the loop.

Example 3—Generated ripple adder with two-dimensional net declaration outside of the generate loop

```

module addergen1 (co, sum, a, b, ci);
  parameter SIZE = 4;
  output [SIZE-1:0] sum;
  output          co;
  input  [SIZE-1:0] a, b;
  input          ci;
  wire   [SIZE :0] c;
  wire   [SIZE-1:0] t [1:3];
  genvar          i;

  assign c[0] = ci;

  // Generated instance names are:
  // xor gates: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
  //             bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
  // and gates: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
  //             bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
  // or  gates: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
  // Generated instances are connected with
  // multi-dimensional nets t[1][3:0] t[2][3:0] t[3][3:0]
  // (12 multi-dimensional nets total)
  generate
    for(i=0; i<SIZE; i=i+1) begin:bit
      xor g1 ( t[1][i],  a[i],  b[i]);
      xor g2 (  sum[i], t[1][i],  c[i]);
      and g3 ( t[2][i],  a[i],  b[i]);
      and g4 ( t[3][i], t[1][i],  c[i]);
      or  g5 (  c[i+1], t[2][i], t[3][i]);
    end
  endgenerate

  assign co = c[SIZE];
endmodule

```

Example 4—Generated ripple adder with net declaration inside of the generate loop

```

module addergen1 (co, sum, a, b, ci);
  parameter SIZE = 4;
  output [SIZE-1:0] sum;
  output          co;
  input  [SIZE-1:0] a, b;
  input          ci;
  wire   [SIZE :0] c;

  genvar          i;

  assign c[0] = ci;

  // Generated instance names are:
  // xor gates: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
  //             bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
  // and gates: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
  //             bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
  // or  gates: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
  // Generated instances are connected with
  // generated nets: bit[0].t1 bit[1].t1 bit[2].t1 bit[3].t1
  //                 bit[0].t2 bit[1].t2 bit[2].t2 bit[3].t2
  //                 bit[0].t3 bit[1].t3 bit[2].t3 bit[3].t3
  generate
    for(i=0; i<SIZE; i=i+1) begin:bit
      wire  t1, t2, t3; // generated net declaration

      xor g1 ( t1, a[i], b[i]);
      xor g2 ( sum[i], t1, c[i]);
      and g3 ( t2, a[i], b[i]);
      and g4 ( t3, t1, c[i]);
      or  g5 ( c[i+1], t2, t3);
    end
  endgenerate

  assign co = c[SIZE];
endmodule

```

The generated instance names in a multi-level generate loop are shown in example 5. The generated name for the scope at each generate loop is created by adding the "[genvar's value]" string to the end of the generate block identifier for the loop. The generated names are now generated identifiers (see [2.7.2](#)) which can be used in hierarchical path names (see [12.4](#)).

Example 5—A multi-level generate loop

```

parameter SIZE = 2;
genvar      i, j, k, m;
generate
  for (i=0; i<SIZE; i=i+1) begin:B1 // scope B1[i]
    M1 N1(); // instantiates B1[i].N1
    for (j=0; j<SIZE; j=j+1) begin:B2 // scope B1[i].B2[j]
      M2 N2(); // instantiates B1[i].B2[j].N2
      for (k=0; k<SIZE; k=k+1) begin:B3 // scope B1[i].B2[j].B3[k]
        M3 N3(); // instantiates B1[i].B2[j].B3[k].N3
      end
    end
  if (i>0)
    for (m=0; m<SIZE; m=m+1) begin:B4 // scope B1[i].B4[m]
      M4 N4(); // instantiates B1[i].B4[m].N4
    end
  end
endgenerate

// some of the generated instance names are:
// B1[0].N1           B1[1].N1
// B1[0].B2[0].N2    B1[0].B2[1].N2
// B1[0].B2[0].B3[0].N3 B1[0].B2[0].B3[1].N3
// B1[0].B2[1].B3[0].N3
// B1[1].B4[0].N4    B1[1].B4[1].N4

```

12.1.3.3 generate-conditional

A generate-conditional is an if-else-if generate construct that permits modules, user defined primitives, Verilog gate primitives, continuous assignments, initial blocks and always blocks to be conditionally instantiated into another module based on an expression that is deterministic at the time the design is elaborated.

Example 6 shows the implementation of a parameterized module. If either of the multiplier's `a_width` or `b_width` parameters are less than 8 (bits), a CLA multiplier is instantiated. If both of the multiplier's `a_width` or `b_width` parameters are greater than or equal to 8 (bits), a Wallace tree multiplier is instantiated.

Example 6—An implementation of a parameterized multiplier module

```

module multiplier(a,b,product);
parameter a_width = 8, b_width = 8;
localparam product_width = a_width+b_width; // can not be modified
// directly with the defparam statement
// or the module instance statement #
input    [a_width-1:0]    a;
input    [b_width-1:0]    b;
output   [product_width-1:0]    product;

generate
  if((a_width < 8) || (b_width < 8))
    CLA_multiplier #(a_width,b_width) u1(a, b, product);
    // instantiate a CLA multiplier
  else
    WALLACE_multiplier #(a_width,b_width) u1(a, b, product);
    // instantiate a Wallace-tree multiplier
endgenerate
// The generated instance name is u1

endmodule

```

12.1.3.4 generate-case

A generate case construct permits modules, user defined primitives, Verilog gate primitives, continuous assignments, initial blocks and always blocks to be conditionally instantiated into another module based on a select one-of-many case construct. The selecting case expression must be deterministic at the time the design is elaborated.

Example 7—Generate with a case to handle widths less than 3

```

generate
  case (WIDTH)
    1: adder_1bit x1(co, sum, a, b, ci);
    // 1-bit adder implementation
    2: adder_2bit x1(co, sum, a, b, ci);
    // 2-bit adder implementation
    default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);
    // others - carry look-ahead adder
  endcase
// The generated instance name is x1

endgenerate

```

Example 8—A module of memory dimm

```

module dimm(adr, ba, rasx, casx, csx, wex, cke, clk, dqm, data, dev_id);

  parameter [31:0] MEM_WIDTH = 16, MEM_SIZE = 8; // in mbytes

  input [10:0] adr;
  input      ba;
  input      rasx, casx, csx, wex, cke, clk;
  input [ 7:0] dqm;
  inout [63:0] data;
  input [ 4:0] dev_id;

  genvar      i;
  generate
    case ({MEM_SIZE, MEM_WIDTH})
      {32'd8, 32'd16}: // 8Meg x 16 bits wide.
        begin
          for (i=0; i<4; i=i+1) begin:word
            sms_16b216t0 p(.clk(clk), .csb(csx), .cke(cke), .ba(ba),
                          .addr(adr[10:0]), .rasb(rasx), .casb(casx),
                          .web(wex), .udqm(dqm[2*i+1]), .ldqm(dqm[2*i]),
                          .dqi(data[15+16*i:16*i]), .dev_id(dev_id[4:0]));
            // The generated instance names are word[3].p, word[2].p,
            // word[1].p, word[0].p, and the task read_mem
          end
          task read_mem;
            input [31:0] address;
            output [63:0] data;
            begin // call read_mem in sms module
              word[3].p.read_mem(address, data[63:48]);
              word[2].p.read_mem(address, data[47:32]);
              word[1].p.read_mem(address, data[31:16]);
              word[0].p.read_mem(address, data[15:0]);
            end
          endtask
        end
      {32'd16, 32'd8}: // 16Meg x 8 bits wide.
        begin
          for (i=0; i<8; i=i+1) begin:byte
            sms_16b208t0 p(.clk(clk), .csb(csx), .cke(cke), .ba(ba),
                          .addr(adr[10:0]), .rasb(rasx), .casb(casx),
                          .web(wex), .dqm(dqm[i]),
                          .dqi(data[7+8*i:8*i]), .dev_id(dev_id[4:0]));
            // The generated instance names are byte[7].p, byte[6].p,
            // byte[5].p, byte[4].p, byte[3].p, byte[2].p, byte[1].p,
            // byte[0].p and the task read_mem
          end
          task read_mem;
            input [31:0] address;
            output [63:0] data;
            begin // call read_mem in sms module
              byte[7].p.read_mem(address, data[63:56]);
              byte[6].p.read_mem(address, data[55:48]);
              byte[5].p.read_mem(address, data[47:40]);
              byte[4].p.read_mem(address, data[39:32]);
              byte[3].p.read_mem(address, data[31:24]);
              byte[2].p.read_mem(address, data[23:16]);
              byte[1].p.read_mem(address, data[15: 8]);
              byte[0].p.read_mem(address, data[ 7: 0]);
            end
          endtask
        end
      // Other memory cases ...
    endcase
  endgenerate
endmodule

```

12.4 Hierarchical names

Every identifier in a Verilog HDL description shall have a unique *hierarchical path name*. The hierarchy of modules and the definition of items such as tasks and named blocks within the modules shall define these names. The hierarchy of names can be viewed as a tree structure, where each module instance, generated instance, task, function, or named `begin-end` or `fork-join` block defines a new hierarchical level, or scope, in a particular branch of the tree.

At the top of the name hierarchy are the names of one or more root modules of which no instances have been created. This root or these parallel root modules make up one or more hierarchies in a *design description* or *description*. Inside any module, each module instance (including an arrayed or generated instance), task definition, function definition, and named `begin-end` or `fork-join` block shall define a new branch of the hierarchy. Named blocks within named blocks and within tasks and functions shall create new branches. Automatic tasks and functions are exceptions, and do not create visible branches that can be referenced (see 10.2.1 and 10.3.1).

Each node in the hierarchical name tree shall be a separate scope with respect to identifiers. A particular identifier can be declared at most once in any scope. See [12.6](#) for a discussion of scope rules and [3.12](#) for a discussion of name spaces.

Any named Verilog object or *hierarchical name reference* can be referenced uniquely in its full form by concatenating the names of the modules, module instance names, tasks, functions, or named blocks that contain it. The period character shall be used to separate each of the names in the hierarchy, except for escaped identifiers embedded in the hierarchical name reference, which are followed by separators composed of white space and a period-character. The complete path name to any object shall start at a top-level (root) module. This path name can be used from any level in the hierarchy or from a parallel hierarchy. The first node name in a path name can also be the top of a hierarchy that starts at the level where the path is being used (which allows and enables downward referencing of items). Objects declared in automatic tasks and functions are exceptions, and cannot be accessed by hierarchical name references.

The syntax for hierarchical path names is given in [Syntax 12-4](#).

```

escaped_identifier ::= (From Annex A - A.9.3)
    \ { Any_ASCII_character_except_white_space } white_space
hierarchical_identifier ::=
    { identifier [ [ constant_expression ] ] . } identifier
identifier ::=
    simple_identifier
    | escaped_identifier
simple_identifiera ::= [ a-zA-Z_ ] { [ a-zA-Z0-9_$ ] }
white_space ::= (From Annex A - A.9.4)
    space | tab | newline | eofb

```

^aA `simple_identifier` shall start with an alpha or underscore (`_`) character, shall have at least one character, and shall not have any spaces.

^bEnd of file.

Syntax 12-4—Syntax for hierarchical path names

Examples:

Example 1—The code in this example defines a hierarchy of module instances and named blocks.

```

module mod (in);
input in;

always @(posedge in) begin : keep
reg hold;
    hold = in;
end
endmodule

module wave;
reg stim1, stim2;

cct a(stim1, stim2); // instantiate cct

initial begin :wave1
    #100 fork :innerwave
        reg hold;
    join
    #150 begin
        stim1 = 0;
    end
end
endmodule

module cct (stim1, stim2);
input stim1, stim2;

// instantiate mod
mod amod(stim1), bmod(stim2);
endmodule

```

[Figure 1](#) illustrates the hierarchy implicit in this Verilog code.

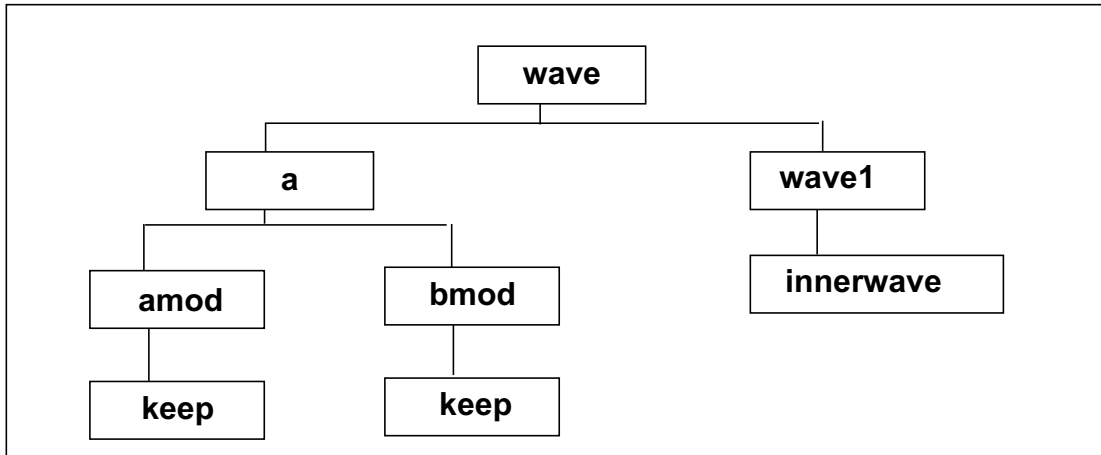


Figure 1—Hierarchy in a model

[Figure 2](#) is a list of the hierarchical forms of the names of all the objects defined in the code.

```

wave
wave.stim1
wave.stim2
wave.a
wave.a.stim1
wave.a.stim2
wave.a.amod
wave.a.amod.in
wave.a.amod.keep
wave.a.amod.keep.hold
wave.a.bmod
wave.a.bmod.in
wave.a.bmod.keep
wave.a.bmod.keep.hold
wave.wave1
wave.wave1.innerwave
wave.wave1.innerwave.hold
  
```

Figure 2—Hierarchical path names in a model

Hierarchical name referencing allows free data access to any object from any level in the hierarchy. If the unique hierarchical path name of an item is known, its value can be sampled or changed from anywhere within the description.

Example 2—The next example shows how a pair of named blocks can refer to items declared within each other.

```

begin
  fork :mod_1
    reg x;
    mod_2.x = 1;
  join
  fork :mod_2
    reg x;
    mod_1.x = 0;
  join
end
  
```

12.5 Upwards name referencing

The name of a module or module instance is sufficient to identify the module and its location in the hierarchy. A lower-level module can reference items in a module above it in the hierarchy. Variables can be referenced if the name of the higher-level module or its instance name is known. For tasks, functions, and named blocks, Verilog shall look in the enclosing module for the name until it is found or until the root of the hierarchy is reached. It shall only search in higher enclosing modules for the name, not instances.

The syntax for an upward reference is given in [Syntax 12-5](#).

```
upward_name_reference ::=
    module_identifier.item_name
item_name ::=
    function_identifier
    | block_identifier
    | net_identifier
    | parameter_identifier
    | port_identifier
    | task_identifier
    | variable_identifier
```

Syntax 12-5—Syntax for upward name referencing

Upwards name references can also be done with names of the form

```
module_instance_name.item_name
```

A name of this form shall be resolved as follows:

- a) Look in the current module for a module instance named `module_instance_name`. If found, this name reference shall be treated as a downward reference, and the item name shall be resolved in the corresponding module.
- b) Look in the parent module for a module instance named `module_instance_name`. If found, the item name shall be resolved from that instance, which is the sibling of the module containing the reference.
- c) Repeat step b), going up the hierarchy.

Example:

In this example, there are four modules, *a*, *b*, *c*, and *d*. Each module contains an integer *i*. The highest-level modules in this segment of a model hierarchy are *a* and *d*. There are two copies of module *b* because module *a* and *d* instantiate *b*. There are four copies of *c*. *i* because each of the two copies of *b* instantiates *c* twice.

```

module a;
integer i;
b a_b1();
endmodule

module b;
integer i;
c b_c1(), b_c2();
initial // downward path references two copies of i:
    #10 b_c1.i = 2; // a.a_b1.b_c1.i, d.d_b1.b_c1.i
endmodule

module c;
integer i;
initial begin // local name references four copies of i:
    i = 1; // a.a_b1.b_c1.i, a.a_b1.b_c2.i,
           // d.d_b1.b_c1.i, d.d_b1.b_c2.i
    b.i = 1; // upward path references two copies of i:
           // a.a_b1.i, d.d_b1.i
    end
endmodule

module d;
integer i;
b d_b1();
initial begin // full path name references each copy of i
    a.i = 1; d.i = 5;
    a.a_b1.i = 2; d.d_b1.i = 6;
    a.a_b1.b_c1.i = 3; d.d_b1.b_c1.i = 7;
    a.a_b1.b_c2.i = 4; d.d_b1.b_c2.i = 8;
end
endmodule

```

12.6 Scope rules

The following four elements define a new scope in Verilog:

- Modules
- Tasks
- Functions
- Named blocks

An identifier shall be used to declare only one item within a scope. This rule means it is illegal to declare two or more variables that have the same name, or to name a task the same as a variable within the same module, or to give a gate instance the same name as the name of the net connected to its output.

If an identifier is referenced directly (without a hierarchical path) within a task, function, or named block, it shall be declared either locally within the task, function, or named block, or within a module, task or named block that is higher in the same branch of the name tree that contains the task, function, or named block. If it is declared locally, then the local item shall be used; if not, the search shall continue upward until an item by that name is found or until a module boundary is encountered. If the item is a variable, it shall stop at a module boundary; if the item is a task, function, or named block it continues to search higher-level modules until found. The search shall cross named block, task, and function boundaries but not module boundaries. This fact means that tasks and functions can use and modify the variables within the containing module by name, without going through their ports.

If an identifier is referenced with a hierarchical name, the path can start with a module name, instance name, task, function, or named block. The names shall be searched first at the current level, then in higher-level modules until found. Since both module names and instance names can be used, precedence is given to instance names if there is a module named the same as an instance name.

Because of the upward searching, path names which are not strictly on a downward path can be used.

Example:

Example 1—In [Figure 3](#), each rectangle represents a local scope. The scope available to upward searching extends outward to all containing rectangles—with the boundary of the module A as the outer limit. Thus block G can directly reference identifiers in F, E, and A; it cannot directly reference identifiers in H, B, C, and D.

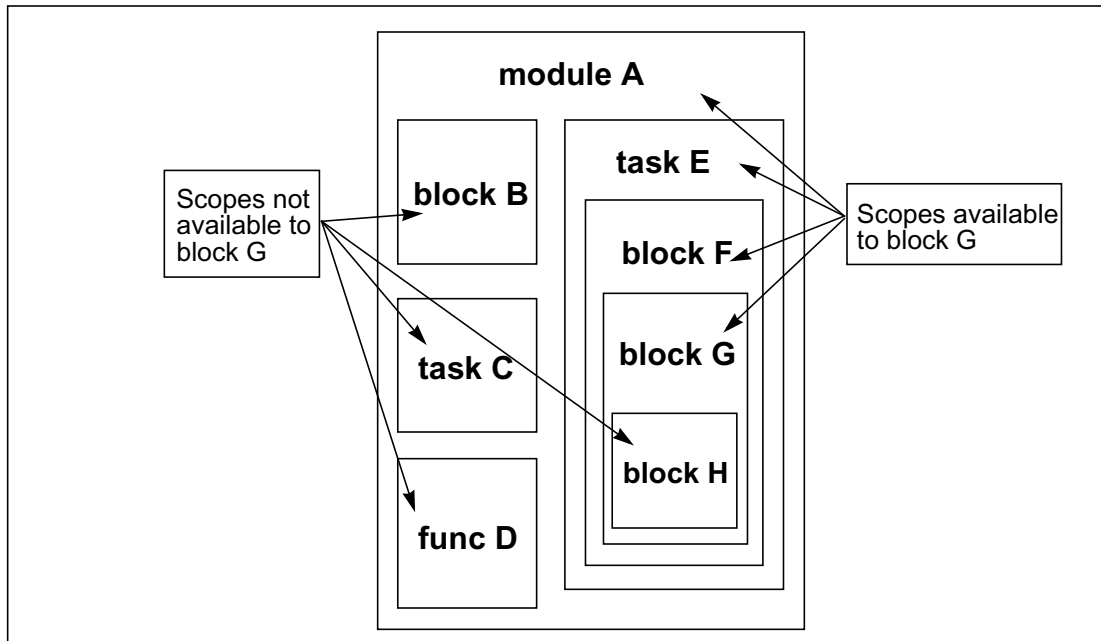


Figure 3—Scopes available to upward name referencing

Example 2—The following example shows an incompletely defined downward reference that can be accessed.

```

task t;
reg r, s;
begin : b
    t.b.r = 0; // poorly defined but found by upward search
    t.s = 0;  // fully defined downward reference
end
endtask

```

13. Configuring the contents of a design

13.1 Introduction

To facilitate both the sharing of Verilog designs between designers and/or design groups, and the repeatability of the exact contents of a given simulation (or other tool) session, the concept of *configurations* is used in the Verilog language. A configuration is simply an explicit set of rules to specify the exact source description to be used to represent each instance in a design. The operation of selecting a source representation for an instance is referred to as *binding* the instance.

The example below shows a simple configuration problem.

Example:

<pre>file top.v module top(); adder a1(...); adder a2(...); endmodule</pre>	<pre>file adder.v module adder(...); // rtl adder // description ... endmodule</pre>	<pre>file adder.vg module adder(...); // gate-level adder // description ... endmodule</pre>
---	--	--

Consider using the `rtl` adder description in `adder.v` for instance `a1` in module `top` and the gate-level adder description in `adder.vg` for instance `a2`. In order to specify this particular set of instance bindings and to avoid having to change the source description to specify a new set, a configuration can be used.

```
config cfg1; // specify rtl adder for top.a1, gate-level adder for top.a2
  design rtlLib.top;
  default liblist rtlLib;
  instance top.a2 liblist gateLib;
endconfig
```

The elements of a *config* are explained in subsequent sections, but this simple example illustrates some important points about *configs*. As evidenced by the **config-endconfig** syntax, the config is a design element, similar to a module, which exists in the Verilog name space. The config contains a set of rules which are applied when searching for a source description to *bind* to a particular instance of the design.

A Verilog design description starts with a top-level module (or modules), which is not instantiated elsewhere in the design. From this module's source description, the instantiated modules (or children) are found, and then the source descriptions for the module definitions of these subinstances shall be located, and so on until every instance in the design is mapped to a source description.