Separate Compilation Discussion

The objectives of supporting separate compilation are:

- Support compilation of collections of SystemVerilog files. This is known as a unit of compilation.
- Support restricted global reference between units of compilation of SystemVerilog code.
- Support definition of items that can be shared within a unit of compilation.
- Support collecting declarative items that into a restricted namespace that can be referenced globally and imported. This is the purpose of the package definition.

In order to support separate compilation and to simplify the current semantics of $root within SystemVerilog the following changes are proposed:

- Change the definition of $root within Section 18 (18.1, 18.2, 18.3, 18.6, 18.9) to limit the items that can be placed in $root to:
  - any item that can be defined in a package
  - an import of one or more items from a package
  - modules, macromodules, primitives, programs, interfaces, and packages
  - directives: timeunit, timeprecision, and bind
- This implies the removal of instances (module, interface, program) (Section 18), statements, genvar, and assert and cover statements (Section 17.12) from $root
- Rework relation between $root and library statements (Section 21.3).
- Include packages and programs within libraries within Section 21.2.
- Addition of two new sections on packages and separate compilation

Within Section 17.12 change (as shown in red):

A concurrent assertion statement can be specified in:

— an always block or initial block as a statement, wherever these blocks can appear\

— a module as a *module_or_generate_item*

— an interface as an *interface_or_generate_item*

— a program as a *non_port_program_item*

— ~~$root~~

Within Section 18.1 change (as shown in red):

— A global declaration space, visible to all modules, interfaces, and programs at all levels of hierarchy within the unit of compilation (see Section 18.3)
— Separate compilation support
— A concept of packages to simplify sharing of data, types and functions

Within Section 18.2 change as shown in red.

18.2 The $root top level

In SystemVerilog there is a top level called $root, which is the whole source text within a compilation unit. This allows declarations outside any named modules or interfaces, unlike Verilog.

~~SystemVerilog requires an elaboration phase. All modules and interfaces must be parsed before elaboration.~~

~~The order of elaboration shall be: First, look for explicit instantiations in $root. If none, then look for implicit instantiations (i.e. uninstantiated modules). Next, traverse non-generate instantiations depth-first, in source order. Finally, execute generate blocks depth-first, in source order.~~

The source text can include the declaration ~~and use of~~ modules and interfaces. Modules can include the declaration and use of other modules and interfaces. Interfaces can include the declaration and use of other interfaces. A module or interface need not be declared before it is used in text order.

~~A module can be explicitly instantiated in the $root top-level.~~ All uninstantiated modules become implicitly instantiated within the top level, which is compatible with Verilog.

The following paragraphs compare the $root top level and modules.

The $root top level:

— has a single occurrence per compilation unit

— can be distributed across any number of files

— can contain any item that can be defined within a package and the definitions are in a namespace local to the compilation unit and can be accessed through the hierarchy within the unit of compilation

— can contain declarations of modules, macromodules, primitives, programs, interfaces, and packages which are in a global namespace

— can contain one timeunit and timeprecision directive within the unit of compilation

— variable and net definitions are in a local ~~global~~ name space and can be accessed throughout the hierarchy within the unit of compilation

— task and function definitions are in a local ~~global~~ name space and can be accessed throughout the hierarchy within the unit of compilation

~~— shall not contain `initial` or `always` procedures~~

~~— can contain procedural statements, which shall be executed one time, as if in an `initial` procedure~~

— can contain ~~assertion declarations, assertion statements and~~ bind directives

— shall not contain instances, procedural statements, genvar declaration, or assert or cover statements

Modules:

— can have any number of module definitions

— can have any number of module instances, which create new levels of hierarchy

— can be distributed across any number of files, and can be defined in any order

— variable and net definitions are in the module instance name space and are local to that scope

— task and function definitions are in the module instance name space and are local to that scope

— can contain any number of **`initial`** and **`always`** procedures

— shall not contain procedural statements that are not within an **`initial`** procedure, **`always`** procedure, task, or function

When an identifier is referenced within a scope, SystemVerilog follows the Verilog name search rules, and then searches in the $root ~~global~~ name space. An identifier in the $root ~~global~~ name space can be explicitly selected by pre-pending `$root.` to the identifier name. For example, a $root ~~global~~ variable named `system_reset` can be explicitly referenced from any level of hierarchy using `$root.system_reset`.

~~The $root space can be used to model abstract functionality without modules. The following example illustrates using the $root space with just declarations, statements and functions.~~

```
typedef int myint;

function void main ();
        myint i,j,k;
        $display ("entering main...");
        left (k);
        right (i,j,k);
        $display ("ending... i=%0d, j=%0d, k=%0d", i, j, k);
endfunction

function void left (output myint k);
        k = 34;
        $display ("entering left");
endfunction

function void right (output myint i, j, input myint k);
        $display ("entering right");
        i = k/2;
        j = k+i;
endfunction

main();
```

ADD new sections 18.3 and 18.4 after Section 18.2 (renumber succeeding sections and syntax boxes):

18.3 Separate Compilation Support

SystemVerilog supports separate compilation through the idea of a separately compiled unit. The unit of compilation is defined as a collection of one or more files. This collection of files share a local root and $root refers to the definitions within the unit of compilation. The exact mechanism for defining which files go into each individual compilation unit is tool specific. The requirement is that a tool provides a mechanism to define the list of files which make up the compilation unit. Two extreme cases are:

1. all files make a single compilation unit (in which case the items in $root are accessible anywhere within the design)
2. each file is a separate compilation unit (in which case the items in $root are accessible only to the items defined within the file)

The items that can be shared between units of compilation are modules, macromodules, primitives, programs, interfaces, and packages. All other items which are defined within the local root cannot be accessed by name outside the unit of compilation. Access to the items in $root that are not shared can be accessed using the PLI which must provide an iterator to iterate through all of the units of compilation. Collision of items which are shared obey the standard rules as if the items where loaded together as a single text stream.

References within a unit of compilation can access objects within the $root space either by the object's name (<object_name>) or by $root.<object_name>. In resolving a reference, SystemVerilog follows the standard name search rules within the current separate compilation unit, and then searches the $root for the current separate compilation unit, and then searches the scope containing modules, macromodules, primitives, programs, interfaces.

Verilog supports the idea that compiler directives once seen by a tool will apply to all forthcoming modules or files in the design. This behavior shall be supported within a separately compiled unit; however, compiler directives from one separately compiled unit shall not impact the behavior of another separately compiled unit.

18.4 Packages

SystemVerilog packages provide an additional mechanism for sharing parameters, data, type, task, function, sequence, and property declarations amongst multiple SystemVerilog modules, interfaces and programs. Packages are explicitly named scopes appearing at the outer level of the source text (at the same level as modules, primitives, interfaces, etc.). Types, variables, tasks, functions, sequences, and properties may be declared within a package. Such declarations may be referenced within modules, macromodules, interfaces, programs, and other packages by either import or hierarchical name.

```
package_declaration ::=                 // A.1.3
        { attribute_instance } package package_identifier
        { package_item } endpackage [ : package_identifier ]

package_item ::=                        // New A.1.9
         net_declaration
        | data_declaration
        | parameter_declaration
        | local_parameter_declaration
        | spec_param_declaration
        | task_declaration
        | function_declaration
        | dpi_import_export
        | package_import_declaration
        | class_declaration
        | extern_constraint_declaration
        | extern_method_declaration
        | sequence_declaration
        | property_decleration
        | anonymous_program

anonymous_program ::= program ; { anonymous_program_item } endprogram

anonymous_program_item ::=
         task_declaration
        | function_declaration
        | class_declaration
```

Syntax 18-1--Package syntax (excerpt from Annex A)

The package declaration creates a top-level region to contain declarations intended to be shared among one or more modules, macromodules, interfaces, or programs. Items within packages are generally type definitions, tasks, and functions. Items within packages are constrained to include only hierarchical references that are contained within the same unit of compilation that contains the package. This means that sequences and properties must be fully parameterized within a package or reference items in the same unit of compilation. It is also possible to populate packages with parameters, variables and nets. This may occasionally be useful for globals that aren't conveniently passed down through the hierarchy. Any variables that have initial values defined for them will be initialized in the same way the initialization occurs for variables declared in $root.

The following is an example of a package:

```
package ComplexPkg;
      typedef struct {
            float i, r;
      } Complex;

      function Complex add(input Complex a, b)
            add.r = a.r + b.r;
            add.i = a.i + b.i;
      endfunction

      function Complex mul(input Complex a, b)
            mul.r = (a.r * b.r) + (a.i * b.i);
```

```
            mul.i = (a.r * b.i) + (a.i * b.r);
        endfunction
    endpackage : ComplexPkg
```

18.3.1 Referencing data in packages

Packages must be defined before they are referenced to allow the types they define to be recognized by the modules that import from them.

One of the ways to utilize declarations made in a packages is to reference them using the namespace operator "::".

```
ComplexPkg::Complex cout = ComplexPkg::mul(a, b);
```

An alternate method for utilizing declarations is the import statement.

```
package_import_declaration ::=
        import package_import_item { , package_import_item } ;

package_import_item ::=
          package_identifier :: identifier
        | package_identifier :: *
```

Syntax 18-2--Import syntax (excerpt from Annex A)

The import statement provides direct visibility of symbols within Packages. It allows those symbols declared within Packages to be visible within the current scope by their declared simple name. Package hierarchical names to the imported symbols can be created as if the symbol were defined in the importing scope. Two forms of the import statement are provided: explicit import, and wildcard import. Explicit import allows control over precisely which symbols are imported:

```
import ComplexPkg::Complex;
import ComplexPkg::add;
```

Explicit imports are treated similarly to a declaration. An explicit import shall be illegal if another symbol by the same name has already been declared or imported into the same scope unless the symbol is from the same package. Similarly, after importing a symbol by a given name, it shall be illegal to then declare a symbol by that same name within the same scope.

Wildcard import allows symbols defined within a package to be imported provided the symbol is not otherwise defined in the importing scope:

```
import ComplexPkg::*;
```

All the symbols within a package implied by a wildcard import are candidates for import. They, in fact, become imported only if there are no other symbols by the same name declared or imported in the same scope. Similarly, their visibility may be limited by a subsequent declaration of the same name in the same scope. If the same symbol is defined in two wildcard imports in the same scope, the symbol shall be undefined within that scope.

8.13.2 Search order Rules

Table 8.x describes the search order rules for the declarations imported from a package. To understand the table, consider the following package declarations:

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

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

Syntax	Description	Scope containing a local declaration of c	Scope not containing a local declaration of c	Scope contains a declaration of c imported using import q::c	Scope contains a declaration of c imported as import q::*

`p::c;`
`p::TRUE;`
 	
`e.g.:`
`u = p::c;`
`y = p::TRUE;`
	A qualified package identifier is visible in any scope (without the need for an import clause).	**OK**.
		
		Direct reference to c refers to the locally declared c.
		
		p::c refers to the c in package p.
			OK
			
			Direct reference to c is illegal since it is undefined.
			
			p::c refers to the c in package p.
				OK.
				
				Direct reference to c refers to the c imported from q.
				
				p::c refers to the c in package p.
					OK.
					
					Direct reference to c refers to the c imported from q.
					
					p::c refers to the c in package p.

`import` `p::*;`
 	
`. . .`
 	
`y = FALSE;`
	All declarations inside package p become potentially directly visible in the importing scope:
	
	• `c`
	• `BOOL`
	• `FALSE`
	• `TRUE`
		OK.
		
		Direct reference to c refers to the locally declared c.
		
		Direct reference to other identifiers (e.g., FALSE) refer to those implicitly imported from package p.
			OK.
			
			Direct reference to c refers to the c imported from package p.
				OK.
				
				Direct reference to c refers to the c imported from package q.
					OK / ERROR
					
					c is undefined in the importing scope. Thus, a direct reference to c is illegal and results in an error.
					
					The import clause is otherwise allowed.

`import` `p::c;`
 	
`. . .`
 	
`if``( !`
`c ) ...`
	The imported identifiers become directly visible in the importing scope:
	
	• `c`
		ERROR.
		
		It shall be illegal to import an identifier defined in the importing scope.
			OK.
			
			Direct reference to c refers to the c imported from package p.
				ERROR.
				
				It shall be illegal to import an identifier defined in the importing scope.
					OK / ERROR
					
					It shall be illegal to reference c before the import of p::c.
					
					Otherwise, direct reference to c refers to the c imported from

					package p.

Table 8.x Scoping Rules for Package Importation

When using the **import** p::c form of importation, the use of a variable forces the import of that variable into the local scope, thus creating an error if another package with the same variable name is imported later. This is shown in the following example:

```
module foo;
      import q::*;
      wire a=c; // This statement forces the import of
      q::c;
      import p::c; // The conflict with q::c and p::c
      creates an error.
endmodule;
```

Within Section 18.3 change (as shown in red):

18.3 Module declarations

SystemVerilog adds the capability to nest module declarations, ~~and to instantiate modules in the $root toplevel space, outside of other modules~~.

```
module m1(...); ... endmodule

module m2(...); ... endmodule

module m3(...);

      m1 i1(...); // instantiates the local m1 declared below
      m2 i4(...); // instantiates m2 - no local declaration
      module m1(...); ... endmodule // nested module declaration,
                                    // m1 module name is in m3's
                                  name space
endmodule

~~m1 i2(...);    // module instance in the $root space,~~
                 ~~// instantiates the module m1 that is not nested in~~
                 ~~another module~~
```

Within Section 18.6 change (as shown in red):

If a **timeunit** is not specified in the module or interface definition, then the time unit is shall be determined using the following rules of precedence:

1) If the module or interface definition is nested, then the time unit ~~is~~ shall be inherited from the enclosing module or interface.
2) Else, if a `'timescale` directive has been previously specified, then the time unit ~~is~~ shall be set to the units of the last `'timescale` directive.
3) Else, if the `$root` top level has a time unit, then the time unit ~~is~~ shall be set to the time units of the unit of compilation ~~root module~~.
4) Else, if any unit of compilation has a time unit, then the time unit shall be set to the time of the unit of compilation (it is an error to have different time units in different units of compilation used within a single design)
5) Else, the default time unit is shall be used.

Within Section 18.9 change (as shown in red):

18.9 Name spaces

SystemVerilog has six ~~five~~ name spaces for identifiers. ~~Verilog's global definitions name space collapses onto the module name space and exists as the top-level scope, `$root`. Module, primitive, package, program, and interface identifiers are local to the module name space where there are defined.~~ The six ~~five~~ name spaces are described as follows:

1) The *definitions name space* unifies all the module, macromodule, primitive, program, interface, and package identifiers defined within $root among all compilation units. Once a name is used to define a module, macromodule, primitive, program, interface, or package within $root the name shall not be used again to declare another module, macromodule, primitive, program, interface, or package within $root.

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

3) The *module name space* is introduced by ~~`$root and`~~ the **`module`**, **`macromodule`**, **`interface`**, **`package,`** **`program,`** and **`primitive`** constructs. It unifies the definition of module, macromodule, interface, program, functions, tasks, named blocks, instance names, parameters, named events, net type of declaration, variable type of declaration and user defined types.

4) The *block name space* is $root or is introduced by named or unnamed blocks, the **`specify`**, **`function`**, and **`task`** constructs. It unifies the definitions of the named blocks, functions, tasks, parameters, named events, variable type of declaration and user defined types.

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

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

Within Section 21.2 change (as shown in red):

A library is a named collection of cells. A cell is a module, macromodule, primitive, interface, program, package, or configuration. A configuration is a specification of which source files bind to each instance in the design.

Within Section 21.3 change (as shown in red):

~~21.3 Library map files~~

~~Verilog 2001 specifies that library declarations, include statements, and config declarations are normally in a mapping file that is read first by a simulator or other software tool. SystemVerilog does not require a special library map file. Instead, the mapping information can be specified in the `$root` top level.~~