Hi Stu,

Thank you for the careful thought and consideration you put into the review of the updated proposal. Hopefully we can resolve most of these issues by providing a corrected version of the proposal.

Please see my comments below.

Regards
David

-----Original Message-----
From: Stuart Sutherland [mailto:stuart@sutherland-hdl.com]
Sent: Tuesday, November 04, 2003 12:23 PM
To: David W. Smith
Subject: Packages and Separate compilation proposal
David,

From both the editor's point of view and as a SystemVerilog user, I would like to raise a number of comments and suggestions regarding to the "Packages_Sep_V8 .pdf" proposed change document distributed.

First, the document clearly defines the changes that need to be made to the LRM. Thank you! (editor's point of view).

Second, way back in the early days of defining SV 3.0, Stefen Boyd and I are the ones who most vehemently objected to the way $root was defined in SUPERLOG and being carried over to SV 3.0 (I added Stefen to this e-mail). Stefen wanted $root thrown out completely. I wanted global definitions and functions, but did not want to allow globals to spread anywhere within a single source file (e.g. at the beginning, end, and between module/interface declarations in the same file) or to be spread across any number of files. Stefen and I both lost. I heartily welcome this effort to "fix" the problems of SV 3.0's $root. Thank you, again!

I have a number of concerns and questions regarding the change document ...

1. I have been following the discussion threads on the use of the terms "root", "local root", "compilation unit", $root, and $unit, "packages". "scope" "namespace", and "name space". I agree that these terms are not being consistently used. I assume it is because there were many individual contributions to the text of the document, spread over a considerable amount of time. Before this change document is finalized, I would like to suggest:

a) Decide if you want to call these pseudo-global spaces "local root" or "compilation unit", but don't use both terms. Then use a $name that matches your decision.

b) Add near the beginning of section 18 of the LRM an introduction these terms that provides a simplified definition and gives cross references to the sections where the terms

are discussed more fully. These definitions will allow terms to be referenced before being fully defined, without having readers assume something from a term that is contrary to the intent. This addition could be part of 18.1 (informative text) or a new section 18.2.

c) After defining these terms, do a thorough review of the entire change document to make sure the terms are used consistently and correctly.

DWS: I agree that these terms need to be rationalized.

- The compilation unit refers to the collection of source files that get compiled together.
- The local root (suggest renaming to compilation-unit scope) refers to the scope containing declarations defined within a compilation unit.
- The $unit is used to access the declarations within the compilation-unit scope.

I think that, if we use these terms consistently, they make sense as separate items. I suggest reordering the sections and combining Sections 18.2 and 18.3 to be a single item. Most of the material in the current Section 18.2 is comparing modules and $root. This is no longer needed. The following is suggested. Replace Section 18.2 with:

18.2 is packages
18.3 is Compilation unit Support

2. The change to made in Section 10.6 says "To access functions defined in any other scope, function in the local root are not accessible". This sentence does not make sense. I cannot even figure out what was trying to be said, in order make suggestions on how to reword it.

DWS: I agree and recommend that it be changed to:

> To access functions defined in any other scope~~, functions in the local root are not accessible including `$root`,~~ the foreign code shall have to change DPIcontext appropriately.

The local root top level

3. The changes within section 18.2 leave the bullet item "- can be distributed across any number of files". I am greatly disappointed to see that this dangerous characteristic of the original $root is not be fixed. With the addition of packages, I now side with Stefen, and say kill the concept of a root space completely.

DWS: I agree that the current wording is confusing. I recommend that these comparisons be removed. The remaining definition of compilation-unit scope (as defined above) is far more restrictive than what $root was. All instantiation and statements have been removed. The remaining ability to declare items are a convenience for defining items local to the compilation unit that cannot be accessed anywhere else and for convenience for using identifiers in a package in the declaration of top-level modules, packages, etc… If you do not wish to use the compilation-unit scope then you can explicitely access items using the scope resolution operator.

4. Starting with the changes in section 18.2, the terms "namespace" and "name space" are used frequently. Are these intentionally different? If so, what is the difference? If not,

in which way should these spaces be referred?

DWS: There is no difference intended. Unfortunately this confusion has already shown up in other changes in the LRM and needs to be fixed during the edit/review cycle. I suggest that each place in the proposal where namespace occurs it get changed to “name space” to be consistent with 1364.

5. Section 18.2 makes references to "name space local to the compilation unit", "global namespace", "local name space" and "module instance name space". Many sections later, in Section 18.9, there are definitions of name spaces, but none of these spaces are defined.

DWS: Clearly there is a consistency issue here. The references are not specific in that they refer to “a global name space” and “a local name space”. I suggest rewriting the whole section since its comparison to a module is no longer appropriate. I have included the following definitions at the beginning of Section 18.3 “Compilation unit Support”. Here are the suggested definitions:

- compilation unit: a collection of one or more SystemVerilog source files compiled together
- compilation-unit scope: a scope that is local to the compilation unit. It contains all declarations that lie outside of any other scope
- $unit: the name used to explicitly access the identifiers in the compilation-unit

6. Section 18.2, under the list of items in a local root, there is the bullet "- can contain one timeunit and timeprecision directive within the unit of compilation". There are two problems with this statement:

a) It conflicts with the first paragraph in section 18.6 (as shown in this same change document), which states that a local unit is a time scope, and that the timeunit and timeprecision declarations can be repeated within a time scope.

b) The wording of this bullet, at least the way I read it, means that an entire unit of compilation can only have a single timeunit and timeprecision declaration. This means I cannot compile two modules at the same time that have different time units or precision declared within the modules.

DWS: I agree that there is an inconsistency. This is taken care of in the rewrite by removing the discussion. The definition of how timeunit_declarations are left to 18.6.

6. Section 18.2, under the list of things that can be contained in a module, the last bullet starting with "shall not contain procedural statements that not within..." is no longer needed. This bullet distinguished modules from the SV 3.0 $root that could contain procedural statements outside of procedural blocks. The changes in this document remove that ability from the root space (thank you!).

DWS: I agree. This is taken care of in the rewrite. The discussion of what cannot be supported is removed (no change of semantics but I agree with your observation that the bullet is no longer required).

Separate Compilation Support

7. Section 18.3, second paragraph, states "The items that can be shared between units of compilation...". How are they shared? How can I end up with multiple "units of compilation"? Isn't what ever I read into my simulation or synthesis tool for a given run the "unit of compilation" for that run? Or is it that I can have a hierarchy with a top-level "unit of compilation" that is made up of several lower-level "units of compilation"? Section 18.3 leaves a lot to the imagination, and apparently I am not able to imagine what the author's of this section were imagining when they wrote it.

DWS: There can be multiple units of compilation that can be compiled independently. When these are combined in to a design the design is elaborated and all compilation units compose the design. Compilation units do not nest nor have a hierarchy of compilation units. I think we cleared this up in our conversation. The fully elaborated design is composed of all of the compilation units (that include all of the declarations used within the design). This seems quite natural and is similar (although the resolution and elaboration semantics are different) to how other languages work. Items are shared in that they live in a package and can then be accessed or referenced from different compilation units using the import or scope resolution operator.

$unit always refers to the compilation-unit scope associated with the compilation unit the scope in which the $unit is used is defined. Any tools that must expose this will have to provide a mechanism for each of the compilation-unit scopes to be disambiguated if the contents of the compilation-unit scope are visible (debuggers for example could use a name given to the compilation unit when it is defined to the tool).

8. Section 18.3, second paragraph states "...the PLI must provide an iterator to iterate through all of the units of compilation". The 1364-2001 standard defines that the PLI works with an elaborated data structure. It seems to me that this elaborated data structure is the one and only unit of compilation. Further, within this elaborated data structure, global objects can only exist once, regardless of whether there was a single compilation or separate compilations. For example, in the final elaborated data structure that the PLI sees, I can only have one global variable called "resetN", even if every module had been compiled separately with a global variable called "resetN". The idea of iterating through multiple local roots means I will find multiple global resetN variables.

DWS: The elaborated data structure contains the instantiated hierarchy of all of the items defined within the compilation units. The items within the compilation-unit scopes are treated as independent packages that are referenced by the items that are in the same compilation unit. There are NO truly global variables (resetN) must defined in a package to be globally available (and appropriately imported or referenced). If each compilation-unit scope contains a resetN then they will all be different and will show up in the PLI data model as independent variables. The real issue is how packages are going to be added to the PLI model and, related, how the compilation-unit scopes will appear. This will have to be handled when the PLI is enhanced to support packages and compilation-unit scopes.

9. Section 18.3, fourth paragraph (counting the deleted one), states "...compiler directives from one separately compiled unit shall not impact the behavior of another separately compiled unit". This paragraph should include a cautionary note that the affects of compiler directives may be different between separate compilation of design units versus single compilation of all the files that make up a complete design.

DWS: I agree. The following sentence was added:

This may result in a difference of behavior between compiling the units separately or as a single compilation unit containing the entire source.

Packages

10. Section 18.4, throughout this section and subsections, the term "symbol" is used. What is a "symbol"? It is not defined anywhere that I am aware of. I suspect that the correct term in this context, and as used in the rest of the LRM, is "identifier".

DWS: I agree. The section has been rewritten using identifier instead of symbol.

11. Section 18.4, second paragraph states that "The package declaration creates a top-level region...". Does this mean a package is part of the design hierarchy tree? It infers this, and it infers that I can then reference something in a package using a relative or full hierarchical path name, including starting with $root.

DWS: A package is not part of the design hierarchy tree. It is a scope that contains declarations that can be used (and shared) among modules, programs, etc.. The reference is with $unit and not $root since this is not part of the design instantiation tree. I would recommend changing the sentence to:

> The package declaration creates a scope that contains declarations intended to be shared among one or more compilation units, modules, macromodules, interfaces, or programs. Items

12. Section 18.4, second paragraph, last sentence states "...the same way the initialization occurs for variables declared in the local root". Where is variable initialization in local root defined? Add a cross reference.

DWS: I agree. This has been changed to:

> Variable declaration assignments within the package shall occur before any **`initial`** or **`always`** blocks are started, in the same way as variables declared in a compilation unit or module.

13. Section 18.4.1 first paragraph states "Packages must be defined before they are referenced...". If packages are a hierarchical unit, as inferred in 18.4, second paragraph, then this is both a radical departure from the nature of Verilog, and should not be necessary. All named hierarchical scopes in Verilog can be referenced before being defined. Identifiers within any Verilog named scope can also be referenced before being defined.

DWS: Packages are not part of the hierarchical unit as discussed above. Since the items within the package are generally variable and type declarations this is consistent with Verilog. I suggest changing this to:

> Packages must exist in order for the items they define to be recognized by the scopes in which they are imported.

14. Section 18.4.1, the paragraph immediately below Syntax 18-2 box, which reads "An alternate method for utilizing declarations is the import statement.", is out of place. Should this paragraph come before the syntax box?

DWS: This appears to be a problem with the PDF display of the document. It reads as you suggest in the source. The new version reads correctly.

15. Section 18.4.1, fourth paragraph uses the term "simple name". I can guess what this term refers to, but I might guess wrong. It should be defined.

DWS: I do not believe we need the term “declared simple name”. I recommend changing it to:

> It allows identifiers declared within packages to be visible within the current scope without a package name qualifier.

16. Section 18.4.1, seventh paragraph states "Wildcard import allows symbols defined...". Should this say "Wildcard import allows all symbols defined..."

DWS: I agree and recommend:

> A wildcard import allows all symbols declared within

17. Section 18.4.1, eighth paragraph, last sentence states "If the same symbol is defined in two wildcard imports in the same scope, the symbol shall be undefined with that scope.". Two comments:

a) This conflict should be a required warning or error (I prefer error). I should not have to wait until I get an error on line 541 of my source code saying that I referenced an undeclared identifier, and then wonder why it was not declared when I thought had imported it from a package (not realizing that another package used the same identifier name).

b). The mandatory "shall be undefined" conflicts with the second paragraph in section 18.4.2 (after the table) which states that the same symbol in two packages can end up using the symbol from one of the packages. These conflicting paragraphs would be resolved if conflicting symbols in two wildcard imports are a mandatory error.

DWS: The core concept here is that use of import name::* does not force the names to exist within the importing scope. It makes the names available for use if there are no conflicts and no prior definitions. Table 18.1 indicates that the names will be undefined and if c is used it will be an error (due to the lack of definition). If the same name is wildcard imported from two different packages then it is ambiguous which one it comes from. If an explicit import and wildcard is used then it is clear which one is used. I would recommend the following change:

> If the same identifier is wildcard imported into a scope from two different packages, the identifier shall be undefined within that scope and result in an error if the identifier is used.

18. Section 18.4.2, example code: there are two declarations of "const c...". Per the BNF, "const" must be followed by a data type, not an identifier.

DWS: I agree. The following is a correction to the example:

```
package p;
      typedef enum { FALSE, TRUE } BOOL;
      const BOOL c = FALSE;
endpackage;

package q;
      const int c = 0;
endpackage;
```

19. Section 18.4.2, table 8.x (note that this will become table 18.1 when I bring in to the LRM). The heading of the first column in the table is labeled "Syntax". Everything in that column is code examples, not BNF. I suggest making the column heading "Code example" or "Example".

DWS: I agree. I suggest replacing Syntax with Example.

20. Section 18.4.2, column 1, first row after the headings:

a) The lines "p::c;" and p::TRUE;" are not complete procedural statements, and should not have semicolons.

b) The line "e.g.:" is in source code font. I had to read it three times before figuring out that it is not code. I suggest completely removing the lines "p::c;", p::TRUE;" and "e.g.:". The actual code examples in the last two lines of the table cell are all that is needed.

DWS: I agree. I suggest we change the first column, first row after the heading to:

~~p::c;~~
~~p::TRUE;~~

~~e.g.:~~
`u = p::c;`
y = p::TRUE;

21. Section 18.4.2, table 8.x, third row (not including heading row), last column, states "it shall be illegal to reference c before the import of p::c". This conflicts with Verilog's implicit net declaration rules, where "c" could have already have inferred as a net by appearing on the LHS of a continuous assignment, as a connection to a module instance port, or as a connection to primitive instance terminal. The impact of implicit nets and references to imported identifiers from packages is not covered at all in section 18.4.

DWS: In Section 3.5 of 1364-2001 it states that:

> In the absence of an explicit declaration, an implicit net of default net type shall be assumed in the following circumstances:

The presence of an import results in an explicit declaration. As such, if a declaration is explicitly or implicitly imported it does exist and the implicit net declaration will not be created. There is no backward compatibility problem here since the import mechanism does not exist in 1364-2001. The same is true for implicit scalar net declarations. This seems like a pretty straightforward result of Section 3.5 of 1364-2001 and the discussion on packages.

22. Section 18.6 first paragraph. The first sentence conflicts with the last sentence. Or does it? The first sentence uses the words "time unit" and "time precision" (note the spaces) and says there can be at most one of these. The last sentence uses the words "timeunit" and timeprecision" and says they can be repeated. Very confusing. Historically, the last sentence was added to SV 3.0 to allow $root, which could be scattered through hundreds of source code files (bad idea--should be illegal), to contain more than one timeunit and/or timeprecision declaration, provided they did not conflict.

DWS: I think that definition is consistent (but confusing). More confusing is the BNF which does not support what is here. I believe this needs to be clarified by SV-BC independent of this proposal.

23. Section 18.6, numbered item 1) has the parenthetical clause "(program and packages cannot be nested)". This is bad grammar in that it mixes singular and plural nouns. "program" should be plural if this clause is referring to the construct rather than the keyword. "packages" should be singular if the clause is referring to the keyword rather than the construct. I prefer the former (referring to the constructs).

DWS: I agree. The correct clause is:

(programs and packages cannot be nested).

24. Section 18.9, numbered item 1), two things:

a) The last sentence states "...the name shall not be used again...within local root in any compilation unit". Should this be in "any compilation unit" or "any other compilation unit"? There are also punctuation errors in this paragraph, but I will fix those when I edit the changes into the LRM.

b) Does the restrictions in this paragraph still allow for Verilog-2001's concept of multiple modules or primitives with the same name but made unique through the use of libraries? If I compile a library file or directory, is it then a compilation unit? If so, and I compile another library with a module of the same name, it would become another compilation unit that has a module with the same name, which this rule makes illegal. That would break Verilog-2001.

DWS: The attempt here was to be explicitly compatible with 1364 will acknowledging that names can be reused within nested definitions. The issue of how module names are reused when in libraries is a problem in 3.12 of 1364-2001 LRM. Once that is fixed, we can modify it here to be consistent. I suggest the following wording:

1) The *definitions name space* unifies all the non-nested **module**, **macromodule**, **primitive**, **program**, and **interface** identifiers defined outside of all other declarations. Once a name is used

to define a **module**, **macromodule**, **primitive**, **program**, or **interface** within one compilation unit the name shall not be used again (in any compilation unit) to declare another non-nested **module**, **macromodule**, **primitive**, **program**, or **interface** outside of all other declarations. This is compatible with the *definitions name space* as defined in of IEEE 1364-2001.

25. Section 18.9, numbered item 6): Should for loops with variable declarations as part of the initial assignment be included in the list of block name spaces?

DWS: Probably, I will let SV-BC provide the errata to fix this section since it is not related to the separate compilation.

26. Section 18.9, numbered item 7) does not mention ref ports. I suggest changing the third sentence to "The connection can be unidirectional (input or output), bidirectional (inout) or a hierarchical reference (ref)." and the sixth sentence to "The port type declarations are input, output, inout and ref.".

DWS: Probably, I will let SV-BC provide the errata to fix this section since it is not related to the separate compilation.

Finally, I did not review the changes to the BNF. I trust someone else has.

DWS: Brad has done a couple of review passes.

Again, I am glad that the atrocities of SV 3.0's $root are being addressed for SV 3.1a. I like the way packages solve the problem. I would heartily support totally removing the concept of $root and local root from the standard.

Stu

~~~~~~
Stuart Sutherland
stuart@sutherland-hdl.com
phone: 503-692-0898

Sutherland HDL Inc.
22805 SW 92nd Place
Tualatin, OR 97062

Sutherland HDL, Inc. -- Training Engineers to be Verilog, SystemVerilog and VHDL Wizards! www.sutherland-hdl.com
~~~~~~