

2.7.2 Generated identifiers [deleted]

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 **declared previously in the scope where the instantiation appears or in any scope whose declarations can be directly referenced from that scope (see 12.7)**, 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 in the scope where the assignment statement appears or in any scope whose declarations can be directly referenced from that scope (see 12.7)**, then an implicit scalar net of default net type shall be assumed. See [6.1.2](#) for a discussion of continuous assignment statements.

Note that the implicit net declaration belongs to the scope in which the net reference appears. For example, if the implicit net is declared by a reference in a generate block, then the net is implicitly declared only in that generate block. Subsequent references to the net from outside the generate block or in another generate block within the same module would either be illegal or would create another implicit declaration of a different net (depending on whether the reference meets the above criteria). See 12.4 for information about generate blocks.

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

Elements of net arrays can be used in the same fashion as a scalar or vector net. They are useful for connecting to ports of module instances inside loop generate constructs (see 12.4).

3.12 Name spaces

In Verilog HDL, there are **several** name spaces; two are global and **the rest** are local. The global name spaces are *definitions* and *text macros*. The *definitions name space* unifies all the **module** (see [12.1](#)), **macro-module** (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.

The local name spaces are: *block*, *module*, *generate block*, *port*, *specify block*, and *attribute*. Once a name is defined within the *block*, *module*, *generate block*, *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, **module instances**, **generate blocks**, parameters, named events, **genvars**, 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 generate block name space is introduced by generate constructs (see 12.4). It unifies the definition of functions, tasks, named blocks, module instances, generate blocks, local parameters, named events, genvars, net type of declaration, and variable type of declaration.

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.

4.3 Minimum, typical, and maximum delay expressions

[In syntax 4-2 in this section, remove "genvar_identifier" from the list of alternatives in the constant_primary production. Do the same in Annex A.8.4].

10.3.5 Use of constant functions (no change)

Constant function calls are used to support the building of complex calculations of values at elaboration time. 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 } local_parameter_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
    | { attribute_instance } loop_generate_construct
    | { attribute_instance } conditional_generate_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
    | generate_region
    | specify_block
    | { 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 **do not appear in any module instantiation statement**, as described in [12.1.2](#). Note that this applies even if the module instantiation appears in a generate block that is not itself instantiated (see 12.4). A model shall contain at least one top-level module.

12.4 Generate constructs

Generate constructs are used to either conditionally or multiply instantiate generate blocks into a model. A generate block is a collection of one or more module items. A generate block may not contain port declarations, parameter declarations, specify blocks and specparam declarations. All other module items, including other generate constructs, are allowed in a generate block. Generate constructs provide the ability for parameter values to affect the structure of the model. They also allow for modules with repetitive structure to be described more concisely, and they make recursive module instantiation possible.

There are two kinds of generate constructs: loops and conditionals. Loop generate constructs allow a single generate block to be instantiated into a model multiple times. Conditional generates, which include if-generate and case-generate constructs, instantiate at most one generate block from a set of alternative generate blocks. The term *generate scheme* refers to the method for determining which or how many generate blocks are instantiated. It includes the conditional expressions, case alternatives and loop control statements that appear in a generate construct.

Generate schemes are evaluated during elaboration of the model. Elaboration occurs after parsing the HDL and before simulation, and involves expanding module instantiations, computing parameter values, resolving hierarchical names (see 12.5), establishing net connectivity and in general preparing the model for simulation. Although generate schemes use syntax that is similar to behavioral statements, it is important to note that they do not execute at simulation time. They are evaluated at elaboration time and the result is determined before simulation begins. Therefore, all expressions in generate schemes must be constant expressions, deterministic at elaboration time. [*For more details on elaboration, see 12.8 - assuming a section on elaboration is added*].

The elaboration of a generate construct results in zero or more instances of a generate block. An instance of a generate block is similar in some ways to an instance of a module. It creates a new level of hierarchy. It brings the objects, behavioral constructs and module instances within the block into existence. These constructs act the same as they would if they were in a module brought into existence with a module instantiation, except that object declarations from the enclosing scope can be referenced directly (see 12.7). Names in instantiated named generate blocks can be referenced hierarchically as described in 12.5.

The keywords `generate` and `endgenerate` may be used in a module to define a *generate region*. A generate region is a textual span in the module description where generate constructs may appear. Use of generate regions is optional. There is no semantic difference in the module when a generate region is used. A parser may choose to take note of the generate region to produce different error messages for misused generate construct keywords. Generate regions do not nest and they may only occur directly within a module. If the generate keyword is used, it must be matched by an endgenerate keyword.

The syntax for generate constructs is given in Syntax 12-3.

```

module_or_generate_item ::=
    { attribute_instance } module_or_generate_item_declaration
    | { attribute_instance } local_parameter_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
    | { attribute_instance } loop_generate_construct
    | { attribute_instance } conditional_generate_construct

generate_region ::=
    generate { module_or_generate_item } endgenerate

genvar_declaration ::=
    genvar list_of_genvar_identifiers ;
list_of_genvar_identifiers ::=
    genvar_identifier { , genvar_identifier }

loop_generate_construct ::=
    for ( genvar_initialization ; genvar_expression ; genvar_iteration )
        generate_block

genvar_initialization ::=
    genvar_identifier = constant_expression

genvar_expression ::=
    genvar_primary
    | unary_operator { attribute_instance } genvar_primary
    | genvar_expression binary_operator { attribute_instance } genvar_expression
    | genvar_expression ? { attribute_instance } genvar_expression : genvar_expression
    | string

genvar_iteration ::=
    genvar_identifier = genvar_expression

genvar_primary ::=
    constant_primary
    | genvar_identifier

conditional_generate_construct ::=
    if_generate_construct
    | case_generate_construct

if_generate_construct ::=
    if ( constant_expression ) generate_block_or_null
    [ else generate_block_or_null ]

case_generate_construct ::=
    case ( constant_expression )
        case_generate_item { case_generate_item } endcase

case_generate_item ::=
    constant_expression { , constant_expression } : generate_block_or_null
    | default [ : ] generate_block_or_null

generate_block ::=
    module_or_generate_item
    | begin [ : generate_block_identifier ] { module_or_generate_item } end

generate_block_or_null ::=
    generate_block ;

```

Syntax 12-3—Syntax for generate constructs

12.4.1 Loop generate construct

A loop generate construct permits a generate block to be instantiated multiple times using syntax that is similar to a for-loop statement. The loop index variable must be declared in a genvar declaration prior to its use in a loop generate scheme.

The genvar is used as an integer during elaboration to evaluate the generate loop and create instances of the generate block, but it does not exist at simulation time. A genvar shall not be referenced anywhere other than in a loop generate scheme.

Both the initialization and iteration assignments in the loop generate scheme shall assign to the same genvar. The initialization assignment shall not reference the loop index variable on the right hand side.

Within the generate block of a loop generate construct, there is an implicit localparam declaration. This is an integer parameter that has the same name and type as the loop index variable, and its value within each instance of the generate block is the value of the index variable at the time the instance was elaborated. This parameter can be used anywhere within the generate block that a normal parameter with an integer value can be used. It can be referenced with a hierarchical name.

Because this implicit localparam has the same name as the genvar, any reference to this name inside the loop generate block will be a reference to the localparam, not to the genvar. As a consequence, it is not possible to have two nested loop generate constructs that use the same genvar.

Generate blocks in loop generate constructs can be named or unnamed, and they can consist of only one item which need not be surrounded by begin/end keywords. Even if the begin/end keywords are absent, it is still a generate block which, like all generate blocks, comprises a separate scope and a new level of hierarchy when it is instantiated.

If the generate block is named, it is a declaration of an array of generate block instances. The index values in this array are the values assumed by the genvar during elaboration. Note that this can be a sparse array since the genvar values do not have to form a contiguous range of integers. The array is considered to be declared even if the loop generate scheme resulted in no instances of the generate block. If the generate block is not named, the declarations within it cannot be referenced using hierarchical names other than from within the hierarchy instantiated by the generate block itself.

It shall be an error if the name of a generate block instance array conflicts with any other declaration, including any other generate block instance array. It shall be an error if the loop generate scheme does not terminate. It shall be an error if a genvar value is repeated during the evaluation of the loop generate scheme. It shall be an error if any bit of the genvar is set to X or Z during the evaluation of the loop generate scheme.

Examples:*Example 1:*

```

for (i=0; i<5; i=i+1) begin:a
  for (i=0; i<5; i=i+1) begin:b
    // error -- using "i" as loop index for
    // two nested generate loops
  end
end
-----

reg a;
for (i=1; i<0; i=i+1) begin: a
  // error -- "a" conflicts with name of reg "a"
end
-----

for (i=1; i<5; i=i+1) begin: a
end

for (i=10; i<15; i=i+1) begin: a
  // error -- "a" conflicts with name of previous
  // loop even though indices are unique
end

```

Example 2—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

```

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;

  // Hierarchical gate 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 nets total)

  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

  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;

  // Hierarchical gate 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
  // Gate instances are connected with nets named:
  //             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

  for(i=0; i<SIZE; i=i+1) begin:bit
    wire  t1, t2, t3;

    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

  assign co = c[SIZE];
endmodule

```

The hierarchical generate block instance names in a multi-level generate loop are shown in example 5. For each block instance created by the generate loop, the generate block identifier for the loop is indexed by adding the "[genvar value]" to the end of the generate block identifier. These generate scope names can be used in hierarchical path names (see [12.5](#)).

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
  end
  if (i>0) begin:B4                      // scope B1[i].B4
    for (m=0; m<SIZE; m=m+1) begin:B5 // scope B1[i].B4.B5[m]
      M4 N4();                          // instantiates B1[i].B4.B5[m].N4
    end
  end
end
endgenerate

// Some examples of hierarchical names for the module instances:
// 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.B5[0].N4                 B1[1].B4.B5[1].N4

```

12.4.2 Conditional generate constructs

The conditional generate constructs, `if-generate` and `case-generate`, select at most one generate block from a set of alternative generate blocks based on constant expressions evaluated during elaboration. The selected generate block, if any, is instantiated into the model.

Generate blocks in conditional generate constructs can be named or unnamed, and they can consist of only one item which need not be surrounded by `begin/end` keywords. Even if the `begin/end` keywords are absent, it is still a generate block which, like all generate blocks, comprises a separate scope and a new level of hierarchy when it is instantiated.

Because at most one of the alternative generate blocks is instantiated, it is permissible for there to be more than one block with the same name within a single conditional generate construct. It is not permissible for any of the named generate blocks to have the same name as generate blocks in any other conditional or loop generate construct in the same scope, even if the blocks with the same name are not selected for instantiation. It is not permissible for any of the named generate blocks to have the same name as any other declaration in the same scope, even if that block is not selected for instantiation.

If the generate block selected for instantiation is named, then this name declares a generate block instance and is the name for the scope it creates. Normal rules for hierarchical naming apply. If the generate block selected for instantiation is not named, it still creates a scope, but the declarations

within it cannot be referenced using hierarchical names other than from within the hierarchy instantiated by the generate block itself.

If a generate block in a conditional generate construct consists of only one item which is itself a conditional generate construct, and if that item is not surrounded by begin/end keywords, then this generate block is not treated as a separate scope. The generate construct within this block is said to be directly nested. The generate blocks of the directly nested construct are treated as if they belong to the outer construct, and therefore can have the same name as the generate blocks of the outer construct. This allows complex conditional generate schemes to be expressed without creating unnecessary levels of generate block hierarchy. The most common use of this would be to create an if-else-if generate scheme with any number of else-if clauses, all of which can have generate blocks with the same name because only one will be selected for instantiation. It is permissible to combine if-generate and case-generate constructs in the same complex generate scheme. Note that direct nesting applies only to conditional generate constructs nested in conditional generate constructs. It does not apply in any way to loop generate constructs.

Example:

```

module test;

    parameter p = 0, q = 0;
    wire a, b, c;

    if (p == 1)
        if (q == 0)
            begin : u1
                and g1 (a, b, c);
            end
        else if (q == 2)
            begin : u1
                or g1 (a, b, c);
            end
        else
            ;
    else if (p == 2)
        case (q)
            0, 1, 2:
                begin : u1
                    xor g1 (a, b, c);
                end
            default:
                begin : u1
                    xnor g1 (a, b, c);
                end
        endcase
    endmodule

```

Depending on the values of parameters *p* and *q*, this generate construct will select at most one of the generate blocks named *u1*. The hierarchical name of the gate instantiation in that block would be *test.u1.x1*. Note that when nesting if-generate constructs there can be ambiguity as to which "if" construct an "else" belongs to. The "else" always belongs to the nearest "if" construct. As in the example above, an "else" with a null generate block can be inserted to make a subsequent "else" belong to an outer "if" construct. Begin/end keywords can also be used to disambiguate, but this would violate the criteria for direct nesting and an extra level of generate block hierarchy would be created.

Conditional generate constructs make it possible for a module to contain an instantiation of itself. The same can be said of loop generate constructs, but it is more easily done with conditional generates. With proper use of parameters, the resulting recursion can be made to terminate, resulting in a legitimate model hierarchy. Note that because of the rules for determining top-level modules, a module containing an instantiation of itself will not be a top-level module.

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)) begin: mult
    CLA_multiplier #(a_width,b_width) u1(a, b, product);
    // instantiate a CLA multiplier
  end
  else begin: mult
    WALLACE_multiplier #(a_width,b_width) u1(a, b, product);
    // instantiate a Wallace-tree multiplier
  end
endgenerate
// The hierarchical instance name is mult.u1

endmodule

```

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

```

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

  case ({MEM_SIZE, MEM_WIDTH})
    {32'd8, 32'd16}: // 8Meg x 16 bits wide.
      begin: memory
        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 hierarchical instance names are memory.word[3].p,
          // memory.word[2].p, memory.word[1].p, memory.word[0].p,
          // and the task memory.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: memory
        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 hierarchical instance names are memory.byte[7].p,
          // memory.byte[6].p, ... , memory.byte[1].p, memory.byte[0].p,
          // and the task memory.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
endmodule

```

12.4.3 External names for unnamed generate blocks

Although an unnamed generate block has no name that can be used in a hierarchical name, it needs to have a name by which external interfaces can refer to it. A name will be assigned for this purpose to each unnamed generate block as follows.

Each generate construct in a given scope is assigned a number. The number will be 1 for the construct that appears textually first in that scope, and will increase by 1 for each subsequent generate construct in that scope. All unnamed generate blocks will be given the name “genblk<n>” where <n> is the number assigned to its enclosing generate construct. If such a name would conflict with an explicitly declared name, then leading zeroes are added in front of the number until the name does not conflict.

Note that each generate construct is assigned its number as described above even if it does not contain any unnamed generate blocks.

Example:

```
module top;

    parameter genblk2 = 0;
    genvar i;

    if (genblk2)
        reg a; // top.genblk1.a
    else
        reg b; // top.genblk1.b

    if (genblk2)
        reg a; // top.genblk02.a
    else
        reg b; // top.genblk02.b

    for (i = 0; i < 1; i = i + 1)
    begin : g1
        if (1)
            reg a; // top.g1[0].genblk1.a
        end

    for (i = 0; i < 1; i = i + 1)
        if (1)
            reg a; // top.genblk4[0].genblk1.a

    if (1)
        reg a; // top.genblk5.a

endmodule
```

12.5 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 block 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.

A design description contains one or more top-level modules (see 12.1.1). Each such module forms the top of a name hierarchy. 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 instance**), **generate block 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). **Unnamed generate blocks are also exceptions. They create branches that are visible only from within the block and within any hierarchy instantiated by the block. See 12.4.3 for a discussion of unnamed generate blocks.**

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 [12.7](#) 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, **generate blocks**, 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. **Objects declared in unnamed generate blocks are also exceptions. They can be referenced by hierarchical names only from within the block and within any hierarchy instantiated by the block.**

Names in a hierarchical path name that refer instance arrays or loop generate blocks may be followed immediately by a constant expression in square brackets. This expression selects a particular instance of the array and is therefore called an *instance select*. The expression shall evaluate to one of the legal index values of the array. If the array name is not the last path element in the hierarchical name, the instance select expression is required.

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.

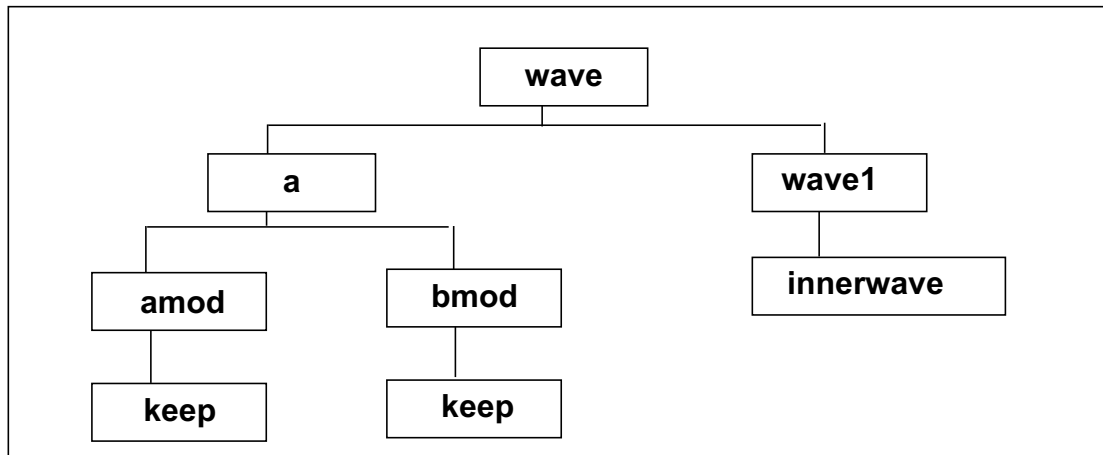


Figure 1—Hierarchy in a model

[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
  
```

12.6 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, **named blocks and generate 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

```
scope_name.item_name
```

where **scope_name** is either a module instance name or a generate block name. A name of this form shall be resolved as follows:

- a) Look in the current scope for a scope named **scope_name**. If not found and the current scope is not the module scope, look for the name in the enclosing scope, repeating as necessary until the name is found or the module scope is reached. If still not found, proceed to b). Otherwise this name reference shall be treated as a downward reference from the scope in which the name is found.
- b) Look in the parent module's outermost scope for a scope named **scope_name**. If found, the item name shall be resolved from that scope.
- c) Repeat step b), going up the hierarchy.

There is an exception to these rules for hierarchical names on the left hand side of defparam statements. See 12.2.1 for details.

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

```

12.7 Scope rules

The following elements define a new scope in Verilog:

- Modules
- Tasks
- Functions
- Named blocks
- **Generate 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. **For generate blocks, this rule applies whether or not the generate block is instantiated. An exception to this is made for generate blocks in a conditional generate construct. See 12.4.3 for a discussion of naming conditional generate blocks.**

If an identifier is referenced directly (without a hierarchical path) within a task, function, **named block or generate block**, it shall be declared either locally within the task, function, **named block or generate block**, or within a module, task, **function, named block or generate block** that is higher in the same branch of the name tree that contains the task, function, **named block or generate 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, **named block or generate block** it continues to search higher-level modules until found. The search shall cross **generate block**, 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, **named block or named generate 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.

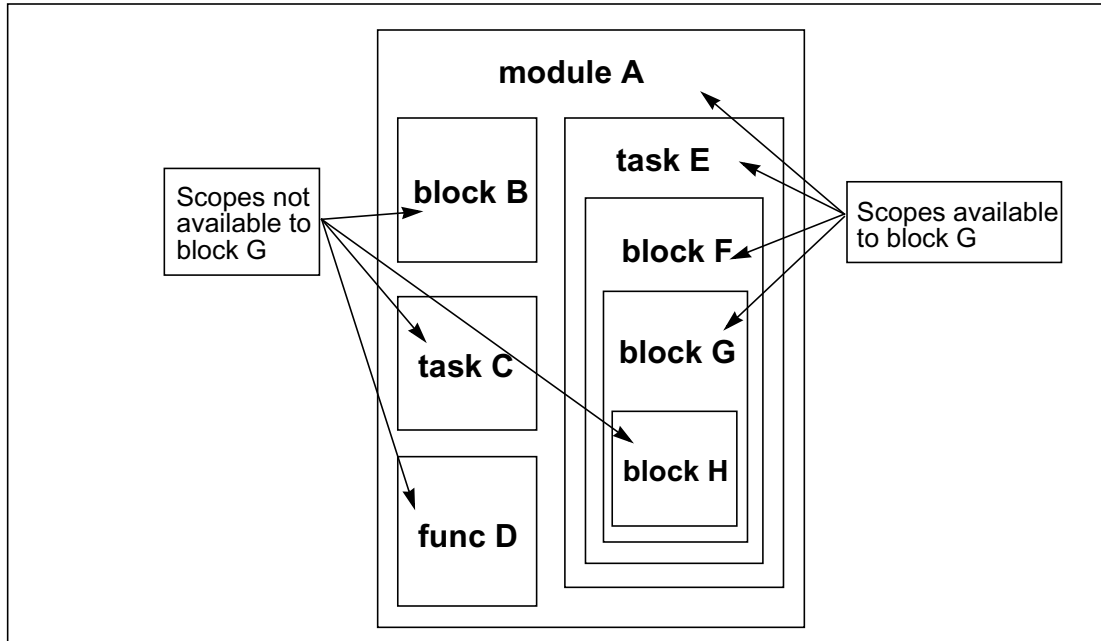


Figure 3—Scopes available to upward name referencing

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) (see 12.1.1). 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.

