

# Draft Proposal for Generate Modifications in 1364-2005

This file contains four main sections:

1. Grammar Modifications

Changes to the BNF, mostly for clarification of generates. Some minor language changes.

2. General Modifications

Changes that need to be made throughout the document: new terminology, clarifications, corrections.

3. New LRM section 12.4

Text for a new LRM section 12.4 to replace the current 12.1.3

4. New LRM section 12.8

Text for a new LRM section dealing with how generates are elaborated.

## 1. Grammar Modifications

The grammar for generate statements is the basis of the understanding of what they are. I believe that much of the difficulty we are having in coming up with language to convey this understanding is due to the way the grammar was constructed. By slightly altering this structure and the terms used in the BNF, I believe we can create a stronger basis from which a description of generates will flow much more naturally. This grammar describes the same language as was described by the original grammar, with only slight intentional modifications: the generate / endgenerate keywords are now optional, generate blocks can no longer exist on their own without an *if*, *case* or *for* construct, for-generate blocks no longer require an explicit name, and localparams can be declared inside generate blocks. Some justification and explanation for my changes follows:

The use of the word "statement" in the BNF for the generate constructs is misleading. Until now, that word was used only for behavioral statements. We should keep it that way. Generate constructs look like their behavioral counterparts, but we should avoid confusing the two. A better word is "construct", which is currently used for always and initial blocks.

Although generate constructs are very much a new concept in Verilog, we should try to demystify them as much as we can. To this end, we should treat them grammatically as peers of the other items that can appear at the same places in a module. For this reason, I have put them into the module\_or\_generate\_item production alongside always blocks, initial blocks, module instantiations, etc. By doing this, the requirement that generate/endgenerate keywords surround all generate constructs is removed, but the language is not changed in any other way, and it is much clearer from the grammar where generate constructs can appear.

A single BNF production, called generate\_block, is the thing which is brought into existence by any of the generate constructs. This gives us one specific term, "generate block", to describe that which is generated. Generate blocks may or may not be named, and they may even consist of only one item that is not surrounded by begin/end keywords. But still calling it a generate block should simplify the wording of our descriptions tremendously. It also has the advantage of conveying the sense that it constitutes a separate scope, which we have proposed as a necessary interpretation.

I have grouped the if-generate and case-generate constructs together under the heading of "conditional generate". These two types of generate both select at most one generate block from a choice of blocks, and both check a condition to determine which block to select. Also, grouping them under one heading will make writing the descriptive paragraphs easier.

The original grammar required generate/endgenerate keywords to surround all generate constructs, but it also allowed for any other module item to appear inside them. We have discussed making these keywords optional since they have no real semantic purpose. My grammar uses the term "generate\_region" instead of "generated\_instance" for this. The word "instance" has other implications that do not quite apply here. By using the word "region", I hope to make it easier to explain the minor role that these keywords serve. Also, unlike the original grammar, my grammar makes it clear that any module or generate item can appear inside them.

I have removed the ability to have a generate block exist on its own without any conditional or looping construct around it. It has been speculated that this was an unintended artifact of the original grammar and it serves no real purpose.

I have changed the production name "genvar\_case\_item" to "case\_generate\_item". It has nothing to do with genvars. I suspect this was a typo in the original grammar.

I have removed "genvar\_identifier" from the constant\_primary production. In this proposal, the expressions within the control structure of a for-generate loop, which include genvar identifiers, are covered by a new genvar\_primary production. References to the genvar in constant expressions inside the loop generate block are actually references to an implicit localparam within the block. See my Sect. 12.4 for details.

Note: These BNF productions need to be changed wherever they appear in the LRM.

```
non_port_module_item ::=
    module_or_generate_item
    | specify_block
    | generate_region
    | { attribute_instance } parameter_declaration ;
    | { attribute_instance } specparam_declaration
```

```
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
```

```
generate_region ::=
    generate { module_or_generate_item } endgenerate
```

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

```
genvar_initialization ::=
```

```

    genvar_identifier = constant_expression

genvar_iteration ::=
    genvar_identifier = genvar_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_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_or_null ::=
    generate_block | ;

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

constant_primary ::=
    constant_concatenation
    | constant_function_call
    | ( constant_mintypmax_expression )
    | constant_multiple_concatenation
    | number
    | parameter_identifier
    | specparam_identifier

```

## 2. General Modifications

Delete Sect. 2.7.2 "Generated Identifiers".

In Sect. 3.5, in the second of the three rules, change “**explicitly declared previously in one of the declaration statements of the instantiating module**” to “**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)**”

In Sect 3.5, in the third of the three rules, change “**declared previously**” to “**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)**”.

Add the following paragraph to the end of Sect. 3.5

**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 is 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 in its own scope (depending on whether the reference meets the above criteria). See Sect. 12.4 for information about generate blocks.**

Change Sect. 3.10.1 to:

**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.**

In Sect. 3.12:

In the third paragraph, change “**There are five local name spaces:**” to “**The local name spaces are:**”.

Also in the third paragraph, change the two occurrences of “block, module, port, specify block,” to “block, module, port, generate block, specify block,”

In the “module name space” paragraph, add “genvars” to the list of declarations allowed in the name space after “named events”, and change “instance names” to “module instances, generate blocks”.

Add a new paragraph after the “module name space” paragraph as follows:

**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, localparameters, named events, genvars, net type of declaration, and variable type of declaration.**

In Sect. 10.3.5, remove "They shall not be declared inside a generate scope".

In Sect. 12.1.1, change "are not instantiated" to "do not appear in any module instantiation statement". Add the sentence, "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".**

Delete Sect. 12.1.3, rename Sect. 12.4, 12.5 and 12.6 to be 12.5, 12.6 and 12.7 respectively, and insert Sect. 12.4 (Generate constructs) as provided in Part 3 of this document.

In Sect. 12.5 (formerly 12.4, Hierarchical Names):

In the first paragraph, remove the first sentence and add in its place:

**A design description contains one or more top-level modules (see 12.1.1). Each such module forms the top of a name hierarchy.**

First paragraph, change "**generated instance**" to "**generate block instance**".

Second paragraph, remove "**or generated**" from the parenthetical phrase, and add "**, generate block instance**" just after the closing parenthesis.

Second paragraph, add the following sentence to the end after ETF #325 is applied:

**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.**

Fourth paragraph, first sentence, add "**generate blocks**" to the list of the types of names that can be in a hierarchical name, after "**module instance names**".

Fourth paragraph, add the following sentence to the end after ETF #325 is applied:

**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.**

Add new paragraph:

Names in a hierarchical path name that refer to instance arrays or loop generate blocks ~~must~~ 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.

In Sect. 12.6 (formerly 12.5, Upwards name referencing):

First paragraph, change "**For tasks, functions and named blocks,**" to "**For tasks, functions, named blocks and generate blocks**".

Change all occurrences of "**module\_instance\_name**" to "**scope\_name**".

Prior to "**A name of this form shall be resolved as follows**", add "**where scope\_name is either a module instance name or a generate block name.**"

Replace rules a) and b) with:

- 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.**

After rules a), b) and c), insert the following paragraphs:

**“If `scope_name` is an instance select then only the name portion is used to find a match. If the name matches but the instance select value is illegal, it shall be an error.”**

**There is an exception to these rules for hierarchical names on the left hand side of `defparam` statements. See section 12.8 for details.**

In Sect. 12.7 (formerly 12.6, Scope rules):

Change “**The following four elements**” to “**The following elements**”.

Add "**Generate blocks**" to the list of items between the first and second paragraphs.

First paragraph, add the following sentence to the end:

**For generate blocks, this rule applies whether or not it is instantiated.**

Second paragraph, add the following sentence to the end:

**An exception to this is made for generate blocks in a conditional generate construct. See Sect. 12.4.3 for a discussion of naming conditional generate blocks.**

Third paragraph, replace each occurrence of "task, function or named block" with "task, function, named block or generate block". Replace “module, task or named block” with “module, task, function, named block or generate block”. Replace "named block, task, and function boundaries" with "generate block, named block, task and function boundaries".

Fourth paragraph, replace "or named block" with "named block or named generate block".

Add Sect. 12.8 (Elaboration flow) as provided in Part 4 of this document.

In Sect. 13.1, last paragraph, change “**which is not instantiated elsewhere in the design**” to “**which does not appear in any module instantiation statement in the design**”.

### 3. Section 12.4 Generate constructs

## 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 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.

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 the model 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.



end

-----

```
reg a;  
for (i=1; i<0; i=i+1) begin: a // error -- "a" conflicts with the 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 the name of previous loop  
end // name even though indices are unique
```

[Keep the other examples currently in the corresponding section of the LRM, but remove from the code comments and their descriptive paragraphs all references to generated names. Replace these with comments showing the hierarchical names to the items in the generate blocks. Put generate and endgenerate keywords on a separate line and eliminate them altogether in half the examples. Eliminate Example 2, as it is too similar to Example 1.]

## 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 the elaboration. The selected generate block, if any, is instantiated into the model.

Generate blocks in conditional generate constructs may 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 names 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 statement which is itself a conditional generate construct, and if that statement is not surrounded by begin/end key-

words, 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, all of which can have 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 scope would be created.

Conditional generate constructs make it possible for a module to contain an instantiation of itself. The same can be said of instance arrays and loop generates, 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.

[Keep examples 6 and 7, modify them in the same ways that the loop generate examples were modified].

### 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.a

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

## 4. Section 12.8 Elaboration flow

### 12.8 Elaboration flow

Elaboration is the process that occurs between parsing and simulation. It binds modules to module instances, builds the model hierarchy, computes parameter values, resolves hierarchical names, establishes net connectivity and prepares all of this for simulation. This section is concerned with the part of elaboration where the model hierarchy is expanded and parameter values are resolved. Because of the interdependence of instance arrays and generate constructs with defparam statements that have hierarchical parameter names, the result of elaboration can be ambiguous without rules to guide the process.

Instance arrays and generate constructs use parameter values to determine the structure of the model hierarchy. They can bring other module instances into existence depending on parameter values. If such a module instance contains a defparam statement that modifies a parameter above it in the hierarchy, it is possible that the new parameter value would dictate that the module is not to be instantiated. To avoid this kind of paradox, it shall be an error for a defparam statement to modify a parameter that could affect its own instantiation. This restriction is described formally below.

Elaboration of the model hierarchy proceeds in phases. In each phase, the hierarchy is expanded as much as possible without evaluating or descending into instance arrays and generate constructs. The first phase has as its starting points all of the top-level modules. The hierarchy beneath each starting point is expanded. When an instance array or a generate construct is encountered, it is saved on a list, but it is not elaborated and the hierarchy beneath it is not expanded. This list then becomes the list of starting points for the next phase, and this process is repeated until all of the hierarchy is expanded.

By the end of each phase, the values of parameters must be determined. These parameter values must be finalized before proceeding to the next phase because the hierarchy in the next phase can depend on them. At no time shall a defparam statement modify a parameter whose value was determined during a previous phase of elaboration.

Another elaboration task is the resolution of hierarchical names. For most hierarchical names, this can be done after all of the hierarchy is expanded as described in 12.6. This process is not affected by the phased approach to elaboration. However, hierarchical names on the left hand side of defparam statements must be treated specially.

Hierarchical names in defparam statements must be resolved during the first phase of elaboration in which a resolution is found, even if a downward resolution could be found in a later phase of elaboration. If this is not done, and the name does not resolve in a later phase, then the name would resolve upwards to a parameter whose value has already been determined. In other words, the defparam must take effect during the phase of elaboration where its target parameter is evaluated, but it may not be known how the name will resolve until elaboration is finished.

To avert this paradox, hierarchical names in defparams will be resolved in the earliest phase possible and the defparam shall have its effect in that phase. It shall be an error if the rules for hierarchical name resolution would produce a different resolution after the hierarchy is fully elaborated.

Example:

```

module m;
    mid1 n();
endmodule

module mid1;
    parameter p = 2;

    defparam m.n.p = 1;

    generate
        if (p == 1) begin : m
            mid2 n();
        end
    endgenerate
endmodule

module mid2();
    parameter p = 3;
endmodule

```

In this example, the defparam is encountered in the first phase of elaboration, before the if-generate statement is elaborated. At this point, the name m.n.p resolves upwards to the parameter p in the module mid1. The defparam sets this parameter to 1 and so in the second phase of elaboration, the if-generate condition is true and its generate block m is instantiated. This elaborates a hierarchy whereby the name m.n.p would resolve downward to the parameter p in module mid2. This shall be reported as an error by the elaborator. If the condition had not been true, then there would be no downward resolution possible and there would be no error.