SystemVerilog 3.1 Draft 6

Accellera's Extensions to Verilog®

Abstract: a set of extensions to the IEEE 1364-2001 Verilog Hardware Description Language to aid in the creation and verification of abstract architectural level models

SystemVerilog 3.1 Draft 6

Accellera's Extensions to Verilog®

Abstract: a set of extensions to the IEEE 1364-2001 Verilog Hardware Description Language to aid in the creation and verification of abstract architectural level models

This is a ballot draft for internal use within Accellera and the Accellera SystemVerilog committees only. Information within this document has not been approved by Accellera, and is subject to change.

1370 Trancas Street #163
Napa, CA 94558
Phone: (707) 251-9977
Fax: (707) 251-9877

Acknowledgements

This SystemVerilog Language Reference Manual was developed by experts from many different fields, including design and verification engineers, Electronic Design Automation (EDA) companies, EDA vendors, and members of the IEEE 1364 Verilog standard working group.

The SystemVerilog Language Reference Manual (LRM) was specified by the Accellera SystemVerilog committee. Four subcommittees worked on various aspects of the SystemVerilog 3.1 specification:

— The Basic Committee (SV-BC) worked on errata and clarification of the SystemVerilog 3.0 LRM.

— The Enhancement Committee (SV-EC) investigated and specified new modeling and testbench features.

— The Assertions Committee (SV-AC) specified the assertions constructs for SystemVerilog 3.1.

— The C Application Programming Interface (API) Committee (SV-CC) developed and specified the Direct Programming Interface (DPI), the assertions API and the coverage API for SystemVerilog.

The committee chairs were:

Vassilios Gerousis, SystemVerilog 3.0 and 3.1 Committee General Chair
Dave Kelf, SystemVerilog 3.0 Committee Co-Chair
Johny Srouji, SystemVerilog 3.1 Basic Committee Chair; Karen Pieper, Co-Chair
David Smith, SystemVerilog 3.1 Enhancement Committee Chair; Stefen Boyd, Co-Chair
Faisal Haque, SystemVerilog 3.1 Assertions Committee Chair; Steve Meier, Co-Chair
Swapnajit Mittra, SystemVerilog 3.1 C API Committee Chair; Ghassan Khoory, Co-Chair
Stuart Sutherland, SystemVerilog 3.0 and 3.1 Language Reference Manual Editor
Stefen Boyd, SystemVerilog 3.0 and 3.1 BNF Annex. Editor

Committee members included (listed alphabetically by last name):

SystemVerilog 3.0 Committee	**SystemVerilog 3.1 Basic Committee**	**SystemVerilog 3.1 Enhancement Committee**	**SystemVerilog 3.1 Assertions Committee**	**SystemVerilog 3.1 C API Committee**
Stefen Boyd*	Kevin Cameron	Stefen Boyd*	Roy Armoni	John Amouroux
Dennis Brophy	Cliff Cummings*	Dennis Brophy	Surrendra Dudani	Kevin Cameron
Kevin Cameron	Dan Jacobi	Michael Burns	Cindy Eisner	Joao Gaeda
Cliff Cummings*	Jay Lawrence	Kevin Cameron	Harry Foster	Ghassan Khoory
Simon Davidmann	Matt Maidment	Cliff Cummings*	Faisal Haque	Andrzej Litwiniuk
Tom Fitzpatrick*	Francoise Martinolle*	Peter Flake	John Havlicek	Francoise Martinole*
Peter Flake	Karen Pieper*	Jeff Freedman	Richard Ho	Swapnajit Mittra
Harry Foster	Brad Pierce	Neil Korpusik	Adam Krolnik*	Michael Rohleder
Vassilios Gerousis	David Rich	Jay Lawrence	David Lacey	John Stickley
Paul Graham	Steven Sharp*	Francoise Martinolle*	Joseph Lu	Stuart Swan
Dave Kelf	Johny Srouji	Don Mills	Erich Marschner	Bassam Tabbara
David Knapp*	Gord Vreugdenhil*	Mehdi Mohtashemi	Steve Meier	Kurt Takara
Adam Krolnik*		Phil Moorby	Prakash Narain	Doug Warmke
Mike McNamara*		Karen Pieper*	Andrew Seawright	
Phil Moorby		Brad Pierce	Bassam Tabbara	
Prakash Narian		Arturo Salz		
Anders Nordstrom*		David Smith		
Rajeev Ranjan		Stuart Sutherland*		
John Sanguinetti				
David Smith				
Alec Stanculescu*				
Stuart Sutherland*				
Bassam Tabbara				
Andy Tsay				

* indicates this person was also an active member of the IEEE 1364 Verilog Standard Working Group.

Table of Contents

Section 1 Introduction to SystemVerilog 1

Section 2 Literal Values 3
2.1 Introduction (informative) 3
2.2 Literal value syntax 3
2.3 Integer and logic literals 3
2.4 Real literals 4
2.5 Time literals 4
2.6 String literals 4
2.7 Array literals 5
2.8 Structure literals 5

Section 3 Data Types 6
3.1 Introduction (informative) 6
3.2 Data type syntax 7
3.3 Integer data types 7
3.4 Real and shortreal data types 8
3.5 Void data type 8
3.6 chandle data type 8
3.7 String data type 9
3.8 Event data type 13
3.9 User-defined types 14
3.10 Enumerations 15
3.11 Structures and unions 19
3.12 Class 21
3.13 Singular type 22
3.14 Casting 22
3.15 $cast dynamic casting 23

Section 4 Arrays 25
4.1 Introduction (informative) 25
4.2 Packed and unpacked arrays 25
4.3 Multiple dimensions 26
4.4 Indexing and slicing of arrays 27
4.5 Array querying functions 28
4.6 Dynamic arrays 28
4.7 Array assignment 29
4.8 Arrays as arguments 30
4.9 Associative arrays 31
4.10 Associative array methods 34
4.11 Associative array assignment 36
4.12 Associative array arguments 36
4.13 Associative array literals 36

Section 5 Data Declarations 38
5.1 Introduction (informative) 38
5.2 Data declaration syntax 38
5.3 Constants 38
5.4 Variables 39
5.5 Scope and lifetime 40
5.6 Nets, regs, and logic 41

5.7 Signal aliasing ... 42

Section 6 Attributes ... 44
6.1 Introduction (informative) ... 44
6.2 Default attribute type ... 44

Section 7 Operators and Expressions ... 45
7.1 Introduction (informative) ... 45
7.2 Operator syntax ... 45
7.3 Assignment operators ... 45
7.4 Operations on logic and bit types ... 46
7.5 Wild equality and wild inequality ... 46
7.6 Real operators ... 47
7.7 Size ... 47
7.8 Sign ... 47
7.9 Operator precedence and associativity ... 47
7.10 Built-in methods ... 48
7.11 Concatenation ... 49
7.12 Unpacked array expressions ... 49
7.13 Structure expressions ... 50
7.14 Aggregate expressions ... 52
7.15 Conditional operator ... 52

Section 8 Procedural Statements and Control Flow ... 53
8.1 Introduction (informative) ... 53
8.2 Statements ... 53
8.3 Blocking and nonblocking assignments ... 55
8.4 Selection statements ... 56
8.5 Loop statements ... 57
8.6 Jump statements ... 58
8.7 Final blocks ... 58
8.8 Named blocks and statement labels ... 59
8.9 Disable ... 60
8.10 Event control ... 61
8.11 Procedural assign and deassign removal ... 62

Section 9 Processes ... 63
9.1 Introduction (informative) ... 63
9.2 Combinational logic ... 63
9.3 Latched logic ... 64
9.4 Sequential logic ... 64
9.5 Continuous assignments ... 64
9.6 fork...join ... 64
9.7 Process execution threads ... 65
9.8 Process control ... 66

Section 10 Tasks and Functions ... 68
10.1 Introduction (informative) ... 68
10.2 Tasks ... 69
10.3 Functions ... 71
10.4 Task and function scope and lifetime ... 73
10.5 Task and function argument passing ... 73
10.6 Import and export functions ... 76

Section 11 Classes ... 78
11.1 Introduction (informative) ... 78
11.2 Syntax ... 78
11.3 Overview ... 79
11.4 Objects (class instance) ... 79
11.5 Object properties ... 80
11.6 Object methods ... 81
11.7 Constructors ... 81
11.8 Static properties ... 82
11.9 Static methods ... 82
11.10 This ... 82
11.11 Assignment, re-naming and copying ... 83
11.12 Inheritance and subclasses ... 84
11.13 Overridden members ... 85
11.14 Super ... 85
11.15 Casting ... 86
11.16 Chaining constructors ... 86
11.17 Data hiding and encapsulation ... 87
11.18 Constant Properties ... 87
11.19 Abstract classes and virtual methods ... 88
11.20 Polymorphism: dynamic method lookup ... 88
11.21 Class scope resolution operator :: ... 89
11.22 Out of block declarations ... 90
11.23 Parameterized classes ... 91
11.24 Typedef class ... 92
11.25 Classes, structures, and unions ... 92
11.26 Memory management ... 92

Section 12 Random Constraints ... 94
12.1 Introduction (informative) ... 94
12.2 Overview ... 94
12.3 Random variables ... 96
12.4 Constraint blocks ... 98
12.5 Randomization methods ... 106
12.6 In-line constraints — randomize() with ... 107
12.7 Disabling random variables with rand_mode() ... 108
12.8 Controlling constraints with constraint_mode() ... 109
12.9 Dynamic constraint modification ... 110
12.10 Random number system functions ... 111
12.11 Random stability ... 112
12.12 Manually seeding randomize ... 114

Section 13 Inter-Process Synchronization and Communication ... 115
13.1 Introduction (informative) ... 115
13.2 Semaphores ... 115
13.3 Mailboxes ... 116
13.4 Parameterized mailboxes ... 119
13.5 Event ... 120
13.6 Event sequencing: wait_order() ... 121
13.7 Event variables ... 122

Section 14 Scheduling Semantics ... 125
14.1 Execution of a hardware model and its verification environment ... 125
14.2 Event simulation ... 125

14.3 The stratified event scheduler ... 125
14.4 The PLI callback control points ... 129

Section 15 Clocking Domains ... 130
15.1 Introduction (informative) ... 130
15.2 Clocking domain declaration ... 130
15.3 Input and output skews ... 132
15.4 Hierarchical expressions ... 133
15.5 Signals in multiple clocking domains ... 133
15.6 Clocking domain scope and lifetime ... 133
15.7 Multiple clocking domains example ... 133
15.8 Interfaces and clocking domains ... 134
15.9 Clocking domain events ... 136
15.10 Cycle delay: ## ... 136
15.11 Default clocking ... 136
15.12 Input sampling ... 137
15.13 Synchronous events ... 138
15.14 Synchronous drives ... 138

Section 16 Program Block ... 141
16.1 Introduction (informative) ... 141
16.2 The program construct ... 141
16.3 Multiple programs ... 143
16.4 Eliminating testbench races ... 143
16.5 Blocking tasks in cycle/event mode ... 144
16.6 Program control tasks ... 144

Section 17 Assertions ... 145
17.1 Introduction (informative) ... 145
17.2 Immediate assertions ... 145
17.3 Concurrent assertions overview ... 147
17.4 Boolean expressions ... 148
17.5 Sequences ... 150
17.6 Declaring sequences ... 153
17.7 Sequence operations ... 155
17.8 Manipulating data in a sequence ... 171
17.9 System functions ... 174
17.10 The property definition ... 175
17.11 Multiple clock support ... 177
17.12 Concurrent assertions ... 179
17.13 Clock resolution ... 184
17.14 Binding properties to scopes or instances ... 187

Section 18 Hierarchy ... 190
18.1 Introduction (informative) ... 190
18.2 The $root top level ... 190
18.3 Module declarations ... 192
18.4 Nested modules ... 192
18.5 Port declarations ... 195
18.6 Time unit and precision ... 196
18.7 Module instances ... 197
18.8 Port connection rules ... 200
18.9 Name spaces ... 202
18.10 Hierarchical names ... 202

Section 19 Interfaces 203
19.1 Introduction (informative) 203
19.2 Interface syntax 204
19.3 Ports in interfaces 208
19.4 Modports 209
19.5 Tasks and functions in interfaces 212
19.6 Parameterized interfaces 218
19.7 Access without ports 220

Section 20 Parameters 222
20.1 Introduction (informative) 222
20.2 Parameter declaration syntax 222

Section 21 Configuration Libraries 224
21.1 Introduction (informative) 224
21.2 Libraries 224
21.3 Library map files 224

Section 22 System Tasks and System Functions 225
22.1 Introduction (informative) 225
22.2 Expression size system function 225
22.3 Shortreal conversions 225
22.4 Array querying system functions 226
22.5 Assertion severity system tasks 227
22.6 Assertion control system tasks 228
22.7 Assertion system functions 228
22.8 Random number system functions 229
22.9 Program control 229
22.10 Coverage system functions 229
22.11 Enhancements to Verilog-2001 system tasks 229
22.12 $readmemb and $readmemh 230

Section 23 VCD Data 231

Section 24 Compiler Directives 232
24.1 Introduction (informative) 232
24.2 ‘define macros 232
24.3 `include 233

Section 25 Features under consideration for removal from SystemVerilog 234
25.1 Introduction (informative) 234
25.2 Defparam statements 234
25.3 Procedural assign and deassign statements 234

Section 26 Direct Programming Interface (DPI) 236
26.1 Overview 236
26.2 Two layers of the DPI 237
26.3 Global name space of imported and exported functions 238
26.4 Imported functions 238
26.5 Calling imported functions 243
26.6 Exported functions 244

Section 27 SystemVerilog Assertion API 246
27.1 Requirements 246

27.2 Extensions to VPI enumerations ... 246
27.3 Static information ... 247
27.4 Dynamic information ... 250
27.5 Control functions ... 252

Section 28 SystemVerilog Coverage API ... 255
28.1 Requirements ... 255
28.2 SystemVerilog real-time coverage access ... 256
28.3 FSM recognition ... 261
28.4 VPI coverage extensions ... 263

Annex A Formal Syntax ... 267

Annex B Keywords ... 301

Annex C Linked Lists ... 303

Annex D DPI C-layer ... 309

Annex E Include files ... 334

Annex F Inclusion of Foreign Language Code ... 339

Annex G Glossary ... 343

Annex H Bibliography ... 345

Section 1
Introduction to SystemVerilog

This document specifies the Accellera extensions for a higher level of abstraction for modeling and verification with the Verilog Hardware Description Language. These additions extend Verilog into the systems space and the verification space and was built on top of the work of the IEEE Verilog 2001 committee.

Throughout this document:

— "Verilog" or "Verilog-2001" refers to the IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language

— "SystemVerilog" refers to the Accellera extensions to the Verilog-2001 standard.

This document numbers the generations of Verilog as follows:

— **"Verilog 1.0"** is the IEEE Std. 1364-1995 Verilog standard, which is also called Verilog-1995

— **"Verilog 2.0"** is the IEEE Std. 1364-2001 Verilog standard, commonly called Verilog-2001; this generation of Verilog contains the first significant enhancements to Verilog since its release to the public in 1990

— **"SystemVerilog 3.x"** is Verilog-2001 plus an extensive set of high-level abstraction extensions, as defined in this document

  — SystemVerilog 3.0, approved as an Accellera standard in June 2002, includes enhancements primarily directed at high-level architectural modeling

  — SystemVerilog 3.1, approved as an Accellera standard in **add final date** , includes enhancements primarily directed at advanced verification and C language integration

The Accellera initiative to extend Verilog is an ongoing effort under the direction of the Accellera HDL+ Technical Subcommittee. This committee will continue to define additional enhancements to Verilog beyond SystemVerilog 3.1.

SystemVerilog is built on top of Verilog 2001. SystemVerilog improves the productivity, readability, and reusability of Verilog based code. The language enhancements in SystemVerilog provide more concise hardware descriptions, while still providing an easy route with existing tools into current hardware implementation flows.

SystemVerilog 3.0 adds several new constructs to Verilog-2001, including:

— C data types to provide better encapsulation and compactness of code

  — int, typedef, struct, union, enum

— Enhancements to existing Verilog constructs, to provide tighter specifications

  — Extensions to always blocks to include linting type features

  — Logic (0, 1, X, Z) and bit (0, 1) data types

  — Automatic/static specification on a per variable instance basis

  — Procedural break, continue, return

— Interfaces to encapsulate communication and facilitate "Communication Oriented" design

— Dynamic processes for modeling pipelines

— A $root top level hierarchy which can have global definitions

SystemVerilog 3.1 adds verification enhancements in the following important areas:

— Verification Functionality: Reusable, reactive testbench data-types and functions.

  - Built-in types: string, associative array, and dynamic array.
  - Pass by reference subroutine arguments.
- Synchronization: Mechanisms for dynamic process creation, process control, and inter-process communication.
  - Enhancements to existing Verilog events.
  - Built-in synchronization primitives: Semaphore, Mailbox.
- Classes: Object-Oriented mechanism that provides abstraction, encapsulation, and safe pointer capabilities.
- Dynamic Memory: Automatic memory management in a re-entrant environment that frees users from explicit de-allocation.
- Cycle-Based Functionality: Clocking domains and cycle-based attributes that help reduce development, ease maintainability, and promote reusability.
  - Cycle-based signal drives and samples
  - Synchronous samples
  - Race-free program context

Assertion mechanism for verifying design intent and functional coverage intent.

- Property and sequence declarations
- Assertions and Coverage statements with action blocks.

Section 2
Literal Values

2.1 Introduction (informative)

The lexical conventions for SystemVerilog literal values are extensions of those for Verilog. SystemVerilog adds literal time values, literal array values, literal structures and enhancements to literal strings.

2.2 Literal value syntax

```
time_literal[7] ::=                                              // from Annex A.8.4
        unsigned_number time_unit
      | fixed_point_number time_unit
time_unit ::= s | ms | us | ns | ps | fs | step
number ::=                                                       // from Annex A.8.7
        decimal_number
      | octal_number
      | binary_number
      | hex_number
      | real_number
decimal_number ::=
        unsigned_number
      | [ size ] decimal_base  unsigned_number
      | [ size ] decimal_base  x_digit { _ }
      | [ size ] decimal_base  z_digit { _ }
binary_number ::= [ size ] binary_base  binary_value
octal_number ::= [ size ] octal_base  octal_value
hex_number ::= [ size ] hex_base  hex_value
sign ::= + | -
size ::= non_zero_unsigned_number
non_zero_unsigned_number[1] ::= non_zero_decimal_digit { _ | decimal_digit}
real_number[1] ::=
        fixed_point_number
      | unsigned_number [ . unsigned_number ] exp [ sign ] unsigned_number
fixed_point_number[1] ::= unsigned_number . unsigned_number
exp ::= e | E
unsigned_number[1] ::= decimal_digit { _ | decimal_digit }
string_literal ::= " { Any_ASCII_Characters } "                  // from Annex A.8.8
```

Syntax 2-1—Literal values (excerpt from Annex A)

2.3 Integer and logic literals

Literal integer and logic values can be sized or unsized, and follow the same rules for signedness, truncation and left-extending as Verilog-2001.

SystemVerilog adds the ability to specify unsized literal single bit values with a preceding apostrophe (**'**), but without the base specifier. All bits of the unsized value are set to the value of the specified bit. In a self-determined context these literals have a width of 1 bit, and the value is treated as unsigned.

```
'0, '1, 'X, 'x, 'Z, 'z     // sets all bits to this value
```

2.4 Real literals

The default type is **real** for fixed point format (e.g. `1.2`), and exponent format (e.g. `2.0e10`).

A cast can be used to convert literal **real** values to the **shortreal** type (e.g., `shortreal'(1.2)`). Casting is described in Section 3.14.

2.5 Time literals

Time is written in integer or fixed point format, followed without a space by a time unit (**fs ps ns us ms s step**). For example:

```
0.1ns
40ps
```

The time literal is interpreted as a **realtime** value scaled to the current time unit and rounded to the current time precision. Note that if a time literal is used as an actual parameter to a module or interface instance, the current time unit and precision are those of the module or interface instance.

2.6 String literals

A string literal is enclosed in quotes and has its own data type. Non-printing and other special characters are preceded with a backslash. SystemVerilog adds the following special string characters:

\v vertical tab
\f form feed
\a bell
\x02 hex number

A string literal must be contained in a single line unless the new line is immediately preceded by a **** (back slash). In this case, the back slash and the new line are ignored. There is no predefined limit to the length of a string literal.

A string literal can be assigned to a character, or a packed array, as in Verilog-2001. If the size differs, it is right justified.

```
byte c1 = "A" ; bit [7:0] d = "\n" ;
bit [0:11] [7:0] c2 = "hello world\n" ;
```

A string literal can be assigned to an unpacked array of characters, and a zero termination is added like in C. If the size differs, it is left justified.

```
byte c3 [0:12] = "hello world\n" ;
```

Packed and unpacked arrays are discussed in Section 4. The difference between string literals and array literals is discussed in Section 2.7, which follows.

String literals can also be cast to a packed or unpacked array, which shall follow the same rules as assigning a literal string to a packed or unpacked array. Casting is discussed in Section 3.14.

SystemVerilog 3.1 also includes a **string** data type to which a string literal can be assigned. Variables of type

string have arbitrary length; they are dynamically resized to hold any string. String literals are packed arrays (of a width that is a multiple of 8 bits), and they are implicitly converted to the string type when assigned to a string type or used in an expression involving string type operands (see Section 3.7).

2.7 Array literals

Array literals are syntactically similar to C initializers, but with the replicate operator ({{}}) allowed.

```
int n[1:2][1:3] = {{0,1,2},{3{4}}};
```

The nesting of braces must follow the number of dimensions, unlike in C. However, replicate operators can be nested.

```
int n[1:2][1:3] = {2{{3{4}}}};
```

If the type is not given by the context, it must be specified with a cast.

```
typedef int [1:3] triple; // 3 integers packed together
b = triple'{0,1,2};
```

2.8 Structure literals

Structure literals are syntactically similar to C initializers. Structure literals must have a type, either from context or a cast.

```
typedef struct {int a; shortreal b;} ab;
ab c;
c = {0, 0.0}; // structure literal type determined from the left hand context
(c)
```

Nested braces should reflect the structure. For example:

```
ab abarr[1:0] = {{1, 1.0}, {2, 2.0}};
```

Note that the C alternative `{1, 1.0, 2, 2.0}` is not allowed.

Structure literals can also use member name and value, or data type and default value (see Section 7.13):

```
c = {a:0, b:0.0};               // member name and value for that member
c = {default:0};                // all elements of structure c are set to 0
d = ab'{int:1, shortreal:1.0};  // data type and default value for all members
                                // of that type
```

When an array of structures is initialized, the nested braces should reflect the array and the structure. For example:

```
ab abarr[1:0] = {{1, 1.0}, {2, 2.0}};
```


Section 3
Data Types

3.1 Introduction (informative)

To provide for clear translation to and from C, SystemVerilog supports the C built-in types, with the meaning given by the implementation C compiler. However, to avoid the duplication of **`int`** and **`long`** without causing confusion, in SystemVerilog, **`int`** is 32 bits and **`longint`** is 64 bits. The C **`float`** type is called **`shortreal`** in SystemVerilog, so that it is not be confused with the Verilog-2001 **`real`** type.

Verilog-2001 has net data types, which can have 0, 1, X or Z, plus 7 strengths, giving 120 values. It also has variable data types such as **`reg`**, which have 4 values 0, 1, X, Z. These are not just different data types, they are used differently. SystemVerilog adds another 4-value data type, called **`logic`** (see Sections 3.3.2 and 5.6).

SystemVerilog 3.1 adds string, **`chandle`** and **`class`** data types, and enhances the Verilog **`event`** and SystemVerilog 3.0 **`enum`** data types. SystemVerilog 3.1 also extends the user defined types by providing support for object-oriented class.

Verilog-2001 provides arbitrary fixed length arithmetic using **`reg`** data types. The **`reg`** type can have bits at X or Z, however, and so are less efficient than an array of bits, because the operator evaluation must check for X and Z, and twice as much data must be stored. SystemVerilog adds a **`bit`** type which can only have bits with 0 or 1 values. See Section 3.3.2 on 2-state data types.

Automatic type conversions from a smaller number of bits to a larger number of bits involve zero extensions if unsigned or sign extensions if signed, and do not cause warning messages. Automatic truncation from a larger number of bits to a smaller number does cause a warning message. Automatic conversions between **`logic`** and **`bit`** do not cause warning messages. To convert a logic value to a bit, 1 converts to 1, anything else to 0.

User defined types are introduced by **`typedef`** and must be defined before they are used. Data types can also be parameters to modules or interfaces, making them like class templates in object-oriented programming. One routine can be written to reverse the order of elements in any array, which is impossible in C and in Verilog.

Structures and unions are complicated in C, because the tags have a separate name space. SystemVerilog follows the C syntax, but without the optional structure tags.

See also Section 4 on arrays.

3.2 Data type syntax

```
data_type ::=                                                    // from Annex A.2.2.1
        integer_vector_type [ signing ] { packed_dimension } [ range ]
      | integer_atom_type [ signing ]
      | type_declaration_identifier { packed_dimension }
      | non_integer_type
      | struct packed [ signing ] { { struct_union_member } } { packed_dimension }
      | union packed [ signing ] { { struct_union_member } } { packed_dimension }
      | struct [ signing ] { { struct_union_member } }
      | union [ signing ] { { struct_union_member } }
      | enum [ integer_type [ signing ] { packed_dimension } ]
          { enum_identifier [ = constant_expression ] { , enum_identifier [ = constant_expression ] } }
      | string
      | event
      | chandle
      | class_scope_type_identifier
class_scope_type_identifier::=
        class_identifier :: { class_identifier :: } type_declaration_identifier
      | class_identifier :: { class_identifier :: } class_identifier
integer_type ::= integer_vector_type | integer_atom_type
integer_atom_type ::= byte | shortint | int | longint | integer
integer_vector_type ::= bit | logic | reg
non_integer_type ::= time | shortreal | real | realtime
net_type ::= supply0 | supply1 | tri | triand | trior | tri0 | tri1 | wire | wand | wor
signing ::= signed | unsigned
simple_type ::= integer_type | non_integer_type | type_identifier
struct_union_member ::= { attribute_instance } data_type  list_of_variable_identifiers_or_assignments ;
variable_decl_assignment ::=                                     // from Annex A.2.4
        variable_identifier [ variable_dimension ] [ = constant_expression ]
      | variable_identifier [ ] = new [ constant_expression ] [ ( variable_identifier ) ]
      | class_identifier [ parameter_value_assignment ] = new [ ( list_of_arguments ) ]
```

Syntax 3-1—data types (excerpt from Annex A)

3.3 Integer data types

SystemVerilog offers several integer data types, representing a hybrid of both Verilog and C data types:

Table 3-1: Integer data types

shortint	2-state SystemVerilog data type, 16 bit signed integer
int	2-state SystemVerilog data type, 32 bit signed integer
longint	2-state SystemVerilog data type, 64 bit signed integer
byte	2-state SystemVerilog data type, 8 bit signed integer or ASCII character
bit	2-state SystemVerilog data type, user-defined vector size

Table 3-1: Integer data types

logic	4-state SystemVerilog data type, user-defined vector size with different use rules from reg
reg	4-state Verilog-2001 data type, user-defined vector size
integer	4-state Verilog-2001 data type, at least 32 bit signed integer
time	4-state Verilog-2001 data type, 64-bit integer

3.3.1 Integral types

The term integral is used throughout this document to refer to the data types that can represent a single basic integer data type, **packed array**, **packed struct**, **packed union**, **enum**, or **time**.

3.3.2 2-state (two-value) and 4-state (four-value) data types

Types that can have unknown and high-impedance values are called 4-state types. These are **logic**, **reg**, **integer** and **time**. The other types do not have unknown values and are called 2-state types, for example **bit** and **int**.

The difference between **int** and **integer** is that **int** is 2-state logic and **integer** is 4-state logic. 4-state values have additional bits that encode the X and Z states. 2-state data types can simulate faster, take less memory, and are preferred in some design styles.

3.3.3 Signed and unsigned data types

Integer types use integer arithmetic and can be signed or unsigned. This affects the meaning of certain operators such as '<', etc.

```
int unsigned ui;
int signed si;
```

The data types **byte**, **shortint**, **int**, **integer** and **longint** default to **signed**. The data types **bit**, **reg** and **logic** default to **unsigned**, as do arrays of these types.

Note that the **signed** keyword is part of Verilog-2001. The **unsigned** keyword is a reserved keyword in Verilog-2001, but is not utilized.

See also Section 7, on operators and expressions.

3.4 Real and shortreal data types

The **real**[1] data type is from Verilog-2001, and is the same as a C **double**. The **shortreal** data type is a SystemVerilog data type, and is the same as a C **float**.

3.5 Void data type

The **void** data type represents non-existent data. This type can be specified as the return type of functions, indicating no return value.

3.6 chandle data type

The **chandle** data type represents storage for pointers passed using the DPI Direct Programming Interface

[1] The real and shortreal types are represented as described by IEEE 734-1985, an IEEE standard for floating point numbers.

(see Section 26). The size of this type is platform dependent, but shall be at least large enough to hold a pointer on the machine in which the tool is running.

The syntax to declare a handle is as follows:

```
chandle variable_name ;
```

where *variable_name* is a valid identifier. Chandles shall always be initialized to the value **null**, which has a value of 0 on the C side. Chandles are very restricted in their usage, with the only legal uses being as follows:

— Only the following operators are valid on **chandle** variables:
  — Equality (**==**), inequality (**!=**) with another **chandle** or with **null**
  — Case equality (**===**), case inequality (**!==**) with another **chandle** or with **null** (same semantics as **==** and **!=**)
— Can be tested for a boolean value that shall be 0 if the **chandle** is **null** and 1 otherwise
— Only the following assignments can be made to a **chandle**
  — Assignment from another **chandle**
  — Assignment to **null**
— Chandles can be inserted into associative arrays (refer to Section 4.9), but the relative ordering of any two entries in such an associative array can vary, even between successive runs of the same tool.
— Chandles can be used within a class
— Chandles can be passed as arguments to functions or tasks
— Chandles can be returned from functions

The use of chandles is restricted as follows:

— Ports shall not have the **chandle** data type
— Chandles shall not be assigned to variables of any other type
— Chandles shall not be used:
  — In any expression other than as permitted above
  — As ports
  — In sensitivity lists or event expressions
  — In continuous assignments
  — In unions
  — In packed types

3.7 String data type

SystemVerilog includes a **string** data type, which is a variable size, dynamically allocated array of characters. SystemVerilog also includes a number of special methods to work with strings.

Verilog supports string literals, but only at the lexical level. In Verilog, string literals behave like packed arrays of a width that is a multiple of 8 bits. A string literal assigned to a packed array is truncated to the size of the array

In SystemVerilog string literals behave exactly the same as in Verilog However, SystemVerilog also supports

the **string** data type to which a string literal can be assigned. When using the **string** data type instead of a packed array, strings can be of arbitrary length and no truncation occurs. Literal strings are implicitly converted to the **string** type when assigned to a **string** type or used in an expression involving **string** type operands.

Variables of type **string** can be indexed from 0 to N-1 (the last element of the array), and they can take on the special value "", which is the empty string.

The syntax to declare a **string** is:

```
string variable_name [= initial_value];
```

where *variable_name* is a valid identifier and the optional *initial_value* can be a string literal or the value "" for an empty string. For example:

```
string myName = "John Smith";
```

If an initial value is not specified in the declaration, the variable is initialized to "", the empty string.

SystemVerilog provides a set of operators that can be used to manipulate combinations of string variables and string literals. The basic operators defined on the string data type are listed in Table 3-2, which follows.

A string literal can be assigned to a **string**, a character, or a packed array. If their size differs the literal is right justified and zero filled on the left. For example:

```
byte c = "A";                 // assign to c "A"
bit [10:0] a = "\x41";        // assigns to a 'b000_0100_0001
bit [1:4][7:0] h = "hello" ;  // assigns to h "ello"
```

A **string**, string literal, or packed array can be assigned to a **string** variable. The **string** variable shall grow to accommodate the packed array. If the size (in bits) of the packed array is not a multiple of 8, then the packed array is zero filled on the left.

For example:

```
string s1 = "hello";          // sets s1 to "hello"
bit [11:0] b = 12'ha41;
string s2 = b;                // sets s2 to 'h0a41
```

As a second example:

```
reg [15:0] r;
integer i = 1;
string b = "";
string a = {"Hi", b};

r = a;              // OK
b = r;              // OK (implicit cast, implementations can issue a warning)
b = "Hi";           // OK
b = {5{"Hi"}};      // OK
a = {i{"Hi"}};      // OK (non constant replication)
r = {i{"Hi"}};      // invalid (non constant replication)
a = {i{b}};         // OK
a = {a,b};          // OK
a = {"Hi",b};       // OK
a[0] = "h";         // OK same as a[0] = "hi" )
```

Table 3-2: String operators

Operator	Semantics
`Str1 == Str2`	Equality. Checks if the two strings are equal. Result is 1 if they are equal and 0 if they are not. Both strings can be of type **`string`**. Or one of them can be a string literal. If both operands are string literals, the expression is the same Verilog equality operator for integer types. The special value `" "` is allowed.
`Str1 != Str2`	Inequality. Logical Negation of ==

`Str1 < Str2`
`Str1 <= Str2`
`Str1 > Str2`
`Str1 >= Str2`
	Comparison. Relational operators return 1 if the corresponding condition is true using the lexicographical ordering of the two strings `Str1` and `Str2`. The comparison behaves like the ANSI C `strcmp` function (or the compare string method). Both operands can be of type **`string`**, or one of them can be a string literal.

`{Str1,Str2,...,Strn}`	Concatenation. Each operand can be of type **`string`** or a string literal (it shall be implicitly converted to type **`string`**). If at least one operand is of type **`string`**, then the expression evaluates to the concatenated string and is of type **`string`**. If all the operands are string literals, then the expression behaves like a Verilog concatenation of integral types; if the result is then used in an expression involving **`string`** types, it is implicitly converted to the **`string`** type.
`{multiplier{Str}}`	Replication. `Str` can be of type **`string`** or a string literal. Multiplier must be of integral type and can be non-constant. If multiplier is non-constant or **`Str`** is of type **`string`**, the result is a string containing N concatenated copies of `Str`, where `N` is specified by the multiplier. If `Str` is a literal and the multiplier is constant, the expression behaves like numeric replication in Verilog (if the result is used in another expression involving string types, it is implicitly converted to the string type).
`Str.method(...)`	The dot (**`.`**) operator is used to invoke a specified method on strings.

SystemVerilog also includes a number of special methods to work with strings. These methods use the built-in method notation. These methods are described in the following subsections.

3.7.1 len()

```
function int len()
```

— `str.len()` returns the length of the string, i.e., the number of characters in the string (excluding any terminating character).

— If `str` is "", then `str.len()` returns 0.

3.7.2 putc()

```
task putc(int i, string s)
task putc(int i, byte c)
```

— `str.putc(i, c)` replaces the *i*th character in *str* with the given integral value.

— `str.putc(i, s)` replaces the *i*th character in *str* with the first character in *s*.

— `s` can be any expression that can be assigned to a string.

— `putc` does not change the size of `str`: If `i` < 0 or `i` >= `str.len()`, then *str* is unchanged.

Note: `str.putc(j, x)` is identical to `str[j] = x`.

3.7.3 getc()

```
function int getc(int i)
```

— `str.getc(i)` returns the ASCII code of the `i`th character in `str`.

— If `i` < 0 or `i` >= `str.len()`, then `str.getc(i)` returns 0.

Note: `x = str.getc(j)` is identical to `x = str[j]`.

3.7.4 toupper()

```
function string toupper()
```

— `str.toupper()` returns a string with characters in `str` converted to uppercase.

— `str` is unchanged.

3.7.5 tolower()

```
function string tolower()
```

— `str.tolower()` returns a string with characters in `str` converted to lowercase.

— `str` is unchanged.

3.7.6 compare()

```
function int compare(string s)
```

— `str.compare(s)` compares `str` and `s`, as in the ANSI C `strcmp` function, with a compatible return value.

See the relational string operators in Section 3.7, Table 3-2.

3.7.7 icompare()

```
function int icompare(string s)
```

— `str.icompare(s)` compares `str` and `s`, like the ANSI C `strcmp` function, with a compatible return value, but the comparison is case insensitive.

3.7.8 substr()

```
function string substr(int i, int j)
```

— `str.substr(i, j)` returns a new string that is a sub-string formed by characters in position `i` through `j` of `str`.

— If `i` < 0, `j` < `i`, or `j` >= `str.len()`, `substr()` returns " " (the empty string).

3.7.9 atoi(), atohex(), atooct(), atobin()

```
function integer atoi()
function integer atohex()
function integer atooct()
function integer atobin()
```

— `str.atoi()` returns the integer corresponding to the ASCII decimal representation in `str`. For example:

```
str = "123";
int i = str.atoi();  // assigns 123 to i.
```

The string is converted until the first non-digit is encountered.

— `atohex` interprets the string as hexadecimal.

— `atooct` interprets the string as octal.

— `atobin` interprets the string as binary.

3.7.10 atoreal()

```
function real atoreal()
```

— `str.atoreal()` returns the real number corresponding to the ASCII decimal representation in `str`.

3.7.11 itoa()

```
task itoa(integer i)
```

— `str.itoa(i)` stores the ASCII decimal representation of `i` into `str` (inverse of `atoi`).

3.7.12 hextoa()

```
task hextoa(integer i)
```

— `str.hextoa(i)` stores the ASCII hexadecimal representation of `i` into `str` (inverse of `atohex`).

3.7.13 octtoa()

```
task octtoa(integer i)
```

— `str.octtoa(i)` stores the ASCII octal representation of `i` into `str` (inverse of `atooct`).

3.7.14 bintoa()

```
task bintoa(integer i)
```

— `str.bintoa(i)` stores the ASCII binary representation of `i` into `str` (inverse of `atobin`).

3.7.15 realtoa()

```
task realtoa(real r)
```

— `str.realtoa(r)` stores the ASCII real representation of `i` into `str` (inverse of `atoreal`).

3.8 Event data type

The **`event`** data type is an enhancement over Verilog named events. SystemVerilog events provide a handle to a synchronization object. Like Verilog, event variables can be explicitly triggered and waited for. Furthermore, SystemVerilog events have a persistent triggered state that lasts for the duration of the entire time step. In addition, an event variable can be assigned another event variable or the special value **`null`**. When assigned another event variable, both event variables refer to the same synchronization object. When assigned **`null`**, the association between the synchronization object and the event variable is broken. Events can be passed as arguments to tasks.

The syntax to declare an **`event`** is:

```
event variable_name [= initial_value];
```

Where *variable_name* is a valid identifier and the optional *initial_value* can be another event variable or the special value **`null`**.

If an initial value is not specified then the variable is initialized to a new synchronization object.

If the event is assigned **null**, the event becomes nonblocking, as if it were permanently triggered.

Examples:

```
event done;                // declare a new event called done
event done_too = done;     // declare done_too as alias to done
event empty = null;        // event variable with no synchronization object
```

Event operations and semantics are discussed in detail in Section 13.5.

3.9 User-defined types

```
type_declaration ::=                                                    // from Annex A.2.1.3
        typedef [ data_type ] type_declaration_identifier ;
      | typedef hierarchical_identifier . type_identifier type_declaration_identifier ;
      | typedef [ class ] class_identifier ;
      | typedef class_identifier [ parameter_value_assignment ] type_declaration_identifier ;
```

Syntax 3-2—user-defined types (excerpt from Annex A)

The user can define a new type using **typedef**, as in C.

```
typedef int intP;
```

This can then be instantiated as:

```
intP a, b;
```

A type can be used before it is defined, provided it is first identified as a type by an empty **typedef**:

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

Note that this does not apply to enumeration values, which must be defined before they are used.

If the type is defined within an interface, it must be re-defined locally before being used.

```
interface it;
   typedef int intP;
endinterface

it it1 ();
typedef it1.intP intP;
```

User-defined type names must be used for complex data types in casting (see Section 3.14, below), and as parameters.

Sometimes a user defined type needs to be declared before the contents of the type has been defined. This is of use with user defined types derived from **enum**, **struct**, **union**, and **class**. For an example, see Section 11.24. Support for this is provided by the following forms for **typedef**:

```
typedef enum type_declaration_identifier;
typedef struct type_declaration_identifier;
typedef union type_declaration_identifier;
typedef class type_declaration_identifier;
```

```
typedef type_declaration_identifier;
```

Note that, while this is useful for coupled definitions of classes as shown in Section 11.24, it cannot be used for coupled definitions of structures, since structures are statically declared and there is no support for pointers to structures.

The last form shows that the type of the user defined type does not have to be defined in the forward declaration.

A **typedef** inside a **generate** shall not define the actual type of a forward definition that exists outside the scope of the forward definition.

3.10 Enumerations

```
data_type ::=                                                    // from Annex A.2.2.1
        ...
      | enum [ integer_type [ signing ] { packed_dimension } ]
          { enum_identifier [ = constant_expression ] { , enum_identifier [ = constant_expression ] } }
```

Syntax 3-3—enumerated types (excerpt from Annex A)

An enumerated type declares a set of integral named constants. Enumerated data types provide the capability to abstractly declare strongly typed variables without either a data type or data value(s) and later add the required data type and value(s) for designs that require more definition. Enumerated data types also can be easily referenced or displayed using the enumerated names as opposed to the enumerated values.

In the absence of a data type declaration, the default data type shall be **int**. Any other data type used with enumerated types shall require an explicit data type declaration.

An enumerated type defines a set of named values. In the following example, `light1` and `light2` are defined to be variables of the anonymous (unnamed) enumerated int type that includes the three members: `red`, `yellow` and `green`.

```
enum {red, yellow, green} light1, light2; // anonymous int type
```

An enumerated name with x or z assignments assigned to an **enum** with no explicit data type or an explicit 2-state declaration shall be a syntax error.

```
// Syntax error: IDLE=2'b00, XX=2'bx <ERROR>, S1=2'b01??, S2=2'b10??
enum {IDLE, XX='x, S1=2'b01, S2=2'b10} state, next;
```

An **enum** declaration of a 4-state type, such as integer, that includes one or more names with x or z assignments shall be permitted.

```
// Correct: IDLE=2'b00, XX=2'bx, S1=2'b01, S2=2'b10
enum integer {IDLE, XX='x, S1=2'b01, S2=2'b10} state, next;
```

An unassigned enumerated name that follows an enum name with x or z assignments shall be a syntax error.

```
// Syntax error: IDLE=2'b00, XX=2'bx, S1=??, S2=??
enum integer {IDLE, XX='x, S1, S2} state, next;
```

The values can be cast to integer types, and increment from an initial value of 0. This can be overridden.

```
enum {bronze=3, silver, gold} medal; // silver=4, gold=5
```


The values can be set for some of the names and not set for other names. A name without a value is automatically assigned an increment of the value of the previous name.

```
// c is automatically assigned the increment-value of 8
enum {a=3, b=7, c} alphabet;
```

If an automatically incremented value is assigned elsewhere in the same enumeration, this shall be a syntax error.

```
// Syntax error: c and d are both assigned 8
enum {a=0, b=7, c, d=8} alphabet;
```

If the first name is not assigned a value, it is given the initial value of 0.

```
// a=0, b=7, c=8
enum {a, b=7, c} alphabet;
```

A sized constant can be used to set the size of the type. All sizes must be the same.

```
// silver=4'h4, gold=4'h5 (all are 4 bits wide)
enum {bronze=4'h3, silver, gold} medal4;

// Syntax error: the width of the enum has been exceeded
// in both of these examples
      enum {a=1'b0, b, c} alphabet;
      enum [0:0] {a,b,c} alphabet;
```

Any enumeration encoding value that is outside the representable range of the **enum** shall be an error.

Adding a constant range to the **enum** declaration can be used to set the size of the type. If any of the enum members are defined with a different sized constant, this shall be a syntax error.

```
// Error in the bronze and gold member declarations
enum bit [3:0] {bronze=5'h13, silver, gold=3'h5} medal4;

// Correct declaration - bronze and gold sizes are redundant
enum bit [3:0] {bronze=4'h13, silver, gold=4'h5} medal4;
```

Type checking of enumerated types used in assignments, as arguments and with operators is covered in Section 3.10.3. Like C, there is no overloading of literals, so medal and medal4 cannot be defined in the same scope, since they contain the same names.

3.10.1 Defining new data types as enumerated types

A type name can be given so that the same type can be used in many places.

```
typedef enum {NO, YES} boolean;
boolean myvar; // named type
```

3.10.2 Enumerated type ranges

A range of enumeration elements can be specified automatically, via the following syntax:

Table 3-3: Enumeration element ranges

`name`	Associates the next consecutive number with name.
`name = N`	Assigns the constant N to name

Table 3-3: Enumeration element ranges

`name[N]`	Generates N names in the sequence: name0, name1, ..., nameN-1N must be a constant expression
`name[N:M]`	Creates a sequence of names starting with nameN and incrementing or decrementing until reaching name nameM.

For example:

```
enum { add=10, sub[5], jmp[6:8] } ;
```

This example assigns the number 10 to the enumerated type `add`. It also creates the enumerated types `sub0`,`sub1`,`sub2`,`sub3`,and `sub4`, and assigns them the values 11..15, respectively. Finally, the example creates the enumerated types `jmp6`,`jmp7`, and `jmp8`, and assigns them the values 16-18, respectively.

3.10.3 Type checking

SystemVerilog enumerated types are strongly typed, thus, a variable of type **`enum`** cannot be directly assigned a value that lies outside the enumeration set. This is a powerful type-checking aid that prevents users from accidentally assigning nonexistent values to variables of an enumerate type. This restriction only applies to an enumeration that is explicitly declared as a type. The enumeration values can still be used as constants in expressions, and the results can be assigned to any variable of a compatible integral type.

Both the enumeration names and their integer values must be unique. The values can be set to any integral constant value, or auto-incremented from an initial value of 0. It is an error to set two values to the same name, or to set a value to the same auto-incremented value.

Enumerated variables are type-checked in assignments, arguments, and relational operators. Enumerated variables are auto-cast into integral values, but, assignment of arbitrary expressions to an enumerated variable requires an explicit cast.

For example:

```
typedef enum { red, green, blue, yellow, white, black } Colors;
```

This operation assigns a unique number to each of the color identifiers, and creates the new data type Colors. This type can then be used to create variables of that type.

```
Colors c;
c = green;
c = 1;            // Invalid assignment
if ( 1 == c )     // OK. c is auto-cast to integer
```

In the example above, the value `green` is assigned to the variable `c` of type `Colors`. The second assignment is invalid because of the strict typing rules enforced by enumerated types.

Casting can be used to perform an assignment of a different data type, or an out of range value, to an enumerated type. Casting is discussed in Sections 3.10.4, 3.14 and 3.15.

3.10.4 Enumerated types in numerical expressions

Elements of enumerated type variables can be used in numerical expressions. The value used in the expression is the numerical value associated with the enumerated value. For example:

```
typedef enum { red, green, blue, yellow, white, black } Colors;

Colors col;
integer a, b;
```



```
a = blue * 3;
col = yellow;
b = col + green;
```

From the previous declaration, `blue` has the numerical value 2. This example assigns `a` the value of 6 (2*3). Next, it assigns `b` a value of 4 (3+1).

An enum variable or identifier used as part of an expression is automatically cast to the base type of the **`enum`** declaration (either explicitly or using int as the default). An assignment to an enum variable from an expression other than an enum variable or identifier of the same type shall require a cast. Casting to an **`enum`** type shall cause a conversion of the expression to its base type without checking the validity of the value (unless a dynamic cast is used as described in Section 3.15).

```
typedef enum {Red, Green, Blue} Colors;
typedef enum {Mo,Tu,We,Th,Fr,Sa,Su} Week;
Colors C;
Week W;
int I;

C = Colors'(C+1);             // C is converted to an integer, then added to
                              // one, then converted back to a Colors type

C = C + 1; C++; C+=2; C = I;  // Illegal because they would all be
                              // assignments of expressions without a cast

C = Colors'(Su);              // Legal; puts an out of range value into C

I = C + W;                    // Legal; C and W are automatically cast to int
```

SystemVerilog includes a set of specialized methods to enable iterating over the values of enumerated types.

3.10.4.1 first()

The prototype for the `first()` method is:

```
function enum first();
```

The `first()` method returns the value of the first member of the enumeration **`enum`**.

3.10.4.2 last()

The prototype for the `last()` method is:

```
function enum last();
```

The `last()` method returns the value of the last member of the enumeration **`enum`**.

3.10.4.3 next()

The prototype for the `next()` method is:

```
function enum next( int unsigned N = 1 );
```

The `next()` method returns the Nth next enumeration value (default is the next one) starting from the current value of the given variable. A wrap to the start of the enumeration occurs when the end of the enumeration is reached. If the given value is not a member of the enumeration, the `next()` method returns the first member.

3.10.4.4 prev()

The prototype for the `prev()` method is:

```
function enum prev( int unsigned N = 1 );
```

The `prev()` method returns the Nth previous enumeration value (default is the previous one) starting from the current value of the given variable. A wrap to the end of the enumeration occurs when the start of the enumeration is reached. If the given value is not a member of the enumeration, the `prev()` method returns the last member.

3.10.4.5 num()

The prototype for the `num()` method is:

```
function int num();
```

The `num()` method returns the number of elements in the given enumeration.

3.10.4.6 name()

The prototype for the name() method is:

```
function string name();
```

The `name()` method returns the string representation of the given enumeration value. If the given value is not a member of the enumeration, the `name()` method returns the empty string.

3.10.4.7 Using enumerated type methods

The following code fragment shows how to display the name and value of all the members of an enumeration.

```
typedef enum { red, green, blue, yellow } Colors;
Colors c = c.first;
forever begin
   $display( "%s : %d\n", c.name, c );
   if( c == c.last ) break;
   c = c.next;
end
```

3.11 Structures and unions

data_type ::= *// from Annex A.2.2.1*
... | **struct packed** [signing] **{** { struct_union_member } **}** { packed_dimension } | **union packed** [signing] **{** { struct_union_member } **}** { packed_dimension } | **struct** [signing] **{** { struct_union_member } **}** | **union** [signing] **{** { struct_union_member } **}**

struct_union_member ::= { attribute_instance } data_type list_of_variable_identifiers_or_assignments **;**

Syntax 3-4—Structures and unions (excerpt from Annex A)

Structure and union declarations follow the C syntax, but without the optional structure tags before the '{'.

```
struct { bit [7:0] opcode; bit [23:0] addr; }IR;// anonymous structure
```



```
                                                    // defines variable IR
   IR.opcode = 1; // set field in IR.
```

Some additional examples of declaring structure and unions are:

```
typedef struct {
            bit [7:0] opcode;
            bit [23:0] addr;
} instruction; // named structure type
instruction IR; // define variable

typedef union { int i; shortreal f; } num; // named union type
   num n;
n.f = 0.0; // set n in floating point format

typedef struct {
            bit isfloat;
            union { int i; shortreal f; } n; // anonymous type
} tagged; // named structure

tagged a[9:0]; // array of structures
```

A structure can be assigned as a whole, and passed to or from a function or task as a whole.

Section 2.8 discusses assigning initial values to a structure.

A packed structure consists of bit fields, which are packed together in memory without gaps. This means that they are easily converted to and from bit vectors. An unpacked structure has an implementation-dependent packing, normally matching the C compiler.

Like a packed array, a packed structure can be used as a whole with arithmetic and logical operators. The first member specified is the most significant and subsequent members follow in decreasing significance. The structures are declared using the **`packed`** keyword, which can be followed by the **`signed`** or **`unsigned`** keywords, according to the desired arithmetic behavior. The default is unsigned:

```
struct packed signed {
   int a;
   shortint b;
   byte c;
   bit [7:0] d;
} pack1; // signed, 2-state

struct packed unsigned {
   time a;
   integer b;
   logic [31:0] c;
} pack2; // unsigned, 4-state
```

If any data type within a packed structure is 4-state, the whole structure is treated as 4-state. Any 2-state members are converted as if cast. One or more elements of the packed array can be selected, assuming an `[n-1:0]` numbering:

```
pack1 [15:8] // c
```

Non-integer data types, such as **`real`** and **`shortreal`**, are not allowed in packed structures or unions. Nor are unpacked arrays.

A packed structure can be used with a **`typedef`**.

```
typedef struct packed { // default unsigned
   bit [3:0] GFC;
   bit [7:0] VPI;
   bit [11:0] VCI;
   bit CLP;
   bit [3:0] PT ;
   bit [7:0] HEC;
   bit [47:0] [7:0] Payload;
   bit [2:0] filler;
} s_atmcell;
```

A packed union shall contain members that must be packed structures, or packed arrays or integer data types all of the same size (in contrast to an unpacked union, where the members can be different sizes). This ensures that you can read back a union member that was written as another member. A packed union can also be used as a whole with arithmetic and logical operators, and its behavior is determined by the signed or unsigned keyword, the latter being the default. If a packed union contains a 2-state member and a 4-state member, the entire union is 4 state. There is an implicit conversion from 4-state to 2-state when reading and from 2-state to 4-state when writing the 2-state bit member.

For example, a union can be accessible with different access widths:

```
typedef union packed { // default unsigned
   s_atmcell acell;
   bit [423:0] bit_slice;
   bit [52:0][7:0] byte_slice;
} u_atmcell;

u_atmcell u1;
byte b; bit [3:0] nib;
b = u1.bit_slice[415:408]; // same as b = u1.byte_slice[51];
nib = u1.bit_slice [423:420]; // same as nib = u1.acell.GFC;
```

Note that writing one member and reading another is independent of the byte ordering of the machine, unlike a normal union of normal structures, which are C-compatible and have members in ascending address order.

3.12 Class

A *class* is a collection of data and a set of subroutines that operate on that data. The data in a class is referred to as properties, and its subroutines are called methods. The properties and methods, taken together, define the contents and capabilities of a class instance or object.

class_declaration ::= *// from Annex A.1.3*
{ attribute_instance } [**virtual**] **class** [lifetime] class_identifier [parameter_port_list]
[**extends** class_identifier] **;** [timeunits_declaration] { class_item }
endclass [**:** class_identifier]

Syntax 3-5—Classes (excerpt from Annex A)

The object-oriented class extension allows objects to be created and destroyed dynamically. Class instances, or objects, can be passed around via object handles, which add a safe-pointer capability to the language. An object can be declared as an argument of type **`input`**, **`output`**, **`inout`**, or, **`ref`**. In each case, the argument copied is the object handle, not the contents of the object.

A Class is declared using the **`class`**...**`endclass`** keywords. For example:

```
class Packet
```



```
    int address;         // Properties are address, data, and crc
    bit [63:0] data;
    shortint crc;
    Packet next;         // Handle to another Packet

    function new();      // Methods are send and new
    function bit send();
endclass : Packet
```

Any data type can be declared as a class member. Classes are discussed in more detail in Section 11.

3.13 Singular type

A *singular* type includes packed arrays (and structures) and all other data types except unpacked structures, unpacked arrays, and chandles.

3.14 Casting

```
constant_primary ::=                                          // from Annex A.8.4
            ...
          | casting_type ' ( constant_expression )
          | casting_type ' constant_concatenation
          | casting_type ' constant_multiple_concatenation
primary ::=
            ...
          | casting_type ' ( expression )
          | void ' ( function_call )
          | casting_type ' concatenation
          | casting_type ' multiple_concatenation
casting_type ::= simple_type | number | signing               // from Annex A.2.2.1
simple_type ::= integer_type | non_integer_type | type_identifier
```

Syntax 3-6—casting (excerpt from Annex A)

A data type can be changed by using a cast (') operation. The expression to be cast must be enclosed in parenthesis or within concatenation or replication braces.

```
int'(2.0 * 3.0)
shortint'{8'hFA,8'hCE}
```

A decimal number as a data type means a number of bits.

```
17'(x - 2)
```

The signedness can also be changed.

```
signed'(x)
```

A user-defined type can be used.

```
mytype'(foo)
```

When casting to a predefined type, the prefix of the cast must be the predefined type keyword. When casting to

a user-defined type, the prefix of the cast must be the user-defined type identifier.

When a **shortreal** is converted to an **int** or to 32 bits, its value is rounded, as in Verilog. Therefore, the conversion can lose information. To convert a **shortreal** to its underlying bit representation without a loss of information, use $shortrealtobits as defined in Section 22.3. To convert from the bit representation of a shortreal value into a **shortreal**, use $bitstoshortreal as defined in Section 22.3.

Structures can be converted to bits preserving the bit pattern, which means they can be converted back to the same value without any loss of information. The following example demonstrates this conversion. In the example, the $bits attribute gives the size of a structure in bits (the $bits system function is discussed in Section 22.2:

```
typedef struct {
            bit isfloat;
            union { int i; shortreal f; } n; // anonymous type
} tagged; // named structure

typedef bit [$bits(tagged) - 1 : 0] tagbits; // tagged defined above

tagged a [7:0]; // unpacked array of structures

tagbits t = tagbits'(a[3]); // convert structure to array of bits
a[4] = tagged'(t); // convert array of bits back to structure
```

Note that the **bit** data type loses X values. If these are to be preserved, the **logic** type should be used instead.

The size of a union in bits is the size of its largest member. The size of a **logic** in bits is 1.

For compatibility, the Verilog functions $itor, $rtoi, $bitstoreal, $realtobits, $signed, $unsigned can also be used.

3.15 $cast dynamic casting

SystemVerilog provides the $cast system task to assign values to variables that might not ordinarily be valid because of differing data type. $cast can be called as either a task or a function.

The syntax for $cast is:

```
function int $cast( singular dest_var, singular source_exp );
```

or

```
task $cast( singular dest_var, singular source_exp );
```

The *dest_var* is the variable to which the assignment is made.

The *source_exp* is the expression that is to be assigned to the destination variable.

Use of $cast as either a task or a function determines how invalid assignments are handled.

When called as a task, $cast attempts to assign the source expression to the destination variable. If the assignment is invalid, a runtime error occurs and the destination variable is left unchanged.

When called as a function, $cast attempts to assign the source expression to the destination variable, and returns 1 if the cast is legal. If the cast fails, the function does not make the assignment and returns 0. When called as a function, no runtime error occurs, and the destination variable is left unchanged.

It's important to note that $cast performs a run-time check. No type checking is done by the compiler, except to check that the destination variable and source expression are singulars.

For example:

```
typedef enum { red, green, blue, yellow, white, black } Colors;
Colors col;
$cast( col, 2 + 3 );
```

This example assigns the expression `(5 => black)` to the enumerated type. Without `$cast`, this type of assignment is illegal.

The following example shows how to use the $cast to check if an assignment will succeed:

```
if ( ! $cast( col, 2 + 8 ) )     // 10: invalid cast
   $display( "Error in cast" );
```

Alternatively, the preceding examples can be cast using a static SystemVerilog cast operation: For example:

```
col = Colors'(2 + 3);
```

However, this is a compile-time cast, i.e, a coercion that always succeeds at run-time, and does not provide for error checking or warn if the expression lies outside the enumeration values.

Allowing both types of casts gives full control to the user. If users know that it is safe to assign certain expressions to an enumerated variable, the faster static compile-time cast can be used. If users need to check if the expression lies within the enumeration values, it is not necessary to write a lengthy switch statement manually, the compiler automatically provides that functionality via the `$cast` function. By allowing both types of casts, users can control the time/safety trade-offs.

Note: `$cast` is similar to the `dynamic_cast` function available in C++, but, `$cast` allows users to check if the operation will succeed, whereas `dynamic_cast` always raises a C++ exception.

Section 4
Arrays

4.1 Introduction (informative)

An array is a collection of variables, all of the same type, and accessed using the same name plus one or more indices.

In C, arrays are indexed from 0 by integers, or converted to pointers. Although the whole array can be initialized, each element must be read or written separately in procedural statements.

In Verilog-2001, arrays are indexed from left-bound to right-bound. If they are vectors, they can be assigned as a single unit, but not if they are arrays. Verilog-2001 allows multiple dimensions.

In Verilog-2001, all data types can be declared as arrays. The **reg**, **wire** and all other net types can also have a vector width declared. A dimension declared before the object name is referred to as the "vector width" dimension. The dimensions declared after the object name are referred to as the "array" dimensions.

```
reg [7:0] r1 [1:256];   // [7:0] is the vector width, [1:256] is the array size
```

SystemVerilog enhances array declarations in several ways. SystemVerilog supports fixed-size arrays, dynamic arrays, and associative arrays. Fixed-size arrays can be multi-dimensional and have fixed storage allocated for all the elements of the array. Dynamic arrays also allocate storage for all the elements of the array, but the array size can be changed dynamically. Dynamic and associative arrays are one-dimensional. Fixed-size and dynamic arrays are indexed using integer expressions, while associative arrays can be indexed using arbitrary data types. Associative arrays do not have any storage allocated until it is needed, which makes them ideal for dealing with sparse data.

4.2 Packed and unpacked arrays

SystemVerilog uses the term "***packed array***" to refer to the dimensions declared before the object name (what Verilog-2001 refers to as the vector width). The term "***unpacked array***" is used to refer to the dimensions declared after the object name.

```
bit [7:0] c1;         // packed array
real u [7:0];         // unpacked array
```

A packed array is a mechanism for subdividing a vector into subfields which can be conveniently accessed as array elements. Consequently, a packed array is guaranteed to be represented as a contiguous set of bits. An unpacked array may or may not be so represented. A packed array differs from an unpacked array in that, when a packed array appears as a primary, it is treated as a single vector.

If a packed array is declared as signed, then the array viewed as a single vector shall be signed. A part-select of a packed array shall be unsigned.

Packed arrays allow arbitrary length integer types, so a 48 bit integer can be made up of 48 bits. These integers can then be used for 48 bit arithmetic. The maximum size of a packed array can be limited, but shall be at least 65536 (2^{16}) bits.

Packed arrays can only be made of the single bit types (**bit**, **logic**, **reg**, **wire**, and the other net types) and recursively other packed arrays and packed structures.

Integer types with predefined widths cannot have packed array dimensions declared. These types are: **byte**, **shortint**, **int**, **longint**, and **integer**. An integer type with a predefined width can be treated as a single dimension packed array. The packed dimensions of these integer types shall be numbered down to 0, such that the right-most index is 0.


```
byte c2;    // same as bit [7:0] c2;
integer i1; // same as logic signed [31:0] i1;
```

Unpacked arrays can be made of any type. SystemVerilog enhances fixed-size unpacked arrays in that in addition to all other variable types, unpacked arrays can also be made of object handles (see Section 11.4) and events (see Section 13.5).

SystemVerilog accepts a single number, as an alternative to a range, to specify the size of an unpacked array, like C. That is, `[size]` becomes the same as `[size-1:0]`. For example:

`int Array[8][32];` is the same as: `int Array[7:0][31:0];`

The following operations can be performed on all arrays, packed or unpacked. The examples provided with these rules assume that A and B are arrays of the same shape and type.

— Reading and writing the array, e.g., `A = B`

— Reading and writing a slice of the array, e.g., `A[i:j] = B[i:j]`

— Reading and writing a variable slice of the array, e.g., `A[x+:c] = B[y+:c]`

— Reading and writing an element of the array, e.g., `A[i] = B[i]`

— Equality operations on the array or slice of the array, e.g. `A==B, A[i:j] != B[i:j]`

The following operations can be performed on packed arrays, but not on unpacked arrays. The examples provided with these rules assume that A is an array.

— Assignment from an integer, e.g., `A = 8'b11111111;`

— Treatment as an integer in an expression, e.g., `(A + 3)`

When assigning to an unpacked array, the source and target must be arrays with the same number of unpacked dimensions, and the length of each dimension must be the same. Assignment to an unpacked array is done by assigning each element of the source unpacked array to the corresponding element of the target unpacked array. Note that an element of an unpacked array can be a packed array.

For the purposes of assignment, a packed array is treated as a vector. Any vector expression can be assigned to any packed array. The packed array bounds of the target packed array do not affect the assignment. A packed array cannot be assigned to an unpacked array.

4.3 Multiple dimensions

Like Verilog memories, the dimensions following the type set the packed size. The dimensions following the instance set the unpacked size.

```
bit [3:0] [7:0] joe [1:10]; // 10 entries of 4 bytes (packed into 32 bits)
```

can be used as follows:

```
joe[9] = joe[8] + 1; // 4 byte add
joe[7][3:2] = joe[6][1:0]; // 2 byte copy
```

Note that the dimensions declared following the type and before the name (`[3:0][7:0]` in the preceding declaration) vary more rapidly than the dimensions following the name (`[1:10]` in the preceding declaration). When used, the first dimensions (`[3:0]`) follow the second dimensions (`[1:10]`).

In a list of dimensions, the right-most one varies most rapidly, as in C. However a packed dimension varies more rapidly than an unpacked one.

```
bit [1:10] foo1 [1:5];  // 1 to 10 varies most rapidly; compatible with
                           Verilog-2001 arrays
```

```
bit foo2 [1:5] [1:10];  // 1 to 10 varies most rapidly, compatible with C

bit [1:5] [1:10] foo3;  // 1 to 10 varies most rapidly

bit [1:5] [1:6] foo4 [1:7] [1:8];   // 1 to 6 varies most rapidly, followed by
                                       1 to 5, then 1 to 8 and then 1 to 7
```

Multiple packed dimensions can also be defined in stages with **typedef**.

```
typedef bit [1:5] bsix;
bsix [1:10] foo5; // 1 to 5 varies most rapidly
```

Multiple unpacked dimensions can also be defined in stages with **typedef**.

```
typedef bsix mem_type [0:3];  // array of four 'bsix' elements
mem_type bar [0:7];           // array of eight 'mem_type' elements
```

When the array is used with a smaller number of dimensions, these have to be the slowest varying ones.

```
bit [9:0] foo6;
foo5 = foo1[2]; // a 10 bit quantity.
```

As in Verilog-2001, a comma-separated list of array declarations can be made. All arrays in the list shall have the same data type and the same packed array dimensions.

```
bit [7:0] [31:0] foo7 [1:5] [1:10], foo8 [0:255]; // two arrays declared
```

If an index expression is of a 4-state type, and the array is of a 4-state type, an X or Z in the index expression shall cause a read to return X, and a write to issue a run-time warning. If an index expression is of a 4-state type, but the array is of a 2-state type, an X or Z in the index expression shall generate a run-time warning and be treated as 0. If an index expression is out of bounds, a run-time warning can be generated.

Out of range index values shall be illegal for both reading from and writing to an array of 2-state variables, such as **int**. The result of an out of range index value is indeterminate. Implementations shall generate a warning if an out of range index occurs for a read or write operation.

4.4 Indexing and slicing of arrays

An expression can select part of a packed array, or any integer type, which is assumed to be numbered down to 0.

SystemVerilog uses the term "*part select*" to refer to a selection of one or more contiguous bits of a single dimension packed array. This is consistent with the usage of the term "*part select*" in Verilog.

```
reg [63:0] data;
reg [7:0] byte2;
byte2 = data[23:16]; // an 8-bit part select from data
```

SystemVerilog uses the term "*slice*" to refer to a selection of one or more contiguous elements of an array. Verilog only permits a single element of an array to be selected, and does not have a term for this selection.

An single element of a packed or unpacked array can be selected using an indexed name.

```
bit [3:0] [7:0] j;   // j is a packed array
byte k;
k = j[2]; // select a single 8-bit element from j
```

One or more contiguous elements can be selected using a slice name. A slice name of a packed array is a packed array. A slice name of an unpacked array is an unpacked array.


```
bit busA [7:0] [31:0] ;    // unpacked array of 8 32-bit vectors
int busB [1:0];            // unpacked array of 2 integers
busB = busA[7:6];          // select a slice from busA
```

The size of the part select or slice must be constant, but the position can be variable. The syntax of Verilog-2001 is used.

```
int i = bitvec[j +: k];    // k must be constant.
int a[x:y], b[y:z], e;
a = {b[c -: d], e};        // d must be constant
```

Slices of an array can only apply to one dimension, but other dimensions can have single index values in an expression.

4.5 Array querying functions

SystemVerilog provides new system functions to return information about an array. These are: **`$left`**, **`$right`**, **`$low`**, **`$high`**, **`$increment`**, **`$length`**, and **`$dimensions`**. These functions are described in Section 22.4.

4.6 Dynamic arrays

Dynamic arrays are one-dimensional arrays whose size can be set or changed at runtime. The space for a dynamic array doesn't exist until the array is explicitly created at runtime.

The syntax to declare a dynamic array is:

```
data_type array_name [];
```

where `data_type` is the data type of the array elements. Dynamic arrays support the same types as fixed-size arrays.

For example:

```
bit [3:0] nibble[];     // Dynamic array of 4-bit vectors
integer mem[];          // Dynamic array of integers
```

The `new[]` operator is used to set or change the size of the array.

The `size()` built-in method returns the current size of the array.

The `delete()` built-in method clears all the elements yielding an empty array (zero size).

4.6.1 new[]

The built-in function `new` allocates the storage and initializes the newly allocated array elements either to their default initial value or to the values provided by the optional argument.

The prototype of the `new` function is:

```
array_identifier = new[size] [(src_array)];
```

`size`:

The number of elements in the array. Must be a non-negative integral expression.

`src_array`:

Optional. The name of an array with which to initialize the new array. If `src_array` is not specified, the elements of `array_name` are initialized to their default value. `src_array` must be a dynamic array of the same data type as `array_name`, but it need not have the same size. If the size of `src_array` is less than `size`, the extra elements of `array_name` shall be initialized to their default value. If the size of `src_array` is greater than `size`, the additional elements of `src_array` shall be ignored.

This argument is useful when growing or shrinking an existing array. In this situation, `src_array` is `array_name`, so the previous values of the array elements are preserved. For example:

```
integer addr[];   // Declare the dynamic array.
addr = new[100];  // Create a 100-element array.
...
      // Double the array size, preserving previous values.
addr = new[200](addr);
```

The `new` operator follows the SystemVerilog precedence rules. Since both the square brackets `[]` and the parenthesis `()` have the same precedence, the arguments to this operator are evaluated left to right: `size` first, and `src_array` second.

4.6.2 size()

The prototype for the `size()` method is:

```
function int size();
```

The `size()` method returns the current size of a dynamic array, or zero if the array has not been created.

```
int j = addr.size;
addr = new[ addr.size() * 4 ] (addr);  // quadruple addr array
```

Note: The `size` method is equivalent to `$length( addr, 1 )`.

4.6.3 delete()

The prototype for the `delete()` method is:

```
function void delete();
```

The `delete()` method empties the array, resulting in a zero-sized array.

```
int ab [] = new[ N ];         // create a temporary array of size N
// use ab
ab.delete;                    // delete the array contents
$display( "%d", ab.size );    // prints 0
```

4.7 Array assignment

Assigning to a fixed-size unpacked array requires that the source and the target both be arrays with the same number of unpacked dimensions, and the length of each dimension be the same. Assignment is done by assigning each element of the source array to the corresponding element of the target array, which requires that the source and target arrays be of compatible types. Compatible types are types that are assignment compatible.Assigning fixed-size unpacked arrays of unequal size to one another shall result in a type check error.

```
int A[10:1];      // fixed-size array of 10 elements
int B[0:9];       // fixed-size array of 10 elements
int C[24:1];      // fixed-size array of 24 elements

A = B;            // ok. Compatible type and same size
```



```
A = C;            // type check error: different sizes
```

An array of wires can be assigned to an array of variables having the same number of unpacked dimensions and the same length for each of those dimensions, and vice-versa.

```
wire [31:0] W [9:0];
assign W = A;
initial #10 B = W;
```

A dynamic array can be assigned to a one-dimensional fixed-size array of a compatible type, if the size of the dynamic array is the same as the length of the fixed-size array dimension. Unlike assigning with a fixed-size array, this operation requires a run-time check that can result in an error.

```
int A[100:1];           // fixed-size array of 100 elements
int B[] = new[100];     // dynamic array of 100 elements
int C[] = new[8];       // dynamic array of 8 elements

A = B;                  // OK. Compatible type and same size
A = C;                  // type check error: different sizes
```

A dynamic array or a one-dimensional fixed-size array can be assigned to a dynamic array of a compatible type. In this case, the assignment creates a new dynamic array with a size equal to the length of the fixed-size array. For example:

```
int A[100:1];           // fixed-size array of 100 elements
int B[];                // empty dynamic array
int C[] = new[8];       // dynamic array of size 8

B = A;                  // ok. B has 100 elements
B = C;                  // ok. B has 8 elements
```

The last statement above is equivalent to:

```
B = new[ C.size ] (C);
```

Similarly, the source of an assignment can be a complex expression involving array slices or concatenations. For example:

```
string d[1:5] = { "a", "b", "c", "d", "e" };
string p[];
p = { d[1:3], "hello", d[4:5] };
```

The preceding example creates the dynamic array `p` with contents: “`a`”, “`b`”, “`c`”, “`hello`”, “`d`”, “`e`”.

4.8 Arrays as arguments

Arrays can be passed as arguments to tasks or functions. The rules that govern array argument passing by value are the same as for array assignment (see Section 10.5) are the same as for array assignment. When an array argument is passed by value, a copy of the array is passed to the called task or function. This is true for all array types: fixed-size, dynamic, or associative.

Passing fixed-size arrays as arguments to subroutines requires that the actual argument and the formal argument in the function declaration be of the compatible type and that all dimensions be of the same size.

For example, the declaration:

```
task fun(int a[3:1][3:1]);
```

declares task `fun` that takes one argument, a two dimensional array with each dimension of size three. A call to `fun` must pass a two dimensional array and with the same dimension size 3 for all the dimensions. For example, given the above description for `fun`, consider the following actuals:

```
int b[3:1][3:1];  //OK: same type, dimension, and size

int b[1:3][0:2];  //OK: same type, dimension, & size (different ranges)

reg b[3:1][3:1];  //error: incompatible type

int b[3:1];       //error: incompatible number of dimensions

int b[3:1][4:1];  //error: incompatible size (3 vs. 4)
```

A subroutine that accepts a one-dimensional fixed-size array can also be passed a dynamic array of a compatible type of the same size.

For example, the declaration:

```
task bar( string arr[4:1] );
```

declares a task that accepts one argument, an array of 4 strings. This task can accept the following actual arguments:

```
string b[4:1];       //OK: same type and size
string b[5:2];       //OK: same type and size (different range)
string b[] = new[4]; //OK: same type and size, requires run-time check
```

A subroutine that accepts a dynamic array can be passed a dynamic array of a compatible type or a one-dimensional fixed-size array of a compatible type

For example, the declaration:

```
task foo( string arr[] );
```

declares a task that accepts one argument, a dynamic array of 4 strings. This task can accept any one-dimensional array of strings or any dynamic array of strings.

4.9 Associative arrays

Dynamic arrays are useful for dealing with contiguous collections of variables whose number changes dynamically. When the size of the collection is unknown or the data space is sparse, an associative array is a better option. Associative arrays do not have any storage allocated until it is used, and the index expression is not restricted to integral expressions, but can be of any type.

An associative array implements a lookup table of the elements of its declared type. The data type to be used as an index serves as the lookup key, and imposes an ordering.

The syntax to declare an associative array is:

```
data_type array_id [ index_type ];
```

where:

— *data_type* is the data type of the array elements. Can be any type allowed for fixed-size arrays.

— *array_id* is the name of the array being declared.

— *index_type* is the data-type to be used as an index, or `*`. If `*` is specified, then the array is indexed by any integral expression of arbitrary size. An index type restricts the indexing expressions to a particular type.

Examples of associative array declarations are:

```
integer i_array[*];           // associative array of integer (unspecified
                              // index)

bit [20:0] array_b[string];   // associative array of 21-bit vector, indexed
                              // by string

event ev_array[myClass];      // associative array of event indexed by class
                              // myClass
```

Array elements in associative arrays are allocated dynamically; an entry is created the first time it is written. The associative array maintains the entries that have been assigned values and their relative order according to the index data type.

4.9.1 Wildcard index type

Example: `int array_name [*];`

Associative arrays that specify a wildcard index type have the following properties:

— The array can be indexed by any integral data type. Since the indices can be of different sizes, the same numerical value can have multiple representations, each of a different size. SystemVerilog resolves this ambiguity by detecting the number of leading zeros and computing a unique length and representation for every value.

— Non-integral index types are illegal and result in a type check error.

— A 4-state Index containing X or Z is invalid.

— Indices are unsigned.

— Indexing expressions are self-determined; signed indices are not sign extended.

— A string literal index is auto-cast to a bit-vector of equivalent size.

— The ordering is numerical (smallest to largest).

4.9.2 String index

Example: `int array_name [ string ];`

Associative arrays that specify a string index have the following properties:

— Indices can be strings or string literals of any length. Other types are illegal and shall result in a type check error.

— An empty string “” index is valid.

— The ordering is lexicographical (lesser to greater).

4.9.3 Class index

Example: `int array_name [ some_Class ];`

Associative arrays that specify a class index have the following properties:

— Indices can be objects of that particular type or derived from that type. Any other type is illegal and shall result in a type check error.

— A null index is invalid.

— The ordering is deterministic but arbitrary.

4.9.4 Integer (or int) index

Example: `int array_name [ integer ];`

Associative arrays that specify an integer index have the following properties:

— Indices can be any integral expression.

— Indices are signed.

— A 4-state index containing X or Z is invalid.

— Indices smaller than integer are sign extended to 32 bits.

— Indices larger than integer are truncated to 32 bits.

— The ordering is signed numerical.

4.9.5 Signed packed array

Example:
```
typedef bit signed [4:1] Nibble;
int array_name [ Nibble ];
```

Associative arrays that specify a signed packed array index have the following properties:

— Indices can be any integral expression.

— Indices are signed.

— Indices smaller than the size of the index type are sign extended.

— Indices larger than the size of the index type are truncated to the size of the index type.

— The ordering is signed numerical.

4.9.6 Unsigned packed array or packed struct

Example:
```
typedef bit [4:1] Nibble;
int array_name [ Nibble ];
```

Associative arrays that specify an unsigned packed array index have the following properties:

— Indices can be any integral expression.

— Indices are unsigned.

— A 4-state Index containing X or Z is invalid.

— Indices smaller than the size of the index type are zero filled.

— Indices larger than the size of the index type are truncated to the size of the index type.

— The ordering is numerical.

If an invalid index (i.e., 4-state expression has X’s) is used during a read operation or an attempt is made to read a non-existent entry then a warning is issued and the default initial value for the array type is returned, as shown in the table below:

Table 4-1: Value read from a nonexistent associative array entry

Type of Array	Value Read
4-state integral type	’X

Table 4-1: Value read from a nonexistent associative array entry

2-state integral type	’0
enumeration	first element in the enumeration
string	“”
class	null
event	null

If an invalid index is used during a write operation, the write is ignored and a warning is issued.

4.10 Associative array methods

In addition to the indexing operators, several built-in methods are provided that allow users to analyze and manipulate associative arrays, as well as iterate over its indices or keys.

4.10.1 num()

The syntax for the `num()` method is:

```
function int num();
```

The `num()` method returns the number of entries in the associative array. If the array is empty, it returns 0.

```
int imem[*];
imem[ 2’b3 ] = 1;
imem[ 16’hffff ] = 2;
imem[ 4b’1000 ] = 3;
$display( "%0d entries\n", imem.num ); // prints "3 entries"
```

4.10.2 delete()

The syntax for the `delete()` method is:

```
function void delete( [input index] );
```

Where *index* is an optional index of the appropriate type for the array in question.

If the *index* is specified, then the `delete()` method removes the entry at the specified index. If the entry to be deleted does not exist, the method issues no warning.

If the *index* is not specified, then the `delete()` method removes all the elements in the array.

```
int map[ string ];
map[ "hello" ] = 1;
map[ "sad" ] = 2;
map[ "world" ] = 3;
map.delete( "sad" );    // remove entry whose index is "sad" from "map"
map.delete;             // remove all entries from the associative array "map"
```

4.10.3 exists()

The syntax for the `exists()` method is:

```
function int exists( input index );
```

Where *index* is an index of the appropriate type for the array in question.

The `exists()` function checks if an element exists at the specified index within the given array. It returns 1 if the element exists, otherwise it returns 0.

```
if ( map.exists( "hello" ))
   map[ "hello" ] += 1;
else
   map[ "hello" ] = 0;
```

4.10.4 first()

The syntax for the `first()` method is:

```
function int first( ref index );
```

Where *index* is an index of the appropriate type for the array in question.

The `first()` method assigns to the given index variable the value of the first (smallest) index in the associative array. It returns 0 if the array is empty, and 1 otherwise.

```
string s;
if ( map.first( s ) )
   $display( "First entry is : map[ %s ] = %0d\n", s, map[s] );
```

4.10.5 last()

The syntax for the `last()` method is:

```
function int last( ref index );
```

Where *index* is an index of the appropriate type for the array in question.

The `last()` method assigns to the given index variable the value of the last (largest) index in the associative array. It returns 0 if the array is empty, and 1 otherwise.

```
string s;
if ( map.last( s ) )
   $display( "Last entry is : map[ %s ] = %0d\n", s, map[s] );
```

4.10.6 next()

The syntax for the `next()` method is:

```
function int next( ref index );
```

Where *index* is an index of the appropriate type for the array in question.

The `next()` method finds the entry whose index is greater than the given index. If there is a next entry, the index variable is assigned the index of the next entry, and the function returns 1. Otherwise, index is unchanged, and the function returns 0.

```
string s;
if ( map.first( s ) )
   do
      $display( "%s : %d\n", s, map[ s ] );
   while ( map.next( s ) );
```


4.10.7 prev()

The syntax for the `prev()` method is:

```
function int prev( ref index );
```

Where *index* is an index of the appropriate type for the array in question.

The `prev()` function finds the entry whose index is smaller than the given index. If there is a previous entry, the index variable is assigned the index of the previous entry, and the function returns 1. Otherwise, the index is unchanged, and the function returns 0.

```
string s;
if ( map.last( s ) )
   do
      $display( "%s : %d\n", s, map[ s ] );
   while ( map.prev( s ) );
```

If the argument passed to any of the four associative array traversal methods `first`, `last`, `next`, and `prev` is smaller than the size of the corresponding index, then the function returns –1 and shall copy only as much data as can fit into the argument. For example:

```
string   aa[*];
byte     ix;
int      status;
aa[ 1000 ] = "a";
status = aa.first( ix );
   // status is -1
   // ix is 232 (least significant 8 bits of 1000)
```

4.11 Associative array assignment

Associative arrays can be assigned only to another associative array of a compatible type and with the same index type. Other types of arrays cannot be assigned to an associative array, nor can associative arrays be assigned to other types of arrays, whether fixed-size or dynamic.

Assigning an associative array to another associative array causes the target array to be cleared of any existing entries, and then each entry in the source array is copied into the target array.

4.12 Associative array arguments

Associative arrays can be passed as arguments only to associative arrays of a compatible type and with the same index type. Other types of arrays, whether fixed-size or dynamic, cannot be passed to subroutines that accept an associative array as an argument. Likewise, associative arrays cannot be passed to subroutines that accept other types of arrays.

Passing an associative array by value causes a local copy of the associative array to be created.

4.13 Associative array literals

Associative array literals use the `{index:value}` syntax with an optional default index. Like all other arrays, an associative array can be written one entry at a time, or the whole array contents can be replaced using an array literal.

```
constant_primary ::=                                                    // from Annex A.8.1
concatenation ::=
            ...
          | { array_member_label : expression { , array_member_label : expression } }
array_member_label ::=
            default
          | type_identifier
          | constant_expression
```

Syntax 4-7—Associative array literal syntax (excerpt from Annex A)

For example:

```
// an associative array of strings indexed by 2-state integers,
// default is "foo".
string words [int] = {default: "foo"};

// an associative array of 4-state integers indexed by strings, default is -1.
integer table [string] = {"Peter":20, "Paul":22, "Mary":23, default:-1 };
```

If a default value is specified, then reading a non-existent element shall yield the specified default value. Otherwise, the default initial value is as described in Table 4-1 shall be returned.

Section 5
Data Declarations

5.1 Introduction (informative)

There are several forms of data in SystemVerilog: literals (see Section 2), parameters (see Section 20), constants, variables, nets, and attributes (see Section 6)

C constants are either literals, macros or enumerations. There is also a **`const`**, keyword but it is not enforced in C.

Verilog 2001 constants are literals, parameters, localparams and specparams. Verilog 2001 also has variables and nets. Variables must be written by procedural statements, and nets must be written by continuous assignments or ports.

SystemVerilog extends the functionality of variables by allowing them to either be written by procedural statements or driven by a single continuous assignment, similar to a **`wire`**. Since the keyword **`reg`** no longer describes the users intent in many cases, the keyword **`logic`** is added as a more accurate description that is equivalent to **`reg`**. Verilog-2001 has already deprecated the use of the term *register* in favor of *variable*.

SystemVerilog follows Verilog by requiring data to be declared before it is used, apart from implicit nets. The rules for implicit nets are the same as in Verilog-2001.

A variable can be static (storage allocated on instantiation and never de-allocated) or automatic (stack storage allocated on entry to a task, function or named block and de-allocated on exit). C has the keywords **`static`** and **`auto`**. SystemVerilog follows Verilog in respect of the static default storage class, with automatic tasks and functions, but allows **`static`** to override a default of **`automatic`** for a particular variable in such tasks and functions.

5.2 Data declaration syntax

```
data_declaration ::=                                          // from Annex A.2.1.3
        variable_declaration
      | constant_declaration
      | type_declaration
block_variable_declaration ::=
        [ lifetime ] data_type  list_of_variable_identifiers ;
      | lifetime data_type  list_of_variable_decl_assignments ;
variable_declaration ::=
        [ lifetime ] data_type  list_of_variable_identifiers_or_assignments ;
lifetime ::= static | automatic
```

Syntax 5-1—Data declaration syntax (excerpt from Annex A)

5.3 Constants

Constants are named data items which never change. There are three kinds of constants, declared with the keywords **`localparam`**, **`specparam`** and **`const`**, respectively. All three can be initialized with a literal.

```
localparam byte colon1 = ":" ;
specparam int delay = 10 ; // specparams are used for specify blocks
const logic flag = 1 ;
```

A parameter or local parameter can only be set to an expression of literals, parameters or local parameters, genvars, or a constant function of these. Hierarchical names are not allowed.

A specparam can also be set to an expression containing one or more specparams.

A constant declared with the **const** keyword, can only be set to an expression of literals, parameters, local parameters, genvars, a constant function of these, or other constants. The parameters, local parameters or constant functions can have hierarchical names. This is because the static constants are calculated after elaboration.

```
const logic option = a.b.c ;
```

A constant expression contains literals and other named constants.

An instance of a class (an object handle) can also be declared with the **const** keyword.

```
const class_name object = new(5,3);
```

This means that the object acts like a variable that cannot be written. The arguments to the new method must be constant expressions.

SystemVerilog enhancements to **parameter** constant declarations are presented in Section 20. SystemVerilog does not change **localparam** and **specparam** constants declarations. A **const** form of constant differs from a **localparam** constant in that the **localparam** must be set during elaboration, whereas a **const** can be set during simulation, such as in an automatic task.

5.4 Variables

A variable declaration consists of a data type followed by one or more instances.

```
shortint s1, s2[0:9];
```

A variable can be declared with an initializer, which must be a constant expression.

```
int i = 0;
```

In Verilog-2001, an initialization value specified as part of the declaration is executed as if the assignment were made from an initial block, after simulation has started. Therefore, the initialization can cause an event on that variable at simulation time zero.

In SystemVerilog, setting the initial value of a static variable as part of the variable declaration (including static class members) shall occur before any **initial** or **always** blocks are started, and so does not generate an event. If an event is needed, an **initial** block should be used to assign the initial values.

Initial values in SystemVerilog are not constrained to simple constants; they can include run-time expressions, including dynamic memory allocation. For example, a static class handle or a mailbox can be created and initialized by calling its new method (see Annex 11.4), or static variables can be initialized to random values by calling the `$urandom` system task. This requires a special pre-initial pass at run-time.

The following table contains the default values for SystemVerilog variables.

Table 5-1: Default values

Type	Default Initial value
4 state integral	`'X`
2 state integral	`'0`

Table 5-1: Default values

Type	Default Initial value
real, shortreal	`0.0`
Enumeration	First value in the enumeration
`string`	`""` (empty string)
`event`	New event
`class`	**`null`**
`chandle` (Opaque handle)	**`null`**

5.5 Scope and lifetime

Any data declared outside a module, interface, task, or function, is global in scope (can be used anywhere after its declaration) and has a static lifetime (exists for the whole elaboration and simulation time).

SystemVerilog data declared inside a module or interface but outside a task, process or function is local in scope and static in lifetime (exists for the lifetime of the module or interface). This is roughly equivalent to C static data declared outside a function, which is local to a file.

Data declared in an automatic task, function or block has the lifetime of the call or activation and a local scope. This is roughly equivalent to a C automatic variable.

Data declared in a static task, function or block defaults to a static lifetime and a local scope. If an initializer is used, the keyword **`static`** must be specified to make the code clearer.

Note that in SystemVerilog, data can be declared in unnamed blocks as well as in named blocks. This data is visible to the unnamed block and any nested blocks below it. Hierarchical references cannot be used to access this data by name.

Verilog-2001 allows tasks and functions to be declared as **`automatic`**, making all storage within the task or function automatic. SystemVerilog allows specific data within a static task or function to be explicitly declared as **`automatic`**. Data declared as automatic has the lifetime of the call or block, and is initialized on each entry to the call or block.

SystemVerilog also allows data to be explicitly declared as **`static`**. Data declared to be **`static`** in an automatic task, function or in a process has a static lifetime and a scope local to the block. This is like C static data declared within a function.

```
module ms1;
   int st0; // static
   initial begin
      int st1; //static
      static int st2; //static
      automatic int auto1; //automatic
   end
   task automatic t1();
      int auto2; //automatic
      static int st3; //static
      automatic int auto3; //automatic
   endtask
endmodule
```

Note that automatic or dynamic variables cannot be used to trigger an event expression or be written with a

nonblocking assignment. Automatic variables and dynamic constructs—objects handles, dynamic arrays, associative arrays, strings, and event variables—are limited to the procedural context.

See also Section 10 on tasks and functions.

5.6 Nets, regs, and logic

Verilog-2001 states that a net can be written by one or more continuous assignments, primitive outputs or through module ports. The resultant value of multiple drivers is determined by the resolution function of the net type. A net cannot be procedurally assigned. If a net on one side of a port is driven by a variable on the other side, a continuous assignment is implied. A force statement can override the value of a net. When released, it returns to resolved value.

Verilog-2001 also states that one or more procedural statements can write to variables, including procedural continuous assignments. The last write determines the value. A variable cannot be continuously assigned. The force statement overrides the procedural assign statement, which in turn overrides the normal assignments. A variable cannot be written through a port; it must go through an implicit continuous assignment to a net.

In SystemVerilog, all variables can now be written either by one continuous assignment, or by one or more procedural statements, including procedural continuous assignments. It shall be an error to have multiple continuous assignments or a mixture of procedural and continuous assignments writing to the same variable. All data types can write through a port.

SystemVerilog variables can be packed or unpacked aggregates of other types. Multiple assignments made to independent elements of a variable are examined individually. An assignment where the left-hand-side contains a slice is treated as a single assignment to the entire slice. It shall be an error to have a packed structure or array type written with a mixture of procedural and continuous assignments. Thus, an unpacked structure or array can have one element assigned procedurally, and another element assigned continuously. And, each element of a packed structure or array can each have a single continuous assignment. For example, assume the following structure declaration:

```
struct {
   bit [7:0] A;
   bit [7:0] B;
   byte C;
} abc;
```

The following statements are legal assignments to struct `abc`:

```
assign abc.C = sel ? 8'hBE : 8'hEF;

not    (abc.A[0],abc.B[0]),
       (abc.A[1],abc.B[1]),
       (abc.A[2],abc.B[2]),
       (abc.A[3],abc.B[3]);

always @(posedge clk) abc.B <= abc.B + 1;
```

The following additional statements are illegal assignments to struct `abc`:

```
// Multiple continuous assignments to abc.C
assign abc.C = sel ? 8'hDE : 8'hED;

// Mixing continuous and procedural assignments to abc.A
always @(posedge clk) abc.A[7:4] <= !abc.B[7:4];
```

For the purposes of the preceding rule, a declared variable initialization or a procedural continuous assignment is considered a procedural assignment. A **`force`** statement is neither a continuous or procedural assignment. A

release statement shall not change the variable until there is another procedural assignment, or shall schedule a re-evaluation of the continuous assignment driving it. A single **force** or **release** statement shall not be applied to a whole or part of a variable that is being assigned by a mixture of continuous and procedural assignments.

A continuous assignment is implied when a variable is connected to an input port declaration. This makes assignments to a variable declared as an input port illegal. A continuous assignment is implied when a variable is connected to the output port of an instance. This makes procedural or continuous assignments to a variable connected to the output port of an instance illegal.

SystemVerilog variables cannot be connected to either side of an inout port. SystemVerilog introduces the concept of shared variables across ports with the ref port type. See Section 18.8 (port connections) for more information about ports and port connection rules.

The compiler can issue a warning if a continuous assignment could drive strengths other then `St0`, `St1`, `StX`, or `HiZ` to a variable. In any case, SystemVerilog applies automatic type conversion to the assignment, and the strength is lost.

Note that a SystemVerilog variable cannot have an implicit continuous assignment as part of its declaration, the way a net can. An assignment as part of the logic declaration is a variable initialization, not a continuous assignment. For example:

```
wire w = vara & varb;               // continuous assignment

logic v = consta & constb;          // initial procedural assignment

logic vw; // no initial assignment
assign vw = vara & varb;            // continuous assignment to a logic

real circ;
assign circ = 2.0 * PI * R;         // continuous assignment to a real
```

5.7 Signal aliasing

The Verilog **assign** statement is a unidirectional assignment and can incorporate a delay and strength change. To model a bidirectional short-circuit connection it is necessary to use the **alias** statement. The members of an alias list are signals whose bits share the same physical nets. The example below implements a byte order swapping between bus `A` and bus `B`.

```
module byte_swap (inout wire [31:0] A, inout wire [31:0] B);
   alias {A[7:0],A[15:8],A[23:16],A[31:24]} = B;
endmodule
```

This example strips out the least and most significant bytes from a four byte bus:

```
module byte_rip (inout wire [31:0] W, inout wire [7:0] LSB, MSB);
   alias W[7:0] = LSB;
   alias W[31:24] = MSB;
endmodule
```

The bit overlay rules are the same as those for a packed union with the same member types: each member shall be the same size, and connectivity is independent of the simulation host. The nets connected with an alias statement must be type compatible, that is, they have to be of the same net type. For example, it is illegal to connect a **wand** net to a **wor** net with an **alias** statement. This is a stricter rule than applied to nets joining at ports because the scope of an alias is limited and such connections are more likely to be a design error. Variables and hierarchical references cannot be used in **alias** statements. Any violation of these rules shall be considered a fatal error.

The same nets can appear in multiple alias statements. The effects are cumulative. The following two examples are equivalent. In either case, `low12[11:4]` and `high12[7:0]` share the same wires.

```
module overlap(inout wire [15:0] bus16, inout wire [11:0] low12, high12);
   alias bus16[11:0] = low12;
   alias bus16[15:4] = high12;
endmodule

module overlap(inout wire [15:0] bus16, inout wire [11:0] low12, high12);
   alias bus16 = {high12, low12[3:0]};
   alias high12[7:0] = low12[11:4];
endmodule
```

To avoid errors in specification, it is not allowed to specify an alias from an individual signal to itself, or to specify a given alias more than once. The following version of the code above would be illegal since the top four and bottom four bits are the same in both statements:

```
alias bus16 = {high12[11:8], low12};
alias bus16 = {high12, low12[3:0]};
```

This alternative is also illegal because the bits of `bus16` are being aliased to itself:

```
alias bus16 = {high12, bus16[3:0]} = {bus16[15:12], low12};
```

Alias statements can appear anywhere module instance statements can appear. If an identifier that has not been declared as a data type appears in an alias statement, then an implicit net is assumed, following the same rules as implicit nets for a module instance. The following example uses **`alias`** along with the automatic name binding to connect pins on cells from different libraries to create a standard macro:

```
module lib1_dff(Reset, Clk, Data, Q, Q_Bar);
   ...
endmodule

module lib2_dff(reset, clock, data, a, qbar);
   ...
endmodule

module lib3_dff(RST, CLK, D, Q, Q_);
   ...
endmodule

macromodule my_dff(rst, clk, d, q, q_bar); // wrapper cell
   input rst, clk, d;
   output q, q_bar;
   alias rst = Reset = reset = RST;
   alias clk = Clk = clock = CLK;
   alias d = data = D;
   alias q = Q;
   alias Q_ = q_bar = Q_Bar = qbar;
   `LIB_DFF my_dff (.*); // LIB_DFF is any of lib1_dff, lib2_dff or lib3_dff
endmodule
```

Using a net in an alias statement does not modify its syntactic behavior in other statements. Aliasing is performed at elaboration time and cannot be undone.

Section 6
Attributes

6.1 Introduction (informative)

With Verilog-2001, users can add named attributes (properties) to Verilog objects, such as modules, instances, wires, etc. Attributes can also be specified on the extended SystemVerilog constructs and are included as part of the BNF (see Annex A). SystemVerilog also defines a default data type for attributes.

6.2 Default attribute type

The default type of an attribute with no value is `bit`, with a value of 1. Otherwise, the attribute takes the type of the expression.

Section 7
Operators and Expressions

7.1 Introduction (informative)

The SystemVerilog operators are a combination of Verilog and C operators. In both languages, the type and size of the operands is fixed, and hence the operator is of a fixed type and size. The fixed type and size of operators is preserved in SystemVerilog. This allows efficient code generation.

Verilog does not have assignment operators or increment and decrement operators. SystemVerilog includes the C assignment operators, such as **+=**, and the C increment and decrement operators, **++** and **--**.

Verilog-2001 added signed nets and **reg** variables, and signed based literals. There is a difference in the rules for combining signed and unsigned integers between Verilog and C. SystemVerilog uses the Verilog-2001 rules.

7.2 Operator syntax

```
assignment_operator ::=                                          // from Annex A.6.2
        = | += | -= | *= | /= | %= | &= | |= | ^= | <<= | >>= | <<<= | >>>=
conditional_expression ::=                                       // from Annex A.8.3
        expression1 ? { attribute_instance } expression2 : expression3
unary_operator ::=                                               // from Annex A.8.6
        + | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
binary_operator ::=
        + | - | * | / | % | == | != | === | !== | =?= | !?= | && | || | **
      | < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | << | >>> | <<<
inc_or_dec_operator ::= ++ | --
unary_module_path_operator ::=
    ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
binary_module_path_operator ::=
    == | != | && | || | & | | | ^ | ^~ | ~^
```

Syntax 7-1—Operator syntax (excerpt from Annex A)

7.3 Assignment operators

In addition to the simple assignment operator, =, SystemVerilog includes the C assignment operators and special bitwise assignment operators: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `<<<=`, and `>>>=`. An assignment operator is semantically equivalent to a blocking assignment, with the exception that any left hand side index expression is only evaluated once. For example:

```
a[i]+=2; // same as a[i] = a[i] +2;
```

In SystemVerilog, an expression can include a blocking assignment, provided it does not have a timing control. Note that such an assignment must be enclosed in parentheses to avoid common mistakes such as using `a=b` for `a==b`, or `a|=b` for `a!=b`.

```
if ((a=b)) b = (a+=1);
```



```
a = (b = (c = 5));
```

The semantics of such an assignment expression are those of a function which evaluates the right hand side, casts the right hand side to the left hand data type, stacks it, updates the left hand side and returns the stacked value. The type returned is the type of the left hand side data type. If the left hand side is a concatenation, the type returned shall be an unsigned integral value whose bit length is the sum of the length of its operands.

It shall be illegal to include an assignment operator in an event expression, in an expression within a procedural continuous assignment, or in an expression that is not within a procedural statement.

SystemVerilog includes the C increment and decrement assignment operators `++i`, `--i`, `i++` and `i--`. These do not need parentheses when used in expressions. These increment and decrement assignment operators behave as blocking assignments.

The ordering of assignment operations relative to any other operation within an expression is undefined. An implementation can warn whenever a variable is both written and read-or-written within an integral expression or in other contexts where an implementation cannot guarantee order of evaluation. In the following example:

```
i = 10;
j = i++ + (i = i - 1);
```

After execution, the value of `j` can be 18, 19, or 20 depending upon the relative ordering of the increment and the assignment statements.

7.4 Operations on logic and bit types

When a binary operator has one operand of type **`bit`** and another of type **`logic`**, the result is of type **`logic`**. If one operand is of type **`int`** and the other of type **`integer`**, the result is of type **`integer`**.

The operators **`!=`** and **`==`** return an X if either operand contains an `x` or a `z`, as in Verilog-2001. This is converted to a 0 if the result is converted to type **`bit`**, e.g. in an **`if`** statement.

The unary reduction operators (`& ~& | ~| ^ ~^`) can be applied to any integer expression (including packed arrays). The operators shall return a single value of type **`logic`** if the packed type is four valued, and of type **`bit`** if the packed type is two valued.

```
int i;
bit b = &i;
integer j;
logic c = &j;
```

7.5 Wild equality and wild inequality

SystemVerilog 3.1 introduces the wild-card comparison operators, as described below.

Table 7-1: Wild equality and wild inequality operators

Operator	Usage	Description
=?=	a =?= b	a equals b, X and Z values act as wild cards
!?=	a !?= b	a not equal b, X and Z values act as wild cards

The wild equality operator (`=?=`) and inequality operator (`!?=`) treat X and Z values in a given bit position as a wildcard. A wildcard bit matches any bit value (0, 1,Z, or X) in the value of the expression being compared

against it.

These operators compare operands bit for bit, and return a 1-bit self-determined result. If the operands to the wild-card equality/inequality are of unequal bit length, the operands are extended in the same manner as for the case equality/inequality operators. If the relation is true, the operator yields a 1. If the relation is false, it yields a 0.

The three types of equality (and inequality) operators in SystemVerilog behave differently when their operands contain unknown values (X or Z). The `==` and `!=` operators result in X if any of their operands contains an X or Z. The `===` and `!==` check the 4-state explicitly, therefore, X and Z values shall either match or mismatch, never resulting in X. The `=?=` and `!?=` operators treat X or Z as wild cards that match any value, thus, they too never result in X.

7.6 Real operators

Operands of type **`shortreal`** have the same operation restrictions as Verilog **`real`** operands. The unary operators ++ and -- can have operands of type **`real`** and **`shortreal`** (the increment or decrement is by 1.0). The assignment operators **`+=`**, **`-=`**, **`*=`**, **`/=`** can also have operands of type **`real`** and **`shortreal`**.

If any operand is **`real`**, the result is **`real`**, except before the ? in the ternary operator. If no operand is **`real`** and any operand is **`shortreal`**, the result is **`shortreal`**.

Real operands can also be used in the following expressions:

```
str.realval // structure or union member
realarray[intval] // array element
```

7.7 Size

The number of bits of an expression is determined by the operands and the context, following the same rules as Verilog. In SystemVerilog, casting can be used to set the size context of an intermediate value.

With Verilog, tools can issue a warning when the left and right hand sides of an assignment are different sizes. Using the SystemVerilog size casting, these warnings can be prevented.

7.8 Sign

The following unary operators give the signedness of the operand: **`~ ++ -- + -`**. All other operators shall follow the same rules as Verilog for performing signed and unsigned operations.

7.9 Operator precedence and associativity

Operator precedence and associativity is listed in Table 7-2, below. The highest precedence is listed first.

Table 7-2: Operator precedence and associativity

`()  []  ::  .`	left
`+  -  !  ~  &  ~&  |  ~|  ^  ~^  ^~  ++  --` (unary)	right
`**`	left
`*  /  %`	left
`+  -` (binary)	left

Table 7-2: Operator precedence and associativity (continued)

`<<  >>  <<<  >>>`	left
`<  <=  >  >=  inside  dist`	left
`==  !=  ===  !==  =?=  !?=`	left
`& (binary)`	left
`^  ~^  ^~ (binary)`	left
`| (binary)`	left
`&&`	left
`||`	left
`?: (conditional operator)`	right
`=>`	right
`=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=  <<<=  >>>=  :=  :/  <=`	none
`{}  {{}}`	concatenation

7.10 Built-in methods

SystemVerilog introduces classes and the method calling syntax, in which a task or function is called using the dot notation (`.`):

```
object.task_or_function()
```

The object uniquely identifies the data on which the task or function operates. Hence, the method concept is naturally extended to built-in types in order to add functionality that traditionally was done via system tasks or functions. Unlike system tasks, built-in methods are not prefixed with a `$` since they require no special prefix to avoid collisions with user-defined identifiers. Thus, the method syntax allows extending the language without the addition of new keywords or cluttering the global name space with system tasks.

Built-in methods, unlike system tasks, can not be redefined by users via PLI tasks. Thus, only functions that users should not be allowed to redefine are good candidates for built-in method calls.

In general, a built-in method is preferred over a system task when a particular functionality applies to all data types, or it applies to a specific data type. For example:

```
dynamic_array.size, associative_array.num, and string.len
```

These are all similar concepts, but they represent different things. A dynamic array has a size, an associative array contains a given number of items, and a string has a given length. Using the same system task, such as $length, for all of them would be less clear and intuitive.

A built-in method can only be associated with a particular data type. Therefore, if some functionality is a simple side effect (i.e., `$stop` or `$reset`) or it operates on no specific data (i.e., `$random`) then a system task must be used.

When a function or task built-in method call specifies no arguments, the empty parenthesis, `()`, following the task/function name is optional. This is also true for tasks or functions that require arguments, when all arguments have defaults specified. For a method, this rule allows simple calls to appear as properties of the object or built-in type. Similar rules are defined for functions and tasks in Section 10.5.5.

7.11 Concatenation

Braces (**{ }**) are used to show concatenation, as in Verilog. The concatenation is treated as a packed vector of bits. It can be used on the left hand side of an assignment or in an expression.

```
logic log1, log2, log3;
{log1, log2, log3} = 3'b111;
{log1, log2, log3} = {1'b1, 1'b1, 1'b1}; // same effect as 3'b111
```

Software tools can generate a warning if the concatenation width on one side of an assignment is different than the expression on the other side. The following examples can give warning of size mismatch:

```
bit [1:0] packedbits = {32'b1,32'b1}; // right hand side is 64 bits
int i = {1'b1, 1'b1}; //right hand side is 2 bits
```

Note that braces are also used for initializers of structures or unpacked arrays. Unlike in C, the expressions must match element for element and the braces must match the structures and array dimensions. Each element must match the type being initialized, so the following do not give size warnings:

```
bit unpackedbits [1:0] = {1,1}; // no size warning, bit can be set to 1
int unpackedints [1:0] = {1'b1,1'b1}; //no size warning, int can be set to 1'b1
```

A concatenation of unsized values shall be illegal, as in Verilog. However, an array initializer can use unsized values within the braces, such as {1,1}.

The replication operator (also called a multiple concatenation) form of braces can also be used for initializers . For example, {3{1}} represents the initializer {1, 1, 1}.

Refer to Sections 2.7 and 2.8 for more information on initializing arrays and structures .

SystemVerilog enhances the concatenation operation to allow concatenation of variables of type string. In general, if any of the operands is of type **string**, the concatenation is treated as a string, and all other arguments are implicitly converted to the **string** type (as described in Section 3.7). String concatenation is not allowed on the left hand side of an assignment, only as an expression.

```
string hello = "hello";
string s;
s = { hello, " ", "world" };
$display( "%s\n", s );        // displays 'hello world'
s = { s, " and goodbye" };
$display( "%s\n", s );        // displays 'hello world and goodbye'
```

The replication operator (also called a multiple concatenation) form of braces can also be used with variables of type **string**. In the case of string replication, a non-constant multiplier is allowed.

```
int n = 3;
string s = {n { "boo " }};
$display( "%s\n", s );  // displays 'boo boo boo '
```

Note that unlike bit concatenation, the result of a string concatenation or replication is not truncated. Instead, the destination variable (of type **string**) is resized to accommodate the resulting string.

7.12 Unpacked array expressions

Braces are also used for expressions to assign to unpacked arrays. Unlike in C, the expressions must match element for element, and the braces must match the array dimensions. The type of each element is matched against the type of the expression according to the same rules as for a scalar. This means that the following examples do not give size warnings, unlike the similar assignments above:


```
bit unpackedbits [1:0] = {1,1};  // no size warning as bit can be set to 1
int unpackedints [1:0] = {1'b1, 1'b1}; // no size warning as int can be
                                       // set to 1'b1
```

The syntax of multiple concatenations can be used for unpacked array expressions as well.

```
unpackedbits = {2 {y}} ; // same as {y, y}
```

SystemVerilog determines the context of the braces by looking at the left hand side of an assignment. If the left hand side is an unpacked array, the braces represent an unpacked array literal or expression. Outside the context of an assignment on the right hand side, an explicit cast must be used with the braces to distinguish it from a concatenation.

It can sometimes be useful to set array elements to a value without having to keep track of how many members there are. This can be done with the **`default`** keyword:

```
initial unpackedints = {default:2}; // sets elements to 2
```

For more arrays of structures, it is useful to specify one or more matching types, as illustrated under structure expressions, below.

```
struct {int a; time b;} abkey[1:0];
abkey = {{a:1, b:2ns}, {int:5, time:$time}};
```

The rules for unpacked array matching are as follows:

— For `type:value`, if the element or sub array type of the unpacked array exactly matches this type, then each element or sub array shall be set to the value. The value must be castable to the array element or sub array type. Otherwise, if the unpacked array is multidimensional, then there is a recursive descent into each sub array of the array using the rules in this section and the type and default specifiers. Otherwise, if the unpacked array is an array of structures, there is a recursive descent into each element of the array using the rules for structure expressions and the type and default specifiers.

— For `default:value`, this specifies the default value to use for each element of an unpacked array that has not been covered by the earlier rules in this section. The value must be castable to the array element type.

7.13 Structure expressions

A structure expression (packed or unpacked) can be built from member expressions using braces and commas, with the members in declaration order. It can also be built with the names of the members

```
module mod1;

   typedef struct {
      int x;
      int y;
   } st;

   st s1;
   int k = 1;

   initial begin
      #1 s1 = {1, 2+k};          // by position
      #1 $display( s1.x, s1.y);
      #1 s1 = {x:2, y:3+k);      // by name
      #1 $display( s1);
      #1 $finish;
```

```
   end
endmodule
```

It can sometimes be useful to set structure members to a value without having to keep track of how many members there are, or what the names are. This can be done with the **default** keyword:

```
initial s1 = {default:2};  // sets x and y to 2
```

The {member:value} or {data_type: default_value} syntax can also be used:

```
ab abkey[1:0] = {{a:1, b:1.0}, {int:2, shortreal:2.0}};
```

Note that the **default** keyword applies to members in nested structures or elements in unpacked arrays in structures. In fact, it descends the nesting to a built-in type or a packed array of them.

```
struct {
   int A;
   struct {
      int B, C;
   } BC1, BC2;
}

ABC = {A:1, BC1:{B:2, C:3}, BC2:{B:4,C:5}};
DEF = {default:10};
```

To deal with the problem of members of different types, a type can be used as the key. This overrides the default for members of that type:

```
typedef struct {
   logic [7:0] a;
   bit b;
   bit [31:0] c;
   string s;
} sa;

sa s2;
initial s2 = {bit[31:0]:1, default:0, string:""};  // set all to 0 except the
                                                   // array of bits to 1 and
                                                   // string to ""
```

Similarly, an individual member can be set to override the general default and the type default:

```
initial #10 s1 = {default:'1, s = ""}; // set all to 1 except s to ""
```

SystemVerilog determines the context of the braces by looking at the left hand side of an assignment. If the left hand side is an unpacked structure, the braces represent an unpacked structure literal or expression. Outside the context of an assignment on the right hand side, an explicit cast must be used with the braces to distinguish it from a concatenation.

The matching rules are as follows:

— A member:value: specifies an explicit value for a named member of the structure. The named member must be at the top level of the structure—a member with the same name in some level of substructure shall not be set. The value must be castable to the member type, otherwise an error is generated.

— The type:value specifies an explicit value for a field in the structure which exactly matches the type and has not been set by a field name specifier above. If the same key type is mentioned more than once, the last value is used.

— The `default:value` applies to members that are not matched by either member name or type and are not either structures or unpacked arrays. The value must be castable to the member type, otherwise an error is generated. For unmatched structure members, the type and default specifiers are applied recursively according to the rules in this section to each member of the substructure. For unmatched unpacked array members, the type and default specifiers are applied to the array according to the rules for unpacked arrays.

Every member must be covered by one of these rules.

7.14 Aggregate expressions

Unpacked structure and array variables, literals, and expressions can all be used as aggregate expressions. A multi-element slice of an unpacked array can also be used as an aggregate expression.

Aggregate expressions can be copied in an assignment, through a port, or as an argument to a task or function. Aggregate expressions can also be compared with equality or inequality operators. To be copied or compared, the type of an aggregate expression must be equivalent.

Unpacked structures types are equivalent by the hierarchical name of its type alone. This means in order to have two equivalent unpacked structures in two different scopes, the type must be defined in one of the following ways:

— Defined in a higher-level scope common to both expressions.

— Passed through type parameter.

— Imported by hierarchical reference.

Unpacked arrays types are equivalent by having equivalent element types and identical shape. Shape is defined as the number of dimensions and the number of elements in each dimension, not the actual range of the dimension.

7.15 Conditional operator

```
conditional_expression ::= (From Annex A.8.3)
expression1 ? { attribute_instance } expression2 : expression3
```

As defined in Verilog, if *expression1* is true, the operator returns *expression2*, if false, it returns *expression3*. If *expression1* evaluates to an ambiguous value (x or z), then both *expression2* and *expression3* shall be evaluated and their results shall be combined, bit by bit.

SystemVerilog extends the conditional operator to non bit-level types and aggregate expressions using the following rules:

— If both *expression2* and *expression3* are bit-level types, or a packed aggregate of bit type, the operation proceeds as defined.

— If *expression2* or *expression3* is a bit-level type and the opposing expression can be implicitly cast to a bit-level type, the cast is made and proceeds as defined.

— For all other cases, the type of *expression2* and *expression3* must be equivalent.

If *expression1* evaluates to an ambiguous value, then both *expression2* and *expression3* shall be evaluated and their results shall be combined, element-by-element. If the elements match, the element is returned. If they do not match, then the default-uninitialized value for that element's type shall be returned.

Section 8
Procedural Statements and Control Flow

8.1 Introduction (informative)

Procedural statements are introduced by the following:

`initial` // enable this statement at the beginning of simulation and execute it only once

`final` // do this statement once at the end of simulation

`always`, **`always_comb`**, **`always_latch`**, **`always_ff`** // loop forever (see Section 9 on processes)

`task` // do these statements whenever the task is called

`function` // do these statements whenever the function is called and return 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

Verilog procedural statements are in **`initial`** or **`always`** blocks, tasks or functions. SystemVerilog adds a final block that executes at the end of simulation.

Verilog includes most of the statement types of C, except for `do...while`, `break`, `continue` and `goto`. Verilog has the **`repeat`** statement which C does not, and the **`disable`**. The use of the Verilog **`disable`** to carry out the functionality of break and continue requires the user to invent block names, and introduces the opportunity for error.

SystemVerilog adds C-like **`break`**, **`continue`** and **`return`** functionality, which do not require the use of block names.

Loops with a test at the end are sometimes useful to save duplication of the loop body. SystemVerilog adds a C-like **`do`**...**`while`** loop for this capability.

Verilog provides two overlapping methods for procedurally adding and removing drivers for variables: the **`force`**/**`release`** statements and the **`assign`**/**`deassign`** statements. The **`force`**/**`release`** statements can also be used to add or remove drivers for nets in addition to variables. A force statement targeting a variable that is currently the target of an assign shall override that assign; however, once the force is released, the assign's effect again shall be visible.

The keyword **`assign`** is much more commonly used for the somewhat similar, yet quite different purpose of defining permanent drivers of values to nets.

SystemVerilog **`final`** blocks execute in an arbitrary but deterministic sequential order. This is possible because **`final`** blocks are limited to the legal set of statements allowed for functions. SystemVerilog does not specify the ordering, but implementations should define rules that preserve the ordering between runs. This helps keep the output log file stable since **`final`** blocks are mainly used for displaying statistics.

8.2 Statements

The syntax for procedural statements is:


```
statement_or_null ::=                                          // from Annex A.6.4
                statement
              | { attribute_instance } ;
statement ::= [ block_identifier : ] statement_item
statement_item ::=
                { attribute_instance } blocking_assignment ;
              | { attribute_instance } nonblocking_assignment ;
              | { attribute_instance } procedural_continuous_assignments ;
              | { attribute_instance } case_statement
              | { attribute_instance } conditional_statement
              | { attribute_instance } inc_or_dec_expression ;
              | { attribute_instance } function_call ;
              | { attribute_instance } disable_statement
              | { attribute_instance } event_trigger
              | { attribute_instance } loop_statement
              | { attribute_instance } jump_statement
              | { attribute_instance } par_block
              | { attribute_instance } procedural_timing_control_statement
              | { attribute_instance } seq_block
              | { attribute_instance } system_task_enable
              | { attribute_instance } task_enable
              | { attribute_instance } wait_statement
              | { attribute_instance } procedural_assertion_item
function_statement ::= [ block_identifier : ] function_statement_item
function_statement_item ::=
                { attribute_instance } function_blocking_assignment ;
              | { attribute_instance } function_case_statement
              | { attribute_instance } function_conditional_statement
              | { attribute_instance } inc_or_dec_expression ;
              | { attribute_instance } function_call ;
              | { attribute_instance } function_loop_statement
              | { attribute_instance } jump_statement
              | { attribute_instance } function_seq_block
              | { attribute_instance } disable_statement
              | { attribute_instance } system_task_enable
```

Syntax 8-1—Procedural statement syntax (excerpt from Annex A)

8.3 Blocking and nonblocking assignments

```
blocking_assignment ::=                                          // from Annex A.6.4
        variable_lvalue = delay_or_event_control  expression
      | hierarchical_variable_identifier = new [ constant_expression ] [ ( variable_identifier ) ]
      | class_identifier [ parameter_value_assignment ] = new [ ( list_of_arguments ) ]
      | class_identifier . randomize [ ( ) ] with constraint_block ;
      | operator_assignment
operator_assignment ::= variable_lvalue  assignment_operator  expression
assignment_operator ::=
        = | += | -= | *= | /= | %= | &= | |= | ^= | <<= | >>= | <<<= | >>>=
nonblocking_assignment ::= variable_lvalue <= [ delay_or_event_control ] expression
```

Syntax 8-2—blocking and nonblocking assignment syntax (excerpt from Annex A)

The following assignments are allowed in both Verilog-2001 and SystemVerilog:

```
#1 r = a;
r = #1 a;
r <= #1 a;
r <= a;
@c r = a;
r = @c a;
r <= @c a;
```

SystemVerilog also allows a time unit to specified in the assignment statement, as follows:

```
#1ns r = a;
r = #1ns a;
r <= #1ns a;
```

It shall be illegal to make nonblocking assignments to automatic variables.

The size of the left-hand side of an assignment forms the context for the right hand side expression. If the left-hand side is smaller than the right hand side, information can be lost, and a warning can be given.

8.4 Selection statements

```
conditional_statement ::=                                   // from Annex A.6.6
        [ unique_priority ] if ( expression ) statement_or_null [ else statement_or_null ]
      | if_else_if_statement
if_else_if_statement ::=
        [ unique_priority ] if ( expression ) statement_or_null
        { else [ unique_priority ] if ( expression ) statement_or_null }
        [ else statement_or_null ]
unique_priority ::= unique | priority
case_statement ::=                                          // from Annex A.6.7
        [ unique_priority ] case ( expression ) case_item { case_item } endcase
      | [ unique_priority ] casez ( expression ) case_item { case_item } endcase
      | [ unique_priority ] casex ( expression ) case_item { case_item } endcase
case_item ::=
        expression { , expression } : statement_or_null
      | default [ : ] statement_or_null
```

Syntax 8-3—Selection statement syntax (excerpt from Annex A)

In Verilog, an **`if`** **`(`***expression*`)` is evaluated as a boolean, so that if the result of the expression is 0 or X, the test is considered false.

SystemVerilog adds the keywords **`unique`** and **`priority`**, which can be used before an **`if`**. If either keyword is used, it shall be a run-time error for no condition to match unless there is an explicit **`else`**. For example:

```
unique if ((a==0) || (a==1)) $display("0 or 1");
else if (a == 2) $display("2");
else if (a == 4) $display("4"); // values 3,5,6,7 cause an error

priority if (a[2:1]==0) $display("0 or 1");
else if (a[2] == 0) $display("2 or 3");
else $display("4 to 7"); //covers all other possible values, so no error
```

A **`unique if`** indicates that there should not be any overlap in a series of **`if`**...**`else`**...**`if`** conditions, allowing the expressions to be evaluated in parallel. A software tool shall issue an error if it determines that there is a potential overlap in the conditions.

A **`priority if`** indicates that a series of **`if`**...**`else`**...**`if`** conditions shall be evaluated in the order listed. In the preceding example, if the variable `a` had a value of 0, it would satisfy both the first and second conditions, requiring priority logic.

The **`unique`** and **`priority`** keywords apply to the entire series of **`if`**...**`else`**...**`if`** conditions. In the preceding examples it would have been illegal to insert either keyword after any of the occurrences of **`else`**.

In Verilog, there are three types of case statements, introduced by **`case`**, **`casez`** and **`casex`**. With SystemVerilog, each of these can be qualified by **`priority`** or **`unique`**. A **`priority case`** shall act on the first match only. A **`unique case`** shall guarantee no overlapping case values, allowing the case items to be evaluated in parallel. If the case is qualified as **`priority`** or **`unique`**, the simulator shall issue an error message if an unexpected case value is found.

Note: by specifying **`unique`** or **`priority`**, it is not necessary to code a **`default`** case to trap unexpected case values. For example:

```
bit [2:0] a;
unique case(a) // values 3,5,6,7 cause a run-time error
      0,1: $display("0 or 1");
      2: $display("2");
      4: $display("4");
endcase

priority casez(a)
      2'b00?: $display("0 or 1");
      2'b0??: $display("2 or 3");
      default: $display("4 to 7");
endcase
```

The **unique** and **priority** keywords shall determine the simulation behavior. It is recommended that synthesis follow simulation behavior where possible. Attributes can also be used to determine synthesis behavior.

8.5 Loop statements

loop_statement ::= *// from Annex A.6.8*
  forever statement_or_null
| **repeat (** expression **)** statement_or_null
| **while (** expression **)** statement_or_null
| **for (** variable_decl_or_assignment **;** expression **;** variable_assignment **)** statement_or_null
| **for (** variable_decl_or_assignment { **,** variable_decl_or_assignment } **;** expression **;**
  variable_assignment { **,** variable_assignment } **)** statement_or_null
| **do** statement_or_null **while (** expression **) ;**

variable_decl_or_assignment ::=
  data_type list_of_variable_identifiers_or_assignments
| variable_assignment

Syntax 8-4—Loop statement syntax (excerpt from Annex A)

Verilog provides **for**, **while**, **repeat** and **forever** loops. SystemVerilog enhances the Verilog **for** loop, and adds a **do**...**while** loop.

8.5.1 The do...while loop

```
do statement while(condition) // as C
```

The condition can be any expression which can be treated as a boolean. It is evaluated after the statement.

8.5.2 Enhanced for loop

In Verilog, the variable used to control a **for** loop must be declared prior to the loop. If loops in two or more parallel procedures use the same loop control variable, there is a potential of one loop modifying the variable while other loops are still using it.

SystemVerilog adds the ability to declare the **for** loop control variable within the **for** loop. This creates a local variable within the loop. Other parallel loops cannot inadvertently affect the loop control variable. For example:

```
module foo;

   initial begin
```



```
      for (int i = 0; i <= 255; i++)
         ...
   end

   initial begin
      loop2: for (int i = 15; i >= 0; i--)
         ...
   end
endmodule
```

The local variable declared within a **for** loop is equivalent to declaring an automatic variable in an unnamed block.

Verilog only permits a single initial statement and a single step assignment within a **for** loop. SystemVerilog allows the initial declaration or assignment statement to be one or more comma-separated statements. The step assignment can also be one or more comma-separated assignment statements.

```
for ( int count = 0; count < 3; count++ )
   value = value +((a[count]) * (count+1));

for ( int count = 0, done = 0, int j = 0; j * count < 125; j++ )
   $display("Value j = %d\n", j );
```

8.6 Jump statements

```
jump_statement ::=                                    // from Annex A.6.5
        return [ expression ] ;
      | break ;
      | continue ;
```

Syntax 8-5—Jump statement syntax (excerpt from Annex A)

SystemVerilog adds the C jump statements **break**, **continue** and **return**.

```
break    // out of loop as C
continue // skip to end of loop as C
return expression    // exit from a function
return   // exit from a task or void function
```

The **continue** and **break** statements can only be used in a loop. The **continue** statement jumps to the end of the loop and executes the loop control if present. The **break** statement jumps out of the loop. The **continue** and **break** statements cannot be used inside a **fork**...**join** block to control a loop outside the **fork**...**join** block.

The **return** statement can only be used in a task or function. In a function returning a value, the return must have an expression of the correct type.

Note that SystemVerilog does not include the C **goto** statement.

8.7 Final blocks

The **final** block is like an **initial** block, defining a procedural block of statements, except that it occurs at the end of simulation time and executes without delays. A **final** block is typically used to display statistical information about the simulation.

```
final_construct ::= final function_statement                    // from Annex A.6.2
```

Syntax 8-6—Final block syntax (excerpt from Annex A)

The only statements allowed inside a **`final`** block are those permitted inside a function declaration. This guarantees that they execute within a single simulation cycle. Unlike an **`initial`** block, the **`final`** block does not execute as a separate process; instead, it executes in zero time, the same as a function call.

After one of the following conditions occur, all spawned processes are terminated, all pending PLI callbacks are canceled, and then the final block executes.

— The event queue is empty

— Execution of `$finish`

— Termination of all program blocks, which executes an implicit `$finish`

— PLI execution of `tf_dofinish()` or `vpi_control(vpiFinish,...)`

```
final
   begin
      $display("Number of cycles executed %d",$time/period);
      $display("Final PC = %h",PC);
   end
```

Execution of `$finish`, `tf_dofinish()`, or `vpi_control(vpiFinish,...)` from within a final block shall cause the simulation to end immediately. Final blocks can only trigger once in a simulation.

Final blocks shall execute before any PLI callbacks that indicate the end of simulation.

8.8 Named blocks and statement labels

```
seq_block ::=                                                   // from Annex A.6.3
        begin [ : block_identifier ] { block_item_declaration } { statement_or_null }
        end [ : block_identifier ]
par_block ::=
        fork [ : block_identifier ] { block_item_declaration } { statement_or_null }
        join_keyword [ : block_identifier ]
join_keyword ::= join | join_any | join_none
```

Syntax 8-7—Blocks and labels syntax (excerpt from Annex A)

Verilog allows a **`begin`**...**`end`**, **`fork`**...**`join`**, **`fork`**...**`join_any`** or **`fork`**...**`join_none`** statement block to be named. A named block is used to identify the entire statement block. A named block creates a new hierarchy scope. The block name is specified after the **`begin`** or **`fork`** keyword, preceded by a colon. For example:

```
begin : blockA    // Verilog-2001 named block
   ...
end
```

SystemVerilog allows a matching block name to be specified after the block **`end`**, **`join`**, **`join_any`** or **`join_none`** keyword, preceded by a colon. This can help document which **`end`** or **`join`**, **`join_any`** or **`join_none`** is associated with which **`begin`** or **`fork`** when there are nested blocks. A name at the end of the block is not required. It shall be an error if the name at the end is different than the block name at the begin-

ning.

```
begin: blockB     // block name after the begin or fork
   ...
end: blockB
```

SystemVerilog allows a label to be specified before any statement, as in C. A statement label is used to identify a single statement. The label name is specified before the statement, followed by a colon.

```
labelA: statement
```

A **begin**...**end**, **fork**...**join**, **fork**...**join_any** or **fork**...**join_none** block is considered a statement, and can have a statement label before the block.

```
labelB: fork   // label before the begin or fork
   ...
join : labelB
```

It shall be illegal to have both a label before a **begin** or **fork** and a block name after the **begin** or **fork**. A label cannot appear before the **end**, **join**, **join_any** or **join_none**, as these keywords do not form a statement.

A statement with a label can be disabled using a **disable** statement. Disabling a statement shall have the same behavior as disabling a named block.

See Section 9.6 for additional discussion on **fork**...**join**, **fork**...**join_any** or **fork**...**join_none**.

8.9 Disable

SystemVerilog has **break** and **continue** to break out of or continue the execution of loops. The Verilog-2001 disable can also be used to break out of or continue a loop, but is more awkward than using **break** or **continue**. The **disable** is also allowed to disable a named block, which does not contain the **disable** statement. If the block is currently executing, this causes control to jump to the statement immediately after the block. If the block is a loop body, it acts like a **continue**. If the block is not currently executing, the **disable** has no effect.

SystemVerilog has **return** from a task, but **disable** is also supported. If **disable** is applied to a named task, all current executions of the task are disabled.

```
module ...
always always1: begin ... t1: task1( ); ... end
...
endmodule

always begin
   ...
   disable u1.always1.t1; // exit task1, which was called from always1 (static)
end
```

8.10 Event control

```
delay_or_event_control ::=                                  // from Annex A.6.5
        delay_control
      | event_control
      | repeat ( expression ) event_control
delay_control ::=
        # delay_value
      | # ( mintypmax_expression )
event_control ::=
        @ event_identifier
      | @ ( event_expression )
      | @*
      | @ (*)
event_expression ::=
        [ edge_identifier ] expression [ iff expression ]
      | event_expression or event_expression
      | event_expression , event_expression
edge_identifier ::= posedge | negedge                       // from Annex A.7.4
```

Syntax 8-8—Delay and event control syntax (excerpt from Annex A)

Any change in a variable or net can be detected using the **@** event control, as in Verilog. If the expression evaluates to a result of more than one bit, a change on any of the bits of the result (including an x to z change) shall trigger the event control.

SystemVerilog adds an **`iff`** qualifier to the **`@`** event control.

```
module latch (output logic [31:0] y, input [31:0] a, input enable);
   always @(a iff enable == 1)
      y <= a; //latch is in transparent mode
endmodule
```

The event expression only triggers if the expression after the **`iff`** is true, in this case when `enable` is equal to 1. Note that such an expression is evaluated when `a` changes, and not when `enable` changes. Also note that **`iff`** has precedence over **`or`**. This can be made clearer by the use of parentheses.

If a variable is not of a 4-state type, then **`posedge`** and **`negedge`** refer to transitions from 0 and to 0, respectively.

If the expression denotes a clocking-domain **`input`** or **`inout`** (see Section 15), the event control operator uses the synchronous values, that is, the values sampled by the clocking event. The expression can also denote a clocking-domain name (with no edge qualifier) to be triggered by the clocking event.

A variable used with the event control can be any one of the integral data types (see Section 3.3.1) or string. The variable can be either a simple variable or a **`ref`** argument (variable passed by reference); it can be a member of an array, associative-array, or object (class instance) of the aforementioned types. Objects (handles) and aggregate types are not allowed.

Event control variables can include array subscript expressions, in which case the index expression is evaluated only once when the event control statement is executed. Likewise, an object data member in an event control shall block until that particular data member changes value, not when the handle to the object is modified. For example:


```
Packer p = new; // Packet 1
Packet q = new; // Packet 2
fork
   @(p.status); // Wait for status in Packet 1 to change
      p = q; // Has no effect on the wait in Process 1
join_none
// @(p.status) continues to wait for status of Packet 1 to change
```

8.11 Procedural assign and deassign removal

SystemVerilog currently supports the procedural **`assign`** and **`deassign`** statements. However, these statements may be removed from future versions of the language. See Section 25.3.

Section 9
Processes

9.1 Introduction (informative)

Verilog-2001 has **always** and **initial** blocks which define static processes.

In an **always** block which is used to model combinational logic, forgetting an **else** leads to an unintended latch. To avoid this mistake, SystemVerilog adds specialized **always_comb** and **always_latch** blocks, which indicate design intent to simulation, synthesis and formal verification tools. SystemVerilog also adds an **always_ff** block to indicate sequential logic.

In systems modeling, one of the key limitations of Verilog is the inability to create processes dynamically, as happens in an operating system. Verilog has the **fork**...**join** construct, but this still imposes a static limit.

SystemVerilog has both static processes, introduced by **always**, **initial** or **fork**, and dynamic processes, introduced by built-in **fork**...**join_any** and **fork**...**join_none**.

SystemVerilog creates a thread of execution for each **initial** or **always** block, for each parallel statement in a **fork**...**join** block and for each dynamic process. Each continuous assignment can also be considered its own thread.

SystemVerilog 3.1 adds dynamic processes by enhancing the **fork**...**join** construct in a way that is more natural to Verilog users. SystemVerilog 3.1 also introduces dynamic process control constructs that can terminate or wait for processes using their dynamic, parent-child relationship. These are **wait fork** and **disable fork**.

9.2 Combinational logic

SystemVerilog provides a special **always_comb** procedure for modeling combinational logic behavior. For example:

```
always_comb
   a = b & c;

always_comb
   d <= #1ns b & c;
```

The **always_comb** procedure provides functionality that is different than a normal always procedure:

— There is an inferred sensitivity list that includes every variable read by the procedure.

— The variables written on the left-hand side of assignments shall not be written to by any other process.

— The procedure is automatically triggered once at time zero, after all **initial** and **always** blocks have been started, so that the outputs of the procedure are consistent with the inputs.

The SystemVerilog **always_comb** procedure differs from the Verilog-2001 **always @*** in the following ways:

— **always_comb** automatically executes once at time zero, whereas **always @*** waits until a change occurs on a signal in the inferred sensitivity list.

— **always_comb** is sensitive to changes within the contents of a function, whereas **always @*** is only sensitive to changes to the arguments of a function.

— Variables on the left-hand side of assignments within an **always_comb** procedure shall not be written to by any other processes, whereas **always @*** permits multiple processes to write to the same variable.

Software tools can perform additional checks to warn if the behavior within an **always_comb** procedure does

not represent combinational logic, such as if latched behavior can be inferred.

9.3 Latched logic

SystemVerilog also provides a special **always_latch** procedure for modeling latched logic behavior. For example:

```
always_latch
   if(ck) q <= d;
```

The **always_latch** procedure determines its sensitivity and executes identically to the **always_comb** procedure. Software tools can perform additional checks to warn if the behavior within an **always_latch** procedure does not represent latched logic.

9.4 Sequential logic

The SystemVerilog **always_ff** procedure can be used to model synthesizable sequential logic behavior. For example:

```
always_ff @(posedge clock iff reset == 0 or posedge reset) begin
   r1 <= reset ? 0 : r2 + 1;
   ...
end
```

The **always_ff** block imposes the restriction that only one event control is allowed. Software tools can perform additional checks to warn if the behavior within an **always_ff** procedure does not represent sequential logic.

9.5 Continuous assignments

In Verilog, continuous assignments can only drive nets, and not variables.

SystemVerilog removes this restriction, and permits continuous assignments to drive nets any type of variable. Nets can be driven by multiple continuous assignments, or a mixture of primitives and continuous assignments. Variables can only be driven by one continuous assignment or one primitive output. It shall be an error for a variable driven by a continuous assignment or primitive output to have an initializer in the declaration or any procedural assignment. See also Section 5.6.

9.6 fork...join

The **fork**...**join** construct enables the creation of concurrent processes from each of its parallel statements.

The syntax to declare a **fork**...**join** block is:

```
par_block ::=                                              // from Annex A.6.3
        fork [ : block_identifier ] { block_item_declaration } { statement_or_null }
        join_keyword [ : block_identifier ]
join_keyword ::= join | join_any | join_none
```

Syntax 9-1—Fork...join block syntax (excerpt from Annex A)

One or more statements can be specified, each statement shall execute as a concurrent process.

A Verilog **fork**...**join** block always causes the process executing the fork statement to block until the termination of all forked processes. With the addition of the **join_any** and **join_none** keywords, SystemVerilog provides three choices for specifying when the parent (forking) process resumes execution.

Table 9-1: fork...join control options

Option	Description
join	The parent process blocks until all the processes spawned by this fork complete. .
join_any	The parent process blocks until any one of the processes spawned by this fork complete.
join_none	The parent process continues to execute concurrently with all the processes spawned by the fork. The spawned processes do not start executing until the parent thread executes a blocking statement.

When defining a **fork**...**join** block, encapsulating the entire fork within a **begin**...**end** block causes the entire block to execute as a single process, with each statement executing sequentially.

```
fork
   begin
      statement1;    // one process with 2 statements
      statement2;
   end
join
```

In the following example, two processes are forked, the first one waits for 20ns and the second waits for the named event `eventA` to be triggered. Because the **join** keyword is specified, the parent process shall block until the two processes complete; That is, until 20ns have elapsed and `eventA` has been triggered.

```
fork
   begin
      $display( "First Block\n" );
      # 20ns;
   end
   begin
      $display( "Second Block\n" );
      @eventA;
   end
join
```

A **return** statement within the context of a **fork**...**join** statement is illegal and shall result in a compilation error. For example:

```
task wait_20;
   fork
      # 20;
      return ;    // Illegal: cannot return; task lives in another process
   join_none
endtask
```

Note: SystemVerilog 3.0 provided a **process** statement, which gave the same functionality as the **fork**...**join_none** construct. SystemVerilog 3.1 deprecates the **process** statement, in favor of **fork**...**join_none**.

9.7 Process execution threads

SystemVerilog creates a thread of execution for:

— Each **initial** block

— Each **always** block

— Each parallel statement in a **fork**...**join** (or **join_any** or **join_none**) statement group

— Each dynamic process

Each continuous assignment can also be considered its own thread.

9.8 Process control

SystemVerilog provides constructs that allow one process to terminate or wait for the completion of other processes. The **wait fork** construct waits for the completion of processes. The **disable fork** construct stops the execution of processes.

9.8.1 Wait fork

The **wait fork** statement is used to ensure that all child processes (processes created by the calling process) have completed their execution.

The syntax for **wait fork** is:

```
wait fork ; // from Annex A.6.5
```

Specifying **wait fork** causes the calling process to block until all its sub-processes have completed.

Verilog terminates a simulation run when there is no further activity of any kind. SystemVerilog adds the ability to automatically terminate the simulation when all its program blocks finish executing (i.e, they reach the end of their execute block), regardless of the status of any child processes (see Section 16.6). The **wait fork** statement allows a program block to wait for the completion of all its concurrent threads before exiting.

In the following example, in the task do_test, the first two processes are spawned and the task blocks until one of the two processes completes (either `exec1`, or `exec2`). Next, two more processes are spawned in the background. The **wait fork** statement shall ensure that the task `do_test` waits for all four spawned processes to complete before returning to its caller.

```
task do_test;
   fork
      exec1();
      exec2();
   join_any
   fork
      exec3();
      exec4();
   join_none
   wait fork;        // block until exec1 ... exec4 complete
endtask
```

9.8.2 Disable fork

The **disable fork** statement terminates all active descendants (sub-processes) of the calling process.

The syntax for **disable fork** is:

```
disable fork ; // from Annex A.6.5
```

The **disable fork** statement terminates all descendants of the calling process, as well as the descendants of the process' descendants, that is, if any of the child processes have descendants of their own, the **disable**

fork statement shall terminate them as well.

In the example below, the function `get_first` spawns three versions of a function that wait for a particular device (1, 7, or 13). The function `wait_device` waits for a particular device to become ready and then returns the device's address. When the first device becomes available, the `get_first` function shall resume execution and proceed to kill the outstanding `wait_device` processes.

```
function integer get_first();
   fork
      get_first = wait_device( 1 );
      get_first = wait_device( 7 );
      get_first = wait_device( 13 );
   join_any
   disable fork;
endfunction
```

Verilog supports the **disable** construct, which terminate a process when applied to the named block being executed by the process. The **disable fork** statement differs from **disable** in that **disable fork** considers the dynamic parent-child relationship of the processes, whereas **disable** uses the static, syntactical information of the disabled block. Thus, **disable** shall end all processes executing a particular block, whether the processes were forked by the calling thread or not, while **disable fork** shall end only those processes that were spawned by the calling thread.

Section 10
Tasks and Functions

10.1 Introduction (informative)

Verilog-2001 has static and automatic tasks and functions. Static tasks and functions share the same storage space for all calls to the tasks or function within a module instance. Automatic tasks and function allocate unique, stacked storage for each instance.

SystemVerilog adds the ability to declare automatic variables within static tasks and functions, and static variables within automatic tasks and functions.

SystemVerilog also adds:

— More capabilities for declaring task and function ports

— Function output and inout ports

— Void functions

— Multiple statements in a task or function without requiring a **`begin`**...**`end`** or **`fork`**...**`join`** block

— Returning from a task or function before reaching the end of the task or function

— Passing arguments by reference instead of by value

— Passing argument values by name instead of by position

— Default argument values

— Importing and exporting functions through the Direct Programming Interface (DPI)

10.2 Tasks

```
task_body_declaration ::=                                          // from Annex A.2.7
        [ interface_identifier . ] task_identifier ;
        { task_item_declaration }
        { statement_or_null }
        endtask [ : task_identifier ]
      | [ interface_identifier . ] task_identifier ( task_port_list ) ;
        { block_item_declaration }
        { statement_or_null }
        endtask [ : task_identifier ]

task_declaration ::= task [ lifetime ] task_body_declaration

task_item_declaration ::=
        block_item_declaration
      | { attribute_instance } tf_input_declaration ;
      | { attribute_instance } tf_output_declaration ;
      | { attribute_instance } tf_inout_declaration ;
      | { attribute_instance } tf_ref_declaration ;

task_port_list ::= task_port_item { , task_port_item }
      | list_of_port_identifiers { , task_port_item }

task_port_item ::=
        { attribute_instance } tf_input_declaration
      | { attribute_instance } tf_output_declaration
      | { attribute_instance } tf_inout_declaration
      | { attribute_instance } tf_ref_declaration ;
      | { attribute_instance } port_type list_of_tf_port_identifiers
      | { attribute_instance } tf_data_type list_of_tf_variable_identifiers

tf_input_declaration ::=
        input [ signing ] { packed_dimension } list_of_tf_port_identifiers
      | input tf_data_type list_of_tf_variable_identifiers

tf_output_declaration ::=
        output [ signing ] { packed_dimension } list_of_tf_port_identifiers
      | output tf_data_type list_of_tf_variable_identifiers

tf_inout_declaration ::=
        inout [ signing ] { packed_dimension } list_of_tf_port_identifiers
      | inout tf_data_type list_of_tf_variable_identifiers

tf_ref_declaration ::=
        [ const ] ref tf_data_type list_of_tf_variable_identifiers

tf_data_type ::=
        data_type
      | chandle

lifetime ::= static | automatic                                    // from Annex A.2.1

signing ::= signed | unsigned                                      // from Annex A.2.2.1
```

Syntax 10-1—Task syntax (excerpt from Annex A)

A Verilog task declaration either has the formal arguments in parentheses (like ANSI C) or in declarations and directions.

```
task mytask1 (output int x, input logic y);
```



```
   ...
endtask

task mytask2;
   output x;
   input y;
   int x;
   logic y;
   ...
endtask
```

Each formal argument has one of the following directions:

```
input    // copy value in at beginning
output   // copy value out at end
inout    // copy in at beginning and out at end
ref      // pass reference (see Section 10.5.2)
```

With SystemVerilog, there is a default direction of **input** if no direction has been specified. Once a direction is given, subsequent formals default to the same direction. In the following example, the formal arguments a and b default to inputs, and u and v are both outputs.

```
task mytask3(a, b, output logic [15:0] u, v);
   ...
endtask
```

Each formal argument also has a data type which can be explicitly declared or can inherit a default type. The task argument default type in SystemVerilog is **logic**.

SystemVerilog allows an array to be specified as a formal argument to a task. For example:

```
// the resultant declaration of b is input [3:0][7:0] b[3:0]
task mytask4(input [3:0][7:0] a, b[3:0], output [3:0][7:0] y[1:0]);
   ...
endtask
```

Verilog-2001 allows tasks to be declared as **automatic**, so that all formal arguments and local variables are stored on the stack. SystemVerilog extends this capability by allowing specific formal arguments and local variables to be declared as **automatic** within a static task, or by declaring specific formal arguments and local variables as **static** within an automatic task.

With SystemVerilog, multiple statements can be written between the task declaration and **endtask**, which means that the **begin** .... **end** can be omitted. If **begin** .... **end** is omitted, statements are executed sequentially, the same as if they were enclosed in a **begin** .... **end** group. It shall also be legal to have no statements at all.

In Verilog, a task exits when the endtask is reached. With SystemVerilog, the **return** statement can be used to exit the task before the **endtask** keyword.

10.3 Functions

```
function_data_type[8] ::=                                              // from Annex A.2.6
        integer_vector_type { packed_dimension } [ range ]
      | integer_atom_type
      | type_declaration_identifier { packed_dimension }
      | non_integer_type
      | struct [packed ] {{ struct_union_member } } { packed_dimension }
      | union [ packed ] { { struct_union_member } } { packed_dimension }
      | enum [ integer_type { packed_dimension } ]
        { enum_identifier [ = constant_expression ] { , enum_identifier [ = constant_expression ] } }
      | string
      | chandle
      | void
function_body_declaration ::=
        [ signing ] [ range_or_type ]
        [ interface_identifier . ] function_identifier ;
        { function_item_declaration }
        { function_statement_or_null }
        endfunction [ : function_identifier ]
      | [ signing ] [ range_or_type ]
        [ interface_identifier . ] function_identifier ( function_port_list ) ;
        { block_item_declaration }
        { function_statement_or_null }
        endfunction [ : function_identifier ]
function_declaration ::=
        function [ lifetime ] function_body_declaration
function_item_declaration ::=
        block_item_declaration
      | { attribute_instance } tf_input_declaration ;
      | { attribute_instance } tf_output_declaration ;
      | { attribute_instance } tf_inout_declaration ;
      | { attribute_instance } tf_ref_declaration ;
function_port_item ::=
        { attribute_instance } tf_input_declaration
      | { attribute_instance } tf_output_declaration
      | { attribute_instance } tf_inout_declaration
      | { attribute_instance } tf_ref_declaration
      | { attribute_instance } port_type list_of_tf_port_identifiers
      | { attribute_instance } tf_data_type list_of_tf_variable_identifiers
function_port_list ::= function_port_item { , function_port_item }
range_or_type ::=
        { packed_dimension } range
      | function_data_type
lifetime ::= static | automatic                                        // from Annex A.2.1
signing ::= signed | unsigned                                          // from Annex A.2.2.1
```

Syntax 10-2—Function syntax (excerpt from Annex A)

A Verilog function declaration either has the formal arguments in parentheses (like ANSI C) or in declarations and directions:

```
function logic [15:0] myfunc1(int x, int y);
   ...
endfunction

function logic [15:0] myfunc2;
   input int x;
   input int y;
   ...
endfunction
```

SystemVerilog extends Verilog functions to allow the same formal arguments as tasks. Function argument directions are:

```
input    // copy value in at beginning
output   // copy value out at end
inout    // copy in at beginning and out at end
ref      // pass reference (see Section 10.5.2)
```

Function declarations default to the formal direction **input** if no direction has been specified. Once a direction is given, subsequent formals default to the same direction. In the following example, the formal arguments `a` and `b` default to inputs, and `u` and `v` are both outputs:

```
function logic [15:0] myfunc3(int a, int b, output logic [15:0] u, v);
   ...
endfunction
```

Each formal argument has a data type which can be explicitly declared or can inherit a default type. The default type in SystemVerilog is **logic**, which is compatible with Verilog. SystemVerilog allows an array to be specified as a formal argument to a function, for example:

```
function [3:0][7:0] myfunc4(input [3:0][7:0] a, b[3:0]);
   ...
endfunction
```

SystemVerilog allows multiple statements to be written between the function header and **endfunction**, which means that the **begin**...**end** can be omitted. If the **begin**...**end** is omitted, statements are executed sequentially, as if they were enclosed in a **begin**...**end** group. It is also legal to have no statements at all, in which case the function returns the current value of the implicit variable that has the same name as the function.

10.3.1 Void functions

In Verilog, functions must return values. The return value is specified by assigning a value to the name of the function.

```
function [15:0] myfunc1 (input foo);
   myfunc1 = 16'hbeef; //return value is assigned to function name
endfunction
```

SystemVerilog allows functions to be declared as type **void**, which do not have a return value. For non-void functions, a value can be returned by assigning the function name to a value, as in Verilog, or by using **return** with a value. The **return** statement shall override any value assigned to the function name. When the return statement is used, non-void functions must specify an expression with the return.

```
function [15:0] myfunc2 (input foo);
```

```
   return 16'hbeef; //return value is specified using return statement
endfunction
```

In SystemVerilog, a function return can be a structure or union. In this case, a hierarchical name used inside the function and beginning with the function name is interpreted as a member of the return value. If the function name is used outside the function, the name indicates the scope of the whole function. If the function name is used within a hierarchical name, it also indicates the scope of the whole function.

Function calls are expressions unless of type **void**, which are statements:

```
a = b + myfunc1(c, d); //call myfunc1 (defined above) as an expression

myprint(a); //call myprint (defined below) as a statement

function void myprint (int a);
   ...
endfunction
```

10.3.2 Discarding function return values

In Verilog-2001, values returned by functions must be assigned or used in an expression. Calling a function as if it has no return value can result in a warning message. SystemVerilog allows using the **void** data type to discard a function's return value, which is done by casting the function to the **void** type:

```
void'(some_function());
```

10.4 Task and function scope and lifetime

In Verilog-2001, the default lifetime for tasks and functions is static. Automatic tasks and functions must be explicitly declared, using the automatic keyword.

SystemVerilog adds an optional qualifier to specify the default lifetime of all tasks and functions declared within a module, interface or program (see Section 16). The lifetime qualifier is **automatic** or **static**. The default lifetime is **static**.

```
program automatic test ;
   task foo( int a );   // arguments and variables in foo are automatic
      ...
   endtask
endmodule
```

Class methods are by default **automatic**, regardless of the lifetime attribute of the scope in which they are declared. Classes are discussed in Section 11.

10.5 Task and function argument passing

SystemVerilog provides two means for passing arguments to functions and tasks: by value and by reference. Arguments can also be passed by name as well as by position. Task and function arguments can also be given default values, allowing the call to the task or function to not pass arguments.

10.5.1 Pass by value

Pass by value is the default mechanism for passing arguments to subroutines, it is also the only one provided by Verilog-2001. This argument passing mechanism works by copying each argument into the subroutine area. If the subroutine is automatic, then the subroutine retains a local copy of the arguments in its stack. If the arguments are changed within the subroutine, the changes are not visible outside the subroutine. When the arguments are large, it can be undesirable to copy the arguments. Also, programs sometimes need to share a

common piece of data that is not declared global.

For example, calling the function bellow copies 1000 bytes each time the call is made.

```
function int crc( byte packet [1000:1] );
   for( int j= 0 1; j <= 1000; j++ ) begin
      crc ^= packet[j];
   end
endfunction
```

10.5.2 Pass by reference

Arguments passed by reference are not copied into the subroutine area, rather, a reference to the original argument is passed to the subroutine. The subroutine can then access the argument data via the reference. To indicate argument passing by reference, the argument declaration is preceded by the **ref** keyword. The general syntax is:

```
subroutine( ref type argument );
```

For example, the example above can be written as:

```
function int crc( ref byte packet [1000:1] );
   for( int j= 1; j <= 1000; j++ ) begin
      crc ^= packet[j];
   end
endfunction
```

Note that in the example, no change other than addition of the **ref** keyword is needed. The compiler knows that `packet` is now addressed via a reference, but users do not need to make these references explicit either in the callee or at the point of the call. That is, the call to either version of the `crc` function remains the same:

```
byte packet1[1000:1];
int k = crc( packet1 ); // pass by value or by reference: call is the same
```

When the argument is passed by reference, both the caller and the subroutine share the same representation of the argument, so any changes made to the argument either within the caller or the subroutine shall be visible to each other. The semantics of assignments to variables passed by reference is that changes are seen outside the subroutine immediately (before the subroutine returns). Only variables, not nets, can be passed by reference.

Arguments passed by reference must match exactly, no promotion, conversion, or auto-casting is possible when passing arguments by reference. In particular, array arguments must match their type and all dimensions exactly. Fixed-size arrays cannot be mixed with dynamic arrays and vice-versa.

Passing an argument by reference is a unique argument passing qualifier, different from **input**, **output**, or **inout**. Combining **ref** with any other qualifier is illegal. For example, the following declaration results in a compiler error:

```
task incr( ref input int a ); // incorrect: ref cannot be qualified
```

A **ref** argument is similar to an **inout** argument except that an **inout** argument is copied twice: once from the actual into the argument when the subroutine is called and once from the argument into the actual when the subroutine returns. Passing object handles are no exception and have similar semantics when passed as **ref** or **inout** arguments, thus, a **ref** of an object handle allows changes to the object handle (for example assigning a new object) in addition to modification of the contents of the object.

To protect arguments passed by reference from being modified by a subroutine, the const qualifier can be used together with **ref** to indicate that the argument, although passed by reference, is a read-only variable.

```
task show ( const ref byte [] data );
```

```
    for ( int j = 0; j < data.size ; j++ )
       $display( data[j] ); // data can be read but not written
endtask
```

When the formal argument is declared as a **const ref**, the subroutine cannot alter the variable, and an attempt to do so shall generate compiler error.

10.5.3 Default argument values

To handle common cases or allow for unused arguments, SystemVerilog allows a subroutine declaration to specify a default value for each singular argument.

The syntax to declare a default argument in a subroutine is:

```
subroutine( type argument = default_value );
```

The `default_value` is any expression that is visible at the current scope. It can include any combination of constants or variables visible at the scope of both the caller and the subroutine.

When the subroutine is called, arguments with default values can be omitted from the call and the compiler shall insert their corresponding values. Unspecified (or empty) arguments can be used as placeholders for default arguments, allowing the use of non-consecutive default arguments. If an unspecified argument is used for an argument that does not have a default value, a compiler error shall be issued.

```
task read(int j = 0, int k, int data = 1 );
...
endtask;
```

This example declares a task `read()` with two default arguments, `j` and `data`. The task can then be called using various default arguments:

```
read( , 5 );          // is equivalent to  read( 0, 5, 1 );
read( 2, 5 );         // is equivalent to  read( 2, 5, 1 );
read( , 5, );         // is equivalent to  read( 0, 5, 1 );
read( , 5, 7 );       // is equivalent to  read( 0, 5, 7 );
read( 1, 5, 2 );      // is equivalent to  read( 1, 5, 2 );
read( );              // error; k has no default value
```

10.5.4 Argument passing by name

SystemVerilog allows arguments to tasks and functions to be passed by name as well as by position. This allows specifying non-consecutive default arguments and easily specifying the argument to be passed at the call. For example:

```
function int fun( int j = 1, string s = "no" );
   ...
endfunction
```

The fun function can be called as follows:

```
fun( .j(2), .s("yes") );      // fun( 2, "yes" );
fun( .s("yes") );             // fun( 1, "yes" );
fun( , "yes" );               // fun( 1, "yes" );
fun( .j(2) );                 // fun( 2, "no" );
fun( 2 );                     // fun( 2, "no" );
fun( );                       // fun( 1, "no" );
```

If the arguments have default values, they are treated like parameters to module instances. If the arguments do not have a default, then they must be given or the compiler shall issue an error.

If both positional and named arguments are specified in a single subroutine call, then all the positional arguments must come before the named arguments. Then, using the same example as above:

```
fun( .s("yes"), 2 );       // illegal
fun( 2, .s("yes") );       // OK
```

10.5.5 Optional argument list

When a task or function specifies no arguments, the empty parenthesis, `()`, following the task/function name shall be optional. This is also true for tasks or functions that require arguments, when all arguments have defaults specified.

10.6 Import and export functions

The syntax for the import and export of functions is:

```
dpi_import_export ::=                                          // from Annex A.2.6
        import "DPI" [ dpi_import_property ] [ c_identifier = ] dpi_function_proto
      | export "DPI" [ c_identifier = ] function function_identifier
dpi_import_property ::= context | pure
dpi_function_proto ::=
        named_function_proto
      | [ signing ] function_data_type function_identifier ( list_of_dpi_proto_formals )
list_of_dpi_proto_formals ::=
        [ { attribute_instance } dpi_proto_formal { , { attribute_instance } dpi_proto_formal } ]
dpi_proto_formal ::=
        data_type [ port_identifier dpi_dimension { , port_identifier dpi_dimension } ]
```

Syntax 10-3—Import and export syntax (excerpt from Annex A)

In both **`import`** and **`export`**, *c_identifier* is the name of the foreign function (import/export), *function_identifier* is the SystemVerilog name for the same function. If *c_identifier* is not explicitly given, it shall be the same as the SystemVerilog function *function_identifier*. An error shall be generated if and only if the *c_identifier* has characters that are not valid in a C function identifier.

Several SystemVerilog functions can be mapped to the same foreign function by supplying the same *c_identifier* for several *fnames*. Note that all these SystemVerilog functions must have identical argument types, as defined in the next paragraph.

For any given *c_identifier*, all declarations, regardless of scope, must have exactly the same function signature. The function signature includes the return type, the number, order, direction and types of each and every argument. Each type includes dimensions and bounds of any arrays/array dimensions. For **`import`** declarations, arguments can be open arrays. Open arrays are defined in Section 26.4.6.1. The signature also includes the **`pure`**/**`context`** qualifiers that can be associated with an import definition.

Only one **`import`** or **`export`** declaration of a given *function_identifier* shall be permitted in any given scope. More specifically, for an **`import`**, the import must be the sole declaration of *function_identifier* in the given scope. For an **`export`**, the function must be declared in the scope where the export occurs and there must be only one export of that *function_identifier* in that scope.

For exported functions, the exported function must be declared in the same scope that contains the **`export`** `"DPI"` declaration. Only SystemVerilog functions can be exported (specifically, this excludes exporting a class method)

Note that **`import`** `"DPI"` functions declared this way can be invoked by hierarchical reference the same as any normal SystemVerilog function. Declaring a SystemVerilog function to be exported does not change the semantics or behavior of this function from the SystemVerilog perspective (i.e. there is no effect in SystemVerilog usage other than making this exported function also accessible to C callers).

Only non-void functions with no **`output`** or **`inout`** arguments can be specified as **`pure`**. Functions specified as pure in their corresponding SystemVerilog external declarations shall have no side effects; their results need to depend solely on the values of their input arguments. Calls to such functions can be removed by SystemVerilog compiler optimizations or replaced with the values previously computed for the same values of the input arguments.

Specifically, a pure function is assumed to not directly or indirectly (i.e., by calling other functions):

— Perform any file operations

— Read or write anything in the broadest possible meaning, including I/O, environment variables, objects from the operating system, or from the program or other processes, shared memory, sockets, etc.

— Access any persistent data, like global or static variables.

If a pure function does not obey the above restrictions, SystemVerilog compiler optimizations can lead to unexpected behavior, due to eliminated calls or incorrect results being used.

An unqualified imported function can have side effects but cannot read or modify any SystemVerilog signals other than those provided through its arguments. Unqualified imports shall not be permitted to invoke exported SystemVerilog functions.

Imported functions with the **`context`** qualifier can invoke exported SystemVerilog functions, can read or write to SystemVerilog signals other than those passed through their arguments, either through the use of other interfaces or as a side effect of invoking exported SystemVerilog functions. Context functions shall always implicitly be supplied a scope representing the fully qualified instance name within which the import declaration was present (i.e. an import function always runs in the instance in which the import declaration occurred). This is the same semantics as SystemVerilog functions, which also run in the scope they are defined, rather than in the scope of the caller.

Import context functions can have side effects and can use other SystemVerilog interfaces (including but not limited to VPI). However, note that declaring an import context function does not automatically make any other simulator interface available. For VPI access (or any other interface access) to be possible, the appropriate implementation-defined mechanism must still be used to enable these interface(s). Note also that DPI calls do not automatically create or provide any handles or any special environment that might be needed by those other interfaces. It shall be the user's responsibility to create, manage or otherwise manipulate the required handles/environment(s) needed by the other interfaces. The `svGetScopeName()` and related functions exist to provide a name based linkage from DPI to other interfaces. Exported functions can only be invoked if the current DPI context refers to an instance in which the named function is defined.

To access functions defined in any other scope, including `$root`, the foreign code shall have to change DPI context appropriately. Attempting to invoke an exported SystemVerilog function from a scope in which it is not directly visible shall result in a runtime error. How such errors are handled shall be implementation dependent. If an imported function needs to invoke an exported function that is not visible from the current scope, it needs to change, via svSetScope, the current scope to a scope that does have visibility to the exported function. This is conceptually equivalent to making a hierarchically qualified function call in SystemVerilog. The current SystemVerilog context shall be preserved across a call to an exported function, even if current context has been modified by an application. Note that context is not defined for non-context imports and attempting to use any functionality depending on context from non-context imports can lead to unpredictable behavior.

Section 11
Classes

11.1 Introduction (informative)

SystemVerilog introduces an object-oriented **`class`** data abstraction. Classes allow objects to be dynamically created, deleted, assigned, and accessed via object handles. Object handles provide a safe pointer-like mechanism to the language. Classes offer inheritance and abstract type modeling, which brings the advantages of C function pointers with none of the type-safety problems, thus, bringing true polymorphism into Verilog.

11.2 Syntax

```
class_declaration ::=                                          // from Annex A.1.3
        { attribute_instance } [ virtual ] class [ lifetime ] class_identifier [ parameter_port_list ]
        [ extends class_identifier ] ; [ timeunits_declaration ] { class_item }
        endclass [ : class_identifier]

class_item ::=                                                 // from Annex A.1.8
        { attribute_instance } class_property
      | { attribute_instance } class_method
      | { attribute_instance } class_constraint

class_property ::=
        { property_qualifier } data_declaration
      | const { class_item_qualifier } data_type const_identifier [ = constant_expression ] ;

class_method ::=
        { method_qualifier } task_declaration
      | { method_qualifier } function_declaration
      | extern { method_qualifier } method_prototype

class_constraint ::=
        constraint_prototype
      | constraint_declaration

class_item_qualifier[11] ::=
        static
      | protected
      | local

property_qualifier[11] ::=
        rand
      | randc
      | class_item_qualifier

method_qualifier[11] ::=
        virtual
      | class_item_qualifier

method_prototype ::=
        task named_task_proto ;
      | function named_function_proto ;

extern_method_declaration ::=
        function [ lifetime ] class_identifier :: function_body_declaration
      | task [ lifetime ] class_identifier :: task_body_declaration
```

Syntax 11-1—Class syntax (excerpt from Annex A)

11.3 Overview

A ***class*** is a type that includes data and subroutines (functions and tasks) that operate on that data. A class's data is referred to as *properties*, and its subroutines are called *methods*, both are members of the class. The properties and methods, taken together, define the contents and capabilities of some kind of object.

For example, a packet might be an object. It might have a command field, an address, a sequence number, a time stamp, and a packet payload. In addition, there are various things than can be done with a packet: initialize the packet, set the command, read the packet's status, or check the sequence number. Each Packet is different, but as a class, packets have certain intrinsic properties that can be captured in a definition.

```
class Packet ;
   //data or class properties
   bit [3:0] command;
   bit [40:0] address;
   bit [4:0] master_id;
   integer time_requested;
   integer time_issued;
   integer status;

   // initialization
   function new();
      command = IDLE;
      address = 41'b0;
      master_id = 5'bx;
   endfunction

   // methods
   // public access entry points
   task clean();
      command = 0; address = 0; master_id = 5'bx;
   endtask

   task issue_request( int delay );
      // send request to bus
   endtask

   function integer current_status();
      current_status = status;
   endfunction
endclass
```

A common convention is to capitalize the first letter of the class name, so that it is easy to recognize class declarations.

11.4 Objects (class instance)

A class defines a data type. An object is an instance of that class. An object is used by first declaring a variable of that class type (that holds an object handle) and then creating an object of that class (using the `new` function) and assigning it to the variable.

```
Packet p; // declare a variable of class Packet
p = new;  // initialize variable to a new allocated object of the class Packet
```

The variable `p` is said to hold an object handle to an object of class `Packet`.

Uninitialized object handles are set by default to the special value **`null`.** An uninitialized object can be detected by comparing its handle with **`null`**.

For example: The task `task1` below checks if the object is initialized. If it is not, it creates a new object via the `new` command.

```
class obj_example;
      ...
endclass

task task1(integer a, obj_example myexample);
   if (myexample == null) myexample = new;
endtask
```

Accessing non-static members (Section 11.8) or virtual methods (Section 11.19) via a **`null`** object handle is illegal. The result of an illegal access via a null object is indeterminate, and implementations can issue an error.

SystemVerilog objects are referenced using an *object handle*. There are some differences between a C pointer and a SystemVerilog object handle. C pointers give programmers a lot of latitude in how a pointer can be used. The rules governing the usage of SystemVerilog object handles are much more restrictive. A C pointer can be incremented for example, but a SystemVerilog object handle cannot. In addition to object handles, Section 3.6 introduces the **`chandle`** data type for use with the DPI Direct Programming Interface (see Section 26).

Table 11-1: Comparison of pointer and handle types

Operation	C pointer	SV object handle	SV chandle
Arithmetic operations (such as incrementing)	Allowed	Not allowed	Not allowed
For arbitrary data types	Allowed	Not allowed	Not allowed
Dereference when null	Error	Not allowed	Not allowed
Casting	Allowed	Limited	Not allowed
Assignment to an address of a data type	Allowed	Not allowed	Not allowed
Unreferenced objects are garbage collected	No	Yes	Yes
Default value	Undefined	**`null`**	**`null`**
For classes	(C++)	Allowed	Not allowed

11.5 Object properties

The data fields of an object can be used by qualifying property names with an instance name. Using the earlier example, the commands for the `Packet` object `p` can be used as follows:

```
Packet p = new;
p.command = INIT;
p.address = $random;
packet_time = p.time_requested;
```

Any data-type can be declared as a class property, except for net types since they are incompatible with dynamically allocated data.

11.6 Object methods

An object's methods can be accessed using the same syntax used to access properties:

```
Packet p = new;
status = p.current_status();
```

Note that the assignment to `status` is not:

```
status = current_status(p);
```

The focus in object-oriented programming is the object, in this case the packet, not the function call. Also, objects are self-contained, with their own methods for manipulating their own properties. So the object doesn't have to be passed as an argument to `current_status()`. A class' properties are freely and broadly available to the methods of the class, but each method only accesses the properties associated with its object, i.e., its instance.

11.7 Constructors

SystemVerilog does not require the complex memory allocation and deallocation of C++. Construction of an object is straightforward and garbage collection, as in Java, is implicit and automatic. There can be no memory leaks or other subtle behavior that is so often the bane of C++ programmers.

SystemVerilog provides a mechanism for initializing an instance at the time the object is created. When an object is created, for example

```
Packet p = new;
```

The system executes the `new` function associated with the class:

```
class Packet;
   integer command;

   function new();
      command = IDLE;
   endfunction
endclass
```

Note that `new` is now being used in two very different contexts with very different semantics. The variable declaration creates an object of class `Packet`. In the course of creating this instance, the `new` function is invoked, in which any specialized initialization required can be done. The `new` function is also called the *class constructor.*

The `new` operation is defined as a function with no return type, and like any other function, it must be non-blocking. Even though `new` does not specify a return type, the left-hand side of the assignment determines the return type.

Every class has a default (built-in) `new` method. The default constructor first calls its parent class constructor (**super**.new() as described in Section 11.14) and then proceeds to initialize each member of the current object to its default (or uninitialized value).

It is also possible to pass arguments to the constructor, which allows run-time customization of an object:

```
Packet p = new(STARTUP, $random, $time);
```

where the `new` initialization task in `Packet` might now look like:

```
function new(int cmd = IDLE, bit[12:0] adrs = 0, int cmd_time );
```



```
      command = cmd;
      address = adrs;
      time_requested = cmd_time;
   endfunction
```

The conventions for arguments are the same as for any other procedural subroutine calls, such as the use of default arguments.

11.8 Static properties

The previous examples have only declared instance properties. Each instance of the class (i.e., each object of type `Packet`), has its own copy of each of its six variables. Sometimes only one version of a variable is required to be shared by all instances. These class properties are created using the keyword **`static`**. Thus, for example, in a case where all instances of a class need access to a common file descriptor:

```
class Packet ;
   static integer fileId = $fopen( "data", "r" );
```

Now, `fileID` shall be created and initialized once. Thereafter, every Packet object can access the file descriptor in the usual way:

```
Packet p;
c = $fgetc( p.fileID );
```

11.9 Static methods

Methods can be declared as **`static`**. A static method is subject to all the class scoping and access rules, but behaves like a regular subroutine that can be called outside the class, even with no class instantiation. A static method has no access to non-static members (properties or methods), but it can directly access static class properties or call static methods of the same class. Access to non-static members or to the special this handle within the body of a static method is illegal and results in a compiler error. Static methods cannot be virtual.

```
class id;
   static int current = 0;
   static function int next_id();
      next_id = ++current; // OK to access static class property
   endfunction
endclass
```

A static method is different from a method with static lifetime. The former refers to the lifetime of the method within the class, while the latter refers to the lifetime of the arguments and variables within the task.

```
class TwoTasks;
   static task foo(); ... endtask   // static class method with
                                    // automatic variable lifetime

   task static bar(); ... endtask   // non-static class method with
                                    // static variable lifetime
endclass
```

By default, class methods have automatic lifetime for their arguments and variables.

11.10 This

The **`this`** keyword is used to unambiguously refer to properties or methods of the current instance. The **`this`** keyword denotes a predefined object handle that refers to the object that was used to invoke the subroutine that

this is used within. The **this** keyword shall only be used within non-static class methods, otherwise an error shall be issued. For example, the following declaration is a common way to write an initialization task:

```
class Demo ;
   integer x;

   function new (integer x)
      this.x = x;
   endfunction
endclass
```

The x is now both a property of the class and an argument to the function new. In the function new, an unqualified reference to x shall be resolved by looking at the innermost scope, in this case the subroutine argument declaration. To access the instance property, it is qualified with the **this** keyword, to refer to the current instance.

Note that in writing methods, members can be qualified with **this** to refer to the current instance, but it is usually unnecessary.

11.11 Assignment, re-naming and copying

Declaring a class variable only creates the name by which the object is known. Thus:

```
Packet p1;
```

creates a variable, p1, that can hold the handle of an object of class Packet, but the initial value of p1 is **null**. The object does not exist, and p1 does not contain an actual handle, until an instance of type Packet is created:

```
p1 = new;
```

Thus, if another variable is declared and assigned the old handle, p1, to the new one, as in:

```
Packet p2;
p2 = p1;
```

then there is still only one object, which can be referred to with either the name p1 or p2. Note, new was executed only once, so only one object has been created.

If, however, the example above is re-written as shown below, a copy of p1 shall be made:

```
Packet p1;
Packet p2;
p1 = new;
p2 = new p1;
```

The last statement has new executing a second time, thus creating a new object p2, whose properties are copied from p1. This is known as a *shallow copy.* All of the variables are copied across: integers, strings, instance handles, etc. Objects, however, are not copied, only their handles; as before, two names for the same object have been created. This is true even if the class declaration includes the instantiation operator new:

```
class A ;
   integer j = 5;
endclass

class B ;
   integer i = 1;
   A a = new;
```



```
endclass

function integer test;
   B b1 = new;       // Create an object of class B
   B b2 = new b1;    // Create an object that is a copy of b1
   b2.i = 10;        // i is changed in b2, but not in b1
   b2.a.j = 50;      // change a.j, shared by both b1 and b2
   test = b1.i;      // test is set to 1 (b1.i has not changed)
   test = b1.a.j;    // test is set to 50 (a.j has changed)
endfunction
```

Several things are noteworthy. First, properties and instantiated objects can be initialized directly in a class declaration. Second, the shallow copy does not copy objects. Third, instance qualifications can be chained as needed to reach into objects or to reach through objects:

```
b1.a.j                  // reaches into a, which is a property of b1
p.next.next.next.val    // chain through a sequence of handles to get to val
```

To do a full (deep) copy, where everything (including nested objects) are copied, custom code is typically needed. For example:

```
Packet p1 = new;
Packet p2 = new;
p2.copy(p1);
```

where `copy(Packet p)` is a custom method written to copy the object specified as its argument into its instance.

11.12 Inheritance and subclasses

The previous sections defined a class called `Packet`. This class can be extended so that the packets can be chained together into a list. One solution would be to create a new class called `LinkedPacket` that contains a variable of type Packet called `packet_c`.

To refer to a property of `Packet`, the variable `packet_c` needs to be referenced.

```
class LinkedPacket;
   Packet packet_c;
   LinkedPacket next;

   function LinkedPacket get_next();
      get_next = next;
   endfunction
endclass
```

Since `LinkedPacket` is a specialization of `Packet`, a more elegant solution is to extend the class creating a new subclass that *inherits* the members of the parent class. Thus, for example:

```
class LinkedPacket extends Packet;
   LinkedPacket next;

   function LinkedPacket get_next();
      get_next = next;
   endfunction
endclass
```

Now, all of the methods and properties of `Packet` are part of `LinkedPacket`—as if they were defined in `LinkedPacket`—and `LinkedPacket` has additional properties and methods.

The parent's methods can also be overridden, changing their definitions.

The mechanism provided by SystemVerilog is called *Single-Inheritance*, that is, each class is derived from a single parent class.

11.13 Overridden members

Subclass objects are also legal representative objects of their parent classes. For example, every `LinkedPacket` object is a perfectly legal `Packet` object.

The handle of a `LinkedPacket` object can be assigned to a `Packet` variable:

```
LinkedPacket lp = new;
Packet p = lp;
```

In this case, references to `p` access the methods and properties of the `Packet` class. So, for example, if properties and methods in `LinkedPacket` are overridden, these overridden members referred to through `p` get the original members in the `Packet` class. From `p`, `new` and all overridden members in `LinkedPacket` are now hidden.

```
class Packet;
   integer i = 1;
   function integer get();
      get = i;
   endfunction
endclass

class LinkedPacket extends Packet;
   integer i = 2;
   function integer get();
      get = -i;
   endfunction
endclass

LinkedPacket lp = new;
Packet p = lp;
j = p.i;              // j = 1, not 2
j = p.get();          // j = 1, not -1 or –2
```

To call the overridden method via a parent class object (`p` in the example), the method needs to be declared **`virtual`** (see Section 11.19).

11.14 Super

The **`super`** keyword is used from within a derived class to refer to members of the parent class. It is necessary to use **`super`** to access members of a parent class when those members are overridden by the derived class.

```
class Packet;                          //parent class
   integer value;
   function integer delay();
      delay = value * value;
   endfunction
endclass

class LinkedPacket extends Packet;     //derived class
   integer value;
```



```
    function integer delay();
       delay = super.delay()+ value * super.value;
    endfunction
endclass
```

The member can be a member declared a level up or be inherited by the class one level up. There is no way to reach higher (for example, `super.super.count` is not allowed).

Subclasses (or derived classes) are classes that are extensions of the current class. Whereas superclasses (parent classes or base classes) are classes that the current class is extended from, beginning with the original base class.

Note: When using the **`super`** within `new`, **`super`**`.new` must be the first executable statement in the constructor. This is because the superclass must be initialized before the current class and if the user code doesn't provide an initialization, the compiler shall insert a call to **`super`**`.new` automatically.

11.15 Casting

It is always legal to assign a subclass variable to a variable of a class higher in the inheritance tree. It is never legal to directly assign a superclass variable to a variable of one of its subclasses. However, it is legal to assign a superclass handle to a subclass variable if the superclass handle refers to an object of the given subclass.

To check if the assignment is legal, the dynamic cast function `$cast()` is used (see Section 3.15).

The syntax for `$cast()` is:

```
task $cast( singular dest_handle, singular source_handle );
```

or

```
function int $cast( singular dest_handle, singular source_handle );
```

When used with object handles, `$cast()` checks the hierarchy tree (super and subclasses) of the `source_expr` to see if it contains the class of *dest_handle*. If it does, `$cast()` does the assignment. Otherwise the error handling is as described in Section 3.15.

11.16 Chaining constructors

When a subclass is instantiated, the class method `new()` is invoked. The first action `new()` takes, before any code defined in the function is evaluated, is to invoke the `new()` method of its superclass, and so on up the inheritance hierarchy. Thus, all the constructors are called, in the proper order, beginning with the root base class and ending with the current class.

If the initialization method of the superclass requires arguments, there are two choices. To always supply the same arguments, or to use the **`super`** keyword. If the arguments are always the same, then they can be specified at the time the class is extended:

```
class EtherPacket extends Packet(5);
```

This passes 5 to the `new` routine associated with `Packet`.

A more general approach is to use the **`super`** keyword, to call the superclass constructor:

```
function new();
   super.new(5);
endfunction
```

To use this approach, `super.new(…)` must be the first executable statement in the function `new`.

11.17 Data hiding and encapsulation

So far, all class properties and methods have been made available to the outside world without restriction. Often, it is desirable to restrict access to properties and methods from outside the class by hiding their names. This keeps other programmers from relying on a specific implementation, and it also protects against accidental modifications to properties that are internal to the class. When all data becomes hidden—being accessed only by public methods—testing and maintenance of the code becomes much easier.

In SystemVerilog, unqualified properties and methods are public, available to anyone who has access to the object's name.

A member identified as **`local`** is available only to methods inside the class. Further, these local members are not visible within subclasses. Of course, non-local methods that access local properties or methods can be inherited, and work properly as methods of the subclass.

A **`protected`** property or method has all of the characteristics of a **`local`** member, except that it can be inherited; it is visible to subclasses.

Note that within the class, a local method or property of the class can be referenced, even if it is in a different instance. For example:

```
class Packet;
   local integer i;
   function integer compare (Packet other);
      compare = (this.i == other.i);
   endfunction
endclass
```

A strict interpretation of encapsulation might say that `other.i` should not be visible inside of this packet, since it is a local property being referenced from outside its instance. Within the same class, however, these references are allowed. In this case, `this.i` shall be compared to `other.i` and the result of the logical comparison returned.

Class members can be identified as either **`local`** or **`protected`**; properties can be further defined as **`const`**, and methods can be defined as **`virtual`**. There is no predefined ordering for specifying these modifiers; however, they can only appear once per member. It shall be an error to define members to be both **`local`** and **`protected`**, or to duplicate any of the other modifiers.

11.18 Constant Properties

Class properties can be made read-only by a **`const`** declaration like any other SystemVerilog variable. However, because class objects are dynamic objects, class properties allow two forms of read-only variables: global constants and instance constants.

Global constant properties are those that include an initial value as part of their declaration. They are similar to other **`const`** variables in that they cannot be assigned a value anywhere other than in the declaration.

```
class Jumbo_Packet;
   const int max_size = 9 * 1024; // global constant
   byte payload [];
   function new( int size );
      payload = new[ size > max_size ? max_size : size ];
   endfunction
endclass
```

Instance constants do not include an initial value in their declaration, only the const qualifier. This type of constant can be assigned a value at run-time, but the assignment can only be done once in the corresponding class constructor.


```
class Big_Packet;
   const int size; // instance constant
   byte payload [];
   function new();
      size = $random % 4096; //one assignment in new -> ok
      payload = new[ size ];
   endfunction
endclass
```

Typically, global constants are also declared **static** since they are the same for all instances of the class. However, an instance constant cannot be declared **static**, since that would disallow all assignments in the constructor.

11.19 Abstract classes and virtual methods

A set of classes can be created that can be viewed as all being derived from a common base class. For example, a common base class of type `BasePacket` that sets out the structure of packets but is incomplete would never be instantiated. From this base class, though, a number of useful subclasses could be derived, such as Ethernet packets, token ring packets, GPSS packets, satellite packets. Each of these packets might look very similar, all needing the same set of methods, but they could vary significantly in terms of their internal details.

A base class sets out the prototype for the subclasses. Since the base class is not intended to be instantiated, it can be made *abstract* by specifying the class to be **virtual**:

```
virtual class BasePacket;
```

Abstract classes can also have *virtual* methods. Virtual methods provide prototypes for subroutines, all of the information generally found on the first line of a method declaration: the encapsulation criteria, the type and number of arguments, and the return type if it is needed. Later, when subclasses override virtual methods, they must follow the prototype exactly. Thus, all versions of the virtual method look identical in all subclasses:

```
virtual class BasePacket;
   virtual protected function integer send(bit[31:0] data);
   endfunction
endclass

class EtherPacket extends BasePacket;
   protected function integer send(bit[31:0] data);
      // body of the function
      ...
   endfunction
endclass
```

EtherPacket is now a class that can be instantiated. In general, if an abstract class has any virtual methods, all of the methods must be overridden (and provided with a method body) for the subclass to be instantiated. If any virtual methods have no implementation, the subclass needs to be abstract.

An abstract class can contain methods for which there is only a prototype and no implementation (i.e., an incomplete class). An abstract class cannot be instantiated, it can only be derived. Methods of normal classes can also be declared virtual. In this case, the method must have a body. If the method does have a body, then the class can be instantiated, as can its subclasses.

11.20 Polymorphism: dynamic method lookup

Polymorphism allows the use of a variable in the superclass to hold subclass objects, and to reference the methods of those subclasses directly from the superclass variable. As an example, assume the base class for the

`Packet` objects, `BasePacke`t defines, as virtual functions, all of the public methods that are to be generally used by its subclasses, methods such as send, receive, print, etc. Even though `BasePacket` is abstract, it can still be used to declare a variable:

```
BasePacket packets[100];
```

Now, instances of various packet objects can be created, and put into the array:

```
EtherPacket ep = new;   // extends BasePacket
TokenPacket tp = new;   // extends BasePacket
GPSSPacket gp = new;    // extends EtherPacket
packets[0] = ep;
packets[1] = tp;
packets[2] = gp;
```

If the data types were, for example, integers, bits and strings, all of these types could not be stored into a single array, but with *polymorphism*, it can be done. In this example, since the methods were declared as **`virtual`**, the appropriate subclass methods can be accessed from the superclass variable, even though the compiler didn't know—at compile time—what was going to be loaded into it.

For example, `packets[1]`:

```
packets[1].send();
```

shall invoke the `send` method associated with the `TokenPacket` class. At run-time, the system correctly binds the method from the appropriate class.

This is a typical example of polymorphism at work, providing capabilities that are far more powerful than what is found in a non-object-oriented framework.

11.21 Class scope resolution operator ::

The class scope operator `::` is used to specify an identifier defined within the scope of a class. It has the following form:

```
class_identifier :: { class_identifier :: } identifier
```

Identifiers on the left side of the scope-resolution operator (**`::`**) can be only class names.

Because classes and other scopes can have the same identifiers, the scope resolution operator uniquely identifies a member of a particular class. In addition, to disambiguating class scope identifiers, the `::` operator also allows access to static members (properties and methods) from outside the class, as well as access to public or protected elements of a super-classes from within the derived classes.

```
class Base;
   typedef enum {bin,oct,dec,hex} radix;
   static task print( radix r, integer n ); ... endtask
endclass
...
Base b = new;
int bin = 123;
b.print( Base::bin, bin );    // Base::bin and bin are different
Base::print( Base::hex, 66 );
```

In SystemVerilog, the class scope operator applies to all static elements of a class: static class properties, static methods, typedefs, enumerations, structures, unions, and nested class declarations. Class-scope resolved expressions can be read (in expressions), written (in assignments or subroutines calls) or triggered off (in event expressions). They can also be used as the name of a type or a method call.

Like modules, classes are scopes and can nest. Nesting allows hiding of local names and local allocation of resources. This is often desirable when a new type is needed as part of the implementation of a class. Declaring types within a class helps prevent name collisions, and cluttering the outer scope with symbols that are used only by that class. Type declarations nested inside a class scope are public and can be accessed outside the class.

```
class StringList;
   class Node; // Nested class for a node in a linked list.
      string name;
      Node link;
   endclass
endclass

class StringTree;
   class Node; // Nested class for a node in a binary tree.
      string name;
      Node left, right;
   endclass
endclass
// StringList::Node is different from StringTree::Node
```

The scope resolution operator enables:

— Access to static public members (methods and properties) from outside the class hierarchy.

— Access to public or protected class members of a super-class from within the derived classes.

— Access to type declarations and enumeration labels declared inside the class from outside the class hierarchy or from within derived classes.

11.22 Out of block declarations

It is convenient to be able to move method definitions out of the body of the class declaration. This is done in two steps. Declare, within the class body, the method prototypes—whether it is a function or task, any qualifiers (**`local`**, **`protected`** or **`virtual`**), and the full argument specification plus the **`extern`** qualifier. The **`extern`** qualifier indicates that the body of the method (its implementation) is to be found outside the declaration. Then, outside the class declaration, declare the full method—like the prototype but without the qualifiers—and, to tie the method back to its class, qualify the method name with the class name and a pair of colons:

```
class Packet;
   Packet next;
   function Packet get_next();// single line
      get_next = next;
   endfunction

   // out-of-body (extern) declaration
   extern protected virtual function int send(int value);
endclass

function int Packet::send(int value);
   // dropped protected virtual, added Packet::
   // body of method
      ...
endfunction
```

The out of block method declaration must match the prototype declaration exactly; the only syntactical difference is that the method name is preceded by the class name and scope operator (`::`).

11.23 Parameterized classes

It is often useful to define a generic class whose objects can be instantiated to have different array sizes or data types. This avoids writing similar code for each size or type, and allows a single specification to be used for objects that are fundamentally different, and (like a templated class in C++) not interchangeable.

The normal Verilog parameter mechanism is used to parameterize a class:

```
class vector #(parameter int size = 1;);
   bit [size-1:0] a;
endclass
```

Instances of this class can then be instantiated like modules or interfaces:

```
   vector #(10) vten;         // object with vector of size 10
   vector #(.size(2)) vtwo;   // object with vector of size 2
   typedef vector#(4) Vfour;  // Class with vector of size 4
```

This feature is particularly useful when using types as parameters:

```
class stack #(parameter type T = int;);
   local T items[];
   task push( T a ); ... endtask
   task pop( ref T a ); ... endtask
endclass
```

The above class defines a generic *stack* class that can be instantiated with any arbitrary type:

```
stack is;                  // default: a stack of int's
stack#(bit[1:10]) bs;      // a stack of 10-bit vector
stack#(real) rs;           // a stack of real numbers
```

Any type can be supplied as a parameter, including a user-defined type such as a **`class`** or **`struct`**.

The combination of a generic class and the actual parameter values is called a specialization (or variant). Each specialization of a class has a separate set of **`static`** member variables (this is consistent with C++ templated classes). To share static member variables among several class specializations, they must be placed in a non-parameterized base class.

```
class vector #(parameter 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
```

The variable `count` in the example above can only be accessed by the corresponding `disp_count` method. Each specialization of the class *vector* has its own unique copy of `count`.

To avoid having to repeat the specialization either in the declaration or to create parameters of that type, a **`typedef`** should be used:

```
typedef vector#(4) Vfour;
typedef stack#(Vfour) Stack4;
Stack4 s1, s2;             // declare objects of type Stack4
```


11.24 Typedef class

Sometimes a class variable needs to be declared before the class itself has been declared. For example, if two classes each need a handle to the other. When, in the course of processing the declaration for the first class, the compiler encounters the reference to the second class, that reference is undefined and the compiler flags it as an error.

This is resolved using **`typedef`** to provide a forward declaration for the second class:

```
typedef class C2;       // C2 is declared to be of type class
class C1
      C2 c;
endclass
class C2
   C1 c;
endclass
```

In this example, `C2` is declared to be of type **`class`**, a fact that is re-enforced later in the source code. Note that the **`class`** construct always creates a type, and does not require a **`typedef`** declaration for that purpose (as in **`typedef class`** …). This is consistent with common C++ use.

Note that the `class` keyword in the statement `typedef class C2;` is not necessary, and is used only for documentation purposes. The statement `typedef C2;` is equivalent and shall work the same way.

11.25 Classes, structures, and unions

SystemVerilog adds the object-oriented **`class`** construct. On the surface, it might appear that **`class`** and **`struct`** provide equivalent functionality, and only one of them is needed. However, that is not true; **`class`** differs from **`struct`** in four fundamental ways:

1) SystemVerilog **`struct`** are strictly static objects; they are created either in a static memory location (global or module scope) or on the stack of an automatic task. Conversely, SystemVerilog objects (i.e., class instances) are exclusively dynamic, their declaration doesn't create the object; that is done by calling `new`.

2) SystemVerilog structs are type compatible so long as their bit sizes are the same, thus copying structs of different composition but equal sizes is allowed. In contrast, SystemVerilog objects are strictly strongly-typed. Copying an object of one type onto an object of another is not allowed.

3) SystemVerilog objects are implemented using handles, thereby providing C-like pointer functionality. But, SystemVerilog disallows casting handles onto other data types, thus, unlike C, SystemVerilog handles are guaranteed to be safe.

4) SystemVerilog objects form the basis of an Object-Oriented data abstraction that provides true polymorphism. Class inheritance, abstract classes, and dynamic casting are powerful mechanisms that go way beyond the mere encapsulation mechanism provided by structs.

11.26 Memory management

Memory for objects, strings, and dynamic and associative arrays is allocated dynamically. When objects are created, SystemVerilog allocates more memory. When an object is no longer needed, SystemVerilog automatically reclaims the memory, making it available for re-use. The automatic memory management system is an integral part of SystemVerilog. Without automatic memory management, SystemVerilog's multi-threaded, re-entrant environment creates many opportunities for users to run into problems. A manual memory management system, such as the one provided by C's `malloc` and `free`, would not be sufficient.

For example, consider the following example:

```
myClass obj = new;
fork
   task1( obj );
   task2( obj );
join_none
```

In this example, the main process (the one that forks off the two tasks) does not know when the two processes might be done using the object `obj`. Similarly, neither `task1` nor `task2` knows when any of the other two processes will no longer be using the object `obj`. It is evident from this simple example that no single process has enough information to determine when it is safe to free the object. The only two options available to the user are (1) play it safe and never reclaim the object, or (2) add some form of reference count that can be used to determine when it might be safe to reclaim the object. Adopting the first option can cause the system to quickly run out of memory. The second option places a large burden on users, who, in addition to managing their testbench, must also manage the memory using less than ideal schemes. To avoid these shortcomings, SystemVerilog manages all dynamic memory automatically. Users do not need to worry about dangling references, premature deallocation, or memory leaks. The system shall automatically reclaim any object that is no longer being used. In the example above, all that users do is assign **`null`** to the handle `obj` when they no longer need it. Similarly, when an object goes out of scope the system implicitly assigns **`null`** to the object.

Section 12
Random Constraints

12.1 Introduction (informative)

Constraint-driven test generation allows users to automatically generate tests for functional verification. Random testing can be more effective than a traditional, directed testing approach. By specifying constraints, one can easily create tests that can find hard-to-reach corner cases. SystemVerilog allows users to specify constraints in a compact, declarative way. The constraints are then processed by a solver that generates random values that meet the constraints.

The random constraints are built on top of an object oriented data abstraction.that models the data to be randomized as objects that contain random variables and user-defined constraints. The constraints determine the legal values that can be assigned to the random variables. Objects are ideal for representing complex aggregate data types and protocols such as Ethernet packets.

Section 12.2 provides an overview of object-based randomization and constraint programming. The rest of this section provides detailed information on random variables, constraint blocks, and the mechanisms used to manipulate them.

12.2 Overview

This section introduces the basic concepts and uses for generating random stimulus within objects. SystemVerilog uses an object-oriented method for assigning random values to the member variables of an object, subject to user-defined constraints. For example:

```
class Bus;
   rand bit[15:0] addr;
   rand bit[31:0] data;

   constraint word_align {addr[1:0] == 2'b0;}
endclass
```

The `Bus` class models a simplified bus with two random variables: `addr` and `data`, representing the address and data values on a bus. The `word_align` constraint declares that the random values for `addr` must be such that `addr` is word-aligned (the low-order 2 bits are 0).

The `randomize()` method is called to generate new random values for a bus object:

```
Bus bus = new;

repeat (50) begin
   if ( bus.randomize() == 1 )
      $display ("addr = %16h data = %h\n", bus.addr, bus.data);
   else
      $display ("Randomization failed.\n");
end
```

Calling `randomize()` causes new values to be selected for all of the random variables in an object such that all of the constraints are true (satisfied). In the program test above, a `bus` object is created and then randomized 50 times. The result of each randomization is checked for success. If the randomization succeeds, the new random values for `addr` and `data` are printed; if the randomization fails, an error message is printed. In this example, only the `addr` value is constrained, while the `data` value is unconstrained. Unconstrained variables are assigned any value in their declared range.

Constraint programming is a powerful method that lets users build generic, reusable objects that can later be

extended or constrained to perform specific functions. The approach differs from both traditional procedural and object-oriented programming, as illustrated in this example that extends the `Bus` class:

```
typedef enum {low, mid, high} AddrType;

class MyBus extends Bus;
   rand AddrType atype;
   constraint addr_range
   {
      (atype == low ) => addr inside { [0 : 15] };
      (atype == mid ) => addr inside { [16 : 127]};
      (atype == high) => addr inside {[128 : 255]};
   }
endclass
```

The `MyBus` class inherits all of the random variables and constraints of the `Bus` class, and adds a random variable called `atype` that is used to control the address range using another constraint. The `addr_range` constraint uses implication to select one of three range constraints depending on the random value of `atype`. When a `MyBus` object is randomized, values for `addr`, `data`, and `atype` are computed such that all of the constraints are satisfied. Using inheritance to build layered constraint systems enables the development of general-purpose models that can be constrained to perform application-specific functions.

Objects can be further constrained using the `randomize()` **`with`** construct, which declares additional constraints in-line with the call to `randomize()`:

```
task exercise_bus (MyBus bus);
   int res;

   // EXAMPLE 1: restrict to small addresses
   res = bus.randomize() with {atype == small;};

   // EXAMPLE 2: restrict to address between 10 and 20
   res = bus.randomize() with {10 <= addr && addr <= 20;};

   // EXAMPLE 3: restrict data values to powers-of-two
   res = bus.randomize() with {data & (data - 1) == 0;};
endtask
```

This example illustrates several important properties of constraints:

— Constraints can be any SystemVerilog expression with variables and constants of integral type (**`bit`**, **`reg`**, **`logic`**, **`integer`**, **`enum`**, **`packed struct`**, etc.).

— The constraint solver must be able to handle a wide spectrum of equations, such as algebraic factoring, complex boolean expressions, and mixed integer and bit expressions. In the example above, the power-of-two constraint was expressed arithmetically. It could have also been defined with expressions using a shift operator. For example, `1 << n`, where `n` is a 5-bit random variable.

— If a solution exists, the constraint solver must find it. The solver can fail only when the problem is over-constrained and there is no combination of random values that satisfy the constraints.

— Constraints interact bidirectionally. In this example, the value chosen for `addr` depends on `atype` and how it is constrained, and the value chosen for `atype` depends on `addr` and how it is constrained. All expression operators are treated bidirectionally, including the implication operator (`=>`).

Sometimes it is desirable to disable constraints on random variables. For example, to deliberately generate an illegal address (non-word aligned):

```
task exercise_illegal(MyBus bus, int cycles);
   int res;
```



```
    // Disable word alignment constraint.
    bus.word_align.constraint_mode(0);

    repeat (cycles) begin

    // CASE 1: restrict to small addresses.
    res = bus.randomize() with {addr[0] || addr[1];};
       ...
    end

    // Re-enable word alignment constraint
    bus.word_align.constraint_mode(1);
endtask
```

The `constraint_mode()` method can be used to enable or disable any named constraint block in an object. In this example, the word-alignment constraint is disabled, and the object is then randomized with additional constraints forcing the low-order address bits to be non-zero (and thus unaligned).

The ability to enable or disable constraints allows users to design constraint hierarchies. In these hierarchies, the lowest level constraints can represent physical limits grouped by common properties into named constraint blocks, which can be independently enabled or disabled.

Similarly, the `rand_mode()` method can be used to enable or disable any random variable. When a random variable is disabled, it behaves in exactly the same way as other nonrandom variables.

Occasionally, it is desirable to perform operations immediately before or after randomization. That is accomplished via two built-in methods, `pre_randomize()` and `post_randomize()`, which are automatically called before and after randomization. These methods can be overloaded with the desired functionality:

```
class XYPair;
   rand integer x, y;
endclass

class MyYXPair extends XYPair
   function void pre_randomize();
      super.pre_randomize();
      $display("Before randomize x=%0d, y=%0d", x, y);
   endfunction

   function void post_randomize();
      super.post_randomize();
      $display("After randomize x=%0d, y=%0d", x, y);
   endfunction
endclass
```

By default, `pre_randomize()` and `post_randomize()` call their overloaded parent class methods. When `pre_randomize()` or `post_randomize()` are overloaded, care must be taken to invoke the parent class' methods, unless the class is a base class (has no parent class), otherwise the base class methods shall not be called.

The random stimulus generation capabilities and the object-oriented constraint-based verification methodology enable users to quickly develop tests that cover complex functionality and better assure design correctness.

12.3 Random variables

Class variables can be declared random using the **`rand`** and **`randc`** type-modifier keywords.

The syntax to declare a random variable in a class is:

```
class_property ::=                                          // from Annex A.1.8
        { property_qualifier } data_declaration

property_qualifier[11] ::=
        rand
      | randc
```

Syntax 12-1—Random variable declaration syntax (excerpt from Annex A)

— The solver can randomize singular variables of any integral type.

— Arrays can be declared **rand** or **randc**, in which case all of their member elements are treated as **rand** or **randc**.

— Dynamic and associative arrays can be declared **rand** or **randc**. All of the elements in the array are randomized, overwriting any previous data. If the array elements are object handles, all of the array elements must be non-null. Individual array elements can be constrained, in which case the index expression must be a literal constant.

— The size of a dynamic array declared as **rand** or **randc** can also be constrained. In that case, the array shall be resized according to the size constraint, and then all the array elements shall be randomized. The array size constraint is declared using the `size` method. For example,

```
rand bit [7:0] len;
rand integer data[];
constraint db { data.size == len; }
```

The variable `len` is declared to be 8 bits wide. The randomizer computes a random value for the `len` variable in the 8-bit range of 0 to 255, and then randomizes the first `len` elements of the data array.

If a dynamic array's size is not constrained then `randomize()` randomizes all the elements in the array.

— An object handle can be declared **rand** in which case all of that object's variables and constraints are solved concurrently with the variables and constraints of the object that contains the handle. Objects cannot be declared **randc**.

12.3.1 rand modifier

Variables declared with the **rand** keyword are standard random variables. Their values are uniformly distributed over their range. For example:

```
rand bit [7:0] y;
```

This is an 8-bit unsigned integer with a range of 0 to 255. If unconstrained, this variable shall be assigned any value in the range 0 to 255 with equal probability. In this example, the probability of the same value repeating on successive calls to randomize is 1/256.

12.3.2 randc modifier

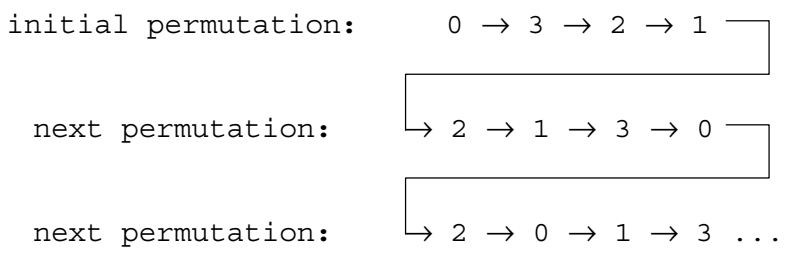
Variables declared with the **randc** keyword are random-cyclic variables that cycle through all the values in a random permutation of their declared range. Random-cyclic variables can only be of type **bit** or enumerated types, and can be limited to a maximum size.

To understand **randc**, consider a 2-bit random variable y:

```
randc bit [1:0] y;
```


The variable `y` can take on the values 0, 1, 2, and 3 (range 0 to 3). Randomize computes an initial random permutation of the range values of `y`, and then returns those values in order on successive calls. After it returns the last element of a permutation, it repeats the process by computing a new random permutation.

The basic idea is that **`randc`** randomly iterates over all the values in the range and that no value is repeated within an iteration. When the iteration finishes, a new iteration automatically starts.



The permutation sequence for any given **`randc`** variable is recomputed whenever the constraints change on that variable, or when none of the remaining values in the permutation can satisfy the constraints.

To reduce memory requirements, implementations can impose a limit on the maximum size of a **`randc`** variable, but it should be no less than 8 bits.

The semantics of random-cyclical variables require that they be solved before other random variables. A set of constraints that includes both **`rand`** and **`randc`** variables shall be solved such that the **`randc`** variables are solved first, and this can sometimes cause `randomize()` to fail.

12.4 Constraint blocks

The values of random variables are determined using constraint expressions that are declared using constraint blocks. Constraint blocks are class members, like tasks, functions, and variables. Constraint block names must be unique within a class.

The syntax to declare a constraint block is:

```
constraint_declaration ::=                                    // from Annex A.1.9
        [ static ] constraint constraint_identifier { { constraint_block } }
constraint_block ::=
        solve identifier_list before identifier_list ;
      | expression dist { dist_list } ;
      | constraint_expression
constraint_expression ::=
        expression ;
      | expression => constraint_set
      | if ( expression ) constraint_set [ else constraint_set ]
constraint_set ::=
        constraint_expression
      | { { constraint_expression } }
dist_list ::= dist_item { , dist_item }
dist_item ::=
        value_range := expression
      | value_range :/ expression
constraint_prototype ::= [ static ] constraint constraint_identifier
extern_constraint_declaration ::=
        [ static ] constraint class_identifier :: constraint_identifier { { constraint_block } }
identifier_list ::= identifier { , identifier }
```

Syntax 12-2—Constraint syntax (excerpt from Annex A)

constraint_identifier is the name of the constraint block. This name can be used to enable or disable a constraint using the `constraint_mode()` method (see Section 12.8).

constraint_block is a list of expression statements that restrict the range of a variable or define relations between variables. A *constraint_expression* is any SystemVerilog expression, or one of the constraint-specific operators: **`=>`**, **`inside`** and **`dist`** (see Sections 12.4.3 and 12.4.4).

The declarative nature of constraints imposes the following restrictions on constraint expressions:

— Calling tasks or functions is not allowed.

— Operators with side effects, such as **`++`** and **`--`** are not allowed.

— **`randc`** variables cannot be specified in ordering constraints (see **`solve`**...**`before`** in Section 12.4.8).

— **`dist`** expressions cannot appear in other expressions (unlike **`inside`**); they can only be top-level expressions.

12.4.1 External constraint blocks

Constraint block bodies can be declared outside a class declaration, just like external task and function bodies:

```
// class declaration
class XYPair;
   rand integer x, y;
   constraint c;
endclass

// external constraint body declaration
constraint XYPair::c { x < y; }
```


12.4.2 Inheritance

Constraints follow the same general rules for inheritance as class variables, tasks, and functions:

— A constraint in a derived class that uses the same name as a constraint in its parent classes overrides the base class constraints. For example:

```
class A;
   rand integer x;
   constraint c { x < 0; }
endclass

class B extends A;
   constraint c { x > 0; }
endclass
```

An instance of class `A` constrains `x` to be less than zero whereas an instance of class `B` constrains `x` to be greater than zero. The extended class `B` overrides the definition of constraint `c`. In this sense, constraints are treated the same as virtual functions, so casting an instance of `B` to an `A` does not change the constraint set.

— The `randomize()` task is virtual. Accordingly, it treats the class constraints in a virtual manner. When a named constraint is overloaded, the previous definition is overridden.

12.4.3 Set membership

Constraints support integer value sets and set membership operators.

The syntax to define a set expression is:

```
inside_expression ::= expression inside range_list_or_array          // from Annex A.8.3
range_list_or_array ::=
            variable_identifier
          | { value_range { , value_range } }
value_range ::=
            expression
          | [ expression : expression ]
```

Syntax 12-3—inside expression syntax (excerpt from Annex A)

expression is any integral SystemVerilog expression.

range_list_or_array is a comma-separated list of integral expressions and ranges. Value ranges are specified in ascending order with a low and high bound, enclosed by square braces **`[ ]`**, and separated by a colon (**`:`**), as in `[low_bound:high_bound]`. Ranges include all of the integer elements between the bounds. If the bound to the left of the colon is greater than the bound to the right the range is empty and contains no values.

The **`inside`** operator evaluates to true if the expression is contained in the set; otherwise it evaluates to false.

Absent any other constraints, all values (either single values or value ranges), have an equal probability of being chosen by the **`inside`** operator.

The negated form denotes that expression lies outside the set: `!(expression` **`inside`** `{ set })`.

For example:

```
rand integer x, y, z;
constraint c1 {x inside {3, 5, [9:15], [24:32], [y:2*y], z};}

rand integer a, b, c;
constraint c2 {a inside {b, c};}

integer fives[0:3] = { 5, 10, 15, 20 };
rand integer v;
constraint c3 { v inside fives; }
```

Set values and ranges can be any integral expression. Values can be repeated, so values and value ranges can overlap. It is important to note that the **inside** operator is bidirectional, thus, the second example above is equivalent to `a == b || a == c`.

12.4.4 Distribution

In addition to set membership, constraints support sets of weighted values called distributions. Distributions have two properties: they are a relational test for set membership, and they specify a statistical distribution function for the results.

The syntax to define a distribution expression is:

```
constraint_block ::=                                    // from Annex A.1.9
            ...
          | expression dist { dist_list } ;
dist_list ::= dist_item { , dist_item }
dist_item ::=
            value_range := expression
          | value_range :/ expression
```

Syntax 12-4—Constraint distribution syntax (excerpt from Annex A)

expression can be any integral SystemVerilog expression.

The distribution operator **dist** evaluates to true if the value of the expression is contained in the set; otherwise it evaluates to false.

Absent any other constraints, the probability that the expression matches any value in the list is proportional to its specified weight.

The distribution set is a comma-separated list of integral expressions and ranges. Optionally, each term in the list can have a weight, which is specified using the **:=** or **:/** operators. If no weight is specified, the default weight is 1. The weight can be any integral SystemVerilog expression.

The **:=** operator assigns the specified weight to the item, or if the item is a range, to every value in the range.

The **:/** operator assigns the specified weight to the item, or if the item is a range, to the range as a whole. If there are `n` values in the range, the weight of each value is `range_weight / n`.

For example:

```
x dist {100 := 1, 200 := 2, 300 := 5}
```

means `x` is equal to 100, 200, or 300 with weighted ratio of 1-2-5. If an additional constraint is added that specifies that `x` cannot be 200:

```
x != 200;
```



```
x dist {100 := 1, 200 := 2, 300 := 5}
```

then x is equal to 100 or 300 with weighted ratio of 1-5.

It is easier to think about mixing ratios, such as 1-2-5, than the actual probabilities because mixing ratios do not have to be normalized to 100%. Converting probabilities to mixing ratios is straightforward.

When weights are applied to ranges, they can be applied to each value in the range, or they can be applied to the range as a whole. For example,

```
x dist { [100:102] := 1, 200 := 2, 300 := 5}
```

means x is equal to 100, 101, 102, 200, or 300 with a weighted ratio of 1-1-1-2-5.

```
x dist { [100:102] :/ 1, 200 := 2, 300 := 5}
```

means x is equal to one of 100, 101, 102, 200, or 300 with a weighted ratio of 1/3-1/3-1/3-2-5.

In general, distributions guarantee two properties: set membership and monotonic weighting, which means that increasing a weight shall increase the likelihood of choosing those values.

Limitations:

— A **dist** operation shall not be applied to **randc** variables.

— A **dist** expression requires that expression contain at least one **rand** variable.

— A **dist** expression can only be a top-level constraint (not a predicated constraint).

12.4.5 Implication

Constraints provide two constructs for declaring conditional (predicated) relations: *implication* and **if**..**else**.

The implication operator (**=>**) can be used to declare an expression that implies a constraint.

The syntax to define an implication constraint is:

```
constraint_expression ::=                                   // from Annex A.1.9
          ...
        | expression => constraint_set
```

Syntax 12-5—Constraint implication syntax (excerpt from Annex A)

The *expression* can be any integral SystemVerilog expression.

The boolean equivalent of the implication operator `a => b` is `(!a || b)`. This states that if the expression is true, then random numbers generated are constrained by the constraint (or constraint block). Otherwise the random numbers generated are unconstrained.

The *constraint_set* represents any valid constraint or an unnamed constraint block. If the expression is true, all of the constraints in the constraint block must also be satisfied.

For example:

```
mode == small => len < 10;
mode == large => len > 100;
```

In this example, the value of `mode` implies that the value of `len` shall be constrained to less than 10 (`mode == small`), greater than 100 (`mode == large`), or unconstrained (`mode != small` and `mode != large`).

In the following example:

```
bit [3:0] a, b;
constraint c { (a == 0) => (b == 1); }
```

Both `a` and `b` are 4 bits, so there are 256 combinations of `a` and `b`. Constraint `c` says that `a == 0` implies that `b == 1`, thereby eliminating 15 combinations: {0,0}, {0,2}, … {0,15}. Therefore, the probability that `a == 0` is thus 1/(256-15) or 1/241.

12.4.6 if..else constraints

`if`...**`else`** style constraints are also supported.

The syntax to define an **`if`**...**`else`** constraint is:

```
constraint_expression ::=                                   // from Annex A.1.9
        ...
      | if ( expression ) constraint_set [ else constraint_set ]
```

Syntax 12-6—If...else constraint syntax (excerpt from Annex A)

expression can be any integral SystemVerilog expression.

constraint_set represents any valid constraint or an unnamed constraint block. If the expression is true, all of the constraints in the first constraint or constraint block must be satisfied, otherwise all of the constraints in the optional **`else`** constraint or constraint-block must be satisfied. Constraint blocks can be used to group multiple constraints.

If..else style constraint declarations are equivalent to implications:

```
if (mode == small)
   len < 10;
else if (mode == large)
   len > 100;
```

is equivalent to

```
mode == small => len < 10 ;
mode == large => len > 100 ;
```

In this example, the value of `mode` implies that the value of `len` is less than 10, greater than 100, or unconstrained.

Just like implication, **`if`**...**`else`** style constraints are bidirectional. In the declaration above, the value of `mode` constraints the value of `len`, and the value of `len` constrains the value of `mode`.

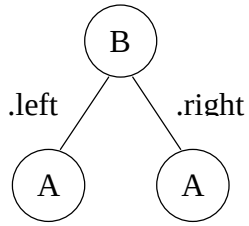
Because the **`else`** part of an **`if`**...**`else`** style constraint declaration is optional, there can be confusion when an **`else`** is omitted from a nested **`if`** sequence. This is resolved by always associating the **`else`** with the closest previous **`if`** that lacks an **`else`**. In the example below, the **`else`** goes with the inner **`if`**, as shown by indentation:

```
if (mode != large)
   if (mode == small)
      len < 10;
   else // the else applies to preceding if
      len > 100;
```


12.4.7 Global constraints

When an object member of a class is declared **rand**, all of its constraints and random variables are randomized simultaneously along with the other class variables and constraints. Constraint expressions involving random variables from other objects are called global constraints.

```
class A;            // leaf node
   rand bit [7:0] v;
endclass

class B extends A; // heap node
   rand A left;
   rand A right;

   constraint heapcond {left.v <= v; right.v <= v;}
endclass
```



This example uses global constraints to define the legal values of an ordered binary tree. Class `A` represents a leaf node with an 8-bit value `x`. Class `B` extends class `A` and represents a heap-node with value `v`, a left subtree, and a right subtree. Both subtrees are declared as **rand** in order to randomize them at the same time as other class variables. The constraint block named `heapcond` has two global constraints relating the left and right subtree values to the heap-node value. When an instance of class `B` is randomized, the solver simultaneously solves for `B` and its left and right children, which in turn can be leaf nodes or more heap-nodes.

The following rules determine which objects, variables, and constraints are to be randomized:

1) First, determine the set of objects that are to be randomized as a whole. Starting with the object that invoked the `randomize()` method, add all objects that are contained within it, are declared **rand**, and are active (see `rand_mode` in Section 12.7). The definition is recursive and includes all of the active random objects that can be reached from the starting object. The objects selected in this step are referred to as the active random objects.

2) Next, select all of the active constraints from the set of active random objects. These are the constraints that are applied to the problem.

3) Finally, select all of the active random variables from the set of active random objects. These are the variables that are to be randomized. All other variable references are treated as state variables, whose current value is used as a constant.

12.4.8 Variable ordering

The solver must assure that the random values are selected to give a uniform value distribution over legal value combinations (that is, all combinations of legal values have the same probability of being the solution). This important property guarantees that all legal value combinations are equally probable, which allows randomization to better explore the whole design space.

Sometimes, however, it is desirable to force certain combinations to occur more frequently. Consider the case where a 1-bit control variable `s` constrains a 32-bit data value `d`:

```
class B;
   rand bit s;
   rand bit [31:0] d;

   constraint c { s => d == 0; }
endclass
```

The constraint `c` says "`s` implies `d` equals zero". Although this reads as if `s` determines `d`, in fact `s` and `d` are determined together. There are 2^{32} valid combinations of `{s,d}`, but `s` is only true for `{1,0}`. Thus, the prob-

ability that s is true is $1/2^{32}$, which is practically zero.

The constraints provide a mechanism for ordering variables so that s can be chosen independently of d. This mechanism defines a partial ordering on the evaluation of variables, and is specified using the solve keyword.

```
class B;
   rand bit s;
   rand bit [31:0] d;
   constraint c { s => d == 0; }
   constraint order { solve s before d; }
endclass
```

In this case, the order constraint instructs the solver to solve for s before solving for d. The effect is that s is now chosen true with 50% probability, and then d is chosen subject to the value of s. Accordingly, d == 0 shall occur 50% of the time, and d != 0 shall occur for the other 50%.

Variable ordering can be used to force selected corner cases to occur more frequently than they would otherwise.

The syntax to define variable order in a constraint block is:

```
constraint_block ::=                                    // from Annex A.1.9
        solve identifier_list before identifier_list ;
```

Syntax 12-7—Solve...before constraint ordering syntax (excerpt from Annex A)

solve and **before** each take a comma-separated list of integral variables or array elements.

The following restrictions apply to variable ordering:

— Only random variables are allowed, that is, they must be **rand**.

— **randc** variables are not allowed. **randc** variables are always solved before any other.

— The variables must be integral values.

— A constraint block can contain both regular value constraints and ordering constraints.

— There must be no circular dependencies in the ordering, such as “solve a before b” combined with “solve b before a”.

— Variables that are not explicitly ordered shall be solved with the last set of ordered variables. These values are deferred until as late as possible to assure a good distribution of values.

— Variables can be solved in an order that is not consistent with the ordering constraints, provided that the outcome is the same. An example situation where this might occur is:

```
x == 0;
x < y;
solve y before x;
```

In this case, since x has only one possible assignment (0), x can be solved for before y. The constraint solver can use this flexibility to speed up the solving process.

12.4.9 Static constraint blocks

A constraint block can be defined as static by including the **static** keyword in its definition.

The syntax to declare a static constraint block is:


```
static constraint constraint_identifier { contraint_expressions }
```

If a constraint block is declared as **static**, then calls to `constraint_mode()` shall affect all instances of the specified constraint in all objects. Thus, if a static constraint is set to OFF, it is off for all instances of that particular class.

12.5 Randomization methods

12.5.1 randomize()

Variables in an object are randomized using the `randomize()` class method. Every class has a built-in `randomize()` virtual method, declared as:

```
virtual function int randomize();
```

The `randomize()` method is a virtual function that generates random values for all the active random variables in the object, subject to the active constraints.

The `randomize()` method returns 1 if it successfully sets all the random variables and objects to valid values, otherwise it returns 0.

Example:

```
class SimpleSum;
   rand bit [7:0] x, y, z;
   constraint c {z == x + y;}
endclass
```

This class definition declares three random variables, `x`, `y`, and `z`. Calling the `randomize()` method shall randomize an instance of class `SimpleSum`:

```
SimpleSum p = new;
int success = p.randomize();
if (success == 1 ) ...
```

Checking the return status can be necessary because the actual value of state variables or addition of constraints in derived classes can render seemingly simple constraints unsatisfiable.

12.5.2 pre_randomize() and post_randomize()

Every class contains built-in `pre_randomize()` and `post_randomize()` functions, that are automatically called by `randomize()` before and after computing new random values.

The built-in definition for `pre_randomize()` is:

```
function void pre_randomize;
   if (super) super.pre_randomize();   // test super to see if the
                                       // object handle exists
      // Optional programming before randomization goes here
endfunction
```

The built-in definition for `post_randomize()` is:

```
function void post_randomize;
   if (super) super.post_randomize();  // test super to see if the
                                       // object handle exists
      // Optional programming after randomization goes here
endfunction
```

When `obj.randomize()` is invoked, it first invokes `pre_randomize()` on `obj` and also all of its random object members that are enabled. `pre_randomize()` then calls `super.pre_randomize()`. After the new random values are computed and assigned, `randomize()` invokes `post_randomize()` on `obj` and also all of its random object members that are enabled. `post_randomize()` then calls **`super`**`.post_randomize()`.

Users can overload the `pre_randomize()` in any class to perform initialization and set pre-conditions before the object is randomized.

Users can overload the `post_randomize()` in any class to perform cleanup, print diagnostics, and check post-conditions after the object is randomized.

If these methods are overloaded, they must call their associated parent class methods, otherwise their pre- and post-randomization processing steps shall be skipped.

The `pre_randomize()` and `post_randomize()` methods are not virtual. However, because they are automatically called by the `randomize()` method, which is virtual, they appear to behave as virtual methods.

12.5.3 Randomization methods notes

— Random variables declared as static are shared by all instances of the class in which they are declared. Each time the `randomize()` method is called, the variable is changed in every class instance.

— If `randomize()` fails, the constraints are infeasible and the random variables retain their previous values.

— If `randomize()` fails, `post_randomize()` is not called.

— The `randomize()` method shall not be overloaded.

— The `randomize()` method implements object random stability. An object can be seeded by the `$srandom()` system call (see Section 12.10.3), specifying the object in the second argument.

— The built-in methods `pre_randomize()` and `post_randomize()` are functions and cannot block.

12.6 In-line constraints — randomize() with

By using the `randomize()`...**`with`** construct, users can declare in-line constraints at the point where the `randomize()` method is called. These additional constraints are applied along with the object constraints.

The syntax for `randomize()`...**`with`** is:

```
blocking_assignment ::=                                   // from Annex A.6.2
        ...
        | class_identifier . randomize [ ( ) ] with constraint_block ;
```

Syntax 12-8—In-line constraint syntax (excerpt from Annex A)

class_identifier is the name of an instantiated object.

The unnamed *constraint_block* contains additional in-line constraints to be applied along with the object constraints declared in the class.

For example:

```
class SimpleSum
   rand bit [7:0] x, y, z;
   constraint c {z == x + y;}
endclass
```



```
task InlineConstraintDemo(SimpleSum p);
   int success;
   success = p.randomize() with {x < y;};
endtask
```

This is the same example used before, however, `randomize()...with` is used to introduce an additional constraint that `x < y`.

The `randomize()...with` construct can be used anywhere an expression can appear. The constraint block following **`with`** can define all of the same constraint types and forms as would otherwise be declared in a class.

The `randomize()...with` constraint block can also reference local variables and task and function arguments, eliminating the need for mirroring a local state as member variables in the object class. The scope for variable names in a constraint block, from inner to outer, is: `randomize()...with` object class, automatic and local variables, task and function arguments, class variables, variables in the enclosing scope. The `randomize()...with` class is brought into scope at the innermost nesting level.

In the example below, the `randomize()...with` class is `Foo`.

```
class Foo;
   rand integer x;
endclass

class Bar;
   integer x;
   integer y;

   task doit(Foo f, integer x, integer z);
      int result;
      result = f.randomize() with {x < y + z;};
   endtask
endclass
```

In the `f.randomize() with` constraint block, `x` is a member of class `Foo`, and hides the `x` in class `Bar`. It also hides the `x` argument in the `doit()` task. `y` is a member of `Bar`. `z` is a local argument.

12.7 Disabling random variables with rand_mode()

The `rand_mode()` method can be used to control whether a random variable is active or inactive. When a random variable is inactive, it is treated the same as if it had not been declared **`rand`** or **`randc`**. Inactive variables are not randomized by the `randomize()` method, and their values are treated as state variables by the solver. All random variables are initially active.

The syntax for the `rand_mode()` method is:

```
task object[.random_variable]::rand_mode( bit on_off );
```

or

```
function int object.random_variable::rand_mode();
```

object is any expression that yields the object handle in which the random variable is defined.

random_variable is the name of the random variable to which the operation is applied. If it is not specified (only allowed when called as a task), the action is applied to all random variables within the specified object.

When called as a task, the argument to the `rand_mode` method determines the operation to be performed:

Table 12-1: rand_mode argument

Value	Meaning	Description
0	OFF	Sets the specified variables to inactive so that they are not randomized on subsequent calls to the `randomize()` method.
1	ON	Sets the specified variables to active so that they are randomized on subsequent calls to the `randomize()` method.

For array variables, `random_variable` can specify individual elements using the corresponding index. Omitting the index results in all the elements of the array being affected by the call.

If the random variable is an object handle, only the mode of the variable is changed, not the mode of random variables within that object (see global constraints in Section 12.4.7).

A compiler error shall be issued if the specified variable does not exist within the class hierarchy or it exists but is not declared as **`rand`** or **`randc`**.

When called as a function, `rand_mode()` returns the current active state of the specified random variable. It returns 1 if the variable is active (`ON`), and 0 if the variable is inactive (`OFF`).

The function form of `rand_mode()` only accepts singular variables, thus, if the specified variable is an unpacked array, a single element must be selected via its index.

Example:

```
class Packet;
   rand integer source_value, dest_value;
   ... other declarations
endclass

int ret;
Packet packet_a = new;
// Turn off all variables in object
packet_a.rand_mode(0);

// ... other code
// Enable source_value
packet_a.source_value.rand_mode(1);

ret = packet_a.dest_value.rand_mode();
```

This example first disables all random variables in the object `packet_a`, and then enables only the `source_value` variable. Finally, it sets the `ret` variable to the active status of variable `dest_value`.

The `rand_mode()` method is built-in and cannot be overridden.

12.8 Controlling constraints with constraint_mode()

The `constraint_mode()` method can be used to control whether a constraint is active or inactive. When a constraint is inactive, it is not considered by the `randomize()` method. All constraints are initially active.

The syntax for the `constraint_mode()` method is:


```
task object[.constraint_identifier]::constraint_mode( bit on_off );
```

or

```
function int object.constraint_identifier::constraint_mode();
```

object is any expression that yields the object handle in which the constraint is defined.

constraint_identifier is the name of the constraint block to which the operation is applied. The constraint name can be the name of any constraint block in the class hierarchy. If no constraint name is specified (only allowed when called as a task), the operation is applied to all constraints within the specified object.

When called as a task, the argument to the `constraint_mode` task method determines the operation to be performed:

Table 12-2: constraint_mode argument

Value	Meaning	Description
0	OFF	Sets the specified constraint block to active so that it is considered by subsequent calls to the `randomize()` method.
1	ON	Sets the specified constraint block to inactive so that it is not enforced on subsequent calls to the `randomize()` method.

A compiler error shall be issued if the specified constraint block does not exist within the class hierarchy.

When called as a function, `constraint_mode()` returns the current active state of the specified constraint block. It returns 1 if the constraint is active (ON), and 0 if the constraint is inactive (OFF).

Example:

```
class Packet;
   rand integer source_value;
   constraint filter1 { source_value > 2 * m; }
endclass

function integer toggle_rand( Packet p );
   if ( p.filter1.constraint_mode() )
      p.filter1.constraint_mode(0);
   else
      p.filter1.constraint_mode(1);

   toggle_rand = p.randomize();
endfunction
```

In this example, the `toggle_rand` function first checks the current active state of the constraint filter1 in the specified `Packet` object `p`. If the constraint is active, then the function shall deactivate it; if it is inactive, the function shall activate it. Finally, the function calls the randomize method to generate a new random value for variable `source_value`.

The `constraint_mode()` method is built-in and cannot be overridden.

12.9 Dynamic constraint modification

There are several ways to dynamically modify randomization constraints:

— Implication and **`if`**...**`else`** style constraints allow declaration of predicated constraints.

— Constraint blocks can be made active or inactive using the `constraint_mode()` built-in method. Initially, all constraint blocks are active. Inactive constraints are ignored by the `randomize()` function.

— Random variables can be made active or inactive using the `rand_mode()` built-in method. Initially, all **`rand`** and **`randc`** variables are active. Inactive variables are ignored by the `randomize()` function.

— The weights in a **`dist`** constraint can be changed, affecting the probability that particular values in the set are chosen.

12.10 Random number system functions

12.10.1 $urandom

The system function `$urandom` provides a mechanism for generating pseudorandom numbers. The function returns a new 32-bit random number each time it is called. The number shall be unsigned.

The syntax for `$urandom` is:

```
function int unsigned $urandom [ (int seed ) ] ;
```

The `seed` is an optional argument that determines the sequence of random numbers generated. The seed can be any integral expression. The random number generator shall generate the same sequence of random numbers every time the same seed is used.

The random number generator is deterministic. Each time the program executes, it cycles through the same random sequence. This sequence can be made nondeterministic by seeding the `$urandom` function with an extrinsic random variable, such as the time of day.

For example:

```
bit [64:1] addr;

$urandom( 254 );              // Initialize the generator
addr = {$urandom, $urandom }; // 64-bit random number
number = $urandom & 15;       // 4-bit random number
```

The `$urandom` function is similar to the `$random` system function, with two exceptions. `$urandom` returns unsigned numbers and is automatically thread stable (see Section 12.11.2).

12.10.2 $urandom_range()

The `$urandom_range()` function returns an unsigned integer within a specified range.

The syntax for `$urandom_range()` is:

```
function int unsigned $urandom_range( int unsigned maxval,
                                      int unsigned minval = 0 );
```

The function shall return an unsigned integer in the range `maxval` .. `minval`.

Example: `val = $urandom_range(7,0);`

If `minval` is omitted, the function shall return a value in the range `maxval` .. 0.

Example: `val = $urandom_range(7);`

If `maxval` is less than `minval`, the arguments are automatically reversed so that the first argument is larger than the second argument.

Example: `val = $urandom_range(0,7);`

All of the three previous examples produce a value in the range of 0 to 7, inclusive.

`$urandom_range()` is automatically thread stable (see Section 12.11.2).

12.10.3 $srandom()

The system function `$srandom()` allows manually seeding the Random Number Generator (RNG) of objects or threads.

The syntax for the `$srandom()` system task is:

```
task $srandom( int seed, [class_identifier obj] );
```

The `$srandom()` system task initializes the local random number generator using the value of the given seed. The optional object argument is used to seed an object instead of the current process (thread). The top level randomizer of each program is initialized with `$srandom(1)` prior to any randomization calls.

12.11 Random stability

The Random Number Generator (RNG) is localized to threads and objects. Because the sequence of random values returned by a thread or object is independent of the RNG in other threads or objects, this property is called *random stability.* Random stability applies to:

— The system randomization calls, `$urandom`, `$urandom_range()`, and `$srandom()`.

— The object randomization method, `randomize()`.

Testbenches with this feature exhibit more stable RNG behavior in the face of small changes to the user code. Additionally, it enables more precise control over the generation of random values by manually seeding threads and objects.

12.11.1 Random stability properties

Random stability encompasses the following properties:

— Thread stability

  Each thread has an independent RNG for all randomization system calls invoked from that thread. When a new thread is created, its RNG is seeded with the next random value from its parent thread. This property is called *hierarchical seeding.*

  Program and thread stability is guaranteed as long as thread creation and random number generation is done in the same order as before. When adding new threads to an existing test, they can be added at the end of a code block in order to maintain random number stability of previously created work.

— Object stability

  Each class instance (object) has an independent RNG for all randomization methods in the class. When an object is created using `new`, its RNG is seeded with the next random value from the thread that creates the object.

  Object stability is guaranteed as long as object and thread creation, as well as random number generation, are done in the same order as before. In order to maintain random number stability, new objects, threads and random numbers can be created after existing objects are created.

— Manual seeding

  All RNG's can be manually seeded. Combined with hierarchical seeding, this facility allows users to define the operation of a subsystem (hierarchy subtree) completely with a single seed at the root thread of the system.

12.11.2 Thread stability

Random values returned from the `$urandom` system call are independent of thread execution order. For example:

```
integer x, y, z;
fork        //set a seed at the start of a thread
   begin $srandom(100); x = $urandom; end
            //set a seed during a thread
   begin y = $urandom; $srandom(200); end
            // draw 2 values from the thread RNG
   begin z = $urandom + $urandom ; end
join
```

The above program fragment illustrates several properties:

— Thread locality. The values returned for `x`, `y` and `z` are independent of the order of thread execution. This is an important property because it allows development of subsystems that are independent, controllable, and predictable.

— Hierarchical seeding. When a thread is created, its random state is initialized using the next random value from the parent thread as a seed. The three forked threads are all seeded from the parent thread.

Each thread is seeded with a unique value, determined solely by its parent. The root of a thread execution subtree determines the random seeding of its children. This allows entire subtrees to be moved, and preserves their behavior by manually seeding their root thread.

12.11.3 Object stability

The `randomize()` method built into every class exhibits object stability. This is the property that calls to `randomize()` in one instance are independent of calls to `randomize()` in other instances, and independent of calls to other randomize functions.

For example:

```
class Foo;
   rand integer x;
endclass

class Bar;
   rand integer y;
endclass

initial begin
   Foo foo = new();
   Bar bar = new();
   integer z;
   void'foo.randomize();
   // z = $random;
   void'bar.randomize();
end
```

— The values returned for `foo.x` and `bar.y` are independent of each other.

— The calls to `randomize()` are independent of the `$random` system call. If one uncomments the line `z = $random` above, there is no change in the values assigned to `foo.x` and `bar.y`.

— Each instance has a unique source of random values that can be seeded independently. That random seed is taken from the parent thread when the instance is created.

— Objects can be seeded at any time using the `$srandom()` system task with an optional object argument.

```
class Foo;
   function new (integer seed);
      //set a new seed for this instance
      $srandom(seed, this);
   endfunction
endclass
```

Once an object is created there is no guarantee that the creating thread can change the object's random state before another thread accesses the object. Therefore, it is best that objects self-seed within their new method rather than externally.

An object's seed can be set from any thread. However, a thread's seed can only be set from within the thread itself.

12.12 Manually seeding randomize

Each object maintains its own internal random number generator, which is used exclusively by its `randomize()` method. This allows objects to be randomized independent of each other and of calls to other system randomization functions. When an object is created, its random number generator (RNG) is seeded using the next value from the RNG of the thread that creates the object. This process is called *hierarchical object seeding*.

Sometimes it is desirable to manually seed an object's RNG using the `$srandom()` system call. This can be done either in a class method, or external to the class definition:

An example of seeding the RNG internally, as a class method is:

```
class Packet;
   rand bit[15:0] header;
   ...
   function new (int seed);
      $srandom(seed, this);
      ...
   endfunction
endclass
```

An example of seeding the RNG externally is:

```
Packet p = new(200); // Create p with seed 200.
$srandom(300, p);    // Re-seed p with seed 300.
```

Calling `$srandom()` in an object's `new()` function, assures the object's RNG is set with the new seed before any class member values are randomized.

Section 13
Inter-Process Synchronization and Communication

13.1 Introduction (informative)

High-level and easy-to-use synchronization and communication mechanism are essential to control the kinds of interactions that occur between dynamic processes used to model a complex system or a highly reactive testbench. Verilog provides basic synchronization mechanisms (i.e., **->** and **@**), but they are all limited to static objects and are adequate for synchronization at the hardware level, but fall short of the needs of a highly dynamic, reactive testbench. At the system level, an essential limitation of Verilog is its inability to create dynamic events and communication channels, which match the capability to create dynamic processes.

SystemVerilog adds a powerful and easy-to-use set of synchronization and communication mechanisms, all of which can be created and reclaimed dynamically. SystemVerilog adds a ***semaphore*** built-in class, which can be used for synchronization and mutual exclusion to shared resources, and a ***mailbox*** built-in class that can be used as a communication channel between processes. SystemVerilog also enhances Verilog's named **event** data type to satisfy many of the system-level synchronization requirements.

Semaphores and mailboxes are built-in types, nonetheless, they are classes, and can be used as base classes for deriving additional higher level classes.

13.2 Semaphores

Conceptually, a *semaphore* is a bucket. When a semaphore is allocated, a bucket that contains a fixed number of keys is created. Processes using semaphores must first procure a key from the bucket before they can continue to execute. If a specific process requires a key, only a fixed number of occurrences of that process can be in progress simultaneously. All others must wait until a sufficient number of keys is returned to the bucket. Semaphores are typically used for mutual exclusion, access control to shared resources, and for basic synchronization.

An example of creating a semaphore is:

```
semaphore smTx;
```

Semaphore is a built-in class that provides the following methods:

— Create a semaphore with a specified number of keys: `new()`

— Obtain one or more keys from the bucket: `get()`

— Return one or more keys into the bucket: `put()`

— Try to obtain one or more keys without blocking: `try_get()`

13.2.1 new()

Semaphores are created with the `new()` method.

The prototype for semaphore `new()` is:

```
function new(int keyCount = 0 );
```

The *KeyCount* specifies the number of keys initially allocated to the semaphore bucket. The number of keys in the bucket can increase beyond *KeyCount* when more keys are put into the semaphore than are removed. The default value for *KeyCount* is 0.

The `new()` function returns the semaphore handle, or **null** if the semaphore cannot be created.

13.2.2 put()

The semaphore `put()` method is used to return keys to a semaphore.

The prototype for `put()` is:

```
task put(int keyCount = 1);
```

keyCount specifies the number of keys being returned to the semaphore. The default is 1.

When the `semaphore.put()` task is called, the specified number of keys are returned to the semaphore. If a process has been suspended waiting for a key, that process shall execute if enough keys have been returned.

13.2.3 get()

The semaphore `get()` method is used to procure a specified number of keys from a semaphore.

The prototype for `get()` is:

```
task get(int keyCount = 1);
```

keyCount specifies the required number of keys to obtain from the semaphore. The default is 1.

If the specified number of keys are available, the method returns and execution continues. If the specified number of key are not available, the process blocks until the keys become available.

The semaphore waiting queue is First-In First-Out (FIFO). This does not guarantee the order in which processes arrive at the queue, only that their arrival order shall be preserved by the semaphore.

13.2.4 try_get()

The semaphore `try_get()` method is used to procure a specified number of keys from a semaphore, but without blocking.

The prototype for `try_get()` is:

```
function int try_get(int keyCount = 1);
```

keyCount specifies the required number of keys to obtain from the semaphore. The default is 1.

If the specified number of keys are available, the method returns 1 and execution continues. If the specified number of key are not available, the method returns 0.

13.3 Mailboxes

A *mailbox* is a communication mechanism that allows messages to be exchanged between processes. Data can be sent to a mailbox by one process and retrieved by another.

Conceptually, mailboxes behave like real mailboxes. When a letter is delivered and put into the mailbox, one can retrieve the letter (and any data stored within). However, if the letter has not been delivered when one checks the mailbox, one must choose whether to wait for the letter or retrieve the letter on subsequent trips to the mailbox. Similarly, SystemVerilog's mailboxes provide processes to transfer and retrieve data in a controlled manner. Mailboxes are created as having either a bounded or unbounded queue size. A bounded mailbox becomes full when it contains the bounded number of messages. A process that attempts to place a message into a full mailbox shall be suspended until enough room becomes available in the mailbox queue. Unbounded mailboxes never suspend a thread in a send operation.

An example of creating a mailbox is:

```
mailbox mbxRcv;
```

Mailbox is a built-in class that provides the following methods:

— Create a mailbox: `new()`

— Place a message in a mailbox: `put()`

— Try to place a message in a mailbox without blocking: `try_put()`

— Retrieve a message from a mailbox: `get()` or `peek()`

— Try to retrieve a message from a mailbox without blocking: `try_get()` or `try_peek()`

— Retrieve the number of messages in the mailbox: `num()`

13.3.1 new()

Mailboxes are created with the `new()` method.

The prototype for mailbox `new()` is:

```
function new(int bound = 0);
```

The `new()` function returns the mailbox handle, or **`null`** if the mailboxes cannot be created. If the `bound` argument is zero then the mailbox is unbounded (the default) and a `put()` operation shall never block. If `bound` is non-zero, it represents the size of the mailbox queue.

The bound must be positive. Negative bounds are illegal and can result in indeterminate behavior, but implementations can issue a warning.

13.3.2 num()

The number of messages in a mailbox can be obtained via the `num()` method.

The prototype for `num()` is:

```
function int num();
```

The `num()` method returns the number of messages currently in the mailbox.

The returned value should be used with care, since it is valid only until the next `get()` or `put()` is executed on the mailbox. These mailbox operations can be from different processes than the one executing the `num()` method. Therefore, the validity of the returned value shall depend on the time that the other methods start and finish.

13.3.3 put()

The `put()` method places a message in a mailbox.

The prototype for `put()` is:

```
task put( singular message);
```

The `message` is any singular expression, including object handles.

The `put()` method stores a message in the mailbox in strict FIFO order. If the mailbox was created with a bounded queue the process shall be suspended until there is enough room in the queue.

13.3.4 try_put()

The `try_put()` method attempts to place a message in a mailbox.

The prototype for `try_put()` is:

```
function int try_put( singular message);
```

The `message` is any singular expression, including object handles.

The `try_put()` method stores a message in the mailbox in strict FIFO order. This method is meaningful only for bounded mailboxes. If the mailbox is not full then the specified message is placed in the mailbox and the function returns 1. If the mailbox is full, the method returns 0.

13.3.5 get()

The `get()` method retrieves a message from a mailbox.

The prototype for `get()` is:

```
task get( ref singular message );
```

The `message` can be any singular expression, and it must be a valid left-hand side expression.

The `get()` method retrieves one message from the mailbox, that is, removes one message from the mailbox queue. If the mailbox is empty then the current process blocks until a message is placed in the mailbox. If there is a type mismatch between the *message* variable and the message in the mailbox, a runtime error is generated.

Non-parameterized mailboxes are type-less, that is, a single mailbox can send and receive different types of data. Thus, in addition to the data being sent (i.e., the message queue), a mailbox implementation must maintain the message data type placed by `put()`. This is required in order to enable the runtime type checking.

The mailbox waiting queue is FIFO. This does not guarantee the order in which processes arrive at the queue, only that their arrival order shall be preserved by the mailbox.

13.3.6 try_get()

The `try_get()` method attempts to retrieves a message from a mailbox without blocking.

The prototype for `try_get()` is:

```
function int try_get( ref singular message );
```

The *message* can be any singular expression, and it must be a valid left-hand side expression.

The `try_get()` method tries to retrieve one message from the mailbox. If the mailbox is empty, then the method returns 0. If there is a type mismatch between the *message* variable and the message in the mailbox, the method returns –1. If a message is available and the message type matches the type of the *message* variable, the message is retrieved and the method returns 1.

13.3.7 peek()

The `peek()` method copies a message from a mailbox without removing the message from the queue.

The prototype for `peek()` is:

```
task peek( ref singular message );
```

The *message* can be any singular expression, and it must be a valid left-hand side expression.

The `peek()` method copies one message from the mailbox without removing the message from the mailbox queue. If the mailbox is empty then the current process blocks until a message is placed in the mailbox. If there is a type mismatch between the *message* variable and the message in the mailbox, a runtime error is generated.

Note that calling `peek()` can cause one message to unblock more than one process. As long as a message remains in the mailbox queue, any process blocked in either a `peek()` or `get()` operation shall become unblocked.

13.3.8 try_peek()

The `try_peek()` method attempts to copy a message from a mailbox without blocking.

The prototype for `try_peek()` is:

```
function int try_peek( ref singular message );
```

The `message` can be any singular expression, and it must be a valid left-hand side expression.

The `try_peek()` method tries to copy one message from the mailbox without removing the message from the mailbox queue. If the mailbox is empty, then the method returns 0. If there is a type mismatch between the *message* variable and the message in the mailbox, the method returns –1. If a message is available and the message type matches, the type of the *message* variable, the message is copied and the method returns 1.

13.4 Parameterized mailboxes

The default mailbox is type-less, that is, a single mailbox can send and receive any type of data. This is a very powerful mechanism that, unfortunately, can also result in run-time errors due to type mismatches between a message and the type of the variable used to retrieve the message. Frequently, a mailbox is used to transfer a particular message type, and, in that case, it is useful to detect type mismatches at compile time.

Parameterized mailboxes use the same parameter mechanism as parameterized classes (see Section 11.23), modules, and interfaces:

```
mailbox #(type = dynamic_type)
```

Where `dynamic_type` represents a special type that enables run-time type-checking (the default).

A parameterized mailbox of a specific type is declared by specifying the type:

```
typedef mailbox #(string) s_mbox;

s_mbox sm = new;
string s;

sm.put( "hello" );
...
sm.get( s );   // s <- "hello"
```

Parameterized mailboxes provide all the same standard methods as *dynamic* mailboxes: `num()`, `new()`, `get()`, `peek()`, `put()`, `try_get()`, `try_peek()`, `try_put()`.

The only difference between a generic (dynamic) mailbox and a parameterized mailbox is that for a parameterized mailbox, the compiler ensures that all `put`, `peek`, `try_peek` and `get` methods are compatible with the mailbox type, so that all type mismatches are caught by the compiler and not at run-time.

13.5 Event

In Verilog, named events are static objects that can be triggered via the `->` operator, and processes can wait for an event to be triggered via the `@` operator. SystemVerilog events support the same basic operations, but enhance Verilog events in several ways. The most salient enhancement is that the triggered state of Verilog named events has no duration, whereas in SystemVerilog this state persists throughout the time-step in which the event triggered. Also, SystemVerilog events act as handles to synchronization queues, thus, they can be passed as arguments to tasks, and they can be assigned to one another or compared.

Existing Verilog event operations (`@` and `->`) are backward compatible and continue to work the same way when used in the static Verilog context. The additional functionality described below works with all events in either the static or dynamic context.

A SystemVerilog event provides a handle to an underlying synchronization object. When a process waits for an event to be triggered, the process is put on a queue maintained within the synchronization object. Processes can wait for a SystemVerilog event to be triggered either via the `@` operator, or by using the `wait()` construct to examine their triggered state. Events are triggered using the `->` or the `->>` operator.

```
event_trigger ::=                                                    // from Annex A.6.5
        -> hierarchical_event_identifier ;
      |->> [ delay_or_event_control ] hierarchical_event_identifier ;
```

Syntax 13-1—Event trigger syntax (excerpt from Annex A)

The syntax to declare named events is discussed in Section 3.8.

13.5.1 Triggering an event

Named events are triggered via the `->` operator.

Triggering an event unblocks all processes currently waiting on that event. When triggered, named events behave like a one-shot, that is, the trigger state itself is not observable, only its effect. This is similar to the way in which an edge can trigger a flip-flop but the state of the edge can not be ascertained, i.e., `if (posedge clock)` is illegal.

13.5.2 Nonblocking event trigger

Nonblocking events are triggered using the `->>` operator.

The effect of the `->>` operator is that the statement executes without blocking and it creates a nonblocking assign update event in the time in which the delay control expires, or the event-control occurs. The effect of this update event shall be to trigger the referenced event in the nonblocking assignment region of the simulation cycle.

13.5.3 Waiting for an event

The basic mechanism to wait for an event to be triggered is via the event control operator, `@`.

```
@ event_identifier;
```

The `@` operator blocks the calling process until the given event is triggered.

For a trigger to unblock a process waiting on an event, the waiting process must execute the `@` statement before the triggering process executes the trigger operator, `->`. If the trigger executes first, then the waiting process remains blocked.

13.5.4 Persistent trigger: triggered property

SystemVerilog can distinguish the event trigger itself, which is instantaneous, from the event's triggered state, which persists throughout the time-step (i.e., until simulation time advances). The `triggered` event property allows users to examine this state.

The `triggered` property is invoked using a method-like syntax:

event_identifier.triggered

The `triggered` event property evaluates to true if the given event has been triggered in the current time-step and false otherwise. If `event_identifier` is **`null`**, then the triggered event property evaluates to false.

The `triggered` event property is most useful when used in the context of a **`wait`** construct:

```
wait ( event_identifier.triggered )
```

Using this mechanism, an event trigger shall unblock the waiting process whether the **`wait`** executes before or at the same simulation time as the trigger operation. The `triggered` event property, thus, helps eliminate a common race condition that occurs when both the trigger and the **`wait`** happen at the same time. A process that blocks waiting for an event might or might not unblock, depending on the execution order of the waiting and triggering processes. However, a process that waits on the triggered state always unblocks, regardless of the order of execution of the wait and trigger operations.

Example:

```
event done, blast;          // declare two new events
event done_too = done;      // declare done_too as alias to done

task trigger( event ev );
   -> ev;
endtask

...

fork
   @ done_too;              // wait for done through done_too
   #1 trigger( done );      // trigger done through task trigger
join

fork
   -> blast;
   wait ( blast.triggered );
join
```

The first fork in the example shows how two event identifiers, `done` and `done_too`, refer to the same synchronization object, and also how an event can be passed to a generic task that triggers the event. In the example, one process waits for the event via `done_too`, while the actual triggering is done via the `trigger` task that is passed `done` as an argument.

In the second fork, one process can trigger the event `blast` before the other process (if the processes in the **`fork`**…**`join`** execute in source order) has a chance to execute, and wait for the event. Nonetheless, the second process unblocks and the fork terminates. This is because the process waits for the event's triggered state, which remains in its triggered state for the duration of the time-step.

13.6 Event sequencing: wait_order()

The **`wait_order`** construct suspends the calling process until all of the specified events are triggered in the

given order (left to right) or any of the un-triggered events are triggered out of order and thus causes the operation to fail.

The syntax for the **wait_order** construct is:

```
wait_statement ::=                                          // from Annex A.6.5
        ...
      | wait_order ( hierarchical_identifier [ , hierarchical_identifier ] ) action_block
action_block ::=
        statement _or_null
      | [ statement ] else statement
```

Syntax 13-2—wait_order event sequencing syntax (excerpt from Annex A)

For **wait_order** to succeed, at any point in the sequence, the subsequent events—which must all be un-triggered at this point, or the sequence would have already failed—must be triggered in the prescribed order. Preceding events are not limited to occur only once. That is, once an event occurs in the prescribed order, it can be triggered again without causing the construct to fail.

Only the first event in the list can wait for the persistent `triggered` property.

The action taken when the construct fails depends on whether or not the optional phrase **else** statement (the fail statement) is specified. If it is specified, then the given statement is executed upon failure of the construct. If the fail statement is not specified, a failure generates a run-time error.

For example:

```
wait_order( a, b, c);
```

suspends the current process until events `a`, `b`, and `c` trigger in the order `a -> b -> c`. If the events trigger out of order, a run-time error is generated.

Example:

```
wait_order( a, b, c ) else $display( "Error: events out of order" );
```

In this example, the fail statement specifies that upon failure of the construct, a user message be displayed, but without an error being generated.

Example:

```
bit success;
wait_order( a, b, c ) success = 1; else success = 0;
```

In this example, the completion status is stored in the variable success, without an error being generated.

13.7 Event variables

An event is a unique data type with several important properties. Unlike Verilog, SystemVerilog events can be assigned to one another. When one event is assigned to another the synchronization queue of the source event is shared by both the source and the destination event. In this sense, events act as full fledged variables and not merely as labels.

13.7.1 Merging events

When one event variable is assigned to another, the two become merged. Thus, executing `->` on either event

variable affects processes waiting on either event variable.

For example:

```
event a, b, c;
a = b;
-> c;
-> a;    // also triggers b
-> b;    // also triggers a
a = c;
b = a;
-> a;    // also triggers b and c
-> b;    // also triggers a and c
-> c;    // also triggers a and b
```

When events are merged, the assignment only affects the execution of subsequent event control or wait operations. If a process is blocked waiting for `event1` when another event is assigned to `event1`, the currently waiting process shall never unblock. For example:

```
fork
   T1: while(1) @ E2;
   T2: while(1) @ E1;
   T3: begin
         E2 = E1;
         while(1) -> E2;
   end
join
```

This example forks off three concurrent processes. Each process starts at the same time. Thus, at the same time that process `T1` and `T2` are blocked, process `T3` assigns event `E1` to `E2`. This means that process `T1` shall never unblock, because the event `E2` is now `E1`. To unblock both threads `T1` and `T2`, the merger of `E2` and `E1` must take place before the fork.

13.7.2 Reclaiming events

When an event variable is assigned the special **`null`** value, the association between the event variable and the underlying synchronization queue is broken. When no event variable is associated with an underlying synchronization queue, the resources of the queue itself become available for re-use.

Triggering a **`null`** event shall have no effect. The outcome of waiting on a **`null`** event is undefined, and implementations can issue a run-time warning.

For example:

```
event E1 = null;
@ E1;                    // undefined: might block forever or not at all
wait( E1.triggered );    // undefined
-> E1;                   // no effect
```

13.7.3 Events comparison

Event variables can be compared against other event variables or the special value **`null`**. Only the following operators are allowed for comparing event variables:

— Equality (`==`) with another event or with **`null`**.

— Inequality (`!=`) with another event or with **`null`**.

— Case equality (`===`) with another event or with **`null`** (same semantics as `==`).

— Case inequality (`!==`) with another event or with **`null`** (same semantics as `!=`).

— Test for a boolean value that shall be 0 if the event is **`null`** and 1 otherwise.

Example:

```
event E1, E2;
if ( E1 )   // same as if ( E1 != null )
   E1 = E2;
if ( E1 == E2 )
   $display( "E1 and E2 are the same event" );
```

Section 14
Scheduling Semantics

14.1 Execution of a hardware model and its verification environment

The balance of the sections of this standard describes the behavior of each of the elements of the language. This section gives an overview of the interactions between these elements, especially with respect to the scheduling and execution of events. Although SystemVerilog is not limited to simulation, the semantics of the language are defined for event directed simulation, and other uses of the hardware description language are abstracted from this base definition.

14.2 Event simulation

The SystemVerilog language is defined in terms of a discrete event execution model. The discrete event simulation is described in more detail in this section to provide a context to describe the meaning and valid interpretation of SystemVerilog constructs. These resulting definitions provide the standard SystemVerilog reference algorithm for simulation, which all compliant simulators shall implement. Note that there is a great deal of choice in the definitions that follow, and differences in some details of execution are to be expected between different simulators. In addition, SystemVerilog simulators are free to use different algorithms than those described in this section, provided the user-visible effect is consistent with the reference algorithm.

A SystemVerilog description consists of connected threads of execution or processes. Processes are objects that can be evaluated, that can have state, and that can respond to changes on their inputs to produce outputs. Processes are concurrently scheduled elements, such as **`initial`** blocks. Example of processes include, but are not limited to, primitives, **`initial`** and **`always`** procedural blocks, continuous assignments, asynchronous tasks, and procedural assignment statements.

Every change in state of a net or variable in the system description being simulated is considered an update event.

Processes are sensitive to update events. When an update event is executed, all the processes that are sensitive to that event are considered for evaluation in an arbitrary order. The evaluation of a process is also an event, known as an evaluation event.

Evaluation events also include PLI callbacks, which are points in the execution model where user-defined external routines can be called from the simulation kernel.

In addition to events, another key aspect of a simulator is time. The term simulation time is used to refer to the time value maintained by the simulator to model the actual time it would take for the system description being simulated. The term time is used interchangeably with simulation time in this section.

To fully support clear and predictable interactions, a single time slot is divided into multiple regions where events can be scheduled that provide for an ordering of particular types of execution. This allows properties and checkers to sample data when the design under test is in a stable state. Property expressions can be safely evaluated, and testbenches can react to both properties and checkers with zero delay, all in a predictable manner. This same mechanism also allows for non-zero delays in the design, clock propagation, and/or stimulus and response code to be mixed freely and consistently with cycle accurate descriptions.

14.3 The stratified event scheduler

A compliant SystemVerilog simulator must maintain some form of data structure that allows events to be dynamically scheduled, executed and removed as the simulator advances through time. The data structure is normally implemented as a time ordered set of linked lists, which are divided and sub-divided in a well defined manner.

The first division is by time. Every event has one and only one simulation execution time, which at any given

point during simulation can be the current time or some future time. All scheduled events at a specific time define a time slot. Simulation proceeds by executing and removing all events in the current simulation time slot before moving on to the next non-empty time slot, in time order. This procedure guarantees that the simulator never goes backwards in time.

A time slot is divided into a set of ordered regions:

1) Preponed

2) Pre-active

3) Active

4) Inactive

5) Pre-NBA

6) NBA

7) Post-NBA

8) Observed

9) Post-observed

10) Reactive

11) Postponed

The purpose of dividing a time slot into these ordered regions is to provide predictable interactions between the design and testbench code.

Except for the Observed and Reactive regions and the Post-observed PLI region, these regions essentially encompass the Verilog 1364-2001 standard reference model for simulation, with exactly the same level of determinism. This means that legacy Verilog code shall continue to run correctly without modification within the new mechanism. The Postponed region is where the monitoring of signals, and other similar events, takes place. No new value changes are allowed to happen in the time slot once the Postponed region is reached.

The Observed and Reactive regions are new in the SystemVerilog 3.1 standard, and events are only scheduled into these new regions from new language constructs.

The Observed region is for the evaluation of the property expressions when they are triggered. It is essential that the signals feeding and producing all the clocks to the property expressions have stabilized, so that the next state of the property expressions can be calculated deterministically. A criterion for this determinism is that the property evaluations must only occur once in any clock triggering time slot. During the property evaluation, pass/fail code shall be scheduled to be executed in the Reactive region of the current time slot.

The sampling time of sampled data for property expressions is controlled in the clock domain block. The new `#1step` sampling delay provides the ability to sample data immediately before entering the current time slot, and is a preferred construct over other equivalent constructs because it allows the `1step` time delay to be parameterized. This `#1step` construct is a conceptual mechanism that provides a method for defining when sampling takes place, and does not require that an event be created in this previous time slot. Conceptually this `#1step` sampling is identical to taking the data samples in the Preponed region of the current time slot.

Code specified in the program block, and pass/fail code from property expressions, are scheduled to occur in the Reactive region.

The Pre-active, Pre-NBA, and Post-NBA are new in the SystemVerilog 3.1 standard but support existing PLI callbacks. The Post-observed region is new in the SystemVerilog 3.1 standard and has been added for PLI support.

The Pre-active region is specifically for a PLI callback control point that allows for user code to read and write values and create events before events in the Active region are evaluated (see Section 14.4).

The Pre-NBA region is specifically for a PLI callback control point that allows for user code to read and write values and create events before the events in the NBA region are evaluated (see Section 14.4).

The Post-NBA region is specifically for a PLI callback control point that allows for user code to read and write values and create events after the events in the NBA region are evaluated (see Section 14.4).

The Post-observed region is specifically for a PLI callback control point that allows for user code to read values after properties are evaluated (in Observed or earlier region).

The flow of execution of the event regions is specified in Figure 14-1.

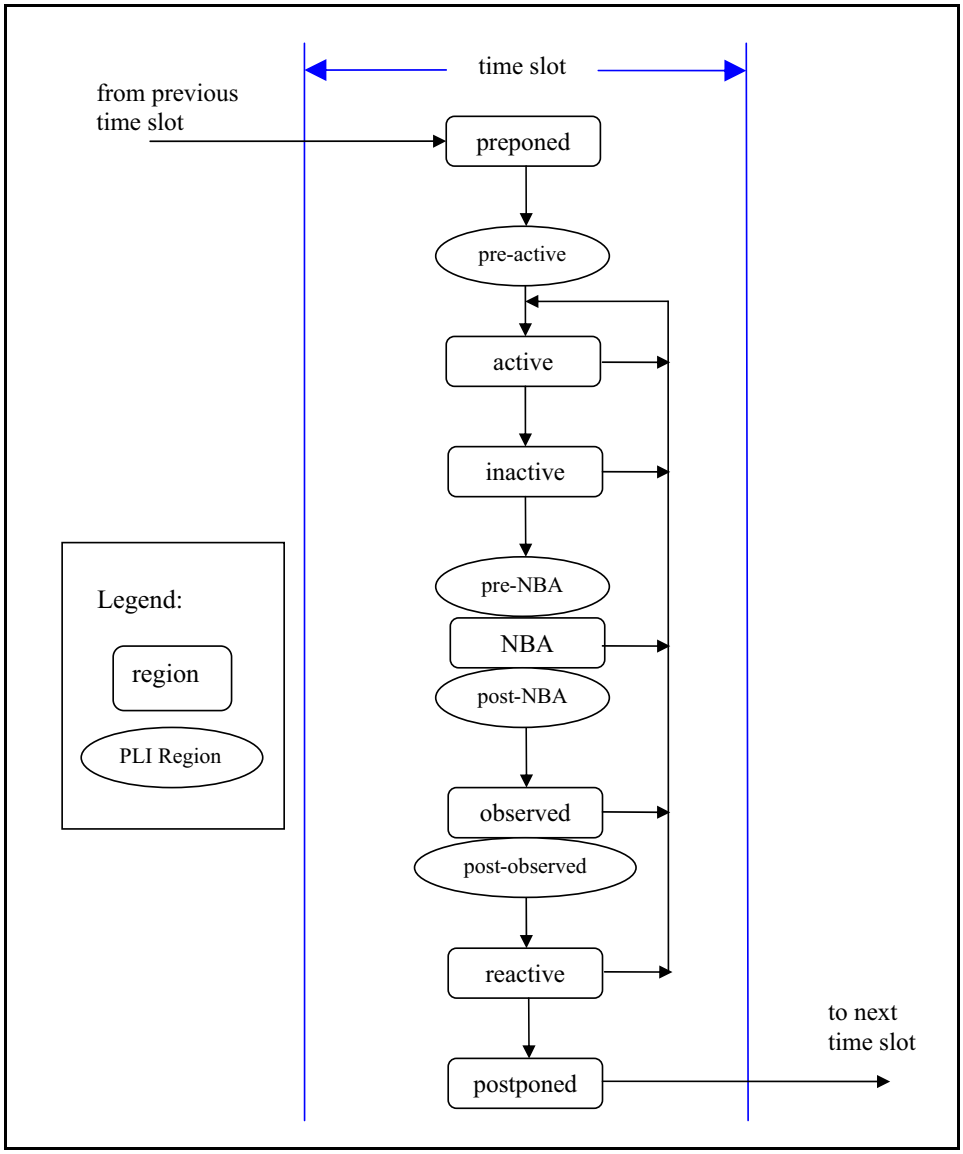


Figure 14-1 — The SystemVerilog flow of time slots and event regions

The Active, Inactive, Pre-NBA, NBA, Post-NBA, Observed, Post-observed and Reactive regions are known as the *iterative* regions.

The Preponed region is specifically for a PLI callback control point that allows for user code to access data at the current time slot before any net or variable has changed state.

The Active region holds current events being evaluated and can be processed in any order.

The Inactive region holds the events to be evaluated after all the active events are processed.

An *explicit zero delay* (#0) requires that the process be suspended and an event scheduled into the Inactive region of the current time slot so that the process can be resumed in the next inactive to active iteration.

A nonblocking assignment creates an event in the NBA region, scheduled for current or a later simulation time.

The Postponed region is specifically for a PLI callback control point that allows for user code to be suspended until after all the Active, Inactive and NBA regions have completed. Within this region, it is illegal to write values to any net or variable, or to schedule an event in any previous region within the current time slot.

14.3.1 The SystemVerilog simulation reference algorithm

```
execute_simulation {
   T = 0;
   initialize the values of all nets and variables;
   schedule all initialization events into time 0 slot;
   while (some time slot is non-empty) {
      move to the next future non-empty time slot and set T;
      execute_time_slot (T);
   }
}

execute_time_slot {
   execute_region (preponed);
   while (some iterative region is non-empty) {
      execute_region (active);
      scan iterative regions in order {
         if (region is non-empty) {
            move events in region to the active region;
            break from scan loop;
         }
      }
   }
   execute_region (postponed);
}

execute_region {
   while (region is non-empty) {
      E = any event from region;
      remove E from the region;
      if (E is an update event) {
         update the modified object;
         evaluate processes sensitive to the object and possibly schedule
            further events for execution;
      } else { /* E is an evaluation event */
         evaluate the process associated with the event and possibly
            schedule further events for execution;
      }
   }
}
```

The Iterative regions and their order are: Active, Inactive, Pre-NBA, NBA, Post-NBA, Observed, Post-

observed and Reactive.

14.4 The PLI callback control points

There are two kinds of PLI callbacks, those that are executed immediately when some specific activity occurs, and those that are explicitly registered as a one-shot evaluation event.

It is possible to explicitly schedule a PLI callback event in any region. Thus, an explicit PLI callback registration is identified by a tuple: (time, region).

The following list provides the mapping from the various current PLI callbacks

Table 14-3: PLI Callbacks

Callback	Identification
`tf_synchronize`	(time, Pre-NBA)
`tf_isynchronize`	(time, Pre-NBA)
`tf_rosynchronize`	(time, Postponed)
`tf_irosynchronize`	(time, Postponed)
`cbReadWriteSynch`	(time, Post-NBA)
`cbAtStartOfSimTime`	(time, Pre-active)
`cbReadOnlySynch`	(time, Postponed)
`cbNBASynch`	(time, Pre-NBA)
`cbAtEndOfSimTime`	(time, Postponed)
`cbNextSimTime`	(time, Pre-active)
`cbAfterDelay`	(time, Pre-active)

Section 15
Clocking Domains

15.1 Introduction (informative)

In Verilog, the communication between blocks is specified using module ports. SystemVerilog adds the interface, a key construct that encapsulates the communication between blocks, thereby enabling users to easily change the level of abstraction at which the inter-module communication is to be modeled.

An interface can specify the signals or nets through which a testbench communicates with a device under test. However, an interface does not explicitly specify any timing disciplines, synchronization requirements, or clocking paradigms.

SystemVerilog adds the **`clocking`** construct that identifies clock signals, and captures the timing and synchronization requirements of the blocks being modeled. A clocking domain assembles signals that are synchronous to a particular clock, and makes their timing explicit. The clocking domain is a key element in a cycle-based methodology, which enables users to write testbenches at a higher level of abstraction. Rather than focusing on signals and transitions in time, the test can be defined in terms of cycles and transactions. Depending on the environment, a testbench can contain one or more clocking domains, each containing its own clock plus an arbitrary number of signals.

The clocking domain separates the timing and synchronization details from the structural, functional, and procedural elements of a testbench. Thus, the timing for sampling and driving clocking domain signals is implicit and relative to the clocking-domain's clock. This enables a set of key operations to be written very succinctly, without explicitly using clocks or specifying timing. These operations are:

— Synchronous events

— Input sampling

— Synchronous drives

15.2 Clocking domain declaration

The syntax for the **`clocking`** construct is:

```
clocking_decl ::= [ default ] clocking [ clocking_identifier ] clocking_event ;        // from Annex A.6.11
                  { clocking_item }
                  endclocking
clocking_event ::=
                  @ identifier
                | @ ( event_expression )
clocking_item :=
                  default default_skew ;
                | clocking_direction list_of_clocking_decl_assign ;
                | { attribute_instance } concurrent_assertion_item_declaration
default_skew ::=
                  input clocking_skew
                | output clocking_skew
                | input clocking_skew output clocking_skew
clocking_direction ::=
                  input [ clocking_skew ]
                | output [ clocking_skew ]
                | input [ clocking_skew ] output [ clocking_skew ]
                | inout
list_of_clocking_decl_assign ::= clocking_decl_assign { , clocking_decl_assign }
clocking_decl_assign ::= signal_identifier [ = hierarchical_identifier ]
clocking_skew ::=
                  edge_identifier [ delay_control ]
                | delay_control
edge_identifier ::= posedge | negedge                                                 // from Annex A.7.4
delay_control ::=                                                                     // from Annex A.6.5
                  # delay_value
                | # ( mintypmax_expression )
```

Syntax 15-1—Class syntax (excerpt from Annex A)

The *delay_control* must be either a time literal or a constant expression that evaluates to a positive integer value.

The *clocking_identifier* specifies the name of the clocking domain being declared.

The *signal_identfier* identifies a signal in the scope enclosing the clocking domain declaration, and declares the name of a signal in the clocking domain. Unless a `hierarchical_expression` is used, both the signal and the `clocking_item` names shall be the same.

The *clocking_event* designates a particular event to act as the clock for the clocking domain. Typically, this expression is either the **`posedge`** or **`negedge`** of a clocking signal. The timing of all the other signals specified in a given clocking domain are governed by the clocking event. All **`input`** or **`inout`** signals specified in the clocking domain are sampled when the corresponding clock event occurs. Likewise, all **`output`** or **`inout`** signals in the clocking domain are driven when the corresponding clock event occurs. Bidirectional signals (**`inout`**) are sampled as well as driven.

The *clocking_skew* determines how many time units away from the clock event a signal is to be sampled or driven. Input skews are implicitly negative, that is, they always refer to a time before the clock, whereas output skews always refer to a time after the clock (see Section 15.3). When the clocking event specifies a simple edge, instead of a number, the skew can be specified as the opposite edge of the signal. A single skew can be specified for the entire domain by using a **`default`** clocking item.

The *hierarchical_identifier* specifies that, instead of a local port, the signal to be associated with the clocking domain is specified by its hierarchical name (cross-module reference).

Example:

```
clocking bus @(posedge clock1);
   default input #10ns output #2ns;
   input data, ready, enable = top.mem1.enable;
   output negedge ack;
   input #1step addr;
endclocking
```

In the above example, the first line declares a clocking domain called `bus` that is to be clocked on the positive edge of the signal `clock1`. The second line specifies that by default all signals in the domain shall use a `10ns` input skew and a `2ns` output skew. The next line adds three input signals to the domain: `data`, `ready`, and `enable`; the last signal refers to the hierarchical signal `top.mem1.enable`. The fourth line adds the signal `ack` to the domain, and overrides the default output skew so that `ack` is driven on the negative edge of the clock. The last line adds the signal `addr` and overrides the default input skew so that `addr` is sampled one step before the positive edge of the clock.

Unless otherwise specified, the default **`input`** skew is `1`**`step`** and the default **`output`** skew is **`0`**. A **`step`** is a special time unit whose value is defined in Section 18.6. A `1`**`step`** input skew allows input signals to sample their steady-state values in the time step immediately before the clock event (i.e., in the preceding Postponed region). Unlike other time units, which represent physical units, a step cannot be used to set or modify either the precision or the timeunit.

15.3 Input and output skews

Input (or inout) signals are sampled at the designated clock event. If an input skew is specified then the signal is sampled at *skew* time units *before* the clock event. Similarly, output (or inout) signals are driven *skew* simulation time units *after* the corresponding clock event. Figure 15-1 shows the basic sample/drive timing for a positive edge clock.

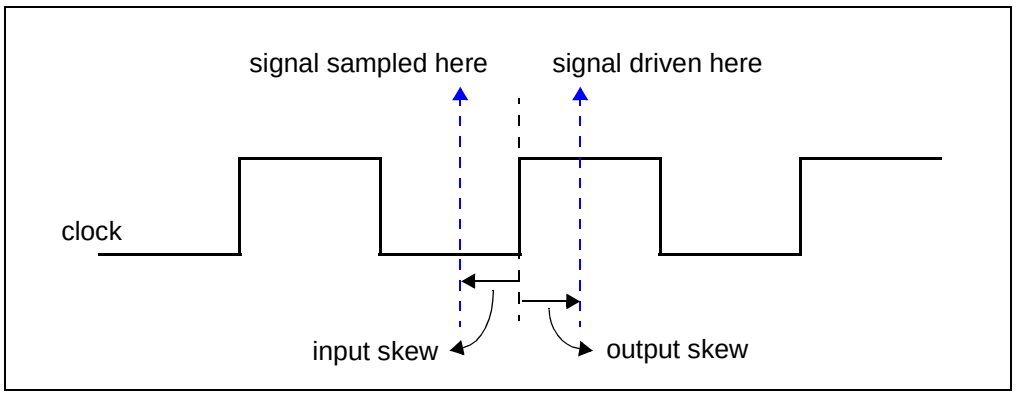


Figure 15-1 — Sample and drive times including skew with respect to the positive edge of the clock.

A skew must be a constant expression, and can be specified as a parameter. If the skew does not specify a time unit, the current time unit is used. If a number is used, the skew is interpreted using the timescale of the current scope.

```
clocking dram @(clk);
   input #1ps address;
   input #5 output #6 data;
```

```
endclocking
```

An input skew of 1**step** indicates that the signal is to be sampled at the end of the previous time step. That is, the value sampled is always the signal's last value immediately before the corresponding clock edge.

An input skew of **#0** forces a skew of zero. Inputs with zero skew are sampled at the same time as their corresponding clocking event, but to avoid races, they are sampled in the Observed region. Likewise, outputs with zero skew are driven at the same time as their specified clocking event, as nonblocking assignments (in the NBA region).

Skews are declarative constructs, thus, they are semantically very different from the syntactically similar procedural delay statement. In particular, a #0 skew, does not suspend any process nor does it execute or sample values in the Inactive region.

15.4 Hierarchical expressions

Any signal in a clocking domain can be associated with an arbitrary hierarchical expression. As described in Section 15.2, a hierarchical expression is introduced by appending an equal sign (**=**) followed by the hierarchical expression:

```
clocking cd1 @(posedge phi1);
      input #1step state = top.cpu.state;
endclocking
```

However, hierarchical expressions are not limited to simple names or signals in other scopes. They can be used to declare slices and concatenations (or combinations thereof) of signals in other scopes or in the current scope.

```
clocking mem @(clock);
   input instruction = { opcode, regA, regB[3:1] };
endclocking
```

15.5 Signals in multiple clocking domains

The same signals—clock, inputs, inouts, or outputs—can appear in more than one clocking domain. Clocking domains that use the same clock (or clocking expression) shall share the same synchronization event, in the same manner as several latches can be controlled by the same clock. Input semantics are described in Section 15.12, and output semantics are described in Section 15.14.

15.6 Clocking domain scope and lifetime

A **clocking** construct is both a declaration and an instance of that declaration. A separate instantiation step is not necessary. Instead, one copy is created for each instance of the block containing the declaration (like an always block). Once declared, the clocking signals are available via the clock-domain name and the dot (**.**) operator:

```
dom.sig  // signal sig in clocking dom
```

Clocking domains cannot be nested. They cannot be declared inside functions or tasks, or at the global (`$root`) level. Clocking domains can only be declared inside a module, interface or program (see Section 16).

Clocking domains have static lifetime and scope local to their enclosing module, interface or program.

15.7 Multiple clocking domains example

In this example, a simple test program includes two clocking domains. The program construct used in this

example is discussed in Section 16.

```
program test(  input phi1, input [15:0] data, output logic write,
               input phi2, inout [8:1] cmd, input enable
            );
   reg [8:1] cmd_reg;

   clocking cd1 @(posedge phi1);
      input data;
      output write;
      input state = top.cpu.state;
   endclocking

   clocking cd2 @(posedge phi2);
      input #2 output #4ps cmd;
      input enable;
   endclocking

   initial begin
      // program begins here
   ...
      // user can access cd1.data , cd2.cmd , etc…
   end
   assign cmd = enable ? cmd_reg: 'x;
endprogram
```

The test program can be instantiated and connected to a device under test (`cpu` and `mem`).

```
module top;
   logic phi1, phi2;
   wire [8:1] cmd; // cannot be logic (two bidirectional drivers)
   logic [15:0] data;

   test main( phi1, data, write, phi2, cmd, enable );
   cpu cpu1( phi1, data, write );
   mem mem1( phi2, cmd, enable );
endmodule
```

15.8 Interfaces and clocking domains

A **`clocking`** encapsulates a set of signals that share a common clock, therefore, specifying a clocking domain using a SystemVerilog **`interface`** can significantly reduce the amount of code needed to connect the testbench. Furthermore, since the signal directions in the clocking domain within the testbench are with respect to the testbench, and not the design under test, a **`modport`** declaration can appropriately describe either direction. A testbench program can be contained within a *program* and its ports can be interfaces that correspond to the signals declared in each clocking domain. The interface's wires shall have the same direction as specified in the clocking domain when viewed from the testbench side (i.e., **`modport`** `test`), and reversed when viewed from the device under test (i.e., **`modport`** `dut`).

For example, the previous example could be re-written using interfaces as follows:

```
interface bus_A (input clk);
      logic [15:0] data;
      logic write;
      modport test (input data, output write);
      modport dut (output data, input write);
endinterface
```

```
interface bus_B (input clk);
      logic [8:1] cmd;
      logic enable;
      modport test (input enable);
      modport dut (output enable);
endinterface


program test( bus_A.test a, bus_B.test b );

      clocking cd1 @(posedge a.clk);
         input a.data;
         output a.write;
         inout state = top.cpu.state;
      endclocking

      clocking cd2 @(posedge b.clk);
         input #2 output #4ps b.cmd;
         input b.enable;
      endclocking

   initial begin
      // program begins here
      ...
      // user can access cd1.a.data , cd2.b.cmd , etc…
   end
endprogram
```

The test module can be instantiated and connected as before:

```
module top;
   logic phi1, phi2;

   bus_A a(phi1);
   bus_B b(phi2);

   test main( a, b );
   cpu cpu1( a );
   mem mem1( b );
endmodule
```

Alternatively, in the program test above, the clocking domain can be written using both interfaces and hierarchical expressions as:

```
   clocking cd1 @(posedge a.clk);
      input data = a.data;
      output write = a.write;
      inout state = top.cpu.state;
   endclocking

   clocking cd2 @(posedge b.clk);
      input #2 output #4ps cmd = b.cmd;
      input enable = b.enable;
   endclocking
```

This would allow using the shorter names (`cd1.data`, `cd2.cmd`, …) instead of the longer interface syntax (`cd1.a.data`, `cd2.b.cmd`,…).

15.9 Clocking domain events

The clocking event of a clocking domain is available directly by using the clocking domain name, regardless of the actual clocking event used to declare the clocking domain.

For example.

```
clocking dram @(posedge phi1);
   inout data;
   output negedge #1 address;
endclocking
```

The clocking event of the `dram` domain can be used to wait for that particular event:

```
@( dram );
```

The above statement is equivalent to `@(posedge phi1)`.

15.10 Cycle delay:

The **##** operator can be used to delay execution by a specified number of clocking events, or clock cycles.

The syntax for the cycle delay statement is:

```
cycle_delay_range ::=                                          // from Annex A.2.10
          ## constant_expression
        | ## [ cycle_delay_const_range_expression ]
cycle_delay_const_range_expression ::=
          constant_expression : constant_expression
        | constant_expression : $
```

Syntax 15-2—Cycle delay syntax (excerpt from Annex A)

The *constant_expression* can be any SystemVerilog expression that evaluates to a positive integer value.

What constitutes a cycle is determined by the default clocking in effect (see Section 15.11). If no default clocking has been specified for the current module, interface, or program then the compiler shall issue an error.

Example:

```
## 5;       // wait 5 cycles (clocking events) using the default clocking

## j + 1;   // wait j+1 cycles (clocking events) using the default clocking
```

15.11 Default clocking

One **clocking** can be specified as the default for all cycle delay operations within a given **module**, **interface**, or **program**.

The syntax for the default cycle specification statement is:

```
module_or_generate_item_declaration ::=                          // from Annex A.1.5
          ...
        | default clocking clocking_identifier ;
clocking_decl ::= [ default ] clocking [ clocking_identifier ] clocking_event ;   // from Annex A.6.11
          { clocking_item }
          endclocking
```

Syntax 15-3—Default clocking syntax (excerpt from Annex A)

The *clocking_identifier* must be the name of a clocking domain.

Only one default clocking can be specified in a program, module, or interface. Specifying a default clocking more than once in the same program or module shall result in a compiler error.

A default clocking is valid only within the scope containing the default clocking specification. This scope includes the module, interface, or program that contains the declaration as well as any nested modules or interfaces. It does not include instantiated modules or interfaces.

Example 1. Declaring a clocking as the default:

```
program test( input bit clk, input reg [15:0] data )
   default clocking bus @(posedge clk);
      inout data;
   endclocking

   initial begin
      ## 5;
      if ( bus.data == 10 )
         ## 1;
      else
         ...
   end
endprogram
```

Example 2. Assigning an existing clocking to be the default:

```
module processor ...
   clocking busA @(posedge clk1); ... endclocking
   clocking busB @(negedge clk2); ... endclocking
   module cpu( interface y )
      default clocking busA ;
      initial begin
         ## 5; // use busA => (posedge clk1)
         ...
      end
   endmodule
endmodule
```

15.12 Input sampling

All clocking domain inputs (input or inout) are sampled at the corresponding clocking event. If the input skew is non-zero, then the value sampled corresponds to the signal value at the Postponed region of the time step skew time-units prior to the clocking event (see Figure 15-1 in Section 15.3). If the input skew is zero, then the value sampled corresponds to the signal value in the Observed region.

Samples happen immediately (the calling process does not block). When a signal appears in an expression, it is replaced by the signal's sampled value, that is, the value that was sampled at the last sampling point.

When the same signal is an input to multiple clocking domains, the semantics are straightforward; each clocking domain samples the corresponding signal with its own clocking event.

15.13 Synchronous events

Explicit synchronization is done via the event control operator, `@`, which allows a process to wait for a particular signal value change, or a clocking event (see Section 15.9).

The syntax for the synchronization operator is given in Section 8.10.

The expression used with the event control can denote clocking-domain input (**`input`** or **`inout`**), or a slice thereof. Slices can include dynamic indices, which are evaluated once, when the `@` expression executes.

These are some example synchronization statements:

— Wait for the next change of signal `ack_1` of clock-domain `ram_bus`

```
@(ram_bus.ack_l);
```

— Wait for the next clocking event in clock-domain `ram_bus`

```
@(ram_bus);
```

— Wait for the positive edge of the signal `ram_bus.enable`

```
@(posedge ram_bus.enable);
```

— Wait for the falling edge of the specified 1-bit slice `dom.sign[a]`. Note that the index `a` is evaluated at runtime.

```
@(negedge dom.sign[a]);
```

— Wait for either the next positive edge of `dom.sig1` or the next change of `dom.sig2`, whichever happens first.

```
@(posedge dom.sig1 or dom.sig2);
```

— Wait for the either the negative edge of `dom.sig1` or the positive edge of `dom.sig2`, whichever happens first.

```
@(negedge dom.sig1 or posedge dom.sig2);
```

The values used by the synchronization event control are the synchronous values, that is, the values sampled at the corresponding clocking event.

15.14 Synchronous drives

Clocking domain outputs (**`output`** or **`inout`**) are used to drive values onto their corresponding signals, but at a specified time. That is, the corresponding signal changes value at the indicated clocking event as modified by the output skew.

The syntax to specify a synchronous drive is similar to an assignment:

```
statement ::= [ block_identifier : ] statement_item                    // from Annex A.6.4
statement_item ::=
                ...
              | { attribute_instance } clocking_drive
clocking_drive ::=                                                     // from Annex A.6.11
                clockvar_expression <= [ cycle_delay ] expression
              | cycle_delay clockvar_expression <= expression
cycle_delay ::= ## expression
clockvar ::= clocking_identifier . identifier
clockvar_expression ::=
                clockvar range
              | clockvar [ range_expression ]
```

Syntax 15-4—Default clocking syntax (excerpt from Annex A)

The *clockvar_expression* is either a bit-select, slice, or the entire clocking domain output whose corresponding signal is to be driven (concatenation is not allowed):

```
dom.sig           // entire clockvar

dom.sig[2]        // bit-select

dom.sig[8:2]      // slice
```

The expression can be any valid expression that is assignment compatible with the type of the corresponding signal.

The *event_count* is an integral expression that optionally specifies the number of clocking events (i.e. cycles) that must pass before the statement executes. Specifying a non-zero `event_count` blocks the current process until the specified number of clocking events have elapsed, otherwise the statement executes at the current time. The `event_count` uses syntax similar to the cycle-delay operator (see Section 15.10), however, the synchronous drive uses the clocking domain of the signal being driven and not the default clocking.

The second form of the synchronous drive uses the intra-assignment syntax. An intra-assignment `event_count` specification also delays execution of the assignment. In this case the process does not block and the right-hand side expression is evaluated when the statement executes.

Examples:

```
bus.data[3:0] <= 4'h5;  // drive data in current cycle

##1 bus.data <= 8'hz;   // wait 1 (bus) cycle and then drive data

##2; bus.data <= 2;     // wait 2 default clocking cycles, then drive data

bus.data <= ##2 r;      // remember the value of r and then drive
                        // data 2 (bus) cycles later
```

Regardless of when the drive statement executes (due to event_count delays), the driven value is assigned to the corresponding signal only at the time specified by the output skew.

15.14.1 Drives and nonblocking assignments

Synchronous signal drives are processed as nonblocking assignments.

A key feature of **inout** clocking domain variables and synchronous drives is that a drive does not change the clock domain input. This is because reading the input always yields the last sampled value, and not the driven value.

15.14.2 Drive value resolution

When more than one synchronous drive is applied to the same clocking domain output (or inout) at the same simulation time, the driven values are checked for conflicts. When conflicting drives are detected a runtime error is issued, and each conflicting bit is driven to X (or 0 for a 2-state port).

For example:

```
clocking pe @(posedge clk);
   output nibble;    // four bit output
endclocking

pe.nibble <= 4'b0101;
pe.nibble <= 4'b0011;
```

The driven value of `nibble` is `4'b0xx1`, regardless of whether `nibble` is a **reg** or a **wire**.

When the same variable is an output from multiple clocking domains, the last drive determines the value of the variable. This allows a single module to model multi-rate devices, such as a DDR memory, using a different clocking domain to model each active edge. For example:

```
reg j;

clocking pe @(posedge clk);
   output j;
endclocking

clocking ne @(negedge clk);
   output j;
endclocking
```

The variable `j` is an output to two clocking domains using different clocking events (**posedge** vs. **negedge**). When driven, the variable `j` shall take on the value most recently assigned by either clocking domain.

Clock-domain outputs driving a net (i.e. through different ports) cause the net to be driven to its resolved signal value. When a clock-domain output corresponds to a wire, a driver for that wire is created that is updated as if by a continuous assignment from a register inside the clock-domain that is updated as a nonblocking assignment.

Section 16
Program Block

16.1 Introduction (informative)

The module is the basic building block in Verilog. Modules can contain hierarchies of other modules, wires, task and function declarations, and procedural statements within always and initial blocks. This construct works extremely well for the description of hardware. However, for the testbench, the emphasis is not in the hardware-level details such as wires, structural hierarchy, and interconnects, but in modeling the complete environment in which a design is verified. A lot of effort is spent getting the environment properly initialized and synchronized, avoiding races between the design and the testbench, automating the generation of input stimuli, and reusing existing models and other infrastructure.

The program block serves three basic purposes:

1) It provides an entry point to the execution of testbenches.

2) It creates a scope that encapsulates program-wide data.

3) It provides a syntactic context that specifies execution in the Reactive region.

The program construct serves as a clear separator between design and testbench, and, more importantly, it specifies specialized execution semantics in the Reactive region for all elements declared within the program. Together with clocking domains, the program construct provides for race-free interaction between the design and the testbench, and enables cycle and transaction level abstractions.

The abstraction and modeling constructs of SystemVerilog simplify the creation and maintenance of test-benches. The ability to instantiate and individually connect each program instance enables their use as generalized models.

16.2 The program construct

A typical program contains type and data declarations, subroutines, connections to the design, and one or more procedural code streams. The connection between design and testbench uses the same interconnect mechanism as used by SystemVerilog to specify port connections, including interfaces. The syntax for the program block is:


```
program_nonansi_header ::=                                   // from Annex A.1.3
        { attribute_instance } program [ lifetime ] program_identifier
        [ parameter_port_list ] list_of_ports ;
program_ansi_header ::=
        {attribute_instance } program [ lifetime ] program_identifier
        [ parameter_port_list ] [ list_of_port_declarations ] ;
program_declaration ::=
        program_nonansi_header [ timeunits_declaration ] { program_item }
        endprogram [ : program_identifier ]
      | program_ansi_header [ timeunits_declaration ] { non_port_program_item }
        endprogram [ : program_identifier ]
      | { attribute_instance } program program_identifier ( .* ) ;
        [ timeunits_declaration ] { program_item }
        endprogram [ : program_identifier ]
      | extern program_nonansi_header
      | extern program_ansi_header
program_item ::=                                             // from Annex A.1.7
        port_declaration ;
      | non_port_program_item
non_port_program_item ::=
        { attribute_instance } continuous_assign
      | { attribute_instance } module_or_generate_item_declaration
      | { attribute_instance } specparam_declaration
      | { attribute_instance } local_parameter_declaration
      | { attribute_instance } parameter_declaration ;
      | { attribute_instance } initial_construct
      | { attribute_instance } concurrent_assertion_item
      | class_declaration
lifetime ::= static | automatic                              // from Annex A.2.1.3
```

Syntax 16-1—Program declaration syntax (excerpt from Annex A)

For example:

```
program test (input clk, input [16:1] addr, inout [7:0] data);
   initial ...
endprogram
```

or

```
program test ( interface device_ifc );
   initial ...
endprogram
```

A more complete example is included in Sections 15.7 and 15.8.

Although the **program** construct is new to SystemVerilog, its inclusion is a natural extension. The **program** construct can be considered a leaf module with special execution semantics. Once declared, a program block can be instantiated in the required hierarchical location (typically at the top level) and its ports can be connected in the same manner as any other module.

Program blocks can be nested within modules or interfaces. This allows multiple cooperating programs to

share variables local to the scope. Nested programs with no ports or top-level programs that are not explicitly instantiated are implicitly instantiated once. Implicitly instantiated programs have the same instance and declaration name. For example:

```
module test(...)
   int shared; // variable shared by programs p1 and p1

   program p1;
      ...
   endprogram

   program p2;
      ...
   endprogram // p1 and p2 are implicitly instantiated once in module test

endmodule
```

A program block can contain one or more **`initial`** blocks. It can not contain **`always`** blocks, UDPs, modules, interfaces, or other programs.

Type and data declarations within the program are local to the program scope and have static lifetime. Program variables can only be assigned using blocking assignments. Non-program variables can only be assigned using nonblocking assignments. Using nonblocking assignments with program variables or blocking assignments with design (non-program) variables shall be an error.

16.3 Multiple programs

It is allowed to have any arbitrary number of program definitions or instances. The programs can be fully independent (without inter-program communication), or cooperative. The degree of communication can be controlled by choosing to share data using nested blocks or hierarchical references (including **`$root`**), or making the data private by declaring it inside the corresponding program block.

16.4 Eliminating testbench races

There are two major sources of non-determinism in Verilog. The first one is that active events are processed in an arbitrary order. The second one is that statements without time-control constructs in behavioral blocks do not execute as one event. However, from the testbench perspective, these effects are all unimportant details. The primary task of a testbench is to generate valid input stimulus for the design under test, and to verify that the device operates correctly. Furthermore, testbenches that use cycle abstractions are only concerned with the stable or steady state of the system for both checking the current outputs and for computing stimuli for the next cycle. Formal tools also work in this fashion.

To avoid the races inherent in the Verilog event scheduler, program statements are scheduled to execute in the Reactive region, after all clocks in the design have triggered and the design has settled to its steady state. In addition, design signals driven from within the program must be assigned using nonblocking assignments. Thus, even signals driven with no delay are propagated into the design as one event. With this behavior, correct cycle semantics can be modeled without races; thereby making program-based testbenches compatible with clocked assertions and formal tools.

Since the program executes in the Reactive region, the clocking domain construct is very useful to automatically sample the steady-state values of previous time steps or clock cycles. Programs that read design values exclusively through clocking domains with non-zero input skews are insensitive to read-write races. It is important to note that simply sampling input signals (or setting non-zero skews on clock domain inputs) does not eliminate the potential for races. Proper input sampling only addresses a single clocking domain. With multiple clocks, the arbitrary order in which overlapping or simultaneous clocks are processed is still a potential source for races. The program construct addresses this issue by scheduling its execution in the Reactive region, after all design events have been processed, including clocks driven by nonblocking assignments.

16.4.1 Zero-skew clocking domain races

When a clocking domain sets both input and output skews to `#0` (see Section 15.3) then its inputs are sampled at the same time as its outputs are driven. This type of zero-delay processing is a common source of non-determinism that can result in races. Nonetheless, even in this case, the program minimizes races by means of two mechanisms. First, by constraining program statements to execute in the Reactive region, after all zero-delay transitions have propagated through the design and the system has reached a quasi steady state. Second, by requiring design variables or nets to be modified only via nonblocking assignments. These two mechanisms reduce the likelihood of a race; nonetheless, a race is still possible when skews are set to zero.

16.5 Blocking tasks in cycle/event mode

Calling program tasks or functions from within design modules is illegal and shall result in an error. This is because the design must not be aware of the testbench. Programs are allowed to call tasks or functions in other programs or within design modules. Functions within design modules can be called from a program, and require no special handling. However, blocking tasks within design modules that are called from a program do require explicit synchronization upon return from the task. That is, when blocking tasks return to the program code, the program block execution is automatically postponed until the Reactive region. The copy out of the parameters happens when the task returns.

Calling blocking tasks in design modules from within programs requires careful consideration. Expressions evaluated by the task before blocking on the first timing control shall use the values after they have been updated by nonblocking assignments. In contrast, if the task is called from a module at the start of the time step (before nonblocking assignments are processed) then those same expressions shall use the values before they have been updated by nonblocking assignments.

```
module ...
   task T;
      S1: a = b;     // might execute before or after the Observe region
      #5;
      S2: b <= 1'b1; // always executes before the Observe region
   endtask
endmodule
```

If task `T`, above, is called from within a module, then the statement `S1` can execute immediately when the Active region is processed, before variable b is updated by a nonblocking assignment. If the same task is called from within a program, then the statement `S1` shall execute when the Reactive region is processed, after variable `b` might have been updated by nonblocking assignments. Statement `S2` always executes immediately after the delay expires; it does not wait for the Reactive region even though it was originally called from the program block.

16.6 Program control tasks

In addition to the normal simulation control tasks (`$stop` and `$finish`), a program can use the `$exit` control task.

16.6.1 $exit()

Each program can be finished by calling the `$exit` system task. When all programs exit, the simulation finishes.

The syntax for the `$exit` system task is:

```
task $exit();
```

When all **`initial`** blocks in a program finish (i.e., they execute their last statement), the program implicitly calls `$exit`. Calling `$exit` causes all processes spawned by the current program to be terminated.

Section 17
Assertions

17.1 Introduction (informative)

SystemVerilog adds features to specify assertions of a system. An assertion specifies a behavior of the system. Assertions are primarily used to validate the behavior of a design. In addition, assertions can be used to provide functional coverage and generate input stimulus for validation.

There are two kinds of assertions: concurrent and immediate.

— Immediate assertions follow simulation event semantics for their execution and are executed like a statement in a procedural block. Immediate assertions are primarily intended to be used with simulation.

— Concurrent assertions are based on clock semantics and use sampled values of variables. One of the goals of SystemVerilog assertions is to provide a common semantic meaning for assertions so that they can be used to drive various design and verification tools. Many tools, such as formal verification tools, evaluate circuit descriptions using a cycle-based semantic, which typically relies on a clock signal or signals to drive the evaluation of the circuit. Any timing or event behavior between clock edges is abstracted away. Concurrent assertions incorporate this clock semantic. While this approach generally simplifies the evaluation of a circuit description, there are a number of scenarios under which this cycle-based evaluation provides different behavior from the standard event-based evaluation of SystemVerilog.

This section describes both types of assertions.

17.2 Immediate assertions

The immediate assertion statement is a test of an expression performed when the statement is executed in the procedural code. The expression is non-temporal and treated as a condition as in an **`if`** statement. The immediate **`assert`** statement is a *statement_item* and can be specified anywhere a procedural statement is specified.

```
procedural_assertion_item ::=                                   // from Annex A.6.10
        ...
        | immediate_assert_statement
immediate_assert_statement ::=
        assert ( expression ) action_block
action_block ::=                                                // from Annex A.6.3
          statement _or_null
        | [ statement ] else statement
```

Syntax 17-1—Immediate assertion syntax (excerpt from Annex A)

The *action_block* specifies what actions are taken upon success or failure of the assertion. The statement associated with the success of the assert statement is the first statement. It is called the *pass statement* and is executed if the expression evaluates to true. The evaluation of the expression follows the same semantic as that of the conditional context of the **`if`** statement. As with the **`if`** statement, if the conditional expression evaluates to `x`, `z` or `0`, then the assertion fails. The pass statement can, for example, record the number of successes for a coverage log, but can be omitted altogether. If the pass statement is omitted, then no user-specified action is taken when the assert expression is true. The statement associated with **`else`** is called a *fail statement* and is executed if the assertion fails. That is, the expression does not evaluate to a known, non-zero value. The **`else`** statement can also be omitted. The action block is executed immediately after the evaluation of the assert expression.

The optional statement label (identifier and colon) creates a named block around the assertion statement (or any other SystemVerilog statement) and can be displayed using the `%m` format specification.

```
assert_foo : assert(foo) $display("%m passed"); else $display("%m failed");
```

Note: The assertion control system tasks are described in Section 22.6.

Since the assertion is a statement that something must be true, the failure of an assertion shall have a severity associated with it. By default, the severity of an assertion failure is *error*. Other severity levels can be specified by including one of the following severity system tasks in the fail statement:

— `$fatal` is a run-time fatal, which shall terminate the simulation with an error code. The first argument passed to `$fatal` shall be consistent with the argument to `$finish`.

— `$error` is a run-time error.

— `$warning` is a run-time warning, which can be suppressed in a tool-specific manner.

— `$info` indicates that the assertion failure carries no specific severity.

The syntax for these system tasks is shown in Section 22.5.

If an assertion fails and no **`else`** clause is specified, the tool shall, by default, call `$error`, unless a tool-specific option, such as a command-line option, is enabled to suppress the failure.

All of these severity system tasks shall print a tool-specific message indicating the severity of the failure, and specific information about the specific failure, which shall include the following information:

— The file name and line number of the assertion statement.

— The hierarchical name of the assertion, if it is labeled, or the scope of the assertion if it is not labeled.

For simulation tools, these tasks shall also include the simulation run-time at which the severity system task is called.

Each system task can also include additional user-specified information using the same format as the Verilog `$display`.

If more than one of these system tasks is included in the **`else`** clause, then each shall be executed as specified.

If the severity system task is executed at a time other than when the assertion fails, the actual failure time of the assertion can be recorded and displayed programmatically. For example:

```
time t;

always @(posedge clk)
   if (state == REQ)
      assert (req1 || req2)
      else begin
         t = $time;
         #5 $error("assert failed at time %0t",t);
      end
```

If the assertion fails at time 10, the error message shall be printed at time 15, but the user-defined string printed shall be “assert failed at time 10”.

The display of messages of warning and info types can be controlled by a tool-specific option, such as a command-line option.

Since the fail statement, like the pass statement, is any legal SystemVerilog procedural statement, it can also be used to signal a failure to another part of the testbench.

```
assert (myfunc(a,b)) count1 = count + 1; else ->event1;
assert (y == 0) else flag = 1;
```

17.3 Concurrent assertions overview

Concurrent assertions describe behavior that spans over time. Unlike immediate assertions, the evaluation model is based on a clock such that a concurrent assertion is evaluated only at the occurrence of a clock tick. The values of variables used in the evaluation are the sampled values. This way, a predictable result can be obtained from the evaluation, regardless of the simulator's internal mechanism of ordering events and evaluating events. This model of execution also corresponds to the synthesis model of hardware interpretation from an RTL description.

The values of variables used in assertions are sampled in the Preponed region of a time slot and the assertions are evaluated during the Observe region. This is explained in Section 14, Scheduling Semantics.

The timing model employed in a concurrent assertion specification is based on clock ticks and uses a generalized notion of clock cycles. The definition of a clock is explicitly specified by the user and can vary from one expression to another.

A *clock tick* is an atomic moment in time and implies that there is no duration of time in a clock tick. It is also given that a clock shall tick only once at any simulation time, and the sampled values for that simulation time are used for evaluation. In an assertion, the sampled value is the only valid value of a variable at a clock tick. Figure 17-1 shows the values of a variable as the clock progresses. The value of signal `req` is low at clock ticks 1 and 2. At clock tick 3, the value is sampled as high and remains high until clock tick 6. The sampled value of variable `req` at clock tick 6 is low and remains low until clock tick 10. Notice that, at clock tick 9, the simulation value transitions to high. However, the sampled value is low.

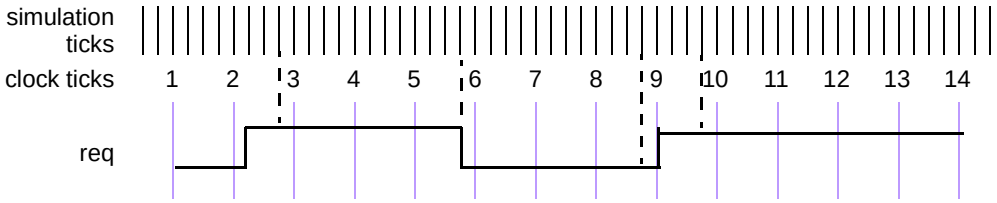


Figure 17-1 — Sampling a variable on simulation ticks

An expression used in an assertion is always tied to a clock definition. The sampled values are used to evaluate value change expressions or boolean sub-expressions that are required to determine a match with respect to a sequence expression.

Note:

— It is important to ensure that the defined clock behavior is glitch free. Otherwise, wrong values can be sampled.

— If a variable that appears in the expression for clock also appears in an expression for the assertion, the values of the two usages of the variable can be different. The value of the variable used in the clock expression is the current value, while for the assertion the sampled value of the variable is used.

The clock expression that controls evaluation of a sequence can be more complex than just a single signal name. An expression such as `(clk && gating_signal) and (clk iff gating_signal)` could be used to represent gated clocks. Other more complex expressions are possible. In order to ensure proper behavior of the system and conform as closely as possible to truly cycle-based semantics, the signals in a clock expression must be glitch-free and should only transition once at any simulation time.

An example of a concurrent assertion is:


```
base_rule1: assert property (cont_prop(rst,in1,in2)) pass_stat else fail_stat;
```

The keyword **property** distinguishes a concurrent assertion from an immediate assertion. The syntax of concurrent assertions is discussed in 17.12.

17.4 Boolean expressions

The expressions used in sequences are evaluated over sampled values of the variables that appear in the expression. The outcome of the evaluation of an expressions is boolean and is interpreted the same way as an expression is interpreted in the condition of a procedural **if** statement. That is, if the expression evaluates to `x`, `z`, or `0`, then it is interpreted as being false. Otherwise, it is true.

There are certain restrictions on the expressions that can appear in concurrent assertions. The restrictions on operand types, variables, and operators are specified in the following sections.

17.4.1 Operand types

The following types are not allowed:

— non-integer types (**time**, **shortreal**, **real** and **realtime**)

— **string**

— **event**

— **chandle**

— **class**

— associative arrays

— dynamic arrays

Fixed size arrays, packed or unpacked, can be used as a whole or as part selects or as indexed bit or part selects. The indices can be constants, parameters, or variables.

The following example shows some possible forms of comparison of over members of structures and unions:

```
typedef int [4] array;
typedef struct { int a, b, c,d } record;
union { record r; array a; } p, q;
```

The following comparisons are legal in expressions:

```
p.a == q.a
```

and

```
p.r == q.r
```

The following example provides further illustration of the use of arrays in expressions.

```
logic [7:0] arrayA [0:15], arrayB[0:15];
```

The following comparisons are legal:

```
arrayA == arrayB;
arrayA != arrayB;
arrayA[i] >= arrayB[j];
arrayB[i][j+:2] == arrayA[k][m-:2];
(arrayA[i] & (~arrayB[j])) == 0;
```

17.4.2 Variables

The variables that can appear in expressions must be static design variables or function calls returning values of types described in Section 17.4.1. The functions should be automatic (or preserve no state information) and pure (no output arguments, no side effects). Static variables declared in programs, interfaces or clocking domains can also be accessed. If a reference is to a static variable declared in a task, that variable is sampled as any other variable, independent of calls to the task.

17.4.3 Operators

All operators that are valid for the types described in Section 17.4.1 are allowed with the exception of assignment operators or increment and decrement operators. SystemVerilog includes the C assignment operators, such as +=, and the C increment and decrement operators, ++ and --. These operators cannot be used in expressions that appear in assertions. This restriction prevents side effects.

17.5 Sequences

```
sequence_expr ::=                                                    // from Annex A.2.10
        cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      | expression { , function_blocking_assignment } [ boolean_abbrev ]
      | ( expression {, function_blocking_assignment } ) [ boolean_abbrev ]
      | sequence_instance [ sequence_abbrev ]
      | ( sequence_expr ) [ sequence_abbrev ]
      | sequence_expr and sequence_expr
      | sequence_expr intersect sequence_expr
      | sequence_expr or sequence_expr
      | first_match ( sequence_expr )
      | expression throughout sequence_expr
      | sequence_expr within sequence_expr
cycle_delay_range ::=
        ## constant_expression
      | ## [ cycle_delay_const_range_expression ]
sequence_instance ::=
        sequence_identifier [ ( actual_arg_list ) ]
formal_list_item ::=
      formal_identifier [ = actual_arg_expr ]
actual_arg_list ::=
        ( actual_arg_expr { , actual_arg_expr } )
      | ( . formal_identifier ( actual_arg_expr ) { , . formal_identifier ( actual_arg_expr ) } )
actual_arg_expr ::=
        event_expression
boolean_abbrev ::=
        consecutive_repetition
      | non_consecutive_repetition
      | goto_repetition
sequence_abbrev ::= consecutive_repetition
consecutive_repetition ::= [* const_or_range_expression ]
non_consecutive_repetition ::= [*= const_or_range_expression ]
goto_repetition ::= [*> const_or_range_expression ]
const_or_range_expression ::=
        constant_expression
      | cycle_delay_const_range_expression
cycle_delay_const_range_expression ::=
        constant_expression : constant_expression
      | constant_expression : $
```

Syntax 17-2—Sequence syntax (excerpt from Annex A)

Properties are often constructed out of sequential behavior. The **`sequence`** feature provides the capability to build and manipulate sequential behavior. A sequence is a list of SystemVerilog boolean expressions in a linear order of increasing time. The boolean expressions must be true at those specific clock ticks for the sequence to be true over time. A boolean expression at a point in time is a simple case of a sequence with time length of one clock cycle. To determine a match of a sequence, the boolean expressions are evaluated at each successive clock tick in an attempt to satisfy the sequence. If all expressions are true, then a match of the sequence occurs.

A sequence expression describes one or more sequences by using *regular expressions.* Such a *regular expression* can concisely specify a set of zero, finitely many, or infinitely many sequences that satisfy the sequence expression.

Sequences and sequence expressions can be composed by concatenation, analogous to a concatenation of lists. The concatenation specifies a delay, using **##**, from the end of the first sequence until the beginning of the second sequence.

The following is the syntax for sequence concatenation.

```
sequence_expr ::=                                                    // from Annex A.2.10
        cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      ...
cycle_delay_range ::=
        ## constant_expression
      | ## [ cycle_delay_const_range_expression ]
cycle_delay_const_range_expression ::=
        constant_expression : constant_expression
      | constant_expression : $
```

Syntax 17-3—Sequence concatenation syntax (excerpt from Annex A)

In this syntax:

— *constant_expression* is computed at compile time and must result in an integer value.

— *constant_expression* can only be 0 or greater.

— The **$** token is used to indicate the end of simulation. For formal verification tools, **$** is used to indicate a finite, unbounded, range.

— When a range is specified with two expressions, the second expression must be greater or equal to the first expression.

The context in which a sequence occurs determines when the sequence is evaluated. The first expression in a sequence is checked at the first occurrence of the clock tick at or after the expression that triggered evaluation of the sequence. Each successive element (if any) in the sequence is checked at the next subsequent occurrence of the clock.

A **##** followed by an optional number or range specifies that the *sequence_expr* should occur later than the current cycle. A number of 1 indicates that the next element should occur a single cycle later than the current cycle. The number 0 specifies that the next expression should occur in parallel with the current clock tick.

The following are examples of delay expressions. `‘true` is a boolean expression that always evaluates to true, and is used for visual clarity. It can be defined as:

```
‘define true 1

##0 a       // means a
##1 a       // means ‘true ##1 a
##2 a       // means ‘true ##1 ‘true ##1 a
##[0:3]a    // means (a) or (‘true ##1 a) or (‘true ##1 ‘true ##1 a) or
               (‘true ##1 ‘true ##1 ‘true ##1 a)
a ##2 b // means a ##1 ‘true ##1 b
```

The sequence:


```
req ##1 gnt ##1 !req
```

specifies that `req` be true on the current clock tick, `gnt` shall be true on the first subsequent tick, and `req` shall be false on the next clock tick after that. The ##1 operator specifies one clock tick separation. A delay of more than one clock tick can be specified, as in:

```
req ##2 gnt
```

This specifies that `req` shall be true on the current clock tick, and `gnt` shall be true on the second subsequent clock tick, as shown in Figure 17-2.

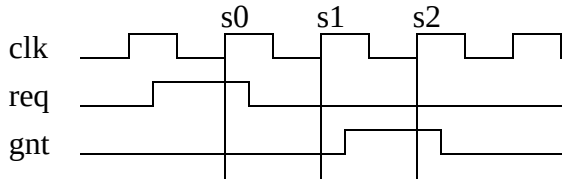


Figure 17-2 — Concatenation of sequences

The following specifies that signal `b` shall be true on the Nth clock tick after signal `a`:

```
a ##N b     // check b on the Nth sample
```

To specify a concatenation of overlapped sequences, where the end point of one sequence coincides with the start of the next sequence, a value of 0 is used, as shown below.

```
a ##1 b ##1 c // first sequence seq1
d ##1 e ##1 f // second sequence seq2
seq1 ##0 seq2 // overlapped concatenation
```

In the above example, `c` is the endpoint of sequence `seq1`, and `d` is the start of sequence `seq2`. When concatenated with 0 clock tick delay, c and d must occur at the same time, resulting in a concatenated sequence equivalent to:

```
a ##1 b ##1 c&&d ##1 e ##1 f
```

It should be noted that no other form of overlapping between the sequences can be expressed using the concatenation operation.

In cases where the delay can be any value in a range, a time window can be specified as follows:

```
req ##[4:32] gnt
```

In the above case, signal `req` must be true at the current clock tick, and signal `gnt` must be true at some clock tick between 4 and 32 after the current clock tick

The time window can extend to a finite, but unbounded, range by using **`$`** as in the example below.

```
req ##[4:$] gnt
```

A sequence can be unconditionally extended by concatenation with `'true`.

```
a ##1 b ##1 c ##3 'true
```

After satisfying signal `c`, the sequence length is extended by 3 clock ticks. Such adjustments in the length of sequences can be required when complex sequences are constructed by combining simpler sequences.

17.6 Declaring sequences

A **sequence** can be declared in

— a module as a *module_or_generate_item*

— an interface as an *interface_or_generate_item*

— a program as a *non_port_program_item*

— a clocking domain as a *clocking_item*

— $root

Sequences are declared using the following syntax.:

```
concurrent_assertion_item_declaration ::=                    // from Annex A.2.10
        ...
      | sequence_declaration
sequence_declaration ::=
        sequence sequence_identifier [ sequence_formal_list ] ;
        { assertion_variable_declaration }
        sequence_spec ;
        endsequence [ : sequence_identifier ]
sequence_formal_list ::=
        ( formal_list_item { , formal_list_item } )
sequence_spec ::=
        multi_clock_sequence
      | sequence_expr
multi_clock_sequence::=
        clocked_sequence { ## clocked_sequence }
clocked_sequence ::=
        clocking_event sequence_expr
sequence_instance ::=
        sequence_identifier [ ( actual_arg_list ) ]
actual_arg_list ::=
        ( actual_arg_expr { , actual_arg_expr } )
      | ( . formal_identifier ( actual_arg_expr ) { , . formal_identifier ( actual_arg_expr ) } )
actual_arg_expr ::=
        event_expression
assertion_variable_declaration ::=
        data_type list_of_variable_identifiers ;
```

Syntax 17-4—Declaring sequence syntax (excerpt from Annex A)

The *clocking_event* specifies the clock for the sequence.

Formal arguments can be optionally specified. A formal argument is untyped, and is used for syntactic replacement of a name or an expression in the sequence.

An actual argument can replace an:

— identifier

— expression

— event control expression

Note that variables used in a sequence that are not formal arguments to the sequence are resolved according to the scoping rules from the scope in which the sequence is declared.

```
sequence s1;
   @(posedge clk) a ##1 b ##1 c;
endsequence
sequence s2;
   @(posedge clk) d ##1 e ##1 f;
endsequence
sequence s3;
   @(negedge clk) g ##1 h ##1 i;
endsequence
```

In this example, sequences `s1` and `s2` are evaluated on each successive posedge of `clk`. The sequence `s3` is evaluated on the negedge of `clk`.

Another example of sequence declaration with arguments is shown below:

```
sequence s20_1(data,en);
   (!frame && (data==data_bus)) ##1 (c_be[0:3] == en);
endsequence
```

Sequence `s20_1` does not specify a clock. In this case, a clock would be inherited from some external source, such as a **`property`** or an **`assert`** statement. A sequence can be referred to by its name. A hierarchical name can be used, consistent with the SystemVerilog naming conventions. A sequence can be referenced in a **`property`**, an **`assert`** statement, or a **`cover`** statement.

To use **`sequence`** as a sub-expression or a part of the expression, simply reference its name. The evaluation of a sequence expression that references a sequence is performed the same way as if the sequence expression contained in the **`sequence`** was a lexical part of the expression, with the formal arguments substituted by the actual ones and the remaining variables that were not arguments substituted from the scope of declaration. An example is shown below:

```
sequence s;
   a ##1 b ##1 c;
endsequence
sequence rule;
   @(posedge sysclk)
   trans ##1 start_trans ##1 s ##1 end_trans;
endsequence
```

Sequence `rule` in the preceding example is equivalent to:

```
sequence rule;
   @(posedge sysclk)
   trans ##1 start_trans ##1 a ##1 b ##1 c ##1 end_trans ;
endsequence
```

Any form of syntactic cyclic dependency of the sequence names is disallowed. The example below illustrates an illegal dependency of `s1` on `s2` and `s2` on `s1`, because it creates a cyclic dependency.

```
sequence s1;
   @(posedge sysclk) (x ##1 s2);
endsequence
sequence s2;
   @(posedge sysclk) (y ##1 s1);
endsequence
```

17.7 Sequence operations

17.7.1 Operator precedence

Operator precedence and associativity is listed in Table 17-1, below. The highest precedence is listed first.

Table 17-1: Operator precedence and associativity

SystemVerilog expression operators	Associativity
, (for assignment)	left
[* [*= [*->	left
and intersect	left
or	left
throughout	left
within	left
##	left

17.7.2 Repetition in sequences

Following is the syntax for sequence repetition.

```
sequence_expr ::=                                                    // from Annex A.2.10
            ...
          | expression { , function_blocking_assignment } [ boolean_abbrev ]
          | ( expression {, function_blocking_assignment } ) [ boolean_abbrev ]
          | sequence_instance [ sequence_abbrev ]
          | ( sequence_expr ) [ sequence_abbrev ]
            ...
boolean_abbrev ::=
            consecutive_repetition
          | non_consecutive_repetition
          | goto_repetition
sequence_abbrev ::= consecutive_repetition
consecutive_repetition ::= [* const_or_range_expression ]
non_consecutive_repetition ::= [*= const_or_range_expression ]
goto_repetition ::= [*-> const_or_range_expression ]
const_or_range_expression ::=
            constant_expression
          | cycle_delay_const_range_expression
cycle_delay_const_range_expression ::=
            constant_expression : constant_expression
          | constant_expression : $
```

Syntax 17-5—Sequence repetition syntax (excerpt from Annex A)

The repetition counts are specified as a range and the minimum and maximum range expressions must be literals or constant expressions.

Three kinds of repetition are provided:

— *consecutive repetition* (`[*`), where a sequence is consecutively repeated with one cycle delay between the repetitions

— *goto repetition* (`[*->`) , where a boolean expression is repeated with one or more cycle delays between the repetitions and the resulting sequence terminates at the last boolean expression

— *non-consecutive repetition* (`[*=`), where a boolean expression is repeated with one or more cycle delays between the repetitions and the resulting sequence can proceed beyond the last boolean expression, but before the occurrence of the boolean expression

To specify the *consecutive repetition* of an expression within a sequence, the expression can simply be repeated, as:

```
a ##1 b ##1 b ##1 b ##1 c
```

Or the number of repetitions can be specified with `[*N]`, as:

```
a ##1 b [*3] ##1 c
```

A *consecutive repetition* specifies that the item or expression must occur a specified number of times. Each repeated item is concatenated (with a delay of 1 clock tick) to the next repeated item. A repeat of N specifies that the sequence must occur N times in succession. For example:

```
a [*3] means a ##1 a ##1 a
```

Using 0 as the repetition number, an empty sequence results, as:

```
a [*0]
```

An empty sequence shall be illegal.

The syntax allows combination of a delay and repetition in the same sequence. The following are both allowed:

```
`true ##3 (a [*3])   // means `true ##1 `true ##1 `true ##1 a ##1 a ##1 a
(`true ##2 a) [*3]   // means (`true ##2 a) ##1 (`true ##2 a) ##1
                     // (`true ##2 a), which in turn means `true ##1 `true ##1
                     // a ##1 `true ##1 `true ##1 a ##1 `true ##1 `true ##1 a
```

A sequence can be repeated as follows:

```
(a ##2 b) [*5]
```

Which is the same as:

```
(a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b)
```

A repetition with a range of maximum and minimum number of times can be expressed with `[* min:max]`. As an example, the following two expressions are equivalent.

```
(a ##2 b)[*1:5]

(a ##2 b)
or (a ##2 b ##1 a ##2 b)
or (a ##2 b ##1 a ##2 b ##1 a ##2 b)
```

```
or (a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b)
or (a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b)
```

The following two expressions are also equivalent.

```
(a[*0:3] ##1 b ##1 c)

(b ##1 c)
or (a ##1 b ##1 c)
or (a ##1 a ##1 b ##1 c)
or (a ##1 a ##1 a ##1 b ##1c)
```

To specify a potentially infinite number of repetitions, the dollar sign (`$`) is used. The repetition:

```
a ##1 b [*1:$] ##1 c
```

means `a` is true on the current sample, then `b` shall be true on every subsequent sample until `c` is true. On the sample in which `c` is true, `b` does not have to be true.

The rules for specifying repeat counts are summarized as:

— Each form of repeat count specifies a minimum and maximum number of occurrences

— expression `[*n:m]`, where `n` is the minimum, `m` is the maximum

— expression `[*n]` is the same as expression `[*n:n]`

— The sequence as a whole cannot be empty

— If `n` is 0, then there must be either a prefix, or a suffix concatenation term (i.e., not the only term in the expression) to the repeated sequence

— The match shall not be empty

The `[*N]` notation indicates *consecutive repetition* of an expression.

The *goto repetition* (non-consecutive exact repetition) specifies the repetition of a boolean expression, such as:

```
a ##1 b [*->min:max] ##1 c
```

This is equivalent to:

```
a ##1 ((!b [*0:$] ##1 b)) [*min:max]) ##1 c
```

Adding the range specification to this allows the construction of useful sequences containing a boolean expression that is true for at most `N` occurrences:

```
a ##1 b[*->1:N] ##1 c //a followed by at most N occurrences of b, followed by c
```

The *non-consecutive repetition* extends the *goto repetition* by extra clock ticks where the boolean expression is not true.

```
a ##1 b [*=min:max] ##1 c
```

This is equivalent to:

```
a ##1 ((!b [*0:$] ##1 b)) [*min:max]) ##1 !b[*0:$] ##1 c
```

The above expression would pass the following sequence, assuming that 3 is within the min:max range.

```
a c c c c b c c b c b d d d c
```


17.7.3 Value change functions

Three functions are provided to detect changes in values between two adjacent clock ticks: `$rose`, `$fell` and `$stable`.

```
$rose ( expression )

$fell ( expression )

$stable ( expression )
```

A value change expression at a clock tick detects the change in value of an expression from the value of that expression at the previous clock tick. The result of a value change expression is true or false and can be used as a boolean expression. At the first clock tick after the assertion is started, the result of these functions are computed by comparing the current value to ‘x’.

`$rose` returns true if the least significant bit of the expression changed to 1. Otherwise, it returns false.

`$fell` returns true if the least significant bit of the expression changed to 0. Otherwise, it returns false.

`$stable` returns true if the value of the expression did not change. Otherwise, it returns false.

Figure 17-3 illustrates two examples of value changes:

— Value change expression `e1` is defined as `$rose(req)`

— Value change expression `e2` is defined as `$fell(ack)`

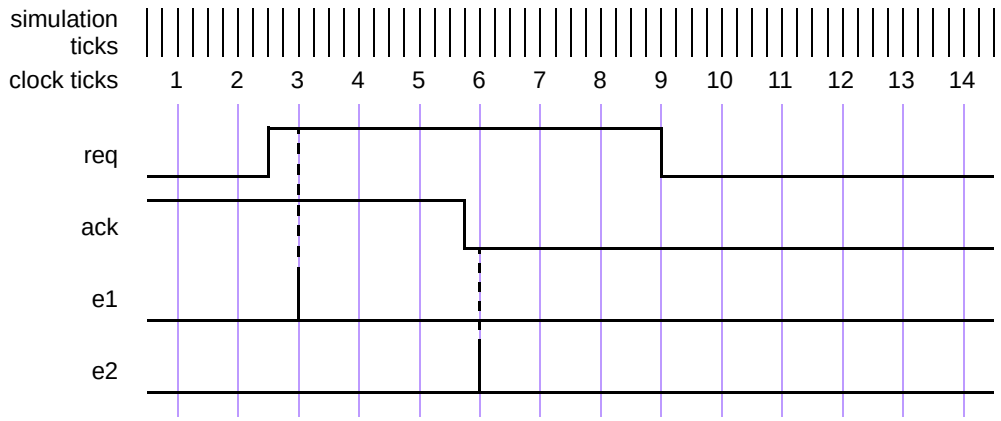


Figure 17-3 — Value change expressions

The clock ticks used for sampling the variables are derived from the clock for the property, which is different from the simulation ticks. Assume, for now, that this clock is defined elsewhere. At clock tick 3, `e1` occurs because the value of `req` at clock tick 2 was low and at clock tick 3, the value is high. Similarly, `e2` occurs at clock tick 6 because the value of `ack` was sampled as high at clock tick 5 and sampled as low at clock tick 6.

17.7.4 AND operation

The binary operator **`and`** is used when both operand expressions are expected to succeed, but the end times of the operand expressions can be different.

```
sequence_expr ::=                                          // from Annex A.2.10
        ...
        | sequence_expr and sequence_expr
```

Syntax 17-6—and operator syntax (excerpt from Annex A)

The two operands of **and** are sequence expressions. The requirement for the success of the **and** operation is that both the operand expressions must succeed. The operand expressions start at the same time. When one of the operand expressions succeeds, it waits for the other to succeed. The end time of the composite expression is the end time of the operand expression that completes last.

When `te1` and `te2` are sequences, then the expression:

```
te1 and te2
```

— Succeeds if `te1` and `te2` succeed.

— The end time is the end time of either `te1` or `te2`, whichever terminates last.

The following example is an expression with the and operator, where the two operands are single sequence evaluations. The operation is illustrated in Figure 17-4.

```
(te1 ##2 te2) and (te3 ##2 te4 ##2 te5)
```

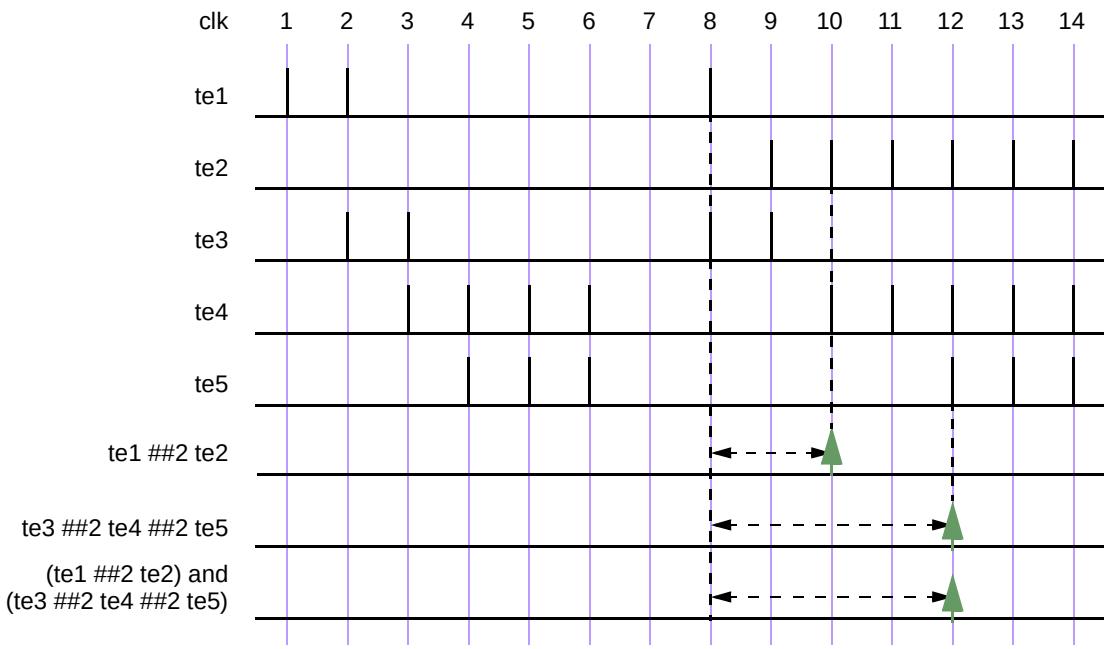


Figure 17-4 — ANDing (and) two sequences

Here, The two operand sequences are `(te1 ##2 te2)` and `(te3 ##2 te4 ##2 te5)`. The first operand sequence requires that first `te1` evaluates to true followed by `te2` two clock ticks later. The second sequence requires that first `te3` evaluates to true followed by `te4` two clock ticks later, followed by `te5` two clock ticks later. Figure 17-4 shows the evaluation attempt at clock tick 8.

This attempt results in a match since both operand sequences match. The end times of matches for the individual sequences are clock ticks 10 and 12. The end time for the entire expression is the last of the two end times,

so a match is recognized for the expression at clock tick 12.

In the following example, an operand sequence is associated with a range of time specification, such as:

```
(te1 ##[1:5] te2) and (te3 ##2 te4 ##2 te5)
```

The first operand sequence consists of an expression with a time range from 1 to 5 and implies that when `te1` evaluates to true, `te2` must follow 1, 2, 3, 4, or 5 clock ticks later. The second operand sequence is the same as in the previous example. To consider all possibilities of a match, the following steps are taken:

1) The first operand sequence starts five sequences of evaluation.

2) The second operand sequence has only one possibility for a match, so only one sequence is started.

3) Figure 17-5 shows the attempt to examine at clock tick 8 when both operand sequences start and succeed. All five sequences for the first operand sequence match, as shown in a time window, at clock ticks 9, 10, 11, 12 and 13 respectively. The second operand sequence matches at clock tick 12.

4) To compute the result for the composite expression, each successful sequence from the first operand sequence is matched against the second operand sequence according to the rules of the and operation to determine the end time for each match.

The result of this computation is five successes, four of them ending at clock tick 12, and the fifth ends at clock tick 13. Figure 17-5 shows the two unique successes at clock ticks 12 and 13.

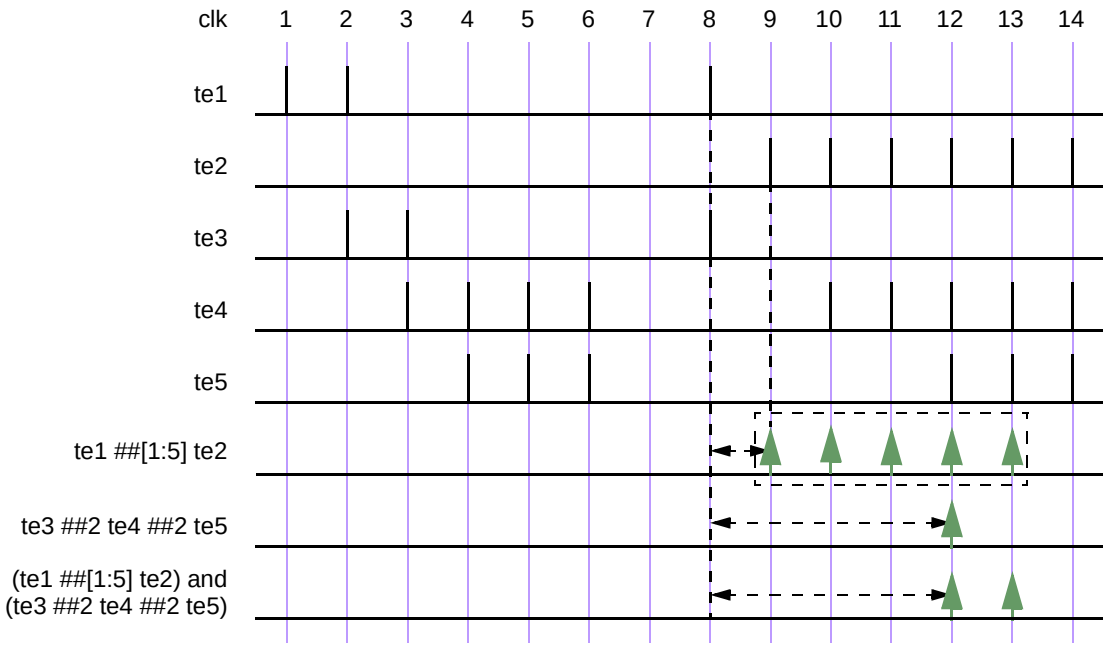


Figure 17-5 — ANDing (and) two sequences, including a time range

If `te1` and `te2` are sampled booleans (not sequences), the expression (`te1` **`and`** `te2`) succeeds if `te1` and `te2` are both evaluated to be true.

An example is illustrated in Figure 17-6, which shows the results for an attempt at every clock tick. The expression matches at clock tick 1, 3, 8, and 14 because both `te1` and `te2` are simultaneously true. At all other clock ticks, the **`and`** operation fails because either `te1` or `te2` is false.

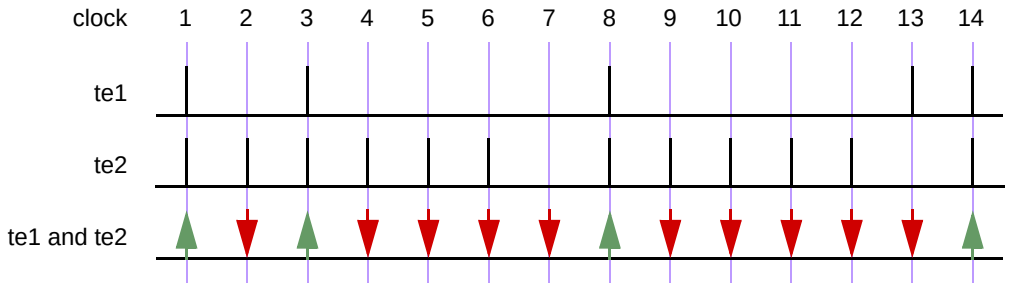


Figure 17-6 — ANDing (and) two boolean expressions

17.7.5 Intersection (AND with length restriction)

The binary operator **`intersect`** is used when both operand expressions are expected to succeed, and the end times of the operand expressions must be the same.

```
sequence_expr ::=                                   // from Annex A.2.10
        ...
        | sequence_expr intersect sequence_expr
```

Syntax 17-7—intersect operator syntax (excerpt from Annex A)

The two operands of **`intersect`** are sequence expressions. The requirements for the success of the **`intersect`** operation are:

— Both the operand expressions must succeed.

— The length of the two operand sequences must be the same.

The additional requirement on the length of the sequences is the basic difference between **`and`** and **`intersect`**.

For each attempted evaluation of *sequence_expr*, there could be multiple matches. When there are multiple matches for each operand sequence expression, the results are computed as follows.

— A match from the first operand is paired with a match from the second operand with the same length.

— If no such pair is found, the result of **`intersect`** is no match.

— If such pairs are found, then the result consists of matched sequences, one for each pair. The end time of each match is determined by the length of the pair.

Figure 17-7 is similar to Figure 17-5, except that **`and`** is replaced by **`intersect`**. Compared with Figure 17-5, there is only a single match in this case.

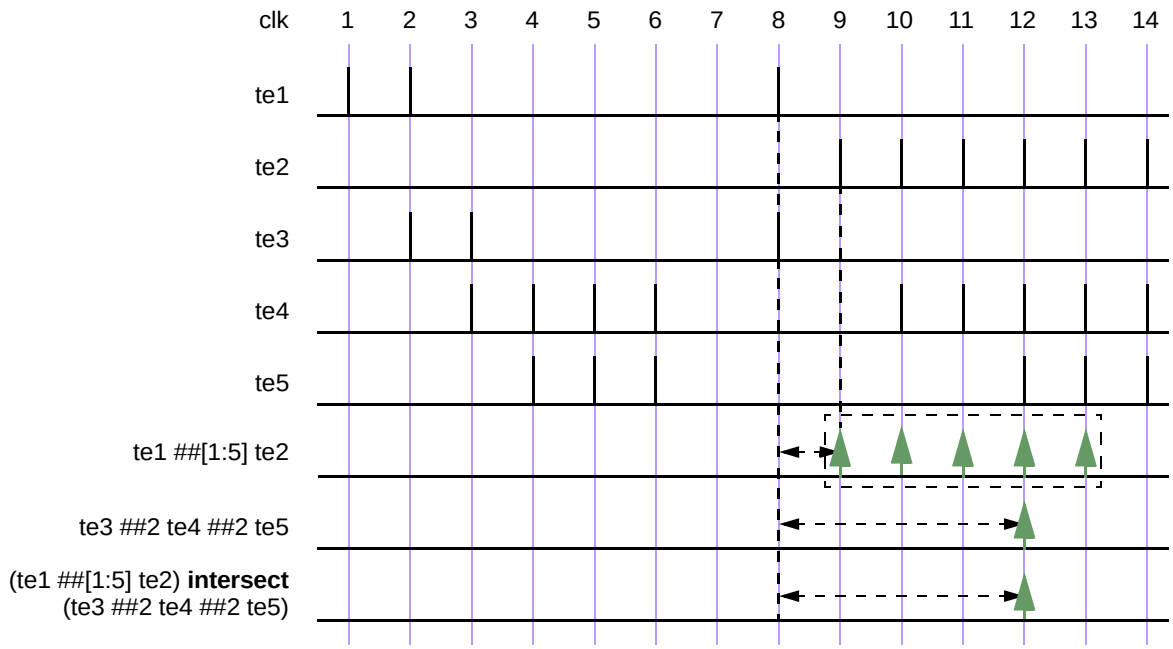


Figure 17-7 — Intersecting two sequences

17.7.6 OR operation

The operator **or** is used when at least one of the two operand sequences is expected to match.

```
sequence_expr ::=                                   // from Annex A.2.10
        ...
        | sequence_expr or sequence_expr
```

Syntax 17-8—or operator syntax (excerpt from Annex A)

The two operands of **or** are sequence expressions.

For the expression:

```
te1 or te2
```

when operands `te1` and `te2` are expressions, the sequence matches whenever at least one of two operands `te1` and `te2` is evaluated to true.

Figure 17-8 illustrates an **or** operation using `te1` and `te2` as simple values. The expression does not match at clock ticks 7 and 13 because `te1` and `te2` are both false at those times. At all other times, the expression matches, as at least one of the two operands is true.

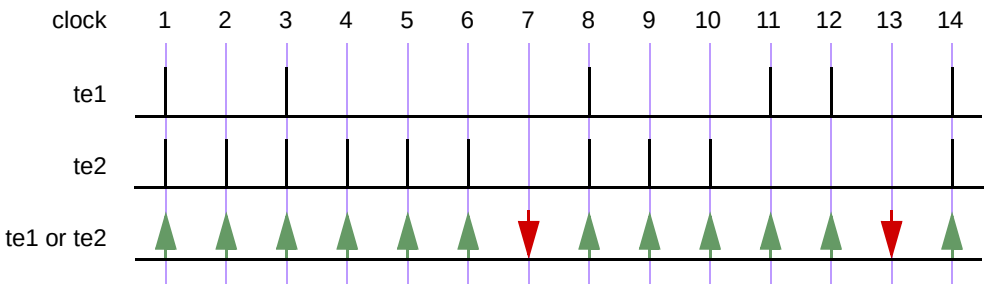


Figure 17-8 — ORing (or) Two Sequences

When `te1` and `te2` are sequences, then the expression

```
te1 or te2
```

matches if at least one of the two operand sequences `te1` and `te2` match. To evaluate this expression, first, the successfully matched sequences of each operand are calculated and assigned to a group. Then, the union of the two groups is computed. The result of the union provides the result of the expression. The end time of a match is the end time of any sequence that matched.

The following example shows an expression with **`or`** operator, where the two operands are sequences. Figure 17-9 illustrates this example.

```
(te1 ##2 te2) or (te3 ##2 te4 ##2 te5)
```

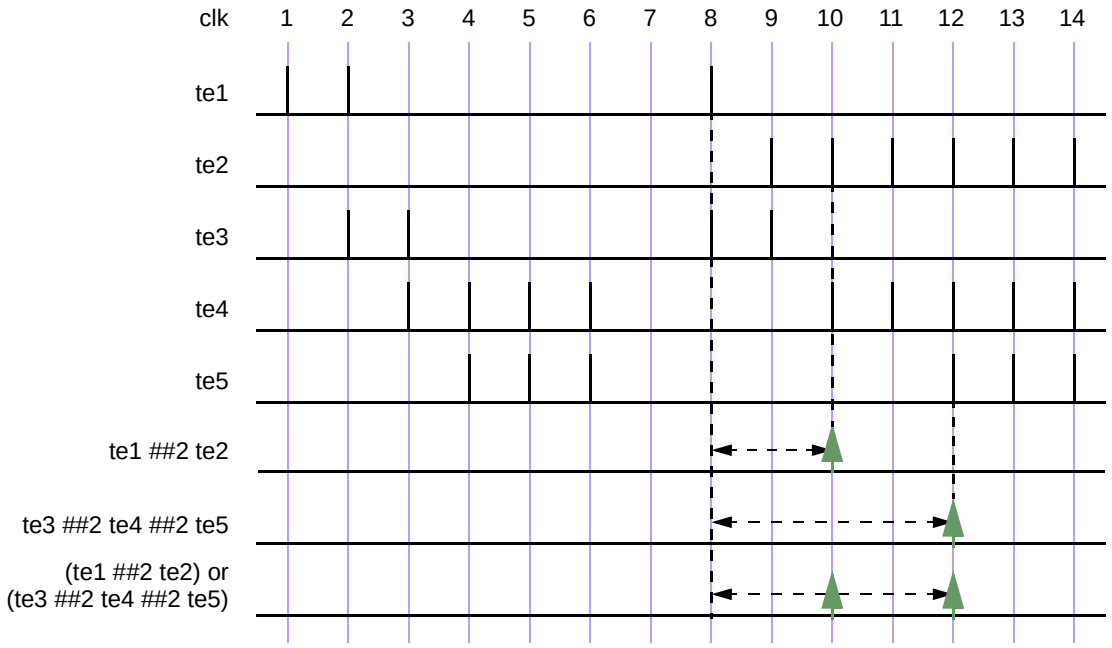


Figure 17-9 — ORing (or) two sequences

Here, the two operand sequences are: `(te1 ##2 te2)` and `(te3 ##2 te4 ##2 te5)`. The first sequence requires that `te1` first evaluates to true, followed by `te2` two clock ticks later. The second sequence requires that `te3` evaluates to true, followed by `te4` two clock ticks later, followed by `te5` two clock ticks later. In Figure 17-9, the evaluation attempt for clock tick 8 is shown. The first sequence matches at clock tick 10 and the second sequence matches at clock tick 12. So, two matches for the expression are recognized.

In the next example, an operand sequence is associated with a time range specification, such as:

```
(te1 ##[1:5] te2) or (te3 ##2 te4 ##2 te5)
```

The first operand sequence consists of an expression with a time range from 1 to 5 and specifies that when `te1` evaluates to true, `te2` must be true 1, 2, 3, 4, or 5 clock ticks later. The sequences from the second operand require that first `te3` must be true followed by `te4` being true two clock ticks later, followed by `te5` being true two clock ticks later. At any clock tick if an operand sequence succeeds, then the composite expressions succeeds. As shown in Figure 17-10, for the attempt at clock tick 8, the first operand sequence matches at clock ticks 9, 10, 11, 12, and 13, while the second operand matches at clock tick 12. The match of the composite expression is computed as a union of the matches of the two operand sequences, which results in matches at clock ticks 9, 10, 11, 12, and 13.

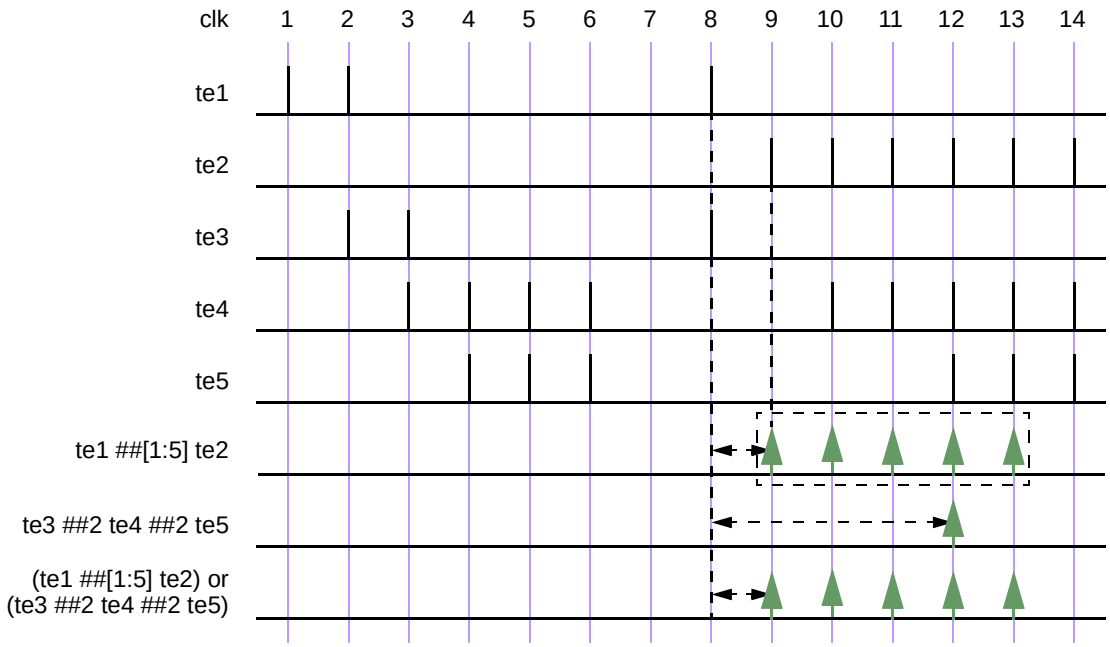


Figure 17-10 — ORing (or) two sequences, including a time range

17.7.7 first_match operation

The **`first_match`** operator matches only the first match of possibly multiple matches for an evaluation attempt of a sequence expression. This allows all subsequent matches to be discarded from consideration. In particular, when the sequence expression is a sub-expression of a larger expression, then applying the **`first_match`** operator has significant effect on the evaluation of the embedding expression.

```
sequence_expr ::=                                   // from Annex A.2.10
        ...
        first_match ( sequence_expr )
```

Syntax 17-9—first_match operator syntax (excerpt from Annex A)

The operand expression can be a sequence expression. *sequence_expr* is evaluated to determine the match for the (**`first_match`** (*`sequence_expr`*)) expression. For a given evaluation attempt, the composite expression matches if *sequence_expr* results in at least one match of a sequence and fails to match if none of the sequences from the expression result in a match. Following the first successful match for the attempt, the **`first_match`** operator stops matching subsequent sequences for sequence_expr. For an attempt, if there are

multiple matches with the same end time as the first detected match, then all those matches are considered as the result of the expression.

The example below shows a variable delay specification.

```
sequence t1;
   te1 ##[2:5]te2;
endsequence
sequence ts1;
   first_match(te1 ##[2:5]te2);
endsequence
```

Each attempt of sequence `t1` can result in matches for up to four following sequences:

```
te1 ##2 te2
te1 ##3 te2
te1 ##4 te2
te1 ##5 te2
```

However, sequence `ts1` can result in a match for only one of the above four sequences. Whichever of the above four sequences matches first becomes the result of sequence **`ts1`**.

As another example:

```
sequence t2;
   (a ##[2:3] b) or (c ##[1:2] d);
endsequence
sequence ts2;
   first_match(t2);
endsequence
```

Each attempt of sequence `t2` can result in matches for up to four following sequences:

```
a ##2 b
a ##3 b
c ##1 d
c ##2 d
```

Sequence `ts2` results in the earliest match. In this case, it is possible to have two matches ending at the same time.

```
a ##2 b
c ##2 d
```

In this case, **`first_match`** results in two sequences.

17.7.8 Conditions over sequences

Sequences often occur under the assumptions of some conditions for correct behavior. A logical condition must hold true, for instance, while processing a transaction. Also, occurrence of certain values is prohibited while processing a transaction. Such situations can be expressed directly using the following construct:

```
sequence_expr ::=                                   // from Annex A.2.10
        ...
        | expression throughout sequence_expr
```

Syntax 17-10—throughout construct syntax (excerpt from Annex A)

expression must evaluate true at every clock tick during the evaluation of *sequence_expr*. If an evaluation of *sequence_expr* starts at time `t1` and ends with a match at time `t2`, then for *sequence_expr* to match, expression must hold true from time `t1` to `t2`.

The **`throughout`** construct is an abbreviation for writing:

```
(expression) [*0:$] intersect sequence_expr
```

In the following example, illustrated in Figure 17-11, if a constraint were placed on the expression as shown below, then the checker `burst_rule1` would fail at clock tick 9.

```
sequence burst_rule1;
   @(posedge mclk)
      $fell(burst_mode) ##0
      (!burst_mode) throughout (##2 ((trdy==0)&&(irdy==0)) [*7]);
endsequence
```

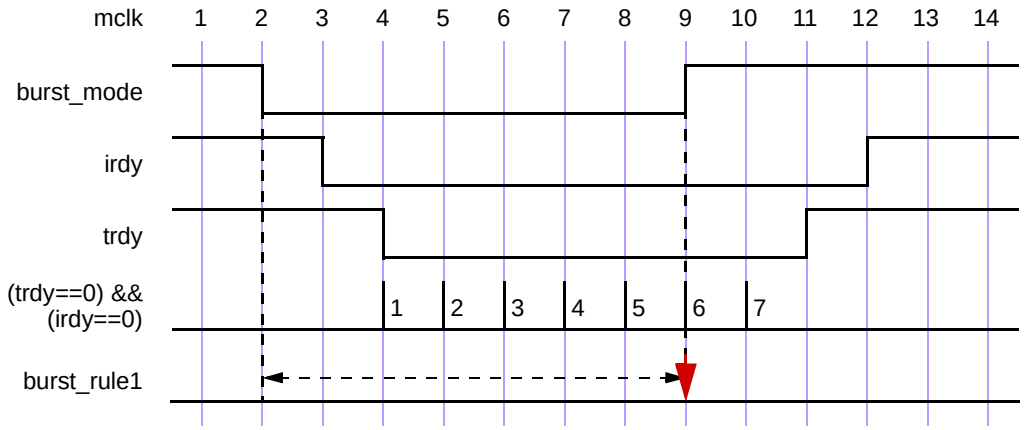


Figure 17-11 — Match with throughout restriction fails

In the above expression, the value of signal `burst_mode` is required to be low during the sequence (from clock tick 2 to 10) and is checked at every clock tick during that period. At clock ticks from 2 to 8, signal `burst_mode` remains low and matches the expression at those clock ticks. At clock tick 9, signal `burst_mode` becomes high, thereby failing to match the expression for `burst_rule1`.

If signal `burst_mode` were to be maintained low until clock tick 10, the expression would result in a match as shown in Figure 17-12.

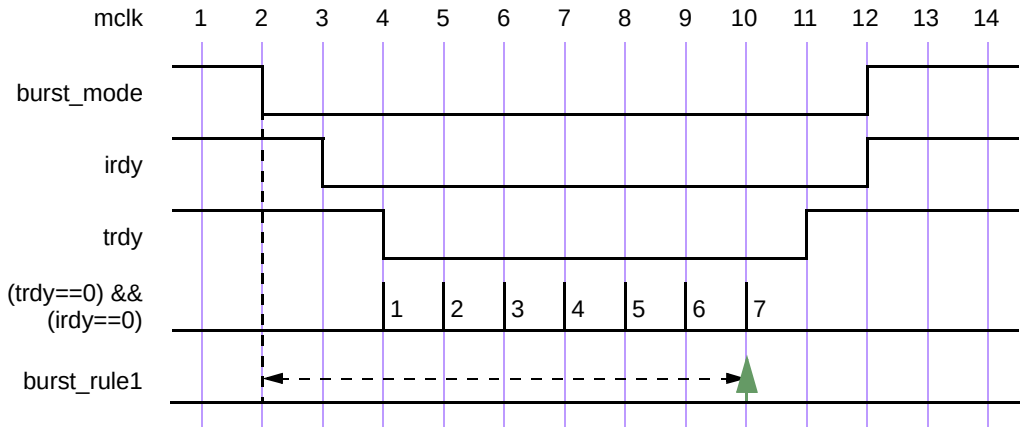


Figure 17-12 — Match with throughout restriction succeeds

17.7.9 Sequence occurrence within another sequence

The containment of a sequence expression within another sequence is expressed as follows:

```
sequence_expr ::=                                          // from Annex A.2.10
        ...
        | sequence_expr within sequence_expr
```

Syntax 17-11—within construct syntax (excerpt from Annex A)

The within construct:

```
sequence_expr1 within sequence_expr2
```

is an abbreviation for writing:

```
(1[*0:$] ##1 sequence_expr1 ##1 1[*0:$]) intersect sequence_expr2
```

The sequence `sequence_expr1` must occur at least once entirely within the sequence `sequence_expr2`. That is, `sequence_expr1` must satisfy the following:

— The start point of `sequence_expr1` must be between the start point and the end point (start and end point being inclusive) of `sequence_expr2`.

— The end point of `sequence_expr1` must be between the start point and the end point (start and end point being inclusive) of `sequence_expr2`.

For example, the sequence expression

```
trdy[*7] within (($fell irdy) ##1 irdy[*8])
```

matches on the trace shown in Figure 17-12.

17.7.10 Detecting and using endpoint of a sequence

There are two ways in which a complex sequence can be decomposed into simpler sub-expressions.

One is to reference the name of a sequence, thereby causing it to be started at the point where it is referenced, as shown below:

```
sequence s;
   a ##1 b ##1 c;
endsequence
sequence rule;
   @(posedge sysclk)
      trans ##1 start_trans ##1 s ##1 end_trans);
endsequence
```

Sequence `s` is evaluated one cycle after the occurrence of `start_trans` in the sequence `rule`.

Another way to use the sequence expression is to detect its end point in another sequence. The end point of a sequence is reached whenever there is a match on its expression. The occurrence of the end point can be tested in any sequence expression by using the method `ended`.

The syntax of the ended method is:


```
sequence_identifier.ended
```

`ended` is a method on a sequence. The result of its operation is true or false. When method `ended` is applied in an expression, it tests whether sequence *seq_name* has reached the end point at that particular point in time. The result of `ended` does not depend upon the starting point of *seq_name*. An example is shown below:

```
sequence e1;
   @(posedge sysclk) $rose(ready) ##1 proc1 ##1 proc2 ;
endsequence
sequence rule;
   @(posedge sysclk) reset ##1 inst ##1 e1.ended ##1 branch_back;
endsequence
```

In this example, sequence expression `e1` must end successfully one clock tick after `inst`. If the method `ended` is replaced with sequence e1, `e1` must start one clock tick after `inst`. Notice that method `ended` only tests for the end point of `e1`, and has no bearing on the starting point of `e1`.

`ended` can be used directly on sequences that do not have formal arguments. To use `ended` on a sequence with arguments, first define a sequence without formal arguments that instantiates the sequence with actual arguments. For example,

```
sequence e2(a,b,c);
   @(posedge sysclk) $rose(a) ##1 b ##1 c;
endsequence
sequence e2_instantiated;
   e2(ready,proc1,proc2);
endsequence
sequence rule2;
   @(posedge sysclk) reset ##1 inst ##1 e2_instantiated.ended ##1 branch_back;
endsequence
```

17.7.11 Implication

The implication construct allows a user to monitor sequences based on satisfying some criteria. Most common uses are to attach a precondition to a sequence, where the evaluation of the sequence is based on the success of a condition.

```
property_expr ::=                                          // from Annex A.2.10
        ...
        | sequence_expr |-> [ not ] sequence_expr
        | sequence_expr |=> [ not ] sequence_expr
multi_clock_property_expr ::=
          ...
        | multi_clock_sequence |=> [ not ] multi_clock_sequence
```

Syntax 17-12—implication syntax (excerpt from Annex A)

This clause is used to precondition monitoring of a sequence expression and is allowed at the property level. The result of the implication is either true or false. The left-hand side operand *sequence_expr* is called *antecedent*, while the right-hand side operand *sequence_expr* is called *consequent*.

The following points should be noted for `|->` implication:

— *antecedent sequence_expr* can result in multiple successful sequences.

— If there is no match of the *antecedent sequence_expr*, implication succeeds vacuously by returning true.

— For each successful match of *antecedent sequence_expr*, *consequent sequence_expr* is separately evaluated, beginning at the end point of the match. That is, the end point of matching sequence from *antecedent sequence_expr* overlaps with start point of the *consequent sequence_expr.*

— All matches of *antecedent sequence_expr* must satisfy *consequent sequence_expr.* The satisfaction of the *consequent sequence_expr* means that there is at least one match of the *sequence_expr.*

— Nesting of implication is not allowed.

Two forms of implication are provided: overlapped using operator `|->`, and non-overlapped using operator `|=>`. For overlapped implication, if there is a match for the *antecedent sequence_expr*, then the first element of the *consequent sequence_expr* is evaluated on the same clock tick. For non-overlapped implication, the first element of the *consequent sequence_expr* is evaluated on the next clock tick. Therefore:

```
sequence_expr |=> [not] sequence_expr
```

is equivalent to:

```
sequence_expr ##1 'true |-> [not] sequence_expr
```

If **`not`** is used on the consequent, the result of *consequent sequence_expr* is reversed.

The use of implication when multi-clock sequences are involved is explained in Section 17.11.

The following example illustrates a bus operation for data transfer from a master to a target device. When the bus enters a data transfer phase, multiple data phases can occur to transfer a block of data. During the data transfer phase, a data phase completes on any rising clock edge on which `irdy` is asserted and either `trdy` or `stop` is asserted. Note that an asserted signal here implies a value of low. The end of a data phase can be expressed as:

```
property data_end;
   @(posedge mclk)
   data_phase |-> ((irdy==0) && ($fell(trdy) || $fell(stop))) ;
endproperty
```

Each time a data phase is true, a match for `data_phase` is recognized. The attempt at clock tick 6 is illustrated in Figure 17-13. The values shown for the signals are the sampled values with respect to the clock. At clock tick 6, `data_end` is true because `stop` gets asserted while `irdy` is asserted.

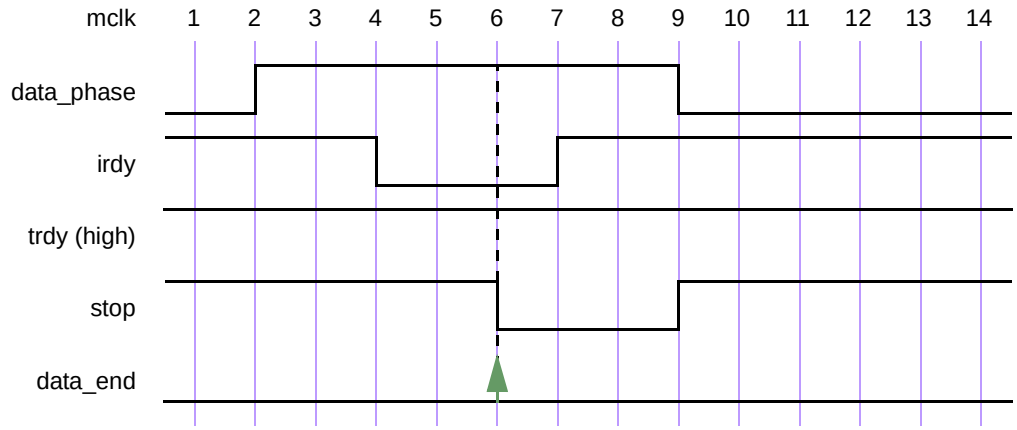


Figure 17-13 — Conditional sequence matching

In another example, `data_end_exp` is used to ensure that `frame` is de-asserted (value high) within 2 clock ticks after `data_end_exp` occurs. Further, it is also required that `irdy` is de-asserted (value high) one clock tick after `frame` is de-asserted.

A property written to express this condition is shown below.

```
`define data_end_exp (data_phase && ((irdy==0)&&($fell(trdy)||$fell(stop))))
property data_end_rule1;
   @(posedge mclk)
   `data_end_exp |-> ##[1:2] $rose(frame) ##1 $rose(irdy);
endproperty
```

property `data_end_rule1` first evaluates `data_end_exp` at every clock tick to test if its value is true. If the value is false, then that particular attempt to evaluate `data_end_rule1` is considered true. Otherwise, the following sequence expression is evaluated. The sequence expression:

```
##[1:2] $rose(frame) ##1 $rose(irdy)
```

specifies looking for the rising edge of `frame` within two clock ticks in the future. After `frame` toggles high, `irdy` must also toggle high after one clock tick. This is illustrated in Figure 17-14. `` `data_end_exp `` is acknowledged at clock tick 6. Next, `frame` toggles high at clock tick 7. Since this falls within the timing constraint imposed by `[1:2]`, it satisfies the sequence and continues to monitor further. At clock tick 8, `irdy` is evaluated. Signal `irdy` transitions to high at clock tick 8, satisfying the sequence specification completely for the attempt that began at clock tick 6.

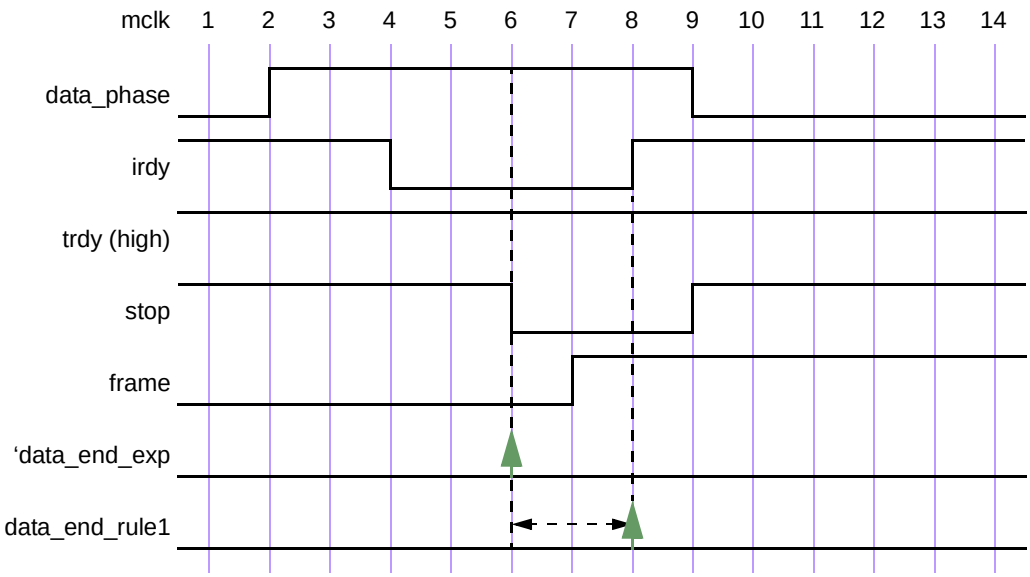


Figure 17-14 — Conditional sequences

Generally, assertions are associated with preconditions so that the checking is performed only under certain specified conditions. As seen from the previous example, the `|->` operator provides this capability to specify preconditions with sequences that must be satisfied before continuing to match those sequences. The next example modifies the preceding example to see the effect on the results of the assertion by removing the precondition for the sequence. This is shown below, and illustrated in Figure 17-15.

```
property data_end_rule2;
   @(posedge mclk) ##[1:2] $rose(frame) ##1 $rose(irdy);
endproperty
```

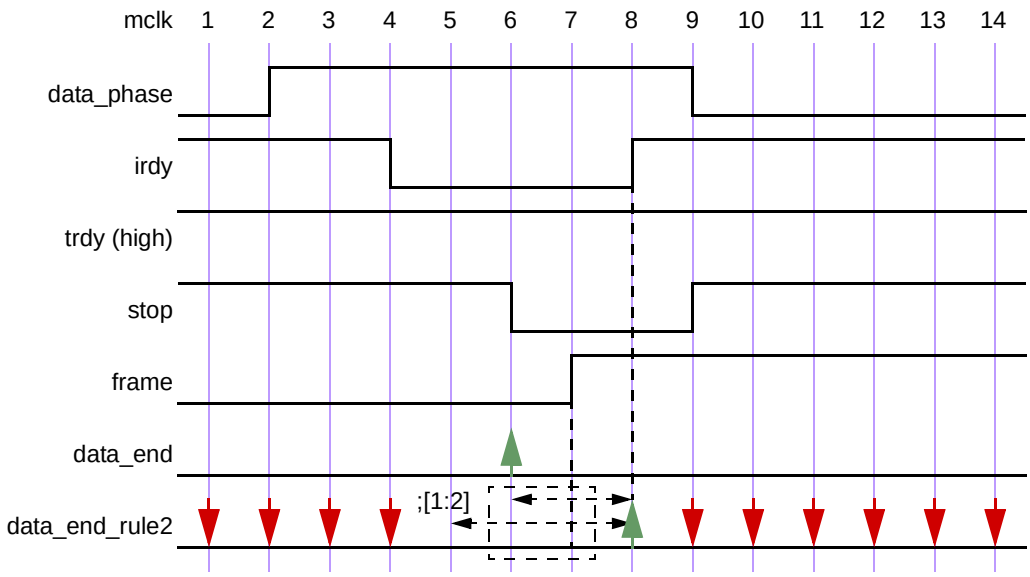


Figure 17-15 — Results without the condition

The property is evaluated at every clock tick. For the evaluation at clock tick 1, the rising edge of signal `frame` does not occur at clock tick 1 or 2, so the property fails at clock tick 1. Similarly, there is a failure at clock ticks 2, 3, and 4. For attempts starting at clock ticks 5 and 6, the rising edge of signal `frame` at clock tick 7 allows checking further. At clock tick 8, the sequences complete according to the specification, resulting in a match for attempts starting at 5 and 6. All later attempts to match the sequence fail because `$rose(frame)` does not occur again. That also means that there is no match at 5, 6, and 7.

Figure 17-15 shows that removing the precondition of checking `` `data_end_exp `` from the assertion causes failures that are not relevant to the verification objective. It is important from the validation standpoint to determine these preconditions and use them to filter out inappropriate or extraneous situations.

An example of implication where the antecedent is a sequence expression follows:

```
(a ##1 b ##1 c) |-> (d ##1 e)
```

If the sequence `(a ##1 b ##1 c)` matches, then the sequence `(d ##1 e)` must also match. On the other hand, if the sequence `(a ##1 b ##1 c)` does not match, then the result is true.

In the next example, all matches of `(a ##[1:3] b ##1 c)` must match `(d ##1 e)`. If there are no matches of `(a ##[1:3] b ##1 c)`, then there is a vacuous success for the property.

Another example of implication is:

```
property p16;
   (write_en & data_valid) ##0
   (write_en && (retire_address[0:4]==addr)) [*2] |->
   ##[3:8] write_en && !data_valid &&(write_address[0:4]==addr);
endproperty
```

Multi-clock sequence implication is explained in Section 17.11.

17.8 Manipulating data in a sequence

The use of static SystemVerilog variables implies that only one copy exists. Therefore, if data values need to

be checked in pipelined designs, then for each data entering the pipeline, a separate variable can be used to store the predicted output of the pipeline for later comparison when the result actually exits the pipe. This storage can be built by using an array of variables arranged in a shift register to mimic the data propagating through a pipeline. However, in more complex situations where the latency of the pipe is variable and out of order, this construction could become very complex and error prone. In other words, variables are needed that are local to and are used within a particular transaction check that can span an arbitrary interval of time and can overlap with other transaction checks. Such a variable must thus be dynamically created when needed within an instance of a sequence and removed when the end of the sequence is reached.

The dynamic creation of a variable and its assignment is achieved by using the local variable declaration in a sequence or property definition and making an assignment in the sequence.

```
sequence_expr ::=                                                    // from Annex A.2.10
        ...
        | ( expression {, function_blocking_assignment } ) [ boolean_abbrev ]
        | expression { , function_blocking_assignment } [ boolean_abbrev ]
```

Syntax 17-13—variable assignment syntax (excerpt from Annex A)

The type of variable is explicitly specified. The variable can be assigned anywhere in the sequence and reassigned later in the sequence. For every attempt, a new copy of the variable is created for the sequence. The variable value can be tested like any other SystemVerilog variable.

Hierarchical references to a local variable are not allowed.

As an example the local variable usage, assume a pipeline that has a fixed latency of 5 clock cycles. The data enters the pipe on `pipe_in` when `valid_in` is true, and the value computed by the pipeline appears 5 clock cycles later on the signal `pipe_out1`. The data as transformed by the pipe is predicted by a function that increments the data. The following sequence expression verifies this behavior:

```
property e;
   int x;
   (valid_in,(x = pipe_in)) |-> ##5 (pipe_out1 == (x+1));
endproperty
```

Property `e` is evaluated as :

1) When `valid_in` is true, `x` is assigned to pipe_in. Property e is true if five cycles later, `x` is equal to `(x+1)`. Property e is false if `pipe_out1` is not equal to `(x+1)`.

2) When valid_in is false, property e evaluates to true.

Variables can be used in sequences or properties.

```
sequence data_check;
   int x;
   a ##1 !a, x = data_in ##1 !b*[0:$] ##1 b && (data_out == x);
endsequence
property data_check_p
   int x;
   a ##1 !a, x = data_in |=> !b*[0:$] ##1 b && (data_out == x);
endproperty
```

Local variables can be written on repeated sequences and accomplish accumulation of values.

```
sequence rep_v;
   int x;
```

```
    `true,x = 0 ##0
    (!a [* 0:$] ##1 a, x = x+data)[*4] ##1 b ##1 c && (data_out == x);
endsequence
```

The local variables declared in one sequence are not visible in the sequence where it gets instantiated. An example below illustrates an illegal access to local variable `v1` of sequence `sub_seq1` in sequence `seq1`.

```
sequence sub_seq1;
   int v1;
   a ##1 !a, v1 = data_in ##1 !b*[0:$] ##1 b && (data_out == v1);
endsequence
sequence seq1;
   c ##1 sub_seq1 ##1 (do1 == v1); // error since v1 is not visible
endsequence
```

To access a local variable of a sub-sequence, a local variable must be declared and passed to the instantiated sub-sequence through an argument. An example below illustrates this usage.

```
sequence sub_seq2(lv);
   a ##1 !a, lv = data_in ##1 !b*[0:$] ##1 b && (data_out == lv);
endsequence
sequence seq2;
   int v1;
   c ##1 sub_seq2(v1) ##1 (do1 == v1); // v1 is now bound to lv
endsequence
```

Note that when a local variable is a formal argument of a sequence definition, it is illegal to declare the variable, as shown below.

```
sequence sub_seq3(lv);
   int lv; // illegal since lv is a formal argument
   a ##1 !a, lv = data_in ##1 !b*[0:$] ##1 b && (data_out == lv);
endsequence
```

There are special considerations on using local variables in parallel branches using operators **`or`**, **`and`**, and **`intersect`**.

1) Variables assigned on parallel threads cannot be accessed in sibling threads. For example:

```
sequence s4;
    int x;
    (a ##1 b, (x = data) ##1 c) or (d ##1 (e==x)); // illegal
endsequence
```

2) In the case of **`or`**, it is the intersection of the variables (names) that pass on past **`or`** operations. More precisely, a local variable passes the **`or`** if, and only if, it passes through both branches of **`or`** operations.

3) All succeeding threads out of **`or`** branches continue as separate threads, carrying with them their own latest samplings of the local variables. These threads do not have to have consistent valuations for the local variables. For example:

```
sequence s5;
   int x,y;
   ((a ##1 b, x = data, y = data1 ##1 c)
      or (d ##1 `true, x = data ##0 (e==x))) ##1 (y==data2);
   // illegal since y is not in the intersection
endsequence
sequence s6;
   int x,y;
```



```
   ((a ##1 b, x = data, y = data1 ##1 c)
      or (d ##1 `true, x = data ##0 (e==x))) ##1 (x==data2);
   // legal since x is in the intersection
endsequence
```

4) In the case of **and** and **intersect**, the symmetric difference of the local variables that are sampled in the two joining threads passes on past the join. More precisely, a local variable that passes through at least one branch of the join shall be passed on past the join unless it is blocked. A local variable is blocked from passing on past the join if either:

   a) The local variable is sampled in and passes through each branch of the join. Or,

   b) The local variable is blocked from passing through at least one of the branches of the join..

   The value passed on is the latest sampled value. The two joining threads are merged into one thread at the join.

```
sequence s7;
   int x,y;
   ((a ##1 b, x = data, y = data1 ##1 c)
      and (d ##1 `true, x = data ##0 (e==x))) ##1 (x==data2);
   // illegal since x is common to both threads
endsequence
sequence s8;
   int x,y;
   (a ##1 b, x = data, y = data1 ##1 c)
      and (d ##1 `true, x = data ##0 (e==x))) ##1 (y==data2);
   // legal since y is in the difference
endsequence
```

5) The intersection and difference of the sets of names should be computed statically at compile time.

17.9 System functions

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is “one-hot”. The following system functions are included to facilitate such common assertion functionality:

— `$onehot (<expression>)` returns true if only one bit of the expression is high.

— `$onehot0(<expression>)` returns true if at most one bit of the expression is high.

— `$inset (<expression>, <expression> {, <expression> } )` returns true if the first expression is equal to at least one of the subsequent expression arguments.

— `$insetz (<expression>,<expression> {, <expression> } )` returns true if the first expression is equal to at least other expression argument. The comparison is performed using casez semantics, so ‘z’ or ‘?’ bits are treated as don’t-cares.

— `$isunknown (<expression>)` returns true if any bit of the expression is ‘x’. This is equivalent to `^<expression> === 'bx`.

All of the above system functions have a return type of bit. A return value of `1'b1` indicates true, and a return value of `1'b0` indicates false.

In addition to accessing values of signals at the time of evaluation of a boolean expression, the past values can be accessed with the `$past` function.

```
$past ( expression [ , number_of_ticks] )
```

The optional argument *number_of_ticks* specifies the number of clock ticks in the past. If *number_of_ticks* is not specified, then it defaults to 1. `$past` returns the sampled value of the expression that was present *number_of_ticks* prior to the time of evaluation of `$past`.

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.

Another useful function provided for the boolean expression is `$countones`, to count the number of 1s in a bit vector expression.

```
$countones ( expression)
```

An `x` and `z` value of a bit is not counted towards the number of ones.

17.10 The property definition

A property defines a behavior of the design. A property can be used for verification as an assumption, a checker, or a coverage specification. In order to use the behavior for verification, an **`assert`** or **`cover`** statement must be used. A property declaration by itself does not produce any result.

A property can be declared in

- a module as a *module_or_generate_item*
- an interface as an *interface_or_generate_item*
- a program as a *non_port_program_item*
- a clocking domain as a *clocking_item*
- $root

To declare a property, the **`property`** construct is used as shown below:


```
concurrent_assertion_item_declaration ::=                    // from Annex A.2.10
        property_declaration
property_declaration ::=
        property property_identifier [ property_formal_list ] ;
        { assertion_variable_declaration }
        property_spec ;
        endproperty [ : property_identifier ]
property_formal_list ::=
        ( formal_list_item { , formal_list_item } )
property_spec ::=
        [clocking_event ] [ disable iff ] ( expression ) [ not ] property_expr
      | [ disable iff ( expression ) ] not multi_clock_property_expr
property_expr ::=
        sequence_expr
      | sequence_expr |-> [ not ] sequence_expr
      | sequence_expr |=> [ not ] sequence_expr
multi_clock_property_expr ::=
        multi_clock_sequence
      | multi_clock_sequence |=> [ not ] multi_clock_sequence
assertion_variable_declaration ::=
        data_type list_of_variable_identifiers ;
property_instance::=                                         // from Annex A.6.10
        property_identifier [ ( actual_arg_list ) ]
```

Syntax 17-14—property construct syntax (excerpt from Annex A)

A **property** is declared with optional formal arguments, as in a sequence declaration. When a property is instantiated, actual arguments can be passed to the property. The property gets expanded with the actual arguments by replacing the formal arguments with the actual arguments. The semantic checks are performed to ensure that the expanded property with the actual arguments is legal.

The result of **property** evaluation is either true or false. There are two kinds of property: sequence, and implication. If the property is just a sequence, the result of a sequence for every evaluation attempt is true or false. This is accomplished by implicitly transforming *sequence_expr* to **first_match**(sequence_expr). That is, as soon as a match of *sequence_expr* is determined, the result is considered to be true, and no other matches are required for that evaluation attempt. However, if the property is an implication, then the semantics of implication determine whether the property is true or false.

The **disable iff** clause allows asynchronous resets to be specified. For a particular attempt, if the expression of the **disable iff** becomes true at any time during the evaluation of the attempt, then the attempt for the property is considered to be a success. Other attempts are not affected by the evaluation of the expression for an attempt.

The **not** clause states that the *property_expr* associated with the property must never evaluate to true. Effectively, it negates *property_expr*. For each attempt, *property_expr* results in either true or false, based on whether there is a match for the sequence. The **not** clause reverses the result of *property_expr*. It should be noted that there is no complementation or any form of negation for the sequence in *property_expr*.

This allows for the following examples:

```
property rule1;
   @(posedge clk) a |-> b ##1 c ##1 d;
endproperty
```

```
property rule2;
   @(clkev) disable iff (foo) not a |-> b ##1 c ##1 d;
endproperty
```

Property `rule2` negates the result of the implication (`a |-> b ##1 c ##1 d`) for every attempt. `clkev` specifies the clock for the property.

A property can optionally specify an event control for the clock. The clock derivation and resolution rules are described in Section 17.13.

A property can be referenced by its name. A hierarchical name can be used, consistent with the SystemVerilog naming conventions. Like sequence declarations, variables used within a property that are not formal arguments to the property are resolved hierarchically from the scope in which the property is declared.

Properties that use more than one clock are described in Section 17.11

17.11 Multiple clock support

Multiple clock sequences and properties can be specified using the following syntax.

```
sequence_spec ::=                                          // from Annex A.2.10
        multi_clock_sequence
      | sequence_expr
multi_clock_sequence::=
        clocked_sequence { ## clocked_sequence }
clocked_sequence ::=
        clocking_event sequence_expr
multi_clock_property_expr ::=
        multi_clock_sequence
      | multi_clock_sequence |=> [ not ] multi_clock_sequence
```

Syntax 17-15—Multiple clock syntax (excerpt from Annex A)

Two cases are allowed:

1) Concatenation of two sequences, where each sequence can have a different clock

2) The antecedent of an implication on one clock, while the consequent is on another clock

The multi-clock concatenation operator `##` synchronizes between the two clocks.

```
@(posedge clk0) sig0 ## @(posedge clk1) sig1
```

When signal `sig0` matches at clock `clk`, `##` moves the time to the nearest clock tick of `clk1` after the match. At the first clock tick of `clk1`, it matches `sig1`. If the two clocks, `clk0` and `clk1`, are identical, then the above sequence is equivalent to:

```
@(posedge clk0) sig0 ##1 sig1
```

For two sequences, such as

```
@(posedge clk0) s0 ## @(posedge clk1) s1
```

For every match of `s0` at clock `clk0`, `##` moves the time to the first clock tick of `clk1`. From that first tick of `clk1`, `s1` is matched.

Multi-clock implication is only allowed with the non-overlapping implication. The semantics are similar to the sequence concatenation with ##. Whenever there is a match of the antecedent sequence, time is advanced to the nearest clock tick of the clock of the consequent sequence. The consequent is then evaluated for satisfaction.

The following are examples of multiple-clock specifications:

```
sequence s1;
   @(posedge clk1) a ##1 b; // single clock sequence
endsequence
sequence s2;
   @(posedge clk2) c ##1 d; // single clock sequence
endsequence
```

1) multiple-clock sequence

```
sequence mult_s;
   @(posedge clk) a ## @(posedge clk1) s1 ## @(posedge clk2) s2;
endsequence
```

2) property with a multiple-clock sequence

```
property mult_p1;
   @(posedge clk) a ## @(posedge clk1) s1 ## @(posedge clk2) s2;
endproperty
```

3) property with a named multiple-clock sequence

```
property mult_p2;
   mult_s;
endproperty
```

4) property with multiple-clock implication

```
property mult_p3;
   @(posedge clk) a ## @(posedge clk1) s1 |=> @(posedge clk2) s2;
endproperty
```

5) property with named sequences at different clocks. In this case, if `s1` contains a clock, then it must be identical to `(posedge clk1)`. Similarly, if `s2` contains a clock, it must be identical to `(posedge clk2)`.

```
property mult_p5
   @(posedge clk1) s1 |=> @(posedge clk2) s2;
endproperty
```

6) property with implication, where antecedent and consequent are named multi-clocked sequences

```
property mult_p6;
   mult_s |=> mult_s;
endproperty
```

17.11.1 Detecting and using endpoint of a sequence in multi-clock context

To detect the end point of a sequence when the clock of the source sequence is different than the desalination sequence, method `matched` on the source sequence is used. The end point of a sequence is reached whenever there is a match on its expression. The occurrence of the end point can be tested in any sequence expression by using the method `ended` when the clocks of the source and destination sequences are the same, while method `matched` is used when the clocks are different.

The syntax of the `matched` method is:

```
sequence_identifier.matched
```

`matched` is a method on a sequence which return true or false. Unlike `ended`, `matched` uses synchronization between the two clocks, by storing the result of the source sequence match until the arrival of the first destination clock tick after the match. When method `matched` is applied, it tests whether the source sequence has reached the end point at that particular point in time. The result of `matched` does not depend upon the starting point of the source sequence.

Like `ended`, `matched` can be used directly on sequences that do not have formal arguments.

An example is shown below:

```
sequence e1;
   @(posedge clk) $rose(ready) ##1 proc1 ##1 proc2 ;
endsequence
sequence e2;
   @(posedge sysclk) reset ##1 inst ##1 e1.matched [*->1] ##1 branch_back;
endsequence
```

In this example, source sequence `e1` is evaluated at clock `clk`, while the destination sequence `e2` is evaluated at clock `sysclk`. In `e2`, the end point of `e1` is tested to occur sometime after the occurrence of `inst`. Notice that method matched only tests for the end point of `e1` and has no bearing on the starting point of `e1`.

17.12 Concurrent assertions

A property on its own is never evaluated for checking an expression. It must be used within a verification statement for this to occur. A verification statement states the verification function to be performed on the property. The statement can be one of the following:

— **`assert`** to specify the property as a checker to ensure that the property holds for the design

— **`cover`** to monitor the property evaluation for coverage

A concurrent assertion statement can be specified in:

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

— a module as a *module_or_generate_item*

— an interface as an *interface_or_generate_item*

— a program as a *non_port_program_item*

— $root


```
procedural_assertion_item ::=                                   // from Annex A.6.10
        assert_property_statement
      | cover_property_statement
concurrent_assertion_item ::=                                   // from Annex A.2.10
        concurrent_assert_statement
      | concurrent_cover_statement
concurrent_assert_statement ::=
        [block_identifier:] assert_property_statement
concurrent_cover_statement ::=
        [block_identifier:] cover_property_statement
assert_property_statement::=
        assert property ( property_spec ) action_block
      | assert property ( property_instance ) action_block
cover_property_statement::=
        cover property ( property_spec ) statement_or_null
      | cover property ( property_instance ) statement_or_null
```

Syntax 17-16—Concurrent assert construct syntax (excerpt from Annex A)

The **assert** statement is used to enforce a **property** as a checker. When the property for the **assert** statement is evaluated to be true, the pass statements of the action block are executed. Otherwise, the fail statements of the *action_block* are executed. For example,

```
property abc(a,b,c);
   disable iff (a==2) not @clk (b ##1 c);
endproperty
env_prop: assert property (abc(rst,in1,in2)) pass_stat else fail_stat;
```

When no action is needed, a null statement (i.e.;) is specified. If no statement is specified for the **else**, then `$error` is used as the statement when the assertion fails.

The *action_block* shall not include any concurrent **assert** or **cover** statement. The *action_block*, however, can contain immediate assertion statements.

Note: The pass and fail statements are executed in the Reactive region. The regions of execution are explained in the scheduling semantics section, Section 14.

To monitor sequences and other behavioral aspects of the design for coverage, the same syntax is used with the **cover** statement. The tools can gather information about the evaluation and report the results at the end of simulation. When the property for the **cover** statement is successful, the pass statements can specify a coverage function, such as monitoring all paths for a sequence.

The **assert** or **cover** statements can be referenced by their optional name. A hierarchical name can be used consistent with the SystemVerilog naming conventions. When a name is not provided, a tool shall assign a name to the statement for the purpose of reporting.

Assertion control tasks are described in Section 22.6.

Coverage results are divided into two: coverage for properties, coverage for sequences.

For sequence coverage, the statement appears as:

```
cover property ( sequence_spec ) statement_or_null
```

The identifier of a particular attempt is called *attemptId*, and the clock tick of the occurrence of the match is called *clock step*.

The results of coverage statement for a property shall contain:

— Number of times attempted

— Number of times succeeded

— Number of times failed

— Number of times succeeded because of vacuity

— Each attempt with an *attemptID* and time

— Each success/failure with an *attemptID* and time

In addition, *statement_or_null* is executed every time a property succeeds.

Vacuity rules are applied only when implication operator is used. A property succeeds non-vacuously only if the consequent of the implication contributes to the success.

Results of coverage for a sequence shall include:

— Number of times attempted

— Number of times matched (each attempt can generate multiple matches)

— Each attempt with attemptId and time

— Each match with clock step, attemptID, and time

In addition, *statement_or_null* gets executed for every match. If there are multiple matches at the same time, the statement gets executed multiple times, one for each match.

17.12.1 Using concurrent assertion statements outside of procedural code

A concurrent assertion statement can be used outside of a procedural context. It can be used within a module as a *module_common_item*, an interface as a *module_common_item*, or a program as a *non_port_item*. A concurrent assertion statement is either an **`assert`** or a **`cover`** statement. Such a concurrent assertion statement uses the **`always`** semantics.

The following two forms are equivalent:

```
assert property ( property_spec ) action_block

always assert property ( property_spec ) action_block ;
```

Similarly, the following two forms are equivalent:

```
cover property ( property_spec ) statement_or_null

always cover property ( property_spec ) statement_or_null
```

For example:

```
module top(input bit clk);
   logic a,b,c;
   property rule3;
      @(posedge clk) a |-> b ##1 c;
   endproperty
   a1: assert property (rule3);
   ...
```



```
endmodule
```

`rule3` is a property declared in module `top`. The assert statement `a1` starts checking the property from the beginning to the end of simulation. The property is always checked. Similarly,

```
module top(input bit clk);
   logic a,b,c;
   sequence seq3;
      @(posedge clk) b ##1 c;
   endsequence
   c1: cover property (seq3);
   ...
endmodule
```

The cover statement `c1` starts coverage of the sequence `seq3` from beginning to the end of simulation. The sequence is always monitored for coverage.

17.12.2 Embedding concurrent assertions in procedural code

A concurrent assertion statement can also be embedded in a procedural block as a *statement_item*. For example:

```
property rule;
   a ##1 b ##1 c;
endproperty

always @(posedge clk) begin
   <statements>;
   assert property (rule);
end
```

If the statement appears in an **`always`** block, the property is always monitored. If the statement appears in an **`initial`** block, then the monitoring is performed only on the first clock tick.

Two inferences are made from the procedural context: clock from the event control of an **`always`** block, and the enabling conditions.

A clock is inferred if the statement is placed in an **`always`** or **`initial`** 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`** or **`initial`** block.

For example:

```
property r1;
   q != d;
endproperty
always @(posedge mclk) begin
   q <= d1;
   r1_p: assert property (r1);
end
```

The above property can be checked by writing statement `r1_p` outside the always block, and declaring the property with the clock as:

```
property r1;
   @(posedge mclk)q != d;
```

```
endproperty
always @(posedge mclk) begin
   q <= d1;
end
r1p: assert property (r1);
```

If the clock is explicitly specified with a property, then it must be identical to the inferred clock, as shown below:

```
property r2;
   @(posedge mclk)(q != d);
endproperty
always @(posedge mclk) begin
   q <= d1;
   r2_p: assert property (r2);
end
```

In the above example, `(posedge mclk)` is the clock for property `r2`.

Another inference made from the context is the enabling condition for a property. Such derivation takes place when a property is placed in an **`if`**...**`else`** block or a **`case`** block. The enabling condition assumed from the context is used as the antecedent of the property.

```
property r3;
   @(posedge sclk)(q != d);
endproperty
always @(posedge mclk) begin
   if (a) begin
      q <= d1;
      r3_p: assert property (r2);
   end
end
```

The above example is equivalent to:

```
property r3;
   @(posedge sclk)a |-> (q != d);
endproperty
r3_p: assert property (r3);
always @(posedge mclk) begin
   if (a) begin
      q <= d1;
   end
end
```

Similarly, the enabling condition is also inferred from **`case`** statements.

```
property r4;
   @(posedge sclk)(q != d);
endproperty
always @(posedge mclk) begin
   case (a)
      1: begin q <= d1;
               r4p: assert property (r4);
         end
      default: q1 <= d1;
   endcase
end
```


The above example is equivalent to:

```
property r4;
   @(posedge sclk)(a==1) |-> (q != d);
endproperty
r4_p: assert property (r4);
always @(posedge mclk) begin
   case (a)
      1: begin q <= d1;
         end
      default: q1 <= d1;
   endcase
end
```

The enabling condition is inferred from procedural code inside an **always** or **initial** block, with the following restrictions:

1) There must not be a preceding statement with a timing control.

2) A preceding statement shall not invoke a task call which contains a timing control on any statement.

3) The concurrent assertion statement shall not be placed in a looping statement, immediately, or in any nested scope of the looping statement.

17.13 Clock resolution

There are a number of ways to specify a clock for a property:

— sequence instance with a clock, for example

```
sequence s2; @(posedge clk) a ##2 b; endsequence
property p2; not s2; endproperty
assert property (p2);
```

— property, for example:

```
property p3; @(posedge clk) not (a ##2 b); endproperty
assert property (p3);
```

— contextually inferred clock from a procedural block, for example:

```
always @(posedge clk) assert property (not (a ##2 b));
```

— clocking domain, for example:

```
clocking master_clk @(posedge clk);
   property p3; not (a ##2 b); endproperty
endclocking
assert property (master_clk.p3);
```

— default clock, for example:

```
default clocking master_clk @(posedge clk);
```

For a multi-clocked assertion, the clocks are explicitly specified. No default clock or inferred clock is used. In addition, multi-clocked properties are not allowed to be defined within a clocking domain.

A multi-clocked property assert statement must not be embedded in procedural code where a clock is inferred. For example, following forms are not allowed.

```
always @(clk) assert property (mult_clock_prop);// illegal
initial @(clk) assert property (mult_clock_prop);// illegal
```

The rules for an assertion with one clock are discussed in the following paragraphs.

The clock for an assertion statement is determined in the decreasing order of priority:

1) Explicitly specified clock for the assertion.

2) Inferred clock from the context of the code when embedded.

3) Default clock, if specified.

A concurrent assertion statement must resolve to a clock. Otherwise, the statement is considered illegal.

Sequences and properties specified in clocking domains resolve the clock by the following rules:

1) Event control of the clocking domain specifies the clock.

2) No explicit event control is allowed in any property or sequence declaration.

3) If a named sequence that is defined outside the clocking domain is used , its clock, if specified, must be identical to the clocking domain's clock.

4) Multi-clock properties are not allowed.

Resolution of clock for a sequence definition assumes that only one explicit event control can be specified. Also, the named sequences used in the sequence definition can, but do not need to, contain event control in their definitions.

```
sequence s;
   //sequence composed of two named sub-sequences
   @(posedge s_clk) e ##1 s1 ##1 s2 ##1 f;
endsequence
sequence s1;
   @(posedge clk1) a ##1 b; // single clock sequence
endsequence
sequence s2;
   @(posedge clk2) c ##1 d; // single clock sequence
endsequence
```

These example sequences are used in the following table to explain the rules for a sequence definition. The clock of any sequence when explicitly specified is indicated by X. The absence of a clock is indicated by a dash.

Table 17-2: Rules for sequence definition

s_clk	clk1	clk2	Resolved clock	Semantic restriction
-	-	-	unclocked	-
X	-	-	s_clk	-
X	X	-	s_clk	`s_clk` and `clk1` must be identical
X	X	X	s_clk	`s_clk`, `clk1` and `clk2` must be identical
X	-	X	s_clk	`s_clk` and `clk2` must be identical
-	X	-	unclocked	-

Table 17-2: Rules for sequence definition

s_clk	clk1	clk2	Resolved clock	Semantic restriction
-	X	X	unclocked	`clk1` and `clk2` must be identical
-	-	X	unclocked	-

Once the clock for a sequence definition is determined, the clock of a property definition is resolved similar to the resolution for a sequence definition. A single clocked property assumes that only one explicit event control can be specified. Also, the named sequences used in the property definition can contain event control in their definitions. The following table specifies the rules for property definition clock resolution. The property has the form:

```
property p;
   @(posedge p_clk) not s1 |=> s2;
endproperty
```

`p_clk` is the property for the clock, `clk1` is the clock for sequence s1 and `clk2` is the clock for sequence `s2`. The same rules apply for operator `|->`.

Resolution of clock for an **`assert`** statement is based on the following assumptions:

Table 17-3: Resolution of clock for an assert statement

p_clk	clk1	clk2	Resolved clock	Semantic restriction
-	-	-	unclocked	-
X	-	-	p_clk	-
X	X	-	p_clk	`p_clk` and `clk1` must be identical
X	X	X	p_clk	`p_clk`, `clk1` and `clk2` must be identical
X	-	X	p_clk	`p_clk` and `clk2` must be identical
-	X	-	unclocked	-
-	X	X	unclocked or multi-clock	`clk1` and `clk2` must be identical. If `clk1` and `clk2` are different for the case of operator `|=>`, then it is considered a multi-clock implication
-	-	X	unclocked	-

Resolution of clock for an **`assert`** statement is based on the following assumptions:

— **`assert`** can appear in an **`always`** block, **`initial`** block or outside procedural context

— clock is inferred from an **`always`** or **`initial`** block

— default clock can be specified using default clocking domain

The following table specifies the rules for clock resolution when **`assert`** appears in an always or initial block, where `i_clk` is the inferred clock from an **`always`** or **`initial`** block, `d_clk` is the default clock, and `p_clk` is the property clock.

Table 17-4: Resolution of clock in an always or initial block

i_clk	d_clk	p_clk	Resolved clock	Semantic restriction
-	-	-	unclocked	Error. An assertion must have a clock
X	-	-	i_clk	-
-	X	-	d_clk	
-	-	X	p_clk	
X	-	X	i_clk	`i_clk` and `p_clk` must be identical
X	X	-	i_clk	-
-	X	X	p_clk	
-	-	X	p_clk	-

When the **`assert`** statement is outside any procedural block, there is no inferred clock. The rules for clock resolution are specified in the table below.

Table 17-5: Resolution of clock outside a procedural block

d_clk	p_clk	Resolved clock	Semantic restriction
–	–	unclocked	Error. An assertion must have a clock
X	-	d_clk	
–	X	p_clk	
X	X	p_clk	

17.14 Binding properties to scopes or instances

To facilitate verification separate from the design, it is possible to specify properties and bind them to specific modules or instances. The following are the goals of providing this feature:

— It allows verification engineers to verify with minimum changes to the design code/files.

— It allows a convenient mechanism to attach verification IP to a module or an instance.

— No semantic changes to the assertions are introduced due to this feature. It is equivalent to writing properties external to a module, using hierarchical path names.

With this feature, a user can bind a module, interface, or program instance to a module or a module instance.

The syntax of the **`bind`** construct is:


```
bind_directive ::=                                          // from Annex A.1.5
        bind module_identifier bind_instantiation ;
      | bind name_of_instance bind_instantiation ;
bind_instantiation ::=
        program_instantiation
      | module_instantiation
      | interface_instantiation
```

Syntax 17-17—bind construct syntax (excerpt from Annex A)

The **bind** directive can be specified in

— a module as a *module_or_generate_item*

— $root.

A program block contains non-design code (either testbench or properties) and executes in the Reactive region, as explained in Section 16.

Example of binding a program instance to a module:

```
bind cpu fpu_props fpu_rules_1(a,b,c);
```

Where:

— `cpu` is the name of module.

— `fpu_props` is the name of the program containing properties.

— `fpu_rules_1` is the program instance name.

— Ports `(a, b,c)` get bound to signals `(a,b,c)` of module `cpu`.

— Every instance of `cpu` gets the properties.

Example of binding a program instance to a specific instance of a module:

```
bind cpu1 fpu_props fpu_rules_1(a,b,c);
```

By binding a program to a module or an instance, the program becomes part of the bound object. The names of assertion-related declarations can be referenced using the SystemVerilog hierarchical naming conventions.

Binding of a module instance or an interface instance works the same way as described for programs above.

```
interface range (input clk,enable, input int minval,expr);
   property crange_en;
      @(posedge clk) enable |-> (minval <= expr);
   endproperty
range_chk: assert property (crange_en);
endinteface

bind cr_unit range r1(c_clk,c_en,v_low,(in1&&in2));
```

In this example, interface `range` is instantiated in the module `cr_unit`. Effectively, every instance of module `cr_unit` shall contain the interface instance `r1`.

Where:

— `cpu1` is the name of module instance (`cpu1` is an instance of module of module `cpu`).

— `fpu_props` is the name of the program containing properties.

— `fpu_rules_1` is the program instance name.

— Ports `(a, b,c)` get bound to signals `(a,b,c)` of module instance `cpu1`.

— Only the `cpu1` instance of `cpu` gets the properties.

By binding a program to a module or an instance, the program becomes part of the bound object. The names of assertion related declarations can be referenced using the SystemVerilog hierarchical naming conventions.

Section 18
Hierarchy

18.1 Introduction (informative)

Verilog has a simple organization. All data, functions and tasks are in modules except for system tasks and functions, which are global, and can be defined in the PLI. A Verilog module can contain instances of other modules. Any uninstantiated module is at the top level. This does not apply to libraries, which therefore have a different status and a different procedure for analyzing them. A hierarchical name can be used to specify any named object from anywhere in the instance hierarchy. The module hierarchy is often arbitrary and a lot of effort is spent in maintaining port lists.

In Verilog, only net, **`reg`**, **`integer`** and **`time`** data types can be passed through module ports.

SystemVerilog adds many enhancements for representing design hierarchy:

— A global declaration space, visible to all modules at all levels of hierarchy

— Nested module declarations, to aid in representing self-contained models and libraries

— Relaxed rules on port declarations

— Simplified named port connections, using `.name`

— Implicit port connections, using **`.*`**

— Time unit and time precision specifications bound to modules

— A concept of interfaces to bundle connections between modules (presented in Section 19)

An important enhancement in SystemVerilog is the ability to pass any data type through module ports, including nets, and all variable types including reals, arrays, and structures.

18.2 The $root top level

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

SystemVerilog requires an elaboration phase. All modules and interfaces must be parsed before elaboration. The order of elaboration shall be: First, look for explicit instantiations in $root. If none, then look for implicit instantiations (i.e. uninstantiated modules). Next, traverse non-generate instantiations depth-first, in source order. Finally, execute generate blocks depth-first, in source order.

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

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

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

The $root top level:

— has a single occurrence

— can be distributed across any number of files

— variable and net definitions are in a global name space and can be accessed throughout the hierarchy

— task and function definitions are in a global name space and can be accessed throughout the hierarchy

— shall not contain **initial** or **always** procedures

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

— can contain assertion declarations, assertion statements and bind directives

Modules:

— can have any number of module definitions

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

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

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

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

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

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

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

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

```
typedef int myint;

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

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

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

main();
```


18.3 Module declarations

```
module_declaration ::=                                          // from Annex A.1.3
        module_nonansi_header [ timeunits_declaration ] { module_item }
        endmodule [ : module_identifier ]
      | module_ansi_header [ timeunits_declaration ] { non_port_module_item }
        endmodule [ : module_identifier ]
      | { attribute_instance } module_keyword [ lifetime ] module_identifier ( .* ) ;
        [ timeunits_declaration ] { module_item } endmodule [ : module_identifier ]
      | extern module_nonansi_header
      | extern module_ansi_header
module_keyword ::= module | macromodule
timeunits_declaration ::=
        timeunit time_literal ;
      | timeprecision time_literal ;
      | timeunit time_literal ;
        timeprecision time_literal ;
      | timeprecision time_literal ;
        timeunit time_literal ;
```

Syntax 18-1—Module declaration syntax (excerpt from Annex A)

In Verilog, a module must be declared apart from other modules, and can only be instantiated within another module. A module declaration can appear after it is instantiated in the source text.

SystemVerilog adds the capability to nest module declarations, and to instantiate modules in the `$root` top-level space, outside of other modules.

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

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

module m3(...);

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

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

18.4 Nested modules

A module can be declared within another module. The outer name space is visible to the inner module, so that any name declared there can be used, unless hidden by a local name, provided the module is declared and instantiated in the same scope.

One purpose of nesting modules is to show the logical partitioning of a module without using ports. Names that are global are in the outermost scope, and names that are only used locally can be limited to local modules.

```
// This example shows a D-type flip-flop made of NAND gates
module dff_flat(input d, ck, pr, clr, output q, nq);
```

```
wire q1, nq1, q2, nq2;

    nand g1b (nq1, d, clr, q1);
    nand g1a (q1, ck, nq2, nq1);

    nand g2b (nq2, ck, clr, q2);
    nand g2a (q2, nq1, pr, nq2);

    nand g3a (q, nq2, clr, nq);
    nand g3b (nq, q1, pr, q);
endmodule


// This example shows how the flip-flop can be structured into 3 RS latches.
module dff_nested(input d, ck, pr, clr, output q, nq);
wire q1, nq1, nq2;

    module ff1;
        nand g1b (nq1, d, clr, q1);
        nand g1a (q1, ck, nq2, nq1);
    endmodule
    ff1 i1;

    module ff2;
        wire q2; // This wire can be encapsulated in ff2
        nand g2b (nq2, ck, clr, q2);
        nand g2a (q2, nq1, pr, nq2);
    endmodule
    ff2 i2;

    module ff3;
        nand g3a (q, nq2, clr, nq);
        nand g3b (nq, q1, pr, q);
    endmodule
    ff3 i3;
endmodule
```

The nested module declarations can also be used to create a library of modules that is local to part of a design.

```
module part1(....);
   module and2(input a; input b; output z);
   ....
   endmodule
   module or2(input a; input b; output z);
   ....
   endmodule
   ....
   and2 u1(....), u2(....), u3(....);
   .....
endmodule
```

This allows the same module name, e.g. `and2`, to occur in different parts of the design and represent different modules. Note that an alternative way of handling this problem is to use configurations.

To support separate compilation, extern declarations of a module can be used to declare the ports on a module without defining the module itself. An extern module declaration consists of the keyword **`extern`** followed by the module name and the list of ports for the module. Both list of ports syntax (possibly with parameters), and original Verilog style port declarations can be used. Note that the potential existence of defparams precludes the checking of the port connection information prior to elaboration time even for list of ports style declara-

tions.

The following example demonstrates the usage of extern module declarations.

```
extern module m (a,b,c,d);
extern module a #(parameter size= 8, parameter type TP = logic [7:0])
                (input [size:0] a, output TP b);

module top ();
   wire [8:0] a;
   logic [7:0] b;

   m m (.*);
   a a (.*);
endmodule
```

Modules `m` and `a` are then assumed to be instantiated as:

```
module top ();
   m m (a,b,c,d);
   a a (a,b);
endmodule
```

If an **`extern`** declaration exists for a module, it is possible to use **`.*`** as the ports of the module. This usage shall be equivalent to placing the ports (and possibly parameters) of the **`extern`** declaration on the module.

For example,

```
extern module m (a,b,c,d);
extern module a #(parameter size = 8, parameter type TP = logic [7:0])
                (input [size:0] a, output TP b);

module m (.*);
   input a,b,c;
   output d;
endmodule

module a (.*);
   ...
endmodule
```

is equivalent to writing:

```
module m (a,b,c,d);
   input a,b,c;
   output d;
endmodule

module a #(parameter size = 8, parameter type TP = logic [7:0])
         (input [size:0] a, output TP b);
endmodule
```

Extern module declarations can appear at any level of the instantiation hierarchy, but are visible only within the level of hierarchy in which they are declared. It shall be an error for the module definition to not exactly match the extern module declaration.

18.5 Port declarations

```
inout_declaration ::=                                          // from Annex A.2.1.2
        inout [ port_type ] list_of_port_identifiers
      | inout data_type list_of_variable_identifiers
input_declaration ::=
        input [ port_type ] list_of_port_identifiers
      | input data_type list_of_variable_identifiers
output_declaration ::=
        output [ port_type ] list_of_port_identifiers
      | output data_type  list_of_variable_port_identifiers
interface_port_declaration ::=
        interface_identifier list_of_interface_identifiers
      | interface_identifier . modport_identifier  list_of_interface_identifiers
ref_declaration ::= ref data_type list_of_port_identifiers
generic_interface_port_declaration ::=
        interface list_of_interface_identifiers
      | interface . modport_identifier list_of_interface_identifiers
port_type ::=                                                  // from Annex A.2.2.1
        data_type
      | net_type [ signing ] { packed_dimension }
      | trireg [ signing ] { packed_dimension }
      | [ signing ] { packed_dimension } range
signing ::= signed | unsigned
```

Syntax 18-2—Port declaration syntax (excerpt from Annex A)

With SystemVerilog, a port can be a declaration of a net, an interface, an event, or a variable of any type, including an array, a structure or a union.

```
typedef struct {
            bit isfloat;
            union { int i; shortreal f; } n;
} tagged; // named structure

module mh1 (input int in1, input shortreal in2, output tagged out);
   ...
endmodule
```

For the first port, if neither a type nor a direction is specified, then it shall be assumed to be a member of a port list, and any port direction or type declarations must be declared after the port list. This is compatible with the Verilog-1995 syntax. If the first port type but no direction is specified, then the port direction shall default to **`inout`**. If the first port direction but no type is specified, then the port type shall default to **`wire`**. This default type can be changed using the `` `default_nettype `` compiler directive, as in Verilog.

```
// Any declarations must follow the port list, because first port does not
// have either a direction or type specified; Port directions default to inout
module mh4(x, y);
   wire x;
   tri0 y;
   ...
endmodule
```


For subsequent ports in the port list, if the type and direction are omitted, then both are inherited from the previous port. If only the direction is omitted, then it is inherited from the previous port. If only the type is omitted, it shall default to **wire**. This default type can be changed using the `default_nettype compiler directive, as in Verilog.

```
// second port inherits its direction and type from previous port
module mh3 (input byte a, b);
   ...
endmodule
```

Generic interface ports cannot be declared using the Verilog-1995 list of ports style. Generic interface ports can only be declared by using a list of port declaration style.

```
module cpuMod(interface d, interface j);
   ...
endmodule
```

18.6 Time unit and precision

SystemVerilog has a time unit and precision declaration which has the equivalent functionality of the `timescale compiler directives in Verilog-2001. Use of these declarations removes the file order dependencies problems with compiler directives. The time unit and precision can be declared by the **timeunit** and **timeprecision** keywords, respectively, and set to a time literal which must be a power of 10 units. For example:

```
timeunit 100ps;
timeprecision 10fs;
```

There shall be at most one time unit and one time precision for any module or interface definition, or in $root. This shall define a time scope. If specified, the **timeunit** and **timeprecision** declarations shall precede any other items in the current time scope. The **timeunit** and **timeprecision** declarations can be repeated as later items, but must match the previous declaration within the current time scope.

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

1) If the module or interface definition is nested, then the time unit is shall be inherited from the enclosing module or interface.

2) Else, if a `timescale directive has been previously specified, then the time unit is shall be set to the units of the last `timescale directive.

3) Else, if the $root top level has a time unit, then the time unit is shall be set to the time units of the root module.

4) Else, the default time unit is shall be used.

The time unit of $root shall only be determined by a **timeunit** declaration, not a `timescale directive.

If a **timeprecision** is not specified in the current time scope, then the time precision is shall be determined following the same precedence as with time units.

The global time precision is the minimum of all the timeprecision statements and the smallest time precision argument of all the `timescale compiler directives (known as the precision of the time unit of the simulation in Section 19.8 of the IEEE 1364-2001 standard) in the design. The step time unit is equal to the global time precision.

18.7 Module instances

```
module_instantiation ::=                                          // from Annex A.4.1.1
        module_identifier [ parameter_value_assignment ] module_instance { , module_instance } ;
parameter_value_assignment ::= # ( list_of_parameter_assignments )
list_of_parameter_assignments ::=
        ordered_parameter_assignment { , ordered_parameter_assignment }
      | named_parameter_assignment { , named_parameter_assignment }
ordered_parameter_assignment ::= expression | data_type
named_parameter_assignment ::=
        . parameter_identifier ( [ expression ] )
      | . parameter_identifier ( data_type )
module_instance ::= name_of_instance ( [ list_of_port_connections ] )
name_of_instance ::= module_instance_identifier { range }
list_of_port_connections ::=
        ordered_port_connection { , ordered_port_connection }
      | dot_named_port_connection { , dot_named_port_connection }
      | { named_port_connection , } dot_star_port_connection { , named_port_connection }
ordered_port_connection ::= { attribute_instance } [ expression ]
named_port_connection ::= { attribute_instance } . port_identifier ( [ expression ] )
dot_named_port_connection ::=
        { attribute_instance } .port_identifier
      | named_port_connection
dot_star_port_connection ::= { attribute_instance } .*
```

Syntax 18-3—Module instance syntax (excerpt from Annex A)

A module can be used (instantiated) in two ways, hierarchical or top level. Hierarchical instantiation allows more than one instance of the same type. The module name can be a module previously declared or one declared later. Actual parameters can be named or ordered. Port connections can be named, ordered or implicitly connected. They can be nets, variables, or other kinds of interfaces, events, or expressions. See below for the connection rules.

Consider an ALU accumulator (`alu_accum`) example module that includes instantiations of an ALU module, an accumulator register (`accum`) module and a sign-extension (`xtend`) module. The module headers for the three instantiated modules are shown in the following example code.

```
module alu (
   output reg [7:0] alu_out,
   output reg zero,
   input [7:0] ain, bin,
   input [2:0] opcode);
   // RTL code for the alu module
endmodule

module accum (
   output reg [7:0] dataout,
   input [7:0] datain,
   input clk, rst_n);
   // RTL code for the accumulator module
endmodule
```



```
module xtend (
   output reg [7:0] dout,
   input din,
   input clk, rst_n);
   // RTL code for the sign-extension module
endmodule
```

18.7.1 Instantiation using positional port connections

Verilog has always permitted instantiation of modules using positional port connections, as shown in the `alu_accum1` module example, below.

```
module alu_accum1 (
   output [15:0] dataout,
   input [7:0] ain, bin,
   input [2:0] opcode,
   input clk, rst_n);
   wire [7:0] alu_out;

   alu alu (alu_out, , ain, bin, opcode);
   accum accum (dataout[7:0], alu_out, clk, rst_n);
   xtend xtend (dataout[15:8], alu_out[7], clk, rst_n);
endmodule
```

As long as the connecting variables are ordered correctly and are the same size as the instance-ports that they are connected to, there shall be no warnings and the simulation shall work as expected.

18.7.2 Instantiation using named port connections

Verilog has always permitted instantiation of modules using named port connections as shown in the `alu_accum2` module example.

```
module alu_accum2 (
   output [15:0] dataout,
   input [7:0] ain, bin,
   input [2:0] opcode,
   input clk, rst_n);
   wire [7:0] alu_out;

   alu   alu   (.alu_out(alu_out), .zero(),
                .ain(ain), .bin(bin), .opcode(opcode));
   accum accum (.dataout(dataout[7:0]), .datain(alu_out),
                .clk(clk), .rst_n(rst_n));
   xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]),
                .clk(clk), .rst_n(rst_n));
endmodule
```

Named port connections do not have to be ordered the same as the ports of the instantiated module. The variables connected to the instance ports must be the same size or a port-size mismatch warning shall be reported.

18.7.3 Instantiation using implicit .name port connections

SystemVerilog adds the capability to implicitly instantiate ports using a .name syntax if the instance-port name and size match the connecting variable-port name and size. This enhancement eliminates the requirement to list a port name twice when the port name and signal name are the same, while still listing all of the ports of the instantiated module for documentation purposes.

In the following `alu_accum3` example, all of the ports of the instantiated alu module match the names of the

variables connected to the ports, except for the unconnected zero port, which is listed using a named port connection, showing that the port is unconnected. Implicit .name port connections are made for all name and size matching connections on the instantiated module.

In the same `alu_accum3` example, the `accum` module has an 8-bit port called `dataout` that is connected to a 16-bit bus called `dataout`. Because the internal and external sizes of `dataout` do not match, the port must be connected by name, showing which bits of the 16-bit bus are connected to the 8-bit port. The `datain` port on the `accum` is connected to a bus by a different name (`alu_out`), so this port is also connected by name. The `clk` and `rst_n` ports are connected using implicit .name port connections. Also in the same `alu_accum3` example, the `xtend` module has an 8-bit output port called `dout` and a 1- bit input port called `din`. Since neither of these port names match the names (or sizes) of the connecting variables, both are connected by name. The `clk` and `rst_n` ports are connected using implicit .name port connections.

```
module alu_accum3 (
   output [15:0] dataout,
   input [7:0] ain, bin,
   input [2:0] opcode,
   input clk, rst_n);
   wire [7:0] alu_out;

   alu   alu   (.alu_out, .zero(), .ain, .bin, .opcode);
   accum accum (.dataout(dataout[7:0]), .datain(alu_out), .clk, .rst_n);
   xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]), .clk, .rst_n);
endmodule
```

A *.port_identifier* port connection is semantically equivalent to the named port connection *.port_identifier*`(name)` port connection with the following exceptions:

— The identifier referenced by *.port_identifier* shall not create an implicit wire declaration.

— It shall be illegal for a *.port_identifier* port connection to create an implicit cast. This includes truncation or padding.

— A conversion between a 2-state and 4-state type of the same bit length is a legitimate cast.

— A port connection between a net type and a variable type of the same bit length is a legitimate cast.

— It shall be an error if a *.port_identifier* port connection between two dissimilar net types would generate a warning message as required by the Verilog-2001 standard.

18.7.4 Instantiation using implicit .* port connections

SystemVerilog adds the capability to implicitly instantiate ports using a **`.*`** syntax for all ports where the instance-port name and size match the connecting variable-port name and size. This enhancement eliminates the requirement to list any port where the name and size of the connecting variable match the name and size of the instance port. This implicit port connection style is used to indicate that all port names and sizes match the connections where emphasis is placed only on the exception ports. The implicit `.*` port connection syntax can greatly facilitate rapid block-level testbench generation where all of the testbench variables are chosen to match the instantiated module port names and sizes.

In the following `alu_accum4` example, all of the ports of the instantiated alu module match the names of the variables connected to the ports, except for the unconnected zero port, which is listed using a named port connection, showing that the port is unconnected. The implicit `.*` port connection syntax connects all other ports on the instantiated module.

In the same `alu_accum4` example, the `accum` module has an 8-bit port called `dataout` that is connected to a 16-bit bus called `dataout`. Because the internal and external sizes of `dataout` do not match, the port must be connected by name, showing which bits of the 16-bit bus are connected to the 8-bit port. The `datain` port on the `accum` is connected to a bus by a different name (`alu_out`), so this port is also connected by name. The `clk` and `rst_n` ports are connected using implicit `.*` port connections. Also in the same `alu_accum4` exam-

ple, the `xtend` module has an 8-bit output port called `dout` and a 1- bit input port called `din`. Since neither of these port names match the names (or sizes) of the connecting variables, both are connected by name. The `clk` and `rst_n` ports are connected using implicit `.*` port connections.

```
module alu_accum4 (
   output [15:0] dataout,
   input [7:0] ain, bin,
   input [2:0] opcode,
   input clk, rst_n);
   wire [7:0] alu_out;

   alu   alu   (.*, .zero());
   accum accum (.*, .dataout(dataout[7:0]), .datain(alu_out));
   xtend xtend (.*, .dout(dataout[15:8]), .din(alu_out[7]));
endmodule
```

An implicit `.*` port connection is semantically equivalent to a default .name port connection for every port declared in the instantiated module. A named port connection can be mixed with a `.*` connection to override the port connection to a different expression or to leave the port unconnected.

When the implicit `.*` port connection is mixed in the same instantiation with named port connections, the implicit `.*` port connection token can be placed anywhere in the port list. The `.*` token can only appear at most once in the port list.

Modules can be instantiated into the same parent module using any combination of legal positional, named, implicit .name connected and implicit `.*` connected instances as shown in `alu_accum5` example.

```
module alu_accum5 (
   output [15:0] dataout,
   input [7:0] ain, bin,
   input [2:0] opcode,
   input clk, rst_n);
   wire [7:0] alu_out;

   // mixture of named port connections and
   // implicit .name port connections
   alu   alu   (.ain(ain), .bin(bin), .alu_out, .zero(), .opcode);

   // positional port connections
   accum accum (dataout[7:0], alu_out, clk, rst_n);

   // mixture of named port connections and implicit .* port connections
   xtend xtend (.dout(dataout[15:8]), .*, .din(alu_out[7]));
endmodule
```

18.8 Port connection rules

SystemVerilog extends Verilog port connections by making all variable data types available to pass through ports. It does this by allowing both sides of a port connection to have the same compatible data type, and by allowing continuous assignments to variables. It also creates a new type of port qualifier, **`ref`**, to allow shared variable behavior across a port by passing a hierarchical reference.

18.8.1 Port connection rules for variables

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-2001.

— 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 can not be left unconnected

18.8.2 Port connection rules for nets

If a port declaration has a **wire** type (which is the default), or any other net type, then its direction controls how it can be connected as follows:

— An **input** can be connected to any expression of a compatible data type. If left unconnected, it shall have the value 'z.

— An **output** can be connected to a net type (or a concatenation of net types) or a compatible variable type (or a concatenation of variable types).

— An **inout** can be connected to a net type (or a concatenation of net types) or left unconnected, but not to a variable type.

Note that where the data types differ between the port declaration and connection, an initial value change event can be caused at time zero.

18.8.3 Port connection rules for interfaces

A port declaration can be a generic interface or named interface type. An interface port instance must always be connected to an interface instance or a higher-level interface port. An interface port cannot be left unconnected.

If a port declaration has a generic interface type, then it can be connected to an interface instance of any type. If a port declaration has a named interface type, then it must be connected to an interface instance of the identical type.

18.8.4 Compatible port types

The same rules for assignment compatibility are used for compatible port types for ports declared as an **input** or an **output** variable, or for **output** ports connected to variables. SystemVerilog does not change any of the other port connection compatibility rules

18.8.5 Unpacked array ports and arrays of instances

For an unpacked array port, the port and the array connected to the port must have the same number of unpacked dimensions, and each dimension of the port must have the same size as the corresponding dimension of the array being connected.

If the size and type of the port connection match the size and type of a single instance port, the connection shall be made to each instance in an array of instances.

If the port connection is an unpacked array, the unpacked array dimensions of each port connection shall be compared with the dimensions of the instance array. If they match exactly in size, each element of the port connection shall be matched to the port left index to left index, right index to right index. If they do not match it shall be considered an error.

If the port connection is a packed array, each instance shall get a part-select of the port connection, starting with all right-hand indices to match the right most part-select, and iterating through the right most dimension first. Too many or too few bits to connect all the instances shall be considered an error.

18.9 Name spaces

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

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

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

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

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

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

18.10 Hierarchical names

Hierarchical names are also called nested identifiers. They consist of instance names separated by periods, where an instance name can be an array element.

```
$root.mymodule.u1 // absolute name
u1.struct1.field1 // u1 must be visible locally or above, including globally
adder1[5].sum
```

Nested identifiers can be read (in expressions), written (in assignments or task/function calls) or triggered off (in event expressions). They can also be used as type, task or function names.

Section 19
Interfaces

19.1 Introduction (informative)

The communication between blocks of a digital system is a critical area that can affect everything from ease of RTL coding, to hardware-software partitioning to performance analysis to bus implementation choices and protocol checking. The interface construct in SystemVerilog was created specifically to encapsulate the communication between blocks, allowing a smooth migration from abstract system-level design through successive refinement down to lower-level register-transfer and structural views of the design. By encapsulating the communication between blocks, the interface construct also facilitates design re-use. The inclusion of interface capabilities is one of the major advantages of SystemVerilog.

At its lowest level, an interface is a named bundle of nets or variables. The interface is instantiated in a design and can be passed through a port as a single item, and the component nets or variables referenced where needed. A significant proportion of a Verilog design often consists of port lists and port connection lists, which are just repetitions of names. The ability to replace a group of names by a single name can significantly reduce the size of a description and improve its maintainability.

Additional power of the interface comes from its ability to encapsulate functionality as well as connectivity, making an interface, at its highest level, more like a class template. An interface can have parameters, constants, variables, functions and tasks. The types of elements in an interface can be declared, or the types can be passed in as parameters. The member variables and functions are referenced relative to the instance name of the interface as instance.member. Thus, modules that are connected via an interface can simply call the task/function members of that interface to drive the communication. With the functionality thus encapsulated in the interface, and isolated from the module, the abstraction level and/or granularity of the communication protocol can be easily changed by replacing the interface with a different interface containing the same members but implemented at a different level of abstraction. The modules connected via the interface don’t need to change at all.

To provide direction information for module ports and to control the use of tasks and functions within particular modules, the **`modport`** construct is provided. As the name indicates, the directions are those seen from the module.

In addition to task/function methods, an interface can also contain processes (i.e. **`initial`** or **`always`** blocks) and continuous assignments, which are useful for system-level modeling and testbench applications. This allows the interface to include, for example, its own protocol checker that automatically verifies that all modules connected via the interface conform to the specified protocol. Other applications, such as functional coverage recording and reporting, protocol checking and assertions can also be built into the interface.

The methods can be abstract, i.e. defined in one module and called in another, using the export and import constructs. This could be coded using hierarchical path names, but this would impede re-use because the names would be design-specific. A better way is to declare the task and function names in the interface, and to use local hierarchical names from the interface instance for both definition and call. Broadcast communication is modeled by **`forkjoin`** tasks, which can be defined in more than one module and executed concurrently.

19.2 Interface syntax

```
interface_declaration ::=                                          // from Annex A.1.3
        interface_nonansi_header [ timeunits_declaration ] { interface_item }
        endinterface [ : interface_identifier ]
      | interface_ansi_header [ timeunits_declaration ] { non_port_interface_item }
        endinterface [ : interface_identifier ]
      | { attribute_instance } interface interface_identifier ( .* ) ;
        [ timeunits_declaration ] { interface_item }
        endinterface [ : interface_identifier ]
      | extern interface_nonansi_header
      | extern interface_ansi_header
interface_nonansi_header ::=
        { attribute_instance } interface [ lifetime ] interface_identifier
        [ parameter_port_list ] list_of_ports ;
interface_ansi_header ::=
        {attribute_instance } interface [ lifetime ] interface_identifier
        [ parameter_port_list ] [ list_of_port_declarations ] ;
modport_declaration ::= modport modport_item { , modport_item } ;  // from Annex A.2.9
modport_item ::= modport_identifier ( modport_ports_declaration { , modport_ports_declaration } )
modport_ports_declaration ::=
        modport_simple_ports_declaration
      | modport_hierarchical_ports_declaration
      | modport_tf_ports_declaration
modport_simple_ports_declaration ::=
        input list_of_modport_port_identifiers
      | output list_of_modport_port_identifiers
      | inout list_of_modport_port_identifiers
      | ref [ data_type ] list_of_modport_port_identifiers
modport_hierarchical_ports_declaration ::=
      interface_instance_identifier [ [ constant_expression ] ] . modport_identifier
modport_tf_ports_declaration ::=
      import_export modport_tf_port
modport_tf_port ::=
        task named_task_proto { , named_task_proto }
      | function named_function_proto { , named_function_proto }
      | task_or_function_identifier { , task_or_function_identifier }
import_export ::= import | export
interface_instantiation ::=                                        // from Annex A.4.1.2
        interface_identifier [ parameter_value_assignment ] module_instance { , module_instance } ;
```

Syntax 19-1—Interface syntax (excerpt from Annex A)

The interface construct provides a new hierarchical structure. It can contain smaller interfaces and can be passed through ports.

The aim of interfaces is to encapsulate communication. At the lower level, this means bundling variables and wires in interfaces, and bundling ports with directions in modports. The modules can be made generic so that the interfaces can be changed. The following examples show these features. At a higher level of abstraction, communication can be done by tasks and functions. Interfaces can include task and function definitions, or just

task and function prototypes with the definition in one module (server/slave) and the call in another (client/master).

A simple interface declaration is as follows (see Syntax 19-1 for the complete syntax):

```
interface identifier;
   ...
   interface_items
   ...
endinterface [ : identifier ]
```

An interface can be instantiated hierarchically like a module, with or without ports. For example:

```
myinterface #(100) scalar1, vector[9:0];
```

Interfaces can be declared and instantiated in modules (either flat or hierarchical) but modules can neither be declared nor instantiated in interfaces.

The simplest use of an interface is to bundle wires, as is illustrated in the examples below.

19.2.1 Example without using interfaces

This example shows a simple bus implemented without interfaces. Note that the logic type can replace wire and reg if no resolution of multiple drivers is needed.

```
module memMod( input    bit req,
                        bit clk,
                        bit start,
                        logic [1:0] mode,
                        logic [7:0] addr,
               inout    wire [7:0] data,
               output   bit gnt,
                        bit rdy );
   logic avail;

   ...
endmodule

module cpuMod(
   input    bit clk,
            bit gnt,
            bit rdy,
   inout    wire [7:0] data,
   output   bit req,
            bit start,
            logic [7:0] addr,
            logic [1:0] mode );
   ...
endmodule

module top;
   logic req, gnt, start, rdy; // req is logic not bit here
   logic clk = 0;
   logic [1:0] mode;
   logic [7:0] addr;
   wire  [7:0] data;

   memMod mem(req, clk, start, mode, addr, data, gnt, rdy);
   cpuMod cpu(clk, gnt, rdy, data, req, start, addr, mode);
```



```
endmodule
```

19.2.2 Interface example using a named bundle

The simplest form of a SystemVerilog interface is a bundled collection of variables or nets. When an interface is used as a port, the variables and nets in it are assumed to be **ref** and **inout** ports, respectively. The following interface example shows the basic syntax for defining, instantiating and connecting an interface. Usage of the SystemVerilog interface capability can significantly reduce the amount of code required to model port connections.

```
interface simple_bus; // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;
endinterface: simple_bus

module memMod(simple_bus a, // Use the simple_bus interface
              input bit clk);
   logic avail;
   // a.req is the req signal in the 'simple_bus' interface
   always @(posedge clk) a.gnt <= a.req & avail;
endmodule

module cpuMod(simple_bus b, input bit clk);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(); // Instantiate the interface

   memMod mem(sb_intf, clk); // Connect the interface to the module instance
   cpuMod cpu(.b(sb_intf), .clk(clk)); // Either by position or by name

endmodule
```

In the preceding example, if the same identifier, `sb_intf`, had been used to name the `simple_bus` interface in the `memMod` and `cpuMod` module headers, then implicit port declarations also could have been used to instantiate the `memMod` and `cpuMod` modules into the top module, as shown below.

```
module memMod (simple_bus sb_intf, input bit clk);
   ...
endmodule

module cpuMod (simple_bus sb_intf, input bit clk);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf();

   memMod mem (.*);  // implicit port connections
   cpuMod cpu (.*);  // implicit port connections
```

```
endmodule
```

19.2.3 Interface example using a generic bundle

A module header can be created with an unspecified interface instantiation as a place-holder for an interface to be selected when the module itself is instantiated. The unspecified interface is referred to as a “generic” interface port.

This generic interface port can only be declared by using the list of port declaration style port declaration style. It shall be illegal to declare such a generic interface port using the old Verilog-1995 list of port style.

The following interface example shows how to specify a generic interface port in a module definition.

```
// memMod and cpuMod can use any interface
module memMod (interface a, input bit clk);
   ...
endmodule

module cpuMod(interface b, input bit clk);
   ...
endmodule

interface simple_bus; // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;
endinterface: simple_bus

module top;
   logic clk = 0;

   simple_bus sb_intf(); // Instantiate the interface

   // Connect the sb_intf instance of the simple_bus
   // interface to the generic interfaces of the
   // memMod and cpuMod modules
   memMod mem (.a(sb_intf), .clk(clk));
   cpuMod cpu (.b(sb_intf), .clk(clk));

endmodule
```

An implicit port cannot be used to connect to a generic interface. A named port must be used to connect to a generic interface, as shown below.

```
module memMod (interface a, input bit clk);
   ...
endmodule

module cpuMod (interface b, input bit clk);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf();

   memMod mem (.*, .a(sb_intf)); // partial implicit port connections
```



```
   cpuMod cpu (.*, .b(sb_intf)); // partial implicit port connections

endmodule
```

19.3 Ports in interfaces

One limitation of simple interfaces is that the nets and variables declared within the interface are only used to connect to a port with the same nets and variables. To share an external net or variable, one that makes a connection from outside of the interface as well as forming a common connection to all module ports that instantiate the interface, an interface port declaration is required. The difference between nets or variables in the interface port list and other nets or variables within the interface is that only those in the port list can be connected externally by name or position when the interface is instantiated.

```
interface i1 (input a, output b, inout c);
   wire d;
endinterface
```

The wires `a`, `b` and `c` can be individually connected to the interface and thus shared with other interfaces.

The following example shows how to specify an interface with inputs, allowing a wire to be shared between two instances of the interface.

```
interface simple_bus (input bit clk); // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;
endinterface: simple_bus

module memMod(simple_bus a); // Uses just the interface
   logic avail;

   always @(posedge a.clk) // the clk signal from the interface
      a.gnt <= a.req & avail; // a.req is in the 'simple_bus' interface
endmodule

module cpuMod(simple_bus b);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf1(clk); // Instantiate the interface
   simple_bus sb_intf2(clk); // Instantiate the interface

   memMod mem1(.a(sb_intf1)); // Connect bus 1 to memory 1
   cpuMod cpu1(.b(sb_intf1));
   memMod mem2(.a(sb_intf2)); // Connect bus 2 to memory 2
   cpuMod cpu2(.b(sb_intf2));

endmodule
```

Note: Because the instantiated interface names do not match the interface names used in the memMod and cpuMod modules, implicit port connections cannot be used for this example.

19.4 Modports

To bundle module ports, there are **modport** lists with directions declared within the interface. The keyword **modport** indicates that the directions are declared as if inside the module.

```
interface i2;
   wire a, b, c, d;
   modport master (input a, b, output c, d);
   modport slave (output a, b, input c, d);
endinterface
```

The **modport** list name (master or slave) can be specified in the module header, where the **modport** name acts as a direction and the interface name as a type.

```
module m (i2.master i);
   ...
endmodule

module s (i2.slave i);
   ...
endmodule

module top;
   i2 i();

   m u1(.i(i));
   s u2(.i(i));
endmodule
```

The **modport** list name (master or slave) can also be specified in the port connection with the module instance, where the **modport** name is hierarchical from the interface instance.

```
module m (i2 i);
   ...
endmodule

module s (i2 i);
   ...
endmodule

module top;
   i2 i();

   m u1(.i(i.master));
   s u2(.i(i.slave));
endmodule
```

The syntax of `interface_name.modport_name instance_name` is really a hierarchical type followed by an instance. Note that this can be generalized to any interface with a given **modport** name by writing **interface**`.modport_name instance_name`.

In a hierarchical interface, the directions in a **modport** declaration can themselves be **modport** plus name.

```
interface i1;
   interface i3;
      wire a, b, c, d;
      modport master (input a, b, output c, d);
      modport slave (output a, b, input c, d);
   endinterface
```



```
   i3 ch1(), ch2();
   modport master2 (ch1.master, ch2.master);
endinterface
```

All of the names used in a **modport** declaration shall be declared by the same interface as is the modport itself. In particular, the names used shall not be those declared by another enclosing interface, and a modport declaration shall not implicitly declare new ports.

The following interface declarations would be illegal:

```
interface i;
   wire x, y;

   interface illegal_i;
      wire a, b, c, d;
      // x, y not declared by this interface
      modport master(input a, b, x, output c, d, y);
      modport slave(input a, b, x, output c, d, y);
   endinterface : illegal_i

   illegal_i ch1, ch2;
   modport master2 (ch1.master, ch2.master);
endinterface : i

interface illegal_i;
   // a, b, c, d not declared by this interface
   modport master(input a, b, output c, d);
   modport slave(output a, b, output c, d);
endinterface : illegal_i
```

Note that if no **modport** is specified in the module header or in the port connection, then all the nets and variables in the interface are accessible with direction **inout** or **ref**, as in the examples above.

19.4.1 An example of a named port bundle

This interface example shows how to use modports to control signal directions as in port declarations. It uses the modport name in the module definition.

```
interface simple_bus (input bit clk); // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;

   modport slave (input req, addr, mode, start, clk,
                  output gnt, rdy,
                  ref data);

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  ref data);

endinterface: simple_bus

module memMod (simple_bus.slave a); // interface name and modport name
   logic avail;

   always @(posedge a.clk) // the clk signal from the interface
      a.gnt <= a.req & avail; // the gnt and req signal in the interface
```

```
endmodule

module cpuMod (simple_bus.master b);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate the interface

   initial repeat(10) #10 clk++;

   memMod mem(.a(sb_intf)); // Connect the interface to the module instance
   cpuMod cpu(.b(sb_intf));
endmodule
```

19.4.2 An example of connecting a port bundle

This interface example shows how to use modports to control signal directions. It uses the modport name in the module instantiation.

```
interface simple_bus (input bit clk); // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;


   modport slave (input req, addr, mode, start, clk,
                  output gnt, rdy,
                  ref data);

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  ref data);

endinterface: simple_bus

module memMod(simple_bus a); // Uses just the interface name
   logic avail;

   always @(posedge a.clk) // the clk signal from the interface
      a.gnt <= a.req & avail; // the gnt and req signal in the interface
endmodule

module cpuMod(simple_bus b);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate the interface

   initial repeat(10) #10 clk++;

   memMod mem(sb_intf.slave); // Connect the modport to the module instance
   cpuMod cpu(sb_intf.master);
```



```
endmodule
```

19.4.3 An example of connecting a port bundle to a generic interface

This interface example shows how to use modports to control signal directions. It shows the use of the interface keyword in the module definition. The actual interface and modport are specified in the module instantiation.

```
interface simple_bus (input bit clk); // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;

   modport slave (input req, addr, mode, start, clk,
                  output gnt, rdy,
                  ref data);

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  ref data);

endinterface: simple_bus

module memMod(interface a); // Uses just the interface
   logic avail;

   always @(posedge a.clk) // the clk signal from the interface
      a.gnt <= a.req & avail; // the gnt and req signal in the interface
endmodule

module cpuMod(interface b);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate the interface

   memMod mem(sb_intf.slave); // Connect the modport to the module instance
   cpuMod cpu(sb_intf.master);
endmodule
```

19.5 Tasks and functions in interfaces

Tasks and functions can be defined within an interface, or they can be defined within one or more of the modules connected. This allows a more abstract level of modeling. For example “read” and “write” can be defined as tasks, without reference to any wires, and the master module can merely call these tasks. In a **`modport`** these tasks are declared as **`import`** tasks.

If a module is connected to a modport containing an exported task or function, and the module does not define that task or function, then an elaboration error shall occur. Similarly if the modport contains an exported task or function prototype, and the task or function defined in the module does not exactly match that prototype, then an elaboration error shall occur.

If the tasks or functions are defined in a module, using a hierarchical name, they must also be declared as **`extern`** in the interface, or as **`export`** in a **`modport`**.

Tasks (not functions) can be defined in a module that is instantiated twice, e.g. two memories driven from the same CPU. Such multiple task definitions are allowed by a **forkjoin extern** declaration in the interface.

19.5.1 An example of using tasks in an interface

```
interface simple_bus (input bit clk); // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;

   task masterRead(input logic [7:0] raddr); // masterRead method
      // ...
   endtask: masterRead

   task slaveRead; // slaveRead method
      // ...
   endtask: slaveRead

endinterface: simple_bus

module memMod(interface a); // Uses any interface
   logic avail;

   always @(posedge a.clk) // the clk signal from the interface
      a.gnt <= a.req & avail // the gnt and req signals in the interface

   always @(a.start)
      a.slaveRead;
endmodule

module cpuMod(interface b);
   enum {read, write} instr;
   logic [7:0] raddr;

   always @(posedge b.clk)
      if (instr == read)
         b.masterRead(raddr); // call the Interface method
      ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate the interface

   memMod mem(sb_intf);
   cpuMod cpu(sb_intf);
endmodule
```

A function prototype specifies the types and directions of the arguments and the return value of a function which is defined elsewhere. Similarly, a task prototype specifies the types and directions of the arguments of a task which is defined elsewhere. In a modport, the import and export constructs can either use task or function prototypes or use just the identifiers. The only exception is when a modport is used to import a function or task from another module, in which case a full prototype shall be used.

The argument types in a prototype must match the argument types in the function or task declaration. The rules for matching are like those in C. The types must be exactly the same, or defined as being the same by a **typedef** declaration, or a series of **typedef** declarations. Two structure declarations containing the same members

are not considered to be the same type.

19.5.2 An example of using tasks in modports

This interface example shows how to use modports to control signal directions and task access in a full read/write interface.

```
interface simple_bus (input bit clk); // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;

   modport slave (input req, addr, mode, start, clk,
                  output gnt, rdy,
                  ref data,
                  import task slaveRead(),
                         task slaveWrite());
            // import into module that uses the modport

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  ref data,
                  import masterRead,
                         masterWrite);
            // import into module that uses the modport

   task masterRead(input logic [7:0] raddr); // masterRead method
      // ...
   endtask

   task slaveRead; // slaveRead method
      // ...
   endtask

   task masterWrite(input logic [7:0] waddr);
      //...
   endtask

   task slaveWrite;
      //...
   endtask

endinterface: simple_bus

module memMod(interface a); // Uses just the interface
   logic avail;

   always @(posedge a.clk) // the clk signal from the interface
      a.gnt <= a.req & avail; // the gnt and req signals in the interface

   always @(a.start)
      if (a.mode[0] == 1'b0)
         a.slaveRead;
      else
         a.slaveWrite;
endmodule

module cpuMod(interface b);
```

```
   enum {read, write} instr = $rand();
   logic [7:0] raddr = $rand();

   always @(posedge b.clk)
      if (instr == read)
         b.masterRead(raddr); // call the Interface method
      // ...
      else
         b.masterWrite(raddr);
endmodule

module omniMod( interface b);
   //...
endmodule: omniMod

module top;
   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate the interface

   memMod mem(sb_intf.slave); // only has access to the slave tasks
   cpuMod cpu(sb_intf.master); // only has access to the master tasks
   omniMod omni(sb_intf); // has access to all master and slave tasks
endmodule
```

19.5.3 An example of exporting tasks and functions

This interface example shows how to define tasks in one module and call them in another, using modports to control task access.

```
interface simple_bus (input bit clk); // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;

   modport slave( input req, addr, mode, start, clk,
                  output gnt, rdy,
                  ref data,
                  export task Read(),
                         task Write());
           // export from module that uses the modport

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  ref data,
                  import task Read(input logic [7:0] raddr),
                         task Write(input logic [7:0] waddr));
           // import requires the full task prototype

endinterface: simple_bus

module memMod(interface a); // Uses just the interface keyword
   logic avail;

   task a.Read; // Read method
      avail = 0;
      ...
      avail = 1;
```



```
   endtask

   task a.Write;
      avail = 0;
      ...
      avail = 1;
   endtask
endmodule

module cpuMod(interface b);
   enum {read, write} instr;
   logic [7:0] raddr;

   always @(posedge b.clk)
      if (instr == read)
         b.Read(raddr); // call the slave method via the interface
         ...
      else
         b.Write(raddr);
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate the interface

   memMod mem(sb_intf.slave); // exports the Read and Write tasks
   cpuMod cpu(sb_intf.master); // imports the Read and Write tasks
endmodule
```

19.5.4 An example of multiple task exports

It is normally an error for more than one module to export the same task name. However, several instances of the same modport type can be connected to an interface, such as memory modules in the previous example. So that these can still export their read and write tasks, the tasks must be declared in the interface using the **`extern forkjoin`** keywords.

The call to `extern forkjoin task countslaves( )`; in the example below behaves as:

```
fork
   top.mem1.a.countslaves;
   top.mem2.a.countslaves;
join
```

For a read task, only one module should actively respond to the task call, e.g. the one containing the appropriate address. The tasks in the other modules should return with no effect. Only then should the active task write to the result variables.

Note multiple export of functions is not allowed, because they must always write to the result.

The effect of a **`disable`** on an extern forkjoin task is as follows:

— If the task is referenced via the interface instance, all task calls shall be disabled.

— If the task is referenced via the module instance, only the task call to that module instance shall be disabled.

— If an interface contains an extern forkjoin task, and no module connected to that interface defines the task, then any call to that task shall report a run-time error and return immediately with no effect.

This interface example shows how to define tasks in more than one module and call them in another using **extern forkjoin**. The multiple task export mechanism can also be used to count the instances of a particular modport that are connected to each interface instance.

```
interface simple_bus (input bit clk); // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;
   int slaves = 0;

   // tasks executed concurrently as a fork/join block
   extern forkjoin task countSlaves();
   extern forkjoin task Read (input logic [7:0] raddr);
   extern forkjoin task Write (input logic [7:0] waddr);

   modport slave (input req,addr, mode, start, clk,
                  output gnt, rdy,
                  ref data, slaves,
                  export Read, Write, countSlaves);
            // export from module that uses the modport

   modport master (  input gnt, rdy, clk,
                     output req, addr, mode, start,
                     ref data,
                     import task Read(input logic [7:0] raddr),
                     task Write(input logic [7:0] waddr));
            // import requires the full task prototype

   initial begin
      slaves = 0;
      countSlaves;
      $display ("number of slaves = %d", slaves);
   end

endinterface: simple_bus

module memMod #(parameter int minaddr=0, maxaddr=0;) (interface a);
   logic avail = 1;
   logic [7:0] mem[255:0];

   task a.countSlaves();
      a.slaves++;
   endtask

   task a.Read(input logic [7:0] raddr); // Read method
      if (raddr >= minaddr && raddr <= maxaddr) begin
         avail = 0;
         #10 a.data = mem[raddr];
         avail = 1;
      end
   endtask

   task a.Write(input logic [7:0] waddr); // Write method
      if (waddr >= minaddr && waddr <= maxaddr) begin
         avail = 0;
         #10 mem[waddr] = a.data;
         avail = 1;
      end
```



```
   endtask
endmodule

module cpuMod(interface b);
   typedef enum {read, write} instr;
   instr inst;
   logic [7:0] raddr;
   integer seed;

   always @(posedge b.clk) begin
      inst = instr'($dist_uniform(seed, 0, 1));
      raddr = $dist_uniform(seed, 0, 3);
      if (inst == read) begin
         $display("%t begin read %h @ %h", $time, b.data, raddr);
         callr:b.Read(raddr);
         $display("%t end read %h @ %h", $time, b.data, raddr);
      end
      else begin
         $display("%t begin write %h @ %h", $time, b.data, raddr);
         b.data = raddr;
         callw:b.Write(raddr);
         $display("%t end write %h @ %h", $time, b.data, raddr);
      end
   end
endmodule

module top;
   logic clk = 0;

   function void interrupt();
      disable mem1.a.Read; // task via module instance
      disable sb_intf.Write; // task via interface instance
      if (mem1.avail == 0) $display ("mem1 was interrupted");
      if (mem2.avail == 0) $display ("mem2 was interrupted");
   endfunction

   always #5 clk++;

   initial begin
      #28 interrupt();
      #10 interrupt();
      #100 $finish;
   end

   simple_bus sb_intf(clk);

   memMod #(0, 127) mem1(sb_intf.slave);
   memMod #(128, 255) mem2(sb_intf.slave);
   cpuMod cpu(sb_intf.master);
endmodule
```

19.6 Parameterized interfaces

Interface definitions can take advantage of parameters and parameter redefinition, in the same manner as module definitions. This example shows how to use parameters in interface definitions.

```
interface simple_bus #(parameter AWIDTH = 8, DWIDTH = 8;)
                      (input bit clk); // Define the interface
```

```
      logic req, gnt;
      logic [AWIDTH-1:0] addr;
      logic [DWIDTH-1:0] data;
      logic [1:0] mode;
      logic start, rdy;

      modport slave( input req, addr, mode, start, clk,
                     output gnt, rdy,
                     ref data,
                     import task slaveRead(),
                            task slaveWrite());
            // import into module that uses the modport

      modport master(input gnt, rdy, clk,
                     output req, addr, mode, start,
                     ref data,
                     import task masterRead(input logic [AWIDTH-1:0] raddr),
                            task masterWrite(input logic [AWIDTH-1:0] waddr));
            // import requires the full task prototype

      task masterRead(input logic [AWIDTH-1:0] raddr); // masterRead method
         ...
      endtask

      task slaveRead; // slaveRead method
         ...
      endtask

      task masterWrite(input logic [AWIDTH-1:0] waddr);
         ...
      endtask

      task slaveWrite;
         ...
      endtask

   endinterface: simple_bus

   module memMod(interface a); // Uses just the interface keyword
      logic avail;

      always @(posedge b.clk) // the clk signal from the interface
         a.gnt <= a.req & avail; //the gnt and req signals in the interface

      always @(b.start)
         if (a.mode[0] == 1'b0)
            a.slaveRead;
         else
            a.slaveWrite;
   endmodule

   module cpuMod(interface b);
      enum {read, write} instr;
      logic [7:0] raddr;

      always @(posedge b.clk)
         if (instr == read)
            b.masterRead(raddr); // call the Interface method
            // ...
```



```
      else
         b.masterWrite(raddr);
endmodule

module top;

   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate default interface
   simple_bus #(.DWIDTH(16)) wide_intf(clk); // Interface with 16-bit data

   initial repeat(10) #10 clk++;

   memMod mem(sb_intf.slave); // only has access to the slaveRead task
   cpuMod cpu(sb_intf.master); // only has access to the masterRead task

   memMod memW(wide_intf.slave); // 16-bit wide memory
   cpuMod cpuW(wide_intf.master); // 16-bit wide cpu
endmodule
```

19.7 Access without ports

In addition to interfaces being used to connect two or more modules, the interface object/method paradigm allows for interfaces to be instantiated directly as static data objects within a module. If the methods are used to access internal state information about the interface, then these methods can be called from different points in the design to share information.

```
interface intf_mutex;

   task lock ();
      ...
   endtask

   function unlock();
      ...
   endfunction
endinterface

function int f(input int i);
   return(i); // just returns arg
endfunction

function int g(input int i);
   return(i); // just returns arg
endfunction

module mod1(input int in, output int out);

   intf_mutex mutex();

   always begin
      #10 mutex.lock();
      @(in) out = f(in);
      mutex.unlock;
   end

   always begin
      #10 mutex.lock();
```

```
        @(in) out = g(in);
        mutex.unlock;
    end
endmodule
```


Section 20
Parameters

20.1 Introduction (informative)

Verilog-2001 provides three constructs for defining compile time constants: the **parameter**, **localparam** and **specparam** statements.

The language provides four methods for setting the value of parameter constants in a design. Each parameter must be assigned a default value when declared. The default value of a parameter of an instantiated module can be overridden in each instance of the module using one of the following:

— Implicit in-line parameter redefinition (e.g. `foo #(value, value) u1 (...);`)

— Explicit in-line parameter redefinition (e.g. `foo #(.name(value), .name(value)) u1 (...);`)

— **defparam** statements, using hierarchical path names to redefine each parameter

20.1.1 Defparam removal

The **defparam** statement might be removed from future versions of the language. See Section 25.2.

20.2 Parameter declaration syntax

```
local_parameter_declaration ::=                                   // from Annex A.2.1.1
        localparam [ signing ] { packed_dimension } [ range ] list_of_param_assignments ;
      | localparam data_type  list_of_param_assignments ;
parameter_declaration ::=
        parameter [ signing ] { packed_dimension } [ range ] list_of_param_assignments
      | parameter data_type  list_of_param_assignments
      | parameter type  list_of_type_assignments
specparam_declaration ::=
        specparam [ range ] list_of_specparam_assignments ;
constant_declaration ::= const data_type const_assignment ;       // from Annex A.2.1.3
list_of_param_assignments ::= param_assignment { , param_assignment }   // from Annex A.2.3
list_of_specparam_assignments ::= specparam_assignment { , specparam_assignment }
list_of_type_assignments ::= type_assignment { , type_assignment }
const_assignment ::= const_identifier = constant_expression       // from Annex A.2.4
param_assignment ::= parameter_identifier = constant_param_expression
specparam_assignment ::=
        specparam_identifier = constant_mintypmax_expression
      | pulse_control_specparam
type_assignment ::= type_identifier = data_type
```

Syntax 20-1—Parameter declaration syntax (excerpt from Annex A)

A module or an interface can have parameters, which are set during elaboration and are constant during simulation. They are defined with data types and default values. With SystemVerilog, if no data type is supplied, parameters default to type **logic** of arbitrary size for Verilog-2001 compatibility and interoperability.

SystemVerilog adds the ability for a parameter to also specify a data type, allowing modules or instances to have data whose type is set for each instance.

```
module ma   #( parameter p1 = 1; parameter type p2 = shortint; )
            (input logic [p1:0] i, output logic [p1:0] o);
   p2 j = 0; // type of j is set by a parameter, (shortint unless redefined)
   always @(i) begin
      o = i;
      j++;
   end
endmodule

module mb;
   logic [3:0] i,o;
   ma #(.p1(3), .p2(int)) u1(i,o); //redefines p2 to a type of int
endmodule
```


Section 21
Configuration Libraries

21.1 Introduction (informative)

Verilog-2001 provides the ability to specify design configurations, which specify the binding information of module instances to specific Verilog HDL source code. Configurations utilize ***libraries***. A library is a collection of modules, primitives and other configurations. Separate ***library map files*** specify the source code location for the cells contained within the libraries. The names of the library map files is typically specified as invocation options to simulators or other software tools reading in Verilog source code.

SystemVerilog adds support for interfaces to Verilog configurations. SystemVerilog also provides an alternate method for specifying the names of library map files.

21.2 Libraries

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

21.3 Library map files

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

Section 22
System Tasks and System Functions

22.1 Introduction (informative)

SystemVerilog adds several system tasks and system functions as described in the following sections.

In addition, SystemVerilog extends the behavior of the following several Verilog-2001 system tasks, as described in Section 22.11.

22.2 Expression size system function

```
size_function ::= // not in Annex A
        $bits ( expression )
```

Syntax 22-1—Size function syntax (not in Annex A)

The `$bits` system function returns the number of bits required to hold a value. A 4 state value counts as one bit. Given the declaration:

```
logic [31:0] foo;
```

Then `$bits(foo)` shall return 32, even if a software tool uses more than 32-bits of storage to represent the 4-state values.

22.3 Shortreal conversions

Verilog 2001 defines a **`real`** data type, and the system functions `$realtobits` and `$bitstoreal` to permit exact bit pattern transfers between a **`real`** and a 64 bit vector. SystemVerilog adds the **`shortreal`** type, and in a parallel manner, `$shortrealtobits` and `$bitstoshortreal` are defined to permit exact bit transfers between a shortreal and a 32 bit vector.

```
[31:0] $shortrealtobits(shortreal_val) ;
shortreal $bitstoshortreal(bit_val) ;
```

`$shortrealtobits` converts from a **`shortreal`** number to the 32-bit representation (vector) of that short-real number. `$bitstoshortreal` is the reverse of `$shortrealtobits`; it converts from the bit pattern to a **`shortreal`** number.

22.4 Array querying system functions

```
array_query_functions ::= // not in Annex A
        array_dimension_function ( array_identifier , dimension_expression )
      | $dimensions ( array_identifier )
array_dimension_function ::=
        $left
      | $right
      | $low
      | $high
      | $increment
      | $length
dimension_expression ::= expression
```

Syntax 22-2—Array querying function syntax (not in Annex A)

SystemVerilog provides new system functions to return information about an array

— `$left` shall return the left bound (msb) of the dimension

— `$right` shall return the right bound (lsb) of the dimension

— `$low` shall return the minimum of `$left` and `$right` of the dimension

— `$high` shall return the maximum of `$left` and `$right` of the dimension

— `$increment` shall return 1 if `$left` is greater than or equal to `$right`, and -1 if `$left` is less than `$right`

— `$length` shall return the number of elements in the dimension, which is equivalent to `$high - $low + 1`

— `$dimensions` shall return the number of dimensions in the array, or 0 for a singular object

The dimensions of an array shall be numbered as follows: The slowest varying dimension (packed or unpacked) is dimension 1. Successively faster varying dimensions have sequentially higher dimension numbers. For instance:

```
//      Dimension numbers
//    3     4        1     2
reg [3:0][2:1] n [1:5][2:8];
```

For an integer or bit type, only dimension 1 is defined. For an integer N declared without a range specifier, its bounds are assumed to be `[$bits(N)-1:0]`.

If an out-of-range dimension is specified, these functions shall return a logic X.

22.5 Assertion severity system tasks

```
assert_severity_tasks ::= // not in Annex A
                fatal_message_task
              | nonfatal_message_task
fatal_message_task ::=
                $fatal ;
              | $fatal ( finish_number [ , message_argument { , message_argument] } ) ;
nonfatal_message_task ::=
                severity_task ;
              | severity_task ( [ message_argument { , message_argument] } ) ;
severity_task ::= $error | $warning | $info
finish_number ::= 0 | 1 | 2
message_argument ::= string | expression
```

Syntax 22-3—Assertion severity system task syntax (not in Annex A)

SystemVerilog assertions have a severity level associated with any assertion failures detected. By default, the severity of an assertion failure is “error”. The severity levels can be specified by including one of the following severity system tasks in the assertion fail statement:

— `$fatal` shall generate a run-time fatal assertion error, which terminates the simulation with an error code. The first argument passed to `$fatal` shall be consistent with the corresponding argument to the Verilog `$finish` system task, which sets the level of diagnostic information reported by the tool.

— `$error` shall be a run-time error.

— `$warning` shall be a run-time warning, which can be suppressed in a tool-specific manner.

— `$info` shall indicate that the assertion failure carries no specific severity.

All of these severity system tasks shall print a tool-specific message, indicating the severity of the failure, and specific information about the failure, which shall include the following information:

— The file name and line number of the assertion statement,

— The hierarchical name of the assertion, if it is labeled, or the scope of the assertion if it is not labeled.

For simulation tools, these tasks shall also report the simulation run-time at which the severity system task is called.

Each of the severity tasks can include optional user-defined information to be reported. The user-defined message shall use the same syntax as the Verilog **`$display`** system task, and can include any number of arguments.

22.6 Assertion control system tasks

```
assert_control_tasks ::= // not in Annex A
              assert_task ;
            | assert_task ( levels [ , list_of_modules_or_assertions ] ) ;
assert_task ::=
              $asserton
            | $assertoff
            | $assertkill
list_of_modules_or_assertions ::=
              module_or_assertion { , module_or_assertion }
module_or_assertion ::=
              module_identifier
            | assertion_identifier
            | hierarchical_identifier
```

Syntax 22-4—Assertion control syntax (not in Annex A)

SystemVerilog provides three system tasks to control assertions.

— `$assertoff` shall stop the checking of all specified assertions until a subsequent `$asserton`. An assertion that is already executing, including execution of the pass or fail statement, is not affected

— `$assertkill` shall abort execution of any currently executing specified assertions and then stop the checking of all specified assertions until a subsequent `$asserton`.

— `$asserton` shall re-enable the execution of all specified assertions

22.7 Assertion system functions

```
assert_boolean_functions ::= // not in Annex A
              assert_function ( expression ) ;
            | $insetz ( expression, expression [ { , expression } ] ) ;
assert_function ::=
              $onehot
            | $onehot0
            | $inset
            | $isunknown
```

Syntax 22-5—Assertion system function syntax (not in Annex A)

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is “one-hot”. The following system functions are included to facilitate such common assertion functionality:

— `$onehot` returns true if one and only one bit of expression is high.

— `$onehot0` returns true if at most one bit of expression is high.

— `$inset` returns true if the first expression is equal to at least one of the subsequent expression arguments.

— `$insetz` returns true if the first expression is equal to at least one other expression argument. Comparison is performed using **`casez`** semantics, so `z` or `?` bits are treated as don’t-cares.

— `$isunknown` returns true if any bit of the expression is `x`. This is equivalent to `^expression === 'bx`.

All of the above system functions shall have a return type of **`bit`**. A return value of `1'b1` shall indicate true, and a return value of `1'b0` shall indicate false.

Three functions are provided for assertions to detect changes in values between two adjacent clock ticks.

```
$rose ( expression )

$fell ( expression )

$stable ( expression )
```

These functions are discussed in Section 17.7.3.

The past values can be accessed with the `$past` function.

```
$past ( expression [ , number_of_ticks] )
```

The number of 1s in a bit vector expression can be determined with the `$countones` function.

```
$countones ( expression)
```

`$past` and `$countones` are discussed in Section 17.9.

22.8 Random number system functions

To supplement the Verilog `$random` system function, SystemVerilog provides three special system functions for generating pseudorandom numbers, `$urandom`, `$urandom_range` and `$srandom`. These system functions are presented in Section 12.10.

22.9 Program control

In addition to the normal simulation control tasks (`$stop` and `$finish`), a program can use the `$exit` control task. When all programs exit, the simulation finishes. The usage of `$exit` is presented in Section 16.6 on program blocks.

22.10 Coverage system functions

SystemVerilog has several built-in system functions for obtaining test coverage information: `$coverage_control`, `$coverage_get_max`, `$coverage_get`, `$coverage_merge` and `$coverage_save`. The coverage system functions are described in Section 28.2.

22.11 Enhancements to Verilog-2001 system tasks

SystemVerilog adds system tasks and system functions as described in the following sections. In addition, SystemVerilog extends the behavior of the following:

— **`%u`** and **`%z`** format specifiers:

  — For packed data, **`%u`** and %z are defined to operate as though the operation were applied to the equivalent vector.

  — For unpacked struct data, **`%u`** and **`%z`** are defined to apply as though the operation were performed on each member in declaration order.

- For unpacked union data, **%u** and **%z** are defined to apply as though the operation were performed on the first member in declaration order.
- **%u** and **%z** are not defined on unpacked arrays.
- The count of data items read by a **%u** or **%z** for an aggregate type is always either 1 or 0; the individual members are not counted separately.

— `$fread`

`$fread` has two variants—a register variant and a set of three memory variants.

The register variant,

```
$fread(myreg, fd);
```

- is defined to be the one applied for all packed data.
- For unpacked struct data, `$fread` is defined to apply as though the operation were performed on each member in declaration order.
- For unpacked union data, `$fread` is defined to apply as though the operation were performed on the first member in declaration order.
- For unpacked arrays, the original definition applies except that unpacked struct or union elements are read as described above.

22.12 $readmemb and $readmemh

`$readmemb` and `$readmemh` are extended to unpacked arrays of packed data. In such cases, they treat each packed element as the vector equivalent and perform the normal operation. `$readmemb` and `$readmemh` are not defined for packed arrays or unpacked arrays of unpacked data.

Section 23
VCD Data

SystemVerilog does not extend the VCD format. Some SystemVerilog types can be dumped into a standard VCD file by masquerading as a Verilog type. The following table lists the basic SystemVerilog types and their mapping to a Verilog type for VCD dumping.

Table 23-1: VCD type mapping

SystemVerilog	Verilog	Size
`bit`	`reg`	Size of packed dimension
`logic`	`reg`	Size of packed dimension
`int`	`integer`	32
`shortint`	`integer`	16
`longint`	`integer`	64
`shortreal`	`real`	
`byte`	`reg`	8
`enum`	`integer`	32

Packed arrays and structures are dumped as a single vector of **`reg`**. Multiple packed array dimensions are collapsed into a single dimension.

If an **`enum`** declaration specified a type, it is dumped as that type rather than the default shown above.

Unpacked structures appear as named **`fork`**...**`join`** blocks, and their member elements of the structure appear as the types above. Since named **`fork`**...**`join`** blocks with variable declarations are seldom used in testbenches and hardware models, this makes structures easy to distinguish from variables declared in **`begin`**...**`end`** blocks, which are more frequently used in testbenches and models.

As in Verilog 2001, unpacked arrays and automatic variables are not dumped.

Note that the current VCD format does not indicate whether a variable has been declared as **`signed`** or **`unsigned`**.

Section 24
Compiler Directives

24.1 Introduction (informative)

Verilog provides the **`‘define`** text substitution macro compiler directive. A macro can contain arguments, whose values can be set for each instance of the macro. For example:

```
‘define NAND(dval) nand #(dval)

‘NAND(3)       i1 (y, a, b); //‘NAND(3) macro substitutes with: nand #(3)

‘NAND(3:4:5)   i2 (o, c, d); //‘NAND(3:4:5) macro substitutes with: nand
#(3:4:5)
```

SystemVerilog enhances the capabilities of the `‘define` compiler directive to support the construction of string literals and identifiers.

Verilog provides the `` `include `` file inclusion compiler directive. SystemVerilog enhances the capabilities to support standard include specification, and enhances the `` `include `` directive to accept a file name constructed with a macro.

24.2 ‘define macros

In Verilog, the `‘define` macro text can include a backslash (`\`) at the end of a line to show continuation on the next line.

In SystemVerilog, the macro text can also include **`‘"`**, **`‘\‘"`** and **`‘‘`**.

An `` `" `` overrides the usual lexical meaning of **`"`**, and indicates that the expansion should include an actual quotation mark. This allows string literals to be constructed from macro arguments.

A **`‘\‘"`** indicates that the expansion should include the escape sequence `\"`, e.g.

```
`define msg(x,y) `"x: `\`"y`\`"`"
```

This expands:

```
$display(`msg(left side,right side));
```

to:

```
$display("left side: \"right side\"");
```

A **`‘‘`** delimits lexical tokens without introducing white space, allowing identifiers to be constructed from arguments, e.g.

```
`define foo(f) f``_suffix
```

This expands:

```
‘foo(bar)
```

to:

```
bar_suffix
```

The `‘include` directive can be followed by a macro, instead of a literal string:

```
‘define home(filename) ‘"/home/foo/myfile‘"
‘include ‘home(myfile)
```

24.3 ‘include

The syntax of the `‘include` compiler directive is:

```
include_compiler_directive ::=
    ‘include "filename"
  | ‘include <filename>
```

When the `filename` is an absolute path, only that `filename` is included and only the double quote form of the `‘include` can be used.

When the double quote (`"filename"`) version is used, the behavior of `‘include` is unchanged from IEEE Std. 1364-2001.

When the angle bracket (`<filename>`) notation is used, then only the vendor defined location containing files defined by the language standard is searched. Relative path names given inside the **< >** are interpreted relative to the vendor-defined location in all cases.

Section 25
Features under consideration for removal from SystemVerilog

25.1 Introduction (informative)

Certain Verilog language features can be simulation inefficient, easily abused, and the source of design problems. These features are being considered for removal from the SystemVerilog language, if there is an alternate method for these features.

The Verilog language features that have been identified in this standard as ones which can be removed from Verilog are **`defparam`** and procedural **`assign`**/**`deassign`**.

25.2 Defparam statements

The SystemVerilog committee has determined, based on the solicitation of input from tool implementers and tools users, that the **`defparam`** method of specifying the value of a parameter can be a source of design errors, and can be an impediment to tool implementation. The **`defparam`** statement does not provide a capability that can not be done by another method, which avoids these problems. Therefore, the committee has placed the **`defparam`** statement on a deprecation list. This means is that a future revision of the Verilog standard might not require support for this feature. This current standard still requires tools to support the **`defparam`** statement. However, users are strongly encouraged to migrate their code to use one of the alternate methods of parameter redefinition.

Prior to the acceptance of the Verilog-2001 Standard, it was common practice to change one or more parameters of instantiated modules using a separate defparam statement. Defparam statements can be a source of tool complexity and design problems.

A **`defparam`** statement can precede the instance to be modified, can follow the instance to be modified, can be at the end of the file that contains the instance to be modified, can be in a separate file from the instance to be modified, can modify parameters hierarchically that in turn must again be passed to other **`defparam`** statements to modify, and can modify the same parameter from two different **`defparam`** statements (with undefined results). Due to the many ways that a **`defparam`** can modify parameters, a Verilog compiler cannot insure the final parameter values for an instance until after all of the design files are compiled.

Prior to Verilog-2001, the only other method available to change the values of parameters on instantiated modules was to use implicit in-line parameter redefinition. This method uses `#(parameter_value)` as part of the module instantiation. Implicit in-line parameter redefinition syntax requires that all parameters up to and including the parameter to be changed must be placed in the correct order, and must be assigned values.

Verilog-2001 introduced explicit in-line parameter redefinition, in the form `#(.parameter_name(value))`, as part of the module instantiation. This method gives the capability to pass parameters by name in the instantiation, which supplies all of the necessary parameter information to the model in the instantiation itself.

The practice of using **`defparam`** statements is highly discouraged. Engineers are encouraged to take advantage of the Verilog-2001 explicit in-line parameter redefinition capability.

See Section 20 for more details on parameters.

25.3 Procedural assign and deassign statements

The SystemVerilog committee has determined, based on the solicitation of input from tool implementers and tools users, that the procedural **`assign`** and **`deassign`** statements can be a source of design errors, and can be an impediment to tool implementation. The procedural **`assign`**/**`deassign`** statements do not provide a capability that can not be done by another method, which avoids these problems. Therefore, the committee has placed the procedural **`assign`**/**`deassign`** statements on a deprecation list. This means that a future revision of the Verilog standard might not require support for theses statements. This current standard still requires tools to

support the procedural **assign**/**deassign** statements. However, users are strongly encouraged to migrate their code to use one of the alternate methods of procedural or continuous assignments.

Verilog has two forms of the **assign** statement:

— Continuous assignments, placed outside of any procedures

— Procedural continuous assignments, placed within a procedure

Continuous assignment statements are a separate process that are active throughout simulation. The continuous assignment statement accurately represents combinational logic at an RTL level of modeling, and is frequently used.

Procedural continuous assignment statements become active when the **assign** statement is executed in the procedure. The process can be de-activated using a **deassign** statement. The procedural **assign**/**deassign** statements are seldom needed to model hardware behavior. In the unusual circumstances where the behavior of procedural continuous assignments are required, the same behavior can be modeled using the procedural force and release statements.

The fact that the **assign** statement to be used both outside and inside a procedure can cause confusion and errors in Verilog models. The practice of using the **assign** and **deassign** statements inside of procedural blocks is highly discouraged.

See Section 8 for more information on procedural assignments.

Section 26
Direct Programming Interface (DPI)

This chapter highlights the Direct Programming Interface and provides a detailed description of the SystemVerilog layer of the interface. The C layer is defined in Annex D.

26.1 Overview

Direct Programming Interface (DPI) is an interface between SystemVerilog and a foreign programming language. It consists of two separate layers: the SystemVerilog layer and a foreign language layer. Both sides of DPI are fully isolated. Which programming language is actually used as the foreign language is transparent and irrelevant for the SystemVerilog side of this interface. Neither the SystemVerilog compiler nor the foreign language compiler is required to analyze the source code in the other's language. Different programming languages can be used and supported with the same intact SystemVerilog layer. For now, however, SystemVerilog 3.1 defines a foreign language layer only for the C programming language. See Annex D for more details.

The motivation for this interface is two-fold. The methodological requirement is that the interface should allow a heterogeneous system to be built (a design or a testbench) in which some components can be written in a language (or more languages) other than SystemVerilog, hereinafter called the foreign language. On the other hand, there is also a practical need for an easy and efficient way to connect existing code, usually written in C or C++, without the knowledge and the overhead of PLI or VPI.

DPI follows the principle of a black box: the specification and the implementation of a component is clearly separated and the actual implementation is transparent to the rest of the system. Therefore, the actual programming language of the implementation is also transparent, though this standard defines only C linkage semantics. The separation between SystemVerilog code and the foreign language is based on using functions as the natural encapsulation unit in SystemVerilog. By and large, any function can be treated as a black box and implemented either in SystemVerilog or in the foreign language in a transparent way, without changing its calls.

26.1.1 Functions

DPI allows direct inter-language function calls between the languages on either side of the interface. Specifically, functions implemented in a foreign language can be called from SystemVerilog; such functions are referred to as *imported functions*. SystemVerilog functions that are to be called from a foreign code shall be specified in export declarations (see Section 26.6 for more details). DPI allows for passing SystemVerilog data between the two domains through function arguments and results. There is no intrinsic overhead in this interface.

All functions used in DPI are assumed to complete their execution instantly and consume 0 (zero) simulation time, just as normal SystemVerilog functions. DPI provides no means of synchronization other than by data exchange and explicit transfer of control.

Every imported function needs to be declared. A declaration of an imported function is referred to as an *import declaration*. Import declarations are very similar to SystemVerilog function declarations. Import declarations can occur anywhere where SystemVerilog function definitions are permitted. An import declaration is considered to be a definition of a SystemVerilog function with a foreign language implementation. The same foreign function can be used to implement multiple SystemVerilog functions (this can be a useful way of providing differing default argument values for the same basic function), but a given SystemVerilog name can only be defined once per scope. Imported functions can have zero or more formal **`input`**, **`output`**, and **`inout`** arguments, and they can return a result or be defined as void functions.

DPI is based entirely upon SystemVerilog constructs. The usage of imported functions is identical as for native SystemVerilog functions. With few exceptions imported functions and native functions are mutually exchangeable. Calls of imported functions are indistinguishable from calls of SystemVerilog functions. This facilitates ease-of-use and minimizes the learning curve.

26.1.2 Data types

SystemVerilog data types are the sole data types that can cross the boundary between SystemVerilog and a foreign language in either direction (i.e., when an imported function is called from SystemVerilog code or an exported SystemVerilog function is called from a foreign code). It is not possible to import the data types or directly use the type syntax from another language. A rich subset of SystemVerilog data types is allowed for formal arguments of import and export functions, although with some restrictions and with some notational extensions. Function result types are restricted to small values, however (see Section 26.4.5).

Formal arguments of an imported function can be specified as open arrays. A formal argument is an open array when a range of one or more of its dimensions, packed or unpacked, is unspecified. An open array is like a multi-dimensional dynamic array formal in both packed and unpacked dimensions, and is thus denoted using the same syntax as dynamic arrays, using `[]` to denote an open dimension. This is solely a relaxation of the argument-matching rules. An actual argument shall match the formal one regardless of the range(s) for its corresponding dimension(s), which facilitates writing generalized code that can handle SystemVerilog arrays of different sizes. See Section 26.4.6.1.

26.1.2.1 Data representation

DPI does not add any constraints on how SystemVerilog-specific data types are actually implemented. Optimal representation can be platform dependent. The layout of 2- or 4-state packed structures and arrays is implementation- and platform-dependent.

The implementation (representation and layout) of 4-state values, structures, and arrays is irrelevant for SystemVerilog semantics, and can only impact the foreign side of the interface.

26.2 Two layers of the DPI

DPI consists of two separate layers: the SystemVerilog layer and a foreign language layer. The SystemVerilog layer does not depend on which programming language is actually used as the foreign language. Although different programming languages can be supported and used with the intact SystemVerilog layer, SystemVerilog 3.1 defines a foreign language layer only for the C programming language. Nevertheless, SystemVerilog code shall look identical and its semantics shall be unchanged for any foreign language layer. Different foreign languages can require that the SystemVerilog implementation shall use the appropriate function call protocol, argument passing and linking mechanisms. This shall be, however, transparent to SystemVerilog users. SystemVerilog 3.1 requires only that its implementation shall support C protocols and linkage.

26.2.1 DPI SystemVerilog layer

The SystemVerilog side of DPI does not depend on the foreign programming language. In particular, the actual function call protocol and argument passing mechanisms used in the foreign language are transparent and irrelevant to SystemVerilog. SystemVerilog code shall look identical regardless of what code the foreign side of the interface is using. The semantics of the SystemVerilog side of the interface is independent from the foreign side of the interface.

This chapter does not constitute a complete interface specification. It only describes the functionality, semantics and syntax of the SystemVerilog layer of the interface. The other half of the interface, the foreign language layer, defines the actual argument passing mechanism and the methods to access (read/write) formal arguments from the foreign code. See Annex D for more details.

26.2.2 DPI foreign language layer

The foreign language layer of the interface (which is transparent to SystemVerilog) shall specify how actual arguments are passed, how they can be accessed from the foreign code, how SystemVerilog-specific data types (such as **`logic`** and **`packed`**) are represented, and how to translate them to and from some predefined C-like types.

The data types allowed for formal arguments and results of imported functions or exported functions are gen-

erally SystemVerilog types (with some restrictions and with notational extensions for open arrays). The user is responsible for specifying in their foreign code the native types equivalent to the SystemVerilog types used in imported declarations or export declarations. Software tools, like a SystemVerilog compiler, can facilitate the mapping of SystemVerilog types onto foreign native types by generating the appropriate function headers.

The SystemVerilog compiler or simulator shall generate and/or use the function call protocol and argument passing mechanisms required for the intended foreign language layer. The same SystemVerilog code (compiled accordingly) shall be usable with different foreign language layers, regardless of the data access method assumed in a specific layer. Annex A defines DPI foreign language layer for the C programming language.

26.3 Global name space of imported and exported functions

Every function imported to SystemVerilog must eventually resolve to a global symbol. Similarly, every function exported from SystemVerilog defines a global symbol. Thus the functions imported to and exported from SystemVerilog have their own global name space of linkage names, different from $root name space. Global names of imported and exported functions must be unique (no overloading is allowed) and shall follow C conventions for naming; specifically, such names must start with a letter or underscore, and can be followed by alphanumeric characters or underscores. Exported and imported functions, however, can be declared with local SystemVerilog names. Import and export declarations allow users to specify a global name for a function in addition to its declared name. Should a global name clash with a SystemVerilog keyword or a reserved name, it shall take the form of an escaped identifier. The leading backslash (\ ) character and the trailing white space shall be stripped off by the SystemVerilog tool to create the linkage identifier. Note that after this stripping, the linkage identifier so formed must comply with the normal rules for C identifier construction. If a global name is not explicitly given, it shall be the same as the SystemVerilog function name. For example:

```
export "DPI" foo_plus = function \foo+ ; // "foo+" exported as "foo_plus"
export "DPI" function foo; // "foo" exported under its own name
import "DPI" init_1 = function void \init[1] (); // "init_1" is a linkage name
import "DPI" \begin = function void \init[2] (); // "begin" is a linkage name
```

The same global function can be referred to in multiple import declarations in different scopes or/and with different SystemVerilog names, see Section 26.4.4.

Multiple export declarations are allowed with the same *c_identifier*, explicit or implicit, as long as they are in different scopes and have the same type signature (as defined in Section 26.4.4 for imported functions). Multiple export declarations with the same *c_identifier* in the same scope are forbidden.

26.4 Imported functions

The usage of imported functions is similar as for native SystemVerilog functions.

26.4.1 Required properties of imported functions - semantic constraints

This section defines the semantic constraints imposed on imported functions. Some semantic restrictions are shared by all imported functions. Other restrictions depend on whether the special properties **`pure`** (see Section 26.4.2) or **`context`** (see Section 26.4.3) are specified for an imported function. A SystemVerilog compiler is not able to verify that those restrictions are observed and if those restrictions are not satisfied, the effects of such imported function calls can be unpredictable.

26.4.1.1 Instant completion

Imported functions shall complete their execution instantly and consume zero-simulation time, similarly to native functions.

26.4.1.2 input and output arguments

Imported functions can have **`input`** and **`output`** arguments. The formal **`input`** arguments shall not be modi-

fied. If such arguments are changed within a function, the changes shall not be visible outside the function; the actual arguments shall not be changed.

The imported function shall not assume anything about the initial values of formal **output** arguments. The initial values of **output** arguments are undetermined and implementation-dependent.

26.4.1.3 Special properties pure and context

Special properties can be specified for an imported function: as **pure** or as **context** (see also Section 26.4.2 or 26.4.3).

A function whose result depends solely on the values of its input arguments and with no side effects can be specified as **pure**. This can usually allow for more optimizations and thus can result in improved simulation performance. Section 26.4.2 details the rules that must be obeyed by pure functions.

An imported function that is intended to call exported functions or to access SystemVerilog data objects other then its actual arguments (e.g. via VPI or PLI calls) must be specified as **context**. Calls of context functions are specially instrumented and can impair SystemVerilog compiler optimizations; therefore simulation performance can decrease if the **context** property is specified when not necessary. A function not specified as **context** shall not read or write any data objects from SystemVerilog other then its actual arguments. For functions not specified as **context**, the effects of calling PLI, VPI, or exported SystemVerilog functions can be unpredictable and can lead to unexpected behavior; such calls can even crash. Section 26.4.3 details the restrictions that must be obeyed by non-context functions.

26.4.1.4 Memory management

The memory spaces owned and allocated by the foreign code and SystemVerilog code are disjoined. Each side is responsible for its own allocated memory. Specifically, an imported function shall not free the memory allocated by SystemVerilog code (or the SystemVerilog compiler) nor expect SystemVerilog code to free the memory allocated by the foreign code (or the foreign compiler). This does not exclude scenarios where foreign code allocates a block of memory, then passes a handle (i.e., a pointer) to that block to SystemVerilog code, which in turn calls an imported function (e.g. C standard function `free`) which directly or indirectly frees that block.

NOTE—In this last scenario, a block of memory is allocated and freed in the foreign code, even when the standard functions `malloc` and `free` are called directly from SystemVerilog code.

26.4.2 Pure functions

A **pure** function call can be safely eliminated if its result is not needed or if the previous result for the same values of input arguments is available somehow and can be reused without needing to recalculate. Only non-void functions with no **output** or **inout** arguments can be specified as **pure**. Functions specified as **pure** shall have no side effects whatsoever; their results need to depend solely on the values of their input arguments. Calls to such functions can be removed by SystemVerilog compiler optimizations or replaced with the values previously computed for the same values of the input arguments.

Specifically, a pure function is assumed not to directly or indirectly (i.e., by calling other functions):

— perform any file operations

— read or write anything in the broadest possible meaning, includes i/o, environment variables, objects from the operating system or from the program or other processes, shared memory, sockets, etc.

— access any persistent data, like global or static variables.

If a pure function does not obey the above restrictions, SystemVerilog compiler optimizations can lead to unexpected behavior, due to eliminated calls or incorrect results being used.

26.4.3 Context functions

Some DPI imported functions require that the context of their call is known. It takes special instrumentation of

their call instances to provide such context; for example, an internal variable referring to the “current instance” might need to be set. To avoid any unnecessary overhead, imported function calls in SystemVerilog code are not instrumented unless the imported function is specified as **context**.

All DPI exported functions require that the context of their call is known. This occurs since SystemVerilog function declarations always occur in instantiable scopes, hence allowing a multiplicity of unique function instances.

For the sake of simulation performance, an imported function call shall not block SystemVerilog compiler optimizations. An imported function not specified as **context** shall not access any data objects from SystemVerilog other than its actual arguments. Only the actual arguments can be affected (read or written) by its call. Therefore, a call of a non-context function is not a barrier for optimizations. A context imported function, however, can access (read or write) any SystemVerilog data objects by calling PLI/VPI, or by calling an export function. Therefore, a call to a context function is a barrier for SystemVerilog compiler optimizations.

Only calls of context imported functions are properly instrumented and cause conservative optimizations; therefore, only those functions can safely call all functions from other APIs, including PLI and VPI functions or exported SystemVerilog functions. For imported functions not specified as **context**, the effects of calling PLI, VPI, or SystemVerilog functions can be unpredictable and such calls can crash if the callee requires a context that has not been properly set. However note that declaring an import context function does not automatically make any other simulator interface automatically available. For VPI access (or any other interface access) to be possible, the appropriate implementation defined mechanism must still be used to enable these interface(s). Note also that DPI calls do not automatically create or provide any handles or any special environment that can be needed by those other interfaces. It is the user’s responsibility to create, manage or otherwise manipulate the required handles/environment(s) needed by the other interfaces.

Context imported functions are always implicitly supplied a scope representing the fully qualified instance name within which the import declaration was present. This scope defines which exported SystemVerilog functions can be called directly from the imported function; only functions defined and exported from the same scope as the import can be called directly. To call any other exported SystemVerilog functions, the imported function shall first have to modify its current scope, in essence performing the foreign language equivalent of a SystemVerilog hierarchical function call.

Special DPI utility functions exist that allow imported functions to retrieve and operate on their scope. See Annex D for more details.

26.4.4 Import declarations

Each imported function shall be declared. Such declaration are referred to as *import declarations*. The syntax of an **import** declaration is similar to the syntax of SystemVerilog function prototypes (see Section 10.6).

Imported functions are similar to SystemVerilog functions. Imported functions can have zero or more formal **input**, **output**, and **inout** arguments. Imported functions can return a result or be defined as void functions.

```
dpi_import_export ::=                                          // from Annex A.2.6
        import "DPI" [ dpi_import_property ] [ c_identifier = ] dpi_function_proto
dpi_import_property ::= context | pure
dpi_function_proto ::=
        named_function_proto
      | [ signing ] function_data_type function_identifier ( list_of_dpi_proto_formals )
list_of_dpi_proto_formals ::=
        [ { attribute_instance } dpi_proto_formal { , { attribute_instance } dpi_proto_formal } ]
dpi_proto_formal ::=
        data_type [ port_identifier dpi_dimension { , port_identifier dpi_dimension } ]
```

Syntax 26-1—DPI import declaration syntax (excerpt from Annex A)

An import declaration specifies the function name, function result type, and types and directions of formal arguments. It can also provide optional default values for formal arguments. Formal argument names are optional unless argument passing by name is needed. An import declaration can also specify an optional function property: **`context`** or **`pure`**.

Note that an import declaration is equivalent to defining a function of that name in the SystemVerilog scope in which the import declaration occurs, and thus multiple imports of the same function name into the same scope are forbidden. Note that this declaration scope is particularly important in the case of imported context functions, see Section 26.4.3; for non-context imported functions the declaration scope has no other implications other than defining the visibility of the function.

c_identifier provides the linkage name for this function in the foreign language. If not provided, this defaults to the same identifier as the SystemVerilog function name. In either case, this linkage name must conform to C identifier syntax. An error shall occur if the *c_identifier*, either directly or indirectly, does not conform to these rules.

For any given *c_identifier* (whether explicitly defined with *c_identifier*=, or automatically determined from the function name), all declarations, regardless of scope, must have exactly the same type signature. The signature includes the return type and the number, order, direction and types of each and every argument. Type includes dimensions and bounds of any arrays or array dimensions. Signature also includes the **`pure`**/**`context`** qualifiers that can be associated with an extern definition.

Note that multiple declarations of the same imported or exported function in different scopes can vary argument names and default values, provided the type compatibility constraints are met.

A formal argument name is required to separate the packed and the unpacked dimensions of an array.

The qualifier **`ref`** cannot be used in import declarations. The actual implementation of argument passing depends solely on the foreign language layer and its implementation and shall be transparent to the SystemVerilog side of the interface.

The following are examples of external declarations.

```
import "DPI" function void myInit();

// from standard math library
import "DPI" pure function real sin(real);

// from standard C library: memory management
import "DPI" function handle malloc(int size); // standard C function
import "DPI" function void free(handle ptr); // standard C function

// abstract data structure: queue
import "DPI" function handle newQueue(input string name_of_queue);

// Note the following import uses the same foreign function for
// implementation as the prior import, but has different SystemVerilog name
// and provides a default value for the argument.
import "DPI" newQueue=function handle newAnonQueue(input string s=null);
import "DPI" function handle newElem(bit [15:0]);
import "DPI" function void enqueue(handle queue, handle elem);
import "DPI" function handle dequeue(handle queue);

// miscellanea
import "DPI" function bit [15:0] getStimulus();
import "DPI" context function void processTransaction(handle elem,
                                    output logic [64:1] arr [0:63]);
```


26.4.5 Function result

Function result types are restricted to small values. The following SystemVerilog data types are allowed for imported function results:

— **void**, **byte**, **shortint**, **int**, **longint**, **real**, **shortreal**, **handle**, and **string**.

— packed bit arrays up to 32 bits and all types that are eventually equivalent to packed bit arrays up to 32 bits.

The same restrictions apply for the result types of exported functions.

26.4.6 Types of formal arguments

A rich subset of SystemVerilog data types is allowed for formal arguments of import and export functions. Generally, C compatible types, packed types and user defined types built of types from these two categories can be used for formal arguments of DPI functions. The set of permitted types is defined inductively.

The following SystemVerilog types are the only permitted types for formal arguments of import and export functions:

— **void**, byte, **shortint**, **int**, **longint**, **real**, **shortreal**, **handle**, and **string**

— scalar values of type **bit** and **logic**

— packed one dimensional arrays of type bit and logic

  Note however, that every packed type, whatever is its structure, is eventually equivalent to a packed one dimensional array. Therefore practically all packed types are supported, although their internal structure (individual fields of structs, multiple dimensions of arrays) shall be transparent and irrelevant.

— enumeration types interpreted as the type associated with that enumeration

— types constructed from the supported types with the help of the constructs:

  — **struct**

  — unpacked array

  — **typedef**

The following caveats apply for the types permitted in DPI:

— Enumerated data types are not supported directly. Instead, an enumerated data type is interpreted as the type associated with that enumerated type.

— SystemVerilog does not specify the actual memory representation of packed structures or any arrays, packed or unpacked. Unpacked structures have an implementation-dependent packing, normally matching the C compiler.

— The actual memory representation of SystemVerilog data types is transparent for SystemVerilog semantics and irrelevant for SystemVerilog code. It can be relevant for the foreign language code on the other side of the interface, however; a particular representation of the SystemVerilog data types can be assumed. This shall not restrict the types of formal arguments of imported functions, with the exception of unpacked arrays. SystemVerilog implementation can restrict which SystemVerilog unpacked arrays are passed as actual arguments for a formal argument which is a sized array, although they can be always passed for an unsized (i.e., open) array. Therefore, the correctness of an actual argument might be implementation-dependent. Nevertheless, an open array provides an implementation-independent solution.

26.4.6.1 Open arrays

The size of the packed dimension, the unpacked dimension, or both dimensions can remain unspecified; such cases are referred to as *open arrays* (or unsized arrays). Open arrays allow the use of generic code to handle different sizes.

Formal arguments of imported functions can be specified as open arrays. (Exported SystemVerilog functions cannot have formal arguments specified as open arrays.) A formal argument is an open array when a range of one or more of its dimensions is unspecified (denoted by using square brackets (`[]`)). This is solely a relaxation of the argument-matching rules. An actual argument shall match the formal one regardless of the range(s) for its corresponding dimension(s), which facilitates writing generalized code that can handle SystemVerilog arrays of different sizes.

Although the packed part of an array can have an arbitrary number of dimensions, in the case of open arrays only a single dimension is allowed for the packed part. This is not very restrictive, however, since any packed type is eventually equivalent to one-dimensional packed array. The number of unpacked dimensions is not restricted.

If a formal argument is specified as an open array with a range of its packed or one or more of its unpacked dimensions unspecified, then the actual argument shall match the formal one—regardless of its dimensions and sizes of its linearized packed or unpacked dimensions corresponding to an unspecified range of the formal argument, respectively.

Here are examples of types of formal arguments (empty square brackets `[]` denote open array):

```
logic
bit [8:1]
bit[]
bit [7:0] array8x10 [1:10] // array8x10 is a formal arg name
logic [31:0] array32xN []  // array32xN is a formal arg name
logic [] arrayNx3 [3:1]    // arrayNx3 is a formal arg name
bit [] arrayNxN []         // arrayNxN is a formal arg name
```

Example of complete import declarations:

```
import "DPI" function void foo(input logic [127:0]);
import "DPI" function void boo(logic [127:0] i []); // open array of 128-bit
```

The following example shows the use of open arrays for different sizes of actual arguments:

```
typedef struct {int i; ... } MyType;

import "DPI" function void foo(input MyType i [][]);
      /* 2-dimensional unsized unpacked array of MyType */

MyType a_10x5 [11:20][6:2];
MyType a_64x8 [64:1][-1:-8];

foo(a_10x5);
foo(a_64x8);
```

26.5 Calling imported functions

The usage of imported functions is identical as for native SystemVerilog functions., hence the usage and syntax for calling imported functions is identical as for native SystemVerilog functions. Specifically, arguments with default values can be omitted from the call; arguments can be passed by name, if all formal arguments are named.

26.5.1 Argument passing

Argument passing for imported functions is ruled by the *WYSIWYG* principle: *What You Specify Is What You Get*, see Section 26.5.1.1. The evaluation order of formal arguments follows general SystemVerilog rules.

Argument compatibility and coercion rules are the same as for native SystemVerilog functions. If a coercion is

needed, a temporary variable is created and passed as the actual argument. For **input** and **inout** arguments, the temporary variable is initialized with the value of actual argument with the appropriate coercion; for **output** or **inout** arguments, the value of the temporary variable is assigned to the actual argument with the appropriate conversion. The assignments between a temporary and the actual argument follow general SystemVerilog rules for assignments and automatic coercion.

On the SystemVerilog side of the interface, the values of actual arguments for formal input arguments of imported functions shall not be affected by the callee; the initial values of formal output arguments of imported functions are unspecified (and can be implementation-dependent), and the necessary coercions, if any, are applied as for assignments. imported functions shall not modify the values of their input arguments.

For the SystemVerilog side of the interface, the semantics of arguments passing is as if **input** arguments are passed by *copy-in*, **output** arguments are passed by *copy-out*, and **inout** arguments were passed by *copy-in*, *copy-out*. The terms *copy-in* and *copy-out* do not impose the actual implementation; they refer only to "hypothetical assignment".

The actual implementation of argument passing is transparent to the SystemVerilog side of the interface. In particular, it is transparent to SystemVerilog whether an argument is actually passed by *value* or by *reference*. The actual argument passing mechanism is defined in the foreign language layer. See Annex D for more details.

26.5.1.1 "What You Specify Is What You Get" principle

The principle "What You Specify Is What You Get" guarantees the types of formal arguments of imported functions — an actual argument is guaranteed to be of the type specified for the formal argument, with the exception of open arrays (for which unspecified ranges are statically unknown). Formal arguments, other than open arrays, are fully defined by import declaration; they shall have ranges of packed or unpacked arrays exactly as specified in the import declaration. Only the declaration site of the imported function is relevant for such formal arguments.

Another way to state this is that no compiler (either C or SystemVerilog) can make argument coercions between a caller's declared formal and the callee's declared formals. this is because the callee's formal arguments are declared in a different language than the caller's formal arguments; hence here is no visible relationship between the two sets of formals. Users are expected to understand all argument relationships and provide properly matched types on both sides of the interface.

Formal arguments defined as open arrays have the size and ranges of the corresponding actual arguments, i.e., have the ranges of packed or unpacked arrays exactly as that of the actual argument. The unsized ranges of open arrays are determined at a call site; the rest of type information is specified at the import declaration.

So, if a formal argument is declared as `bit [15:8] b []`, then it is the import declaration which specifies the formal argument is an unpacked array of packed bit array with bounds 15 to 8, while the actual argument used at a particular call site defines the bounds for the unpacked part for that call.

26.5.2 Value changes for output and inout arguments

The SystemVerilog simulator is responsible for handling value changes for **output** and **inout** arguments. Such changes shall be detected and handled after control returns from imported functions to SystemVerilog code.

For **output** and **inout** arguments, the value propagation (i.e., value change events) happens as if an actual argument was assigned a formal argument immediately after control returns from imported functions. If there is more than one argument, the order of such assignments and the related value change propagation follows general SystemVerilog rules.

26.6 Exported functions

DPI allows calling SystemVerilog functions from another language. However, such functions must adhere to

the same restrictions on argument types and results as are imposed on imported functions. It is an error to export a function that does not satisfy such constraints.

SystemVerilog functions that can be called from foreign code need to be specified in **`export`** declarations. Export declarations are allowed to occur only in the scope in which the function being exported is defined. Only one export declaration per function is allowed in a given scope.

Note that class member functions can not be exported, but all other SystemVerilog functions can be exported.

Similar to import declarations, **`export`** declarations can define an optional *c_identifier* to be used in the foreign language when calling an exported function.

```
dpi_import_export ::=                                          // from Annex A.2.6
        | export "DPI" [ c_identifier = ] function function_identifier
```

Syntax 26-2—DPI export declaration syntax (excerpt from Annex A)

c_identifier is optional here. It defaults to *function_identifier*. For rules describing *c_identifier*, see Section 26.3. Note that all export functions are always context functions. No two functions in the same SystemVerilog scope can be exported with the same explicit or implicit *c_identifier*. The export declaration and the definition of the corresponding SystemVerilog function can occur in any order. Only one export declaration is permitted per SystemVerilog function.

Section 27
SystemVerilog Assertion API

This chapter defines the Assertion Application Programming Interface (API) in SystemVerilog.

27.1 Requirements

SystemVerilog provides assertion capabilities to enable:

— a user's C code to react to assertion events.

— third-party assertion "waveform" dumping tools to be written.

— third-party assertion coverage tools to be written.

— third-party assertion debug tools to be written.

27.1.1 Naming conventions

All elements added by this interface shall conform to the Verilog Procedural Interface (VPI) interface naming conventions.

— All names are prefixed by `vpi`.

— All *type names* shall start with `vpi`, followed by initially capitalized words with no separators, e.g., `vpiAssertCheck`.

— All callback names shall start with `cb`, followed by initially capitalized words with no separators, e.g., `cbAssertionStart`.

— All *function names* shall start with `vpi_`, followed by all lowercase words separated by underscores (_), e.g., `vpi_get_assert_info()`.

27.2 Extensions to VPI enumerations

These extensions shall be appended to the contents of the `vpi_user.h` file, described in IEEE Std. 1364-2001, Annex G. The numbers in the range 700 - 799 are reserved for the assertion portion of the VPI.

27.2.1 Object types

This section lists the object type VPI calls. The VPI reserved range for these calls is `700 - 729`.

```
#define vpiAssertion 700 /* assertion */
```

27.2.2 Object properties

This section lists the object property VPI calls. The VPI reserved range for these calls is `700 - 729`.

```
/* Assertion types */

#define vpiSequenceType       701
#define vpiAssertType         702
#define vpiCoverType          703
#define vpiPropertyType       704
#define vpiImmediateAssertType705
```

27.2.3 Callbacks

This section lists the system callbacks. The VPI reserved range for these calls is `700 - 719`.

1) Assertion

```
#define cbAssertionStart         700
#define cbAssertionSuccess       701
#define cbAssertionFailure       702
#define cbAssertionStepSuccess   703
#define cbAssertionStepFailure   704
#define cbAssertionDisable       705
#define cbAssertionEnable        706
#define cbAssertionReset         707
#define cbAssertionKill          708
```

2) “Assertion system”

```
#define cbAssertionSysInitialized709
#define cbAssertionSysStart      710
#define cbAssertionSysStop       711
#define cbAssertionSysEnd        712
#define cbAssertionSysReset      713
```

27.2.4 Control constants

This section lists the system control constant callbacks. The VPI reserved range for these calls is `730 - 759`.

1) Assertion

```
#define vpiAssertionDisable      730
#define vpiAssertionEnable       731
#define vpiAssertionReset        732
#define vpiAssertionKill         733
#define vpiAssertionEnableStep   734
#define vpiAssertionDisableStep  735
```

2) Assertion stepping

```
#define vpiAssertionClockSteps   736
```

3) “Assertion system”

```
#define vpiAssertionSysStart     737
#define vpiAssertionSysStop      738
#define vpiAssertionSysEnd       739
#define vpiAssertionSysReset     740
```

27.3 Static information

This section defines how to obtain assertion handles and other static assertion information.

27.3.1 Obtaining assertion handles

SystemVerilog extends the VPI module iterator model (i.e., the instance) to encompass assertions, as shown in Figure 27-1.

The following steps highlight how to obtain the assertion handles for named assertions.

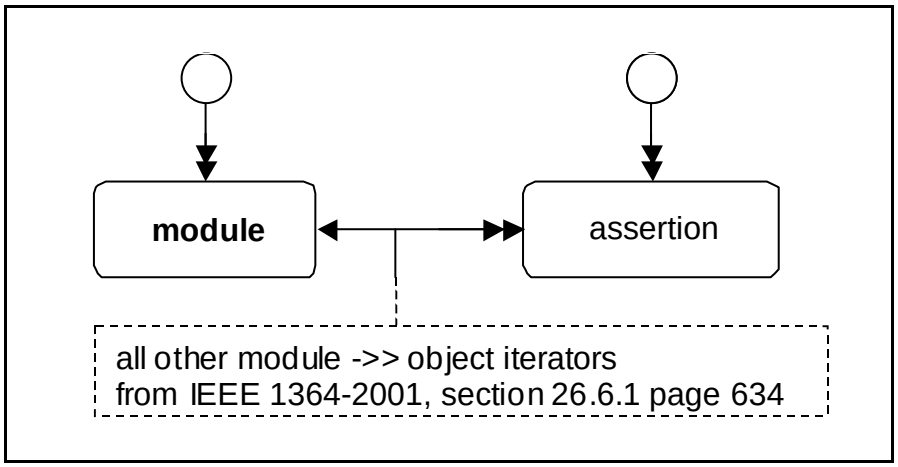


Figure 27-1 — Encompassing assertions

Note: Iteration on assertions from interfaces is not shown in Figure 27-1 since the interface object type is not currently defined in VPI. However the assertion API permits iteration on assertions from interface instance handles and obtaining static information on assertions used in interfaces (see Section 27.3.2.1).

1) Iterate all assertions in the design: use a `NULL` reference handle (ref) to `vpi_iterate()`, e.g.,

```
itr = vpi_iterate(vpiAssertion, NULL);
while (assertion = vpi_scan(itr)) {
   /* process assertion */
}
```

2) Iterate all assertions in an instance: pass the appropriate instance handle as a reference handle to `vpi_iterate()`, e.g.,

```
itr = vpi_iterate(vpiAssertion, instanceHandle);
while (assertion = vpi_scan(itr)) {
   /* process assertion */
}
```

3) Obtain the assertion by name: extend `vpi_handle_by_name` to also search for assertion names in the appropriate scope(s), e.g.,

```
vpiHandle = vpi_handle_by_name(assertName, scope)
```

4) To obtain an assertion of a specific type, e.g. cover assertions, the following approach should be used:

```
vpiHandle assertion;
itr = vpi_iterate(vpiAssertionType, NULL);
while (assertion = vpi_scan(itr)) {
   if (vpi_get(vpiAssertionType, assertion) == vpiCoverType) {
      /* process cover type assertion */
   }
}
```

NOTES

1—As with all VPI handles, assertion handles are handles to a specific instance of a specific assertion.

2—Unnamed assertions cannot be found by name.

27.3.2 Obtaining static assertion information

The following information about an assertion is considered to be static.

— Assertion name

— Instance in which the assertion occurs

— Module definition containing the assertion

— Assertion type

1) Sequence
2) Assert
3) Cover
4) Property
5) ImmediateAssert

— Assertion source information: the file, line, and column where the assertion is defined.

— Assertion clocking domain/expression

27.3.2.1 Using `vpi_get_assertion_info`

Static information can be obtained directly from an assertion handle by using `vpi_get_assertion_info`, as shown below.

```
typedef struct t_vpi_source_info {
   PLI_BYTE* *fileName;
   PLI_INT32 startLine;
   PLI_INT32 startColumn;
   PLI_INT32 endLine;
   PLI_INT32 endColumn;
} s_vpi_source_info, *p_vpi_source_info;
typedef struct t_vpi_assertion_info {
   PLI_BYTE8 *name;     /* name of assertion */
   vpiHandle instance;  /* instance containing assertion */
   PLI_BYTE8 defname;   /* name of module/interface containing assertion
*/
   vpiHandle clock; /* clocking expression */
   PLI_INT32 assertionType; /* vpiSequenceType, ... */
   s_vpi_source_info sourceInfo;
} s_vpi_assertion_info, *p_vpi_assertion_info;
int vpi_get_assertion_info (assert_handle, p_vpi_assertion_info);
```

This call obtains all the static information associated with an assertion.

The inputs are a valid handle to an assertion and a pointer to an existing `s_vpi_assertion_info` data structure. On success, the function returns `TRUE` and the `s_vpi_assertion_info` data structure is filled in as appropriate. On failure, the function returns `FALSE` and the contents of the assertion data structure are unpredictable.

Assertions can occur in modules and interfaces: for assertions defined in modules, the instance field in the `s_vpi_assertion_info` structure shall contain the handle to the appropriate module or interface instance. Note that VPI does not currently define the information model for interfaces and therefore the interface instance handle shall be implementation dependent. The clock field of that structure contains a handle to the event expression representing the clock for the assertion, as determined by Section 17.13.

NOTE: a single call returns all the information for efficiency reasons.

27.3.2.2 Extending `vpi_get()` and `vpi_get_str()`

In addition to `vpi_get_assertion_info`, the following existing VPI functions are also extended:

```
vpi_get(), vpi_get_str()
```

`vpi_get()` can be used to query the following VPI property from a handle to an assertion:

`vpiAssertionDirective`
returns one of `vpiSequenceType`, `vpiAssertType`, `vpiCoverType`, `vpiPropertyType`, `vpiImmediateAssertType`.

`vpiLineNo`
returns the line number where the assertion is declared.

`vpi_get_str()` can be used to obtain the following VPI properties from an assertion handle:

`vpiFileName`
returns the filename of the source file where the assertion was declared.

`vpiName`
returns the name of the assertion.

`vpiFullName`
returns the fully qualified name of the assertion.

27.4 Dynamic information

This section defines how to place assertion system and assertion callbacks.

27.4.1 Placing assertion system callbacks

Use `vpi_register_cb()`, setting the `cb_rtn` element to the function to be invoked and the reason element of the `s_cb_data` structure to one of the following values, to place an assertion system callback.

`cbAssertionSysInitialized`
occurs after the system has initialized. No assertion-specific actions can be performed until this callback completes. The assertion system can initialize before `cbStartOfSimulation` does or afterwards.

`cbAssertionSysStart`
the assertion system has become active and starts processing assertion attempts. This always occur after `cbAssertionSysInitialized`. By default, the assertion system is “started” on simulation startup, but the user can delay this by using assertion system control actions.

`cbAssertionSysStop`
the assertion system has been temporarily suspended. While stopped no assertion attempts are processed and no assertion-related callbacks occur. The assertion system can be stopped and resumed an arbitrary number of times during a single simulation run.

`cbAssertionSysEnd`
occurs when all assertions have completed and no new attempts shall start. Once this callback occurs no more assertion-related callbacks shall occur and assertion-related actions shall have no further effect. This typically occurs after the end of simulation.

`cbAssertionSysReset`
occurs when the assertion system is reset, e.g., due to a system control action.

The callback routine invoked follows the normal VPI callback prototype and is passed an `s_cb_data` containing the callback reason and any user data provided to the `vpi_register_cb()` call.

27.4.2 Placing assertions callbacks

Use `vpi_register_assertion_cb()` to place an assertion callback; the prototype is:

```
vpiHandle vpi_register_assertion_cb(
   vpiHandle,           /* handle to assertion */
   PLI_INT32 reason,    /* reason for which callbacks needed */
   PLI_INT32 (*cb_rtn)( /* callback function */
      PLI_INT32 reason,
      vpiHandle assertion,
      p_vpi_attempt_info info,
      PLI_BYTE8 *userData ),
   PLI_BYTE8 *user_data    /* user data to be supplied to cb */
);
typedef struct t_vpi_assertion_step_info {
   PLI_INT32 matched_expression_count;
   vpiHandle *matched_exprs;  /* array of expressions */
   p_vpi_source_info *exprs_source_info; /* array of source info */
   PLI_INT32 stateFrom, stateTo;/* identify transition */
} s_vpi_assertion_step_info, *p_vpi_assertion_step_info;
typedef struct t_vpi_attempt_info {
   union {
      vpiHandle failExpr;
      p_vpi_assertion_step_info step;
   } detail;
   s_vpi_time attemptTime,
} s_vpi_attempt_info, *p_vpi_attempt_info;
```

where *reason* is any of the following.

`cbAssertionStart`
an assertion attempt has started. For most assertions one attempt starts each and every clock tick.

`cbAssertionSuccess`
when an assertion attempt reaches a success state.

`cbAssertionFailure`
when an assertion attempt fails to reach a success state.

`cbAssertionStepSucCess`
progress one step an attempt. By default, step callbacks are not enabled on any assertions; they are enabled on a per-assertion/per-attempt basis (see Section 27.5.2), rather than on a per-assertion basis.

`cbAssertionStepFailure`
failure to progress by one step along an attempt. By default, step callbacks are not enabled on any assertions; they are enabled on a per-assertion/per-attempt basis (see Section 27.5.2), rather than on a per-assertion basis.

`cbAssertionDisable`
whenever the assertion is disabled (e.g., as a result of a control action).

`cbAssertionEnable`
whenever the assertion is enabled.

`cbAssertionReset`
whenever the assertion is reset.

`cbAssertionKill`
when an attempt is killed (e.g., as a result of a control action).

These callbacks are specific to a given assertion; placing such a callback on one assertion does not cause the callback to trigger on an event occurring on a different assertion. If the callback is successfully placed, a han-

dle to the callback is returned. This handle can be used to remove the callback via `vpi_remove_cb()`. If there were errors on placing the callback, a `NULL` handle is returned. As with all other calls, invoking this function with invalid arguments has unpredictable effects.

Once the callback is placed, the user-supplied function shall be called each time the specified event occurs on the given assertion. The callback shall continue to be called whenever the event occurs until the callback is removed.

The callback function shall be supplied the following arguments:

1) the reason for the callback

2) the handle for the assertion

3) a pointer to an attempt information structure

4) a reference to the user data supplied when the callback was placed.

The `attempt information structure` contains details relevant to the specific event that occurred.

- On disable, enable, reset and kill events, the `info` field is absent (a `NULL` pointer is given as the value of `info`).
- On start and success events, only the `attempt time` field is valid.
- On a failure event, the `attempt time` and `detail.failExpr` are valid.
- On a step callback, the `attempt time` and `detail.step` elements are valid.

On a step callback, the `detail` describes the set of expressions matched in satisfying a step along the assertion, along with the corresponding source references. In addition, the `step` also identifies the source and destination "states" needed to uniquely identify the path being taken through the assertion. *State ids* are just integers, with `0` identifying the origin state, `1` identifying an accepting state, and any other number representing some intermediate point in the assertion. It is possible for the number of expressions in a step to be `0` (zero), which represents an unconditional transition. In the case of a failing transition, the information provided is just as that for a successful one, but the last expression in the array represents the expression where the transition failed.

NOTES

1—In a failing transition, there shall always be at least one element in the expression array.

2—Placing a step callback results in the same callback function being invoked for both success and failure steps.

27.5 Control functions

This section defines how to obtain assertion system control and assertion control information.

27.5.1 Assertion system control

Use `vpi_control()`, with one of the following operators and no other arguments, to obtain assertion system control information.

Usage example: `vpi_control(vpiAssertionSysReset)`

`vpiAssertionSysReset`
discards all attempts in progress for all assertions and restore the entire assertion system to its initial state. Any pre-existing `vpiAssertionStepSuccess` and `vpiAssertionStepFailure` callbacks shall be removed; all other assertion callbacks shall remain.

Usage example: `vpi_control(vpiAssertionSysStop)`

`vpiAssertionSysStop`
considers all attempts in progress as unterminated and disable any further assertions from being started. This control has no effect on pre-existing assertion callbacks.

Usage example: `vpi_control(vpiAssertionSysStart)`

`vpiAssertionSysStart`
restarts the assertion system after it was stopped (e.g., due to `vpiAssertionSysStop`). Once started, attempts shall resume on all assertions. This control has no effect on prior assertion callbacks.

Usage example: `vpi_control(vpiAssertionSysEnd)`

`vpiAssertionSysEnd`
discard all attempts in progress and disables any further assertions from starting. All assertion callbacks currently installed shall be removed. Note that once this control is issued, no further assertion related actions shall be permitted.

27.5.2 Assertion control

Use `vpi_control()`, with one of the following operators, to obtain assertion control information.

— For the following operators, the second argument shall be a valid assertion handle.

Usage example: `vpi_control(vpiAssertionReset, assertionHandle)`

`vpiAssertionReset`
discards all current attempts in progress for this assertion and resets this assertion to its initial state.

Usage example: `vpi_control(vpiAssertionDisable, assertionHandle)`

`vpiAssertionDisable`
disables the starting of any new attempts for this assertion. This has no effect on any existing attempts. or if the assertion already disabled. By default, all assertions are enabled.

Usage example: `vpi_control(vpiAssertionEnable, assertionHandle)`

`vpiAssertionEnable`
enables starting new attempts for this assertion. This has no effect if assertion already enabled or on any existing attempts.

— For the following operators, the second argument shall be a valid assertion handle and the third argument shall be an attempt start-time (as a pointer to a correctly initialized `s_vpi_time` structure).

Usage example: `vpi_control(vpiAssertionKill, assertionHandle, attempt)`

`vpiAssertionKill`
discards the given attempts, but leaves the assertion enabled and does not reset any state used by this assertion (e.g., `past()` sampling).

Usage example: `vpi_control(vpiAssertionDisableStep, assertionHandle, attempt)`

`vpiAssertionDisableStep`
disables step callbacks for this assertion. This has no effect if stepping not enabled or it is already disabled.

— For the following operator, the second argument shall be a valid assertion handle, the third argument shall be an attempt start-time (as a pointer to a correctly initialized `s_vpi_time` structure) and the fourth argument shall be a step control constant.

Usage example: `vpi_control(vpiAssertionEnableStep, assertionHandle, attempt,`

vpiAssertionClockSteps)

vpiAssertionEnableStep
enables step callbacks to occur for this assertion attempt. By default, stepping is disabled for all assertions. This call has no effect if stepping is already enabled for this assertion and attempt, other than possibly changing the stepping mode for the attempt if the attempt has not occurred yet. The stepping mode of any particular attempt cannot be modified after the assertion attempt in question has started.

NOTE—In this release, the only step control constant available is vpiAssertionClockSteps, indicating callbacks on a per assertion/clock-tick basis. The assertion clock is the event expression supplied as the clocking expression to the assertion declaration. The assertion shall “advance” whenever this event occurs and, when stepping is enabled, such events shall also cause step callbacks to occur.

Section 28
SystemVerilog Coverage API

28.1 Requirements

This chapter defines the Coverage Application Programming Interface (API) in SystemVerilog.

28.1.1 SystemVerilog API

The following criteria are used within this API.

1) This API shall be similar for all coverages
   There are a wide number of coverage types available, with possibly different sets offered by different vendors. Maintaining a common interface across all the different types enhances portability and ease of use.

2) At a minimum, the following types of coverage shall be supported:

   a) statement coverage

   b) toggle coverage

   c) fsm coverage

      i) fsm states

      ii) fsm transitions

   c) assertion coverage

3) Coverage APIs shall be extensible in a transparent manner, i.e., adding a new coverage type shall not break any existing coverage usage.

4) This API shall provide means to obtain coverage information from specific sub-hierarchies of the design without requiring the user to enumerate all instances in those hierarchies.

28.1.2 Naming conventions

All elements added by this interface shall conform to the Verilog Procedural Interface (VPI) interface naming conventions.

— All names are prefixed by `vpi`.

— All *type names* shall start with `vpi`, followed by initially capitalized words with no separators, e.g., `vpiCoverageStmt`.

— All callback names shall start with `cb`, followed by initially capitalized words with no separators, e.g., `cbAssertionStart`.

— All *function names* shall start with `vpi_`, followed by all lowercase words separated by underscores (_), e.g., `vpi_control()`.

28.1.3 Nomenclature

The following terms are used in this standard.

Statement coverage — whether a statement has been executed or not, where *statement* is anything defined as a statement in the LRM. *Covered* means it executed at least once. Some implementations also permit

querying the execution count. The granularity of statement coverage can be per-statement or per-statement block (however defined).

FSM coverage — the number of states in a finite state machine (FSM) that this simulation reached. This standard does not require FSM automatic extraction, but a standard mechanism to force specific extraction is available via pragmas.

Toggle coverage — for each bit of every signal (wire and register), whether that bit has both a `0` value and a `1` value. *Full coverage* means both are seen; otherwise, some implementations can query for *partial coverage*. Some implementations also permit querying the toggle count of each bit.

Assertion coverage — for each assertion, whether it has had at least one success. Implementations permit querying for further details, such as attempt counts, success counts, failure counts and failure coverage.

These terms define the “primitives” for each coverage type. Over instances or blocks, the coverage number is merely the sum of all contained primitives in that instance or block.

28.2 SystemVerilog real-time coverage access

This section describes the mechanisms in SystemVerilog through which SystemVerilog code can query and control coverage information. Coverage information is provided to SystemVerilog by means of a number of built-in system functions (described in Section 28.2.2) using a number of predefined constants (described in Section 28.2.1) to describe the types of coverage and the control actions to be performed.

28.2.1 Predefined coverage constants in SystemVerilog

The following predefined `` `define ``s represent basic real-time coverage capabilities accessible directly from SystemVerilog.

— Coverage control

```
`define SV_COV_START       0
`define SV_COV_STOP        1
`define SV_COV_RESET       2
`define SV_COV_QUERY       3
```

— Scope definition (hierarchy traversal/accumulation type)

```
`define SV_COV_MODULE      10
`define SV_COV_HIER        11
```

— Coverage type identification

```
`define SV_COV_ASSERTION   20
`define SV_COV_FSM_STATE   21
`define SV_COV_STATEMENT   22
`define SV_COV_TOGGLE      23
```

— Status results

```
`define SV_COV_OVERFLOW    -2
`define SV_COV_ERROR       -1
`define SV_COV_NOCOV       0
`define SV_COV_OK          1
`define SV_COV_PARTIAL     2
```

28.2.2 Built-in coverage access system functions

28.2.2.1 $coverage_control

```
$coverage_control(control_constant,
                  coverage_type,
                  scope_def,
                  modules_or_instance)
```

This function enables, disables, resets or queries the availability of coverage information for the specified portion of the hierarchy. The return value is a `‘define`d name, with the value indicating the success of the action.

`‘SV_COV_OK`
the request is successful. When querying, if starting, stopping, or resetting this means the desired effect occurred, coverage is available. A successful reset clears all coverage (i.e., using a ...`get() == 0` after a successful …`reset()`).

`‘SV_COV_ERROR`
the call failed with no action, typically due to errors in the arguments, such as a non-existing module or instance specifications.

`‘SV_COV_NOCOV`
coverage is not available for the requested portion of the hierarchy.

`‘SV_COV_PARTIAL`
coverage is only partially available in the requested portion of the hierarchy (i.e., some instances have the requested coverage information, some don’t).

Starting, stopping, or resetting coverage multiple times in succession for the same instance(s) has no further effect if coverage has already been started, stopped, or reset for that/those instance(s).

The hierarchy(ies) being controlled or queried are specified as follows.

`‘SV_MODULE_COV, "unique module def name"`
provides coverage of all instances of the given module (the unique module name is a string), excluding any child instances in the instances of the given module. The module definition name can use special notation to describe nested module definitions.

`‘SV_COV_HIER, "module name"`
provides coverage of all instances of the given module, including all the hierarchy below.

`‘SV_MODULE_COV, instance_name`
provides coverage of the one named instance. The instance is specified as a normal Verilog hierarchical path.

`‘SV_COV_HIER, instance_name`
provides coverage of the named instance, plus all the hierarchy below it.

All the permutations are summarized in Table 28-1.

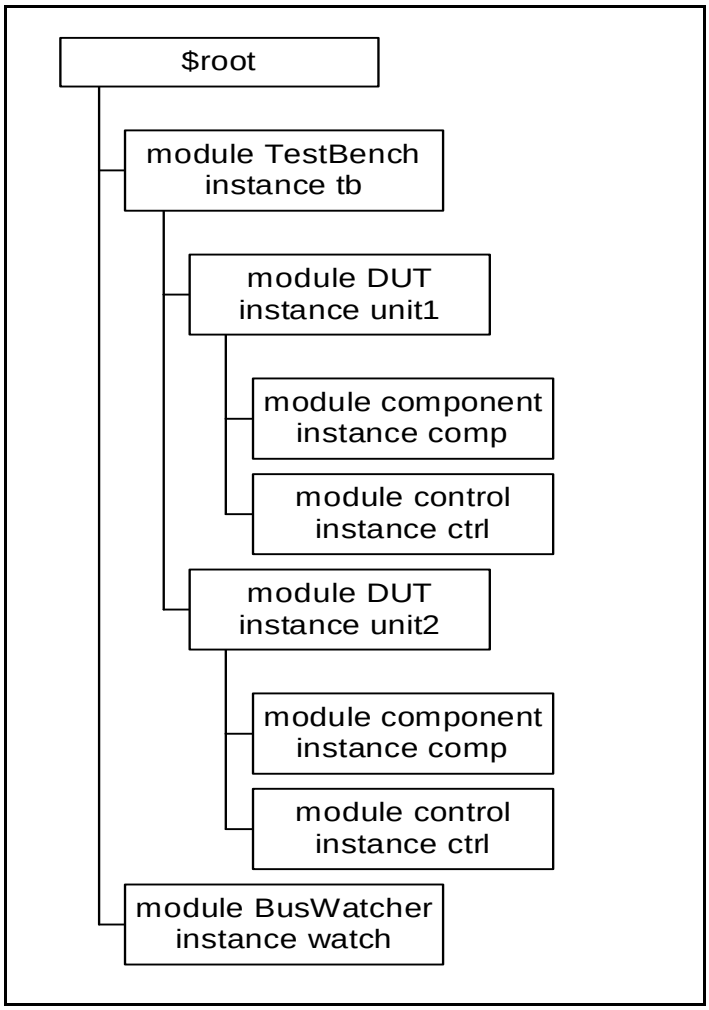
Table 28-1: Instance coverage permutations

Control/query	“Definition name”	instance.name
`‘SV_COV_MODULE`	The sum of coverage for all instances of the named module, excluding any hierarchy below those instances.	Coverage for just the named instance, excluding any hierarchy in instances below that instance.

Table 28-1: Instance coverage permutations (continued)

Control/query	"Definition name"	instance.name
`SV_COV_HIER	The sum of coverage for all instances of the named module, including all coverage for all hierarchy below those instances.	Coverage for the named instance and any hierarchy below it.

NOTE—Definition names are represented as strings, whereas instance names are referenced by hierarchical paths. A hierarchical path need not include any . if the path refers to an instance in the current context (i.e., normal Verilog hierarchical path rules apply).



Example 28-1 — Hierarchical instance example

If coverage is enabled on all instances shown in Example 28-1 —, then:

```
$coverage_control(`SV_COV_CHECK, `SV_COV_TOGGLE, `SV_COV_HIER, $root)
```

checks all instances to verify they have coverage and, in this case, returns `SV_COV_OK.

`$coverage_control(‘SV_COV_RESET, ‘SV_COV_TOGGLE, ‘SV_COV_MODULE, "DUT")`
resets coverage collection on both instances of the DUT, specifically, `$root.tb.unit1` and `$root.tb.unit2`, but leaves coverage unaffected in all other instances.

```
$coverage_control(‘SV_COV_RESET, ‘SV_COV_TOGGLE, ‘SV_COV_MODULE,
                        $root.tb.unit1)
```
resets coverage of only the instance `$root.tb.unit1`, leaving all other instances unaffected.

```
$coverage_control(‘SV_COV_STOP, ‘SV_COV_TOGGLE, ‘SV_COV_HIER,
                        $root.tb.unit1)
```
resets coverage of the instance `$root.tb.unit1` and also reset coverage for all instances below it, specifically `$root.tb.unit1.comp` and `$root.tb.unit1.ctrl`.

`$coverage_control(‘SV_COV_START, ‘SV_COV_TOGGLE, ‘SV_COV_HIER, "DUT")`
starts coverage on all instances of the module DUT and of all hierarchy(ies) below those instances. In this design, coverage is started for the instances `$root.tb.unit1`, `$root.tb.unit1.comp`, `$root.tb.unit1.ctrl`, `$root.tb.unit2`, `$root.tb.unit2.comp`, and `$root.tb.unit2.ctrl`.

28.2.2.2 $coverage_get_max

```
$coverage_get_max(coverage_type, scope_def, modules_or_instance)
```

This function obtains the value representing 100% coverage for the specified coverage type over the specified portion of the hierarchy. This value shall remain constant across the duration of the simulation.

NOTE—This value is proportional to the design size and structure, so it also needs to be constant through multiple independent simulations and compilations of the same design, assuming any compilation options do not modify the coverage support or design structure.

The return value is an integer, with the following meanings.

`-2 (‘SV_COV_OVERFLOW)`
the value exceeds a number that can be represented as an integer.

`-1 (‘SV_COV_ERROR)`
an error occurred (typically due to using incorrect arguments).

`0 (‘SV_COV_NOCOV)`
no coverage is available for that coverage type on that/those hierarchy(ies).

`+pos_num`
the maximum coverage number (where *`pos_num`* `> 0`), which is the sum of all coverable items of that type over the given hierarchy(ies).

The scope of this function is specified as per `$coverage_control` (see Section 28.2.2.1).

28.2.2.3 $coverage_get

```
$coverage_get(coverage_type, scope_def, modules_or_instance)
```

This function obtains the current coverage value for the given coverage type over the given portion of the hierarchy. This number can be converted to a coverage percentage by use of the equation:

$$\mathrm{cov}\,erage\% = \frac{\mathrm{cov}\,erage\_get()}{\mathrm{cov}\,erage\_get\_\max()} * 100$$

The return value follows the same pattern as `$coverage_get_max` (see Section 28.2.2.2), but with *`pos_num`* representing the current coverage level, i.e., the number of the coverable items that have been covered in this/these hierarchy(ies).

The scope of this function is specified as per `$coverage_control` (see Section 28.2.2.1).

The return value is an integer, with the following meanings.

`-2 (‘SV_COV_OVERFLOW)`
the value exceeds a number that can be represented as an integer.

`-1 (‘SV_COV_ERROR)`
an error occurred (typically due to using incorrect arguments).

`0 (‘SV_COV_NOCOV)`
no coverage is available for that coverage type on that/those hierarchy(ies).

+pos_num
the maximum coverage number (where *pos_num* > 0), which is the sum of all coverable items of that type over the given hierarchy(ies).

28.2.2.4 $coverage_merge

```
$coverage_merge(coverage_type, "name")
```

This function loads and merges coverage data for the specified coverage into the simulator. *name* is an arbitrary string used by the tool, in an *implementation-specific* way, to locate the appropriate coverage database, i.e., tools are allowed to store coverage files any place they want with any extension they want *as long* as the user can retrieve the information by asking for a specific saved name from that coverage database. If *name* does not exist or does not correspond to a coverage database from the same design, an error shall occur. If an error occurs during loading, the coverage numbers generated by this simulation might not be meaningful.

The return values from this function are:

`‘SV_COV_OK`
the coverage data has been found and merged.

`‘SV_COV_NOCOV`
the coverage data has been found, but did not contain the coverage type requested.

`‘SV_COV_ERROR`
the coverage data was not found or it did not correspond to this design, or another error.

28.2.2.5 $coverage_save

```
$coverage_save(coverage_type, "name")
```

This function saves the current state of coverage to the tool’s coverage database and associates it with the file named *name*. This file name shall not contain any directory specification or extensions. Data saved to the database shall be retrieved later by using `$coverage_merge` and supplying the same name. Saving coverage shall not have any effect on the state of coverage in this simulation.

The return values from this function are:

`‘SV_COV_OK`
the coverage data was successfully saved.

`‘SV_COV_NOCOV`
no such coverage is available in this design (nothing was saved).

`‘SV_COV_ERROR`
some error occurred during the save. If an error occurs, the tool shall automatically remove the coverage database entry for *name* to preserve the coverage database integrity. It is *not* an error to overwrite a previously existing *name*.

NOTES

1—The coverage database format is implementation-dependent.

2—Mapping of names to actual directories/files is implementation-dependent. There is no requirement that a coverage name map to any specific set of files or directories.

28.3 FSM recognition

Coverage tools need to have automatic recognition of many of the common FSM coding idioms in Verilog/ SystemVerilog. This standard does not attempt to describe or require any specific automatic FSM recognition mechanisms. However, the standard does prescribe a means by which non-automatic FSM extraction occurs. The presence of any of these standard FSM description additions shall override the tool's default extraction mechanism.

Identification of an FSM consists of identifying the following items:

1) the state register (or expression)

2) the next state register (this is optional)

3) the legal states.

28.3.1 Specifying the signal that holds the current state

Use the following pragma to identify the vector signal that holds the current state of the FSM:

```
/* tool state_vector signal_name */
```

where `tool` and `state_vector` are required keywords. This pragma needs to be specified inside the module definition where the signal is declared.

Another pragma is also required, to specify an enumeration name for the FSM. This enumeration name is also specified for the next state and any possible states, associating them with each other as part of the same FSM. There are two ways to do this:

— Use the same pragma:

```
/* tool state_vector signal_name enum enumeration_name */
```

— Use a separate pragma in the signal's declaration:

```
/* tool state_vector signal_name */
reg [7:0] /* tool enum enumeration_name */ signal_name;
```

In either case, `enum` is a required keyword; if using a separate pragma, `tool` is also a required keyword and the pragma needs to be specified immediately after the bit-range of the signal.

28.3.2 Specifying the part-select that holds the current state

A part-select of a vector signal can be used to hold the current state of the FSM. When `cmView` displays or reports FSM coverage data, it names the FSM after the signal that holds the current state. If a part-select holds the current state in the user's FSM, the user needs to also specify a name for the FSM that `cmView` can use. The FSM name is not the same as the enumeration name.

Specify the part-select by using the following pragma:

```
/* tool state_vector signal_name[n:n] FSM_name enum enumeration_name */
```


28.3.3 Specifying the concatenation that holds the current state

Like specifying a part-select, a concatenation of signals can be specified to hold the current state (when including an FSM name and an enumeration name):

```
/* tool state_vector {signal_name , signal_name, ...} FSM_name enum
   enumeration_name */
```

The concatenation is composed of all the signals specified. Bit-selects or part-selects of signals cannot be used in the concatenation.

28.3.4 Specifying the signal that holds the next state

The signal that holds the next state of the FSM can also be specified with the pragma that specifies the enumeration name:

```
reg [7:0] /* tool enum enumeration_name */
signal_name
```

This pragma can be omitted if, and only if, the FSM does not have a signal for the next state.

28.3.5 Specifying the current and next state signals in the same declaration

The tool assumes the first signal following the pragma holds the current state and the next signal holds the next state when a pragma is used for specifying the enumeration name in a declaration of multiple signals, e.g.,

```
/* tool state_vector cs */
reg [1:0] /* tool enum myFSM */ cs, ns, nonstate;
```

In this example, the tool assumes signal cs holds the current state and signal ns holds the next state. It assumes nothing about signal nonstate.

28.3.6 Specifying the possible states of the FSM

The possible states of the FSM can also be specified with a pragma that includes the enumeration name:

```
parameter /* tool enum enumeration_name */
S0 = 0,
s1 = 1,
s2 = 2,
s3 = 3;
```

Put this pragma immediately after the keyword parameter, unless a bit-width for the parameters is used, in which case, specify the pragma immediately after the bit-width:

```
parameter [1:0] /* tool enum enumeration_name */
S0 = 0,
s1 = 1,
s2 = 2,
s3 = 3;
```

28.3.7 Pragmas in one-line comments

These pragmas work in both block comments, between the `/*` and `*/` character strings, and one-line comments, following the `//` character string, e.g.,

```
parameter [1:0] // tool enum enumeration_name
S0 = 0,
s1 = 1,
```

```
    s2 = 2,
    s3 = 3;
```

28.3.8 Example

```
module m3;

reg[31:0] cs;
reg[31:0] /* tool enum MY_FSM */ ns;
reg[31:0] clk;
reg[31:0] rst;

// tool state_vector cs enum MY_FSM

parameter // tool enum MY_FSM
p1=10,
p2=11,
p3=12;

endmodule // m3
```

Signal `ns` holds the next state

Signal `cs` holds the current state

`p1`, `p2`, and `p3` are possible states of the FSM

Example 28-2 — FSM specified with pragmas

28.4 VPI coverage extensions

28.4.1 VPI entity/relation diagrams related to coverage

28.4.2 Extensions to VPI enumerations

— Coverage control

```
#define vpiCoverageStart
#define vpiCoverageStop
#define vpiCoverageReset
#define vpiCoverageCheck
#define vpiCoverageMerge
#define vpiCoverageSave
```

— VPI properties

1) Coverage type properties

```
#define vpiAssertCoverage
#define vpiFsmStateCoverage
#define vpiStatementCoverage
#define vpiToggleCoverage
```

2) Coverage status properties

```
#define vpiCovered
#define vpiCoverMax
#define vpiCoveredCount
```

3) Assertion-specific coverage status properties

```
#define vpiAssertAttemptCovered
#define vpiAssertSuccessCovered
```



```
#define vpiAssertFailureCovered
```

4) FSM-specific methods

```
#define vpiFsmStates
#define vpiFsmStateExpression
```

— FSM handle types (vpi types)

```
#define vpiFsm
#define vpiFsmHandle
```

28.4.3 Obtaining coverage information

To obtain coverage information, the `vpi_get()` function is extended with additional VPI properties that can be obtained from existing handles:

```
vpi_get(<coverageType>, instance_handle)
```

Returns the number of covered items of the given coverage type in the given instance. Coverage type is one of the coverage type properties described in Section 28.4.2. For example, given coverage type `vpiStatement-Coverage`, this call would return the number of covered statements in the instance pointed by *instance_handle*.

```
vpi_get(vpiCovered, assertion_handle)
vpi_get(vpiCovered, statement_handle)
vpi_get(vpiCovered, signal_handle)
vpi_get(vpiCovered, fsm_handle)
vpi_get(vpiCovered, fsm_state_handle)
```

Returns whether the item referenced by the handle has been covered. For handles that can contain multiple coverable entities, such as statement, fsm and signal handles, the return value indicates how many of the entities have been covered.

— For assertion handle, the coverable entities are assertions

— For statement handle, the entities are statements

— For signal handle, the entities are individual signal bits

— For fsm handle, the entities are fsm states

```
vpi_get(vpiCoveredCount, assertion_handle)
vpi_get(vpiCoveredCount, statement_handle)
vpi_get(vpiCoveredCount, signal_handle)
vpi_get(vpiCoveredCount, fsm_handle)
vpi_get(vpiCoveredCount, fsm_state_handle)
```

Returns the number of times each coverable entity referred by the handle has been covered. Note that this is only easily interpretable when the handle points to a unique coverable item (such as an individual statement); when handle points to an item containing multiple coverable entities (such as a handle to a block statement containing a number of statements), the result is the sum of coverage counts for each of the constituent entities.

```
vpi_get(vpiCoveredMax, assertion_handle)
vpi_get(vpiCoveredMax, statement_handle)
vpi_get(vpiCoveredMax, signal_handle)
vpi_get(vpiCoveredMax, fsm_handle)
vpi_get(vpiCoveredMax, fsm_state_handle)
```

Returns the number of coverable entities pointed by the handle. Note that this shall always return 1 (one) when applied to an assertion or FSM state handle.

```
vpi_iterate(vpiFsm, instance-handle)
```

Returns an iterator to all FSMs in an instance.

```
vpi_handle(vpiFsmStateExpression, fsm-handle)
```

Returns the handle to the signal or expression encoding the FSM state.

```
vpi_iterate(vpiFsmStates, fsm-handle)
```

Returns an iterator to all states of an FSM.

```
vpi_get_value(fsm_state_handle, state-handle)
```

Returns the value of an FSM state.

28.4.4 Controlling coverage

```
vpi_control(<coverageControl>, <coverageType>, instance_handle)
vpi_control(<coverageControl>, <coverageType>, assertion_handle)
```

Controls the collection of coverage on the given instance or assertion. Note that statement, toggle and FSM coverage are not individually controllable (i.e., they are controllable only at the instance level and not on a per statement/signal/FSM basis). The semantics and behavior are as per the `$coverage_control` system function (see Section 28.2.2.1). *coverageControl* is one `vpiCoverageStart`, `vpiCoverageStop`, `vpiCoverageReset` or `vpiCoverageCheck`, as defined in Section 28.4.2. *coverageType* is any one of the VPI coverage type properties (Section 28.4.2)

```
vpi_control(<coverageControl>, <coverageType>, name)
```

This saves or merges coverage into the current simulation. The semantics and behavior are specified as per the equivalent system functions `$coverage_merge` (see Section 28.2.2.4) and `$coverage_save` (see Section 28.2.2.5). *coverageControl* is one of `vpiCoverageMerge` or `vpiCoverageSave`, defined in Section 28.4.2.

Annex A
Formal Syntax

(Normative)

The formal syntax of SystemVerilog is described using Backus-Naur Form (BNF). The conventions used are:

— Keywords and punctuation are in **bold** text.

— Syntactic categories are named in non-bold text.

— A vertical bar (|) separates alternatives.

— Square brackets ([ ]) enclose optional items.

— Braces ({ }) enclose items which can be repeated zero or more times.

The full syntax and semantics of Verilog and SystemVerilog are not described solely using BNF. The normative text description contained within the chapters of the IEEE 1364-2001 Verilog standard and this SystemVerilog document provide additional details on the syntax and semantics described in this BNF.

A.1 Source text

A.1.1 Library source text

library_text ::= { library_descriptions }

library_descriptions ::=
    library_declaration
  | include_statement
  | config_declaration

library_declaration ::=
    library library_identifier file_path_spec { **,** file_path_spec }
      [**-incdir** file_path_spec { **,** file_path_spec }] **;**

file_path_spec ::= file_path

include_statement ::= **include** <file_path_spec> **;**

A.1.2 Configuration source text

config_declaration ::=
    config config_identifier **;**
      design_statement
      { config_rule_statement }
    endconfig

design_statement ::= **design** { [library_identifier **.**] cell_identifier } **;**

config_rule_statement ::=
    default_clause liblist_clause
  | inst_clause liblist_clause
  | inst_clause use_clause
  | cell_clause liblist_clause
  | cell_clause use_clause

default_clause ::= **default**

inst_clause ::= **instance** inst_name

inst_name ::= topmodule_identifier { **.** instance_identifier }

cell_clause ::= **cell** [library_identifier **.**] cell_identifier

liblist_clause ::= **liblist** {library_identifier}

use_clause ::= **use** [library_identifier **.**] cell_identifier [**: config**]

A.1.3 Module and primitive source text

source_text ::= [timeunits_declaration] { description }

description ::=
module_declaration
| udp_declaration
| module_root_item
| statement_or_null

module_nonansi_header ::=
{ attribute_instance } module_keyword [lifetime] module_identifier [parameter_port_list]
list_of_ports **;**

module_ansi_header ::=
{ attribute_instance } module_keyword [lifetime] module_identifier [parameter_port_list]
[list_of_port_declarations] **;**

module_declaration ::=
module_nonansi_header [timeunits_declaration] { module_item }
endmodule [**:** module_identifier]
| module_ansi_header [timeunits_declaration] { non_port_module_item }
endmodule [**:** module_identifier]
| { attribute_instance } module_keyword [lifetime] module_identifier **(.*) ;**
[timeunits_declaration] { module_item } **endmodule** [**:** module_identifier]
| **extern** module_nonansi_header
| **extern** module_ansi_header

module_keyword ::= **module** | **macromodule**

interface_nonansi_header ::=
{ attribute_instance } **interface** [lifetime] interface_identifier
[parameter_port_list] list_of_ports **;**

interface_ansi_header ::=
{attribute_instance } **interface** [lifetime] interface_identifier
[parameter_port_list] [list_of_port_declarations] **;**

interface_declaration ::=
interface_nonansi_header [timeunits_declaration] { interface_item }
endinterface [**:** interface_identifier]
| interface_ansi_header [timeunits_declaration] { non_port_interface_item }
endinterface [**:** interface_identifier]
| { attribute_instance } **interface** interface_identifier **(.*) ;**
[timeunits_declaration] { interface_item }
endinterface [**:** interface_identifier]
| **extern** interface_nonansi_header
| **extern** interface_ansi_header

program_nonansi_header ::=
{ attribute_instance } **program** [lifetime] program_identifier
[parameter_port_list] list_of_ports **;**

program_ansi_header ::=
{attribute_instance } **program** [lifetime] program_identifier
[parameter_port_list] [list_of_port_declarations] **;**

program_declaration ::=
program_nonansi_header [timeunits_declaration] { program_item }
endprogram [**:** program_identifier]
| program_ansi_header [timeunits_declaration] { non_port_program_item }
endprogram [**:** program_identifier]
| { attribute_instance } **program** program_identifier **(.*) ;**

[timeunits_declaration] { program_item }
endprogram [**:** program_identifier]
| **extern** program_nonansi_header
| **extern** program_ansi_header

class_declaration ::=
{ attribute_instance } [**virtual**] **class** [lifetime] class_identifier [parameter_port_list]
[**extends** class_identifier] **;** [timeunits_declaration] { class_item }
endclass [**:** class_identifier]

timeunits_declaration ::=
timeunit time_literal ;
| **timeprecision** time_literal ;
| **timeunit** time_literal ;
timeprecision time_literal ;
| **timeprecision** time_literal ;
timeunit time_literal ;

A.1.4 Module parameters and ports

parameter_port_list ::= **#** **(** parameter_declaration { **,** parameter_declaration } **)**

list_of_ports ::= **(** port { **,** port } **)**

list_of_port_declarations ::=
(port_declaration { **,** port_declaration } **)**
| **( )**

non_generic_port_declaration ::=
{ attribute_instance } inout_declaration
| { attribute_instance } input_declaration
| { attribute_instance } output_declaration
| { attribute_instance } ref_declaration
| { attribute_instance } interface_port_declaration

port ::=
[port_expression]
| **.** port_identifier **(** [port_expression] **)**

port_expression ::=
port_reference
| **{** port_reference { **,** port_reference } **}**

port_reference ::=
port_identifier [**[** constant_range_expression **]**]

port_declaration ::=
non_generic_port_declaration
| { attribute_instance } generic_interface_port_declaration

A.1.5 Module items

module_common_item ::=
{ attribute_instance } module_or_generate_item_declaration
| { attribute_instance } interface_instantiation
| { attribute_instance } program_instantiation
| { attribute_instance } concurrent_assertion_item
| { attribute_instance } bind_directive

module_item ::=
non_generic_port_declaration **;**
| non_port_module_item

module_or_generate_item ::=


```
        { attribute_instance } parameter_override
      | { attribute_instance } continuous_assign
      | { attribute_instance } gate_instantiation
      | { attribute_instance } udp_instantiation
      | { attribute_instance } module_instantiation
      | { attribute_instance } initial_construct
      | { attribute_instance } always_construct
      | { attribute_instance } combinational_construct
      | { attribute_instance } latch_construct
      | { attribute_instance } ff_construct
      | { attribute_instance } net_alias
      | { attribute_instance } final_construct
      | module_common_item
      | { attribute_instance } ;
module_root_item ::=
        { attribute_instance } module_instantiation
      | { attribute_instance } local_parameter_declaration
      | interface_declaration
      | program_declaration
      | class_declaration
      | module_common_item
module_or_generate_item_declaration ::=
        net_declaration
      | data_declaration
      | genvar_declaration
      | task_declaration
      | function_declaration
      | dpi_import_export
      | extern_constraint_declaration
      | extern_method_declaration
      | clocking_decl
      | default clocking clocking_identifier ;
non_port_module_item ::=
        { attribute_instance } generated_module_instantiation
      | { attribute_instance } local_parameter_declaration
      | module_or_generate_item
      | { attribute_instance } parameter_declaration ;
      | { attribute_instance } specify_block
      | { attribute_instance } specparam_declaration
      | program_declaration
      | class_declaration
      | module_declaration
parameter_override ::= defparam list_of_defparam_assignments ;
bind_directive ::=
        bind module_identifier bind_instantiation ;
      | bind name_of_instance bind_instantiation ;
bind_instantiation ::=
        program_instantiation
      | module_instantiation
      | interface_instantiation
```

A.1.6 Interface items

interface_or_generate_item ::=
{ attribute_instance } continuous_assign
| { attribute_instance } initial_construct
| { attribute_instance } always_construct
| { attribute_instance } combinational_construct
| { attribute_instance } latch_construct
| { attribute_instance } ff_construct
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**
| module_common_item
| { attribute_instance } modport_declaration
| { attribute_instance } extern_tf_declaration
| { attribute_instance } final_construct
| { attribute_instance } **;**

extern_tf_declaration ::=
extern method_prototype
| **extern forkjoin task** named_task_proto **;**

interface_item ::=
non_generic_port_declaration **;**
| non_port_interface_item

non_port_interface_item ::=
{ attribute_instance } generated_interface_instantiation
| { attribute_instance } specparam_declaration
| interface_or_generate_item
| program_declaration
| class_declaration
| interface_declaration

A.1.7 Program items

program_item ::=
port_declaration **;**
| non_port_program_item

non_port_program_item ::=
{ attribute_instance } continuous_assign
| { attribute_instance } module_or_generate_item_declaration
| { attribute_instance } specparam_declaration
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**
| { attribute_instance } initial_construct
| { attribute_instance } concurrent_assertion_item
| class_declaration

A.1.8 Class items

class_item ::=
{ attribute_instance } class_property
| { attribute_instance } class_method
| { attribute_instance } class_constraint

class_property ::=
{ property_qualifier } data_declaration
| **const** { class_item_qualifier } data_type const_identifier [**=** constant_expression] **;**

class_method ::=

{ method_qualifier } task_declaration
| { method_qualifier } function_declaration
| **extern** { method_qualifier } method_prototype

class_constraint ::=
constraint_prototype
| constraint_declaration

class_item_qualifier[11] ::=
static
| **protected**
| **local**

property_qualifier[11] ::=
rand
| **randc**
| class_item_qualifier

method_qualifier[11] ::=
virtual
| class_item_qualifier

method_prototype ::=
task named_task_proto **;**
| **function** named_function_proto **;**

extern_method_declaration ::=
function [lifetime] class_identifier **::** function_body_declaration
| **task** [lifetime] class_identifier **::** task_body_declaration

A.1.9 Constraints

constraint_declaration ::= [**static**] **constraint** constraint_identifier **{** { constraint_block } **}**

constraint_block ::=
solve identifier_list **before** identifier_list **;**
| expression **dist** { dist_list } **;**
| constraint_expression

constraint_expression ::=
expression **;**
| expression **=>** constraint_set
| **if (** expression **)** constraint_set [**else** constraint_set]

constraint_set ::=
constraint_expression
| **{** { constraint_expression } **}**

dist_list ::= dist_item { **,** dist_item }

dist_item ::=
value_range **:=** expression
| value_range **:/** expression

constraint_prototype ::= [**static**] **constraint** constraint_identifier

extern_constraint_declaration ::=
[**static**] **constraint** class_identifier **::** constraint_identifier **{** { constraint_block } **}**

identifier_list ::= identifier { **,** identifier }

A.2 Declarations

A.2.1 Declaration types

A.2.1.1 Module parameter declarations

local_parameter_declaration ::=
localparam [signing] { packed_dimension } [range] list_of_param_assignments **;**
| **localparam** data_type list_of_param_assignments **;**

parameter_declaration ::=
parameter [signing] { packed_dimension } [range] list_of_param_assignments
| **parameter** data_type list_of_param_assignments
| **parameter type** list_of_type_assignments

specparam_declaration ::=
specparam [range] list_of_specparam_assignments **;**

A.2.1.2 Port declarations

inout_declaration ::=
inout [port_type] list_of_port_identifiers
| **inout** data_type list_of_variable_identifiers

input_declaration ::=
input [port_type] list_of_port_identifiers
| **input** data_type list_of_variable_identifiers

output_declaration ::=
output [port_type] list_of_port_identifiers
| **output** data_type list_of_variable_port_identifiers

interface_port_declaration ::=
interface_identifier list_of_interface_identifiers
| interface_identifier **.** modport_identifier list_of_interface_identifiers

ref_declaration ::= **ref** data_type list_of_port_identifiers

generic_interface_port_declaration ::=
interface list_of_interface_identifiers
| **interface .** modport_identifier list_of_interface_identifiers

A.2.1.3 Type declarations

block_data_declaration ::=
block_variable_declaration
| constant_declaration
| type_declaration

constant_declaration ::= **const** data_type const_assignment **;**

data_declaration ::=
variable_declaration
| constant_declaration
| type_declaration

genvar_declaration ::= **genvar** list_of_genvar_identifiers **;**

net_declaration ::=
net_type [signing]
[delay3] list_of_net_identifiers **;**
| net_type [drive_strength] [signing]
[delay3] list_of_net_decl_assignments **;**
| net_type [**vectored** | **scalared**] [signing]
{ packed_dimension } range [delay3] list_of_net_identifiers **;**
| net_type [drive_strength] [**vectored** | **scalared**] [signing]
{ packed_dimension } range [delay3] list_of_net_decl_assignments **;**
| **trireg** [charge_strength] [signing]
[delay3] list_of_net_identifiers **;**
| **trireg** [drive_strength] [signing]

[delay3] list_of_net_decl_assignments **;**
| **trireg** [charge_strength] [**vectored** | **scalared**] [signing]
{ packed_dimension } range [delay3] list_of_net_identifiers **;**
| **trireg** [drive_strength] [**vectored** | **scalared**] [signing]
{ packed_dimension } range [delay3] list_of_net_decl_assignments **;**

type_declaration ::=
typedef [data_type] type_declaration_identifier **;**
| **typedef** hierarchical_identifier **.** type_identifier type_declaration_identifier **;**
| **typedef** [**class**] class_identifier **;**
| **typedef** class_identifier [parameter_value_assignment] type_declaration_identifier **;**

block_variable_declaration ::=
[lifetime] data_type list_of_variable_identifiers **;**
| lifetime data_type list_of_variable_decl_assignments **;**

variable_declaration ::=
[lifetime] data_type list_of_variable_identifiers_or_assignments **;**

lifetime ::= **static** | **automatic**

A.2.2 Declaration data types

A.2.2.1 Net and variable types

casting_type ::= simple_type | number | signing

data_type ::=
integer_vector_type [signing] { packed_dimension } [range]
| integer_atom_type [signing]
| type_declaration_identifier { packed_dimension }
| non_integer_type
| **struct packed** [signing] **{** { struct_union_member } **}** { packed_dimension }
| **union packed** [signing] **{** { struct_union_member } **}** { packed_dimension }
| **struct** [signing] **{** { struct_union_member } **}**
| **union** [signing] **{** { struct_union_member } **}**
| **enum** [integer_type [signing] { packed_dimension }]
{ enum_identifier [**=** constant_expression] { **,** enum_identifier [**=** constant_expression] } **}**
| **string**
| **event**
| **chandle**
| class_scope_type_identifier

class_scope_type_identifier::=
class_identifier **::** { class_identifier **::** } type_declaration_identifier
| class_identifier **::** { class_identifier **::** } class_identifier

integer_type ::= integer_vector_type | integer_atom_type

integer_atom_type ::= **byte** | **shortint** | **int** | **longint** | **integer**

integer_vector_type ::= **bit** | **logic** | **reg**

non_integer_type ::= **time** | **shortreal** | **real** | **realtime**

net_type ::= **supply0** | **supply1** | **tri** | **triand** | **trior** | **tri0** | **tri1** | **wire** | **wand** | **wor**

port_type ::=
data_type
| net_type [signing] { packed_dimension }
| **trireg** [signing] { packed_dimension }
| [signing] { packed_dimension } range

signing ::= **signed** | **unsigned**

simple_type ::= integer_type | non_integer_type | type_identifier

struct_union_member ::= { attribute_instance } data_type list_of_variable_identifiers_or_assignments **;**

A.2.2.2 Strengths

drive_strength ::=
(strength0 , strength1)
| (strength1 , strength0)
| (strength0 , **highz1**)
| (strength1 , **highz0**)
| (**highz0** , strength1)
| (**highz1** , strength0)

strength0 ::= **supply0** | **strong0** | **pull0** | **weak0**

strength1 ::= **supply1** | **strong1** | **pull1** | **weak1**

charge_strength ::= **(small)** | **(medium)** | **(large)**

A.2.2.3 Delays

delay3 ::= **#** delay_value | **# (** mintypmax_expression [**,** mintypmax_expression [**,** mintypmax_expression]] **)**

delay2 ::= **#** delay_value | **# (** mintypmax_expression [**,** mintypmax_expression] **)**

delay_value ::=
unsigned_number
| real_number
| identifier

A.2.3 Declaration lists

list_of_defparam_assignments ::= defparam_assignment { **,** defparam_assignment }

list_of_genvar_identifiers ::= genvar_identifier { **,** genvar_identifier }

list_of_interface_identifiers ::= interface_identifier { unpacked_dimension }
{ **,** interface_identifier { unpacked_dimension } }

list_of_modport_port_identifiers ::= port_identifier { **,** port_identifier }

list_of_net_decl_assignments ::= net_decl_assignment { **,** net_decl_assignment }

list_of_net_identifiers ::= net_identifier { unpacked_dimension }
{ **,** net_identifier { unpacked_dimension } }

list_of_param_assignments ::= param_assignment { **,** param_assignment }

list_of_port_identifiers ::= port_identifier { unpacked_dimension }
{ **,** port_identifier { unpacked_dimension } }

list_of_udp_port_identifiers ::= port_identifier { **,** port_identifier }

list_of_specparam_assignments ::= specparam_assignment { **,** specparam_assignment }

list_of_tf_port_identifiers ::= port_identifier { unpacked_dimension } [**=** expression]
{ **,** port_identifier { unpacked_dimension } [**=** expression] }

list_of_tf_variable_identifiers ::= port_identifier variable_dimension [**=** expression]
{ **,** port_identifier variable_dimension [**=** expression] }

list_of_type_assignments ::= type_assignment { **,** type_assignment }

list_of_variable_decl_assignments ::= variable_decl_assignment { **,** variable_decl_assignment }

list_of_variable_identifiers ::= variable_identifier variable_dimension
{ **,** variable_identifier variable_dimension }

list_of_variable_identifiers_or_assignments ::=
list_of_variable_decl_assignments
| list_of_variable_identifiers

list_of_variable_port_identifiers ::= port_identifier variable_dimension [**=** constant_expression]
{ **,** port_identifier variable_dimension [**=** constant_expression] }

A.2.4 Declaration assignments

const_assignment ::= const_identifier **=** constant_expression

defparam_assignment ::= hierarchical_parameter_identifier **=** constant_expression

net_decl_assignment ::= net_identifier **=** expression

param_assignment ::= parameter_identifier **=** constant_param_expression

specparam_assignment ::=
    specparam_identifier **=** constant_mintypmax_expression
  | pulse_control_specparam

type_assignment ::= type_identifier **=** data_type

pulse_control_specparam ::=
    PATHPULSE$ = (reject_limit_value [**,** error_limit_value] **) ;**
  | **PATHPULSE$**specify_input_terminal_descriptor**$**specify_output_terminal_descriptor
      = (reject_limit_value [**,** error_limit_value] **) ;**

error_limit_value ::= limit_value

reject_limit_value ::= limit_value

limit_value ::= constant_mintypmax_expression

variable_decl_assignment ::=
    variable_identifier [variable_dimension] [**=** constant_expression]
  | variable_identifier **[ ] = new [** constant_expression **]** [**(** variable_identifier **)**]
  | class_identifier [parameter_value_assignment] **= new** [**(** list_of_arguments **)**]

A.2.5 Declaration ranges

unpacked_dimension ::= **[** dimension_constant_expression **:** dimension_constant_expression **]**
  | **[** dimension_constant_expression **]**

packed_dimension[9] ::=
    [dimension_constant_expression **:** dimension_constant_expression **]**
  | **[ ]**

range ::= **[** msb_constant_expression **:** lsb_constant_expression **]**

associative_dimension ::=
    [data_type **]**
  | **[*]**

variable_dimension ::=
    { unpacked_dimension }
  | **[ ]**
  | associative_dimension

dpi_dimension ::=
    variable_dimension
  | { **[ ]** }

A.2.6 Function declarations

function_data_type[8] ::=
    integer_vector_type { packed_dimension } [range]
  | integer_atom_type
  | type_declaration_identifier { packed_dimension }
  | non_integer_type
  | **struct** [**packed**] **{**{ struct_union_member } **}** { packed_dimension }
  | **union** [**packed**] **{** { struct_union_member } **}** { packed_dimension }
  | **enum** [integer_type { packed_dimension }]
      { enum_identifier [= constant_expression] { , enum_identifier [= constant_expression] } **}**
  | **string**

```
        | chandle
        | void
function_body_declaration ::=
        [ signing ] [ range_or_type ]
            [ interface_identifier . ] function_identifier ;
        { function_item_declaration }
        { function_statement_or_null }
        endfunction [ : function_identifier ]
      | [ signing ] [ range_or_type ]
            [ interface_identifier . ] function_identifier ( function_port_list ) ;
        { block_item_declaration }
        { function_statement_or_null }
        endfunction [ : function_identifier ]
function_declaration ::=
        function [ lifetime ] function_body_declaration
function_item_declaration ::=
        block_item_declaration
      | { attribute_instance } tf_input_declaration ;
      | { attribute_instance } tf_output_declaration ;
      | { attribute_instance } tf_inout_declaration ;
      | { attribute_instance } tf_ref_declaration ;
function_port_item ::=
        { attribute_instance } tf_input_declaration
      | { attribute_instance } tf_output_declaration
      | { attribute_instance } tf_inout_declaration
      | { attribute_instance } tf_ref_declaration
      | { attribute_instance } port_type list_of_tf_port_identifiers
      | { attribute_instance } tf_data_type list_of_tf_variable_identifiers
function_port_list ::= function_port_item { , function_port_item }
named_function_proto::= [ signing ] function_data_type function_identifier ( list_of_function_proto_formals )
list_of_function_proto_formals ::=
        [ { attribute_instance } function_proto_formal { , { attribute_instance } function_proto_formal } ]
function_proto_formal ::=
        tf_input_declaration
      | tf_output_declaration
      | tf_inout_declaration
      | tf_ref_declaration
range_or_type ::=
        { packed_dimension } range
      | function_data_type
dpi_import_export ::=
        import "DPI" [ dpi_import_property ] [ c_identifier = ] dpi_function_proto
      | export "DPI" [ c_identifier = ] function function_identifier
dpi_import_property ::= context | pure
dpi_function_proto ::=
        named_function_proto
      | [ signing ] function_data_type function_identifier ( list_of_dpi_proto_formals )
list_of_dpi_proto_formals ::=
        [ { attribute_instance } dpi_proto_formal { , { attribute_instance } dpi_proto_formal } ]
dpi_proto_formal ::=
```


data_type [port_identifier dpi_dimension { , port_identifier dpi_dimension }]

A.2.7 Task declarations

task_body_declaration ::=
[interface_identifier .] task_identifier ;
{ task_item_declaration }
{ statement_or_null }
endtask [: task_identifier]
| [interface_identifier .] task_identifier (task_port_list) ;
{ block_item_declaration }
{ statement_or_null }
endtask [: task_identifier]

task_declaration ::= **task** [lifetime] task_body_declaration

task_item_declaration ::=
block_item_declaration
| { attribute_instance } tf_input_declaration ;
| { attribute_instance } tf_output_declaration ;
| { attribute_instance } tf_inout_declaration ;
| { attribute_instance } tf_ref_declaration ;

task_port_list ::= task_port_item { , task_port_item }
| list_of_port_identifiers { , task_port_item }

task_port_item ::=
{ attribute_instance } tf_input_declaration
| { attribute_instance } tf_output_declaration
| { attribute_instance } tf_inout_declaration
| { attribute_instance } tf_ref_declaration ;
| { attribute_instance } port_type list_of_tf_port_identifiers
| { attribute_instance } tf_data_type list_of_tf_variable_identifiers

tf_input_declaration ::=
input [signing] { packed_dimension } list_of_tf_port_identifiers
| **input** tf_data_type list_of_tf_variable_identifiers

tf_output_declaration ::=
output [signing] { packed_dimension } list_of_tf_port_identifiers
| **output** tf_data_type list_of_tf_variable_identifiers

tf_inout_declaration ::=
inout [signing] { packed_dimension } list_of_tf_port_identifiers
| **inout** tf_data_type list_of_tf_variable_identifiers

tf_ref_declaration ::=
[**const**] **ref** tf_data_type list_of_tf_variable_identifiers

tf_data_type ::=
data_type
| **chandle**

named_task_proto ::= task_identifier (task_proto_formal { , task_proto_formal })

task_proto_formal ::=
tf_input_declaration
| tf_output_declaration
| tf_inout_declaration
| tf_ref_declaration

A.2.8 Block item declarations

block_item_declaration ::=

{ attribute_instance } block_data_declaration
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**

A.2.9 Interface declarations

modport_declaration ::= **modport** modport_item { **,** modport_item } **;**

modport_item ::= modport_identifier **(** modport_ports_declaration { **,** modport_ports_declaration } **)**

modport_ports_declaration ::=
modport_simple_ports_declaration
| modport_hierarchical_ports_declaration
| modport_tf_ports_declaration

modport_simple_ports_declaration ::=
input list_of_modport_port_identifiers
| **output** list_of_modport_port_identifiers
| **inout** list_of_modport_port_identifiers
| **ref** [data_type] list_of_modport_port_identifiers

modport_hierarchical_ports_declaration ::=
interface_instance_identifier [**[** constant_expression **]**] **.** modport_identifier

modport_tf_ports_declaration ::=
import_export modport_tf_port

modport_tf_port ::=
task named_task_proto { **,** named_task_proto }
| **function** named_function_proto { **,** named_function_proto }
| task_or_function_identifier { **,** task_or_function_identifier }

import_export ::= **import** | **export**

A.2.10 Assertion declarations

concurrent_assertion_item ::=
concurrent_assert_statement
| concurrent_cover_statement
| concurrent_assertion_item_declaration

concurrent_assert_statement ::=
[block_identifier**:**] assert_property_statement

concurrent_cover_statement ::=
[block_identifier**:**] cover_property_statement

assert_property_statement::=
assert property (property_spec **)** action_block
| **assert property (** property_instance **)** action_block

cover_property_statement::=
cover property (property_spec **)** statement_or_null
| **cover property (** property_instance **)** statement_or_null

property_instance ::=
property_identifier [**(** actual_arg_list **)**]

concurrent_assertion_item_declaration ::=
property_declaration
| sequence_declaration

property_declaration ::=
property property_identifier [property_formal_list] **;**
{ assertion_variable_declaration }
property_spec **;**
endproperty [**:** property_identifier]

property_formal_list ::=
(formal_list_item { , formal_list_item })

property_spec ::=
[clocking_event] [**disable iff**] (expression) [**not**] property_expr
| [**disable iff** (expression)] **not** multi_clock_property_expr

property_expr ::=
sequence_expr
| sequence_expr |-> [**not**] sequence_expr
| sequence_expr |=> [**not**] sequence_expr

multi_clock_property_expr ::=
multi_clock_sequence
| multi_clock_sequence |=> [**not**] multi_clock_sequence

sequence_declaration ::=
sequence sequence_identifier [sequence_formal_list] ;
{ assertion_variable_declaration }
sequence_spec ;
endsequence [: sequence_identifier]

sequence_formal_list ::=
(formal_list_item { , formal_list_item })

sequence_spec ::=
multi_clock_sequence
| sequence_expr

multi_clock_sequence::=
clocked_sequence { ## clocked_sequence }

clocked_sequence ::=
clocking_event sequence_expr

sequence_expr ::=
cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
| sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
| expression { , function_blocking_assignment } [boolean_abbrev]
| (expression {, function_blocking_assignment }) [boolean_abbrev]
| sequence_instance [sequence_abbrev]
| (sequence_expr) [sequence_abbrev]
| sequence_expr **and** sequence_expr
| sequence_expr **intersect** sequence_expr
| sequence_expr **or** sequence_expr
| **first_match** (sequence_expr)
| expression **throughout** sequence_expr
| sequence_expr **within** sequence_expr

cycle_delay_range ::=
constant_expression
| ## [cycle_delay_const_range_expression]

sequence_instance ::=
sequence_identifier [(actual_arg_list)]

formal_list_item ::=
formal_identifier [= actual_arg_expr]

actual_arg_list ::=
(actual_arg_expr { , actual_arg_expr })
| (. formal_identifier (actual_arg_expr) { , . formal_identifier (actual_arg_expr) })

actual_arg_expr ::=

event_expression

boolean_abbrev ::=
consecutive_repetition
| non_consecutive_repetition
| goto_repetition

sequence_abbrev ::= consecutive_repetition

consecutive_repetition ::= **[*** const_or_range_expression **]**

non_consecutive_repetition ::= **[*=** const_or_range_expression **]**

goto_repetition ::= **[*->** const_or_range_expression **]**

const_or_range_expression ::=
constant_expression
| cycle_delay_const_range_expression

cycle_delay_const_range_expression ::=
constant_expression **:** constant_expression
| constant_expression **: $**

assertion_variable_declaration ::=
data_type list_of_variable_identifiers ;

A.3 Primitive instances

A.3.1 Primitive instantiation and instances

gate_instantiation ::=
cmos_switchtype [delay3] cmos_switch_instance { **,** cmos_switch_instance } **;**
| enable_gatetype [drive_strength] [delay3] enable_gate_instance { **,** enable_gate_instance } **;**
| mos_switchtype [delay3] mos_switch_instance { **,** mos_switch_instance } **;**
| n_input_gatetype [drive_strength] [delay2] n_input_gate_instance { **,** n_input_gate_instance } **;**
| n_output_gatetype [drive_strength] [delay2] n_output_gate_instance
{ **,** n_output_gate_instance } **;**
| pass_en_switchtype [delay2] pass_enable_switch_instance { , pass_enable_switch_instance } **;**
| pass_switchtype pass_switch_instance { **,** pass_switch_instance } **;**
| **pulldown** [pulldown_strength] pull_gate_instance { **,** pull_gate_instance } **;**
| **pullup** [pullup_strength] pull_gate_instance { **,** pull_gate_instance } **;**

cmos_switch_instance ::= [name_of_gate_instance] **(** output_terminal **,** input_terminal **,**
ncontrol_terminal **,** pcontrol_terminal **)**

enable_gate_instance ::= [name_of_gate_instance] **(** output_terminal **,** input_terminal **,** enable_terminal **)**

mos_switch_instance ::= [name_of_gate_instance] **(** output_terminal **,** input_terminal **,** enable_terminal **)**

n_input_gate_instance ::= [name_of_gate_instance] **(** output_terminal **,** input_terminal { **,** input_terminal } **)**

n_output_gate_instance ::= [name_of_gate_instance] **(** output_terminal { **,** output_terminal } **,**
input_terminal **)**

pass_switch_instance ::= [name_of_gate_instance] **(** inout_terminal **,** inout_terminal **)**

pass_enable_switch_instance ::= [name_of_gate_instance] **(** inout_terminal , inout_terminal **,**
enable_terminal **)**

pull_gate_instance ::= [name_of_gate_instance] **(** output_terminal **)**

name_of_gate_instance ::= gate_instance_identifier { range }

A.3.2 Primitive strengths

pulldown_strength ::=
(strength0 **,** strength1 **)**
| **(** strength1 **,** strength0 **)**
| **(** strength0 **)**

pullup_strength ::=
(strength0 , strength1)
| (strength1 , strength0)
| (strength1)

A.3.3 Primitive terminals

enable_terminal ::= expression

inout_terminal ::= net_lvalue

input_terminal ::= expression

ncontrol_terminal ::= expression

output_terminal ::= net_lvalue

pcontrol_terminal ::= expression

A.3.4 Primitive gate and switch types

cmos_switchtype ::= **cmos** | **rcmos**

enable_gatetype ::= **bufif0** | **bufif1** | **notif0** | **notif1**

mos_switchtype ::= **nmos** | **pmos** | **rnmos** | **rpmos**

n_input_gatetype ::= **and** | **nand** | **or** | **nor** | **xor** | **xnor**

n_output_gatetype ::= **buf** | **not**

pass_en_switchtype ::= **tranif0** | **tranif1** | **rtranif1** | **rtranif0**

pass_switchtype ::= **tran** | **rtran**

A.4 Module, interface and generated instantiation

A.4.1 Instantiation

A.4.1.1 Module instantiation

module_instantiation ::=
module_identifier [parameter_value_assignment] module_instance { , module_instance } ;

parameter_value_assignment ::= # (list_of_parameter_assignments)

list_of_parameter_assignments ::=
ordered_parameter_assignment { , ordered_parameter_assignment }
| named_parameter_assignment { , named_parameter_assignment }

ordered_parameter_assignment ::= expression | data_type

named_parameter_assignment ::=
. parameter_identifier ([expression])
| . parameter_identifier (data_type)

module_instance ::= name_of_instance ([list_of_port_connections])

name_of_instance ::= module_instance_identifier { range }

list_of_port_connections ::=
ordered_port_connection { , ordered_port_connection }
| dot_named_port_connection { , dot_named_port_connection }
| { named_port_connection , } dot_star_port_connection { , named_port_connection }

ordered_port_connection ::= { attribute_instance } [expression]

named_port_connection ::= { attribute_instance } . port_identifier ([expression])

dot_named_port_connection ::=
{ attribute_instance } .port_identifier
| named_port_connection

dot_star_port_connection ::= { attribute_instance } .*

A.4.1.2 Interface instantiation

interface_instantiation ::=
interface_identifier [parameter_value_assignment] module_instance { **,** module_instance } **;**

A.4.4.1 Program instantiation

program_instantiation ::=
program_identifier [parameter_value_assignment] program_instance { **,** program_instance } **;**

program_instance ::= program_instance_identifier { range } **(** [list_of_port_connections] **)**

A.4.2 Generated instantiation

A.4.2.1 Generated module instantiation

generated_module_instantiation ::= **generate** { generate_module_item } **endgenerate**

generate_module_item ::=
generate_module_conditional_statement
| generate_module_case_statement
| generate_module_loop_statement
| [generate_block_identifier :] generate_module_block
| module_or_generate_item

generate_module_conditional_statement ::=
if (constant_expression) generate_module_item [**else** generate_module_item]

generate_module_case_statement ::=
case (constant_expression **)** genvar_module_case_item { genvar_module_case_item }**endcase**

genvar_module_case_item ::=
constant_expression { **,** constant_expression } **:** generate_module_item
| **default** [**:**] generate_module_item

generate_module_loop_statement ::=
for (genvar_decl_assignment **;** constant_expression **;** genvar_assignment **)**
generate_module_named_block

genvar_assignment ::=
genvar_identifier assignment_operator constant_expression
| inc_or_dec_operator genvar_identifier
| genvar_identifier inc_or_dec_operator

genvar_decl_assignment ::=
[**genvar**] genvar_identifier **=** constant_expression

generate_module_named_block ::=
begin : generate_block_identifier { generate_module_item } **end** [: generate_block_identifier]
| generate_block_identifier **:** generate_module_block

generate_module_block ::=
begin [**:** generate_block_identifier] { generate_module_item } **end** [**:** generate_block_identifier]

A.4.2.2 Generated interface instantiation

generated_interface_instantiation ::= **generate** { generate_interface_item } **endgenerate**

generate_interface_item ::=
generate_interface_conditional_statement
| generate_interface_case_statement
| generate_interface_loop_statement
| [generate_block_identifier :] generate_interface_block
| interface_or_generate_item

generate_interface_conditional_statement ::=
if (constant_expression) generate_interface_item [**else** generate_interface_item]

generate_interface_case_statement ::=
case (constant_expression) genvar_interface_case_item { genvar_interface_case_item } **endcase**

genvar_interface_case_item ::=
constant_expression { **,** constant_expression } **:** generate_interface_item
| **default** [**:**] generate_interface_item

generate_interface_loop_statement ::=
for (genvar_decl_assignment **;** constant_expression **;** genvar_assignment **)**
generate_interface_named_block

generate_interface_named_block ::=
begin : generate_block_identifier { generate_interface_item } **end** [: generate_block_identifier]
| generate_block_identifier **:** generate_interface_block

generate_interface_block ::=
begin [**:** generate_block_identifier]
{ generate_interface_item }
end [**:** generate_block_identifier]

A.5 UDP declaration and instantiation

A.5.1 UDP declaration

udp_nonansi_declaration ::=
{ attribute_instance } **primitive** udp_identifier **(** udp_port_list **) ;**

udp_ansi_declaration ::=
{ attribute_instance } **primitive** udp_identifier **(** udp_declaration_port_list **) ;**

udp_declaration ::=
udp_nonansi_declaration udp_port_declaration { udp_port_declaration }
udp_body
endprimitive [: udp_identifier]
| udp_ansi_declaration udp_body **endprimitive** [**:** udp_identifier]
| **extern** udp_nonansi_declaration
| **extern** udp_ansi_declaration
| { attribute_instance } **primitive** udp_identifier **(.*) ;**
{ udp_port_declaration } udp_body **endprimitive** [**:** udp_identifier]

udp_declaration ::=
{ attribute_instance } **primitive** udp_identifier **(** udp_port_list **) ;**
udp_port_declaration { udp_port_declaration }
udp_body
endprimitive
| { attribute_instance } **primitive** udp_identifier **(** udp_declaration_port_list **) ;**
udp_body
endprimitive

A.5.2 UDP ports

udp_port_list ::= output_port_identifier **,** input_port_identifier { **,** input_port_identifier }

udp_declaration_port_list ::= udp_output_declaration **,** udp_input_declaration { **,** udp_input_declaration }

udp_port_declaration ::=
udp_output_declaration ;
| udp_input_declaration ;
| udp_reg_declaration ;

udp_output_declaration ::=
{ attribute_instance } **output** port_identifier
| { attribute_instance } **output reg** port_identifier [**=** constant_expression]

udp_input_declaration ::= { attribute_instance } **input** list_of_udp_port_identifiers
udp_reg_declaration ::= { attribute_instance } **reg** variable_identifier

A.5.3 UDP body

udp_body ::= combinational_body | sequential_body
combinational_body ::= **table** combinational_entry { combinational_entry } **endtable**
combinational_entry ::= level_input_list **:** output_symbol **;**
sequential_body ::= [udp_initial_statement] **table** sequential_entry { sequential_entry } **endtable**
udp_initial_statement ::= **initial** output_port_identifier **=** init_val **;**
init_val ::= **1'b0** | **1'b1** | **1'bx** | **1'bX** | **1'B0** | **1'B1** | **1'Bx** | **1'BX** | **1** | **0**
sequential_entry ::= seq_input_list **:** current_state **:** next_state **;**
seq_input_list ::= level_input_list | edge_input_list
level_input_list ::= level_symbol { level_symbol }
edge_input_list ::= { level_symbol } edge_indicator { level_symbol }
edge_indicator ::= (level_symbol level_symbol) | edge_symbol
current_state ::= level_symbol
next_state ::= output_symbol | **-**
output_symbol ::= **0** | **1** | **x** | **X**
level_symbol ::= **0** | **1** | **x** | **X** | **?** | **b** | **B**
edge_symbol ::= **r** | **R** | **f** | **F** | **p** | **P** | **n** | **N** | *****

A.5.4 UDP instantiation

udp_instantiation ::= udp_identifier [drive_strength] [delay2] udp_instance { **,** udp_instance } **;**
udp_instance ::= [name_of_udp_instance] { range } **(** output_terminal **,** input_terminal { **,** input_terminal } **)**
name_of_udp_instance ::= udp_instance_identifier { range }

A.6 Behavioral statements

A.6.1 Continuous assignment and net alias statements

continuous_assign ::=
  assign [drive_strength] [delay3] list_of_net_assignments **;**
| **assign** [delay_control] list_of_variable_assignments **;**

list_of_net_assignments ::= net_assignment { **,** net_assignment }
list_of_variable_assignments ::= variable_assignment { **,** variable_assignment }
net_alias ::= **alias** net_lvalue **=** net_lvalue **;**
net_assignment ::= net_lvalue **=** expression

A.6.2 Procedural blocks and assignments

initial_construct ::= **initial** statement_or_null
always_construct ::= **always** statement
combinational_construct ::= **always_comb** statement
latch_construct ::= **always_latch** statement
ff_construct ::= **always_ff** statement
final_construct ::= **final** function_statement
blocking_assignment ::=
  variable_lvalue **=** delay_or_event_control expression
| hierarchical_variable_identifier **= new [** constant_expression **]** [**(** variable_identifier **)**]
| class_identifier [parameter_value_assignment] **= new** [**(** list_of_arguments **)**]

| class_identifier **. randomize** [**( )**] **with** constraint_block **;**
| operator_assignment

operator_assignment ::= variable_lvalue assignment_operator expression

assignment_operator ::=
= | += | -= | *= | /= | %= | &= | |= | ^= | <<= | >>= | <<<= | >>>=

nonblocking_assignment ::= variable_lvalue <= [delay_or_event_control] expression

procedural_continuous_assignments ::=
assign variable_assignment
| **deassign** variable_lvalue
| **force** variable_assignment
| **force** net_assignment
| **release** variable_lvalue
| **release** net_lvalue

function_blocking_assignment ::= variable_lvalue = expression

function_statement_or_null ::=
function_statement
| { attribute_instance } **;**

variable_assignment ::=
operator_assignment
| inc_or_dec_expression

A.6.3 Parallel and sequential blocks

action_block ::=
statement_or_null
| [statement] **else** statement_or_null

function_seq_block ::=
begin [**:** block_identifier { block_item_declaration }] { function_statement_or_null }
end [**:** block_identifier]

seq_block ::=
begin [**:** block_identifier] { block_item_declaration } { statement_or_null }
end [**:** block_identifier]

par_block ::=
fork [**:** block_identifier] { block_item_declaration } { statement_or_null }
join_keyword [**:** block_identifier]

join_keyword ::= **join** | **join_any** | **join_none**

A.6.4 Statements

statement_or_null ::=
statement
| { attribute_instance } **;**

statement ::= [block_identifier **:**] statement_item

statement_item ::=
{ attribute_instance } blocking_assignment **;**
| { attribute_instance } nonblocking_assignment **;**
| { attribute_instance } procedural_continuous_assignments **;**
| { attribute_instance } case_statement
| { attribute_instance } conditional_statement
| { attribute_instance } inc_or_dec_expression **;**
| { attribute_instance } function_call **;**
| { attribute_instance } disable_statement
| { attribute_instance } event_trigger

| { attribute_instance } loop_statement
| { attribute_instance } jump_statement
| { attribute_instance } par_block
| { attribute_instance } procedural_timing_control_statement
| { attribute_instance } seq_block
| { attribute_instance } system_task_enable
| { attribute_instance } task_enable
| { attribute_instance } wait_statement
| { attribute_instance } procedural_assertion_item
| { attribute_instance } clocking_drive

function_statement ::= [block_identifier **:**] function_statement_item

function_statement_item ::=
{ attribute_instance } function_blocking_assignment **;**
| { attribute_instance } function_case_statement
| { attribute_instance } function_conditional_statement
| { attribute_instance } inc_or_dec_expression **;**
| { attribute_instance } function_call **;**
| { attribute_instance } function_loop_statement
| { attribute_instance } jump_statement
| { attribute_instance } function_seq_block
| { attribute_instance } disable_statement
| { attribute_instance } system_task_enable

A.6.5 Timing control statements

procedural_timing_control_statement ::=
procedural_timing_control statement_or_null

delay_or_event_control ::=
delay_control
| event_control
| **repeat (** expression **)** event_control

delay_control ::=
delay_value
| **# (** mintypmax_expression **)**

event_control ::=
@ event_identifier
| **@ (** event_expression **)**
| **@***
| **@ (*)**

event_expression ::=
[edge_identifier] expression [**iff** expression]
| event_expression **or** event_expression
| event_expression **,** event_expression

procedural_timing_control ::=
delay_control
| event_control

jump_statement ::=
return [expression] **;**
| **break ;**
| **continue ;**

wait_statement ::=
wait (expression **)** statement_or_null
| **wait fork ;**

| **wait_order (** hierarchical_identifier [**,** hierarchical_identifier] **)** action_block

event_trigger ::=
-> hierarchical_event_identifier **;**
|**->>** [delay_or_event_control] hierarchical_event_identifier **;**

disable_statement ::=
disable hierarchical_task_identifier **;**
| **disable** hierarchical_block_identifier **;**
| **disable fork ;**

A.6.6 Conditional statements

conditional_statement ::=
[unique_priority] **if (** expression **)** statement_or_null [**else** statement_or_null]
| if_else_if_statement

if_else_if_statement ::=
[unique_priority] **if (** expression **)** statement_or_null
{ **else** [unique_priority] **if (** expression **)** statement_or_null }
[**else** statement_or_null]

function_conditional_statement ::=
[unique_priority] **if (** expression **)** function_statement_or_null
[**else** function_statement_or_null]
| function_if_else_if_statement

function_if_else_if_statement ::=
[unique_priority] **if (** expression **)** function_statement_or_null
{ **else** [unique_priority] **if (** expression **)** function_statement_or_null }
[**else** function_statement_or_null]

unique_priority ::= **unique** | **priority**

A.6.7 Case statements

case_statement ::=
[unique_priority] **case (** expression **)** case_item { case_item } **endcase**
| [unique_priority] **casez (** expression **)** case_item { case_item } **endcase**
| [unique_priority] **casex (** expression **)** case_item { case_item } **endcase**

case_item ::=
expression { **,** expression } **:** statement_or_null
| **default** [**:**] statement_or_null

function_case_statement ::=
[unique_priority] **case (** expression **)** function_case_item { function_case_item } **endcase**
| [unique_priority] **casez (** expression **)** function_case_item { function_case_item } **endcase**
| [unique_priority] **casex (** expression **)** function_case_item { function_case_item } **endcase**

function_case_item ::=
expression { **,** expression } **:** function_statement_or_null
| **default** [**:**] function_statement_or_null

A.6.8 Looping statements

function_loop_statement ::=
forever function_statement_or_null
| **repeat (** expression **)** function_statement_or_null
| **while (** expression **)** function_statement_or_null
| **for (** variable_decl_or_assignment { **,** variable_decl_or_assignment } **;** expression **;**
variable_assignment { **,** variable_assignment } **)** function_statement_or_null
| **do** function_statement_or_null **while (** expression **) ;**

loop_statement ::=

forever statement_or_null
| **repeat (** expression **)** statement_or_null
| **while (** expression **)** statement_or_null
| **for (** variable_decl_or_assignment **;** expression **;** variable_assignment **)** statement_or_null
| **for (** variable_decl_or_assignment { **,** variable_decl_or_assignment } **;** expression **;**
variable_assignment { **,** variable_assignment } **)** statement_or_null
| **do** statement_or_null **while (** expression **) ;**

variable_decl_or_assignment ::=
data_type list_of_variable_identifiers_or_assignments
| variable_assignment

A.6.9 Task enable statements

system_task_enable ::= system_task_identifier [**(** [expression] { **,** [expression] } **)**] **;**

task_enable ::= hierarchical_task_identifier [**(** list_of_arguments **)**] **;**

A.6.10 Assertion statements

procedural_assertion_item ::=
assert_property_statement
| cover_property_statement
| immediate_assert_statement

immediate_assert_statement ::=
assert (expression **)** action_block

A.6.11 Clocking domain

clocking_decl ::= [**default**] **clocking** [clocking_identifier] clocking_event **;**
{ clocking_item }
endclocking

clocking_event ::=
@ identifier
| **@ (** event_expression **)**

clocking_item :=
default default_skew **;**
| clocking_direction list_of_clocking_decl_assign **;**
| { attribute_instance } concurrent_assertion_item_declaration

default_skew ::=
input clocking_skew
| **output** clocking_skew
| **input** clocking_skew **output** clocking_skew

clocking_direction ::=
input [clocking_skew]
| **output** [clocking_skew]
| **input** [clocking_skew] **output** [clocking_skew]
| **inout**

list_of_clocking_decl_assign ::= clocking_decl_assign { , clocking_decl_assign }

clocking_decl_assign ::= signal_identifier [= hierarchical_identifier]

clocking_skew ::=
edge_identifier [delay_control]
| delay_control

clocking_drive ::=
clockvar_expression <= [cycle_delay] expression
| cycle_delay clockvar_expression <= expression

cycle_delay ::= ## expression

clockvar ::= clocking_identifier . identifier

clockvar_expression ::=
    clockvar range
  | clockvar [range_expression]

A.7 Specify section

A.7.1 Specify block declaration

specify_block ::= **specify** { specify_item } **endspecify**

specify_item ::=
    specparam_declaration
  | pulsestyle_declaration
  | showcancelled_declaration
  | path_declaration
  | system_timing_check

pulsestyle_declaration ::=
    pulsestyle_onevent list_of_path_outputs **;**
  | **pulsestyle_ondetect** list_of_path_outputs **;**

showcancelled_declaration ::=
    showcancelled list_of_path_outputs **;**
  | **noshowcancelled** list_of_path_outputs **;**

A.7.2 Specify path declarations

path_declaration ::=
    simple_path_declaration **;**
  | edge_sensitive_path_declaration **;**
  | state_dependent_path_declaration **;**

simple_path_declaration ::=
    parallel_path_description = path_delay_value
  | full_path_description = path_delay_value

parallel_path_description ::=
    (specify_input_terminal_descriptor [polarity_operator] => specify_output_terminal_descriptor **)**

full_path_description ::=
    (list_of_path_inputs [polarity_operator] *> list_of_path_outputs **)**

list_of_path_inputs ::=
    specify_input_terminal_descriptor { **,** specify_input_terminal_descriptor }

list_of_path_outputs ::=
    specify_output_terminal_descriptor { **,** specify_output_terminal_descriptor }

A.7.3 Specify block terminals

specify_input_terminal_descriptor ::=
    input_identifier [**[** constant_range_expression **]**]

specify_output_terminal_descriptor ::=
    output_identifier [**[** constant_range_expression **]**]

input_identifier ::= input_port_identifier | inout_port_identifier

output_identifier ::= output_port_identifier | inout_port_identifier

A.7.4 Specify path delays

path_delay_value ::=
    list_of_path_delay_expressions
  | **(** list_of_path_delay_expressions **)**

list_of_path_delay_expressions ::=
t_path_delay_expression
| trise_path_delay_expression , tfall_path_delay_expression
| trise_path_delay_expression , tfall_path_delay_expression , tz_path_delay_expression
| t01_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
tz1_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression
| t01_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
tz1_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression ,
t0x_path_delay_expression , tx1_path_delay_expression , t1x_path_delay_expression ,
tx0_path_delay_expression , txz_path_delay_expression , tzx_path_delay_expression

t_path_delay_expression ::= path_delay_expression

trise_path_delay_expression ::= path_delay_expression

tfall_path_delay_expression ::= path_delay_expression

tz_path_delay_expression ::= path_delay_expression

t01_path_delay_expression ::= path_delay_expression

t10_path_delay_expression ::= path_delay_expression

t0z_path_delay_expression ::= path_delay_expression

tz1_path_delay_expression ::= path_delay_expression

t1z_path_delay_expression ::= path_delay_expression

tz0_path_delay_expression ::= path_delay_expression

t0x_path_delay_expression ::= path_delay_expression

tx1_path_delay_expression ::= path_delay_expression

t1x_path_delay_expression ::= path_delay_expression

tx0_path_delay_expression ::= path_delay_expression

txz_path_delay_expression ::= path_delay_expression

tzx_path_delay_expression ::= path_delay_expression

path_delay_expression ::= constant_mintypmax_expression

edge_sensitive_path_declaration ::=
parallel_edge_sensitive_path_description = path_delay_value
| full_edge_sensitive_path_description = path_delay_value

parallel_edge_sensitive_path_description ::=
([edge_identifier] specify_input_terminal_descriptor =>
specify_output_terminal_descriptor [polarity_operator] : data_source_expression)

full_edge_sensitive_path_description ::=
([edge_identifier] list_of_path_inputs *>
list_of_path_outputs [polarity_operator] : data_source_expression)

data_source_expression ::= expression

edge_identifier ::= **posedge** | **negedge**

state_dependent_path_declaration ::=
if (module_path_expression **)** simple_path_declaration
| **if (** module_path_expression **)** edge_sensitive_path_declaration
| **ifnone** simple_path_declaration

polarity_operator ::= + | -

A.7.5 System timing checks

A.7.5.1 System timing check commands

system_timing_check ::=


```
        $setup_timing_check
      | $hold_timing_check
      | $setuphold_timing_check
      | $recovery_timing_check
      | $removal_timing_check
      | $recrem_timing_check
      | $skew_timing_check
      | $timeskew_timing_check
      | $fullskew_timing_check
      | $period_timing_check
      | $width_timing_check
      | $nochange_timing_check
$setup_timing_check ::=
        $setup ( data_event , reference_event , timing_check_limit [ , [ notify_reg ] ] ) ;
$hold_timing_check ::=
        $hold ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
$setuphold_timing_check ::=
        $setuphold ( reference_event , data_event , timing_check_limit , timing_check_limit
          [ , [ notify_reg ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
          [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ] ) ;
$recovery_timing_check ::=
        $recovery ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
$removal_timing_check ::=
        $removal ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
$recrem_timing_check ::=
        $recrem ( reference_event , data_event , timing_check_limit , timing_check_limit
          [ , [ notify_reg ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
          [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ] ) ;
$skew_timing_check ::=
        $skew ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
$timeskew_timing_check ::=
        $timeskew ( reference_event , data_event , timing_check_limit
          [ , [ notify_reg ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ) ;
$fullskew_timing_check ::=
        $fullskew ( reference_event , data_event , timing_check_limit , timing_check_limit
          [ , [ notify_reg ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ) ;
$period_timing_check ::=
        $period ( controlled_reference_event , timing_check_limit [ , [ notify_reg ] ] ) ;
$width_timing_check ::=
        $width ( controlled_reference_event , timing_check_limit , threshold [ , [ notify_reg ] ] ) ;
$nochange_timing_check ::=
        $nochange ( reference_event , data_event , start_edge_offset ,
          end_edge_offset [ , [ notify_reg ] ] ) ;
```

A.7.5.2 System timing check command arguments

```
checktime_condition ::= mintypmax_expression
controlled_reference_event ::= controlled_timing_check_event
data_event ::= timing_check_event
delayed_data ::=
        terminal_identifier
```

| terminal_identifier **[** constant_mintypmax_expression **]**

delayed_reference ::=
terminal_identifier
| terminal_identifier **[** constant_mintypmax_expression **]**

end_edge_offset ::= mintypmax_expression

event_based_flag ::= constant_expression

notify_reg ::= variable_identifier

reference_event ::= timing_check_event

remain_active_flag ::= constant_mintypmax_expression

stamptime_condition ::= mintypmax_expression

start_edge_offset ::= mintypmax_expression

threshold ::=constant_expression

timing_check_limit ::= expression

A.7.5.3 System timing check event definitions

timing_check_event ::=
[timing_check_event_control] specify_terminal_descriptor [**&&&** timing_check_condition]

controlled_timing_check_event ::=
timing_check_event_control specify_terminal_descriptor [**&&&** timing_check_condition]

timing_check_event_control ::=
posedge
| **negedge**
| edge_control_specifier

specify_terminal_descriptor ::=
specify_input_terminal_descriptor
| specify_output_terminal_descriptor

edge_control_specifier ::= **edge [** edge_descriptor { **,** edge_descriptor } **]**

edge_descriptor[1] ::= **01** | **10** | z_or_x zero_or_one | zero_or_one z_or_x

zero_or_one ::= **0** | **1**

z_or_x ::= **x** | **X** | **z** | **Z**

timing_check_condition ::=
scalar_timing_check_condition
| **(** scalar_timing_check_condition **)**

scalar_timing_check_condition ::=
expression
| **~** expression
| expression **==** scalar_constant
| expression **===** scalar_constant
| expression **!=** scalar_constant
| expression **!==** scalar_constant

scalar_constant ::= **1'b0** | **1'b1** | **1'B0** | **1'B1** | **'b0** | **'b1** | **'B0** | **'B1** | **1** | **0**

A.8 Expressions

A.8.1 Concatenations

concatenation ::=
{ expression { **,** expression } **}**
| **{** struct_member_label **:** expression { **,** struct_member_label : expression } **}**
| **{** array_member_label **:** expression { **,** array_member_label : expression } **}**

constant_concatenation ::=
{ constant_expression { , constant_expression } }
| { struct_member_label : constant_expression { , struct_member_label : constant_expression } }
| { array_member_label : constant_expression { , array_member_label : constant_expression } }

struct_member_label ::=
default
| type_identifier
| variable_identifier

array_member_label ::=
default
| type_identifier
| constant_expression

constant_multiple_concatenation ::= { constant_expression constant_concatenation }

module_path_concatenation ::= { module_path_expression { , module_path_expression } }

module_path_multiple_concatenation ::= { constant_expression module_path_concatenation }

multiple_concatenation ::= { constant_expression concatenation }

A.8.2 Function calls

constant_function_call ::= function_identifier { attribute_instance }
[(list_of_constant_arguments)]

function_call ::= hierarchical_function_identifier { attribute_instance } [(list_of_arguments)]

list_of_arguments ::=
[expression] { , [expression] }
| . identifier ([expression]) { , . identifier ([expression]) }

list_of_constant_arguments ::=
[constant_expression] { , [constant_expression] }
| . identifier ([constant_expression]) { , . identifier ([constant_expression]) }

system_function_call ::= system_function_identifier [(expression { , expression })]

A.8.3 Expressions

base_expression ::= expression

inc_or_dec_expression ::=
inc_or_dec_operator { attribute_instance } variable_lvalue
| variable_lvalue { attribute_instance } inc_or_dec_operator

conditional_expression ::= expression1 ? { attribute_instance } expression2 : expression3

constant_base_expression ::= constant_expression

constant_expression ::=
constant_primary
| unary_operator { attribute_instance } constant_primary
| constant_expression binary_operator { attribute_instance } constant_expression
| constant_expression ? { attribute_instance } constant_expression : constant_expression
| string_literal

constant_mintypmax_expression ::=
constant_expression
| constant_expression : constant_expression : constant_expression

constant_param_expression ::=
constant_expression

constant_range_expression ::=
constant_expression
| msb_constant_expression : lsb_constant_expression

| constant_base_expression **+:** width_constant_expression
| constant_base_expression **-:** width_constant_expression

dimension_constant_expression ::= constant_expression

expression1 ::= expression

expression2 ::= expression

expression3 ::= expression

expression ::=
primary
| unary_operator { attribute_instance } primary
| inc_or_dec_expression
| **(** operator_assignment **)**
| expression binary_operator { attribute_instance } expression
| conditional_expression
| string_literal
| inside_expression

inside_expression ::= expression **inside** range_list_or_array

range_list_or_array ::=
variable_identifier
| **{** value_range { **,** value_range } **}**

value_range ::=
expression
| **[** expression **:** expression **]**

lsb_constant_expression ::= constant_expression

mintypmax_expression ::=
expression
| expression **:** expression **:** expression

module_path_conditional_expression ::= module_path_expression **?** { attribute_instance }
module_path_expression **:** module_path_expression

module_path_expression ::=
module_path_primary
| unary_module_path_operator { attribute_instance } module_path_primary
| module_path_expression binary_module_path_operator { attribute_instance }
module_path_expression
| module_path_conditional_expression

module_path_mintypmax_expression ::=
module_path_expression
| module_path_expression **:** module_path_expression **:** module_path_expression

msb_constant_expression ::= constant_expression

range_expression ::=
expression
| msb_constant_expression **:** lsb_constant_expression
| base_expression **+:** width_constant_expression
| base_expression **-:** width_constant_expression

width_constant_expression ::= constant_expression

A.8.4 Primaries

constant_primary ::=
constant_concatenation
| constant_function_call
| **(** constant_mintypmax_expression **)**

| constant_multiple_concatenation
| genvar_identifier
| number
| parameter_identifier
| specparam_identifier
| casting_type **’ (** constant_expression **)**
| casting_type **’** constant_concatenation
| casting_type **’** constant_multiple_concatenation
| time_literal
| **’0** | **’1** | **’z** | **’Z** | **’x** | **’X**

module_path_primary ::=
number
| identifier
| module_path_concatenation
| module_path_multiple_concatenation
| function_call
| system_function_call
| constant_function_call
| **(** module_path_mintypmax_expression **)**

primary ::=
number
| implicit_class_handle hierarchical_identifier { **[** expression **]** } [**[** range_expression **]**]
[**.** method_identifier { attribute_instance } [**(** expression { **,** expression } **)**]]
| concatenation
| multiple_concatenation
| function_call
| system_function_call
| constant_function_call
| class_identifier **::** { class_identifier **::** } identifier
| **(** mintypmax_expression **)**
| casting_type **’ (** expression **)**
| **void ’ (** function_call **)**
| casting_type **’** concatenation
| casting_type **’** multiple_concatenation
| time_literal
| **’0** | **’1** | **’z** | **’Z** | **’x** | **’X**
| **null**

time_literal[7] ::=
unsigned_number time_unit
| fixed_point_number time_unit

time_unit ::= **s** | **ms** | **us** | **ns** | **ps** | **fs** | **step**

implicit_class_handle[10] ::= [**this**.] | [**super**.]

A.8.5 Expression left-side values

net_lvalue ::=
hierarchical_net_identifier { **[** constant_expression **]** } [**[** constant_range_expression **]**]
| **{** net_lvalue { **,** net_lvalue } **}**

variable_lvalue ::=
hierarchical_variable_identifier { **[** expression **]** } [**[** range_expression **]**]
| **{** variable_lvalue { **,** variable_lvalue } **}**

A.8.6 Operators

unary_operator ::=
+ | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~

binary_operator ::=
+ | - | * | / | % | == | != | === | !== | =?= | !?= | && | || | **
| < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | << | >>> | <<<

inc_or_dec_operator ::= ++ | --

unary_module_path_operator ::=
! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~

binary_module_path_operator ::=
== | != | && | || | & | | | ^ | ^~ | ~^

A.8.7 Numbers

number ::=
decimal_number
| octal_number
| binary_number
| hex_number
| real_number

decimal_number ::=
unsigned_number
| [size] decimal_base unsigned_number
| [size] decimal_base x_digit { _ }
| [size] decimal_base z_digit { _ }

binary_number ::= [size] binary_base binary_value

octal_number ::= [size] octal_base octal_value

hex_number ::= [size] hex_base hex_value

sign ::= + | -

size ::= non_zero_unsigned_number

non_zero_unsigned_number[1] ::= non_zero_decimal_digit { _ | decimal_digit}

real_number[1] ::=
fixed_point_number
| unsigned_number [. unsigned_number] exp [sign] unsigned_number

fixed_point_number[1] ::= unsigned_number . unsigned_number

exp ::= e | E

unsigned_number[1] ::= decimal_digit { _ | decimal_digit }

binary_value[1] ::= binary_digit { _ | binary_digit }

octal_value[1] ::= octal_digit { _ | octal_digit }

hex_value[1] ::= hex_digit { _ | hex_digit }

decimal_base[1] ::= '[s|S]d | '[s|S]D

binary_base[1] ::= '[s|S]b | '[s|S]B

octal_base[1] ::= '[s|S]o | '[s|S]O

hex_base[1] ::= '[s|S]h | '[s|S]H

non_zero_decimal_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

decimal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

binary_digit ::= x_digit | z_digit | 0 | 1

octal_digit ::= x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

hex_digit ::= x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **a** | **b** | **c** | **d** | **e** | **f** | **A** | **B** | **C** | **D** | **E** | **F**

x_digit ::= **x** | **X**

z_digit ::= **z** | **Z** | **?**

A.8.8 Strings

string_literal ::= **"** { Any_ASCII_Characters } **"**

A.9 General

A.9.1 Attributes

attribute_instance ::= **(*** attr_spec { **,** attr_spec } ***)**

attr_spec ::=
    attr_name **=** constant_expression
  | attr_name

attr_name ::= identifier

A.9.2 Comments

comment ::=
    one_line_comment
  | block_comment

one_line_comment ::= **//** comment_text \n

block_comment ::= **/*** comment_text ***/**

comment_text ::= { Any_ASCII_character }

A.9.3 Identifiers

block_identifier ::= identifier

c_identifier[2] ::= [**a-zA-Z_**] { [**a-zA-Z0-9_**] }

cell_identifier ::= identifier

class_identifier ::= identifier

clocking_identifier ::= identifier

config_identifier ::= identifier

constraint_identifier ::= identifier

const_identifier ::= identifier

enum_identifier ::= identifier

escaped_hierarchical_identifier[4] ::=
    escaped_hierarchical_branch { **.**simple_hierarchical_branch | **.**escaped_hierarchical_branch }

escaped_identifier ::= **** {any_ASCII_character_except_white_space} white_space

event_identifier ::= identifier

formal_identifier ::= identifier

function_identifier ::= identifier

gate_instance_identifier ::= identifier

generate_block_identifier ::= identifier

genvar_identifier ::= identifier

hierarchical_block_identifier ::= hierarchical_identifier

hierarchical_event_identifier ::= hierarchical_identifier

hierarchical_function_identifier ::= hierarchical_identifier

hierarchical_identifier ::=
simple_hierarchical_identifier
| escaped_hierarchical_identifier

hierarchical_parameter_identifier ::= hierarchical_identifier

hierarchical_net_identifier ::= hierarchical_identifier

hierarchical_variable_identifier ::= hierarchical_identifier

hierarchical_task_identifier ::= hierarchical_identifier

identifier ::=
simple_identifier
| escaped_identifier

interface_identifier ::= identifier

interface_instance_identifier ::= identifier

inout_port_identifier ::= identifier

input_port_identifier ::= identifier

instance_identifier ::= identifier

library_identifier ::= identifier

method_identifier ::= identifier

modport_identifier ::= identifier

module_identifier ::= identifier

module_instance_identifier ::= identifier

net_identifier ::= identifier

output_port_identifier ::= identifier

parameter_identifier ::= identifier

port_identifier ::= identifier

program_identifier ::= identifier

program_instance_identifier ::= identifier

property_identifier ::= identifier

sequence_identifier ::= identifier

signal_identifier ::= identifier

simple_hierarchical_identifier[3] ::= simple_hierarchical_branch [**.** escaped_identifier]

simple_identifier[2] ::= [**a-zA-Z_**] { [**a-zA-Z0-9_$**] }

specparam_identifier ::= identifier

system_function_identifier[5] ::= **$**[**a-zA-Z0-9_$**]{ [**a-zA-Z0-9_$**] }

system_task_identifier[5] ::= **$**[**a-zA-Z0-9_$**]{ [**a-zA-Z0-9_$**] }

task_or_function_identifier ::= task_identifier | function_identifier

task_identifier ::= identifier

terminal_identifier ::= identifier

text_macro_identifier ::= simple_identifier

topmodule_identifier ::= identifier

type_declaration_identifier ::= type_identifier { unpacked_dimension }

type_identifier ::= identifier

udp_identifier ::= identifier

udp_instance_identifier ::= identifier

variable_identifier ::= identifier

A.9.4 Identifier branches

simple_hierarchical_branch[3] ::=
simple_identifier { **[** unsigned_number **]** } [{ **.** simple_identifier { **[** unsigned_number **]** } }]

escaped_hierarchical_branch[4] ::=
escaped_identifier { **[** unsigned_number **]** } [{ **.** escaped_identifier { **[** unsigned_number **]** } }]

A.9.5 White space

white_space ::= space | tab | newline | eof[6]

NOTES

1) Embedded spaces are illegal.

2) A simple_identifier, c_identifier, and arrayed_reference shall start with an alpha or underscore (_) character, shall have at least one character, and shall not have any spaces.

3) The period (.) in simple_hierarchical_identifier and simple_hierarchical_branch shall not be preceded or followed by white_space.

4) The period in escaped_hierarchical_identifier and escaped_hierarchical_branch shall be preceded by white_space, but shall not be followed by white_space.

5) The $ character in a system_function_identifier or system_task_identifier shall not be followed by white_space. A system_function_identifier or system_task_identifier shall not be escaped.

6) End of file.

7) The unsigned number or fixed point number in time_literal shall not be followed by a white_space.

8) void functions, non integer type functions, and functions with a typedef type cannot have a signing declaration.

9) Open-array (**[ ]**) form shall only be used with dpi_proto_formal

10) implicit_class_handle shall only appear within the scope of a class_declaration or extern_method_declaration.

11) In any one declaration, only one of **protected** or **local** is allowed, only one of **rand** or **randc** is allowed, and **static** and/or **virtual** can appear only once.

Annex B
Keywords

SystemVerilog reserves the following keywords:

alias‡	endprimitive	**modport**†	small
always	**endprogram**‡	module	**solve**‡
always_comb†	**endproperty**‡	nand	specify
always_ff†	endspecify	negedge	specparam
always_latch†	**endsequence**‡	**new**‡	**static**†
and	endtable	nmos	**string**‡
assert†	endtask	nor	strong0
assert_strobe†	**enum**†	noshowcancelled	strong1
assign	event	not	**struct**†
automatic	**export**†	notif0	**super**‡
before‡	**extends**‡	notif1	supply0
begin	**extern**†	**null**‡	supply1
bind‡	**final**‡	or	table
bit†	**first_match**‡	output	task
break†	for	**packed**†	**this**‡
buf	force	parameter	**throughout**‡
bufif0	forever	pmos	time
bufif1	fork	posedge	**timeprecision**†
byte†	**forkjoin**†	primitive	**timeunit**†
case	function	**priority**†	tran
casex	generate	**program**‡	tranif0
casez	genvar	**property**‡	tranif1
cell	highz0	**protected**‡	tri
chandle‡	highz1	pull0	tri0
class‡	if	pull1	tri1
clocking‡	**iff**†	pulldown	triand
cmos	ifnone	pullup	trior
config	**import**†	pulsestyle_onevent	trireg
const†	incdir	pulsestyle_ondetect	**type**†
constraint‡	include	**pure**‡	**typedef**†
context‡	initial	**rand**‡	**union**†
continue†	inout	**randc**‡	**unique**†
cover‡	input	rcmos	unsigned
deassign	**inside**‡	**ref**‡	use
default	instance	real	**var**‡
defparam	**int**†	realtime	vectored
design	integer	reg	**virtual**‡
disable	**interface**†	release	**void**†
dist‡	**intersect**‡	repeat	wait
do†	join	return	**wait_order**‡
edge	**join_any**‡	rnmos	wand
else	**join_none**‡	rpmos	weak0
end	large	rtran	weak1
endcase	liblist	rtranif0	while
endclass‡	library	rtranif1	wire
endclocking‡	**local**‡	scalared	**with**‡
endconfig	localparam	**sequence**‡	**within**‡
endfunction	**logic**†	**shortint**†	wor
endgenerate	**longint**†	**shortreal**†	xnor
endinterface†	macromodule	showcancelled	xor
endmodule	medium	signed	

† keywords added to the IEEE 1364 Verilog-2001 standard as part of SystemVerilog 3.0
‡ keywords added to the IEEE 1364 Verilog-2001 standard as part of SystemVerilog 3.1

Note: The keyword var is reserved for future extensions.

Annex C
Linked Lists

(Informative)

The List package implements a classic list data-structure, and is analogous to the STL (Standard Template Library) List container that is popular with C++ programmers. The container is defined as a parameterized class, meaning that it can be customized to hold data of any type.

C.1 List definitions

list—A list is a doubly linked list, where every element has a predecessor and successor. A list is a sequence that supports both forward and backward traversal, as well as amortized constant time insertion and removal of elements at the beginning, end, or middle.

container—A container is a collection of data of the same type. Containers are objects that contain and manage other data. Containers provide an associated iterator that allows access to the contained data.

iterator—Iterators are objects that represent positions of elements in a container. They play a role similar to that of an array subscript, and allow users to access a particular element, and to traverse through the container.

C.2 List declaration

The List package supports lists of any arbitrary predefined type, such as **`integer`**, **`string`**, or class object.

Any iterator that refers to the position of an element that is removed from a list becomes invalid, thus, unable to iterate over the list.

To declare a specific list, users must first include the generic List class declaration from the standard include area and then declare the specialized list type:

```
`include <List.vh>
...
List#(type) dl;   // dl is a List of 'type' elements
```

C.2.1 Declaring list variables

List variables are declared by providing a specialization of the generic list class:

```
List#(integer) il;              // Object il is a list of integer
typedef List#(Packet) PList;    // Class Plist is a list of Packet objects
```

The List specialization declares a list of the indicated type. The type used in the list declaration determines the type of the data stored in the list elements.

C.2.2 Declaring list iterators

List iterators are declared by providing a specialization of the generic List_Iterator class:

```
List_Iterator#(string) s;       // Object s is a list-of-string iterator
List_Iterator#(Packet) p, q;    // p and q are iterators to a list-of-Packet
```

C.3 Linked list class prototypes

The following class prototypes describe the generic List and List_Iterator classes. Only the public interface is included here.

C.3.1 List_Iterator class prototype


```
class List_Iterator#(parameter type T);
   extern function void next();
   extern function void prev();
   extern function int neq( List_Iterator#(T) iter );
   extern function int eq( List_Iterator#(T) iter );
   extern function T data();
endclass
```

C.3.2 List class prototype

```
class List#(parameter type T);
   extern function new();
   extern function int size();
   extern function int empty();
   extern function void push_front( T value );
   extern function void push_back( T value );
   extern function T front();
   extern function T back();
   extern function void pop_front();
   extern function void pop_back();
   extern function List_Iterator#(T) start();
   extern function List_Iterator#(T) finish();
   extern function void insert( List_Iterator#(T) position, T value );
   extern function void insert_range( List_Iterator#(T) position,
                                      first, last );
   extern function void erase( List_Iterator#(T) position );
   extern function void erase_range( List_Iterator#(T) first, last );
   extern function void set( List_Iterator#(T) first, last );
   extern function void swap( List#(T) lst );
   extern function void clear();
   extern function void purge();
endclass
```

C.4 List_Iterator methods

The List_Iterator class provides methods to iterate over the elements of lists. These methods are described below.

C.4.1 next()

```
function void next();
```

`next` changes the iterator so that it refers to the next element in the list.

C.4.2 prev()

```
function void prev();
```

`prev` changes the iterator so that it refers to the previous element in the list.

C.4.3 eq()

```
function int eq( List_Iterator#(T) iter );
```

`eq` compares two iterators, and returns 1 if both iterators refer to the same list element. Otherwise, it returns 0.

```
if( i1.eq(i2) ) $display("both iterators refer to the same element");
```

C.4.4 neq()

```
function int neq( List_Iterator#(T) iter );
```

`neq` is the negation of `eq()`; it compares two iterators, and returns 0 if both iterators refer to the same list element. Otherwise, it returns 1.

C.4.5 data()

```
function T data();
```

`data` returns the data stored in the element at the given iterator location.

C.5 List methods

The List class provides methods to query the size of the list, obtain iterators to the head or tail of the list, retrieve the data stored in the list, and methods to add, remove, and reorder the elements of the list.

C.5.1 size()

```
function int size();
```

`size` returns the number of elements stored in the list.

```
while ( list1.size > 0 ) begin   // loop while still elements in the list
   ...
end
```

C.5.2 empty()

```
function int empty();
```

`empty` returns 1 if the number elements stored in the list is zero, 0 otherwise.

```
if ( list1.empty )
   $display( "list is empty" );
```

C.5.3 push_front()

```
function void push_front( T value );
```

`push_front` inserts the specified value at the front of the list.

```
List#(int) numbers;
numbers.push_front(10);
numbers.push_front(5);     // numbers contains { 5 , 10 }
```

C.5.4 push_back()

```
function void push_back( T value );
```

`push_back` inserts the specified value at the end of the list.

```
List#(string) names;
names.push_back("Donald");
names.push_back("Mickey");    // names contains { "Donald", "Mickey" }
```


C.5.5 front()

```
function T front();
```

`front` returns the data stored in the first element of the list (valid only if the list is not empty).

C.5.6 back()

```
function T back();
```

`back` returns the data stored in the last element of the list (valid only if the list is not empty).

```
List#(int) numbers;
numbers.push_front(3);
numbers.push_front(2);
numbers.push_front(1);
$display( numbers.front, numbers.back );  // displays 1 3
```

C.5.7 pop_front()

```
function void pop_front();
```

`pop_front` removes the first element of the list. If the list is empty, this method is illegal and can generate an error.

C.5.8 pop_back()

```
function void pop_back();
```

`pop_back` removes the last element of the list. If the list is empty, this method is illegal and can generate an error.

```
while ( lp.size > 1 ) begin   // remove all but the center element from
                              // an odd-sized list lp
   lp.pop_front();
   lp.pop_back();
end
```

C.5.9 start()

```
function List_Iterator#(T) start();
```

`start` returns an iterator to the position of the first element in the list.

C.5.10 finish()

```
function List_Iterator#(T) finish();
```

`finish` returns an iterator to a position just past the last element in the list. The last element in the last can be accessed using `finish.prev`.

```
List#(int) lst;      // display contents of list lst in position order
for ( List_Iterator#(int) p = lst.start; p.neq(lst.finish); p.next )
   $display( p.data );
```

C.5.11 insert()

```
function void insert( List_Iterator#(T) position, T value );
```

`insert` inserts the given `data (value)` into the list at the position specified by the iterator (before the element, if any, that was previously at the iterator's position). If the iterator is not a valid position within the list, then this operation is illegal and can generate an error.

```
function void add_sort( List#(byte) L, byte value );
   for ( List_Iterator#(byte) p = L.start; p.neq(L.finish) ; p.next )
      if ( p.data > value ) begin
         lst.insert( p, value ); // Add to sorted list (ascending order)
      return;
   end
endfunction
```

C.5.12 insert_range()

```
function void insert_range( List_Iterator#(T) position, first, last );
```

`insert_range` inserts the elements contained in the list range specified by the iterators first and last at the specified list position (before the element, if any, that was previously at the position iterator). All the elements from first up to, but not including, last are inserted into the list. If the last iterator refers to an element before the first iterator, the range wraps around the end of the list. The range iterators can specify a range either in another list or in the same list as being inserted.

If the position iterator is not a valid position within the list, or if the range iterators are invalid (i.e., they refer to different lists or to invalid positions), then this operation is illegal and can generate an error.

C.5.13 erase()

```
function void erase( List_Iterator#(T) position );
```

`erase` removes form the list the element at the specified position. After **`erase()`** returns, the position iterator becomes invalid.

```
list1.erase( list1.start );   // same as pop_front
```

If the position iterator is not a valid position within the list, this operation is illegal and can generate an error.

C.5.14 erase_range()

```
function void erase_range( List_Iterator#(T) first, last );
```

`erase_range` removes from a list the range of elements specified by the first and last iterators. This operation removes elements from the first iterator's position up to, but not including, the last iterator's position. If the last iterator refers to an element before the first iterator, the range wraps around the end of the list.

```
list1.erase_range( list1.start, list1.finish ); // Remove all elements from
                                                 // list1
```

If the range iterators are invalid (i.e., they refer to different lists or to invalid positions), then this operation is illegal and can generate an error.

C.5.15 set()

```
function void set( List_Iterator#(T) first, last );
```

`set` assigns to the list object the elements that lie in the range specified by the first and last iterators. After this method returns, the modified list shall have a size equal to the range specified by first and last. This method copies the data from the first iterator's position up to, but not including, the last iterator's position. If the last iterator refers to an element before the first iterator, the range wraps around the end of the list.

```
list2.set( list1.start, list2.finish );      // list2 is a copy of list1
```


If the range iterators are invalid (i.e., they refer to different lists or to invalid positions), then this operation is illegal and can generate an error.

C.5.16 swap()

```
function void swap( List#(T) lst );
```

`swap` exchanges the contents of two equal-size lists.

```
list1.swap( list2 ); // swap the contents of list1 to list2 and vice-versa
```

Swapping a list with itself has no effect. If the lists are of different sizes, this method can issue a warning.

C.5.17 clear()

```
function void clear();
```

`clear` removes all the elements from a list, but not the list itself (i.e., the list header itself).

```
list1.clear();    // list1 becomes empty
```

C.5.18 purge()

```
function void purge();
```

`purge` removes all the list elements (as in clear) and the list itself. This accomplishes the same effect as assigning null to the list. A purged list must be re-created using new before it can be used again.

```
list1.purge();    // same as list1 = null
```

Annex D
DPI C-layer

D.1 Overview

The SystemVerilog Direct Programming Interface (DPI) allows direct inter-language function calls between SystemVerilog and any foreign programming language with a C function call protocol and linking model:

— Functions implemented in C and given import declarations in SystemVerilog can be called from SystemVerilog; such functions are referred to as *imported functions*.

— Functions implemented in SystemVerilog and specified in export declarations can be called from C; such functions are referred to as *exported functions*.

The SystemVerilog DPI supports only SystemVerilog data types, which are the sole data types that can cross the boundary between SystemVerilog and a foreign language in either direction. On the other hand, the data types used in C code shall be C types; hence, the duality of types.

A value that is passed through the Direct Programming Interface is specified in SystemVerilog code as a value of SystemVerilog type, while the same value shall be specified in C code as a value of C type. Therefore, a pair of matching type definitions is required to pass a value through DPI: the SystemVerilog definition and the C definition.

It is the user's responsibility to provide these matching definitions. A tool (such as a SystemVerilog compiler) can facilitate this by generating C type definitions for the SystemVerilog definitions used in DPI for imported and exported functions.

Some SystemVerilog types are directly compatible with C types; defining a matching C type for them is straightforward. There are, however, SystemVerilog-specific types, namely packed types (arrays, structures, and unions), 2-state or 4-state, which have no natural correspondence in C. DPI does not require any particular representation of such types and does not impose any restrictions on SystemVerilog implementations. This allows implementors to choose the layout and representation of packed types that best suits their simulation performance.

While not specifying the actual representation of packed types, this C-layer interface defines a canonical representation of packed 2-state and 4-state arrays. This canonical representation is actually based on legacy Verilog Programming Language Interface's (PLI's) avalue/bvalue representation of 4-state vectors. Library functions provide the translation between the representation used in a simulator and the canonical representation of packed arrays. There are also functions for bit selects and limited part selects for packed arrays, which do not require the use of the canonical representation.

Formal arguments in SystemVerilog can be specified as open arrays solely in import declarations; exported SystemVerilog functions can not have formal arguments specified as open arrays. A formal argument is an open array when a range of one or more of its dimensions is unspecified (denoted in SystemVerilog by using empty square brackets (`[]`)). This corresponds to a relaxation of the DPI argument-matching rules (Section 26.5.1). An actual argument shall match the corresponding formal argument regardless of the range(s) for its corresponding dimension(s), which facilitates writing generalized C code that can handle SystemVerilog arrays of different sizes.

The C-layer of DPI basically uses normalized ranges. *Normalized ranges* mean `[n-1:0]` indexing for the packed part (packed arrays are restricted to one dimension) and `[0:n-1]` indexing for a dimension in the unpacked part of an array. Normalized ranges are used for the canonical representation of packed arrays in C and for System Verilog arrays passed as actual arguments to C, with the exception of actual arguments for open arrays. The elements of an open array can be accessed in C by using the same range of indices as defined in System Verilog for the actual argument for that open array and the same indexing as in SystemVerilog.

Function arguments are generally passed by some form of reference or by value. All formal arguments, except open arrays, are passed by direct reference or value, and, therefore, are directly accessible in C code. Only small values of SystemVerilog input arguments (see Annex D.7.7) are passed by value. Formal arguments

declared in SystemVerilog as open arrays are passed by a handle (type `svOpenArrayHandle`) and are accessible via library functions. Array-querying functions are provided for open arrays.

Depending on the data types used for imported or exported functions, either binary level or C-source level compatibility is granted. Binary level is granted for all data types that do not mix SystemVerilog packed and unpacked types and for open arrays which can have both packed and unpacked parts. If a data type that mixes SystemVerilog packed and unpacked types is used, then the C code needs to be re-compiled using the implementation-dependent definitions provided by the vendor.

The C-layer of the Direct Programming Interface provides two include files. The main include file, `svdpi.h`, is implementation-independent and defines the canonical representation, all basic types, and all interface functions. The second include file, `svdpi_src.h`, defines only the actual representation of packed arrays and, hence, its contents are implementation-dependent. Applications that do not need to include this file are binary-level compatible.

D.2 Naming conventions

All names introduced by this interface shall conform to the following conventions.

— All names defined in this interface are prefixed with `sv` or `SV_`.

— Function and type names start with `sv`, followed by initially capitalized words with no separators, e.g., `svBitPackedArrRef`.

— Names of symbolic constants start with `sv_`, e.g., `sv_x`.

— Names of macro definitions start with `SV_`, followed by all upper-case words separated by a underscore (_), e.g., `SV_CANONICAL_SIZE`.

D.3 Portability

Depending on the data types used for imported or exported functions, the C code can be binary-level or source-level compatible. Applications that do not use SystemVerilog packed types are always binary compatible. Applications that don't mix SystemVerilog packed and unpacked types in the same data type can be written to guarantee binary compatibility. Open arrays with both packed and unpacked parts are also binary compatible.

The values of SystemVerilog packed types can be accessed via interface functions using the canonical representation of 2-state and 4-state packed arrays, or directly through pointers using the implementation representation. The former mode assures binary level compatibility; the latter one allows for tool-specific, performance-oriented tuning of an application, though it also requires recompiling with the implementation-dependent definitions provided by the vendor and shipped with the simulator.

D.3.1 Binary compatibility

Binary compatibility means an application compiled for a given platform shall work with every SystemVerilog simulator on that platform.

D.3.2 Source-level compatibility

Source-level compatibility means an application needs to be re-compiled for each SystemVerilog simulator and implementation-specific definitions shall be required for the compilation.

D.4 Include files

The C-layer of the Direct Programming Interface defines two include files corresponding to these two levels of compatibility: `svdpi.h` and `svdpi_src.h`.

Binary compatibility of an application depends on the data types of the values passed through the interface. If all corresponding type definitions can be written in C without the need to include the `svdpi_src.h` file, then an application is binary compatible. If the `svdpi_src.h` file is required, then the application is not binary compatible and needs to be recompiled for each simulator of choice.

Applications that pass solely C-compatible data types or standalone packed arrays (both 2-state and 4-state) require only the svdpi.h file and, therefore, are binary compatible with all simulators. Applications that use complex data types which are constructed of both SystemVerilog packed arrays and C-compatible types also require the `svdpi_src.h` file and, therefore, are not binary compatible with all simulators. They are source-level compatible, however. If an application is tuned for a particular vendor-specific representation of packed arrays and therefore needs vendor specific include files, then such an application is not source-level compatible.

D.4.1 svdpi.h include file

Applications which use the Direct Programming Interface with C code usually need this main include file. The include file `svdpi.h` defines the types for canonical representation of 2-state (`bit`) and 4-state (`logic`) values and passing references to SystemVerilog data objects. The file also provides function headers and defines a number of helper macros and constants.

This document fully defines the `svdpi.h` file. The content of `svdpi.h` does not depend on any particular implementation or platform; all simulators shall use the same file. For more details on `svdpi.h`, see Annex D.9.1.

Applications which only use `svdpi.h` shall be binary-compatible with all SystemVerilog simulators.

D.4.2 `svdpi_src.h` include file

This is an auxiliary include file. `svdpi_src.h` defines data structures for implementation-specific representation of 2-state and 4-state SystemVerilog packed arrays. The interface specifies the contents of this file, i.e., what symbols are defined. The actual definitions of those symbols, however, are implementation-specific and shall be provided by vendors.

Applications that require the `svdpi_src.h` file are only source-level compatible, i.e., they need to be compiled with the version of `svdpi_src.h` provided for a particular implementation of SystemVerilog. If, however, an application makes use of the details of the implementation-specific representation of packed arrays and thus it requires vendor specific include files, then such an application is not source-level compatible.

D.5 Semantic constraints

Note that the constraints expressed here merely restate those expressed in Section 26.4.1.

Formal and actual arguments of both imported functions and exported functions are bound by the principle “What You Specify Is What You Get.” This principle is binding both for the caller and for the callee, in C code and in SystemVerilog code. For the callee, it guarantees the actual arguments are as specified for the formal ones. For the caller, it means the function call arguments shall conform with the types of the formal arguments, which might require type-coercion on the caller side.

Another way to state this is that no compiler (either C or SystemVerilog) can make argument coercions between a caller’s declared formals and the callee’s declared the formals. This is because the callee’s formal arguments are declared in a different language than the caller’s formal arguments; hence there is no visible relationship between the two sets of formals. Users are expected to understand all argument relationships and provide properly matched types on both sides of the interface (see Annex D.6.2).

In SystemVerilog code, the compiler can change the formal arguments of a native SystemVerilog function and modify its code accordingly, because of optimizations, compiler pragmas, or command line switches. The situation is different for imported functions. A SystemVerilog compiler can not modify the C code, perform any coercions, or make any changes whatsoever to the formal arguments of an imported function.

A SystemVerilog compiler shall provide any necessary coercions for the actual arguments of every imported function call. For example, a SystemVerilog compiler might truncate or extend bits of a packed array if the widths of the actual and formal arguments are different. Similarly, a C compiler can provide coercion for C types based on the relationship of the arguments in the exported function’s C prototype (formals) and the exported function’s C call site (actuals). However, a C compiler can not provide such coercion for SystemVerilog types.

Thus, in each case of an inter-language function call, either C to SystemVerilog or SystemVerilog to C, the compilers expect but cannot enforce that the types on either side are compatible. It is therefore the user's responsibility to ensure that the imported/exported function types exactly match the types of the corresponding functions in the foreign language.

D.5.1 Types of formal arguments

The principle "What You Specify Is What You Get" guarantees the types of formal arguments of imported functions — an actual argument is guaranteed to be of the type specified for the formal argument, with the exception of open arrays (for which unspecified ranges are statically unknown). Formal arguments, other than open arrays, are fully defined by imported declaration; they shall have ranges of packed or unpacked arrays exactly as specified in the imported declaration. Only the SystemVerilog declaration site of the imported function is relevant for such formal arguments.

Formal arguments defined as open arrays have the size and ranges of the actual argument, i.e., have the ranges of packed or unpacked arrays exactly as that of the actual argument. The unsized ranges of open arrays are determined at a call site; the rest of the type information is specified at the import declaration. See also Annex D.6.1.

So, if a formal argument is declared as `bit [15:8] b []`, then it is the import declaration which specifies the formal argument is an unpacked array of packed bit array with bounds `15` to `8`, while the actual argument used at a particular call site defines the bounds for the unpacked part for that call.

D.5.2 input arguments

Formal arguments specified in SystemVerilog as **`input`** must not be modified by the foreign language code. See also Section 26.4.1.2.

D.5.3 output arguments

The initial values of formal arguments specified in SystemVerilog as **`output`** are undetermined and implementation-dependent. See also Section 26.4.1.2.

D.5.4 Value changes for output and inout arguments

The SystemVerilog simulator is responsible for handling value changes for **`output`** and **`inout`** arguments. Such changes shall be detected and handled after the control returns from C code to SystemVerilog code.

D.5.5 context and non-context functions

Also refer to Section 26.4.3.

Some DPI imported functions or other interface functions called from them require that the context of their call be known. It takes special instrumentation of their call instances to provide such context; for example, a variable referring to the "current instance" might need to be set. To avoid any unnecessary overhead, imported function calls in SystemVerilog code are not instrumented unless the imported function is specified as context in its SystemVerilog import declaration.

All DPI export functions require that the context of their call is known. This occurs since SystemVerilog function declarations always occur in instantiable scopes, hence allowing a multiplicity of unique function instances in the simulator's elaborated database. Thus, there is no such thing as a non-context export function.

For the sake of simulation performance, a non-context imported function call shall not block SystemVerilog compiler optimizations. An imported function not specified as context shall not access any data objects from SystemVerilog other then its actual arguments. Only the actual arguments can be affected (read or written) by its call. Therefore, a call of non-context imported function is not a barrier for optimizations. A context imported function, however, can access (read or write) any SystemVerilog data objects by calling PLI/VPI, nor by calling an embedded export function. Therefore, a call to a context function is a barrier for SystemVerilog compiler optimizations.

Only the calls of context imported functions are properly instrumented and cause conservative optimizations;

therefore, only those functions can safely call all functions from other APIs, including PLI and VPI functions or exported SystemVerilog functions. For imported functions not specified as context, the effects of calling PLI, VPI, or SystemVerilog functions can be unpredictable and such calls can crash if the callee requires a context that has not been properly set.

Special DPI utility functions exist that allow imported functions to retrieve and operate on their context. For example, the C implementation of an imported function can use `svGetScope()` to retrieve an svScope corresponding to the instance scope of its corresponding SystemVerilog import declaration. See Annex D.8 for more details.

D.5.6 pure functions

See also Section 26.4.2.

Only non-void functions with no `output` or `inout` arguments can be specified as **`pure`**. Functions specified as **`pure`** in their corresponding SystemVerilog import declarations shall have no side effects; their results need to depend solely on the values of their input arguments. Calls to such functions can be removed by SystemVerilog compiler optimizations or replaced with the values previously computed for the same values of the input arguments.

Specifically, a `pure` function is assumed not to directly or indirectly (i.e., by calling other functions):

— perform any file operations

— read or write anything in the broadest possible meaning, includes i/o, environment variables, objects from the operating system or from the program or other processes, shared memory, sockets, etc.

— access any persistent data, like global or static variables.

If a `pure` function does not obey the above restrictions, SystemVerilog compiler optimizations can lead to unexpected behavior, due to eliminated calls or incorrect results being used.

D.5.7 Memory management

See also Section 26.4.1.4.

The memory spaces owned and allocated by C code and SystemVerilog code are disjoined. Each side is responsible for its own allocated memory. Specifically, C code shall not free the memory allocated by SystemVerilog code (or the SystemVerilog compiler) nor expect SystemVerilog code to free the memory allocated by C code (or the C compiler). This does not exclude scenarios in which C code allocates a block of memory, then passes a handle (i.e., a pointer) to that block to SystemVerilog code, which in turn calls a C function that directly (if it is the standard function `free`) or indirectly frees that block.

NOTE—In this last scenario, a block of memory is allocated and freed in C code, even when the standard functions `malloc` and `free` are called directly from SystemVerilog code.

D.6 Data types

This section defines the data types of the C-layer of the Direct Programming Interface.

D.6.1 Limitations

Packed arrays can have an arbitrary number of dimensions; though they are eventually always equivalent to a one-dimensional packed array and treated as such. If the packed part of an array in the type of a formal argument in SystemVerilog is specified as multi-dimensional, the SystemVerilog compiler linearizes it. Although the original ranges are generally preserved for open arrays, if the actual argument has a multidimensional packed part of the array, it shall be normalized into an equivalent one-dimensional packed array.

NOTE—The actual argument can have both packed and unpacked parts of an array; either can be multidimensional.

D.6.2 Duality of types: SystemVerilog types vs. C types

A value that crosses the Direct Programming Interface is specified in SystemVerilog code as a value of SystemVerilog type, while the same value shall be specified in C code as a value of C type. Therefore, each data type that is passed through the Direct Programming Interface requires two matching type definitions: the SystemVerilog definition and C definition.

The user needs to provide such matching definitions. Specifically, for each SystemVerilog type used in the import declarations or export declarations in SystemVerilog code, the user shall provide the equivalent type definition in C reflecting the argument passing mode for the particular type of SystemVerilog value and the direction (`input`, `output`, or `inout`) of the formal SystemVerilog argument. For values passed by reference, a generic pointer `void *` can be used (conveniently `typedef`ed in `svdpi.h` or `svdpi_src.h`) without knowing the actual representation of the value.

D.6.3 Data representation

DPI imposes the following additional restrictions on the representation of SystemVerilog data types.

— SystemVerilog types that are not packed and that do not contained packed elements have C compatible representation.

— Basic integer and real data types are represented as defined in Annex D.6.4.

— Enumeration types are represented as the types associated with them. Enumerated names are not available on C side of interface.

— Representation of packed types is implementation-dependent.

— Unpacked arrays embedded in a structure have C compatible layout regardless of the type of elements. Similarly, standalone arrays passed as actuals to a sized formal argument have C compatible representation.

— Standalone array passed as an actual to an open array formal

  — if the element type is scalar or packed then the representation is implementation dependent

  — otherwise the representation is C compatible. Therefore an element of an array shall have the same representation as an individual value of the same type. Hence, an array's elements can be accessed from C code via normal C array indexing similarly to doing so for individual values.

— The natural order of elements for each dimension in the layout of an unpacked array shall be used, i.e., elements with lower indices go first. For SystemVerilog range [L:R], the element with SystemVerilog index `min(L,R)` has the C index 0 and the element with SystemVerilog index `max(L,R)` has the C index `abs(L-R)`.

NOTE—This does not actually impose any restrictions on how unpacked arrays are implemented; it only says an array that does not satisfy this condition shall not be passed as an actual argument for a formal argument which is a sized array; it can be passed, however, for a formal argument which is an unsized (i.e., open) array. Therefore, the correctness of an actual argument might be implementation-dependent. Nevertheless, an open array provides an implementation-independent solution; this seems to be a reasonable trade-off.

D.6.4 Basic types

Table D-1 defines the mapping between the basic SystemVerilog data types and the corresponding C types.

Table D-1: Mapping data types

SystemVerilog type	C type
`byte`	`char`
`shortint`	`short int`

Table D-1: Mapping data types (continued)

SystemVerilog type	C type
`int`	`int`
`longint`	`long long`
`real`	`double`
`shortreal`	`float`
`handle`	`void*`
`string`	`char*`

The representation of SystemVerilog-specific data types like packed **`bit`** and **`logic`** arrays is implementation-dependent and generally transparent to the user. Nevertheless, for the sake of performance, applications can be tuned for a specific implementation and make use of the actual representation used by that implementation; such applications shall not be binary compatible, however.

D.6.5 Normalized ranges

Packed arrays are treated as one-dimensional; the unpacked part of an array can have an arbitrary number of dimensions. Normalized ranges mean `[n-1:0]` indexing for the packed part and `[0:n-1]` indexing for a dimension of the unpacked part of an array. Normalized ranges are used for accessing all arguments but open arrays. The canonical representation of packed arrays also uses normalized ranges.

D.6.6 Mapping between SystemVerilog ranges and normalized ranges

The SystemVerilog ranges for a formal argument specified as an open array are those of the actual argument for a particular call. Open arrays are accessible, however, by using their original ranges and the same indexing as in the SystemVerilog code.

For all other types of arguments, i.e., all arguments but open arrays, the SystemVerilog ranges are defined in the corresponding SystemVerilog import or export declaration. Normalized ranges are used for accessing such arguments in C code. The mapping between SystemVerilog ranges and normalized ranges is defined as follows.

1) If a packed part of an array has more than one dimension, it is linearized as specified by the equivalence of packed types (see Section 4.2).

2) A packed array of range `[L:R]` is normalized as `[abs(L-R):0]`; its most significant bit has a normalized index `abs(L-R)` and its least significant bit has a normalized index 0.

3) The natural order of elements for each dimension in the layout of an unpacked array shall be used, i.e., elements with lower indices go first. For SystemVerilog range `[L:R]`, the element with SystemVerilog index `min(L,R)` has the C index 0 and the element with SystemVerilog index `max(L,R)` has the C index `abs(L-R)`.

NOTE—The above range mapping from SystemVerilog to C applies to calls made in both directions, i.e., SystemVerilog-calls to C and C-calls to SystemVerilog.

For example, if `logic [2:3][1:3][2:0] b [1:10] [31:0]` is used in SystemVerilog, it needs to be defined in C as if it were declared in SystemVerilog in the following normalized form: `logic [17:0] b [0:9] [0:31]`.

D.6.7 Canonical representation of packed arrays

The Direct Programming Interface defines the canonical representation of packed 2-state (type `svBitVec32`) and 4-state arrays (type `svLogicVec32`). This canonical representation is derived from on the Verilog legacy

PLI's avalue/bvalue representation of 4-state vectors. Library functions provide the translation between the representation used in a simulator and the canonical representation of packed arrays.

A packed array is represented as an array of one or more elements (of type `svBitVec32` for 2-state values and `svLogicVec32` for 4-state values), each element representing a group of 32 bits.The first element of an array contains the 32 least-significant bits, next element contains the 32 more-significant bits, and so on. The last element can contain a number of unused bits. The contents of these unused bits is undetermined and the user is responsible for the masking or the sign extension (depending on the sign) for the unused bits.

Table D-2 defines the encoding used for a packed `logic` array represented as `svLogicVec32`.

Table D-2: Encoding of bits in `svLogicVec32`

c	d	Value
0	0	0
0	1	1
1	0	z
1	1	x

D.7 Argument passing modes

This section defines the ways to pass arguments in the C-layer of the Direct Programming Interface.

D.7.1 Overview

Imported and exported function arguments are generally passed by some form of a reference, with the exception of small values of SystemVerilog input arguments (see Annex D.11.7), which are passed by value. Similarly, the function result, which is restricted to small values, is passed by value, i.e., directly returned.

Actual arguments passed by reference typically are passed without changing their representation from the one used by a simulator. There is no inherent copying of arguments (other than any copying resulting from coercing).

Access to packed arrays via canonical representation involves copying arguments and does incur some overhead, however. Alternatively, for the sake of performance the application can be tuned for a particular tool and access the packed arrays directly through pointers using implementation representation, which could compromise binary and/or source compatibility. Data can be, however, moved around (copied, stored, retrieved) without using canonical representation while preserving binary or source level compatibility at the same time. This is possible by using pointers and size of data and when the detailed knowledge of the data representation is not required.

NOTE—This provides some degree of flexibility and allows the user to control the trade-off of performance vs. portability.

Formal arguments, except open arrays, are passed by direct reference or value, and, therefore, are directly accessible in C code. Formal arguments declared in SystemVerilog as open arrays are passed by a handle (type `svOpenArrayHandle`) and are accessible via library functions.

D.7.2 Calling SystemVerilog functions from C

There is no difference in argument passing between calls from SystemVerilog to C and calls from C to SystemVerilog. Functions exported from SystemVerilog can not have open arrays as arguments. Apart from this restriction, the same types of formal arguments can be declared in SystemVerilog for exported functions and imported functions. A function exported from SystemVerilog shall have the same function header in C as would an imported function with the same function result type and same formal argument list. In the case of arguments passed by reference, an actual argument to SystemVerilog function called from C shall be allocated using the same layout of data as SystemVerilog uses for that type of argument; the caller is responsible for the

allocation. It can be done while preserving the binary compatibility, see Annex D.11.5 and Annex D.11.11.

D.7.3 Argument passing by value

Only small values of formal input arguments (see Annex D.11.7) are passed by value. Function results are also directly passed by value. The user needs to provide the C-type equivalent to the SystemVerilog type of a formal argument if an argument is passed by value.

D.7.4 Argument passing by reference

For arguments passed by reference, their original simulator-defined representation shall be used and a reference (a pointer) to the actual data object is passed. The actual argument is usually allocated by a caller. The caller can also pass a reference to an object already allocated somewhere else, for example, its own formal argument passed by reference.

If an argument of type `T` is passed by reference, the formal argument shall be of type `T*`. However, packed arrays can also be passed using generic pointers `void*` (`typedef`ed accordingly to `svBitPackedArrRef` or `svLogicPackedArrRef`).

D.7.5 Allocating actual arguments for SystemVerilog-specific types

This is relevant only for calling exported SystemVerilog functions from C code. The caller is responsible for allocating any actual arguments that are passed by reference.

Static allocation requires knowledge of the relevant data type. If such a type involves SystemVerilog packed arrays, their actual representation needs to be known to C code; thus, the file `svdpi_src.h` needs to be included, which makes the C code implementation-dependent and not binary compatible.

Sometimes binary compatibility can be achieved by using dynamic allocation functions. The functions `svSizeOfLogicPackedArr()` and `svSizeOfBitPackedArr()` provide the size of the actual representation of a packed array, which can be used for the dynamic allocation of an actual argument without compromising the portability (see Annex D.11.11). Such a technique does not work if a packed array is a part of another type.

D.7.6 Argument passing by handle—open arrays

Arguments specified as open (unsized) arrays are always passed by a handle, regardless of direction of the SystemVerilog formal argument, and are accessible via library functions. The actual implementation of a handle is simulator-specific and transparent to the user. A handle is represented by the generic pointer `void *` (`typedef`ed to `svOpenArrayHandle`). Arguments passed by handle shall always have a `const` qualifier, because the user shall not modify the contents of a handle.

D.7.7 input arguments

`input` arguments of imported functions implemented in C shall always have a const qualifier.

`input` arguments, with the exception of open arrays, are passed by value or by reference, depending on the size. ‘Small’ values of formal input arguments are passed by value. The following data types are considered *small*:

— **`byte`**, **`shortint`**, **`int`**, **`longint`**, **`real`**, **`shortreal`**

— **`handle`**, **`string`**

— `bit` (i.e., 2-state) packed arrays up to 32 bits (canonical representation shall be used, like for a function result; thus a small packed bit array shall be represented as **`const`** `svBitVec32`)

`input` arguments of other types are passed by reference.

If an **`input`** argument is a packed **`bit`** array passed by value, its value shall be represented using the canonical representation `svBitVec32`. If the size is smaller than 32 bits, the most significant bits are unused and their contents are undetermined. The user is responsible for the masking or the sign extension, depending on the

sign, for the unused bits.

D.7.8 inout and output arguments

`inout` and **`output`** arguments, with the exception of open arrays, are always passed by reference. Specifically, packed arrays are passed, accordingly, as `svBitPackedArrRef` or `svLogicPackedArrRef`. The same rules about unused bits apply as in Annex D.11.7.

D.7.9 Function result

Types of a function result are restricted to the following SystemVerilog data types (see Table D-1 for the corresponding C type):

— **`byte`**, **`shortint`**, **`int`**, **`longint`**, **`real`**, **`shortreal`**, **`handle`**, **`string`**

— packed **`bit`** arrays up to 32 bits.

If the function result type is a packed `bit` array, the returned value shall be represented using the canonical representation `svBitVec32`. If a packed `bit` array is smaller than 32 bits, the most significant bits are unused and their contents are undetermined.

D.8 Context functions

Some DPI imported functions require that the context of their call is known. For example, those calls can be associated with instances of C models that have a one-to-one correspondence with instances of SystemVerilog modules that are making the calls. Alternatively, a DPI imported function might need to access or modify simulator data structures using PLI or VPI calls, or by making a call back into SystemVerilog via an export function. Context knowledge is required for such calls to function properly. It can take special instrumentation of their call to provide such context.

To avoid any unnecessary overhead, imported function calls in SystemVerilog code are not instrumented unless the imported function is specified as context in its SystemVerilog import declaration. A small set of DPI utility functions are available to assist programmers when working with context functions (see Annex D.8.3). If those utility functions are used with any non-context function, a system error shall result.

D.8.1 Overview of DPI and VPI context

Both DPI functions and VPI/PLI functions might need to understand their context. However, the meaning of the term is different for the two categories of functions.

DPI imported functions are essentially proxies for native SystemVerilog functions. Native SystemVerilog functions always operate in the scope of their declaration site. For example, a native SystemVerilog function `f()` can be declared in a module `m` which is instantiated as `top.i1_m`. The `top.i1_m` instance of `f()` can be called via hierarchical reference from code in a distant design region. Function `f()` is said to execute in the *context* (aka. instantiated scope) of `top.i1_m`, since it has unqualified visibility only for variables local to that specific instance of `m`. Function `f()` does not have unqualified visibility for any variables in the calling code's scope.

DPI imported functions follow the same model as native SystemVerilog functions. They execute in the context of their surrounding declarative scope, rather than the context of their call sites. This type of context is termed *DPI context*.

This is in contrast to VPI and PLI functions. Such functions execute in a context associated with their call sites. The VPI/PLI programming model relies on C code's ability to retrieve a context handle associated with the associated system task's call site, and then work with the context handle to glean information about arguments, items in the call site's surrounding declarative scope, etc. This type of context is termed *VPI context*.

Note that all DPI export functions require that the context of their call is known. This occurs since SystemVerilog function declarations always occur in instantiable scopes, hence giving rise to a multiplicity of associated function instances in the simulator's database. Thus, there is no such thing as a non-context export function. All export function calls must have their execution scope specified in advance by use of a context-setting API

function.

D.8.2 Context of imported and export functions

DPI imported and export functions can be declared anywhere a normal SystemVerilog function can be declared. Specifically, this means that they can be declared in **`module`**, **`program`**, **`interface`**, or **`generate`** declarative scope.

A context imported function executes in the context of the instantiated scope surrounding its declaration. This means that such functions can see other variables in that scope without qualification. As explained in Annex D.8.1, this should not be confused with the context of the function's call site, which can actually be anywhere in the SystemVerilog design hierarchy. The context of an imported or exported function corresponds to the fully qualified name of the function, minus the function name itself.

Note that context is transitive through imported and export context functions declared in the same scope. That is, if an imported function is running in a certain context, and if it in turn calls an exported function that is available in the same context, the exported function can be called without any use of svSetScope(). For example, consider a SystemVerilog call to a native function `f()`, which in turn calls a native function g(). Now replace the native function `f()` with an equivalent imported context C function, `f'()`. The system shall behave identically regardless if `f()` or `f'()` is in the call chain above `g()`. `g()` has the proper execution context in both cases.

D.8.3 Working with DPI context functions in C code

DPI defines a small set of functions to help programmers work with DPI context functions. The term scope is used in the function names for consistency with other SystemVerilog terminology. The terms *scope* and *context* are equivalent for DPI functions.

There are functions that allow the user to retrieve and manipulate the current operational scope. It is an error to use these functions with any C code that is not executing under a call to a DPI context imported function.

There are also functions that provide users with the power to set data specific to C models into the SystemVerilog simulator for later retrieval. These are the "put" and "get" user data functions, which are similar to facilities provided in VPI and PLI.

The put and get user data functions are flexible and allow for a number of use models. Users might wish to share user data across multiple context imported functions defined in the same SV scope. Users might wish to have unique data storage on a per function basis. Shared or unique data storage is controllable by a user-defined key.

To achieve shared data storage, a related set of context imported functions should all use the same userKey. To achieve unique data storage, a context import function should use a unique key. Note that it is a requirement on the user that such a key be truly unique from all other keys that could possibly be used by C code. This includes completely unknown C code that could be running in the same simulation. It is suggested that taking addresses of static C symbols (such as a function pointer, or address of some static C data) always be done for user key generation. Generating keys based on arbitrary integers is not a safe practice.

Note that it is never possible to share user data storage across different contexts. For example, if a Verilog module `m` declares a context imported function `f`, and `m` is instantiated more than once in the SystemVerilog design, then `f` shall execute under different values of `svScope`. No such executing instances of `f` can share user data with each other, at least not using the system-provided user data storage area accessible via `svPutUserData()`.

A user wanting to share a data area across multiple contexts must do so by allocating the common data area, then storing the pointer to it individually for each of the contexts in question via multiple calls to `svPutUserData()`. This is because, although a common user key can be used, the data must be associated with the individual scopes (denoted by `svScope`) of those contexts.

```
/* Functions for working with DPI context functions */
```



```
/* Retrieve the active instance scope currently associated with the executing
 * imported function.
 * Unless a prior call to svSetScope has occurred, this is the scope of the
 * function's declaration site, not call site.
 * The return value is undefined if this function is invoked from a non-context
 * imported function.
 */
svScope svGetScope();

/* Set context for subsequent export function execution.
 * This function must be called before calling an export function, unless
 * the export function is called while executing an extern function. In that
 * case the export function shall inherit the scope of the surrounding extern
 * function. This is known as the "default scope".
 * The return is the previous active scope (as per svGetScope)
 */
svScope svSetScope(const svScope scope);

/* Gets the fully qualified name of a scope handle */
const char* svGetNameFromScope(const svScope);

/* Retrieve svScope to instance scope of an arbitrary function declaration.
 * (can be either module, program, interface, or generate scope)
 * The return value shall be NULL for unrecognized scope names.
 */
svScope svGetScopeFromName(const char* scopeName);

/* Store an arbitrary user data pointer for later retrieval by svGetUserData()
 * The userKey is generated by the user. It must be guaranteed by the user to
 * be unique from all other userKey's for all unique data storage requirements
 * It is recommended that the address of static functions or variables in the
 * user's C code be used as the userKey.
 * It is illegal to pass in NULL values for either the scope or userData
 * arguments. It is also an error to call svPutUserData() with an invalid
 * svScope. This function returns -1 for all error cases, 0 upon success. It is
 * suggested that userData values of 0 (NULL) not be used as otherwise it can
 * be impossible to discern error status returns when calling svGetUserData()
 */
int svPutUserData(const svScope scope, void *userKey, void* userData);

/* Retrieve an arbitrary user data pointer that was previously
 * stored by a call to svPutUserData(). See the comment above
 * svPutUserData() for an explanation of userKey, as well as
 * restrictions on NULL and illegal svScope and userKey values.
 * This function returns NULL for all error cases, and a non-Null
 * user data pointer upon success.
 * This function also returns NULL in the event that a prior call
 * to svPutUserData() was never made.
 */
void* svGetUserData(const svScope scope, void* userKey);

/* Returns the file and line number in the SV code from which the extern call
 * was made. If this information available, returns TRUE and updates fileName
 * and lineNumber to the appropriate values. Behavior is unpredictable if
 * fileName or lineNumber are not appropriate pointers. If this information is
 * not available return FALSE and contents of fileName and lineNumber not
 * modified. Whether this information is available or not is implementation
 * specific. Note that the string provided (if any) is owned by the SV
 * implementation and is valid only until the next call to any SV function.
```

```
 * Applications must not modify this string or free it
 */
int svGetCallerInfo(char **fileName, int *lineNumber);
```

D.8.4 Example 1 — Using DPI context functions

```
SV Side:
      // Declare an imported context sensitive C function with cname "MyCFunc"
      import "DPI" context MyCFunc = function integer MapID(int portID);

C Side:
         // Define the function and model class on the C++ side:
         class MyCModel {
         private:
            int locallyMapped(int portID); // Does something interesting...
         public:
            // Constructor
            MyCModel(const char* instancePath) {
               svScope svScope = svGetScopeByName(instancePath);

               // Associate "this" with the corresponding SystemVerilog scope
               // for fast retrieval during runtime.
               svPutUserData(svScope, (void*) MyCFunc, this);
            }

         friend int MyCFunc(int portID);
      };

      // Implementation of imported context function callable in SV
      int MyCFunc(int portID) {
         // Retrieve SV instance scope (i.e. this function's context).
         svScope = svGetScope();

         // Retrieve and make use of user data stored in SV scope
         MyCModel* me = (MyCModel*)svGetUserData(svScope, (void*) MyCFunc);
         return me->locallyMapped(portID);
      }
```

D.8.5 Relationship between DPI and VPI/PLI interfaces

DPI allows C code to run in the context of a SystemVerilog simulation, thus it is natural for users to consider using VPI/PLI C code from within imported functions.

There is no specific relationship defined between DPI and the existing Verilog programming interfaces (VPI and PLI). Programmers must make no assumptions about how DPI and the other interfaces interact. In particular, note that a `vpiHandle` is not equivalent to an `svOpenArrayHandle`, and the two must not be interchanged and passed between functions defined in two different interface standards.

If a user wants to call VPI or PLI functions from within an imported function, the imported function must be flagged with the context qualifier.

Not all VPI or PLI functionality is available from within DPI context imported functions. For example, a SystemVerilog imported function is not a system task, and thus making the following call from within an imported function would result in an error:

```
/* Get handle to system task call site in preparation for argument scan */
vpiHandle myHandle = vpi_handle(vpiSysTfCall, NULL);
```


Similarly, receiving `misctf` callbacks and other activities associated with system tasks are not supported inside DPI imported functions. Users should use VPI or PLI if they wish to accomplish such actions.

However, the following kind of code is guaranteed to work from within DPI context imported functions:

```
/* Prepare to scan all top level modules */
vpiHandle myHandle = vpi_iterate(vpiModule, NULL);
```

D.9 Include files

The C-layer of the Direct Programming Interface defines two include files. The main include file, `svdpi.h`, is implementation-independent and defines the canonical representation, all basic types, and all interface functions. The second include file, `svdpi_src.h`, defines only the actual representation of packed arrays and, hence, is implementation-dependent. Both files are shown in Annex B.

Applications which do not need to include `svdpi_src.h` are binary-level compatible.

D.9.1 Binary compatibility include file `svdpi.h`

Applications which use the Direct Programming Interface with C code usually need this main include file. The include file `svdpi.h` defines the types for canonical representation of 2-state (**`bit`**) and 4-state (**`logic`**) values and passing references to SystemVerilog data objects, provides function headers, and defines a number of helper macros and constants.

This document fully defines the `svdpi.h` file. The content of `svdpi.h` does not depend on any particular implementation or platform; all simulators shall use the same file. The following subsections (and Annex D.10.3.1) detail the contents of the `svdpi.h` file.

D.9.1.1 Scalars of type `bit` and `logic`

```
/* canonical representation */

#define sv_0   0
#define sv_1   1
#define sv_z   2  /* representation of 4-st scalar z */
#define sv_x   3  /* representation of 4-st scalar x */

/* common type for 'bit' and 'logic' scalars. */
typedef unsigned char svScalar;

typedef svScalar svBit;    /* scalar */
typedef svScalar svLogic;  /* scalar */
```

D.9.1.2 Canonical representation of packed arrays

```
/* 2-state and 4-state vectors, modelled upon PLI's avalue/bvalue */
#define SV_CANONICAL_SIZE(WIDTH) (((WIDTH)+31)>>5)

typedef unsigned int
        svBitVec32;/* (a chunk of) packed bit array */

typedef struct { unsigned int c; unsigned int d;}
        svLogicVec32; /* (a chunk of) packed logic array */

/* Since the contents of the unused bits is undetermined, the following macros
can be handy */
#define SV_MASK(N) (~(-1<<(N)))

#define SV_GET_UNSIGNED_BITS(VALUE,N)\
   ((N)==32?(VALUE):((VALUE)&SV_MASK(N)))
```

```
#define SV_GET_SIGNED_BITS(VALUE,N)\
   ((N)==32?(VALUE):\
    (((VALUE)&(1<<((N)1)))?((VALUE)|~SV_MASK(N)):((VALUE)&SV_MASK(N))))
```

D.9.1.3 Implementation-dependent representation

```
/* a handle to a scope (an instance of a module or an interface) */
typedef void *svScope;

/* a handle to a generic object (actually, unsized array) */
typedef void* svOpenArrayHandle;

/* reference to a standalone packed array */
typedef void* svBitPackedArrRef;
typedef void* svLogicPackedArrRef;

/* total size in bytes of the simulator's representation of a packed array */
/* width in bits */
int svSizeOfBitPackedArr(int width);
int svSizeOfLogicPackedArr(int width);
```

D.9.1.4 Translation between the actual representation and the canonical representation

```
/* functions for translation between the representation actually used by
   simulator and the canonical representation */

/* s=source, d=destination, w=width */

/* actual <-- canonical */
void svPutBitVec32   (svBitPackedArrRef   d, const svBitVec32*   s, int w);
void svPutLogicVec32 (svLogicPackedArrRef d, const svLogicVec32* s, int w);

/* canonical <-- actual */
void svGetBitVec32   (svBitVec32*   d, const svBitPackedArrRef   s, int w);
void svGetLogicVec32 (svLogicVec32* d, const svLogicPackedArrRef s, int w);
```

The above functions copy the whole array in either direction. The user is responsible for providing the correct width and for allocating an array in the canonical representation. The contents of the unused bits is undetermined.

Although the put/get functionality provided for `bit` and `logic` packed arrays is sufficient, yet basic, it requires unnecessary copying of the whole packed array when perhaps only some bits are needed. For the sake of convenience and improved performance, bit selects and limited (up to 32 bits) part selects are also supported, see Annex D.10.3.1 and Annex D.10.3.2.

D.9.2 Source-level compatibility include file `svdpi_src.h`

Only two symbols are defined: the macros that allow declaring variables to represent the SystemVerilog packed arrays of type `bit` or `logic`.

```
#define SV_BIT_PACKED_ARRAY(WIDTH,NAME) ...
#define SV_LOGIC_PACKED_ARRAY(WIDTH,NAME) ...
```

The actual definitions are implementation-specific. For example, a SystemVerilog simulator might define the later macro as follows.

```
#define SV_LOGIC_PACKED_ARRAY(WIDTH,NAME) \
                              svLogicVec32 NAME [ SV_CANONICAL_SIZE(WIDTH) ]
```


D.9.3 Example 2 — binary compatible application

SystemVerilog:

```
typedef struct {int a; int b;} pair;
import "DPI" function void foo(input int i1, pair i2, output logic [63:0] o3);

export "DPI" function exported_sv_func;

function void exported_sv_func(input int i, output int o [0:7]);
   begin ... end
endfunction
```

C:

```
#include "svdpi.h"

typedef struct {int a; int b;} pair;

extern void exported_sv_func(int, int *); /* imported from SystemVerilog */

void foo(const int i1, const pair *i2, svLogicPackedArrRef o3)
{
   svLogicVec32 arr[SV_CANONICAL_SIZE(64)]; /* 2 chunks needed */
   int tab[8];

   printf("%d\n", i1);
   arr[1].c = i2->a;
   arr[1].d = 0;
   arr[2].c = i2->b;
   arr[2].d = 0;
   svPutLogicVec32 (o3, arr, 64);

   /* call SystemVerilog */
   exported_sv_func(i1, tab); /* tab passed by reference */
   ...
}
```

D.9.4 Example 3— source-level compatible application

SystemVerilog:

```
typedef struct {int a; bit [6:1][1:8] b [65:2]; int c;} triple;
   // troublesome mix of C types and packed arrays
import "DPI" function void foo(input triple i);

export "DPI" function exported_sv_func;

function void exported_sv_func(input int i, output logic [63:0] o);
   begin ... end
endfunction
```

C:

```
#include "svdpi.h"
#include "svdpi_src.h"

typedef struct {
        int a;
```

```
        sv_BIT_PACKED_ARRAY(6*8, b) [64]; /* implementation specific
                                          representation */
        int c;
        } triple;

/* Note that 'b' is defined as for 'bit [6*8-1:0] b [63:0]' */

extern void exported_sv_func(int, svLogicPackedArrRef); /* imported from
                                                           SystemVerilog */

void foo(const triple *i)
{
   int j;
   /* canonical representation */
   svBitVec32 arr[SV_CANONICAL_SIZE(6*8)]; /* 6*8 packed bits */
   svLogicVec32 aL[SV_CANONICAL_SIZE(64)];

   /* implementation specific representation */
   SV_LOGIC_PACKED_ARRAY(64, my_tab);

   printf("%d %d\n", i->a, i->c);
   for (j=0; j<64; j++) {
      svGetBitVec32(arr, (svBitPackedArrRef)&(i->b[j]), 6*8);
   ...
}
...
/* call SystemVerilog */
exported_sv_func(2, (svLogicPackedArrRef)&my_tab); /* by reference */
svGetLogicVec32(aL, (svLogicPackedArrRef)&my_tab, 64);  ... }
```

NOTE—a, b, and c are directly accessed as fields in a structure. In the case of b, which represents unpacked array of packed arrays, the individual element is accessed via the library function svGetBitVec32(), by passing its address to the function.

D.10 Arrays

Normalized ranges are used for accessing SystemVerilog arrays, with the exception of formal arguments specified as open arrays.

D.10.1 Multidimensional arrays

Packed arrays shall be one-dimensional. Unpacked arrays can have an arbitrary number of dimensions.

D.10.2 Direct access to unpacked arrays

Unpacked arrays, with the exception of formal arguments specified as open arrays, shall have the same layout as used by a C compiler; they are accessed using C indexing (see Annex D.6.6).

D.10.3 Access to packed arrays via canonical representation

Packed arrays are accessible via canonical representation; this C-layer interface provides functions for moving data between implementation representation and canonical representation (any necessary conversion is performed on-the-fly (see Annex D.9.1.3)), and for bit selects and limited (up to 32-bit) part selects. (Bit selects do not involve any canonical representation.)

D.10.3.1 Bit selects

This subsection defines the bit selects portion of the svdpi.h file (see Annex D.9.1 for more details).

```
/* Packed arrays are assumed to be indexed n-1:0,
```



```
   where 0 is the index of least significant bit */

/* functions for bit select */

/* s=source, i=bit-index */
svBit svGetSelectBit(const svBitPackedArrRef s, int i);
svLogic svGetSelectLogic(const svLogicPackedArrRef s, int i);

/* d=destination, i=bit-index, s=scalar */
void svPutSelectBit(svBitPackedArrRef d, int i, svBit s);
void svPutSelectLogic(svLogicPackedArrRef d, int i, svLogic s);
```

D.10.3.2 Part selects

Limited (up to 32-bit) part selects are supported. A part select is a slice of a packed array of types **`bit`** or **`logic`**. Array slices are not supported for unpacked arrays. Additionally, 64-bit wide part select can be read as a single value of type unsigned **`long long`**.

Functions for part selects only allow access (read/write) to a narrow subrange of up to 32 bits. A canonical representation shall be used for such narrow vectors. If the specified range of part select is not fully contained within the normalized range of an array, the behavior is undetermined.

For the convenience, bit type part selects are returned as a function result. In addition to a general function for narrow part selects (<= 32-bits), there are two specialized functions for 32 and 64 bits.

```
/*
 * functions for part select
 *
 * a narrow (<=32 bits) part select is copied between
 * the implementation representation and a single chunk of
 * canonical representation
 * Normalized ranges and indexing [n-1:0] are used for both arrays:
 * the array in the implementation representation and the canonical array.
 *
 * s=source, d=destination, i=starting bit index, w=width
 * like for variable part selects; limitations: w <= 32
 */
```

NOTE—For the sake of symmetry, a canonical representation (i.e., an array) is used both for **`bit`** and **`logic`**, although a simpler **`int`** can be used for **`bit`**-part selects (<= 32-bits):

```
/* canonical <-- actual */
void svGetPartSelectBit(svBitVec32* d, const svBitPackedArrRef s, int i,
                        int w);
svBitVec32 svGetBits(const svBitPackedArrRef s, int i, int w);
svBitVec32 svGet32Bits(const svBitPackedArrRef s, int i); // 32-bits
unsigned long long svGet64Bits(const svBitPackedArrRef s, int i); // 64-bits
void svGetPartSelectLogic(svLogicVec32* d, const svLogicPackedArrRef s, int i,
                        int w);

/* actual <-- canonical */
void svPutPartSelectBit(svBitPackedArrRef d, const svBitVec32 s, int i,
                        int w);
void svPutPartSelectLogic(svLogicPackedArrRef d, const svLogicVec32 s, int i,
                        int w);
```

D.11 Open arrays

Formal arguments specified as open arrays allows passing actual arguments of different sizes (i.e., different range and/or different number of elements), which facilitates writing more general C code that can handle SystemVerilog arrays of different sizes. The elements of an open array can be accessed in C by using the same range of indices and the same indexing as in SystemVerilog. Plus, inquiries about the dimensions and the original boundaries of SystemVerilog actual arguments are supported for open arrays.

NOTE—Both packed and unpacked array dimensions can be unsized.

All formal arguments declared in SystemVerilog as open arrays are passed by handle (type `svOpenArrayHandle`), regardless of the direction of a SystemVerilog formal argument. Such arguments are accessible via interface functions.

D.11.1 Actual ranges

The formal arguments defined as open arrays have the size and ranges of the actual argument, as determined on a per-call basis. The programmer shall always have a choice whether to specify a formal argument as a sized array or as an open (unsized) array.

In the former case, all indices are normalized on the C side (i.e., `0` and up) and the programmer needs to know the size of an array and be capable of determining how the ranges of the actual argument map onto C-style ranges (see Annex D.6.6).

Tip: programmers can decide to use `[n:0]name[0:k]` style ranges in SystemVerilog.

In the later case, i.e., an open array, individual elements of a packed array are accessible via interface functions, which facilitate the SystemVerilog-style of indexing with the original boundaries of the actual argument.

If a formal argument is specified as a sized array, then it shall be passed by reference, with no overhead, and is directly accessible as a normalized array. If a formal argument is specified as an open (unsized) array, then it shall be passed by handle, with some overhead, and is mostly indirectly accessible, again with some overhead, although it retains the original argument boundaries.

NOTE—This provides some degree of flexibility and allows the programmer to control the trade-off of performance vs. convenience.

The following example shows the use of sized vs. unsized arrays in SystemVerilog code.

```
// both unpacked arrays are 64 by 8 elements, packed 16-bit each
logic [15: 0] a_64x8 [63:0][7:0];
logic [31:16] b_64x8 [64:1][-1:-8];

import "DPI" function void foo(input logic [] i [][]);
      // 2-dimensional unsized unpacked array of unsized packed logic

import "DPI" function void boo(input logic [31:16] i [64:1][-1:-8]);
      // 2-dimensional sized unpacked array of sized packed logic

foo(a_64x8);
foo(b_64x8); // C code can use original ranges [31:16][64:1][-1:-8]

boo(b_64x8); // C code must use normalized ranges [15:0][0:63][0:7]
```

D.11.2 Array querying functions

These functions are modelled upon the SystemVerilog array querying functions and use the same semantics (see Section 22.4).

If the dimension is 0, then the query refers to the packed part (which is one-dimensional) of an array, and dimensions `> 0` refer to the unpacked part of an array.

```
/* h= handle to open array, d=dimension */
int svLeft(const svOpenArrayHandle h, int d);
int svRight(const svOpenArrayHandle h, int d);
int svLow(const svOpenArrayHandle h, int d);
int svHigh(const svOpenArrayHandle h, int d);
int svIncrement(const svOpenArrayHandle h, int d);
int svLength(const svOpenArrayHandle h, int d);
int svDimensions(const svOpenArrayHandle h);
```

D.11.3 Access functions

Similarly to sized arrays, there are functions for copying data between the simulator representation and the canonical representation and to obtain the actual address of SystemVerilog data object or of an individual element of an unpacked array. This information might be useful for simulator-specific tuning of the application.

Depending on the type of an element of an unpacked array, different access methods shall be used when working with elements.

— Packed arrays (`bit` or `logic`) are accessed via copying to or from the canonical representation.

— Scalars (1-bit value of type `bit` or `logic`) are accessed (read or written) directly.

— Other types of values (e.g., structures) are accessed via generic pointers; a library function calculates an address and the user needs to provide the appropriate casting.

— Scalars and packed arrays are accessible via pointers only if the implementation supports this functionality (per array), e.g., one array can be represented in a form that allows such access, while another array might use a compacted representation which renders this functionality unfeasible (both occurring within the same simulator).

SystemVerilog allows arbitrary dimensions and, hence, an arbitrary number of indices. To facilitate this, variable argument list functions shall be used. For the sake of performance, specialized versions of all indexing functions are provided for 1, 2, or 3 indices.

D.11.4 Access to the actual representation

The following functions provide an actual address of the whole array or of its individual elements. These functions shall be used for accessing elements of arrays of types compatible with C. These functions are also useful for vendors, because they provide access to the actual representation for all types of arrays.

If the actual layout of the SystemVerilog array passed as an argument for an open unpacked array is different than the C layout, then it is not possible to access such an array as a whole; therefore, the address and size of such an array shall be undefined (zero (`0`), to be exact). Nonetheless, the addresses of individual elements of an array shall be always supported.

NOTE—No specific representation of an array is assumed here; hence, all functions use a generic pointer `void *`.

```
/* a pointer to the actual representation of the whole array of any type */
/* NULL if not in C layout */
void *svGetArrayPtr(const svOpenArrayHandle);

int svSizeOfArray(const svOpenArrayHandle); /* total size in bytes or 0 if not
                                               in C layout */

/* Return a pointer to an element of the array
   or NULL if index outside the range or null pointer */
```

```
void *svGetArrElemPtr(const svOpenArrayHandle, int indx1, ...);

/* specialized versions for 1-, 2- and 3-dimensional arrays: */
void *svGetArrElemPtr1(const svOpenArrayHandle, int indx1);
void *svGetArrElemPtr2(const svOpenArrayHandle, int indx1, int indx2);
void *svGetArrElemPtr3(const svOpenArrayHandle, int indx1, int indx2,
                       int indx3);
```

Access to an individual array element via pointer makes sense only if the representation of such an element is the same as it would be for an individual value of the same type. Representation of array elements of type `scalar` or *packed value* is implementation-dependent; the above functions shall return `NULL` if the representation of the array elements differs from the representation of individual values of the same type.

D.11.5 Access to elements via canonical representation

This group of functions is meant for accessing elements which are packed arrays (`bit` or `logic`).

The following functions copy a single vector from a canonical representation to an element of an open array or other way round. The element of an array is identified by indices, bound by the ranges of the actual argument, i.e., the original SystemVerilog ranges are used for indexing.

```
/* functions for translation between simulator and canonical representations*/
/* s=source, d=destination */
/* actual <-- canonical */
void svPutBitArrElemVec32 (const svOpenArrayHandle d, const svBitVec32* s,
                           int indx1, ...);
void svPutBitArrElem1Vec32(const svOpenArrayHandle d, const svBitVec32* s,
                           int indx1);
void svPutBitArrElem2Vec32(const svOpenArrayHandle d, const svBitVec32* s,
                           int indx1, int indx2);
void svPutBitArrElem3Vec32(const svOpenArrayHandle d, const svBitVec32* s,
                           int indx1, int indx2, int indx3);

void svPutLogicArrElemVec32 (const svOpenArrayHandle d, const svLogicVec32* s,
                           int indx1, ...);
void svPutLogicArrElem1Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
                           int indx1);
void svPutLogicArrElem2Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
                           int indx1, int indx2);
void svPutLogicArrElem3Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
                           int indx1, int indx2, int indx3);

/* canonical <-- actual */
void svGetBitArrElemVec32 (svBitVec32* d, const svOpenArrayHandle s,
                           int indx1, ...);
void svGetBitArrElem1Vec32(svBitVec32* d, const svOpenArrayHandle s,
                           int indx1);
void svGetBitArrElem2Vec32(svBitVec32* d, const svOpenArrayHandle s,
                           int indx1, int indx2);
void svGetBitArrElem3Vec32(svBitVec32* d, const svOpenArrayHandle s,
                           int indx1, int indx2, int indx3);

void svGetLogicArrElemVec32 (svLogicVec32* d, const svOpenArrayHandle s,
                           int indx1, ...);
void svGetLogicArrElem1Vec32(svLogicVec32* d, const svOpenArrayHandle s,
                           int indx1);
void svGetLogicArrElem2Vec32(svLogicVec32* d, const svOpenArrayHandle s,
                           int indx1, int indx2);
```



```
void svGetLogicArrElem3Vec32(svLogicVec32* d, const svOpenArrayHandle s,
                             int indx1, int indx2, int indx3);
```

The above functions copy the whole packed array in either direction. The user is responsible for allocating an array in the canonical representation.

D.11.6 Access to scalar elements (bit and logic)

Another group of functions is needed for scalars (i.e., when an element of an array is a simple scalar, **`bit`**, or **`logic`**):

```
svBit   svGetBitArrElem (const svOpenArrayHandle s, int indx1, ...);
svBit   svGetBitArrElem1(const svOpenArrayHandle s, int indx1);
svBit   svGetBitArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svBit   svGetBitArrElem3(const svOpenArrayHandle s, int indx1, int indx2,
                             int indx3);

svLogic svGetLogicArrElem (const svOpenArrayHandle s, int indx1, ...);
svLogic svGetLogicArrElem1(const svOpenArrayHandle s, int indx1);
svLogic svGetLogicArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svLogic svGetLogicArrElem3(const svOpenArrayHandle s, int indx1, int indx2,
                             int indx3);

void svPutLogicArrElem (const svOpenArrayHandle d, svLogic value, int indx1,
                             ...);
void svPutLogicArrElem1(const svOpenArrayHandle d, svLogic value, int indx1);
void svPutLogicArrElem2(const svOpenArrayHandle d, svLogic value, int indx1,
                             int indx2);
void svPutLogicArrElem3(const svOpenArrayHandle d, svLogic value, int indx1,
                             int indx2, int indx3);

void svPutBitArrElem (const svOpenArrayHandle d, svBit value, int indx1, ...);
void svPutBitArrElem1(const svOpenArrayHandle d, svBit value, int indx1);
void svPutBitArrElem2(const svOpenArrayHandle d, svBit value, int indx1,
                             int indx2);
void svPutBitArrElem3(const svOpenArrayHandle d, svBit value, int indx1,
                             int indx2, int indx3);
```

D.11.7 Access to array elements of other types

If an array's elements are of a type compatible with C, there is no need to use canonical representation. In such situations, the elements are accessed via pointers, i.e., the actual address of an element shall be computed first and then used to access the desired element.

D.11.8 Example 4— two-dimensional open array

SystemVerilog:

```
typedef struct {int i; ... } MyType;

import "DPI" function void foo(input MyType i [][]); /* 2-dimensional unsized
                                                        unpacked array of MyType */

MyType a_10x5 [11:20][6:2];
MyType a_64x8 [64:1][-1:-8];

foo(a_10x5);
foo(a_64x8);
```

C:

```
#include "svdpi.h"

typedef struct {int i; ... } MyType;

void foo(const svOpenArrayHandle h)
{
   MyType my_value;
   int i, j;
   int lo1 = svLow(h, 1);
   int hi1 = svHigh(h, 1);
   int lo2 = svLow(h, 2);
   int hi2 = svHigh(h, 2);

   for (i = lo1; i <= hi1; i++) {
      for (j = lo2; j <= hi2; j++) {

         my_value = *(MyType *)svGetArrElemPtr2(h, i, j);
         ...
         *(MyType *)svGetArrElemPtr2(h, i, j) = my_value;
         ...
      }
   ...
   }
}
```

D.11.9 Example 5 — open array

SystemVerilog:

```
typedef struct { ... } MyType;

import "DPI" function void foo(input MyType i [], output MyType o []);

MyType source [11:20];
MyType target [11:20];

foo(source, target);
```

C:

```
#include "svdpi.h"

typedef struct ... } MyType;

void foo(const svOpenArrayHandle hin, const svOpenArrayHandle hout)
{
   int count = svLength(hin, 1);
   MyType *s = (MyType *)svGetArrayPtr(hin);
   MyType *d = (MyType *)svGetArrayPtr(hout);

   if (s && d) { /* both arrays have C layout */

   /* an efficient solution using pointer arithmetic */
   while (count--)
      *d++ = *s++;

   /* even more efficient:
```



```
      memcpy(d, s, svSizeOfArray(hin));
   */

} else { /* less efficient yet implementation independent */

      int i = svLow(hin, 1);
      int j = svLow(hout, 1);
      while (i <= svHigh(hin, 1)) {
         *(MyType *)svGetArrElemPtr1(hout, j++) =
         *(MyType *)svGetArrElemPtr1(hin, i++);
      }

   }

}
```

D.11.10 Example 6 — access to packed arrays

SystemVerilog:

```
import "DPI" function void foo(input logic [127:0]);
import "DPI" function void boo(input logic [127:0] i []); // open array of
                                                          // 128-bit
```

C:

```
#include "svdpi.h"

/* one 128-bit packed vector */
void foo(const svLogicPackedArrRef packed_vec_128_bit)
{
   svLogicVec32 arr[SV_CANONICAL_SIZE(128)]; /* canonical representation */

   svGetLogicVec32(arr, packed_vec_128_bit, 128);
   ...
}

/* open array of 128-bit packed vectors */
void boo(const svOpenArrayHandle h)
{
   int i;
   svLogicVec32 arr[SV_CANONICAL_SIZE(128)]; /* canonical representation */

   for (i = svLow(h, 1); i <= svHigh(h, 1); i++) {

      svLogicPackedArrRef ptr = (svLogicPackedArrRef)svGetArrElemPtr1(h, i);
      /* user need not know the vendor representation! */

      svGetLogicVec32(arr, ptr, 128);
      ...
   }
   ...
}
```

D.11.11 Example 7 — binary compatible calls of exported functions

This example demonstrates the source compatibility include file `svdpi_src.h` is not needed if a C function dynamically allocates the data structure for simulator representation of a packed array to be passed to an exported SystemVerilog function.

SystemVerilog:

```
export "DPI" function myfunc;
...
function void myfunc (output logic [31:0] r); ...
...
```

C:

```
#include "svdpi.h"
extern void myfunc (svLogicPackedArrRef r); /* exported from SV */

   /* output logic packed 32-bits */
...
svLogicVec32 my_r[SV_CANONICAL_SIZE(32)];
/* my array, canonical representation */

/* allocate memory for logic packed 32-bits in simulator's representation */
svLogicPackedArrRef r =
   (svLogicPackedArrRef)malloc(svSizeOfLogicPackedArr(32));
myfunc(r);
/* canonical <-- actual */
svGetLogicVec32(my_r, r, 32);
/* shall use only the canonical representation from now on */
free(r); /* don't need any more */
...
```


Annex E
Include files

This annex shows the contents of the `svdpi.h` and `svdpi_src.h` include files.

E.1 Binary-level compatibility include file svdpi.h

```
/* canonical representation */

#define sv_0   0
#define sv_1   1
#define sv_z   2  /* representation of 4-st scalar z */
#define sv_x   3  /* representation of 4-st scalar x */

/* common type for 'bit' and 'logic' scalars. */
typedef unsigned char svScalar;

typedef svScalar svBit;    /* scalar */
typedef svScalar svLogic;  /* scalar */

/* Canonical representation of packed arrays */
/* 2-state and 4-state vectors, modelled upon PLI's avalue/bvalue */
#define SV_CANONICAL_SIZE(WIDTH) (((WIDTH)+31)>>5)

typedef unsigned int
        svBitVec32;/* (a chunk of) packed bit array */

typedef struct { unsigned int c; unsigned int d;}
        svLogicVec32; /* (a chunk of) packed logic array */

/* Since the contents of the unused bits is undetermined, the following macros can
be handy */
#define SV_MASK(N) (~(-1<<(N)))

#define SV_GET_UNSIGNED_BITS(VALUE,N)\
   ((N)==32?(VALUE):((VALUE)&SV_MASK(N)))

#define SV_GET_SIGNED_BITS(VALUE,N)\
   ((N)==32?(VALUE):\
    (((VALUE)&(1<<((N)1)))?((VALUE)|~SV_MASK(N)):((VALUE)&SV_MASK(N))))

/* implementation-dependent representation */

/* a handle to a scope (an instance of a module or interface) */
typedef void* svScope;

/* a handle to a generic object (actually, unsized array) */
typedef void* svOpenArrayHandle;

/* reference to a standalone packed array */
typedef void* svBitPackedArrRef;
typedef void* svLogicPackedArrRef;

/* total size in bytes of the simulator's representation of a packed array */
/* width in bits */
int svSizeOfBitPackedArr(int width);
int svSizeOfLogicPackedArr(int width);
```

```
/* Translation between the actual representation and the canonical representation
*/

/* functions for translation between the representation actually used by
   simulator and the canonical representation */

/* s=source, d=destination, w=width */

/* actual <-- canonical */
void svPutBitVec32   (svBitPackedArrRef   d, const svBitVec32*   s, int w);
void svPutLogicVec32 (svLogicPackedArrRef d, const svLogicVec32* s, int w);

/* canonical <-- actual */
void svGetBitVec32   (svBitVec32*   d, const svBitPackedArrRef   s, int w);
void svGetLogicVec32 (svLogicVec32* d, const svLogicPackedArrRef s, int w);

/* Bit selects */

/* Packed arrays are assumed to be indexed n-1:0,
   where 0 is the index of least significant bit */

/* functions for bit select */

/* s=source, i=bit-index */
svBit svGetSelectBit(const svBitPackedArrRef s, int i);
svLogic svGetSelectLogic(const svLogicPackedArrRef s, int i);

/* d=destination, i=bit-index, s=scalar */
void svPutSelectBit(svBitPackedArrRef d, int i, svBit s);
void svPutSelectLogic(svLogicPackedArrRef d, int i, svLogic s);


/*
 * functions for part select
 *
 * a narrow (<=32 bits) part select is copied between
 * the implementation representation and a single chunk of
 * canonical representation
 * Normalized ranges and indexing [n-1:0] are used for both arrays:
 * the array in the implementation representation and the canonical array.
 *
 * s=source, d=destination, i=starting bit index, w=width
 * like for variable part selects; limitations: w <= 32
 */
/* canonical <-- actual */
void svGetPartSelectBit(svBitVec32* d, const svBitPackedArrRef s, int i,
                        int w);
svBitVec32 svGetBits(const svBitPackedArrRef s, int i, int w);
svBitVec32 svGet32Bits(const svBitPackedArrRef s, int i); // 32-bits
unsigned long long svGet64Bits(const svBitPackedArrRef s, int i); // 64-bits
void svGetPartSelectLogic(svLogicVec32* d, const svLogicPackedArrRef s, int i,
                          int w);

/* actual <-- canonical */
void svPutPartSelectBit(svBitPackedArrRef d, const svBitVec32 s, int i, int w);
void svPutPartSelectLogic(svLogicPackedArrRef d, const svLogicVec32 s, int i,
                        int w);

/* Array querying functions */
```



```
/* These functions are modelled upon the SystemVerilog array querying functions
and use the same semantics*/
/* If the dimension is 0, then the query refers to the packed part (which is one-
dimensional) of an array, and dimensions > 0 refer to the unpacked part of an
array.*/

/* h= handle to open array, d=dimension */
int svLeft(const svOpenArrayHandle h, int d);
int svRight(const svOpenArrayHandle h, int d);
int svLow(const svOpenArrayHandle h, int d);
int svHigh(const svOpenArrayHandle h, int d);
int svIncrement(const svOpenArrayHandle h, int d);
int svLength(const svOpenArrayHandle h, int d);
int svDimensions(const svOpenArrayHandle h);

/* a pointer to the actual representation of the whole array of any type */
/* NULL if not in C layout */
void *svGetArrayPtr(const svOpenArrayHandle);

int svSizeOfArray(const svOpenArrayHandle); /* total size in bytes or 0 if not in C
                                       layout */

/* Return a pointer to an element of the array
   or NULL if index outside the range or null pointer */

void *svGetArrElemPtr(const svOpenArrayHandle, int indx1, ...);

    /* specialized versions for 1-, 2- and 3-dimensional arrays: */
void *svGetArrElemPtr1(const svOpenArrayHandle, int indx1);
void *svGetArrElemPtr2(const svOpenArrayHandle, int indx1, int indx2);
void *svGetArrElemPtr3(const svOpenArrayHandle, int indx1, int indx2, int indx3);

/* Functions for translation between simulator and canonical representations*/
/* These functions copy the whole packed array in either direction. The user is
responsible for allocating an array in the canonical representation. */
/* s=source, d=destination */
/* actual <-- canonical */
void svPutBitArrElemVec32 (const svOpenArrayHandle d, const svBitVec32* s,
                           int indx1, ...);
void svPutBitArrElem1Vec32(const svOpenArrayHandle d, const svBitVec32* s, int
indx1);
void svPutBitArrElem2Vec32(const svOpenArrayHandle d, const svBitVec32* s, int
indx1,
                           int indx2);
void svPutBitArrElem3Vec32(const svOpenArrayHandle d, const svBitVec32* s,
                           int indx1, int indx2, int indx3);

void svPutLogicArrElemVec32 (const svOpenArrayHandle d, const svLogicVec32* s,
                             int indx1, ...);
void svPutLogicArrElem1Vec32( const svOpenArrayHandle d, const svLogicVec32* s,
                              int indx1);
void svPutLogicArrElem2Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
                             int indx1, int indx2);
void svPutLogicArrElem3Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
                             int indx1, int indx2, int indx3);

/* canonical <-- actual */
void svGetBitArrElemVec32 (svBitVec32* d, const svOpenArrayHandle s, int indx1,
...);
```

```
void svGetBitArrElem1Vec32(svBitVec32* d, const svOpenArrayHandle s, int indx1);
void svGetBitArrElem2Vec32(svBitVec32* d, const svOpenArrayHandle s, int indx1,
                           int indx2);
void svGetBitArrElem3Vec32(svBitVec32* d, const svOpenArrayHandle s,
                           int indx1, int indx2, int indx3);

void svGetLogicArrElemVec32 (svLogicVec32* d, const svOpenArrayHandle s, int
indx1,
                             ...);
void svGetLogicArrElem1Vec32(svLogicVec32* d, const svOpenArrayHandle s, int
indx1);
void svGetLogicArrElem2Vec32(svLogicVec32* d, const svOpenArrayHandle s, int
indx1,
                           int indx2);
void svGetLogicArrElem3Vec32(svLogicVec32* d, const svOpenArrayHandle s,
                           int indx1, int indx2, int indx3);

svBit   svGetBitArrElem (const svOpenArrayHandle s, int indx1, ...);
svBit   svGetBitArrElem1(const svOpenArrayHandle s, int indx1);
svBit   svGetBitArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svBit   svGetBitArrElem3(const svOpenArrayHandle s, int indx1, int indx2, int
indx3);

svLogic svGetLogicArrElem (const svOpenArrayHandle s, int indx1, ...);
svLogic svGetLogicArrElem1(const svOpenArrayHandle s, int indx1);
svLogic svGetLogicArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svLogic svGetLogicArrElem3(const svOpenArrayHandle s, int indx1, int indx2, int
indx3);

void svPutLogicArrElem (const svOpenArrayHandle d, svLogic value, int indx1, ...);
void svPutLogicArrElem1(const svOpenArrayHandle d, svLogic value, int indx1);
void svPutLogicArrElem2(const svOpenArrayHandle d, svLogic value, int indx1,
                        int indx2);
void svPutLogicArrElem3(const svOpenArrayHandle d, svLogic value, int indx1, int
indx2,
                        int indx3);

void svPutBitArrElem (const svOpenArrayHandle d, svBit value, int indx1, ...);
void svPutBitArrElem1(const svOpenArrayHandle d, svBit value, int indx1);
void svPutBitArrElem2(const svOpenArrayHandle d, svBit value, int indx1, int
indx2);
void svPutBitArrElem3(const svOpenArrayHandle d, svBit value, int indx1, int
indx2,
                      int indx3);

/* Functions for working with DPI context functions */

/* Retrieve the active instance scope currently associated with the executing
   imported function.
   Unless a prior call to svSetScope has occurred, this is the scope of the
   function's declaration site, not call site.
   Returns NULL if called from C code that is *not* an imported function. */
svScope svGetScope();

/* Set context for subsequent export function execution.
   This function must be called before calling an export function, unless
   the export function is called while executing an extern function. In that
   case the export function shall inherit the scope of the surrounding extern
   function. This is known as the "default scope".
```



```
   The return is the previous active scope (as per svGetScope) */
svScope svSetScope(const svScope scope);

/* Gets the fully qualified name of a scope handle */
const char* svGetNameFromScope(const svScope);

/* Retrieve svScope to instance scope of an arbitrary function declaration.
 * (can be either module, program, interface, or generate scope)
 * The return value shall be NULL for unrecognized scope names.
 */
svScope svGetScopeFromName(const char* scopeName);

/* Store an arbitrary user data pointer for later retrieval by svGetUserData()
 * The userKey is generated by the user. It must be guaranteed by the user to
 * be unique from all other userKey's for all unique data storage requirements
 * It is recommended that the address of static functions or variables in the
 * user's C code be used as the userKey.
 * It is illegal to pass in NULL values for either the scope or userData
 * arguments. It is also an error to call svPutUserData() with an invalid
 * svScope. This function returns -1 for all error cases, 0 upon success. It is
 * suggested that userData values of 0 (NULL) not be used as otherwise it can
 * be impossible to discern error status returns when calling svGetUserData()
 */
int svPutUserData(const svScope scope, void *userKey, void* userData);

/* Retrieve an arbitrary user data pointer that was previously
 * stored by a call to svPutUserData(). See the comment above
 * svPutUserData() for an explanation of userKey, as well as
 * restrictions on NULL and illegal svScope and userKey values.
 * This function returns NULL for all error cases, 0 upon success.
 * This function also returns NULL in the event that a prior call
 * to svPutUserData() was never made.
 */
void* svGetUserData(const svScope scope, void* userKey);

/* Returns the file and line number in the SV code from which the extern call
 * was made. If this information available, returns TRUE and updates fileName  *
and lineNumber to the appropriate values. Behavior is unpredictable if
 * fileName or lineNumber are not appropriate pointers. If this information is  *
not available return FALSE and contents of fileName and lineNumber not
 * modified. Whether this information is available or not is implementation
 * specific. Note that the string provided (if any) is owned by the SV
 * implementation and is valid only until the next call to any SV function.
 * Applications must not modify this string or free it
 */
int svGetCallerInfo(char **fileName, int *lineNumber);

Source-level compatibility include file svdpi_src.h

/* macros for declaring variables to represent the SystemVerilog */
/* packed arrays of type bit or logic */
/* WIDTH= number of bits,NAME = name of a declared field/variable */
#define SV_BIT_PACKED_ARRAY(WIDTH,NAME)/* actual definition goes here */
#define SV_LOGIC_PACKED_ARRAY(WIDTH,NAME)/* actual definition goes here */
```

Annex F
Inclusion of Foreign Language Code

This annex describes common guidelines for the inclusion of Foreign Language Code into a SystemVerilog application. This intention of these guidelines is to enable the redistribution of C binaries in shared object form.

Foreign Language Code is functionality that is included into SystemVerilog using the DPI Interface. As a result, all statements of this annex apply only to code included using this interface; code included by using other interfaces (e.g., PLI or VPI) is outside the scope of this document. Due to the nature of the DPI Interface, most Foreign Language Code is usually be created from C or C++ source code, although nothing precludes the creation of appropriate object code from other languages. This annex adheres to this rule, it's content is independent from the actual language used.

In general, Foreign Language Code is provided in the form of object code compiled for the actual platform. The capability to include Foreign Language Code in object-code form shall be supported by all simulators as specified here. Overview

This annex defines how to:

— specify the location of the corresponding files within the file system

— specify the files to be loaded (in case of object code) or

— provide the object code (as a shared library or archive)

Although this annex defines guidelines for a common inclusion methodology, it requires multiple implementations (usually two) of the corresponding facilities. This takes into account that multiple users can have different viewpoints and different requirements on the inclusion of Foreign Language Code.

— A vendor that wants to provide his IP in form of Foreign Language Code often requires a self-contained method for the integration, which still permits an integration by a third party. This use-case is often covered by a bootstrap file approach.

— A project team that specifies a common, standard set of Foreign Language code, might change the code depending on technology, selected cells, back-annotation data, and other items. This-use case is often covered by a set of tool switches, although it might also use the bootstrap file approach.

— An user might want to switch between selections or provide additional code. This-use case is covered by providing a set of tool switches to define the corresponding information, although it might also use the bootstrap file approach.

NOTE—This annex defines a set of switch names to be used for a particular functionality. This is of informative nature; the actual naming of switches is not part of this standard. It might further not be possible to use certain character configurations in all operating systems or shells. Therefore any switch name defined within this document is a recommendation how to name a switch, but not a requirement of the language.

F.1 Location independence

All pathnames specified within this annex are intended to be location-independent, which is accomplished by using the switch `-sv_root`. It can receive a single directory pathname as the value, which is then prepended to any relative pathname that has been specified. In absence of this switch, or when processing relative filenames before any `-sv_root` specification, the current working directory of the user shall be used as the default value.

F.2 Object code inclusion

Compiled object code is required for cases where the compilation and linking of source code is fully handled by the user; thus, the created object code only need be loaded to integrate the Foreign Language Code into a SystemVerilog application. All SystemVerilog applications shall support the integration of Foreign Language

Code in object code form. Figure F-1depicts the inclusion of object code and its relations to the various steps involved in this integration process.

.

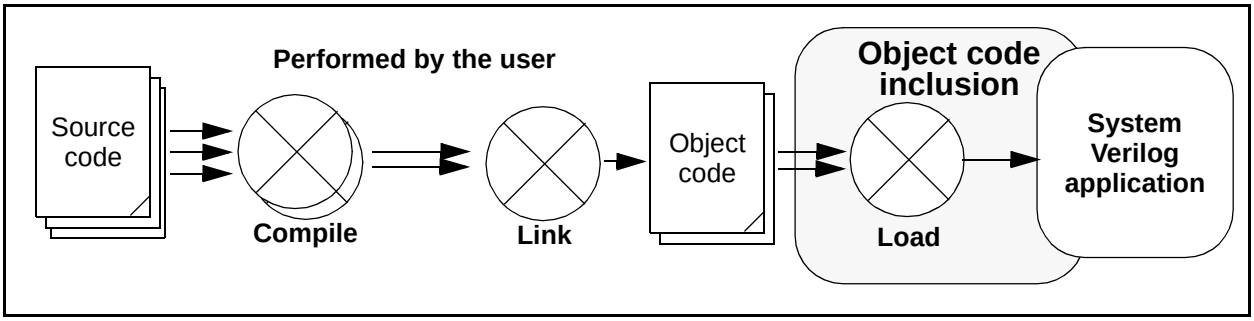


Figure F-1 — Inclusion of object code into a SystemVerilog application

Compiled object code can be specified by one of the following two methods:

1) by an entry in a bootstrap file; see Annex F.2.1 for more details on this file and its content. Its location shall be specified with one instance of the switch `-sv_liblist` *`pathname`*. This switch can be used multiple times to define the usage of multiple bootstrap files.

2) by specifying the file with one instance of the switch `-sv_lib` *`pathname_without_extension`* (i.e., the filename shall be specified without the platform specific extension). The SystemVerilog application is responsible for appending the appropriate extension for the actual platform. This switch can be used multiple times to define multiple libraries holding object code.

Both methods shall be provided and made available concurrently, to permit any mixture of their usage. Every location can be an absolute pathname or a relative pathname, where the value of the switch `-sv_root` is used to identify an appropriate prefix for relative pathnames (see Annex F.1 for more details on forming pathnames).

The following conditions also apply.

— The compiled object code itself shall be provided in form of a shared library having the appropriate extension for the actual platform.

NOTE—Shared libraries use, for example, `.so` for Solaris and `.sl` for HP-UX; other operating systems might use different extensions. In any case, the SystemVerilog application needs to identify the appropriate extension.

— The provider of the compiled code is responsible for any external references specified within these objects. Appropriate data needs to be provided to resolve all open dependencies with the correct information.

— The provider of the compiled code shall avoid interferences with other software and ensure the appropriate software version is taken (e.g., in cases where two versions of the same library are referenced). Similar problems can arise when there are dependencies on the expected runtime environment in the compiled object code (e.g., in cases where C++ global objects or static initializers are used).

— The SystemVerilog application need only load object code within a shared library that is referenced by the SystemVerilog code or by registration functions; loading of additional functions included within a shared library can interfere with other parts.

In case of multiple occurrences of the same file (files having the same pathname or which can easily be identified as being identical; e.g., by comparing the inodes of the files to detect cases where links are used to refer the same file), the above order also identifies the precedence of loading; a file located by method 1) shall override files specified by method 2).

All compiled object code need to be loaded in the specification order similarly to the above scheme; first the content of the bootstrap file is processed starting with the first line, then the set of `-sv_lib` switches is processed in order of their occurrence. Any library shall only be loaded once.

F.2.1 Bootstrap file

The object code bootstrap file has the following syntax.

1) The first line contains the string `#!SV_LIBRARIES`.

2) An arbitrary amount of entries follow, one entry per line, where every entry holds exactly one library location. Each entry consists only of the *`pathname_without_extension`* of the object code file to be loaded and can be surrounded by an arbitrary number of blanks; at least one blank shall precede the entry in the line. The value *`pathname_without_extension`* is equivalent to the value of the switch `-sv_lib`.

3) Any amount of comment lines can be interspersed between the entry lines; a comment line starts with the character # after an arbitrary (including zero) amount of blanks and is terminated with a newline.

F.2.2 Examples

1) If the pathname root has been set by the switch `-sv_root` to `/home/user` and the following object files need to be included:

```
/home/user/myclibs/lib1.so
/home/user/myclibs/lib3.so
/home/user/proj1/clibs/lib4.so
/home/user/proj3/clibs/lib2.so
```

then use either of the methods in Example F-1. Both methods are equivalent.

```
#!SV_LIBRARIES
 myclibs/lib1
 myclibs/lib3
 proj1/clibs/lib4
 proj3/clibs/lib2
```

Bootstrap file method

```
...
-sv_lib myclibs/lib1
-sv_lib myclibs/lib3
-sv_lib proj1/clibs/lib4
-sv_lib proj3/clibs/lib2
...
```

Switch list method

Example F-1 — Using a simple bootstrap file or a switch list

2) If the current working directory is `/home/user`, using the series of switches shown in Example F-2 (left column) result in loading the following files (right column).

```
-sv_lib svLibrary1
-sv_lib svLibrary2
-sv_root /home/project2/shared_code
-sv_lib svLibrary3
-sv_root /home/project3/code
-sv_lib svLibrary4
```

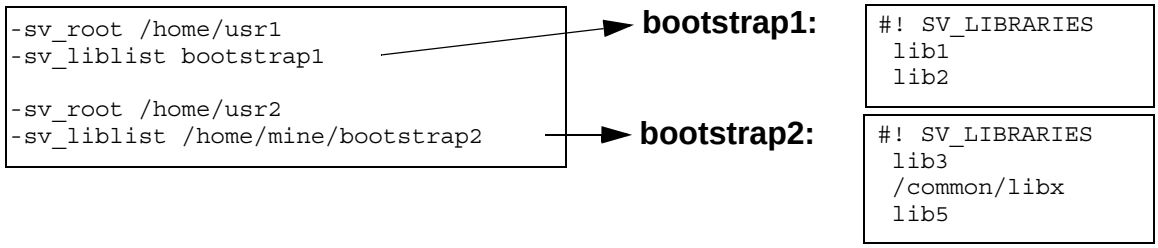
Switches

```
/home/user/svLibrary1.so
/home/user/svLibrary2.so

/home/project2/shared_code/svLibrary3.so

/home/project3/code/svLibrary4.so
```

Files

Example F-2 — Using a combination of `-sv_lib` and `-sv_root` switches

3) Further, using the set of switches and contents of bootstrap files shown in Example F-3:



Example F-3 — Mixing `-sv_root` *and bootstrap files*

results in loading the following files:

```
/home/usr1/lib1.ext
/home/usr1/lib2.ext
/home/usr2/lib3.ext
/common/libx.ext
/home/usr2/lib5.ext
```

where *ext* stands for the actual extension of the corresponding file.

Annex G
Glossary

(Informative)

Assertion — An assertion is a statement that a certain property must be true. For example, that a read_request must always be followed by a read_grant within 2 clock cycles. Assertions allow for automated checking that the specified property is true, and can generate automatic error messages if the property is not true. SystemVerilog provides special assertion constructs, which are discussed in Section 17.

DPI — Direct Programming Interface. This is an interface between SystemVerilog and foreign programming languages permitting direct function calls from SystemVerilog to foreign code and from foreign code to SystemVerilog. It has been designed to have low inherent overhead and permit direct exchange of data between SystemVerilog and foreign code.

Elaboration — Elaboration is the process of binding together the components that make up a design. These components can include module instances, primitive instances, interfaces, and the top-level of the design hierarchy. SystemVerilog requires a specific order of elaboration, which is presented in Section 18.2.

Enumerated type — Enumerated data types provide the capability to declare a variable which can have one of a set of named values. The numerical equivalents of these values can be specified. Enumerated types can be easily referenced or displayed using the enumerated names, as opposed to the enumerated values. Section 3.10 discusses enumerated types.

Interface — An interface encapsulates the communication between blocks of a design, allowing a smooth migration from abstract system-level design through successive refinement down to lower-level register-transfer and structural views of the design. By encapsulating the communication between blocks, the interface construct also facilitates design re-use. The inclusion of interface capabilities is one of the major advantages of SystemVerilog. Interfaces are covered in Section 19.

LRM — LRM is an abbreviation for Language Reference Manual. “SystemVerilog LRM” refers to this document. “Verilog LRM” refers to the IEEE manual “1364-2001 IEEE Standard for Verilog Hardware Description Language 2001”. See Annex H for information about this manual.

Packed array — Packed array refers to an array where the dimensions are declared before an object name. Packed arrays can have any number of dimensions. A one-dimensional packed array is the same as a vector width declaration in Verilog. Packed arrays provide a mechanism for subdividing a vector into subfields, which can be conveniently accessed as array elements. A packed array differs from an unpacked array, in that the whole array is treated as a single vector for arithmetic operations. Packed arrays are discussed in detail in Section 4.

Process — A process is a thread of one or more programming statements which can be executed independently of other programming statements. Each initial procedure, always procedure and continuous assignment statement in Verilog is a separate process. These are static processes. That is, each time the process starts running, there is an end to the process. SystemVerilog adds specialized always procedures, which are also static processes, and dynamic processes. When dynamic processes are started, they can run without ending. Processes are presented in Section 9.

SystemVerilog — SystemVerilog refers to the Accellera standard for a set of abstract modeling and verification extensions to the IEEE 1364-2001 Verilog standard. The many features of the SystemVerilog standard are presented in this document.

Unpacked array — Unpacked array refers to an array where the dimensions are declared after an object name. Unpacked arrays are the same as arrays in Verilog, and can have any number of dimensions. An unpacked array differs from a packed array, in that the whole array cannot be used for arithmetic operations. Each element must be treated separately. Unpacked arrays are discussed in Section 4.

Verilog — Verilog refers to the IEEE 1364-2001 Verilog Hardware Description Language (HDL), commonly called Verilog-2001. This language is documented in the IEEE manual “1364-2001 IEEE Standard for Verilog

Hardware Description Language 2001”. See Annex H for information about this manual.

VPI — Verilog Procedural Interface. The 3rd generation Verilog Programming Language Interface (PLI), providing object-oriented access to Verilog behavioral, structural, assertion and coverage objects.

Annex H
Bibliography

(Informative)

[B1] IEEE Std. 754-1985, IEEE Standard for Binary Floating-Point Arithmetic 1985. ISBN 1-5593-7653-8. IEEE Product No. SH10116-TBR.

[B2] IEEE Std. 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog¨ Hardware Description Language 1995. ISBN 0-7381-3065-6. IEEE Product No. WE94418-TBR.

[B3] IEEE Std. 1364-2001, IEEE Standard for Verilog Hardware Description Language 2001. ISBN 0-7381-2827-9. IEEE Product No. SH94921-TBR.

Index

Symbols

!?= wild inequality **46**
177
clocked sequence **177**
#1step **126**
$assertkill **228**
$assertoff **228**
$asserton **228**
$bits **23, 225**
$bitstoshortreal **23**
$cast **23**
$cast() **86**
$countones **175**
$dimensions **28, 226**
$error **146, 227**
$exit **144**
$fatal **146, 227**
$fell **158**
$high **28, 226**
$increment **28, 226**
$info **146, 227**
$inset **174, 228**
$insetz **174, 228**
$isunknown **174, 229**
$left **28, 226**
$length **28, 226**
$low **28, 226**
$onehot **174, 228**
$onehot0 **174, 228**
$past **174**
$right **28, 226**
$root **190–191**
$rose **158**
$shortrealtobits **23**
$srandom() **112**
$stable **158**
$urandom **111**
$urandom_range() **111**
$warning **146, 227**
’ cast operator **22**
*= operator **47**
+= operator **47**
.* port connections **199**
.name port connections **198**
/= operator **47**
-= operator **47**
=> implication **102**
=?= wild equality **46**
\ line continuation **232**
\a bell **4**
\f form feed **4**
\v vertical tab **4**
\x02 hex number **4**
‘ ‘ double back tick **232**
‘define **232**
|=> multi-clock sequence **177**

Numerics

2-state types **8**
4-state types **8**

A

Active region **126**
aggregate expressions **52**
alias **42**
always @* **63**
always_comb **63**
always_ff **63–64**
always_latch **63–64**
and **158–159**
anding sequences **158**
array literals **5**
array part selects **27**
array querying functions **28, 226**
array slices **27**
arrays **25**
assert **145**
assertion API **246–254**
assertion system functions **228**
assertion system tasks **227–228**
assertions **145–189, 343**
assign **53, 62, 234**
assignment operators **45**
associative array methods
- delete() **34**
- exists() **34**
- first() **35**
- last() **35**
- next() **35**
- num() **34**
- prev() **36**

associative arrays **31–37**
atobin() **12–13**
atohex() **12–13**
atoi() **12**
atooct() **12–13**
atoreal() **13**
attributes **44**
automatic **38, 40, 68**
automatic tasks **70**

B

back() **306**
before **99**
bell **4**
bind **187**

bintoa() **13**
bit **6–8**
block name **59–60**
blocking assignments **55**
boolean expression **150**
break **53, 58, 60**
built-in methods **48**
byte **7–8**

C

casting **22–23**
chandle **6, 8, 80**
check **146**
class **21, 78–93**
clear() **308**
clock tick **147**
combinational logic **63**
compare() **12**
concatenation **49**
concurrent assertions **147**
conditional operator **52**
configurations **224**
const **38, 87**
constants **38**
constraint blocks **98**
constraint_mode() method **96, 109**
context **77, 238–240**
continue **53, 58, 60**
continuous assignment **64**
cover **180–181**
coverage API **255–265**

D

data declarations **38**
data types **6**
data() **305**
deassign **53, 62, 234**
decrementor operator **45**
defparam **222, 234**
delete() **29, 34**
Direct Programming Interface (DPI) **236–245**
disable **60**
disable fork **63, 66**
dist **99, 101**
distribution **101**
do...while loop **53, 57**
double **8**
dynamic array methods
  delete() **29**
  size() **29**
dynamic arrays **28**
dynamic processes **63**

E

edge event **158**
elaboration **190, 343**
empty() **305**
encapsulation **87**
enum **15–16**
enumerated type methods
  first() **18**
  last() **18**
  name() **19**
  next() **18**
  num() **19**
  prev() **19**
enumerated types **15–16, 343**
eq() **304**
erase() **307**
erase_range() **307**
exists() **34**
export **76, 212, 244–245**
extends **86**
extern **90, 212, 216**

F

final **53, 58**
finish() **306**
first() **18, 35**
first_match **164**
float **8**
for loops **57**
force **53**
fork...join **64**
forkjoin **203, 213, 216**
form feed **4**
front() **306**
functions **72**
functions in interfaces **212**
functions, arg passing by name **75**
functions, default args **75**
functions, exporting **76, 212, 244–245**
functions, importing **76, 212, 240**

G

garbage collection **81**
getc() **12**
goto **58**

H

handle **80**
hextoa() **13**
hierarchical names **202**

I

icompare() **12**
if...else **103**
if..else **102**
iff **61, 147**
immediate assertions **145**

implication **102**
import **76, 212, 240**
Inactive region **126**
incrementor operator **45**
inheritance **84, 100**
insert() **306**
insert_range() **307**
inside **99–100**
int **6–8**
integer **8**
integer literals **3**
integral **8**
interface **203–220, 343**
intersect **161**
introduction to SystemVerilog **1**
itoa() **13**

J
join_any **63, 65**
join_none **63, 65**

L
labels **60**
last() **18, 35**
latched logic **64**
len() **11**
libraries **224**
library map files **224**
linked ists **303–308**
list methods
- back() **306**
- clear() **308**
- data() **305**
- empty() **305**
- eq() **304**
- erase() **307**
- erase_range() **307**
- finish() **306**
- front() **306**
- insert() **306**
- insert_range() **307**
- neq() **305**
- next() **304**
- pop_back() **306**
- pop_front() **306**
- prev() **304**
- purge() **308**
- push_front() **305**
- set() **307**
- size() **305**
- start() **306**
- swap() **308**

literal values **3**
local **87**
localparam **222**
logic **6, 8**
longint **6–8**
LRM **343**

M
matched **178**
memory management **92**
methods **81**
methods, built-in **48**
modport **203, 209**
module instantiation **198–199**
multiple dimension arrays **26**

N
name() **19**
named blocks **59**
named port connections **198**
NBA region **126**
neq() **305**
nested identifiers **202**
nested modules **192**
new **81**
next() **18, 35, 304**
nonblocking assignments **55**
null **9, 13, 93**
num() **19, 34**

O
object handle **79–80**
object-oriented **78**
Observed region **126**
octtoa() **13**
operator associativity **47**
operator precedence **47, 155**
or **162**
oring sequences **162**
overview of SystemVerilog **1**

P
packed arrays **25–26, 46, 343**
parameter **91, 222**
parameter type **223**
part selects **27**
PLI callbacks **129**
pointer **80**
polymorphism **88**
pop_back() **306**
pop_front() **306**
port connections, .* **199**
port connections, .name **198**
port declarations **195**
post_randomize() method **96, 106**
Post-NBA region **126**
Post-observed region **126**

Postponed region **126**
pre_randomize() method **96, 106**
Pre-active region **126**
precedence **47**
Pre-NBA region **126**
Preponed region **126**
prev() **19, 36, 304**
priority **56–57**
process **343**
process execution threads **65**
program block **141–144**
property **148, 175**
protected **87**
pure **77, 238–239**
purge() **308**
push_back() **305**
push_front() **305**
putc() **11**

R

rand **96**
rand_mode() method **96, 108**
randc **96**
random constraints **94–114**
random distribution **101**
random implication **102**
Random Number Generator (RNG) **112**
randomization methods
- constraint_mode() **96, 109**
- post_randomize() **96, 106**
- pre_randomize() **96, 106**
- rand_mode() **96, 108**
- randomize() **94, 106**

randomize() method **94, 106**
randomize()...with **107**
Reactive region **126**
real **4, 6, 8, 47**
real literals **4**
realtoa() **13**
ref **74**
reg **6, 8**
regions
- Active **126**
- Inactive **126**
- NBA **126**
- Observed **126**
- Post-NBA **126**
- Post-observed **126**
- Postponed **126**
- Pre-active **126**
- Pre-NBA **126**
- Preponed **126**
- Reactive **126**

release **53**
repetition **155**
return **53, 58, 60, 70, 72**
RNG (Random Number Generator) **112**

S

scheduling semantics **125–129**
sequence **150, 153**
sequence expression **151**
sequential logic **64**
set() **307**
shortint **7–8**
shortreal **4, 6, 8, 47**
signed types **8**
singular **22**
size() **29, 305**
slices **27**
solve...before **99, 105**
sparse arrays, see associative arrays
specparam **222**
start() **306**
statement labels **59**
static **38–40, 68, 82**
static processes **63**
static tasks **70**
step **4, 126**
stratified event scheduler **125**
string **9–13**
string literals **4**
string methods
- atobin() **12–13**
- atohex() **12–13**
- atoi() **12**
- atooct() **12–13**
- atoreal() **13**
- bintoa() **13**
- compare() **12**
- getc() **12**
- hextoa() **13**
- icompare() **12**
- itoa() **13**
- len() **11**
- octtoa() **13**
- putc() **11**
- realtoa() **13**
- substr() **12**
- tolower() **12**
- toupper() **12**

struct **19**
structure literals **5**
structures **19**
subclasses **84**
substr() **12**
super **85**
swap() **308**

SystemVerilog, overview **1**
SystemVerilog,version numbers **1**

T

tasks **69**
tasks in interfaces **212**
tasks, arg pass by name **75**
tasks, default args **75**
this **82**
threads **65**
time **8**
time literals **4**
time unit **4**
tolower() **12**
top level **190**
toupper() **12**
type **223**
typedef **6, 14, 92**

U

union **20**
unions **19**
unique **56–57**
unpacked arrays **25–26, 343**
unsigned types **8**
unsized literals **4**
user-defined types **14**

V

variable initialization **39**
VCD **231**
vertical tab **4**
virtual **87–88**
void **8**
void functions **68, 72**

W

wait fork **63, 66**
while **53, 57**
wild-card operators **46**
with **107**


accellera