

## Changes - 5/5/2005 by Cliff Cummings

Table of Contents - Added entry for syntax 13-7 cellname

13.2.1 - "NOTES" to "Library map file details" (per proposal from Tom Fitzpatrick - Mantis 658)

13.1.1 - removed "macromodule" from the proposal (inclusion of "module" in the 1364-2001 standard assumes both module and macromodule).

13.2.1 - added permission for one-line and block comments in Verilog library map files (per email from Adam Krolnik - 5/3/2005)

13.2.1 - added a restriction with examples at the end of this section to clarify proper usage of Verilog comments within library files (per email from Adam Krolnik - 5/3/2005)

Syntax 13.2 - ~~——|config\_declaration~~

Already deleted from the BNF - forgot to delete it from this syntax box

13.2.4 - New Compiler directives in library map files section (per email from Adam Krolnik - 5/3/2005)

13.2.2 - Update to allow the above compiler directives (includes examples from Adam Krolnik - 5/3/2005)

13.2.1.1 - Added a large example with explanation.

Annex B - required keyword changes

Index - change index entries for config hierarchical config, cell and endconfig

**Adam also mentions two other points that I do not wish to propose changing. I believe there is sufficient capabilities in the current definition of configurations to cover these points:**

3. If parsers can be forgiving on the need for semicolons, can't we as writers of the LRM be somewhat forgiving too? I.e. why can't we keep or suggest optional semicolons?

**I like the semicolons**

5. It would be nice to exclude a file from being included in a wildcarded file\_path specification. The only way I have seen/used is the following:  
library DONT\_USE\_THIS\_FILE design/my/bad/old/design/files/x\_old.v;  
library RTLlib design/...;

**I believe includes and library paths are sufficient to cover this.**

## Table of Contents

<b>WAS:</b> 13.5.3 Using the <b>cell</b> clause .....	215
<b>PROPOSED:</b> 13.5.3 Using the <b>cellname</b> clause .....	215
<b>WAS:</b> 13.5.5 Using a hierarchical <b>config</b> .....	215
<b>PROPOSED:</b> 13.5.5 Using a hierarchical <b>configuration</b> .....	215
<b>WAS:</b> Syntax 13-7—Syntax for <b>cell</b> clause .....	211
<b>PROPOSED:</b> Syntax 13-7—Syntax for <b>cellname</b> clause .....	211

## A.1 Source text

### A.1.1 Library source text

```
library_text ::= { library_descriptions }
library_descriptions ::=
    library_declaration
    | include_statement
    — | config_declaration
library_declaration ::=
    library library_identifier file_path_spec [ { , file_path_spec } ]
    [ -incdir file_path_spec [ { , file_path_spec } ] ];
file_path_spec ::= file_path
include_statement ::= include <file_path_spec>;
```

### A.1.2 Configuration source text

```
config_declaration ::=
    config configuration config_identifier ;
    design_statement
    { config_rule_statement }
    endconfig endconfiguration
design_statement ::= design { [library_identifier.]cell_identifier } ;
config_rule_statement ::=
PROPOSED: (add missing semicolons)
    default_clause liblist_clause ;
    | inst_clause liblist_clause ;
    | inst_clause use_clause ;
    | cell_clause liblist_clause ;
    | cell_clause use_clause ;
default_clause ::= default
inst_clause ::= instance inst_name
inst_name ::= topmodule_identifier { .instance_identifier }
cell_clause ::= cellcellname [ library_identifier.]cell_identifier
liblist_clause ::= liblist [ { library_identifier } ]
use_clause ::= use [ library_identifier.]cell_identifier [configconfiguration]
```

### A.1.3 Module and primitive source text

```
source_text ::= { description }
description ::=
    module_declaration
```

```
| udp_declaration
| config_declaration
module_declaration ::=
  { 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
```

## Annex B

(normative)

### List of keywords

#### WAS:

cell  
config  
endconfig

#### PROPOSED:

cellname  
configuration  
endconfiguration

**Index** *(need to update references)*

**WAS:**

cell 206

multiple 209

**PROPOSED:**

cellname 206

multiple 209

-----

**WAS:**

clause

cell 211

using 215

**PROPOSED:**

clause

cellname 211

using 215

-----

**WAS:**

config 206

**PROPOSED:**

configuration 206

-----

**WAS:**

endconfig 206

**PROPOSED:**

endconfiguration 206

-----

**WAS:**

hierarchical config

using 215

**PROPOSED:**

hierarchical configuration

using 215

-----

## 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.

*PROPOSED ADDITION - New paragraph:*

Configurations were added to the IEEE Std 1364-2001 (Verilog-2001) and included the keywords **config-endconfig** and **cell**, but starting with the IEEE Std 1364-2005, the Verilog-2001 version of these keywords were replaced with **configuration-endconfiguration** and **cellname**. Configuration and library descriptions that use any of the older Verilog-2001 keywords should be updated with the new keywords, but for those designs that cannot be modified, the older Verilog-2001 configuration keywords can still be used by surrounding the necessary Verilog-2001 style configuration and library files with the **`begin\_keywords 1364-2001** and **`end\_keywords** compiler directives as described in 19.11.

The example below shows a simple configuration problem.

*Example:*

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

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 configuration 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 endconfiguration
```

**WAS:** 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 namespace. 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.~~

**PROPOSED:** The elements of a *configuration* are explained in subsequent sections, but this simple example illustrates some important points about *configurations*. As evidenced by the **configuration-endconfiguration** syntax, the configuration is a design element, similar to a module, which exists in the Verilog namespace. The configuration contains a set of rules that 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.

### 13.1.1 Library notation

**WAS:** In order to map a Verilog instance to a source description, the concept of a symbolic *library*, which is simply a logical collection of design elements (such as modules, primitives, or **configs**) can be used.

**PROPOSED:** In order to map a Verilog instance to a source description, the concept of a symbolic *library*, which is simply a logical collection of design elements (such as modules, primitives, or **configurations**) can be used. A library is not a Verilog source file description, but it is a separate file to describe how source description design elements are bound to specified source file descriptions.

These design elements can be referred to as *cells*. The cell name shall be the same as the name of the module/macromodule/primitive/**configuration** being processed. Syntax 13-1 specifies a cell from a given library.

```
library_cell ::=  
  [library_identifier.]cell_identifier[:configuration]
```

Syntax 13-1 Syntax for cell

This notation gives a symbolic method of referring to source descriptions; the method of mapping source descriptions into libraries is shown in greater detail in 13.2.1. The optional **configuration** extension shall be used explicitly to refer to a **configuration** in the case where a **configuration** has the same name as a module/macromodule/primitive.

For the purposes of this example, suppose the files top.v and adder.v, the rtl descriptions, have been mapped into the library rtlLib, and the file adder.vg, the gate-level description of the adder, has been mapped into the library gateLib. The actual mechanism for mapping source descriptions to libraries is detailed in 13.2.

### 13.1.2 Basic configuration elements

**WAS:** The **design** statement in **configuration** cfg1 of the first example of 13.1 specifies the top-level module in the **design** and what source description is to be used. In this example, the rtlLib.top notation indicates the top-level module description shall be taken from rtlLib. Since top.v and adder.v were mapped to this library, the actual description for the module is known to come from top.v.

**PROPOSED:** The **design** statement in **configuration** cfg1 of the first example of 13.1 specifies the top-level module in the **configuration** and what source description is to be used. In this example, the rtlLib.top notation indicates the top-level module description **for this configuration** shall be taken from rtlLib. Since top.v and adder.v were mapped to this library, the actual description for the module is known to come from top.v.

The **default** statement coupled with the **liblist** clause specifies, by default, all subinstances of top (i.e., top.a1 and top.a2) shall be taken from rtlLib, which means the descriptions in top.v and adder.v, which were mapped to this library, shall be used. For a basic design, which can be completely rtl, this can be sufficient to specify completely the binding for the entire design. However, here the top.a2 instance of adder to the gate-level description shall be bound.

The **instance** statement specifies, for the particular instance top.a2, the source description shall be taken from gateLib. The instance statement overrides the default rule for this particular instance. Since adder.vg was mapped to gateLib, this statement dictates the gate-level description in adder.vg be used for instance top.a2.

## 13.2 Libraries

As mentioned in the previous section, a library is a logical collection of cells ~~which~~that are mapped to particular source description files. The symbolic `lib.cell[:configconfiguration]` notation supports the separate compilation of source files by providing a file system-independent name to refer to source descriptions when instances in a design are bound. It also allows multiple tools, which can have different invocation use-models, to share the same configuration.

### 13.2.1 Specifying libraries - the library map file

**PROPOSED** - *Add comment-clarification at the end of this paragraph*

When parsing a source description file (or files), the parser shall first read the library mapping information from a pre-defined file prior to reading any source files. The name of this file and the mechanism for reading it shall be tool-specific, but all compliant tools shall provide a mechanism to specify one or more library mapping files to be used for a particular invocation of the tool. If multiple mapping files are specified, then they shall be read in the order in which they are specified. [Verilog one-line and block comments shall be permitted in library map files.](#)

For the purposes of this discussion, assume the existence of a file named `lib.map` in the current working directory, which is automatically read by the parser prior to parsing any source files specified on the command line. The syntax for declaring a library in the library map file is shown in Syntax 13-2.

```
escaped_hierarchical_identifier* ::= (From Annex A - A.1.1)
library_text ::=
    { library_descriptions }
library_descriptions ::=
    library_declaration
    | include_statement
    | config_declaration
library_declaration ::=
    library library_identifier file_path_spec [ { , file_path_spec } ]
    [ -incdir file_path_spec [ { , file_path_spec } ] ;
file_path_spec ::=
    file_path
include_statement ::=
    include <file_path_spec> ;
```

- ❖ The period in `escaped_hierarchical_identifier` and `escaped_hierarchical_branch` shall be preceded by white\_space, but shall not be followed by white\_space.

*Syntax 13-2 Syntax for declaring library in the library map file*

#### ~~NOTES~~ Library map file details

1 The `file_path` uses file system-specific notation to specify an absolute or relative path to a particular file or set of files. The following shortcuts/wildcards can be used:

- ? single character wildcard (matches any single character)
- \* multiple character wildcard (matches any number of characters in a directory/file name)
- ... hierarchical wildcard (matches any number of hierarchical directories)
- .. specifies the parent directory
- . specifies the directory containing the `lib.map`

Paths ~~which~~that end in `/` shall include all files in the specified directory. Identical to `/*`.

Paths ~~which~~that do not begin with `/` are relative to the directory in which the current `lib.map` file is located.

2 The paths `/*.v` and `*.v` are identical and both specify all files with a `.v` suffix in the current directory.

### PROPOSED:

3 To avoid confusion between file path wild cards and Verilog comments, it shall be illegal to put any type of Verilog comment inside of any library or include statements. Verilog comments shall either precede or follow these library statements.

```
library rtlLib1 /*.v;      // legal comment after use of wild-card file specification
library rtlLib2 /*.v;      /* legal block-comment */
library rtlLib3 /*.v;      // legal comment after file path that is legal for some computers
library rtlLib4 /* illegal Verilog block-comment within library statement */ *.v;
library rtlLib5 // Illegal single line comment - library command completes on the next line
*.v;
```

Any file encountered by the compiler ~~which~~that does not match any library's `file_path` specification shall by default be compiled into a library named `work`.

To perform the library mapping discussed in the example in 13.1, use the following library definitions in the `lib.map` file:

```
library rtlLib *.v;        // matches all files in the current directory with a .v suffix
library gateLib /*.vg;     // matches all files in the current directory with a .vg suffix
```

#### 13.2.1.1 File path resolution

If a file name potentially matches multiple file path specifications, the path specifications shall be resolved in the following order:

- a) File path specifications which end with an explicit filename
- b) File path specifications which end with a wildcarded filename
- c) File path specifications which end with a directory

If a file name matches path specifications in multiple library definitions (after the above resolution rules have been applied), it shall be an error.

Consider the following example that includes files in three different directories and a `lib.map` file in the `testbench (tb)` directory.

```
file: /proj/tb/lib.map
file: /proj/tb/tb.v (module tb ... // testbench module)

file: /proj/rtl/and2.v (module and2 ... // rtl 2-input and-gate)
file: /proj/rtl/and3.v (module and3 ... // rtl 3-input and-gate)
file: /proj/rtl/or2.v (module or2 ... // rtl 2-input or-gate)
file: /proj/rtl/or3.v (module or3 ... // rtl 3-input or-gate)
file: /proj/rtl/inv.v (module inv ... // rtl inverter)
file: /proj/rtl/dff.v (module dff ... // rtl dff model)

file: /proj/gates/or2.vg (module or2 ... // gate-version 2-input or-gate)
file: /proj/gates/or3.vg (module or3 ... // gate-version 3-input or-gate)
```

IF the `lib.map` file for the preceding example contains the following code, this is legal and `myLib` will have the `tb.v` file and all of the `rtl` models except for the `or3.v` model. The `myLib` library will also include the `gates/or3.vg` model.

```
library myLib tb.v, ../rtl/*.v, ../gates/or3.vg // legal library declaration
```

IF the lib.map file for the preceding example contains the following code, this is also legal and myLib will again have the tb.v file, all of the rtl models except for the or3.v model, and the gates/or3.vg model.

```
library myLib tb.v, ../rtl/, ../gates/or3.* // legal library declaration
```

IF the lib.map file for the preceding example contains the following code, this shall be an error because two or3 modules are explicitly called from two different files.

```
library myLib tb.v, ../rtl/or2.v, ../rtl/or3.v, ../gates/or3.vg // ILLEGAL library declaration
```

IF the lib.map file for the preceding example contains the following code, this shall be an error because two or3 modules are referenced using wildcards from two different files.

```
library myLib tb.v, ../rtl/or2.v, ../rtl/or?.v, ../gates/or3.* // ILLEGAL library declaration
```

IF the lib.map file for the preceding example contains the following code, this shall be an error because multiple copies of or2 and or3 modules are being referenced from two different directories.

```
library myLib tb.v, ../rtl/, ../gates/ // ILLEGAL library declaration
```

Using these rules with the library definitions in the lib.map file, all source files encountered by the parser/compiler can be mapped to a unique library. Once the source descriptions have been mapped to libraries, the cells defined therein are available for binding.

NOTE - Tool implementers may find it convenient to provide a command-line argument to explicitly specify the library into which the file being parsed is to be mapped, which shall override any library definitions in the lib.map file. If these libraries do not exist in the lib.map file, they can only be accessed via an explicit [config configuration](#).

If multiple cells with the same name map to the same library, then the LAST cell encountered shall be written to the library. This is to support a separate-compile use-model (see 13.4.3), where it is assumed encountering a cell after it has previously been compiled is intended to be a recompiling of the cell. In the case where multiple modules with the same name are mapped to the same library in a single invocation of the compiler, then a warning message shall be issued.

### 13.2.2 Using multiple library mapping files

In addition to specifying library mapping information, a lib.map file can also include references to other lib.map files. The **include** command is used to insert the entire contents of a library mapping file in another file during parsing. The result is as though the contents of the included mapping file appear in place of the **include** command.

**WAS:** The syntax of a lib.map file is limited to library specifications, include statements, and standard Verilog comment syntax. Syntax 13-3 shows the syntax for the **include** command.

**PROPOSED:** The syntax of a lib.map file is limited to library specifications, include statements, standard Verilog comment syntax and certain Verilog compiler directives as defined in 13.2.4. Syntax 13-3 shows the syntax for the **include** command.

```
include_statement ::= (From Annex A - A.1.1)  
include <file_path_spec> ;
```

Syntax 13-3 Syntax for include command

If the file path specification, whether in an include or library statement, describes a relative path, it shall be relative to the location of the file ~~which~~that contains the file path. Library providers shall include a local library mapping file in addition to the source contents of the library. Individual users can then simply

include the provider's library mapping file in their own map file to gain access to the contents of the provided library.

### 13.2.3 Mapping source files to libraries

For each cell definition encountered during parsing/compiling, the name of the source file being parsed is compared to the file path specifications of the library declarations in all of the library map files being used. The cell is mapped into the library whose file path specification matches the source file name.

**PROPOSED:** *Add this new section*

### 13.2.4 Compiler directives in library map files

Library map files shall permit the use of the following Verilog compiler directives:

- ``define` and ``undef` macro definitions(see 19.3). An example of macro definition usage follows:

```
`define RTLPATH /proj1/rtl
library rtlLib `RTLPATH/*.v;
```

Note - some Verilog tools allow macro definition from a command line switch of the form `+define+RTLPATH="/proj1/rtl"` and these tools could define library paths without the ``define` macro definition shown in the preceding example.  
*(informative note)*

- ``ifdef`, ``else`, ``elsif`, ``endif`, ``ifndef` conditional compilation compiler directives (see 19.4). Conditional compilation could be used to help specify the desired contents of a library map file as shown in the following example:

```
library rtlLib /proj1/rtl/*.v;

`ifdef VER20
    include /path/ip/block/2.0/lib.map;
`else
    include /path/ip/block/1.0/lib.map;
`endif
```

- ``include` compiler directive (see 19.5). Other libraries can be included using the library map file keyword `include`, but ``include` can include other blocks of legal library map file commands, including the compiler directives listed here.
- ``begin_keywords`, ``end_keywords` compiler directives (see 19.11)

## 13.3 Configurations

As mentioned in the introduction of this chapter, a configuration is simply a set of rules to apply when searching for library cells to which to bind instances. The syntax for configurations is shown in 13.3.1.

### 13.3.1 Basic configuration syntax

**PROPOSED** (*additional paragraphs*):

A configuration file is a Verilog source file that is generally used with a library map file to specify all the Verilog modules that are used by simulation or other tools for the current invocation of the tool. As mentioned in 13.2, a library is a logical collection of cells that are mapped to particular source description files. The configuration is used to bind the description files specified by one or more library map files (or the default work-library map file) to the instances of a Verilog description. A properly coded library map

file (or library map files) and Verilog configuration file (or Verilog configuration files) can contain sufficient information to run a simulation without explicitly calling any other Verilog module-files.

Although it is possible to mix configurations and Verilog modules in the same file, it is generally not useful to do so, because a configuration is typically used to bind a source file to a Verilog module and it would be rare to bind a source file with both configuration-bind information and a module source description in the same configuration file.

The configuration syntax is shown in Syntax 13-4.

```
config_declaration ::= (From Annex A -A.1.2)
    config configuration config_identifier ;
    design_statement
    { config_rule_statement }
    endconfig endconfiguration
design_statement ::=
    design { [library_identifier.]cell_identifier } ;
config_rule_statement ::=
PROPOSED: (add missing semicolons)
    default_clause liblist_clause ;
    | inst_clause liblist_clause ;
    | inst_clause use_clause ;
    | cell_clause liblist_clause ;
    | cell_clause use_clause ;
```

Syntax 13-4 Syntax for configuration

### 13.3.1.1 Design statement

**WAS:** The **design** statement names the library and cell of the top-level module or modules in the design hierarchy configured by the **config**. There shall be one and only one design statement, but multiple top-level modules can be listed in the design statement. The cell or cells identified can not be configurations themselves. It is possible the design identified can have the same name as **configs**, however.

**PROPOSED:** The **design** statement names the library and cell of the top-level module or modules in the design hierarchy configured by the **configuration**. **Since configurations can be hierarchically nested (see 13.3.2 and 13.5.5), the "top-level" module or modules in the configuration may not be the top-level modules of the fully elaborated design.** There shall be one and only one design statement, but multiple top-level modules can be listed in the design statement. The cell or cells identified can not be configurations themselves. It is possible the design identified can have the same name as **configurations**, however.

The **design** statement shall appear before any **configconfiguration** rule statements in the **configconfiguration**.

If the library identifier is omitted, then the library **which**that contains the **configconfiguration** shall be used to search for the cell.

### 13.3.1.2 The default clause

The syntax for the **default** clause is specified in Syntax 13-5.

```
default_clause ::= (From Annex A - A.1.2)
    default
```

### Syntax 13-5 Syntax for default clause

The **default** clause selects all instances **which** that do not match a more specific selection clause. The **use** expansion clause (see 13.3.1.6) can not be used with a **default** selection clause. For other expansion clauses, there can not be more than one **default** clause **which** that specifies the expansion clause.

For simple design configurations, it might be sufficient to specify a **default liblist** (see 13.3.1.5).

#### 13.3.1.3 The instance clause

The **instance** clause is used to specify the specific instance to which the expansion clause shall apply. The syntax for the **instance** clause is specified in Syntax 13-6.

```
inst_clause ::= (From Annex A - A.1.2)
             instance inst_name
inst_name ::=
            toplevel_identifier{.instance_identifier}
```

### Syntax 13-6 Syntax for instance clause

The instance name associated with the **instance** clause is a Verilog hierarchical name, starting at the toplevel module of the **config** configuration (i.e., the name of the cell in the **design** statement).

#### 13.3.1.4 The cellname clause

The **cellname** selection clause names the cell to which it applies. The syntax for the **cellname** clause is specified in Syntax 13-7.

```
cell_clause ::= (From Annex A - A.1.2)
             cellname [ library_identifier.]cell_identifier
```

### Syntax 13-7 Syntax for cellname clause

If the optional library name is specified then the selection rule applies to any instance **which** that is bound or is under consideration for being bound to the selected library and cell. It is an error if a library name is included in a **cellname** selection clause and the corresponding expansion clause is a library list expansion clause.

#### 13.3.1.5 The liblist clause

The **liblist** clause defines an ordered set of libraries to be searched to find the current instance. The syntax for the **liblist** clause is specified in Syntax 13-8.

```
liblist_clause ::= (From Annex A - A.1.2)
                liblist [{library_identifier}]
```

### Syntax 13-8 Syntax for liblist clause

*liblists* are inherited hierarchically downward as instances are bound. When searching for a cell to bind to the current unbound instance, and in the absence of an applicable binding expansion clause, the specified library list is searched in the specified order.

The current library list is selected by the selection clauses. If no library list clause is selected, or the selected library list is empty, then the library list contains the single name which is the library in which the cell containing the unbound instance is found (i.e., the parent cell's library).

### 13.3.1.6 The use clause

The **use** clause specifies a specific binding for the selected cell. The syntax for the **use** clause is specified in Syntax 13-9.

```
use_clause ::= (From Annex A - A.1.2)
use [library_identifier.]cell_identifier[:configconfiguration]
```

#### Syntax 13-9 Syntax for use clause

A **use** clause can only be used in conjunction with an **instance** or **cellname** selection clause. It specifies the exact library and cell to which a selected cell or instance is bound.

The **use** clause has no effect on the current value of the library list. It can be common in practice to specify multiple **configconfiguration** rule statements, one of which specifies a binding and the other of which specifies a library list.

If the lib.cell being referred to by the **use** clause is a **configconfiguration** ~~which~~that has the same name as a module/macromodule/primitive in the same library, then the optional **configconfiguration** suffix can be added to the lib.cell to specify the **configconfiguration** explicitly.

If the library name is omitted, the library shall be inherited from the parent cell.

NOTE—The binding statement can create situations where the unbound instance's module name and the cell name to which it is bound are different.

### 13.3.2 Hierarchical configurations

For situations where it is desirable to specify a special set of configuration rules for a subsection of a design, it is possible to bind a particular instance directly to a configuration using the binding clause:

```
instance top.a1.foo use lib1.foo:configconfiguration;
// bind to the configconfiguration foo in library lib1
```

specifies the instance top.a1.foo is to be replaced with the design hierarchy specified by the configuration lib1.foo:**configconfiguration**. The **design** statement in lib1.foo:**configconfiguration** shall specify the actual binding for the instance top.a1.foo, and the rules specified in the **configconfiguration** shall determine the configuration of all other subinstances under top.a1.foo.

It shall be an error for an instance clause to specify a hierarchical path to an instance ~~which~~that occurs within a hierarchy specified by another **configconfiguration**.

```
configconfiguration bot;
design lib1.bot;
default liblist lib1 lib2;
instance bot.a1 liblist lib3;
endconfig endconfiguration

configconfiguration top;
design lib1.top;
default liblist lib2 lib1;
instance top.bot use lib1.bot:configconfiguration;
instance top.bot.a1 liblist lib4;
// ERROR - can't set liblist for top.bot.a1 from this configconfiguration
endconfig endconfiguration
```

## 13.4 Using libraries and `configurations`

The following subclause describes potential use-models for referencing `configurations` on the command line. It is included for clarification purposes.

The traditional Verilog simulation use-model takes a file-based approach, where the source descriptions for all cells in the design are specified on the command line for each invocation of the tool. With the advent of compiled-code simulators, the configuration mechanism shall also support a use-model `whichthat` allows for the source files to be pre-compiled and then for the pre-compiled design objects to be referenced on the command line. This subclause shall explain how configurations can be used in both of these scenarios.

### 13.4.1 Precompiling in a single-pass use-model

The single-pass use-model is the traditional use-model with which most users are familiar. In this use-model, all of the source description files shall be provided to the simulator via the command line, and only these source descriptions can be used to bind cell instances in the current design. A precompiling strategy in this scenario actually parses every cell description provided on the command line and maps it into the library without regard to whether the cell actually is used in the design. The tool can optionally check to see if the cell already exists in the library, and if it is up-to-date (i.e. the source description has not changed since the last time the cell was compiled) the tool can skip recompiling the cell. After all cells on the command line have been compiled, then the tool can locate the top-level cell (discussed in Section 12), and proceed down the hierarchy, binding each instance as it is encountered in the hierarchy.

NOTE With this use-model it is not necessary for library objects to persist from one tool invocation to another (although for performance considerations it is recommended they do).

### 13.4.2 Elaboration-time compiling in a single-pass use-model

An alternate strategy `whichthat` can be used with a single-pass tool is to parse the source files only to find the top-level module(s), without actually compiling anything into the library during this scanning process. Once the top-level module(s) has been found, then it can be compiled into the library, and the tool can proceed down the hierarchy, only compiling the source descriptions necessary to bind the design successfully. Based on the binding rules in place, only the source files `whichthat` match the current library specification need to be parsed to find the current cell's source description to compile. As with the precompiled single-pass usemodel, it is not necessary for library cells to persist from one invocation to another using this strategy.

### 13.4.3 Precompiling using a separate compilation tool

When using a separate compilation tool, it is essential library cells persist, and the compiled forms shall therefore exist somewhere in the file system. The exact format and location for holding these compiled forms shall be vendor/tool-specific. Using this separate compiler strategy, the source descriptions shall be parsed and compiled into the library using one or more invocations of the compiler tool. The only restriction is all cells in a design shall be precompiled prior to binding the design (typically via an invocation of a separate tool). Using this strategy, the tool `whichthat` actually does the binding only needs to be told the top-level module(s) of the design to be bound, and then it shall use the precompiled form of the cell description(s) from the library to determine the subinstances and descend hierarchically down the design binding each cell as it is located.

### 13.4.4 Command line considerations

In each of the three preceding strategies, the binding rules can either be specified via a `configuration`, or the default rules (from the library map file) can be used. In the single-pass use-models, the `configuration` can be specified by including its source description file on the command line. In the case where the `configuration` includes a design statement, then the specified cell shall be the top-level

module, regardless of the presence of any uninstantiated cells in the rest of the source files. When using a separate compilation tool, the tool **whichthat** actually does the binding only needs to be given the *lib.cell* specification for the top-level cell(s) and/or the **configconfiguration** to be used. In this strategy, the **configconfiguration** itself shall also be precompiled.

### 13.5 Configuration examples

Consider the following set of source descriptions:

*Example:*

<pre>file top.v <b>module</b> top(...); ... adder a1(...); adder a2(...); <b>endmodule</b> <b>module</b> foo(...); ... // rtl <b>endmodule</b></pre>	<pre>file adder.v <b>module</b> adder(...); ... // rtl foo f1(...); foo f2(...); <b>endmodule</b> <b>module</b> foo(...); ... // rtl <b>endmodule</b></pre>	<pre>file adder.vg <b>module</b> adder(...); ... // gate-level foo f1(...); foo f2(...); <b>endmodule</b> <b>module</b> foo(...); ... // gate-level <b>endmodule</b></pre>	<pre>file lib.map <b>library</b> rtlLib top.v; <b>library</b> aLib adder.*; <b>library</b> gateLib adder.vg;</pre>
--	---	--	--

All of the examples in this section shall assume the top.v, adder.v and adder.vg files get compiled with the given lib.map file. This yields the following library structure:

```
rtlLib.top // from top.v
rtlLib.foo // from top.v
aLib.adder // from adder.v
aLib.foo // rtl from adder.v
gateLib.adder // from adder.vg
gateLib.foo // from adder.vg
```

#### 13.5.1 Default configuration from library map file

With no configuration, the libraries are searched according to the library declaration order in the library map file. This means all instances of module adder shall use aLib.adder (since aLib is the first library specified **whichthat** contains a cell named adder), and all instances of module foo shall use rtlLib.foo (since rtlLib is the first library **whichthat** contains foo).

#### 13.5.2 Using the default clause

To always use the foo definition from file adder.v, use the following simple configuration:

```
configconfiguration cfg1;
WAS: design rtlLib.top
PROPOSED: (add missing semicolon)
design rtlLib.top;
default liblist aLib rtlLib;
endconfig endconfiguration
```

The **default liblist** statement overrides the library search order in the lib.map file, so aLib is always searched before rtlLib. Since the gateLib library is not included in the liblist, the gate-level descriptions of adder and foo shall not be used.

To use the gate-level representations of adder and foo, add to the **configconfiguration** as follows:

```
configconfiguration cfg2;
```

**WAS:** `design rtlLib.top`  
**PROPOSED:** *(add missing semicolon)*  
`design rtlLib.top;`  
`default liblist gateLib aLib rtlLib;`  
`endconfig endconfiguration`

This shall cause the gate representation always to be taken before the rtl representation, using the module definitions for adder and foo from adder.vg. The rtl view of top shall be taken since there is no gate representation available.

### 13.5.3 Using the `cellname` clause

To modify the `configconfiguration` to use the rtl view of adder and the gate-level representation of foo from gateLib:

`configconfiguration cfg3;`  
**WAS:** `design rtlLib.top`  
**PROPOSED:** *(add missing semicolon)*  
`design rtlLib.top;`  
`default liblist aLib rtlLib;`  
`cellname foo use gateLib.foo;`  
`endconfig endconfiguration`

The `cellname` clause selects all cells named foo and explicitly binds them to the gate representation in gateLib.

### 13.5.4 Using the instance clause

To modify the `configconfiguration` so the top.a1 adder (and its descendants) use the gate representation and the top.a2 adder (and its descendants) use the rtl representation from aLib:

`configconfiguration cfg4`  
**WAS:** `design rtlLib.top`  
**PROPOSED:** *(add missing semicolon)*  
`design rtlLib.top;`  
`default liblist gateLib rtlLib;`  
`instance top.a2 liblist aLib;`  
`endconfig endconfiguration`

Since the `liblist` is inherited, all of the descendants of top.a2 inherit its `liblist` from the instance selection clause.

### 13.5.5 Using a hierarchical `configconfiguration`

Now suppose all this work has only been on the adder module by itself and a `configconfiguration` which that uses the rtlLib.foo cell for f1, and the gateLib.foo cell for f2 has already been developed. Then use:

`configconfiguration cfg5;`  
`design aLib.adder;`  
`default liblist gateLib aLib;`  
`instance adder.f1 liblist rtlLib;`  
`endconfig endconfiguration`

To use this configuration cfg5 for the top.a2 instance of adder and take the full default aLib adder for the top.a1 instance, use the following `configconfiguration`:

```

config configuration cfg6;
  design rtlLib.top;
  default liblist aLib rtlLib;
WAS: instance top.a2 use work.cfg5:config configuration
PROPOSED: (add missing semicolon)
  instance top.a2 use work.cfg5:config configuration;
endconfig endconfiguration

```

**WAS:** The binding clause specifies the `work.cfg5:config configuration` is to be used to resolve the bindings of instance `top.a2` and its descendants. It is the design statement in `config cfg5` ~~which~~that defines the exact binding for the `top.a2` instance itself. The rest of `cfg5` defines the rules to bind the descendants of `top.a2`. Notice the instance clause in `cfg5` is relative to its own top-level module, `adder`.

**PROPOSED:** The binding clause specifies that the configuration `work.cfg5:configuration` is to be used to resolve the bindings of instance `top.a2` and its descendants. It is the design statement in `configuration cfg5` that defines the exact binding for the `top.a2` instance itself. The rest of `cfg5` defines the rules to bind the descendants of `top.a2`. Notice the instance clause in `cfg5` is relative to its own top-level module, `adder`.

### 13.6 Displaying library binding information

It shall be possible to display the actual library binding information for module instances during simulation. The format specifier `%l` or `%L` shall print out the library.cell binding information for the module instance containing the display (or other textual output) command. This is similar to the `%m` format specifier ~~which~~that prints out the hierarchical path name of the module containing it.

It shall also be able to use the VPI interface to display the binding information. The following new `vpiProperties` shall exist for objects of type `vpiModule`:

- `vpiUseBinding` - the library.cell binding information for a module instance
- `vpiLibrary` - the library name into which the module was compiled
- `vpiCell` - the name of the cell bound to the module instance
- `vpiConfig` - the library.cell name of the `config configuration` controlling the binding of the module instance

These properties shall be of string type, similar to the `vpiName` and `vpiFullName` properties.

### 13.7 Library mapping examples

In the absence of a configuration, it is possible to perform basic control of the library searching order when binding a design.

When a `config configuration` is used, the `config configuration` overrides the rules specified here.

#### 13.7.1 Using the command line to control library searching

In the absence of a configuration, it shall be necessary for all compliant tools to provide a mechanism of specifying a library search order on the command line ~~which~~that overrides the default order from the library mapping file. This mechanism shall include specification of library names only, with the definitions of these libraries to be taken from the library mapping file.

NOTE It is recommended all compliant tools use `"-L <library_name>"` to specify this search order.

#### 13.7.2 File path specification examples

*Example:*

Given the following set of files:

```
/proj/lib1/rtl/a.v
/proj/lib2/gates/a.v
/proj/lib1/rtl/b.v
/proj/lib2/gates/b.v
```

From the /proj library, the following absolute file\_path\_specs are resolved as shown:

```
/proj/lib*/*/a.v =/proj/lib1/rtl/a.v, /proj/lib2/gates/a.v
../a.v =/proj/lib1/rtl/a.v, /proj/lib2/gates/a.v
/proj/.../b.v =/proj/lib1/rtl/b.v, /proj/lib2/gates/b.v
.../rtl/*.v =/proj/lib1/rtl/a.v, /proj/lib1/rtl/b.v
```

From the /proj/lib1 directory, the following relative file\_path\_specs are resolved as shown:

```
../lib2/gates/*.v = /proj/lib2/gates/a.v, /proj/lib2/gates/b.v
./rtl/?v = /proj/lib1/rtl/a.v, /proj/lib1/rtl/b.v
./rtl/ = /proj/lib1/rtl/a.v, /proj/lib1/rtl/b.v
```

### 13.7.3 Resolving multiple path specifications

*Example:*

```
library lib1 "/proj/lib1/foo*.v";
library lib2 "/proj/lib1/foo.v";
library lib3 "../lib1/";
library lib4 "/proj/lib1/*ver.v";
```

When evaluated from the directory /proj/tb directory, the following source files shall map into the specified library:

```
../lib1/foobar.v - lib1 // potentially matches lib1 and lib3. Since lib1 includes a
                        filename and lib3 only specifies a directory; lib1 takes precedence
/proj/lib1/foo.v - lib2 // takes precedence over lib1 and lib3 path specifications
/proj/lib1/bar.v - lib3
/proj/lib1/barver.v - lib4 // takes precedence over lib3 path specification
/proj/lib1/foover.v - ERROR // matches lib1 and lib4
/test/tb/tb.v - work // does not match any library specifications.
```