Chapter 2

SystemVerilog Declaration Spaces

Verilog only has limited places in which designers can declare variables and other design information. SystemVerilog extends Verilog's declaration spaces in several ways. These extensions make it much easier to model complex design data, and reduce the risk of hard-to-find coding errors. SystemVerilog also enhances how simulation time units are defined.

The topics discussed in this chapter include:

- Packages definitions and importing definitions from packages
- $unit compilation declaration space
- Declarations in unnamed blocks
- Enhanced time unit definitions

Before examining in detail the many new data types that SystemVerilog offers, it is important to know *where* designers can define important information that is used in a design. To illustrate these new declaration spaces, this chapter will use several SystemVerilog data types that are not discussed until the following chapters. In brief, some of the new types used in this chapter are:

`logic` — a 1-bit 4-state variable, like the Verilog **`reg`** type; can be declared as any vector size (discussed in Chapter 3).

`enum` — an enumerated net or variable with a labeled set of values; similar to the C enum type, but with additional syntax and semantics for modeling hardware (discussed in Chapter 4).

typedef — a user-defined data type, constructed from built-in types or other user-defined types, similar to the C typedef (discussed in Chapter 4).

struct — a collection of variables that can be referred to individually or collectively, similar to the C struct type (discussed in Chapter 5).

2.1 Packages

Verilog requires local declarations

In Verilog, declarations of variables, nets, tasks and functions must be declared within a module, between the **module**...**endmodule** keywords. The objects declared within a module are local to the module. For modeling purposes, these objects should be referenced within the module in which they are declared. Verilog also allows hierarchical references to these objects from other modules for verification purposes, but these cross-module references do not represent hardware behavior, and are not synthesizable. Verilog also allows local variables to be defined in named blocks (formed with **begin**...**end** or **fork**...**join**), tasks and functions. These declarations are still defined within a module, however, and, for synthesis purposes, only accessible within the module.

Verilog does not have a place to make global declarations, such as global functions. A declaration that is used in multiple design blocks must be declared in each block. This not only requires redundant declarations, but it can also lead to errors if a declaration, such as a function, is changed in one design block, but not in another design block that is supposed to have the same function. Many designers use include files and other coding tricks to work around this shortcoming, but that, too, can lead to coding errors and design maintenance problems.

SystemVerilog adds user-defined types to Verilog

SystemVerilog adds user-defined types, using **typedef**. It is often desirable to use the definition of user-defined types in multiple modules. Using Verilog rules, where declarations are always local to a module, it would be necessary to duplicate a user-defined type definition in each and every module in which the definition is used. Redundant local definitions would not be desirable for user-defined types.

2.1.1 Package definitions

SystemVerilog adds packages to Verilog

To enable sharing a user-defined type definition across multiple modules, SystemVerilog adds ***packages*** to the Verilog language. The concept of packages is leveraged from the VHDL language. SystemVerilog packages are defined between the keywords **package** and **endpackage**.

The synthesizable constructs that a packages can contain are:

- **parameter** and **localparam** constant definitions
- **const** variable definitions
- **typedef** user-defined types
- Fully automatic **task** and **function** definitions
- **import** statements from other packages
- Operator overload definitions

Packages can also contain global variable declarations, static task definitions and static function definitions. These are not synthesizable, however, and are not covered in this book.

package definitions are independent of modules

A package is a separate declaration space. It is not embedded within a Verilog module. A simple example of a package definition is:

Example 2-1: A package definition

```
package definitions;

  parameter VERSION = "1.1";

  typedef enum {ADD, SUB, MUL} opcodes_t;

  typedef struct {
    logic [31:0] a, b;
    opcodes_t    opcode;
  } instruction_t;

  function automatic [31:0] multiplier (input [31:0] a, b);
    // code for a custom 32-bit multiplier goes here
    return a * b;  // abstract multiplier (no error detection)
  endfunction
endpackage
```

parameters in packages cannot be redefined

Packages can contain **parameter**, **localparam** and **const** constant declarations. The **parameter** and **localparam** constants are Verilog constructs. A **const** constant is a SystemVerilog constant, which is discussed in section 3.10 on page 71. In Verilog, a **parameter** constant can be redefined for each instance of a module, whereas a **localparam** cannot be directly redefined. In a package, however, a **parameter** constant cannot be redefined, since it is not part of a module instance. In a package, **parameter** and **localparam** are synonymous.

2.1.2 Referencing package contents

Modules and interfaces can reference the definitions and declarations in a package four ways:

- Direct reference using a scope resolution operator
- Import specific package items into the module or interface
- Wildcard import package items into the module or interface
- Import package items into the `$unit` declaration space

The first three methods are discussed in this section. Importing into `$unit` is discussed later in this chapter, in section 2.2 on page 14.

Package references using the scope resolution operator

:: is used to reference items in packages

SystemVerilog adds a `::` "*scope resolution operator*" to Verilog. This operator allows directly referencing a package by the package name, and then selecting a specific definition or declaration within the package. The package name and package item name are separated by double colons (`::`). For example, a SystemVerilog module port can be defined as an `instruction_t` type, where `instruction_t` is defined in the package `definitions`, illustrated in example 2-1 on page 9.

Example 2-2: Explicit package references using the `::` scope resolution operator

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
  always_ff @(posedge clock) begin
```

```
    case (IW.opcode)
      definitions::ADD : result = IW.a + IW.b;
      definitions::SUB : result = IW.a - IW.b;
      definitions::MUL : result = definitions::
                                      multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

Explicit package reference help document source code

Explicitly referencing package contents can help to document the design source code. In example 2-2, above, the use of the package name makes it is very obvious where the definitions for `instruction_t`, `ADD`, `SUB`, `MUL` and `multiplier` can be found. However, when a package item, or items, needs to be referenced many times in a module, explicitly referencing the package name each time may be too verbose. In this case, it may be desirable to import package items into the design block.

Importing specific package items

import statements make package items visible locally

SystemVerilog allows specific package items to be imported into a module, using an **`import`** statement. When a package definition or declaration is imported into a module or interface, that item becomes visible within the module or interface, as if it were a locally defined name within that module or interface. It is no longer necessary to explicitly reference the package name each time that package item is referenced.

Importing a package definition or declaration can simplify the code within a module. Example 2-2 is modified below as example 2-3, using import statements to make the enumerated type labels local names within the module. The case statement can then reference these names without having to explicitly name the package each time.

Example 2-3: Importing specific package items into a module

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
```

```
  import definitions::ADD;
  import definitions::SUB;
  import definitions::MUL;
  import definitions::multiplier;

  always_comb begin
    case (IW.opcode)
      ADD : result = IW.a + IW.b;
      SUB : result = IW.a - IW.b;
      MUL : result = multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

Importing an enumerated type definition does not import the labels used within that definition.

In example 2-3, above, the following import statement would not work:

```
import definitions::opcode_t;
```

enumerated labels must be imported in order to reference locally

This import statement would make the user-defined type, `opcode_t`, visible in the module. However, it would not make the enumerated labels used within `opcode_t` visible. Each enumerated label must be explicitly imported, in order for the labels to become visible as local names within the module. When there are many items to import from a package, using a wildcard import may be more practical.

Wildcard import of package items

all items in a package can be made visible using a wildcard

SystemVerilog allows package items to be imported using a wildcard, instead of naming specific package items. The wildcard token is an asterisk (`*`). For example:

```
import definitions::*;  // wildcard import
```

A wildcard import does not automatically import all package contents.

wildcard imports do not automatically import the entire package

When package items are imported using a wildcard, only items actually used in the module or interface are actually imported. Definitions and declarations in the package that are not referenced are not imported.

Local definitions and declarations within a module or interface take precedence over a wildcard import. An import that specifically names package items also takes precedence over a wildcard import. From a designer's point of view, a wildcard import simply adds the package to the search rules for an identifier. Software tools will search for local declarations first (following Verilog search rules for within a module), and then search in any packages that were imported using a wildcard. Finally, tools will search in SystemVerilog's `$unit` declaration space. The `$unit` space is discussed in section 2.2 on page 14 of this chapter.

Example 2-4, below, uses a wildcard import statement. This effectively adds the package to the identifier search path. When the case statement references the enumerated labels of ADD, SUB, and MUL, as well as the function `multiplier`, it will find the definitions of these names in the `definitions` package.

Example 2-4: Using a package wildcard import

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
  import definitions::*;  // wildcard import

  always_comb begin
    case (IW.opcode)
      ADD : result = IW.a + IW.b;
      SUB : result = IW.a - IW.b;
      MUL : result = multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

In examples 2-3, and 2-4, for the IW module port, the package name must still be explicitly referenced. It is not possible to add an **`import`** statement between the module keyword and the module

port definitions. There is a way to avoid having to explicitly reference the package name in a port list, however, using the `$unit` declaration space. The `$unit` space is discussed in 2.2.

2.1.3 Synthesis guidelines

for synthesis, package tasks and functions must be automatic

When a module references a task or function that is defined in a package, synthesis will duplicate the task or function functionality and treat it as if it had been defined within the module. To be synthesizable, tasks and functions defined in a package must be declared as **`automatic`**, and cannot contain static variables. This is because storage for an automatic task or function is effectively allocated each time it is called. Thus, each module that references an automatic task or function in a package sees a unique copy of the task or function storage that is not shared by any other module. This ensures that the simulation behavior of the pre-synthesis reference to the package task or function will be the same as post-synthesis behavior, where the functionality of the task or function has been implemented within one or more modules.

For similar reasons, synthesis does not support variables declarations in packages. In simulation, a package variable will be shared by all modules that import the variable. One module can write to the variable, and another module will see the new value. This type of inter-module communication without passing values through module ports is not synthesizable.

2.2 $unit compilation-unit declarations

SystemVerilog has compilation units

SystemVerilog adds a concept called a ***compilation unit*** to Verilog. A compilation unit is all source files that are compiled at the same time. Compilation units provide a means for software tools to separately compile sub-blocks of an overall design. A sub-block might comprise a single module or multiple modules. The modules might be contained in a single file or in multiple files. A sub-block of a design might also contain interface blocks (presented in Chapter 10) and testbench program blocks (covered in the companion book, *SystemVerilog for Verification*[1]).

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

compilation-unit scopes contain external declarations

SystemVerilog extends Verilog's declaration space by allowing declarations to be made outside of package, module, interface and program block boundaries. These external declarations are in a ***compilation-unit scope***, and are visible to all modules that are compiled at the same time.

The compilation-unit scope can contain:

- Time unit and precision declarations (see 2.4 on page 28)
- Variable declarations
- Net declarations
- Constant declarations
- User-defined data types, using **`typedef`**, **`enum`** or **`class`**
- Task and function definitions

The following example illustrates external declarations of a constant, a variable, a user-defined type, and a function.

Example 2-5: External declarations in the compilation-unit scope (not synthesizable)

```
/******************* External declarations ******************/
parameter VERSION = "1.2a";   // external constant

reg resetN = 1;               // external variable (active low)

typedef struct packed {       // external user-defined type
  reg [31:0] address;
  reg [31:0] data;
  reg [ 7:0] opcode;
} instruction_word_t;

function automatic int log2 (input int n);  // external function
  if (n <=1) return(1);
  log2 = 0;
  while (n > 1) begin
    n = n/2;
    log2++;
  end
  return(log2);
endfunction

/******************** module definition ********************/
// external declaration is used to define port types
module register (output instruction_word_t  q,
                 input  instruction_word_t  d,
```

```
                input  wire                 clock );

  always @(posedge clock, negedge resetN)
    if (!resetN) q <= 0;  // use external reset
    else q <= d;
endmodule
```

External compilation-unit scope declarations are not global

A declaration in the compilation-unit scope is not the same as a global declaration. A true global declaration, such as a global variable or function, would be shared by all modules that make up a design, regardless of whether or not source files are compiled separately or at the same time.

SystemVerilog's compilation-scope only exists for source files that are compiled at the same time. Each time source files are compiled, a compilation-unit scope is created that is unique to just that compilation. For example, if module `CPU` and module `controller` both reference an externally declared variable called `reset`, then two possible scenarios exist:

- If the two modules are compiled at the same time, there will be a single compilation-unit scope. The externally declared `reset` variable will be common to both modules.
- If each module were compiled separately, then there would be two compilation-unit scopes, possibly with two different `reset` variables.

In the latter scenario, the compilation that included the external declaration of `reset` would appear to compile OK. The other file, when compiled separately, would have its own, unique $unit compilation space, and would not see the declaration of `reset` from the previous compilation. Depending on the context of how `reset` is used, the second compilation might fail, due to an undeclared variable, or it might compile OK, making `reset` an implicit net. *This is a dangerous possibility!* If the second compilation succeeds by making `reset` an implicit net, there will now be two signals called `reset`, one in each compilation. The two different `reset` signals would not be connected in any way.

2.2.1 Coding guidelines

$unit should only be used for importing packages

1. Do not make any declarations in the $unit space! All declarations should be made in named packages.
2. When necessary, packages can be imported into $unit. This is useful when a module or interface contains multiple ports that are of user-defined types, and the type definitions are in a package.

Directly declaring objects in the $unit compilation-unit space can lead to design errors when files are compiled separately. It can also lead to spaghetti code if the declarations are scattered in multiple files that can be difficult to maintain, re-use, or to debug declaration errors.

2.2.2 SystemVerilog identifier search rules

Declarations in the compilation-unit scope can be referenced anywhere in the hierarchy of modules that are part of the compilation unit.

the compilation-unit scope is third in the search order

SystemVerilog defines a simple and intuitive search rule for when referencing an identifier:

1. First, search for local declarations, as defined in the IEEE 1364 Verilog standard.
2. Second, search for declarations in packages which have been wildcard imported into the current scope.
3. Third, search for declarations in the compilation-unit scope.
4. Fourth, search for declarations within the design hierarchy, following IEEE 1364 Verilog search rules.

The SystemVerilog search rules ensure that SystemVerilog is fully backward compatible with Verilog.

2.2.3 Source code order

Data identifiers and type definitions must be declared before being referenced.

Variables and nets in the compilation-unit scope

undeclared identifiers have an implicit net type

There is an important consideration when using external declarations. Verilog supports implicit type declarations, where, in specific contexts, an undeclared identifier is assumed to be a net type (typically a **wire** type). Verilog requires the type of identifiers to be explicitly declared before the identifier is referenced when the context will not infer an implicit type, or when a type other than the default net type is desired.

external declarations must be defined before use

This implicit type declaration rule affects the declaration of variables and nets in the compilation-unit scope. Software tools must encounter the external declaration before an identifier is referenced. If not, the name will be treated as an undeclared identifier, and follow the Verilog rules for implicit types.

The following example illustrates how source code order can affect the usage of a declaration external to the module. This example will not generate any type of compilation or elaboration error. For module `parity_gen`, software tools will automatically infer `parity` as an implicit net type local to the module, since the reference to `parity` comes before the external declaration for the signal. On the other hand, module `parity_check` comes after the external declaration of `parity` in the source code order. Therefore, the `parity_check` module will use the external variable declaration.

```
module parity_gen (input wire [63:0] data );
  assign parity = ^data; // parity is an
endmodule                // implicit local net


reg parity;  // external declaration is not
             // used by module parity_gen
             // because the declaration comes
             // after it has been referenced


module parity_check (input  wire [63:0] data,
                     output logic       err);
  assign err = (^data != parity); // parity is
                                  // the $unit
endmodule                         // variable
```

2.2.4 Coding guidelines for importing packages into $unit

SystemVerilog allows module ports to be declared as user-defined types. The coding style recommended in this book is to place those definitions in one or more packages. Example 2-2 on page 10, listed earlier in this chapter, illustrates this usage of packages. An excerpt of this example is repeated below.

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
```

Explicitly referencing the package as shown above can be tedious and redundant when many module ports are of user-defined types. An alternative style is to import a package into the $unit compilation-unit scope, prior to the module declaration. This makes the user-defined type definitions visible in the SystemVerilog search order. For example:

```
// import specific package items into $unit
import definitions::instruction_t;

module ALU
(input  instruction_t  IW,
 input  logic          clock,
 output logic [31:0]   result
);
```

A package can also be imported into the $unit space using a wildcard import. Keep in mind that a wildcard import does not actually import all package items. It simply adds the package to the SystemVerilog source path. The following code fragment illustrates this style.

```
// wildcard import package items into $unit
import definitions::*;

module ALU
(input  instruction_t  IW,
 input  logic          clock,
 output logic [31:0]   result
);
```

Importing packages into $unit with separate compilation

The same care must be observed when importing packages into the $unit space as with making declarations and definitions in the $unit space. When using $unit, file order dependencies can be an issue, and multiple $units can be an issue.

file order compilation dependencies

When items are imported from a package (either with specific package item imports or with a wildcard import), the import statement must occur before the package items are referenced. If the package import statements are in a different file than the module or interface that references the package items, then the file with the import statements must be listed first in the file compilation order. If the file order is not correct, then the compilation of the module or interface will either fail, or will incorrectly infer implicit nets instead of seeing the package items.

multiple file compilation versus single file compilation

Synthesis compilers, lint checkers, some simulators, and possibly other tools that can read in Verilog and SystemVerilog source code can often compile one file at a time or multiple files at a time. When multiple files are compiled as single compilation, there is a single $unit space. An import of a package (either specific package items or a wildcard import) into $unit space makes the package items visible to all modules and interfaces read in after the import statement. However, if files are compiled separately, then there will be multiple separate $unit compilation units. A package import in one $unit will not be visible in another $unit.

using import statements in every file

A solution to both of these problems with importing package items into the $unit compilation-unit space is to place the import statements in every file, before the module or interface definition. This solution works great when each file is compiled separately. However, care must still be taken when multiple files are compiled as a single compilation. It is illegal to import the same package items more than once into the same $unit space (The same as it is illegal to declare the same variable name twice in the same name space).

conditional compilation with $unit package imports

A common C programming trick can be used to make it possible to import package items into the $unit space with both single file compilation and multiple file compilation. The trick is to use conditional compilation to include the import statements the first time the statements are compiled into $unit, and not include the statements if they are encountered again in the same compilation. In order to tell if the import statements have already been compiled in the current

$unit space, a **`define** flag is set the first time the import statements are compiled.

In the following example, the `definitions` package is contained in a separate file, called `definitions.pkg` (Any file name and file extension could be used). After the endpackage keyword, the package is wildcard imported into the $unit compilation-unit space. In this way, when the package is compiled, the definitions within the package are automatically made visible in the current $unit space.

Within the `definitions.pkg` file, a flag is set to indicate when this file has been compiled. Conditional compilation surrounds the entire file contents. If the flag has not been set, then the package will be compiled and imported into $unit. If the flag is already set (indicating the package has already been compiled and imported into the current $unit space), then the contents of the file are ignored.

Example 2-6: Package with conditional compilation (file name: `definitions.pkg`)

```
`ifndef DEFS_DONE  // if the already-compiled flag is not set...
  `define DEFS_DONE  // set the flag
  package definitions;

    parameter VERSION = "1.1";

    typedef enum {ADD, SUB, MUL} opcodes_t;

    typedef struct {
      logic [31:0] a, b;
      opcodes_t    opcode;
    } instruction_t;

    function automatic [31:0] multiplier (input [31:0] a, b);
      // code for a custom 32-bit multiplier goes here
      return a * b;  // abstract multiplier (no error detection)
    endfunction
  endpackage

  import definitions::*;  // import package into $unit

`endif
```

The line:

```
`include "definitions.pkg"
```

should be placed at the beginning of every design or testbench file that needs the definitions in the package. When the design or testbench file is compiled, it will include in its compilation the package and import statement. The conditional compilation in the `definitions.pkg` file will ensure that if the package has not already been compiled and imported, it will be done. If the package has already been compiled and imported into the current $unit space, then the compilation of that file is skipped over.

For this coding style, the package file should be passed to the software tool compiler indirectly, using a **`include** compiler directive.

package compilation should be indirect, using `include

This conditional compilation style uses the Verilog **`include** directive to compile the `definitions.pkg` file as part of the compilation of some other file. This is done in order to ensure that the import statement at the end of the `definitions.pkg` file will import the package into the same $unit space being used by the compilation of the design or testbench file. If the `definitions.pkg` file were to be passed to the software tool compiler directly on that tool's command line, then the package and import statement could be compiled into a different $unit space than what the design or testbench block is using.

The file name for the example listed in 2-6 does not end with the common convention of `.v` (for Verilog source code files) or `.sv` (for SystemVerilog source code files). A file extension of `.pkg` was used to make it obvious that the file is not a design or testbench block, and therefore is not a file that should be listed on the simulator, synthesis compiler or other software tool command line. The `.pkg` extension is an arbitrary name used for this book. The extension could be other names, as well.

Examples 2-7 and 2-8 illustrate a design file and a testbench file that include the entire file in the current compilation. The items within the package are then conditionally included in the current $unit compilation-unit space using a wildcard import. This makes the package items visible throughout the module that follows, including in the module port lists.

Example 2-7: A design file that includes the conditionally-compiled package file

```
`include "definitions.pkg" // compile the package file

module ALU
(input  instruction_t  IW,
 input  logic          clock,
 output logic [31:0]   result
);
  always_comb begin
    case (IW.opcode)
      ADD : result = IW.a + IW.b;
      SUB : result = IW.a - IW.b;
      MUL : result = multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

Example 2-8: A testbench file that includes the conditionally-compiled package file

```
`include "definitions.pkg" // compile the package file

module test;
  instruction_t test_word;
  logic [31:0]  alu_out;
  logic         clock = 0;

  ALU dut (.IW(test_word), .result(alu_out), .clock(clock));

  always #10 clock = ~clock;

  initial begin
    @(negedge clock)
    test_word.a = 5;
    test_word.b = 7;
    test_word.opcode = ADD;
    @(negedge clock)
    $display("alu_out = %d (expected 12)", alu_out);
    $finish;
  end
endmodule
```

`include works with both sinlge-file and multi-file compilation

In a single file compilation, the package will be compiled and imported into each $unit compilation-unit. This ensures that each $unit sees the same package items. Since each $unit is unique, there will not be a name conflict from compiling the package more than once.

In a multiple file compilation, the conditional compilation ensures that the package is only compiled and imported once into the common $unit compilation space that is shared by all modules. Whichever design or testbench file is compiled first will import the package, ensuring that the package items are visible for all subsequent files.

The conditional compilation style shown in this section does not work with global variables, static tasks, and static functions.

package variables are shared variables (not synthesizable)

Packages can contain variable declarations. A package variable is shared by all design blocks (and test blocks) that import the variable. The behavior of package variables will be radically different between single file compilations and multiple file compilations. In multiple file compilations, the package is imported into a single $unit compilation space. Every design block or test block will see the same package variables. A value written to a package variable by one block will be visible to all other blocks. In single file compilations, each $unit space will have a unique variable that happens to have the same name as a variable in a different $unit space. Values written to a package variable by one design or test block will not be visible to other design or test blocks.

static tasks and functions in packages are not synthesizable

Static tasks and functions, or automatic tasks and functions with static storage, have the same potential problem. In multiple file compilations, there is a single $unit space, which will import one instance of the task or function. The static storage within the task or function is visible to all design and verification blocks. In single file compilations, each separate $unit will import a unique instance of the task or function. The static storage of the task or function will not be shared between design and test blocks.

This limitation on conditionally compiling import statements into $unit should not be a problem in models that are written for synthesis, because synthesis does not support variable declarations in packages, or static tasks and functions in packages (see section 2.1.3 on page 14).

2.2.5 Synthesis guidelines

The synthesizable constructs that can be declared within the compilation-unit scope (external to all module and interface definitions) are:

- **`typedef`** user-defined type definitions
- Automatic functions
- Automatic tasks
- **`parameter`** and **`localparam`** constants
- Package imports

using packages instead of $unit is a better coding style

While not a recommended style, user-defined types defined in the compilation-unit scope are synthesizable. A better style is to place the definitions of user-defined types in named packages. Using packages reduces the risk of spaghetti code and file order dependencies.

external tasks and functions must be automatic

Declarations of tasks and functions in the $unit compilation-unit space is also not a recommended coding style. However, tasks and functions defined in $unit are synthesizable. When a module references a task or function that is defined in the compilation-unit scope, synthesis will duplicate the task or function code and treat it as if it had been defined within the module. To be synthesizable, tasks and functions defined in the compilation-unit scope must be declared as **`automatic`**, and cannot contain static variables. This is because storage for an automatic task or function is effectively allocated each time it is called. Thus, each module that references an automatic task or function in the compilation-unit scope sees a unique copy of the task or function storage that is not shared by any other module. This ensures that the simulation behavior of the pre-synthesis reference to the compilation-unit scope task or function will be the same as post-synthesis behavior, where the functionality of the task or function has been implemented within the module.

A **`parameter`** constant defined within the compilation-unit scope cannot be redefined, since it is not part of a module instance. Synthesis treats constants declared in the compilation-unit scope as literal values. Declaring parameters in the $unit space is not a good modeling style, as the constants will not be visible to modules that are compiled separately from the file that contains the constant declarations.

2.3 Declarations in unnamed statement blocks

local variables in named blocks

Verilog allows local variables to be declared in named **begin**...**end** or **fork**...**join** blocks. A common usage of local variable declarations is to declare a temporary variable for controlling a loop. The local variable prevents the inadvertent access to a variable at the module level of the same name, but with a different usage. The following code fragment has declarations for two variables, both named `i`. The **for** loop in the named **begin** block will use the local variable `i` that is declared in that named block, and not touch the variable named `i` declared at the module level.

```
module chip (input clock);
  integer i;  // declaration at module level

  always @(posedge clock)
    begin: loop       // named block
      integer i;      // local variable
      for (i=0; i<=127; i=i+1) begin
        ...
      end
    end
endmodule
```

hierarchical references to local variables

A variable declared in a named block can be referenced with a hierarchical path name that includes the name of the block. Typically, only a testbench or other verification routine would reference a variable using a hierarchical path. Hierarchical references are not synthesizable, and do not represent hardware behavior. The hierarchy path to the variable within the named block can also be used by VCD (Value Change Dump) files, proprietary waveform displays, or other debug tools, in order to reference the locally declared variable. The following testbench fragment uses hierarchy paths to print the value of both the variables named i in the preceding example:

```
module test;
  reg clock;
  chip chip (.clock(clock));

  always #5 clock = ~clock;

  initial begin
    clock = 0;
    repeat (5) @(negedge clock) ;
    $display("chip.i = %0d", chip.i);
    $display("chip.loop.i = %0d", chip.loop.i);
```

```
    $finish;
  end
endmodule
```

2.3.1 Local variables in unnamed blocks

local variables in unnamed blocks

SystemVerilog extends Verilog to allow local variables to be declared in unnamed blocks. The syntax is identical to declarations in named blocks, as illustrated below:

```
module chip (input clock);
  integer i;  // declaration at module level

  always @(posedge clock)
    begin              // unnamed block
      integer i;       // local variable
      for (i=0; i<=127; i=i+1) begin
        ...
      end
    end
endmodule
```

Hierarchal references to variables in unnamed blocks

local variables in unnamed blocks have no hierarchy path

Since there is no name to the block, local variables in an unnamed block cannot be referenced hierarchically. A testbench or a VCD file cannot reference the local variable, because there is no hierarchy path to the variable.

named blocks protect local variables

Declaring variables in unnamed blocks can serve as a means of protecting the local variables from external, cross-module references. Without a hierarchy path, the local variable cannot be referenced from anywhere outside of the local scope.

inferred hierarchy paths fro debugging

This extension of allowing a variable to be declared in an unnamed scope is not unique to SystemVerilog. The Verilog language has a similar situation. User-defined primitives (UDPs) can have a variable declared internally, but the Verilog language does not require that an instance name be assigned to primitive instances. This also creates a variable in an unnamed scope. Software tools will infer an instance name in this situation, in order to allow the variable within the UDP to be referenced in the tool's debug utilities. Software tools may also assign an inferred name to an unnamed block, in order to allow the tool's waveform display or debug utilities to reference the local variables in that unnamed block. The SystemVer-

ilog standard neither requires nor prohibits a tool inferring a scope name for unnamed blocks, just as the Verilog standard neither requires nor prohibits the inference of instance names for unnamed primitive instances.

Section 7.7 on page 192 also discusses named blocks; and section 7.8 on page 194 introduces statement names, which can also be used to provide a scope name for local variables.

2.4 Simulation time units and precision

The Verilog language does not specify time units as part of time values. Time values are simply relative to each other. A delay of 3 is larger than a delay of 1, and smaller than a delay of 10. Without time units, the following statement, a simple clock oscillator that might be used in a testbench, is somewhat ambiguous:

```
forever #5 clock = ~clock;
```

What is the period of this clock? Is it 10 picoseconds? 10 nanoseconds? 10 milliseconds? There is no information in the statement itself to answer this question. One must look elsewhere in the Verilog source code to determine what units of time the `#5` represents.

2.4.1 Verilog's timescale directive

Verilog specifies time units to the software tool

Instead of specifying the units of time with the time value, Verilog specifies time units as a command to the software tool, using a **`timescale** compiler directive. This directive has two components: the time units, and the time precision to be used. The precision component tells the software tool how many decimal places of accuracy to use.

In the following example,

```
`timescale 1ns / 10ps
```

the software tool is instructed to use time units of 1 nanosecond, and a precision of 10 picoseconds, which is 2 decimal places, relative to 1 nanosecond.

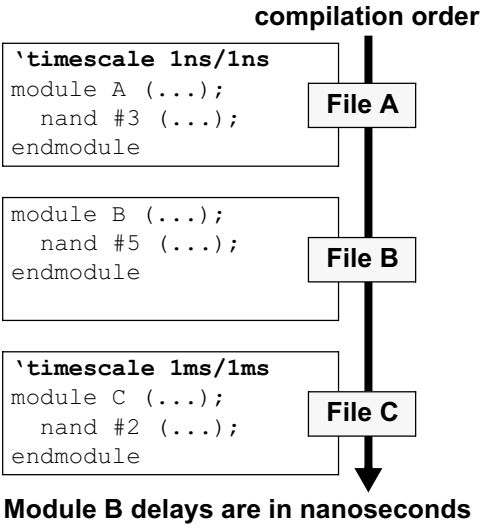
multiple 'timescale directives

The **`timescale** directive can be defined in none, one or more Verilog source files. Directives with different values can be specified for different regions of a design. When this occurs, the software tool must resolve the differences by finding a common denominator in all the time units specified, and then scaling all the delays in each region of the design to the common denominator.

the 'timescale directive is file order dependent

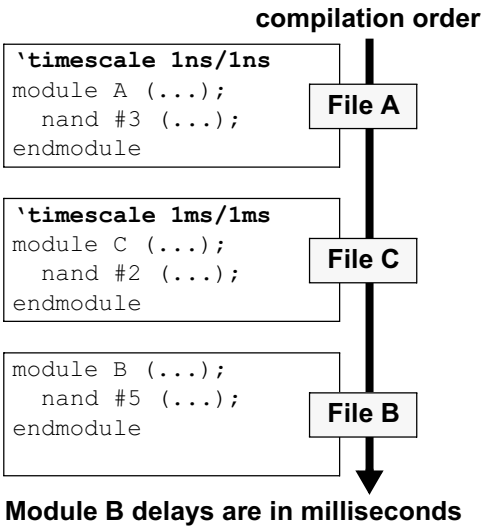
A problem with the **`timescale** directive is that the command is not bound to specific modules, or to specific files. The directive is a command to the software tool, and remains in effect until a new **`timescale** command is encountered. This creates a dependency on which order the Verilog source files are read by the software tool. Source files without a **`timescale** directive are dependent on the order in which the file is read relative to previous files.

In the following illustration, files A and C contain **`timescale** directives that set the software tool's time units and time precision for the code that follows the directives. File B, however, does not contain a **`timescale** directive.



If the source files are read in the order of File A then B and then C, the **`timescale** directive that is in effect when module B is compiled is 1 nanosecond units with 1 nanosecond precision. Therefore, the delay of 5 in module B represents a delay of 5 nanoseconds.

If the source files are read in by a compiler in a different order, however, the effects of the compiler directives could be different. The illustration below shows the file order as A then C and then B.



In this case, the **`timescale** directive in effect when module B is compiled is 1 millisecond units with 1 millisecond precision. Therefore, the delay of 5 represents 5 milliseconds. The simulation results from this second file order will be very different than the results of the first file order.

2.4.2 Time values with time units

time units specified as part of the time value

SystemVerilog extends the Verilog language by allowing time units to be specified as part of the time value.

```
forever #5ns clock = ~clock;
```

Specifying the time units as part of the time value removes all ambiguity as to what the delay represents. The preceding example is a 10 nanoseconds oscillator (5 ns high, 5 ns low).

The time units that are allowed are listed in the following table.

Table 2-1: SystemVerilog time units

Unit	Description
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	femtoseconds
step	the smallest unit of time being used by the software tool (used in SystemVerilog testbench clocking blocks)

No space is allowed between the time value and the time unit.

When specifying a time unit as part of the time value, there can be no white space between the value and time unit.

```
#3.2ps      // legal
#4.1 ps     // illegal: no space allowed
```

2.4.3 Scope-level time unit and precision

SystemVerilog allows the time units and time precision of time values to be specified locally, as part of a module, interface or program block, instead of as commands to the software tool (interfaces are discussed in Chapter 10 of this book, and program blocks are presented in the companion book, *SystemVerilog for Verification*[1]).

timeunit and timeprecision as part of module definition

In SystemVerilog, the specification of time units is further enhanced with the keywords **timeunit** and **timeprecision**. These keywords are used to specify the time unit and precision information within a module, as part of the module definition.

```
module chip (...);
  timeunit 1ns;
  timeprecision 10ps;
```

1. Spear, Chris *"SystemVerilog for Verification"*, Norwell, MA: Springer 2006, 0-387-27036-1.

```
  ...
endmodule
```

The **timeunit** and **timeprecision** keywords allow binding the unit and precision information directly to a a module, interface or program block, instead of being commands to the software tool. This resolves the ambiguity and file order dependency that exist with Verilog's **`timescale** directive.

The units that can be specified with the **timeunit** and **timeprecision** keywords are the same as the units and precision that are allowed with Verilog's **`timescale** directive. These are the units that are listed in table 2-1 on page 31, except that the special **step** unit is not allowed. As with the **`timescale** directive, the units can be specified in multiples of 1, 10 or 100.

The **timeunit** and **timeprecision** statements must be specified immediately after the module, interface, or program declaration, before any other declarations or statements.

timeunit and timeprecision must be first

The specification of a module, interface or program **timeunit** and **timeprecision** must be the first statements within a module, appearing immediately after the port list, and before any other declarations or statements. Note that Verilog allows declarations within the port list. This does not affect the placement of the **timeunit** and **timeprecision** statements. These statements must still come immediately after the module declaration. For example:

```
module adder (input  wire [63:0] a, b,
              output reg  [63:0] sum,
              output reg         carry);
  timeunit 1ns;
  timeprecision 10ps;
  ...
endmodule
```

2.4.4 Compilation-unit time units and precision

external timeunit and timeprecision

The **timeunit** and/or the **timeprecision** declaration can be specified in the compilation-unit scope (described earlier in this chapter, in section 2.2 on page 14). The declarations must come before any other declarations. A **timeunit** or **timeprecision**

declaration in the compilation-unit scope applies to all modules, program blocks and interfaces that do not have a local **timeunit** or **timeprecision** declaration, and which were not compiled with the Verilog **`timescale** directive in effect.

At most, one **timeunit** value and one **timeprecision** value can be specified in the compilation-unit scope. There can be more than one **timeunit** or **timeprecision** statements in the compilation-unit scope, as long as all statements have the same value.

Time unit and precision search order

time unit and precision search order

With SystemVerilog, the time unit and precision of a time value can be specified in multiple places. SystemVerilog defines a specific search order to determine a time value's time unit and precision:

- If specified, use the time unit specified as part of the time value.
- Else, if specified, use the local time unit and precision specified in the module, interface or program block.
- Else, if the module or interface declaration is nested within another module or interface, use the time unit and precision in use by the parent module or interface. Nested module declarations are discussed in Chapter 9 and interfaces are discussed in Chapter 10.
- Else, if specified, use the **`timescale** time unit and precision in effect when the module was compiled.
- Else, if specified, use the time unit and precision defined in the compilation-unit scope.
- Else, use the simulator's default time unit and precision.

backward compatibility

This search order allows models using the SystemVerilog extensions to be fully backward compatible with models written for Verilog.

The following example illustrates a mixture of delays with time units, **timeunit** and **timeprecision** declarations at both the module and compilation-unit scope levels, and **`timescale** compiler directives. The comments indicate which declaration takes precedence.

Example 2-9: Mixed declarations of time units and precision (not synthesizable)

```
timeunit 1ns;          // external time unit and precision
timeprecision 1ns;

module my_chip ( ... );

  timeprecision 1ps; // local precision (priority over external)

  always @(posedge data_request) begin
    #2.5 send_packet;  // uses external units & local precision
    #3.75ns check_crc; // specific units take precedence
  end

  task send_packet();
    ...
  endtask

  task check_crc();
  ...
  endtask
endmodule

`timescale 1ps/1ps  // directive takes precedence over external
module FSM ( ... );
  timeunit 1ns;     // local units take priority over directive

  always @(State) begin
    #1.2 case (State) // uses local units & timescale precision
      WAITE: #20ps ...;  // specific units take precedence
    ...
  end
endmodule
```

2.5 Summary

This chapter has introduced SystemVerilog packages and the $unit declaration space. Packages provide a well-defined declaration space where user-defined types, tasks, functions and constants can be defined. The definitions in a package can be imported into any number of design blocks. Specific package items can be imported, or the package definitions can be added to a design block's search path using a wildcard import.

The $unit declaration space provides a quasi global declaration space. Any definitions not contained within a design block, testbench block or package falls into the $unit compilation-unit space. Care must be taken when using $unit to avoid file order dependencies and differences between separate file compilation and multi-file compilation. This chapter provided coding guidelines for the proper usage of the $unit compilation-unit space.

SystemVerilog also allows local variables to be defined in unnamed **`begin`**...**`end`** blocks. This simplifies declaring local variables, and also hides the local variable from outside the block. Local variables in unnamed blocks are protected from being read or modified from code that is not part of the block.

SystemVerilog also enhances how simulation time units and precision are specified. These enhancements eliminate the file order dependencies of Verilog's **`` `timescale``** directive.

http://www.springer.com/978-0-387-33399-1

SystemVerilog for Design Second Edition
A Guide to Using SystemVerilog for Hardware Design and Modeling
Sutherland, S.; Davidmann, S.; Flake, P.
2006, XXX, 418 p., Hardcover
ISBN: 978-0-387-33399-1