SystemVerilog 3.1 Draft 3

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

Editor's note:
Draft 2 reflects changes made to the released SystemVerilog 3.0 LRM, as well as to draft 1 of this document. The primary source of the changes in this draft are from the SV-EC (changes 1 through 41) and SV-BC (in Edits_As_Of_02_12_201.doc).

Legend:
~~– magenta strike through text~~ indicates text to be deleted from the 3.0 LRM.
– blue text with change bars indicates text that was added for draft 1.
– blue underlined text with change bars indicates text that was added for draft 2.
– blue double-underlined text with change bars indicates text that was added for draft 3.
–red text in boxes indicate editor notes that need to resolve, or that the editor needs to implement in a future draft.

SystemVerilog 3.~~0~~1 Draft 3

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

~~Approved by the Accellera Board of Directors on 3 June 2002.~~ **This is a preliminary draft for review and development purposes within the Accellera SystemVerilog committees only. Information within this document has not been approved by Accellera, and is subject to change.**

Acknowledgements

Editor’s Note: This page will need to be updated at the conclusion of defining SystemVerilog 3.1

This SystemVerilog 3.0 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 primary contributors to the development of SystemVerilog 3.0 include:

Vassilios Gerousis, Chair
Dave Kelf, Co-chair
Stefen Boyd
Dennis Brophy
Kevin Cameron
Cliff Cummings
Simon Davidmann
Tom Fitzpatrick
Peter Flake
Harry Foster
Paul Graham
David Knapp
Adam Krolnik
Mike McNamara
Phil Moorby
Prakash Narian
Anders Nordstrom
Rajeev Ranjan
John Sanguinetti
David Smith
Alec Stanculescu
Stuart Sutherland
Bassam Tabbara
Andy Tsay

Stuart Sutherland served at the technical editor for this document. Stefen Boyd served as editor of the BNF annex.

Table of Contents

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.~~0~~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.~~0~~1.

SystemVerilog 3.0 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, char, 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 test-bench data-types and functions.

  - Built-in types: string, associative array, and dynamic array.
  - Pass by reference subroutine parameters.
- **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

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 ::=          // from Annex A.8.4
            unsigned_number time_unit
          | fixed_point_number time_unit
time_unit ::= s | ms | us | ns | ps | fs

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 ::= non_zero_decimal_digit { _ | decimal_digit}
real_number ::=
            fixed_point_number
          | unsigned_number [ . unsigned_number ] exp [ sign ] unsigned_number
fixed_point_number ::= unsigned_number . unsigned_number
exp ::= e | E
unsigned_number1 ::= decimal_digit { _ | decimal_digit }
```

EC-CH1 string ::= " { Any_ASCII_Characters~~_except_new_line~~ } "   // 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.

BC-7
BC-7b

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**). For example:

```
0.1ns
40ps
```

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.

```
char 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.

```
char 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 annex C).

2.7 Array literals

BC-7b

Arrays 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

BC-7c

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.

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 will not be confused with the Verilog-2001 **real** type.

Verilog-2001 has net data types, which may 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 section 3.3.2.

EC-CH102

SystemVerilog 3.1 adds **string, handle** 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 ] { packed_dimension }
            | type_declaration_identifier
            | non_integer_type
            | struct { { struct_union_member } }
            | union { { struct_union_member } }
            | enum { enum_identifier [ = constant_expression ]
              { , enum_identifier [ = constant_expression ] } }
            | void
integer_type ::= integer_vector_type | integer_atom_type
integer_atom_type ::= byte | char | shortint | int | longint | integer
integer_vector_type ::= bit | logic | reg
non_integer_type ::= time | shortreal | real | realtime | $built-in
signing ::= [ signed ] | [ unsigned ]
simple_type ::= integer_type | non_integer_type | type_identifier
struct_union_member ::= data_type  list_of_variable_identifiers_or_assignments ;
```

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

EC-CH6
EC-CH31
BC46

`char`	2-state C-compatible data type, ~~usually an~~ 8 bit signed integer (ASCII) ~~or a short int (Unicode)~~
`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
`bit`	2-state SystemVerilog data type, user-defined vector size
`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
~~`string`~~	~~arbitrary length character string~~
~~`class`~~	~~object-oriented class~~

EC-CH2

3.3.1 Integral types

EC-CH3

The term *integral* is used throughout this document to refer to the data types that can represent a single ~~integral value. These are all the~~ basic integer data type~~s~~, packed `struct`, packed `union`, `enum`, ~~and~~ or `time`.

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

Types which can have unknown and high impedance values are called 4-state types. These are **`logic`**, **`reg`** and **`integer`**. 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 should 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 **`char`**, **`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 Other basic data types~~

Editor’s Note: I took the liberty of elevating the three sub-subsections within 3.4 of the SV 3.0 LRM to subsection level, to be more consistent with the levels describing the new string and event data types. Hence:
- SV 3.0 LRM Section 3.4.1 “Time data types” becomes SV 3.1 LRM Section 3.4
- SV 3.0 LRM Section 3.4.2 “Real and shortreal data types” becomes SV 3.1 LRM Section 3.5
- SV 3.0 LRM Section 3.4.3 “Void data type” becomes SV 3.1 LRM Section 3.6

3.4 Time data types

Time is a special data type. It is a 64 bit integer of time steps. The default time step follows the rules of IEEE Verilog standard. The time step can be changed by the **`timeprecision`** declaration. It can also be changed by a **`` `timescale``** directive.

The **`timeprecision`** declaration affects the local accuracy of delays.

```
module m;
   timeprecision 0.1ns;
   initial #10.11ns a = 1; // round to #10.1ns according to time precision
endmodule
```

The **`timeunit`** declaration is used to set the current time unit. When a literal time is expressed in SystemVerilog, it can be given with explicit time units, e.g. `12ns`. If no time units are specified, the literal number is multiplied by the current time unit. Time values are scaled to the time precision of the module, following the rules of Verilog-2001. An integer or real variable is cast to a time value by using the integer or real as a delay.

For example:

```
#10.11; // multiply by time unit and round according to time precision
```

See section 17.6 for more information on setting the time units and time precision.

3.5 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`**.

3.6 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.

EC-CH102

3.7 Handle data type

The **`handle`** data type represents storage for pointers passed across the DirectC interface. The size of this type is platform dependent and must be at least large enough to hold a pointer on the machine in which the simulator is running. The syntax to declare a handle is as follows:

Editor's Note: Is the "DirectC" .name to be used in SystemVerilog?

```
handle variable name ;
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

Editor's Note: I took the liberty of adding "handle" to the keyword list in Annex B

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

Editor's Note: Is "null" also a SystemVerilog keyword?.

— only the following operators are valid on handle variables:

— equality (==), inequality (!=) with another handle or with null

— case equality (===), case inequality with another handle or with null (same semantics as == and !=)

— only the following assigments can be made to a handle

— assignment from another handle

— assigment to null

— handles can be inserted into associative arrays (refer to section 4.9), but no guarantees will be made on relative ordering of any two entries in such an associative array, even between successive runs of the same simulation.

— handles can be used within a class

— handles may be passed as arguments to functions or tasks

— handles can be returned from functions

The use of handles are restricted as follows:

— ports may not have the handle data type

— handles may not be assigned to variables of any other type

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

- handles cannot be used:
  - in any expression other than as permited above
  - as a ports
  - in sensitivity lists or event expressions
  - in continuous assigments
  - in structures or unions
  - in packed arrays

3.8 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, which are described in annex C.

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 (see annex C).

EC-CH4 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. ~~Uninitialized variables of type **string** are initialized to “”.~~

The syntax to declare a string is:

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

Editor’s Note: Replace preceding syntax line with BNF excerpt, once available.

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.

EC-CH21 A string literal is implicitly converted to string type when it is assigned to a variable of type **string** or is used in an expression involving string type operands. ~~A string literal and a concatenation or replication of string literals are the only types of packed arrays that are allowed to be assigned to variables of type **string**.~~ A variable of type **string** can be assigned an expression of type **string**, string literal, or packed array.

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:

```
    char 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 will 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
```

For example:

EC-CH5

EC-CH22

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

r = a;                  // OK
b = r;                  // ~~Error~~ OK (implicit cast, some implementations
                        //    may 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 may be of type **`string`**. Or one of them may 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 may be of type **`string`**, or one of them may be a string literal.

`{Str1,Str2,...,Strn}`	Concatenation. Each string may be of type **`string`** or a string literal (it will be implicitly converted to **`string`**). If at least one string is of type **`string`**, then the expression evaluates to the concatenated string and is of type **`string`**. If all the strings 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.

Table 3-2: String operators

EC-CH8

Operator	Semantics
`{multiplier{Str}}`	Replication. `Str` may 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. See annex C for detailed descriptions of the various string methods available.

SystemVerilog also includes a number of special methods to work with strings.

— **`len()`** — returns the length of the string

— **`putc()`** — replaces a character in a string

— **`getc()`** — returns the ASCII code of a character in a string

— **`toupper()`** — returns a string with all characters converted to uppercase

— **`tolower()`** — returns a string with all characters converted to lowercase

— **`compare()`** — compares two strings character by character

— **`icompare()`** — compares two strings character by character in a case insensitive mode

— **`substr()`** — returns a sub-string from within a string

— **`atoi()`**— returns the integer corresponding to the ASCII decimal representation of a string

— **`atohex()`**— returns the integer corresponding to the ASCII hexadecimal representation of a string interprets the string as hexadecimal.

EC-CH7

— **`atooct()`**— returns the integer corresponding to the ASCII ~~ocatal~~ octal representation of a string interprets the string as octal.

— **`atobin()`**— returns the integer corresponding to the ASCII binary representation of a string

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

— **`itoa(`***`i`***`)`** stores the ASCII decimal representation of an integer as a string (inverse of `atoi`).

EC-CH9

— **`hextoa(`***`i`***`)`** stores the ASCII hexadecimal representation of an integer as a string (inverse of `atohex`).

— **`octtoa(`***`i`***`)`** stores the ASCII octal representation of an integer as a string (inverse of `atooct`).

— **`bintoa(`***`i`***`)`** stores the ASCII binary representation of an integer as a string (inverse of `atobin`).

— **`realtoa(`***`r`***`)`** stores the ASCII representation of a real as a string (inverse of `atoreal`).

These built-in string methods are described in annex C.

3.9 Event data type

EC-CH11

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, however, SystemVerilog events can also have a persistent triggered state~~, that is, the synchronization object can be either ON or OFF~~ that lasts for the duration of the entire time step. Also, event variables can be assigned the special value **null**, which breaks the association between the synchronization object and the event variable, or be assigned another event variable, in which case more than one event variable will refer to the same synchronization object. Events can be passed as arguments to tasks.

The syntax to declare an event is:

EC-CH11

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

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

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

EC-CH11

~~If an initial value is not specified then the variable is initialized to a new synchronization object whose triggered state is OFF~~.

~~If the event is assigned **null**, the event behaves as if it were permanently triggered (ON state).~~

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

The declaration **event bit** creates a persistent event (as described in section 12.6.2).

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 bit blast;           // persistent event
event bit empty = null;    // persistent event variable
```

Event operations and semantics are discussed in detail in section 12.6.

3.10 User-defined types

```
type_declaration ::=          // from Annex A.2.1.3
          typedef data_type type_declaration_identifier ;
        | typedef interface_identifier { [ constant_expression ] } . type_identifier
             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 3.12, below), and as parameters.

3.11 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)

Editor’s Note: Update preceding BNF excerpt with new BNF, once available.

An enumerated type provides the capability to declare sets 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.

BC10
BC26-1

~~The format control string "%n" can be used to display the enumerated name. Any format control string which can be used to display an integer value can be used to display an enumerated value.~~

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
```

BC17b

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 and 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;
```

BC17d

```
// 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 [3:0] {bronze=5'h13, silver, gold=3'h5} medal4;

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

EC-CH26

~~The type is checked in assignments, arguments and relational operators (which check the values).~~ Type checking of enumerated types used in assignments, as arguments and with operators is covered in section 3.11.3. 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.

3.11.1 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
```

EC-CH12

~~SystemVerilog also provides a shorthand notation for declaring enumerated types:~~

```
~~enum enum_type [integer_type [signing]{packed_dimension}] { value_list };~~
```

~~This modified form declares the enumeration and creates a type called *enum_type*. This shorthand notation is similar to the way in which C++ extends C, and allows an enumerated type to be created as part of the enumeration declaration, without the need for a typedef.~~

~~For example, to create an enumerated type called StreetLight:~~

```
~~enum StreetLight {red, yellow, green};~~
```

~~To create an enumeration type called Colors whose values are of type bit[1:0].~~

```
~~enum Colors bit [1:0] { unknown = 'x, red = 1, green, blue };~~
```

~~The shorthand form cannot be used to declare both a type and variables of that type. For example, the following is an error:~~

```
~~enum Boolean { FALSE, TRUE } myvar;~~
```

3.11.2 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
`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:

EC-CH13

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

3.11.3 Type checking

SystemVerilog enumerated types are strongly typed, thus, a variable of type **enum** cannot be 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:

EC-CH14

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

EC-CH27

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 3.14 and 3.15.

3.11.4 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:

EC-CH15

```
~~enum Colors { red, green, blue, yellow, white, black };~~
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).

EC-CH32

~~3.10.5 Increment and decrement operators on enumerated types~~

~~SystemVerilog attaches a special semantics to the operators ++, --, +=, and -= when applied to variables of enumerated type, as described below:~~

-

~~Table 3-4: Increment and decrement operations on enumerated types~~

~~Operator~~	~~Description~~
~~`++enumVar`~~	~~Assigns the next enumeration member (according to the definition order) to enumVar. A wrap around to the first enumeration value occurs when incrementing the last enumeration value.~~
~~`--enumVar`~~	~~Assigns to enumVar the previous enumeration member (according to the definition order). A wrap around to the last enumeration value occurs when decrementing the first enumeration value.~~
~~`enumVar += N`~~	~~Assigns to enumVar its Nth next value. A wrap to the start of the list occurs when the end of the list is reached.~~
~~`enumVar -= N`~~	~~Assigns to enumVar its Nth previous member. A wrap to the end of the list occurs when the start of the list is reached.~~

~~Note that~~

```
enumVar += 5;
```

~~is different from~~

```
enumVar = enumVar + 5;
```

~~The former is legal while the latter is illegal and requires an explicit cast (see sections 3.14 and 3.15), either as:~~

```
enumVar = EnumType'(enumVar + 5);   // static cast (fast, unsafe)
$cast ( enumVar, enumVar + 5);      // dynamic cast (safe, slower)
```

EC-CH32

3.11.5 Methods for iterating over enumerated types

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

3.11.5.1 first()

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

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

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

3.11.5.2 last()

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

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

The **last()** function return the value of the last member of the enumeration enum.

3.11.5.3 next()

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

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

The **next()** function 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()`** function returns the first member.

3.11.5.4 prev()

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

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

The **`prev()`** function 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()`** function returns the last member.

3.11.5.5 num()

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

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

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

3.11.5.6 name()

The syntax 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()`** function returns the empty string.

Example: 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
```

3.12 Structures and Unions

```
data_type ::=           // from Annex A.2.2.1
              ...
             | struct { { struct_union_member } }
             | union { { struct_union_member } }
struct_union_member ::= 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.

BC-5

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, which defaults to 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
```

BC-no #

If any data type within a packed structure is ~~masked~~ 2-state, the whole structure is treated as ~~masked~~ 2-state. Any ~~unmasked~~ 4-state members are converted as if cast, i.e. an X will be read as 0 if it is in a member of type bit. One or more elements of the packed array may 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;
```

BC-5
BC8-2b
BC-8-8

~~A packed union contains members that are packed structures or arrays of the same size.~~ A packed union shall contain members that ~~are~~ must be packed structures, or packed arrays or integer data types of the same size. This ensures that you can read back a union member that was written as another member. ~~If any member is 4-state, the whole union is 4-state.~~ 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.

Editor’s Note: BC-5 and BC8-8 modified the same sentence. I merged the two changes together.

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.

3.13 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.

The object-oriented class extension allows objects to be created and destroyed dynamically. Classes can also be passed around by reference via handles, adding a safe-pointer capability.

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.

3.14 Casting

```
primary ::=     // from Annex A.8.4
            ...
          | simple_type_or_number ' ( expression )
          | simple_type_or_number ' { expression { , expression } }
          | simple_type_or_number ' { expression { expression } }

simple_type_or_number ::= // from Annex A.2.2.1
            simple_type | number
simple_type ::= // from Annex A.2.2.1
            integer_type | non_integer_type | type_identifier
```

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

A data type may 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**, its value is rounded as in Verilog. So the conversion can lose information. When a **shortreal** is converted to 32 bits, its bit pattern is preserved, which means it can be converted back to the same value without any loss of information. This technique can also be used for structures, where 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:

EC-CH34

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

or

EC-CH34

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

EC-CH71

A singular type includes packed arrays (and structures) and all other data types except unpacked structures, unpacked arrays, and handles (used for the C interface).

The *dest_var* is the variable to which the assignment is made. ~~It can be any scalar singular (non-unpacked array) type (bit, integer, string, enumerated type, event, or object handle).~~

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.

EC-CH28

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

EC-CH29

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 ~~set to its corresponding uninitialized value, which depends on the type of the variable~~ left unchanged.

EC-CH34

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 ~~scalars~~ singulars.

For example:

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

To check if the assignment will succeed, one can use:

```
    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 may 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. Unpacked arrays can be made up of any type.

Integer types with predefined widths cannot have packed array dimensions declared. These types are: **char**, **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;
```

EC-CH30

Unpacked arrays can be made of any scalar (non-unpacked-array) type. ~~VeraLite~~ SystemVerilog enhances fixed-size unpacked arrays in that in addition to all other SystemVerilog types, unpacked arrays may also be made of object handles (see section 11.4) and events (see section 12.6).

Note: ~~VeraLite~~ SystemVerilog accepts a single number (not a range) to specify the size of an unpacked arrays, like C. SystemVerilog should accept this type of declaration as a shorthand notation, 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];`

BC56

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]`

BC56

— 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 ~~may~~ can be a packed array.

BC42-4

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 bit int)
```

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 will 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 will 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 may 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 may be variable. The syntax of Verilog-2001 is used.

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

Slices of an array can only apply to one dimension, but other dimensions may 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 22.3.

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 syntax of the new function is:

EC-CH35

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

EC-CH76

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 syntax 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 syntax 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

EC-CH36
EC-CH75

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
```

EC-CH75

```
    A = B;            // ok. Compatible type and same size
    A = C;            // ~~compile-time~~ type check error: different sizes
```

EC-CH40
EC-CH75

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 ~~to~~ with a fixed-size array, this operation requires a run-time check that may result in an error.

EC-CH75

EC-CH75

```
    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 ~~100~~ 8 elements

    A = B;                  // ok. Compatible type and same size
    A = C;                  // ~~run-time~~ 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[5:1] = { "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

EC-CH77

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 parameters to subroutines requires that the actual parameter 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 parameter, 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 parameter, an array of 4 strings. This task will accept the following actual parameters:

— `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 parameter, a dynamic array of 4 strings. This task will accept any one-dimensional array of strings or any dynamic array of strings.

4.9 Associative arrays

EC-CH41

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~~s~~ 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.

EC-CH41

The syntax to declare ~~associative~~ 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`* (optional) is the data-type to be used as an index. If no index 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 Unspecified index type

Example: **int** `array_name [];`

Associative arrays that do not specify an index type have the following properties:

EC-CH37

— The array can be indexed by any integral data type~~, including integers, packed arrays of arbitrary length, string literals, and packed structures~~. Since the indices can be of different sizes, the same numerical value may 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.

EC-CH38

— Non-integral index types are illegal and result in a ~~compiler~~ 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:

EC-CH38

— Indices can be strings or string literals of any length. Other types are illegal and shall result in a ~~compiler~~ 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:

EC-CH38

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

EC-CH78

Table 4-1: ~~Invalid array index default initial value~~ Value read from a nonexistent associative array entry

Type of Array	Value Read
4-state integral type	'X
2-state integral type	'0
enumeration	first element in the enumeration

EC-CH78

**Table 4-1: ~~Invalid array index default initial value~~
Value read from a nonexistent associative array entry**

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:

EC-CH72

```
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", map.num );  // prints "3 entries"
```

4.10.2 delete()

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

EC-CH39

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

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

EC-CH41

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

EC-CH70

EC-CH73

If the index is not specified then the ~~delete~~ **`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( var index );
```

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

EC-CH41

The **`first()`** ~~function~~ 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( var index );
```

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

The **`last()`** function 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( var index );
```

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

The **`next()`** function 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( var 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, 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 will copy only as much data as will fit into the argument. For example:

```
string   aa[];
char     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.

Section 5 Data Declarations

5.1 Introduction (informative)

There are several forms of data in SystemVerilog: literals (see section 2), parameters (see section 19), 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 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 char colon1 = ":" ;
specparam int delay = 10 ; // specparams are used for specify blocks
const logic flag = 1 ;
```

A local parameter is a constant which is calculated at elaboration time, and can depend upon parameters or other local parameters at the top level or in the same module or interface.

A specify parameter is also calculated at elaboration time, but it may be modified by the PLI, and so cannot be used to set parameters or local parameters.

A constant declared with the const keyword is calculated after elaboration. This means that it can contain an expression with any hierarchical path name. This constant is like a variable which cannot be written.

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

A constant expression contains literals and other named constants.

SystemVerilog enhancements to **`parameter`** constant declarations are presented in section 19. 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 may 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 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.

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 dynamic process is also automatic.

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, but in the unnamed blocks a hierarchical name cannot be used to access it.

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 msl;
   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 variables cannot be used to trigger an event expression or be written with a nonblocking assignment.

See also section 10 on tasks and functions.

5.6 Nets, regs, and logic

A net can only 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. The value can be overridden by a **force** statement. If a net on one side of a port is driven by a variable on the other side, a continuous assignment is implied.

A **reg** variable can only be written by one or more procedural statements, including procedural (quasi-) continuous assignments. The last write determines the value. The **force** statement overrides the **assign** statement which overrides the normal assignments. A **reg** variable cannot be written through a port.

A **logic** variable can be written either by one continuous assignment or primitive output, or by one or more procedural statements. The last write determines the value. A **logic** variable can be written through a port. It shall be an error to have a continuous assignment and a procedural assignment write to the same **logic** variable, even through ports. The **assign** statement overrides normal procedural assignments to a **logic** variable, until deassigned.

Note the difference between a net declaration with assignment and a variable initialization:

```
wire w = vara & varb; // continuous assignment
reg r = consta & constb; // initial assignment
logic v = consta & constb; // initial assignment
```

EC-CH24

5.7 Signal Aliasing

The SystemVerilog **assign** statement is a unidirectional assignment and may 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 wires. The example below implements a byte order swapping between bus A and bus B.

Editor’s Note: I took the liberty of adding “alias” to the keyword list in annex B, under this change number.

```
module byte_swap (inout A, inout B);
   wire [31:0] A,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 W, inout LSB, inout MSB);
   wire [31:0] W;
   wire [7:0] MSB,LSB;
   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 should be the same size and connectivity is independent of the simulation host. The types of nets connected with an alias statement must be compatible, all the nets have to be of the same type or "wire", i.e. it would be 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 is 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]` will share the same wires:

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

module overlap(inout bus16, inout low12, inout high12);
   wire [15:0] bus16;
   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, so 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 a module instance would appear, and any undeclared nets in the alias statement are assumed to be scalar as they would with 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 it’s 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 SystemVerilog interfaces. SystemVerilog also defines a default data type for attributes.

6.2 Attribute syntax for interfaces

```
interface_declaration ::=          // from Annex A.1.3
        { attribute_instance } interface interface_identifier [ parameter_port_list ]
        [ list_of_ports ] ; [unit] [precision] { interface_item }
        endinterface [: interface_identifier]
      | { attribute_instance } interface interface_identifier [ parameter_port_list ]
        [ list_of_port_declarations ] ; [unit] [precision] { non_port_interface_item }
        endinterface [: interface_identifier]
interface_item ::=          // from Annex A.1.6
        port_declaration
      | non_port_interface_item
attribute_instance ::= (* attr_spec { , attr_spec } *)          // from Annex A.9.1
attr_spec ::=
        attr_name = constant_expression
      | attr_name
attr_name ::= identifier
```

Syntax 6-1—Interface attribute syntax (excerpt from Annex A)

An example of defining an attribute for an interface declaration is:

```
(* interface_att = 10 *) interface bus1.... endinterface
```

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.

The **modport** declaration can be preceded by an attribute instance, like any other interface item.

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 incrementor and decrementor operators. SystemVerilog includes the C assignment operators, such as `+=`, and the C incrementor and decrementor 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

```
unary_operator ::=          // from Annex A.8.6
            + | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
binary_operator ::=
            + | - | * | / | % | == | != | === | !== | && | || | **
           | < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | << | >>> | <<<
inc_or_dec_operator ::= ++ | --
unary_module_path_operator ::=
      ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
binary_module_path_operator ::=
      == | != | && | || | & | | | ^ | ^~ | ~^
assignment_operator ::=
            = | += | -= | *= | /= | %= | &= | |= | ^= | <<= | >>= | <<<= | >>>=
```

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

7.3 Assignment, incrementor and decrementor operations

In addition to the simple assignment operator, `=`, SystemVerilog includes the C assignment operators and special bitwise assignment operators: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `<<<=`, and `>>>=`. Assignment operators may only be used with blocking assignments.

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));
```

SystemVerilog also includes the C incrementor and decrementor operators `++i`, `--i`, `i++`, and `i--` (provided there is no timing control). These can be used in expressions without parentheses. These increment and decrement operations behave as blocking assignments.

The behavior of the **++** and **--** operators (pre/post increment/decrement) is incompletely defined in the ANSI C standard. This can lead to unexpected behavior when a single statement modifies the same variable more than once. For example, the following C code fragment may produce different outputs with different C compilers:

```
int i = 1;
printf( "%d %d %d %d %d %d\n", i++, i++, ++i, --i, i--, i-- );
```

SystemVerilog defines the semantics for computing all arguments and operands. The size of the **++** and **--** operators is self-determined. Arguments with the same precedence are evaluated in strict left-to-right order. In addition, the **++** and **--** operators operate on their corresponding variables as they are evaluated. Thus, the semantics of post and pre increment (**++**) is roughly equivalent to the code shown below (decrement is analogous).

```
function integer pre_inc (var integer a); // ++a
   a += 1;
   pre_inc = a;
endfunction

function integer post_inc (var integer a); // a++
   post_inc = a;
   a += 1;
endfunction
```

The above description states a semantic definition for these operators. SystemVerilog's semantics are compatible with Verilog operators, which are also left to right associative, and may have side-effects. For example:

```
$display( f( a ) + g( b ) );
```

EC-CH43 ~~If functions f() and g() have side effects on variables a or b, Verilog must enforce the left-to-right semantics to avoid the ambiguous results.~~

EC-CH43 Verilog enforces left-to-right evaluation in accordance with the associativity to avoid the ambiguous results; functions `f()` and `g()` may have side effects (global or hierarchical reference) on variables `a` or `b`.

BC44-3 The type returned by an assignment operator shall be the type of the LHS. If the LHS is a concatenation, the type returned shall be an unsigned integral value whose bit length is the sum of its operands.

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.

EC-CH82

These operators compare operands bit for bit, and return a 1-bit self-determined result. ~~If the operands are not the same length, the shorter operand is zero-filled.~~ 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 will 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 will 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, some tools may 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

BC19-1

EC-CH25

EC-CH23

BC19-1

EC-CH23

EC-CH25

`() [] .`	left
`Unary ! ~ ++ -- + - & ~&` ~~`&&`~~ `| ~|` ~~`||`~~ `^ ~^ ^~`	right
`**`	left
`* / %`	left
`+ -`	left
`<< >> <<< >>>`	left
`< <= > >= inside dist`	left
`== != === !== =?= !?=`	left
`&` ~~`&~`~~	left
`^ ~^ ^~`	left
`|` ~~`|~`~~	left
`&&`	left
`||`	left
`?:`	right
`=>`	right
`= += -= *= /= %= &= ^= |= <<= >>= <<<= >>>=`	none
`{,}`	concatenation

Editor's Note: BC19-1 said to add ^~ to lines 2 and 11. I made the second change to line 10 was instead of 11.

Note that `&` is higher precedence than `^`, following the Verilog standard.

EC-CH33

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.

EC-CH87

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.

EC-CH87

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.

EC-CH87

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.

EC-CH74

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** (or **logic** if any operand is of type **logic**). 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 may 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 **string** type (as described in section 3.8). 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.

Section 8 Procedural Statements and Control Flow

8.1 Introduction (informative)

EC-CH79

~~Procedural statements are introduced by one of~~ One introduces procedural statements by the following:

Editor's Note: The deleted sentence should be kept. The new wording is too informal for a technical language reference manual.

initial // do this statement once at the beginning of simulation

Editor's Note: The added comment is not correct. Statements within an initial procedure do not necessarily execute at the beginning of simulation, as there can be time and/or event controls before the statement

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

EC-CH79

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

Editor's Note: I took the liberty of adding "final" to the list of keywords in Annex B.

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 will override that assign; however, once the force is released, the assign's effect again will 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.

built-in

```
statement ::= [ block_identifier : ] statement_item          // from Annex A.6.4
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 } transition_to_state statement_or_null
        | { attribute_instance } inc_or_dec_expression
        | { attribute_instance } function_call       /* must be void function */
        | { 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 } process_statement~~
        | { attribute_instance } proc_assertion
statement_or_null ::=
          statement
        | { attribute_instance } ;
procedural_timing_control_statement ::=
          delay_or_event_control  statement_or_null
```

EC-CH88

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

8.2 Blocking and nonblocking assignments

```
blocking_assignment ::=          // from Annex A.6.2
          variable_lvalue = delay_or_event_control expression
        | 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, and information may be lost, a warning can be given.

8.3 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]

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

unique_priority ::= **unique** | **priority**

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 warning 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 will cause a warning

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 warning
```

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.

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 a warning message if an unexpected case value is found. 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 will cause a run-time warning
      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 may also be used to determine synthesis behavior.

8.4 Loop statements

```
loop_statement ::=          // from Annex A.6.8
        forever statement
      | repeat ( expression ) statement_or_null
      | while ( expression ) statement_or_null
      | for ( variable_decl_or_assignment ; expression ; variable_assignment ) statement_or_null
      | do statement 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)

Editor’s Note: A new BNF for the **`for`** loop will be required (see changes described below).

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

8.4.1 The do...while loop

Editor’s Note: Subheading titles were added for clarity, due to additional text on for loops..

```
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.

8.4.2 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 can be referenced hierarchically by adding a statement label before the **for** loop (see section 8.7).

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 );
```

8.5 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 **`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.

EC-CH79

8.6 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
```

Editor's Note: insert final BNF once the BNF is complete.

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.

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 ~~similar routines~~ `vpi_control(vpiFinish,...)`

Editor's Note: I took the liberty of replacing "similar routines" with the actual PLI routine name.

```
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 will cause the simulation to end immediately. Final blocks can only trigger once in a simulation.

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

8.7 Named blocks and statement labels

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

EC-CH89

Syntax 8-6—Blocks and labels syntax (excerpt from Annex A)

EC-CH89

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
```

EC-CH89

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 beginning.

```
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. A statement label does not create a hierarchy scope. The label name is specified before the statement, followed by a colon.

```
labelA: statement
```

EC-CH89

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. This is not the same as a block name, however, because it does not create a hierarchy scope.

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

EC-CH89

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.

EC-CH88

~~8.7 Processes~~

~~Each **initial** and **always** block is a process. Each branch of a **fork** within such a block is also a process. These are static processes and they can be explicitly named with a statement label as shown above.~~

~~A dynamic process can be created using the **process** keyword. This forks off a statement without waiting for completion.~~

```
~~process statement~~
```

~~See Section 9 for more information about processes.~~

8.8 Disable

SystemVerilog has **break** and **continue** for a clean way 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. ~~The **disable**, **break** and **continue** statements shall not affect any nonblocking assignments which have been started.~~

BC44-9

BC44-9

~~It shall be illegal to disable a function, because the return value would be uncertain. However, a function may disable its calling block.~~

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
```

8.9 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 ::=
          expression [ iff expression ]
        | hierarchical_identifier [ iff expression ]
        | [ edge ] expression [ iff expression ]
        | event_expression or event_expression
        | event_expression , event_expression
edge ::= posedge | negedge ~~| changed~~
```

BC44-15

Syntax 8-7—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) will 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
```

BC42-12

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 ~~`clk`~~ 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 or net is not of type **logic**, then **posedge** and **negedge** refer to transitions from 0 and to 0 respectively. If the variable or net is a packed array or structure, it is zero if all elements are 0.

BC44-15

~~SystemVerilog also allows the **@** event control to explicitly state any change, using the **changed** keyword.~~

```
~~@(myvar)          // triggers on any change to myvar~~

~~@(changed myvar)  // triggers on any change to myvar~~
```

~~The **@(changed** expression) differs from **@(**expression) in that the **changed** keyword explicitly defines that the event control only triggers on a change of the result of the expression. In certain types of expressions, **@(**expression) can trigger on changes to operands of the expression that do not affect the result.~~

SystemVerilog allows assignment expressions to be used in an event control, e.g. `@((a = b + c))`. The event control shall only be sensitive to changes in the result of the expression on the right-hand side of the assignment. It shall not be sensitive to changes on the left-hand side expression.

8.10 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 24.3.

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.

BC42-14

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.

EC-CH89

SystemVerilog has both static processes, introduced by **always**, **initial** or **fork**, and dynamic processes, introduced by ~~**process**~~ 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 may also be considered its own thread. Execution of each thread may be interrupted between statements at a semicolon, but a single statement (not a block) containing no user task or function call is uninterrupted. This allows atomic test-and-set using assignment operators in an if statement.

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 **$wait_child**, **$suspend_thread**, and **$terminate**.

EC-CH79

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 will preserve the ordering between runs. This helps keep the output log file stable since final blocks are mainly used for displaying statistics.

9.2 Level sensitive 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 may 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 may 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 may be inferred.

9.3 Latch sensitive 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 differs from a normal **`always`** procedure in the following ways:

— There is an inferred sensitivity list that includes every variable read by the procedure.

— The variables written on the left-hand side of assignments may 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.

BC42-17 Software tools may perform additional checks to warn if the behavior within an **`always_latch`** procedure does not represent latched logic.

9.4 Edge sensitive 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 may 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, **`logic`** variables, and any other type of variables, except **`reg`** variables. Nets can be driven by multiple continuous assignments, or a mixture of primitives and continuous assignments. **`logic`** variables and other data types 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.

~~9.6 Dynamic processes~~

Editor's Note: The **process** statement is deprecated, in favor of **fork**...**join none** (see below).

~~The SystemVerilog dynamic process adds capability that behaves like a **fork** without a **join**. A dynamic process is started as a separate thread, and execution of the current procedure or task continues while the process is executing. The process does not block the flow of execution of statements within the procedure or task. Dynamic processes allow the creation of multi-threaded processes, as opposed to multiple procedures, which are static parallel processes.~~

~~A dynamic process shall be created by the **process** keyword, which is used as follows:~~

```
~~process statement~~
```

~~For example, the following task initiates an endless loop and returns immediately to the caller. The task can be launched any number of times to display a selected location at every strobe.~~

```
~~task monitorMem(input int address);~~
   ~~process forever @strobe $display("address %h data %h", mem[address] );~~
~~endtask~~
```

~~The following example illustrates using a dynamic process to model a pipeline.~~

```
~~// pipeline module~~
~~module p(input clk, flush, input int x_in, y_in, z_in);~~
   ~~parameter int latency = 6, throughput = 2;~~
   ~~int z_out;~~
   ~~int processes = 0;~~

   ~~always begin~~
      ~~while (!flush) begin~~
      ~~process begin~~
         ~~int v2, v3, v4, v5; // lifetime matches process~~
         ~~processes++;~~
         ~~v2 = x_in + y_in;~~
         ~~v3 = x_in - z_in;~~
         ~~v4 = v2 * v3;~~
         ~~v5 = v4 * x_in;~~
         ~~repeat(latency) @ (posedge clk);~~
         ~~z_out <= v5;~~
         ~~processes--;~~
      ~~end~~
      ~~repeat(throughput) @(posedge clk);~~
   ~~end~~
   ~~wait(processes == 0); //wait for flush~~
   ~~end~~
~~endmodule~~
```

~~In the proceeding example, the **while** loop contains a delay of two clock cycles, from the **repeat** statement, and this determines the pipeline throughput. Each iteration spawns a process which lasts six clock cycles, the latency of the pipeline. The variable `processes` keeps a count of the number of currently active processes. The pipeline flush is not complete until this count has fallen to zero.~~

~~SystemVerilog 3.0 does not provide a mechanism to disable a process once it has been started, but all instances of a named block within a dynamic process can be disabled by referring to a named block.~~

9.7 fork...join

The **fork...join** construct provides the primary mechanism for creating concurrent processes.

The syntax to declare a **fork...join** block is:

EC-CH89

```
fork
   statement1;
   statement2;
   ...
   statementn;
~~join [all | any | none]~~
join | join_any | join_none
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

EC-CH44

~~The statement(s) can be any valid statement or block of statements enclosed by **begin...end**.~~

One or more statements can be specified, each statement will execute as a concurrent process.

EC-CH45

~~The spawned processes start executing in strict source order: the first statement (statement1), starts executing first, followed by the second (statement2), and so on.~~

EC-CH89

In Verilog a **fork...join** block always causes the process executing the fork statement to block until all the forked off processes terminate. SystemVerilog adds ~~join options~~ the **join_any** and **join_none** keywords that ~~control how the fork is to be carried out.~~

~~The join options of **all**, **any** or **none**~~ specify when the parent (forking) process resumes execution. ~~If the join option is not specified, then SystemVerilog defaults to all, which is the same behavior as Verilog.~~

Editor's Note: I implemented change EC-CH89 for the preceding two paragraphs slightly differently than specified to make the wording flow better.

Table 9-1: fork...join control options

EC-CH89

Option	Description
~~all~~ join	The parent process blocks until all the processes spawned by this fork complete. This is the same as a Verilog **fork...join** .
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.

EC-CH45

~~A **fork...join** none statement causes all the spawned processes as well as the parent process to execute concurrently, but the children processes do not start executing until the parent process executes a blocking statement (see **$suspend_thread** in section 9.9.3). Nevertheless, the spawned processes will start executing in source order: starting with the first statement first, and ending with the last.~~

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
```

EC-CH89

In the following example, two processes are forked off, the first one will wait for 20ns and the second will wait for the named event eventA to be triggered. Because ~~no join option~~ the **join** keyword is specified~~ (the same as **all**)~~, the parent process will block until the two processes complete, that is, 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:

```
    function int wait_20;
       fork
          # 20;
          return 4;   // Illegal: cannot return; function lives in another process
       join_none
    endfunction
```

EC-CH89

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 or the ~~more natural~~ **fork**...**join_none** form.

9.8 Process execution threads

SystemVerilog creates a thread of execution for:

— Each **initial** block

— Each **always** block

EC-CH89

— Each parallel statement in a **fork**...**join** (or **join_any** or **join_none**) statement group

— Each dynamic process

Each continuous assignment may also be considered its own thread.

Execution of each thread can be interrupted between statements at a semicolon, but a single statement (not a block) containing no user task or function call shall not be interrupted. This allows atomic test-and-set using assignment operators in an **if** statement.

9.9 Process control

SystemVerilog provides several constructs that allow one process to terminate or wait for the completion of other processes. The **$wait_child** construct waits for the completion of processes. The **$terminate** construct stops the execution of processes. The **$suspend_thread** system task temporarily suspends a thread.

EC-CH100

9.9.1 ~~$wait_child()~~ wait fork

The ~~**$wait_child** system task~~ **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_child~~ wait fork is:

```
~~task $wait_child();~~
wait fork ;
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

~~Calling $wait_child~~ Specifying wait fork causes the calling process to block until all its sub-processes have completed.

EC-CH100 ~~By default, SystemVerilog terminates a simulation run when all its programs finish executing (i.e, they reach the end of their execute block), regardless of the status of any child processes. The $wait_child task allows a program to wait for the completion of all its concurrent threads before exiting.~~

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 15.9.1). 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 ~~call to $wait_child~~ wait fork statement will 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                EC-CH89
   fork
      exec3();
      exec4();
   join_none               EC-CH89
   ~~$wait_child()~~ wait fork;// block until exec1 ... exec4 complete     EC-CH100
endtask
```

9.9.2 ~~$terminate~~ Disable fork

EC-CH100

The ~~$terminate~~ disable fork statement terminates all active descendants (sub-processes) of the calling process.

The syntax for ~~$terminate~~ disable fork is:

```
~~$terminate~~ disable fork;
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

The ~~$terminate command~~ disable fork statement terminates all descendants of the calling process, as well as the descendants of the process' descendants, that is, if any of the child processes have descendants of their own, the ~~$terminate command~~ disable fork statement will terminate them as well.

In the example below the function get_first spawns three versions of a function that will wait for a particular device (1, 7, or 13). The function wait_device function 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 will 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
      ~~$terminate~~ disable fork;
    endfunction
```

EC-CH89

Verilog supports the **disable** construct, which will end a process when applied to the named block being executed by the process. ~~$terminate~~ The **disable fork** statement differs from **disable** in that ~~$terminate~~ **disable fork** considers the dynamic parent-child relationship of the processes, whereas **disable** uses the static syntactical information of the disabled block. Thus, **disable** will end all processes executing a particular block, whether the processes were forked by the calling thread or not, while ~~$terminate~~ **disable fork** will end only those processes that were spawned by the calling thread.

Editor's Note: EC-CH100 did not include the changes to the preceding paragraph. I added those because they seemed appropriate..

9.9.3 $suspend_thread()

The **$suspend_thread** system task temporarily suspends the current thread.

The syntax for $suspend_thread is:

```
    task $suspend_thread();
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

The **$suspend_thread** system task temporarily suspends the current process allowing other ready processes to execute. Calling **$suspend_thread** is conceptually similar to a zero delay statement (#0), however, **$suspend_thread** conveys the intent more clearly ~~and may also be called after nonblocking assignments (see section 15.7) where a zero delay is ill-advised~~ when called during the verification phase.

EC-CH100

The following example forks multiple threads each calling `my_task()`. After each thread is forked, the calling thread is suspended, which allows the newly forked thread to start executing (call `my_task`) before forking the next thread.

```
    for( int j=0; j<10; j++ )
    begin
      fork
        my_task(i);
      join_none
      $suspend_thread();
    end
```

EC-CH89

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

EC-CH103

— Passing arguments by reference instead of by value

— Passing argument values by name instead of by position

— Default argument values

10.2 Tasks

```
task_declaration ::=          // from Annex A.2.7
          task [ automatic ] [ interface_identifier . ] task_identifier ;
          { task_item_declaration }
          { statement }
          endtask [ : task_identifier ]
        | task [ automatic ] [ interface_identifier . ] task_identifier ( task_port_list ) ;
          { block_item_declaration }
          { statement }
          endtask [ : task_identifier ]
task_item_declaration ::=
          block_item_declaration
        | { attribute_instance } input_declaration ;
        | { attribute_instance } output_declaration ;
        | { attribute_instance } inout_declaration ;
task_port_list ::= task_port_item { , task_port_item }
task_port_item ::=
          { attribute_instance } input_declaration
        | { attribute_instance } output_declaration
        | { attribute_instance } inout_declaration
task_prototype ::=
     task ( { attribute_instance } task_proto_formal
          { , { attribute_instance } task_proto_formal } )
named_task_proto ::= task task_identifier ( task_proto_formal { , task_proto_formal } )
task_proto_formal ::=
          input data_type [ variable_declaration_identifier ]
        | inout data_type [ variable_declaration_identifier ]
        | output data_type [ variable_declaration_identifier ]
```

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
```

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
```

BC no #

Each formal argument also 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.~~ The task argument default type in SystemVerilog is reg.

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_declaration ::= // from Annex A.2.6
  function [**automatic**] [signing] [range_or_type]
  [interface_identifier **.**] function_identifier **;**
  { function_item_declaration }
  { function_statement }
  endfunction [**:** function_identifier]
 | **function** [**automatic**] [signing] [range_or_type]
  [interface_identifier **.**] function_identifier **(** function_port_list **) ;**
  { block_item_declaration }
  { function_statement }
  endfunction [**:** function_identifier]

function_item_declaration ::=
  block_item_declaration
 | { attribute_instance } input_declaration **;**
 | { attribute_instance } output_declaration **;**
 | { attribute_instance } inout_declaration **;**

function_port_item ::=
  { attribute_instance } input_declaration
 | { attribute_instance } output_declaration
 | { attribute_instance } inout_declaration

function_port_list ::= function_port_item { **,** function_port_item }

function_prototype ::= **function** data_type **(** list_of_function_proto_formals **)**

named_function_proto::= **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 ::=
  input data_type [variable_declaration_identifier]
 | **inout** data_type [variable_declaration_identifier]
 | **output** data_type [variable_declaration_identifier]
 | variable_declaration_identifier

range_or_type ::=
  { packed_dimension } range
 | data_type

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 formal arguments to be inputs or outputs. Function arguments are all passed by value, i.e. copied.

```
input   // copy value in at beginning

output  // copy value out at end

inout   // copy in at beginning and out at end
```

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

Editor’s Note: The subsection title was added by the editor, both for clarity and to give balance with the addition of subsection 10.3.2 that was added.for draft 1.

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

EC-CH103 EC-CH90 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~~s~~ in a ~~compilation error~~ warning message. SystemVerilog allows using the **void** data type to discard a function's return value. This can be done by casting the function to the **void** type, or by assigning the function return to the **void** type:

```
void'(some_function());
```

EC-CH90

```
~~void = some_function();~~
```

~~With SystemVerilog, a non-void function call can also be used as a statement without explicitly discarding the return, but this can result in a warning message.~~

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 module attribute to specify the default lifetime of all tasks and functions declared within the module. The lifetime attribute can be set to **automatic** or **static**. The default is **static** for modules, and **automatic** for the program block (see section 15).

Class methods are by default **automatic**, regardless of the lifetime attribute of the module in which they are declared. Classes are discussed in section 11.

Editor's Note: No syntax or examples of these new attributes was provided by the SV-EC.

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 EC-CH103 ~~a~~ default values, allowing the call to the task or function to not pass arguments.

10.5.1 Pass by value

EC-CH103 *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 may 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 will copy 1000 bytes each time the call is made.

EC-CH103

```
function int crc( char [1000:1] packet );
   for( int j= ~~0~~ 1; j < ~~1094~~ 1000; j++ ) begin
```

```
         crc ^= packet[j];
      end
   endfunction
```

10.5.2 Pass by reference

EC-CH92

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 ~~indirectly~~ via the reference. To indicate argument passing by reference, the argument declaration is preceded by the **var** keyword. The general syntax is:

```
      subroutine( var type argument );
```

For example, the example above can be written as:

EC-CH103

```
   function int crc( var char [1000:1] packet );
      for( int j= ~~0~~ 1; j < ~~1094~~ 1000; j++ ) begin
         crc ^= packet[j];
      end
   endfunction
```

EC-CH103
EC-CH92

Note that in the example, no change other than addition of the **var** keyword is needed. The compiler knows that packet is now addressed ~~indirectly~~ 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:

```
   char packet[1000:1];
   int k = crc( packet1 ); // pass by value or by reference: call is the same
```

EC-CH103

When the argument is passed by reference, both the caller and the ~~callee~~ subroutine share the same representation of the argument, so any changes made to the argument either within the caller or the ~~callee~~ subroutine will be visible to each other.

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 parameter passing qualifier, different from **input, output**, or **inout.** Combining **var** with any other qualifier is illegal. For example, the following declaration results in a compiler error:

```
   task incr( var input int a ); // incorrect: var cannot be qualified
```

10.5.3 Default argument values

EC-CH91

To handle common cases or allow for unused arguments, SystemVerilog allows a subroutine declaration to specify a default value for each ~~scalar (non-packed-array)~~ singular argument.

The syntax to declare a default argument in a subroutine is:

```
   subroutine( type argument = default_value );
```

EC-CH103

The *default_value* is any expression that is visible at the current scope. It may include any combination of constants or variables visible at the scope of both the caller and the ~~callee~~ subroutine.

When the subroutine is called, arguments with default values can be omitted from the call and the compiler will 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;
```

EC-CH103

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 parameter 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.

EC-CH74

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.

Section 11
Classes

Editor’s Note: This entire section is new for draft 1. Only the Section titles have been highlighted as new text.

Editor’s Note: The material inserted here is as it was provided by the SV-EC committee. The editor feels it is written in a tutorial style, and needs considerable rewording to be appropriate for the SystemVerilog LRM. See section 11.4 as one example where re-wording may be needed.

11.1 Introduction (informative)

EC-CH104

~~SystemVerilog 3.0 includes structures for data encapsulation.~~ SystemVerilog ~~3.1 adds~~ introduces the object-oriented **class** framework. Classes allow objects to be dynamically created and deleted, to be assigned, and to be accessed via handles, which provide a safe pointer-like mechanism to the language. With inheritance and abstract classes, this framework brings the advantages of C function pointers with none of the type-safety problems, thus, bringing true polymorphism into Verilog.

11.2 Syntax

Editor’s Note: Add BNF excerpt, when available

11.3 Overview

EC-CH104

~~A **class** is a collection of data and a set of subroutines that operate on that data.~~ A **class** is a type that includes data and subroutines 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 one can do 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 one can capture in a definition.

```
class Packet ;
   bit [3:0] command;   // data portion
   bit [40:0] address;
   bit [4:0] master_id;
   integer time_requested;
   integer time_issued;
   integer status;

   function new();   // initialization
      command = IDLE;
      address = 41'b0;
```

```
        master_id = 5'bx;
     endfunction

     task clean();
        command = 0; address = 0; master_id = 5'bx;
     endtask
                    // public access entry points
     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)

EC-CH106

The ~~last~~ previous section only provided the definition of a class *Packet*. That is a new, complex data type, but one can't do anything with the class itself. First, one needs to create an instance of the class, a single `Packet` *object*. The first step is to create a variable that can hold an object handle:

```
  Packet p;
```

Nothing has been created yet. The declaration of *p* is simply a variable that can hold a handle of a *Packet* object. For *p* to refer to something, an instance of the class must be created using the **new** function.

```
  Packet p;
  p = new;
```

Editor's Note: The Editor vehemently objects to reserving the keywords **new**, **this** and **super**! (see Annex B)

Uninitialized object handles are set by default to the special value **null.** One can detect an uninitialized object 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
```

EC-CH104

Accessing non-static members or virtual methods 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.

System Verilog objects are referenced using an “object handle”. There are some differences between a C pointer and a System Verilog object handle. C pointers give programmers a lot of latitude in how a pointer may be used. The rules governing the usage of System Verilog object handles are much more restrictive. A C pointer may be incremented for example, but a System Verilog object handle may not. In addition to object handles, section 3.7 introduces the **handle** data type for use with the DirectC interface.

Editor’s Note: Is the “DirectC” .name to be used in SystemVerilog?

Table 11-2: Comparison of pointer and handle types

C pointer	SV object handle	SV handle	Operation
Allowed	Not allowed	Not allowed	Arithmetic operations (such as incrementing)
Allowed	Not allowed	Not allowed	For arbitrary data types
Error	Not allowed	Not allowed	Dereference when null
Allowed	Limited	Not allowed	Casting
Allowed	Not allowed	Not allowed	Assignment to an address of a data type
No	Yes	Yes	Unreferenced objects are garbage collected
Undefined	null	null	Default value
(C++)	Allowed	Not allowed	For classes

11.5 Object properties

After having created an object in the last section, one can use its data fields by qualifying property names with an instance name. Looking at the earlier example, the commands for the Packet object *p* can be used as follows:

```
Packet p = new;
p.command = INIT;
p.address = $random;
time = p.time_requested;
```

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 we did not say:

```
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 may be done. The new task is also called the class constructor.

The **new** operation is defined as a **function** with no return type, thus, it must be nonblocking. 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 11.13) 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:

EC-CH104

```
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 procedural subroutine calls, including the use of default arguments.

11.8 Class properties

So far, we 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 one needs only one version of a variable 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 = $open( "data", "r" );
```

Now, semId will be created and initialized once. Thereafter, every Packet object can access the file descriptor in the usual way:

EC-CH104

```
Packet p;
c = $fgetc( ~~p.semId~~ p.fileID );
```

11.9 This

There are times when one needs to unambiguously refer to properties or methods of the current instance. For example, the following declaration is a common way to write an initialization task:

EC-CH104

```
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 will be resolved by looking at the innermost scope, in this case the subroutine argument declaration. To access the instance property, we qualify it with **this** to refer to the current instance.

Note that in writing methods, one can always qualify members with **this** to refer to the current instance, but it is usually unnecessary.

11.10 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 will not contain an actual handle, until an instance of type Packet is created:

```
p1 = new;
```

Thus, if one declares another variable and assign the old handle, p1, to the new one:

```
Packet p2;
p2 = p1;
```

then there's still only one object, which can be referred to with either the name p1 or p2. Note, **new** was executed once, so only one object has been created.

EC-CH104

If, however, the ~~last expression~~ example above is re-written ~~slightly differently~~ as shown below, it will make a copy of p1:

```
Packet p1;
Packet p2;
p1 = new;
p2 = new p1;
```

~~This statement has **new** executing twice, thus creating two objects, p1 and p2. With this syntax, however, p2 will be a copy of p1, but it will be what is~~ The last statement has new executing a second time, thus creating a new object p2 whose properties are copied from p1, 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
```

EC-CH104

```
~~task~~ integer function 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, 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)
~~endtask~~ endfunction
```

Editor’s Note: Verilog syntax is “function integer”. Is the “integer function” above correct?

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. Thus, we might have

```
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.

11.11 Inheritance and subclasses

The previous sections defined a class called `Packet`. Assume one wanted to extend this class 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, one needs to reference the variable packet_c.

```
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, we could have:

```
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.

One can also override the parent’s methods, changing their definitions.

The mechanism provided by SystemVerilog is called *Single-Inheritance*, that is, each class is derived from a single parent class.

11.12 Overridden members

Subclass objects are also legal representative objects of their parent classes. For example, every `LinkedPacket` object is a perfectly legal `Packet` object.

One can assign the handle of a `LinkedPacket` object 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, when one references these overridden members through `p` one gets 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
```

EC-CH106

~~To get the overridden method~~ To call the overridden method via a parent class object (p in the example), the ~~parent~~ method needs to be declared **`virtual`** (see section 11.18).

11.13 Super

EC-CH106

The **`super`** keyword is used from within a derived class to refer to properties of the parent class. It is necessary to use **`super`** ~~when the property of the derived class has been overridden and cannot be accessed directly~~ to access properties of a parent class when those properties 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 property may be a member declared a level up or a member inherited by the class one level up. There is no way to reach higher (for example, **super.super**.count is not allowed).

EC-CH106 Subclasses are classes that are extensions of the current class. Whereas ~~super-classes~~ superclasses are classes that the *current* class is extended from, beginning with the original base class.

EC-CH106 Note: When using the **super** within **new**, **super.new** must be the first executable statement in the constructor. This is because the ~~super-class~~ superclass must be initialized before the current class and if the user code doesn't provide an initialization, the compiler will insert a call to **super.new** automatically.

11.14 Casting

EC-CH104 EC-CH106 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 ~~super-class~~ superclass variable to a variable of one of its subclasses. However, it may be legal to place the contents of the superclass handle in a subclass variable.

EC-CH106 To check if the assignment is legal, the dynamic cast function **$cast()** is ~~needed~~ used (see section 3.15).

The syntax for **$cast()** is:

EC-CH104

```
   task $cast( ~~scalar~~ singular dest_handle, ~~scalar~~ singular source_handle );
```

or

```
   function int $cast( ~~scalar~~ singular dest_handle, ~~scalar~~ singular source_handle
   );
```

EC-CH106 EC-CH104 ~~This function checks the hierarchy tree (super and subclasses) of the *source_handle* to see if it contains the class *dest_handle*. If it does, **$cast()** does the assignment; if it is not, **$cast()** generates a fatal error runtime error occurs and leaves the destination variable unchanged.~~

~~The second version of this function allows checking the results without generating an error:~~

```
   ~~int success = $cast(destination_handle, source_handle );~~
```

EC-CH104 ~~This version does the assignment and returns 1 if the assignment is valid. Otherwise, it sets the destination handle to null and returns 0 no runtime error occurs, the destination variable is left unchanged, and the function returns 0.~~

EC-CH106 When used with object handles, **$cast()** checks the hierarchy tree (super and subclasses) of the *source_expr* to see if it contains the class *dest_var*. If it does, **$cast()** does the assignment. Otherwise the error handling is as described in section 3.14.

11.15 Chaining constructors

When a subclass is instantiated, one of the system's first actions is to invoke the class method **new()**. The first,

EC-CH106

implicit action **new()** takes is to invoke the **new()** method of its ~~super-class~~ superclass, and so on up the inheritance hierarchy. Thus, all the constructors are called, in the proper order, beginning with the base class and ending with the current class.

EC-CH106

If the initialization method of the ~~super-class~~ superclass requires arguments, one has two choices. To always supply the same arguments or to use the **super** keywords. 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 will pass 5 to the **new** routine associated with Packet.

EC-CH106

A more general approach is to use the **super** keyword, to call the ~~super-class~~ superclass constructor:

```
function new();
   super.new(5);
endfunction
```

To use this approach, **super.new**(...) must be the first executable statement in the function **new**.

11.16 Data hiding and encapsulation

EC-CH104

So far, all class properties and methods have been made available to the outside world without restriction. ~~However, for most data (and subroutines) one wants to hide them from the outside world~~ 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, unlabeled 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 even to subclasses and cannot be inherited. 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, one can reference a *local* method or property of the class, 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 will be compared to other.i and the result of the logical comparison will be returned.

EC-CH104

~~In summary:~~

~~— Wherever possible, use **local** members. Hide members that the outside world doesn't need to know about.~~

- ~~Use **protected** members if the outside world doesn't have a need to know, but subclasses might.~~
- ~~Public access should only be allowed when it is absolutely necessary, and the access should be limited as much as possible. Generally, don't provide direct access to properties but rather provide access methods — provide, for example, only read access if a variable should never be written. This provides an extra level of protection and preserves flexibility for future changes.~~

EC-CH16

11.17 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.

11.18 Abstract classes and virtual methods

Often one creates a set of classes that can be viewed as all derived from a common base class. For example, we might start with a common base class of type `BasePacket` that sets out the structure of packets but is incomplete; one would never want to instantiate it. From this base class, though, one might derive a number of useful subclasses: 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.

EC-CH104

The first step is to create the base class that sets out the prototype for these subclasses. Since the base class ~~doesn't need to instantiate the base class~~ is not intended to be instantiated, it can be made *abstract* by specifying , it can be declared to be *abstract* by declaring the class to be **virtual**:

```
virtual class BasePacket;
```

By themselves, abstract classes are not tremendously interesting, but 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 will 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
```

EC-CH104

EtherPacket is now a class that can be instantiated. In general, if an abstract class has ~~several~~ any virtual methods, all of the methods must be overridden (and provided with a method body) for the subclass to be instantiated. ~~If all of the methods are not overridden,~~ If any virtual methods have no implementation, the subclass needs to be abstract.

EC-CH106

An abstract class may 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. ~~However, if the subclass overrides the virtual method, then the new method must exactly match the super-class's prototype.~~

EC-CH104

11.19 Polymorphism: dynamic method lookup

EC-CH106

Polymorphism allows one to use ~~super-class~~ superclass to hold subclass objects, and to reference the methods of those subclasses directly from the ~~super-class~~ superclass variable. As an example, consider the base class for the `Packet` objects, `BasePacket`. Assuming that it 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, one can create instances of various packet objects, and put these into the array just created:

```
EtherPacket ep = new;
TokenPacket tp = new;
GPSSPacket gp = new;
packets[0] = ep;
packets[1] = tp;
packets[2] = gp;
```

If the data types were, for example, integers, bits and strings, one couldn't store all of these types into a single array, but with *polymorphism* one can do it. In this example, since the methods were declared as **`virtual`**, one can access the appropriate subclass methods from the superclass variable even though the compiler didn't know—at compile time—what was going to be loaded into. For example, `packets[1]`:

```
packets[1].send();
```

will 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.

11.20 Out of block declarations

EC-CH104

~~It is generally good coding practice to keep the class declaration to about a page. This makes the class easy to understand and to remember; declarations that go on for pages are hard to follow, and it is easy to miss short methods buried among the multi-page declarations.~~

~~To make this practical, it is best to move long~~ 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 *attributes* (**`local`**, **`protected`**, **`public`**, or **`virtual`**), and the full argument specification plus the **`extern`** qualifier. The **`extern`** qualifier indicates that the body of the method (it's implementation) is to be found outside the declaration. Then, outside the class declaration, declare the full method—like the prototype but without the attributes—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
```

EC-CH104

~~The first lines of each part of the method declaration are nearly identical, except for the attributes and class-reference fields.~~ 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 (**`::`**).

11.21 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( var 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
```

11.22 Typedef class

Sometimes a class variable needs to be declared before the class itself has been declared. For example, two classes may 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 will work the same way.

11.23 Classes, structures, and unions

EC-CH104

~~SystemVerilog 3.0 includes **struct** and **union**.~~ SystemVerilog ~~3.1~~ 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 ~~3.1~~ 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 ~~3.1~~ objects are strictly strongly-typed. Copying an object of one type onto an object of another is not allowed.

3) SystemVerilog ~~3.1~~ objects are implemented using handles, thereby providing C-like pointer functionality. But, SystemVerilog ~~3.1~~ disallows casting handles onto other data types, thus, unlike C, SystemVerilog ~~3.1~~ handles are guaranteed to be safe.

4) SystemVerilog ~~3.1~~ objects form the basis of an Object-Oriented framework 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.

11.24 Memory management

EC-CH104

Memory for objects, strings, and dynamic and associative arrays is allocated dynamically. When objects are created, SystemVerilog allocates more memory. When an object is ~~not~~ no longer needed ~~anymore~~, SystemVerilog automatically reclaims the memory, making it available for re-use. The automatic memory management system is an integral part of SystemVerilog. One might be tempted to think that a manual memory management system, such as the one provided by C's `malloc` and `free`, might be sufficient. However, SystemVerilog's multi-threaded, re-entrant environment create many opportunities for users to *shoot themselves in the foot*. For example, consider the following example:

EC-CH89

```
myClass obj = new;
fork
   task1( obj );
   task2( obj );
join_none
```

In this example, the main process (the one that forks off the two tasks) doesn't 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 will cause the system to quickly run out of memory. The second option places a large burden on users, who, in addition to managing their test-bench, must also manage the memory using less than ideal schemes. To avoid these shortcomings, SystemVerilog manages all dynamic memory automatically. Users no longer need to worry about dangling references, premature deallocation, or memory leaks. The system will 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
Inter-Process Synchronization and Communication

Editor's Note: This entire section is new for draft 1. Only the Section titles have been highlighted as new text.

12.1 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 test-bench. 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 test-bench. 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.

EC-CH107

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** ~~primitive~~ built-in class, which can be used for synchronization and mutual exclusion to shared resources, and a **mailbox** ~~primitive~~ 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. Lastly, SystemVerilog adds the **wait_var** mechanism that can be used to synchronize processes using dynamic data.

EC-CH107

Semaphores and mailboxes are built-in types, nonetheless, they are classes, and can be used as base classes for deriving additional higher level classes.

Editor's Note: Are semaphore and mailbox really "primitives" in the Verilog HDL and PLI sense of "primitive"?

12.2 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.

Semaphore is a built-in class that provides the following methods:

— Create a semaphore with a specified number of keys: `new()`

EC-CH107

— Obtain ~~a key~~ one or more keys from the bucket: `get()`

— Return ~~a key~~ one or more keys into the bucket: `put()`

— Try to obtain a key without blocking: `try_get()`

12.2.1 new()

Semaphores are created with the `new()` method.

The syntax for semaphore `new()` is:

```
function new(int key_count = 0 );
```

Editor’s Note: Replace preceding syntax line with BNF excerpt, once available.

The *key_count* specifies the number of keys initially allocated to the semaphore *bucket*. The number of keys in the bucket can increase beyond *key_count* when more keys are put into the semaphore than are removed. The default value for *key_count* is 0.

The **`new()`** function returns the semaphore handle, or **`null`** if the semaphore cannot be created.

12.2.2 put()

The semaphore **`put()`** method is used to return keys to a semaphore.

The syntax for **put()** is:

```
task put(int keyCount = 1);
```

Editor’s Note: Replace preceding syntax line with BNF excerpt, once available.

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 will execute if enough keys have been returned.

12.2.3 get()

EC-CH107 The semaphore **`get()`** ~~function~~ method is used to procure a specified number of keys from a semaphore.

The syntax for **`get()`** is:

```
task get(int keyCount = 1);
```

Editor’s Note: Replace preceding syntax line with BNF excerpt, once available.

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 task 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).

12.3 try_get()

The semaphore **`try_get()`** method is used to procure a specified number of keys from a semaphore, but without blocking.

The syntax for **try_get()** is:

```
function int try_get(int keyCount = 1);
```

Editor’s Note: Replace preceding syntax line with BNF excerpt, once available.

keyCount specifies the required number of keys to obtain from the semaphore. The default is 1.

EC-CH107 If the specified number of keys are available, the ~~task~~ method returns 1 and execution continues. If the specified number of key are not available, the ~~function~~ method returns 0.

12.4 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 will be suspended until enough room becomes available in the mailbox queue. Unbounded mailboxes never suspend a thread in a send operation.

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()**

EC-CH107 — Retrieve the number of messages in the mailbox: **num()**

12.4.1 new()

Mailboxes are created with the **new()** method.

The syntax for mailbox **new()** is:

```
function new(int bound = 0);
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

EC-CH107 The **new()** function returns the mailbox ~~identifier~~ 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 will never block. If *bound* is non-zero, it represents the size of the mailbox queue.

EC-CH107 The bound must be positive. Negative bounds are illegal and may result in indeterminate behavior, but implementations can issue a warning.

12.4.2 num()

The number of messages in a mailbox can be obtained via the **num()** method.

The syntax for **num()** is:

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

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

The **num()** method returns the number of messages currently in the mailbox.

12.4.3 put()

The **put()** method places a message in a mailbox.

The syntax for **put()** is:

EC-CH107

```
    task put(~~scalar~~ singular message);
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

EC-CH107

The *message* is any ~~scalar~~ singular (non-unpacked array) 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 will be suspended until there is enough room in the queue.

12.4.4 try_put()

The **try_put()** method attempts to place a message in a mailbox.

The syntax for **try_put()** is:

EC-CH107

```
    function int try_put(~~scalar~~ singular message);
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

EC-CH107

The *message* is any ~~scalar~~ singular (non-unpacked array) 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.

12.4.5 get()

The **get()** method retrieves a message from a mailbox.

The syntax for **get()** is:

EC-CH107

```
    task get( ~~var scalar~~ ref singular message );
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

EC-CH107

The *message* can be any ~~scalar~~ singular (non-unpacked array) expression, and it must be a valid l-value.

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.

EC-CH107

~~Simple~~ Non-parameterized mailboxes are type-less, that is, a single mailbox can send and receive ~~any type~~ 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.

12.4.6 try_get()

The **try_get()** method attempts to retrieves a message from a mailbox without blocking.

The syntax for **try_get()** is:

EC-CH107

```
    function int try_get( ~~var scalar~~ ref singular message );
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

EC-CH107

The *message* can be any ~~scalar~~ singular (non-unpacked array) expression, and it must be a valid l-value.

EC-CH107

The **try_get()** method tries to retrieve one message from the mailbox. If the mailbox is empty, then the ~~function~~ method returns 0. If there is a type mismatch between the *message* variable and the message in the mailbox, the ~~function~~ 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 ~~function~~ method returns 1.

12.4.7 peek()

The **peek()** method copies a message from a mailbox without removing the message from the queue.

The syntax for **peek()** is:

EC-CH107

```
task peek( ~~var scalar~~ ref singular message );
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

EC-CH107

The *message* can be any ~~scalar~~ singular (non-unpacked array) expression, and it must be a valid l-value.

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()** may 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 will become unblocked.

12.4.8 try_peek()

The **try_peek()** method attempts to copy a message from a mailbox without blocking.

The syntax for **try_peek()** is:

EC-CH107

```
function int try_peek( ~~var scalar~~ ref singular message );
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

EC-CH107

The *message* can be any ~~scalar~~ singular (non-unpacked array) expression, and it must be a valid l-value.

EC-CH107

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 ~~function~~ method returns 0. If there is a type mismatch between the *message* variable and the message in the mailbox, the ~~function~~ 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 ~~function~~ method returns 1.

EC-CH107

~~Mailboxes are a built-in type, nonetheless, they are classes, and can be used as base classes for deriving more higher level classes.~~

Editor's Note: Is the preceding paragraph supposed to be a part of the try_peek() subsection? It seems like it belongs at the end of 19.4, before any of the subsections.

12.5 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 11.21), modules, and interfaces:

```
    mailbox#(type = dynamic_type)
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

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()**.

EC-CH107

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_get** and **get** calls are compatible with the mailbox type so that all type mismatches are caught by the compiler and not at run-time.

EC-CH17

~~12.6 Event~~

~~In Verilog, named events are triggered via the -> operator, and processes can block until an event is triggered via the @ operator. A Verilog **event** is a SystemVerilog **event** that uses a ONE_SHOT trigger. But, a SystemVerilog **event** is much more general than a Verilog **event**. The most salient semantic difference is that Verilog named events do not have a value nor a duration, whereas SystemVerilog events have a value (ON, OFF) and a persistency that can be controlled via the trigger options. Also, SystemVerilog events are handles to synchronization objects, thus, they can be passed as arguments to tasks, and they can be dynamically allocated and reclaimed, whereas named events are static and cannot be passed as arguments. More than a basic data type, SystemVerilog events behave like object handles; they can be assigned to one another, they can be assigned the value **null**; they can be arguments to tasks (but not functions), and they can be dynamically allocated and reclaimed.~~

~~Existing Verilog event operations (@ and ->) are backward compatible and will continue to work the same way, but they will be restricted to named events with static lifetime. The new functionality described below will work with all events, static or dynamic.~~

~~A SystemVerilog **event** provides a handle to a synchronization object, the **$sync()** system task can be used to wait for an event (like @), and the **$trigger()** can be used to trigger the event.~~

~~12.5.1 $sync()~~

~~The **$sync()** system task is used to either check the persistent status of an event, or to block the caller until one or more events are triggered.~~

~~**$sync()** can be called either a as task or as a function. The syntax to call **$sync()** is:~~

```
    ~~task $sync(ALL | ANY | ORDER, event ev_id1, ..., ev_idN);~~
```

~~or~~

```
~~function int $sync(CHECK, event ev_id1, ..., ev_idN);~~
```

~~Where *ev_id1*, ..., *ev_idN* are the event identifiers on which **$sync** is to operate.~~

~~The first argument determines the type of operation that **$sync()** is to perform, as described by the table below.~~

Table 12-1: $sync operations

~~**ALL**~~	~~Suspends the calling process until *all* of the specified events are triggered.~~ ~~For example:~~ `~~$sync(ALL, a, b, c);~~` ~~suspends the current process until the 3 events a, b, and c are triggered.~~
~~**ANY**~~	~~Suspends the calling process until *any* one of the specified events are triggered.~~ ~~For example:~~ `~~$sync(ANY, a, b, c);~~` ~~suspends the current process until either event a, or event b, or event c is triggered.~~
~~**ORDER**~~	~~Suspends the calling process until all of the specified events are triggered (like ALL) but the events must be received in the given order (left to right). If an event is received out of order, the process unblocks and generates a run-time error.~~ ~~When **$sync()** is called, only the first event in the list can be in the ON state. If any other event is ON, it generates a run-time error.~~ ~~For example:~~ `~~$sync(ORDER, a, b, c);~~` ~~suspends the current process until events trigger in the order a -> b -> c.~~

~~**CHECK**~~	~~Called as a function that returns 1 if all the specified events are in the ON state, and 0 otherwise.~~
	~~This call is only meaningful with persistent events; those triggered via the ON or OFF trigger option (see section 12.5.2).~~
	~~For example:~~
	`~~if ( $sync(CHECK, eventA) )~~`
	`~~$display("The event A is ON\n");~~`
	~~The message is only displayed if eventA is in the ON state.~~

~~12.5.2 $trigger()~~

~~The **$trigger()** system task is used to change the triggered state of event variables. This state may be persistent or not, depending on the trigger option. A non-persistent trigger state is not visible, only its effect can be felt. Like the way in which a clock edge triggers a latch but the state of the edge can not be ascertained: `if ( posedge clock )` is illegal.~~

~~The syntax to call **$trigger()** is:~~

```
~~task $trigger( option, event ev_id1, ..., ev_idN );~~
```

```
    ~~option: ONE_SHOT | ONE_BLAST | HAND_SHAKE | ON | OFF~~
```

~~Where *ev_id1*, ..., *ev_idN* are the event identifiers on which **$trigger** is to operate.~~

~~The first argument determines the type of operation that **$trigger()** is to perform, as described by the table below.~~

Table 12-2: $trigger operations

~~**ONE_SHOT**~~	~~Triggers the specified events by turning them **ON** momentarily, causing all processes currently waiting on the specified events to unblock. Subsequent calls to **$sync()** on the specified events will block.~~ ~~In order for this call to **$trigger()** to unblock a **$sync()** call, the call to **$sync()** must execute before the call to **$trigger()**.~~ ~~This trigger option is the same as a Verilog **->** operation, except that **$trigger()** can atomically trigger more than one event.~~
~~**ONE_BLAST**~~	~~Similar to **ONE_SHOT** except that the **ON** state persists until simulation time advances. Thus, a **ONE_BLAST $trigger()** will unblock processes that execute **$sync()** either before or at the same simulation time as **$trigger()**.~~
~~**HAND_SHAKE**~~	~~Unblocks only one process, even if more than one **$sync()** call is blocked waiting on the same event. The first process to have executed the **$sync()** call is unblocked (FIFO ordering).~~ ~~If at least one process is blocked in **$sync()** waiting on the specified event, the **$trigger(HAND_SHAKE)** unblocks one process.~~ ~~If there are no processes blocked no the specified event, the event will store the trigger, keeping track of how many times the event has been triggered using **HAND_SHAKE**. Then, when a process eventually calls **$sync()** on the given event, the trigger is removed from the event (its count is decremented) and the process unblocks immediately.~~
~~**ON**~~	~~Turns the event **ON**. All currently waiting as well as subsequent calls to **$sync()** on the specified event will unblock.~~ ~~The **ON** condition persist until it is explicitly set to **OFF**.~~
~~**OFF**~~	~~Turns the event **OFF**. Subsequent calls to **$sync()** on the specified event will block.~~

EC-CH17

~~12.7 Event variables~~

~~Event variables serve as the link between **$trigger()** and **$sync()**. They are a unique data type with several important properties.~~

~~12.5.3 Disabling events~~

~~If an event variable is assigned the special **null** value, the event is ignored in subsequent calls to **$sync()**. That is, when the **event** is set to **null**, no process can wait for the event again.~~

~~For example:~~

```
    ~~event E1 = null;~~
    ~~$sync(ALL, E1);~~
```

~~The call **$sync** doesn't block because event E1 is no longer blocking.~~

~~12.5.4 Merging Events~~

~~When one event variable is assigned to another, the two become *merged*. Thus, calling **$trigger()** on either variable affects **$sync()** calls waiting on both event variables.~~

~~For example:~~

```
~~event a, b, c;~~
~~a = b;~~
~~$trigger(ON, c);~~
~~$trigger(ON, a);  // also triggers b~~
~~$trigger(ON, b);  // also triggers a~~
~~a = c;~~
~~b = a;~~
~~$trigger(ON, a);  // also triggers b and c~~
~~$trigger(ON, b);  // also triggers a and c~~
~~$trigger(ON, c);  // also triggers a and b~~
```

~~When merging events, the assignment only affects subsequent calls to **$trigger()** and **$sync()**. If a process is blocked waiting for event1 when another event is assigned to event1, the call to **$sync()** will never unblock. For example:~~

```
~~fork~~
   ~~T1: while(1) $sync(ALL, E2);~~
   ~~T2: while(1) $sync(ALL, E1);~~
   ~~T3: begin~~
         ~~E2 = E1;~~
         ~~while(1) $trigger(ON, 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 will 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.~~

EC-CH17

12.6 Event

In Verilog, named events are static objects that can be triggered via the -> operator, and processes can block until an event is triggered via the @ operator. SystemVerilog events support the same basic operations, but enhance Verilog events in several ways. The most salient semantic difference is that Verilog named events do not have a value or duration, whereas SystemVerilog events can have a persistency that lasts throughout the time-step on which they are triggered. Also, SystemVerilog events act as handles to synchronization queues, thus, they can be passed as arguments to tasks, and they can be dynamically allocated and reclaimed. ~~In this respect, SystemVerilog events behave like object handles; they can be assigned to one another, they can be assigned the special value null, they can be arguments to tasks (but not functions), and they can be dynamically allocated and reclaimed.~~

EC-CH107

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 FIFO queue maintained within the synchronization object. Processes can wait for a SystemVerilog event to be triggered either via the @ operator or the wait() construct. Events are always triggered using the -> operator.

SystemVerilog provides for two different types of events: persistent events and non-persistent events. These two are described below.

12.6.1 Non-Persistent Events

EC-CH107

Non-persistent events are the same as named Verilog events. They behave like a one-shot, that is, their triggered state is not observable, only its effect. This is similar to the way in which an edge can trigger a ~~latch~~ flip-flop but the state of the edge can not be ascertained: if(posedge clock) is illegal.

Triggering a non-persistent event causes all processes currently waiting on the event to unblock. For a trigger to unblock a process that is waiting on non-persistent event, that process must execute the wait (or @) before the triggering process executes the trigger operator, ->. A process that executes wait() on a non-persistent event after the event has been triggered will block.

The syntax to declare a non-persistent event is:

```
event event_identifier;
```

Editor’s Note: Add BNF excerpt, once available.

12.6.2 Persistent Events: event bit

Persistent events are similar to non-persistent events except that once triggered, the triggered state persists throughout the time-step, that is, until simulation time advances. Thus, a persistent event will unblock all processes that execute the wait() construct either before or at the same simulation time as the trigger operation.

The persistent trigger behavior helps eliminate a common race condition that occurs when both the trigger and the wait operations happen at the same time. A process that blocks on a regular (non-persistent) event may or may not unblock depending on the execution order of the waiting and triggering processes, while a persistent event always unblock the waiting process, regardless of the order of execution.

The syntax to declare a persistent event is:

```
event bit event_identifier;
```

Editor’s Note: Add BNF excerpt , once available.

Persistent and non-persistent events support the same set of operators, but they are different types. A persistent event may only be assigned (or passed as an argument) to another persistent event and vice-versa (see Section 11.6.2).

12.6.3 Triggering an Event

All events regardless of their type (persistent or non-persistent) are triggered via the -> operator.

The syntax to trigger an event is:

```
-> event_identifier;
```

Editor’s Note: Add BNF excerpt , once available.

If the event_identifier is a persistent event then the event will remain in the triggered state until the simulation time advances. Otherwise, the persistent state is unobservable.

Triggering a persistent event more than once at the same simulation time has no effect. However, triggering a non-persistent event more than once, at the same simulation time, results in multiple triggers.

12.6.4 Waiting for an Event

There are two mechanisms that can be used to wait for an event. The @ operator and the wait construct.

The syntax for this use of the @ operator is:

```
@ event_identifier;
```

Editor’s Note: Add BNF excerpt , once available.

The @ operator always blocks the calling process until an event is triggered.

The syntax for this use of the wait construct is:

```
wait( event_identifier );
```

Editor’s Note: Add BNF excerpt , once available.

The wait construct blocks if the given event is a non-persistent event or the persistent event has not been triggered at the current simulation time.

Both mechanisms can be used to wait for either a persistent or a non-persistent event. The wait construct is only meaningful when the event is persistent.

Examples:

```
event done; // declare a new event
event done_too = done; // declare done_too as alias to done
event bit blast; // persistent event
task trigger( event ev );
   -> ev;
endtask
...
fork
   @ done_too; // wait for done through done_too
   trigger( done ); // trigger done through task trigger
join

event bit blast; // persistent event
fork
   -> blast; // trigger blast event
   wait( blast ); // wait for blast event
join
```

The first fork in the examples 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 will trigger either event. In the example, the first 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.

EC-CH107

When the second fork executes, the first process may triggers the event `blast` before the second process (assuming the processes in a **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 blast is a persistent event so it remains in its triggered state for the duration of the time-step. Note that if blast was declared as a non-persistent event the second process would never unblock.

EC-CH17

12.7 Event synchronization utilities

12.7.1 $wait_all()

The $wait_all system tasks suspends the calling process until all of the specified events are triggered.

The syntax for the $wait_all task is:

```
$wait_all( event_identifier {, event_identifier } )
```

For example:

```
$wait_all( a, b, c);
```

suspends the current process until the 3 events a, b, and c are triggered.

Any of the specified events may be triggered more than once; the only requirement to unblock the calling process is that each event be triggered at least once.

12.7.2 $wait_any()

The $wait_any system tasks suspends the calling process until any of the specified events are triggered.

The syntax for the $wait_any task is:

```
$wait_any( event_identifier {, event_identifier } )
```

For example:

```
$wait_any( a, b, c);
```

suspends the current process until either event a, or event b, or event c is triggered.

12.7.3 $wait_order()

The $wait_order system task suspends the calling process until all of the specified events are triggered (similar to $wait_all), but the events must be triggered in the given order (left to right). If an event is received out of order, the process unblocks and generates a run-time error.

The syntax for the $wait_order task is:

```
$wait_order( event_identifier {, event_identifier } )
```

When $wait_order() is called, only the first event in the list can be in the triggered state. If any other persistent event is in triggered state, it generates a run-time error.

For example:

```
$wait_order( a, b, c);
```

suspends the current process until events trigger in the order a –> b –> c.

EC-CH17

12.8 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.

12.8.1 Disabling Events

EC-CH107

If an event variable is assigned the special **null** value, the event is ignored in subsequent calls to wait(). That is, when the event is set to **null**, no process can wait for the event again.

For example:

```
event E1 = null;
@ E1;
```

The statement @ E1 does not block because event E1 is no longer blocking.

12.8.2 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 merging events, the assignment only affects subsequent executions of ->and wait(). If a process is blocked waiting for event1 when another event is assigned to event1, the wait() will 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 will 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.

12.9 $wait_var()

The **$wait_var()** system task is a procedural blocking statement that waits for *any* of the variables in its argument list to change (the value of the variables must change, assigning the same value to a variable does not cause a change).

The syntax for **$wait_var()** is:

EC-CH107

```
task $wait_var(~~scalar~~ singular variable^1,..., variable^N);
```

Editor’s Note: Replace preceding syntax line with BNF excerpt, once available.

The variables *variable*^{1},..., *variable*^{N} can be any one of the integral data types (see section 3.3.1) or **string.** Each variable may be either a simple variable, or a **var** parameter (variable passed by reference) or a member of an array, associative-array, or object (class) of the aforementioned types. Objects (handles) are not allowed.

EC-CH107

Arguments to **$wait_var()** can be ~~an~~ array subscript expressions, in which case the index expression is evaluated only once when **$wait_var()** is executed. Likewise, passing an object data member to **$wait_var()** will block until that particular data member changes value, not when the handle to the object is modified. For example:

EC-CH89

```
Packer p = new; // Packet 1
Packet q = new; // Packet 2

fork
   $wait_var(p.status);     // Wait for status in Packet 1 to change
   p = q;                   // Has no effect on the wait in Process 1
join_none

// $wait_var continues to wait for status of Packet 1 to change
```

The example below forks two concurrent processes. The first process is suspended until the second element of array `data` changes. The second process randomly changes the values within array `data`. When `data[2]` changes value, the first process prints its message.

```
bit [7:0] data [100];

fork
   begin
      $wait_var(data[2]);
      $display( "Data[2] has changed to: %d\n", data[2]);
   end
   begin
      for( int j = 0; j < 100; j++ )
         begin
            data[i] = $random;
            #10;
         end
   end
join
```

Section 13 Clocking Domains

Editor’s Note: This entire section is new for draft 1. Only the Section titles have been highlighted as new text.

13.1 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 test-bench 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 test-benches 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 test-bench may contain one or more clocking domains, each containing its own clock plus an arbitrary number signals.

EC-CH101

The clocking domain separates the timing and synchronization details from the structural, functional, and procedural elements of a test-bench. 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

13.2 Clocking domain declaration

The syntax for the **clocking** construct is:

EC-CH80

```
clocking_decl ::= [ default ] clocking [identifier] clocking_event ;
   { clocking_item }
endclocking
```

EC-CH80

```
clocking_event ::= @ identifier
                 | @ ( event_expression )
event_expression ::= // this item is already defined in the BNF
```

EC-CH46

```
clocking_item := default default_skew;
| clocking_direction signal_or_assign_list ;

default_skew ::= input skew
               | output skew
               | input skew output skew

clocking_direction ::= input [ skew ]
                     | output [ skew ]
```

```
                             | input [ skew ] output [ skew ]
                             | inout

signal_or_assign_list ::= signal_or_assign { , signal_or_assign }

signal_or_assign ::= signal_identifier [ = hierarchical_expression ]
```

EC-CH80

~~skew ::= [edge] # delay_expression// edge valid only if event_expression is simple edge~~

```
skew ::= edge [ # delay_expression ]    // edge valid only if
       | # delay_expression             // clocking event is simple edge

edge ::= posedge | negedge

delay_expression ::= unsigned_number | time_literal
```

Editor's Note: Update preceding BNF excerpt with new BNF, once available.

EC-CH80

The *delay_expression* must be either a time literal or a constant expression that evaluates to a positive integer value.

The *identifier* specifies the name of the clocking domain being declared.

The *signal_identfier* identifies a port 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 port and the interface signal will share the same name.

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. Bi-directional signals (**inout**) are sampled as well as driven.

The *skew* parameters determine 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 13.3). When the clocking event specifies a simple edge, instead of a number, the skew may be specified as the opposite edge of the signal. A single *skew* may be specified for the entire domain by using a **default** clocking item.

The *hierarchical_name* 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:

EC-CH46

```
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 will use a `10`**ns** input skew and a `2`**ns** 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.

EC-CH81

Unless otherwise specified, the default **`input`** skew is `1`**`step`** and the default **`output`** skew is **`0`**. A **`step`** is a special time unit defined to be the smallest possible delay throughout the simulation, that is, the smallest global precision. Like all other time units, **`step`** is not a keyword. A `1`**`step`** input skew allows input signals to sample their steady-state values immediately before the clock event (i.e., at read-only-synchronize immediately before time advanced to the clock event). Unlike other time units, which represent physical units, a step cannot be used to set or modify the either the precision or the timeunit.

Editor's Note: The addition in the preceding paragraph from EC-CH81 of a step being the "smallest global precision" is ambiguous. What if no global timeprecision was specified? Is the module's precision used? Is the 'timescale precision used? What's the precedence. I suspect what was intended is that **"a step is an increment of 1 simulator time unit. The simulator time unit is defined in the Verilog 1364 standard".** [1364-2001 section 19.8 defines the simulator time as "The smallest time_precision argument of all the `timescale compiler directives in the design determines the precision of the time unit of the simulation.". The PLI sections also refer to "simulator time unit" in several places, and refer to 19.8 for the definition of the simulator time unit".

13.3 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 13-1 shows the basic sample/drive timing for a positive edge clock.

Figure 13-1—Sample and drive times including skew with respect to the positive edge of the clock.

Editor's Note: Figure still needs to be recreated.

EC-CH47

A skew must be a constant expression ~~and can be specified either as an unsigned integer value or as a time literal~~, 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 @(changed clk);
   input #1ps address;
   input #5 output #6 data;
endclocking
```

Editor's Note: Has the changed keyword been removed from SystemVerilog?

EC-CH48

An input skew of 1**step** indicates that the signal is to be sampled ~~an infinitesimal *delta* before the clock event~~ 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.

EC-CH47

~~When skews are not specified, input signals default to a skew of **1step**, and output signals default to a skew of **#0**.~~

EC-CH84

An input skew of **#0** forces a skew of zero. Input~~ signal~~s with zero skew are sampled at the same time as their corresponding ~~clock edge~~ clocking event, but to avoid races, ~~the sampling is done *after* all nonblocking assignments (NBA) have been processed (see Section 15.7)~~they are sampled at the start of the verification phase (after processing non-blocking assignments). Likewise, output~~ signal~~s with zero ~~output~~ skew are driven at the same time as their specified ~~clock edge~~ clocking event, but ~~immediately before *read-only synchronize time* (before advancing time)~~ at the end of the verification phase. A detailed explanation for this event ordering is covered in Section 15.7.

13.4 Hierarchical expressions

Any signal in a clocking domain can be associated with an arbitrary hierarchical expression. As described above, 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, concatenations, or combinations of signals in other scopes or in the current scope.

```
clocking mem @(changed clock);
   input instruction = { opcode, regA, regB[3:1] };
endclocking
```

Editor's Note: Has the changed keyword been removed from SystemVerilog?

13.5 Signals in multiple clocking domains

EC-CH85

~~The same port may be used in more than one clocking domain. For input signals, the semantics are clear; each clocking domain samples the signal using a different clock. However, for output signals, there are two possibilities, the output port is either driven to a resolved value or to the latest value assigned (as a procedural assignment). Typically, this is not an issue since signals in different clocking domains truly are separate signals and each corresponds to a separate port (in a different module or program). But, sometimes the same port signal may be driven by more than one clock edge, for example, dual data rate memories are driven on both positive and negative clock edges. Output signals implement **logic** semantics, that is, the last signal write determines the value. These semantics are typically useful, but users can easily accomplish value resolution by using separate ports for the same net.~~

The same signals—clock, inputs, inouts, or outputs—may appear in more than one clocking domain. Clocking domains that use the same clock (or clocking expression) will 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 13.13, and output semantics are described in section 13.14.

13.6 Clocking domain scope and lifetime

EC-CH86

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
```

EC-CH49

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 ~~a~~ program (see section 15).

EC-CH49

Clocking domains have static lifetime and scope local to their enclosing module, interface or program.

13.7 Multiple clocking domain example

In this example, a simple test module includes two clocking domains. The program construct used in this example is discussed in section 15. In this example, it can be considered a module.

```
program test(  input phi1, input [15:0] data, output write,
               input phi2, inout [8:1] cmd, input enable
            );

   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

   // program begins here
   ...
   // user can access cd1.data , cd2.cmd , etc…
endprogram
```

The test module can be instantiated and connected to a device under test (`cpu` and `mem`).

```
module top;
   logic phi1, phi2;

   test main( phi1, data, write, phi2, cmd, enable );
   cpu cpu1( phi1, data, write );
   mem mem1( phi2, cmd, enable );
endmodule
```

13.8 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 test-bench. Furthermore, since the signal directions in the clocking domain within the test-bench are with respect to the test-bench, and not the design under test, a **modport** declaration can appropriately describe either direction. Conceptually, one can envision a test-bench program as being contained within a *program module*, and whose ports are interfaces that correspond to the signals declared in each clocking domain. The interface's wires will have the same direction as specified in the clocking domain when viewed from the test-bench 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);
      wire [15:0] data;
      wire write;
      modport test (input data, output write);
      modport dut (output data, input write);
endinterface

interface bus_B (input clk);
      wire [8:1] cmd;
      wire 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

      // program begins here
      ...
      // user can access cd1.a.data , cd2.b.cmd , etc…
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, 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`,…).

13.9 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)`.

EC-CH18

~~13.10 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:~~

```
      ~~## expression [ @ clocking_name ] ;~~
```

~~The *expression* can be any SystemVerilog expression that evaluates to a positive integer value.~~

~~The optional *clocking_name* must be the name of a clocking domain. If it is not specified then the default clocking is used (see section 13.11). If neither *clocking_name* nor default clocking has been specified then the compiler will issue an error.~~

~~Example:~~

```
   ~~## 5 @busA;       // wait 5 cycles using clocking busA~~
   ~~## j + 1 @busB    // wait j+1 cycles using clocking busB~~
   ~~## 3;             // wait 3 cycles using the default clocking~~
```

~~13.11 Default ##~~

~~One clocking event can be specified as the default for all cycle delay operations within a given module or program.~~

~~The syntax for the default cycle specification statement is:~~

```
      ~~default ## clocking_name ;~~
```

~~The *clocking_name* must be the name of a clocking domain.~~

~~Only one default clocking can be specified in a program or module. Specifying a default clocking more than~~

~~once in the same program or module will result in a compiler error.~~

~~A default clocking specified in a module is only valid in that particular module and not in any of its sub-modules.~~

```
    ~~program test( input bit clk, input reg [15:0] data )~~

        ~~clocking bus @(posedge clk);~~
            ~~inout data;~~
        ~~endclocking~~

        ~~default ## bus;~~

        ~~## 5;~~
        ~~if ( bus.data == 10 )~~
            ~~## 1;~~
        ~~else~~
            ~~...~~

    ~~endprogram~~
```

EC-CH18

13.10 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:

EC-CH51

```
    ## [ expression ];
```

Editor's Note: Update preceding syntax with BNF excerpt, once available.

The expression can be any SystemVerilog expression that evaluates to a positive integer value.

EC-CH52

What represents a cycle is determined by the default clocking in effect (see section 13.11). If no default clocking has been specified for the current module, interface, or program then the compiler will issue an error.

Example:

EC-CH50

```
    ## [5];     // wait 5 cycles using the default clocking

    ## [j + 1]; // wait j+1 cycles using the default clocking
```

13.11 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:

```
    default clocking_decl ;          // clocking declaration
```

or

```
    default clocking clocking_name ; // existing clocking
```

Editor's Note: Update preceding syntax with BNF excerpt, once available.

The clocking_name 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 will result in a compiler error.

EC-CH53 ~~A default clocking specified in a module is only valid in that particular module and not in any of its sub-modules.~~ 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 other 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
   ## [5];
   if ( bus.data == 10 )
      ## [1];
   else
      ...
endprogram
```

EC-CH54

Example 2. Assigning an existing clocking to be the default:

```
clocking busA @(posedge clk1); ... endclocking
clocking busB @(negedge clk2); ... endclocking
module processor ...
   module cpu( interface y )
      default clocking busA ;
      initial begin
         ## [5]; // use busA => (posedge clk1)
         ...
      end
   ~~endprogram~~ endmodule
endmodule
```

EC-CH55 EC-CH54

EC-CH101 Editor’s Note: The remaining subsections in.this section were originally in section 14 of 3.1 draft 2

13.12 ~~Synchronization~~ Synchronous events

EC-CH95 EC-CH93

Explicit synchronization is done via the event control operator, **@**, operator, which allows a process to wait for ~~an explicit~~ a particular signal value change, or a clocking event (see section 13.9).

EC-CH94 ~~The syntax is for the synchronization operator is:~~

EC-CH56

```
~~@##([specific_edge] signal {or [specific_edge] signal});~~
```

~~Where *specific_edge* identifies the edge at which the synchronization occurs and can be:~~

EC-CH57

— ~~**negedge** : a negative (or falling) edge of the given (1-bit) signal~~

— ~~**posedge** : a positive (or rising) edge of the given (1-bit) signal.~~

~~If no edge is specified, the synchronization occurs on the next change in the specified signal.~~

~~The *signal* specifies the clocking-domain signal to which the synchronization is linked. It can be any signal in~~

~~a clocking domain, or a slice thereof. If the signal or the slice represents a 1-bit value, it's possible to synchronize to **posedge** or **negedge**, otherwise the synchronization is only to the next change. Slices can include dynamic indices, which are evaluated once, when the @## expression executes.~~

~~If the operator has more than one expression, joined by the **or** keyword, then the synchronization occurs when any of the expressions is satisfied.~~

EC-CH94

The syntax is for the synchronization operator is:

```
event_control ::=
                    @ event_identifier
                  | @ ( event_expression )
                  | @*
                  | @ (*)

event_expression ::=
                    expression [ iff expression ]
                  | hierarchical_identifier [ iff expression ]
                  | [ edge ] expression [ iff expression ]
                  | event_expression or event_expression
                  | event_expression , event_expression
```

Editor's Note: Replace preceding syntax line with BNF excerpt, once available.

The expression can denote clocking-domain input, 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);
```

EC-CH58 The values used by the synchronization ~~primitive~~ event control are the synchronous values, that is, the values sampled at the corresponding clocking event.

Editor's Note: The "primitive" is a keyword with unique meaning in Verilog. It shouldn't be used in the line above

EC-CH95

13.13 ~~Signal~~ Input sampling

~~All **input** or **inout** signals in a clocking domain are sampled at the clocking event of the corresponding clocking. If the signal has a non-zero input skew then the value of the signal is sampled *skew* time units before the clock edge (see figure 13-1 in section 13.3).~~

EC-CH60

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 read-only-sync [ROSYNC] of the time step skew time-units prior to the clocking event (see figure 13-1 in section 13.3). If the input skew is zero then the value sampled corresponds to the signal value at the start of the verification phase.

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.

EC-CH60 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.

EC-CH95

13.14 ~~Signal~~ Synchronous drives

EC-CH96 ~~Drives are used to propagate the value of **output** or **inout** signals at their corresponding clock edge. A drive is an assignment in which the left hand side is a signal in a clocking domain.~~

~~The syntax to drive a signal is:~~

EC-CH61

```
~~@##delay signal_expression = expression;~~
```

~~or~~

```
~~signal_expression <= expression;~~
```

~~The *delay* optionally specifies the number of clocking events (i.e. cycles) that pass before the signal is driven. When no delay is specified, the default is **@0**, i.e., the current cycle.~~

~~The *signal_expression* is either a bit-select, slice, or the entire signal in a clocking that is to be driven (concatenation is *not* allowed):~~

```
~~dom.sig         // entire signal~~

~~dom.sig[2]      // bit-select~~

~~dom.sig[8:2]    // slice~~
```

~~The *expression* can be any valid expression that is type compatible with the signal.~~

~~For example:~~

EC-CH62

```
~~bus.data[3:0] = 4'h5;  // drive on current cycle~~
~~@##1 bus.data = 8'hz;  // wait 1 cycle and then drive~~
```

~~The value driven onto an output signal is not applied until the signal's drive edge (typically the clocking event) plus any output skew has transpired.~~

EC-CH96

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:

```
[ ## event_count ] clockvar_expression = expression;
```

or

```
clockvar_expression = [ ## event_count ] expression;
```

Editor's Note: Replace preceding syntax lines with BNF excerpt, once available.

The clockvar_expression is either or 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 a syntax similar to the cycle-delay operator (see section 13.10), 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 statement, but the right-hand side expression is evaluated before the process blocks, instead of after.

Examples:

```
bus.data[3:0] = 4'h5;   // drive in current cycle

##1 bus.data = 8'hz;    // wait 1 (bus) cycle and then drive

##[2]; bus.data = 2;    // wait 2 default clocking cycles, then drive

bus.data = ##2 r;       // sample r, wait 2 (bus) cycles, the drive
```

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.

EC-CH98

13.14.1 ~~Blockingand nonblocking drives~~ Drives and nonblocking assignments

~~All zero-delay signal drives (no cycle delay and no skew) are queued and propagated in one fell swoop, right before *read-only synchronize time*. Zero-delay signal drives resemble Verilog nonblocking assignments, thus, reading the value of an inout signal immediately after it has been driven will yield the previous (sampled) value, not the driven value:~~

```
~~if( bus.data == 31 )~~
   ~~bus.data <= 27;~~
~~y = bus.data;        // y is 31 (not 27)~~
```

~~It is illegal to drive a clocking domain signal with zero delay using = (blocking drive). If the drive specifies a delay or an output skew then the blocking drive is allowed.~~

EC-CH98

Synchronous signal drives are queued and processed at the end of the verification phase, like nonblocking assignments, that is, they are propagated in one fell swoop without process execution in between drives.

Editor's Note: "one fell swoop" may not be appropriate for an international standard.

A key feature of **inout** clocking domain variables and synchronous drives is that a driven signal value does not change the clock domain input. This is because reading the input always yields the last sampled value, and not the current signal value. In this respect, an **inout** clocking domain variable resembles nonblocking assignments since reading the variable immediately after it has been assigned will yield the previous value, not the assigned value.

```
// bus.data is a clock domain inout, y is a variable
if( bus.data == 5 )                if( y == 5 )
   bus.data = 0;                      y <= 0;
$display( bus.data );              $display( y );    // both display 5
```

13.14.2 Drive value resolution

EC-CH99

~~When the same output signal in a clocking domain is driven more than once at the same time, the drives 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).~~

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).

EC-CH97

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. Naturally, clock-domain outputs driving a net (i.e., through different ports) cause the net to be driven to its resolved signal value.

EC-CH83

~~13.14.3 Drive / assignment ambiguity~~

~~The signal drive operator syntax may appear to be ambiguous with certain event control expressions in SystemVerilog. For example:~~

EC-CH63

```
~~integer j = 4;~~
~~@##j a = b;~~
```

~~The last statement above has the same syntactical form as a signal drive. But, it has two different meanings: in Verilog the process blocks until j changes value, whereas a signal drive causes the process to block for j cycles.~~

~~Nevertheless, the compiler can easily resolve the ambiguity by examining the type of operand involved in the signal drive (a above). If the operand is defined in a clocking domain, the signal is synchronous and should be driven using cycle semantics via a signal drive. Otherwise, the statement is a regular event control assignment.~~

EC-CH59

~~Section 14~~
~~Signal Synchronous Operations~~

EC-CH101

Editor's Note: Other than the intro, this entire section was moved to the end of section 13.

~~14.1 Introduction (informative)~~

EC-CH59

~~The clocking domain (see section 13) separates the timing and synchronization details from the structural, functional, and procedural elements of a test-bench. 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 signal synchronous operations to be written very succinctly, without explicitly using clocks or specifying timing. These signal synchronous operations are:~~

— ~~Synchronization~~

— ~~Sampling~~

— ~~Driving~~

Section 15 Program Block

Editor’s Note: This entire section is new for draft 1. Only the section titles have been highlighted as new text.

15.1 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 test-bench, the emphasis is not in the hardware-level details such wires, hierarchy, and interconnect, but in modeling the large environment in which a device needs to be verified. A lot of effort is spent in getting the environment properly initialized and synchronized, avoiding races between the hardware and the test-bench, automating the generation of input stimuli, and in reusing existing models and other infrastructure.

A typical test-bench contains type definitions, data declarations, subroutines, some form of structured connections to the design, and a program block. The **program** block serves two basic purposes:

1) It provides an entry point where the test-bench begins execution.

2) It creates a scope that encapsulates program-wide data.

A Verilog **module** provides both of these functions: it creates a new scope, and can include an **initial** block to serve as the test-bench entry point. Thus, a module is a natural choice for modeling the program block. However, such a *“test-bench module”* differs from a regular Verilog module in several ways. First, the communication between the test-bench and the design takes place via special *ports* that in addition to type, direction, and size, can also specify a clocking scheme (see section 13). Second, it provides for race-free cycle and transaction level abstractions as well as event abstractions. The **program** construct serves as a clear separator between the design and the test-bench, and, more importantly, it indicates the special nature of the *test-bench module*, thus, enabling specialized execution semantics for all elements within the program.

EC-CH67

The abstraction and modeling constructs simplify the creation and maintenance of test-benches. Furthermore, since modeling the environment can be a significant part of a test-bench, the same set of abstract test-bench constructs can be effective in writing models at a higher level of abstraction than currently provided by SystemVerilog. The ability to instantiate and individually connect each instance of a program enables their use as generalized models.

15.2 The program construct

The connection between design and test-bench uses the same interconnect mechanism as used by SystemVerilog to specify port connections, including interfaces. The syntax for the program block is:

EC-CH64

```
program program_name ( list_of_port_declarations );
   program_declarartions
   program_code
endprogram
```

Editor’s Note: Replace preceding syntax lines with BNF excerpt, once available.

Editor’s Note: Is the Verilog-2001 module parameter list also supported?.

For example:

```
program test (input clk, input [16:1] addr, inout [7:0] data);
   ...
endprogram
```

or

```
program test ( interface device_ifc );
   ...
endprogram
```

EC-CH64

The `list_of_port_declarations` allowed by a program is the same as the one allowed for any Verilog module. A more complete example is included in sections 13.7 and 13.8.

Although the **program** construct is new to SystemVerilog, its inclusion is a natural extension. The program construct can be considered the declaration of a special type of module (i.e., a module with a test-bench attribute). Once the program block has been declared, it can be instantiated in the proper hierarchical location (typically at the top level) and its ports can be connected in the same manner as any other module.

Some of the test-bench constructs and data-types cannot be used in declarative contexts such as module ports, gates, or continuos assignments. These constructs will be limited to the procedural context (i.e., the test-bench environment). This limitation is not new either, it simply extends the rules set forth by SystemVerilog, which disallows automatic variables from triggering event expressions or be written using nonblocking assignments. Likewise, all the dynamic test-bench constructs—objects handles, dynamic and associative arrays, strings, and events—will be limited to the procedural context.

15.3 Static data initialization

In SystemVerilog, setting the initial value of a static variable as part of the variable declaration requires that the initialization occurs before any **initial** or **always** blocks are started. Likewise, SystemVerilog allows static data in a program block (including static class members) to specify an initial value as part of their declaration, and requires that all such data be initialized before the program block begins execution. It is important to note that SystemVerilog initial values are not constrained to simple constants, but may include run-time expressions, including dynamic memory allocation. For example, a static class can be initialized via its **new** method (see section 11.4), or a **mailbox** may be initialized by calling its **new** method (see section 11.4).

Note: While this does not represent a conflict with SystemVerilog 3.0, it may require a special pre-initial pass at run-time, which may need changes to the initial SystemVerilog simulation cycle. This is one of the requirements that differentiates a program from a module.

Editor's Note: The preceding paragraph seems rather odd for a standard. Is it necessary to state this at all?

15.4 Scope and lifetime

The following test-bench constructs all have **module** or **program** scope. They share the name space at the hierarchical scope in which they are declared, so no two of them can have the same name:

— Class declarations

— Enumerated types and enumeration Values

— Clocking domains (see section 13)

— Program block

The program block contains a single implicit **initial** block, and no **always** blocks or other programs or modules. Programs blocks cannot be nested.

All constructs declared within the **program** are local in scope (local to the program block) and have static lifetime.

Global declarations (outside the program block or any other module) reside in **$root** and have static lifetime.

Class declarations create a new scope.

EC-CH65 ~~Tasks and functions cannot be nested within themselves, but they can contain block statements that create a scope. Block statements do not have to be named to create a new scope.~~

The program scope rules are consistent with SystemVerilog. The declaration in the closest enclosing scope is matched: A scope nested inside another scope has visibility of (and may reference) all elements visible or declared in its parent scope. A name declared inside a scope hides all elements with the same name that are visible or declared in the parent scope.

15.5 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. Users can control the degree of communication by choosing to share data via **$root** or hierarchical reference, or making the data private by declaring it inside the corresponding program block. EC-CH66

EC-CH67 ~~The abstraction and modeling constructs simplify the creation and maintenance of test-benches. Furthermore, since modeling the environment can be a significant part of a test-bench, the same set of abstract test-bench constructs can be effective in writing models at a higher level of abstraction than currently provided by SystemVerilog. The ability to instantiate and individually connect each instance of a program enables their use as generalized models.~~

15.6 Eliminating zero-skew races

If both input and output skews are set to `#0` (see section 13.3) then input signals are sampled at the same time as their corresponding clock edge, and output signals are driven at the same time as their corresponding clock edge. That is, both samples and drives happen at the same time. This type of zero-delay processing is a typical source of non-determinism that often results in races. However, races are minimized by means of two mechanisms. First, by constraining test-bench processes to execute only *after* nonblocking assignments, once all zero-delay transitions have propagated through the design and the system has reached a steady state. Second, by queuing all outgoing signal drives until the end of the test-bench execution cycle, and then propagating all the drives as one event. This is described in section 13.14.1.

Supporting signals with zero input or output skew without races is an important feature of the test-bench environment. This is because test-benches with no timing information are quite common, particularly during the early phases of a design, when designers are mostly focused on functionality and not timing.

15.7 Eliminating races and SystemVerilog event queue

There are two major sources of non determinism in Verilog. The first one is that active events can be taken off the queue and 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 test-bench perspective, these effects are all unimportant details. The primary task of a test-bench is to generate valid input stimulus for the design under test, and to verify that the device operates correctly. Furthermore, test-benches 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 non determinism and races inherent in the Verilog event queue management, test-bench processes execute only after the system has settled to its steady state. This is after *nonblocking assignments* have been processed, thus, treating all transitions towards the steady state in the same consistent manner (from the test-bench perspective). Accordingly, signals driven from the test bench with no delay are propagated into the design as one event immediately before *read-only synchronize time*. With this behavior, the correct cycle semantics can be modeled without races, thereby making the test-bench environment compatible with the assertions mechanisms and formal tools.

It is important to note that simply setting non-zero skews on the signals does not eliminate the potential for races. Non-zero skews only address a single clocking domain. When multiple clocks are used, the arbitrary order in which overlapping or simultaneous clocks are processed is still a potential source for races. The solution requires a special execution time after *all* events have been processed, including all clocks driven by non-blocking assignments.

EC-CH68

~~In order to standardize the cycle behavior, the execution after nonblocking assignments described above must be added to the SystemVerilog's event cycle. This is a requirement from many other subsystems such as monitors, checkers, waveform tools, and temporal assertions. However, it is the test-bench that exacerbates this need because in addition to examining the current state, it must also react and provide new stimuli for the next cycle, which is often driven with no delay.~~

15.8 Blocking tasks in cycle/event mode

EC-CH69

Calling tasks or functions in the program block from ~~other~~ *design* modules is not allowed. The rationale for this is that the design must not be aware of the test-bench. However, calling subroutines in ~~other~~ *design* modules from within the program is allowed. Calling a function presents no problem and can be treated like a regular function call. However, calling a blocking task outside the program block from inside the program does require explicit synchronization upon return from the task. That is, postpone execution until after nonblocking assignments.

15.9 Program control tasks

In addition to the normal simulation control tasks (**`$stop`** and **`$finish`**), a **`program`** can use the **`$exit`** control task.

15.9.1 $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();
```

Editor's Note: Replace preceding syntax lines with BNF excerpt, once available.

When a program executes its last statement, it implicitly calls **`$exit`**. Calling **`$exit`** causes all processes spawned by the current program to be terminated.

~~Section 16~~
~~Assertions [SV 3.0]~~

Editor’s Note: This entire section is superceded by the following section (added for SV 3.1 draft 3).

~~16.1 Introduction (informative)~~

~~An assertion is a statement that a property must be true. There are two kinds of assertions: concurrent assertions which state that the property must be always be true, e.g. throughout a simulation, and procedural assertions which are incorporated in procedural code and apply only for a limited time or under limited conditions.~~

~~There are various applications of assertions. They can be included in the design, to document the assumptions made by the designer and to facilitate “white box” testing. They can be outside the design, either in a testbench to check the response of the design to the stimulus, or to control a tool such as a stimulus generator or a model checker.~~

~~Concurrent assertions can be coded as modules in a library, but this limits the complexity of the property that can be expressed easily. It is more difficult to code procedural assertions as a library of tasks in Verilog, because events cannot be arguments, each assertion may need static data, and tasks block.~~

~~16.2 Procedural assertions~~

```
proc_assertion ::=          // from Annex A.6.10
        immediate_assert
      | strobed_assert
      | clocked_immediate_assert
      | clocked_strobed_assert
immediate_assert ::= assert ( expression )
        statement_or_null
        [ else statement_or_null ]
strobed_assert ::= assert_strobe ( expression )
        restricted_statement_or_null
        [ else restricted_statement_or_null ]
clocked_immediate_assert ::= assert ( expr_sequence ) step_control
        statement_or_null
        [ else statement_or_null ]
clocked_strobed_assert ::= assert_strobe ( expr_sequence ) step_control
        restricted_statement_or_null
        [ else restricted_statement_or_null ]
restricted_statement_or_null ::=
        restricted_statement
      | { attribute_instance } ;
restricted_statement ::=
        [ block_identifier : ] restricted_statement_item
restricted_statement_item ::=
        { attribute_instance } proc_assertion
      | { attribute_instance } system_task_enable
      | { attribute_instance } delay_or_event_control statement
      | { attribute_instance } restricted_seq_block
restricted_seq_block ::= begin [ : block_identifier ] { block_item_declaration }{ restricted_statement }
        end [ : block_identifier ]
expr_sequence ::=
        expression
      | [ constant_expression ]
      | range
      | expr_sequence ; expr_sequence
      | expr_sequence * [ constant_expression ]
      | expr_sequence * range
      | ( expr_sequence )
step_control ::=
        @@ event_identifier
      | @@ ( event_expression )
```

~~*Syntax 16-1—Assertion syntax (excerpt from Annex A)*~~

~~SystemVerilog provides four kinds of procedural assertions, which allow the user to test boolean expressions or sequences of boolean expressions, and perform some action based on whether the expression or sequence is true or false. Immediate assertions test the value of a boolean expression at the time the statement is executed, and may be used in always and initial blocks, tasks and functions. Strobed assertions schedule the evaluation~~

~~of the expression to be delayed until the end of the current timeslice, to allow for glitches to settle. Strobed assertions may be used in **initial** and **always** blocks and tasks, but not in functions, since functions must return immediately.~~

~~To test sequences of expressions, it is necessary to specify a sampling clock event on which to test each element of the sequence. Therefore, Clocked Immediate and Clocked Strobed assertions are added to allow progressive evaluation of sequences of expressions. Since these clocked assertions, by definition, take time, they cannot be used in functions. Clocked immediate assertions evaluate each expression in the sequence when the clock event triggers, and clocked strobed assertions evaluate each expression at the end of the timeslice at which the event triggers.~~

~~16.3 Immediate assertions~~

~~The immediate assert statement is a test of an expression performed when the statement is executed in the procedural code. The expression is treated as a condition like in an if statement.~~

~~[identifier **:**] **assert (** expression **)** [pass_statement] [**else** fail_statement]~~

~~The pass statement is executed if the assertion succeeds, i.e. the expression evaluates to true. As with the if statement, if the expression evaluates to 'X', 'Z' or '0', then the assertion fails. The pass statement may, for example, record the number of successes for a coverage log, but may be omitted altogether. If the pass statement is omitted, then no action is taken if the assert expression is true. The fail statement is executed if the assertion fails (i.e. the expression does not evaluate to a known, non-zero value) and can be omitted. The optional assertion label (identifier and colon) creates a notional named block around the assertion statement (or any other SystemVerilog statement) and can be displayed using the %m format code.~~

```
~~assert_foo : assert (foo) $display("%m passed"); else $display("%m failed");~~
```

~~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 may be specified by including one of the following severity system tasks in the fail statement.~~

— ~~**$fatal** is a run-time Fatal, which terminates 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 22.4.~~

~~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 an assertion fails and no **else** clause is specified, the tool shall, by default, call **$error**, unless a tool-specific command-line option is enabled to suppress the failure.~~

~~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 will be printed at time 15, but the user-defined string printed will be “assert failed at time 10”.~~

~~The display of messages of warning and info types can be controlled by a tool-specific 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;~~
```

~~The assert statement serves as guidance to non-simulation tools that the condition should be true. The second statement above is equivalent to:~~

```
~~if ( y!=0) begin flag = 1; end~~
```

~~16.4 Strobed assertions~~

~~If an immediate assertion is in code triggered by a timing control that happens at the same time as a blocking assignment to the data being tested, there is a risk of the wrong value being sampled. For example:~~

```
~~always @(posedge clock) a = a + 1; // blocking assignment~~
~~always @(posedge clock) begin~~
   ~~....~~
   ~~assert (a < b);~~
~~end~~
```

~~This can be solved by using a strobed assertion, which waits in the background until the end of the time slot, like the Verilog **$strobe** system task.~~

```
~~always @(posedge clock) begin~~
   ~~....~~
   ~~cas:assert_strobe (a < b);~~
~~end~~
```

~~Strobed assertions can have pass or fail statements like immediate assertions. However, the statements are restricted to another assertion statement, a system task call, a statement preceded by a delay control or an event control, or sequential block containing them. This is because the statement happens after the assertion is evaluated, at the end of the time slot, and hence must not create more events at that time slot or change values. Statements which cause additional events to occur at the current time shall be an error.~~

~~The example below illustrates the effect of blocking and nonblocking assignments on immediate and strobed assertions. The immediate assertions are like **$display** statements and the strobed assertions are like **$strobe** statements.~~

```
~~module test;~~
~~reg [3:0] a=0; c=0, d=0;~~
~~reg clk = 0;~~
~~wire b;~~

~~initial begin~~
   ~~#10 clk = 1;~~
   ~~forever #5 clk = !clk; // posedge clk at 10,20,30,40...~~
~~end~~

~~assign b = a+1;~~

~~always @(posedge clk) begin~~
   ~~a1: assert(c<3); // fails at time 40~~
   ~~c = c+1;~~
   ~~a2: assert(c<3); // fails at time 30~~
   ~~a <= a+1;~~
   ~~a3: assert(a<3); // fails at time 40~~
   ~~a4: assert(b<3); // fails at time 40~~
   ~~a5: assert_strobe(a<3); // fails at time 30~~
   ~~a6: assert_strobe(b<3); // fails at time 30~~
~~end~~

~~always @(a) begin // models transient behavior on comb. nets~~
   ~~d = a+2; // spikes to 2 at 0, 3 at 10, 4 at 20~~
   ~~assert(d<3); // fails at time 10~~
   ~~d = d-1; // settles to 1 at 0, 2 at 10, 3 at 20~~
   ~~assert(d<3); // fails at time 20~~
~~end~~

~~always @(d) assert_strobe (d<3); // fails at time 20~~

~~endmodule~~
```

~~16.5 Sequential assertions~~

~~In addition to assertions about single expressions, it is often useful to assert sequences of expressions over time. One way of doing this is to use nested immediate assertions, where each subsequent assertion is the pass statement of the previous assertion.~~

```
~~always @(posedge clk or negedge rst)~~
   ~~if(state == REQ)~~
      ~~a7: assert(req1) // no semicolon~~
      ~~@(posedge clk) assert(gnt)~~
      ~~@(posedge clk) assert(!req1);~~
```

~~The above example verifies the sequence that, if state is equal to `REQ`, the `req1` signal must be true immediately, then on the next posedge `clk`, `gnt` must be true and on the following posedge `clk`, `req` must be false. Note that the assertion statement itself is nonblocking, so the sequence in assertion `a7` is equivalent to:~~

EC-CH88

```
~~always @(posedge clk or negedge rst)~~
   ~~if(state == REQ)~~
   ~~a8: assert(req1)~~
   ~~process~~
   ~~fork~~
      ~~@(posedge clk) assert (gnt)~~
         ~~@(posedge clk) assert(!req1);~~
   ~~join_none~~
```

~~To simplify this complex nested assertion, a *sequential regular expression* is used in the assert statement. Sequential regular expressions require a *step control event expression* to specify the timing between evaluations of each element in the regular expression. Using a sequential regular expression, the assertion `a8` could be rewritten as:~~

```
~~always @(posedge clk or negedge rst)~~
~~if(state == REQ)~~
   ~~a9: assert(req1;gnt;!req1) @@(posedge clk);~~
   ~~// note the @@ token to distinguish the step control from the pass statement~~
```

~~A sequential regular expression is a semicolon-delimited list of expressions. The first expression in the list is evaluated immediately when the assert statement is executed. The other subsequent expressions are evaluated one at a time on successive occurrences of the step control event expression. In assertion `a9` above, `req1` is evaluated immediately when the assert statement is executed, just as for an immediate assertion, then `gnt` is evaluated on the next posedge `clk` event, and so on.~~

~~The '**@@**' token is introduced to distinguish the step control from an ordinary event control at the start of the pass statement. Consider the following:~~

```
~~always @(posedge clock or negedge rst)~~
   ~~if(state == REQ)~~
      ~~a10: assert (req1)~~
      ~~@(posedge clk) // This is an event control in the pass statement~~
         ~~$display("Hello at time %t", $time);~~
```

~~In this example, the "`@(posedge clock)`" in the pass statement causes the display action to occur on the next posedge of clock after the assertion succeeds. Therefore, a new token is required to distinguish the assertion sequence step control from the pass statement.~~

~~Note that, since the first expression is evaluated immediately, assertion `a9` above is equivalent to:~~

```
~~always @(posedge clk or negedge rst)~~
   ~~if(state == REQ)~~
      ~~assert(req1)~~
         ~~assert(1;gnt;!req1) @@(posedge clk);~~
```

~~The sequence notation "`(1;<expression_or_sequence>)`" is a convenient shorthand, indicating that the `<expression_or_sequence>` is to be evaluated on the next occurrence of the step control event. This is because the expression '`1`' is evaluated immediately and is always true.~~

~~Sequential assertions using the **`assert`** keyword are called *clocked immediate assertions*, since the expressions are evaluated as with immediate assertions. Similarly, clocked strobed assertions may be written using the **`assert_strobe`** keyword, in which each expression in the sequence is evaluated either at the end of the timeslice in which the assertion is executed or in which the step control event occurs. The pass and fail statements of clocked strobed assertions have the same restrictions as strobed assertions.~~

~~Specifying an explicit step control for a sequence makes it possible to use clocked assertions in combinational always blocks.~~

```
~~always @(foo,bar)~~
   ~~assert_strobe (a;b;c) @@(posedge clk);~~
   ~~// look for a when foo or bar changes, then look for b on next posedge clk~~
```

~~Since it is common for combinational always blocks to be executed multiple times in a single timestep as the signals in the event trigger expression settle, it is common to use strobed assertions in combinational always blocks. Immediate assertions are commonly used in clocked always blocks.~~

~~Note that to avoid races, the variables read in clocked immediate assertions should be written by nonblocking assignments. Expressions in clocked strobed assertions are always sampled at the end of the timestep, so no~~

~~race conditions should occur.~~

~~An assertion could be executed twice in the same timestep via a zero-delay loop or a combinational always block, for example. If a clocked immediate assertion is executed more than once at the same timestep, the first expression in the sequence will be re-evaluated. If a clocked strobed assertion is executed more than once at the same timestep, the first expression in the sequence will be evaluated once at the end of the timestep.~~

~~An assertion shall only spawn a single process to evaluate the next expression in the sequence at the next step control event. If the step control event occurs multiple times at the same timestep, then in a clocked immediate assertion the current expression in the sequence shall be re-evaluated. In a clocked strobed assertion, the current expression will still be evaluated only once at the end of the timestep. The next expression in the sequence shall not be evaluated until the step control occurs in a later timestep.~~

~~As mentioned above, the execution of a sequential assertion spawns a process that monitors each event in the sequence when the step control event occurs. If the sequential assert statement is executed again before the sequence spawned by the original execution has expired, then a new process shall be spawned that looks for the sequence starting at the current timestep. It is therefore possible to have multiple processes in-flight, each monitoring the same sequence, but offset in time. It is possible for these multiple processes to be satisfied by the same sequential behavior, even though the processes are offset in time. In such a case, both processes shall terminate at the same timestep, in which both sequences are satisfied. Consider:~~

```
~~module top;~~
   ~~reg clk = 0;~~
   ~~reg a,b,c;~~

   ~~initial begin~~
      ~~#10 clk = 1;~~
      ~~forever begin~~
         ~~clk = 0;~~
         ~~clk = 1; // 2 posedges clk at 10,20,30,40...~~
         ~~#5 clk = 0;~~
         ~~#5 clk = 1;~~
      ~~end~~
   ~~end~~

   ~~always @(posedge clk)~~
      ~~assert(a;b;c) @@(posedge clk);~~
      ~~// 'a' is evaluated only once at 10, 'b' once at 20, 'c' once at 30~~
```

~~Note that the step control expression may be any valid event expression in SystemVerilog. The following assertions all use valid step control expressions:~~

```
~~bit clk;~~
~~event ev1;~~

~~always @(posedge clk or negedge reset) begin~~
   ~~assert (a;b;c) @@(negedge clk); // sequence sampled on negedge clk~~
   ~~assert (a;b;c) @@(clk); // sequence sampled on any edge of clk~~
   ~~assert (a;b;c) @@(ev1); // sequence sampled when event ev1 fires~~
   ~~a11: assert(a;b;c) @@(posedge clk iff !rst);~~
      ~~// sequence sampled on posedge clk if rst == 0~~
~~end~~
```

~~Note the use of the **iff** operator in assertion a11 above. In effect, this allows a "gated clock" to control the assertion without the user having to declare the gated clock explicitly (see section 8.9). Because this could have significant impact on the ability of Formal Verification tools to evaluate the assertion successfully, it is recommended that this construct be used only for simulation.~~

~~This flexibility also allows nested assertions to use different clocks:~~

```
~~always @(posedge clk) begin~~
   ~~assert (a;b) @@(posedge clk) // on posedge clk~~
   ~~assert (1;c;d) @@(negedge clk); // look for c and d on negedge clk~~
   ~~assert (e;f) @@(posedge clk2)~~
   ~~assert (1;g;h) @@(ev1);~~
~~end~~
```

~~16.6 More expression sequences~~

~~A number of steps can be skipped either by writing expressions which are always true:~~

```
~~assert (a;1;1;c) @@(posedge clk); // two steps between a and c~~
```

~~or by using the notation [n] to count the number of steps:~~

```
~~assert (a;[2];c) @@(posedge clk); // two steps between a and c~~
~~assert (a;[1];[1];c) @@(posedge clk); // two steps between a and c~~
```

~~Note that in [n], the n must be a non-negative literal or a constant expression. [0] has no effect. The number of steps to be skipped may also be expressed using [min:max], where the minimum number of steps must be greater than or equal to zero. Both min and max must be a literal or constant expression.~~

```
~~assert (a;[0:10];b) @@(posedge clk);~~
~~// b occurs between the next and 11th clock edges, inclusive~~
```

~~If an expression must be repeated a defined number of times, this can be expressed with a trailing *[n]. If it can be repeated a minimum or maximum number of times, this can be expressed with a trailing *[min:max]. These repetition counts must also be literals or constant expressions.~~

```
~~assert ((a; b)*[5]) @@(posedge clk); // a;b;a;b;a;b;a;b;a;b~~
~~assert ((a*[0:3];b;c)) @@(posedge clk); // equivalent to~~
                        ~~//    (b;c) or (a;b;c) or (a;a;b;c) or (a;a;a;b;c).~~
```

~~This means that a sequence `a;ab;a;b;c;` will pass. The expression sequence is not equivalent to `((a && !b)* [0:3];b;c)`, which would fail the same sequence.~~

~~The rules for specifying repeat counts are summarized as:~~

- ~~Each form of repeat count specifies a minimum and maximum number of occurrences~~
- ~~expr*[n:m], where n is the minimum, m is the maximum~~
- ~~expr*[n], same as expr*[n:n]~~
- ~~[n], same as 1*[n:n]~~
- ~~The sum of the minimum repeat counts for all terms in a sequence must be greater than 0~~
- ~~The sequence as a whole cannot be empty~~
- ~~The last term in a sequence shall not have a min:max range of repetition. If it does, it shall be an error.~~

~~16.7 Aborting assertions externally~~

~~A named assertion can be disabled like any other named SystemVerilog block. If this is done before the expression sequence has finished, it means that neither the pass statement nor the fail statement shall be executed.~~

```
~~disable cas;~~
```

~~Note that if the **disable** is applied at the same simulation time step as the last clock step of a sequence, there is a race in the case of an immediate assertion, but a strobed assertion is always disabled.~~

~~If the pass or fail statement is executing when the disable is executed, the statement shall be disabled, just as if the statement were in another named block that gets disabled.~~

~~If a sequential assertion has been executed multiple times before the sequence has expired, then all instances of the assertion shall be disabled when the assertion is disabled.~~

~~16.8 Controlling assertions~~

~~System tasks are provided to limit assertion checking to part of the design and part of the simulation time.~~

~~The **$assertoff** system task stops the checking of all specified assertions. When these assertions are encountered before a subsequent **$asserton**, the assert statement shall be ignored. Neither the pass statement nor the fail statement shall be executed. An assertion that is already executing, including execution of the pass or fail statement, is not affected by **$assertoff**.~~

~~The **$assertkill** system task disables all specified assertions and prevents them from executing until a subsequent **$asserton**. As with disable, the checking of the sequence is aborted, and neither the pass nor fail statement is executed.~~

~~The **$asserton** system task re-enables the execution of all specified assertions.~~

~~The assertion control system tasks may be used with or without arguments. When invoked with no arguments, the system task refers to all assertions throughout the model. Refer to section 22.5 for the syntax of these system tasks.~~

~~Assertions are on by default until turned off. When an assertion control task is specified with arguments, the first argument indicates how many levels of the hierarchy below each specified module instance to turn on or off. Subsequent arguments specify which scopes of the model in which to control assertions. These arguments can specify entire modules or individual named assertions within a module. Setting the first argument to 0 causes all assertions in the specified module and in all module instances below the specified module to be affected. The argument 0 applies only to subsequent arguments which specify module instances, and not to individual assertions.~~

~~16.9 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:~~

BC42-18

— ~~**$onehot** (<expression>) returns true if only one and only one bit of expression is high.~~

— ~~**$onehot0**(<expression>) returns true if at most one bit of expression is low.~~

— ~~**$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. 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.~~

Section 16 Assertions

Editor's Note: This entire section is new for draft 3, and replaces the SV 3.0 section on assertions. Only the Section titles have been highlighted as new text.

Editor's Note: I still need to check all formatting for this section. Also need to map all cross references to the old assertion section to this new section.

16.1 Introduction (informative)

System Verilog adds features to specify assertions (or properties) of a system. An assertion specifies a specific behavior of the system. There are two kinds of assertions: concurrent or immediate.

Immediate assertions follow event semantics for their execution and get 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 may 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 semantics. 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.

16.2 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 treated as a condition like in an `if` statement. The syntax of the immediate assertion statement is as follows.

```
immediate_assertion::=
        [ identifier : ] 'check' '(' expression ')' action_block
action_block::=
        statement_or_null [ 'else' statement_or_null ]
statement_or_null::=
        statement
        | ';'
```

Syntax 16-2—Immediate assertion syntax

The statement associated with the success of the assert statement is called `pass` statement, and is executed if the expression evaluates to true. As with the `if` statement, if the expression evaluates to '`X`', '`Z`' or '`0`', then the assertion fails. The pass statement may, for example, record the number of successes for a coverage log, but may be omitted altogether. If the pass statement is omitted, then no user specified action is taken when the ~~assert~~ check expression is true. The statement associated with **`else`** is called a fail statement, and is executed if the assertion fails (i.e. the expression does not evaluate to a known, non-zero value) and can be omitted. The optional assertion 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 code.

```
assert_foo : check (foo) $display("%m passed"); else $display("%m failed");
```

Note: The pass and fail statements are executed as part of verification code. The distinction between design and verification code is being discussed in other commitees, and a special scheduling mechanism to support the two types of code will also be devised. The main objective here is to prevent modification of design behavior as a result of assertion monitoring activities.

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 may be specified by including one of the following severity system tasks in the fail statement:

— **`$fatal`** is a run-time Fatal, which terminates the simulation with an error code. The first argument passed to **`$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 16.4 of System Verilog3.0 LRM.

Need software cross reference above

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 an assertion fails and no **`else`** clause is specified, the tool shall, by default, call **`$error`**, unless a tool-specific command-line option is enabled to suppress the failure.

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)
      check(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 will be printed at time 15, but the user-defined string printed will be “assert failed at time 10”.

The display of messages of warning and info types can be controlled by a tool-specific 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.

```
    check (myfunc(a,b)) count1 = count + 1; else ->event1;
    check (y == 0); else flag = 1;
```

16.3 Concurrent assertions

Concurrent assertions describe behavior that spans over time. 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 valued. 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 timing model employed in 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. In addition, a user can choose to use the simulation time as a clock to express asynchronous events.

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 may tick only once at any simulation time. The value of a variable in an expression at a clock tick is sampled at the end of one simulation timestep (i.e. at read-only synchronization time, as defined by the PLI) before the clock tick. In an assertion, the sampled value is the only valid value of a variable at a clock tick. Figure 16-2 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 9. The value of variable `req` at clock tick 9 is low and remains low.

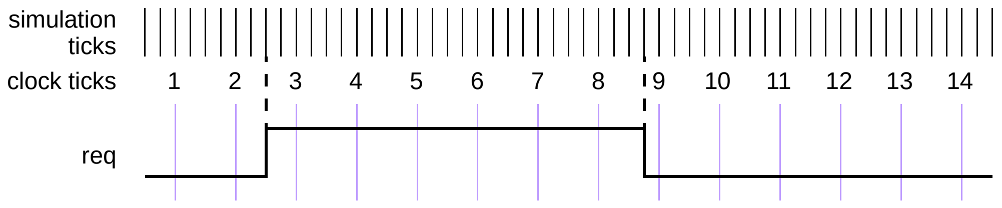


Figure 16-2—Sampling a Variable on Simulation Ticks

The sampled value of a signal with respect to its clock is the value of the variable at the end of the simulation time (i.e. read-only sync) before the clock event occurs.

An expression is always tied to a clock definition. The values of variables are sampled only at clock ticks. These 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 may get sampled.

— The two words "clock tick" and "sampling event" are used synonymously in this document.

The clock expression that controls evaluation of a sequence may be more complex than just a single signal name. An expression such as (clk && gate) could be used to represent a gated clock. 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 may only transition once at any simulation time. The clock expressions must be evaluated with zero delay.

16.4 Sequences

A sequence is a list of SystemVerilog boolean expressions in a linear order of increasing time. These boolean expressions must be true at those specific points in time 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 unit.To determine a match of a sequence, the boolean expressions are evaluated at each successive sample point 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* that concisely specify a range of possibilities of when an expression needs to hold true. These sequential regular expressions can actually describe a set of one or more sequences that satisfy the sequential expression.

The basic composition of a sequence consist of a boolean expression concatenated by another boolean expression. The concatenation specifies a delay between the two boolean expressions. The following is the syntax for sequence concatenation.

```
sequence_expr  ::=
          sequence_phrase  { ; [range] sequence_phrase }
sequence_phrase  ::=
          sequence_element
          | range sequence_element
sequence_element :: =
          boolean_item
          | (sequence_expr )
boolean_item ::=
          boolean_expr
          | true
range ::=
          [ constant_range_expression ]
          | [ constant_range_expression : constant_range_expression ]
          | [constant_range_expression : inf ]
```

Syntax 16-3—Sequence concatenation syntax

In this syntax:

— `constant_range_expression` is a compile-time constant expression that results in an integer value

— `constant_range_expression` can only be 0 or greater and **true** unconditionally evaluates to true boolean value.

— The keyword **true** unconditionally evaluates to true boolean value.

— The keyword **`inf`** is used to indicate the end of simulation. For formal verification tools, **`inf`** is interpreted as infinity.

— When a range is specifies 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 element in a sequence is checked at the first occurrence of the clock at or after the event 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 range specifies that the sequence_expr should occur later than the 'current' cycle. A range of [1] indicates that the next element should occur a single cycle later than the ‘current’ cycle. A ‘;’ without a range is equivalent to a ‘;’ with a range [1]. A range of [0] specifies that the next element should occur in parallel with the ‘current’ cycle.

When a range specifier appears at the start of the sequence without ‘;’, its meaning is identical to as if the ‘;’ is prepended to the sequence. The semantics are the same.

The following are examples of unary delay expressions. A unary delay, i.e. an expression with delay as the prefix, must be enclosed in parenthesis.

```
([0] a)  means a
([1] a)  means true; a
([2] a)  means true;true;a
([0:3]a) means(a) or (true;a) or (true;true;a) or (true;true;true;a)
```

An example of a delay expression is as follows:

```
a;[2] b means a ; true ; b
```

Note that the following two are equivalent:

```
a; true ;[2] b    means    a ; true ; true; b
a; (true;[2] b)   means    a ; true ; true; b
```

A sequence:

```
req; gnt;!req
```

This sequence specifies that **`req`** be true on the current clock tick, **`gnt`** will be true on the first subsequent tick and **`req`** will be false on the next tick after that. The ‘;’ operator specifies one clock tick separation. When a number is appended to semicolon, The number of samples is prepended to the expression in the sequence, as in

```
req;[2]gnt
```

This specifies that req will be true on the current sample, and gnt will be true on the second subsequence sample, as shown in figure Figure 16-3.

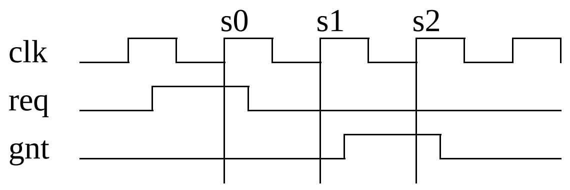


Figure 16-3—Concatenation

The following specifies that ‘b’ will be true on the Nth sample after ‘a:

```
a;[N]b      // check b on the Nth sample
```

To specify 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 ;b ;c // first sequence seq1
d ;e ;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] sampling, c and d must occur at the same time, resulting in the concatenated sequence being is equivalent to:

```
a;b ;c&&d ;e ;f
```

In cases where the concatenation can occur anytime between two points in time, a time window can be specified as follows:

```
req;[4:32] gnt
```

In the above case, signal gnt must be true at some sampling event between sampling events ranging from 4 to 32 after the current sample.

The time window can extend to the end of simulation in the example below.

```
req;[4:inf] gnt
```

A sequence can be unconditionally extended by using **`true`**.

```
a ;b ;c ;[3]true
```

After signal c, the signal length is extended by 3 sample events. Such adjustments in the length of sequences are required when complex sequences constructed by combining simpler sequences.

16.5 Declaring sequences

Sequences can be reused by declaring them as objects of type **`sequence`** with optional parameters:

```
seq_declaration ::=
        sequence [event_control] named_seq { , named_seq} ;
named_seq ::=
        identifier [ ( identifier { , identifier} ) ] = ( sequence_expr )
```

Syntax 16-4—Declaring sequence syntax

The event_control specifies the clock for the sequence.

The declaration can optionally include arguments that allow the same sequence to be instantiated multiple times with different argument values. The actual arguments can be boolean or sequence expressions.

Note that variables referenced within a seq that are not formal arguments to the sequence are resolved hierarchically from the scope in which the seq is instantiated.

```
sequence @(posedge clk) s1 = (a;b;c), s2 = (d;e;f);
sequence @(nededge clk) s3 = (g;h;i);
```

In this example, sequences s1 and s2 are sampled on each successive posedge clk. The sequence s3 is sampled on negedge clk.

Another example of sequence declaration with arguments is shown below:

```
sequence s20_1(data,en) = (!frame && (data==data_bus) ; (c_be[0:3] == en));
```

A sequence can be referred in properties by referencing its name. A hierarchical name can be be used consistent with the System Verilog naming conventions.

16.6 Sequence operations

16.6.1 Repetition in sequences

Following is the syntax for sequence concatenation (sequence_phrase from concatenation has been extended with repetition clauses).

```
sequence_phrase ::=
        sequence_element
        |sequence_element * range
        | boolean_expr =* range
```

Syntax 16-5—Sequence concatenation syntax

The repetition counts are specified with range and must be literals or constant expressions.

To specify the repetition of an expression within a sequence, the expression may simply be repeated, as:

```
a;b;b;b;c
```

or the number of repetitions may be specified with a trailing “*[N]”, as:

```
a;b*[3];c
```

A repeat specifies that the item or expression should 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 should occur N times in succession - e.g.,

```
a*[3]   means a ; a ; a
```

The syntax allows combination of a delay and a repeat in the same sequence with no separation by ‘;’, but requires that the repeated item be delimited by parentheses. The following are both allowed:

```
true;[3](a*[3])     means          true;true;true;a;a;a
(true;[2]a)*[3]     means          (true;[2]a);(true;[2]a);(true;[2]a)
                    which means    true;true;a;true;true;a;true;true;a
```

As an example, with named sequences

```
sequence seq1 = ([2]a);     means         true ; true ; a
sequence seq2 = (b;seq1);   means         b;([2]a)
```

```
                                which  means   b;(true ; true ; a)
                                which  means   b;true;true;a
```

A sequence can be repeated as follows:

```
(a ; b)*[5]
```

is same as:

```
a;b;a;b;a;b;a;b;a;b
```

A repetition with a range of maximum and minimum number of times can be expressed with a trailing *[min:max]. As an example, the following two expression are equivalent.

```
(a ; b)*[1:5]
(a;b)or(a;b;a;b;)or(a;b;a;b;a;b)or(a;b;a;b;a;b;a;b)or(a;b;a;b;a;b;a;b;a;b)
```

The following two expression are also equivalent.

```
(a*[0:3];b;c)
(b;c) or (a;b;c) or (a;a;b;c) or (a;a;a;b;c).
```

To specify potentially infinite number of repetitions, the keyword **inf** is used. So,

```
a; b*[1:inf];c
```

means ‘a’ is true on the current sample, then ‘b’ will 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 “*[N]” notation indicates consecutive repetition of an expression. It is also possible to specify non-consecutive repetition of a *boolean* expression with:

```
a;b*=[min:max];c
```

This is equivalent to:

```
a;((!b*[0:inf];b ~~)~~)*[min:max]);c
```

Adding the range specification to this allows the construction of useful sequences containing a boolean expression that is true for at most N samples:

```
a;b*=[1:N];c // a followed by at most N occurrences of b, followed by c
```

The rules for specifying repeat counts are summarized as:

— Each form of repeat count specifies a minimum and maximum number of occurrences

— expr*[n:m], where n is the minimum, m is the maximum

— expr*[n] is the same as expr*[n:n]

— The sequence as a whole cannot be empty

— If n is 0, then there must be either a prefix, or a post fix concatenation term

16.6.2 Value change functions

Three functions are provided to detect changes in values between two adjacent clock ticks: **$rose**, **$fell** and

`$stable`.

```
value_change_functions::=
        $rose ( expression )
      | $fell ( expression )
      | $stable (expression )
```

Syntax 16-6—Value change function syntax

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.

`$rose` returns true if the least significant bit of the expression changed from 0 to 1. Otherwise, it returns false.

`$fell` returns true if the least significant bit of the expression changed from 1 to 0. Otherwise, it returns false.

`$stable` returns true if the value of the expression did not change. Otherwise, it returns false.

Figure 16-4 illustrates two examples of value changes:

— value change expression `e1` is defined as `$rose (req)`

— value change expression `e2` is defined as `$fell (ack)`

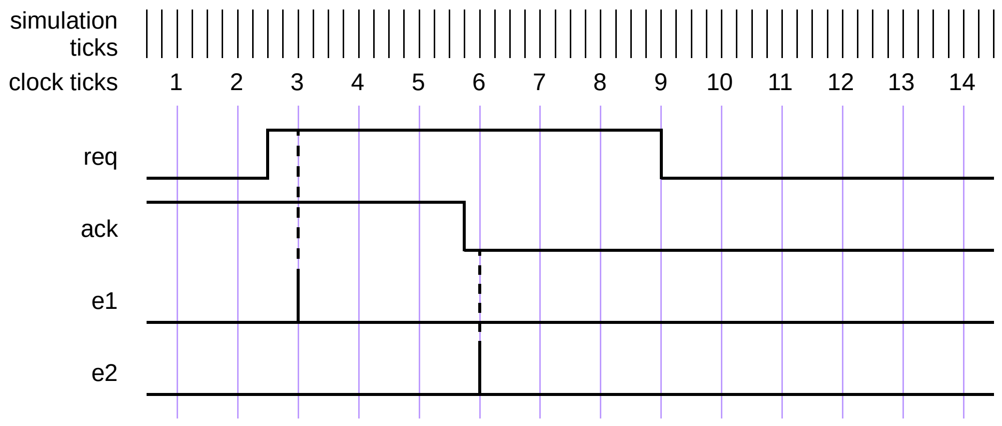


Figure 16-4—Value Change Expressions

The clock used for sampling the events is different than the simulation ticks. Assume, for now, that this clock is defined in this language 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.

16.6.3 AND operation

The binary operator **`and`** is used when both operand expressions are expected to succeed, but the end times of the operand expressions may be different.

```
sequence_expr ::=
        sequence_expr and sequence_expr
```

Syntax 16-7—and operator syntax

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. 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.

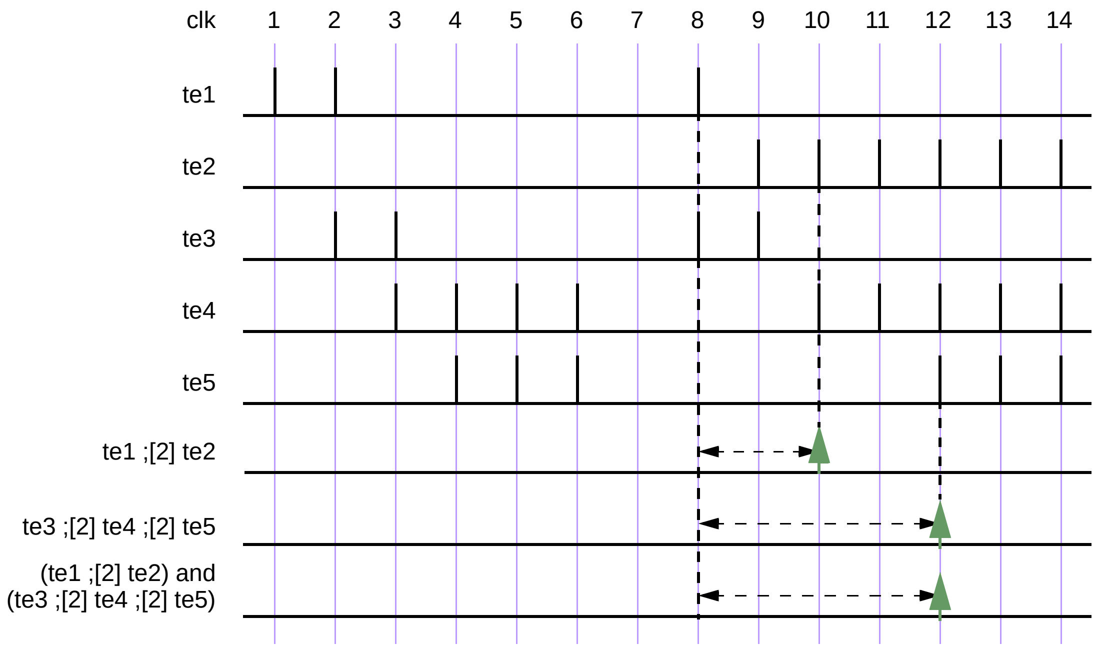
— The end time is the end time of either `te1` or `te2`, whichever terminates last.

First, let us consider the case when both operands are single sequence evaluations.

An example is illustrated in Figure 16-5. Consider the following expression with operator **and** where the two operands are sequences.

```
(te1 ;[2] te2) and (te3 ;[2] te4 ;[2] te5)
```

Figure 16-5—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 16-5 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.

Now, consider an example where 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, following steps are taken:

1) The first operand sequence starts five sequences of evaluation.

2) The second operand sequence has only one possibility of match, so only one sequence is started.

3) Figure 16-6 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 ticks 12, and the fifth ends at clock tick 13. Figure 16-6 shows the two unique successes at clock ticks 12 and 13.

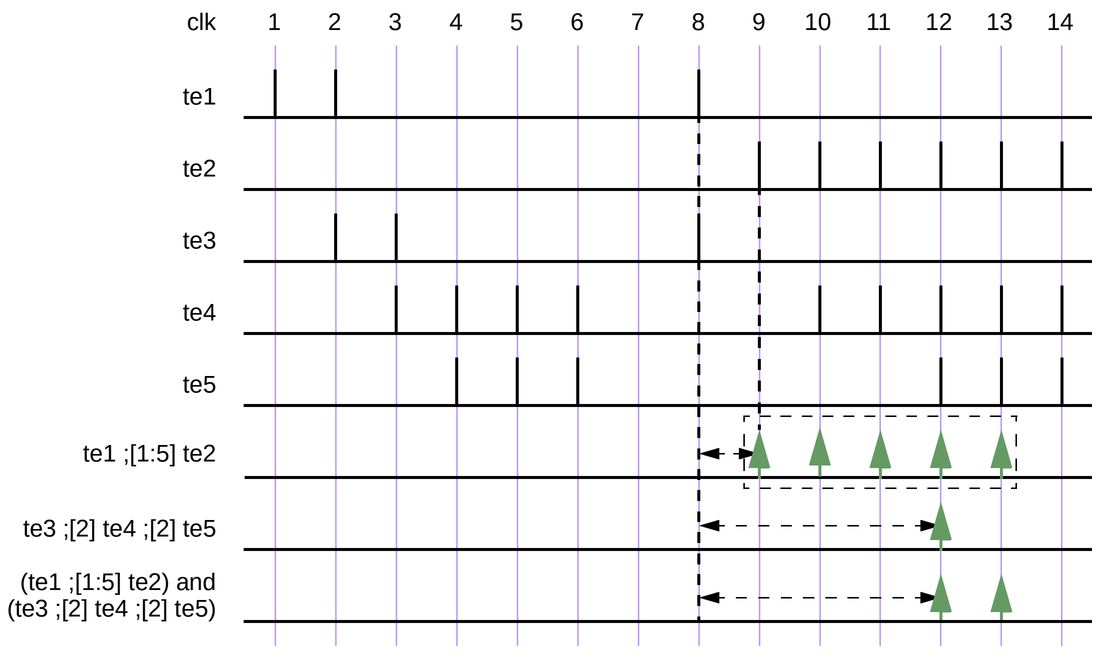


Figure 16-6—ANDing (and) Two Sequences Including a Time Range

If `te1` and `te2` are sampled booleans (not sequences), the expression succeeds if `te1` and `te2` are both evaluated to be true.

An example is illustrated in Figure 16-7 to show the results for attempt at every clock tick. The expression matches at clock tick 1, 3 and 8 because both `te1` and `te2` are simultaneously true. At all other clock ticks, the **and** operation fails because either `te1` or `te2` is false.

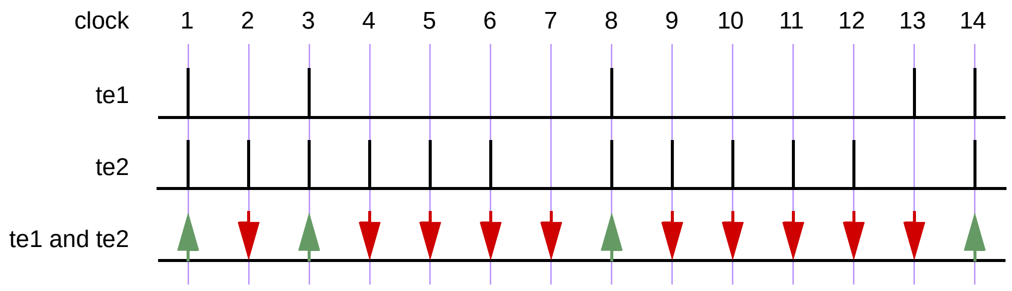


Figure 16-7—ANDing (and) Two Boolean Expressions

16.6.4 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 ::=
        sequence_expr intersect sequence_expr
```

Notice the equence is corrected to sequence

Syntax 16-8—intersect operator syntax

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.

16.6.5 OR operation

The operator **`or`** is used when at least one of the two operand sequences is expected to match.

```
sequence_expr ::=
        sequence_expr or sequence_expr
```

Syntax 16-9—or operator syntax

The two operands of **`or`** are sequence expressions.

Let us consider these operand expressions as values, events and sequences separately to illustrate the details of **`or`** operations. For the expression

```
    te1 or te2
```

when the operand expressions `te1` and `te2` are events or values, the expression matches whenever at least one of two operands `te1` and `te2` is evaluated to true.

Figure 16-8 illustrates **`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.

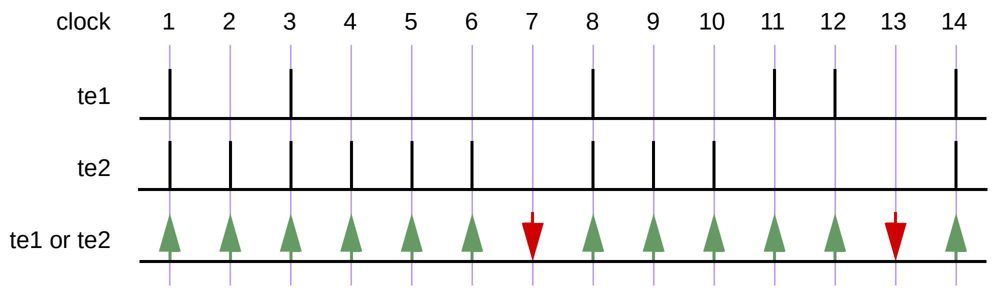


Figure 16-8—ORing (or) Two Sequences

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.

An example is illustrated in Figure 16-9. Consider an expression with **`or`** operator where the two operands are sequences.

```
(te1 ;[2] te2) or (te3 ;[2] te4 ;[2] te5)
```

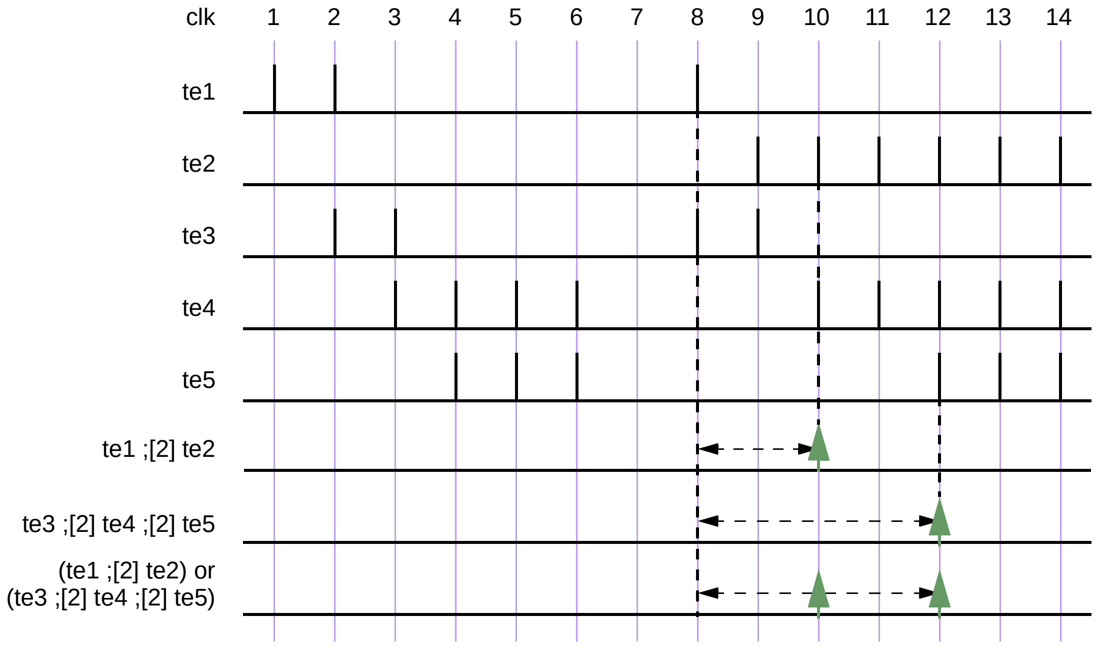


Figure 16-9—ORing (or) Two Sequences

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 16-9, 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.

Consider an example where an operand sequence is associated with 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 16-10, 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 ticks 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.

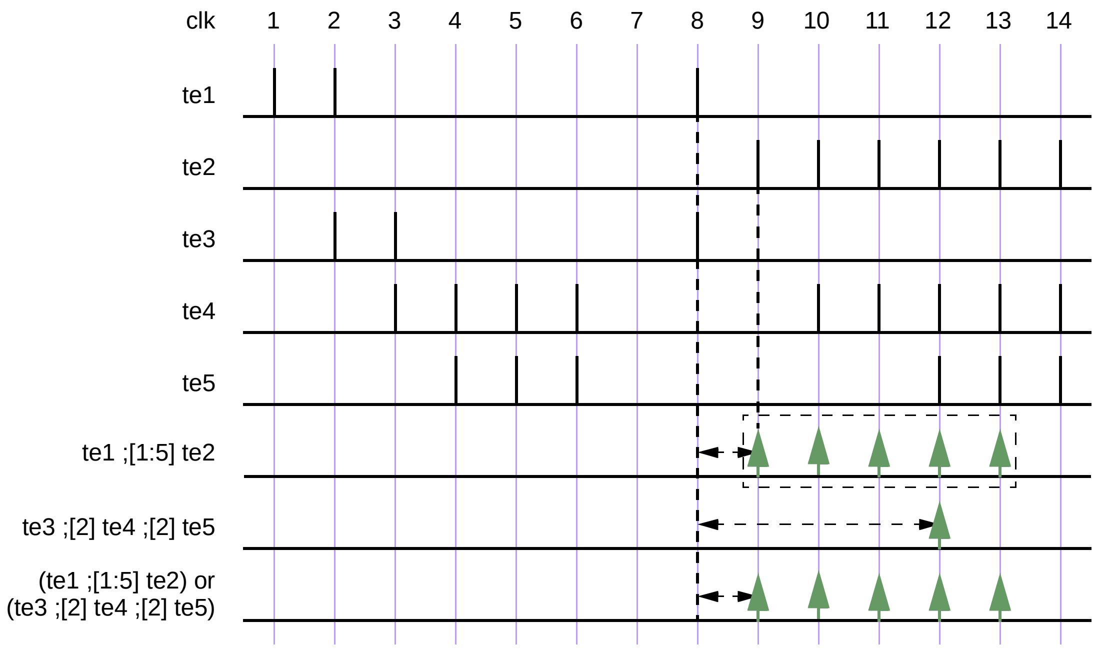


Figure 16-10—ORing (or) Two Sequences Including a Time Range

16.6.6 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 you to discard all subsequent matches 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 ::=
        first_match ( sequence_expr )
```

Syntax 16-10—first_match operator syntax

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.

Please note that **`first_match`** applies to each attempt for the sequence individually.

Consider an example with a variable delay specification as shown below.

```
    sequence t1 = (te1 ;[2:5]te2);
    sequence ts1 = (first_match(te1 ;[2:5]te2));
```

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.

16.6.7 Boolean implication (sequences based on boolean condition)

This 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.

Syntax 16-11—if Boolean implication syntax

```
sequence_expr::=
        boolean_expr => sequence_expr
```

This clause is used to precondition monitoring of a sequence expression. The condition `boolean_expr` must be satisfied in order to monitor `sequence_expr`. If the condition `boolean_expr` fails then `sequence_expr` is skipped for monitoring and results in a sequence true of length one. `boolean_expr` is a logical expression that results in true or false, and `sequence_expr` is a sequence expression that can result in one or more matches. *If the expression evaluates to true, then the first element of the sequence_expr is evaluated on the same clock tick.*

If the condition is evaluated to true, then the evaluation of `sequence_expr` is conducted. The sequence matches of `sequence_expr` become the matches of implication.

Consider 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:

```
sequence @(posedge mclk) data_end =
                  ((data_phase) => ((irdy==0)&&($fell(trdy)||$fell( stop))));
```

Each time a data phase completes, a match for `data_end` is recognized. The attempt at clock tick 6 is illustrated in Figure 16-11. The values shown for the signals are the sampled values with respect to the clock. At clock tick 6 `data_end` is matched because `stop` gets asserted while `irdy` is asserted.

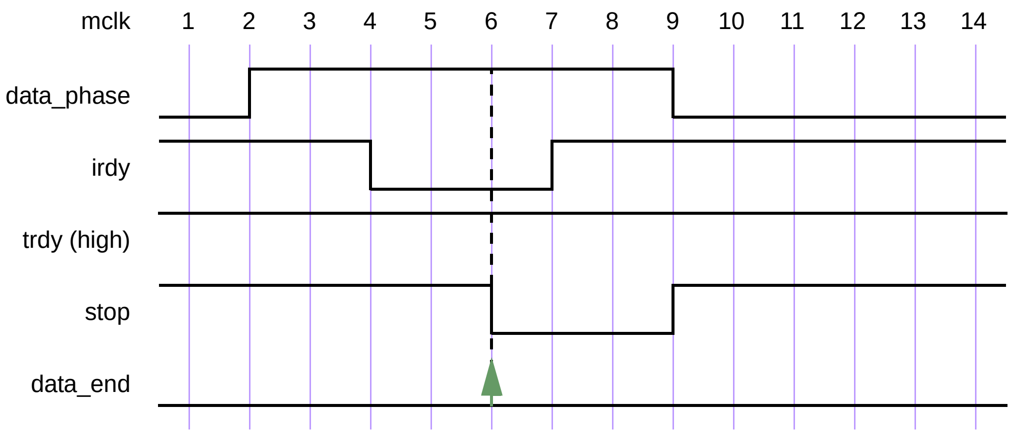


Figure 16-11—Conditional Sequence Matching

`data_end` can be used to ensure that `frame` is de-asserted within 2 clock ticks after `data_end` occurs. Further, it is also required that `irdy` gets de-asserted one clock tick after `frame` gets de-asserted.

A sequence expression is written to express this condition as shown below.

```
‘define data_end  (data_phase &&((irdy==0)&&($fell(trdy)||$fell(stop))))
sequence @(posedge mclk)
   data_end_rule1 =( (‘data_end1) => ([1:2] $rose(frame) ; $rose(irdy)) );
```

`sequence data_end_rule1` first evaluates `data_end` 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 a match with a sequence true of length one.. Otherwise, the following sequence expression is evaluated. The sequence expression:

```
[1:2] $rose(frame) ; $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 16-12. Sequence `data_end` 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.

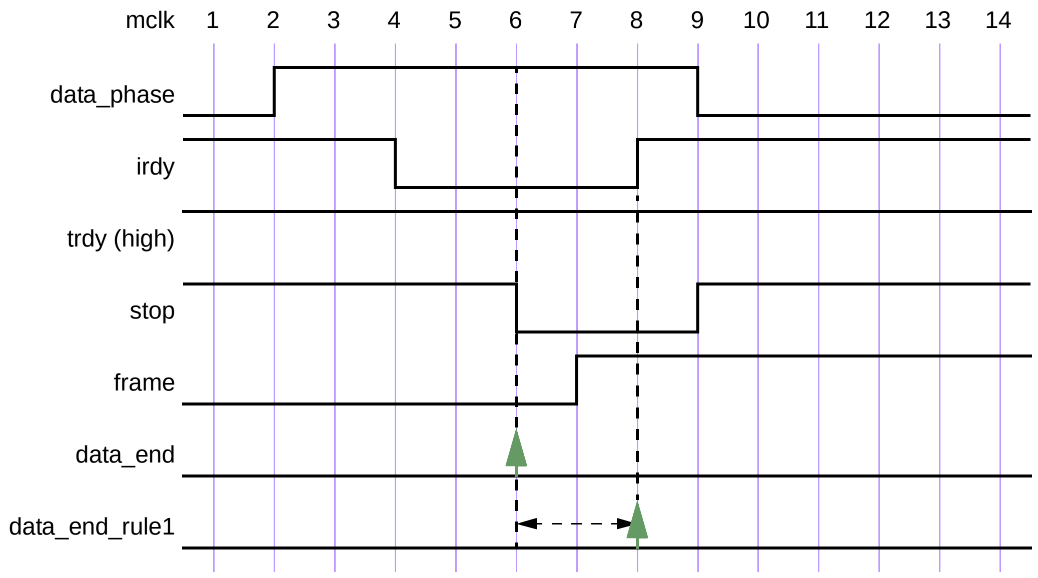


Figure 16-12—Conditional Sequences

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. Let us modify the above 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 16-13.

```
    sequence @(posedge mclk) data_end_rule2 = ( ([1:2] $rose frame) ; $rose irdy );
```

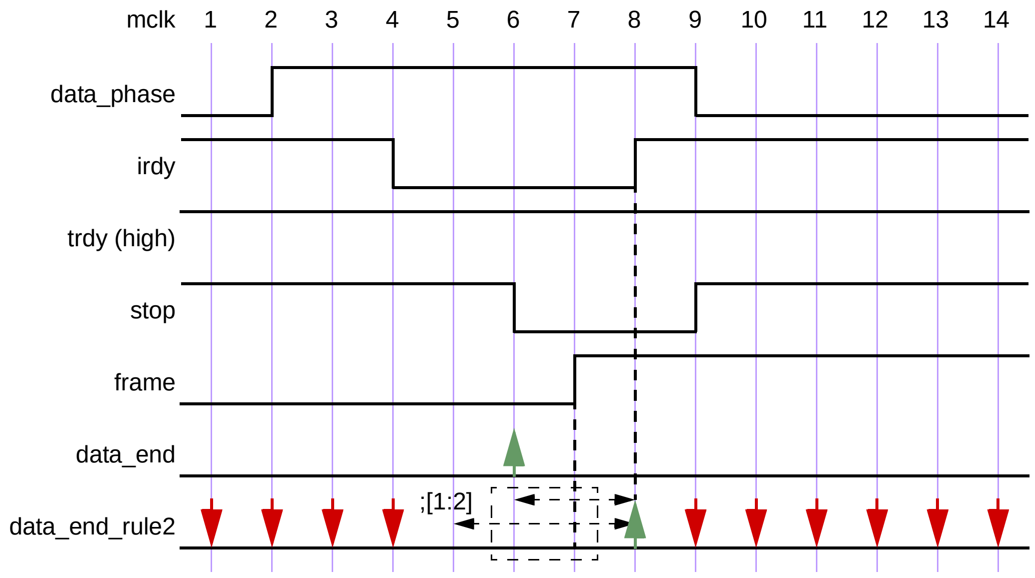


Figure 16-13—Results without the Condition

The sequence 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 evaluation fails and the result for the sequence is a failed match 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.

As one can see from Figure 16-13, removing the precondition of checking event data_end from the assertion causes failures that are not relevant to the verification objective. It becomes important from the validation standpoint to determine these preconditions and use them in the assertion to filter out inappropriate or extraneous situations.

Multi-way conditions are expressed by disjunction, using the **or** operator as illustrated by the example below.

```
sequence s(len) =  ((!trans * [1:inf] ;trans) * [len]);
sequence word_trans =
                   (((lp == BLK1)=> s(BLK1)) or
                    ((lp == BLK2)=> s(BLK2)) or
                    ((lp == BLK3)=> s(BLK3)) or
                   (((lp!=BLK1)||(lp!=BLK2)||(lp!=BLK3))=> s(BLK_DEFAULT)));
```

16.6.8 Sequential implication (sequences based on sequential conditions)

A sequential implication can also be specified using the **=>** clause from the preceding section. The syntax is:

```
sequence_expr::=
        sequence_expr_cond => sequence_expr1
```

Syntax 16-12—Sequential implication syntax

sequence_expr_cond can be any sequence expression.

This feature is useful for chaining sequential implications.

The following points should be noted for sequential implication.

— sequence_expr_cond can result in multiple successful sequences.

— If no sequence succeeds, implication succeeds vacuously by returning a true sequence of length one.

— For each successful match of sequence_expr_cond, sequence_expr1 is separately evaluated, beginning at the end point of the match.That is, the end point of matching sequence from sequence_expr_cond coincides with start point of sequence_expr1

— All matches of sequence_expr_cond must also match sequence_expr1.

For example:

```
(a;b;c) => (d;e)
```

If the sequence (a;b;c) matches then the sequence (d;e) must also match. On the other hand, if the sequence (a;b;c) does not match, then the result is true.

Consider now:

```
(a;[1:3] b;c) => (d;e)
```

In the above example, all matches of (a;[1:3] b; c) must match (d;e). If there are no matches of (a;[1:3] b; c), then there is a vacuous match for the entire expression, resulting in true.

The following example illustrates chaining of sequential implications.

```
sequence next_event(e) = (!e * [1:inf] ; e);
property p16 =
  ((write_en & data_valid)=>
   ((next_event(write_en&&(retire_address[0:4]==addr)))=>
    ([3:8] write_en && !data_valid &&(write_address[0:4]==addr))));
```

16.6.9 Conditions over sequences

Sequences of events often occur under the assumptions of some conditions for correct behavior. A logical condition must hold true, for instance, while processing a transaction. Also frequently, occurrence of certain events is prohibited while processing a transaction. Such situations can be expressed directly using the following construct:

```
sequence_expr::=
          throughout boolean_expr within sequence_expr
```

Notice the equence is corrected to sequence

Syntax 16-13—throughout construct syntax

`boolean_expr` is an expression which must evaluate true at every clock tick while monitoring `sequence_expr`. If a sequence for `sequence_expr` starts at time t1 and ends at time t2, then `expression` must hold true from time t1 to t2. If either the sequence expression does not match or the boolean expression becomes false while the sequence is being evaluated, the composite sequence does not match and a property stated over this composite sequence would declare a failure.

The **`throughout`** construct is an abbreviation for writing:

```
(boolean_expr) *[0:inf] intersect sequence_expr
```

Consider the example illustrated in Figure 16-14. If an additional constraint were placed on the expression as shown below, then the checker `burst_rule` would fail at clock tick 9.

```
sequence @(posedge mclk) burst1 =
     ( (fell burst_mode)=>
       (!burst_mode) throughout ([2] ((trdy==0)&&(irdy==0)) * [7]) );
property burst_rule1 = (burst1) ;
```

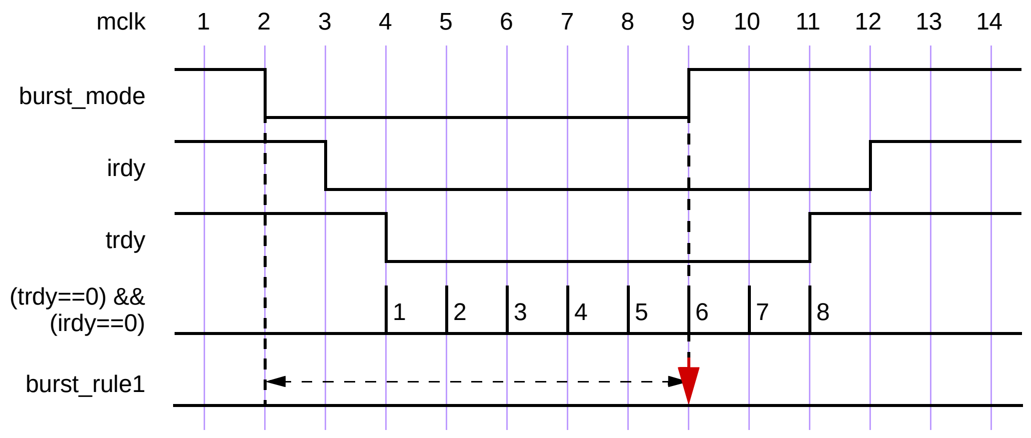


Figure 16-14—Match with throughout-within Restriction Fails

In the above expression, the value of signal `burst_mode` is required to be low during the sequence (from clock tick 2 to 11), 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 11, the expression would result in a match as shown in Figure 16-15.

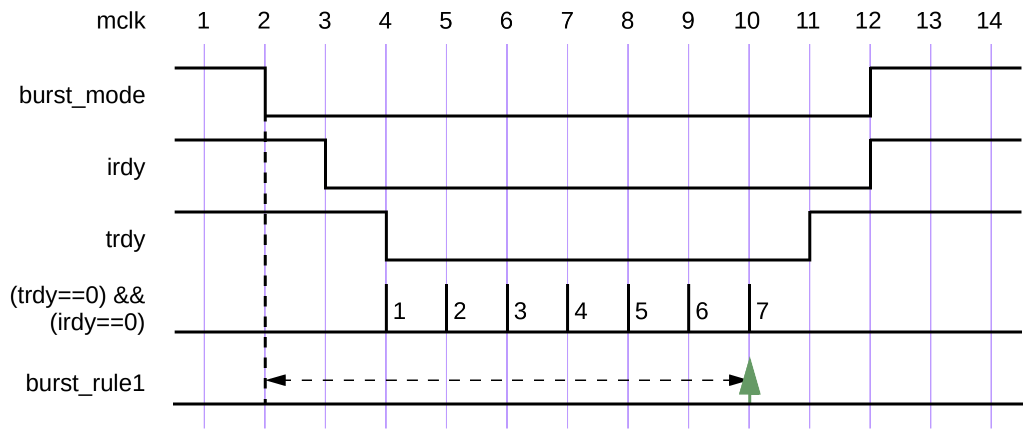


Figure 16-15—Match with throughout-within Restriction Succeeds

16.6.10 Sequence occurrence within another sequence

The containment of a sequence expression within another sequence is expressed as follows:

```
sequence_expr ::=

sequence_expr1 within sequence_expr2
```

Syntax 16-14—Sequence within another sequence syntax

The **within** construct is an abbreviation for writing:

```
(true ;[0:inf](sequence_expr1);[0:inf] true) intersect sequence_expr2
```

The sequence `sequence_expr1` must occur 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`.

16.6.11 Detecting and using endpoint of a sequence

There are two ways in which a complex **sequence** can be decomposed into simpler sub-expressions.

To use **sequence** as a sub-expression, or a part of the expression is by simply referencing its name. The evaluation of a sequence expression that references a **sequence** expression is performed the same way as if the **sequence** expression was a lexical part of the expression. In other words, the *s*equence expression is "invoked" from the expression where it is referenced. An example is shown below:

```
    sequence @(rose sysclk) s = (a;b;c),
        rule = ((trans)=> (start_trans;s;end_trans));
```

This is equivalent to:

```
    sequence @(rose sysclk) s = (a;b;c),
        rule = ((trans)=> (start_trans;a;b;c;end_trans)) ;
```

Any form of syntactic cyclic dependency of the sequence names is disallowed. The example below illustrates dependency of s1 on s2, and s2 on s1, which creates a cyclic dependecncy.

```
    sequence @(rose sysclk) s1 = (x;s2),
                            s2 = (y;s1);
```

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 operator **ended**.

```
boolean_expr_op ::=
        ended seq_name
```

Syntax 16-15—ended operator syntax

ended is a boolean operator. The result of its operation is true or false. When **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 @(posedge sysclk) e1 = ($rose ready;proc1;proc2),
                          rule = ( (reset)=> (inst;ended e1;branch_back));
```

In this example sequence expression e1 must end successfully one clock tick after inst. If the keyword **ended** wasn't there, sequence expression e1 must start one clock tick after inst. Notice that the ended operator only tests for the end point of e1, and has no bearing on the starting point of e1.

16.7 Declaring boolean expressions

Because sequences are composed of boolean expressions, it is useful to allow boolean expressions to be declared as objects of type **`bool`**.

bool_declaration ::=

  bool [range_or_type] named_bool { **,** named_bool } **;**

named_bool ::=

  identifier [**(** identifier { **,** identifier } **)**] **=** boolean_expression

Syntax 16-16—bool type declaration syntax

The boolean object can then be declared as:

```
bool b1(a,b) = a && b && c;
```

and used in a sequence as:

```
(b1(foo,bar);c;d)
(b1(.a(f1),.b(b1));c;d)
```

Note that, in the boolean expression b1, the formal arguments 'a' and 'b' are replaced by the corresponding actual arguments when the bool is instantiated. Any variables referenced within the bool that are not formal arguments get resolved via standard rules from the scope in which the bool is instantiated.

A **bool** expression can be referenced in properties by its name. A hierarchical name can be be used consistent with the System Verilog naming conventions.

boolean_expression is an extension of the System Verilog expression and defined as below.

boolean_expr_op ::=

  expression

  | bool_instance

  | '**true**'

  | '**ended**' seq_name

  | value_change_functions

  | '**$past**' '**(**' expression ['**,**' number_of_ticks] '**)**'

  | '**$countones**' '**(**' expression '**)**'

Syntax 16-17—boolean_expression syntax

This should also be BNF below

boolean_expr::= System Verilog expression where an operand can be System Verilog operand or boolean_expr_op

The **bool** feature differs from other features of System Verilog, such as the macro or function feature. It provides the following capabilites:

— It is a named object.

— It can be instantiated with positional or named parameters.

— It may be defined over multiple physical lines (without requiring line continuation character).

— Boolean expression allows all sequence related boolean operations such as **ended**, **$past** and **$rose.**

16.8 Manipulating data in a sequence

The use of System Verilog 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 we may need a separate variable to store the predicted output of the pipeline for later comparison when the result actually exits the pipe. We can build such a storage 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, we need variables that are local to and are used within a particular transaction check which can span an arbitrary interval of time and may 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 variable creation and destruction can be achieved using the variable declaration at the head of a sequence:

```
sequence_expr ::=
        ( ( {variable_declaration {, variable_declaration}} ) sequence_expr )
variable_declaration ::=
        type identifier = expression
```

Syntax 16-18—variable declaration syntax

The type of name is explicitly specified. The value of the expression is sampled at the time of the beginning of `sequence_expr` and stored in the dynamically created variable `identifier`. Inside `sequence_expr`, the value of the variable remains unchanged for the entire duration of the sequence. Variable `identifier` can be used in `sequence_expr` as any other variable. For every attempt, a new instance of variable `identifier` is created for the `sequence_expr`.

For example, 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.

```
sequence e = ( (valid_in) =>
               ((int x = pipe_in) ([5] (pipe_out1 == (x+1))) );
```

Suppose now that the output of this pipe is chained to another pipe of latency 3 that computes the value as predicted by data and `pipe_val`. The transfer to the second pipe happens only if the result of the first pipe satisfies some Boolean variable `pipe_cont`. We can modify and extend the above example as follows:

```
sequence e_two_pipes =
  ( (valid_in) => ((int x = pipe_in)
                    ([5] ( pipe_out1 == (x+1));
                   (pipe_out1==pipe_cont)) =>
                      ((int y == x+1)
                        ([3](pipe_out2==(y+pipe_val)))))));
```

The nested **variable** declaration uses the variable name `y` which is assigned a value from the enclosing declared variable `x`.

Note that, for practical debugging and verification performance reasons, it may be preferable to verify each of the two pipelines by a separate sequence rather than in one single sequence as in the above example.

If the pipeline supported out-of-order execution in which the outputs can exit with variable latency and in a different order than the data entered, it is a simple matter to add an `id` to each input data and then check that data is correct when the id appears on the output. The first example modified to include the `id` on input and output is as follows:

```
sequence e_with_id = ( (valid_in) =>((int x = pipe_in, int id = id_in)
                              ([1:inf] ( (id_out == id && valid_out)=>
                                           (pipe_out1 == x+1)))));
```

In this example, notice the use of two dynamic variables, `x` and `id`, assigned in the same declaration by separating them by a comma.

16.9 System functions

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] **)**

Syntax 16-19—$past function syntax

The 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 ‘x’.

Another useful function provided for the boolean expression is **`$countones`**, to count the number of 1s in a bit vector expression.

$countones (expression **)**

Syntax 16-20—$countones function syntax

The ‘x’ and ‘z’ value of a bit is not counted towards the number of ones.

16.10 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, a verification directive must be used. A property declaration by itself does not produce any result.

To declare a property, the **`property`** construct is used as shown below.

```
prop_declaration ::=
        property named_prop { , named_prop } ;
named_prop ::=
        identifier [ ( identifier { , identifier } )] = prop_expr
prop_expr ::=
        [initial] [accept ( expression ) ] [never] clocked_sequence
        | identifier [( expression_list )]      // identifier must be a property
clocked_sequence ::=
         [event_control] sequence_expr
```

Syntax 16-21—property construct syntax

A **`property`** declaration is parameterized, like a **`sequence`** and **`bool`** 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 a clocked_sequence for every evaluation attempt is true or false. This is accomplished by implicitly tranforming 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.

The **`accept`** clause allows you to specify asynchronous resets. For a particular attempt, if the accept boolean expression becomes true at any time during the evaluation of the attempt, then the attempt for the property is considered to be a success.

The **`never`** clause states that the expression associated with the property must never evaluate to true. Effectively, it negates the property expression. For each attempt, clocked_sequence results in either true of false, based on whether there is a match for the sequence. The **`never`** clause reverses the result of clocked_sequence. It should be noted that there is no complementation or any form of negation for the sequence itself.

The **`initial`** clause states that the property should only be evaluated on the first clock tick. Thereafter, there should be no evaluation of the property. Without the **`initial`** clause the property is evaluated for every clock tick.

This allows for the following examples:

```
property rule1 = @(posedge clk) ( (a) =>(b;c;d));
property rule2 = (accept = foo) never @(clkev) ((a)=>(b;c;d));
```

A property can be referred to by directives by referencing its name. A hierarchical name can be be used consistent with the System Verilog naming conventions.

A property by default is not evaluated for checking the expression. A verification directive states the verification function to be performed on the property. The directive 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

```
property_directive ::=
        [identifier : ] assert prop_expr  action_block
        | [identifier : ] cover prop_expr  statement_or_null
```

Syntax 16-22—Verification directive syntax

The **`assert`** directive is used to enforce a property as a checker. When the property for the **`assert`** directive 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) = accept(a==2) never @clk (b;c);
env_prop: assert abc(rst,in1,in2) pass_stat else fail_stat;
```

When no action is needed, a null statement (i.e. ;) is specified. The default for else_stat is **$error**. If else_stat is specified, it overrides the default action.

To monitor sequences and other behavioral aspects of the design for coverage, the same syntax is used with the **`cover`** directive. The tools can gather information about the evaluation and report the results at the end of simulation. When the property for the **`cover`** directive is successful, the pass statements may specify a coverage function, such as monitoring all paths for a sequences.

A directive can directly specify an expression, without first declaring it as a property. For example,

```
input_prop: assert accept(in1=2) never @clk (f;g);
cover_item: cover @clk2 (m;n) pass_stat;
```

In the above example, two properties are specified, one with the assert clause and the other with the cover clause.

Please note that a property specification can be just a bool or a sequence.

A directive can be referenced by its optional name. A hierarchical name can be be used consistent with the System Verilog naming conventions. When a name is not provided, a tool shall assign a name to the directive.

16.10.1 Declaring properties outside of procedural code

A property related statement can be used directly within a module as a *module_item* or within *interface* as an *interface_item*.

A property related statement is one of the following:

— bool definition

— sequence definition

— property definition

— property directive

— template instantiation

— template declaration

— property directive

For example:

```
module top(input bit clk);
reg a,b,c;
property rule3 = @(posedge clk) (if(a) (b;c));
...
endmodule
```

`rule3` is a property declared in module top.

16.10.2 Embedding properties in procedural code

A property can also be embedded in a procedural block. A property related statement allowed in a procedural block is one of the following:

- bool definition
- sequence definition
- property definition
- property directive
- template instantiation
- property directive

A property related statement can be declared or instantiated directly in a procedural block as in:

```
always @(posedge clk) begin
   property rule = (a;b;c);
   <statements>;
   <statements>;
   end
```

A procedural property is equivalent to a declarative property in syntax and semantics. Two assumptions are made from the procedural context: clock from the event control of an always block, and the enabling conditions.

A clock for the property related statement is assumed if it is placed in an always block with the event control of the form `@(posedge expr)` or `@(negedge expr)`. In such cases, the event control expression is assumed to be the clock.

For example:

```
always @(posedge mclk)begin
  q <= d1;
  property r1 = (q != d);
end
```

The above property r1 can be written outside the always block with identical semantics as:

```
always @(posedge mclk)begin
  q <= d1;
end
property r1 = @(posedge mclk)(q != d);
```

If the clock is explicitly specified with a property, then it overrides the assumed clock, as shown below:

```
always @(posedge mclk)begin
  q <= d1;
  property r2 = @(posedge sclk)(q != d);
```

```
end
```

In the above example, `(posedge sclk)` is the clock for property r2.

Another possible assumption 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.

```
always @(posedge mclk)begin
  if (a) begin
     q <= d1;
     property r2 = @(posedge sclk)(q != d);
  end
end
```

The above example is equivalent to:

```
always @(posedge mclk)begin
  if (a) begin
     q <= d1;
  end
end
property r2 = @(posedge sclk)(a => (q != d));
```

Similarly, enabling condition is also derived from case statements.

```
always @(posedge mclk)begin
  case (a)begin
     1:begin q <= d1;
             property r2 = @(posedge sclk)(q != d);
       end
     default: q1 <= d1;
  endcase
end
```

The above example is equivalent to:

```
always @(posedge mclk)begin
  case (a)begin
     1:begin q <= d1;
       end
     default: q1 <= d1;
  endcase
  property r2 = @(posedge sclk)(a==1) => (q != d);
end
```

16.11 Grouping assertions as a library

The syntax for library groupings is as follows:

```
template_declaration ::=
          template template_identifier [( template_formal_list )] ;
             { template_item_declaration }
          endtemplate [ : template_identifier ]
template_formal_list ::=
          task_formal_arg { , task_formal_arg }
task_formal_arg ::=
          [data_type] formal_identifier [= boolean_expr | sequence_expr | event_expr | string]
template_item_declaration ::=
          property_decl
          | property_directive
          | seq_decl
          | bool_decl
          | clocking_decl
```

Syntax 16-23—Library groupings syntax

This sub-section describes how to group statements to construct a library of properties and expressions. Such a group is called **`template`** which is given a name and can be instantiated with parameters. When instantiated with parameters, the parameters provide the binding to the actual design objects or other definitions specified elsewhere in the description.

A formal parameter is used to replace a name in the template body. The formal parameter can have an optional specification of type. data_type refers to the System Verilog data types.

The default values for a formal parameter can be specified by using an equal sign with the left-hand side of the equal sign as the formal parameter name and right-hand side as the default value. For example,

```
template hold(exp, min = 0, max = 15, clk);
    sequence @(posedge clk) ova_e_hold = ( past(exp)==exp)*[min:max] );
endtemplate
```

The body of the template may contain:

— **`property`**, **`sequence`** and **`bool`** declarations

— directives

— clock domain declarations

Note: A clock domain declarion using clocking_decl has been described in elsewhere in System Verilog LRM.

Need a cross reference above

A **`template`** is instantiated with the following syntax:

```
template_instantiation ::=
          template_identifier [instance_name] [(list_of_port_connections)];
```

Syntax 16-24—template instantiation syntax

The actual parameters can be given as an ordered list, as a named list. In an ordered list, the parameters are listed in the same order as in the template definition.

For example, the hold template defined above can be instantiated with:

```
hold ordered(counter, 2, 5, rose clk);
```

Or it can be instantiated with:

```
hold named(.exp(counter), .min(2), .max(5), .clk(rose clk));
```

The template instance name is optional. When the name is not specified, the name is the global sequence number of the instance in the form *seq_number*. For example, the first template instance compiled would be assigned the name ti1.

As template instances are expanded, the names of declarations in the template body are constructed by appending the definition name with the template instance name and a dot character. Such an expansion of a name uniquely identifies its definition. The following example illustrates the name expansion of definitions.

```
template range();
   bool c1 = ( enable );
   sequence @(posedge clk2) crange_en = ( ((c1) => (minval <= expr) );
   range_chk: assert (crange_en);
endtemplate
range t1();
range t2();
property term_check = ((t1.c1) => (p_low ; p_end));
```

The definitions `c1`, `crange_en`, and `range_chk` are expanded as shown below.

```
bool t1.c1 = ( enable );
sequence @(rose clk2) t1.crange_en = ( (t1.c1) => (minval <= expr) );
t1_range_chk: assert (t1.crange_en);
bool t2.c1 = ( enable );
@ (rose clk2) sequence t2.crange_en = ( (t2.c1) => (minval <= expr) );
t2.range_chk: assert (t2.crange_en);
property term_chk = ((t1.c1) => (p_low ; p_end;))
```

Using this naming scheme, an expression defined within a template can be referenced outside the template via a standard hierarchical reference.

The actual parameters may not resolve all signals specified within the template. When the template is instantiated, the parameters and the unresolved signals get bound to the design objects in the instantiating scope.

If a formal parameter is specified with a default value in the template definition, then the corresponding actual parameter may be optionally omitted. In the example below, the formal parameter `max` is not supplied when the template is instantiated. The default value of 15 for `max` declared in the template is used.

```
template hold(exp, min = 0, max = 15, clk);
    sequence @(rose clk) e_hold = ( ($past(exp) == exp) * [min:max] );
endtemplate
hold hold_instance(s, 5, , rose clk);
```

If the default parameter value is not declared in the template definition, omission of the corresponding actual parameter value in the template instantiation will result in an error.

16.12 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 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 with XMRs.

— It disallows design code to be attached along with the property.

With this feature, a user can bind a program, where the program contains a group of properties, to a module or an instance.

The syntax of the **bind** construct is:

```
bind_directive ::=
        bind module_instance_name program instantiation ;
module_instance_name ::=
        name of a module or instance
program instantiation ::=
        program_name program_instance_name ( port_arguments )
```

Syntax 16-25—bind construct syntax

A program contains non-design code (either testbench or properties) and executes in the verification phase (The details of the program construct are being discussed in sv-ec committee)

Example of binding to a module:

```
    bind cpu fpu_props fpu_rules_1(a,b,c);
```

— 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 to a specific instance of a module:

```
    bind cpu1 fpu_props fpu_rules_1(a,b,c);
```

— 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 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 System Verilog hierarchical naming conventions.

Section 17 Hierarchy

17.1 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 may 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 18)

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.

17.2 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

BC42-20 — ~~shall~~ can contain procedural statements, which will be executed one time, as if in an **initial** procedure

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();
```

17.3 Module declarations

```
module_declaration ::=          // from Annex A.1.3
              { attribute_instance } module_keyword  module_identifier [ parameter_port_list ]
              [ list_of_ports ] ; [unit] [precision] { module_item }
              endmodule
            | { attribute_instance } module_keyword  module_identifier [ parameter_port_list ]
              [ list_of_port_declarations ] ; [unit] [precision] { non_port_module_item }
              endmodule
module_keyword ::= module | macromodule          // from Annex A.1.3
timeunits_declaration ::=          // from Annex A.1.3
              timeunit time_literal ;
            | timeprecision time_literal ;
            | timeunit time_literal ;
              timeprecision time_literal ;
            | timeprecision time_literal ;
              timeunit time_literal ;
module_or_generate_item_declaration ::=          // from Annex A.1.5
              net_declaration
            | data_declaration
            | event_declaration
            | genvar_declaration
            | task_declaration
            | function_declaration
module_item ::=          // from Annex A.1.5
              port_declaration ;
            | non_port_module_item
non_port_module_item ::=          // from Annex A.1.5
              { attribute_instance } generated_module_instantiation
            | { attribute_instance } local_parameter_declaration
            | { attribute_instance } module_or_generate_item
            | { attribute_instance } parameter_declaration ;
            | { attribute_instance } specify_block
            | { attribute_instance } specparam_declaration
            | module_declaration
module_or_generate_item ::=          // from Annex A.1.5
              { 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_statement
            | { attribute_instance } latch_statement
            | { attribute_instance } ff_statement
            | module_common_item
module_common_item ::=          // from Annex A.1.5
              { attribute_instance } module_or_generate_item_declaration
            | { attribute_instance } interface_instantiation
```

Syntax 17-1—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 may 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
```

17.4 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.

17.5 Port declarations

inout_declaration ::= **inout** [port_type] list_of_port_identifiers // from Annex A.2.1.2

input_declaration ::= **input** [port_type] list_of_port_identifiers // from Annex A.2.1.2

output_declaration ::= // from Annex A.2.1.2
  output [port_type] list_of_port_identifiers
  | output data_type list_of_variable_port_identifiers

interface_port_declaration ::= // from Annex A.2.1.2
  interface list_of_interface_identifiers
  | **interface .** modport_identifier list_of_interface_identifiers
  | **identifier** list_of_interface_identifiers
  | **identifier .** modport_identifier list_of_interface_identifiers

port_type ::= // from Annex A.2.2.1
  data_type { packed_dimension }
  | net_type [signing] { packed_dimension }
  | **trireg** [signing] { packed_dimension }
  | **event**
  | [signing] { packed_dimension } range

signing ::= [**signed**] | [**unsigned**] // from Annex A.2.2.1

Syntax 17-2—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);
   int x;
   char 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 char a, b);
   ...
endmodule
```

A software tool can use the port direction to check against writing to an input port or not writing to an output port.

Ports which are of a net type can have multiple drivers, which are resolved according to the net’s resolution function. A driver can be an **`output`** port of an instantiation, or a continuous assignment.

If the port is of type **`logic`** or any other variable data type, then the port has the value of the last assignment to it. If the port is an **`inout`**, then these assignments can be inside or outside the module. If the port is an **`output`**, then these assignments shall only be inside the module. This provides a way to model a port which is meant to be a single driver.

17.6 Time unit and precision

The time unit can be set by the **`timeunit`** keyword to a time which must be a power of 10 units. For example:

```
timeunit 100ps;
```

The time unit is determined as follows:

1) If a **timeunit** has been specified in the current module, then the time unit is set to module’s time units.

2) Else, if the module definition is nested, then the time unit is inherited from the enclosing module.

3) Else, if a **‘timescale** directive has been specified, then the time unit is set to the units of last **‘timescale** directive.

4) Else, if the **$root** top level has a time unit, then the time unit set to the time units of the root module.

5) Else, the simulator’s default time units are used.

The simulator’s default time units follow the rules of Verilog.

The time precision is set by the **`timeprecision`** keyword to a time which must be a power of 10 units e.g.

```
timeprecision 100fs;
```

If the **`timeprecision`** is not specified, then the precision is determined following the same precedence as with time units.

It is an error to set a precision larger than the current unit.

The **`timeunit`** and **`timeprecision`** keywords shall precede any other item in the top level, module, or interface, because the other items can contain delays and therefore can be dependent on the time unit.

17.7 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 17-3—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
```

17.7.1 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 will be no warnings and the simulation will work as expected.

17.7.2 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 will be reported.

17.7.3 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
```

Implicit .name port connections do not have to be ordered the same as the ports of the instantiated module.

The following rules apply to implicit .name port connections:

— For an implicit .name port connection to be legal, the connecting variable name must match the port name of the instantiated module.

— For an implicit .name port connection to be legal, the connecting variable size must match the port size of the instantiated module.

— For an implicit .name port connection to be legal, the connecting variable data type must be compatible to the port data type of the instantiated module. See section 17.7.5 for a description of compatible data types for implicit port connections.

— Implicit .name port connections cannot be used in the same instantiation with positional port connections.

— Implicit .name port connections may be used in the same instantiation with named port connections.

— Implicit .name port connections cannot be used in the same instantiation with implicit .* port connections.

— The order of the implicit .name port connections does not have to match the port-order of the instantiated module.

— All connecting variables must be explicitly declared, either as a port in the parent module (following the rules of Verilog-2001) or as an explicit net or variable of one or more bits.

17.7.4 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` 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 **.*** 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
```

The following rules apply to implicit **.*** port connections:

— For an implicit **.*** port connection to be legal, all implicitly connected ports must have a connecting variable name to match the port name of the instantiated module.

— For an implicit **.*** port connection to be legal, all implicitly connected ports must have a connecting variable size to match the port size of the instantiated module.

— For an implicit **.*** port connection to be legal, the connecting variable data type must be compatible to the port data type of the instantiated module. See section 17.7.5 for a description of compatible data types for implicit port connections.

— Implicit **.*** port connections cannot be used in the same instantiation with positional port connections.

— Implicit `.*` port connections may be used in the same instantiation with named port connections.

— Implicit `.*` port connections cannot be used in the same instantiation with implicit .name port connections.

— If implicit `.*` port connections are used in an instantiation, all unconnected ports must be shown using named port connections.

— 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.

— All connecting variables must be explicitly declared, either as a port in the parent module (following the rules of Verilog-2001) or as an explicit net or variable of one or more bits.

Modules may 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
```

17.7.5 Compatible data types for implicit port connections

BC42-25 Implicit port connections are permitted between any two data types that are allowed by SystemVerilog port connection rules, as long as the SystemVerilog ~~simulator is not required to report~~ standard does not require a warning about the connection. Any SystemVerilog instantiation that would cause a warning to be issued must be connected by name if other ports of the instance are instantiated using an implicit port connection style.

BC42-26 If, for example, a top-level module connects a signal named `net1` of any data type to an instantiated submodule with a port also named `net1` of same data type, SystemVerilog ~~will run this simulation~~ shall run without warning, because the data types are the same across ports. It is legal to make this type of connection using an implicit port connection style.

BC42-27 If, for example, a top-level module connects a signal named `net2` of type **`wire`** to an instantiated submodule with a port also named `net2` of type **`reg`**, ~~Verilog simulator run this simulation without~~ shall not generate a warning, because the data types are compatible across ports. It is legal to make this type of connection using an implicit port connection style.

BC42-28 If, for example, a top-level module connects a signal named `net3` of type **`tri1`** to an instantiated submodule with a port named `net3` of type **`tri0`**, ~~Verilog simulators issue~~ shall result in a warning and the top-level data type (**`tri1`**) is used during simulation, as described in the IEEE Verilog-2001 Standard. It is legal to make this type of connection using named port connections but it shall be a syntax error to make this connection using an implicit port connection style. Any port connection that results in a required warning message shall not be permitted to be instantiated using an implicit port connection style.

A top-level module shall not implicitly connect a signal of any data type to a port by the same name of another data type if connecting the data types is illegal as defined by this SystemVerilog standard.

17.8 Port connection rules

If a port declaration has a variable data type such as **`logic`**, 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 unconnected, it has the initial value corresponding to the data type.

— An **`output`** can be connected to a variable (or a concatenation) of a compatible data type, and has shared variable behavior if multiple outputs are connected (last write wins); An **`output logic`** can be connected to a net (to provide a resolution function in the case of multiple drivers).

— An **`inout`** can be connected to a variable (or a concatenation) of the same data type.

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 unconnected, it has the value `'z`.

— An **`output`** can be connected to a net type (or a concatenation of net types) or left unconnected, but not to a **`logic`** variable.

— An **`inout`** can be connected to a net type (or a concatenation of net types) or left unconnected, but not to a **`logic`** variable.

Note that where the data types differ between the port declaration and connection, an initial value change event may be caused at time zero.

If a port declaration has a generic **`interface`** type, then it can be connected to an interface of any type. If a port declaration has a named interface type, then it must be connected to a generic interface or an interface of the same type.

A mismatch between vector width across a port connection is resolved as follows:

— If the port is a net vector, then the Verilog connection rules for nets are followed.

— If the port is an **`inout`** port variable, then a port connection must have the same size and representation on both sides of the port. It shall be an error if there is a mismatch.

— If the port is an **`input`** or an **`output`** variable, then the Verilog assignment rules are followed.

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 the array.

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.

17.9 Name spaces

There is one name space hierarchy in SystemVerilog. A type name may be not be the same as an instance name. Type names include modules, interfaces, and data types. Instance names include tasks, functions, procedures, variables, constants and labels as well as module and interface instances.

Pre-defined (built-in) names begin with **`$`**. For example **`$root`** is the name of the top level of the hierarchy.

Names are initially global. A new scope is defined by:

— a module or interface

— a task or function

— a sequential or parallel block

— a structure or union

Tasks and function definitions cannot be nested within themselves, but can be defined in modules or interfaces. The declaration in the closest enclosing scope is matched.

17.10 Hierarchical names

Hierarchical names are also called nested identifiers. They consist of instance names separated by periods, where an instance name may 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 18 Interfaces

18.1 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.

BC42-29 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 ~~modelling~~ modeling and test bench 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.

18.2 Interface syntax

modport_declaration ::= modport list_of_modport_identifiers **;** // from Annex A.2.9

list_of_modport_identifiers ::= modport_item { **,** modport_item }

modport_item ::= modport_identifier **(** modport_port { **,** modport_port } **)**

modport_port ::= // from Annex A.2.9
  input [port_type] port_identifier
  | **output** [port_type] port_identifier
  | **inout** [port_type] port_identifier
  | interface_identifier **.** port_identifier
  | import_export **task** named_task_proto
  | import_export **function** named_fn_proto
  | import_export task_or_function_identifier { **,** task_or_function_identifier }

import_export ::= **import** | **export**

interface_port_declaration ::= // from Annex A.2.1.2
  interface list_of_interface_identifiers
  | **interface .** modport_identifier list_of_interface_identifiers
  | **identifier** list_of_interface_identifiers
  | **identifier .** modport_identifier list_of_interface_identifiers

interface_or_generate_item ::= // from Annex A.1.6
  { attribute_instance } continuous_assign
  | { attribute_instance } initial_construct
  | { attribute_instance } always_construct
  | { attribute_instance } combinational_statement
  | { attribute_instance } latch_statement
  | { attribute_instance } ff_statement
  | { attribute_instance } local_parameter_declaration
  | { attribute_instance } parameter_declaration **;**
  | module_common_item
  | { attribute_instance } modport_declaration

interface_item ::= // from Annex A.1.6
  port_declaration
  | non_port_interface_item

non_port_interface_item ::= // from Annex A.1.6
  { attribute_instance } generated_interface_instantiation
  | { attribute_instance } local_parameter_declaration
  | { attribute_instance } parameter_declaration **;**
  | { attribute_instance } specparam_declaration
  | interface_or_generate_item
  | interface_declaration

interface_instantiation ::= // from Annex A.4.1.2
  interface_identifier [parameter_value_assignment] module_instance { **,** module_instance } **;**

Syntax 18-1—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).

An interface is declared as follows:

```
interface <identifier>; <interface_items> endinterface [: <name> <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.

18.2.1 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    logic[7:0] data,
               output   bit gnt,
                        bit rdy );
   logic avail;

   ...
endmodule

module cpuMod(
   input    bit clk,
            bit gnt,
            bit rdy,
   inout    logic [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, data;

   memMod mem(req, clk, start, mode, addr, data, gnt, rdy);
   cpuMod cpu(clk, gnt, rdy, data, req, start, addr, mode);

endmodule
```

18.2.2 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 **inout** ports. 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
```

BC19-2

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
```

BC19-2

18.2.3 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. 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
```

BC19-2

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
```

BC19-2

18.3 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.

18.4 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();                                   BC22-2

   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();                                   BC22-2

   m u1(.i(i.master));
   s u2(.i(i.master));
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();                          BC22-2
   modport master2 (ch1.master, ch2.master);
endinterface
```

Note that if no **modport** is specified in the module header or in the port connection, then all the wires and variables in the interface are accessible with direction **inout**, as in the examples above.

18.4.1 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,
                  inout data);

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout 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
```

18.4.2 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,
                  inout data);

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout 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
```

18.4.3 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,
                  inout data);

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout 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
```

18.5 Tasks and functions in interfaces

Tasks and functions may be defined within an interface, or they may 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 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) may 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.

18.5.1 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.slave); // only has access to the slaveRead task
   cpuMod cpu(sb_intf.master); // only has access to the masterRead task
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.

18.5.2 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,
               inout data,
               import task slaveRead(),
                      task slaveWrite());
            // import into module that uses the modport

   modport master(input gnt, rdy, clk,
               output req, addr, mode, start,
               inout data,
               import task masterRead(input logic[7:0] raddr),
                      task masterWrite(input logic[7:0] waddr));
            // import requires the full task prototype

   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
      b.gnt <= b.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 slaveRead task
   cpuMod cpu(sb_intf.master); // only has access to the masterRead task
   omniMod omni(sb_intf); // has access to all master and slave tasks
endmodule
```

18.5.3 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,
                     inout data,
                     export task Read(),
                            task Write());
              // export from module that uses the modport

      modport master(input gnt, rdy, clk,
                     output req, addr, mode, start,
                     inout 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
```

18.5.4 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 may 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. Normally, only one module responds to the task call, e.g. the one containing the appropriate address. Only then should the task write to the result variables. Note multiple export of functions is not allowed, because they must always write to the result.

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;
   // 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,
                  inout data,
                  export task Read(),
                         task Write());
         // export from module that uses the modport

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout 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(interface a); // Uses just the interface keyword
   logic avail;

   task a.countSlaves;
      a.slaves++;
   endtask

   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 mem1(sb_intf.slave); //exports the countSlaves, Read and Write tasks
   memMod mem2(sb_intf.slave); //exports the countSlaves, Read and Write tasks
   cpuMod cpu(sb_intf.master); //imports the Read and Write tasks
endmodule
```

18.6 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,
                     inout data,
                     import task slaveRead(),
                            task slaveWrite());
            // import into module that uses the modport

      modport master(input gnt, rdy, clk,
                     output req, addr, mode, start,
                     inout 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
```

18.7 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 may 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
```

BC19-2

Section 19 Parameters

19.1 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

19.1.1 Defparam removal

The **`defparam`** statement may be removed from future versions of the language. See section 24.2.

19.2 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 ;
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 }
param_assignment ::= parameter_identifier = constant_param_expression          // from Annex A.2.4
specparam_assignment ::=
        specparam_identifier = constant_mintypmax_expression
      | pulse_control_specparam
type_assignment ::= type_identifier = data_type
```

Syntax 19-1—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, which is 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
```

EC-CH25

Section 20 Random Constraints

Editor’s Note: This entire section is new for draft 2. Only the Section titles have been highlighted as new text.

Editor’s Note: Things left to do in this section:
— Add the correct formatting for keywords and other special fonts.
— Revise wording for proper IEEE style of “shall” and “can”.
— Add notes for where BNF syntax needs to be inserted.
— Change hard-coded cross references to linked cross references.
— Add index tags for the entire chapter

20.1 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. This proposal 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 framework 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.

The next section provides an overview of object-based randomization and constraint programming. The rest of this document provides detailed information on random variables, constraint blocks, and the mechanisms used to manipulate them.

20.2 Overview

This section introduces the basic concepts and uses for generating random stimulus within objects. The proposed language 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] == ‘2b0;}
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 type;
   constraint addr_range
   {
      (type == low ) => addr inside { [0 : 15] };
      (type == mid ) => addr inside { [16 : 127]};
      (type == 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 type 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 type. When a MyBus object is randomized, values for *addr*, *data*, and *type* are computed such that all of the constraints are satisfied. Using inheritance to build layered constraint systems allows uses to develop 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 {type == 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…).

— Constraint expressions follow SystemVerilog syntax and semantics, including precedence, associativity, sign extension, truncation, and wrap-around.

— 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 may fail only when the problem is over-constrained and there is no combination of random values that satisfy the constraints.

— Constraints interact bi-directionally. In this example, the value chosen for addr depends on type and how it is constrained, and the value chosen for type depends on addr and how it is constrained. All expression operators are treated bi-directionally, including the implication operator (=>).

Sometimes it is desirable to disable constraints on random variables. For example, consider the case where we want to deliberately generate an illegal address (non-word aligned):

```
task exercise_illegal(MyBus bus, int cycles);
   int res;

   // Disable word alignment constraint.
   $constraint_mode( OFF, bus.word_align );

   repeat (cycles) begin

   // CASE 1: restrict to small addresses.
   res = bus.randomize() with {addr[0] || addr[1];};
      ...
   end

// Re-enable word alignment constraint.
$constraint_mode( ON, bus.word_align );
endtask
```

The **`$constraint_mode()`** system task 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 a 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 non-random 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();
EC-CH42  ~~printf~~ $display("Before randomize x=%0d, y=%0d~~\n~~", x, y);
   endtask

   function void post_randomize();
      super.post_randomize();
```

```
        ~~printf~~ $display("After randomize x=%0d, y=%0d~~\n~~", x, y);
      endtask
   endclass
```

Editor's Note: "function" is paired with "endtask". Are these tasks or functions?.

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).

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.

20.3 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:

```
rand variable;
```

or

```
randc variable;
```

Editor's Note: Add BNF excerpt , once available.

— The solver can randomize scalar variables of any integral type such as integer, enumerated types, and packed array variables of any size.

— 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. If the array elements are of type object, all of the array elements must be non-**null**. Individual array elements may be constrained, in which case the index expression must be a literal constant.

— The size of a dynamic array declared as **rand** or **randc** may also be constrained**.** In that case, the array will be resized according to the size constraint, and then all the array elements will 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 variable can be declared **rand** in which case all of that object's variables and constraints are solved concurrently with the other class variables and constraints. Objects cannot be declared **randc**.

20.3.1 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 will 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.

20.3.2 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, char,** or enumerated types, and may 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.

$$\begin{tikzcd}
\text{initial permutation:} & 0 \arrow[r] & 3 \arrow[r] & 2 \arrow[r] & 1 \arrow[dlll] \\
\text{next permutation:} & 2 \arrow[r] & 1 \arrow[r] & 3 \arrow[r] & 0 \arrow[dlll] \\
\text{next permutation:} & 2 \arrow[r] & 0 \arrow[r] & 1 \arrow[r] & 3 \ \ldots
\end{tikzcd}$$

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 may impose a limit on the maximum size of a **randc** variable, but it should be no less than 8 bits.

The semantics of cyclical variables requires that they be solved before other random variables. A set of constraints that includes both **rand** and **randc** variables will be solved such that the **randc** variables are solved first, and this may sometimes cause **randomize()** to fail.

20.4 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. They must be defined after the variable declarations in the class, and before the task and function declarations in the class. Constraint block names must be unique within a class.

The syntax to declare a constraint block is:

```
constraint constraint_name { contraint_expressions }
```

Editor’s Note: Add BNF excerpt , once available.

constraint_name is the name of the constraint block. This name can be used to enable or disable a constraint using the system task **$constraint_mode()**.

constraint_expressions 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`.

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 20.12).

— `dist` expressions cannot appear in other expressions (unlike `inside`); they can only be top-level expressions.

20.5 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; }
```

20.6 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 effectively 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.

— The built-in methods **pre_randomize()** and **post_randomize()** are functions and cannot block.

20.7 Set membership

Constraints support integer value sets and set membership operators.

The syntax to define a set expression is:

```
    expression inside { value_range_list };
```

or

```
    expression inside array;   // fixed-size, dynamic, or associative array
```

Editor’s Note: Add BNF excerpt , once available.

expression is any integral SystemVerilog expression.

value_range_list is a comma-separated list of integral expressions and ranges. Ranges are specified with a low and high bound, enclosed by square braces [ ], and separated by a colon: [*low_bound* : *high_bound*]. Ranges include all of the integer elements between the bounds. The bound to the left of the colon MUST be less than or equal to the bound to the right, otherwise 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};}
```

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 is equivalent to `a == b || a == c`.

20.8 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:

```
    expression dist { value_range_ratio_list };
```

Editor’s Note: Add BNF excerpt , once available.

expression can be any integral SystemVerilog expression.

The distribution operator **dist** evaluates to true if 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 value_range_ratio_list is a comma-separated list of integral expressions and ranges (the same as the value_range_list for set membership). 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 may 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 will increase the likelihood of choosing those values.

Limitations:

— A **dist** operation may not be applied to **randc** variables.

— A **dist** expression requires that expression contain at least one **rand** variable.

20.9 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:

```
expression => constraint;
expression => constraint_block;
```

Editor’s Note: Add BNF excerpt , once available.

The *expression* can be any integral SystemVerilog expression.

The implication operator **=>** evaluates to true if the expression is false or the constraint is satisfied; otherwise it evaluates to false.

The *constraint* is any valid constraint, and *constraint_block* represents an anonymous 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* is less than 10 or greater than 100. If *mode* is neither small nor large, the value of *len* is unconstrained.

The boolean equivalent of (a **=>** b) is (!a || b). Implication is a bidirectional operator. Consider 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}. The probability that a == 0 is thus 1/(256-15) or 1/241.

It is important to that the constraint solver be designed to cover the whole random value space with uniform probability. This allows randomization to better explore the whole design space than in cases where certain value combinations are preferred over others.

20.10 if-else constraints

If-else style constraint are also supported.

The syntax to define an **if-else** constraint is:

```
if (expression) constraint; [else constraint;]
if (expression) constraint_block [else constraint_block]
```

Editor’s Note: Add BNF excerpt , once available.

expression can be any integral SystemVerilog expression.

constraint is any valid constraint. If the expression is true, the first constraint must be satisfied; otherwise the optional else-constraint must be satisfied.

constraint_block represents an anonymous constraint block. If the *expression* is true, all of the constraints in the first constraint block must be satisfied, otherwise all of the constraints in the optional *else-constraint-block* must be satisfied. Constraint blocks may 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 bi-directional. In the declaration above, the value of *mode* constraints the value of *len*, and the value of *len* constrains the value of *mode*.

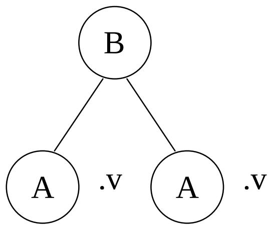
20.11 Global constraints

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 sub-tree, and a right sub-tree. Both sub-trees 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 sub-tree 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 may be leaf nodes or more heap-nodes.

The following rules determines 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**). 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.

20.12 Variable ordering

The solver assures that the random values are selected to give a uniform value distribution over legal value combinations (that is, all combinations of values have the same probability of being chosen). This important property guarantees that all value combinations are equally probable.

Sometimes, however, it is desirable to force certain combinations to occur more frequently. Consider this 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 probability that *s* is true is $1/2^{32}$, which is practically zero.

The constraints provide a mechanism for order variables so that *s* can be chosen independent 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 will occur 50% of the time, and d != 0 will 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:

```
solve variable_list before variable_list ;
```

Editor’s Note: Add BNF excerpt , once available.

variable_list is a comma-separated list of integral scalar 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 scalar values.

— A constraint block may 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 will be solved with the last set of ordered variables. These values are deferred until as late as possible to assure a good distribution of value.

— 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.

20.13 Randomization methods

20.13.1 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 will randomize an instance of class SimpleSum:

```
SimpleSum p = new;
int success = p.randomize();
if (success == 1 ) ...
```

Checking results is always needed because the actual value of state variables or addition of constraints in derived classes may render seemingly simple constraints unsatisfiable.

20.13.2 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.

Built-in definition for **pre_randomize()**:

```
function void pre_randomize;
   if (super) super.pre_randomize();
      // Optional programming before randomization goes here.
endfunction
```

Built-in definition for **post_randomize()**:

```
function void post_randomize;
   if (super) super.post_randomize();
      // 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 recursively 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 recursively calls

super.post_randomize().

Users may overload the **pre_randomize()** in any class to perform initialization and set pre-conditions before the object is randomized.

Users may 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 will be skipped.

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 be called.

— The **randomize()** method may not be overloaded.

— The **randomize()** method implements object random stability. An object can be seeded by the **$srandom()** system call, specifying the object in the second argument.

20.14 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:

```
result = object_name.randomize() with constraint_block;
```

Editor's Note: Add BNF excerpt , once available.

object_name is the name of an instantiated object.

The anonymous *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 may also reference local variables and task and function parameters, 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 parameters, class variables, variables in the enclosing scope. The **randomize() with** class is brought into scope at the innermost nesting level.

For example, see below, where 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 parameter in the doit() task. y is a member of Bar. z is a local parameter.

20.15 Disabling random variables

The **$rand_mode()** system task 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.

20.15.1 $rand_mode()

The syntax for the **$rand_mode()** subroutine is:

```
task $rand_mode( ON | OFF, object [.random_variable] );
```

or

```
function int $rand_mode( object.random_variable );
```

Editor’s Note: Add BNF excerpt , once available.

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, the action is applied to all random variables within the specified object.

The first argument to the **$rand_mode** task determines the operation to be performed:

Table 20-1: $rand_mode first argument

Constant	Value	Description
OFF	0	Sets the specified variables to inactive so that they are not randomized on subsequent calls to the **randomize()** method.

Table 20-1: $rand_mode first argument

Constant	Value	Description
ON	1	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 variable is an object, only the mode of the variable is changed, not the mode of random variables within that object (see Global Constraints in Section 20.11).

A compiler error is issued if the specified variable does not exist within the class hierarchy or it exists but is not declared as **rand** or **randc.**

The function form of **$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 scalar variables, thus, if the specified variable is an 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.
$rand_mode(OFF, packet_a);

// ... other code
// Enable source_value.
$rand_mode(ON, packet_a.source_value );

ret = $rand_mode( packet_a.dest_value );
```

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*.

20.16 Disabling constraints

The **$constaint_mode()** system task 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.

20.16.1 $constraint_mode()

The syntax for the **$constraint_mode()** subroutine is:

```
task $constraint_mode( ON | OFF, object [.constraint_name] );
```

or

```
function int $constraint_mode( object. constraint_name );
```

Editor's Note: Add BNF excerpt , once available.

object is any expression that yields the object handle in which the constraint is defined.

constraint_name 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, the operation is applied to all constraints within the specified object.

The first argument to the **$constraint_mode** task determines the operation to be performed:

Table 20-2: $constraint_mode first argument

Constant	Value	Description
OFF	0	Sets the specified constraint block to active so that it is considered by subsequent calls to the **randomize()** method.
ON	1	Sets the specified constraint block to inactive so that it is not enforced on subsequent calls to the **randomize()** method.

A compiler error is issued if the specified constraint block does not exist within the class hierarchy**.**

The function form of **$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( $constraint_mode( p.filter1 ) )
      $constraint_mode( OFF, p.filter1 );
   else
      $constraint_mode( ON, p.filter1 );

   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 will deactivate it; if it's inactive the function will activate it. Finally, the function calls the randomize method to generate a new random value for variable *source_value*.

20.17 Static constraint blocks

Constraint blocks can be defined as static by including the **static** keyword in their definition.

The syntax to declare a static constraint block is:

```
static constraint constraint_name { contraint_expressions }
```

Editor's Note: Add BNF excerpt , once available.

If a constraint block is declared as **static**, then calls to **$constraint_mode()** 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.

20.18 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()** system task. 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()** system task. 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.

20.19 Random number system functions

20.19.1 $urandom

The system function **$urandom** provides a mechanism for generating random numbers. The function returns a new 32-bit random number each time it is called. The number is unsigned.

The syntax for **$urandom** is:

```
function unsigned int $urandom [ (int seed ) ] ;
```

Editor's Note: Add BNF excerpt , once available.

The *seed* is an optional argument that determines which random number is generated. The seed can be any integral expression. The random number generator generates the same number 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 non-deterministic 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 it's automatically thread stable (see Section 20.20.2).

20.19.2 $urandom_range()

The **$urandom_range()** function returns an unsigned integer within a specified range.

The syntax for **$urandom_range** is:

```
function unsigned int $urandom_range(  unsigned int maxval,
                                        unsigned int minval = 0 );
```

Editor’s Note: Add BNF excerpt , once available.

The function returns an unsigned integer in the range *maxval* .. *minval*.

Example: `val = $urandom_range(7,0);`

If *minval* is omitted, the function returns 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 three previous examples produce a value in the range of 0 to 7, inclusive.

$urandom_range() is automatically thread stable (see Section 20.20.2).

20.19.3 $srandom()

The system function **$srandom()** allows manually seeding the RNG of objects or threads.

The syntax for the **$srandom()** system task is:

```
task $srandom( int seed, [object obj] );
```

Editor’s Note: Add BNF excerpt , once available.

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.

20.20 Random stability

The Random Number Generator (RNG) is localized to threads and objects. Because the stream 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()**.

Test-benches 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.

20.20.1 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 is 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 sub-tree) completely with a single seed at the root thread of the system.

20.20.2 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 preserve their behavior by manually seeding their root thread.

20.20.3 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();
~~begin~~ end
```

Editor's Note: I took the liberty of changing the final "begin" to "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 void 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 may be set from any thread. However, a thread's seed can only be set from within the thread itself.

20.21 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:

internally:

```
class Packet;
   rand bit[15:0] header;
   ...
   function void new (int seed);
      $srandom(seed, this);
      ...
   ~~endtask~~ endfunction
endclass
```

Editor's Note: I took the liberty of changing the final "endtask" to "endfunction"

or externally:

```
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 randomized.

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.

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)` will return 32, even if a software tool uses more than 32-bits of storage to represent the 4-state values.

22.3 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 scalar 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.

22.4 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.

22.5 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

22.6 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 low.

— **`$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.

Section 23 Compiler Directives

23.1 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 strings as macro arguments

23.2 ‘define macros

In SystemVerilog, the **`define** macro text can include a backslash (\) at the end of a line to show continuation on the next line.

The macro text can also include an isolated quote, which must be preceded by a back tick, **`"**. This allows macro arguments to be included in strings. If the strings are to contain **\\"**, the macro text should be written **`\\`"**. Otherwise, the backslash will be treated as the start of an escaped identifier.

The macro text can also include a double back tick, **``**, to allow identifiers to be constructed from arguments, e.g.

```
`define foo(f) f``_suffix
```

This expands:

```
foo(bar)
```

to:

```
bar_suffix
```

Note that there must be no space before the parenthesis. Otherwise, it is treated as macro text.

The ‘**include** directive can be followed by a macro, instead of a literal string:

```
`define f1 "/home/foo/myfile"
`include `f1
```

Section 24
Features under consideration for removal from SystemVerilog

24.1 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`**.

24.2 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 may 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 19 for more details on parameters.

24.3 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 may 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.

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 may 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 ::=
BC19-3    **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

BC19-5 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

module_declaration ::=
{ attribute_instance } module_keyword module_identifier [parameter_port_list]
BC19-6 ~~{~~ list_of_ports ~~}~~ **;** [timeunits_declaration] { module_item }
endmodule
| { attribute_instance } module_keyword module_identifier [parameter_port_list]
[list_of_port_declarations] **;** [timeunits_declaration] { non_port_module_item }
endmodule

module_keyword ::= **module** | **macromodule**

interface_declaration ::=
{ attribute_instance } **interface** interface_identifier [parameter_port_list]
BC19-7 ~~{~~ list_of_ports ~~}~~ **;** [timeunits_declaration] { interface_item }
endinterface [**:** interface_identifier]
| { attribute_instance } **interface** interface_identifier [parameter_port_list]
[list_of_port_declarations] **;** [timeunits_declaration] { non_port_interface_item }
endinterface [**:** interface_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 } **)**
| **( )**

port ::=
[port_expression]
| **.** port_identifier **(** [port_expression] **)**

port_expression ::=
port_reference
BC19-8 | **{** port_reference { **,** port_reference } **}**

port_reference ::=
port_identifier
| port_identifier **[** constant_expression **]**
| port_identifier **[** range_expression **]**

port_declaration ::=
{ attribute_instance } inout_declaration
| { attribute_instance } input_declaration
| { attribute_instance } output_declaration
| { attribute_instance } interface_port_declaration

A.1.5 Module items

module_common_item ::=
{ attribute_instance } module_or_generate_item_declaration
| { attribute_instance } interface_instantiation

module_item ::=
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_statement
| { attribute_instance } latch_statement
| { attribute_instance } ff_statement
| module_common_item

module_root_item ::=
{ attribute_instance } module_instantiation
| { attribute_instance } local_parameter_declaration
| interface_declaration
| module_common_item

module_or_generate_item_declaration ::=
net_declaration
| data_declaration
| event_declaration
| genvar_declaration
| task_declaration
| function_declaration

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
| module_declaration

parameter_override ::= **defparam** list_of_param_assignments **;**

A.1.6 Interface items

interface_or_generate_item ::=
{ attribute_instance } continuous_assign
| { attribute_instance } initial_construct
| { attribute_instance } always_construct
| { attribute_instance } combinational_statement
| { attribute_instance } latch_statement
| { attribute_instance } ff_statement
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**
| module_common_item

| { attribute_instance } modport_declaration

interface_item ::=

BC19-9 port_declaration ;

| non_port_interface_item

non_port_interface_item ::=

{ attribute_instance } generated_interface_instantiation

BC19-10 ~~| { attribute_instance } local_parameter_declaration~~

~~| { attribute_instance } parameter_declaration ;~~

| { attribute_instance } specparam_declaration

| interface_or_generate_item

| interface_declaration

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

input_declaration ::= **input** [port_type] list_of_port_identifiers

output_declaration ::=

output [port_type] list_of_port_identifiers

| output data_type list_of_variable_port_identifiers

interface_port_declaration ::=

interface list_of_interface_identifiers

| **interface .** modport_identifier list_of_interface_identifiers

BC19-11/ BC27 | identifier list_of_interface_identifiers

| identifier **.** 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

event_declaration ::= **event** list_of_event_identifiers **;**

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 **;**
BC19-12 | **typedef** interface_identifier { [constant_expression] } **.** type_identifier
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

data_type ::=
integer_vector_type [signing] { packed_dimension } [range]
| integer_atom_type [signing] { packed_dimension }
| type_declaration_identifier
| non_integer_type
| **struct** [**packed**] [signing] **{** { struct_union_member } **}**
| **union** [**packed**] [signing] **{** { struct_union_member } **}**
| **enum** [integer_type [signing] { packed_dimension }]
{ enum_identifier [**=** constant_expression] { **,** enum_identifier [**=** constant_expression] } **}**
| **void**

integer_type ::= integer_vector_type | integer_atom_type

integer_atom_type ::= **byte** | **char** | **shortint** | **int** | **longint** | **integer**

integer_vector_type ::= **bit** | **logic** | **reg**

non_integer_type ::= **time** | **shortreal** | **real** | **realtime** | $built-in

net_type ::= **supply0** | **supply1** | **tri** | **triand** | **trior** | **tri0** | **tri1** | **wire** | **wand** | **wor**

port_type ::=
BC19-13 data_type ~~{ packed_dimension }~~
| net_type [signing] { packed_dimension }
| **trireg** [signing] { packed_dimension }
| **event**
| [signing] { packed_dimension } range

BC19-14

signing ::= ~~{~~ **signed** ~~}~~ | ~~{~~ **unsigned** ~~}~~

simple_type_or_number ::= simple_type | number

simple_type ::= integer_type | non_integer_type | type_identifier

struct_union_member ::= 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

BC19-15

delay3 ::= # delay_value | # (~~delay_value~~ mintypmax_expression [**,** ~~delay_value~~ mintypmax_expression [**,**
  ~~delay_value~~ mintypmax_expression]] **)**

delay2 ::= # delay_value | # **(** ~~delay_value~~ mintypmax_expression [**,** ~~delay_value~~ mintypmax_expression] **)**

delay_value ::=
  unsigned_number
  ~~| parameter_identifier~~
  ~~| specparam_identifier~~
  ~~| mintypmax_expression~~
  | real_number
  | identifier

A.2.3 Declaration lists

BC19-16

list_of_event_identifiers ::= event_identifier ~~[unpacked_dimension~~ { unpacked_dimension }~~]~~
  { **,** event_identifier ~~[unpacked_dimension~~ { unpacked_dimension }~~]~~ }

list_of_genvar_identifiers ::= genvar_identifier { **,** genvar_identifier }

list_of_interface_identifiers ::= interface_identifier { unpacked_dimension }
  { **,** interface_identifier { unpacked_dimension } }

list_of_net_decl_assignments ::= net_decl_assignment { **,** net_decl_assignment }

BC19-17

list_of_net_identifiers ::= net_identifier ~~[unpacked_dimension~~ { unpacked_dimension }~~]~~
  { **,** net_identifier ~~[unpacked_dimension~~ { 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_type_assignments ::= type_assignment { **,** type_assignment }

list_of_variable_decl_assignments ::= variable_decl_assign_identifier { **,** variable_decl_assign_identifier }

list_of_variable_identifiers ::= variable_declaration_identifier { **,** variable_declaration_identifier }

list_of_variable_identifiers_or_assignments ::=
  list_of_variable_decl_assignments
  | list_of_variable_identifiers

list_of_variable_port_identifiers ::= port_identifier { unpacked_dimension } [= constant_expression]
{ **,** port_identifier { unpacked_dimension } [**=** constant_expression] }

A.2.4 Declaration assignments

const_assignment ::= const_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

A.2.5 Declaration ranges

unpacked_dimension ::= **[** dimension_constant_expression **:** dimension_constant_expression **]**

packed_dimension ::= **[** dimension_constant_expression **:** dimension_constant_expression **]**

range ::= **[** msb_constant_expression **:** lsb_constant_expression **]**

A.2.6 Function declarations

function_declaration ::=
function [**automatic**] [signing] [range_or_type]
[interface_identifier **.**] function_identifier **;**
{ function_item_declaration }
{ function_statement }
endfunction [**:** function_identifier]
| **function** [**automatic**] [signing] [range_or_type]
[interface_identifier **.**] function_identifier **(** function_port_list **) ;**
{ block_item_declaration }
{ function_statement }
endfunction [**:** function_identifier]

function_item_declaration ::=
block_item_declaration
| { attribute_instance } input_declaration **;**
| { attribute_instance } output_declaration **;**
| { attribute_instance } inout_declaration **;**

function_port_item ::=
{ attribute_instance } input_declaration
| { attribute_instance } output_declaration
| { attribute_instance } inout_declaration

function_port_list ::= function_port_item { **,** function_port_item }

function_prototype ::= **function** data_type **(** list_of_function_proto_formals **)**

named_function_proto::= **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 ::=
    input data_type [variable_declaration_identifier]
    | **inout** data_type [variable_declaration_identifier]
    | **output** data_type [variable_declaration_identifier]
    | variable_declaration_identifier

range_or_type ::=
    { packed_dimension } range
    | data_type

A.2.7 Task declarations

task_declaration ::=
    task [**automatic**] [interface_identifier **.**] task_identifier **;**
    { task_item_declaration }
    { statement }
    endtask [**:** task_identifier]
    | **task** [**automatic**] [interface_identifier **.**] task_identifier **(** task_port_list **) ;**
    { block_item_declaration }
    { statement }
    endtask [**:** task_identifier]

task_item_declaration ::=
    block_item_declaration
    | { attribute_instance } input_declaration **;**
    | { attribute_instance } output_declaration **;**
    | { attribute_instance } inout_declaration **;**

task_port_list ::= task_port_item { **,** task_port_item }
BC19-19 | list_of_port_identifiers { , task_port_item }

task_port_item ::=
    { attribute_instance } input_declaration
    | { attribute_instance } output_declaration
    | { attribute_instance } inout_declaration
BC19-19 | { attribute_instance } port_type list_of_port_identifiers

task_prototype ::=
    task ({ attribute_instance } task_proto_formal { **,** { attribute_instance } task_proto_formal } **)**

named_task_proto ::= **task** task_identifier **(** task_proto_formal { **,** task_proto_formal } **)**

task_proto_formal ::=
    input data_type [variable_declaration_identifier]
    | **inout** data_type [variable_declaration_identifier]
    | **output** data_type [variable_declaration_identifier]

A.2.8 Block item declarations

block_item_declaration ::=
    { attribute_instance } block_data_declaration
    | { attribute_instance } event_declaration
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration **;**

A.2.9 Interface declarations

BC19-28 modport_declaration ::= **modport** list_of_modport_identifiers **;**

list_of_modport_identifiers ::= modport_item { **,** modport_item }

modport_item ::= modport_identifier **(** modport_port { **,** modport_port } **)**

modport_port ::=
    input [port_type] port_identifier

| **output** [port_type] port_identifier
| **inout** [port_type] port_identifier
| interface_identifier **.** port_identifier
| import_export **task** named_task_proto
BC19-20 | import_export **function** named_function_proto
| import_export task_or_function_identifier { **,** task_or_function_identifier }

import_export ::= **import** | **export**

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] **)**
BC19-22 | **.** 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.4.2 Generated instantiation

A.4.2.1 Generated module instantiation

generated_module_instantiation ::= **generate** { generate_module_item } **endgenerate**

generate_module_item_or_null ::= generate_module_item | **;**

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_or_null [**else** generate_module_item_or_null]

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_or_null
| **default** [**:**] generate_module_item_or_null

generate_module_loop_statement ::=
for (genvar_decl_assignment **;** constant_expression **;** genvar_assignment **)**
generate_module_named_block

genvar_assignment ::=

BC19-25

~~genvar_identifier = constant_expression~~
~~|~~ 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_or_null ::= generate_interface_item | **;**

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_or_null [**else** generate_interface_item_or_null]

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_or_null
| **default** [**:**] generate_interface_item_or_null

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_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 statements

continuous_assign ::= **assign** [drive_strength] [delay3] list_of_net_assignments **;**

list_of_net_assignments ::= net_assignment { **,** net_assignment }

net_assignment ::= net_lvalue = expression

A.6.2 Procedural blocks and assignments

BC42-34

initial_construct ::= **initial** statement

always_construct ::= **always** statement

combinational_statement ::= **always_comb** statement

latch_statement ::= **always_latch** statement

ff_statement ::= **always_ff** statement

blocking_assignment ::=
 variable_lvalue = delay_or_event_control expression
 | 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 ::= variable_lvalue = expression

A.6.3 Parallel and sequential blocks

function_seq_block ::=
 begin [**:** block_identifier { block_item_declaration }] { function_statement } **end**

par_block ::=
 fork [**:** block_identifier] { block_item_declaration } { statement } **join** [**:** block_identifier]

seq_block ::=
 begin [**:** block_identifier] { block_item_declaration } { statement } **end** [**:** block_identifier]

A.6.4 Statements

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

BC19-27 | { attribute_instance } inc_or_dec_expression **;**
| { attribute_instance } function_call[7]
| { 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
BC42-35 | { attribute_instance } **process** statement
| { attribute_instance } proc_assertion

statement_or_null ::=
statement
| { attribute_instance } **;**

BC19-27 function_statement ::= [block_identifier **:**] function_statement_item **;**

Editor's Note: Does adding the semicolon cause a problem with the function_blocking_assignment ; (below)?.

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[7]
| { 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 ::=
delay_or_event_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 ::=
expression [**iff** expression]
| hierarchical_identifier [**iff** expression]
| [edge] expression [**iff** expression]

| event_expression **or** event_expression
| event_expression **,** event_expression

edge ::= **posedge** | **negedge** | **changed**

jump_statement ::=
return [expression] **;**
| **break ;**
| **continue ;**

wait_statement ::=
wait (expression **)** statement_or_null

event_trigger ::=
-> hierarchical_event_identifier **;**

disable_statement ::=
disable hierarchical_task_identifier **;**
| **disable** hierarchical_block_identifier **;**

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

| **repeat (** expression **)** function_statement_or_null
| **while (** expression **)** function_statement_or_null
| **for (** variable_decl_or_assignment **;** expression **;** variable_assignment **)**
function_statement_or_null
| **do** function_statement **while (** expression **)**

loop_statement ::=
forever statement
| **repeat (** expression **)** statement_or_null
| **while (** expression **)** statement_or_null
| **for (** variable_decl_or_assignment **;** expression **;** variable_assignment **)** statement_or_null
| **do** statement **while (** expression **)**

variable_decl_or_assignment ::=
BC19-36 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 [**(** expression { **,** expression } **)**] **;**

A.6.10 Assertion statements

proc_assertion ::=
immediate_assert
| strobed_assert
| clocked_immediate_assert
| clocked_strobed_assert

immediate_assert ::= **assert (** expression **)**
statement_or_null
[**else** statement_or_null]

strobed_assert ::= **assert_strobe (** expression **)**
restricted_statement_or_null
[**else** restricted_statement_or_null]

clocked_immediate_assert ::= **assert (** expr_sequence **)** step_control
statement_or_null
[**else** statement_or_null]

clocked_strobed_assert ::= **assert_strobe (** expr_sequence **)** step_control
restricted_statement_or_null
[**else** restricted_statement_or_null]

restricted_statement_or_null ::=
restricted_statement
| { attribute_instance } **;**

restricted_statement ::=
[block_identifier **:**] restricted_statement_item

restricted_statement_item ::=
{ attribute_instance } proc_assertion
| { attribute_instance } system_task_enable
| { attribute_instance } delay_or_event_control statement
| { attribute_instance } restricted_seq_block

restricted_seq_block ::= **begin** [**:** block_identifier] { block_item_declaration }{ restricted_statement }
end [**:** block_identifier]

expr_sequence ::=
expression

| [constant_expression]
| range
| expr_sequence **;** expr_sequence
| expr_sequence ***** [constant_expression]
| expr_sequence ***** range
| **(** expr_sequence **)**

step_control ::=
@@ event_identifier
| **@@** **(** event_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
| input_identifier **[** constant_expression **]**
| input_identifier **[** range_expression **]**

specify_output_terminal_descriptor ::=
output_identifier
| output_identifier **[** constant_expression **]**

| output_identifier **[** 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_descriptor1 ::= **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 } **}**

constant_concatenation ::= **{** constant_expression { **,** 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 **}**

net_concatenation ::= **{** net_concatenation_value { **,** net_concatenation_value } **}**

net_concatenation_value ::=
    hierarchical_net_identifier
  | hierarchical_net_identifier **[** expression **]** { **[** expression **]** }
  | hierarchical_net_identifier **[** expression **]** { **[** expression **]** } **[** range_expression **]**
  | hierarchical_net_identifier **[** range_expression **]**
  | net_concatenation

variable_concatenation ::= **{** variable_concatenation_value { **,** variable_concatenation_value } **}**

variable_concatenation_value ::=
    hierarchical_variable_identifier
  | hierarchical_variable_identifier **[** expression **]** { **[** expression **]** }
  | hierarchical_variable_identifier **[** expression **]** { **[** expression **]** } **[** range_expression **]**
  | hierarchical_variable_identifier **[** range_expression **]**
  | variable_concatenation

A.8.2 Function calls

constant_function_call ::= function_identifier { attribute_instance }
    (constant_expression { **,** constant_expression } **)**

function_call ::= hierarchical_function_identifier{ attribute_instance } **(** expression { **,** expression } **)**

genvar_function_call ::= genvar_function_identifier { attribute_instance }
    (constant_expression { **,** 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 variable_lvalue
  | variable_lvalue 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

constant_mintypmax_expression ::=
    constant_expression
  | constant_expression **:** constant_expression **:** constant_expression

constant_param_expression ::=
    constant_expression
  | data_type

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
  | { attribute_instance } inc_or_dec_expression
  | **(** operator_assignment **)**
  | expression binary_operator { attribute_instance } expression
  | conditional_expression
  | string

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
| 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
| hierarchical_identifier
| hierarchical_identifier **[** expression **]** { **[** expression **]** }
| hierarchical_identifier **[** expression **]** { **[** expression **]** } **[** range_expression **]**
| hierarchical_identifier **[** range_expression **]**
| concatenation
| multiple_concatenation
| function_call
| system_function_call
| constant_function_call
| **(** mintypmax_expression **)**
| **{** expression { , expression } **}**
| **{** expression **{** expression **} }**
| simple_type_or_number **’ (** expression **)**
| simple_type_or_number **’ {** expression { , expression } **}**
| simple_type_or_number **’ {** expression **{** expression **} }**
| time_literal
| **’0** | **’1** | **’z** | **’Z** | **’x** | **’X**

time_literal ::=
unsigned_number time_unit
| fixed_point_number time_unit

time_unit ::= **s** | **ms** | **us** | **ns** | **ps** | **fs**

A.8.5 Expression left-side values

net_lvalue ::=
hierarchical_net_identifier
| hierarchical_net_identifier **[** constant_expression **]** { **[** constant_expression **]** }
| hierarchical_net_identifier **[** constant_expression **]** { **[** constant_expression **]** }
[constant_range_expression **]**
| hierarchical_net_identifier **[** constant_range_expression **]**
| hierarchical_net_identifier **(** [constant_expression { **,** constant_expression }] **)**
| net_concatenation

variable_lvalue ::=
variable_lvalue_item [inc_or_dec_operator]
| hierarchical_variable_identifier **(** [constant_expression { **,** constant_expression }] **)**

variable_lvalue_item ::=
hierarchical_variable_identifier

| hierarchical_variable_identifier **[** expression **]** { **[** expression **]** }
| hierarchical_variable_identifier **[** expression **]** { **[** expression **]** } **[** range_expression **]**
| hierarchical_variable_identifier **[** range_expression **]**
| variable_concatenation

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

EC-CH1 string ::= **"** { Any_ASCII_Characters~~_except_new_line~~ } **"**

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

arrayed_identifier ::=
    simple_arrayed_identifier
  | escaped_arrayed_identifier

block_identifier ::= identifier

cell_identifier ::= identifier

config_identifier ::= identifier

const_identifier ::= identifier

enum_identifier ::= identifier

escaped_arrayed_identifier ::= escaped_identifier **[** range **]**

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

function_identifier ::= identifier

gate_instance_identifier ::= arrayed_identifier

generate_block_identifier ::= identifier genvar_function_identifier ::= identifier[8]

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_net_identifier ::= hierarchical_identifier

hierarchical_variable_identifier ::= hierarchical_identifier

hierarchical_task_identifier ::= hierarchical_identifier

identifier ::=
  simple_identifier
  | escaped_identifier

interface_identifier ::= identifier

inout_port_identifier ::= identifier

input_port_identifier ::= identifier

instance_identifier ::= identifier

library_identifier ::= identifier

memory_identifier ::= identifier

modport_identifier ::= identifier

module_identifier ::= identifier

module_instance_identifier ::= arrayed_identifier

net_identifier ::= identifier

output_port_identifier ::= identifier

parameter_identifier ::= identifier

port_identifier ::= identifier

real_identifier ::= identifier

simple_arrayed_identifier ::= simple_identifier **[** range **]**

simple_hierarchical_identifier[3] ::= simple_hierarchical_branch [**.**escaped_identifier]

simple_identifier[2] ::= [**a-zA-Z_**] { [**a-zA-Z0-9_$**] }

specparam_identifier ::= identifier

state_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 { packed_dimension }

type_identifier ::= identifier

udp_identifier ::= identifier

udp_instance_identifier ::= arrayed_identifier

variable_decl_assign_identifier ::= variable_identifier { unpacked_dimension } [**=** constant_expression]

variable_declaration_identifier ::= variable_identifier { unpacked_dimension }

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 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) Must be a void function
8) Hierarchy is not allowed

Annex B
Keywords

EC-CH19
BC-CH20
EC-CH24
EC-CH25
BC19-59b
EC-CH79
EC-CH89
EC-CH102

SystemVerilog reserves the following keywords:

alias‡	endprimitive	modport†	small
~~all~~‡	endprogram‡	module	solve‡
always	endspecify	nand	specify
always_comb†	endtable	negedge	specparam
always_ff†	endtask	new‡	static†
always_latch†	~~endtransition~~†	nmos	string‡
and	enum†	~~none~~‡	strong0
~~any~~‡	event	nor	strong1
assert†	export†	noshowcancelled	struct†
assert_strobe†	extends‡	not	super‡
assign	extern†	notif0	supply0
~~async†~~	final‡	notif1	supply1
automatic	for	null‡	table
before‡	force	or	task
begin	forever	output	this‡
bit†	fork	packed†	time
break†	forkjoin†	parameter	timeprecision†
buf	function	pmos	timeunit†
bufif0	generate	posedge	tran
bufif1	genvar	primitive	tranif0
byte†	handle‡	priority†	tranif1
case	highz0	~~process~~†	~~transition~~†
casex	highz1	program‡	tri
casez	if	protected‡	tri0
cell	iff†	pull0	tri1
changed†	ifnone	pull1	triand
char†	import†	pulldown	trior
class‡	incdir	pullup	trireg
clocking‡	include	pulsestyle_onevent	type†
cmos	initial	pulsestyle_ondetect	typedef†
config	inout	public‡	union†
const†	input	rand‡	unique†
constraint‡	inside‡	randc‡	unsigned
continue†	instance	rcmos	use
deassign	int†	real	var‡
default	integer	realtime	vectored
defparam	interface†	reg	virtual‡
design	join	release	void†
disable	join_any‡	repeat	wait
do†	join_none‡	return	wand
else	large	rnmos	weak0
end	liblist	rpmos	weak1
endcase	library	rtran	while
endclass‡	local‡	rtranif0	wire
endclocking‡	localparam	rtranif1	with‡
endconfig	logic†	scalared	wor
endfunction	longint†	shortint†	xnor
endgenerate	longreal†	shortreal†	xor
endinterface†	macromodule	showcancelled	
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

Annex C
String Methods

EC-CH105

~~(Informative)~~

Editor’s Note: This entire section is new for draft 1. Only the Section titles have been highlighted as new text.

C.4 Introduction

SystemVerilog 3.1 adds the **string** data type, which is a variable length array. SystemVerilog 3.1 also supports a wide range of methods that operate and manipulate variables of the **string** type. These methods use an object-oriented-like notation, that allow the creation of a large number of built-in, type-specific functions without cluttering the global name space. These methods are described in the following sections.

Editor’s Note: Is it being *suggested* that SystemVerilog tools build in these methods? If so, something to that effect should be stated in this intro.

C.5 len()

```
function integer 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.

C.6 putc()

```
task putc(integer i, string s)
task putc(integer i, char 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** doesn't 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`.

C.7 getc()

```
function int getc(integer 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 ]`.

C.8 toupper()

```
function string toupper()
```

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

— *str* is unchanged.

C.9 tolower()

```
function string tolower()
```

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

— *str* is unchanged.

C.10 compare()

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

— `str.compare(s)` compares *str* and *s*, character by character and returns the difference between the first character in which they differ.

— If the strings are equal, `str.compare(s)` returns 0. (like `strcmp` in ANSI C).

See the relational string operators in section 3.8, table 3-2.

C.11 icompare()

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

— `str.icompare(s)` behaves is similar to `compare()`, but the comparison is case insensitive.

C.12 substr()

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

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

— If 0 <= *i* <= *j* < `str.len()`, `substr()` returns `“”` (the empty string).

C.13 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 to the first non-digit is encountered.

— **atohex** interprets the string as hexadecimal.

— **atooct** interprets the string as octal.

— **atobin** interprets the string as binary.

C.14 atoreal()

```
function real atoreal()
```

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

C.15 itoa()

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

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

EC-CH10

C.16 hextoa()

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

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

C.17 octtoa()

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

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

C.18 bintoa()

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

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

C.19 realtoa()

```
task realtoa(integer i)
```

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

Annex D Linked Lists

(Informative)

Editor’s Note: This entire section is new for draft 1. Only the Section titles have been highlighted as new text.

The List package is analogous to the C++ STL (*Standard Template Library*) List container that is popular with C++ programmers. However, instead of C++ *templates*, the generic code is done using macros. This will be changed to use a parameterized list.

Editor’s Note: What is being said in the preceding paragraph? Is it being *suggested* that SystemVerilog tools include these macros? Or do the tools just provide a macro file that can be included? When will this be “changed to use a parameterized list”, and who is going to do it?

D.20 List definitions

list —A list is a doubly linked list, where every element has a predecessor and successor. It 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 objects of the same type (for example, a container of network packets, a container of microprocessor instructions, etc.). Containers are objects that contain and manage other objects and provide *iterators* that allow the contained objects to be addressed. A container has methods for accessing its elements. Every container has an associated iterator type that can be used to iterate through the container’s elements.

iterator—Iterators provide the interface to containers. They also provide a means to traverse the container elements. Iterators are pointers to nodes within a list. If an iterator points to an object in a range of objects and the iterator is incremented, the iterator then points to the next object in the range.

D.21 List declaration

The List package supports lists of any arbitrary predefined type, such as integer, string, and class object.

To use a particular type of linked one must declare the list, thus:

```
‘include <ListMacros.vrh>
...
‘MakeVeraList(type)
```

Editor’s Note: Can the SystemVerilog standard use the name “Vera”?

D.21.1 Declaring list variables

A list variable must be declared before using it. This is done via the *VeraList* construct:

```
VeraList_type list1, list2, ..., listN;
```

The VeraList construct declares lists of the indicated type. Data stored in the list elements must be of the same type as the list declaration.

D.21.2 Declaring list iterators

All list iterators must be declared before using them via the VeraListIterator construct:

```
VeraListIterator_type iterator1, ..., iteratorN;
```

The *VeraListIterator* construct declares list iterators of the indicated type. An iterator has to be declared as with any other variable declaration.

D.22 Size methods

This section describes the list methods that analyze list sizes.

D.22.1 size()

The **size()** method returns the number of elements in the list container:

```
list1.size();
```

D.22.2 empty()

The **empty()** method returns 1 if the number elements in the list container is 0:

```
list1.empty();
```

D.23 Element access methods

This section describes the list methods used to access list elements.

D.23.1 front()

The **front()** method returns the first element in the list:

```
list1.front();
```

D.23.2 back()

The **back()** method returns the last element in the list:

```
list1.back();
```

D.24 Iteration methods

This section describes the list methods used for iteration.

D.24.1 start()

The **start()** method returns an iterator pointing to the first element in the list:

```
list1.start();
```

D.24.2 finish()

The **finish()** method returns an iterator pointing to the very end of the list, (i.e. past the end value(last element) of the list. The last element can be accessed **list.finish().prev()**.

D.25 Modifying methods

This section describes the list methods used to modify list containers.

D.25.1 assign()

The **assign()** method assigns elements of one list to another.

```
list1.assign(start_iterator, finish_iterator);
```

The method assigns the elements that lie between the two iterators to list1.

If the finish iterator points to an element before the start iterator, the range wraps around the end of the list.

The range iterators must be valid list iterators. If either points to a non-existent element or if they point to different lists, an error is generated.

D.25.2 swap()

The **swap()** method swaps the contents of two lists.

```
list1.swap(list2);
```

The method assigns the elements of list1 to list2, and vice versa.

Swapping a list with itself has no effect. Swapping lists of different sizes generates an error.

D.25.3 clear()

The **clear()** method removes all the elements of the specified list and releases all the memory allocated for the list (except for the list header).

```
list1.clear();
```

D.25.4 purge()

The **purge()** method removes all the elements of the specified list, *and* releases all the memory allocated for the list (including the list header), therefore avoiding possible memory leaks.

```
list1.purge();
```

To use a list that has been purged, the list must be re-created by calling **new()**.

Both the **purge()** and **clear()** methods delete all the elements in the list. However, the **purge()** method deletes the list header as well. Since the **clear()** method does not delete the list header,

subsequent list addition methods such as **push_back()** will work without having to do a **new()** on the list. If you intend to use the same list again, use **list1.clear()**. If the list is being deleted forever, never to be used gain, **list1.purge()** is recommended.

D.25.5 erase()

The **erase()** method removes the indicated element:

```
new_iterator = list1.erase(position_iterator);
```

The element in the indicated position of list1 is removed from the list.

After the element is removed, subsequent elements are moved up (there is no resultant empty element). Upon calling the **erase()** method, the position iterator is made invalid and the method returns a new iterator.

The position iterator must be a valid list iterator. If it points to a non-existent element, or an element from another list, an error is generated.

D.25.6 erase_range()

The **erase_range()** method removes the elements in the indicated range:

```
list1.erase_range(start_iterator, finish_iterator);
```

The **erase_range()** method removes the elements in the range from list1. Note that the elements from start up to, but not including, finish are removed. After the elements are removed, subsequent elements are moved up (there is no resultant empty element). If the finish

iterator points to an element before the start iterator, the range wraps around the end of the list. Any iterators pointing to elements within the range are made invalid.

The range iterators must be valid list iterators. If either points to a non-existent element or if they point to different lists, an error is generated.

D.25.7 push_back()

The **push_back()** method inserts data at the end of the list:

```
list1.push_back(data);
```

The data is added as another element at the end of list1. If the list already has the maximum allowed elements, the element is not added and an overflow error is generated.

The data must be of type a compatible with the list type.

D.25.8 push_front()

The **push_front()** method inserts data at the front of the list:

```
list1.push_front(data);
```

The data is added as another element at the end of list1. If the list already has the maximum allowed elements, the element is not added and an overflow error is generated.

The data must be of type a compatible with the list type.

D.25.9 pop_front()

The **pop_front()** method removes the first element of the list:

```
list1.pop_front();
```

The first element of list1 is removed. If list1 is empty, an error message is generated.

D.25.10 pop_back()

The **pop_back()** method removes the last element of the list:

```
list1.pop_back();
```

The last element of list1 is removed. If list1 is empty, an error message is generated.

D.25.11 insert()

The **insert()** method inserts data before the indicated position:

```
list1.insert(position_iterator, data);
```

The method inserts the given data before the indicated position. Subsequent elements are moved backward. The position iterator must point to an element in the call list.

The data must be of type a compatible with the list type.

D.25.12 insert_range()

The **insert_range()** method inserts elements in a given range before the indicated position:

```
list1.insert_range(position_iterator, start_iterator, finish_iterator);
```

The method inserts the elements in the range between start and finish before the position given by position. Note that the elements from start up to, but not including, finish are inserted. If the finish iterator points to an element before the start iterator, the range wraps around the end of the list. The range iterators can specify a range in another list or a range in list1.

The position iterator must point to an element in the calling list. the range iterators must be valid list iterators. If either points to a non-existent element or if they point to different lists, an error is generated.

D.26 Iterator methods

This section describes the methods used by iterators.

D.26.1 next()

The **next()** method moves the iterator so that it points to the next item in the list:

```
I1.next();
```

D.26.2 prev()

The **prev()** method moves the iterator so that it points to the previous item in the list:

```
I1.prev();
```

D.26.3 eq()

The **eq()** method compares two iterators:

```
I1.eq(I2);
```

The method returns 1 if both iterators point to the same location in the same list. Otherwise, it returns 0.

D.26.4 neq()

The **neq()** method compares two iterators:

```
I1.neq(I2);
```

The method returns 1 if the iterators point to different locations (either different locations in the same list or any location in different lists). Otherwise, it returns 0.

D.26.5 data()

The **data()** method returns the data stored at a particular location:

```
I1.data();
```

The method returns the data stored at the location pointed to by iterator I1.

The data type is of the same type used in declaring the list via **MakeVeraList**(type).

Annex E
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 16.

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 17.2.

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 may be specified. Enumerated types can be easily referenced or displayed using the enumerated names, as opposed to the enumerated values. Section 3.11 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 18.

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 F 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, introduced by the process keyword. 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 F for information about this manual.

Annex F 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

accellera