cadence®

SystemVerilog Reference

Product Version 9.2

July 2010

Printed in the United States of America.

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Contents

1

Overview of SystemVerilog

The IEEE 1800 standard for SystemVerilog describes a large set of extensions to the existing IEEE Verilog-2001 standard. This set of enhancements provides new capabilities for modeling hardware at the RTL and system level, along with a powerful set of new features for verifying model functionality.

This reference guide tells you how to enable the SystemVerilog constructs, and describes the constructs in the IEEE 1800 standard that are supported by the simulator. For information about the simulator, refer to the *Verilog Simulation User Guide.*

Availability of Constructs within Simulators

Cadence offers several simulators. All of the features described in this book are supported within the Incisive Enterprise Simulator - XL (IES-XL). However, some of the constructs described in this document are not available with the Incisive Enterprise Simulator - L (IES-L). These constructs include classes, semaphores, program blocks, and clocking blocks.

SystemVerilog in Simulation

For information about how to simulate a design that contains SystemVerilog constructs, including information about how to view and debug SystemVerilog constructs using Tcl or the Cadence® Simulation Analysis Environment (SimVision), refer to *SystemVerilog in Simulation*.

SystemVerilog VPI Extensions

Because the SystemVerilog VPI standard is still evolving, Cadence does not support SystemVerilog VPI extensions in the current release.

Existing user and third-party PLI and VPI applications that already work for designs without SystemVerilog language extensions will continue to work on those designs. However, these existing applications might fail if applied to designs that contain SystemVerilog constructs.

SystemVerilog Assertions

Note: SystemVerilog assertions are available only if you have an Incisive license.

Support for SystemVerilog assertions is documented in the *Assertion Writing Guide* and the *SVA Quick Reference*.

SystemVerilog Coverage

Note: SystemVerilog coverage is available only if you have an Incisive license.

Support for SystemVerilog coverage is documented in the “Functional Coverage” chapter of the *ICC User Guide*.

SystemVerilog with AMS

You can simulate a design that contains both AMS and SystemVerilog code, but with the following limitations:

- If an AMS scope is instantiated inside a SystemVerilog scope, it cannot have any non-digital ports.
- If a SystemVerilog scope is instantiated inside an AMS scope, it cannot have non-digital objects connected to its ports.

You can use the multi-step invocation mode (*ncvlog*, *ncelab*, *ncsim*) to simulate mixed AMS/SystemVerilog designs by using separate invocations of *ncvlog* with the appropriate option, then compiling the SystemVerilog and AMS portions of the design.

SystemVerilog Examples

This document contains small examples for each of the supported SystemVerilog constructs. For complete examples, refer to the following:

- *SystemVerilog Engineering Notebook*—Examples of various SystemVerilog constructs. You can download the examples and run them using the simulator.
- *SystemVerilog DPI Engineering Notebook*—Examples of SystemVerilog DPI. You can download the examples and run them using the simulator.
- *Examples Reference Guide*—Lists the examples located within your installation.

Language Support

This document uses the following terms:

- *Verilog* or *Verilog-2001*—Refers to the IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language
- *IEEE 1800*—Refers to the IEEE 1800 Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language.

Getting Help

This section describes:

About Online Help

Documentation for SystemVerilog is provided in HTML and in PDF format.

The online documentation system consists of

- The Cadence documentation window

  This window lets you find and open any of the books shipped with the products that you ordered. You can list books by product name, by product family, or by document type. For example, you can list all manuals, all Product Notes documents, or all Known Problems and Solutions documents. When you select a document, it is opened in your Web browser. The system automatically starts the browser, if necessary.

- Manuals in HTML format

  Each HTML document has both hyperlinked cross-references and a toolbar with buttons that let you navigate through the documentation system. Using the buttons on the toolbar, you can redisplay the documentation window, move forward and backward through chapters, display the Table of Contents, open a PDF file for printing, or open the search page. You can also send an e-mail directly to Cadence publications with comments about the document that you are viewing.

- A PDF (Portable Document Format) file for each document, so you can print the entire document or sections of a document.
- A powerful search tool that lets you search for information in documents for a product family or for specific products. You can also search individual documents.

Click the *Help* button on the toolbar of any document to open the *Cadence Documentation User Guide*, which contains a complete description of how to use the online documentation system.

Invoking the Documentation System

There are two ways to open the online help for the simulator:

- From the Cadence documentation window.

  To invoke the Cadence documentation window on UNIX or Linux, use the Cadence Help command.

  ```
  % cdnshelp
  ```

  On the Cadence documentation window, click on a category name to show the documents in that category. Then double-click a manual title to load that manual into your web browser.

- From the Help menu on the graphical user interface.

  If you are using a graphical user interface, such as NCLaunch or the SimVision analysis environment, pull down the *Help* menu and select the name of the online manual that you want to view. For example, if you are simulating a Verilog design, select *Verilog Simulation User Guide* to open the online help for the simulator.

  Help can also be accessed from forms on the graphical interface. Click on the *Help* button on the bottom right of the form to get online help.

  To open the Cadence documentation window, click the Library button at the top of any document displayed in the browser.

Printing Documents From the Online Documentation System

The *View/Print PDF* button opens a PDF file in Adobe Acrobat® Reader. You can print the entire document or print a section of the document by specifying a range of page numbers.

A hyperlinked list of contents (a “bookmark” list) is available for navigating the print document online. Hyperlinked cross-references in the HTML version are reformatted to include page

numbers in the printed copy, so that you can find the referenced page easily when reading the printed version.

Searching Documents

The built-in search mechanism lets you search various groupings of books or individual books. You can:

- Search all installed documents.
- Search all documents for specific products or product families.
- Search one or more specific books.

The Search tool lets you perform many different types of full-text search queries. You can search for text phrases or exact words, use Boolean AND, OR, and NOT operators, use special operators such as CASE (for case-sensitive searches) or NEAR (to search for words near each other), or use wildcard characters for substitution.

Getting Help on Commands to Run Tools

You can display a list of options for any of the simulator tools and utilities by typing the tool or utility name followed by the `-help` option.

The `-help` option displays a list of the command options for the specified tool with a brief description of each option.

Syntax:

```
% tool_name -help
```

Examples:

```
% ncvlog -help
% ncvhdl -help
% ncelab -help
% ncsim -help
% ncupdate -help
```

Getting Help on Tool Messages

Use the *nchelp* utility to display extended help on the brief messages generated by the compiler, elaborator, and simulator.

Syntax:

```
% nchelp [options] tool_name message_code
```

You can enter the *message_code* argument in lowercase or uppercase.

Examples:

```
% nchelp ncvlog BADCLP
% nchelp ncvlog badclp
% nchelp ncelab cuvwsp
% nchelp ncsim NOSNAP
```

Other Documentation

A wealth of other documentation related to Cadence products is available on the Cadence Online Support website, a technical support service for Cadence software users. The service is available to all customers who have a software support services agreement. Cadence Online Support contains product information, datasheets, information about what's new in the latest release, application notes, white papers, information about Cadence services, such as training, customer support, and methodology services, and so on.

http://support.cadence.com

Customer Support

There are several ways that you can get help with your Cadence product:

- Customer support

  Cadence is committed to keeping your design teams productive by providing answers to technical questions, the latest software updates, and education services to keep your skills updated. For information about Cadence support, go to the following web site:

  http://www.cadence.com/support

- Cadence Online Support

  Customers with a maintenance contract with Cadence can obtain current information about the tools at the following web site:

  http://support.cadence.com

- Feedback about documentation

  Contact Cadence Customer Support to file a CCR if you find:

  - An error in the manual
  - An omission of information in a manual
  - A problem using the Cadence Help documentation system

2

Compiling SystemVerilog Constructs

This section describes the ways you can compile a SystemVerilog design using the simulator.

Using ncvlog

To compile a design that contains SystemVerilog constructs, use the `-sv` option with `ncvlog`:

```
% ncvlog -sv systemverilog_source_files
```

Note: SystemVerilog has added many new keywords. Therefore, Verilog legacy code that uses SystemVerilog keywords as identifiers will not compile if you use the `-sv` switch. In these cases, you can either

- Specify the set of reserved keywords in effect, using the SystemVerilog `` `begin_keywords `` and `` `end_keywords `` compiler directives
- Remove and restore specific keywords, using the compiler `ncvlog -rmkeyword` command-line option, or the Cadence `` `remove_keyword `` and `` `restore_keyword `` compiler directives

Refer to "`begin_keywords and `end_keywords" on page 292 and "`remove_keyword and `restore_keyword" on page 297 for more information.

See the *Verilog Simulation User Guide* for details about simulating Verilog designs.

Using the irun Utility

You can use the *irun* utility to run the simulator by specifying all input files and all command-line options on a single command line. The *irun* utility determines the language of a file by its extension, then maps the file to its appropriate compiler. For example:

```
% irun -linedebug -access rwc -gui vlogfile1.v vlogfile2.v systemv1.sv systemv2.sv
```

In this example, *irun* will compile the `.v` files using `ncvlog` and the `.sv` files using `ncvlog -sv`. After the input files have been compiled, *irun* automatically invokes *ncelab* to elaborate the design. In the example command line, the `-access` option is passed to the elaborator to

provide read access to simulation objects. After the elaborator has generated a simulation snapshot, `ncsim` is invoked with SimVision.

For each file type, there is a command-line option that you can use to change, or add to, the list of defined file extensions. For example, the default extensions for SystemVerilog files are `.sv`, `.svp`, `.SV`, and `.SVP`. You can add `.mysv` to the list of SystemVerilog file extensions by using the `-sysv_ext` option. For example:

```
irun top.sv dut.v test.mysv -sysv_ext .sv, .svp, .SV, .SVP, .mysv
```

When you use an extension option, the built-in defaults are removed, and you must specify all of the different extensions to be recognized.

Note: Compiling files using the `irun` utility is an alternative to using the `` `begin_keywords `` and `` `end_keywords `` directives when you want to distinguish Verilog files from SystemVerilog files. However, this utility does not automatically support files that contain a mixture of Verilog and SystemVerilog. For those cases, you can use the `` `begin_keywords `` and `` `end_keywords `` compiler directives within the `.sv` file, and compile the file using *irun*.

For more information about *irun*, refer to the *irun User Guide*. For information about using the *irun* utility with DPI, refer to "Using the irun Utility with DPI" on page 321.

SystemVerilog and the PLI tf_nodeinfo() Interface

The PLI `tf_nodeinfo()` interface is not compatible with SystemVerilog designs. Therefore, you cannot compile SystemVerilog designs with the `-nomempack` option, or elaborate SystemVerilog designs with the `-arr_access` option.

For more information about these options, refer to the *Verilog Simulation User Guide*.

3

List of Supported Constructs

Table 3-1 on page 25 lists the SystemVerilog constructs that are supported in the current release. This table summarizes the supported constructs, with a reference to the relevant section numbers in the IEEE 1800-2005 standard. Refer to the construct's relevant section within this book for information about what the current release supports, and any limitations that might apply.

Note: The section numbers shown in this table are the section numbers where most of the information in the LRM can be found. In many cases, a section can contain information related to more than one construct, and information related to a particular construct can be in more than one section.

Table 3-1 SystemVerilog Constructs Supported in the Current Release

IEEE 1800 Section	SystemVerilog Construct
3.3	Literal Value Assignments on page 31
3.5 and 10.3 (for time units) 19.10 (for `timeunit` and `timeprecision`)	Time Unit and Time Precision on page 33
4	Data Types Overview on page 37 ■ `const` ■ `logic`, `bit` ■ `byte`, `shortint`, `int`, `longint`
4.2	uwire Nets on page 77
4.6	Chandle Data Type on page 43
4.7	Strings on page 43
4.9	typedef Declaration on page 56
4.10	enum Data Type on page 60

Table 3-1 SystemVerilog Constructs Supported in the Current Release, *continued*

IEEE 1800 Section	SystemVerilog Construct
4.11	Structures on page 67 Unions on page 73
4.14	Static Casting on page 78
5	Arrays on page 83
5.2	Packed and Unpacked Arrays on page 84
5.5	Array Querying Functions on page 87
5.6	Dynamic Arrays on page 89
5.9	Associative Arrays on page 99
5.14	Queues on page 105
5.15.1	Array Manipulation Methods on page 114
6.6	Declaring Local Variables in Unnamed Blocks on page 126
6.7 and 11.5	Continuous Assignments to Variables on page 127
7	Classes on page 131 Classes are supported within the Incisive Enterprise Simulator - XL (IES-XL). However, classes are not available with the Incisive Enterprise Simulator - L (IES-L).
8	Operators and Expressions on page 147
8.2	Increment and decrement operators
8.3	Assignment Operators on page 148
8.5	Wild Equality and Wild Inequality Operators on page 148
8.13	Assignment Patterns on page 150
8.15	Aggregate Expressions on page 153
8.19	Set Membership Operator on page 149
10.10	`iff` event control qualifier

10.4	Unique and Priority Decision Statements on page 155
	`unique if, unique case`
	`priority if, priority case`

10.5.1	do...while Loop on page 158

Table 3-1 SystemVerilog Constructs Supported in the Current Release, *continued*

IEEE 1800 Section	SystemVerilog Construct
10.5.2	for Loop on page 158
10.5.3	foreach Loop on page 159
10.6	`return, break, and continue Jump Statements on page 162`
10.7	final Blocks on page 163
11.2, 11.3, and 11.4	`always_* Blocks on page 163`
11.6	fork...join on page 164
11.8.2	`disable fork on page 167`
11.8.1	`wait fork on page 167`
12.2 and 12.3	Multiple Statements in Tasks and Functions on page 171
12.3	Default Direction in Task and Function Declarations on page 172
12.3.2	Discarding Function Return Values
12.3	Function Output Arguments on page 171 `function func(input integer a, input integer b, output integer x, output integer y);`
12.3.1	Void Functions on page 173 `function void myprint (integer a);`
12.4.5	Optional Arguments for Tasks and Functions on page 178
12.4.4	Passing Task and Function Arguments by Name on page 177
12.4.2	Passing Task and Function Arguments by Reference on page 173
12.4.3	Specifying Default Argument Values for Tasks and Functions on page 177
13	Random Constraints on page 181
13.3	Random Variables on page 181
13.4	Constraint Blocks on page 184
13.5	Randomization Methods on page 192

Table 3-1 SystemVerilog Constructs Supported in the Current Release, *continued*

IEEE 1800 Section	SystemVerilog Construct
13.6	In-Line Constraints (randomize() with) on page 193
13.7	Activating and Inactivating Random Variables with rand_mode() on page 193
13.8	Activating and Inactivating Constraints with constraint_mode() on page 194
13.10	In-Line Random Variable Control on page 195
13.11	Randomizing Scope Variables (std::randomize()) on page 196
13.12	Random Number System Functions and Methods on page 200
13.13	Random Stability on page 203
13.15	Random Weighted Case (randcase) on page 203
13.16	Random Sequence Generator (randsequence) on page 204
14.2	Semaphores on page 209 Semaphores are supported within the Incisive Enterprise Simulator - XL (IES-XL). However, semaphores are not available with the Incisive Enterprise Simulator - L (IES-L).
14.3	Mailboxes on page 212 Mailboxes are supported within the Incisive Enterprise Simulator - XL (IES-XL). However, mailboxes are not available with the Incisive Enterprise Simulator - L (IES-L).
14.5.2	Non-Blocking Event Trigger on page 219
14.5.4	Persistent Trigger on page 220
14.7	Event Variables on page 221
15	Clocking Blocks on page 223 Clocking blocks are supported within the Incisive Enterprise Simulator - XL (IES-XL). However, clocking blocks are not available with the Incisive Enterprise Simulator - L (IES-L).

Table 3-1 SystemVerilog Constructs Supported in the Current Release, *continued*

IEEE 1800 Section	SystemVerilog Construct
16	Program Blocks on page 233 Program blocks are supported within the Incisive Enterprise Simulator - XL (IES-XL). However, program blocks are not available with the Incisive Enterprise Simulator - L (IES-L).
17	SystemVerilog Assertions Support for SystemVerilog assertions is documented in the *Assertion Writing Guide* and the *SVA Quick Reference*. **Note:** SystemVerilog assertions are available only if you have an Incisive license.
18	SystemVerilog Coverage Support for SystemVerilog coverage is documented in the "Functional Coverage" chapter of the *ICC User Guide*. **Note:** SystemVerilog coverage is available only if you have an Incisive license.
19	Hierarchy on page 243 ■ Packages ■ Port declarations ■ Compilation units
19.11.3	.name Implicit Port Connection on page 34
19.11.4	Dot Star (.*) Implicit Port Connection on page 35
19.4	Out-of-Module Reference ($root) on page 281
20	Interfaces on page 261
22.4	Expression Size System Function ($bits) on page 281
23.2	`define on page 291
23.4	`begin_keywords and `end_keywords on page 292
26	Direct Programming Interface on page 301

Table 3-1 SystemVerilog Constructs Supported in the Current Release, *continued*

IEEE 1800 Section	SystemVerilog Construct

10.8
(for `begin...end` and `fork...join`)
12.2 (for tasks)
12.3 (for functions)
7.2 (for classes)
15.2 (for clocking blocks)
16.2 (for program blocks)
19.2 (for packages)
19.5 (for modules)
20.2 (for interfaces)
	Matching End Names on page 31
	For example:
	`begin: blockA`
	`...`
	`end: blockA`

Not applicable (This is a Cadence extension)	`remove_keyword and `restore_keyword on page 297

4

Convenience Enhancements

Convenience enhancements are constructs that help make it easier to model in Verilog.

Literal Value Assignments

SystemVerilog lets you specify unsized literal single-bit values with a preceding apostrophe ('), but without the base specifier. This enhancement lets you fill a vector of any width with any logic value, without having to specify the vector size of the literal value. All bits of the vector on the left-hand side of the assignment are set to the specified value.

- `'0` – Fill all bits with `0`
- `'1` – Fill all bits with `1`
- `'Z` or `'z` – Fill all bits with `Z`
- `'X` or `'x` – Fill all bits with `X`

For example:

```
reg [127:0] data;

data = '1;    // Sets all bits of data to 1
data = 'z;    // Sets all bits of data to Z
data = 'X;    // Sets all bits of data to X
```

Matching End Names

SystemVerilog lets you specify a matching ending name for named blocks of code. This feature helps make the code more readable and easier to maintain.

The end name is preceded by a colon, and the specified name must be exactly the same as the name with which it is paired. A warning message is issued if the names are different.

You can specify a matching end name after the following keywords:

- `end`

```
begin : block_identifier
...
end : block_identifier
```

- join

```
fork : block_identifier
...
join : block_identifier
```

- endtask

```
task task_identifier
...
endtask : task_identifier
```

- endfunction

```
function function_identifier
...
endfunction : function_identifier
```

- endclass

```
class class_identifier
...
endclass : class_identifier
```

- endclocking

```
clocking clocking_identifier
. . .
endclocking: clocking_identifier
```

- endprogram

```
program program_identifier
. . .
endprogram: program_identifier
```

- endpackage

```
package package_identifier
. . .
endpackage: package_identifier
```

- endmodule

```
module module_identifier
...
endmodule : module_identifier
```

- endinterface

```
interface interface_identifier
...
endinterface : interface_identifier
```

Time Unit and Time Precision

In Verilog-2001, time values are specified with a number without a time unit. For example,

```
initial
  #5 clock = 1;

always
  #50 clock = ~clock;
```

The time unit and time precision can be specified with the `` `timescale `` compiler directive. You can specify this directive in one or more files, and you can specify directives with different time unit and time precision values for different modules in the design. When the source files are compiled and a `` `timescale `` directive is encountered, that directive remains in effect until another `` `timescale `` directive is encountered. Therefore, the time units and time precision that are used for a source file without a `` `timescale `` directive depend on the order in which the source files are compiled, which can cause simulation results to vary for different simulation runs.

SystemVerilog provides two enhancements to control the specification of time units for time values, which remove ambiguity and the file order dependency problem associated with the `` `timescale `` directive.

- A time unit can be specified with a time value.

  The time unit can be `s`, `ms`, `us`, `ns`, `ps`, or `fs`. There can be no whitespace between the time value and the time unit. For example:

  ```
  0.1ns
  40ps
  #10ns clock = ~clock;
  r = <= #1ns a;
  ```

  Note: The SystemVerilog LRM states that the time unit can also be `step`. This is not supported in the current release.

- Time units and time precision can be specified within a module with the keywords `timeunit` and `timeprecision`.

  As with the `` `timescale `` directive, the units that can be specified with the `timeunit` and `timeprecision` keywords are `s`, `ms`, `us`, `ns`, `ps`, or `fs`, and the units can be specified in multiples of 1, 10, or 100. There can be no whitespace between the time value and the time unit.

  These declarations can be specified within a module, package, or interface. The declarations must appear immediately after the module, package, or interface declaration. For example:

  ```
  module test #(parameter MSB = 3, LSB = 0)
               (output reg [MSB:LSB] x,
  ```

```
                input wire [MSB:LSB] a,
                input wire [7:0] b,
                input wire enable);

    timeunit 1ns;
    timeprecision 10ps;

    initial
      begin
        $display($time,,,"x=%d", x );
        #20 $finish;
      end

    endmodule
```

The scope of a `timeunit` or `timeprecision` declaration is limited to the design unit in which it is declared. There can be only one time unit and one time precision for the design unit. The `timeunit` and/or `timeprecision` declaration can be repeated as later items, but the values must match the original values exactly.

In the current release, the time unit and precision for a time value are determined according to the following search order:

1. Use the time unit specified as part of the time value.
2. Use the time unit and precision specified with the `timeunit` and `timeprecision` keywords.
3. Use the time unit and precision specified with the `` `timescale `` compiler directive that is currently in effect.
4. Use the simulator's default time unit and precision.

.name Implicit Port Connection

In Verilog-2001, you can instantiate a module using named port connections. The syntax requires you to specify the port name used in the module declaration, followed by the name used in the instantiating module. For example:

```
module top;
  wire data, clk, clr, q, qb;

  flop u1 (.data(data), .clock(clk), .clear(clr), .q(q), .qb(qb));
  ...
endmodule

module flop (input data, clock, clear, output q, qb);
  ...
  ...
endmodule
```

When you connect module instance ports in this way, the width of the port does not have to match the width of the net or variable connected to the port.

SystemVerilog simplifies this syntax for cases where the name of the port matches the name of the net or variable connected to the port. If the names match, and if the data types on each side of the port are compatible, you can specify only the port name. For example, in the code shown above, the `data` port is connected to the `data` net. This connection can be specified as `.data`.

Verilog named port connections must be used if the names do not match. Implicit `.name` port connections can be combined with named port connections. In the example above, the port connections can be written as follows:

```
flop u1 (.data, .clock(clk), .clear(clr), .q, .qb);
```

Note: The SystemVerilog LRM specifies that the size of the port must match the size of the net or variable connected to the port. In the Cadence implementation, this restriction is not imposed, but a warning is issued if the size of the net or variable does not match the size of the port.

As in Verilog-2001, you cannot mix positional port connections and named port connections in a module instantiation.

Dot Star (.*) Implicit Port Connection

In addition to using `.name` implicit port connections in the port list of a module instantiation (see ".name Implicit Port Connection" on page 34), SystemVerilog also provides the `.*` construct to further simplify the syntax for connecting ports by name.

The `.*` implicit port connection is used in the port list of a module instantiation as a shorthand for named port connections. Each unconnected port in the module definition is connected to a variable, wire, or interface with the same name declared in the instantiating module.

As with `.name` implicit port connections, the name of the port must match the name of the net or variable connected to the port, and the data types connected together must be compatible. Verilog-2001 named port connections must be used for any connections that cannot be inferred by `.*`.

Note: The SystemVerilog LRM specifies that the size of the port must match the size of the net or variable connected to the port. In the Cadence implementation, this restriction is not imposed, but a warning is issued if the size of the net or variable does not match the size of the port.

Note: In the current release, the following restrictions apply to the use of `.*` implicit port connections:

- `.*` cannot be used inside a generate block.
- `.*` can be used only in the instantiation of a Verilog or SystemVerilog module or interface. For example, the construct cannot be used in the instantiation of an analog block, a VHDL block, or a SystemC® block.

The following example uses the `.*` syntax in the module instantiation:

```
module top;
  wire data, clk, clr, q, qb;

  flop u1 (.*, .clock(clk), .clear(clr));
  ...
endmodule

module flop (input data, clock, clear, output q, qb);
  ...
  ...
endmodule
```

You can use only one `.*` token in the port list. When the implicit `.*` port connection is mixed with named port connections, as in the example shown above, or with `.name` implicit port connections, you can place the `.*` token anywhere in the port list.

As in Verilog-2001, you cannot mix positional port connections with named port connections in a module instantiation.

5

Data Types

Data Types Overview

In Verilog-2001, all logic values manipulated during simulation are 4-state. That is, the logic values in the simulation belong to the set `0`, `1`, `X` (unknown), and `Z` (high impedance). The logic value represented by a variable or net always belongs to this set of four values at any time.

Verilog 2001 defines the following data types for storing integers:

Data type	Description	Default
`reg`	4-state, user-defined vector size	`unsigned`
`integer`	32-bit 4-state integer	`signed`
`time`	64-bit 4-state integer	`unsigned`

SystemVerilog introduces one new 4-state data type (`logic`) and several 2-state data types. In the current release, the following 4-state and 2-state data types have been implemented:

Data type	Description	Default
`logic`	1-bit 4-state integer, user-defined vector size See "logic Data Type" on page 40.	`unsigned`
`bit`	1-bit 2-state integer, user-defined vector size See "bit Data Type" on page 41.	`unsigned`
`byte`	8-bit 2-state integer or ASCII character See "byte, shortint, int, and longint Data Types" on page 42.	`signed`

Data type	Description	Default
`shortint`	16-bit 2-state integer	`signed`
`int`	32-bit 2-state integer	`signed`
`longint`	64-bit 2-state integer	`signed`

Overview of Verilog Data Types

Verilog data objects have two attributes:

- The *kind* of the object (parameter, variable, or net)

  The object kind indicates what you can do with the object. For example, only parameters can be modified with `defparam` statements, and only variables can be assigned by procedural assignments.

  Note: The simulator supports the `defparam` statement only for parameters and `defparam` expressions that are legal in Verilog. The `defparam` statement is not supported for SystemVerilog data types.

- The *data type* of the object (integer, real, scalar bit, bit vector, and so on)

  The data type of an object indicates the values that the object can take. For example, an object of type `real` can take the value `3.14`, while an object of a bit vector type can take the value `4'b0xz1`.

These two object attributes are largely orthogonal. For example, a net can be of almost any data type, and a bit vector can be the data type of almost any kind of object.

A data type is a set of values. The Verilog data types fall into two groups:

- Primitive Data Types, whose values cannot be defined in terms of other values or data types.
- User-Defined Data Types, whose values are constructed from other values or data types.

Primitive Data Types

Verilog data types include a small number of *primitive* data types that serve as a basis for the value system. The values of a primitive data type cannot be defined in terms of other values or data types.

Verilog-2001 has two primitive data types: a 4-state bit type, and a real type. SystemVerilog introduces a name, `logic`, for the existing 4-state data type. SystemVerilog also introduces a 2-state bit type called `bit`. The extended language has three primitive data types: `logic`, `bit`, and `real`. These primitive data type names are all reserved words.

The following variable declarations use the primitive data type names:

```
real realvar;      // A real-valued variable
logic logicvar;    // A 4-state variable
bit bitvar;        // A 2-state variable
```

See "logic Data Type" on page 40 for details about the `logic` data type. See "bit Data Type" on page 41 for details about the `bit` data type.

User-Defined Data Types

In addition to primitive data types, the data type system includes data types that have values constructed from other values. These data types are called *user-defined* data types because you must describe how the values are constructed. For example, a Verilog bit vector is a user-defined data type because you must include a range to indicate the number of bits and how to index the vector.

You can describe the characteristics of a user-defined data type directly in an object declaration. For example, the following declarations define two 8-bit-wide 4-state variables called `byte_var1` and `byte_var2`:

```
logic [7:0] byte_var1;
logic [7:0] byte_var2;
```

SystemVerilog enhances the language by introducing the `typedef` declaration, which you can also use to describe the characteristics of a user-defined data type. A `typedef` declaration gives the data type a name that you can use in other declarations. The following example shows how to define the same two variables using a `typedef` declaration:

```
typedef logic [7:0] bits8;    // The name of this data type is bits8
bits8 byte_var1;
bits8 byte_var2;
```

See "typedef Declaration" on page 56 for details about the `typedef` declaration.

Some user-defined data types are so common that they have been given special predefined status in the language. Their names are reserved words, and you can use them without specifying the construction of their values. The Verilog-2001 predefined data types are `integer`, `time`, and `realtime`. The `integer` data type represents a 4-state 32-bit signed integer. SystemVerilog introduces new predefined data types, `byte`, `shortint`, `int`, and `longint` to represent 2-state signed integers of 8, 16, 32, and 64 bits, respectively. See

"byte, shortint, int, and longint Data Types" on page 42 for more information about these new data types.

logic Data Type

In Verilog-2001, all logic values manipulated during simulation are 4-state. That is, the logic values in the simulation belong to the set `0`, `1`, `X` (unknown), and `Z` (high impedance). The logic value represented by a variable or net always belongs to this set of four values at any time.

SystemVerilog gives this 4-state data type a name: `logic`. This keyword is simply the name of the 4-state bit type; it does not imply an object kind. The `logic` keyword can be used in any context in which a data type is allowed when you want to declare a 4-state object. For example, `logic` is used to explicitly state the data type in the following parameter, variable, and net declarations:

```
parameter logic p = 1'b0;     // 1-bit wide 4-state parameter
logic v;                      // 1-bit wide 4-state variable
logic [63:0] v2;              // 64-bit wide 4-state variable
wire logic w;                 // 1-bit wide 4-state net
```

These declarations are equivalent to the following Verilog-2001 declarations:

```
parameter p = 1'b0;
reg v;
reg [63:0] v;
wire w;
```

You can also use the `logic` data type on ports. Output ports that are declared as a `logic` data type are considered variables by default. In the following example, the `answer` port is declared as a `logic` data type, and is considered a variable by default.

```
module mymod(result, one, two);
  output logic result;
  input wire one;
  input wire two;

  assign result = one & two;
endmodule
```

Note: When you use the `logic` data type on an output port, a variable is added to its connections. The added variables are then subject to continuous assignments. The continuous assignments for these added variables can affect optimization, and can cause performance degradation, depending on the number of levels in your design, and the number of continuous assignments. To prevent the addition of variables to a port's connections, you can declare the port to also be a wire. For example:

```
...
output wire logic inputA;
...
```

Refer to "Continuous Assignments to Variables" on page 127 for more information about continuous assignments.

logic and reg Data Types

In Verilog-2001, the `reg` keyword is special, in that it implies both a data type (4-state logic) and an object kind (variable). Unlike other keywords that imply data type, such as `integer`, `reg` cannot be used to declare a parameter or a net.

In SystemVerilog, `reg` is a data type with the same semantics as `logic`. Both keywords specify that an object is 4-state, without saying anything about what kind of object it is. That is, `reg` does not imply that an object is a variable, as opposed to a net or parameter. For example, the following declarations are supported:

```
reg [31:0][7:0] mdv;
typedef reg signed [31:0] myIntT;
struct packed { reg [1:9] m1; reg m2; } packedStructVar;
enum reg [1:0] {high, moderate, low, off} eVar = off;

function reg func(input x);
      ...
endfunction

parameter reg x = 0;    // Same as: parameter logic x = 0;
module m(inout reg x);  // Module port is a net, not a variable
```

Note: The IEEE 1800 standard states that the `reg` keyword cannot immediately follow a net type keyword, such as `wire` or `tri`. The `reg` keyword can be used in a net or port declaration if there are lexical elements between the net type and `reg` keywords. For example:

```
wire reg x;               // Illegal--reg follows net type wire
inout tri reg y;          // Illegal--reg follows net type tri
wor scalared reg[3:0] z;  // Legal--scalared separates wor and reg
```

In these cases, use the `logic` keyword instead of `reg`.

bit Data Type

SystemVerilog adds support for a 2-state logic data type called `bit`. This keyword is simply the name of the 2-state bit type. As with `logic`, it does not imply an object kind. The `bit` data type differs from `logic` in that `bit` only stores the 2-state values of 0 and 1.

You can declare parameters and variables to be of type `bit`. Nets cannot be declared as `bit`.

The `bit` data type is used in exactly the same way the `logic` type is used. For example:

```
parameter bit p = 0;     // Parameter p can only be 0 or 1
bit v;                   // 1-bit wide 2-state variable
bit [63:0] v2;           // 64-bit wide 2-state variable
```

Treatment of 2-state objects varies, depending on the object kind:

- `bit` parameters are initialized to the value specified in the parameter declaration. You can override the initial value in the same way that 4-state parameters are overridden.
- `bit` variables are initialized to `0`.
- It is possible to assign an `X` or a `Z` to a `bit` variable; `X` and `Z` values are converted to 0.

byte, shortint, int, and longint Data Types

In Verilog-2001, the `integer` data type is a 32-bit vector of 4-state values that holds signed integer numbers. The Cadence data type extensions allow `integer` to be used as a general-purpose data type. You can use it in any context in which a data type is allowed. For example:

```
wire integer w;      // wire of type integer
input integer p;     // input port of type integer
```

SystemVerilog introduces new 2-state data types for storing integer numbers:

- `byte`

  A vector of eight 2-state bit values for holding either a signed integer number or a single ASCII character.

- `shortint`

  A vector of 16 2-state bit values for holding a signed integer number.

- `int`

  A vector of 32 2-state bit values for holding a signed integer number.

- `longint`

  A vector of 64 2-state bit values for holding a signed integer number.

Nets cannot be declared as a 2-state data type.

A variable declared as a 2-state data type is initialized to `0`.

When an object declared as a 2-state data type is assigned a value, any `X` or `Z` values in the elements of the new value are converted to `0`.

The following are some example declarations using the new data types:

```
int v;                  // A variable of type int
input int p2;            // An input port of type int
parameter int w2 = 289; // A parameter of type int
shortint errors;        // A variable of type shortint
parameter longint reset_count = 0; // A parameter of type longint

enum byte {steady, rising, falling} barometer; // An 8-bit 2-state enumeration
                                               // variable
```

Chandle Data Type

SystemVerilog adds a special `chandle` data type, which is used to store and pass C or C++ pointers in and out of DPI imported or exported tasks and functions. The syntax for a `chandle` declaration is as follows:

```
chandle variable_name;
```

where *`variable_name`* is a valid identifier. The `chandle` data types are initialized to the value `null`, with a value of zero on the C side.

For example, the following uses `chandle` data types to pass C pointers as arguments to imported DPI functions:

```
import "DPI-C" function chandle func10_dpi_ch ( inout chandle hndl);
```

Note: A `chandle` pointer cannot be relocated by the simulator, because it points directly into user memory. When you restart a snapshot, `chandle` pointers restore the value they had when the snapshot was last saved. This result might cause unexpected behavior if your DPI code tries to deference a `chandle` pointer after a restore operation, because the `chandle` pointer might have changed since the last save operation. A warning message is issued when you try to restore a snapshot that contains `chandle` pointers.

Note: In the current release, you cannot declare a `chandle` inside an unpacked structure.

Strings

SystemVerilog introduces a `string` data type, which represents a variable-length text string. The syntax for a string data type is as follows:

```
string string_identifier [= initial_value];
```

where the *`initial_value`* can be a string literal or an empty string `""`.

The `string` data type has the following characteristics:

- A string literal can be assigned to a `string` data type.

- The length of the `string` data type can vary during simulation.

  When a string literal is assigned to an integral variable or an unpacked array of bytes of a different size, the string literal is truncated. The `string` data type eliminates this situation—when a value is assigned to a `string` variable, its length adjusts accordingly.

- A single character of a `string` variable is of type `byte`.
- The indexes of a `string` variable are numbered from `0` to *`N`*`-1`, where *`N`* is the length of the string. The `0` corresponds to the first character of the string, and `N-1` corresponds to the last character of the string.
- If a string literal contains the special "`\0`" character, this character is ignored.

String Operators

The following table describes the SystemVerilog string operators that are supported in the current release.

Operator	Description
`s1 == s2`	The *equality* operator checks whether two strings are equal.
`s1 != s2`	The *inequality* operator checks whether two strings are different.

`s1 < s2`
`s1 <= s2`
`s1 > s2`
`s1 >= s2`
	The *comparison* operators use the `compare()` string method. If the given condition is true, these relational operators return 1. Both operands can be of type `string`, or one of them can be a string literal.

`{s1, s2,...,sN}`	The *concatenation* operator can take `strings` or string literals. If at least one operand is of type `string`, the expression evaluates to the concatenated string and is of type `string`. If all operands are string literals, the expression behaves like a Verilog concatenation of integral types. If the result is then used in another expression that involves `string` types, the expression is converted to type `string`.

Operator	Description
`{multiplier{s1}}`	The *replication* operator replicates a string by the number of times specified by `multiplier`. The `s1` value must be a `string` or string literal. The `multiplier` must be an integral type, and can be constant or non-constant: ■ If `multiplier` is non-constant or `s1` is a string type, the result is a string containing N concatenated copies of `s1`, where N is the `multiplier`. ■ If `multiplier` is constant and `s1` is a literal, the expression behaves like a numeric replication in Verilog. If the result is used in another expression involving string types, it is implicitly converted to a string type.
`s1[index]`	The *indexing* operator returns the ASCII code for the given index. If the given index is out of range, the operator returns 0.

String Methods

SystemVerilog provides built-in methods for working with strings. The current release supports the following string methods:

Method	Syntax	Description
`len()`	`str.len()`	Returns the length of the specified string
`putc()`	`str.putc(i, c)`	Replaces the `i`th character of the string with the given integral value `c`
`getc()`	`str.getc(i)`	Returns the ASCII code of the `i`th character of the specified string
`toupper()`	`str.toupper()`	Returns the uppercase of the specified string
`tolower()`	`str.tolower()`	Returns the lowercase of the specified string
`itoa()`	`str.itoa(i)`	Stores the ASCII representation of `i` into the string
`atoi()`	`str.atoi()`	Returns the integer corresponding to the ASCII decimal representation in `str`

Method	Syntax	Description
atobin()	str.atobin()	Returns the binary value corresponding to the ASCII binary representation in str
atohex()	str.atohex()	Returns the hexadecimal value corresponding to the ASCII hexadecimal representation in str
atooct()	str.atooct()	Returns the octal value corresponding to the ASCII octal representation in str
hextoa()	str.hextoa(i)	Stores the ASCII hexadecimal representation of the ith character of the specified string
octtoa()	str.octtoa(i)	Stores the ASCII octal representation of the ith character of the specified string
bintoa()	str.bintoa(i)	Stores the ASCII binary representation of the ith character of the specified string
compare()	str.compare(s)	Compares str and s
icompare()	str.icompare(s)	Compares str and s, but the comparison is case-insensitive
atoreal()	str.atoreal()	Returns the real number corresponding with the ASCII decimal representation in str
realtoa()	str.realtoa(r)	Stores the ASCII real representation of r into str
substr	str.substr(i, j)	Returns a new string that is a substring formed using the characters in positions i and j of str If i < 0, j < i, or j >= str.len(), this method returns an empty string.

Example 5-1 Using String Methods and Operators

The following example illustrates the supported string methods and operations.

```
module top;

// Declares a string data type.
string mystring = "hello world";
string newstring= "he\0llo big world"; // The \0 character will be ignored.
string string_pi = "3.1415";
real real_pi;
int i;
```

```
initial begin
   $display ("Value of string is: %s", mystring);
   i = mystring.len();                // Returns the length of the string.
   $display("%d characters long", i);

   // Displays the ASCII code for the given characters.
   $display("%s", mystring.getc(0));
   $display("%s", mystring.getc(1));
   $display("%s", mystring.getc(2));
   $display("%s", mystring.getc(3));
   $display("%s", mystring.getc(4));

   // Changes the string to uppercase
   mystring = mystring.toupper();
   $display ("My new string in uppercase: %s", mystring);

   // Displays the string in lowercase
   $display("My new string in lowercase: %s", mystring.tolower());
   $display("First string is:%s", mystring);
   $display("Second string is:%s", newstring);

   // Compares two strings
    if (newstring == mystring)
     $display("The strings are the same.");
    else
     $display("These strings are not the same.");
    if (newstring != mystring)
     $display("Again, they are not the same.");
    else
     $display("These strings are the same.");

    // Uses the indexing operator to replace characters
    mystring[0] = "Y";
    mystring[5] = "W";
    $display("New string %s", mystring);

    // substr() method extracts "WORLD" from "YELLOWWORLD"
    $display("Short string is %s", mystring.substr(6,10));

    // Converts string to a real value
    real_pi = string_pi.atoreal();
    $display("String pi is %s", string_pi);
    $display("Real pi is %f", real_pi);

end

endmodule
```

This example produces the following output:

```
Value of string is: hello world
11 characters long
 h
 e
 l
 l
 o
My new string in uppercase: HELLO WORLD
My new string in lowercase: hello world
```

```
First string is: HELLO WORLD
Second string is: hello big world

These strings are not the same.
Again, they are not the same.

New string YELLOWWORLD

Short string is WORLD

String pi is 3.1415
Real pi is 3.141500
```

Strings and System Tasks

The following system tasks have been enhanced so they can accept `string` variables as arguments: `$display`, `$write`, `$strobe`, `$monitor`, `$fdisplay`, `$fwrite`, `$fstrobe`, `$fmonitor`, `$sscanf`, and `$fopen`.

Using Strings with Classes

The current release supports strings within classes as

- Default class members (public or automatic)
- Static, local, or protected class members
- Local and global constant class members

You can perform the following operations on a string that is a member of a class:

- Initializing the string within a constructor
- Passing the string as an argument to an automatic or static function or task that is within the class. The actual string can be a member of the same class, or can be declared outside the class.
- Using the string with any of the supported operators and methods. See "String Operators" on page 44 and "String Methods" on page 45.
- Using the string as an argument to system tasks and functions
- Using the string as the return type for an automatic or static function that is declared inside the class
- Using the string with the `this` and `super` keywords
- You can declare a function that returns the string as `extern`.

- A virtual function can return the string and take the string as an argument.

The following example illustrates how to use strings within a class.

Example 5-2 Using Strings with Classes

```
module top;
class demo_class;
  static string s; // Static class member
  string s1;
  string s2;
  string s3;
  function new();
    s1 = "first_string"; // Initialized within a constructor
    s2 = "second_string";
  endfunction
  function string concat_string(); // Function that returns a string
    return {s1.toupper(),"",s2.toupper()}; //Uses string methods and operators
  endfunction
  task format_string(string s1); // String passed as an argument
   $sformat(s, "Output of %s", s1); // String as an argument of a system task
  endtask
endclass
demo_class dc;
initial begin
  dc = new;
  dc.s3 = {dc.s1,"", dc.s2};
  if (dc.s3.toupper() == dc.concat_string())
    $display("Correct");
  else
    $display("Not Correct");
  dc.format_string("format_string");
  $display(dc.s);
end
endmodule
```

This code will produce the following output:

```
Correct
Output of format_string
```

Using Strings with Packages

A string can be declared as a public, static, local, or protected member of a class that is declared inside a package. The following is supported for strings that are members of a class declared within a package:

- Initializing the string inside a constructor
- Passing the string as an argument to a task or function inside the class
- Using the string as an argument to system tasks and functions
- Applying the `this` and `super` keywords to this type of string

- A virtual function can return this type of string and take the string as an argument
- Using any of the supported operators and methods on this type of string; see "String Operators" on page 44 and "String Methods" on page 45
- Declaring a function that returns this type of string within a class

Using Strings within begin...end Blocks

Strings can be declared as static within `begin...end` blocks for the following types of statements:

`initial`	`always`	`case`, `casex`, and `casez`
`repeat`	`while`	`forever`
`for`	`fork...join`	`do...while`

You can perform the following operations on strings that are declared within a `begin...end` block:

- Assigning a string literal or string variable to another string
- Passing a string as an argument to tasks or functions
- Using the string with any of the supported operators and methods. See "String Operators" on page 44 and "String Methods" on page 45.

Declaring a Fixed Array of Strings

You can declare a fixed array of type string within the following scopes: modules, program blocks, tasks, functions, packages, and classes.

You can perform the following operations on the index of a fixed array of strings:

- Assigning the string index to a string literal or string variable
- Using the string index as an argument to system tasks or functions
- Passing the string index as an argument (`input`, `output`, or `inout`) to a user-defined task or function
- Using the string index with any of the supported operators and methods; see "String Operators" on page 44 and "String Methods" on page 45

The following example illustrates how to use a fixed array of strings.

Example 5-3 Fixed Array of Strings

```
module top;
  string s1[10];
  string s = "one";
  function string func(string s2);
   return s2.substr(1,4);
  endfunction
initial begin
  #0;
  s1[0] = "zero"; // String literal is initialized to the index of the string
array
  s1[1] = s; // String variable is initialized to the index of the string array
  $display("Print index %c", s1[0][1]); // Uses the indexing operator
  if(s1[0]==s1[1]) // Uses the equality operator
    $display("Incorrect");
  else
    $display("Relational operator is working correctly");
  s1[2] = {s1[0],"",s1[1]}; // Uses concatenation operator
  $display("Print concatenation result", s1[2]);
  $display("Length of first index:%d", s1[1].len()); // Uses the len method
  s1[3] = s1[2].toupper(); // Uses the toupper method
  $display("Print toupper", s1[3]);
  s1[4] = func(s1[3]); // Index of string array is passed as an argument to a
                       // function that returns a string that is assigned to the
                        // index of a string array.
  $display("After function return", s1[4]);
  s1[5] = "hi";
  $sformat(s1[5],"%s","hello"); // Index of string array as an argument to a
                                // system task
  $display("after sformat", s1[5]);
end
endmodule
```

This example will produce the following output:

```
Print index e
Relational operator is working correctly
Print concatenation result zero one
Length of first index: 3
Print toupper ZERO ONE
After function return ERO
After sformat hello
Index 3 changed
```

Declaring Arrays and Queues of Strings

You can declare fixed arrays, dynamic arrays, queues, and associative arrays of type `string` within module, program block, task, function, package, and class scopes.

You can perform the following operations on the index of an array or queue of strings:

- Assigning the string index to a string literal or string variable

- Using the string index as an argument to system tasks or functions
- Passing the string index as an argument (`input`, `output`, or `inout`) to a user-defined task or function
- Using the string index with any of the supported operators and methods; see "String Operators" on page 44 and "String Methods" on page 45

An element of an array or queue of strings is a string. The current release supports the same functionality for strings as for the elements of an array or queue of strings. In turn, whatever is not supported for strings is also not supported for elements of an array of strings. This rule applies to only the *elements* of an array or queue of strings. The *entire* array or queue is subject to the same limitations as defined for that type of array. See "Arrays" on page 83.

Using Elements of a Dynamic Array of Strings

An element of a dynamic array of strings is a string. The current release supports the same functionality for strings and for elements of a dynamic array of strings. In turn, whatever is not supported for strings is also not supported for elements of a dynamic array of strings.

Note: This applies only to elements of a dynamic array of strings. The entire dynamic array, however, is subject to the same limitations as defined for dynamic arrays. See "Arrays" on page 83.

For example, for the following dynamic array of strings:

```
string dyn_arr[];
string s1;
...
dyn_arr = new[10];
...
```

You can perform the following operations on this array, because they are supported for the `string` data type.

- Bit select on an element of a dynamic array of strings

  ```
  dyn_arr[0] = "abc";
  ```

- Pass it as an `inout`/`input`/`output` parameter to a function

  ```
  func(dyn_arr[0])
  ```

- Assign an element of the dynamic array of strings to a string

  ```
  s1 = dyn_arr[0];
  ```

- Functions can return an element of a dynamic array of strings

  ```
  s1 = func(dyn_arr[0]);
  ```

- Event controls are supported for elements of a dynamic array of strings

```
always @ (dyn_arr[0])
```

Using Strings as Parameters and localparams

A string can be declared as a `parameter` or `localparam` inside a module. In the current release, a string parameter can be

- Assigned while instantiating a module
- Assigned to other string variables
- An argument to system tasks and functions
- Passed as an argument to user-defined functions and tasks
- Used with the supported String Operators and String Methods—except for the `itoa()`, `hextoa()`, `octtoa()`, and `bintoa()` methods, because it is illegal to change string parameter values at simulation time

Example 5-4 Using String Parameters

```
module sub;
  parameter string test = "hello"; // Declares a string parameter
  string test1;
  function string func(string s);
    func = s;
    $display(func);
  endfunction
  initial begin
   $display(test);
   test1 = test; // String parameter is assigned to a string variable
   $display(test1);
   if (test1 == test) // Equality operator
   $display("correct");
   test1 = {test, "hello"}; // Concatenation operator
   $display(test1);
   $display("byte : %c",test[1]); // Indexing operator
   $display("len : ",test.len()); // len method
   $display("getc : ",test.getc(0)); // getc method
   $display("toupper : ",test.toupper()); // toupper method

   if (!test1.compare(test)) // compare method
     $display("correct");
     $display("atoi : ", test.atoi()); // atoi method
     test1 = func(test); // string parameter passed as an argument
     $display(test1);
     test1 = "";
     $sformat(test1,"%s",test); // String parameter passed as an argument to
                                // sformat
     $display(test1);
  end
endmodule

module top;
```

```
  sub #("hi")t1();
endmodule
```

This example produces the following simulation results:

```
hi
hi
correct
hihello
byte : i
len : 2
getc : 104
toupper : HI
atoi :   0
hi
hi
hi
```

Using Out-of-Module References to Strings

Out-of-module references (OOMRs) to strings are supported in the following cases:

- OOMR to a string in an assignment statement
- OOMR to a string in a relational expression
- OOMR to a string in a conditional concatenation expression
- OOMR to a string in a `case` statement

For example:

```
module x ();
  string s, s1, s2, s3, s4, s5;
  function automatic string fs(input string s1, output string s2, inout string
    s3, ref string s4);
    s2 = s1;
    s3 = s2.toupper();
    s4 = s3;
    s4[0] = s1[0];
    s4 = {s4,"-----"};
    return s3;
  endfunction
endmodule

module y (input var string s1, output var string s2);
  initial begin
    #0;
    $display(s1);
    s2 = "changed";
  end
endmodule

module test;
  x xx1();
  y yy1(xx1.s1, xx1.s5);
  initial begin
```

```
    xx1.s = "hello";
    xx1.s1 = xx1.s;
    if (xx1.s1==xx1.s)
      $display ("correct");
    void'($sscanf(xx1.fs(xx1.s,xx1.s1,xx1.s2,xx1.s3),"%s", xx1.s2));
    $display(xx1.s,xx1.s1, xx1.s2, xx1.s3, xx1.s4);
    xx1.s4 = xx1.s1 != xx1.s3 ? xx1.s1 : xx1.s2;
    $display(xx1.s4);
    case (xx1.s1)
      "HELLO" : begin
        $display("incorrect");
      end
      xx1.s4 : begin
        $display("correct");
      end
      default $display("No match for any case!!!!");
    endcase
    #1;
    $display(xx1.s5);
  end
endmodule
```

The output from this code looks like the following:

```
ncsim> run
correct
hellohelloHELLOhELLO-----
hello
correct
hello
changed
```

Limitations:

- Built-in methods are not supported for OOMRs to strings.
- OOMRs to strings that are members of a class are not supported

Limitations on Strings

The following summarizes the features in the LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Declaring the `string` data type, or assigning a `string` literal or `string` variable to a `string`, is supported within a module, program block, task, function, package, or class. The current release does not support strings within unpacked structures.

  Strings within unpacked structures are supported.

- You cannot use string indexing on class selects that are arguments to system tasks or functions. For example, where `c1` is a class:

  ```
  $sformat(str,"%x", c1[2].s); // Invalid
  ```

- If a string is a member of a dynamic or fixed array of a class, you cannot use it as an argument to a system task or function. For example:

```
c c1[10];
$sformat(str, "%s", c1[2].s); // Invalid
```

- Strings cannot be declared in a `for` loop that is inside a `generate` statement. For example:

```
module test;
  integer j=0;
  genvar i;

  generate
    for(i=4; i>1; i=i-1) begin : gen1
      typedef logic [0:7] w_type;

      logic [7:0] A;
      string B ; // Invalid

    end
  endgenerate
...
endmodule
```

- You cannot use string members of a class in port mapping or with the scope resolution operator.
- Strings are not supported within parameterized mailboxes.
- String variables cannot be passed by reference to a task or function.

typedef Declaration

A `typedef` declaration gives a data type a name that you can use in other declarations. The advantage of using a `typedef` is that you can characterize the data type in a single place, then refer to that description in as many other declarations as you like. By referring to a common definition for the data type, you can guarantee that a set of objects share the same data type characteristics, such as width, array size, and signedness. Another common use for naming data types is to provide descriptive information for the object declarations that use them. This technique can help to make the code more self-documenting and easier to read.

Limitations on Typedefs

This section summarizes the features in the SystemVerilog standard that are not supported in the current release.

- The LRM states that a type can be used before it is defined. For example:

```
typedef foo;
```

```
foo f = 1;
typedef int foo;
```

This form is not supported.

- The LRM allows references to type identifiers defined within an interface through ports, provided that they are locally redefined before being used. This feature is not supported.

Creating a New Data Type Definition

A `typedef` declaration begins with the keyword `typedef`. The nature of the rest of the declaration depends on what kind of data type you want to declare. The syntax is essentially the same as that of an object declaration, with the data type name appearing in the location where you normally put the object name.

In the following example, a type named `bits8` is used to declare two variables with the same data type characteristics—4-state, 8 bits wide. In this example, the name `bits8` is a synonym for `logic [7:0]`.

```
typedef logic [7:0] bits8;    // The name of this data type is "bits8"
    bits8 byte_var1;
    bits8 byte_var2;
```

The following declarations illustrate the syntax for different kinds of data types:

```
// A scalar type called logic_type
typedef logic logic_type;

// An unsigned vector type called vector_type
typedef logic [31:0] vector_type;

// A signed vector type called signed_vector_type
typedef logic signed [31:0] signed_vector_type;

// An array type called array_type
typedef logic [7:0] array_type [15:0][15:0];
```

A `typedef` declaration does not create a new data type. It simply introduces a new name for a data type. The data type itself is defined by the form of its values and the associated operators. The fact that a data type has a name does not imply any special treatment. The same compatibility and conversion rules apply to its values.

In the Cadence implementation, you can use a data type name in the declaration of parameters, variables, nets, ports, functions, and tasks. The following code fragments show where the data type name appears in the different kinds of Verilog declarations:

```
// The name of this 32-bit bit vector data type is addressT
typedef logic [31:0] addressT;

// Variable
addressT v1;
```

```
// Net
wire addressT w1;

// Port
inout addressT p1;

// Parameter
parameter addressT default_value = 32'h00000000;

// Task or function argument
input addressT value_in;

// Function
function addressT checksum;
```

Example 5-5 Defining Shared Data Type Characteristics

In this example, a `typedef` declaration is used to define a set of shared data type characteristics in a single place. The data type name is then used in other declarations and in another `typedef` declaration.

```
localparam BUS_WIDTH = 32;    // Address and data are the same size
localparam DMA_BURST = 4;     // How many data words are transferred

// Create a name, busT, for data and address buses
typedef logic [BUS_WIDTH:1] busT;

// Declare two variables of type busT
busT IOaddress_reg, IOdata_reg;

// Declare two nets of type busT
wire busT IOaddress, IOdata;

// Declare a data type called dmaT, constructed from type busT
typedef busT dmaT [DMA_BURST-1:0];
```

The shared width characteristics of the data and address buses are captured in the data type definition of `busT`. This data type name is then used to declare two variables, `IOaddress_reg` and `IOdata_reg`; two nets, `IOaddress` and `IOdata`; and another data type, the array type `dmaT`.

Example 5-6 Providing Descriptions for Object Declarations

In this example, the data type names declared in `typedef` declarations do not define any data type characteristics, but provide descriptive information for the object declarations that use them.

```
typedef integer flagsT;
typedef flagsT maskT;

typedef enum maskT {
    SIGBUS  = 32'h00000001,
    SIGSEGV = 32'h00000002,
    SIGTRAP = 32'h00000004,
    SIGILL  = 32'h00000008,
```

```
    SIGFPE  = 32'h00000010
} flag_bitT;

task set_flag ( inout flagsT flags, input flag_bitT flag_bit );
    flags = flags | flag_bit;
endtask

task clear_flag ( inout flagsT flags, input flag_bitT flag_bit );
    flags = flags & ~flag_bit;
endtask
```

These declarations define a set of flag bits and utility routines for setting and clearing a given flag bit. The `flagsT` data type is simply another name for `integer`, and the `maskT` data type is simply another name for `flagsT`, so there are three different names for the same data type.

The new names provide additional descriptive information for the object declarations that use them. These distinct names are used as the enumeration base type and the task argument types, to make the semantic nature of these items very clear.

Handling Data Type Visibility

Like macros, data type names must be declared before they are used. A data type declaration can appear outside of a module declaration, or in any location in which an object declaration is allowed.

The information about a data type—that is, the information in its declaration—must be known at the time when you compile the declarations that use it. Because hierarchical references cannot be resolved until the design hierarchy has been created, you cannot use a hierarchical name to refer to a data type.

The following visibility rules apply to data type names:

- If a data type name is declared inside a block, the name is visible only within that block.
- If a data type name is declared inside a module, the data type name is visible only within that module.
- If a data type name is declared outside of a module, the data type declaration is treated lexically in the order in which it is encountered in the description. That is, the data type name is visible in any module that follows the data type declaration.

  The following example shows a data type that is declared outside of a module so that it can be used in declaring and instantiating a module with ANSI-style parameter declarations:

```
typedef enum { FALSE, TRUE } booleanT;

module test #( parameter booleanT do_random_test = FALSE );
    ...
```

```
    ...
endmodule

module top;
    test #(TRUE) t1 ();
    ...
    ...
endmodule
```

Like other forms of declaration, the name of a data type must be unique in the immediate scope in which it is declared. Two data type declarations that introduce the same name within the same scope are not allowed, even if the data types they describe are identical. This is true even if the declarations appear in the scope by including the same file with `` `include ``. Two data type declarations introducing the same name might appear in separate compilation units.

enum Data Type

SystemVerilog introduces enumerated data type declarations. Enumerated data types let you declare a variable that has a list of valid values. Each value in the list has an associated user-defined name. You can use these meaningful names to specify the values that the variable can have.

Using enumerations can make the code easier to read and debug. Like constant names defined with `` `define `` macros, you can use the constant names in your Verilog code. However, unlike `` `define `` macros, you can also see the values as names in the Waveform Viewer, use the constant names in Tcl commands, and see the values as names in the output of Tcl commands.

With SystemVerilog enumerations, you can also use a shorthand that allows a sequence of enumeration values to be defined using a simple array-like notation. You might find it convenient in defining the enumeration values in an enumeration data type. For example, a declaration as the following:

```
typedef enum { a[3] } et;
```

contains the enumerated type range `a[3]`. This enumeration declaration is equivalent to the longer:

```
typedef enum { a0, a1, a2 } et;
```

Similarly, the declaration:

```
typedef enum { a[4:2] } et;
```

is equivalent to:

```
typedef enum { a4, a3, a2 } et;
```

If there is initialization in the context of this shorthand, it applies to the first enumeration value. For example:

```
typedef enum { a[3] = 5 } et;
```

is equivalent to:

```
typedef enum { a0 = 5, a1, a2 } et;
```

Example 5-7 Using Enumeration Declaration Shorthand

SystemVerilog Code:

```
module top;
  typedef enum { x[2], y[3:1] = 12 } t;
  t v;
  int i;
  initial  begin
    $display("Values for enumeration type t...");
    for (i = 1, v = v.first(); i <= v.num(); v = v.next(), i++)
      $display("  %s with the value %0d", v.name(), v);
  end
endmodule
```

Produces the following output:

```
Values for enumeration type t...
  x0 with the value 0
  x1 with the value 1
  y3 with the value 12
  y2 with the value 13
  y1 with the value 14
```

Limitations on Enumerations

This section summarizes the features in the SystemVerilog standard that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Hierarchical references to enumeration constants are not supported.
- The LRM states that if a value is explicitly specified for an enumeration constant, the value is a constant expression that can include references to parameters, local parameters, genvars, other named constants, and so on. In the current release, the constant expression is restricted to be a simple number, optionally preceded by a +/-.

Declaring an Enumeration

The definition of an enumerated data type is introduced by the keyword `enum`. The definition includes a list of the enumeration constants, and, optionally, a data type and/or vector width.

As with all data types, you can define an enumeration in an object declaration, as shown in the following examples:

```
enum { clear, warning, error } status;
enum { clear, warning, error } status = clear;
enum int { overflow, underflow, io_error } error_codes;
enum logic [1:0] { overflow, underflow, io_error } error_codes;
```

You can also define an enumeration in a `typedef` declaration. For example:

```
typedef enum { overflow, underflow, io_error } error_codes_t;
typedef enum bit { FALSE, TRUE } boolean;
typedef enum bit { FALSE=1'b0, TRUE=1'b1 } boolean;
typedef enum logic [2:0] { MOVop, SUBop, ANDop, ADDop, ORop, LDop, XORop }
opcode_t;
```

Depending on how you declare an enumeration, it is a bit or a bit vector.

- If you do not include a range or a data type name, the base type defaults to `int`. For example, the following declaration

  ```
  typedef enum { overflow, underflow, io_error } error_codes_t;
  ```

  is equivalent to

  ```
  typedef enum int { overflow, underflow, io_error } error_codes_t;
  ```

- If you specify the `logic` or `bit` type without a range, as in the following declaration, the enumeration is a scalar.

  ```
  typedef enum bit { FALSE, TRUE } boolean;
  ```

- If you specify a range, as in the following declaration, the enumeration is a vector.

  ```
  typedef enum logic [2:0] { MOVop, SUBop, ANDop, ADDop, ORop, LDop, XORop }
  opcode_t;
  ```

The following code fragments show different ways to declare the `error_code` enumeration as a 32-bit signed bit vector:

```
// The default is int
typedef enum { OVERFLOW, UNDERFLOW, MEMLIMIT, BUSERR } error_code;

// Explicitly including the data type name
typedef enum int { OVERFLOW, UNDERFLOW, MEMLIMIT, BUSERR } error_code;

// With a bit type, range, and signed specification
typedef enum logic signed [31:0] { OVERFLOW, UNDERFLOW, MEMLIMIT, BUSERR }
        error_code;
```

```
// With another user-defined data type
typedef logic signed [31:0] my_integer;
typedef enum my_integer { OVERFLOW, UNDERFLOW, MEMLIMIT, BUSERR } error_code;
```

Specifying Enumeration Constants

Enumeration constants are names for a set of constant values. These constants belong to the same namespace as the enclosing declaration. This means that if two enumeration declarations are visible in the same place, all of their enumeration constant names must be unique.

The value represented by a name in the enumeration list is the same type as the enum type itself, which defaults to `int`. The first name in the list has a value of `0`, the second name has a value of `1`, the third name has a value of `2`, and so on. For example, in the following enumeration, the enumeration constants `suspended` and `active` have the values `0` and `1`, respectively.

```
enum { suspended, active } process_status;
```

You can explicitly declare a value for names in the enumerated list. For example:

```
typedef enum logic [2:0] {WAIT=3'b001, LOAD=3'b010, READY=3'b100} states_m;
```

If you do not specify a value for a name, the value of the previous name in the list is incremented by one. In the following example, the value of `A` is `1`, the value of `B` is `2`, the value of `C` is `5`, and the value of `D` is `6`.

```
enum {A=1, B, C=5, D} enum_list;
```

Each name in the list must have a unique value. For example:

```
enum {E=1, F, G=2, D} enum_list2;
```

causes the following compilation error, because both `B` and `C` have the value `2`:

```
ncvlog: *E,SVEDUP (test.sv,8|14): Enumeration constant 'G' has the same value
as enumeration constant 'F'. For a given enumeration type, each enumeration
constant must have a unique value.
```

If the data type is a 4-state type, you can assign the values `X` or `Z` to a name. For example:

```
enum logic {H=0, I=1, J=1'bx, D=1'bz} enum_list3;
```

If you assign the value `X` or `Z` to a name, the following name must have an explicit value assigned. For example, the following is an error because `D` does not have an explicit value.

```
enum logic {K=0, L=1, M=1'bx, D=1'bz} enum_list4;
```

Treating Enumeration Objects as Bit Vectors

Enumeration data types are a special form of bit or bit vector. That is, depending on how you declare the enumerated data type, the enumeration objects are bits or bit vectors. For example, in the following declaration, the enumeration constants are scalars.

```
enum logic { OVERFLOW, UNDERFLOW } error_code;
```

In the following declaration, the enumeration constants are vectors.

```
typedef enum logic [2:0] {
        idle = 3'b001,
        read_cycle = 3'b010,
        write_cycle = 3'b100 } fsm_states;

fsm_states state = idle;
```

All vector operations and semantics apply to vector enumeration objects. For example, enumeration objects that are vectors can be the subject of bit-selects and part-selects. The normal semantics for initialization, assignment (except for type checking, see "Enumeration Type Checking"), and value conversion (sign extension and truncation) also apply. For example, the variable `state`, in the example shown above, will start simulation as all X's, and transition to the value of `idle` in the first simulation cycle.

The value of an enumeration constant with an explicit encoding is determined by the standard vector assignment semantics, as though the value were assigned to an object of that data type. If there is a width mismatch between the value and the vector width, truncation or extension of the value will occur. An error is issued only when an application of the assignment changes the value. For example:

```
enum logic[1:0]{
  WAIT = 1,
  LOAD = 2,
  READY = 4} stateA;
```

In this example, the values of `WAIT` and `LOAD` do not cause an error message because they can be represented in two bits (unsigned). However, the value of `READY` cannot be represented in this way, which causes the following compilation error message for this enumeration constant:

```
ncvlog: *E,SVECTR (test.sv,14|10): Truncation occurred converting the
enumeration constant expression for 'READY' into a value of this enumeration
type.
```

Enumeration Type Checking

SystemVerilog enumeration data types are strongly typed, in that when an `enum` object is assigned a value, the data type of that value must match the data type of the `enum` object. For example:

```
enum {s, m, l} sizes;
typedef enum {red, green, blue } colorT;
colorT color;
int i;
initial begin
 color = green;  // Valid. Both have the same data type.
 color = m; // Invalid. Data type mismatch.
...
end
```

When an `enum` object is used in an expression, it is automatically converted to the underlying numeric type for the `enum`. For example:

```
i = green + 1; // green is automatically converted to int 1, so i = 2.
```

An `enum` variable cannot be assigned a value that is outside the enumeration set, unless you use an explicit cast. For example:

```
color = colorT'(1); // Valid. Static cast converts 1 to colorT.
```

Enumeration Type Methods

SystemVerilog provides a set of methods that you can use to display information about an enumeration.

Method	Description
`first()`	Returns the value of the first member in an enumeration.
`last()`	Returns the value of the last member in an enumeration.
`next()`	Returns the value of the next member in the enumeration, based on the given value. If the given value is the last member in the enumeration, returns the value of the first member. If the given value is not a member of the enumeration, returns the default initial value. This method also takes an optional parameter, which indicates how many values to go forward.

Method	Description
`prev()`	Returns the value of the previous member in the enumeration, based on the given value. If the given value is the first member in the enumeration, returns the value of the last member. If the given value is not a member of the enumeration, returns the default initial value. This method also takes an optional parameter, which indicates how many values to go backward.
`num()`	Returns the number of elements in an enumeration.
`name()`	Returns the string representation of the given enumeration value. If the given value is not a member of the enumeration, returns the default initial value.

For example:

```
module top;

typedef enum byte {seattle, austin, calcutta = 15, louisville, london = 100 }
Cities;

Cities city = city.first();

initial
 forever begin
 $display("%n has the internal value %d", city, city);
 if (city == city.last()) break;
 city = city.next();
end

initial begin
 $display("Total number of cities is %d", city.num());
 city = calcutta;
 $display("%n + 2 is %n", city, city.next(2));
 city = louisville;
 $display ("%n - 3 is %n", city, city.prev(3));
end
endmodule
```

This code produces the following output:

```
seattle has the internal value    0
austin has the internal value    1
calcutta has the internal value   15
louisville has the internal value   16
london has the internal value  100
Total number of cities is           5
calcutta + 2 is london
louisville - 3 is seattle
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

In the current release, the following features are not supported:

- Cascaded enumeration methods are not supported. For example:

```
city = city.first().next(); // Not supported
...
city = city.first(); // Use this code instead
city = city.next();
```

- Enumeration methods are not supported for array elements or members of a structure. For example:

```
Cities cityArray[10];
city = cityArray[2].next(); // Not supported

...
city = cityArray[2]; // Use this code instead
city = city.next();
```

- Enumeration methods are not supported for hierarchical references or OOMRs. For example, the following code

```
module m;
  enum {cold, hot} temp = hot;
endmodule

module top;
  m i();
  initial $display(i.temp); // OOMR to enum variable is supported
  initial $display(i.temp.first()); // enum method on OOMR not supported
endmodule
```

  causes the following error message:

```
ncelab: *E,NLMETH (./test.sv,8|30): Built-in method calls are not presently
supported on non-local objects (i.e., referenced with hierarchical names),
array elements or structure members).
```

Structures

Verilog does not have a convenient way to group related data objects under a common name. SystemVerilog introduces C-like *structures* to Verilog that can group related data objects, where each member in the structure is called a *field*. Fields can be standard data types, such as `time`, `int`, and `logic`; user-defined data types; or other structures.

Structures are declared using the `struct` keyword, and a simple structure declaration looks like the following:

```
struct {
structure_fields;
...
} structure_name;
```

In SystemVerilog, there are two kinds of structures: *packed* and *unpacked*. By default, structures are *unpacked*. You can explicitly declare a structure as *packed* by using the

`packed` keyword. A packed structure consists of bit fields that are packed together in memory without gaps. The following lists some of the differences between packed and unpacked structures:

- Packed structures, unlike unpacked structures, can be treated as normal vectors. The first member in a packed structure has the most significance, while the following members have decreasing significance. Packed structures can be indexed like normal vectors—for example, `packedStruct[3]`. Packed structures can also be used as a whole, with arithmetic and logical operators.
- Packed structures can have only integral values that can be represented as a vector, such as `int` and `byte`. Packed structures cannot contain unpacked structures, `real` or `shortreal` variables, unpacked unions, or unpacked arrays. SystemVerilog cannot pack a structure if any of the fields cannot be represented as a vector. These restrictions do not apply to unpacked structures.

You can reference the whole group of fields by using the structure's name, or access a member of the structure by using its field name. Structure members are accessed as follows:

```
<structure_name>.<field_name>
```

For example, the following assigns a value of 0 to `field1` in structure `test1_var`:

```
test1_var.field1 = 0;
```

Structures can contain data objects of different types and sizes. For example, the following illustrates a simple declaration for a packed structure:

```
typedef struct packed{
   integer field1;
   logic[7:0] field2;
}test1;
```

Packed Structures

The Cadence implementation supports the following functionality for packed structures:

- Packed structures that can be applied to variables, parameters, and nets. For example:

```
test1 test1_var;

parameter test1 test1_parameter = 2;

wire test1 test1_wire;
```

- SystemVerilog unpacked arrays of packed structures
- Type parameter members
- SystemVerilog packed arrays of packed structures. For example:

```
wire struct packed { logic [2:0][1:0] a1; } [7:0] myArray;
```

- Mixing 2-state and 4-state packed structures. If any data type within a packed structure is 4-state, the whole structure is 4-state. When reading 2-state members, there is an implicit conversion from 4-state to 2-state. When writing 2-state members, there is an implicit conversion from 2-state to 4-state. For example:

```
struct packed { logic m1; bit m2; } v;
```

- Assigning patterns to members of packed structures. See "Assignment Operators" on page 148 for information about assignment patterns.

Limitations on Packed Structures

This section summarizes the features in the SystemVerilog standard that are not supported in the current release for packed structures. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Out-of-module references to members of a packed structure are not supported. For example, given the following code, an out-of-module reference of the form `top.mystruct.member` is not supported.

```
module top;
  struct packed {
    int member;
  } mystruct;
endmodule
```

Declaring a Packed Structure

Packed structures are defined by using the `packed` keyword. For example:

```
struct packed {
    logic m1;
    logic m2;
    } packed_test;
```

You can create a data type from a structure by using the `typedef` keyword. When you declare a structure as a user-defined type, storage is not allocated. For fields within a structure to store values, you must declare a variable of that data type. For example, the previous example can look like this:

```
// Structure definition
typedef struct packed {
    logic m1;
    logic m2;
    } p_test;

// Variable of the data type
p_test packed_test;
```

You can define a structure where its fields are other structures. For example:

```
struct packed {
    p_test x;
    p_test y;
} coordinate;
```

Unpacked Structures

The Cadence implementation supports the following functionality for unpacked structures:

- Unpacked structure members with the following data types:
  - Any data type allowed for members of a packed structure
  - The `real` data type
  - The `string` data type
  - Unpacked structures
  - Type parameters
  - A fixed-size unpacked array data type
- Variables of an unpacked structure type. For example:

  ```
  struct { integer i1, i2; } mystruct;
  ```
- Member selection for unpacked structure variables—for example, `struct.`*`member`*
- Unpacked structures as task and function arguments and function return types
- Unpacked structure assignments for variables

  These variable assignments are subject to the type-compatibility requirements enforced by the SystemVerilog LRM. For example:

  ```
  // Unpacked structure data types
  typedef struct { real x; real y;} cartesianCoordinateT;
  typedef struct { real magnitude; real angle; } polarCoordinateT;

  // Variables of the unpacked structure data type
  cartesianCoordinateT cartesian;
  polarCoordinateT polar1, polar2;

  // Conversion function
  function polarCoordinateT convert(input cartesianCoordinateT x);
  ...
  endfunction

  polar1 = polar2;               // Supported
  polar1 = cartesian;            // Causes compilation error due to type
  mismatch
  polar1 = convert(cartesian);   // Supported
  ```
- Assignment patterns to members of packed structures; for example:

```
struct { integer age, weight; }
       alfred = '{weight: 155, age: 36 },
       ella =   '{27, 155},
       ginger = '{age:27, default:155};
```

See "Assignment Operators" on page 148 for more information about assignment patterns.

- Conditional, logical equality (==), and case equality (===) operators on unpacked structures; for example:

```
struct { byte b; real r; } v1, v2, v3;
...

// Assigns v1 to either v2 or v3, depending on the how v1 compares to v3
using
// logical equality
v1 = (v1 == v3) ? v2 : v3;
```

- Arrays of unpacked structures; for example:

```
typedef struct{
   integer length, width, depth;
} dimensionsT;

dimensionsT arrayOfDimensions[250]; // Supported
```

Using Unpacked Structures in Classes

In the current release, unpacked structures are supported within classes. The following summarizes the supported functionality:

- Unpacked `struct` variables can be declared within classes, including classes that are declared within packages or modules.
- User-defined unpacked `struct` types can be used within a class.
- Unpacked `struct` members can be declared within a class.
- You can assign a class `struct` member to another class `struct` member of the same type:

```
C cc1, cc2;
cc1.st1 = cc2.st1;
```

- Within a class, you can declare a function that returns a `struct` and that has `input`, `output`, `inout`, or `ref` arguments that are structures.
- You can use the logical equality operators, `==` and `!=`, with class `struct` variables:

```
if (cc.s1 == cc1.st1)
```

- Unpacked structures can be declared as static members of a class:

```
class C;
  static struct {
    ...
  }st1;
endclass
```

- Unpacked structures can be declared as public, local, or protected members of a class.
- You can use type parameters as struct members:

```
class c#(type t1 = bit, type t2 = bit);
  typedef struct {
    t1 m1;
    t2 m2;
  } st;
  ...
endclass
```

- You can nest unpacked structures within a class:

```
class C;
  struct {
   int a;
   struct {
     real b;
   }st2;
  }st1;
endclass
```

Limitations on Unpacked Structures

The following list summarizes the features in the SystemVerilog standard that are not supported in the current release.

- The following data types are not supported for members of an unpacked structure:
  - `string`
  - associative arrays
  - dynamic arrays
  - classes
  - queues
  - events
- The Cadence implementation does not support initializing members of an unpacked structure. For example:

```
typedef packed {
   logic [3:0] ab1 = 4'b0101;  // Not supported
   }myInitType;
myInitType myInitVar;
```

- Nets of an unpacked structure data type are not supported.

```
wire struct {integer i1, i2; } wu; // Not supported
```

- Parameters of an unpacked structure data type are not supported.

```
parameter struct {integer a1, a2; } ab; // Not supported
```

- Unpacked structure variables are not supported within constant functions.
- Net ports that have an unpacked structure data type are not supported.

```
module m3(i1, i2, i3, o1, o2, o3);
   typedef struct {integer m1; logic [7:0] m2; } upsT;
   input upsT i1;       // Not supported, i1 is a net port
   input wire upsT i2;  // Not supported, i2 is a net port
   input var upsT i3;   // Supported, i3 is a variable port
   output upsT o1;      // Supported, o1 is a variable port
   output wire upsT o2; // Not supported, o2 is a net port
   output var upsT o3;  // Supported, o3 is a variable port
endmodule
```

- Out-of-module references to an unpacked structure variable, or to members of an unpacked structure, are not supported, including unpacked structures that are declared within classes.
- Unpacked structures are not supported within packages.
- Non-blocking assignments are not supported on class objects that contain unpacked structures. Non-blocking assignments are also not supported on unpacked structures or unpacked structure members that are elements of a class.
- Unpacked `struct` members cannot be used in sensitivity lists:

```
always @ (cc.st1)
```

- You cannot use class struct variables with relational operators (<, <=, >, >=).

```
if (cc.st1 < cc1.st1)
```

- You cannot use the `$bits()` function on class `struct` variables.

Debugging Structures

For information about how to debug structures using the Tcl command-line interface or the SimVision analysis environment, refer to *SystemVerilog in Simulation*.

Unions

The SystemVerilog *union* data type is similar to the C union data type. A union is a storage element; it is a collection of variables of different types and sizes. A SystemVerilog union is

similar to a SystemVerilog structure in that it can be packed or unpacked, it can have members, and its members are accessed in the same way as structures. However, members of a union share the same storage space, which means a union can store values of different types at different times. The same stored value can be viewed as if it was a different type by accessing the value through different members.

Unions are declared using the `union` keyword. A simple union declaration looks like the following:

```
union [tagged] [packed [signing]] {union_members} union_name;
```

For example:

```
union {
  int a;
  bit [52:0] b;
} myunion;
```

Note: The result for unpacked unions might not match that of a packed union, or an unpacked union from another tool.

Limitations on Unions

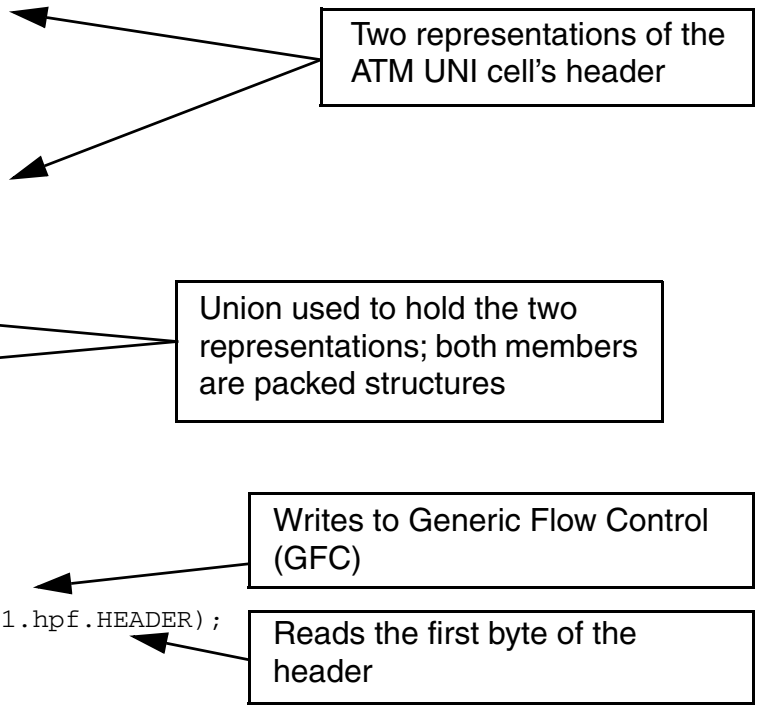
The current release supports packed and unpacked unions as described in the LRM, with the following exceptions:

- Tagged unions are not supported.
- Unpacked unions cannot be passed using the direct programming interface (DPI).This limitation is not mentioned in the IEEE 1800 standard, but is part of the IEEE 2008 draft LRM.

Example 5-8 Packed Union

The following example illustrates how unions can provide different representations of the same data.

```
typedef struct packed {
   bit         [3:0]   GFC;
   bit         [7:0]   VPI;
   bit         [15:0]  VCI;
   bit         [2:0]   PTI;
   bit                 CLP;
   bit         [7:0]   HEC;
   bit [0:47] [7:0]  PAYLOAD;
   } uniType;

typedef struct packed {
   bit [0:4] [7:0]  HEADER;
   bit [0:47][7:0]  PAYLOAD;
} head_pay_format;

typedef union packed {
   uniType uni;
   head_pay_format hpf;
} ATMcell;

module test();

ATMcell cell1;

initial begin

   cell1.uni.GFC = 4'h5;
   $display("header is %h",cell1.hpf.HEADER);
   $stop;
end
endmodule
```



When you run this example, the simulator displays the first byte of the header:

```
% irun -access rwc packed_atm.sv
...
ncsim> run
header is 5000000000
Simulation stopped via $stop(1) at time 0 FS + 0
./packed_atm.sv:29   $stop;
```

Example 5-9 Unpacked Union

The following example illustrates how untagged unions can be updated by one member type, and read as a value of another member type—causing a type loophole, where the union stores a value of one type and misinterprets the bits as another type.

```
typedef union{
  logic [7:0] L8;
  integer  I32;
  real     c_d;
} unpackedunion;

module test();

  unpackedunion uu;

  initial begin
   uu.L8 = 8'h8a;
   $display("logic is %h",uu.L8);
   $display("integer is %h",uu.I32);
   $display("real is %e",uu.c_d);
   $stop;
end
endmodule
```

Unpacked union with three members of different data types

When you run this example, results are not consistent:

```
% irun -access rwc unpacked_union.sv
...
ncsim> run

logic is 8a
integer is ffffff8a
real is -NaN
Simulation stopped via $stop(1) at time 0 FS + 0
./unpacked_union.sv:17   $stop;
ncsim> value uu
'{L8:8'h8a, I32:-118, c_d:nan}
```

Valid read

However, all results are not guaranteed to return the correct data type

uwire Nets

Verilog net types, such as `wire` and `trireg`, determine how the value of a net is computed from its drivers. The current release supports a new net type, `uwire`, for single-driver nets.

Note: Cadence implemented this new net type in the IUS5.4 production release, before it was approved by the IEEE for inclusion in the SystemVerilog standard. Cadence called this new net type `wone`. Beginning with the first IUS5.4 hotfix release (IUS5.4-S1), if you try to use the `wone` keyword, an error is generated telling you to use the equivalent `uwire` net type instead.

You can use the `uwire` net type to enforce a restriction that a given net has at most one driver. A `uwire` net behaves like a single-driver `wire` net. The value of a `uwire` net is the value of its driver, if it has one. If the net has no driver, its value is `Z`. During elaboration, a check is performed for multiple drivers for the `uwire` net. The elaborator generates an error if multiple drivers are detected.

A `uwire` net cannot be connected to a bidirectional terminal of a tran network.

It is an error to connect a `uwire` net to an AMS `wreal` net.

The following design defines a `uwire` net called `toggle`. It has two drivers: a gate output in module `top`, and an implicit continuous assignment for the connection to the output port of the `toggle_driver` module:

```
`timescale 1ns/1ns

module top;

    uwire toggle;

    toggle_driver td (toggle);

    not #1 delay_toggle (toggle, toggle);

    initial
    begin
        $monitor($stime, "toggle = %b", toggle);
        #10 $finish;
    end
endmodule

module toggle_driver (output reg toggle = 1'b0);

    always #1 toggle = ~toggle;

endmodule
```

Because the `uwire` net has two drivers, the elaborator generates the following error:

```
ncelab: *E,UWIREM: Multiple drivers detected on a 'uwire' net (top.toggle).
```

You do not need to specify the `uwire` net type on all of the ports connected to your net. The `uwire` net type has precedence over all other net types, except for `supply0` and `supply1`.

The coercion of a net to `uwire` can cause a change in behavior for some types of nets. In such cases, the elaborator issues a warning. For example, connecting a `uwire` to a `trireg` results in the following kind of warning:

```
ncelab: *W,UWIREC (./test.v,7|23): Incompatible port connection to 'uwire' net
top.toggle; 'uwire' dominates, and 'trireg' semantics are not in effect.
```

When different net types are connected through a module port, and one or both of the net types is `uwire`, the resulting net type is determined as follows:

Net Type 1	Net Type 2	Resolution	Warning?
uwire	uwire	uwire	No
uwire	wire/tri	uwire	No
uwire	wand/triand	uwire	Yes
uwire	wor/trior	uwire	Yes
uwire	trireg	uwire	Yes
uwire	tri0	uwire	Yes
uwire	tri1	uwire	Yes
uwire	supply0	supply0	No
uwire	supply1	supply1	No

You can use the `` `default_nettype `` compiler directive to make the `uwire` net type the default for all of your nets.

Static Casting

SystemVerilog adds the ability to change the data type of a value using a cast (`'`) operation. Static casts have the following syntax:

```
casting_type ' (expression)
```

In this type of cast, the value of *`expression`* is converted to the data type specified by the *`casting_type`*.

Example 5-10 Simple Static Cast

```
typedef enum { red=0, green=1, blue=2} colorT;
colorT v;
v = colorT'(1); // Type cast
$display("v: %s", v.name()); // Displays the value 'green'
```

In this example, the type cast `colorT' (1)` converts the value `1` to the data type `colorT`, which is equivalent to `green`.

Example 5-11 Static Cast with a Scope Resolution Operator

```
module simple;
  class abc;
   typedef enum bit {RD, WR} dir_t;
   dir_t dir;
  endclass
  abc a=new;
  initial begin
   abc::dir_t dir;
   dir=abc::dir_t' (1); //Type cast
   #1 $display("Direction is %n", dir);
  end
endmodule
```

In this example, the type cast `dir=abc::dir_t' (1)` converts the value 1 to the data type `dir_t`, which is equivalent to `WR`.

Example 5-12 Static Cast with a Vector

```
typedef logic [3:0] vec4T;
reg [7:0] vec8V;
...
vec8V = 8'b11111111;
$display("vec8v truncated to vec4T: %b", vec4T'(vec8V));
// Displays value "1111"
...
```

In this example, the static type cast `vec4T'(vec8V)` converts the value of `vec8V`—`8'b11111111`—to a vector of width 4.

Casting to Real Data Types

SystemVerilog allows casting of *`casting_type`* to be `real` or `realtime` types, their typedefs, or a type parameter resolving to one of their types. Similarly, the data type of *`expression`* can be one of these data types. The intended use models are converting an integer value to a real value, and a real value to an integer value within an expression.

Example 5-13 Static Cast to real Types

SystemVerilog code:

```
module top;
  real x = 3.14;
  initial $display(int'(x));
    int i = 3;
    real y = real'(i);
    initial $display(y + 0.14);
endmodule
```

Produces the following output:

```
ncsim> run
3
3.14
ncsim: *W,RNQUIE: Simulation is complete.
```

Casting to Vector Width

Casting to a vector width causes the value of an expression to be truncated or extended so that its width matches the width specified in the cast. When casting to a width, a static cast has the following abstract syntax:

constant_primary '(*expression*)

where *constant_primary* is evaluated to a positive number and the result of *expression* is converted to be a vector of that width.

You can also use casting to `signed` or `unsigned`, which causes the value of an expression to be converted to either a signed or unsigned quantity. When changing the sign, a static cast has the following abstract syntax:

signing '(*expression*)

where *signing* is one of the reserved words `signed` or `unsigned`. The effect is to convert the result of *expression* to be either signed or unsigned.

Example 5-14 Casting to Vector Width and Changing Sign

SystemVerilog code:

```
module top;
  int x = -1;
  reg [3:0] y = 7;
  initial begin
    $display("%0d cast to unsigned is %0d", x, unsigned'(x));
    $display("%b cast to width 2 is %b", y, 2'(y));
  end
endmodule
```

Produces the following output:

```
ncsim> run
```

```
-1 cast to unsigned is 4294967295
0111 cast to width 2 is 11
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

Casting a Class Handle

Using SystemVerilog static cast lets you convert a class handle to an assignment-compatible class type. Thus, you can explicitly cast a class handle to have a different class data type. If the class handle cannot be cast to the target class type due to the SystemVerilog assignment compatibility rules, an error is generated.

Example 5-15 Static Cast of a Class Handle

```
class automobile;
  int odometer;
endclass

class volvo extends automobile;
  string import_date;
  endclass

volvo my_volvo = new;
// Explicitly cast 'my_volvo' to be an 'automobile'
automobile my_auto = automobile'(my_volvo);
// This will cause a type-check error:
initial begin
  // Cannot cast 'my_auto' to be a 'volvo'
  $display(volvo'(my_auto));
end
```

Limitations on Type Casting

In the current release, type casts have the following limitation:

- The *casting_type* cannot be a reference to a type parameter.

Limitations on Data Types

This section summarizes the syntax constructs in the SystemVerilog LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- The `fps_covergroup_identifier` data type alternative is not supported in the current release.
- `shortreal` is not allowed as a non_integer_type.

6

Arrays

Arrays are used to hold elements of a declared data type. In Verilog-2001, arrays can be multidimensional, and can be declared for all data types. Verilog-2001 arrays use the following syntax:

```
data_type vector_width array_name array_dimension
```

For example:

```
reg [4:0] student_id [0:7]; // Array of 8 student_ids. Each student_id is
                            // 5 bits wide.
integer msk[0:63];          // Array of 64 integer values
wire array_w[7:0][5:0];     // Multidimensional array of wires
```

SystemVerilog array declarations use the following syntax:

```
data_type packed_dimensions array_name unpacked_dimensions
```

where

- *packed_dimensions*—Dimensions preceding the name of the array indicate a packed array.
- *unpacked_dimensions*—Dimensions following the name of the array indicate an unpacked array.

For example:

```
integer chk [3:0][7:0][3:0];  // Multidimensional unpacked array
bit [7:0][3:0] chk1;          // Multidimensional packed array
```

SystemVerilog enhances arrays by

- Allowing unpacked arrays of any data type, including the `event` data type
- Allowing multidimensional packed arrays
- Adding *dynamic arrays*, which let you change the size of one of the dimensions in an unpacked array

  Storage for a dynamic array is allocated during simulation.

- Adding *associative arrays*

  These arrays are best used when the size of a collection of variables is unknown, or when data space is limited. Storage for an element in an associative array is not allocated until that element is accessed.

- Adding *queues*

  Queues are used to collect elements of a declared data type. Queues are declared like arrays, but use `$` for the range.

Packed and Unpacked Arrays

In Verilog, vectors can consist of single-bit data types, such as `reg` or `wire`, and the vector range is declared just before the signal name. In SystemVerilog, vector declarations are called *packed arrays*. For example, the following declares a packed array called `addr` that is 41 bits wide:

```
reg [0:40] addr; // One-dimensional packed array
```

Packed arrays are used to divide a vector into its subfields so that the subfields can be accessed as array elements. Packed arrays can be made up of single-bit types, other packed arrays, or packed structures. Packed arrays are always represented as a contiguous set of bits.

SystemVerilog enhances arrays by letting you declare multidimensional packed arrays. For example, the following declares an array of four 6-bit sub-arrays:

```
bit [3:0][5:0] addr2; // Two-dimensional packed array
```

The Cadence implementation supports the following functionality for packed arrays:

- Packed arrays whose element type is another packed array. Specifically, the Cadence implementation supports multidimensional vectors.

  ```
  logic [8:1] [16:1] v1 [63:0] [31:0];
  reg [128:1] x;
  ...
  x = v1[4][7];
  ...
  ```

- Packed arrays of enumerations. In the following example, `p` is a packed array of enumeration vectors.

  ```
  typedef enum bit [7:0] {x, y, z} t;
  parameter t [31:0] p = 0;
  ```

- Packed structures with packed array members. In the following example, `w1` is a packed structure with a multidimensional packed array member:

```
wire struct packed {
logic [2:0][1:0] m;
} w1;
```

- Packed arrays of packed structures. In the following example, `w3` is a packed array of packed structures, where each structure has a two-dimensional packed array member.

```
wire struct packed {
logic [2:0][1:0] m;
} [7:0] w3;
```

- Verilog arrays of packed arrays.
- Assignment patterns for assigning to elements of a packed array. See "Assignment Patterns" on page 150 for information about assignment patterns.

Unpacked arrays are declared by placing the dimensions right after the array name, which is similar to the Verilog style of array declarations. Unlike packed arrays, an unpacked array can be of any type, and might or might not be represented as a contiguous set of bits. For example:

```
wire n [0:32];        // One-dimensional unpacked array of 33 one-bit nets
int m [7:0][3:0]; // Two-dimensional unpacked array of 32-bit int variables
```

The Cadence implementation supports the following functionality for unpacked arrays:

- Unpacked arrays whose size is specified by a single positive number.

  Instead of a range, SystemVerilog allows the use of a single positive number to denote the size of an unpacked array, where `[size]` is the same as `[0:size-1]`. For example:

```
logic v[4];                  // Same as: logic v[0:3];
wire [5:0] w[10][0:4][20];   // Same as: wire [5:0] w[0:9][0:4][0:19];

parameter aSize = 3000;
typedef int myArray[aSize];  // Same as: typedef int myArray[0:aSize-1];

logic v1[0];                 // Illegal. Array size must be positive.
```

  Note: This feature is not available for packed array dimensions. For example, the following declaration is invalid:

```
logic [8] eightBits;         // Invalid. [8] is a vector dimension.
```

- Assignments between unpacked array variables. The current release enforces the SystemVerilog type-compatibility requirements for assignments to unpacked arrays. For example:

```
integer fourintA[4];
integer fourintB[4];
integer fiveint[5];
```

```
int fourint[4];
...
task showSum(input integer x[4]);
...
endtask
...
fourintA = fourintB;  // Valid
showSum(fourintB);    // Valid
fourintA = fiveint;   // Invalid
showSum(fourint);      // Invalid
```

- Unpacked arrays as arguments to tasks and functions, and as function return types. For example, the following function accepts an unpacked array input argument and returns an unpacked array function.

```
typedef reg [31:0] registerSetT[16];
registerSet myregs;
function registerSetT negateRegisters(input registerSetT regs);
...
endfunction
...
myRegs = negateRegisters(myRegs);
```

- Conditional logical equality (==) and case equality (===) operations on unpacked arrays.
- Assignment patterns for assigning to elements of an unpacked array. For example:

```
int upa [0:3];
intial upa = '{0,1,2,3}; // Assignment to an unpacked array
```

  See "Assignment Patterns" on page 150 for information about assignment patterns.

- Unpacked arrays that are involved in assignments, function and task argument passing and value return, and conditional and equality operations cannot be given by an out-of-module reference, or be variable inside a class.

Limitations on Packed and Unpacked Arrays

This section summarizes the features in the SystemVerilog standard that are not supported in the current release.

- In Verilog, you can select only a single element of an array. SystemVerilog enhances arrays by allowing the selection of one or more contiguous elements of an array. This selection is called a *slice*. Selecting an array slice in SystemVerilog is similar to performing a constant part select or indexed part select in Verilog.

  The Cadence implementation supports only SystemVerilog slices of packed arrays, where the array is a one-dimensional array of scalars. The implementation supports slicing only the last dimension of a multidimensional vector.

  The implementation does not support slices on unpacked arrays.

For example:

```
module top;
  logic [7:0][4:0] mdv;
  logic arr [9:0];
  logic [1:0] twoScalars;
  logic [9:0] tenScalars;

  initial begin
    twoScalars = mdv[7][1:0];  // Valid, slice of a one-dimensional
                               // packed array of scalars
  tenScalars= mdv[7:6];        // Invalid, slice does not occur
                               // at last dimension
  twoScalars= arr[7:6];         // Invalid, slice of an unpacked array
  end
endmodule
```

Array Querying Functions

The current release supports the SystemVerilog system functions that are used to return information about the dimensions of a given array or integral data type, or of data objects of such a data type.

In the current release:

- Array query functions are supported for fixed arrays and integral data types. They are not supported for dynamic arrays.
- Array query functions can be used in class objects and to access class properties. They are also supported for packed arrays in class objects. However, they are not supported for unpacked arrays in class objects.

The supported system functions are the following:

`$left`	Returns the left bound (most significant bit) of the dimension.
`$right`	Returns the right bound (least significant bit) of the dimension.
`$low`	Returns the minimum of `$left` and `$right` of the dimension.
`$high`	Returns the maximum of `$left` and `$right` of the dimension.
`$increment`	Returns 1 if `$left` is greater than or equal to `$right`, and `-1` if `$left` is less than `$right`.

`$size`	Returns the number of elements in the dimension. This is also equal to `$high - $low + 1`.
`$dimensions`	For packed, unpacked, dynamic, and static arrays, this function returns the number of dimensions in the array. For the `string` data type, and other non-array data types that are equivalent to simple bit vector types, this function returns 1. For all other types, this function returns zero. **Note:** In the current release, you cannot use array querying functions with dynamic arrays.
`$unpacked_dimensions`	For static and dynamic arrays, this function returns the number of unpacked dimensions. For all other types, this function returns zero. **Note:** In the current release, you cannot use array querying functions with dynamic arrays.

Note the following for array-querying functions:

- The `$dimensions` and `$unpacked_dimensions` functions take one argument—the array identifier. For example:

```
// Returns the number of dimension for myarr, or zero if it is integral
a = $dimensions(myarr);
```

  All of the other functions take two arguments—the array identifier (required) and dimension (optional). For example:

```
// Returns the most significant bit of myarr's second dimension
a = $left(myarr, 2);
```

  Note: If the dimension is not specified, it defaults to 1. If you specify a dimension that is out of range, this function returns `x`.

- All of these functions return an integral data type.
- When used with fixed arrays, these functions can act as constant functions and can be passed as an elaboration parameter.

The following example shows the use of these system functions:

```
module test();

  logic [6:9] [10:14] word;
  parameter p = $left(word, 2);
  int a = $left(word, 1) ;
```

```
  initial begin
    void'($size(word, 2));
    $display("a = %d \n", a);
    $display("Size = %d \n", $size(word, 1));
    $display("Left bound = %d \n", $left(word, 1));
    $display("Right bound = %d \n", $right(word, 1));
    $display("Low = %d \n", $low(word, 1));
    $display("High = %d \n", $high(word, 1));
    $display("Increment = %d \n", $increment(word, 1));
    $display("Dimensions = %d \n", $dimensions(word));
    $display("Unpacked dimensions = %d \n", $unpacked_dimensions(word));
  end
endmodule
```

This code produces the following simulation results:

```
a =                            6
Size =                         4
Left bound =                   6
Right bound =                  9
Low =                          6
High =                         9
Increment =                   -1
Dimensions =                   2
Unpacked dimensions =          0
```

Dynamic Arrays

SystemVerilog enhances Verilog arrays with the addition of *dynamic arrays*. A dynamic array is one dimension of an unpacked array, whose number of elements can be set or changed during simulation. Storage for a dynamic array is allocated during simulation. The syntax for dynamic array declarations is as follows:

```
data_type array_name[];
```

For example:

```
int x[];         // Dynamic array of ints
bit [4:0] y[];   // Dynamic array of 5-bit vectors
string dynstr[]; // Dynamic array of strings

typedef int da[]; // User-defined dynamic array
da d;
```

Access Methods for Dynamic Arrays

SystemVerilog offers the following built-in methods for use with dynamic arrays:

- `new[]`—A function that creates a dynamic array of the specified size. It initializes the newly-created array elements with the elements of a specified array, or to an initial default value. The syntax for the `new[]` function is as follows:

  ```
  dyn_array = new [size][(old_dyn_array)]
  ```

where

- [*size*] is an expression that specifies the number of elements in the array, and must be a non-negative integral expression. The index of a dynamic array is always [0:*size*-1].
- [*old_dyn_array*] is an optional argument. When specified, the elements of *dyn_array* are initialized to the elements of *old_dyn_array*. Otherwise, the elements of *dyn_array* are initialized to their initial default value. The *old_dyn_array* must be a dynamic array of the same type as *dyn_array*, but can have a different size.

For example:

```
integer myaddr[];     // Declares the dynamic array
myaddr = new[50];     // Creates a 50-element array, with an index of 0 to
                      // 49, and array elements are initialized to x.
                      // The index of a dynamic array is always [0:size-1].
myaddr= new[60](myaddr); // Resizes the array, while preserving its
                         // previous content
...
integer newaddr[];
newaddr = new[70](myaddr); // Copies the content of myaddr into newaddr
```

- size()—A method that returns the size of the dynamic array.

The built-in size() method returns the current size of the dynamic array, or returns zero if the array is empty. The syntax for the size() method is as follows:

```
array_name.size()
```

For example:

```
int ab[];
...
ab = new[10];                // Creates a 10-element array
$display("%d", ab.size());  // Displays 10
```

- delete()—A method that removes all storage for a dynamic array.

The built-in delete() method removes all storage within a given dynamic array, resulting in an empty array. The syntax for the delete() method is as follows:

```
array_name.delete()
```

For example:

```
ab.delete();               // Deletes the array created in the last example
$display("%d", ab.size());  // Displays zero
```

Note: You can also use the {} construct to empty an array. For example:

```
a = {}   // Deletes the content of the array
```

Example 6-1 Accessing Out-of-Bound Elements of a Dynamic Array

If you try to write to an out-of-bound index of a dynamic array, the simulation issues a warning message. If you try to read an out-of-bound index of a dynamic array, the simulator does not issue an error message, but displays the default value. For example:

```
module top;
  string dyn_arr[]; // Dynamic array of strings
  initial begin
    dyn_arr = new[10];
    dyn_arr[0] = "ABC";
    $display("value = %d\n", dyn_arr[0]); // Displays ABC
    #5;
    dyn_arr[13] = "ABC"; // Writes to an out-of-bound index. Causes warning.
    $display("value=%d\n", dyn_arr[14]); // Reads an out-of-bound index.
                                         // Displays a null string, which
                                         // is the default value for strings.
    end
endmodule
```

Example 6-2 Fixed Arrays of Dynamic Arrays

You can declare a fixed array of dynamic arrays. For example, the following declares a fixed array called `fa_da` where each element is a dynamic array, and each element in the dynamic array is of type `int`:

```
int fa_da[0:3][]; // Fixed array of dynamic arrays
```

For example:

```
module top;
   string fada[0:3][]; // Declares a fixed array of dynamic arrays
   int i,j,k;

initial begin
  k=100;
   for(i=0;i<2;i++)
   begin
    fada[i]=new[4]; // Allocates a dynamic array of size 4
    $display("Value is %d",fada[i].size()); // Displays the size of fada
    for(j=0;j<2;j++)
     begin
      fada[i][j]="Hello";
       $display("Value is %s",fada[i][j]); // Displays the value of fada
     end
   end
end
endmodule
...
ncsim> run
Value is 4
Value is Hello
Value is Hello
Value is 4
Value is Hello
Value is Hello
```

```
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

Example 6-3 Dynamic Arrays of Fixed Arrays

The following example shows full array assignment, passing an array by value, passing an array by reference, port variable connection, reading and writing of full bit select:

```
module top;
   integer da[][0:4];
   integer da1[][0:4];
   integer da2[][0:4][0:4];
   integer fa[0:4];
   initial begin
      $display ("value = %d\n"; // Displays X
      $display ("$bits = %d\n, $bits(da)); // Displays 0
      // Creates a dynamic array containing a fixed array of 5 integers:
      da = new[10];
      // Copies a full dynamic array to another dynamic array:
      da = da1;
      // Assigns a fixed array to a bit select of a dynamic array:
      da[3] = fa;
      // Writes to a bit select:
      da[3][2] = 1;
      // Reads from a bit select:
      $display ("value of element da[3][2] = %d\n", da[3][2]);
      da2 = new[5];
      da2[3][4] = fa;
endmodule
```

Example 6-4 Dynamic Arrays of Queues

This example shows full array assignment, passing an array by value, passing an array by reference, port variable, reading and writing of full bit select:

```
module top;
  integer da[][$];
  integer da1[][$];
  integer queue[$];
  initial begin
    $display ("value = %d\n", da[0][0]);  // displays x
    $display ("$bits = %d\n", $bits(da)); // displays 0
    da = new[10] ; //creates a dynamic array that contains a queue within it
    da = da1;      // copying full dynamic array to another
    da[3] = queue; // fixed array assignment
    da[4] = queue[4:5] // part select of queues
    da[5] = da[4][4:5] // part select of queues
    da[6] = da[2].find (x) with (x> 3);
    da[3][0] = 1; // writing bit select
    $display ("value of element da[3][2] = %d\n", da[3][2]); // read bit select
endmodule
```

Example 6-5 Assigning an Entire Fixed Array to a Dynamic Array

An assignment of a fixed array to a dynamic array of a compatible type creates a new dynamic array with a size equal the length of the fixed array. Correspondence of elements is defined as leftmost to leftmost and rightmost to rightmost, irrespective of index values, as shown in the following example:

```
module top;
  int da[];
  int fa[5];
  initial begin
    fa[0]=2;
    da=fa;
  end
endmodule
```

```
module top;
  int da[];
  int fa[4:1];
  initial begin
    da = new[4];
    da[0] = 4;
    fa = da;
    $display ("fa[4] = %d\n", fa[4]); // da[0] maps to fa[4]
  end
endmodule
```

Example 6-6 Assigning an Entire Dynamic Array to a Fixed Array

You can assign a dynamic array to a fixed array of an equivalent type of the same size. The size checking is performed at run time, and, if the size of the arrays are different, an error is issued with no operation performed. Correspondence of elements is defined as leftmost to leftmost and rightmost to rightmost, irrespective of index values:

```
module top;
  int da[];
  int fa[6:4];
  initial begin
    da=new[3];
    da[0]=2;
    fa=da; // da[0] corresponds to fa[6]
  end
endmodule
```

Example 6-7 Passing Dynamic Arrays by Reference to Tasks and Functions

Passing a piece of a dynamic array by reference to a task or function is not supported. You can pass only whole objects by reference to a task or function.

```
module top;
  string dyn_arr[];
  initial begin
   dyn_arr = new[10];
   func(dyn_arr); // Calls a time-consuming task
   $display("size = %d\n", dyn_arr.size());
```

```
  end
  initial
  $monitor("value - %s time =", dyn_arr[3], $time);
  task automatic func(ref string dyn_arr[]); // Passes dynamic array by
                                             // reference
   dyn_arr[3] = "abc"; // Task changes the value of dyn_array;
                       // the new value is visible to the module
   #2;
   dyn_arr = new[20];  // Resizes the array
   #3;
   dyn_arr.delete();   // Deletes the dynamic array
   #5;
  endtask
endmodule
```

This example produce the following simulation results:

```
value - abc time =           0
value - time =               2
value - time =               5
size =                       0
```

Example 6-8 Passing Fixed Arrays by Value to Tasks and Functions

You can pass a fixed array by value to a task or function that accepts dynamic arrays of a compatible type. For example, the following example passes fixed array `fa` to task `t1`, which is a task that is set up to accept a dynamic array:

```
module top;
  task t1(int da[]);
   $display("%d", da[0]);
  endtask
  int fa[5];
  initial begin
    fa[0]=2;
    t(fa); // Passes fixed array to task t1
  end
endmodule
```

Example 6-9 Passing Dynamic Arrays by Value to Tasks and Functions

You can pass a dynamic array by value to a task or function that accepts fixed arrays of a compatible type. For example, the following passes dynamic array `da` to task `t2`, which is a task that is set up to accept a fixed array:

```
module top;
  task t2(int fa[10]);
   $display("%d", fa[0]);
  endtask
  int da[];
  initial begin
    da = new[10];
    da[0] =2;
    t2(da); // Passes dynamic array to task t2
  end
endmodule
```

Example 6-10 Assigning Fixed-Size Arrays to Dynamic Arrays

You can assign a fixed-size array to a dynamic array, if each element is of an equivalent type. For example:

```
module top;
  int da[];
  int fa[5];
  initial begin
    fa[0] = 2;
    da = fa; // Creates a new dynamic array of the same size as fa
  end
endmodule
```

Example 6-11 Assigning Dynamic Arrays to Fixed-Size Arrays

In the current release, you can assign a dynamic array to a fixed-sized array, if they have same size and are of equivalent types.

For example:

```
module top;
  int da[];
  int fa[6:4];
  initial begin
    da = new[3];
    da[0] = 2;
    fa = da; // da[0] maps to fa[6]
  end
endmodule
```

Example 6-12 Assigning an Entire Dynamic Array to Another

You can assign a dynamic array to another dynamic array with an equivalent element type:

```
class C;
  int a;
endclass

class C1 extends C;
  int b;
endclass

C da_cls[];
C1 da_cls1[];
C da_cls2[];
int fa_int[0:3];
int da_int[];
int da_int1[];
bit [0:31]da_bit[];
string da_str[];
string da_str1[];
da_int = new[4]];
da_int = da_bit  /* gives TYPEERR at compile time*/
da_cls = da_cls1 /* gives TYPEERR at compile time*/
da_int = fa_int  /* gives TYPEERR at compile time*/
fa_int = da_int  /* gives TYPEERR at compile time */
```

```
da_int = da_int1 /* assigns an entire queue to another*/
da_cls = da_cls2 /* assigns an entire queue to another*/
da_str = da_str1 /* assigns an entire queue to another*/
```

Limitations on Dynamic Arrays

This section summarizes the features in the SystemVerilog standard that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- The Cadence implementation supports dynamic arrays of
  - `bit`
  - `logic`
  - `byte`
  - `shortint`
  - `int`
  - `longint`
  - `integer`
  - `string`
  - `real`
  - Enumerated data types
  - Packed arrays of `bit`, `logic`, and `reg`
  - Packed and unpacked structures
  - Mailboxes
  - Semaphores
  - Fixed and associative arrays
  - Classes
- The Cadence implementation does not support dynamic arrays within multidimensional arrays.
- Dynamic arrays are supported
  - On module ports

- At the top level of a module, interface, package, class, or program block
- In global compilation units
- Within tasks, functions, and class methods

Dynamic arrays are not supported within structures. For example, if a structure is defined within a module, a dynamic array declaration within the structure is not currently allowed.

- Dynamic arrays are supported within `begin...end` blocks that are in the following:
  - Modules, program blocks, and interfaces, including `if...else`, `while`, and `for` loops declared within these scopes
  - Tasks and functions
  - `for generate, if generate, case generate`
  - `fork...join`, `repeat`, and `forever` loops
  - `always` blocks
  - `final` statements
  - `casex` and `casez` statements
  - Named blocks
- Dynamic arrays can be declared as public, `static`, `local`, or `protected` members of a class that is declared within a package.
- Multidimensional dynamic arrays are not supported; for example:

```
reg [0:7] arr[];    // Supported
int int_array[];    // Supported
reg [0:7] arr1[][];  // Unsupported
int int_arr[][];    // Unsupported
```

- Assignments made to a part select of a dynamic array are not supported in the current release.

```
module test_top;
  int arrA[0:7];
  int dynarr[];
  int dynarr1[];
  initial begin
    dynarr  = new[8];
    dynarr1 = new[8];
    dynarr[0:2]  = arrA[0:2];    // Unsupported
    dynarr1[0:2] = dynarr[0:2];  // Unsupported
  end
endmodule
```

- The Cadence implementation specifies an upper limit of `(2**31-1)` for dynamic arrays. In the following example, the parser issues a warning if N is greater than `(2**31-1)`. In this case, the elaborator and simulator will continue with undefined behavior.

```
dyn_arr = new[N];
```

- While the Cadence implementation supports OOMRs and hierarchical references for dynamic arrays, there is a limitation which does not allow calling methods of the dynamic array class using hierarchical references. However, you can make such method calls using local objects after assigning an out-of-scope instance of dynamic array to a local declaration of the same kind.

- Passing a part select or element of a dynamic array by reference to a task or function is not supported. You can pass only whole objects by reference to a task or function. For example:

```
task automatic func(ref dynarr[]); // Legal
task automatic func1(ref dynaarr[0]); // Not supported
```

  Dynamic arrays cannot be passed as `input`, `output`, or `inout` types to a task or function.

- You cannot use an event control on an entire dynamic array, or on a part select of a dynamic array. However, you can use an event control on an individual element of a dynamic array:

```
int da[];
always @(da)         // On entire array. Not supported.
...
always @(da[0:2])    // On part select of the array. Not supported.
...
always @(da[0])     // On an array element. Supported.
...
```

  You cannot use an event control on a part select of a fixed array of dynamic arrays.

Note: All of these limitations apply also to dynamic arrays of strings and classes.

- The current release supports the `sum()` array reduction method, but only with dynamic arrays, as shown in the following example:

```
module top;
  byte da[];
  int i;
  initial begin
    da = new[5];
    for (i = 0; i < 5; i ++)
      da[i] = i;
    i = da.sum();
    $display(i); // i becomes 10
  end
endmodule
```

- The current release does not support the `and()`, `or()`, `product()`, or `xor()` methods.

- The current release does not support assigning fixed arrays to dynamic arrays and vice versa via port variables.
- Array assignments involving complex expressions like concatenation are not supported.
- Usage of an entire array is allowed only in assignment statement (not in sensitivity lists or in system task and functions).

Associative Arrays

SystemVerilog enhances Verilog arrays with the addition of *associative arrays*. An associative array is declared using a data type as its special array size. The syntax for declaring an associative array is as follows:

```
data_type array_id [index_type]
```

where:

- *data_type* is the data type of the array elements.
- *array_id* is the array name.
- *index_type* specifies the data type that will be used as an index.

The following are examples of associative array declarations:

```
logic my_array[integer];  // Associative array with an integer index
...
typedef int foo;
foo myfoo[int];           // Associative array constructed from a typedef

typedef int a_t[string];  // User-defined associative array
a_t a;
```

Storage for members of an associative array is allocated when that member is created. Associative arrays are best used when you have limited data space, or when the size of the collection of variables is unknown.

For a list of unsupported features, refer to "Limitations on Associative Arrays" on page 102.

Access Methods for Associative Arrays

This section uses the following example to describe the built-in methods that provide access into associative arrays:

```
module MyMod;
  int myArr[byte];
  int myVar;
  byte myIndex;
```

```
  byte i;
  initial begin
    myVar = 5;
    myIndex = 45;
    myArr[34] = 1;
    myArr[-66] = myVar;
    myArr[myIndex] = 13;
  end
endmodule
```

- `num()` method—Returns the number of entries for an associative array. Returns zero for empty arrays. For example, the following prints "`3 items in my array`":

  ```
  myVar = myArr.num();
  $display("%d items in my array", myVar);
  ```

- `delete()` method—Removes elements in an associative array. The syntax for this method is as follows:

  ```
  array_name.delete([input index]);
  ```

  where *`index`* is an optional index of the appropriate type. If *`index`* is specified, the `delete()` method removes the entry at the specified index. If *`index`* is unspecified, the `delete()` method removes all of the elements in the array.

  For example:

  ```
  myArr.delete(34); // Deletes entry whose index is 34
  ```

- `exists()` method—Indicates whether an array element exists for a particular index. Returns 1 if the element exists; otherwise returns 0. The syntax for this method is as follows:

  ```
  array_name.exists(input index);
  ```

  where *`index`* is an index of the appropriate type.

  For example, the following displays "Index 45 exists", because an element exists with an index of `45`:

  ```
  if (myArr.exists(45))
    $display("Index 45 exists");
  else
      $display("Index 45 does not exist");
  ```

- `first()` method—Assigns to the given index variable the value of the first or smallest index in the associative array. Returns 0 if the array is empty; otherwise returns 1. The syntax for this method is as follows:

  ```
  array_name.first(ref index);
  ```

  where *`index`* is an index of the appropriate type.

  For example:

  ```
  if (myArr.first(i));
    $display ("The first index of the array is %d", i);
  ```

- `last()` method—Assigns to the given index the value of the last or largest index in the associative array. Returns 0 if the array is empty; otherwise returns 1. The syntax for this method is as follows:

  ```
  array_name.last(ref index);
  ```

  where *index* is an index of the appropriate type.

  For example:

  ```
  if (myArr.last(i));
    $display ("The last index of the array is %d", i);
  ```

- `next()` method—Locates an entry with an index that is greater than the specified index. If it finds an entry, the method assigns the index of the located entry to the index variable, then returns 1. If it cannot find an entry, the method returns 0, and the index variable is unchanged. The syntax for this method is as follows:

  ```
  array_name.next(ref index);
  ```

  where *index* is an index of the appropriate type.

  For example:

  ```
  if (myArr.first(i));
    do
     $display ("%d in the current index", i);
    while
      (myArr.next(i));
  ```

- `prev()` method—Locates an entry with an index that is smaller than the specified index. If it finds an entry, the method assigns the index of the located entry to the index variable, then returns 1. If it cannot find an entry, the method returns 0, and the index variable is unchanged. The syntax for this method is as follows:

  ```
  array_name.prev(ref index);
  ```

  where *index* is an index of the appropriate type.

  For example:

  ```
  if (myArr.last(i));
    do
     $display ("%d in the current index", i);
    while
     (myArr.prev(i));
  end
  ```

Example 6-13 Assigning One Associative Array to Another

An associative array can be assigned to another associative array with an equivalent element type and with the same index type.

```
class C;
  int a;
endclass
```

```
class C1 extends C;
  int b;
endclass

int aa_int_cls [C];
int aa_int_cls1[C1];
C aa_cls [int];
C1 aa_cls1[int];
C aa_cls2[int];
int aa_w[*];
int aa_int[int];
int aa_int1[int];
int aa_int2[longint];
bit [0:31]aa_bit[int];
aa_w = aa_int;    /* wildcard associative arrays can be assigned to another
wildcard associative array. TYPEERR at compile time.*/
aa_int = aa_bit    /* gives TYPEERR at compile time*/
aa_int = aa_int2;  /* gives TYPEERR at compile time*/
aa_int_cls = aa_int_cls1 /* gives TYPEERR at compile time*/
aa_cls = aa_cls1   /* gives TYPERR at compile time */
aa_int = aa_int1   /* assigns an entire associative array to another*/
aa_cls = aa_cls    /* assigns an entire associative array to another*/
```

Example 6-14 Passing Associative Arrays by Value to Task and Functions

An associative array can be assigned to another associative array with an equivalent element type and with the same index type. For example, for the following declaration:

```
task fun (int aa [int])
```

consider these actuals:

```
int aa [int]     // allowed
int aa [longint] // not allowed
```

Limitations on Associative Arrays

This section summarizes the associative array features in the SystemVerilog standard that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Although the LRM states that an associative array can use any data type allowed for fixed-size arrays, the Cadence implementation supports only the following:
  - `int`
  - `integer`
  - `logic`
  - `bit`
  - `reg`

- `shortint`
- `byte`
- `longint`
- `time`
- `class`
- `string`
- `real`
- Events
- Packed and unpacked structures
- Mailboxes
- Semaphores
- Enumerated data types
- User-defined packed types
- Unpacked structures
- Packed structures
- Virtual interfaces

The Cadence implementation does not support the following complex data types for associative arrays:

- Multidimensional arrays

- The Cadence implementation supports associative array declarations on module ports, and in modules, interfaces, classes, packages, programs, tasks, functions, global compilation units, and class methods.

  Associative arrays are not supported within generates, unless they are enclosed in a `begin...end` block.

- Associative arrays are supported within `begin...end` blocks that are in the following:

  - Modules, program blocks, and interfaces, including `if...else`, `while`, and `for` loops declared within these scopes.

    Associative arrays are not supported in any other sub-scopes of a module, interface, or program.

- Tasks and functions
- `for generate, if generate, case generate`
- `fork...join`, `repeat`, and `forever` loops
- `always` blocks
- `final` statements
- `casex` and `casez` statements
- Named blocks

- Associative arrays can be declared as public, `static`, `local`, or `protected` members of a class that is inside a package, module, interface, or program block.
- The Cadence implementation supports strings, integers, user-defined packed types, class handles, and wildcards (`*`) as index types to associative arrays. However, the user-defined types described in LRM are not supported. The Cadence implementation does not support complex user-defined types.
- The Cadence implementation supports limited indexing into associative arrays, through bit selects. A bit select can be a constant or dynamic value. For example:

  ```
  a = myArray[0];
  myArray[1] = a;
  ```

  If you try to read a value that does not exist, the simulator returns the default initial value for the array type, as specified in Table 5-1 of the LRM. If you specify an invalid index during a write operation, the simulator ignores the write.
- The LRM describes the concatenation syntax and how to specify default values for associative arrays. These features are not supported in the current release.

  ```
  int a1[int] {default:1};  // Unsupported
  ```
- The Cadence implementation does not support OOMRs or hierarchical references. You can reference an associative array only from the scope in which it was declared. However, if an associative array is declared within a package, you can reference it from other scopes.
- In the Cadence implementation, you can update values in an associative array through calls to the methods discussed in "Access Methods for Associative Arrays" on page 99, or through standard blocking assignments. You can only use blocking assignments to assign a single bit-select item; blocking assignments cannot be a part of a complex expression, such as concatenation or part selects.

  The Cadence implementation does not support other methods of updating values, which include forces, continuous assignments, or non-blocking assignments.

- Passing a piece of an associative array by reference to a task or function is not supported. You can pass only whole objects by reference to a task or function.
- Assignments to a part select of an associative array are not supported in the current release. For example:

```
int aa[int];
int aa1[int];
initial begin
  aa[0:3] = aa1[0:3]; // Not supported--part-select assignment
end
```

- You cannot use an event control on an entire associative array, or on a part select of an associative array. However, you can use an event control on an individual element of an associative array:

```
int aa[byte];
always @(aa)        // Not supported on entire array
always @(aa[0:2])   // Not supported on part select of the array
always @(aa[0])     // Supported on an array element
```

- In the current release, associative arrays can be declared as `rand`, but not as `randc`. For more information, see "Limitations on Random Variables" on page 183.
- Array assignments involving complex expressions like concatenation are not supported.
- Usage of an entire array is allowed only in assignment statement (not in sensitivity lists or in system task and functions).

Queues

For examples that you can download and run, refer to the *SystemVerilog Engineering Notebook*.

SystemVerilog introduces the queue construct, which is a variable-sized collection of elements of a declared data type. Queues are similar to one-dimensional, unpacked arrays whose size can increase or decrease automatically.

A queue is declared like an array, but uses a dollar sign (`$`) for its range. The maximum size of an array can be limited by specifying an optional *`constant_expression`* as its last index. The syntax for declaring a queue is as follows:

```
data_type queue_id [$[:constant_expression]]
```

For example:

```
int myArray[0:63];   // A standard array of 64 integers
int myQueue[$];      // A queue of integers
bit q1[$:63];        // A queue with a maximum size of 64 bits

typedef int foo;     // A queue constructed from a typedef
```

```
foo q2[$];

typedef int q_t[$];  // A user-defined queue
q_t q;
```

Queue members are identified by numbers that represent their position in the queue. Zero represents the leftmost (first) member of the queue, and `$` represents the rightmost (last) member.

Access Methods for Queues

This section describes the built-in methods that provide access into queues.

- `size()` method—Returns the number of elements in a queue. Returns 0 for empty queues. For example:

  ```
  int ww55[$];
  q = ww55.size();  // Returns 0
  ```

  Note: When used in a constraint expression, the built-in `.size()` method of a queue has the appearance of a state variable. The constraint will not set the size of a `rand` queue. This is a limitation outlined by the LRM.

- `insert()` method—Inserts the specified item at the specified index position. For example, the following inserts element `0` at position `15` within the queue called `ww55`.

  ```
  ww55.insert(0,15);
  ```

- `delete()` method—Deletes the element at the specified position. For example, the following deletes the element at position `0` within the queue called `ww55`:

  ```
  ww55.delete(0);
  ```

  The argument to the `delete()` method is optional. If the argument is missing, the entire contents of the queue are deleted.

  ```
  ww55.delete(); // Deletes contents of myq
  ```

  Note: You can also use the `'{}` construct to empty a queue. For example:

  ```
  q = '{}   // Deletes the content of the queue
  ```

- `pop_front()` method—Removes and returns the first element in a queue. For example, the following removes and returns the first element in `ww55`:

  ```
  q = ww55.pop_front();
  ```

- `pop_back()` method—Removes and returns the last element in a queue. For example, the following removes and returns the last element in `ww55`.

  ```
  q = ww55.pop_back();
  ```

- `push_front()` method—Inserts the specified element at the beginning of the queue. For example, the following inserts `1` at the beginning of queue `ww55`:

```
ww55.push_front(1);
```

- `push_back()` method—Inserts the specified element at the end of the queue. For example, the following inserts `15` at the end of queue `ww55`:

```
ww55.push_back(15);
```

Example 6-15 Queues of Fixed Arrays

This example shows full array assignment to a queue, passing a queue by value, passing a queue by reference, port variable connection, reading and writing of full bit select:

```
module top;
  integer q[$][0:4];
  integer q1[$][0:4];
  integer q2[$][0:4][0:4];
  integer fa[0:4];
  initial begin
    $display ("value = %d\n", q[0][0]); // x
    q = q1;                 // copying full queue to another
    q[0] = fa;              // fixed array assignment
    q[0]][2] = 1;           // writing bit select
    $display ("value of element q[0][2] = %d\n", q[0][2]); // reading bit
select
    q2[0][0] = fa
endmodule
```

Example 6-16 Fixed Arrays of Queues

This example shows full array assignment to a queue, passing a queue by value, passing a queue by reference, port variable connection, reading and writing of full bit select:

```
module top;
  integer fa_q[0:4][$];
  integer fa_q1[0:4][$];
  integer fa_q2[0:4][0:4][$]
  integer fa[0:4];
  integer q[$];
  initial begin
    $display ("value = %d\n", fa_q[0][0]); // x
    fa_q = fa_q1;          // copying full fixed array of queues to another
    fa_q[0]= q;            // queue assignment
    fa_q[0]][2] = 1;       // writing bit select
    $display ("value of element fa_q[0][2] = %d\n", fa_q[0][2]); // read bit
select
endmodule
```

Example 6-17 Queues of Associative Arrays

This example shows full array assignment to a queue, passing a queue by value, passing a queue by reference, port variable connection, reading and writing of full bit select:

```
module top;
  integer q[][*]];
  integer q1[$][*];
```

```
  integer aa[*];
  initial begin
    $display ("value = %d\n", q[0][0]); // x
    q = q1; // copying full dynamic array to another
    q[0]] = aa;                     // associative array assignment
    q1 = q[4:5]                     // part select of queues
    q1 = q.find (x) with (x> 3);    //gives an error
    q1 = q.find (x) with (x[0]> 3); // works
    q[0][0] = 1;                    // writing bit select
    $display ("value of element q[0][0] = %d\n", q[0][0]); // reading bit
select
endmodule
```

Example 6-18 Assigning One Queue to Another

A queue be assigned to another queue with an equivalent element type and with the same maximum size.

```
class C;
  int a;
endclass

class C1 extends C;
  int b;
endclass

C q_cls[$];
C1 q_cls1[$];
C q_cls2[$];
int q_int[$];
int q_int1[$];
int q_int2[$:3];
bit [0:31]aa_bit[$];
string q_str[$];
string q_str1[$];
q_int = q_bit /* gives TYPEERR at compile time*/
q_int = q_int2; /* gives TYPEERR at compile time, maximum size differs*/
q_cls = q_cls1 /* gives TYPEERR at compile time*/
q_int = q_int1 /* assigns an entire queue to another*/
q_cls = q_cls2 /* assigns an entire queue to another*/
q_str = q_str1 /* assigns an entire queue to another*/
```

Example 6-19 Passing a Queue by Value to Tasks and Functions

You can pass a queue by value to another queue with an equivalent element type and with the same maximum size. For example, for the following declaration:

```
task fun (int q [$])
```

consider the these actuals:

```
int q [$]       // allowed
longint q_1 [$] // not allowed, TYPEERR
int q[$:3]      // not allowed, TYPERR
```

Example 6-20 Using Slices of Queue on Right-Hand Side of an Assignment

You can use queue slices on the right-hand side of an assignment, if the element types of queues on both the right-hand side and left-hand side are equivalent. For example:

```
int q[$];
int q1[$:9];
int q2 [$:2];
int q3[$:3];
int q4[$:4];
q = q1[1:7];  // Gives TYPERR error at elab time because maximum size of
              // queues differ.
q = q [7:1]   // Yields an empty queue and gives a warning during simulation
q = q[1:1]    // Yields a queue with one item
q = q[-1:5] is same as q = q[0:5];
q = q [1:b] where b > $(last index) is same as q[1:$]
q = q2 [4:5]; // Gives an error because maximum size differs.
q[0] = 3;
q[1] = 4;
q[2] = 5;
q = q[1:2] ;  // Yields a queue with only two items in it.
```

Example 6-21 Passing Slices of Queue by Value to Tasks and Functions

You can pass a queue slice by value to another queue whose element is of equivalent type. For example, a task declared as follows:

```
task fun (int q [$])
```

is passed the following queue slices:

```
int q [$]         // Declaration
q[0:6]            // Allowed
longint q_1 [$]   // Declaration
q_1[0:8]          // Not allowed; gives TYPEERR at compile time
```

Example 6-22 Assigning a Queue to a Dynamic Array

You can assign a queue to a dynamic array of a compatible type. The assignment creates a new dynamic array with a size equal to the length of the queue. Element correspondence is defined as leftmost to leftmost, rightmost to rightmost, irrespective of index values, as shown in the following example:

```
module top;
  int da[];
  int q[$];
  initial
  begin
    q[0]=2;
    da=q;
    end
  initial
  $display ("Size of da is %d\n", da.size()); // Size of da is 1
endmodule
```

Example 6-23 Passing a Queue by Value to Tasks and Functions

You can pass a queues by value to another queue with an equivalent element type and with the same maximum size. For example, for the following declaration:

```
task fun (int q [$])
```

consider the following actuals:

```
int q [$]       // allowed
longint q_1 [$] // not allowed, TYPEERR
int q[$:3]      // not allowed, TYPERR
```

Example 6-24 Passing a Queue by Value to a Task or Function as Dynamic Array

A subroutine that accepts a dynamic array can be passed a queue of a compatible type. Rules for assigning a queue to a dynamic array, defined above, are applicable. For example:

```
module top;
  task t(int da[]);
    $display("%d",da[0]); // Value is 2
  endtask
  int q[$];
  initial
  begin
    q[0]=2;
    t(q);
  end
endmodule
```

Example 6-25 Connecting a Queue to a Dynamic Array via a Port Variable

In the following example, the actual argument is a dynamic array, and the formal argument is an output variable of a queue and of compatible type in the port connection:

```
module top;
  int da[];
  bot b1(da);
  initial #1 $display ("size of da is %d\n", da.size()); // Size is 2
endmodule

module bot (output var int q[$]);
  initial q[0] = 1;
  initial q[1] = 1;
endmodule
```

In this example, the actual argument is a queue, and the formal argument is an input variable of a dynamic array and of equivalent type in the port connection:

```
module top;
  int q[$];
  initial q[0] = 1
  initial q[1] = 1;
  bot b1 (q);
endmodule

module bot (input var int da[]);
```

```
  initial #1 $display ("size of da is %d\n", da.size()); // Size is 2
endmodule
```

Note: The following scenarios are not supported, and generate an error for incompatible type:

- The actual argument is a dynamic array and the formal argument is an input variable of a queue and of compatible type in the port connection.
- The actual argument is a queue, and the formal argument is an output variable of a dynamic array and of equivalent type in the port connection.

Limitations on Queues

This section summarizes the features in the SystemVerilog standard that are not supported for queues in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- The Cadence implementation supports only the following data types for queues:
  - `int`
  - `integer`
  - `logic`
  - `bit`
  - `reg`
  - `shortint`
  - `byte`
  - `longint`
  - `time`
  - `class`
  - `string`
  - `real`
  - Packed unions
  - Unpacked structs
  - Packed and unpacked structures

- Fixed and associative arrays
- Mailboxes
- Semaphores
- Enumerated data types
- User-defined packed types
- Virtual interfaces

The Cadence implementation does not support the following complex data types for queues

- Multidimensional arrays
- Events

- The Cadence implementation supports queue declarations on module ports and within modules, interfaces, classes, packages, program blocks, tasks, functions, global compilation units, and class methods. Queues are not supported within generates, unless enclosed in a `begin...end` block.
- Queues are supported within `begin...end` blocks that are in the following:
  - Modules, program blocks, and interfaces, including `if...else`, `while`, and `for` loops declared within these scopes.

    Queues are not supported in any other sub-scopes of a module, interface, or program.
  - Tasks and functions
  - `for generate, if generate, case generate`
  - `fork...join`, `repeat`, and `forever` loops
  - `always` blocks
  - `final` statements
  - `casex` and `casez` statements
  - Named blocks
- Queues can be declared as public, `static`, `local`, or `protected` members of a class that is inside a package, module, interface, or program block.
- The Cadence implementation supports limited indexing into queues, through bit-selects. A bit select can be a constant or dynamic value. For example:

```
a = myQueue[0];
myQueue[1] = a;
```

If you try to read a value that does not exist, the simulator returns the default initial value for the queue, as specified in Table 5-1 of the LRM. If you specify an invalid index during a write operation, the simulator ignores the write.

- The LRM specifies a concatenation-like syntax for initializing queues and part-select operations. The Cadence implementation does not support these features. For example, the following is not supported:

```
int q1[$] = {1,2,3};  // Invalid
```

- The Cadence implementation does not support OOMRs or hierarchical references. You can reference a queue only from the scope in which it was declared. However, if the queue is declared within a package, you can reference it from other scopes.

- In the Cadence implementation, you can update values in a queue through calls to the methods discussed in "Access Methods for Queues" on page 106, or through standard blocking assignments. You can use blocking assignments to assign only a single bit-select item; blocking assignments cannot be a part of a complex expression, such as concatenation or part selects.

  The Cadence implementation does not support other methods of updating values, which include forces, continuous assignments, or non-blocking assignments.

- Passing a piece of a queue by reference to a task or function is not supported. You can pass only whole objects by reference to tasks and functions.

- You can pass a queue or a part select of a queue by value to: a task, function, method, or another queue that has the same element type and maximum size.

```
package p;
  typedef reg [0:45] q_type[$];  // Queue typedef
   function void copy_print_q (q_type q); // Passes queue by value to
                                          // a function
   ...
   endfunction
...
endpackage
```

- You can use `$` in operations on a queue's index. For example:

```
myQueue[$+1] = 1;
e = myQueue[$-1];
```

  However, the following is not supported:

```
myqueue = myqueue {myqueue[0:pos-1], e, myqueue[pos:$]};
          //Not supported. $ in concatenation expressions.
```

- Assigning an entire queue to another queue is supported, but they must be compatible—that is, they have the same element type and maximum size:

```
int q[$];
int q1[$];
...
q = q1; // Whole assignment
```

Note: Entire queues are supported only in assignments. You cannot use entire queues in other places such as a system task, system function, or sensitivity list.

- Part selects of a queue are not supported on the left-hand side of an assignment. Part selects of queues are supported on the right-hand side of an assignment, provided the element types on the right and left sides are equivalent.

```
int q[$];
int q1[$:9];
int q2[$];
...
q = q1[1:7]; // Illegal; maximum sizes are different
q[0:3] = q1[0:3]; // Not supported; part select on left side
q = q2[0:5]; // Supported; right and left sides of equivalent types
```

- Indexed part selects of queues are not supported.
- You cannot use an event control on an entire queue, or on a part select of a queue. However, you can use an event control on an individual element of a queue.

```
int q[$];
always @(q)      // Not supported on entire queue
always @(q[0:2]) // Not supported on part select
always @(q[0])   // Supported on an element of the queue
```

- Usage of an entire array is allowed only in assignment statement (not in sensitivity lists or in system task and functions).

Array Manipulation Methods

SystemVerilog provides array locator and array reduction methods, which are built-in methods used to search through the indices of an array for a given expression. In SystemVerilog, array methods operate on any unpacked array, including queues and associative arrays, but have a queue return type.

Array manipulation methods have the following syntax:

array_id.array_method (*iterator_argument*) with (*expression*)

where:

- *array_id* specifies the array to traverse.
- *array_method* specifies the array method.
- *iterator_argument* specifies the name of the variable to use in the `with` expression. This variable is implicitly declared, and its scope is the `with` expression.

This variable is optional. If you do not designate a variable, the method uses the default name `item`.

- *`expression`* is used to iterate over the elements of the given array.

For example:

```
int IQ[$], qi[$];

qi = IQ.find(x) with (x > 5);        // Searches for all elements greater than 5
qi = IQ.find_index with (item == 3); // Searches for indices of items equal to 3
```

Table 6-1 Array Locator Methods

Method	Description
`find()`	Finds all of the elements that satisfy the given expression. Traverses the array from front to back.
`find_first()`	Finds the first element that satisfies the given expression. Traverses the array from front to back.
`find_first_index()`	Returns the index of the first element that satisfies the given expression. Traverses the array from front to back.
`find_index()`	Returns the indexes of all elements that satisfy the given expression. Traverses the array from front to back.
`find_last()`	Returns the last element that satisfies the given expression. Traverses the array from back to front.
`find_last_index()`	Returns the index of the last element that satisfies the given expression. Traverses the array from back to front.
`min()`	Returns the element with the minimum value or whose expression evaluates to a minimum.
`max()`	Returns the element with the maximum value or whose expression evaluates to a maximum.
`unique()`	Returns all elements with unique values or whose expression evaluates to a unique value. The queue returned contains one and only one entry for each of the values found in the array. The ordering of the returned elements is unrelated to the ordering of the original array.

Method	Description
`unique_index()`	Returns the indices of all elements with unique values or whose expression evaluates to a unique value. The queue returned contains one and only one entry for each of the values found in the array. The ordering of the returned elements is unrelated to the ordering of the original array. The index returned for duplicate valued entries may be the index for one of the duplicates.

Note:

- If relational operators (<, >, ==) are defined for the element type of the given array, then it is optional to use the `with` clause.

  For a QDA of element type integral (`int`, `bit`, `packed array`, `packed struct`, `logic`) or `real`, the `with` clause may be skipped as the required relational operators can be applied directly to the array elements.

- If relational operators (<, >, ==) are not defined for the element type of the given array, then it is mandatory to use the `with` clause.

  A queue of queue of `int` (that is, `int q[$][$]`) `q` has an element type as queue of `int`, on which the operators < and > are not defined. Therefore, the `with` clause is mandatory.

- When the `with` clause is specified, the relational operators (<, >, ==) are defined for the type of the expression.

  Expressions are not expected to be boolean, except for the `find` locator methods. So, we may have expressions of the form (`item.size() + 5`), where `item` may be a queue. And because the relational operators (<, >, ==) apply to the entire expression, such an expression is supported.

Example 6-26 Using Array Locator Methods

1.

```
module top;
  int myAA[*], qi[$];
  initial begin
      myAA[2] = 2;
      myAA[4] = 1;
      myAA[8] = 0;
      myAA[16] = 100;
      myAA[32] = 344;
      myAA[64] = 344;
      myAA[128] = 35;
```

```
      myAA[256] = 100;
      myAA[512] = 456780;
      myAA[1024] = 345;

      qi = myAA.find(x) with (x < 6); // Returns a queue of {2,1,0}
      qi = myAA.find_first with (item == 100); // Returns a queue of {100}
      qi = myAA.find with (item == 100); // Returns a queue of {100,100}
      qi = myAA.find_last with (item > 340); // Returns a queue of {345}
      qi = myAA.find_first_index with (item == 35); // Returns a queue of {128}
      qi = myAA.find_first_index(s) with (s > 456700); // Returns a queue of
                                                       // {512}
      qi = myAA.find_last_index with (item == 344); // Returns a queue of {64}
      qi = myAA.find_first_index(s) with (s == 100); // Returns a queue of {16}
  end
endmodule
```

2.

```
int f[6];   // values: 1, 6, 2, 6, 8, 6
int d[];    // values: 2, 4, 6, 8, 10
int q[$];   // values: 1, 3, 5, 7
int tq[$];

tq = q.min();     // {1}
tq = d.max();     // {10}
tq = f.unique();  // {1, 6, 2, 8}
```

3.

```
module arr_me;

string SA[10], qs[$];
int IA[int], qi[$];

initial begin
SA[1] = "Bob";
SA[2] = "Abc";
SA[3] = "Henry";
SA[4] = "John"};
SA[5] = "Bob";
IA[2]=3;
IA[3]=13;
IA[5]=43;
IA[8]=36;
IA[55]=237;
IA[28]=39;

// Find smallest item
qi = IA.min;       // 3

// Find string with largest numerical value
qs = SA.max with ( item.atoi ); // to test

// Find all unique string elements
qs = SA.unique;    // { "Bob", "Abc", "Henry", "John" }

// Find all unique strings in lowercase
qs = SA.unique( s ) with ( s.tolower ); // { "bob", "abc", "henry", "john" }

end
endmodule
```

Table 6-2 Array Reduction Methods

Method	Description
and	Returns the bitwise AND (&) of all of the array elements or, if a with clause is specified, returns the bitwise AND of the values yielded by evaluating the expression for each array element.
or	Returns the bitwise OR (|) of all of the array elements or, if a with clause is specified, returns the bitwise OR of the values yielded by evaluating the expression for each array element.
xor	Returns the bitwise XOR (^) of all of the array elements or, if a with clause is specified, returns the bitwise XOR of the values yielded by evaluating the expression for each array element..

Array reduction methods may be applied to any unpacked array of integral values to reduce the array to a single value. The expression within the optional with clause is used to specify the values to use in the reduction. The values produced by evaluating this expression for each array element are used by the reduction method. This is in contrast to the array locator methods where the with clause is used as a selection criteria.

The method returns a single value of the same type as the array element type or, if specified, the type of the expression in the with clause. The with clause may be omitted if the corresponding arithmetic or boolean reduction operation is defined for the array element type. If a with clause is specified, the corresponding arithmetic or boolean reduction operation is defined for the type of the expression.

Example 6-27 Using Array Reduction Methods

```
byte b[] = { 1, 2, 3, 4 };
int y;
y = b.xor with ( item + 4 ); // y becomes 12 => 5 ^ 6 ^ 7 ^ 8
```

Limitations on Array Methods

- In the current release, array manipulation methods are supported only for queues, dynamic arrays, and associative arrays.
- As per the SystemVerilog LRM, the index locator methods for associative arrays are not allowed for wildcard-indexed associative arrays.

Array Equality Operators

The following operators are allowed on queues and dynamic arrays:

- Logical equality operators (==, !=)

  If, due to unknown or high-impedance bits in the operands, the relation is ambiguous, the result of logical equality and logical inequality operators (== and !=) is a 1-bit unknown value (x).

- Case equality operators (===, !==)

  For the result of the case equality and case inequality operators (=== and !==) to be equal, bits that are x or z must be included in the comparison and match. The result of these operators always is a known value, either 1 or 0.

If the two operands of a comparison operator are aggregate expressions, they should be of equivalent type. Dynamic array, associative array, and queue types are equivalent if they are the same kind of array (dynamic, associative, or queue), have equivalent index types (for associative arrays), and have equivalent element types.

If the types are not equivalent, an error message is issued. For example, if a dynamic array is compared with a queue, the simulator reports an error.

Example 6-28 Using Array Equality Operators

1.

```
module top;
int da[];
int da1[];
  da = new [10];
  da1 = new[10];
  if (da == da1)
    $display (“equal”);
  else
    $display (“unequal”);  // output is unequal as the entries in da have value X
  if (da === da1) //case equality operator,this  will match up x’s also.
    $display (“equal;”);  // output equal will be printed.
  else
    $display (“unequal”);
```

2.

```
Class C;
  Int a;
endclass
  C q[$];
  C q1[$:6];
  Initial begin
    q[0]  = new;
```

```
    q[0].a = 45;
    q1[0] =new;
    q1[0].a = 45;
    if (q == q1)
      $display ("equal");
    else
      $display ("unequal");  // output is unequal as the entries in q have different
                                class handles
    q1[0] = q[0];
    if (q == q1)
      $display ("equal");   // output is equal as the entries in q have same
                                class handles
    else
      $display ("unequal");
```

3.

```
string aa[int];
string aa1[int];
initial begin
  aa[0] = "CADENCE";
  aa1[10] = "CADENCE"
  if (aa == aa1)
    $display ("equal");
  else
    $display ("unequal");  // output is unequal as the entries in aa have
                              different index
  aa1[0] = "CADENCE";
  aa1[10] = "";
  if (aa == aa1)
    $display ("equal");    // output is equal as the entries in aa have the same
                              data and index values
  else
    $display ("unequal");  //
```

Arrays as Function Return Types

In SystemVerilog, you can declare and define functions that return queues, associate arrays, and dynamic arrays, both single and multi-dimensional arrays.

The following multi-dimensional arrays are the currently supported as function return types:

- Dynamic arrays of fixed arrays
- Fixed arrays of dynamic arrays
- Fixed arrays of dynamic arrays of fixed arrays
- Dynamic arrays of queues

In addition, the following multi-dimensional arrays are supported when you provide the `-SVEA` option to the simulator:

- Queues of associative arrays

- Fixed arrays of queues

Note: If there is a current limitation on data types for these arrays, that limitation also applies to the return values of functions.

Example 6-29 Functions Returning Queues, Dynamic Arrays, and Associative Arrays

```
module top ();

typedef int ida[];
typedef int iqa[$];
typedef int iaa[*];

ida d;
iqa q;
iaa a;

initial begin
  d = retDA(5);
  q = retQA(5);
  a = retAA(5);
end

function ida retDA(int size);
  static ida ld;
  /*function body manipulating ld*/
  return ld;
endfunction

function iqa retQA(int size);
  static iqa lq;
  /*function body manipulating lq*/
  return lq;
endfunction

function iaa retAA(int size);
  static iaa la;
  /*function body manipulating la*/
  return la;
endfunction

endmodule
```

Example 6-30 Functions Returning Multi-Dimensional Arrays

```
module top ();

typedef int ifa[5][$];
typedef int ida[][$];

ifa f;
ida d;

initial begin
  f = retFA(5);
  d = retDA(5);

end

function ifa retFA(int size);
  static int lf[5][$];
  /*function body manipulating lf*/
  return lf;
endfunction
```

```
function ida retDA(int size);
  static int ld[][$];
  /*function body manipulating ld*/
  return ld;
endfunction

endmodule
```

Debugging Queues and Arrays

For information about how to debug queues and arrays using the Tcl command-line interface or the SimVision analysis environment, refer to *SystemVerilog in Simulation*.

7

Data Declarations

Value Parameters

For value parameters, the value is defined during elaboration and persists throughout simulation. Value parameters are defined with data types and default values. Data parameters can be defined using the non-ANSI style:

```
parameter data_type parameter_name = value;
```

For example:

```
module m();
  parameter type tp = c1; // Where c1 is a previously-declared class data type
endmodule
```

Value parameters can also be defined using the ANSI style for classes or any design unit:

```
#(parameter data_type parameter_name = value);
```

For example:

```
class elementManager #(parameter integer elementCount = 100, parameter type
  elementType = int);
  ...
endclass
```

Note: The `parameter` keyword is optional in the port list format:

```
class elementManager #(integer elementCountB = 50);
...
```

The current release supports only integral data types for value parameters. Integral data types include `shortint`, `int`, `longint`, `byte`, `bit`, `logic`, `reg`, `integer`, `time`, packed array data types, packed structure data types, and enumeration data types.

Value parameters cannot be declared with a data type that, in any way, references a type parameter.

Type Parameters

The current release supports type parameters, which are described in the IEEE standard. For a list of limitations, refer to "Limitations on Type Parameters" on page 125.

Type parameters are parameters that specify a data type, using the keyword `type`. The parameter is bound to the specific data type during elaboration and throughout simulation.

Notes:

- Type parameters are not local parameters; they cannot be preceded by the keyword `local`. Furthermore, type parameters are not specparams; they cannot be placed within `specify` blocks or preceded by the prefix `spec`.
- A `defparam` statement cannot assign to a type parameter.

  Note: The simulator supports the `defparam` statement only for parameters and `defparam` expressions that were legal in Verilog. The `defparam` statement is not supported for SystemVerilog data types.
- In an assignment to, or override of, a type parameter, the right-hand expression represents a data type.

Defining Type Parameters

Type parameters can be defined in any design unit, such as modules, packages, interfaces, and programs; as well as in any block, such as tasks, functions, `begin...end`, and `fork...join`. Because packages and blocks cannot be overridden, defining type parameters within them might not be useful.

Type parameters can be defined using the non-ANSI style:

```
parameter type parameter_name = data_type;
```

For example:

```
module m();
  parameter type tp = c1; // Where c1 is a previously-declared class data type
endmodule
```

Type parameters can also be defined using the ANSI style for classes or any design unit:

```
#(parameter type parameter_name = data_type);
```

For example:

```
module elementManager #(integer elementCount = 100, parameter type
  elementType = int);
  ...
```

```
  var elementType elements[elementCount];
  ...
endmodule
```

Note: The `parameter` keyword is optional in the port list format.

In this example, the `elementManager` module defines a value parameter called `elementCount`, and a type parameter called `elementType`. The `elements` variable is also declared; this variable is an array whose size is given by the `elementCount` parameter, and whose data type is given by the `elementType` parameter.

When this module is instantiated, the data type will be associated with its type parameter. For example:

```
module test();
  class C;
  ...
  endclass

  // Following overrides value and type parameters
  elementManager #(200, C) inst();

  // Following provides overrides using named assignments
  elementManager #(.elementType(int), .elementCount(200)) inst2();

  // Following uses default values
  elementManager #() inst3();
endmodule
```

The `inst` instance of the `elementManager` module assigns the value of 200 to the `elementCount` parameter, and associates the `C` data type with the `elementType` parameter. Within this instance, the elements variable is an array of 200 class handles for class `C`.

Additional Examples

You can use type parameters as the data type of a type parameter:

```
parameter type tp1 = C; // C is a previously declared class data type
tp myvar; // Variable using type parameter as data type
```

You can use type parameters as an array index type:

```
parameter type tp2 = C2; // Where C2 is a previously declared class data type
tp1 aa[tp2]; // Used as the element and array index type
```

You can use type parameters within the built-in `mailbox` class:

```
mailbox #(tp1) mbox; // Uses type parameter in a mailbox
```

Limitations on Type Parameters

In the current release:

- All assignments to type parameters must resolve to integral types, class data types, strings, or real types; they cannot resolve to unpacked structs, queues, dynamic or associative arrays, fixed arrays, mailboxes, or semaphores.
- Type parameters of non-class data types can be used as elements of a queue, dynamic or associative array, index of an associative array, or mailbox.

  Type parameters of non-class data types cannot be used as members or elements of a vector or static array. They cannot have built-in methods called on them, such as `str.len()`.
- References to type parameters are restricted to type parameter assignments and overrides.
- Variables declared with a type parameter as their data type cannot be used
  - Within a clocking block as an input, output, or inout declaration
  - In OOMRs
- Type parameters can be used as class data types, but not as class templates. For example, the following results in an error:

```
class C4#(b = 1);
...
endclass
parameter type tp5 = C4;
tp5 #(37) myvar2; // Unsupported
```

Const Constants

In SystemVerilog, you can declare a variable as a constant by using the `const` keyword. Constants declared as `const` are assigned after elaboration. The `const` keyword is supported for variable declarations, in `ref` arguments, and on class properties.

The current release enforces the following LRM limitations:

- The value of a `const` variable can only be set in an initializer on the variable declaration. Assigning a `const` variable in any other context will result in an error.
- Assigning to a `const ref` argument will result in an error.

Example

```
typedef union packed {
  struct packed {
  byte b1;
  byte b2;
  } by_byte;
  bit [16:0] B16;
```

```
  } u_1;

// Following assignment causes an error, because it assigns to a
// const ref argument

function automatic void myfunc (const ref int fi);
  fi = 12;  // Error
endfunction

module test();

  const logic [3:0] d =4'ha; // Packed variable const
  const u_1 u1 = 16'habcd; // Union const
  event e1;
  const event e2 = e1;  // Named event const
  logic [3:0] b  = 4'h0;

// Following assignments cause an error, because they are made outside of an
// initializer of the variable declaration

initial begin
  d[0] = b[0];  // Assignment through bit select
  d[2:1] = b[2:1];  // Assignment through part select
  d = 4'ha;  // Packed variable
  u1.B16 = 16'hfedc; // Assignment through union member
  e2 = null; // Event
  myfunc(10);
end

endmodule
```

When you run this example, you get error messages for assigning to a `const` ref argument, and for assigning a value to a constant variable outside of variable initialization:

```
irun non_class_const_support.sv
...
d[0] = b[0];  // Error detected, bit select
ncvlog: *E,CONASN (non_class_const_support.sv,20|2): Constant variable cannot
be assigned outside of an initialization.
...
fi = 12;  // Error detected, const ref arg
|
ncvlog: *E,CONASN (non_class_const_support.sv,8|3): Constant variable cannot
be assigned outside of an initialization.
```

Declaring Variables with Initializers

Variables can be declared in the following ways:

```
int myint, myint2; // Short form; data type followed by the instances
var myvar; // Using the keyword var. Data type is optional.
           // Defaults to type logic.
int a = 0; // With an initializer
```

In SystemVerilog, initialization values for static variables are executed *before* `initial` or `always` blocks. This behavior is unlike Verilog, where static variables behave like any other `initial` block. Also, initializers in SystemVerilog do not need to be simple constants.

In the current release, static variables with an initializer that are declared within a procedural scope—such as tasks, functions, and named/unnamed blocks—must use the `static` keyword:

```
task mytask;
static int b = 1; // Static keyword is required
...
endtask
```

This limitation was introduced in the Accellera 3.1a LRM, but was removed in the IEEE 1800 standard.

Declaring Local Variables in Unnamed Blocks

In Verilog-2001, you can declare local variables in named `begin...end` or `fork...join` blocks. A local variable declared in a named block can be referenced by using a hierarchical path. For example, in the following code, there are two variables named `i`, which are referenced by the hierarchical names `foo.i` and `foo.loop.i`.

```
module foo (...);
integer i;
...
initial
  begin: loop
    integer i;
    for (i = 0; i < 10; i = i + 1)
      begin
      ...
      end
  end
```

In SystemVerilog, you can declare local variables in unnamed blocks, as well as in named blocks. For example:

```
initial
  begin
    integer i;
    for (i = 0; i < 10; i = i + 1)
      begin
```

```
    ...
    end
  end
```

Variables declared in an unnamed block are visible to the unnamed block and any nested blocks below it. However, because the block has no name, the variables cannot be referenced using hierarchical paths.

Continuous Assignments to Variables

In Verilog-2001, a net can be written by one or more continuous assignments or primitive outputs, or through module ports. The left-hand side of a continuous assignment can only be a net data type, such as `wire`. The continuous assignment is a *driver* of the net, and nets can have any number of drivers. A net cannot be procedurally assigned. Variables, on the other hand, cannot be used on the left-hand side of continuous assignments. Variables can only be used on the left-hand side of procedural assignments.

SystemVerilog removes this restriction and permits continuous assignments to variables, in addition to nets. You can assign to a variable with a continuous assignment, and whenever any of the inputs in the right-hand side expression of the assignment changes, the expression is evaluated and the result becomes the new value of the variable.

In the following example, the `answer` port is a variable that is the target of a continuous assignment. Whenever `left` or `right` changes value, the `answer` variable is updated automatically with the new value of `left & right`.

```
module and2(answer, left, right);
  output logic answer;
  input wire left;
  input wire right;

  assign answer = left & right;
endmodule
```

Note: Using the `logic` data type on a port can affect optimization and performance.

A variable can also be connected to an output port in a module instantiation, or to the output of a primitive. Semantically, this connection acts as an implicit continuous assignment, in which the variable is being continuously assigned the value of the output port.

In the following example, the `result` variable in the `top` module is passed to an output port in a module instantiation. There is no explicit continuous assignment to `result`, but one is implied by the connection to the output port in the module instantiation. The `result` variable is automatically updated with each change in the value of the `answer` output port in the instanced module.

```
module and2(answer, left, right);
  output answer;
```

```
  ...
endmodule

module top;
  wire left;
  wire right;
  reg result;
  and2 and2 (result, left, right);
endmodule
```

The SystemVerilog LRM specifies that a variable can have only a single source for its value. If a variable is used on the left-hand side of a continuous assignment, that assignment is the only one permitted for that variable. See "Restrictions on Continuous Assignments to Variables" on page 130 for more information.

See "Limitations on Continuous Assignments" on page 132 for information about restrictions on continuous assignments to variables in the current release.

Restrictions on Continuous Assignments to Variables

The LRM states that you cannot have multiple continuous assignments to the same variable, mix continuous assignments and procedural assignments for the same variable, or use a variable on the left-hand side of a continuous assignment and connect the same variable to the output port of a module.

For an atomic variable—that is, a scalar or real variable—two rules apply:

- There can be at most one continuous assignment to the variable.

```
logic v1;
...
assign v1 = 0;   // Continuous assignment to variable v1
assign v1 = 1;   // Error--second continuous assignment to v1
...
```

- If there is a continuous assignment to the variable, the variable cannot also have any procedural assignments—blocking or non-blocking procedural assignments, procedural continuous assignments, or declared variable initialization.

```
logic v2;
...
assign v2 = 0;  // Continuous assignment to variable v2
initial
   begin
     v2 = 1;    // Error--mix of continuous and procedural assignments
                // for v2.
   end
...

logic v3, v4;
...
assign v3 = 0;  // Continuous assignment to variable v3
always @(v4)
```

```
   v3 <= v4;     // Error--mix of continuous and procedural assignments
                 // for v3.
...

logic v7 = 1;    // Treated as a procedural assignment
...
assign v7 = 0;   // Error. Mix of continuous and procedural assignments
                 // for v7
...
```

For whole vectors, the rules for scalars given above apply. If the vector is not updated as a whole, each element of the vector can have its own continuous assignment. No element can have multiple continuous assignments. It is illegal to use both continuous assignments and procedural assignments to the elements of a vector.

```
logic [1:0] v1;
...
// This is legal: one continuous assignment for index 1 and another for index 0.
assign v1[1] = 0;
assign v1[0] = 0;
...

logic [1:0] v2;
wire w;
...
assign v2[1] = 0; // Continuous assignment for index 1
always @(w)
  v2[2] <= w;     // Error; mix of continuous and procedural assignments
                  // for the elements of v2, even though the vector indexes
                  // are different.
...

parameter integer p = 1;
logic [1:0] v3;
...
assign v3[p:0] = 0;   // Counts as a continuous assignment for each element
                      // of the part select.
assign v3[1] = 0;     // An error if the value of parameter p after elaboration
                      // is 1, because index 1 will then have multiple
                      // continuous assignments.
```

For unpacked arrays, the rules are similar to those for vectors, with the exception that the individual array elements can be updated in different ways. For example, one element might be updated with a continuous assignment, while another element might be the subject of a procedural assignment. For any individual array element, however, at most one continuous assignment can be used. A continuous assignment cannot be combined with a procedural assignment.

```
logic v1[1:0];
...
// This is legal: one continuous assignment for index 1 and one for index 0.
assign v1[1] = 0;
assign v1[0] = 0;
...

logic v2[1:0];
wire w;
```

```
...
assign v2[1] = 0;  // Continuous assignment for index 1
always @(w)
   v2[2] <= w;     // Legal; mixing continuous and procedural assignments
                   // for the elements of v2 is OK, as long as the indexes
                   // are different.
...

parameter integer p = 1;
logic v3[1:0];
...
initial
   begin
      v3[p] = 0;  // Procedural assignment
   end
assign v3[1] = 0; // An error if the value of parameter p after elaboration is 1,
                  // because index 1 will have a procedural assignment and a
                  // continuous assignment.
```

Limitations on Continuous Assignments

In the current release, there are two restrictions on the implementation of continuous assignments to variables.

- Delays on continuous assignments to variables are not supported. Any delay specified in a continuous assignment is ignored.
- If a variable is driven by a continuous assignment and the variable is forced and then released, the continuous assignment is not immediately re-evaluated upon release. The variable retains its existing value, in the same way that the variable will if it is driven by procedural assignments rather than by a continuous assignment.

Automatic Design Unit Qualifier

You can use the `automatic` keyword with module, program, interface, and package declarations. The `automatic` qualifier causes all tasks and functions in that design unit to be treated as automatic by default, unless explicitly declared to have static storage class. All variables in those tasks and functions are also as automatic by default.

Because IES does not support automatic variables outside of tasks and functions, variables declared in procedural blocks, in `initial` and `always` blocks remain static by default. The compiler issues a warning when such a variable is declared as automatic.

Note: The *ncdc* decompiler does not decompile the qualifier on the design unit. Instead, it decompiles the design as it was treated by IUS, with the qualifier added to each of the tasks and functions that was treated as automatic.

8

Classes

SystemVerilog introduces a new `class` data type, which is used in object-oriented (OO) programming. A class is a user-defined data type that can encapsulate data members and methods. Data members and methods are used together to define the functionality and characteristics of an object.

Classes bring the following aspects of OO programming to SystemVerilog: inheritance, encapsulation, polymorphism, and abstract-type modeling.

This chapter provides a basic overview of SystemVerilog classes. For more information, see the IEEE 1800 standard.

Note: Classes are supported within the Incisive Enterprise Simulator - XL (IES-XL). However, classes are not available with the Incisive Enterprise Simulator - L (IES-L).

Declaring a Class Data Type

The following is an example of a simple class:

```
module myModule;
   class MyClass;
      integer p1;

      task myTask(input integer i);
        p1 = i;
      endtask

      function integer myFunc();
        myFunc = p1;
      endfunction
   endclass:MyClass
endmodule
```

This example defines a class called `MyClass`, which has one data member called `p1` and two methods called `myTask()` and `myFunc()`.

Note: In the current release, classes cannot be declared globally. Class data types and class variables can be declared within a module or package. For more information about packages, see “Packages” on page 243.

To use `MyClass`, you must create a variable using `MyClass` as its data type. For example:

```
module test_top;

   class MyClass;
   ...
   endclass:MyClass

   // Creates variables of MyClass type
   MyClass c1;
   ...

endmodule
```

This example creates a class variable called `c1`, whose value is an *instance handle* to an instance of class `MyClass`. An instance handle is a value that points to or represents a particular class object. A class object can have only one instance handle, and an instance handle can point to only one class object. However, multiple class variables can have the same instance handle.

Variables must be initialized. For example:

```
module test_top;
   class MyClass;
   ...
   endclass:MyClass
   MyClass c1;
   initial begin
      // Initializes the variable
      c1 = new;
   end
   ...
endmodule
```

This example uses the `new` function to initialize the `c1` variable to an instance of the `MyClass` class.

Tip

In SystemVerilog, uninitialized variables are given a default value of null. To determine whether a variable is uninitialized, you can check its value versus null. For example:

```
if (c1 == null) c1 = new;
```

For more information about the `new` function, see "Working with Constructors" on page 135, and the IEEE 1800 standard.

You can create a class variable and initialize it at the same time. For example, the following statement declares and initializes a variable:

```
MyClass c1 = new;
```

You can use the variable to access class data members and methods within a class object. For example, the following shows different ways to use variable `c1` to access commands within `MyClass`:

```
c1.p1 = 5;      // Accesses data member p1 in MyClass
c1.myTask(2);   // Accesses myTask() in MyClass and assigns 2 to input b
```

Note: Refer to the IEEE 1800 standard for information about how you can declare static class data members and methods.

Working with Constructors

This section describes the `new` function, which was introduced in "Declaring a Class Data Type" on page 133. The `new` function, which is also called a *class constructor*, is used to initialize an object at the time of its declaration. For example:

```
ClassA c = new;
```

This declaration creates a new instance of the `ClassA` class, and initializes the instance using the `new` function. Every class has a built-in `new` function. However, you can also specify any special initialization requirements by defining your own `new` function within a class. For example:

```
class ClassA;
    ...
    integer a;
    function new();
      a = 0;   // Insert special initialization here
    endfunction
endclass:ClassA
```

You can also customize an instance at run time by passing arguments to a `new` function. For example:

```
ClassA c = new(5, 2);
```

where the corresponding class declaration might look like the following:

```
class ClassA;
    integer a1, a2;
    function new(input int arg1, input int arg2);
      a1 = arg1;
      a2 = arg2;
    endfunction
endclass:ClassA
```

Calling virtual functions from constructors can cause unexpected results.

Inheritance

In SystemVerilog, you can create new classes that are based on existing classes. For example, the following declares a class called `ExtClass` that contains a variable of `Class_AB`:

```
class Class_AB;
   integer p1;
   task t (input integer i);
    p1 = i;
   endtask

   virtual function integer f();
     f = p1;
   endfunction
endclass: Class_AB

class ExtClass;
   Class_AB ext_ab;   // Variable of type Class_AB
   ...
endclass:ExtClass
```

Although this example is legal, SystemVerilog offers a more efficient way to extend a class—through the `extends` keyword. When you extend a class, the derived class inherits all of the data members and methods of the parent class. However, a derived class can add its own data members and methods. In the following example, `ExtendClass` extends `Class_AB`, inherits its data members, and adds properties of its own:

```
class ExtendClass extends Class_AB;
   integer p2;
   task t (input integer i);
     p1 = 2 * i;
     p2 = 4 * i;
   endtask

   virtual function integer f ();
     f = this.p1 + 1;
   endfunction
endclass:ExtendClass
```

Virtual class members are described in "Protecting Class Members" on page 136.

Note: The IEEE 1800 standard describes `$cast` dynamic casting, which is used to cast a handle from a base class to a derived class. The current release supports only `$cast` dynamic casting on classes.

Protecting Class Members

This section describes the keywords you can use to protect data members against accidental modification.

- `local`—Local members are available only to methods within the same class. Local members are not available to subclasses, but can be accessed from different instances within the same class. For example (taken from the LRM):

```
class Packet;
   local integer i;
   function integer compare (Packet other);
      compare = (this.i == other.i);
   endfunction
endclass
```

  For more information about the `local` keyword, see the IEEE 1800 standard.

- `protected`—Protected members are similar to local members, but are visible inside subclasses.

  For more information about the `protected` keyword, see the IEEE 1800 standard.

- `const`—Use the `const` keyword to declare read-only members. For example, the following defines `a1` as a `const` and as a protected member:

```
class ObsC;
    const protected int a1;
    ....
endclass
```

  There are two types of `const` declarations: *global* and *instance*.

  - Global constants assign a value to a member at the time of the declaration. For example, the following declares `const a1` with an initial value as a part of its declaration:

```
class ObsC;
    const protected int a1 = 2;
    ....
endclass
```

    A global `const` can be assigned a value only within its declaration.

  - Instance constants assign a value within the `new()` function—also called the constructor—of the class. For example:

```
class ObsC;
    const protected int a1;
    ...
    function new();
      a1 = 2;
    endfunction
endclass
```

  For more information about the `const` keyword, see the IEEE 1800 standard.

Note: These keywords do not have a predefined order. However, you can specify them only once per member, and you cannot have a member that is declared both as `protected` and as `local`.

Abstract Classes and Virtual Methods

Abstract classes and virtual methods are described in the IEEE standard.

Use the `virtual` keyword to specify that a class can be extended by other classes, but cannot be instantiated. For example, the following declares a virtual class called `virClass` that can be extended by derived classes, but cannot be instantiated:

```
virtual class virClass;
   ...
endclass
```

The IEEE 1800 standard also describes how a virtual class can provide only a prototype for a virtual method, by specifying it without a body. For example:

```
virtual class BasePacket;
  virtual function integer send(bit[31:0] data); // Provides a prototype
  endfunction
endclass

class EtherPacket extends BasePacket;
  function integer send(bit[31:0] data);
  // Body of declaration
  ...
  endfunction
endclass
```

Prior to IUS 8.2, the simulator implemented this functionality, in that it treated virtual methods without declarations or statements as prototypes. Starting with the 8.2 release, the simulator treats virtual methods as prototypes only when they use the `pure virtual` keyword.

For example:

```
virtual class BasePacket;
  pure virtual function integer send(bit[31:0] data); // Provides a prototype
endclass

class EtherPacket extends BasePacket;
  function integer send(bit[31:0] data);
  // Body of declaration
  ...
  endfunction
endclass
```

The `pure` keyword must be used before the `virtual` keyword. Any other ordering will produce an error. The `pure virtual` syntax is set to become a part of the next IEEE standard for SystemVerilog.

Parameterized Classes

Much like functions and modules, SystemVerilog classes can be parameterized. Parameterized classes eliminate having to write code repeatedly for objects that are similar to each other, but might have different parameters. Classes can be parameterized by using

- Value Parameters
- Type Parameters (limitations apply; refer to "Limitations on Type Parameters" on page 123)

You can instantiate instances of a parameterized class as you can for modules or interfaces.

Declaring Parameterized Classes

A parameterized class declaration is called a *generic* class, which is similar to template classes in C++. Parameterized class declarations have parameter port lists. Each parameter declaration within that port list has a default value. You can use the typical Verilog module instantiation and parameter override, by position or name, to provide parameter values. If a specialization does not override a particular parameter, the parameter takes its default value as defined in the class declaration.

For example, the following defines a generic `register` class:

```
class register #(parameter int i = 5);
...
endclass: register
```

In the current release, the `parameter` keyword is optional, and you can declare a parameter port list as `#()`:

```
class registerB #(int size = 8);
...
class registerC #();
...
```

You can instantiate the `register` class with new parameter values, creating a new data type. The combination of the generic class and its actual parameter values is called a *class specialization* or *variant*. Specializations use the following syntax:

```
<class_name> #() id;
```

The following defines two specializations of the `register` class:

```
register #() r1 = new;  // Specialization of register using default
                        // parameter values
register #(7) r2 = new; // Specialization of register where i = 7
```

The current release also supports a non-standard syntax for creating specializations:

```
class_name id;
```

For example:

```
register r3;
```

Example: Creating Class Specializations

The following module contains a generic `vector` class, which includes a `size` parameter. When the class is instantiated with new parameter values, it is called a *class specialization*.

```
class vector #(int size = 1);
  bit [size-1:0] a;
endclass
module test();
  vector vi = new;
  vector #(10) vten;           // Object with vector of size 10; uses parameter
                               // association by position
  vector #(.size(2)) vtwo;     // Object with vector of size 2; uses parameter
                               // association by name
  vector #() vone;             // Object with default vector of size 1

  vector voneNS;               // Object with default vector of size 1,
                               // uses non-standard syntax

endmodule
```

Class specializations can be used with `typedef` statements. They can also be used to create class variables:

```
typedef vector #(10) Vten;  // Class with vector of size 10
typedef vector #() V_one;    // Class type with vector of size 1 (uses default)
vector #(4) var1; // Class variable var1 of data type vector #(4)
```

A `typedef` statement that designates a class specialization can be used as the data type for a class variable:

```
typedef vector #(4) class_spec_4_t;
class_spec_4_t var2;
```

Extending Parameterized Classes

You can extend a parameterized class using the `extends` keyword. The `extends` *`class_type`* declaration can be a class specialization, a nested class specialization, or a class specialization of a class that is declared within a package. The class from which a class specialization extends can be a parameterless class type, or a specialization of a class with parameters.

Example: Extending Parameterized Classes

In the following example, class `C` extends `D#(4)`, which is a class specialization of a class that is declared within a package called `types`. The `D#(4)` specialization sets the value of parameter `l` to `4` so, for class `C` and any specialization of `C`, the value of parameter `l` will always be `4`.

```
package types;
  class D #(parameter l = 0);
  int md = l;
  endclass

  class C #(parameter logic [6:0] p = 0) extends D #(4);
   logic [p:0]mc = p;
    function new();
     $display ("mc = %d, p = %d", mc, p);
    endfunction // New
  endclass
endpackage

module top;
  import types::*;
  class A extends C #(2);
   task print();
     $display ("md from base D = %d", md);
   endtask
  endclass

  A a = new;

  initial begin
   $display ("a.md = %d", a.md);
   a.print();
   a.md = 8;
   a.print();
  end
endmodule
```

When this example is simulated, it produces the following output:

```
% irun test.sv
...
ncsim> run
mc = 2, p =   2
a.md =           4
md from base D =           4
md from base D =           8
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

Parameter assignments of `D` specializations can reference any parameters in the parameter list of `C`. For example, the following declares class `E`, which extends a specialization of `D`. The specialization `D #(q)` references parameter `q`, which is in the parameter port list of `C`:

```
class E #(parameter p = 1) extends D #(p);
..
endclass
E #(4) e4; // E specialization, where q = 4 and p = 4
```

The current release does not support OOMRs as specialization values for the `extends` class type.

```
E #(u1.q) voomr;          // Invalid. Contains OOMR.
```

For more information about the `typedef` class within a parameterized class, refer to "Limitations on Parameterized Classes" on page 144.

Static Variables and Class Specializations

When a parameterized class contains a `static` variable, each specialization of that class will have its own unique copy of the variable. For example:

```
class vector #(int size = 1);
  bit [size-1:0] a;
  static int count = 0;
  function void disp_count();
   $display ("count: %d of size %d", count, size);
  endfunction
endclass
```

In this example, the `count` variable can be accessed only through the `disp_count()` method. Each specialization of the `vector` class has its own unique copy of `count`.

Example: Sharing Static Variables among Class Specializations

The current release supports using non-parameterized base classes to share static members and methods among different class specializations. In the following example, static member `count` is in a non-parameterized class called `base`, and is shared among variables `ci`, `di`, `ei`, and `fi`, regardless of the specialization of generic class `C`.

```
class base;
 static int count;
endclass

class C #(int i = 5 , string str = "8.1") extends base ;
 function void Cfunc();
  count = count + i ;
  $display ("i:%d, str:%s, count:%d",i,str,count);
 endfunction
endclass

module test #(int m =2);

 int val = 8;

 C ci = new;
 C #(.str("8.2")) di = new;
 C #(.i(7)) ei = new;
 C #(.i(5)) fi = new;

 initial begin
  ci.Cfunc;
```

```
  #5  di.Cfunc;
  #10  ei.Cfunc;
  #15  fi.Cfunc;
 end

endmodule
```

When this example is simulated, it produces the following output

```
% irun test.sv
...
i:          5, str:8.1, count:           5
i:          5, str:8.2, count:          10
i:          7, str:8.1, count:          17
i:          5, str:8.1, count:          22
```

Scoped Types and Expressions

You can use the class scope resolution operator (`::`) within class specializations to access identifiers within the scope of a class.

For example, if `classA` contains data type `dataB`, you can access `dataB` from outside `classA` using `classA::dataB`. Class specializations can use the scope resolution operator to select scoped data types.

Example: Selecting Scoped Data Types within Class Specializations

The following example shows how to access a scoped data type from a class specialization. It declares a parameterized class `C`, which has a data type called `myvect`.

```
class C #(parameter int lb =  7);
  typedef bit [lb:0]myvect;
  ...
endclass

module test();

  C #(15) :: myvect a; //a is a 15-bit vector
  C #(.lb(31)) :: myvect b; //b is a 31-bit vector

  initial begin
    $display("a is %b",a);
    $display("b is %b",b);
  end
endmodule
```

This example produces the following results:

```
% irun test.sv
...
a is 0000000000000000
b is 00000000000000000000000000000000
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

Class Specialization Type Checking

Class specialization data types are subject to type checking. In SystemVerilog, class data type assignments are unidirectional.

You can assign a derived class type to any of its parent classes. However, you must use dynamic casting to assign an object of a parent class type to an object of a derived class type. To assign one class type specialization to another class specialization of the same class declaration, the specializations must match. Class specializations match if the following are true for each parameter in the specializations:

- If the parameter is a type parameter, the types in each class specialization are matching types.
- If the parameter is a value parameter, the parameter type and value are the same in each class specialization.

All matching specializations of a generic class represent the same data type. The set of matching specializations of a generic class are defined by the context of the class declaration. If the generic class is declared within a package, it is visible throughout the whole system, and all of the matching specializations for that generic class are of the same type. For other contexts, such as modules and programs, each instance of the scope in which the generic class declaration is contained creates a unique generic class, which defines a new set of matching specializations.

Limitations on Parameterized Classes

The following summarizes the parameterized class features in the LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- OOMRs to variables of a class specialization data type are not supported.
- The operators currently allowed on non-parameterized classes—assignment, equality, inequality, and the conditional operators—are also supported for class specializations.
- The parameter expression within a parameterized `typedef` statement must be a single expression. It cannot contain a `min:typ:max` expression.
- You cannot specialize a class data type that has already been specialized. For example:

```
class vector #(int size = 1, type T = bit);
 T [size-1:0] a;
endclass

typedef vector #(10) Vtenbit;
typedef Vtenbit #(.T(logic)) Vtenlogic; // Invalid
```

- The following declarations, listed in the LRM, are not supported within a parameterized class declaration:
  - `class_constraint`
  - `covergroup_declaration`
- According to the LRM, the `defparam` statement might be removed from future releases of the SystemVerilog language. The simulator supports the `defparam` statement only for parameters and `defparam` expressions that are legal in Verilog. The `defparam` statement is not supported for SystemVerilog data types.

  You cannot use `defparam` statements on class parameters.

  However, the effects of a `defparam` statement can propagate to a class parameter, if the `defparam` statement is applied to a module parameter that depends on a class parameter. In the current release, `defparam` statements are supported for module value parameters, but only if the module value parameter depends on a class data type that is declared within the same module instance. Furthermore, if a `defparam` is applied to a module value parameter, the parameter can be used only in a class data type parameter assignment list.

  For example, the following is supported:

```
module top;
  parameter p = 1;
  class C #(q = 0);
  ...
  endclass
  C#(p) h = new;
  m u1 ();
endmodule

module m;
  defparam top.p = 3; // Valid. Change in value of p propagates to C#(p),
endmodule             // which is a data type declared in the same
                      // scope as p.
```

  The following is not supported:

```
module top;
  parameter p = 1;
  class C #(q = p);
  ...
  endclass
  C#() h = new;
  m u1 ();
endmodule

module m;
  defparam top.p = 3; // Invalid. Change in value of p propagates to q,
endmodule             // which is declared in the class declaration scope.
```

  The following is not supported:

```
module top;
  parameter p = 1;
  class base #(parameter q = 0);
  ...
  endclass
  class C extends base #(p);
  ...
  endclass

  C#() h = new;
  m u1 ();
endmodule

module m;
  defparam top.p = 3; // Invalid. Change in value of p propagates to the
 endmodule            // data type base#(p), in extended class C.
```

The following is not supported:

```
module top;
  parameter p = 1;
  class C;
    logic [p:0] m1;
  endclass
  C#() h = new;
  m u1 ();
endmodule

module m;
  defparam top.p = 3; // Change in value of p propagates to [p:0],
endmodule             // which is declared in class C. Not supported
                      // because the range and parameter are in
                      // different scopes.
```

Additional Features

For more information about the following aspects of SystemVerilog classes, refer to the IEEE 1800 standard. The Cadence implementation of classes supports all of these features:

- Overridden Members—Describes how SystemVerilog classes handle overridden members.
- Super—Describes how to refer to members of a parent class from a derived class.
- Casting—Describes how to assign a subclass variable to a variable in a higher class.
- Chaining Constructors—Describes the initialization sequence within inherited classes.
- Polymorphism—Describes how a base-class variable can hold a sub-class object and reference the sub-class methods directly.
- Class Scope Resolution Operator—Describes how to uniquely identify members of a class using the `::` operator.

- Typedef Class—Describes how to declare a class variable before declaring the class itself.
- Classes and Structures—Describes the difference between classes and structures.

  Note: This section also describes global class definitions, which are unsupported in the current release. Instead, the current release supports packages. Refer to "Packages" on page 243.
- Memory Management—Describes the SystemVerilog automatic memory management system.

Limitations on Classes

The following summarizes the class features in the LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Global class definitions are unsupported. Instead, the current release supports packages. Refer to "Packages" on page 243.
- Out-of-module references (OOMRs) to class variables are not supported. For example:

```
package p;
   class class_type;
   endclass:class_type
endpackage

import p::class_type;

module top;
   class_type v;
   ...
   initial
     v = bot.c   // Not supported in the current release
   ...
endmodule

module bot (...);
   class_type c = new;
   ...
endmodule
```

  References to class variables in a package are supported. For example:

```
package p;
   class class_type;
   endclass:class_type

   class_type c;
   ...
endpackage
```

```
import p::*;

module top;
   class_type v;
   ...
   initial
     v = c   // Supported in the current release
   ...
endmodule
```

- Class variables passed to OOMR tasks or functions, or to tasks or functions declared in a package, are not supported in the current release. For example:

```
package p;
   class class_type;
   endclass:class_type
endpackage

import p::class_type;

module top;
   class_type v;
   ...
   initial
     bot.mytask(v)   // Class variable passed to OOMR task not supported
                     // in the current release
   ...
endmodule

module bot (...);
   task (input class_type arg_mytask);
   ...
   endtask
...
endmodule;
```

Debugging Classes

For information about how to debug classes using the Tcl command-line interface or the SimVision analysis environment, see *SystemVerilog in Simulation*.

9

Operators and Expressions

Verilog does not have increment and decrement operators, and lacks the C assignment operators, such as `+=`. SystemVerilog adds these operators to the language.

Supported Operators

The current release supports the following:

Assignment Operators on page 150	`=  +=  -= *=  /=  %=  &=  |=  ^=  <<= >>=  <<<=  >>>=`
Increment and Decrement Operators	`++` and `--`
Wild Equality and Wild Inequality Operators on page 150	`==? !=?`
Case Equality Operators for Real Values on page 151	`===` and `!==`
"Set Membership Operator" on page 151	`inside`
Aggregate Expressions on page 155	

Note: This table summarizes the supported operators. If an operator is supported in the current release but has limitations, a link to its documentation is provided. If the current release supports the operator without limitations, refer to the IEEE 1800 standard for its documentation.

Assignment Operators

SystemVerilog includes the simple assignment operator, =, and the following C assignment operators and special bit-wise assignment operators:

+= -= *= /= %=

&= |= ^= <<= >>=

<<<= >>>=

These operators combine an operation with the assignment. For example:

```
abc += xyz;      // Add right-hand side to left-hand side, then assign
                 // Same as xyz = xyz + abc;

i *= 3;         // Same as i = i * 3;

a[i] += 2;      // Same as a[i] = a[i] + 2;
```

You can use these assignment operators as statements, but you cannot use them in expressions.

Wild Equality and Wild Inequality Operators

SystemVerilog adds an equality operator (==?) called the *wildcard equality operator*. This operator is similar to the == operator, in that it performs different kinds of bit-wise comparisons. The ==? operator treats Xs and Zs on the right-hand side of the operation as wildcards. Bit positions with Xs and Zs on the right-hand side will automatically match as a result of a wildcard, essentially masking them out of the comparison. The remaining bit positions will participate in a logical comparison, as with ==.

The wildcard inequality operator is !=?.

Note: The Accellera 3.1a LRM introduced symmetric operators (=?= and !?=). These operators, which have been removed in the IEEE 1800 standard, have been retained in the Cadence implementation for backward compatibility, but their use is discouraged.

Examples:

```
reg [3:0] op;
op = 4'b100X;

if (op ==? 4'b1XXX)   // True
if (op ==? 4'b1000)   // Unknown
if (op ==? 4'b100X)   // True
if (op ==? 4'b000X)   // False
if (op !=? 4'b000X)   // True
```

Case Equality Operators for Real Values

Cadence extends case equality operators (`===` and `!==`) to allow real valued operands in Verilog-AMS designs. You can use the case equality operators for real values in the same manner as for logic values. Equality is determined by examining the bit pattern of the operands and returning `1` if they are equal, and `0` otherwise. Thus, an expression comparing two real values `r1` and `r2` as the following:

```
r1 === r2
```

is equivalent to the expression

```
$realtobits(r1) == $realtobits(r2)
```

and the expression

```
r1 !== r2
```

is equivalent to the expression

```
$realtobits(r1) != $realtobits(r2)
```

Example of comparing the Verilog-AMS `` `wrealXState `` and `` `wrealZState `` values:

```
module foo(in, out);
  input var real in;
  output var logic out;

    always @(in)
      begin
        if(in === `wrealZState)
          out = 1'b1;
        else if(in === `wrealXState)
          out = 1'bx;
        else
          out = 1'b0;
      end
endmodule
```

Note: Because case equality operators are not allowed on real operands by the SystemVerilog and Verilog LRM, a message is issued on their use to warn you that your HDL is not standard Verilog. This warning is printed only for modules that are not compiled with the `-ams` option.

Set Membership Operator

The set membership operator, `inside`, uses the following syntax:

```
expression inside {open_range_list}
```

The *`expression`* value can be any singular expression, which can be a real type, or any integral type, including enums, packed structs, and packed arrays.

The *open_range_list* can contain individual expressions of singular type, and ranges containing such expressions. Expressions that are aggregate arrays are not supported.

The inside operator checks the *expression* value against the value or range of values in the *open_range_list* to determine whether the expression is part of the set specified in the list. The members of the set are scanned until a match is found, then the operation returns 1'b1. Once a match is found, no further checks are made.

If no matches are found, the operation returns 1'b0. If no matches are found, but some of the compare results were X, the operation returns 1'bx.

Note: If there are X values in a comparison to a range, the simulator gives the same result for that comparison as if it had been written as ((value >= left) && (value <= right)). If either part of the comparison is 0, the result of the comparison against the range will be 0, even if the other part is X.

You can use the inside operator in any context where expressions are allowed in the language, such as constant expressions, continuous assignments, procedural code, and assertions.

For information about using the inside operator in randomization constraints, see "Set Membership" on page 185.

Limitations on Set Membership Operations

- The current release does not support event handles, class handles, chandles, and strings.
- Range bounds containing $ as a value are not supported
- Aggregate arrays are not supported as *open_range_list* values

Assignment Patterns

SystemVerilog introduces a construct called an *assignment pattern* that can be used to describe patterns of assignments to array elements and structure members. In an assignment pattern, you specify the correspondence between a collection of expressions and the elements of an array or the members of a structure. An assignment pattern uses the following syntax, which can be used on the left-hand or right-hand side of an assignment:

```
typeName'{key:value;{,key:value}}
```

or

```
'{key:value;{,key:value}}
```

Note: The current release does not support assignment patterns on the left-hand side of assignments.

An assignment pattern consists of braces, keys, and values. The whole construct is prefixed with a simple apostrophe, to distinguish the construct from a Verilog concatenation, or with a full type qualification (`typeName'`). Full type qualifications can be used wherever an expression syntax is allowed. On the other hand, you can prefix an assignment pattern with a simple apostrophe, but only when the data type can be determined by the target of the assignment pattern, or by the data being assigned. For example:

```
typedef struct packed {int a, b;} packT;
  packT p1 = packT'{a:0, b:1};   // Legal. Provides the datatype.
  packT p2 =      '{a:0, b:1}; // Legal. Datatype from LHS.
```

In the current release, assignment patterns can consist of array literals and structure literals. Array and structure literals are assignment patterns that use constant member expressions. The current release does not support assignment patterns that use variables.

There are six different techniques for specifying an assignment pattern:

- Indexed—Uses an index value to locate the member to which to assign the value. For example:

  ```
  typedef logic [1:0] t;
  ...
  t p1 = t'{1:0, 0:1};
  ```

  Associations by index value do not apply to structures.

- Type—Assigns the value to subelements with the specified data type. For example, the following assigns the value `t'{0,0}` to all subelements whose data type is `t`:

  ```
  typedef t [4:2] t2;
  t2 p2 = t2'{t:t'{0,0}};
  ```

- Default—Uses the `default` keyword to assign a value to all of the subelements of the data object that have not been matched by either an index or type key. For example:

  ```
  t p3 = '{default:1};       // Sets all elements to 1
  t p4 = '{1:0, default:1};  // Sets all elements, except the element at index
                             // 1, to 1
  ```

- Positional—Assigns values by position. The first value is assigned to the first subelement, the second value is assigned to the second subelement, and so on. For example:

  ```
  t p5 = t'{0, 1};    // Is the same as the following assignment
  t p6 = t'{1:0, 0:1};
  ```

- Member—Assigns the value to the subelement with the specified member name. For example:

  ```
  typedef struct packed {bits8 m; logic [23:0] n; } t3;
  ```

```
t3 p7 = '{n:5, m:3}; // Assigns 5 to the structure member named n, and 3 to
                     // the structure member named m
```

Associations by member name do not apply to arrays.

- Replication—Copies the given expression a constant number of times. For example:

```
t pi = t'{2{0}}; // Same as t'{0,0};
```

Mixing association styles in one assignment pattern is not allowed by the SystemVerilog standard. For example, when assigning to a structure, you cannot mix associations by member name (`member:value`) and positional notation, as in the following example:

```
typedef logic [7:0] bits8;
typedef struct packed { bits8 m; logic [23:0] n; } t;
...
parameter t pc = '{m:3, 5};  // Illegal
```

Limitations on Assignment Patterns

The following assignment pattern features in the SystemVerilog standard are not supported in the current release:

- The following types of assignment patterns are not supported:
  - Associations by type
- Assignment patterns can be used only as the right-hand side of an assignment. Using an assignment pattern as the left-hand side of an assignment is not supported. For example:

```
logic a, b, c, d;
typedef logic [3:0] t;
`{a, b, c, d} = t; // Not supported
```

- The assignment pattern syntax cannot be used for port expressions. For example:

```
typedef struct packed {int a; byte b;} packT;

module top;
  bottom b (`{a:0, b:1}); // Not supported
endmodule
```

However, you can use assignment patterns as arguments to tasks and functions. For example:

```
module bottom (input packT packer);
  task taskT (output packT out, input packT in);
  out = in;
  endtask

  packT one, two;
  initial task(one, '{416, 73}); // Supported
endmodule
```

- Using a full type qualification (`typeName'`) is required for
  - Assignment patterns assigned to out-of-module references
  - Port connection expressions

Aggregate Expressions

Aggregate expressions, data types, and data objects are used to reference a collection of singular values. In SystemVerilog, aggregate expressions can be used in

- Unpacked structure and array data objects
- Unpacked structure and array constructors
- Multi-element slices of unpacked arrays

In the current release, aggregate expressions can be

- Copied through assignments
- Used as arguments to tasks and functions
- Compared, using equality or inequality operators, against other aggregate expressions of the same type
- Used with conditional operators

However, for these uses, the current release supports only unpacked structures and fixed-size arrays of the following data type elements:

- Integral data types
- `real`
- Unpacked structures
- Unpacked fixed-size arrays, but the element data type must be one of the first three data types listed above

The current release does not support out-of-module references to these variables, and aggregates from array slicing or a member of an unpacked structure.

10

Procedural Statements

In Verilog, procedural statements are introduced using the following keywords:

`initial`	Enables the statement once at the beginning of the simulation
`final`	Executes the statement once at the end of simulation
`always`	Includes `always_comb`, `always_latch`, and `always_ff`
`task`	Executes the statement whenever the task is called
`function`	Executes the statement whenever the function is called, and returns a value

SystemVerilog has the following types of control flow within a process:

- Selection, loops, and jumps
- Task and function calls
- Sequential and parallel blocks
- Timing control

Unique and Priority Decision Statements

According to the Verilog-2001 standard, `case` statements must evaluate the case selection statements in the order in which they are listed. The `if...else` decisions must also be evaluated in source code order. This rule implies that there is a priority to the case selection items or series of `if...else...if` statements. To maintain priority ordering in the hardware implementation, priority-encoded logic is required. Often, however, the order of the decision statements is not important, and synthesis tools might optimize out the priority-encoded logic if the tool determines that the branches of the decision are mutually exclusive (unique). Strict coding guidelines must be followed to avoid mismatches between the interpretation of the model by simulation and synthesis tools.

In addition, the Verilog-2001 standard does not require that these decision statements always execute a branch of code.

SystemVerilog adds the keywords `unique` and `priority`, which can be used before `if` and `case`/`casex`/`casez` statements.

- The `unique` keyword indicates that the order of the decision statements is not important, and that they can be evaluated in parallel.

  Examples:

```
always_comb
unique case(a)
  0:       $display("0");
  1:       $display("1");
  default: $display("Value is not 0 or 1");
endcase

always_comb
unique if (a == 0) $display("0");
else if (a == 1) $display("1");
else $display("Value is not 0 or 1");
```

  The SystemVerilog LRM specifies that when a `case` or `if` statement is specified as `unique`, the software tools will verify that all of the decision conditions are mutually exclusive, and that they must generate a warning if more than one condition is true, or can be true. Tools are also required to generate a warning if the `case` or `if` statement is evaluated and no branch is executed. For example, the simulator generates a warning message for the following code because both the first and second conditions are true.

```
a = 3;
b = 4;

unique if (a < b)
  c = 1;
else if (a < 5)
  c = 2;
else if (a > b)
  c = 3;
```

- The `priority` keyword indicates that the order of the decision statements is important, and that tools must maintain the priority encoding. The selection items or series of `if...else...if` statements are evaluated in order, and the first match is used.

  Examples:

```
reg [2:0] a;

priority if (a[2:1] == 0) $display("0 or 1");
else if (a[2] == 0) $display("2 or 3");
else $display("4 to 7");

reg [2:0] a;

priority casez(a)
  3'b00?:  $display("0 or 1");
  3'b0??:  $display("2 or 3");
  default: $display("4 to 7");
endcase
```

  The SystemVerilog LRM specifies that when a `case` or `if` statement is specified as `priority`, there must be at least one true condition. Tools must generate a warning if the `case` or `if` statement is evaluated and no branch is executed. For example, the simulator generates a warning message for the following code because neither condition is true.

```
a = 3;

priority if (a = 0)
  c = 0;
else if (a = 1)
  c = 1;
```

Suppressing Semantic Checks for Unique and Priority Statements

By default, the simulator performs the checks that SystemVerilog requires for `unique` and `priority` constructs. This semantic checking can have an impact on simulation performance.

Use the `ncelab -svperf` command-line option to enhance performance by disabling the checking for `unique` and `priority` violations.

Examples:

The following option is the default. Both `unique` and `priority` constructs are checked, and warning messages are generated for all violations.

```
ncelab -svperf -up              // Same as -svperf -u-p
```

The following option disables checking of both `unique` and `priority` constructs.

```
ncelab -svperf +up              // Same as -svperf +u+p
```

The following option disables checking of `unique` constructs, but enables checking of `priority` constructs.

```
ncelab -svperf +u               // Same as +svperf+u-p
```

do...while Loop

Verilog provides `for`, `while`, `repeat`, and `forever` loops. SystemVerilog enhances loops by providing the `do...while` loop and `foreach` loop.

In the Verilog `while` loop, the condition expression is tested at the beginning of the loop. If the expression evaluates to false, the loop is not executed. If the expression starts out as false, the loop is not executed at all.

The `do...while` loop construct has the following syntax.

```
do statement_or_statement_block while (condition)
```

For a `do...while` loop, the condition is evaluated after the statement or statement block has been executed. The statements in the loop will be executed at least once whenever the loop is encountered. For example:

```
i=0;
  do begin
    a[i] = i;
    i = i + 1;
  end
  while (i < 10);
```

for Loop

In Verilog-2001, the variable used to control a `for` loop must be declared before the loop. This technique can cause problems if you inadvertently use the same loop control variable to control loops in two or more concurrent procedural blocks, because one loop can modify the variable while another loop is still using it. To avoid this situation, you must either declare different variables at the module level, or declare local variables within named `begin...end` blocks.

Verilog-2001 also allows only one initial assignment statement and one step assignment statement in a `for` loop.

SystemVerilog enhances `for` loops in two ways:

- The loop control variable can be declared within the `for` loop itself. In the following example, two loop control variables with the same name are declared.

```
module foo;
...
  initial
    begin
      for (integer i = 0; i < 10; i++)
      ...
    end

  initial
    begin
      for (integer i = 15; i >= 0; i--)
      ...
    end
...
endmodule
```

This example creates a local variable within the loop that other loops cannot affect.

Variables declared in a `for` loop initialization statement are automatic variables. Because they are created when the loop is invoked, and destroyed when the loop exits, they cannot be used outside of the `for` loop in which they are declared, and they cannot be referenced using a hierarchical name.

- Multiple initializer and step statements are allowed. Multiple initial assignment statements and step assignment statements are separated by commas. For example:

```
for (integer i = 0, integer j = 0; i < 10; i++, j++)
```

In the current release, the following is not supported:

- `genvar` variables within `for generate` loops
- Modports within `for` loops

foreach Loop

Verilog provides `for`, `while`, `repeat`, and `forever` loops. SystemVerilog enhances loops by providing the `foreach` loop. The `foreach` construct specifies iteration over the elements of an array. It is defined in the IEEE 1800 standard, as follows:

```
foreach (array_identifier [loop_variables]) statement
```

The `foreach` construct specifies the following:

- *`array_identifier`*—Designates an array, which can be a fixed-size, dynamic, or associative array. The *`array_identifier`* must be a simple identifier.
- *`loop_variables`*—A list of loop variables, where each variable corresponds to a dimension of the array.

Loop variables cannot use the same identifier as the array, and the number of loop variables must not exceed the number of array dimensions. If a loop variable is not specified, there is no iteration over the dimensions of the array.

The type of the loop variable automatically matches the type of the array index.

Although the IEEE 1800 standard specifies that *array_identifier* must be a simple identifier, the current release allows hierarchical identifiers that have the following syntax:

```
ps_or_hierarchical_array_identifier ::=
    [implicit_class_handle . | class_scope | package_scope]
    hierarchical_array_identifier
implicit_class_handle ::= this | super | this . super
class_scope ::= class_type ::
package_scope ::= package_identifier ::
    | $unit ::
hierarchical_array_identifier ::= hierarchical_identifier
hierarchical_identifier ::=
    [$root] {identifier constant_bit_select . } identifier
constant_bit_select ::= { [constant_expression] }
```

This extension allows array specifications that have the same syntax, even if they are not hierarchical names. For example, it allows non-constant indexes and names that are class or structure members (see Example 10-2 on page 163).

Note: This extension covers only the procedural `foreach` statement, not the constraint `foreach`.

Example 10-1 foreach Loop versus repeat Loop

The `foreach` construct is similar to the Verilog `repeat` loop. However, the `repeat` loop takes a number to designate how many times the loop must be executed, while the `foreach` loop uses the array bounds to specify the repeat count.

For example:

```
module myforeach;
  int a[5][3][4];

  initial begin
    foreach (a[lv]) begin
    $display ("Value of lv is %d",lv);
    end
    #1 $finish;
  end
endmodule
```

The `foreach` iterates `lv` from 0 to 4 to produce the following simulation output:

```
Value of lv is 0
Value of lv is 1
```

```
Value of lv is 2
Value of lv is 3
Value of lv is 4
```

Example 10-2 foreach Syntax

This example illustrates the following syntax for `foreach`:

```
foreach(items[i].ops[ii]);
```

where `i` and `ii` are variables. In this case, only the variables in the final bracket `[ii]` are treated as indexes for the `foreach` loop. The `[i]` variables are treated as a part of the hierarchical identifier for the array.

Note: This syntax is supported in the current release, although it is not specified by the IEEE 1800 standard. In the standard, indexes in the *`array_identifier`* must be constant.

In the following example, `foreach` loops are nested to iterate over both arrays.

```
module top;
  class c;
   int arr[];
  endclass

  c carr[5];

  initial
  begin
   foreach (carr[i])
  begin
    carr[i] = new;
    carr[i].arr = new[i];
    foreach (carr[i].arr[j])
      carr[i].arr[j] =  i * 10 + j;
    end
// The foreach loops are nested in order to iterate over both arrays
    foreach (carr[i])
     foreach (carr[i].arr[j])
       $display("carr[%0d].arr[%0d] = %0d", i, j, carr[i].arr[j]);
  end
endmodule
...

% irun -sv foreach.v

...
ncsim> run
carr[1].arr[0] = 10
carr[2].arr[0] = 20
carr[2].arr[1] = 21
carr[3].arr[0] = 30
carr[3].arr[1] = 31
carr[3].arr[2] = 32
carr[4].arr[0] = 40
carr[4].arr[1] = 41
carr[4].arr[2] = 42
carr[4].arr[3] = 43
```

```
ncsim: *W,RNQUIE: Simulation is complete.
```

Limitations on the foreach Loop

The current release has the following limitations on the `foreach` loop:

- The LRM states that loop variables are implicitly declared as automatic variables.

  The current release does not support automatic variables outside of automatic tasks and functions. Therefore, loop variables are declared as automatic only when the `foreach` construct is within an automatic task or function. Otherwise, the loop variable is implicitly declared as a static variable.

- The *`array_identifier`* cannot be an associative array with wildcard indexes. This limitation is expected to be part of the next revision of the standard.

return, break, and continue Jump Statements

Verilog-2001 provides the `disable` statement, which is used to jump to the end of a named statement group or to exit from a task. The `disable` statement requires additional block names, and can create code that is non-intuitive.

SystemVerilog includes the C jump statements `return`, `break`, and `continue`.

- The `return` statement can be used only in a task or function. This statement can be used to
  - Exit a task
  - Exit a non-void function and return a value. The `return` must have an expression of the correct type

    ```
    return expression;
    ```

  - Exit a void function

    ```
    return;
    ```

- The `break` statement can be used only in a loop. This statement jumps out of the loop.

  ```
  break;
  ```

- The `continue` statement can be used only in a loop. This statement jumps to the end of the loop and executes the loop control, if present. Named `begin...end` blocks are not required.

  ```
  continue;
  ```

A block of code cannot be disabled if it contains a `return`, `break`, or `continue` that can exit the block.

final Blocks

A `final` block is similar to an `initial` block, except that it executes when simulation ends, without delays. The only statements allowed within a `final` block are those that are legal in functions.

Note: The end of simulation does not cause an implicit call to `$finish`.

You can use a `final` block to display statistical information about the simulation.

Example:

```
final
   begin
      $display("Ending Simulation Time: %d", $time);
      $display("Clock Cycles: %d", nCycles);
   end
```

iff Event Control Qualifier

System Verilog adds an `iff` qualifier to the `@` event control. This construct provides conditional qualification of the event control.

The event expression is evaluated when the expression before the `iff` qualifier changes value, but it is triggered only if the expression after the `iff` qualifier is true.

Examples:

```
// Event expression is evaluated when d changes value, and triggers
// if enable is equal to 1.
always @(d iff enable == 1)
  q <= d;

// Event expression is evaluated at posedge clk, and triggers if enable
// is equal to 1 and preload changes value.
always @(posedge clk iff enable == 1, preload)
  q <= d;
```

always_* Blocks

System Verilog adds three specialized procedural blocks that reduce the ambiguity of the general purpose Verilog `always` block. These specialized blocks can be used to indicate design intent to simulation, synthesis, and formal verification software tools.

- `always_comb`—Indicates that the intent of the procedural block is to model combinational logic.
- `always_latch`—Indicates that the intent of the procedural block is to model latched logic behavior.
- `always_ff`—Indicates that the intent of the procedural block is to model sequential logic behavior.

fork...join

The Verilog `fork...join` block statement spawns multiple processes that execute in parallel. Each statement is a separate process, and statements that follow a `fork...join` are blocked from execution until all of the spawned processes have completed execution.

SystemVerilog enhances the `fork...join` statement by adding `fork...join_any` and `fork...join_none` blocks. These additions provide three options for specifying when the parent forking process is to resume execution.

`fork...join`	Statements that follow a `fork...join` are blocked from execution until all of the spawned processes have completed execution.
`fork...join_any`	Statements that follow a `fork...join_any` are blocked from execution until any one of the spawned processes has completed execution.
`fork...join_none`	Statements that follow a `fork...join_none` are not blocked from execution while the spawned processes are executing.

The current release also includes extensions to `fork...join` and `fork...join_none`.

fork...join

The `fork...join` statement has been extended so that the statement is now allowed in automatic tasks and functions, including inside local scopes.

fork...join_none

The `fork...join_none` statement has been extended to be allowed in functions.

In SystemVerilog, delaying statements—such as delay controls, event controls, `wait` statements, and task calls—are not allowed in functions. Before this release, `fork...join_none` statements were not allowed within functions, because they are not useful without delaying statements. This restriction has been removed.

In the current release

- `fork...join_none` statements are allowed inside functions
- Delaying statements are allowed inside `fork...join_none` statements that are within functions
- Tasks whose delays only occur within a `fork...join_none` are considered non-time-consuming tasks by the direct programming interface (DPI)

When called by the same type of process, the behavior of a `fork...join_none` statement in a function is the same as the behavior of a `fork...join_none` statement in a task. As with tasks, the process that called the function is considered the parent thread of all threads created by the `fork...join_none` statement.

The following restrictions apply to `fork...join_none`:

- Only the following processes can call a `fork...join_none` statement: `initial` blocks, `always` blocks, variable declaration initializers, and processes created by `fork`.

  In particular, `fork...join_none` statements cannot be executed by the following, because doing so will cause a run-time error: non-blocking assignments, regular and procedural continuous assignments, `force`, nonblocking event triggers, `final` blocks, particular system tasks (`$monitor`, `$strobe`, `$async$*`), or an evaluation of a user-defined system task or function argument by PLI. For example:

```
module top;

function f;
input i;
  fork
    #10 $display("in f");
  join_none
  f = i;
endfunction

integer r, x;
event e;

initial
  begin
  // Subprocesses that produce run-time errors

  // Nonblocking assignments
  r <= f(x);
   r <= #3 f(x);
  r <= #(f(x)) 0;
```

```
  r <= @(f(x)) 0;
  r <= repeat(2) @x f(x);
  r <= repeat(2) @(f(x)) 0;
  r <= repeat(f(x)) @x 0;
  r[f(x)] <= 0;

  // Nonblocking event triggers
  ->> #(f(x)) e;
  ->> @(f(x)) e;
  ->> repeat(2) @(f(x)) e;
  ->> repeat(f(x)) @x e;

  // System tasks that create subprocesses
  $monitor(f(x));
  $strobe(f(x));

  // Force statement
  force r = f(x);

  // Procedural continuous assignment
  assign r = f(x);
  end

  always @e $display(r);

  endmodule
```

- The Verilog `disable` statement will not disable subprocesses created by a `fork...join_none`. However, these subprocesses can be disabled using the `disable fork` statement.
- The current release supports `fork...join_none` subprocesses inside local scopes (`begin...end` or `fork...join`) that declare their own automatic variables, and local automatic scopes inside `fork...join_none` subprocesses. However, you cannot use `return`, `break`, or `continue` statements to exit these kinds of scopes. For example, the following illustrates an unsupported `break` statement within a local automatic scope.

```
task automatic t;
  int i;
   for (i = 0; i < 10; i++)
    begin: loop1
     int j; // Local automatic variable
     j = i;
     fork
      #1 $display(j); // j is visible to the following join_none subprocess
      join_none
      if (j == last)
         break; // Unsupported; uses break to leave the loop1 scope
    end: loop1
endtask
```

Limitations on fork...join_any

In the current release, the following restrictions apply to `fork...join_any`:

- The Verilog `disable` statement will not disable subprocesses created by a `fork...join_any` statement. However, these subprocesses can be disabled using the `disable fork` statement.
- The current release supports `fork...join_any` subprocesses inside local scopes (`begin...end` or `fork...join`) that declare their own automatic variables, and local automatic scopes inside `fork...join_any` subprocesses. However, you cannot use `return`, `break`, or `continue` statements to exit these kinds of scopes.

wait fork

Use the SystemVerilog `wait fork` statement to ensure that all spawned processes have completed their execution.

In the following example, the `getvalue` task waits for all four processes to complete before returning to its caller.

Example 10-3 Using the wait fork Construct

```
task automatic getvalue;
  fork
    taskA(); // Start taskA and taskB at the same time.
    taskB();
  join_any   // Block until either taskA or taskB completes.

  fork
    taskC(); // Start taskC and taskD at the same time.
    taskD();
  join_none

  wait fork; // Block until all four tasks have completed.
endtask
```

Note: The `wait fork` statement waits only for processes or threads that were directly spawned by the waiting process or thread. The `wait fork` statement does not wait for any descendents or subprocesses. To wait for all descendents of a process, each descendent must wait for all of its own spawned processes to complete execution before the descendent can terminate.

disable fork

SystemVerilog provides the `disable fork` statement, which disables all active threads of a calling process, including any subprocesses that were spawned by any of those threads.

Unlike the `disable` statement, which terminates the execution of a named block, regardless of its relationship to the calling process, the `disable fork` statement terminates only the processes that were forked by the calling thread.

In the following example, the `disable fork` statement terminates all three threads forked by the `getvalue` process, regardless of whether they have finished execution:

```
task automatic getvalue;
  input [8:0] value;
  input [8:0] num;

  begin
     #(num) $display("Delay of %d. Value of %d.", num, value);
  end
endtask

initial begin
fork
 #1 getvalue (11, 8);
 #1 getvalue (4, 6);
 #1 getvalue (8, 5);
join_none
 #5 disable fork;
 ...
end
```

However, in the following example, the `disable fork` statement is blocked until one of the `getvalue` processes has completed execution. It then terminates the two outstanding `getvalue` processes.

```
...
initial begin
fork
 #1 getvalue (11, 8);
 #1 getvalue (4, 6);
 #1 getvalue (8, 5);
join_any
 #5 disable fork;
 ...
end
```

11

Tasks and Functions

Several enhancements to tasks and functions are included in the SystemVerilog LRM.

Multiple Statements in Tasks and Functions

In Verilog-2001, you must enclose multiple statements in a task or function within a `begin...end` block. SystemVerilog does not require you to enclose multiple statements with `begin...end`.

For example, Verilog 2001 requires:

```
function bounds_err(input integer a, input integer lower);
  integer upper;

  begin
    upper = lower + 255;
    if (a > upper || a < lower)
      bounds_err = 1;
    else
      bounds_err = 0;
end

endfunction
```

SystemVerilog infers the `begin...end`, and executes the multiple statements sequentially.

```
function bounds_err(input integer a, input integer lower);
  integer upper;

  upper = lower + 255;
  if (a > upper || a < lower)
    bounds_err = 1;
  else
    bounds_err = 0;

endfunction
```

Function Output Arguments

In Verilog-2001, functions can have only inputs. The only output is the single return value.

SystemVerilog allows the formal arguments of functions to be declared with the same directional specifiers as tasks, so that a function can have any number of outputs in addition to the return value. The arguments can be declared as

- `input`—Copy the value in at the beginning of the function call.
- `output`—Copy the value out at the end of the function call.
- `inout`—Copy the value in at the beginning, and out at the end, of the function call.

Note: SystemVerilog also lets you declare a task or function argument as `ref`. Refer to "Passing Task and Function Arguments by Reference" on page 175.

Examples:

```
function void myfunc;
  input integer a;
  input integer b;
  output integer x;
  output integer y;
...
endfunction

function void myfunc(input integer a, input integer b, output integer x,
                     output integer y);
...
endfunction
```

You cannot call a function with `output` or `inout` arguments from

- An event expression
- An expression within a procedural continuous assignment
- An expression that is not within a procedural statement

Default Direction in Task and Function Declarations

When using ANSI-style declarations in tasks and functions, if the direction of an argument is not specified, the direction is the direction of the previous argument in the list. If the direction of the first argument is not specified, the default is `input`. In the following example, the formal argument `a` defaults to `input`. No direction is specified for argument `b`, so it is also an `input`. Arguments `x` and `y` are both outputs.

```
function void myfunc (integer a, integer b, output integer x, y);
...
endfunction
```

Void Functions

In Verilog-2001, a function must return a value. The return value is specified by assigning a value to an implicitly declared internal variable with the same name as the function. For example:

```
function myfunc (input a, b);
  myfunc = a * b - 1;
endfunction
```

The function call is an operand in an expression. For example:

```
x = y + myfunc(c, d);
```

SystemVerilog allows functions to be declared as data type `void`, which do not have a return value. For example:

```
function void myprint (integer a);
...
endfunction
```

Void functions have the syntax and semantic restrictions of non-void functions, but they are called as statements, like Verilog tasks. For example:

```
myprint(a);
```

Discarding Function Return Values

In SystemVerilog, you can discard a function's return value by casting the function to the `void` data type. For example, the following discards the return value of `myclass.randomize()`:

```
void'(myclass.randomize());
```

Passing Task and Function Arguments by Reference

In Verilog-2001, arguments are passed to tasks and functions by value using a copy-in/copy-out mechanism. When a task or function is called, the argument is copied into the task or function. The argument value then becomes local to the task or function, and is not visible outside the task or function. When the task or function is finished executing, the argument is copied out to the caller of the task or function. Unfortunately, passing by value is undesirable when you have large arguments, or if you have programs that need to share data that is not declared as global.

SystemVerilog lets you pass arguments by reference. Instead of copying an argument value, a reference to the original argument is passed to the task or function. Passing by reference offers the following:

- Unlike the pass by value mechanism, changes to a referenced argument are visible outside the task or function. Conversely, changes to a referenced argument outside the task or function are also visible within the task or function.
- Tasks can be reused to generate waveforms and transactions on different variables, which is beneficial in testbench environments.

The syntax for passing argument values by reference is as follows:

```
subroutine (ref type argument);
```

For example:

```
task my_task(ref reg in1, ref reg in2);
  begin
  ...
  end
endtask
```

The `ref` keyword cannot be used with other direction keywords—`input`, `output`, or `inout`. For example, the following is illegal:

```
task check_status (ref input integer in1);
...
```

The LRM states that arguments that are passed by reference must match exactly and that auto-casting, promotion, or conversion is not allowed. For example, you cannot pass an integer by reference when the formal argument is not of type `int`.

In the following example, a module called `submod` defines an array called `my_array` and a variable of type `int` called `my_int`. These objects are passed as arguments in a call to `mytask`. In `mytask`, the formal arguments are defined as `ref` arguments. Within the `submod` module, the formal argument `in1` is an alias for `my_array`, and the formal argument `in2` is an alias for `my_int`.

```
...

module submod;
  parameter LEFT  = 1;
  parameter RIGHT = 0;
  reg [LEFT:RIGHT] my_vector;
  integer my_int;

  initial begin
    #1 my_task(.in1(my_vector), .in2(my_int)); // Passes objects as arguments
  end

  task automatic my_task(
    ref reg [1:0] in1, // References my_vector using alias in1
    ref integer in2 ); // References my_int using alias in2
    begin
      $display( "in1( %d )", in1 );
      $display( "in2( %d )", in2 );
    end
```

```
  endtask
endmodule
```

The current release also supports:

- Passing automatic variables by reference

  The following example passes the value of automatic variable `b`:

```
task automatic mytask(ref int a);
  int b;
  b = a;

  if (b > 1)
     mytask(b); // Supported
endtask
```

- Automatic tasks or functions that pass a `ref` argument by reference

```
task automatic mytask (ref int r, inout int io);
 int depth;
 if (depth > 0) begin
  mytask(r, io); // Valid to pass r by reference
  mytask(io, r); // Valid to pass io and r by reference
end
endtask
```

- Passing an enum by reference

```
typedef enum {red, green, blue} color;

task mytask(ref color color_var);
...
endtask
```

- Passing variables by reference to automatic tasks

Limitations on Passing Task and Function Arguments by Reference

The following summarizes the features in the LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- `ref` arguments can be scalars and vectors of type `bit`, `logic`, and `reg`. A `ref` argument can also be of type `int`, `shortint`, `longint`, `byte`, `real`, and a packed structure. The Cadence implementation supports passing arrays of type `int`, `shortint`, `longint`, `byte`, `real`, packed structures, enumerations, `logic`, `bit`, and `reg`. For example:

```
ref bit a;       // A scalar bit is valid
ref bit [3:1] a; // A bit vector is valid
ref bit a[0:3];  // A bit array is valid
ref int a;       // A scalar int is valid
```

```
ref int a[0:3];  // An int array is valid
```

Although it is not explicitly stated in the LRM, the Cadence implementation does not support the passing of a member of an array, or a member of a packed structure, by reference to a task or function.

- Although it is not explicitly stated in the LRM, the Cadence implementation does not support part selects that travel through `ref` arguments. For example, the following is not supported:

```
module test_top;
...
reg [15:0] a;
task rotate(ref [3:0] op);
...
endtask

rotate(a[15:12]); // Invalid
endmodule
```

- Although it is not explicitly stated in the LRM, the Cadence implementation does not support out-of-module references to `ref` arguments. For example, the following illustrates illegal out-of-module reference (OOMR) access to a task's reference arguments:

```
module test_top;
reg  [31:0] a;
integer b;

initial begin
  #1 my_task(.in1(a), .in2(b));
  #1 test_top.my_task.in1 = 1;   // Illegal OOMR write to ref argument
  #1 b = my_task.in2;            // Illegal OOMR read from ref argument
end

task my_task( ref reg [1:32] in1, ref integer in2 );
 begin
  //I llegal OOMR read from ref argument
  #1 $display( "in1( %d )", test_top.my_task.in1 );

  // Illegal OOMR read from ref argument
  #1  $display( "in2( %d )", my_task.in2 );
  end
endtask

endmodule
```

Specifying Default Argument Values for Tasks and Functions

SystemVerilog lets you specify default values for task and function formal arguments with a direction of `input`, `inout`, or `ref`. Specifying default values is allowed only with ANSI-style declarations.

When the task or function is called, an argument with a default value can be omitted from the call, and the default value will be used for that call. An error is generated if a formal argument without a default value is not passed in a value when the task or function is called.

The following example, taken from the SystemVerilog LRM, declares a task called `read`. Two of the formal arguments have default values.

```
task read(int j = 0, int k, int data = 1);
...
endtask;
```

This task can be called using various arguments, as follows:

```
read( , 5 );         // Same as read( 0, 5, 1 );
read( , 5, );        // Same as read( 0, 5, 1 );
read( 2, 5 );        // Same as read( 2, 5, 1 );
read( , 5, 7 );      // Same as read( 0, 5, 7 );
read( 1, 5, 2 );     // Same as read( 1, 5, 2 );
read( );             // Error because argument k has no default value
```

You can use any value compatible with the formal parameter expression as a default argument to a task or function.

Passing Task and Function Arguments by Name

In Verilog-2001, values must be passed to task or function arguments in the same order in which the formal arguments are defined. This rule can cause errors if values are passed in the wrong order.

SystemVerilog lets you pass arguments by name, as well as by position. Named arguments can be passed in any order. The syntax is the same as that used for named port connections to a module instance.

Example:

```
task read(int j = 0, int k, int data = 1);
...
endtask;
```

The following calls to the `read` task pass arguments by name. If an argument is not specified, the default value is used. A value for argument `k` must be specified, because no default value is defined.

```
read( .j(2), .k(5), .data(3) );        // read(2, 5, 3);
read( .k(5), .data(3), .j(2) );        // read(2, 5, 3);
read( .j(2), .k(5) );                  // read(2, 5, 1);
read( .k(5) );                         // read(0, 5, 1);
```

If both positional and named arguments are specified in a call, all positional arguments must precede the named arguments. For example:

```
read( 2, 5, .data(7) );     // read( 2, 5, 7 );
read( .k(5), 2, 5 );        // Error because named arg precedes positional arg
```

Optional Arguments for Tasks and Functions

In Verilog-2001, tasks and functions must have at least one argument. SystemVerilog removes this restriction. For example:

```
module top;
int count;
function int myfunc (input int a = 0);
  ...
endfunction

initial begin
  count = myfunc(3);  // Valid in Verilog-2001 and SystemVerilog
  count = myfunc();   // Valid in SystemVerilog, but not in Verilog-2001
end
endmodule
```

File I/O System Tasks/Functions and SystemVerilog

Some built-in file I/O system tasks and functions have been enhanced to support SystemVerilog constructs. The following table list the tasks and functions, and the supported SystemVerilog constructs.

File I/O System Task/Function	Supported SystemVerilog Constructs
`$swrite`, `$swriteb`, `$swriteh`, `$swriteo`	Data types `shortint`, `int`, `longint`, `byte`, `bit`, `logic`, `string`, `chandle`
	Union members and packed unions
	Struct members and packed structs
	Enums
	Queues
	Associative arrays, dynamic arrays
	Classes
	Clocking blocks, semaphores, programs
	Interfaces
`$fwrite`, `$fwriteb`, `$fwriteh`, `$fwriteo`	Data types `shortint`, `int`, `longint`, `byte`, `bit`, `logic`, `string`, `chandle`
	Union members and packed unions
	Struct members and packed structs
	Enums
	Queues
	Associative arrays, dynamic arrays
	Classes
	Clocking blocks, semaphores, programs
	Events
	Interfaces

File I/O System Task/Function	Supported SystemVerilog Constructs
`$sscanf` `$fscanf`	shortint, int, longint, byte, bit, logic, string
	chandle (cannot put a value)
	Union members and packed unions
	Packed structs and packed struct members
	Enums
	Dynamic arrays
	Classes
	Clocking blocks
	Semaphores (cannot put a value)
	Programs
	Events (cannot put a value)
	Interfaces
`$fread`	shortint, int, longint, byte, bit, logic
	Union members of type reg, integer, shortint, int, longint, byte, bit, logic
	Struct members of type reg, integer, shortint, int, longint, byte, bit, logic for individual members
	Classes (with valid types)
	Clocking block (with valid types)
	Semaphores (cannot put a value)
	Programs (with valid types)
	Events (cannot put a value)
	Interfaces (with valid types)

In addition, you can find a complete description of the Verilog file I/O tasks and functions in the “Modeling Your Hardware” topic of the *Verilog Simulation User Guide*.

12

Random Constraints

The IEEE 1800 standard describes random constraints, which can be used to generate constraint-driven tests. Unlike traditional directed testing, random testing can help create tests for unique, hard-to-find cases.

In SystemVerilog, you can specify constraints that restrict the values that can be assigned to `rand` or `randc` variables. A solver within the simulator processes constraints, and chooses values for the `rand` or `randc` variables that satisfy the constraints. If some constraints are overly restrictive, in that some random values cannot be satisfied, the solver issues an overconstrained message and terminates the randomization attempt.

Similar to methods, constraints are treated like class members. You can specify a constraint in a class or a derived class. For example, the following defines the `myClass` class with three variables: `a`, `b`, and `c`.

```
class myClass;
  rand bit [3:0] a, b, c;
  constraint c1 {c == a + b;}
endclass
```

The `c1` constraint limits the value of `c` to the sum of `a` and `b`.

To generate random values for the variables in this class, you can use the `randomize()` method, which is described in the LRM, and in "The randomize() Method" on page 194:

```
task chkSuccess(myClass myc);
  int success;
  success = myc.randomize();  // Generates random values for a, b, and c
endtask
```

Note: The constraint solver and random number generator (RNG) used in SystemVerilog randomization might include optimizations not included in previous releases, so the random number stream can differ from release to release.

Random Variables

You can declare random class variables using the `rand` or `randc` keywords, where

- The `rand` modifier defines standard random variables, where the solver distributes values equally over a range. For example, if the following declaration is not associated with a constraint, the solver picks a random value between 0 and 3 with a uniform distribution:

  ```
  rand bit [1:0] d;
  ```

  In this example, there is a 1 in 4 chance that the solver will repeat a value during successive calls to `randomize()`.

- The `randc` modifier defines random-cyclic variables, where the solver goes through all of the values in a random permutation of the specified range. For the following example, the solver generates a random initial permutation using the possible range of values for `e`.

  ```
  randc bit [1:0] e;
  ```

  An initial permutation for this declaration might be 2, 0, 1, 3. If there are successive calls to `randomize()`, the solver returns the values in this order. When the solver runs out of values within the initial permutation, it automatically generates a new permutation.

  The `randc` variables can only be of type `bit` or enumerated types.

The current release supports the following:

- `rand` handles for classes
- `rand` arrays, where the array element type can be a
  - handle
  - `integer`, `shortint`, `byte`, or `bit`
  - `bit` vector
- Multidimensional `rand` arrays
- Aliased `rand` handles, where two or more handles refer to the same class instance.

```
module top;
 class class1;
  rand int rint1;
 endclass
 class class2;
  rand class1 ch1a;
  class1 ch1b;
  constraint con1 { ch1a.rint1 == 200; }
 endclass
  class2 ch2;
  int res;
 initial begin
  ch2 = new;
  ch2.ch1a = new;
  ch2.ch1b = ch2.ch1a;  // ch2.ch1a and ch2.ch1b refer to the same object
```

```
  ch2.ch1b.rint1 = 100;
  res = ch2.randomize();
  // ch2.ch1b.rint1 is now 200, even though ch2.ch1b is not rand
  $display("ch2.ch1b.rint1 = %d", ch2.ch1b.rint1);
 end
endmodule
```

- Null `rand` handles

```
module top;
 class class1;
  rand int rint1;
  constraint con1 { rint1 == 200; }
 endclass
 class class2;
  rand class1 ch1a;
  rand class1 ch1b;
 endclass
 class2 ch2;
 int res;
 initial begin
  ch2 = new;
  ch2.ch1a = new;
  // ch2.ch1b is null
  ch2.ch1a.rint1 = 100;
  res = ch2.randomize();  // Does not consider ch2.ch1b, since it is null
  $display("ch2.ch1a.rint1 = %d", ch2.ch1a.rint1);
 end
endmodule
```

Limitations on Random Variables

The following summarizes the random variable features in the LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- In the Cadence implementation, `rand` is supported for fixed-size arrays, associative arrays, and dynamic arrays. The elements of these arrays can be integral types, such as integer, bit, and bit vector, or class handles.
- `rand` is supported for associative arrays. However, the array index and element type of the associative arrays must be an integral type, such as `integer`, `bit`, and `bit` vector; [*] is also supported. Strings and class handle are not supported.

  For `rand` associative arrays, the index must be an integral type, and the width is limited to 64 bits.
- Constraint blocks can contain non-rand associative arrays. For example:

```
...
opcode code[int];
constraint c1 {opcode[RESET] == 1 -> rand_var != CLEAR;}
```

Constraint blocks that contain `rand` associative arrays cannot contain an expression that requires a new item to be inserted into an associative array. Only the values of existing associative array items can be changed by a randomize call. Although this rule is not specified in the IEEE 1800 Standard, it is outlined in Mantis 2113. This limitation also applies to non-rand associative arrays within constraint blocks. For example:

```
module top;
class class1;
 rand int aa[int];
 constraint c1 { aa[100] == 123; }

 // In the next constraint, the randomize call will fail, because there is
 // no item at aa[200]
 constraint c2 { aa[200] == 234; }
endclass

class1 ch1 = new;

 initial begin
  ch1.aa[100] = 12;  // Insert an item at aa[12]
  assert(ch1.randomize());

  ch1.aa[200] = 13;  // Insert an item at aa[13]
  assert(ch1.randomize());  // This randomize call will be OK
  end
endmodule
```

- `randc` arrays are not supported.
- `rand` is not supported for unpacked structures.
- In the Cadence implementation, `randc` variables have a maximum size of 16 bits, and `randc` enums have a maximum size of 32 bits.
- `rand` variables are supported in function calls. However, `rand` arrays are not supported in function calls.

Constraint Blocks

A constraint block is a class member that lists expressions used to limit the range of a variable, or to define the relationship between variables. The following is the simplified syntax for a constraint block.

```
constraint constraint_identifier
      {constraint_expression [;constraint_expression]; }
```

For example:

```
class myClass;
  rand integer len;
  constraint db {len > 0;}  // Specifies that len must be greater than zero
endclass:myClass
```

Limitations on Constraint Blocks

The following summarizes the constraint block features in the LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- In a constraint expression, the left and right operands of the `**` (exponentiation) operator must be constants.
- The LRM states that only 2-state values are supported within constraints, but the LRM does not describe what happens to 4-state variable values. In the Cadence implementation, 4-state values are converted to 2-state values. This implementation supports the LRM, which specifies that X and Z values are converted to zero.

External Constraint Blocks

The current release supports external constraint blocks.

Inheritance

The current release supports inheritance in constraint blocks, which is similar to inheritance for class variables, tasks, and functions. The current release supports constraint inheritance as described in the LRM. That is, a constraint in a derived class that uses the same name as a constraint in its parent classes overrides the base class constraints. Otherwise, all constraints are inherited as-is from the parent class.

Set Membership

The current release supports the `inside` operator, which is defined in the SystemVerilog LRM.

For example, the following specifies that the legal random values that can be assigned to `rand` variable `a` are 3, 4, or 5:

```
class set_example;
  rand integer a;
  constraint setA {a inside {3, 4, 5};}
endclass
```

In the following example, the `randc` variable `my_randc` will cycle through values in the ranges 0 through 20, and 60 through 90:

```
class set_example1;
  randc byte my_randc;
```

```
  constraint con1 {my_randc inside {[0:20], [60:90]};}
endclass
```

The following illustrates a set membership constraint that contains dynamic arrays. Associative arrays and queues are also supported.

```
class c;

bit [7:0] da[];
rand bit [1:0] control;
rand bit [15:0] addr;

constraint restrict_idx_search {
          (control == 0) -> { addr inside {[0:10]}; }
          (control == 1) -> { addr inside { da}; }
          (control == 2) -> { addr[15:8] inside { da}; }
          }
constraint sb { solve control before addr;}
endclass : c
...
```

You can also use the `inside` operator outside of the constraint block, as shown in the following code:

```
module m;
  initial
    if (1 inside { 1, 2 })
      $display("It works");
endmodule
```

Distribution

Randomization constraints can specify sets of weighted values called *distributions*. Assuming that no other constraints are specified, the probability of selecting a legal value in the list is proportional to its specified weight.

Defining a Distribution Expression

To define a distribution expression, use the `dist` operator. The syntax is similar to that used for set membership, in that you specify a set of legal values as a comma-separated list of single values or ranges. You can specify a weight for each term in the list by using one of the following operators:

- `:=`

  Assigns a weight to the item. If the item is a range, assigns the weight to every value in the range.

- `:/`

Assigns a weight to the item. If the item is a range, assigns the weight to the range as a whole.

Specifying a weight is optional. If you do not specify a weight, the default is `:= 1`.

In the following example, the set of legal random values that can be assigned to variable `b` is 100, 200, or 300. Because no weights are specified, the weighted ratio is 1-1-1.

```
class dist_example;
  rand integer b;
  constraint c1 {b dist {100, 200, 300};}
endclass
```

In the following example, the constraint specifies that `b` is equal to 100, 200, or 300, but with a weighted ratio of 1-2-3.

```
constraint c2 {b dist {100 := 1, 200 := 2, 300 := 3};}
```

The following example specifies that `b` is equal to 100, 101, 102, 200, or 300 with a weighted ratio of 1-1-1-2-3.

```
constraint c3 {b dist {[100:102] := 1, 200 := 2, 300 := 3};}
```

The following example uses the `:/` operator, which applies the weight to the range as a whole. Variable `b` must be 100, 101, 102, 200, or 300 with a weighted ratio of 1/3-1/3-1/3-2-3.

```
constraint c4 {b dist {[100:102] :/ 1, 200 := 2, 300 := 3};}
```

You can use the `dist` operator within conditional statements. For example:

```
constraint c1 {if (cmd==READ) addr dist {12'h0 :=1, 12'h0 :=1, 12'h0 :=1};
```

You can use the `dist` operator on a member of a structure. For example:

```
type struct {
  bit [7:0] data;
  bit [15:0] addr;
  } op;

  rand op foo;

  constraint data_dist {foo.addr dist 100 := 1, 200 := 2, 300 := 3;} //Supported
```

Limitations on Distribution Expressions

The following distribution features in the LRM are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- There are no limitations on the distribution expression, which is to the left of the `dist` operator. The distribution expression can be any legal expression. There are also no limitations on range expressions.

Weight expressions, however, cannot contain `rand` variables, `randc` variables, or function calls.

- A `rand` variable can have only one distribution constraint.
- If a `:=` weight operator applies to a range, the range expressions cannot include any `rand` variables.

Implication

The implication operator (`->`) constrains random values only if a condition is successful. The syntax for the constraint is as follows:

```
expression -> constraint_set
```

where

- *`expression`* is any valid SystemVerilog integral expression
- *`constraint_set`* is any valid constraint or unnamed constraint set

If the *`expression`* is true, the random numbers are constrained by *`constraint_set`*. If *`expression`* is false, the solver ignores the *`constraint_set`*.

The following expression constrains `payload` to less than 100 when mode is `SMALL`, greater than 10000 when mode is `LARGE`, or unconstrained when mode is neither `SMALL` nor `LARGE`:

```
constraint payC {mode == SMALL -> payload < 100; mode == LARGE -> payload > 10000;}
```

Note: The implication operator is bidirectional. In the previous example, the value of `payload` constrains `mode`, but the value of `mode` also constrains `payload`.

Because the implication operator is bidirectional (`a->b`), if expression `b` is false, then `a` must also be false.

if...else Constraints

Similar to the Implication operator, the `if...else` style of specifying constraints uses a conditional expression, in that random values are constrained when a condition is met. The syntax for `if...else` constraints is as follows:

```
if(expression) constraint_set [else constraint_set]
```

where

- *`expression`* is any valid SystemVerilog integral expression

- *constraint_set* is any valid constraint or unnamed constraint block; a *constraint_set* can contain a group of constraints

If the *expression* is true, the random numbers are constrained by the first *constraint_set*. If *expression* is false, the random numbers are constrained by the second, optional *constraint_set*.

An if...else statement and an implication statement are equivalent. For example, the following constraints have the same effect, where payload must be constrained to less than 100 when mode is SMALL, and greater than 10000 when mode is LARGE:

```
constraint payD {mode == SMALL -> payload < 100; mode == LARGE -> payload > 10000;}

constraint payE {if (mode == SMALL) payload < 100;
                 else if (mode == LARGE) payload > 10000;}
```

Because the else is optional, there might be confusion when associating an if with an else in nested if...else statements. To avoid confusion, SystemVerilog associates an else with the last if that is missing an else. In the following example, the else relates to the second if:

```
constraint payF {if (mode != SMALL) if (mode == LARGE) payload > 10000;
                 else payload < 100;}
```

Iterative Constraints

An iterative constraint expression uses loop variables and indexing expressions to specify iteration over elements in an array. The syntax for an iterative constraint expression is as follows:

```
foreach (array_identifier [loop_variables]) constraint_set
loop_variables::=[index_variable_identifier]
                 {,[index_variable_identifier]}
```

For example, the following constrains the elements of the array called myArray to be in the set (2, 3, 5, 6, 10):

```
class myClass;
 rand byte myArray[];
 constraint myCons {foreach (myArray[i]) myArray[i] inside {2, 3, 5, 6, 10};}
 ...
endclass
```

Note: In the IEEE 1800 standard, the array in a foreach statement must be a simple identifier. The current release extends this requirement to allow hierarchical identifiers. This implementation is not supported in the IEEE 1800 standard, but it is set to be a part of future releases of the SystemVerilog language. For more information, see “foreach Loop” on page 159.

Limitations on iterative constraints in the current release

- The index variable must be an integer type.
- Function calls are supported within `foreach` constraint expressions. However, if the `foreach` expression is within an in-line constraint, it cannot contain function calls that have a `foreach` loop index as an argument.

Global Constraints

Global constraints are constraint expressions that contain random variables from other objects. In the current release, constraint expressions can contain variables that are declared in modules, packages, and classes. For example, the following illustrates the use of `rand` class handles. This example applies two constraints to the `rand` variable `rand_int_1`, so that its value is between 11 and 19.

```
module top;

 class c1;
  rand int rand_int_1;
  constraint con1 { rand_int_1 > 10; }
 endclass

 class c2;
  rand c1 c1h;
  constraint con2 { c1h.rand_int_1 < 20; }
 endclass

 c2 c2h;
 integer res;

 initial begin
  repeat(5) begin
   c2h = new;
   c2h.c1h = new;
   res = c2h.randomize();
   $display("res = %d, c2h.c1h.rand_int_1 = %d", res, c2h.c1h.rand_int_1);
  end
 end
endmodule
```

solve...before Constraints

Generally, random values are chosen with equal probability—that is, all values have the same chance of being chosen by the solver. However, you can use the `solve...before` keywords in a randomization constraint to specify that a particular combination of legal values occurs more often than others. A `solve...before` constraint has the following syntax:

```
constraint constraint_identifier
      {solve identifier_list before identifier_list;}
```

The *identifier_list* contains only random variables with integral values. You cannot impose variable ordering on randc variables, because randc variables are always solved first. For example:

```
class myclass;
  rand bit r1;
  rand bit [7:0] r2;
  constraint con1 { r1 -> r2 == 1; }
  constraint order {solve r1 before r2; }
endclass
```

In this example, the solve...before directive lets you specify that the value of r1 must be chosen independently from the value of r2, giving r1 equal probability of having the value 0 or 1. Without the solve...before directive, r1 and r2 are determined together, giving r1 almost zero probability of being 1.

The current implementation of solve...before supports all of the legal uses outlined by the LRM.

Static Constraint Blocks

The current release supports the static keyword, which is used to declare static constraint blocks. When a constraint block is declared as static, the constraint_mode() method affects all of the instances of the specified constraint, in all objects.

Functions in Constraints

Function calls are supported in constraint expressions.

In the current release, when a function is called from a constraint expression

- The function must return an integral type
- The arguments to the function call cannot contain rand variables
- The function cannot contain a randomize call

For example:

```
class class1;
   rand int rint1;
   function int fun1(input int arg1);
     fun1 = 2 * arg1;
   endfunction

   constraint c1 { rint1 == fun1(2); }

endclass
```

Note: According to the LRM, the functions that are called in constraints cannot have side effects, but this requirement is not strictly checked.

Randomization Methods

The current release supports all of the behavior described in the IEEE 1800 standard.

The randomize() Method

The built-in `randomize()` method generates random values for all active random variables within an object, and are subject to the active constraints within the object. The `randomize()` function returns 1 if it successfully assigns all of the random variables within an object to a valid value. Otherwise, it returns 0.

The following example defines a class with random variables `i1`, `i2`, and `i3`. It then proceeds to validate whether the solver was able to generate valid random values for these variables:

```
module top;
  integer debug;
  integer success, x1;

  class class1;
    rand integer i1, i2, i3;  // Define random variables
    integer i10;
    constraint c1 { i1 > 10; i1 < 20; }
    constraint c2 { i2 > i1; i2 < 100; }
    constraint c3 { i3 > i2; i3 < i10; }
  endclass

  class1 p1;

  initial begin
    debug = 1;
    p1 = new;
    p1.i10 = 1000;
    for (x1 = 0; x1 < 10; x1 = x1 + 1) begin
      p1.i1 = 0;
      p1.i2 = 0;
      p1.i3 = 0;

      success = p1.randomize();  // Generate random values for i1, i2, and i3

      if (debug == 1) begin  // Validate whether randomize() was successful
        $display("\nsuccess = %d", success);
      end
    end
  end
endmodule
```

Note: When a constraint contains `randc` variables, the solver generates the `randc` variable values first, and these variables become state variables within the constraint. In this case,

`randomize()` might fail to find a solution. In the current release, the solver will not repeat its attempts to reach a solution to the constraint.

pre_randomize() and post_randomize()

The `pre_randomize()` and `post_randomize()` methods are discussed in the IEEE 1800 standard.

The current release supports the built-in `pre_randomize()` and `post_randomize()` methods, which are called by the `randomize()` method before and after it computes random variables. You can override these methods when you want to perform operations before or immediately after randomization. If these methods are overridden, they must call their associated parent class methods. Otherwise, pre- and post- randomization processing steps are skipped.

In-Line Constraints (randomize() with)

Declaring in-line constraints with the `randomize()...with` construct is supported, including the `local::` qualifier. For example:

```
res = bus.randomize() with {atype == low;};
```

In-line constraints can refer to local variables. For example:

```
class c;
  function void do_rand(input int limit);
   int rc;
   rc = randomize(class_property) with {class_property < limit;} //Supported
  endfunction
endclass
```

Activating and Inactivating Random Variables with rand_mode()

Random variables can be activated or inactivated using the `rand_mode()` method. All random variables are initially active. A random variable that has been inactivated is treated as a state variable, and is not randomized by the `randomize()` method.

You can call the `rand_mode()` method as a task or a function.

- When called as a task, the argument to `rand_mode` (`1` or `0`) determines the operation to be performed. The syntax is as follows:

  object[.*random_variable*].rand_mode(*value*);

where:

- *object* is the object in which the variable is defined.
- *random_variable* is the name of the variable to activate or inactivate; if no variable is specified, the action applies to all random variables in the object
- *value* can be either 1 or 0

  The value 1 activates the specified variable, or all variables if no variable is specified. The value 0 inactivates the specified variable, or all variables if no variable is specified.

- When called as a function, rand_mode() returns 1 if the variable is active, or 0 if the variable is inactive. The syntax is as follows:

```
object.random_variable.rand_mode();
```

For example:

```
class Foo;
  rand integer p1, p2;
endclass

Foo myFoo = new;
initial begin
  int result;
  myFoo.rand_mode(0);            // Inactivate all variables in object myFoo
  myFoo.p1.rand_mode(1);         // Activate variable p1 in object myFoo
  result = myFoo.p1.rand_mode(); // Sets result to 1 because p1 is enabled
end
```

Limitations on rand_mode()

The Cadence implementation supports using the rand_mode() method for packed arrays, packed structures, full unpacked arrays, and elements of fixed-size unpacked arrays. However, the rand_mode() method is not supported for unpacked structures.

Activating and Inactivating Constraints with constraint_mode()

Constraints can be activated or inactivated using the constraint_mode() method. All constraints are initially active. Constraints that are inactive are ignored by the randomize() method.

The constraint_mode() method can be called as a task or a function.

- When called as a task, the argument to `constraint_mode` (`1` or `0`) determines the operation to be performed. The syntax is as follows:

  ```
  object[.constraint_identifier].constraint_mode(value);
  ```

  where

  - *`object`* is the object in which the variable is defined
  - *`constraint_identifier`* is the name of the constraint to activate or inactivate; if no constraint is specified, the action applies to all constraints in the object
  - *`value`* can be either `1` or `0`.

    The value `1` activates the specified constraint, or all constraints if no constraint is specified. The value `0` inactivates the specified constraint, or all constraints if no constraint is specified.

- When called as a function, `constraint_mode()` returns `1` if the constraint is active, or `0` if the constraint is inactive. The syntax is as follows:

  ```
  object.constraint_identifier.constraint_mode();
  ```

For example:

```
class c;
  rand bit cm1,cm2;
  constraint con1 {cm1 < 4;}
  constraint con2 {cm2 == cm1;}
endclass

c cd = new;

initial begin
  int result;
  cd.constraint_mode(0);          // Disables all constraints in object cd
  cd.con1.constraint_mode(1);     // Enables constraint con1 in object cd
  result = cd.con1.rand_mode();  // Sets result to 1 because con1 is active
end
```

In-Line Random Variable Control

You can temporarily control the randomized variables within a class object or instance by calling the `randomize()` method with arguments. When the `randomize()` method is called with arguments, the arguments specify the variables that should be randomized within a class object—all other variables within the object are considered state variables, and are not randomized. For example:

```
module top();
  class c;
    rand integer p1, p2;
    integer b1, b2;
    ...
```

```
  endclass

c pc1;

  initial begin
    byte res;
    pc1 = new;
    res = pc1.randomize();               // Randomize p1 and p2
    res = pc1.randomize(p1);             // Randomize only p1
    res = pc1.randomize(b1);             // Randomize only b1
    res = pc1.randomize(p1, p2, b1, b2); // Randomize p1, p2, b1, and b2
  end
endmodule
```

Randomizing Scope Variables (std::randomize())

SystemVerilog introduces a scope randomization function, `std::randomize()`, which lets you assign unconstrained or constrained random values to variables that are visible in the current scope.

The syntax of the scope randomization function is as follows:

```
[std::]randomize( [variable_identifier_list] )
       [with {constraint_expression [;constraint_expression]; }];
```

Although it is not specifically mentioned in the LRM, each constraint expression must be followed by a semicolon. For example:

```
success = randomize (Bytes) with {Bytes > 0};  // Invalid
success = randomize (Bytes) with {Bytes > 0;}; // Valid
```

Only integral type variables can be randomized. Wires and non-integral types, such as `real`, are not allowed. The following variable data types are supported:

- `integer`
- `logic`
- `reg`
- `int`
- `bit`
- `byte`
- `shortint`
- `longint`
- Enumerated data types

- Packed structures

  The Cadence implementation supports packed structures, but with the following limitations:

  - Packed structures must be at the module level. Packed structures that are defined within classes cannot be used in calls to scope randomize.
  - Packed structures that are defined in a package cannot be used in calls to scope randomize.
  - Packed structures cannot be used in distribution expressions or set membership expressions.
  - Nested structures are unsupported.
  - When a member of a packed structure is an array, a bit or part select of that packed structure is unsupported. For example:

```
module test_top;
  integer r;

  typedef struct packed {
    bit [7:0] rl;
  } struct_t;

  struct_t sl;

  initial begin
    r = randomize(s1) with {sl.rl[2:1]==3;}; // Unsupported
    r = randomize(sl) with {sl.rl == 6;};    // Supported
  end
endmodule
```

In the following example, `std::randomize()` is called with two variables as arguments, `wide` and `i`. The function assigns new random values to these variables.

```
logic [127:0] wide;
integer i;
int success;

success = randomize(wide, i);
```

The scope randomize function returns `1` if all of the random variables have been set to valid values. Otherwise, it returns `0`.

The scope randomize function can be called with no arguments. In this case, the function acts as a checker, and simply returns its status.

Note: The scope randomization construct is not guaranteed to give the same result across simulators from different EDA vendors. One EDA vendor can, and probably will, return different sequences of random numbers than the Cadence simulator. Furthermore, it is not

guaranteed that the Cadence simulator will give the same sequence of random numbers from one major release to the next.

Specifying Constraints

Constraints determine the legal values that can be assigned to the local scope variables. To specify constraints, use the `with` clause. Enclose the constraint expressions in curly braces.

```
randomize(a, b, c) with {constraint_expression;};
randomize(a, b, c) with {constraint_expression; constraint_expression;};
```

In the following example, the `wide` and `i` variables are given random values, subject to the constraint that `i` must be less than 32.

```
logic [127:0] wide;
integer i;
int success;

success = randomize(wide, i) with {i < 32;};
```

In the following example, the `wide` and `i` variables are given random values, subject to the constraint that `i` must be less than 32, and the `0`th bit of `wide` must be `0`.

```
success = randomize(wide, i) with {i < 32; wide[0] == 0;};
```

The randomization returns 1 if it succeeds, and 0 if it is overconstrained. For example, the following call will return 0:

```
success = randomize(i) with {i < 32; i > 32;};
```

The current implementation of the scope randomization construct supports the SystemVerilog constraint expressions described in "Constraint Blocks" on page 186.

Examples:

- Set membership—For example, the set of legal random values that can be assigned to variable `a` is 3, 4, or 5:

```
module top;
  integer i;
  int success;
  integer a, a2;
  integer num_iterations;

  initial
    begin
      a = 0;
      num_iterations = 8;

      process::self.srandom(20);

      for (i = 0; i < num_iterations; i = i + 1)
        begin
```

```
            // a must be 3, 4, or 5.
            success = randomize(a) with { a inside {3, 4, 5}; };
            if (success != 1) $display("Failed");
            if ( !( (a >= 3) && (a <= 5) )) $display("Failed");
          end
      end
endmodule
```

The following call to the `randomize` function specifies that the legal values that can be assigned to variable `a` are 3, 4, 5, 10, 11, 12, 13, 14, 15, and 16.

```
success = randomize(a) with { a inside {3, 4, 5, [10:16]}; };
```

The negated form of the `inside` operator specifies that the expression is excluded from the set of legal values. The syntax is:

```
with { !(expression inside {set_of_values} ); };
```

In the following example, the first `inside` operator specifies that `a` must be 10–16. The second `inside` operator specifies that `a` cannot be 11–15. Therefore, the set of legal values that can be assigned to `a` is 10 or 16.

```
success = randomize(a) with {
          a inside { [10:16] };
          !(a inside { [11:15] });
          };
```

- Distribution—For example, the following specifies that the set of legal random values that can be assigned to variable `a2` is 100, 200, or 300. Because no weights are specified, the weighted ratio is 1-1-1.

```
success = randomize(a2) with {
          a2 dist {100, 200, 300};
          };
```

The following specifies that `a2` is equal to 100, 200, or 300, with a weighted ratio of 1-2-3.

```
success = randomize(a2) with {
          a2 dist {100 := 1, 200 := 2, 300 := 3};
          };
```

The following specifies that `a2` is equal to 100, 101, 102, 200, or 300, with a weighted ratio of 1-1-1-2-3.

```
success = randomize(a2) with {
          a2 dist { [100:102] := 1, 200 := 2, 300 := 3 };
          };
```

The following example uses the `:/` operator to apply the weight to the range as a whole. Variable `a2` must be 100, 101, 102, 200, or 300, with a weighted ratio of 1/3-1/3-1/3-2-3.

```
success = randomize(a2) with {
          a2 dist { [100:102] :/ 1, 200 := 2, 300 := 3 };
          };
```

- Implication—The following specifies that `payload` must be constrained to less than `100` when `mode` is `SMALL`, greater than `10000` when mode is `LARGE`, or unconstrained when `mode` is neither `SMALL` nor `LARGE`:

```
success = randomize(payload) with {mode == SMALL -> payload < 100;
                                   mode == LARGE -> payload > 10000;};
```

- if...else constraints—The following specifies that `payload` must be constrained to less than `100` when `mode` is `SMALL`, and greater than `10000` when `mode` is `LARGE`:

```
success = randomize(payload) with {if (mode == SMALL) payload < 100;
                                   else if (mode == LARGE) payload > 10000;};
```

Limitations on In-Line Scope Randomization Constraints

In-line scope randomization constraints have the same limitations as constraints that are declared in classes. For more information, see "Limitations on Constraint Blocks" on page 187.

Random Number System Functions and Methods

Several enhancements to random number system functions and methods are described in the SystemVerilog LRM. This section lists the enhancements that are supported in the current release.

The $urandom Function

SystemVerilog offers a function called `$urandom`, which supplements the Verilog `$random` task used to generate random numbers. Unlike `$random`, the `$urandom` function generates *unsigned*, 32-bit random numbers and offers thread stability. The syntax for `$urandom` is

```
value = $urandom [( seed )] ;
```

where *`seed`* is an optional, integral expression used to determine the sequence of the generated numbers. For example:

```
module top;
  int value;
  initial begin
    value = $urandom (2); // Sets the seed for the current process
                          // thread to 2
    value = $urandom;     // Generates a 32-bit random number
  end
endmodule
```

The following example from the IEEE 1800 standard is incorrect, because `$urandom` is a function.

```
bit [64:1] addr;

$urandom( 254 );                // Initializes the generator
addr = {$urandom, $urandom };  // 64-bit random number
number = $urandom & 15;         // 4-bit random number
```

The second line should be:

```
value = $urandom( 254 ); // Initializes the generator
```

or, you can cast the function call to `void`:

```
void'$urandom(254); // Initializes the generator
```

The number generator generates pseudo-random numbers, in that the sequence of generated numbers can be reproduced exactly by using the same seed. To generate truly random numbers, you can specify a seed using a non-deterministic source, such as the current time of day, or you can read values of seeds from a file.

The `$urandom` function offers thread stability, in that the random number values do not depend on the order of thread execution. Refer to the SystemVerilog LRM for more information.

The $urandom_range Function

SystemVerilog offers a new function called `$urandom_range`, which generates a random number within a specified range, and offers thread stability. The syntax for `$urandom_range` is as follows:

```
integer value;
value = $urandom_range(maxval, minval);
```

where *`maxval`* and *`minval`* are unsigned, 32-bit integral expressions that specify the size of the range. In the following example, `$urandom_range` returns a random value from 3 to 6.

```
address = $urandom_range(6,3);
```

If *`minval`* is omitted, zero is used as a default. In the following example, `$urandom_range` returns a random number from 0 to 6.

```
address = $urandom_range(6);
```

If *`minval`* is greater than *`maxval`*, the two values are swapped so that the first value is always larger than, or equal to, the second value. In the following example, `$urandom_range` returns a random value from 2 to 6.

```
address = $urandom_range(2,6);
```

In the following example, `$urandom_range` always returns 4:

```
address = $urandom_range(4,4);
```

The srandom() Method

Random number streams in SystemVerilog are associated with processes (threads). Each thread has its own independent RNG for all randomization calls invoked from that thread, and the RNG is guaranteed to produce the same sequence of random values from one simulation run to the next.

The LRM describes how each RNG is seeded. The seed for each Verilog process is obtained from the next random number generated by the process's parent process. If the process does not have a parent—for example, an `initial` block in a module instance—the seed is obtained from the next random number generated by the RNG of the module instance. You can manually set the RNG seed for subsequent calls to a process's RNG by using the SystemVerilog `srandom()` method.

You can set a seed for a stream with a call to the `srandom()` method of the current process, as follows:

```
// Set the seed for the random number stream associated
// with the current process to 300.
process::self.srandom(300);
```

where

- `process` is a predefined, built-in class that implements fine-grained process control.
- `self` is a static member function of the built-in process class that returns an object representing the current process.
- `srandom` is a built-in member function of every object that manually sets the seed for the random number generation for that object—in this case, the current process.

  Note: While the simulator recognizes this special form for seeding the RNG, the built-in process class is not implemented in the current release, and the `self` static member function is not supported in a general context.

Additional System Functions and Methods

The current release supports the following random number system functions and methods:

- `get_randstate()`
- `set_randstate()`

Random Stability

The current release supports all of the properties of random stability, which are described in the IEEE 1800 standard.

Random Weighted Case (randcase)

The `randcase` statement is a case statement that randomly selects one of its branches, based on a branch weight. The probability of selecting a branch is determined by its weight, divided by the sum total of all weights.

The weights must be non-negative integral expressions, and they cannot be greater than 64 bits wide. Negative signed expressions are treated as unsigned expressions.

For example:

```
randcase
   2 : result = 1;
   5 : result = 2;
   3 : result = 3;
endcase
```

In this example, the first branch is given a weight of 2, the second 5, and the third 3. The sum of all weights is 10. The probability of selecting each branch is as follows:

- 20% probability of selecting branch `result = 1;`
- 50% probability of selecting branch `result = 2;`
- 30% probability of selecting branch `result = 3;`

If a weight of 0 is specified for a branch, that branch will be ignored. If all branches specify 0 weights, no branch is taken, and a warning is generated.

Weights are not limited to constants, but can be arbitrary expressions. For example:

```
randcase
   a + b : result = 1;
   a - b : result = 2;
   32'b0 : result = 3;
endcase
```

Random Sequence Generator (randsequence)

To determine whether an utterance of a language is valid, parsers use the language's BNF notation to generate a program that can validate the utterance. SystemVerilog offers a different approach, through *random sequence generators*.

A random sequence generator randomly generates a valid representation of a language, which you can use to stimulate a design under test. To specify a random sequence generator, use the `randsequence` keyword.

Declaring a randsequence Block

A `randsequence` block is composed of *productions*. A production has a name, and contains a list of production items. Production items are defined in the order in which they will be streamed, and can be classified into *terminals* and *non-terminals*. A terminal is an item that cannot be divided, and needs only its associated block of code; while a non-terminal item can be defined in terms of terminals, or other non-terminals.

The following is the simplified syntax for a `randsequence` block:

```
randsequence ([production_identifier])
 production_list;
endsequence
```

For the complete syntax, refer to the IEEE 1800 standard, Syntax 13-12.

The following example defines a `randsequence` block:

```
randsequence(test)
  // Defines test in terms of its non-terminals
  test: one two done;

  // Defines one as a choice between "up" and "down"
  one: up | down;

  // Defines two as a choice between "smile" and "frown"
  two: smile | frown;

  // Defines terminals that display the production name
  done: { $display("done"); };
  up: { $display("up"); };
  down: { $display("down"); };
  smile: { $display("smile"); };
  frown: { $display("frown"); };
endsequence
```

This example has the following possible sequences:

```
up smile done
up frown done
```

```
down smile done
down frown done
```

In this example, the `test` production is defined in terms of three non-terminals: `test`, `one`, and `two`. When the non-terminals are generated, they are decomposed into their productions. A production item can contain multiple productions, separated by the | symbol. This | symbol indicates a group of choices, from which the generator will choose at random. In this example, `one` specifies a choice between `up` and `down`, and `two` specifies a choice between `smile` and `frown`.

The remaining productions in this example are terminals, in that they are specified solely by their code block. In this example, the `done`, `up`, `down`, `smile`, and `frown` terminals display their production name.

if...else Production Statements

Productions can be made conditional using an `if...else` production statement, which uses the following syntax:

```
if(expression) true_productionitem [else false_productionitem]
```

where *`expression`* is any expression that evaluates to a Boolean value. When the expression is evaluated, if it is true, the *`true_productionitem`* is generated; otherwise, the optional *`false_productionitem`* is generated.

For example:

```
module top;
 int i1 = 100;
 initial begin
 randsequence (P1)
   P1 : if(i1 == 100) P2 else P3;
   P2 : {$display("P2");};
   P3 : {$display("P3");};
 endsequence
 end
endmodule
```

Case Production Statements

The `case` production statement, which is used to select a production from a set of alternatives, has the following syntax:

```
case (expression)
  expression {,expression}: production_item1;
  expression {,expression}: production_item2;
  expression {,expression}: production_item3;
   ...
   default: default_production;
endcase
```

The value of the case expression is evaluated and compared to each of the production expressions, in the order in which they are written. For the first production expression that matches, its corresponding production is generated. If none of the production expressions match the case expression, the optional *default_production* is generated. For example:

```
module top;
 int i1 = 100;
 initial begin
  randsequence (P1)
    P1 : case (i1)
     100 : P2;
     default : P2;
     300 : P3;
         endcase;
    P2 : {$display("P2");};
    P3 : {$display("P3");};
  endsequence
 end
endmodule
```

Note: You can have only one `default` statement within a case production statement.

Repeat Production Statements

To generate a production a set number of times, use the `repeat` production statement. This statement has the following syntax:

```
repeat(expression) production_item
```

where *expression* is a non-negative integral value that specifies how many times to generate the specified *production_item*. For example, the following displays `"My sequence"` ten times:

```
module top;
  int i1 = 100;
  initial begin
   randsequence (P1)
    P1 : repeat (10) P2;
    P2 : {$display("P2");};
   endsequence
  end
endmodule
```

A `repeat` production statement cannot be terminated prematurely. To terminate a `repeat` production statement, you can terminate the entire `randsequence` block using a `break` statement.

Limitations on randsequence Blocks

This section summarizes the `randsequence` features in the SystemVerilog LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- The LRM states that you can use the `rand join` production control to randomly interleave multiple production sequences, while preserving the order of each sequence.

  Cadence does not support interleaved productions.

- The LRM states that you can use arguments within productions. The LRM gives the following example:

```
randsequence(main)
 main: first second gen;
 first:  add | dec;
 second: pop | push;
 ...
 gen (string s = "done"): {$display(s);}; // Argument within a production
endsequence
```

  Cadence does not support arguments within productions.

- The LRM states that productions can return values using the `return` statement. The LRM gives the following example:

```
randsequence (bin_op)
 ...
 bit[7:0] value : {return $urandom};           // Returns an 8-bit value
 string operator : add:=5{return "+";}
                   | dec := 2 {return "-" ;}   // Returns a string
                   | mult := 1 {return "*";}
                  ;
endsequence
```

  Cadence does not support productions with return values.

Debugging Random Constraints

For information about how to debug random constraints using the Tcl command-line interface, *ncsim*, or the SimVision analysis environment, refer to *SystemVerilog in Simulation*.

13

Interprocess Synchronization and Communication

SystemVerilog introduces a powerful set of synchronization and communication mechanisms that you can use to control the interactions that occur between dynamic processes used to model a complex system or a highly dynamic, reactive testbench. These additions include semaphore and mailbox built-in classes, and enhancements to the named event data type.

Semaphores

Note: Semaphores are supported within the Incisive Enterprise Simulator - XL (IES-XL). However, semaphores are not available with the Incisive Enterprise Simulator - L (IES-L).

A semaphore is a built-in class that can be used for synchronization and the mutual exclusion of resources. If you have a shared resource that can only be accessed by a set number of processes *at any given time*, you can use a semaphore to enforce the set of rules to access the resource.

When a semaphore is created, it contains a specific number of keys. When a process wants to use the resource, it must first obtain a key from the semaphore. The waiting queue for a semaphore is first-in first-out. This does not guarantee the order in which processes will arrive at the queue, only that the semaphore will preserve the order of their arrival. Once the maximum number of processes has been reached, all others must wait until a sufficient number of keys is returned.

The following is the prototype for the semaphore class:

```
class semaphore;
  function new(int keyCount = 0);
  task put(int keyCount = 1);
  task get(int keyCount = 1);
  function int try_get(int keyCount = 1);
endclass
```

Semaphores are a part of the built-in `std` package, so they are implicitly imported into the compilation-unit scope of every compilation unit. This means that semaphores are available in any other scope.

Note: The `semaphore` class can be redefined, because the `semaphore` identifier is not a reserved keyword and can be used as a regular identifier.

Examples of creating a queue, dynamic array, or associative array of semaphores.

```
semaphore sm_aa[int]; // AA of semaphore indexed by int
semaphore sm_da[];    //Dynamic array of semaphore
semaphore sm_q[$];    //queue of semaphore
```

You can declare semaphores in the scopes of modules, program blocks, classes, packages, static tasks, static functions, and interfaces.

The `semaphore` class provides the following methods:

- `new()`—The `new()` constructor creates the semaphore, and has an integer argument, `keyCount`, that is used to create the desired number of keys. The `keycount` argument defaults to `0`. For example, the following creates a semaphore called `sem1` with `4` keys.

```
    semaphore wantKeys; // Declares a semaphore data type
    ...
    initial begin
    wantKeys = new(4); // Initializes the semaphore with 4 keys
...
```

- `get()`—The `get()` task obtains keys for a semaphore. The number of keys to obtain is passed as an argument to `get()`. The default number of keys is `1`. For example, the following shows a process that asks for all of the keys and blocks until they are available.

```
    ...
    wantKeys.get(4); // Attempt to procure all keys
    ...
```

  A call to `get()` can be time-consuming because, if all of the desired keys are not available, `get()` blocks subsequent statements and waits for all of the remaining keys.

- `put()`—The `put()` task returns keys so that other processes can use them. For example:

```
    ...
    wantKeys.get(4);
    ...
    wantKeys.put(4);
    ...
```

- `try_get()`—The `try_get()` function is used to obtain keys without blocking. Unlike the `get()` task, the `try_get()` function checks key availability without blocking subsequent calls. If the `try_get()` function is successful in procuring the desired number of keys, it returns `1`. Otherwise, it returns `0`. For example:

```
    ...
    if (wantKeys.tryget(4))
      pr();
    else
    ...
```

Example 13-1 Using Semaphore Methods

```
class c;
  semaphore s[*];
  semaphore s_da[];
  semaphore s_q[$];
endclass

module top;
int i;
c obj;

initial
begin
  obj = new;
  obj.s[1] = new(2);
  obj.s[2] = new(1);
  obj.s_da = new[3];
  obj.s_da[1] = new(1);
  obj.s_q[0] = new(2);
  obj.s.first(i);
  $display("trying to get : %d",obj.s[i].try_get(1));
  $display("trying to get : %d",obj.s_q[0].try_get(1));
  $display("trying to get: %d",obj.s_da[i].try_get(3);
  obj.s.next(i);
  $display("trying to get: %d",obj.s[i].try_get(2));
end
```

Simulation output:

```
ncsim> run
trying to get: 1
trying to get: 1
trying to get: 0
trying to get: 0
```

Limitations on Semaphores

In the current release, the following are supported for semaphores:

- Semaphores are supported in modules, program blocks, classes, packages, static tasks, static functions, and interfaces.
- Semaphores can be declared as public, `static`, `local`, or `protected` members of a class.
- Semaphores can be passed as arguments to tasks, functions, and class methods.

Because semaphores are classes, they are subject to the same limitations as classes. In addition to the class limitations, semaphores also have the following limitations:

- Semaphores cannot be used within `generate` statements or declared on ports.
- Semaphores cannot be used in the index of an associative array.

- While the Cadence implementation supports OOMRs and hierarchical references of semaphores, there is a limitation which does not allow calling methods of the semaphore class using hierarchical references. However, you can make such method calls using local objects after assigning an out-of-scope instance of semaphores to a local declaration of the same kind.

  The following is an example of using OOMRs of semaphores:

```
module top;
  semaphore sm;
  integer i = 2;
  sm = new(i);
  sm.get(i);
  bot bot_inst();
endmodule

module bot;
  semaphore sm_1;
  integer j = 2;
  sm_1 = top.sm;
  sm_1.put(j);
endmodule
```

- The LRM states that semaphores “can be used as base classes for deriving additional higher level classes.” This feature is not supported in the current release.
- According to the LRM, semaphores can use the `std::` syntax. With this syntax, the code can easily distinguish between semaphores and any overrides of semaphores. The current release does not support this syntax.

Mailboxes

Note: Mailboxes are supported within the Incisive Enterprise Simulator - XL (IES-XL). However, mailboxes are not available with the Incisive Enterprise Simulator - L (IES-L).

For a complete example using a mailbox that you can download and run, see the *SystemVerilog Engineering Notebook*.

Mailboxes provide a form of *direct* communication between processes, where data can be exchanged between a sending process and its designated recipient.

SystemVerilog mailboxes operate like real mailboxes. A process places a message inside a mailbox so that another process can retrieve it. The message stays within the mailbox until it is retrieved. If the receiving process checks the mailbox before the sending process has deposited the message, the receiving process can either wait for the message to arrive, or check back at a later time.

A SystemVerilog mailbox is a built-in class. The following is the prototype for the `mailbox` class:

```
class mailbox #(type T = dynamic_singular_type);
  function new(int bound = 0);
  function int num();
  task put(T message);
  function int try_put (T message);
  task get(ref T message);
  function int try_get(ref T message);
  task peek (ref T message);
  function int try_peek (ref T message);
endclass
```

Note: The `ref` arguments used in the `get()`, `try_get()`, `peek()`, and `try_peek()` methods are subject to the limitations described in "Limitations on Passing Task and Function Arguments by Reference" on page 175.

The following is the syntax for creating a mailbox:

```
mailbox mailbox_name;
```

For example:

```
mailbox mymailbox;
mailbox mbox_aa[int]; // AA of mailbox indexed by int
mailbox mbox_da[];    //Dynamic array of mailbox
mailbox mbox_q[$];    //Queue of mailbox
```

You can declare mailboxes in the scopes of module, program block, classes, packages, static tasks, static functions, and interfaces.

In the current release, you can use the Tcl `describe` command on the handle of a mailbox to display the number of messages inside the mailbox.

```
describe -handle 2
```

Mailbox Methods

The built-in mailbox class provides the following methods:

- `new()`—The `new()` constructor creates the mailbox. It has an integer argument, `bound`, that is used to determine whether the mailbox is bounded or unbounded. If the `bound` argument is set to a non-zero number, the mailbox is bounded, and the number represents the maximum number of messages that the mailbox can contain. When a process tries to place a message in a mailbox that has reached its bound limit, the process will be suspended until space is available.

  When the `bound` argument is set to zero, the mailbox is unbounded, and can contain an unlimited number of messages. Unbound mailboxes never suspend a thread in a send operation.

  The default bound argument is 0.

Example:

```
mymailbox = new(0);       // mailbox is unbounded, default
mymailbox = new(5);       // mailbox queue can contain 5 messages
mymailbox = new(-10);     // generates warning; default of 0 is used instead
mymailbox_da = new [5]    // dynamic array of mailbox of sized 5
```

- `num()`—The `num()` method returns the number of messages currently in the mailbox.

  For example:

  ```
  i = mymailbox.num();
  ```

- `put()`—The `put()` method places a message in the mailbox. If a process tries to use the `put()` method on a mailbox that is full, the call will block until space is available. For example, the following tries to put a message called `msg1` in `myMailbox`.

  ```
  mymailbox.put(c1); //c1 is a class handle
  ```

  Note: The `put()` method stores the messages in the mailbox in FIFO order. In other words, the first message that is put in is the first one to pop out.

- `try_put()`—The `try_put()` method is similar to `put()`, except that it is non-blocking. If the mailbox is not full, the method places the message in the mailbox and returns a positive integer. If the mailbox is full, this method returns `0`.

  Note: The `try_put()` method stores the messages in the mailbox in FIFO order.

- `get()`—The `get()` method retrieves one message, if one is available, from the specified mailbox's queue. If the message is not available, the call blocks until the message is available. If the type of the message variable does not match the type of message in the mailbox, a run-time error occurs. For example:

  ```
  class c;
  endclass

  class d extends c;
  endclass

  class x;
  endclass

  c c1;
  d d1;
  x x1;
  ...
  mymailbox.put(d1);
  mymailbox.get(x1); // Invalid. Types do not match.
  mymailbox.get(c1); // Valid.
  ```

- `try_get()`—The `try_get()` method is similar to `get()`, except that it is non-blocking. If the mailbox is empty, this method returns 0. If the type of the message variable does not match the type of message in the mailbox, this method returns a

negative integer. If a message is available, and its type matches the message variable type, this method retrieves the message and returns a positive integer. For example:

```
j = mymailbox.try_get(c1); // Value of j will be 0.
mymailbox.put(c1);
j = mymailbox.try_get(x1); // x1 does not change and value of j will be -1.
j = mymailbox.try_get(c2); // Value of j will be 1.
```

- `peek()`—The `peek()` method copies one message, if it is available, from the mailbox. This method is useful when you want to copy a message from a mailbox without deleting it from the mailbox. If the message is not available, the call blocks until the message is available.

- `try_peek()`—The `try_peek()` method is similar to the `peek()` method, except that it is non-blocking. If the mailbox is empty, this method returns 0. If the type of the message variable does not match the type of message in the mailbox, this method returns a negative integer. If a message is available, and its type matches the message variable type, this method copies the message and returns a positive integer.

Note: You can pass bit selects of an array as arguments to the mailbox methods.

Example 13-2 Using Mailbox Methods

```
class c;
mailbox mbx1[*];
mailbox mbx1_da[];
endclass

module top;
integer i,j,k,l;
int p1, p2;
longint idx;
c obj;

initial
begin
  obj = new;
  obj.mbx1_da = new[5];
  obj.mbx1_da[1] = new(2);
  obj.mbx1[1] = new(0);
  obj.mbx1[2] = new(-2);

  $display ("handle of mbx[1] = %d , mbx[2] = %d\n", obj.mbx1[1], obj.mbx1 [2]);
  $display ("handle of mbx_da[1] = %d, mbx_da[2] \n", obj.mbx1_da[1],

  obj.mbx1_da[2]);

  #1;
  obj.mbx1[1].put(1);
  p1= obj.mbx1[1].try_put(2);
  if (p1 == 0)
    $display("mbx1[1] is full");
  obj.mbx1[2].put(3);
  p2=obj.mbx1[2].try_put(4);
  if (p2 == 0)
    $display("mbx1[2] is full");
  $display("number of messages in a mailbox mbx1[1] \n",obj.mbx1[1].num());
  $display("number of messages in a mailbox mbx2[1] \n",obj.mbx1[2].num());
```

```
  $display("number of messages in a mailbox mbx1 \n",obj.mbx1.num());
  $display("size of  mailbox mbx1_qa  \n",obj.mbx1_da.size());

  #1;
  obj.mbx1.first(idx);
  obj.mbx1[idx].peek(i);
  p1 = obj.mbx1[idx].try_peek(j);
  if (p1 == 0)
    $display("no more messages in mbx1[1]");
  obj.mbx1.next(idx);
  obj.mbx1[idx].peek(k);
  p2 = obj.mbx1[idx].try_peek(l);
  if (p2 == 0)
    $display("no more messages in mbx1[2]");
  $display("number of messages in a mailbox mbx1[1] \n",obj.mbx1[1].num());
  $display("number of messages in a mailbox mbx2[1] \n",obj.mbx1[2].num());
  $display("number of messages in a mailbox mbx1 \n",obj.mbx1.num());

end

always @(i)
  $display ("value of i after fetching the message = %d\n", i);

always @(j)
  $display ("value of j after fetching the message = %d\n", j);

always @(k)
  $display ("value of k after fetching the message = %d\n", k);

always @(l)
  $display ("value of l before fetching the message = %d\n", l);

Endmodule
```

Simulation output:

```
ncsim> run
obj.mbx1[2] = new(-2);

ncsim: *W,MBXKBG (./test.v,18|17): Negative key value encountered for a mailbox.
The default of 0 will be used instead [SystemVerilog].

handle of mbx[1] = 7 , mbx[2] = 9
handle of mbx_da[1] = 4 , mbx_da[2] = 9

number of messages in a mailbox mbx1[1]

2

number of messages in a mailbox mbx2[1]

2

number of messages in a mailbox mbx1_da[1]

$display("number of messages in a mailbox mbx1_da[1] \n",obj.mbx1_da[2].num());

ncsim: *W,MBOHDL (./test.v,35|75): A method was encountered on a null mailbox
[SystemVerilog].

0

number of messages in a mailbox mbx1

2

number of messages in a mailbox mbx1_da

5

number of messages in a mailbox mbx1[1]
```

```
2
number of messages in a mailbox mbx2[1]
2
number of messages in a mailbox mbx1
2
value of i after fetching the message = 1
value of j after fetching the message = 1
value of k after fetching the message = 3
value of l before fetching the message = 3
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> heap -report
Dynamic Array: 1 [ 48 bytes ]
Assoc Array: 1 [ 96 bytes ] including 1 prototypes and 2 elements
Class: 1 [ 32 bytes ]
Mailbox: 3 [ 304 bytes ]
Mailbox Entry: 5 [ 160 bytes ]
total objects: 15 [ 640 bytes ]
ncsim> exit
```

Limitations on Mailboxes

The following summarizes the mailbox features in the SystemVerilog standard that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Only the following data types can be passed as arguments to the `put()`, `try_put()`, `get()`, `try_get()`, `peek()`, and `try_peek()` mailbox methods:
  - Class handles
  - `shortint`, `int`, `longint`, `byte`, `integer`
  - `bit`, `logic`, `reg`
  - Packed arrays of types `bit`, `logic`, and `reg`
- Mailboxes and fixed arrays of mailboxes can be declared only within modules, program blocks, tasks, functions, classes, interfaces, and packages.
- While the Cadence implementation supports OOMRs and hierarchical references of mailboxes, there is a limitation which does not allow calling methods of the mailbox class using hierarchical references. However, you can make such method calls using local objects after assigning an out-of-scope instance of mailboxes to a local declaration of the same kind. The following example illustrates this scenario:

```
// This is not allowed:
top.mbox.put();
// This is allowed:
l_mbox = top.mbox;
l_mbox.put();
```

where `mbox` is an instance of the mailbox defined in module `top`.

The following is an example of using OOMRs of mailboxes:

```
module top;
  mailbox #(int) mbox;
  int i = 10, j;
  initial begin
    mbox = new;
    mbox.put(i);
    mbox.get(j);
    $display("From top:  mbox", j);
  end
  bot bot_inst();
endmodule
module bot;
  mailbox #(int) mboxl;
  initial begin
    mboxl = top.mbox;
    $display("From bot: mboxl >> ", mboxl.num());
  end
endmodule
```

- Mailboxes cannot be passed as module ports to tasks and functions. You can, however, pass mailboxes as arguments to tasks, functions, and class methods.
- Although mailboxes are a built-in class, they cannot be extended.
- Parameterized mailboxes are not supported.
- Initializing a fixed array of mailboxes is not support.
- If `mb1` and `mb2` are fixed arrays of mailboxes, `mb1=mb2` is not supported.
- Strings are not supported within parameterized mailboxes.

Events

In the current release, events can be

- Passed by value to tasks and functions
- Declared within a class or package

The current release does not support events within unpacked structures.

Non-Blocking Event Trigger

In Verilog, event triggers act like blocking assignments. They immediately trigger the event at the point at which the trigger is executed, and block the execution of subsequent code until the event finishes execution.

The triggering of named events in Verilog can often lead to race conditions. The following example illustrates the problem.

```
module events;
  event e, go;

  always @(go)
    begin
      $display($time,,"Always 1 - triggering e");
      -> e;
    end

  always @(go)
    begin
      $display($time,,"Always 2 - about to wait on e");
      @(e) $display($time,,"Always 2 - wakeup on e");
      $finish;
    end

  always #5 -> go;
endmodule
```

The run-time output of this description might be

```
5 Always 1 - triggering e
5 Always 2 - about to wait on e
10 Always 1 - triggering e
10 Always 2 - wakeup on e
```

or

```
5 Always 2 - about to wait on e
5 Always 1 - triggering e
5 Always 2 - wakeup on e
```

SystemVerilog introduces a nonblocking event trigger operator, `->>`. The syntax is as follows:

```
->> [delay_or_event_control] hierarchical_event_identifier;
```

The `->>` operator does not block the execution of subsequent code. The statement creates a nonblocking assign update event at the time in which the delay control expires, or the even- control occurs. The triggered event is scheduled to occur at the end of the simulation time slot in the nonblocking assignment region of the simulation cycle.

The following code shows the first `always` block from the example shown above rewritten to use the nonblocking trigger operator.

```
  always @(go)
    begin
      $display($time,,"Always 1 - triggering e");
```

```
      ->> e;  // Nonblocking trigger guaranteed to run after
              // both always blocks.
    end
```

The legal outcomes of the above description are

```
5 Always 2 - about to wait on e
5 Always 1 - triggering e
5 Always 2 - wakeup on e
```

or

```
5 Always 1 - triggering e
5 Always 2 - about to wait on e
5 Always 2 - wakeup on e
```

Note: Although the order of the `always` blocks is still arbitrary, both orderings cause the wakeup on event `e` to occur at time 5.

Persistent Trigger

In Verilog, events have no logic value or duration. Processes can watch for an event to trigger, but if a process is not watching at the exact moment that an event is triggered, the event will go undetected. SystemVerilog introduces the `triggered` property, which enhances the event data type by allowing an event to persist throughout the time step in which the event is triggered. The syntax is as follows:

hierarchical_event_identifier`.triggered`

The `triggered` event property evaluates to true, as long as the given event is triggered within the current time unit. Otherwise, the triggered event property evaluates to false.

The addition of the `triggered` property helps resolve a common race condition, illustrated in the following example:

```
begin
-> eventA
...
end
...
@(eventA)
...
```

In this example, `eventA` must occur before the rest of the code can execute, which can cause a race condition when the event control and `eventA` occur at the same time. In this case, the `wait` might either unblock, or wait until the next `eventA`. You can eliminate this race condition using the `triggered` property and a `wait` statement:

```
begin
-> eventA
...
end
...
```

```
wait(eventA.triggered)
...
```

This code will unblock the calling process as long as the `wait` executes before or at the same time unit as the event trigger.

When the `.triggered` property is used within the `wait()` construct, the condition waits for the *`event`*`.triggered` property to become true. If you reach the `wait()` statement and the property is true, execution proceeds immediately; otherwise, it will wait for the property to become true.

The event.triggered construct is most useful when used within the wait() construct. While it seems natural to replace @(event) with @(event.triggered), combining the semantics of the .triggered property and edge-activated event control @() could produce undesired results.

Note: The current release does not support the `.triggered` property on OOMRs to a named event. You can only reference a named event from the scope in which it was declared.

Event Variables

In SystemVerilog, events behave like variables, in that they can be assigned to one another, assigned a special `null` value, or compared against each other.

If you have two event type variables called `e1` and `e2`, you can

- Assign one to the other

  When an event is assigned to another event, the two events become merged and they share the synchronization queue of the event on the right-hand side of the assignment. However, the assignment affects only subsequent event control or wait operations. For example, if there are processes waiting on an event at the time it is merged with another event, the waiting process will never unblock.

```
e1 = e2;      // Merges the two event variables
              // After this, events waiting on e1 will never unblock
fork
  begin
   -> e1;     // Also triggers e2
  end

  begin
   wait(e1.triggered);  // Unblocks wait process
   ...
  end
```

```
begin
 wait(e2.triggered);  // Also unblocks wait process
end
```

- Reclaim an event

  You can reset an event type variable by assigning it a special `null` value. This disconnects the event variable from its synchronization queue, making the event variable's resources available again. For example:

  ```
  event e1 = null; // Resets e1
  ```

  Event controls and wait operations on a `null` event are undefined, and triggering a `null` event has no effect. For example:

  ```
  @ e1;                 // Undefined
  wait (e1.triggered); // Undefined
  -> e1;               // No effect
  ```

- Compare them

  You can use comparison operators (`==`, `!=`, `===`, and `!==`) to compare an event variable to another event variable, or to `null`. Or, you can check for a Boolean value that will be 0 if the event is `null`, or 1 otherwise. For example:

  ```
  if (e1 == null)
    $display("e1 is null");
  if (e1)
    e1 = e2; // Merges e1 and e2 if e1 is not null
  ```

14

Clocking Blocks

Note: Clocking blocks are supported within the Incisive Enterprise Simulator - XL (IES-XL). However, clocking blocks are not available with the Incisive Enterprise Simulator - L (IES-L).

In Verilog-2001, module ports are used to model communication between blocks. SystemVerilog extends this feature by introducing the interface construct, which encapsulates the communication between blocks. Although an interface can specify the signals and nets through which a testbench communicates with its DUT, it cannot explicitly specify timing and synchronization requirements. To address this limitation, SystemVerilog introduces the *clocking block* construct. A clocking block

- Identifies a *clocking domain*, which encapsulates clock signals, and the timing and synchronization requirements of blocks wherein the clock is used.
- Separates time-related details from the structural, functional, and procedural elements of a testbench.
- Helps define the testbench based on transactions and cycles (cycle-based methodology), as opposed to signals and transition times (event-based methodology).
- Simplifies the creation of a testbench that does not have race conditions with the DUT.

For a complete example using clocking blocks that you can download and run, refer to the *SystemVerilog Engineering Notebook*.

Declaring a Clocking Block

A simple clocking block declaration is as follows; see the SystemVerilog LRM, Syntax for the complete syntax.

```
clocking [clocking_identifier] clocking_event;
clocking_items
endclocking[: clocking_identifier]
```

For example, the following defines a clocking block called `c1` that is to be clocked at signal `clk`.

```
module test_top;

wire clk;
wire a, b;

clocking c1 @clk;
  default input #1ns;
  default output #2ns;
  input posedge a;
  output negedge b;
endclocking: c1

endmodule
```

The second line of the clocking block declaration specifies a default input skew of `1ns`, which means the testbench must sample input signals at `1ns` before the clock edge. The third line specifies a default output skew of `2ns`, which means that outputs must be driven `2ns` after the clock edge. The fourth and fifth lines show how you can reference a clock edge to sample a value or drive a stimulus.

The items and events specified within a clocking block declaration are order-independent—there is no required ordering within the scope of a design unit.

Clocking blocks can be declared within modules, interfaces, and program blocks. Clocking blocks cannot be nested. They cannot be declared within functions, tasks, or packages, or outside all declarations in a compilation unit.

A clocking block is both a declaration and an instance of that declaration. You do not need a separate instantiation.

You can access an individual signal within a declared clocking block by using its name, the dot (`.`) operator, and the signal identifier. For example:

```
c_counter.sig // Accesses sig in clocking block c_counter
```

You can access the clocking event of a clocking block directly, by using the clocking block name. For example:

```
clocking dram@(posedge phi1);
     inout data;
endclocking
...
always @(dram) // Equivalent to @(posedge phi1)
...
```

The following lists the features that are supported in the IEEE 1800 standard, but are not supported in the Cadence implementation.

- The LRM specifies that names for clocking blocks are optional. The Cadence implementation supports only unnamed, `default` clocking blocks that do not contain clocking items. Unnamed, non-default clocking blocks are not allowed, and produce an

error message in the simulator. See "Defining Default Clocking Blocks" on page 231 for more information about default clocking blocks.

- The LRM allows clocking block declarations within `generate` loop statements. This is not supported in the Cadence implementation.

Types of Clocking Items

Cadence provides support for multi-dimensional vectors and unpacked arrays as types of clocking items. Unpacked arrays much be of fixed sizes (that is, Verilog memories). Clocking items that are non-fixed size arrays (for example, dynamic arrays, queues, strings, and associative arrays) are not supported. The supported array element types include vectors, logic scalars and reals. The array element type can be another unpacked array type (that is, multi-dimensional unpacked arrays are supported).

Restrictions of Clocking Items Types

- A clocking item cannot be associated with an unpacked array if the array is addressed using an OOMR.
- A clocking item cannot be associated with an unpacked array of nets.

These limitations are illustrated by the following examples. The lines labeled LINE 9 and LINE 10 will not be supported with this enhancement; LINE 11 will be supported.

```
module data;
  int x[8];
  endmodule

module m(input clk, input var int y[8]);
  data d();
  wire w[8];
  clocking c @(posedge clk);
    input a = d.x; // LINE 9: UNSUPPORTED (OOMR TO UNPACKED ARRAY)
    input w;       // LINE 10: UNSUPPORTED (UNPACKED ARRAY OF NETS)
    input y;       // LINE 11: SUPPORTED
  endclocking
endmodule
```

- Associating a clocking item with an unpacked array that is implemented as a sparse array is not supported.

Example 14-1 Simple Input Case of Clocking Items

This example shows an input clocking item taking on its new value on the positive edge of a clock change:

```
module top;
  int a[2];
```

```
  reg clk;
  clocking cb @(posedge clk);
    input a;
  endclocking
  initial begin
    #1 clk = 0;
    #1 a[0] = 1;
    #1 a[1] = 2;
    #1 clk = 1;
  end
  always @(cb.a[0], cb.a[1])
  $display("At %0d cb.a is {%0d, %0d}", $stime, cb.a[0], cb.a[1]);
endmodule
```

Output:

```
At 4 cb.a is {1, 2}
```

Example 14-2 Simple Output Case of Clocking Items

This example shows an output clocking item driving one of the array elements, with the new value taking effect when the clocking block is activated by an event:

```
module top;
  int a[2];
  event e;
  clocking cb @(e);
    output a;
  endclocking
  initial begin
    #1 cb.a[0] <= 1;
    #1 a[1] = 2;
    #1 ->e;
  end
  always @(a[0], a[1])
    $display("At %0d a is {%0d, %0d}", $stime, a[0], a[1]);
  endmodule
```

Output:

```
At 2 a is {0, 2}
At 3 a is {1, 2}
```

Defining Default Skews and Clocking Direction

SystemVerilog allows default skews for a single clocking block to be specified on multiple lines or on a single line. For example, you can use:

```
clocking c1 @clk;
     default input #10ns;
     default output #2ns;
     ...
endclocking: c1
```

or

```
clocking c1 @clk;
     default input #10ns output #2ns;
     ...
endclocking: c1
```

The SystemVerilog LRM gives the following syntax for defining default skews and clocking direction on a single line:

```
input [clocking_skew] output [clocking_skew];
```

The Cadence implementation allows the reverse order. For example:

```
output [clocking_skew] input [clocking_skew];
```

A clocking block cannot have multiple default skews with the same type. Although types can have independent delay controls and edge skews, they can have only one of each. For example, you cannot have multiple default `input` skews:

```
clocking c1 @clk;
     default input #10ns;
     default input #2ns; // Invalid
```

However, an edge skew default does not conflict with a delay control skew default. For example, the following specifies that all inputs will have a `1ns` input skew, but will also have a positive edge skew.

```
clocking c1 @clk;
     default input #1ns;
     default input posedge; // Valid
```

Defining Clocking Items

SystemVerilog allows you to declare clocking items on multiple lines. For example, the following lines

```
clocking c1 @clk;
    input posedge a;
    output negedge a;
endclocking
```

are the same as

```
clocking c1 @clk;
    input posedge output negedge a;
endclocking
```

When you declare clock items on multiple lines, SystemVerilog treats them as a summation of the various directions and skews. A skew is not unset if a clocking item is declared on multiple lines. For example:

```
clocking c1 @clk;
    input posedge a;
    input a;        // This line does not unset anything
endclocking
```

Each direction of a clocking item—input or output—can have only one skew edge, either `posedge` or `negedge`, and one delay control. For example, you cannot have multiple skew edges and delay controls for the same direction of a clocking item:

```
clocking c1 @clk;
    input posedge a;
    input negedge a; // Invalid
    output negedge a;
    output #1 a;
    output #2 a;     // Invalid
endclocking
```

Using Hierarchical Expressions

In SystemVerilog, you can use a *hierarchical expression* in place of a local port. This feature lets you specify that the signal that will be associated with the clocking block is specified by its hierarchical name. A hierarchical expression is introduced using the equal sign (=). For example:

```
clocking c1 @clk;
     input a = top.cpu.state;
endclocking
```

You cannot designate multiple hierarchical expressions for the same clocking item. In the following example, the second designation causes an error message, because `a` already has the hierarchical expression `top.m.a`:

```
clocking c1 @clk;
    input a = top.m.a;
    input posedge a = top.m.b; //Invalid
endclocking
```

The following lists differences between the IEEE 1800 standard and the Cadence implementation:

- The Cadence implementation does not support the following use of hierarchical expressions:

  ```
  clocking c @clk;
       input a.b;
  endclocking
  ```

  Instead, the Cadence implementation supports the following:

  ```
  clocking c @clk;
       input in = a.b;
  endclocking
  ```

- The Cadence implementation does not support the concatenation `{...}` syntax used in the example in the IEEE 1800 standard. Instead, the Cadence implementation supports the following usage:

```
clocking mem @(clock);
     input instruction = top.cpu.instr;
endclocking
```

Defining Default Clocking Blocks

You can specify a default clocking block for all cycle delay operations that occur within a given module, interface, or program. You can only specify one default clocking block within a module, interface, or program, and that clocking block is only valid within the scope containing the clocking specification. In the following example, default clocking block `c1` is only valid within the scope of module `m1`:

```
module m1;
     wire a;
     wire clk;
     default clocking c1;
        input a;
     endclocking
endmodule
```

The LRM Syntax 15-4 allows a short-form designation of a default clocking block, where the default is designated separately from its declaration. For example:

```
module top;
     wire a;
     wire clk;
     default clocking c1; // Designates the default clocking block

     clocking c1 @clk;   // Clocking block declaration
       input a;
     endclocking
endmodule
```

Note: The implied use of the short-form designation is for nested modules and interfaces. The simulator does not support nested design units, so the Cadence implementation supports only clocking blocks in non-nested design units.

Specifying Cycle Delays and Clocking Drives

The `##` operator is used in the testbench to delay execution by a specified number of clocking events or clock cycles. This feature is called a *cycle delay*. For example:

```
## 2 // Wait 2 clocking events, using the default clocking block
```

The makeup of a clocking event depends on the default clocking block. If a default clocking block has not been specified for the current module, interface, or program, the compiler issues an error message.

Cycle delays can be used in two kinds of statements: `procedural_timing_control_statement` and `clocking_drive`.

The Verilog `procedural_timing_control_statement` thread allows cycle delays to be specified alone. For example:

```
initial
  begin
   ##1;
  end
```

However, the Verilog `procedural_timing_control_statement` thread does not allow cycle delays on the right-hand side of statements. For example:

```
default clocking c1 @clk;
 output a;
endclocking

initial
 begin
  c1.a <= ##1 b; // Not allowed by procedural_timing_control_statement
 end
```

SystemVerilog introduces the `clocking_drive` statement, which allows cycle delays on the right-hand side of statements. For example, the following statement is valid in SystemVerilog. It specifies to remember the value of `b`, then drive `Data` two clock cycles later:

```
cl.Data <= ##2 b;
```

The following summarizes the Cadence implementation for clocking drives:

- You can only use cycle delays on the right-hand side of non-blocking assignments, and the left-hand value must be a clocking item. For example:

```
cl.a = ##1 b; // Invalid, the statement must be non-blocking
a <= ##1 b;   // Invalid, the left-hand side must be a clocking item
cl.a <= ##1 b; // Valid, if cl.a is a clocking item
```

- When a cycle delay is specified on the left-hand side of a non-blocking assignment, and the left-hand value is a clocking item, it is considered a clocking drive. Otherwise, it is considered a `procedural_timing_control_statement` thread. For example:

```
##1 cl.a <= b; // Considered a clocking drive
##1 a <= b;    // Considered a procedural_timing_control_statement
               // followed by a non-blocking assignment
```

- A cycle delay, regardless of whether it is on the left-hand or right-hand side of an assignment, is defined by the clocking block of the signal being driven. For example:

```
##1 bus.Data <= 8' hz; // Wait 1 bus cycle, then drive Data
```

```
bus.Data <= ##2 8'hz;  // Wait 2 bus cycles, then drive Data
```

- When there is a cycle delay on both sides of an assignment, it is considered a `procedural_timing_control`, followed by a clocking drive. For example:

  ```
  ##1 c1.a <= ##2 b;
  ```

  is the same as

  ```
  ##1;
  c1.a <= ##2 b;
  ```

Debugging Clocking Blocks

For information about how to debug clocking blocks using the Tcl command-line interface or the SimVision analysis environment, refer to *SystemVerilog in Simulation*.

15

Program Blocks

Note: Program blocks are supported within the Incisive Enterprise Simulator - XL (IES-XL). However, program blocks are not available with the Incisive Enterprise Simulator - L (IES-L).

SystemVerilog introduces a *program block* construct. A program block, similar to a module, facilitates the creation of a testbench, but has special syntax and semantic restrictions. A program block

- Provides an entry point to the execution of testbenches
- Acts as a scope for the data contained in the program block
- Provides a syntactic context that schedules events in the reactive region
- Uses a special `$exit()` system task

For a complete example using program blocks that you can download and run, refer to the *SystemVerilog Engineering Notebook*.

Declaring a Program Block

A simple `program` declaration is as follows; see the SystemVerilog LRM, Syntax 16-1 for the complete syntax:

```
program program_identifier[(port_list)];
program_items
endprogram[: program_identifier]
```

Although program blocks and modules use different keywords, they follow the same general format. For example, port declarations and end labeling are the same in both constructs.

Supported Constructs for Program Blocks

Program blocks are limited in the type of constructs they can contain. Specifically, the simulator supports the following BNF constructs within a program block:

- `class_constructor_declaration`

- class_declaration
- clocking_declaration
- concurrent_assertion_item
- concurrent_assertion_item_declaration
- continuous_assign
- covergroup_declaration
- data_declaration
- function_declaration
- genvar_declaration
- initial_construct
- local_parameter_declaration
- module_or_generate_item_declaration
- net_declaration
- non_port_program_item
- overload_declaration
- package_or_generate_item_declaration
- parameter_declaration
- specparam_declaration
- timeunits_declaration

Unsupported Constructs

The IEEE 1800 standard indicates that you cannot include instantiation objects, `generate` blocks, `specify` blocks, `defparams`, `always` blocks, UDPs, modules, interfaces, or other programs within a program block. Specifically, the simulator does not support the following BNF constructs within a program declaration:

- generated_module_instantiation
- specify_block
- program_declaration
- module_declaration

- parameter_override
- gate_instantiation
- udp_instantiation
- module_instantiation
- interface_instantiation
- program_instantiation
- bind_directive
- net_alias
- final_construct
- always_construct

Nesting Program Blocks

The IEEE 1800 standard states that program declarations can be nested within modules or interfaces. For example:

```
module test(...)
  int shared;

  program p;
  . . .
  endprogram: p

  program p1;
  . . .
  endprogram: p1

endmodule:test
```

Note: The Cadence implementation *does not* support program block declarations that are nested within modules, packages, or interface declarations. Also, program block declarations are allowed only at the top-most level.

Working with Variable Assignments

Use blocking assignments (=) to update the values of variables that are local to a program block. Use non-blocking assignments (<=) to update non-program variables, such as module variables. If you use a non-blocking assignment with a program variable, or a blocking assignment with a non-program variable, you will get an error message. For example:

```
module design(input wire A);
 int B, C;
endmodule

program test(output reg A);

 integer int_number;
 reg a;

 initial begin

 // Valid variable assignments
 top.t.int_number = 15;      // Program variable
 top.d.C <= 3;              // Non-program variable

 // Invalid variable assignments
 top.t.a <= int_number + 1;    // Program variable
 top.d.B = 5;                  // Non-program variable

 end

endprogram

module top;
    wire A;
    design d(A);
    test t(A);
endmodule
```

Referencing Program Block Variables

References to program block variables can exist within program blocks. However, the IEEE 1800 standard does not allow references to program block variables from outside a program block.

You can reference program block instances from within modules in the traditional hierarchical fashion. However, you cannot reference program block instances from within program blocks.

Instantiating Program Blocks

Program blocks, modules, and primitives are instantiated in the same way. The current release does not support arrays of instances within program blocks. The current implementation does not support program block instantiation to other languages, such as VHDL or SystemC.

New Program Design Unit

Although program blocks are very similar to modules, the LRM definition classifies them as a different type of Verilog design unit. To account for this difference, the NC library system has been enhanced to manage the new design unit type.

If you use the `-messages` option when you compile your source files, the output displays `program`. For example:

```
% ncvlog -nocopyright -messages -sv test.v
file: test.v
     program worklib.P
           errors: 0, warnings: 0
...
```

The default view name for a program is `program`. For example, `worklib.P:program`.

You can query the library system for program objects using the *ncls* utility with the `-program` option.

```
% ncls -program
```

Understanding the $exit() Control Task

Aside from normal simulation tasks, like `$stop` and `$finish`, a program can use the `$exit` control task to terminate a program block. The following summarizes the functionality of the `$exit` control task within the Cadence implementation:

- `$exit()` is a system task that can be called only within program blocks.

  You cannot invoke or enable the `$exit()` call from within a function. You can, however, invoke the `$exit()` task from an `initial` block or task within a program block.

- Program blocks can call `$exit()` explicitly or implicitly.

  To implicitly call `$exit()`, a program block must contain an `initial` block. In the implicit case, the `$exit()` task is called after all `initial` blocks execute their last statement, regardless of whether that statement has events or processes that occur at a later time. In the following example, `top.p1.r` goes through only two transitions, from unset to `1`, then from `1` to `0`. The remaining transitions are disabled, because the program block contains a call to `$exit()`.

  ```
  program P1;
      integer r;
      initial
          #1 r = 1;
      initial
          #10 $display("Keep this going!");
  endprogram
  ```

```
program P2;
    initial
      begin
        #2 top.p1.r = 0;
        #2 top.p1.r = 1;
        $display("P2 - First Initial Block.");
      end
    initial
        #3 $exit();

    initial
        forever @(top.p1.r)
        $display("r: %b",top.p1.r);
endprogram

module top;
P1 p1();
P2 p2();
endmodule
```

- A call to `$exit()` terminates the program block.

  A call to `$exit()` disables all `initial` blocks in the specified program block, and their sub-processes, following the standard rules for disable. In the following example, the call to `$exit()` disables both `initial` blocks, including the first block that has already run.

  ```
  program P1;
          initial
               $display("First");

          initial
              #2 $exit();
  endprogram
  ```

  The simulator does not terminate continuous assignments, tasks, and functions that are defined within the program block, but are called from other blocks. The simulator will, however, terminate outstanding non-blocking assignments that are sub-processes of any of the `initial` blocks.

  If the `$exit()` call is within a task, the simulator terminates the program block that has the `initial` block with the call to `$exit()`—not the program block in which the task is defined.

- The simulator calls `$finish()` when all program blocks have exited, either implicitly or explicitly.

  In the following example, each program block has an `initial` block that runs with an implicit call to `$exit()`, but only the last implicit call to `$exit()` triggers the `$finish()` call.

  ```
  ...
  program P2;
        initial
            begin
              $display("In program block P2.");
  ```

```
          end

endprogram

program P;
     initial
          begin
            $display("In program block P.");
          end

endprogram
...
```

- Calling `$exit()` terminates all processes spawned by the current program.

16

Assertions

SystemVerilog assertions are available only if you have an Incisive license. Support for SystemVerilog assertions is documented in the *Assertion Writing Guide* and in the *SVA Quick Reference*.

Immediate Assertions

Support for SystemVerilog concurrent assertions is documented in the *Assertion Writing Guide* and in the *SVA Quick Reference*.

Concurrent Assertions

Support for SystemVerilog concurrent assertions is documented in the *Assertion Writing Guide* and in the *SVA Quick Reference*.

17

Hierarchy

Packages

For information about how to compile a design with packages, refer to "Compiling a Design with Packages" in *SystemVerilog in Simulation*.

SystemVerilog introduces a package construct to the Verilog language. A package is a new Verilog design unit containing declarations that can be shared among modules, macromodules, interfaces, programs, or other packages.

In the following simple example, the `global_types` package defines some commonly-used types. The package is imported by the `error_checks` module, and the `boolean` type from the package is used as the type of the `suppress_warnings` variable:

```
package global_types;
  typedef enum logic [1:0] { FALSE, TRUE } boolean;
  typedef enum logic [2:0] { H=1'b1, L=1'b0, Z=1'bz, X=1'bx } logic_state;
endpackage
import global_types::*;
module error_checks;
  ...
  boolean suppress_warnings;
  ...
endmodule
```

A common use of packages is to group a type declaration with a set of tasks or functions that operate on that type. In this scenario, a module can declare objects of the type, and use the package tasks and functions to operate on the object data.

A second common use for packages is to define a utility for common use. This sort of package often uses persistent state and non-reentrancy in its implementation.

The following package example shows a combination of both sorts of use. This example implements a common error reporting utility.

```
package messages;
  typedef [80*8:1] message_type;
  integer error_count = 0;
  integer warning_count = 0;
  integer error_limit = 1;
```

```
  task report_warning;
    input message_type message;
    begin
      $display("Warning at %0t: %0s", $time, message);
      warning_count = warning_count + 1;
    end
  endtask

  task report_error;
    input message_type message;
    begin
      $display("Error at %0t: %0s \n", $time, message);
      error_count = error_count + 1;
      if (error_count == error_limit) end_simulation;
    end
  endtask

  task end_simulation;
    begin
      $display (" !! ERROR LIMIT EXCEEDED !!");
      $display (" Warnings: %d Errors: %d\n", warning_count, error_count);
      $finish;
    end
  endtask

endpackage

import messages::* ;
module testbench;
  ...
  if (bad_condition) report_error("Unexpected ...");
  ...
endmodule
```

A package defines a single, global set of items that can be used by any design unit that imports that package. Unlike modules, packages cannot be used as structural building blocks to create multiple copies. However, a package can build on another package by importing the other package.

The declarations in a package are independent of the structural design hierarchy. Packages do not contain hierarchy, nor do they contain references to global typedefs or items declared in modules and primitives. The declarations within a package cannot contain hierarchical references, unless they refer to items created within the package, or to items made visible by importing another package. Packages cannot reference items defined within compilation unit scopes. However, structural elements can depend on items in a package. For example, a module can connect a global supply in a package to a lower-level component.

In SystemVerilog, you cannot have multiple packages with the same name, even if the packages are compiled into different libraries. The parser generates an error if there is more than one package with the same name.

Declaring a Package

The IEEE 1800 standard describes the package declaration syntax. Not all of the package items specified in this syntax are supported in the current release.

Attribute instances on package declarations and the items within a package are supported.

The list of declarations supported within a package in the current release is as follows:

- net_declaration
- data_declaration
- task_declaration
- function_declaration
- dpi_import_export
- class_declaration
- class_constructor_declaration
- parameter_declaration
- local_parameter_declaration
- timeunits_declaration
- concurrent_assertion_item_declaration

The following declarations, listed in the LRM, are not supported in the current release:

- anonymous_program
- extern_constraint_declaration
- covergroup_declaration
- overload_declaration

Referencing Data in a Package

There are two ways to use the declarations contained in a package:

- Reference a package declaration by using its *package item reference full name*. The syntax is:

  package_identifier::*item_name*

In the following example, the `boolean` type and the `logic_state` type are referenced using their package item reference full name.

```
package global_types;
  typedef enum logic [1:0] { FALSE, TRUE } boolean;
  typedef enum logic [2:0] { H=1'b1, L=1'b0, Z=1'bz, X=1'bx } logic_state;
endpackage

module error_checks;
  ...
  global_types::boolean suppress_warnings = global_types::FALSE;
  global_types::logic_state initial_state = global_types::X;
  ...
endmodule
```

- Use the `import` statement to provide direct visibility of identifiers within a package. The `import` statement allows all or selected identifiers declared in a package to be visible within the current scope. If an identifier declared in a package is imported, you can refer to the item by its simple name, without using a package name qualifier.

Note: You cannot use a Verilog out-of-module reference to refer to an item declared in a package. For example, if variable `var` is declared in a package called `pack`, and the variable is imported into module `top`, you cannot use `top.var` to refer to the package item. You must use the package reference name (`pack::var`) or the simple name (`var`).

Controlling Visibility of Names within Packages: The import Statement

The `import` statement provides control over how and which package items are imported.

The `import` statement can be placed

- Outside a design unit—a package, module, UDP, interface, or program

  The scope of an `import` statement that appears outside a design unit declaration extends to the end of the compilation unit. Such an `import` statement affects all following design units in the compilation unit.

- Inside any declarative scope of a design unit

  The scope of an `import` statement inside a declarative scope extends from where it is declared to the end of the declarative scope.

The `import` statement has two forms—wildcard import and explicit import.

Wildcard import Statement

The syntax for wildcard import is:

```
import package_identifier::*;
```

For example:

```
import IObus_package::*;
```

Example

In the following example, there are three source files:

- `package.v` contains the shared types and declarations.
- `rtl.v` is the RTL code of the design. This file must import the package `globals`, because it uses the types and variables declared in this package.
- `test.v` is the testbench code, which also must import the package. In the source file `test.v`, there is a single `import` statement that applies to both the `stimulus` module and the `testbench` module.

For example:

```
// File: package.v
package globals;
  typedef enum logic { FALSE, TRUE } boolean;
  integer error_count;
endpackage

// File: rtl.v
import globals::* ;
module rtl;
  ...
endmodule

// File: test.v
import globals::* ;
module stimulus;
  ...
endmodule

module testbench;
  ...
endmodule
```

Importing the package with `::*` provides potential direct visibility of any of its contents in the importing scope. In the following example, `gnd` and `vdd`, if referenced inside the importing module by their simple names, will have their declaration from the package made locally directly visible in the importing module. Because `gnd` is not referenced inside the `top` module, its declaration is not imported. Because `vdd` is referenced by `r = vdd` and `globals::`, it is imported. However, the local declaration that follows causes a duplicate symbol error.

```
// File: package.v
package globals;
  integer gnd;
  reg vdd;
endpackage
```

```
module top;
  reg r;

  import globals::*;

  initial
   r = vdd;   // This is globals::vdd; it is imported because there
              // is no local declaration for vdd before this reference.

  reg vdd;   // Declare vdd. Error because there are 2 visible declarations
              // for vdd.

endmodule
```

In the following example, `globals::vdd` is not imported, because `vdd` is locally declared before it is referenced.

```
// File: package.v
package globals;
  int gnd;
  reg vdd;
endpackage

module top;
  reg r;

  import globals::*;
  reg vdd; // Declare vdd.

  initial
    r = vdd;      // This is reg vdd; globals::vdd is not imported because vdd
                  // is locally declared before this reference.
endmodule
```

If the same symbol is imported from two or more different packages with a wildcard import, a direct reference to that symbol in the importing design unit is an error, because the declaration it refers to is ambiguous. However, a package item reference full name can be used to disambiguate the origin of the declaration.

In the following example, the first `initial` block contains an error, because there is a direct reference to `vdd`, which is defined in two packages.

```
package p;
 reg vdd;
endpackage

package q;
 reg vdd;
endpackage

import p::*;
import q::*;
module top;
  reg r;
  reg r1;
  initial
    r = vdd;     // Error because vdd is ambiguously defined
```

```
  initial
    r1 = p::vdd; // vdd from package p

  initial
    r2 = q::vdd; // vdd from package q

endmodule
```

Explicit import Statement

The explicit `import` statement allows precise control of the symbols to be imported. The syntax is as follows:

`import` *`package_identifier`*`::`*`identifier`*`[,`*`package_identifier`*`::`*`identifier ...`*`];`

With an explicit import, only the symbols referenced by the import are made directly visible. All other package items are not directly visible. A package reference full name must be used to refer to the package items not made directly visible by the `import` statement.

In the following example, the `top` module explicitly imports two symbols: `error_count` and `vdd`. They are both directly visible inside the `top` module, even though only `vdd` is used to initialize register `r`. The `ground` declaration cannot be directly referenced by its simple name, because it is not an imported item, and a package item reference full name (`globals::ground`) is used to refer to the `ground` declaration.

```
import globals::error_count, globals::vdd;
module top;
  reg r;
  reg r0 = globals::ground;

  initial
  r = vdd;
endmodule
```

An explicit import is illegal if the imported declaration is already declared in the same scope, or if it is explicitly imported from another package. However, it is legal to import the same declared item from the same package multiple times.

Debugging Packages

For information about how to debug packages using the Tcl command-line interface or the SimVision analysis environment, refer to *SystemVerilog in Simulation*.

Compilation Units

For an example that you can download and run, refer to the “Disabling DPI Tasks and Functions” example in the *SystemVerilog DPI Engineering Notebook*.

SystemVerilog adds a concept called *compilation units*. A compilation unit is a collection of one or more source files compiled together.

SystemVerilog extends Verilog by allowing declarations outside of a module, interface, package, or program. Each compilation unit has a *compilation unit scope*, which contains all of the external declarations made across all files within the compilation unit. Unlike global declarations, which are shared by all of the modules that make up a design, compilation unit scope declarations are visible only to the source files that make up the compilation unit.

By default, all files on a given compilation command line make up a single compilation unit. To create a separate compilation unit for each source file, you must compile each source file separately.

For example:

file_a.v

```
reg r;
module a;
  initial begin
  r= 1'b0;
end
endmodule
```

file_b.v

```
module b;
  initial begin
    r = 1;
  end
endmodule
```

file_c.v:

```
module c;
  assign r = 1'b1;
endmodule
```

Default implementation:

```
% ncvlog -sv file_a.v file_b.v file_c.v
```

Separate compilation units:

```
% ncvlog -sv file_a.v
% ncvlog -sv file_b.v
% ncvlog -sv file_c.v
```

With the default implementation, all three files specified on the command line are grouped under a single compilation unit, which makes `r` accessible to modules `b` and `c`, even if they are not defined in the same file. However, if any of these files are compiled separately, references to `r` in files `file_b.v` and `file_c.v` will become out-of-date.

As defined by the LRM, compilation unit scopes cannot access items within other compilation unit scopes.

Related topics:

- "Disabling DPI Tasks and Functions" example in the *SystemVerilog DPI Engineering Notebook*.
- "Compiling Source Files into Compilation Units" in *SystemVerilog in Simulation*.
- "Viewing Compilation Units in the Design Browser" in *SystemVerilog in Simulation*.
- "Debugging Compilation Units in Tcl" in *SystemVerilog in Simulation*.

Supported External Declarations

SystemVerilog extends Verilog by allowing declarations outside of a module, interface, package, or program. For example:

```
declarations
module;
  ...
endmodule
```

All of these external declarations in a compilation unit make up the *compilation-unit scope*. They can by accessed by any of the constructs defined within the compilation unit.

In the current release, compilation-unit scopes can include the following types of external declarations:

- `bind` directives
- classes
- Structures
- Package import declarations
- `timeprecision` and `timeunit`
- `` `timescale `` and `` `include `` directives
- Task and function declarations
- Variable and net declarations
- Constant declarations
- Parameters

- User-defined types that use `typedef`, `enum`, or `class`
- Specialized classes

Explicitly Referencing External Declarations

In SystemVerilog, you can explicitly reference a declaration within a compilation unit scope using `$unit` and the class scope resolution operator. For example:

```
bit b;
task foo;
  int b;
  b = 5 + $unit::b;
endtask
module
...
endmodule
```

Limitations on Compilation Units

Properties and sequences are not supported within a compilation unit.

Port Declarations

In Verilog-2001, input ports cannot be declared as variables. Output ports can be declared as variables, but must be connected to a wire. SystemVerilog removes these restrictions.

In SystemVerilog a port can be declared as an interface, or a variable or net of any allowed data type. The syntax for this type of declaration is as follows:

```
port_direction port_kind data_type
```

where *port_kind* can be the net type keywords or the `var` keyword, which are used to declare net and variable assignments.

For example, the following declares two ports of the packed structure type `my_type`.

```
typedef struct packed {
  logic b;
  int i;
} my_type;
...
module mysub(input var my_type in, output my_type out);
  always @(in)
  out = in;
endmodule
```

The following table outlines the default rules that are used when keywords are omitted from a port declaration.

Port Direction	Port Kind	Data Type	Default
Unspecified	Specified	Unspecified	In a port list, the port direction is inherited from the previous port. For the first port in a list or a standalone declaration, the port direction defaults to `inout`.
Specified	Unspecified	Unspecified	The port defaults to a net of net type `wire`. You can change the default net type using the Verilog `` `default_nettype `` compiler directive.
`input` `inout`	Unspecified	Either	The port kind defaults to a net of net type `wire`. You can change the default net type using the Verilog `` `default_nettype `` compiler directive.
`output`	Unspecified	Either	The default port kind is based on the port data type. If the data type is not specified, the port kind defaults to a net of the default net type. If the data type is specified, the port kind defaults to variable.

The current release supports the following variable data types on ports:

- `bit`
- `shortint`
- `int`
- `longint`
- `logic`
- `byte`
- `enum`
- `reg`
- `real`
- `string`
- Classes

- Packed structures
- Queues
- Dynamic arrays
- Associative arrays
- Packed arrays
- Unpacked structures
- Unpacked arrays

Note: The current release does not support the event data type on ports.

Declarations of Input and Output Ports

Input Ports with no Port Kind

You can declare input ports of the following data types:

- Unpacked arrays (explicit or implicit data types)
- Unpacked structures
- Type parameters

Ports declared of these data types are treated as if the port kind were `var`.

The `var` keyword is not required for input ports of the following data types because these types are not legal on input `wire` ports.

- Queue
- Associative array
- Dynamic array
- String
- Class
- Mailbox
- Event
- Semaphore

- Process
- Chandle
- Union (unpacked)
- Real

The following example shows declaration of input variable ports (no `var` keyword required):

```
typedef  struct {
  ogic m1;
  int m2;
} us_t;

module  #(type parameter T = logic)  m (
  input logic my_array [1:10],   // variable input array port explicit datatype
  input us_t  my_ustruct,        // variable input unpacked struct port
  input T  my_TP                 // variable input port with type parameter
)
```

The following example shows acceptable syntax for variable input ports of the data types listed above:

```
typedef  union {
  logic m1;
  int m2;
} union_t;

class C;
endclass

module  m (
  input logic my_dynamic_array [ ], // variable input dynamic array port
  input byte my_queue [$],          // variable input queue of bytes port
  input integer  my_aa [string],    // variable input port associative array
                                       of integers indexed by strings
  input string my_str,              // variable input string port
  input real my_real,               // variable input real port
  input mailbox my_mail,            // variable input mailbox port
  input event  my_event,            // variable input event port
  input semaphore my_sem,           // variable input semaphore port
  input process my_process,         // variable input process port
  input chandle  my_ch,             // variable input chandle port
  input string my_str,              // variable input string port
  input union_t  my_un              // variable input union port
)
```

Implicit Array Data Types

In the following example, the port `w1` of module `mid` is an input net array port because it has an implicit array data type. The simulator treats it as a variable input port. The simulation of this example, whether it is a net or a variable, displays:

```
top 00 00
top.m 00 00
```

```
module top;
  reg[1:0] w [1:10];
  assign w[1] = 2'b00;
  mid m (w);
  initial #1 $displayb ("%m:", w[1], w[0]);
endmodule // top

module mid (input [1:0] w1 [1:10]);
  initial #1 $displayb ("%m:",w1[1], top.w[1]);
endmodule // mid
```

In the following example, there are two drivers to the same array element `1` of the port `m.w1`:

- The continuous assignment from `top.w` to the port `w1`
- The continuous assignment in `top.m`:

```
assign w1[1] = 2'b01;
```

The net `w1` is resolved to the value `0x` and the simulation displays:

```
top: 0x xx
top.m : 0x 0x
```

However, because the variable semantics are applied, the simulator displays the value assigned from `top` to `mid`.

```
module top;
  reg [1:0] w [1:10];
  assign w[1] = 2'b00;
  mid m (w);
  initial #1 $displayb ("%m:", w[1], w[0]);
endmodule // top

module mid (input [1:0] w1 [1:10]);
  assign w1[1] = 2'b01;
  initial #1 $displayb ("%m:",w1[1], top.w[1]);
endmodule // mid

ncsim> run
top.m 0000
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

If the input port `w1` was an output variable port, the simulator would display the value assigned from `mid` to `top`:

```
module top;
  reg [1:0] w [1:10];
  assign w[1] = 2'b00;
  mid m (w);
  initial #1 $displayb ("%m:", w[1], w[0]);
endmodule // top

module mid (output reg [1:0] w1 [1:10]);
  assign w1[1] = 2'b01;
  initial #1 $displayb ("%m:",w1[1], top.w[1]);
endmodule // mid
```

```
ncsim> run
top.m 0101
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

Input Ports with Explicit var Keyword

These ports are supported with no extensions.

Input Port with Explicit wire Keyword

Wire ports with data types of unpacked arrays (implicit or explicit array data types) and unpacked structures are not supported. All other data types listed above are illegal for input `wire` ports.

Output Ports with no Port Kind

Same extensions as described in Input Ports with no Port Kind on page 256 and in Implicit Array Data Types on page 257.

Output Ports with var Kind

These ports are supported with no extensions.

Output Ports with wire Kind

These ports have `wire` semantics with no extensions.

Inout Ports

These ports are wire ports.

Reference Ports

These ports are variable ports and are not supported.

Port connections

The connections for ports of data types listed in Input Ports with no Port Kind on page 256 are limited to whole variable connections. In particular, port assignment compatible expressions, such as assignment patterns expressions, or array slices, or array indexed expression of an object, are not supported. See the LRM specification of port connection rules in LRM Port Connection Rules for Variable Ports on page 260 and LRM Port Connection Rules for Net Ports on page 261.

In general, variable in a parent module connected to a variable input port in the child instance is supported. However, wire in a parent module connected to a variable input port in a child instance is not supported.

Variable in a parent module connected to a variable output port in the child instance is supported. However, wire in a parent module connected to a variable output port is not supported.

LRM Port Connection Rules for Variable Ports

Both sides of a port connection must have assignment compatible data types. If a port declaration has a variable data type, then its direction controls how it can be connected when instantiated, as follows:

- An input port can be connected to any expression of a compatible data type. A continuous assignment shall be implied when a variable is connected to an input port declaration. Assignments to variables declared as an input port shall be illegal. If left unconnected, the port shall have the default initial value corresponding to the data type.
- An output port can be connected to a variable (or a concatenation) of a compatible data type. A continuous assignment shall be implied when a variable is connected the output port of an instance. Procedural or continuous assignments to a variable connected to the output port of an instance shall be illegal.
- An output port can be connected to a net (or a concatenation) of a compatible data type. In this case, multiple drivers shall be permitted on the net as in Verilog.
- A variable data type is not permitted on either side of an inout port.
- A `ref` port shall be connected to an equivalent variable data type. References to the port variable shall be treated as hierarchal references to the variable it is connected to in its instantiation. This kind of port cannot be left unconnected.

LRM Port Connection Rules for Net Ports

If a port declaration has a net type, such as `wire`, then its direction controls how it can be connected, as follows:

- An input port can be connected to any expression of a compatible data type. If left unconnected, it shall have the value `'z`.
- An output port can be connected to a net or variable (or a concatenation of nets or variables) of a compatible data type.
- An inout port can be connected to a net (or a concatenation of nets) of a compatible data type or left unconnected, but cannot be connected to a variable.

18

Interfaces

For a complete example using interfaces that you can download and run, refer to the *SystemVerilog Engineering Notebook*.

One of the major extensions to the Verilog language proposed in the SystemVerilog LRM is the *interface* construct. This construct was created to encapsulate the communication between blocks of a digital system.

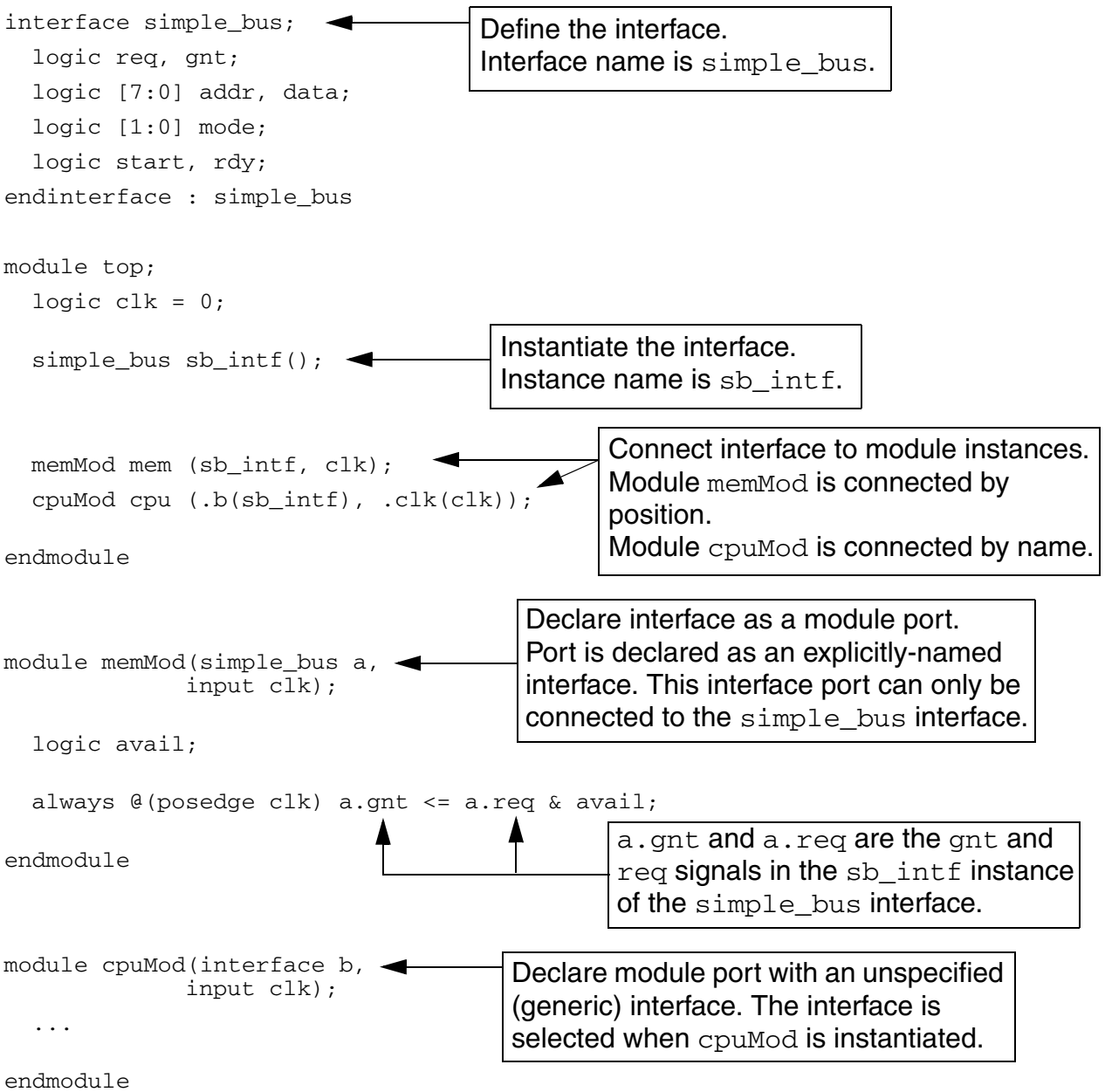
At its lowest level, a SystemVerilog interface is a named bundle of nets or variables that encapsulates the connectivity between blocks. By declaring an interface, you can define a group of signals once in one modeling block. The interface can then be instantiated in the design and accessed through a module port as a single item. This feature eliminates redundant declarations of the same signals in multiple modules, which can significantly reduce the size of a description. Grouping signals together in one place also improves design maintainability. For example, if a change to the port specification is required, the change can be made in one place instead of in multiple modules.

At a higher level, an interface can encapsulate functionality in addition to connectivity. An interface can contain data type declarations, tasks and functions, `initial` and `always` blocks, continuous assignments, and so on, so you can define communication protocols, protocol checking routines, and other verification routines in one place.

This section provides details on the functionality provided in the current release for SystemVerilog interfaces.

Example 18-1 on page 264 shows a sample design with a simple interface that bundles a collection of signals. This example, modified from an example shown in the SystemVerilog LRM, shows the basic syntax for defining, instantiating, and connecting an interface.

Example 18-1 Simple Interface Example



Declaring an Interface

The syntax for an interface declaration is as follows:

```
interface interface_identifier [(port_list)];
  interface_items
endinterface [: interface_identifier]
```

Any construct that you can use in a module can be used in an interface, with the following exceptions. The IEEE 1800 standard does not allow the following constructs in an interface, and these constructs are not supported in the simulator:

- `defparam` statements
- `specify` blocks
- `alias` statements
- Instances of program blocks, modules, and primitives

Example 18-2 Nested Interface Example

```
interface ITest;
   int a;
   int b;
   always @(a) $display( $time, ":  %m.a = %0d", a );
   always @(b) $display( $time, ":  %m.b = %0d", b );
endinterface

interface ITop;
   ITest   TestIf1();
   ITest   TestIf2();
endinterface

module SubMod(ITest If_test, input int in1);
   initial begin
    #210  If_test.a = (1000 * in1) + 11;
    #10  If_test.b = (1000 * in1) + 22;
   end
endmodule

module TestEnv(ITop If_port);
 SubMod sm1(If_port.TestIf1,1);
 SubMod sm2(If_port.TestIf2,2);
   initial begin
    #110  If_port.TestIf1.a = 103;
    #10  If_port.TestIf2.a = 203;
    #10  If_port.TestIf1.b = 105;
    #10  If_port.TestIf2.b = 205;
   end
endmodule

module test_top();
   ITop main_if();
   TestEnv env(main_if);
   initial begin
     #10  main_if.TestIf1.a = 107;
     #10  main_if.TestIf2.a = 207;
     #10  main_if.TestIf1.b = 109;
     #10  main_if.TestIf2.b = 209;
   end
endmodule
```

Inner interface

Outer interface

Connects the inner interface to the module instance.

Connects outer interface to the module instance.

Connects child interface instances relative to the interface port.
In the current release, you cannot use OOMRs, and you can use only two dot components: the first must be the interface port, and the second must be the child interface.

Connects the outer interface "`main_if`".

When simulated, this example produces the following results:

```
ncsim> run
  10:  test_top.main_if.TestIf1.a = 107
  20:  test_top.main_if.TestIf2.a = 207
  30:  test_top.main_if.TestIf1.b = 109
  ...
  220:  test_top.main_if.TestIf2.b = 2022
ncsim: *W,RNQUIE: Simulation is complete.
```

Creating Design Units

Although interfaces are very similar to modules, the LRM definition classifies them as a different type of Verilog design unit. The NC library system has been enhanced to manage the new design unit type.

If you use the `-messages` option when you compile your source files, the output displays `interface`. For example,

```
% ncvlog -nocopyright -messages -sv source.v
file: source.v
        interface worklib.simple_bus
                errors: 0, warnings: 0
...
...
```

The default view name for an interface is `interface`. For example,

```
worklib.simple_bus:interface
```

The *ncls* utility includes a `-interface` command-line option so that you can query the library system for compiled interfaces. For example:

```
% ncls -nocopyright -interface
        interface worklib.simple_bus:interface (VST)
        interface worklib.simple_bus:interface (SIG) <0x7a6d9d7a>
```

Using the Interface as a Module Port

You can declare interface ports on modules in the following ways:

- As a specific type of interface, by using the following syntax:

  ```
  module module_name (interface_name port_name, other_module_ports);
  ```

  In the example shown in Example 18-1 on page 264, the `memMod` module contains an explicitly-named interface port.

  ```
  module memMod(simple_bus a, input clk);
  ```

  This port can be connected only to the interface named `simple_bus`.

  You can also use non-ANSI style declarations. For example:

  ```
  module memMod(a, clk);
    simple_bus a;
    input clk;
    ...
  ```

- As a generic interface port, by using the following syntax:

  ```
  module module_name (interface port_name, other_module_ports);
  ```

In the example shown in Example 18-1 on page 264, the `cpuMod` module contains a generic interface port.

```
module cpuMod(interface b, input clk);
```

This port can be connected to any interface when the module is instantiated.

The LRM states that a generic interface reference "can only be declared by using the list of port declaration style of reference. It shall be illegal to declare such a generic interface reference using the old Verilog-1995 list of port style." The simulator allows these non-ANSI style declarations. For example:

```
module cpuMod(b, clk);
  interface b;
  input clk;
  ...
```

If you use non-ANSI style generic declarations, a warning message is issued saying that this is not official syntax according to the current LRM.

- You can declare interface array ports, where a single dimension array limit is specified after the port name:

  ```
  module module_name (interface port_name array_dimension);
  ```

  In the following example, the `Dut` module connects to an explicitly named interface port with a dimension width of 4. This port can be connected only to the interface named `Ifc`:

  ```
  module Dut (Ifc prta [4:1]);
  ...
  ```

  For more information, refer to "Interface Array Ports" on page 269

Note: A port that is declared as an interface must be connected to an interface instance. An error is generated if an interface port is left unconnected.

Limitations on Interfaces

The following interface features are allowed in the IEEE 1800 standard, but are not supported in the current release of the simulator:

- Interfaces connected to ports of the interface
- Gates and UDPs
- Wires and ports are supported, but the wires and ports declared in an interface must be digital wires. Analog wires within interfaces are not supported.

- Instances of interfaces

  In the current release, simple interface instances can be nested within another interface. For example:

```
interface Inner;
  int a;
endinterface

interface Outer;
  Inner inner1();   // Supported
endinterface
```

  However, arrays of instances are not supported within interfaces, and will generate a parser error. For example:

```
interface Outer;
  Inner inner[1:4]();   // Not supported
endinterface
```

```
ncvlog: *E,INFINS (test.v,13|19): Instances of modules, interface arrays,
program blocks, and primitives are not allowed within interface or program
block definitions.'[SystemVerilog]'.
```

- An interface is instantiated in the same way that modules and primitives are instantiated. Interfaces must be instantiated at the highest point in the hierarchy where they can be used. In the current release, interface instances are limited to single instances. Interface arrays and interfaces in for-generates or if-generates are not supported.

Interface Array Ports

The current release supports interface array port connections, where you place one-dimensional array limits after the port name. For example:

```
module Dut ( Ifc prta [7:1] );
...
endmodule
```

For a complete example using interfaces that you can download and run, refer to the *SystemVerilog Engineering Notebook*.

Supported Uses for Interface Array Ports

In the current release, you can

- Assign a bit select of an interface array port to a virtual interface variable:

```
virtual interface Ifc vi = prta[4];
```

- Call a task or function through a bit select of an interface array port:

```
initial prta[5].mytask();
```

- Access an interface variable using a bit select of an interface array port:

```
initial prta[6].a = 5;
```

- Pass an interface array port to a submodule:

```
Submod sm1 ( smprta(prta) );
```

Note: All bit selects must be constant expressions. The constant expression index cannot have an X or Z value, and must lie within the defined array limits.

Using Arrays of Interfaces in Interface Array Ports

In the current release, support for interface array port connections follows the Verilog 2001 array of instances port connection rules. For example, the following declares an array of interfaces, and passes it to an instance of module `Dut`, which connects the array of interfaces to the interface array port called `prta`:

```
interface Ifc();
...
endinterface

module test_top();
  Ifc ifca [4:1] (); // Array of interfaces instantiation
  Dut dut1(.prta(ifca)); // Module instance
  ...
endmodule

module Dut (Ifc prta [4:1]); // Interface array port
...
endmodule
```

According to the LRM, the array widths must match in interface array port connections. For example, the following is illegal, because the widths of the array of interfaces and the module instance do not match:

```
module top();
  Ifc ifca [3:1] (); // Array of interfaces instantiation
  Dut duta [4:1]( .prt(ifca) ); // Module instance. Width is not the same.
endmodule

module Dut ( Ifc prt );
..
```

The following causes an error, because the array of interfaces and the interface array port have different widths:

```
module top();
  Ifc ifca [4:1] (); // Array of interfaces instantiation
  Dut dut1 ( .prta(ifca) );
endmodule

module Dut ( Ifc prta [5:6] ); // Interface array port. Widths do not match.
...
```

The following also causes an error, because the array of interfaces width is not equal to the module instance width times the interface array port width:

```
module top();
  Ifc ifca [22:1] (); // Should be 21:1
  Dut duta [3:1] ( .prta(ifca) );
endmodule : top

module Dut ( Ifc prta [7:1] );
...
endmodule
```

You can also pass a bit select of an array of interfaces to the module instance. For example, the following passes a bit select of array `ifca` to module instance `dut1`, connecting it to the interface array port `prta`:

```
module top();
  Ifc ifca [4:1] (); // Array of interfaces instantiation
  Dut dut1(.prta(ifca[2]); // Module instance.

module Dut (Ifc prta); // Scalar interface port
...
endmodule
```

However, according to the LRM, you cannot pass a bit select interface array to an interface array port:

```
module top();
  Ifc ifca [4:1];
  Dut dut1(.prta(ifca[4]);

module Dut (Ifc prta [5:6]); // Illegal
...
endmodule
```

The current release also supports implied bit selects of an array of interfaces to module instances:

```
module top();
Ifc ifca [2:4] ();
Dut duta [9:7] (.prt(ifca)); // Connects ifca to interface port of every child
endmodule                    // array using an implied bit select.

module Dut (Ifc prt);
...
endmodule
```

Arrays of interfaces can also connect through implied part selects:

```
module top();
  Ifc ifca [6:1] ();
  Dut duta [3:1] (.prta(ifca)); // Connects ifca to interface port of every
endmodule                     // child array using an implied part select.

module Dut (Ifc prta [2:1]);
  ...
endmodule
```

According to the LRM, you cannot pass an interface instance to an interface array port:

```
module top();
  Ifc ifc1();
  Dut dut1 ( .prt(ifc1) ); // Illegal. Passes interface instance ifc1
endmodule : top

module Dut ( Ifc prt [3:1] );
...
```

Limitations on Interface Array Ports

The following summarizes the interface array port features in the LRM that are not supported in the current release.

- Named interface array ports with modports:

  ```
  module Dut (Ifc smprta[6:1].Master);
  ```

- Generic interface array ports with modports:

  ```
  module Dut (interface ifcprta[6:1].Master);
  ```

- Whole array assignments to virtual interface array variables:

  ```
  module top();
   Ifc ifca [3:1] ();
   Dut dut1 ( .prta(ifca) );
  endmodule

  module Dut ( Ifc prta [3:1] );
  virtual interface Ifc vidut[3:1] = prta; // Not supported.
  endmodule
  ```

  To work around this, the virtual interface array must be loaded using bit selects:

  ```
  module Dut ( Ifc prta [3:1] );
    virtual interface Ifc vidut[3:1];
    initial begin
     vidut[3] = prta[3];
     vidut[2] = prta[2];
     vidut[1] = prta[1];
    end
  endmodule
  ```

Referencing an Interface

You cannot reference an interface using a hierarchical path. For example, the following is not allowed:

```
module top;
  ...
  memMod mem(some_other_top.the_memory_interface);
  ...
endmodule
```

You can reference objects declared in an interface from any module that declares the interface by using an interface reference. The syntax is as follows:

```
port_name.interface_signal_name
```

In the example shown in Example 18-1 on page 264, the `memMod` module has an interface port with the port name `a`.

```
module memMod(simple_bus a, input clk);

  logic avail;

  always @(posedge clk) a.gnt <= a.req & avail;

endmodule
```

Within the `memMod` module, the `gnt` and `req` signals in the interface are referenced as `a.gnt` and `a.req`, respectively.

Working with Modports

While an interface provides a way to define a group of nets or variables in one place to encapsulate the connectivity between blocks, different modules connected to the interface might require different views of the interface. For example, a particular signal can be an input for one module, while the same signal can be an output for another module.

SystemVerilog provides the `modport` construct, which allows you to customize the interface for the different modules that are connected to it. Using this construct, you can

- Provide direction information for module ports
- Specify which signals defined in the interface are accessible to a module

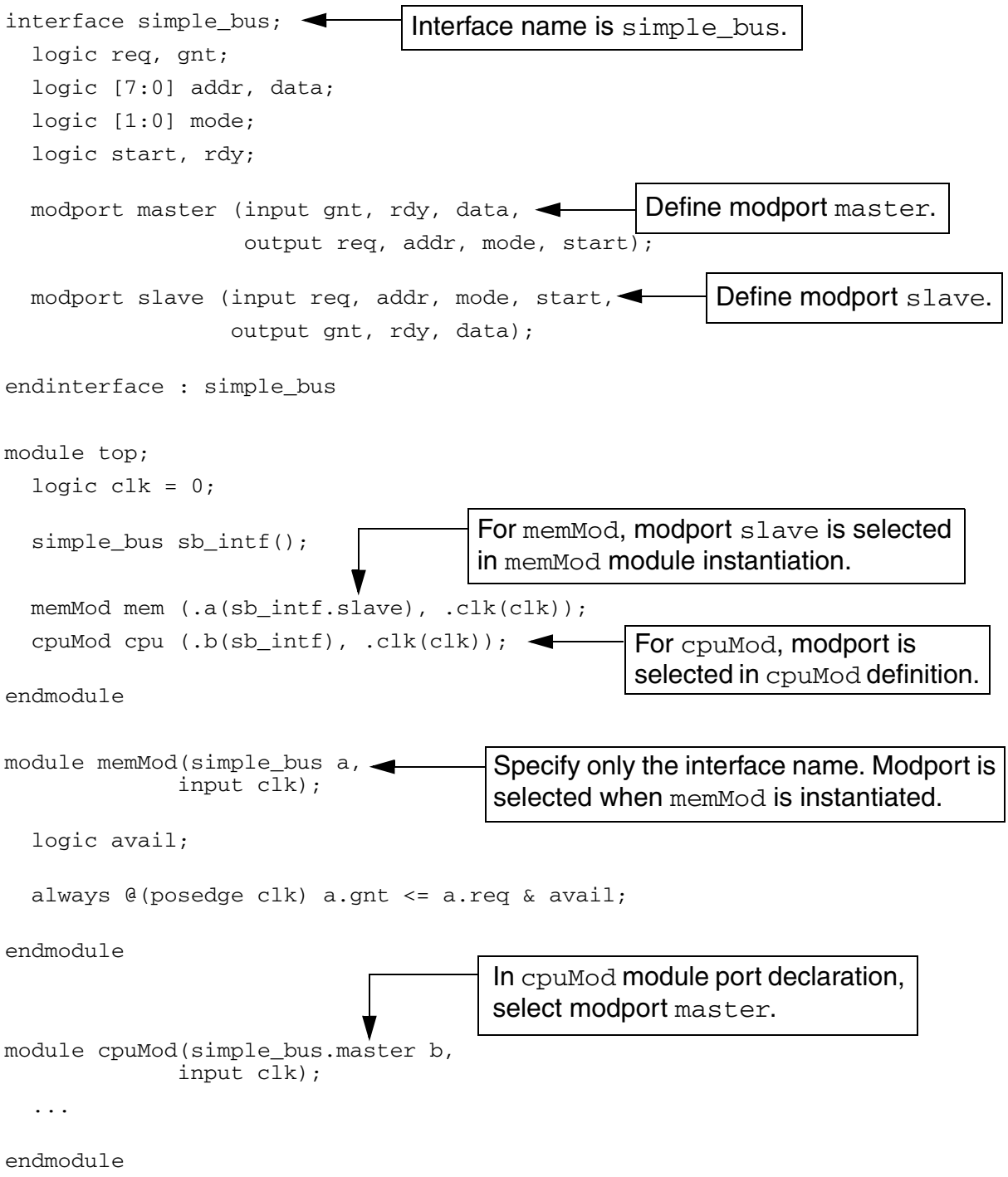
The following example, modified from examples shown in the SystemVerilog LRM, includes an interface with two modports named `master` and `slave`. This example shows

- The basic syntax for defining a modport
- The two ways of selecting which modport a module is to use

  The `slave` modport is selected for the `memMod` module by specifying the modport when the module is instantiated in the `top` module.

  The `master` modport is selected for the `cpuMod` module by specifying the modport in the module port declaration of the `cpuMod` module.

Example 18-3 Interface Example with Modports



Defining a Modport

Modports are defined within an interface by using the `modport` keyword. You can define any number of modports in an interface.

In the current release, you can define a modport that specifies the port direction for ports—`input`, `output`, or `inout`. For example:

```
interface myintf;
  wire a, b, c, d;

  modport master (input a,
                  input b,
                  output c,
                  output d);

  modport slave (output a,
                 output b,
                 input c,
                 input d);
endinterface
```

Modport ports can refer only to nets or variables. Vector sizes or data types are not included in the modport definitions. Only the port direction is specified. All modport ports must have a corresponding net or variable declaration within the interface in which they are declared.

As with module port declarations, you can specify multiple ports with one direction keyword. For example:

```
modport master (input a, b, output c, d);
```

This syntax lets you define multiple modports with one keyword, as shown in the following example:

```
modport master (input a, b, output c, d),
        slave  (output a, b, input c,d);
```

Selecting Which Modport to Use

You can specify which modport a module interface port must use in two places. You can specify he modport name

- In the module header, as part of the module port declaration.

  You can specify an explicitly-named interface and modport using the following syntax:

  ```
  module module_name (interface_name.modport_name port_name,
                      other_ports);
  ```

  In this case, the interface name selects the interface, and the modport name selects the modport.

You can also specify a generic interface and modport using the following syntax:

```
module module_name (interface.modport_name port_name, other_ports);
```

In the example shown in Example 18-3 on page 274, the modport for the `cpuMod` module is specified in the port declaration for the module, as follows:

```
module cpuMod(simple_bus.master b, input clk);
```

The instance of the `cpuMod` module in the `top` module does not specify the name of the modport. It just connects the module port to the instance of the interface.

```
module top;
  ...
  cpuMod cpu (.b(sb_intf), .clk(clk));
  ...
```

- In the port connection with the module instance, using the following syntax:

```
module_name instance_name (interface_instance_name.modport_name,
                           other_ports);
```

The module definition can use either an explicitly-named interface port, or a generic interface port. For example, in Example 18-3 on page 274, the definition of the `memMod` module uses an explicitly-named interface port.

```
module memMod(simple_bus a, input clk);
```

The modport for the `memMod` module is specified in the instantiation statement for the module, as follows:

```
memMod mem (.a(sb_intf.slave), .clk(clk));
```

Note: You can specify which modport to use in both places. If you do, the modport identifier must be the same. A warning is generated if you specify a modport in one place, and specify a different modport in the other place. The warning tells you that the modport specified in the module header as part of the module port declaration is being used instead of the modport specified on the module instance.

Limitations on modports

The following are allowed in the IEEE 1800 standard, but are not supported for modports in the current release of the simulator:

- Modports inside `for` loops
- Expressions within modports

Declaring Tasks and Functions in Interfaces

You can declare tasks and functions in an interface by using the same syntax and the same statements that you used to define tasks and functions in a module. SystemVerilog refers to tasks and functions defined in an interface as *interface methods*.

In the current release, you can define interface methods, then call the methods from modules connected to the interface using an interface reference of the form:

```
interface_port_name.task_function_name(arguments);
```

For example:

```
interface myintf;
  logic start;
  other_interface_signals
  ...

  task mytask;
    ...
  endtask : mytask
endinterface : myintf

module mymod(myintf a);
  ...
  always @(a.start)
     a.mytask;
  ...
endmodule
```

Limitations on Tasks and Functions in Interfaces

The following SystemVerilog enhancements related to tasks and functions in interfaces are not supported in the current release:

- Defining a task or function in one module, then exporting the task or function through an interface modport to other modules. The `export` construct in a modport is not supported.
- Defining a task or function in one module, then exporting the task or function to an interface without using a modport. The `extern` construct is not supported.
- Exporting a task name from multiple modules into the same interface. The `extern forkjoin` construct is not supported.

Virtual Interfaces

In SystemVerilog, you can declare a *virtual interface*, which is a variable that represents an interface instance. Virtual interfaces are meant to separate code that operates on an interface

from the actual code itself. That way, instead of directly manipulating the set of signals within an interface, you are manipulating a virtual set of signals.

Syntax and Usage

The syntax for a virtual interface is as follows:

```
virtual interface virtualinterface_identifier
```

For example:

```
interface Sbus;
  int a;
endinterface


module test_top();
  Sbus sbif1(); // Interface instances
  Sbus sbif2();
    class c;
     virtual interface Sbus sbus = null;  // Virtual interface of type Sbus
                                          // initialized to null.
     virtual interface Sbus vbus = sbif2; // Virtual interface of type Sbus
                                          // initialized to an interface
                                          // instance of the same type.

     function new(virtual interface Sbus channel); // Function argument
       sbus = channel;
       $display("My interface");
     endfunction
    endclass

   initial begin
    c myclass;
    myclass = new(sbif1);
   end

endmodule
```

Note: The IEEE 1800 standard does not mention that, when declaring a virtual interface, you must include any specializations for the interface. For example, if there are any parameter values specified on the interface instance, they must be included in the virtual interface declaration.

You can use a virtual interface in the same context as a variable.

Virtual Interface Support

The current release supports the following:

- Virtual interfaces that are declared within a class, module, program block, task, package, compilation unit, or function

For example:

```
interface Ifc();
int a;
always @(a) $display( $time, ":  %m.a = %0d", a ); endinterface

// Virtual interface in compilation unit scope
virtual interface Ifc cuvi;

// Virtual interface in package
package p;
virtual interface Ifc pkvi;
endpackage

module test_top();
import p::pkvi;
Ifc ifc1();
Ifc ifc2();
initial begin
   #1 pkvi = ifc1;
   #1 cuvi = ifc2;
   #1 pkvi.a = 111;
   #1 cuvi.a = 222;
end
endmodule
```

- Passing virtual interfaces by value or reference to a task or function
- Comparing or assigning virtual interfaces to other virtual interfaces, null, or interface instances
- Dereferencing scalar variables, wires, packed arrays, and unpacked arrays
- Arrays of virtual interfaces
- Virtual interfaces that call tasks and functions within an interface

For example:

```
interface bus;
   logic a, b;
   function void msg();
    $display("Can call this function.");
   endfunction
endinterface

module test_top();
...
class test_stim;
   virtual interface bus vi = null;
   function new(virtual bus vbus);
    vi = vbus;
   endfunction
   task run(logic stim);
    ...
    vi.msg(); // Calls function from interface bus
   endtask
endclass
```

```
...
endmodule
```

- Nested interface instances

  The current release has limited virtual interface support for nested interface instances. To next interface instances, you must:

  a. Define the inner and outer interfaces.

  b. Instantiate the outer interface and its inner children.

  c. Declare the virtual interface variables for the outer and inner interfaces.

  d. Initialize the outer variable to point to the desired outer interface instance.

  e. Using a virtual interface select from the outer virtual variable, access the desired interface instance. Then, write this to the inner virtual interface variable.

  This technique enables access to the inner variables.

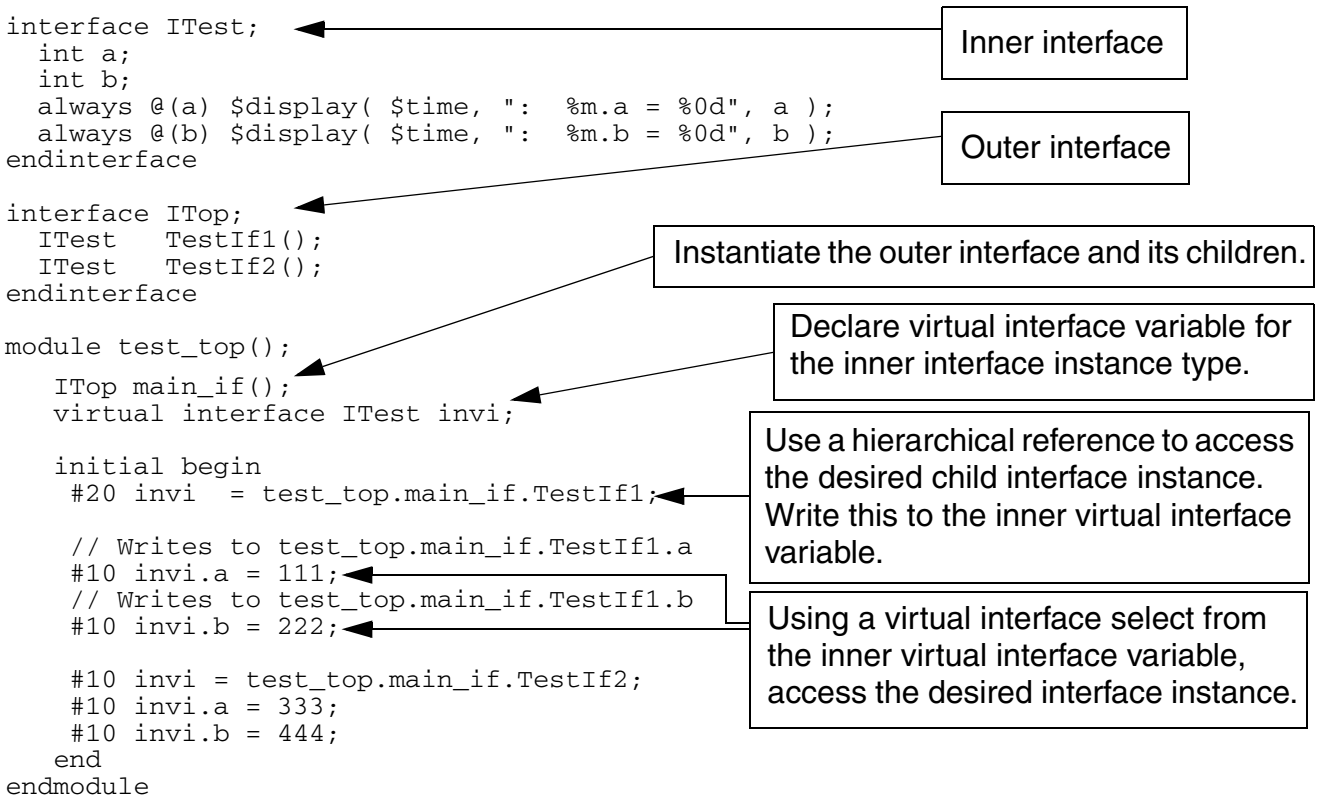
  Note: You can also use a hierarchical reference to access the desired interface instance.

  f. Using a virtual interface select from the inner virtual interface variable, access the desired interface instance.

  g. If necessary, repeat steps e and f.

  Refer to Example 18-4 on page 281.

Example 18-4 Support for Nested Interface Instances



When simulated, this example produces the following results:

```
ncsim> run
 30:  test_top.main_if.TestIf1.a = 111
 40:  test_top.main_if.TestIf1.b = 222
 60:  test_top.main_if.TestIf2.a = 333
 70:  test_top.main_if.TestIf2.b = 444
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

However, in the current release:

- You cannot use a virtual interface select to write to a child instance. For example:

```
#10 outvi.TestIf2 = outvi.TestIf1; // Invalid
```

- Virtual interfaces cannot directly reference child interface items. For example:

```
#10 outvi.TestIf1.a = 555; // Invalid
```

Limitations on Virtual Interfaces

The following summarizes the virtual interface features in the LRM that are not supported in the current release.

- Virtual interface variable expressions cannot be used as `generate` items inside a `forgen` loop.
- Out-of-module references to virtual interfaces are not supported.
- Virtual interfaces cannot reference queues, dynamic arrays, associative arrays, strings, arrays of classes, or semaphores within an interface.

Working with Interfaces and Timing

The IEEE 1800 standard states that, when a module is connected to an interface, signals declared in the interface can be used as terminal descriptors in module path delays, and in timing checks described in a `specify` block.

In the current release, you cannot reference a signal in an interface as the terminal node within a `specify` block or in SDF annotation, including module path delays, interconnect delays, and timing checks.

19

System Functions

Out-of-Module Reference ($root)

In Verilog, an out-of-module reference (OOMR) can be ambiguous. For example, if you have a `parent` instance that contains a `child` instance at both the top level and in the current module, the hierarchical path `parent.child.var` can mean the top-level `parent.child.var` or the local `parent.child.var`.

In Verilog-2001, this ambiguity is resolved by giving priority to the local scope. An OOMR is resolved by first using a search relative to the enclosing scope. If the object is found using this relative search, no other search is performed, and access to the top-level path is prevented.

SystemVerilog solves this problem by introducing the `$root` qualifier, which forces an OOMR reference to be absolute. The `$root` qualifier lets you refer explicitly to a top-level instance, or to an instance path starting from the root of the instantiation tree.

For example:

```
$root.parent.child.var   // Item var within instance child within
                          // top-level instance parent
```

Expression Size System Function ($bits)

SystemVerilog adds a `$bits` system function that returns the number of bits represented by an expression. The syntax is as follows:

```
$bits(expression);
```

Examples:

```
int bitSize;
reg [31:0] x;
reg [7:0] y [0:31];
reg [3:0] z [7:0] [0:15];
...
...
bitSize = $bits(x);          // bitSize returns 32
bitSize = $bits(y[3]);       // bitSize returns 8
bitSize = $bits(z[3][4]);    // bitSize returns 4
```

When used with fixed-size types, the `$bits` system function can be used as an elaboration-time constant. For example:

```
reg [3:0] a;
reg [$bits(a)-1:0] b;
```

When used with dynamic arrays during simulation, the `$bits()` system function returns the size of the whole dynamic array in bits, which is allocated at that simulation time. For example:

```
logic c[];

initial begin
  $display("$bits(c)=%4d", $bits(c)); // Displays "$bits(c)=0"
  c = new[10];
  $display("$bits(c)=%4d", $bits(c)); // Displays "$bits(c)=10"
  c = new[20];
  $display("$bits(c)=%4d", $bits(c)); // Displays "$bits(c)=20"
end
```

Limitations on the $bits System Function

The following lists the current limitations for the `$bits` system function:

- The `$bits` system function cannot be used in constant expressions that involve dynamic arrays. For example:

  ```
  logic c[];
  reg [$bits(c)-1:0] d; // Invalid
  ```

- Declarations that involve `$bits` cannot be evaluated if there is a circular dependency between the declarations. Although the LRM does not explicitly state this limitation, the Cadence implementation does not support the following instances that can cause circular dependencies:

  - Forward references in constant expressions are not supported. For example:

    ```
    reg [$bits(b)-1:0] a; // Invalid
    reg [$bits(a)-1:0] b;
    ```

  - OOMRs within `$bits` constant expressions. For example:

    ```
    reg [$bits(top.cpu1.regfile2.u1.ctl)-1:0] ctl; // Invalid
    ```

- You cannot use the `$bits` function on class `struct` variables.

$sformatf and $psprintf

The `$sformatf` and `$psprintf` system functions are used to pass formatted strings as arguments to objects of SystemVerilog data types. These two functions provide similar functionality, except that the string result of `$psprintf()` is passed back to the user as the function return value, not placed in the first argument as for `$sformat()`. Thus, `$psprintf` is a function that returns a string. This function can be used with the message macros to display messages with run-time formatted content. `$psprintf()` cannot be overridden by a user-defined system function in PLI.

Although the rest of this section refers to `$sformatf`, it implies `$psprintf` as well.

The `$sformatf` system function is defined as follows:

```
$sformatf(format_string[, list_of_arguments])
```

The `$sformatf` system function interprets the first argument, *`format_string`*, as a format string and processes the remaining arguments as format specifiers for the *`format_string`*. The resulting formatted string is passed as the result of the `$sformatf` function. This behavior is different from `$sformat`, where the formatted string is passed back as the first argument for `$sformat`.

For example:

```
string s;
byte r1 = 1100001;
byte r2 = 1100010;
int r3 = 99;
string format_str ="%c%c%d";
str = $sformatf (format_str, r1, r2, r3);
```

Here, the values of `r1`, `r2`, and `r3` are formatted according to the `format_str` string, and the resulting string is assigned to the `str` string. Using `$sformat`, the result looks like the following:

```
$sformat(str, format_str, r1, r2, r3);
```

In this case, the resulting string is assigned to the first argument string, `str`.

Note: *ncelab* issues a warning if the number of actual arguments is not the same as the number of format specifiers.

The following summarizes how the simulator handles cases where there are too many or not enough arguments listed for the given format string.

- When a format string is not specified, the first argument is interpreted as a string `%s` if it is of type literal, expression of integral, or a string data type. Otherwise, *ncelab* issues an error message, and a snapshot is not created.

- When the number of actual arguments is not the same as the number of format specifiers, *ncsim* issues a warning, and uses the default format for the extra arguments.
- When there are more format specifiers than actual arguments, *ncsim* issues a warning, ignores the extra format specifiers, and proceeds.

Limitations on $sformatf

In the current release, the following are not supported:

- The `%p` format specifier is supported in `$sformatf` for the following data types: scalars (`int`, `real`, `string`), queues, danymic arrays, static arrays.
- Unpacked arrays of bytes are not supported as arguments to `$sformatf`.

Sampled Value Functions in Procedural Blocks

According to the SystemVerilog IEEE 1800 standard, sampled value functions can be used in assertions and procedural blocks. This section describes support for sampled value functions in procedural blocks.

The sampled value functions access the sampled value of an expression at the current clock cycle, or at a specific number of cycles in the past.

In the current release, the `$fell`, `$past`, `$rose`, `$sampled` and `$stable` functions are supported only in `always` blocks. These functions are not supported within other procedural blocks and with optional gating expressions and clocking events.

Refer to the “Writing SystemVerilog Assertions” chapter of the *Assertion Writing Guide* for information about sampled value function support for SystemVerilog Assertions.

$rose and $fell Sampled Value Functions

```
$rose (expression])
$fell (expression)
```

`$rose` returns `true` if the LSB of the expression changed to `1`; otherwise, it returns `false`.

`$fell` returns `true` if the LSB of the expression changed to `0`, otherwise, it returns `false`.

When these functions are called at or before the first clock tick of the clocking event, the result is computed by comparing the current sampled value of the expression to `X`.

The following example illustrates the use of `$rose` in an `always` block with inferred clock:

```
always @(posedge clk) begin
  if($rose(b))
    reg1 <= a;
end
```

In this example, `posedge clk` is used as the clocking event for `$rose`. Sampling of the input expression `b` is scheduled in the preponed region of the time slot of `posedge clk` event. Execution of `$rose` is scheduled in active region, while variable `reg1` is assigned in NBA region.

The following example illustrates the use of `$fell` in an `always` block sensitive to the default clock:

```
always @(*) begin
  if($fell(b))
    reg1 <= a;
end
```

In this example, because event control expression is not in the form of `posedge clk` or `negedge clk`, the default clock is used for sampling of the input expression `b` for `$fell`.

$past Sampled Value Function

```
$past (expression [, number_of_ticks])
```

`$past` returns the sampled value of an expression at a previous time. For example:

```
always @ (posedge clk)
 reg_a <= $past(b)
```

In this example, the clocking event `posedge clk` is applied to `$past`. The `$past` function is evaluated in the current occurrence of `posedge clk`, and returns the value of `b` sampled at the previous occurrence of `posedge clk`.

$sampled Sampled Value Function

```
$sampled (expression)
```

`$sampled` returns the value of the expression sampled in the preponed region of the simulation time step in which the function is called. The value is stable throughout the simulation step. Clocking event is not required for this function. The following example illustrates the use of `$sampled` in an `always` block:

```
always @(clk)
  reg1 = $sampled(b);
```

In this example, `b` is sampled in the preponed region every time `clk` changes.

$stable Sampled Value Function

```
$stable (expression)
```

`$stable` returns `true` if the value of an expression did not change during the current clock cycle; otherwise it returns `false`. For example:

```
always @ (posedge clk)
 reg_a = $stable(a) ? b : a;
```

In this example, the clocking event `posedge clk` is applied to `$stable`. Expression `a` is sampled in the preponed region of the time slot of the clocking event `posedge clk`.

Arguments to Sampled Value Functions

- *`expression`* specifies the object whose value you want to obtain.

  In the current release, *`expression`* cannot contain dynamic objects, arrays, unpacked structures, and unpacked unions. Also, *`expression`* cannot contain variables or index variables in bit or part select expressions.

- *`number_of_ticks`* is an optional argument that specifies the number of clock ticks in the past. This value must be an integer constant expression, with a value of `1` or greater. If you do not specify *`number_of_ticks`*, it defaults to 1. If the specified clock tick in the past is before the start of simulation, the returned value from the `$past` function is a value of `x`.

  Refer to "Clocking Events for Sampled Value Functions" on page 288 for information about how to determine the clocking event for `$past` and `$sample`.

Clocking Events for Sampled Value Functions

The following rules are used to determine the clocking event for a sampled value function:

- The inferred clock for the procedural code, if any, is used.

  A clock is inferred if the statement is placed in an `always` block with an event control abiding by the following rules:

  - The clock to be inferred must be placed as the first term of the event control as an edge specifier—`posedge` *`expression`* or `negedge` *`expression`*.
  - The variables in *`expression`* must not be used anywhere in the `always` block.

- Otherwise, default clocking is used.

Using sampled value functions inside a `generate` block is not supported when a default clock is applied according to the notes above, and a default clock is specified in the user's code after the `generate` block.

It is an error if no clocking event for a sampled value function can be determined.

Sampled Value Function Example

```
‘define HIGH 1

module test ();

 reg x = 1’b0;
 reg y = 1’b0;
 reg clk = 1’b0;
 reg a = 1’b0;
  initial begin
    #0
    clk = 0;
    x = 0;
    y = 0;
    #20;
    x = 1;
    y = 0;
    #20;
    x = 1;
    y = 1;
    #20
    x = 1;
    y = 1;
    #60
    x = 1;
    y = 1;
    $finish;
  end


  always begin
    #10 clk  = ~clk;
  end

  dut inst1(clk, x, y);
endmodule // test


module dut (clk,x,y);
 input clk, x, y;
 reg x_past, y_past, x_stable, y_stable;
 reg x1, y1, x2, y2;
 reg [0:3] xv;
 parameter P1 = 2;

 default clocking @(posedge clk);
 endclocking

 /* SVF supported usage :
  *   $past( expression1 [, number_of_ticks])
```

```
 *   $stable( expression ) */

always @ (posedge clk) begin
  x_past   = $past(x);  // Number_of_ticks here is 1 and
                        // clocking event here is posedge clk
  y_past   = $past(y, 2);
  x_stable = $stable(x);
  y_stable = $stable(y);
  $monitor($time, "  Value of x in past at each posedge clk : %b ", $past(x));
end


always ##2 begin
 $display($time,"  x: %b  y: %b ",x, y);
 x1 = $past($past(x,2),`HIGH);
 y1 = $past(x && y, 2);
 x2 = $stable(test.x);
 y2 <= $stable(test.y);
end


always begin // Default clock
 xv[0] <= $past((x ^ y), P1);
 xv[1] <= $past(test.a);
 #5 xv[2] = x & $past({x,y}, 2);
 if (x)
  xv[3] = $past(y, P1+1);
 else
 xv[3] = !($stable(y));
end

endmodule : dut
```

Assertion System Functions

Built-in assertion functions are now supported in HDL. You can use these functions in any Boolean expression to perform common tests—for example, to determine whether a signal is one-hot.

SVA system functions have the following syntax:

```
assert_function(expression);
```

where *expression* is a bit vector expression.

The following system functions are included to facilitate common assertion functionality.

Table 19-1 SVA System Functions

Function	Description
`$onehot`	Evaluates the expression and returns true if only one bit is high. Otherwise, the function returns false.
`$onehot0`	Evaluates the expression and returns true if zero or one bit is high. Otherwise, the function returns false.
`$isunknown`	Evaluates the expression and returns true if any bit is `X` or `Z`. Otherwise, the function returns false.
`$countones()`	Evaluates the expression and returns the number of bits whose value is 1; `X` and `Z` values are not counted.

All of these system functions, except `$countones`, have a return type of `bit`.

In the current release, the following data types are not supported for SVA system functions:

- Dynamic data types
- `real`s
- Unpacked structures
- Unpacked unions

See “Writing SystemVerilog Assertions” in the *Assertion Writing Guide* for information about SVA system function support.

20

Compiler Directives

`define

SystemVerilog enhances the `define text substitution macro compiler directive.

- The macro text can include `". This notation indicates that the macro expansion must include a quotation mark. For example:

```
`define msg(x) $time,,, `"Value of x is %d`", x
...
...
$display(`msg(count));
```

  In this example, the macro expands to

```
$display($time,,, "Value of count is %d", count);
```

- The macro text can include `\`". This notation indicates that the macro expansion must include the escape sequence \". For example:

```
`define msg(x) $time,,, `"Value of `\`"x`\`" is %d`", x
...
...
$display(`msg(count));
```

  In this example, the macro expands to

```
$display($time,,, "Value of \"count\" is %d", count);
```

- The macro text can include ``. This notation is used to delimit an identifier name without introducing white space. For example:

```
`define join(a) a``_join
```

This example expands

```
`join(my)
```

to

```
my_join
```

`begin_keywords and `end_keywords

SystemVerilog has added many new keywords and, unfortunately, these additions can render some Verilog source files illegal. Specifically, Verilog files that have identifiers that match a SystemVerilog keyword will not compile on a SystemVerilog compiler. For example, if you have identifiers such as `bit`, `priority`, and `do` in your Verilog file, it will not compile with a SystemVerilog compiler. SystemVerilog extends the `` `begin_keywords `` and `` `end_keywords `` compiler directives that were introduced in the 1364-2005 standard by adding the "1800-2005" *`version_specifier`*, which defines the set of identifiers to use as reserved keywords for a particular block of code.

The `` `begin_keywords `` and `` `end_keywords `` directives

- Can surround only modules, primitives, interfaces, programs, or packages

  These directives must be specified at the top level, and cannot be used within any of these constructs.

- Affect only the treatment of identifiers within a block of code; they do not affect semantics, tokens, or other aspects of the language

- Can be nested

  Nested pairs of `` `begin_keywords `` and `` `end_keywords `` directives are stacked, which means that when the compiler encounters an `` `end_keyword ``, it goes back to the *`version_specifier`* that was in effect prior to the matching `` `begin_keywords `` directive.

These directives have the following syntax:

```
`begin_keywords "version_specifier"
   ...
`end_keywords
```

where *`version_specifier`* can be one of the following:

- `1364-1995`—Indicates that only the identifiers listed as reserved keywords in the IEEE 1364-1995 standard are considered to be reserved words.

- `1364-2001`—Indicates that only the identifiers listed as reserved keywords in the IEEE 1364-2001 standard are considered to be reserved words.

- `1364-2005`—Indicates that only the identifiers listed as reserved keywords in the IEEE 1364-2005 standard are considered to be reserved words.
- `1800-2005`—Indicates that only the identifiers listed as reserved keywords in the IEEE 1800 standard are considered to be reserved words.

Examples:

The following does not cause an error, because `priority` is not a keyword in IEEE 1364-2001, so it can be used as an identifier:

```
`begin_keywords "1364-2001"
module test1 (...);
   output priority;   // Valid
   ...
endmodule
`end_keywords
```

The following causes an error because `priority` is a keyword in IEEE 1800, so it cannot be used as an identifier.

```
`begin_keywords "1800-2005"
module test1 (...);
   output priority;   // Invalid
   ...
endmodule
`end_keywords
```

The following example shows how to use `` `begin_keywords `` and `` `end_keywords `` with program blocks. The following causes an error, because `program` and `endprogram` are not keywords in IEEE 1364-2005:

```
`begin_keywords "1364-2005"

 program p;      // Invalid
   ...
  endprogram: p   // Invalid
`end_keywords
```

To fix the error, use the 1800-2005 *`version_identifier`* instead:

```
`begin_keywords "1800-2005"
  program p;
   ...
  endprogram: p
`end_keywords
```

Limitations on ‘begin_keywords and ‘end_keywords

This section summarizes the keywords features of the IEEE 1800 standard that are not supported in the Cadence implementation. Differences between the IEEE 1800 specification and the Cadence implementation are also listed.

- You cannot use the *version_specifier* to expand the set of keywords that are implied by the -sv or -v1995 command options for ncvlog.

  For example, when you invoke ncvlog without any options, the IEEE 1364-2001 keywords are used by default. In this case, you cannot use the 1800-2005 *version_specifier*, but you can use the 1364-1995 *version_specifier*.

  To work around this limitation, you can use the irun utility to compile your Verilog and SystemVerilog files on a single line—without having to use the -sv switch. The irun utility determines the language of a file by its extension, and maps the file to its appropriate compiler. By default, Verilog files must use the .v extension and SystemVerilog must use the .sv extension. For example:

  ```
  irun vlogfile1.v vlogfile2.v systemv1.sv systemv2.sv
  ```

  In this example, irun will compile the .v files using the ncvlog command, and the .sv files using ncvlog -sv.

  Note: Compiling files using the irun utility is an alternative to using the `begin_keywords and `end_keywords directives when you want to distinguish Verilog files from SystemVerilog files. However, this utility does not automatically support files that contain a mixture of Verilog and SystemVerilog. For those cases, you can use the `begin_keywords and `end_keywords compiler directives within the .sv file, and compile the file using irun.

- The IEEE 1800 specification indicates that the `begin_keywords must be paired with an `end_keywords directive. The specification also indicates that a `begin_keywords is in effect until it reaches its matching `end_keywords directive.

  The Cadence implementation does not require a matching `end_keywords directive. In cases where a matching `end_keywords directive does not exist, the parser issues a warning message, but continues to parse the code using the set of keywords denoted by the last `begin_keyword. When the parser reaches the end of the source file, it will begin parsing the next source file, and the keywords denoted by the last `begin_keywords will remain in effect. The warning message issued by the parser can be upgraded to an error message using the -ncerror command-line option.

- The IEEE 1800 specification does not indicate how the `begin_keywords directive relates to the `resetall directive, which resets all compiler directives to their default values.

  In the Cadence implementation, the `resetall directive does not affect the current set of keywords specified by the `begin_keywords directive.

Reserved Keywords for IEEE 1800

This section lists the set of reserved keywords for the IEEE 1800 standard.

Table 20-1 IEEE 1800 Reserved Keywords

alias	endmodule	matches	small
always	endpackage	medium	solve
always_comb	endprimitive	modport	specify
always_ff	endprogram	module	specparam
always_latch	endproperty	nand	static
and	endspecify	negedge	string
assert	endsequence	new	strong0
assign	endtable	nmos	strong1
assume	endtask	nor	struct
automatic	enum	noshowcancelled	super
before	event	not	supply0
begin	expect	notif0	supply1
bind	export	notif1	table
bins	extends	null	tagged
binsof	extern	or	task
bit	final	output	this
break	first_match	package	throughout
buf	for	packed	time
bufif0	force	parameter	timeprecision
bufif1	foreach	pmos	timeunit
byte	forever	posedge	tran
case	fork	primitive	tranif0
casex	forkjoin	priority	tranif1
casez	function	program	tri
cell	generate	property	tri0
chandle	genvar	protected	tri1

Table 20-1 IEEE 1800 Reserved Keywords, *continued*

class	highz0	pull0	triand
clocking	highz1	pull1	trior
cmos	if	pulldown	trireg
config	iff	pullup	type
const	ifnone	pulsestyle_ onevent	typedef
constraint	ignore_bins	pulsestyle_ ondetect	union
context	illegal_bins	pure	unique
continue	import	rand	unsigned
cover	incdir	randc	use
covergroup	include	randcase	uwire
coverpoint	initial	randsequence	var
cross	inout	rcmos	vectored
deassign	input	real	virtual
default	inside	realtime	void
defparam	instance	ref	wait
design	int	reg	wait_order
disable	integer	release	wand
dist	interface	repeat	weak0
do	intersect	return	weak1
edge	join	rnmos	while
else	join_any	rpmos	wildcard
end	join_none	rtran	wire
endcase	large	rtranif0	with
endclass	liblist	rtranif1	within
endclocking	library	scalared	wor
endconfig	local	sequence	xnor

Table 20-1 IEEE 1800 Reserved Keywords, *continued*

endfunction	localparam	shortint	xor
endgenerate	logic	shortreal	
endgroup	longint	showcancelled	
endinterface	macromodule	signed	

`remove_keyword and `restore_keyword

The `begin_keywords and `end_keywords compiler directives described in the IEEE 1800 standard and in "`begin_keywords and `end_keywords" on page 294 can be used to specify the complete set of reserved keywords in effect when a design unit is parsed. For example, the following compiler directive specifies that only the identifiers listed as reserved keywords in the IEEE 1364-2001 standard are considered to be reserved words. This compiler directive will remove all of the keywords introduced in SystemVerilog.

```
`begin_keywords "1364-2001"
```

However, in transitioning to SystemVerilog, you might have difficulty with a limited number of keywords in the new language that conflict with identifiers commonly used in your code. To provide a way to remove and restore specific keywords, Cadence has implemented a compiler command-line option, ncvlog -rmkeyword, and the `remove_keyword and `restore_keyword compiler directives.

The -rmkeyword command-line option and the `remove_keyword directive can be used to remove a keyword from any set of keywords. That is, their use is not restricted to removing a keyword from the IEEE 1800 set of keywords.

ncvlog -rmkeyword

Using the -rmkeyword command-line option is convenient if you want to remove a particular keyword so that it will always be treated as an identifier in all parts of the design. The syntax is as follows:

```
-rmkeyword keyword
```

For example:

```
-rmkeyword logic
```

Only one keyword can be specified with -rmkeyword. Use the option multiple times to remove multiple keywords. For example:

```
% ncvlog -rmkeyword logic -rmkeyword do test.v
```

`remove_keyword and `restore_keyword Compiler Directives

Use the `remove_keyword and `restore_keyword directives if you need to mix SystemVerilog code with non-SystemVerilog code.

The `remove_keyword directive removes a specified keyword from the set of keywords. This feature disables the functionality provided by the keyword, but allows the keyword to be used as an identifier.

For example:

```
`remove_keyword logic
```

You can specify only one keyword. To remove more than one keyword, you must specify each keyword with a different compiler directive. For example:

```
`remove_keyword logic
`remove_keyword do
```

The following syntax generates an error:

```
`remove_keyword logic do    // Illegal syntax
```

The `restore_keyword directive restores a keyword that was previously removed by `remove_keyword.

These compiler directives, as for the `begin_keywords and `end_keywords directives, can only be specified outside a design element—module, primitive, configuration, interface, program, or package. The `remove_keyword directive affects all source code that follows the directive, even across code file boundaries.

Interaction with Other Compiler Directives

The `remove_keyword directive and the -rmkeyword command-line option take precedence over the `begin_keywords and `end_keywords directives. If a keyword is removed, it is removed from all variations of the language specification, and cannot be reactivated by starting a new set of keywords using `begin_keywords or `end_keywords. The only way to restore a keyword is by using `restore_keyword.

The `remove_keyword directive also takes precedence over the `resetall directive. The set of keywords is not affected by `resetall.

Limitations on Remove and Restore Keywords

This section lists some limitations on the use of the -rmkeyword command-line option and the `remove_keyword and `restore_keyword directives.

- Removed keywords are still highlighted as keywords in the SimVision GUI.
- The compiler directives are not specified in any standard, and they might not be supported by other tools. You can hide the directives from other tools by conditionally compiling the code using `` `ifdef INCA `` or `` `ifdef CDS_TOOL_DEFINE `` in the source code.

21

Direct Programming Interface

SystemVerilog introduces a foreign language interface called the Direct Programming Interface (DPI). DPI provides a simple, straightforward, and efficient way to connect SystemVerilog and C language code, and to build a design or testbench with components written in SystemVerilog and C. The current release extends this feature so that DPI can be used to connect SystemVerilog and SystemC language code.

The following directory contains examples of a SystemVerilog testbench that instantiates a SystemC reference model and uses DPI calls:

```
install_dir/tools/systemc/examples/...
```

For more information about these examples, refer to the *Examples Reference Guide.*

For more DPI examples that you can download and run, refer to the *SystemVerilog DPI Engineering Notebook*.

Importing Functions and Tasks using DPI

With DPI, SystemVerilog code can directly call a C or SystemC function. The functions implemented in C or SystemC are called *imported functions*. The imported function name is imported into the SystemVerilog language using an *import declaration*.

An import declaration specifies the task or function name, the return data type for functions, and the types and directions of the formal arguments. The number of arguments must match the number of arguments in the C or SystemC function, and the data types of the arguments must be compatible with the C or SystemC function data types.

An import declaration defines a task or function in the scope in which the declaration occurs. You cannot, therefore, import the same task or function name into the same module multiple times. However, the same C or SystemC function can be imported into multiple modules.

Imported tasks and functions are called in the same way that native SystemVerilog tasks and functions are called. Calls of imported tasks and functions are indistinguishable from calls of SystemVerilog tasks or functions.

In the following example, a C function called `hello()` is declared in a SystemVerilog module with an import declaration and then called.

C Function

```
#include<stdio.h>
void hello() {
printf("Greetings from the C function!\n");
}
```

SystemVerilog

```
module top;

  import "DPI-C" task hello();
  ...
  initial
  if (sig == 1) hello();
  ...
endmodule
```

pure and context Properties

By default, a C or SystemC function can be imported as a SystemVerilog function or task. Both can have `input`, `output`, and `inout` arguments. Functions can have a return value, or be declared as `void`. The imported task or function cannot access SystemVerilog data objects, other than its actual arguments. A call to the imported task or function can read or write only the actual arguments.

Imported C and SystemC functions can be declared as `pure` or `context`. The following is a brief description of these properties. See the IEEE 1800 standard for a detailed description.

- `pure` functions

  An imported function can be specified as `pure` if the result of the function depends solely on the values of its input arguments. Only non-void functions with no `output` or `inout` arguments can be specified as `pure`. These functions can have no side effects. For example, they cannot read or write anything, access global or static variables, or call other functions.

  For example:

  ```
  import "DPI-C" pure function int calc_parity (input int a);
  ```

  Specifying a function as `pure` can often result in improved simulation performance, because more optimizations can be performed.

  Because a task does not have a return value or a result, a function imported as a task cannot be specified as `pure`.

- `context` functions

  An imported task or function must be specified as `context` if it

- Accesses SystemVerilog data objects other than its actual arguments—for example, through PLI calls
- Calls exported tasks or functions
- Accesses SystemC portions of the design

For example:

```
import "DPI-SC" context function int myclassfunc_func1 ( );
```

Calls of `context` tasks and functions are specially instrumented, so they can impair compiler optimizations. Simulation performance can be affected if the `context` property is specified when it is not necessary.

Violating the rules specified in this section, and incorrectly declaring an imported task or function as `pure` ***or*** `context`***, can result in unpredictable simulation behavior.***

Importing C Functions and Tasks

The syntax in the LRM for declaring a C function imported as a SystemVerilog function is as follows:

```
import {"DPI" | "DPI-C" } [context | pure] [c_identifier =] function
function_data_type function_identifier ([tf_port_list]);
```

Note: The current release supports both "`DPI`" and "`DPI-C`" strings for specifying DPI tasks and functions. However, "`DPI-C`" is the recommended usage, as it supports the IEEE 1800 standard.

In the current release, import declarations that use `DPI-C` are allowed only within modules, packages, interfaces, program blocks, and compilation unit scopes.

For example:

```
import "DPI-C" function int calc_parity (input int a);
```

In this example:

- `int` is the data type of the function return value.
- `calc_parity` is the function identifier used in the SystemVerilog code.
- The function has one input, which is of type `int`.

The syntax for declaring a C function imported as a SystemVerilog task is as follows:

```
import {"DPI"|"DPI-C"} [context] [c_identifier =] task task_identifier
([tf_port_list]);
```

For example:

```
import "DPI-C" task calc_task (input int in1, output int out1);
```

Specifying a Local Name for the C Function

By default, the task or function identifier in the import declaration is assumed to be the same as the function identifier in the C code. However, you can use the *c_identifier* to specify a name to represent the C function name. The specified name must conform to C identifier syntax.

In the following example, the `calc_parity_func` C function is given the name `calc_parity` in SystemVerilog.

```
import "DPI-C" calc_parity_func = function int calc_parity (input int a);
```

Return Types for Imported C Tasks

In the current release, the following data types are supported for imported function return types:

- `void`, `byte`, `shortint`, `int`, `longint`, `real`, `string`, `chandle`
- Scalar values of types `bit` and `logic`

Prior to IUS 8.1, imported tasks did not have return values, and C and SystemC functions that corresponded to an imported task had to return a `void` type. This requirement has changed.

Because support for the `disable` construct within DPI-based designs has been added, C and SystemC functions that correspond to an imported or exported task must return an `int` value. For example, the following defines a C task called `imp_task`, which will be imported into SystemVerilog:

```
int imp_task_c (int x, int y){ /* Return type is int */
..int dis_ret;
  dis_ret = exp_task_c(x,y); /*Return type is int */
  return (dis_ret);
...
}
```

For backward compatibility, you can use the `-dpi_void_task` option with `ncelab` or `irun` on existing DPI designs. Designs will not be affected by this new requirement, and will behave as they did prior to IUS 8.1. However, you must adhere to this new style of DPI function declaration to use the `disable` functionality with DPI-based designs.

See "Disabling DPI Tasks and Functions" on page 329 for more information about the `disable` function.

Formal Arguments for Imported C Functions and Tasks

An imported task or function can have zero or more formal arguments.

By default, each formal argument is assumed to be an input to the C function. You can override this default by explicitly declaring each formal argument as an `input`, `output`, or `inout` argument. For example:

```
import "DPI-C" context task calc_task(input int in1, output int out1);
```

The SystemVerilog data types specified in the import declaration must be compatible with the actual C function data types. No checking is performed to ensure that the data types are compatible, and improper declarations can result in unpredictable behavior and erroneous values.

The LRM lists the data types that are allowed for formal arguments of imported and exported tasks or functions.

In the current release, the following data types are supported as formal arguments for DPI-C imported functions and tasks:

- `byte`, `shortint`, `int`, `longint`, `real`, `string`, and `chandle`
- Scalar values of types `bit` and `logic`
- One-dimensional packed arrays of types `bit` and `logic`
- One-dimensional unpacked arrays of type `byte`, unsigned `byte`, `int`, and unsigned `int`.

  Note: This array type is not supported as a formal argument to imported SystemC functions and tasks.
- Multi-dimensional packed arrays of types `bit` and `logic`
- One-dimensional open arrays of the following datatypes are supported:
  - `int`, `shortint`, `longint`
  - `string`, `byte`
  - `string , byte`
  - Bit vector, logic vector
  - Bit, logic

A formal argument is considered to be an open array when one or more of its dimensions are unspecified. The actual argument can be a fixed or dynamic array.

- Unpacked structs with members of type `longint`, `shortint`, `bitvector`, `int`, `unsigned bit`, `logic`, and `enums` of type `int`, `unsigned int`, `bit`, `logic`, `longint`, `shortint`, and `bitvector`. Special rules apply; see "Using Unpacked Structs as Formal Arguments in DPI-C Import Functions" and "Using Unpacked Structs as Formal Arguments in DPI-SC Import Functions" on page 308.

Using Unpacked Structs as Formal Arguments in DPI-C Import Functions

Unpacked structs with members of type `int`, `unsigned bit`, or `logic` can be used as formal arguments to DPI-C imported functions and tasks. They cannot be used as formal arguments to exported functions and tasks. You will get an error message if a SystemVerilog unpacked structure contains a member that is not of type `int`, `unsigned bit`, or `logic`, and is used as a formal argument to a DPI-C import function or task.

The following are required for DPI-C imported functions and tasks:

- The formal argument in C code must be a pointer to a C struct.
- The layout of the C and SystemVerilog structs must match—that is, the structs must have the same number and ordering of fields.

For examples, refer to "Unpacked Structs as Formal Arguments to DPI-C Import Functions" on page 337 or to the *SystemVerilog DPI Engineering Notebook*.

Using Unpacked Structs as Formal Arguments in DPI-SC Import Functions

Unpacked structs with members of type `int`, `unsigned bit`, or `logic` can be used as formal arguments to DPI-SC imported functions and tasks. They cannot be used as formal arguments to exported functions and tasks. You will get an error message if a SystemVerilog unpacked struct contains a member that is not of type `int`, `unsigned bit`, and `logic` and is used as a formal argument to a DPI-SC import function or task.

The following are required for DPI-SC import functions and tasks:

- The formal argument in SystemC code must be a C struct that can be passed by value, pointer, or reference. The C struct can contain only C data type fields; SystemC data types are not supported. The SystemVerilog data types used within a SystemVerilog unpacked struct will be mapped to C data types, using the DPI-C mapping rules. For example, `bit` will map to `svBit`, and `logic` will map to `svLogic`. However, the current release does not support mapping `bit` to `bool`, or `logic` to `sc_logic`.

To facilitate the conversion of C types to SystemC types, a library of conversion functions has been added called `sc_dpi_convert` class. For more information about this library, refer to "SystemC and HDL Design Hierarchies" in the *SystemC Simulation User Guide*.

- If a SystemVerilog unpacked struct is used as an argument to a DPI-SC imported task or function, it cannot contain members that use SystemC data types.
- The layout of the C and SystemVerilog structs must match—that is, the structs must have the same number and ordering of fields.
- In DPI-SC imports, the names of the structs must match.

  Unlike C, where the linker matches functions by name, regardless of argument types, the C++ linker matches the entire signature of a function, including its argument types.

For an example, refer to "Unpacked Structs as Formal Arguments to DPI-SC Import Functions" on page 339.

Importing SystemC Functions and Tasks

A SystemC function can be imported as a SystemVerilog function or task. To import a task or function, you must:

- Define the following before the first `#include` in your SystemC file:

  ```
  #define NCSC_INCLUDE_TASK_CALLS
  ```

- Declare the task or function using an `import` declaration in SystemVerilog. See "Import Declaration Syntax" on page 310.
- Register the task or function in SystemC using the appropriate `NCSC_REGISTER_DPI*` registration macros. See the "SystemC and HDL Design Hierarchies" chapter of the *SystemC Simulation User Guide* for more information.

  See "DPI Examples" on page 332 for examples.

You can import the following types of SystemC function:

- Global C++ functions
- Static and non-static member methods of arbitrary classes
- Static and non-static member methods of classes that inherit from `sc_object`

  If a class member method inherits from `sc_object`, you must first specify its scope in SystemC before invoking it from SystemVerilog. See "Setting the SystemC Scope" on page 310.

Import Declaration Syntax

The syntax for declaring a SystemC function imported as a SystemVerilog function is as follows:

```
import "DPI-SC" [context | pure] [c_identifier =] function function_data_type
function_identifier ([tf_port_list]);
```

The "DPI-SC" qualifier is for tasks and functions that are interoperable with SystemC. When you use the "DPI-SC" qualifier in your import declaration:

- SystemC functions and tasks are invoked transparently from SystemVerilog
- The simulator maps the SystemC data types to SystemVerilog data types transparently

For example:

```
import "DPI-SC" context function void scmod_run (input sc_int[31:0] i1);
```

In this example:

- void is the data type of the function return value.
- scmod_run is the function identifier used in the SystemVerilog code.
- The function has one input, which is of type sc_int.

The syntax for declaring a SystemC function imported as a SystemVerilog task is as follows:

```
import "DPI-SC" [context | pure] [c_identifier =] task task_id
([tf_port_list]);
```

For example:

```
import "DPI-SC" task sccalc_task (input int in1, output int out1);
```

In the current release, import declarations that use "DPI-SC" are allowed within modules, packages, interfaces, program blocks, and compilation unit scopes.

Setting the SystemC Scope

The base class for all objects in the SystemC design hierarchy is sc_object. If you import a non-static member method of a class that inherits from sc_object, you must set the scope of its corresponding sc_object instance in SystemC before you can invoke it from SystemVerilog. In SystemVerilog, you can use the scSetScopeByName() function to pass the scope information to SystemC. For example:

```
scSetScopeByName("sctop.scm");
```

This statement specifies that any subsequent calls to SystemC sc_object class member methods will use the sctop.scm scope.

Once a scope is set, it persists until another `scSetScopeByName()` is used.

Note: You do not need to set the scope for any other type of imported SystemC function or task, such as global functions, static class member methods, or member methods of an arbitrary class that do not inherit from `sc_object`.

See "DPI Examples" on page 332 for an extensive example.

Return Types for Imported SystemC Functions

For SystemC import declarations, the current release supports the same data types listed in "Return Types for Imported C Tasks" on page 306.

You can use SystemC data types as return types for imported functions, but they require a `typedef`. See "Using typedef with SystemC Data Types" on page 319 for guidelines on using a `typedef` with SystemC data types.

Prior to IUS 8.1, imported tasks did not have return values, and C and SystemC functions that corresponded to an imported task had to return a `void` type. This requirement has changed.

Because support for the `disable` construct within DPI-based designs has been added, C and SystemC functions that correspond to an imported or exported task must return an `int` value. For example, the following defines a C task called `imp_task`, which will be imported into SystemVerilog:

```
int imp_task_c (int x, int y){ /* Return type is int */
  int dis_ret;
  dis_ret = exp_task_c(x,y); /*Return type is int */
  return (dis_ret);
...
}
```

For backward compatibility, you can use the `-dpi_void_task` option with `ncelab` or `irun` for existing DPI designs. Designs will not be affected by this new requirement, and will behave as they did prior to IUS 8.1. However, you must adhere to this new style of DPI function declaration to use the `disable` functionality with DPI-based designs.

Formal Arguments for Imported SystemC Functions and Tasks

For SystemC import declarations, the current release supports the same data types listed in "Formal Arguments for Imported C Functions and Tasks" on page 307.

You can use the following SystemC data types as formal arguments for imported functions. The simulator will implicitly map them to SystemVerilog data types.

Note: Special rules apply for unpacked structures. See "Using Unpacked Structs as Formal Arguments in DPI-SC Import Functions" on page 308.

Table 21-1 Default Data Type Mapping for Formal Arguments

SystemC Data Type	SystemVerilog Data Type
`sc_logic` `sc_lv`	`logic`

`bool`
`sc_bv`
`sc_int`
`sc_uint`
`sc_bigint`
`sc_biguint`
	`bit`

Note: When used as formal arguments, `sc_lv`, `sc_bv`, `sc_int`, `sc_uint`, `sc_bigint`, and `sc_biguint` must be declared as vectors.

For example:

```
import "DPI-SC" task task1 (input sc_int[31:0] i1); // Valid
import "DPI-SC" task task1 (input sc_int i1); // Invalid
```

SystemC data types can be used only within "`DPI-SC`" `import` and `export` declarations. Using them in any other context within a SystemVerilog design will result in an error. You can, however, use a `typedef` to override the default data mapping listed in Table 21-1 on page 312. See "Using typedef with SystemC Data Types" on page 319 for guidelines on using a `typedef` with SystemC data types.

Generating a Header File for Imported Functions and Tasks

Using the *ncelab* option `-dpiimpheader`, you can generate an imported header file that contains all C function prototypes for corresponding imported functions or tasks declared with the keyword `DPI`, `DPI-C`, or `DPI-SC` in SystemVerilog. The generated header file can be included with your C application code to check the consistency of the function prototypes with their actual declarations or definitions.

Because an imported function/task declaration in SystemVerilog code contains the SystemVerilog data types as the arguments to the imported function/task, the equivalent generated C header file uses data mapping between the two languages as defined in Table 21-1 on page 312.

In the imported header file, the return value of an imported task is `int` reflecting the `disabled` value, if no elaborator option `-dpi_void_task` is specified. If you provide a `-dpi_void_task` option to *ncelab*, the return type in the generated imported header file is `void`.

If you have unpacked structures as arguments, some naming convention apply as names of unpacked structures may differ in the two languages. In the created imported header file, a SystemVerilog structure is named as *`name_of_module_or_package__name_of_struct_defined_in_sv`*. You need to edit this name if a file with the same name is already included in your C application.

Exporting SystemVerilog Functions and Tasks using DPI

A C function that corresponds to a `context import` subroutine can directly call a SystemVerilog task or function. Such SystemVerilog tasks and functions are called *exported* tasks and functions. Exported tasks and functions can also be invoked from SystemC processes.

For example, a SystemVerilog function called `hello_sv()` is declared in SystemVerilog, and exported to C using an `export` declaration:

SystemVerilog

```
module top;
export "DPI-C" function hello_sv;
import "DPI-C" context task test();
...
  function void hello_sv(input int a);
  ...
   $display("Hello, world %d\n", a);
  ...
  endfunction
endmodule
```

C Function

```
#include<stdio.h>
extern void hello_sv(int _a1);

 void test(){
 ...
 hello_sv(100);
 ...
 }
```

Exporting Functions and Tasks to C

C functions can call SystemVerilog functions and tasks that are declared using an export declaration.

Note the following for exported functions and tasks:

- In the current release, export declarations are allowed only within modules, packages, interfaces, program blocks, and compilation unit scopes. Export declarations must appear in the scope in which the task or function is defined.

- When using DPI to export functions and tasks, you must include a header file in your C code. You can create your own header file, or generate a header file using the `-dpiheader` switch with `ncelab`. See "Using DPI with the Simulator" on page 322 for more information.
- Exported functions and tasks are subject to the same argument type and result restrictions as imported functions. See "Formal Arguments for Functions or Tasks Exported to C" on page 316 for more information.

Export Declaration Syntax

The syntax in the LRM for declaring a SystemVerilog function or task that will be exported to C is as follows:

```
export {"DPI" | "DPI-C"} [c_identifier = ] function function_identifier;
export {"DPI" | "DPI-C"} [c_identifier = ] task task_identifier;
```

Note: The current release supports both "`DPI`" and "`DPI-C`" strings for specifying DPI tasks and functions. However, "`DPI-C`" is the recommended usage, because it supports the IEEE 1800 standard.

By default, the function or task identifier in the export declaration is the same as the function or task identifier in the C code. However, you can use the *`c_identifier`* to specify a name to represent the C function or task name. The C identifier must conform to the C identifier syntax.

For example, the following specifies that the SystemVerilog function `myfunction` will be exported, and that C will use the identifier `my_cfunction`:

```
export "DPI-C" my_cfunction = function myfunction;
```

An export declaration and its corresponding SystemVerilog function definition can appear in any order. However, the export declaration must occur within the scope for which the function or task is defined. For example:

```
module top;
 export "DPI-C" c_name= function sv_name1;
   function int sv_name1( );
      begin
      ...
      end
   endfunction
endmodule
```

This declaration can also look like the following:

```
module top;
   function int sv_name1( );
      begin
      ...
      end
```

```
   endfunction
 export "DPI-C" c_name= function sv_name1;
endmodule
```

A *c_identifier* cannot be used within the same scope for more than one exported function or task. Also, there can be only one export declaration per task or function within the same scope. In the following example, although there are multiple export declarations that correspond to the same C identifier `c_name`, the example is still valid, because the declarations exist in different scopes and have the same signatures:

```
module top;

   export "DPI" c_name= function sv_name1;

   function int sv_name1( inout int a1, input real a2, output shortint a3,
                          inout byte a4);
     begin
       $display(" \n Value inside the sv_name1 \
       ...
     end
   endfunction
   ...
endmodule

module mid1;

   export "DPI" c_name= function sv_name2;

   function int sv_name2( inout int a1, input real a2, output shortint a3,
                          inout byte a4);
     begin
       $display(" \n Value inside the sv_name2 \
       ...
     end
   endfunction
   ...
endmodule
```

This example also illustrates how exported functions and tasks can have `input`, `output`, and `inout` arguments.

An exported function or task can be called only from within a `context` imported function or task.

Return Types for Functions Exported to C

In the current release, the following data types are supported for exported function and task return types:

- `void`, `byte`, `shortint`, `int`, `longint`, `real`, `chandle`, and `string`
- Scalar values of type `bit` and `logic`

Because support for the `disable` construct within DPI-based designs was added, C and SystemC functions that correspond to an imported or exported task must return an `int` value. For example, the following defines a C task called `imp_task`, which will be imported into SystemVerilog:

```
int imp_task_c (int x, int y){ /* Return type is int */
  int dis_ret;
  dis_ret = exp_task_c(x,y); /* Return type is int */
  return (dis_ret);
...
}
```

For backward compatibility, you can use the `-dpi_void_task` option with `ncelab` or `irun` on existing DPI-C designs. Designs will not be affected by this new requirement and will behave as they did prior to IUS 8.1. However, you must adhere to this new style of DPI-C function declaration to use the `disable` functionality with DPI-based designs.

See "Disabling DPI Tasks and Functions" on page 329 for more information about the `disable` function.

Formal Arguments for Functions or Tasks Exported to C

The SystemVerilog data types specified in the export declaration must be compatible with the actual C function data types. The data types are not checked to ensure that they are compatible, and improper declarations can result in unpredictable behavior and erroneous values.

The LRM lists the data types that are allowed for formal arguments of imported and exported tasks or functions.

In the current release, the following data types are supported as formal arguments for exported functions and tasks:

- `byte`, `shortint`, `longint`, `int`, `real`, `string`, and `chandle`
- Scalar values of types `bit` and `logic`
- One-dimensional packed arrays of types `bit` and `logic`
- One-dimensional unpacked arrays of type `byte`, unsigned `byte`, `int`, and unsigned `int`

  Note: This array type is not supported as a formal argument to imported SystemC functions and tasks.
- Multi-dimensional packed arrays of types `bit` and `logic`
- Packed structs of types `bit` and `logic`

Exporting SystemVerilog Functions and Tasks to SystemC

SystemC processes can call SystemVerilog functions and tasks that are declared using an `export` declaration. Exported functions and tasks can also be called from `context` imported functions and tasks that are qualified with "`DPI-SC`". Every exported function must

- Define the following before the first `#include` in your SystemC file:

  ```
  #define NCSC_INCLUDE_TASK_CALLS
  ```

- Have an `export` declaration

  See "Export Declaration Syntax" on page 318.

- Be declared in SystemC as external by using the `extern "C"` keyword, which will indicate that the task or function's definition resides in another source file

  See Example on page 335 for an example.

- Specify its SystemVerilog scope using the `svSetScope()` function in SystemC

  The `svSetScope()` function has the following syntax:

  ```
  svSetScope(svGetScopeFromName("scope");
  ```

  However, if you are calling an exported function from within an imported function, `svSetScope()` might not be required; see Example on page 335.

- Exported functions can be invoked from the following places in a SystemC design:

  - From a SystemC `end_of_elaboration` callback that is invoked during simulation

    The callback function can invoke exported functions.

  - From a SystemC `start_of_simulation` callback

    The callback function can invoke exported functions.

  - From a method process or a thread process

    During process execution, a SystemC design can call exported functions.

  - From the `sc_main()` function

    `$call_systemc_function()` and `$call_systemc_process_nb()` can invoke a SystemC function, or schedule a SystemC process that can further invoke exported functions.

You can also call the following accessor functions from each of the places listed above:

svSetScope	svGetNameFromScope	svGetScopeFromName
svPutUserData	svGetUserData	svGetScope

Export Declaration Syntax

The syntax for declaring a SystemVerilog function or task that will be exported to SystemC is as follows:

```
export "DPI-SC" [c_identifier = ] function function_identifier;
export "DPI-SC" [c_identifier = ] task task_identifier;
```

The "DPI-SC" qualifier is for tasks and functions that are interoperable with SystemC. When you use the "DPI-SC" qualifier in your export declaration:

- SystemVerilog functions and tasks are invoked transparently from SystemC.
- The simulator maps the SystemC data types to SystemVerilog data types transparently

By default, the function or task identifier in the export declaration is the same as the function identifier in the SystemC code. However, you can use the *c_identifier* to specify a name to represent the SystemC function. The C identifier must conform to SystemC identifier syntax. For example, the following specifies that the SystemVerilog function myfunction will be exported, and that SystemC will use the identifier my_scexp.

```
export "DPI-SC" my_scexp = function sc_exp;
```

Return Types for Functions Exported to SystemC

For SystemC export declarations, the current release supports the same data types listed in "Return Types for Functions Exported to C" on page 315.

You can also use SystemC data types as return types for exported functions but they require a typedef. See "Using typedef with SystemC Data Types" on page 319 for guidelines on using a typedef with SystemC data types.

Formal Arguments for Functions or Tasks Exported to SystemC

For SystemC export declarations, the current release supports the same data types listed in "Formal Arguments for Functions or Tasks Exported to C" on page 316.

You can also use SystemC data types as formal arguments for exported functions. The simulator will map them to a SystemVerilog data type. See Table 21-1 on page 312 for a list of supported SystemC data types.

SystemC data types can be used only within "DPI-SC" import and export declarations. Using them in any other context within a SystemVerilog design will result in an error. You can, however, use a typedef to override the default data mapping listed in Table 21-1 on page 312. See "Using typedef with SystemC Data Types" on page 319 for guidelines on using a typedef with SystemC data types.

Limitations on Functions and Tasks Exported to SystemC

In the current release

- Export declarations are allowed only within modules, packages, interfaces, and compilation unit scopes
- Export functions can be invoked from SystemC methods and threads
- Export tasks can be invoked from SystemC threads

  These exported tasks can have SystemVerilog blocking constructs that consume simulation time. For more information about tasks that consume time, refer to "Tasks That Consume Time" on page 321.

Using typedef with SystemC Data Types

SystemC data types can be used within only "DPI-SC" import and export declarations. Using them in any other context, without a typedef, will result in an error.

If you use a SystemVerilog typedef to create a new definition for a SystemC data type, the type definition must satisfy the data type mapping listed in tables 21-2 and 21-3; these tables apply to both import and export declarations. Otherwise, you will get a compilation error. If the typedef is used outside "DPI-SC" import or export declarations, the regular SystemVerilog type definition semantics will apply.

Table 21-2 Data Type Mapping for Function Return Types

SystemC Data Type	Valid SystemVerilog Data Types
sc_logic	logic, bit

SystemC Data Type	Valid SystemVerilog Data Types
bool	bit, logic
sc_int sc_uint sc_bigint sc_biguint	int, shortint, longint

Table 21-3 Data Type Mapping for Formal Arguments

SystemC Data Type	SystemVerilog Data Type
sc_logic sc_lv	logic, bit
bool sc_bv	bit, logic
sc_int sc_uint sc_bigint sc_biguint	bit, int, shortint, longint

Note: When using a typedef to map sc_lv, sc_bv, sc_int, sc_unit, sc_bigint, or sc_biguint to type bit or logic, you must declare these types as vectors. For example:

```
typedef bit sc_int;
import "DPI-SC" task1 (input sc_int[31:0] i1); // Valid
import "DPI-SC" task2 (input sc_int i1); // Invalid
...
typedef int sc_int;
import "DPI-SC" task3 (input sc_int a1); // Valid
```

Example 1

```
typedef int sc_int; // Maps sc_int to type int, which is a valid data type
import "DPI-SC" void func1 (input sc_int a);
```

The simulator will map sc_int to type int before it invokes the SystemC func1 function.

Example 2

```
typedef string sc_int; // sc_int cannot be mapped to strings
import "DPI-SC" void func2 (input sc_int a);
```

This example results in a compilation error, because the typedef maps sc_int to an invalid type.

Example 3

```
typedef bit sc_int;
import "DPI-SC" function void openArrayFunc(input sc_int[7:0] a[]);
```

This example results in an error message, because open arrays are not supported for `typedefs` that correspond to SystemC data types.

Tasks That Consume Time

In the current release, SystemVerilog can invoke a DPI import call chain that eventually consumes time in SystemC or in SystemVerilog. Likewise, SystemC can invoke a DPI export call chain that eventually consumes time in SystemVerilog or in SystemC. These types of export call chains can be invoked only from a SystemC thread process. Nested call chains of arbitrary depth are also supported in this release.

To consume time in SystemC, a task that is imported using "`DPI`", "`DPI-C`", or "`DPI-SC`" must use the `wait(...)` construct. For more information about the different forms of the `wait(...)` construct, see the "SystemC and HDL Design Hierarchies" chapter of the *SystemC Simulation User Guide* for more information.

Note: Import tasks defined in SystemVerilog program blocks cannot call `wait(...)` in SystemC. This call will result in an error message.

To consume time in SystemVerilog, a task that is exported using "`DPI`", "`DPI-C`", or "`DPI-SC`" can use the following SystemVerilog constructs:

- `fork...join`, `fork...join_none`, and `fork...join_any`
- `wait fork` and `wait`
- Semaphores

  Semaphores provide a `get()` method that blocks execution of a process until a key is available.
- Mailboxes

  Mailboxes provide a `put()` method that suspends a process until there is enough room in the queue.
- Cycle delays

  You can introduce cycle delays by using default clocking blocks.

  The ## operator delays execution by a specified number of clocking events or clocking cycles.

- Blocking and non-blocking assignments that use delay-based timing controls

  The # symbol, a delay-based timing control, specifies how long to wait before executing a statement. For example:

  ```
  #5 a = b;
  a = #5 9 b;
  a <= #4 b;
  ```

- Statements that include event-based timing controls

  The @ symbol represents an event-based timing control. It specifies that a statement can be executed when a signal value changes, or when a specified event has triggered. For example:

  ```
  @ (event_identifier)
  @(*)
  @ a(iff en == 1)
  ```

Note: Disabling a SystemVerilog process that is in the middle of a import call chain is not supported, and will result in a fatal error during run-time.

For example, the following call chains can successfully consume time in the current release:

SystemVerilog:

Calls an imported task → Issues `wait(...)`
Calls an imported task → Calls an exported task → Issues `@`
Calls an imported task → Calls an exported task → Calls an imported task → Issues `wait(...)`

SystemC:

Calls an exported task → Issues `@`
Calls an exported task → Calls an imported task → Issues `wait(...)`
Calls an exported task → Calls an imported task → Calls an exported task → Issues `@`

The semantics for consuming time in a DPI call chain is the same as if the call was occurring in one language—the language that originated the DPI call chain. In addition, the DPI call chain can resume execution in the same delta cycle in which the time-consuming condition is satisfied—or example, if an event is triggered—without artificial delta delays.

See the “SystemC and HDL Design Hierarchies” chapter of the *SystemC Simulation User Guide* for more information.

Using DPI with the Simulator

- Using the irun Utility with DPI on page 323

- Using the Incisive Simulator with DPI on page 325

Using the irun Utility with DPI

With the *irun* utility, you can run the simulator by specifying all input files and all command-line options on a single command line. The *irun* utility determines the language of a file by its extension, and maps the file to its appropriate compiler.

Importing C Tasks and Functions using irun

To use the *irun* utility with DPI to import C tasks and functions, use the following command:

```
% irun -dpiimpheader design_files c_files
```

Where *design_files* are the design files that contain the DPI `import` statement, which imports the tasks and functions contained in the specified *c_files*. For example:

```
% irun -dpiimpheader systemv1.sv systemv2.sv cfile1.c cfile2.c
```

In this example, *irun* compiles the `.sv` files using `ncvlog -sv`, and the C files using a C compiler. After the input files have been compiled, *irun* automatically invokes *ncelab* to elaborate the design, and *ncsim* to simulate the design.

Exporting Tasks and Functions to C using irun

To use the *irun* utility with DPI to export SystemVerilog tasks and functions, use the following command:

```
% irun design_files -dpiheader header_file_name -cpost c_files -end
```

where

- *design_files*—The design files that contain the tasks and functions to export. These files are compiled using `ncvlog -sv` before proceeding to elaboration.

  Note: By default, SystemVerilog files must have the `.sv`, `.svp`, `.SV`, or `.SVP` extension. For more information about how to modify the default extensions, refer to "Compiling SystemVerilog Constructs" on page 23 or the *irun User Guide*.

- *header_file_name*—You must include a header file in your C files that reference exported tasks and functions. The `-dpiheader header_file_name` option specifies this header file. If the specified header file does not exist, `ncelab` creates it for you. If you have your own header file, save the header file in the directory where *irun* will be invoked.

For an example of how to include the header file in your C files, refer to “Debugging DPI Exported Functions and Tasks” in *SystemVerilog in Simulation*.

Important

You must create the header file before the C files are compiled.

- *c_files*—The C files that reference the exported SystemVerilog tasks and functions. If your C files contain export functionality, they must be specified between the `-cpost` and `-end` options, so that the *irun* utility knows to compile these files with a C compiler *after* elaboration.

For example:

```
% irun top.sv -access +rwc -dpiheader myheader.h -cpost add.c -end
```

In this example, *irun* compiles the `top.sv` file using `ncvlog -sv`, then proceeds to elaboration. During elaboration, all `export` declarations are dumped into the `myheader.h` file. After elaboration, *irun* compiles the *add.c* file using a C compiler, then proceeds to simulation.

For information about how to export SystemVerilog tasks and functions using the Incisive simulator, refer to “Exporting SystemVerilog Tasks and Functions using the Incisive Simulator” on page 326.

For more information about `irun`, refer to the *irun User Guide*.

Importing SystemC Tasks and Functions Using irun

To use the *irun* utility with DPI to import SystemC tasks and functions, use the following command:

```
% irun -dpiimpheader design_files systemc_files -sysc
```

For example:

```
% irun -dpiimpheader systemv1.sv system2.sv systemcfile.cpp -sysc
```

In this example, *irun* compiles the `.sv` files using `ncvlog -sv`, and compiles the `.cpp` file using the `ncsc_run` compiler interface.

Note: By default, SystemVerilog files must have the `.sv`, `.svp`, `.SV`, or `.SVP` extension. For more information about how to modify the default extensions, see “Compiling SystemVerilog Constructs” on page 23, or refer to the *irun User Guide*.

Exporting Tasks and Functions to SystemC using irun

To use the *irun* utility with DPI to export SystemVerilog tasks and functions to SystemC, use the following command:

```
% irun -Wcxx -fPIC systemc_files sv_files -sysc
```

where

- `-Wcxx`—Passes the user-specified arguments to C++.
- `-fPIC`—Is required for all C and C++ designs that have `extern` declarations.
- *`systemc_files`*—The SystemC files that reference the exported SystemVerilog tasks and functions.
- *`sv_files`*—The SystemVerilog files that contain the tasks and functions to export. These files are compiled using `ncvlog -sv` before proceeding to elaboration.

Note: By default, SystemVerilog files must have the `.sv`, `.svp`, `.SV`, or `.SVP` extension. For more information about how to modify the default extensions, see "Compiling SystemVerilog Constructs" on page 23, or refer to the *irun User Guide*.

For example:

```
% irun -Wcxx -fPIC -dpiimpheader sc.cpp test.sv -sysc
```

For more information about *irun*, refer to the *irun User Guide*.

Using the Incisive Simulator with DPI

This section describes how to import C tasks and functions into SystemVerilog, and how to export SystemVerilog tasks and functions to C.

To import SystemC functions, or to export SystemVerilog functions to SystemC, see "Using the irun Utility with DPI" on page 323.

Importing C Tasks and Functions using the Incisive Simulator

To use the Incisive simulator with DPI to import C tasks and functions, do the following:

1. Compile the design using the following command:

   ```
   % ncvlog -sv design_files
   ```

   where *`design_files`* are the design files that contain the DPI `import` statement, which imports the C tasks and functions contained in the specified *`c_files`*. For example:

```
% ncvlog -sv top.v
```

2. Elaborate the design using `ncelab`. For example:

```
% ncelab -access +RWC -dpiimpheader top
```

3. In the current directory, create a single shared object library. See "Compiling and Linking C Objects into a Single Shared Object Library" on page 327.

4. Simulate the design using `ncsim`. For example:

```
% ncsim -messages top
```

   If you compiled and linked your C code into a shared library other than `libdpi.`*`ext`*, see "Simulating the Elaborated Snapshot of Libraries Other than libdpi.ext" on page 328.

Exporting SystemVerilog Tasks and Functions using the Incisive Simulator

To use the Incisive simulator with DPI to export SystemVerilog tasks and functions:

1. Create a header file.

   When using DPI to export tasks and functions, you must first include a header file in your C code. During elaboration, all `export` declarations are dumped into the header file. You can use your own header file, or use `ncelab` to generate one for you.

   If you have your own header file, save it in the directory where the Incisive simulator will be invoked.

Important

   You must create the header file before the C objects are compiled.

   Include the header file in your C files that reference the exported tasks and functions. You must include the header file so that the C symbols corresponding to the exported tasks and functions are externally visible. For an example of how to include the header file in your C files, see "Debugging DPI Exported Functions and Tasks in *SystemVerilog in Simulation*.

2. Compile and link your C objects into a single shared object library corresponding to your exported tasks and functions. For instructions, see "Compiling and Linking C Objects into a Single Shared Object Library" on page 327.

3. Go to "Running the Simulator in Multi-Step Mode" on page 327.

Compiling and Linking C Objects into a Single Shared Object Library

If you are exporting SystemVerilog tasks and functions using the Incisive simulator, you must compile and link your C objects into a shared library that corresponds to the exported tasks and functions.

Note: Save the shared library in the directory where *ncsim* or *irun* will be invoked, or include the path to the shared library in the library path environment variable—`LD_LIBRARY_PATH` for Solaris and Linux, `SHLIB_PATH` for HP-UX, or `LIBPATH` for AIX.

To compile and link your C objects into a shared library using a `gcc` compiler:

```
% gcc -fPIC -shared -o libdpi.so add.c -I $CDS_INST_DIR/tools/inca/include
```

To compile and link your C objects into a shared library using a `cc` compiler:

```
% cc -KPIC -G -o libdpi.so -I $CDS_INST_DIR/tools/inca/include add.c
```

where

- `add.c` is the C file to compile
- `libdpi.so` is the shared library to build

  Note: By default, the simulator looks for a single shared library called `libdpi.so` for Solaris, Linux, and AIX; or `libdpi.sl` for HP-UX. However, the `-sv_root` and `-sv_lib` options to *ncsim* let you specify a shared library other than `libdpi.`*`ext`*. These options are discussed in "Running the Simulator in Multi-Step Mode" on page 327.
- `$CDS_INST_DIR` is an environment variable that points to your installation

Running the Simulator in Multi-Step Mode

If you are using the multi-step invocation mode of the Incisive simulator and you have your own header file, do the following:

1. Use the `ncvlog` command with the `-sv` command-line option to compile the SystemVerilog code. For example:

   ```
   % ncvlog -mess test.v -sv
   ```
2. Use the `ncelab` command to elaborate the design. For example:

   ```
   % ncelab -mess top
   ```
3. Simulate the elaborated snapshot. For example:

   ```
   % ncsim -mess top
   ```

   If you compiled and linked your C code into a shared library other than `libdpi.`*`ext`*, see "Simulating the Elaborated Snapshot of Libraries Other than libdpi.ext" on page 328.

If you need to generate a header file using `ncelab`, do the following:

1. Use the `ncvlog` command with the `-sv` command-line option to compile the SystemVerilog code. For example:

   ```
   ncvlog -mess test.v -sv
   ```

2. Use the `ncelab` command with the `-dpiheader` command-line option to generate the header file. For example:

   ```
   ncelab -mess top -dpiheader /home/john/myheader.h
   ```

   The `-dpiheader` switch generates a header file that contains declarations for all of the C identifiers that correspond to exported tasks and functions contained in the elaborated snapshot.

   The elaborator saves the header file in the directory specified by the `-dpiheader` switch.

   In your C files that reference the exported tasks and functions, you must include the header file so that the C symbols corresponding to the exported functions are externally visible. For an example of how to include the header file in your C files, see “Debugging DPI Exported Functions and Tasks” in *SystemVerilog in Simulation*.

   You must generate the header file before you compile your C objects.

3. Compile and link your objects into a single shared object library that corresponds to your exported tasks and functions. For instructions, see “Compiling and Linking C Objects into a Single Shared Object Library” on page 327.

4. Simulate the elaborated snapshot. For example:

   ```
   % ncsim -mess top
   ```

   If you compiled and linked your C code into a shared library other than `libdpi.`*`ext`*, see “Simulating the Elaborated Snapshot of Libraries Other than libdpi.ext” on page 328.

Simulating the Elaborated Snapshot of Libraries Other than libdpi.ext

If you compiled and linked your C code into a shared library other than `libdpi.`*`ext`*, you must specify the path and name of the shared library using the `-sv_root` and `-sv_lib` options to *ncsim*. For example:

```
% ncsim -mess top -sv_root /a/b/c -sv_lib mylib
```

where

- `-sv_root` is a single directory that will be prefixed to any relative path specified by `-sv_lib`
- `-sv_lib` can be a relative or full pathname that corresponds to a shared library; if you provide a relative path, it is prefixed by the path specified by `-sv_root`

For example, the following loads `/a/b/c/mylib.`*`ext`* during simulation:

```
% ncsim -mess top -sv_root /a/b/c -sv_lib mylib
```

Using the `-sv_root` and `-sv_lib` options with *ncsim* allows you to load multiple shared libraries during simulation. For example, the following loads `/x/y/myl.`*`ext`* and `../myl1.`*`ext`* during simulation:

```
% ncsim -mess top -sv_root /x/y/ -sv_lib myl -sv_root ../ -sv_lib myl1
```

Disabling DPI Tasks and Functions

For an example that you can download and run, refer to the *SystemVerilog DPI Engineering Notebook*.

In SystemVerilog, tasks can be disabled by using a `disable` statement. An imported task or function is considered disabled when it or its parent task is the target of a SystemVerilog `disable` statement. In the current release, you can use the `disable` statement to disable process call chains that contain DPI imported tasks or functions and exported tasks.

To account for disabled imported tasks, or the exported task that they call, the C code must follow a certain protocol to programmatically acknowledge that a task has been disabled:

- Imported C or SystemC tasks must return an `int` value of 1 to indicate a disable. Otherwise, they return a value of 0. If a disabled task does not return a value of 1, the simulator issues an error message.
- When an imported C or SystemC function is disabled, it must call the `svAckDisabled` API function. If a disabled function does not call this API function before it returns to its parent, the simulator issues an error message.
- When an exported task is the target of a `disable`, its parent imported task is not considered disabled when the exported task returns. The exported task returns 0, and any calls to `svIsDisabledState()` return 0. When the parent import task and export task are disabled, the export task must return an integer value of 1.

Imported tasks and functions, and exported tasks, can determine whether they are disabled by calling the `svIsDisabledState()` API function.

The following illustrates an imported C task that calls an exported SystemVerilog task.

In SystemVerilog:

```
...
import "DPI-C" context task imp_t1(inout int a);
export "DPI-C" task exp_t1;

task exp_t1(output int a);
...
endtask
```

In C:

```
extern int exp_t1 (int *a);

int imp_t1(int *a);
{
int ret;
...
ret = exp_t1();
...
return ret;
}
```

In this example, the `imp_t1` and `exp_t1` functions return an `int`. The `exp_t1` function will return 0 unless its parent function, `imp_t1`, is disabled, in which case `exp_t1` will return 1. The `imp_t1` function must check for a disable, and acknowledge any disables by returning 1:

```
int imp_t1(int *a);
{
int ret;
...
ret = svIsDisabledState();
...
return ret;
}
```

According to the LRM, the `disable` statement cannot be used to disable exported functions.

Debugging DPI Import and Export Functions

For information about how to debug DPI import and export functions using the Tcl command-line interface, see *SystemVerilog in Simulation*.

DPI Accessor Functions

The following DPI accessor functions are supported in the current release.

Table 21-4 Supported DPI Accessor Functions

svAckDisabledState	svGetArrayPtr	svGetArrElemPtr1
svGetBitArrElem1	svGetBitArrElem1Vec32	svGetBitArrElem1VecVal
svGetBitselBit	svGetBitselLogic	svGetCallerInfo
svGetLogicArrElem1	svGetLogicArrElem1Vec32	svGetLogicArrElem1VecVal
svGetNameFromScope	svGetPartselBit	svGetPartSelectBit
svGetPartSelectLogic	svGetPartselLogic	svGetScope
svGetScopeFromName	svGetSelectBit	svGetSelectLogic
svGetUserData	svHigh	svIsDisabledState
svLeft	svLow	svPutBitArrElem1
svPutBitArrElem1Vec32	svPutBitArrElem1VecVal	svPutBitselBit
svPutBitselLogic	svPutLogicArrElem1	svPutLogicArrElem1Vec32
svPutLogicArrElem1VecVal	svPutPartselBit	svPutPartSelectBit
svPutPartSelectLogic	svPutPartselLogic	svPutSelectBit
svPutSelectLogic	svPutUserData	svRight
svSetScope	svSizeOfArray	

The following DPI accessor functions are not supported in the current release.

Table 21-5 Unsupported DPI Accessor Functions

svGet32Bits	svGetArrElemPtr3	svGetBitArrElem
svGetArrElemPtr2	svGetBitArrElem2Vec32	svGetBitArrElem2VecVal
svGetBitArrElem2	svGetBitArrElem3Vec32	svGetBitArrElem3VecVal
svGetBitArrElem3	svGetBitArrElemVecVal	svGetBits
svGetBitArrElemVec32	svGetLogicArrElem	svGetLogicArrElem2
svGetBitVec32	svGetLogicArrElem2VecVal	svGetLogicArrElem3
svGetLogicArrElem2Vec32	svGetLogicArrElem3VecVal	svGetLogicArrElemVec32
svGetLogicArrElem3Vec32	svGetLogicVec32	svIncrement

svGetLogicArrElemVecVal	svLength	svPutBitArrElem
svPutBitArrElem2	svPutBitArrElem2Vec32	svPutBitArrElem2VecVal
svPutBitArrElem3Vec32	svPutBitArrElem3VecVal	svPutBitArrElemVec32
svPutBitArrElemVecVal	svPutBitVec32	svPutLogicArrElem
svPutLogicArrElem2	svPutLogicArrElem2Vec32	svPutLogicArrElem2VecVal
svPutLogicArrElem3	svPutLogicArrElem3Vec32	svPutLogicArrElem3VecVal
svPutLogicArrElemVec32	svPutLogicArrElemVecVal	svPutLogicVec32
svSizeOfBitPackedArr	svSizeOfLogicPackedArr	
svDimensions	svDpiVersion	
svGet64Bits	svGetArrElemPtr	

DPI Examples

This section provides extensive examples of using DPI with C and SystemC.

For more DPI examples that you can download and run, see the *SystemVerilog DPI Engineering Notebook*.

Using DPI with C

In SystemVerilog:

```
module top;
 int top_res;
 bit b1 = 1'b1;
 bit b2 = 1'b1;

 import "DPI-C" context xxx = function int imp_func(input bit bin1, inout bit bin2);
 export "DPI-C" ccc = function exp_func;

 function int exp_func(input bit eb1, output bit eb2);
   if(eb1 != eb2)
    begin
      eb2 = eb1;
      return -43;
    end
   else
      return 54;
 endfunction

 initial
   begin
     #2 top_res = imp_func(b1,b2);
     $display("top_res : %d\n", top_res);
   end

endmodule
```

Imports the C function called `xxx` and calls it `imp_func` in SystemVerilog.

Exports the SystemVerilog function called `exp_func` and calls it `ccc` in C.

Defines `exp_func` that will be exported to C.

Invokes the imported C function `xxx`.

In C:

```
#include <stdio.h>
#include "myheader.h"
  extern int ccc (unsigned char, unsigned char *);

  int xxx (svBit b1, svBit *b2)
  {
  int res;
  printf("Before calling ccc ****** b1: %d, b2 : %d\n", b1, *b2);
  res = ccc(b1,b2);
  printf("Res:%d\n", res);
  printf("After calling ccc ****** b1: %d, b2 : %d\n",b1, *b2);

  printf("\n\nBefore calling ccc ****** b1: %d, b2 : %d\n", b1, *b2);
  res = ccc(b1,b2);
  printf("Res:%d\n", res);
  printf("After calling ccc ****** b1: %d, b2 : %d\n",b1, *b2);
  return res
  }
```

Includes the header file generated by `ncelab`.

Declares the exported SystemVerilog function `xxx` as extern .

Defines the imported `xxx` function.

Invokes the exported function `exp_func`.

In this example:

- In SystemVerilog, the `top` module's `imp_func` function invokes the `xxx` C function.

- In C, the `xxx` function calls the `ccc` function twice, before and after probing the values of `b1` and `b2`. The call to the `ccc` function invokes the SystemVerilog `exp_func` function with `b1` and `b2` as `svBit` type parameters. The return value of the `exp_func` function is assigned to `res`.

For example:

```
irun -sv test.v testme.c
```

produces the following simulation result:

```
Before calling ccc ******b1 : 1, b2 :0
Res : -43
After calling ccc ******b1: 1, b2: 1

Before calling ccc ******b1 : 1, b2 :1
Res : 54
After calling ccc ******b1: 1, b2: 1
top_res: 54
```

Using DPI with SystemC

In SystemVerilog (test.sv file):

```
module scmod
  (* integer foreign = "SystemC"; *);
endmodule

module svtop;

  scmod scm();
  import "DPI-SC" context function void scmod_run ( input sc_int[31:0] a);
  export "DPI-SC" function sc_exp;

  function void sc_exp ( input sc_int[31:0] i);
   $display(" Inside sc_exp : value is %d ",i);
  endfunction

 initial begin
  scSetScopeByName("svtop.scm");
  scmod_run(17);
 end
endmodule
```

Imports the SystemC `sc_object` class member method `scmod_run` and uses a SystemC type as a formal argument.

Exports the SystemVerilog function called `sc_exp`.

Defines `sc_exp`, which will be exported to C.

Sets the scope and invokes the imported `sc_object` class member method `scmod_run`.

In SystemC (sc.cpp file):

```
#define NCSC_INCLUDE_TASK_CALLS
#include "systemc.h"

extern "C" {
  extern void sc_exp(sc_int<32> a);
}

class scmod : public sc_module {
  public:
  SC_CTOR(scmod) {
  }
  void scmod_run ( sc_int<32> i1) {
   cout << "Inside SystemC value is " << i1 << endl;
  sc_exp(i1);
  }
};

NCSC_MODULE_EXPORT(scmod)
NCSC_REGISTER_DPI_MEMBER_ALIAS(scmod_run,scmod,scmod_run)
```

`#define` must come before the first `#include` of `systemc.h`.

Declares the exported SystemVerilog function `sc_exp` as `extern`.

Defines the imported `sc_object` class member method `scmod_run`.

Invokes the exported function `sc_exp`.

Registers member method that inherits from `sc_object`.

For example:

```
% irun -Wcxx -fPIC sc.cpp test.sv -sysc
```

produces the following simulation result:

```
Inside SystemC value is 17
Inside sc_exp : value is 17
```

Using scSetScopeByName in SystemVerilog

In SystemVerilog (test.sv file):

```
`timescale 1 ns/1 ps

  module sctop (* integer foreign = "SystemC"; *);
  endmodule

module top;
  int r1;
  byte b1 = "A";
  sctop sc();
  import "DPI-SC" function int myglobalfunc (input byte a);
  import "DPI-SC" function int myclassfunc (input byte a);
  initial begin
   #10
   r1=myglobalfunc( b1);
   #10
   scSetScopeByName("top.sc");
   r1=myclassfunc(b1);
   $finish;
end
endmodule
```

Imports two SystemC functions.

Calls an imported function. Does not require `scSetScopeByName()`, because `myglobalfunc()` is global.

Sets the scope to `top.sc`.

Calls an imported function. Requires `scSetScopeByName()`, because `myclassfunc()` is a member method of a class that inherits from `sc_object`.

In SystemC (sc.cpp file):

```
#define NCSC_INCLUDE_TASK_CALLS
#include "systemc.h"


int myglobalfunc(char i1) {
  cerr << sc_time_stamp() << " starting myglobalfunc - i1 = " << i1
  <<  endl;
  return 0;
}

class sctop : public sc_module {
public:
  SC_CTOR(sctop) {
  }
  int myclassfunc(char i1) {
  cerr << sc_time_stamp() << " starting myclassfunc - i1 = " << i1
  <<  endl;
  return 0;
}

};

NCSC_MODULE_EXPORT(sctop)
NCSC_REGISTER_DPI(myglobalfunc)
NCSC_REGISTER_DPI_MEMBER_ALIAS(myclassfunc,sctop,myclassfunc)
```

`#define` must come before the first `#include` of `systemc.h`.

Defines the global imported function `myglobalfunc()`.

Defines the sc_object class member method imported function `myclassfunc()`.

Registers global functions.

Registers member methods that inherit from `sc_object`.

For example:

```
% irun sc.cpp test.sv -sysc
```

produces the following results:

```
10 ns starting myglobalfunc - i1 = A
20 ns starting myclassfunc - i1 = A
```

Unpacked Structs as Formal Arguments to DPI-C Import Functions

This example demonstrates the guidelines listed in "Using Unpacked Structs as Formal Arguments in DPI-C Import Functions" on page 308.

In SystemVerilog (test.sv file):

```
module top;

typedef struct {
  int a;
  logic b;
  bit c;
} mystruct ;

import "DPI-C" task t1(input mystruct b);
  mystruct aa;
  initial begin
  aa.a = 10;
  aa.b = 1'bz;
  aa.c = 1'b1;
  t1(aa);
  end
endmodule
```

Layout of SystemVerilog and C structs match.

In C:

```
#include<stdio.h>
#include<svdpi.h>

typedef struct {
  int a;
  svLogic b;
  svBit c;
}myCstruct;


int t1(myCstruct *p) {
  printf("p->a = %d\n",p->a);
  printf("p->b = %d\n",p->b);
  printf("p->c = %d\n",p->c);
}
```

Formal argument in C is a pointer to a C struct.

For example:

```
% irun testme.c test.sv
```

produces the following results:

```
p->a = 10
p->b = 2
p->c = 1
```

Unpacked Structs as Formal Arguments to DPI-SC Import Functions

In SystemVerilog (test.sv file):

```
module top;
 typedef struct {
  int a;
  logic b;
  bit c;
} mystruct;

import "DPI-SC" function void sc_imp_func(
input mystruct a1,
input sc_int[0:31] a2
);
mystruct aa;

initial begin
  aa.a = 10;
  aa.b = 1'bz;
  aa.c = 1'b1;
  sc_imp_func(aa, 32);
end
endmodule
```

Layout and names of SystemVerilog and C structs match.

In SystemC (testme.cpp):

```
#define NCSC_INCLUDE_TASK_CALLS
#include <systemc.h>
#include "sysc/cosim/sc_dpi_convert.h"

typedef struct {
  int a;
  svLogic b;
  svBit c;
} mystruct;

void sc_imp_func(mystruct a1, sc_int<32> a2) {

 int a;
 sc_logic b;
 bool c;

 sc_dpi_convert* converter = sc_dpi_converter();
 a = a1.a;
 // convert from svLogic to sc_logic
 converter->sc_from_logic(b, a1.b);

 // convert from svBit to bool
 converter->sc_from_bit(c, a1.c);
 cerr << "mystruct:" << a << "," << b << "," << c << endl;
}
```

Struct contains only C data type fields.

Formal argument in SystemC is a C struct.

`sc_dpi_convert` class used to convert C types to SystemC types.

For example:

```
% irun testSC.cpp testSC.sv -sysc
```

produces the following results:

```
mystruct:10,Z,1
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X