SystemVerilog 3.1
Testbench Extensions

Version 2.2
November 25, 2002

SYNOPSYS®

SystemVerilog 3.1
VeraLite

1 Introduction

This document specifies VeraLite, the test-bench extensions to SystemVerilog that have been accepted by the Accellera committee to become part of SystemVerilog 3.1. VeraLite is a subset of verification constructs from the Vera hardware verification language. Vera was initially designed as a set of enhancements to "Verilog 1.0", thus, the syntactical and lexical elements of both languages are the same. While many of the test-bench constructs are unique to Vera, the semantics of the constructs common to both languages, including data-types and operators, remain largely the same. This makes VeraLite an ideal candidate for inclusion into SystemVerilog. Nonetheless, Vera, since its inception, has evolved as a separate language. Verilog too has evolved, first into "Verilog 2001", and more recently into SystemVerilog-3.0. As such, Vera and SystemVerilog exhibit several conflicts and areas of functional overlap. This document includes resolutions to all known conflicts between SystemVerilog and Vera, as well as improvements that simplify the language. Since Vera was based largely on "Verilog 1.0", the first set of conflicts is resolved by updating VeraLite to be compatible with "Verilog 2001", thus, allowing VeraLite to become a natural extension to SystemVerilog.

VeraLite enhances SystemVerilog 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

Note that VeraLite constructs are applicable only in the behavioral context. The usage of VeraLite makes sense and should be allowed only in initial or always blocks.

2 Lexical Elements

VeraLite source code consists of a stream of lexical tokens. Lexical tokens consist of one or more characters. The types of lexical element in VeraLite are:

- White Space
- Comments
- Statement Blocks
- Operators
- Identifiers
- Strings
- Numbers
- Keywords

2.1 White Space

White space is any sequence of spaces, tabs, newlines, and formfeeds. White space is used in VeraLite as a token separator. Except within a string, white space is ignored.

2.2 Comments

VeraLite supports two forms of comments: a single-line comment and a block comment. A single-line comment starts with a `//` (double slash) and ends with a newline.

A block comment starts with `/*` and ends with a `*/`. Everything between the start and end tags is a comment. Block comments cannot be nested.

2.3 Statement Blocks

VeraLite supports two forms of execute statement blocks: **`"begin … end"`** and **`"fork … join"`** blocks. Empty execute blocks statements are illegal. For example, the following generates a syntax error:

```
if(1) begin
    ;
end
```

The syntax for fork/join statement blocks is:

```
fork
    process1();
    process2();
        ...
    processN();
join
```

VeraLite introduces two new declaration statement blocks: "**class … endclass**" and "**program … endprogram**". These are decribed later.

2.4 Operators

Operators are sequences of one, two, or three characters. Operators, which are used in expressions, have three forms: Unary, Binary, and Conditional. Unary operators appear to the left of their operand. Binary operators appear between their two operands. The conditional operator has two operator character (? :) separating three operands.

2.5 Identifiers

An identifier is a sequence of letters [a-zA-Z], digits [0-9], dollar signs [$], or underscore characters [_].
Identifiers are case-sensitive and cannot begin with a digit or $.

Strings

A string literal is a sequence of characters enclosed by double quotes(" "). 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.
VeraLite also includes a string data-type to which a string literal can be assigned. Variables of type string have arbitrary length; they are dynamically resized to hold any string. String literals are packed arrays (of a width that is a multiple of 8 bits), and they are implicitly converted to the string type when assigned to a string type or used in an expression involving string type operands (see Section 6).

2.7 Numbers

A number can be expressed in two forms, a simple ***DECIMAL*** form and a ***NUMBER*** form that may specify size and base.
The ***DECIMAL*** form is a simple decimal number specified as a sequence of digits from 0 to 9. An optional plus or a minus sign at the start of the number can be used to specify positive or negative numbers. Underscores are ignored and may be used for clarity.

The **NUMBER** format takes this form:

```
<size>'<base><digits>
```

<size> is a non zero decimal number that specifies the number of bits in the constant. If <size> is omitted and the high order bit is X or Z then they are extended to the size of the expression containing the number. The maximum size is 65535.

<base> is a single case-insensitive character specifies the numerical base of the number. Legal base specifications are d, h, o, b for decimal, hexadecimal, octal, and binary respectively.

<digits> is a sequence of digits that are legal for the specified base format. These are:

```
'b (binary): [01xXzZ_]
'd (decimal): [0123456789_]
'o (octal): [01234567xXzZ_]
'h (hexadecimal): [0123456789abcdefABCDEFxXzZ_]
```

X and x represent unknown values, and Z and z represent high impedance values in binary, octal, or hexadecimal form. The underscores are ignored and can be used to increase readability.
If the most significant specified digit of a number is X or Z, then they are extended to fill the higher order digits.
For example, 4'bx is equivalent to 8'bxxxx, and 8'bz00 is equivalent to 8'bzzzzzz00.
If not all the bits are specified and the highest specified bit is not x or z, then zero filling takes place.

2.8 Keywords

Keywords are predefined identifiers used to define language constructs. The VeraLite subset recognizes the keywords shown in the table below. Keywords unique to VeraLite (not in SystemVerilog 3.0) are shown in **boldface**.

all	**endclass**	integer	return
any	**endclocking**	join	static
async	enfunction	**local**	**string**
begin	**endprogram**	negedge	**super**
bit	endtask	**new**	task
break	enum	**none**	**this**
case	event	**null**	typedef
casex	**extends**	or	**var**
casez	extern	output	void
class	for	posedge	**virtual**
clocking	fork	program	while
continue	function	**protected**	
default	if	**public**	
else	inout	reg	
end	input	repeat	

2.8.1 VeraLite Predefined Constants

VeraLite introduces several predefined constants. The table below lists the predefined constant identifiers

ALL	HAND_SHAKE	ONE_BLAST
ANY	OFF	ONE_SHOT
CHECK	ON	ORDER

These predefined constants are defined using the following enumerated types:

enum TriggerModes { OFF, ON, ONE_SHOT, ONE_BLAST, HAND_SHAKE };
enum CheckMode { CHECK = 0 };
enum SyncModes { ALL = 1, ANY, ORDER };

3 Data Types and Variable Declaration

VeraLite supports all the SystemVerilog standard types and user defined types. It also enhances several existing data types, and extends the user defined types by providing support for object-oriented class. VeraLite supports all of the following standard data types listed below. The data types introduced or enhanced by VeraLite are underlined.

3.1 Basic integer data types

- **char** 2-state 8 bit signed integer (SystemVerilog 3.0)
- **shortint** 2-state 16 bit signed integer (SystemVerilog 3.0)
- **int** 2-state 32 bit signed integer (SystemVerilog 3.0)
- **longint** 2-state 64 bit signed integer (SystemVerilog 3.0)
- **byte** 2-state 8 bit signed integer (SystemVerilog 3.0)
- **bit** 2-state user-defined vector size (SystemVerilog 3.0)
- **logic** 4-state user-defined vector size (SystemVerilog 3.0)
- **reg** 4-state user-defined vector size (Verilog)
- **integer** 4-state 32 bit signed integer (Verilog)

3.2 Other basic data types

- **Time** 64-bit time step (SystemVerilog 3.0)
- **real** Floating point (SystemVerilog 3.0)
- **shortreal** Floating point (SystemVerilog 3.0)
- **event** Enhanced Verilog events (Verilog/VeraLite)
- **string** Arbitrary length character string (VeraLite)

3.3 User Defined data types

- **struct** Structures (SystemVerilog 3.0)
- **union** C-like union (SystemVerilog 3.0)
- **enum** Enhanced Enumerations (SystemVerilog 3.0)
- **class** Object-Oriented class (VeraLite)

Variables of basic integer type, string, event, and enum can be declared with an optional initial value. For example:

```
integer count = 7;
bit [8:1] address = 8'hff;
```

3.3.1 Integral Types

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 types, **packed struct**, **packed union**, **enum,** and **Time**.

3.4 String

The **string** data type is variable size array of characters. VeraLite offers a wide range of methods and operators that manipulate strings.

The syntax to declare a string is:

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

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

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

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

String operators and semantics are discussed in detail in Section 8.4.

3.5 Event

The **event** data type is an enhancement to Verilog named events. VeraLite events provide a handle to a synchronization object. Like Verilog, event variables can be explicitly triggered and waited for, however, VeraLite events can also have a persistent triggered state, that is, the synchronization object can be either ON or OFF. 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:

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

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

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

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

Event operations and semantics are discussed in detail in Section 12.4.

3.6 Enumerations

An enumerated type provides the capability to declare sets of integral named constants.

To declare an enumeration, SystemVerilog-3.0 uses the following syntax:

enum [integer_type [signing] {packed_dimension}] **{** value_list **}**
and, value_list is: enum_value [= constant]{,enum_value [= constant]}

For example:

```
enum {red, yellow, green} light1, light2; // anonymous int type
enum logic [1:0] {red = 2, yellow, green, unknown = 'x};
```

And, to create an enumerated type, SystemVerilog requires the use of **typedef**. For example:

typedef enum { red, green, blue } Colors;

VeraLite extends SystemVerilog with 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.

Create an enumerated type called StreetLight:
```
enum StreetLight {red, yellow, green};
```

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 modified form cannot be used to declares both a type and variables of that type. For example, the following is an error:
enum Boolean { FALSE, TRUE } myvar;

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 variable 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.

Enumerated types are discussed in more detail in Section 4.9.

3.7 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 7.

4 Arrays

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

4.1 Fixed-size Arrays

SystemVerilog 3.0 supports multi-dimensional fixed-size arrays. These arrays can be *packed* or *unpacked,* but their sizes are fixed and must be specified by constant expressions. Packed arrays can only be made of single bit types (2-state or 4-state): **bit, logic, reg, wire**, or net-type, and are stored as a continuous set of bits. A packed array declaration has the dimensions specified to the left of the variable being declared. Unpacked arrays can be made of any variable type; their dimensions are specified to the right of the variable being declared.

```
bit [16:1] p_a;          //  'p_a' is a packed array of 16 bits
int u_a [16:1];          // 'u_a' is an unpacked array of 16 int's
reg [7:0] bytes [9:0];   // 'bytes' is an unpacked array of 10 packed arrays of 8 reg's.
```

Fixed-size arrays can have multiple dimensions in both their packed and unpacked dimensions.

```
bit [1:10][1:2] xyz [1:5][1:10];      // 'xyz' is an array of 5 * 10  packed array of 10 * 2 bit's
```

Packed arrays allow arbitrary length integer types that can be used in arithmetic expressions. The maximum size of a packed array is at least 65536 bits.

The result of using an expression containing unknown bits (**`x`** or **`z`**) as an array index depends on the operation and on type of the array. A read returns **`x`** if the array is of a 4-state type, and **`0`** if the array is of a 2-state type. A write is ignored and causes a warning to be issued regardless of the array type.
If an index expression is out of bounds, a run-time warning is issued. These bounds checks are always enabled by VeraLite, but may be disabled in SystemVerilog.

Unpacked arrays can be made of any scalar (non-unpacked-array) type. VeraLite 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 7.1) and events (see Section 12.5).

Note: VeraLite 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];
```

4.2 Array Initialization

An array can be initialized as part of its declaration. The values used for array initialization are subject to the same rules as the initialization of the scalar variables that make up the array.
Array initialization in SystemVerilog uses braces { } to denote each array dimension, but unlike C, the nesting of braces of must follow the number of dimensions.

For example, a single dimensional array can be initialized as:

```
integer array[5:1] = {0, 1, 2, 3, 4};
```

And a multi-dimensional array can be initialized as:

```
int arr_2_by_3 [1:2][1:3] = {{0,1,2},{5,6,67}};
```

Fixed-sized array can be initialized as part of their declaration provided that only constant expressions are used as initializers. Thus, arrays of events or object handles may not be initialized in their declaration.
Concatenation and replication of constant expressions can be used in array initialization.

4.3 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 [*];

data_type

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).

4.3.1 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:

```
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. s*rc_array* must be a dynamic array of the same data type as *array_nam*e, but it need not have the same size. If the size of *src_array* is less than *size*, the extra elements of *array_nam*e are initialized to their default value. If the size of *src_array* is greater than *size*, the additional elements of *src_array* are 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);
```

4.3.2 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).

4.3.3 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
```

4.4 Array Assignment

Assigning to a fixed-size unpacked array requires that the source and the target both be arrays with the same number of unpacked dimensions, and the length of each dimension be the same. Assignment is done by assigning each element of the source array to the corresponding element of the target array, which requires that the source and target arrays be of compatible types.

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

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

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, a fixed size array, this operation requires a run-time check.

```
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 elements

A = B;        // ok.  Compatible type and same size
A = C;        // run-time 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] };
```

This example creates the dynamic array *p* with contents: “a”, “b”, “c”, “hello”, “d”, “e”.

4.5 Arrays as Arguments

Arrays can be passed as arguments to tasks or functions. The rules that govern array argument passing by value (see Section 9.3) are the same as for array assignment.

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, and 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 a 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.

4.6 Associative Arrays

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

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

The syntax to declare associative an associative array is:
data_type array_id **[** [*index_type*] **]**;

data_type	The data type of the array elements. Can be any type allowed for fixed-size arrays.
array_id	The name of the array being declared.
index_type	Optional. 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.

4.6.1 Unspecified Index Type

Example: **int** array_name [];

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

— The array can be indexed by any integral data type, including integers, packed arrays of arbitrary length, string literals, and packed structs. Since the indices can be of different sizes, the same numerical value may have multiple representations, each of a different size. VeraLite resolves this ambiguity by detecting the number of leading zeros and computing a unique length and representation for every value.
— Non-integral index types are illegal and result in a compiler 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).

4.6.2 String Index

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

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

— Indices can be strings or string literals of any length. Other types are illegal and result in a compiler error.
— An empty string **“”** index is valid.
— The ordering is lexicographical (lesser to greater).

4.6.3 *Class* Index

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

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

— Indices can be objets of that particular type or derived from that type. Any other type is illegal and results in a compiler error.
— A **null** index is invalid.
— The ordering is deterministic but arbitrary.

4.6.4 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.

4.6.5 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.

4.6.6 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:

4-state integral type	**'x**
2-state integral type	**'0**
enumeration	first element in the enumeration
string	**""**
class	**null**
event	**null**

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

4.7 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.

4.7.1 num()

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

```
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"
```

4.7.2 delete()

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

```
task delete( [input index] );
```

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

If the *index* is specified then the delete method removes the entry at the specified index. If the entry to be deleted does not exist, the task issues no warning.
If the *index* is not specified then the delete method removes all the elements in the array.

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

4.7.3 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;
```

4.7.4 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.

The **first()** function 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] );
```

4.7.5 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] );
```

4.7.6 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 ) );
```

4.7.7 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)
```

4.8 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.

4.9 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.

5 Enumerated Types

Enumerated types are user-defined sets of named integral valued constants (see Section 3.6).

For example:

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

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. VeraLite 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.

Elements within enumerated type definitions are assigned identifiers, which are numbered consecutively, starting from 0. In the Colors example above, red is assigned 0, green is assigned 1, and so on. An explicit value in an enumerated type declaration affects all subsequent values that have no explicit value.
A range of enumeration elements can be specified automatically, via the following syntax:

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

For example:

```
enum opcode { 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.

5.1 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:

```
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).

5.2 Dynamic Casting: $cast()

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

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

```
   function int $cast( scalar dest_var, scalar source_exp );
or
   task $cast( scalar dest_var, scalar source_exp );
```

dest_var:

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

source_exp:

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

Use of **$cast()** as either a task or a function determines how invalid assignments are handled. When called as task, **$cast()** attempts to assign the source expression to the destination variable. If the assignment is invalid, a fatal runtime error occurs.
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.

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.
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, users can specify this operation using a static SystemVerilog cast operation:

```
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, they can choose the faster compile-time cast. If users need to check if the expression lies within the enumeration values, they need not 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 tradeoffs.

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.

5.3 Increment and Decrement Operators on Enumerated Types

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

Operator	Description
++enumVar	Assigns the next enumeration member (according to the definition order) to enumVar. A wrap around to the first enumeartion 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 enumeartion value occurs when decrementing the first enumeration value.
enumVar **+= N**	Assigns to enumVar its N^{th} 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 N^{th} 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 Section 5.2), either as:

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

6 String

VeraLite introduces the **string** data type, a variable size, dynamically allocated array of characters. 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). In VeraLite a string literal behaves the same way. However, VeraLite also supports the **string** data type to which a string literal can be assigned. In SystemVerilog, a string literal assigned to a packed array is truncated to the size of the array, whereas in VeraLite, strings can be of arbitrary length and no truncation occurs.

In VeraLite string literals behave exactly the same as in Verilog, except that they are implicitly converted to the string type when assigned to a string type or used in an expression involving string type operands (see Section 8.4).

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 **“”.**

6.1 String Methods

In addition to the operators allowed on strings (see Section 8.4), VeraLite supports a wide range of methods that operate and manipulate variables of **string** type. These methods use an object-oriented-like notation, which may appear as a departure from Verilog, that allow creation of a large number of built-in, type-specific functions without cluttering the global namespace. These methods are described in the following sections.

6.1.1 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.

6.1.2 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.

6.1.3 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].

6.1.4 toupper()

```
function string toupper()
```

— *str*.toupper() returns a string with characters in *str* converted to uppercase.
— *str* is unchanged.

6.1.5 tolower()

```
function string tolower()
```

— *str*.tolower() returns a string with characters in *str* converted to lowercase.
— *str* is unchanged.

6.1.6 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 Table 3.

6.1.7 icompare()

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

— *str*.icompare(s) behaves is similar to **compare()**, but the comparison is case insensitive.

6.1.8 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).

6.1.9 atoi(), atohex(), atooct(), atobin()

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

— *str*.atoi() returns the integer corresponding to the ASCII decimal representation in *str*.
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.

6.1.10 atoreal()

```
function real atoreal()
```

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

6.1.11 itoa()

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

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

7 Classes

SystemVerilog-3.0 includes **structs** for data encapsulation. VeraLite adds 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.

A **class** is a collection of data and a set of 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.

7.1 Objects (Class Instance)

The last 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** task.

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

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
```

7.2 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;
```

7.3 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.

7.4 Constructors

VeraLite 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. VeraLite 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 non-blocking. Even though **new** does not specify a return type, the left-hand side of the assignment determines the return type.

Every class has a default (built-in) **new** method. The default constructor first calls its parent class constructor (**super.new()** as described in Section 7.10) and then proceeds to initialize each member of the current object to its default (or uninitialized value).

It is also possible to pass arguments to the constructor, which allows run-time customization of an object:

```
Packet p = new(STARTUP, $random, $time);
```

where the **new** initialization task in *Packet* might now look like:

```
function new(int cmd = IDLE, bit[12:0] adrs = 0, int time );
     command = cmd;
     address = adrs;
     time_requested = time;
endfunction
```

The conventions for arguments are the same as for procedural subroutine calls, including the use of default arguments.

7.5 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:

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

7.6 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:

```
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.

7.7 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.

If, however, the last expression above is re-written slightly differently, it will make a copy of `p1`:

```
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 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

task 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
```

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.

7.8 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 VeraLite is called *Single-Inheritance,* that is, each class is derived from a single parent class.

7.9 Overriden 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
```

To get the overridden method, the parent method needs to be declared virtual (see below).

7.10 super

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.

```
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). Subclasses are classes that are extensions of the current class. Whereas super-classes are classes that the *current* class is extended from, beginning with the original base class.

Note: When using the **super** within **new**, **super.new** must be the first executable statement in the constructor. This is because the super-class 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.

7.11 Casting

It is always legal to assign subclass variable to a variable of a class higher in the inheritance tree. It is never legal to directly assign a super-class 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.

To check if the assignment is legal, the dynamic cast function **$cast()** is needed (see Section 5.2).

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

```
   task $cast( scalar dest_handle, scalar source_handle );
or
   function int $cast( scalar dest_handle, scalar source_handle );
```

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.

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

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

This version does the assignment and returns 1 if the assignment is valid. Otherwise, it sets the destination handle to null and returns 0.

7.12 Chaining Constructors

When a subclass is instantiated, one of the system's first actions is to invoke the class method **new().** The first, implicit action **new()** takes is to invoke the **new()** method of its super-class, 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.
If the initialization method of the super-class requires arguments, one has two choices. To always supply the same arguments or to use the **super** keywors. 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.

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

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

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

7.13 Data Hiding and Encapsulation

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. 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 VeraLite, 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.

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.

7.14 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.

7.15 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.
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, 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
```

EtherPacket is now a class that can be instantiated. In general, if an abstract class has several virtual methods, all of the methods must be overridden for the subclass to be instantiated. If all of the methods are not overridden, the subclass needs to be abstract.
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.

7.16 Polymorphism: Dynamic Method Lookup

Polymorphism allows one to use super-class variables to hold subclass objects, and to reference the methods of those subclasses directly from the super-class 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.

7.17 Out of Block Declarations

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 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
```

The first lines of each part of the method declaration are nearly identical, except for the attributes and class-reference fields.

7.18 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
```

7.19 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.

7.20 Classes, Structs, and Unions

SystemVerilog-3.0 includes **struct** and **union**. VeraLite 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, VeraLite 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, VeraLite objects are strictly strongly-typed. Copying an object of one type onto an object of another is not allowed.
3. VeraLite objects are implemented using handles, thereby providing C-like pointer functionality. But, VeraLite disallows casting handles onto other data types, thus, unlike C, VeraLite handles are guaranteed to be safe.
4. VeraLite 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.

7.21 Memory Management

Memory for objects, strings, and dynamic and associative arrays is allocated dynamically. When objects are created, VeraLite allocates more memory. When an object is not needed anymore, VeraLite automatically reclaims the memory, making it available for re-use. The automatic memory management system is an integral part of VeraLite. 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:

```
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, VeraLite 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.

8 Operators and Expressions

VeraLite supports all the "SystemVerilog 3.0" operators plus a few additional ones.
Table 1 lists all VeraLite operators. Operators common to VeraLite and SystemVerilog have the same semantics as in "SystemVerilog 3.0". Operators that do not exist in "SystemVerilog 3.0" or that have extended behavior are shown in **boldface.**

Operator	*Description*	*Semantics*
`{}`	RHS numeric concatenation	Same as SystemVerilog 3.0
`{}`	LHS numeric concatenation	Same as SystemVerilog 3.0
`{{}}`	Numeric Replication	Same as SystemVerilog 3.0
`{}`	**String concatenation**	**Not in SystemVerilog 3.0**[1]
`{{}}`	**String replication**	**Not in SystemVerilog 3.0**
`+ - * /`	Arithmetic	Same as SystemVerilog 3.0
`%`	Modulus	Same as SystemVerilog 3.0
`++ --`	Increment/Decrement (post)	Same as SystemVerilog 3.0[2]
`++ --`	Increment/Decrement (pre)	Same as SystemVerilog 3.0
`+= -= *= /= %= <<= >>= &= |= ^= ~&= ~|= ~^= <<<= >>>=`	Compound assignment	Same as SystemVerilog 3.0
`=`	Simple Assignment	Same as SystemVerilog 3.0
`< <= > >=`	Relational	Same as SystemVerilog 3.0
`! && || == !=`	Logical operators	Same as SystemVerilog 3.0
`=== !==`	Case equality, inequality	Same as SystemVerilog 3.0
`=?= !?=`	**Wild equality, inequality**	**Not in SystemVerilog 3.0**
`~`	Bit-wise negation	Same as SystemVerilog 3.0
`& | ^ ^~`	Bit-wise binary operators	Same as SystemVerilog 3.0
`& | ^ ~& ~| ~^ ^~`	Reduction operators	Same as SystemVerilog 3.0
`<< >>`	Shift	Same as SystemVerilog 3.0
`<<< >>>`	Arithmetic Shift	Same as SystemVerilog 3.0
`**`	Exponentiation	Same as SystemVerilog 3.0
`?:`	Conditional	Same as SystemVerilog 3.0

Table 1: VeraLite Operators

8.1 Operator Precedence

Table 2 lists the precedence and associativity of all VeraLite operators. Highest precedence operators are listed first. Precedence is the same as in SystemVerilog 3.0.

Operator	Associativity
`() [] .`	left
Unary `! ~ ++ -- & ~& | ~| ^ ~^`	right

[1] These operators don't exist in SystemVerilog when applied to string variables (see Section 8.4).
[2] The behavior of ++ and -- is incompletely specified in SystemVerilog (see Section 8.3).

**	left
* / %	left
+ -	left
<< >> <<< >>>	left
< <= > >=	left
== != === !== =?= !?=	left
& &~	left
^ ~^	left
| |~	left
&&	left
||	left
? :	right
= += != *= /= %= &= |= ^= <<= >>= <<<= >>>=	none

Table 2: Operator Precedence

8.2 Wild Equality and Wild Inequality

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

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

The wild equality operator (=?=) and inequality operator (!?=) treat **X** and **Z** values in a given bit position as a wildcard. A wildcard bit matches any bit value (**0**, **1**, **Z**, or **X**) in the value of the expression being compared against it.
These operators compare operands bit for bit, and return a 1-bit self-determined result. If the operands are not the same length, the shorter operand is zero-filled. 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**. Finally, the =?= and !?= operators treat **X** or **Z** as wildcards that match any value, thus, they too never result in **X**.

8.3 Side effecting operators: ++ and --

The behavior of the ++ and -- operators (pre/post increment/decrement) is incompletely defined in SystemVerilog 3.0. This can lead to unexpected behavior when a single statement modifies the same variable more than once. The ANSI-C standard specifically leaves this behavior undefined, allowing every compiler to do it differently, and indeed they do. For example, the following C code fragment produces the output shown below:

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

VeraLite	1 2 4 3 3 2	gcc -g	1 2 4 3 3 2
cc (solaris)	1 2 1 1 3 2	gcc -O2	1 2 1 1 3 2
cc (dec)	1 1 2 1 1 1	cc (linux)	0 -1 -1 -2 0 1

VeraLite 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); begin    // ++a
 a += 1;
 pre_inc = a;
end
endfunction

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

The above description states a semantic definition for these operators. VeraLite'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));

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.

8.4 String Operators

VeraLite introduces the string data type, and 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.

In VeraLite 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.

For example:

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

```
b = r;              // Error
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 same as a[0] = "hii" )
```

Operator	Semantics
Str_1 **==** Str_2	**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.
Str_1 **!=** Str_2	**Inequality**. Logical Negation of ==
Str_1 **<** Str_2 Str_1 **<=** Str_2 Str_1 **>** Str_2 Str_1 **>=** Str_2	**Comparison**. Relational operators return 1 if the corresponding condition is true using the lexicographical ordering of the two strings Str_1 and Str_2. 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.
{Str_1,Str_2,...,Str_n**}**	**Concatenation**. Each Str_i may be of type string or a string literal (it will be implicitly converted to string). If at least one Str_i is of type string, then the expression evaluates to the concatenated string and is of type string. If all the Str_i are string literals then the expression behaves like Verilog concatenation of integral types; if the result is then used in an expression involving string types, it is implicitly converted to string type.
{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 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 string type).
Str.method(...)	The dot (.) operator is used to invoke a specified method on strings. See Section 6.1 for detailed descriptions of the various string methods available.

Table 3: String Operators

8.5 Concatenation and Replication

The concatenation operation is specified using braces (**{ }**), as in Verilog. The concatenation is treated as a packed vector of bits (or **logic** if any operand is of type **logic**). Concatenation can be used on the left hand side of an assignment or in an expression.

```
logic a, b, c;
{a, b, c} = 3'b111;
{a, b, c} = {1'b1, 1'b1, 1'b1};  // same effect as 3'b111
```

In addition, VeraLite 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 8.4). 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'
```

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

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.

9 Subroutines

Like Verilog, VeraLite supports two means of encapsulating often-executed program fragments:
— **Functions** Execute the subroutine whenever called and returns a value right away
— **Tasks** Execute the subroutine whenever called, returns no value, and may block.

VeraLite extends SystemVerilog-3.0 subroutines with:
— Lifetime attribute for modules
— Discarding function values
— Parameter passing by Value or Reference
— Default arguments

9.1 Scope and Lifetime

In SystemVerilog-3.0 Subroutines declared outside a module or interface have global scope, are available everywhere and exist in the **$root** scope. Subroutines definitions cannot be nested, they must be made either at the top level (**$root** scope) or within a module (module scope).

Verilog-2001 allows tasks and functions to be declared as **automatic**, making all storage within the task or function automatic. SystemVerilog-3.0 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. Subroutines with **automatic** variables are re-entrant and can be called recursively.

SystemVerilog-3.0 also allows data to be explicitly declared as **static**. Data declared as **static** in an automatic task, function or in a process has a static lifetime and a scope local to the block.

In SystemVerilog-3.0 the default lifetime for tasks and functions is **static**. Yet, most test-bench subroutines are **automatic**, and having to explicitly declare each of them so can be cumbersome. This is resolved by adding 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 14).

Also, class methods are by default automatic, regardless of the lifetime attribute of the module in which they are declared. Classes are discussed in Section 7.

9.2 Discarding Function Return Values

Values returned by functions must be assigned or used in an expression. Calling a function as if it has no return value results in a compilation error. VeraLite allows using the **void** data type to discard a function's return value. In SystemVerilog-3.0, this can be done with a cast to the **void** type:

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

VeraLite provides a different syntactical alternative in which the return value is assigned to the **void** keyword:

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

The VeraLite form is preferred since it shows the intent more clearly than a cast, and doesn't force users to use an extra set of parenthesis.

9.3 Parameter Passing

VeraLite provides two means for passing arguments to functions and tasks: *by value* and *by reference*.

9.3.1 Pass By Value

Pass by value is the default mechanism for passing arguments to subroutines, it is also the only one provided by SystemVerilog-3.0. 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.

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

9.3.2 Pass By Reference

Arguments *passed by reference* are not copied into the subroutine area, rather, a reference to the original argument is passed to the subroutine. The subroutine can then access the argument data 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:

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

Not that in the example, no change other than addition of the **var** keyword is needed. The compiler knows that *packet* is now addressed indirectly vi 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
```

Also, because the argument is passed by reference, both the caller an the callee share the same representation of the argument, so any changes made to the argument either within the caller or the callee 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
```

9.4 Default Arguments

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

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

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

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.

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 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 is issued.

```
task read(int j = 0, int k, int data = 1 );
    ...
endtask;
```

This example declares a task read() with two default arguments, `j` and `data`.

The task can the 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
```

9.5 Argument Passing by Name

VeraLite 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 a defaults, they are treated like parameters to module instances. If the arguments do not have a default then they must be given or the compiler will issue an error.

10 Sequential Control

This section reviews the behavioral constructs for sequential flow control. These are all fully compatible with SystemVerilog. VeraLite enhances the SystemVerilog **for** loop by allowing multiple assignment statements as the initial and increment statements.

10.1 if-else Statements

The if-else statement is used to decide whether a statement is executed or not.
The syntax to declare an **if-else** statement is:

```
if (expression) if_block [else else_block]
```

where *expression* is any valid expression that evaluates to true, false, or perhaps **x**.
The *if_block* or *else_block* can be any statement or block of statements. If a code block is used, the entire block is executed.
If the *expression* evaluates to true, the *if_block* is executed. If it evaluates to false (or **x**) the *else_block* is executed.
If the *else_block* is omitted, the expression is evaluated and the *if_block* is executed only if it evaluates to true. Otherwise, the program continues execution with the first line after the *if_block*. If-else statements can be nested.

```
if (operator == 0) y = a+b;
else if (operator == 1) y = a-b;
else if (operator == 2) y = a*b;
else y = 'bx;
```

This example uses several if-else statements. The final else statement is associated with the *if_block* immediately preceding it.

10.2 case Statements

The case statement provides for multi-way branching.
The syntax to declare a **case** statement is:

```
case (primary_expression)
   case1_expression : statement
   case2_expression : statement
   ...
   caseN_expression : statement
   [default : statement]
endcase
```

The *primary_expression* is evaluated and its value is successively checked against each *case_expression*. When an exact match is found, the statement corresponding to the matching case is executed, and control is passed to the first line of code after the endcase. If other matches exist, they are not executed.

The *case_expression* can be any valid expression. Expressions separated by commas allow multiple expressions to share the same statement block.

All case expressions must have the same bit length **x** and **z** values are actual values and are not ignored.

The *statement(s)* can be any valid statement or block of statements. If a code block is used, the entire block is executed.
A case statement must have at least one case item aside from the default case, which is optional. The default case must be the last item in a case statement.

```
case( bus[3:0] )
    4'b00ZZ: packet = NONE;
    4'b0001, 4'b1001: packet = READ;
    4'b0010, 4'b1010: packet = WRITE;
    4'b00XX: packet = UNKNOWN;
    default: $display("Bad packet %h detected\n", bus[3:0]);
endcase
```

The **casex** and **casez** statements allow treating as wildcards any **x** or **z** values in both the *primary_expression* and each *case_expression*. **casex**, treats both **x** or **z** values as *don't-care* in both *primary_expression* and each *case_expression*, while **casez** treats **z** values as *don't-care*. Like in the **case** statement, if no match is found, the default statement is executed.

The **if-else, case, casez** and **casex** statements can be preceded by the **unique** or **priority** qualifiers to indicate that only one statement is allowed to match the expression or that the expressions be evaluated in the given order, respectively.

10.3 repeat loops

The repeat loop executes a statement an integral number of times.
The syntax to declare a repeat loop is:

```
repeat (expression) statement
```

The *expression* can be any valid integral expression, including constants.
The *statement* can be any valid statement or block of statements. If a code block is used, the entire block is executed.
The value of the expression is evaluated before the loop starts. Changing a variable within the expression does not change the number of times to be executed.

```
repeat(4) shift_op <<= 3;
```

10.4 for loops

Generalized looping construct that controls execution of its associated statement via the three constructs *initial*, *condition,* and *increment.* The syntax to declare a for loop is:

```
for(initial; condition; increment) statement
```

The *initial* statement is one or more comma-separated assignment statements that are executed before the loop starts. Normally used to control the number of loops to be executed. The first *initial* statement can optionally declare a single loop control variable.

The *condition* can be any valid expression that can be evaluated as a boolean. It is evaluated each time the loop executes,
The *increment* specifies one or more assignments, normally used to modify the value of the loop-control variable. They execute each time the loop executes, after *statement*.
The *statement* can be any valid statement or block of statements. If a code block is used, the entire block is executed.
The for loop sets the initial value of the loop control variable. It evaluates the condition. If the condition is true, the loop executes a one time. When the loop finishes one iteration, the update expression is executed. Typically this expression changes the value of the loop control variable. Then the condition is checked again and the process continues. The loop continues as long as the condition evaluates to true. When it does not evaluate to true, the loop stops and control is passed to the first line after the loop.

```
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 );
```

VeraLite enhances SystemVerilog-3.0 by allowing the initial and increment statements to contain more than one assignment statement; these statements separated by commas (,). Initial statements may alos declare and initialize more than one variable.

10.5 while loops

The syntax to declare a while loop is:

```
while (condition) statement
```

The *condition* can be any valid expression that can be evaluated as a boolean.
The *statement* can be any valid statement or block of statements. If a code block is used, the entire block is executed.
The loop iterates while *condition* is true. When the condition is false or **X**, control passes to the first line of code after the loop. The *condition* is checked at the start of each loop.

```
operator = 0;
while (operator < 5) begin
   operator += 1;
end
```

10.6 do loops

The syntax to declare a do loop is:

```
do statement while (condition)
```

The *condition* can be any valid expression that can be evaluated as a boolean.
The *statement* can be any valid statement or block of statements. If a code block is used, the entire block is executed.

The loop executes *statement* and then iterates while *condition* is true. When the condition is false or **x**, control passes to the first line of code after the loop. The *condition* is checked at the end of each loop.

```
do
   operator <<= 1;
while (nshift < 5)
```

10.7 Jump Statements

The C-like **break** and **continue** statements can be used for flow control within loops. The **return** statement allows termination of a task or function.

10.7.1 break

The break statement forces immediate termination of a loop. A **break** statement can only be executed from inside a loop. Executing **break** terminates the loop immediately and passes control to the first line after the loop.

```
while (test_flag) begin
   if (done)
      break;
end
```

10.7.2 continue

The **continue** statement forces the next iteration of a loop to take place.
A **continue** statement can only be executed from inside a loop. In a **repeat** loop, the **continue** statement passes control back to the top of the loop. If the loop is complete, control is then passed to the first line after the loop.
In a **for** loop the **continue** statement passes control to the *increment* statement.
In a **while** loop, the **continue** statement passes control to the *condition*.

```
for ( int j = 0; j < 10; j++ ) begin
    if (skip_loop) continue;
          ...
end
```

10.7.3 return

Normally, functions and tasks return control to the caller after executing the last statement of the block. The **return** statement exits the subroutine and passes control back to the caller.
The **return** statement takes two forms:

— **return;**
— **return** *expression;*

The first form is allowed in a task or function.
The second form is only allowed in a function, and causes *expression* to be implicitly assigned to the function's return variable.

If a **return** statement with no expression is executed inside a function, the value returned is the default (or uninitialized) value for the corresponding data type.

11 Processes

Verilog-2001 provides **always** and **initial** blocks that define static processes.
An **always** block is used to model combinational logic. SystemVerilog-3.0 enhances **always** blocks with specialized **always_comb**, **always_latch**, and **always_ff** blocks, which indicate combinational behavior, latched behavior, or sequential logic behavior, respectively.

In systems modeling, one of the key limitations of Verilog is its inability to create processes dynamically, as happens in an operating system. Verilog has the **fork** .. **join** construct, but this still imposes a static limit. SystemVerilog-3.0 adds dynamic processes, which are introduced by the **process** statement. VeraLite adds dynamic processes by enhancing the **fork .. join** construct, in a way that is more natural to Verilog users.

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. Each thread executes uninterrupted until it encounters a blocking statement, such as waiting for an event, a delay statement, etc…

Verilog provides process control via the **disable** construct, but is limited to processes associated with named blocks and cannot distinguish between different processes executing the same block. VeraLite 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**.

11.1 fork .. join

The **fork .. join** construct provides the primary mechanism for creating concurrent processes. The syntax to declare a **fork…join** block is:

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

The *statement(s)* can be any valid statement or block of statement enclosed by **begin .. end**. One or more statements can be specified, each statement will execute as a concurrent process. The spawned processes start executing in strict source order: the first statement (*statement*^{1}), starts executing first, followed by the second *(statement*^{2}*),* and so on.

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

The join option (**all, any**, **none)** specifies when the parent (forking) process resumes execution. If the join option is not specified, the VeraLite uses the Verilog default, which is the same as **all**.

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

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 11.5). 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
```

Example: 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 was specified (same as **all**), the parent process will block until the two prcesses complete, that is, 20ns have elapased 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 results 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
```

The **fork..join none** construct overlaps with SystemVerilog-3.0's **process** statement. While the two can coexist, we propose deprecation of **process** in favor or the more natural VeraLite form.

11.2 Process Control

VeraLite 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.

11.3 $wait_child()

The $**wait_child()** system task is used to ensure that all child processes (processes created by the calling process) have completed their execution.

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

```
task $wait_child();
```

Calling **$wait_child()** causes the calling process to block until all its sub-processes have completed.
By default, VeraLite terminates a simulation run when all its programs finish executing (i.e, they reach the end of their execute block), regardless of the status of any child processes. The **$wait_child()** task allows a program to wait for the completion of all its concurrent threads before exiting.

Example: In the task *do_test*, the first two processes are spawned and 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** 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
     fork
          exec3();
          exec4();
```

```
        join none
        $wait_child();  // block until exec1 ... exec4 complete
    endtask
```

11.4 $terminate

The terminate statement terminates all active descendants (sub-processes) of the calling process. The syntax for **$terminate** is:

```
$terminate;
```

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

Example: In the code 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 return 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;
endfunction
```

Verilog supports the **disable** construct, which will end a process when applied to the named block being executed by the process. However, **$terminate** differs from **disable** in that **$terminate** 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** will end only those processes that were spawned by the calling thread.

11.5 $suspend_thread()

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

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

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

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 non-blocking assignments (see Section 14.5) where a zero delay is ill-advised.

Example: This 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 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
```

12 Inter-Process Synchronization and Communication

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.

VeraLite brings to SystemVerilog a powerful and easy-to-use set of synchronization and communication mechanisms, all of which can be created and reclaimed dynamically. VeraLite adds a **semaphore** primitive, which can be used for synchronization and mutual exclusion to shared resources, and a **mailbox** primitive that can be used as a communication channel between processes. VeraLite also enhances Verilog's named **event** data type to satisfy many of the system-level synchronization requirements. Lastly, VeraLite adds the **wait_var** mechanism that can be used to synchronize processes using dynamic data.

12.1 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()**
— Obtain a key from the bucket: **get()**
— Return a key into the bucket: **put()**
— Try to obtain a key without blocking: **try_get()**

12.1.1 new()

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

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

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

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.

12.1.2 put()

The semaphore **put()** method is used to return keys to a semaphore.
The syntax for **put()** is:

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

keyCount specifies the number of keys being returned to the semaphore. The default is 1.

When the **semaphore.put()**task is called, the specified number of keys are returned to the semaphore. If a process has been suspended waiting for a key, that process will execute if enough keys have been returned.

12.1.3 get()

The semaphore **get()** function is used to procure a specified number of keys from a semaphore.
The syntax for **get()** is:

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

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

If the specified number of keys are available, the 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).

12.1.4 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);
```

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 1 and execution continues. If the specified number of key are not available, the function returns 0.

12.2 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, VeraLite's mailboxes 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()**

12.2.1 new()

Mailboxes are created with the **new()** method.
The syntax for mailbox **new()** is:

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

The **new()** function returns the mailbox identifier, 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.

12.2.2 num()

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

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

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

12.2.3 put()

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

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

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

The *message* is any scalar (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.

12.2.4 try_put()

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

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

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

The *message* is any scalar (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.

12.2.5 get()

The **get()** method retrieves a message from a mailbox.
The syntax for **get()** is:

```
task get( var scalar message );
```

The *message* can be any scalar (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.

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

12.2.6 try_get()

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

```
function int try_get( var scalar message );
```

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

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

12.2.7 peek()

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

```
task peek( var scalar message );
```

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

The **get()** 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.

12.2.8 try_peek()

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

```
function int try_peek( var scalar message );
```

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

The **try_peek()** method tries to copy one message from the mailbox without removing the message from themailbox queue. If the mailbox is empty then the function returns 0. If there is a type mismatch between the *message* variable and the message in the mailbox, the function 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 returns 1.

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

12.3 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 7.18), modules, and interfaces:

mailbox#(**type =** ***dynamic_type***)

Where *dynamic_type* represents a special type that enables run-time type-checking (the default).

A parameterized mailbox of a specific type is declared by specifying the type:

```
typedef mailbox #(string) s_mbox;

s_mbox   sm = new;
string      s;

sm.put( "hello" );
…
sm.get( s );     // s <- "hello"
```

Parameterized mailboxes provide all the same standard methods as *dynamic* mailboxes:

num	**get**	**put**	**try_peek**
new	**peek**	**try_get**	**try_put**

The only difference between a generic (dynamic) mailbox and a parameterized mailbox is that for a parameterized mailbox the compiler ensures that all **put** and **get** calls are compatible with the mailbox type so that all type mismatches are caught by the compiler and not at run-time.

12.4 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 VeraLite event that uses a ONE_SHOT trigger. But, a VeraLite **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 VeraLite events have a value (ON, OFF) and a persistency that can be controlled via the trigger options. Also, VeraLite 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, VeraLite 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 VeraLite **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.

12.4.1 $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_id*^{1}, ..., *ev_id*^{N} 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.

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 12.4.2).
	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.

12.4.2 $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_id[1], ..., ev_id[N] );

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

Where ev_id^1, ..., ev_id^N 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.

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.

12.5 Event Variables

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

12.5.1 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.

12.5.2 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.

12.6 $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:

```
task $wait_var(scalar variable[1],..., variable[N]);
```

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.

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:

```
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.
```

Example: 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
```

13 Clocking Domains

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. VeraLite 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.

13.1 Clocking Domain Declaration : clocking

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

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

cloking_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*]

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

edge ::= **posedge** | **negedge**

delay_expression ::= *unsigned_number* | *time_literal*

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 13.2). 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:

```
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 10ns input skew and a 2ns output skew. The next line adds three input signals to the domain: data, ready, and enable;the last signal refers to the hierarchical signal top.mem1.enable. The fourth line adds the signal ack to the domain, and overrides the default output skew so that ack is driven on the negative edge of the clock. The last line adds the signal addr and overrides the default input skew so that addr is sampled one step before the positive edge of the clock.

Unless otherwise specified, the default **input** skew is **1step** and default **output** skew is **0**. A **step** is a special time unit defined to be the smallest possible delay throughout the simulation. A **1step**

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).

13.2 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 A shows the basic sample/drive timing for a positive edge clock.

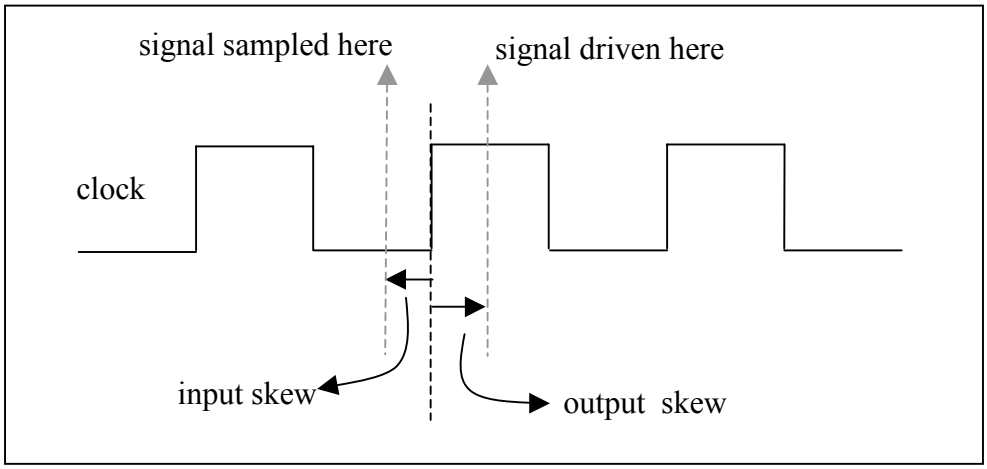

Figure A: Sample and drive times including skew with respect to the positive edge of the clock.

A skew must be a constant expression and can be specified either as an unsigned integer value or as a time literal. 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
```

An input skew of **1step** indicates that the signal is to be sampled an infinitesimal *delta* before the clock event. That is, the value sampled is always the signal's last value immediately before the corresponding clock edge.

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

An input skew of **`#0`** forces a skew of zero. Input signals with zero skew are sampled at the same time as their corresponding clock edge, but to avoid races the sampling is done *after* all non-blocking assignments (NBA) have been processed (see Section 14.5). Likewise, output signals with zero output skew are driven at the same time as their specified clock edge, but immediately before *read-only synchronize time* (before advancing time). A detailed explanation for this event ordering is covered in Section 14.5.

13.3 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
```

13.4 Signal in Multiple Clocking Domains

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.

13.5 Clocking Domain Scope and Lifetime

A **clocking** construct is both a declaration and an instance of that declaration. A separate instantiation step is not necessary. Once declared, the clocking signals are available via the clock-domain name and the dot (.) operator:

```
dom.sig      // signal sig in clocking dom
```

Clocking domains cannot be nested. They cannot be declared inside functions or tasks, or at the global ($root) level. Clocking domains can only be declared inside a module or a program (see Section 14).

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

13.6 Multiple Clocking Domain Example

In this example, a simple test module includes two clocking domains.

```
program[3] 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 @(posedege phi2);
        input #2 output #4ps cmd;
        input enable;
    endclocking

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

And, 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
```

13.7 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);
```

[3] The **program** construct is discussed in Section 14. In this example it can be considered a module.

```
    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 @(posedege a.clk);
        input a.data;
        output a.write;
        inout state = top.cpu.state;
    endclocking

    clocking cd2 @(posedege 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
```

And, 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 @(posedege a.clk);
        input data = a.data;
        output write = a.write;
        inout state = top.cpu.state;
    endclocking

    clocking cd2 @(posedege b.clk);
        input #2 output #4ps cmd = b.cmd;
        input enable = b.enable;
    endclocking
```

And this would allow using the shorter names (cd1.data, cd2.cmd, …) instead of the longer interface syntax (cd1.a.data, cd2.b.cmd,…).

13.8 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).

13.9 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 13.9.1). 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
```

13.9.1 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
```

Signal Operations

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 signal operations to be written very succinctly, without explicitly using clocks or specifying timing. These signal operations are:

— Synchronization
— Sampling
— Driving

13.10 Synchronization

Explicit synchronization is done via the @ operator, which allows a process to wait for an explicit signal value change.

The syntax is for the synchronization operator is:

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

Where *specific_edge* identifies the edge at which the synchronization occurs and can be:
— **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 **posedege** 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.

These are some example synchronization statements:

— Wait for the next change of signal ack_1 of clock domain ram_bus

```
@(ram_bus.ack_1);
```

— Wait for the next clocking event in clock-domain ram_bus

```
@(ram_bus);
```

— Wait for the positive edge of the signal ram_bus.enable

```
@(posedge ram_bus.enable);
```

— Wait for the falling edge of the specified 1-bit slice dom.sign[a]. Note that the index *a* is evaluated at runtime.

```
@(negedge dom.sign[a]);
```

— Wait for either the next positive edge of dom.sig1 or the next change of dom.sig2, whichever happens first.

```
@(posedge dom.sig1 or dom.sig2);
```

— Wait for the either the negative edge of dom.sig1 or the positive edge of dom.sig2, whichever happens first.

```
@(negedge dom.sig1 or posedge dom.sig2);
```

The values used by the synchronization primitive are the synchronous values, that is, the values sampled at the corresponding clocking event.

13.11 Signal 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 A).

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.

13.12 Signal Drives

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:

```
   @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:

```
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.

13.12.1 Blocking and Non-Blocking Drives

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 non-blocking 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 ) begin
   bus.data <= 27;
   y = bus.data;       // y is 31 (not 27)
end
```

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.

13.12.2 Drive Value Resolution

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).

13.12.3 Drive / Assignment Ambiguity

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

```
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.

14 Program Block

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 SystemVerilog **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 SystemVerilog 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.

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:

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

For example:

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

or

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

The `list_of_ports` allowed by a program is the same as the one allowed for any SystemVerilog module. A more complete example is included in Sections 13.6 and 13.7.

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 to, it simply extends the rules set forth by SystemVerilog, which disallows automatic variables from triggering event expressions or be written using non-blocking assignments. Likewise, all the dynamic test-bench constructs – objects handles, dynamic and associative arrays, strings, and events – will be limited to the procedural context.

14.1 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, VeraLite allows static data (including static class members) to specify an initial value as part of their declaration, and, like SystemVerilog, VeraLite requires that all such data be initialized before the *program* begins execution. It is important to note that VeraLite 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 7.4), or a **Mailbox** may be initialized by calling its **new** method (see Section 12.2.1). While this does not represent a conflict with SystemVerilog, 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.

14.2 Scope and Lifetime

The following test-bench constructs all have module or program scope. They all 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.

Tasks and Functions cannot be nested within themselves, but they can contain block statements that do 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.

14.3 Multiple Programs

It is allowed to have any arbitrary number of program definitions or instances. The programs can be fully independent (without inter-program communication), or cooperative. Users can control the degree of communication by choosing to share data via $root or making the data private by declaring it inside the corresponding program block.

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.

14.4 Eliminating Zero-Skew Races

If both input and output skews are set to `#0` 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* non-blocking 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 13.12.1.
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.

14.5 Eliminating Races and SystemVerilog Event Queue

There are two major sources of nondeterminism 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 nondeterminsm 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 *non-blocking assignments* have been processed, thus, treating all transitions towards the steady state in the same consistent manner (from the testbench 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.

In order to standardize the cycle behavior, the execution after non-blocking assignments described above must be added to the SystemVerilog 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.

14.6 Blocking Tasks in Cycle/Event mode

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 non-blocking assignments.

14.7 Program Control Tasks

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

14.7.1 $exit()

Each program can be finished by calling the **$exit()** system task. When all programs exit, the simulation finishes.

The syntax for the **$exit()** system task is:

```
task $exit();
```

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

15 Linked Lists (8-1)

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.

15.1 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.

15.2 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)
```

15.2.1 Declaring Lists 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.

15.2.2 Declaring List Iterators

You must declare all list iterators 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.

15.3 Size Methods

This section describes the list methods that analyze list sizes.

15.3.1 size()

The **size()** method returns the number of elements in the list container:

```
list1.size();
```

15.3.2 empty()

The **empty()** method returns 1 if the number elements in the list container is 0:

```
list1.empty();
```

15.4 Element Access Methods

This section describes the list methods used to access list elements.

15.4.1 front()

The **front()** method returns the first element in the list:

```
list1.front();
```

15.4.2 back()

The **back()** method returns the last element in the list:

```
list1.back();
```

15.5 Iteration Methods

This section describes the list methods used for iteration.

15.5.1 start()

The **start()** method returns an iterator pointing to the first element in the list:

```
list1.start();
```

15.5.2 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()**.

15.6 Modifying Methods

This section describes the list methods used to modify list containers.

15.6.1 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.

15.6.2 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.

15.6.3 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();
```

15.6.4 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.

15.6.5 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.

15.6.6 erase_ra0nge()

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.

15.6.7 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.

15.6.8 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.

15.6.9 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.

15.6.10 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.

15.6.11 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.

15.6.12 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.

15.7 Iterator Methods

This section describes the methods used by iterators.

15.7.1 next()

The **next()** method moves the iterator so that it points to the next item in the list:

```
I1.next();
```

15.7.2 prev()

The **prev()** method moves the iterator so that it points to the previous item in the list:

```
I1.prev();
```

15.7.3 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.

15.7.4 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.

15.7.5 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).