System Verilog 3.1 Donation
Part I: Testbench

Version 1.1, May 2002

SYNOPSYS®

1

VeraLite: The Language

This chapter covers the basics of VeraLite. It introduces the lexical elements and some of the basic components of the language. The following sections are included:

- Lexical Elements
- Data Types and Variable Declaration
- Arrays
- Enumerated types
- Operators
- Variable Assignment

Lexical Elements

VeraLite source code consists of a stream of lexical elements. Lexical element types are:

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

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.

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 finishes out the line. The syntax is:

```
any_vera_statement; //One line comment
```

A block statement starts with a /* and ends with a */. Everything between the start and end tags is a comment.

The syntax is:

```
/*Blocks of comments
can take up
multiple lines */
```

Note:

Block comments *cannot* be nested.

Statement Blocks

VeraLite supports two methods of creating statement blocks: braces, and fork/join.

The syntax for statement blocks using braces is:

```
{
     // vera_statements
}
```

Note:

An empty statement is not legal in VeraLite. For example, the following generates a parse error:

```
if(1)
;
else
;
```

The syntax for fork/join statement blocks is:

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

Forks and joins are discussed in more detail in fork and join on page 4-2.

Identifiers

An identifier is a sequence of letters [a-zA-Z], digits[0-9] and underscores[_]. Identifiers are case-sensitive and cannot begin with a digit.

Strings

A string is a sequence of characters enclosed by double quotes. A string 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 new line are ignored. There is no maximum string size limit for constants.

Numbers

In VeraLite, a number can be formed using either the **LIT_INTEGER** or **NUMBER** format.

The **LIT_INTEGER** format is a simple decimal number specified as a sequence of digits from 0 to 9. Negative signs are allowed to specify negative integers. Underscores are ignored and may be used for clarity. The upper limit for integer sizes is dependent on the host machine, but it is generally 32 bits. The syntax is:

```
[0123456789]+
```

The **NUMBER** format takes these forms:

```
<size>'<base><number>
```

<size>:

The <*size*> specifies the number of bits in the number. If the <*size*> is omitted, the number of bits for <*number*> defaults to the host machine word size. A plus or minus sign before the <*size*> specification signifies the number's polarity. The maximum size is 65535.

<base>:

The <*base*> is always preceded by a single quote (‘). The <*base*> can be one of the following: d(ecimal), h(exadecimal), o(ctal), or b(inary). The base identifier can be either upper or lower case.

<number>:

- The valid elements of <*number*> for each <*base*> are:

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

The X and x represent unknown values, and Z and z represent high impedance values in binary, octal, or hexadecimal form. Underscores are ignored.

If the most significant specified digit of a <*number*> representation is an x or a z, the VeraLite compiler extends the x or z to fill the higher order bits or digits.

For example, 8’bx is equivalent to 8’bxxxxxxxx, 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.

Data Types and Variable Declaration

VeraLite's standard data types are:

- integer
- bit
- string
- event

VeraLite's user-defined data types are:

- Enumerated types
- virtual port (port)
- class

All basic types can be declared as class members and can be used to form associative and non-associative arrays.

The term **scalar** is used in this document to refer to a disjunctive list of the data types: "integer," "bit," "bit[]," and "enum." For example, instead of the following four prototypes:

```
function integer function_name(...);
function bit function_name(...);
function bit[msb:lsb] function_name(...);
function enum function_name(...);
```

As a shorthand notation, **scalar** is specified:

```
function scalar function_name(...);
```

Variables are seen globally if they are declared at the top (program) level. If they are declared in blocks (begin/end, {}, fork/join), they are seen locally.

Standard Data Types

integer

Integers are signed variables. The upper limit for integer sizes is dependent on the host machine. On 32 bit machines, the allowed range is between -2^{31} and 2^{31} -1. An integer may become `X` (unknown) when it is not initialized or when an undefined value is stored.

The syntax to declare an integer is:

```
integer variable_name [=initial_value];
```

variable_name:

The *`variable_name`* can be a valid identifier.

initial_value:

Specifying the *`initial_value`* is optional.

For expressions involving both bit and integer types, the integer types are first converted to 32-bit unsigned integers.

bit

Bits can have the value *0*, *1*, *z*, or *x*.

Table 1-1 Value Levels and Conditions

Value Level	Condition
0	logic 0
1	logic 1
z	high impedance
x	unknown

The syntax to declare a bit is:

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

variable_name:

The *variable_name* can be a valid identifier.

initial_value:

Specifying the *initial_value* is optional.

VeraLite also supports bit fields.

The syntax to declare a bit field is:

```
bit [high:0] variable_name [=initial_value];
```

High:

High specifies the upper limit on the field. The maximum size of a bit field is 65535 bits. When declaring bit fields, you cannot use variables for the high specifier.

variable_name:

The *variable_name* can be a valid identifier.

initial_value:

Specifying the *initial_value* is optional.

Note:

Verilog users: The keyword, **reg**, is equivalent to the keyword, **bit**. Therefore, **reg** and **bit** can be used interchangeably.

string

Strings are character data types that have a wide range of operators associated with them for manipulating characters.

The syntax to declare a string is:

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

variable_name:

The *variable_name* can be a valid identifier.

initial_value:

Specifying the *initial_value* is optional.

String operators are discussed in on page 1-28.

event

Events are pointers to a synchronization object. A synchronization object can be either ON or OFF. An event can point either to an object or be null. Events are passed as arguments to method calls to specify the trigger point. Two or more events can point to the same synchronization object.

The syntax to declare an event is:

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

variable_name:

The *variable_name* can be a valid identifier.

initial_value

The *initial_value* can be either null or another event. The default *initial_value* is a new synchronization object set to OFF. Events assigned to null act as if always ON.

User-Defined Data Types

Enumerated types

Enumerated types are named, integer constants.

The syntax to declare an enumerated type is:

```
enum category = list;
```

OR

```
enum category {list};
```

category:

The *category* is the name of the enumerated type. It is used to assign list values to variables.

list:

list is a list of category values separated by commas. They are assigned sequential integer values in the order listed.

Enumerated types cannot be declared inside classes or subroutines.

Enumerated types are discussed in more detail in “Enumerated Types” on page 1-18.

class

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 subroutines are called *methods*. The properties and methods, taken together, define the contents and capabilities of a class instance or object. Classes are discussed in more detail in Chapter 11.

Arrays

VeraLite supports one-dimensional and multi-dimensional arrays, which are lists of variables that are all of the same type and called with the same name. Arrays in VeraLite can be static (global) or dynamic (local). You can also create associative arrays that have the advantage that each entry of the array is allocated only when it is accessed.

Fixed-size Arrays

Any variable type can be declared as an array.

The syntax to declare arrays is:

```
integer array_name[size];
bit [high:0] array_name[size];
port_name array_name[size];
event array_name[size];
```

size:

The *size* specifies the number of elements in the array. The maximum number of elements in an array is 2^{31}-1 elements. For larger arrays, you should use associative arrays.

Accessing an array with an unknown bit (‘x’) in the index causes a simulation error. Also, writing to an array with an unknown in the index is ignored, and reading with an unknown in the index returns ‘X’s.

Note that a bit field of an array element cannot be referenced directly. To reference a bit field of an array element, use a temporary variable. For example:

```
tmp = memory[42];
```

```
if (tmp[3:2] == 0) ...
```

Array Initialization

An array can be initialized when declared. The values used for array initialization are subject to the same limitations as the initialization of scalar variables.

Single-dimensional array example:

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

Table 1-2

Data Types	Supported
integer	yes
bit	yes
enum	yes
string	yes
event	no
bind	no
port	no
object	no

Concatenation is not supported in array initialization. An attempt to concatenate will result in a compilation error.

Example of an illegal declaration:

```
#define OPCODE 8'ha
bit [16:0] array1[3] = { {OPCODE, 8'h00}, {OPCODE, 8'h01}, {OPCODE,
      8'h02}};
```

Note:

In VeraLite v1.0, you cannot initialize an array in the declaration.

Multi-dimensional Arrays

The declaration of a multi-dimensional array variable is similar to that of a single dimensional array, with the addition of multiple dimensions after the variable name. Any data type that can be used for a single dimensional array can also be used for a multi-dimensional array.

Examples:

```
integer matrix [2][5];
Color colors [3][4][2];
event myevent [2][2];
```

The following program illustrates the use of a three dimensional array.

```
task cube_add(integer cube[2][2][2], integer offset)
{
      integer i, j, k;

      for (i=0; i<2; ++i){
            for (j=0; j<2; ++j){
                  for (k=0; k<2; ++k){
                        cube[i][j][k] += offset;
                  }
            }
      }
}

program array
{
      integer cube[2][2][2], i, j, k;

      for (i = 0; i < 2; ++i) {
            for (j = 0; j < 2; ++j) {
                  for (k = 0; k < 2; ++k) {
                        cube[i][j][k] = i+j+k;
                  }
            }
      }

cube_add(cube,4);
```

```
for (i = 0; i < 2; ++i){
      for (j = 0; j < 2; ++j) {
            for (k = 0; k < 2; ++k) {
                        printf("cube[%d][%d][%d] = %d\n", i, j, k,
                              cube[i][j][k]);
                  }
            }
      }
}
```

When referencing elements in a multi-dimensional array, multiple indices must be specified as follows:

```
vname[index_1]...[index_n]
```

When passing a multi-dimensional array as a parameter to a function, the formal argument must be of the same type as the parameter passed in.

For example, the declaration:

```
task fun(integer x[2][2])
```

creates a task “fun” that takes one parameter, a two dimensional array where each dimension is two. Any call to “fun” must pass in a two dimensional array where each dimension is two.

Also, the multi-dimensional array does not support bit slicing and associative array declarations. An associative array can only have one dimension.

The following generates compilation errors:

```
integer assoc_matrix[][2]; //Invalid
integer double_assoc_matrix[][]; //Invalid
```

Array Initialization

The values used for array initialization are subject to the same limitations as the initialization of scalar variables. For example:

```
integer x[2][2]=
{
     {0,1},
     {2,3}
};
```

Initialization is identical to how C and C++ initialize multi-dimensional arrays. The order of the data being loaded for the above example is:

```
x[0][0], x[0][1],
x[1][0], x[1][1]
```

Table 1-3

Data Types	Supported
integer	yes
bit	yes
enum	yes
string	yes
event	no
bind	no
port	no
object	no

Concatenation is not supported in array initialization. An attempt to concatenate will result in a compilation error.

Example of an illegal declaration:

```
#define OPCODE 8'ha
bit [16:0] array1[3] = { {OPCODE, 8'h00}, {OPCODE, 8'h01}, {OPCODE,
      8'h02}};
```

Associative Arrays

Associative arrays are arrays whose dimensions are not specified. The syntax to declare associative arrays is:

```
integer array_name[];
bit [high:0] array_name[];
port_name array_name[];
event array_name[];
```

Array elements in associative arrays are allocated dynamically, when you access a particular address. The array index tracks those elements that have been assigned values and stores those values within the array. The index is an unsigned number with a maximum value of 2^64-2. When using integer and bit associative arrays, if you try to access an element that has not been assigned a value, an 'X' is returned.

Note:

Using associative arrays slightly slows down simulation time. The effect is usually unnoticeable.

Users can implement the system function, **assoc_index()** to manipulate or analyze associative arrays.

The syntax for assoc_index is:

```
function integer assoc_index(CHECK | DELETE | FIRST | NEXT,
      assoc_array_name [, var bit[63:0] index]);
```

Key words:

CHECK, DELETE, FIRST, or NEXT determines the function of **assoc_index()**.

Table 1-4

Option	Description
CHECK	Checks if an element exists at the specified index within the array. If it does, a 1 is returned. If it does not, a 0 is returned. If the index is omitted, the function returns the number of allocated elements in the array.
DELETE	Deletes the element specified at the specific index. If it is successful, a 1 is returned. If the element does not exist, a 0 is returned. If the index is omitted, all the elements in the array are deleted. Only the array elements are deleted, and not the array itself.
FIRST	Returns the element associated with the first valid index. The index is assigned the value of the first valid element in the array. The function returns a 0 if it fails, and 1 if an element is returned.
NEXT	When NEXT is used, the function searches for the first valid array element with an index greater than the passed parameter index. If an element is found, the function returns 1, and assigns the new index to the parameter index. If none exists, it leaves the value of index unchanged, and returns to 0.

assoc_array_name:

The *assoc_name* is the name of the associate array being analyzed. It must be a valid array reference.

index:

The *index* is the numerical index of the element being analyzed.

The function **assoc_index()** returns an integer (1 or 0), or void.

- A “1” is returned when the function call is successful.
- A “0” is returned when the function call is unsuccessful.

In the case of **assoc_index()**, it is not mandatory to assign the return value to a variable.

Enumerated Types

Enumerated types are a user-defined list of named integer constants. As discussed in “Enumerated types” on page 1-10

The syntax to declare an enumerated type is:

```
enum category = list;
```

OR

```
enum category {list}
```

For example:

```
enum colors = red, green, blue, yellow, white, black;
```

This operation assigns a unique number to each of the color identifiers, allowing us to create a new data type of type colors.

```
colors new_color;
integer val;
new_color = green;
new_color = 1; // Invalid assignment.
```

This example assigns the color green to the colors variable new_color. The second assignment is invalid because of the strict typing rules used by enumerated types.

Different enumerated types cannot share the same name. For instance, you cannot define an element called RANDOM for two different enumerated type categories list and packet. RANDOM can only be defined in one of the categories, not both.

Elements within enumerated type definitions are assigned identifiers, which are numbered consecutively, starting from 0. In our example, red is assigned 0, green is assigned 1, and so on.

Any explicit value in an enumerated type declaration affects all subsequent enums without an explicit value.

You can further specify identifiers in the element list in several ways:

- name: This associates the next consecutive integer with name.
- name[N]: This generates `N` names in the sequence (name0, name1, ..., nameN-1) where N must be a constant integer.
- name[n:m] :This creates a sequence of names starting with namen and counting up (or down) to namem.
- name=N: This assigns the constant `N` to `name`.

For example:

```
enum instructions = 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.

Enumerated Types in Numerical Expressions

Elements of an enumerated type or an enumerated type variable can be used in numerical expressions. The value used in the expression is the numerical value assigned to the enumerated type element.

For example:

```
colors new_color;
integer val1, val2;

val1 = blue * 3;
new_color = yellow;
val2 = new_color + green;
```

From our previous declaration, `blue` has a numerical identifier of `2`. This example assigns `val1` a value of `6` (2*3). This example then assigns `val2` a value of `4` (3+1).

Note:

Assignments to enumerated type variables are strongly typed. Thus, assigning numerical expressions to enumerated type variables causes compilation errors.

Increment and Decrement Operations on Enumerated Types

The operators ++, --, +=, and -= have special meanings on enumerated type variables.

Table 1-5

Enum Variable/ Operator	Assignment
`enum_var++`	Assigns the next member (as defined by the definition order) to *enum_var*. The first member is selected if *enum_var* is currently holding the last member
`enum_var--`	Assigns the previous member (as defined by the definition order) to *enum_var*. The last member is selected if *enum_var* is currently holding the first member.
`enun_var+=val`	Assigns the *val*-th next member to *enum_var*. A wrap to the beginning of the list occurs when the end of the list is reached.
`enum_var-=val`	Assigns the *val*-th previous to *enum_var*. A wrap to the end of the list occurs when the beginning of the list is reached.

Note:

"enum_var += val;" is different than "enum_var = enum_var + val;" The former is legal while the latter is illegal because "enum_var + val" evaluates to a numerical expression which cannot be assigned to an enumerated type variable.

Operators

VeraLite uses a set of standard operators for expressions and concatenation. The increment (++) and decrement (--) operators behave just like C/C++. All VeraLite operators which are defined in Verilog work the same way as the Verilog operators. For example:

```
-1%4 = -1
```

Table 1-6 lists the basic VeraLite operators.

Table 1-6 VeraLite Operators

Operator	Semantics
{}	concatenation
´{}	concatenation left of assignment
+ - * /	arithmetic
%	modulus
++ --	increment, decrement
> >= < <=	relational
+= -=	add and assign, subtract and assign
=	assignment
!	logical negation
&&	logical and
||	logical or
==	logical equality
!=	logical inequality
===	case equality
!==	case inequality
=?=	wild equality
!?=	wild inequality
~	bitwise negation
&	bitwise and
&~	bitwise nand
|	bitwise or

Table 1-6 VeraLite Operators(Continued)

Operator	Semantics
|~	bitwise nor
^	bitwise exclusive or
^~	bitwise exclusive nor
&	unary and
~&	unary nand
|	unary or
~|	unary nor
^	unary exclusive or
~^	unary exclusive nor
<<	left shift
>>	right shift
?:	conditional

Operator Precedence

The precedence order of VeraLite operators is defined in Table 1-7.

Table 1-7 Precedence Order of VeraLite Operators

Operator	Precedence
()	Highest precedence
.	
++ --	
& ~& | ~| ^ ~^	
* / %	
+ -	
<< >>	
< <= > >=	
=?= !?= == != === !==	
& &~	
^ ^~	
| |~	
&&	

Table 1-7 Precedence Order of VeraLite Operators(Continued)

Operator	Precedence
`||`	
`?:`	
`= += -= *= /= %=`	
`<<= >>= &= |= ^= ~&= ~|= ~^=`	Lowest precedence

All operators associate left to right. That is, if multiple operators with the same precedence are used (as in A + B - C), the expression is evaluated left to right (A + B, then - C). When operators differ in precedence, the highest precedence operator is executed first. Parentheses change the operator precedence.

Arithmetic Operators

The unary arithmetic operators (+ and -) take precedence over the binary arithmetic operators (+, -, *, /, and &).

If an operand has any bit with a value of x, the entire result is x.

Relational Operators

The relational operators are:

- a < b (a less than b)
- a > b (a greater than b)
- a <= b (a less than or equal to b)
- a >= b (a greater than or equal to b)

The relational operators yield a scalar value of 0 if the relation is false, or a 1 if the relation evaluates to true. If there is are unknown bits in the relation (a value of x), the relation yields an unknown value (x).

Note that relational operators have a lower precedence than the arithmetic operators.

Equality Operators

The equality operators are:

- a === b (a equal to b, including x and z values)
- a !==b (a not equal to b, including x and z values)
- a == b (a equal to b, not including x or z values)
- a != b (a not equal to b, not including x or z values)
- a =?= b (a equals b, x and z values are wildcards)
- a !?= b (a not equal to b, x and z values are wildcards)

The wild equality operator (=?=) and inequality operator (!?=) treat an *x* value or z value in a given bit position (for bit values) as a wildcard. They match 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. If the operands are not the same length, 0's fill the empty spaces. If the relation is true, the operator yields a 1. If the relation is false, it yields a 0.

If the operands have an x or z value, the result is unknown (x) when using the == and != operands. When using the === and !== operands, x and z values must match exactly.

Logical AND and Logical OR Operators

The logical AND and OR operators are:

- && (AND)
- || (OR)

The AND and OR operators are logical connectives. Expressions connected by these operators are evaluated left to right. If the relation is true, the operation yields a 1. If the relation is false, it yields a 0. If the result is unknown, it yields an unknown value (x).

Bitwise Operators

Bit-wise operators compare 1 bit in one operand to the equivalent bit in another operand to calculate 1 bit for the result. If the operands are not the same length, the smaller operand is zero-filled in the most significant bit positions. The operator logic tables follow.

~	
0	1
1	0
x	x

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x

|	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x

^~	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x

Reduction Operators

The reduction operators perform a bit-wise operation on a single operand and yield a single bit result. The first step applies the operator between the first and second bits of the operand. Subsequent calls apply the operator between the result and the next bit in the operand. The logic tables for the reduction operators are the same as for the bit-wise operators. The logic tables for the AND, OR, NAND, NOR, exclusive OR, and exclusive NOR follow.

	&	|	~&	~|
no bits set	0	0	1	1
all bits set	1	1	0	0
some bits set	0	1	1	0
bit vector of 1, x, and z	x	1	x	0
bit vector of 0, x, and z	0	x	1	x

	^	~^
odd number of bits set	1	0
even number of bits set (or none)	0	1

Conditional Operator

The conditional operator follows this format:

```
conditional_expression ::=  expression1 ? exprssion 2:
     expression3
```

If `expression1` evaluates to true (known value other than 0), then `expression2` is evaluated and used as the result. If `expression1` evaluates to false, (0) then `expression3` is evaluated and used as

the result of the conditional expression. If `expression1` evaluates to an ambiguous value (x or z), then both `expression2` and `expression3` are evaluated and their results are combined, bit by bit, using the ?: truth table to calculate the final result (unless `expression2` or `expression3` is real, in which case the result is 0). If the lengths of `expression2` and `expression3` are different, then the shorter operand is lengthened to match the longer and zero filled from the left. This logic table shows the results of unknown conditional statements.

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

Operators for manipulating strings

VeraLite provides a set of operators that can be used to manipulate combinations of string variables and string constants. Table 1-8 lists the valid operators.

Table 1-8 Valid VeraLite Operators

Operator	Meaning
`==`	Check equality of two strings
`!=`	Check inequality of two strings
`{str1,str2..}`	Generate a concatenated string with str1, str2, ...
`{num{str}}`	Generate a string duplicated num times.

Note:

You can compare string variables to null.

Concatenation

The syntax for concatenation is:

```
{var1, var2, ..., varN}
```

The result of the concatenation is *variable*. The arguments are concatenated sequentially. Any combination of valid data types (integer, bit, string and enum) is valid for concatenation.

For example:

```
bit [6:0] data;
bit parity;
bit [7:0] foo;
foo = {data, parity};
```

Multiple concatenation is also supported:

```
{var1 {var2}}
```

or

```
{var1,...,varN}
```

For example:

```
{ 32 {1'b1 } }
{ 4 {addr, data} }
```

This example concatenates `addr` and `data` four times.

VeraLite uses the left brace to both open a block, and for concatenation. This creates a conflict when using the left brace for concatenation on the left-hand side of assignments. Therefore, VeraLite uses a single quote to prefix the left brace when it is used for concatenation on the left. For example:

```
'{data,packet,parity} = 256'b0;
```

Variable Assignment

Variable assignment is the primitive operation to set a value for a variable.

The syntax to assign values to variables is:

```
variable_name operator assign_expression
```

For example:

```
i = 0;
a = 1'b0;
temp[3:0] = 4'b1000;
memory [53] = 8'b0x0x0x0x;
for (i = 0; i < 10; i += 2)
```

There are two types of assign operators: the = operator is called the simple assignment operator; all others are called compound assignment operators.

For the compound assignment operator, the expression a *operator* = b is equivalent to a = a *operator* b.

For example, the following are equivalent:

```
i = i + 5;
i += 5;
```

VeraLite supports the C-style ++ and -- operators.

For example:

```
result = 5;
a = result++;
a = ++result;
```

The second line accesses the variable and then increments it. The third line increments the variable and then accesses it.

VeraLite does not support assignment recursion.

This is an illegal assignment:

```
a = b = c;
```

The VeraLite source code is type checked at compile time. Event variables cannot be used in assignments because they are used only for triggering purposes. All other combinations of integer and bit variables are valid.

Note:

To help avoid mistakes, *assignments* are not expressions. Hence, statements like these are invalid:

```
if(a=b) a=c+d;        // should be a==b
while(a=b) a=$random(); // should be a==b
```

2

Programming Overview

This chapter documents the basic elements of VeraLite programming. It details the fundamental program structure used in all VeraLite programs.

This chapter includes these sections:

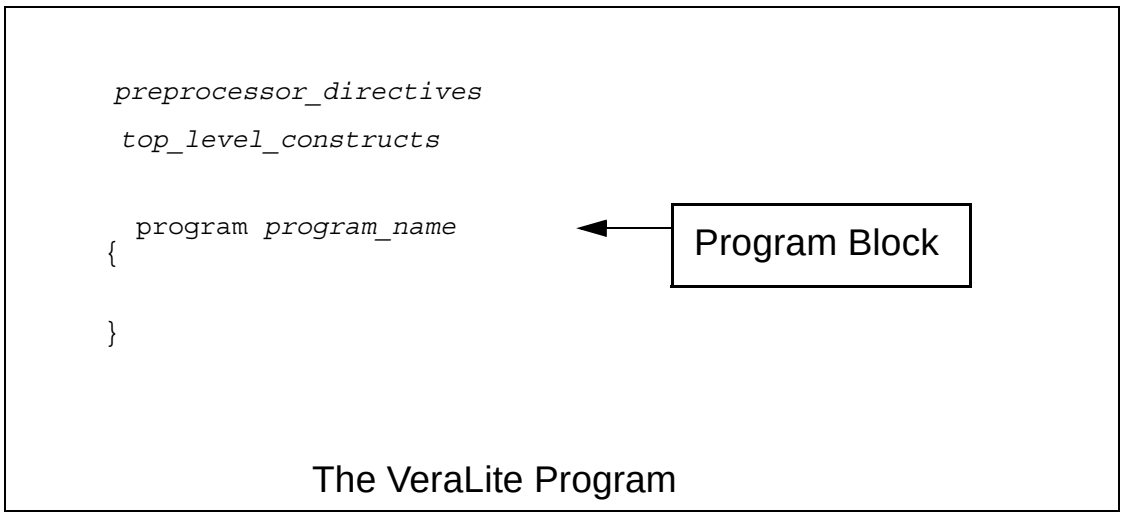
- Overview
- Subroutines

Overview

A program involves the integration of several key components of a testbench. The constituents are:

- a required program block,
- preprocessor directives,
- top level constructs.

The program block, the various constructs and the preprocessor directives can occur in any order.



The VeraLite Program

Program Block

The program block is indicated by the keyword, **program**.

The program block contains:

```
program program_name

{
     variable declarations
     program block code
}
```

The main program block is where:

- global variables are declared,
- executable statements are carried out,
- calls to subroutines are made.

VeraLite, like C, supports both top level and lower level scope. Variables declared in the main program block are global, whereas any variable defined in a task or function has local scope.

Top level constructs

A program can have any number of the top level constructs:

- class prototypes
- extern declarations
- enumerated type definitions
- class definitions
- subroutines

- HDL subroutines
- interface declarations
- system clock definitions
- port definitions
- bind definitions
- coverage_definitions definitions

Preprocessor Directives

The preprocessor directives:

- #define text_macro
- #include "*filename*"
- #include <vera_defines.vrh>

can occur anywhere in the program.

Referencing Variables

Forward reference is allowed with interface signals, global variables, and task and function calls that are present at the top level.

Enumerated types, classes, coverage_def, functions, global variables, hdl_tasks and tasks defined in other files require an extern declaration in the file referencing that symbol (see External Declarations on page 2-13).

Subroutines

VeraLite supports two means of encapsulating often-executed program fragments: functions and tasks. All functions and tasks are re-entrant and therefore can be called recursively.

VeraLite subroutines definitions cannot be nested. This means that all subroutine declarations must be made at the top level. Since tasks and functions are global by default, VeraLite supports declaring subroutines as **local**. They are then separately compiled.

You can also declare external subroutines. These separately compiled object files are linked at simulation time.

This section includes:

- Functions
- Tasks
- return Statement
- Static Variables
- Subroutine Arguments
- External Declarations

Functions

Functions are provided for implementing mathematical functions containing some number of arguments and one return value. Functions can be used in expressions in order to perform frequently used calculations, or to encapsulate the calculation.

The syntax to declare a function is:

```
function data_type function_name (type
     argument_list){statements;}
```

data_type:

The *data_type* can be any of the valid VeraLite data types, (**integer**, **bit**, **string**, **event**, **port**, or **enum**). The value returned will be of the same data type with which the function is declared.

function_name:

The *function_name* is the name by which the function is called throughout the program.

argument_list:

An argument is a variable, including the data type, that is passed to the function when the function is called. All data types can be passed. Array arguments can be associative, as well as **var**, (see Reference Passing on page 2-11 for discussion of **var**). Array arguments are strongly typed. The array type, width, and size of the call must exactly match the array type, width and size of the declaration. Multiple arguments are separated by commas.

statements:

The *statements* can be any statement, including function calls, timing modifiers, and variable assignments

Functions are designed to return a single value. They can return values of any data type as well as data structures. Functions can also return bit arrays. However, functions cannot return arrays of other types (either fixed size or associative). To set the return value, assign a value to the name of the function somewhere within the body of the function.

This is an example function declaration:

```
function bit [3:0] even_byte_parity (bit [31:0] data)
{
```

```
        bit [3:0] tmp;
        tmp[3] = ^data[31:24];
        tmp[2] = ^data[23:16];
        tmp[1] = ^data[15: 8];
        tmp[0] = ^data[ 7: 0];
        even_byte_parity = tmp;
}
```

This example declares the function **even_byte_parity()** with the argument `data`. The final line of the function contains the line that sets the return value.

Functions can be called in expressions from within the main program or from within other functions.

The syntax to call a function is:

```
variable = function(argument_list);
```

For example:

```
parity = even_byte_parity(Data);
```

By default, function names are global. Functions declared as local can only be used in the file where they are defined. To invoke a function defined in another file, you must use the **extern** declaration (see External Declarations on page 2-13).

The syntax to declare a local function is:

```
local function data_type function_name (type argument_list)
      {statements;}
```

For example:

```
extern local function bit[3:0] g_decode (integer i);
```

Discarding Function Return Values

Function return values are enforced by the VeraLite compiler. Calling a function as if it has no return value results in compilation errors. To discard a function's return value, use the void construct.

The syntax for the void construct is:

```
void = function(argument_list);
```

Tasks

Tasks are identical to functions except they do not return a value.

The syntax to declare a task is:

```
task task_name (type argument_list){statements;}
```

task_name:

The *task_name* is the name by which the task is called throughout the program.

argument_list:

An *argument* is a variable, including the data type, that is passed to the function when the function is called. All data types can be passed, including ports. Array arguments can be associative, as well as **var**, (see Reference Passing on page 2-11 for discussion of **var**). Array arguments are strongly typed. Array type, width, and size must match exactly between the declaration and the call. Multiple arguments are separated by commas.

statements:

The *statements* can be any VeraLite statement, including function calls, timing modifiers, and variable assignments.

This is an example task declaration:

```
task handshake_port0(bit direction, bit [7:0] data1, bit[7:0]
     data2)
{

     @0,1000 port0.req == 1'b1;
     port0.ack = 1'b1;
     @1 port0.ack <= 1'b0;

     if(direction) port0.data = data1;
          else port0.data = data2;
}
```

Tasks can be invoked as statements.

The syntax to invoke a task is:

```
task_name(argument_list);
```

For example:

```
print_data(new_data);
```

By default, task names are global. Tasks declared as local can only be used in the file where they are defined. To invoke a task defined in another file, you must use the extern declaration (see External Declarations on page 2-13).

The syntax to declare a local task is:

```
local task task_name (argument_list){statements;}
```

For example:

```
local task print_data (bit[7:0] data)
{

     printf("Local data = %h", data);
}
```

return Statement

Normally, functions and tasks return control to the caller after the last statement of the block is executed. VeraLite provides the **return** statement to manually pass control back to the caller.

The syntax for **return** is:

```
return;
```

When the **return** statement is executed, the subprocess is terminated as if it had been exited normally. If the **return** statement is executed in a function before a value has been assigned, an undefined value is returned.

If a **return** statement is executed at the top code level, the simulation is terminated.

Static Variables

By default, variables are local to the function or task that uses them. They are allocated when the function or task is called. This construct allows tasks and functions to be re-entrant and recursive.

If you want a variable to be shared across all invocations of a function or task, use the **static** declaration.

The syntax to declare a **static** variable is:

```
static data_type variable_name;
```

Any data type can be declared as a **static** variable.

Note:

In the case of concurrent accesses, there may be races if multiple threads assign to the same variable. Also, static variables cannot be declared in a global context.

Subroutine Arguments

VeraLite provides two means of accessing arguments in functions and tasks: “pass by value” and “pass by reference.”

Value Passing

“Pass by value” is the default method through which arguments are passed into functions and tasks. Each subroutine retains a local copy of the argument. If the arguments are changed within the subroutine, the changes do not affect the caller.

Reference Passing

In “pass by reference,” functions and tasks directly access the specified variables passed as arguments.

The syntax to pass a subroutine argument by reference is:

```
subroutine (var data_type variable);
```

In “pass by reference,” subroutines operate directly on the **var** arguments. The caller sees any changes to variables made within the subroutine. Variables of any type can be passed by reference.

This is an example of “pass by reference”:

```
task IO_read_indirect(bit[63:0] addr, var bit[31:0] io_data)
```

```
{
      // ... (modifies both addr and io_data)
      // ... (only the change in io_data will be seen by caller)
}
      // caller
      IO_read_indirect(my_addr, my_data );
```

In this example, the variable `io_data` is passed by reference. The task modifies all of the arguments passed, but only the change made to `io_data` is seen outside the task.

Note:

In the case of concurrent accesses, a race condition may arise if multiple threads assign to the same variable.

Default Arguments

To handle common cases or allow for unused arguments, VeraLite allows you to define default values for each scalar argument.

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

```
subroutine(type arg=default_value){statements}
```

default_value:

The *`default_value`* can be any expression visible at the current code level. It can include any combination of constants and global variables.

When the subroutine is called, you can omit an argument that has a default defined for it. Use an asterisk (*) as a placeholder in the subroutine call. If an asterisk is used for a variable that does not have a default value, a compilation error occurs.

This is an example of a subroutine with default arguments:

```
task read(integer i = 0, integer k, bit[5:0] data = 6'b0)
{

      //..

}
read(100, 5, *);
read(*, 5, 6'b000111);
```

This example declares a task **read()** with default arguments. The first call to **read()** is equivalent to read(100, 5, 6'b0). The second call to **read()** is equivalent to read(0,5,6'b000111).

External Declarations

External declaration of subroutines enables the use of multiple source files. Large functions and tasks can be compiled separately, which facilitates debugging.

Declaring External Subroutines

You can create subroutines in multiple source files. You must declare these subroutines as external at the top level.

The syntax to declare a subroutine as external is:

extern task | **function** *subroutine* (*argument_list*);

Note:

When using external subroutines, the argument types that are passed must match exactly. So take extra care when passing arguments to external subroutines.

External Default Arguments

The default values can be set locally, and independently, for each compilation unit using extern declarations with default values. A general library, which can then be customized for a particular user or testbench, can be implemented by using include files with different defaults.

For example, the task **write()** may be defined in a separate library, which is compiled independently. The VeraLite file in which the task **write()** will be used must declare **write()** as being external. Default values can be set in this **extern** declaration:

```
// file A (library)
      task write (integer i, k, bit[5:0] data)
      {
            // write definition
      }
// file B (testbench)
      extern task write(integer i = 10,integer k,bit[5:0]
            data=6'b1);

task xyz ()
{
      write (*, 5);
      //continue task declaration
}
```

3

Sequential Control

This chapter discusses the VeraLite constructs used for sequential flow control. It includes these sections:

- if-else Statements
- case Statements
- repeat loops
- for loops
- while loops
- break and continue

if-else Statements

The if-else statement is the general form of selection statement.

The syntax to declare an if-else statement is:

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

expression:

The *expression* can be any valid VeraLite expression.

block:

The *if_block* or *else_block* can be any valid VeraLite 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, the *else_block* is executed.

If the *else_block* is omitted, the conditional is evaluated and the *if_block* is executed only if it evaluated to true. Otherwise, the program continues execution with the first line after the *if_block*.

Nested if-else statements are supported.

Example:

```
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. Note that the final else statement is associated with the *if_block* immediately preceding it.

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]

}
```

primary_expression:

The *primary_expression* is evaluated. The value of the *primary_expression* 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 case block. If other matches exist, they are not executed.

case_expression:

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

All case expressions must be the same bit length. ‘X’ and ‘Z’ values are actual signal values and are not ignored.

statement:

The *statement* can be any valid VeraLite 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.

An example case block:

```
case ( bus[3:0] )
{
      4'b00ZZ: packet = null;
      4'booo1, 4'b1001: packet = READ;
      4'b0010, 4'b1010: packet = WRITE;
      4'b00XX: packet = UNKNOWN;

      default:
      {
            printf("Error: illegal packet %h detected\n",
                  bus[3:0]);
            packet_error();
      }
}
```

To use 'X' or 'Z' as a "don't care," use the **casex** or **casez** statements. When using **casex**, 'X' and 'Z' values in both the primary_expression and case_expressions are treated as "don't care". When using **casez**, 'Z' values in both the primary_expression and case_expressions are treated as "don't care". If no match is found, the default statement is executed.

repeat loops

The repeat loop executes a statement a fixed number of times.

The syntax to declare a repeat loop is:

```
repeat (expression) statement
```

expression:

The *expression* can be any valid expression, including constants.

statement:

The *statement* can be any valid statement or block of statements. If a code block is used, the entire block is executed.

Repeat statements can be used to repeat any statement a fixed number of times. The value of the expression is evaluated before the repetitions start. Changing a variable within the expression does not change the number of loops to be executed.

Repeat statements are often used to implement a wait or pause in the simulation.

For example:

```
repeat (10) @(posedge CLOCK);
```

This example pauses the simulation 10 clock cycles.

for loops

The syntax to declare a for loop is:

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

initial:

The *initial* is an assignment statement used to set the loop control variables.

condition:

The *condition* can be any valid expression.

increment:

The *increment* defines how the loop control variable changes each time the loop is repeated. It can be any valid expression.

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 single 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 of VeraLite code after the loop.

You can specify multiple variables in the initial statement, separating them with commas. Multiple variables can also be used in the condition expression. These variables (with their initialized values) are passed to the loop and can be used within the loop for loop control or in VeraLite expressions.

VeraLite does not allow assignments within the conditional. The conditional c=1 is invalid. Instead, you must use c==1.

Some examples of for loops:

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

for(count=0, done=0, i=0;i*count<125;i++)
      printf("Value i = %d\n",i);
```

while loops

The syntax to declare a while loop is:

while (*condition*) *statement*

condition:

The *condition* can be any valid expression.

statement:

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 the condition is true. When the condition is false, control passes to the first line of code after the loop. The condition is checked at the top of each loop.VeraLite does not allow assignments within the conditional. The conditional c=1 is invalid. Instead, you must use c==1.

This is an example of a while loop:

```
operator = 0;
while (operator<5)
{
      operator=operator+1;
      printf("Operator is %d", operator);
}
```

This loop continues until operator equals 5. Each time through the loop, operator is increased by 1. The check is made at the top of each loop. After 5 passes through the loop, the loop ends, and control is passed to the first line of code after the loop.

If the condition is a non-zero constant, the loop becomes infinite. Infinite loops can only be broken using the **break** statement (see break on page 3-8).

break and continue

The break and continue statements are used for flow control within loops.

break

The break statement is used to force the immediate termination of a loop, bypassing the normal loop test.

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

```
break;
```

When the break statement is executed from inside a loop, the loop is immediately terminated and control passes to the first line of VeraLite code after the loop. If the break statement is executed outside of a loop, a syntax error is generated.

This is an example of the break statement:

```
while (test_flag)
{
      if (done) break;
      ...
}
```

This example breaks if the condition is satisfied. Control returns to the first line after the loop.

continue

The continue statement forces the next iteration of a loop to take place, skipping any code in between.

The syntax to declare a continue statement is:

```
continue;
```

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 of code after the loop.

In a for loop, the continue statement causes the conditional test and increment portions of the loop to execute.

In a while loop, the continue statement passes control to the conditional test.

This is an example of a **continue** statement:

```
for (i=0;i<10;i++)
{
      if (skip_loop) continue;
      ...
}
```

4

Concurrency Control

This chapter discusses how VeraLite handles concurrency. It explains how to model parallel, independent activities and details the VeraLite constructs used to control those concurrent threads. Included are these sections:

- fork and join
- Synchronizing concurrent processes with event variables
- Semaphores
- Mailboxes
- Timeout Limit

fork and join

Fork/join blocks provide the primary mechanism for creating concurrent processes.

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

```
fork
{
      statement1
}

{
      statement2

}

...

{
      statementN
}
join [all | any | none]
```

statement:

The *statement* can be any valid VeraLite statement or sequence of statements.

Keywords:

The **all** | **any** | **none** options specify when the code after the fork/join block executes. They are optional.

The default is **all**.

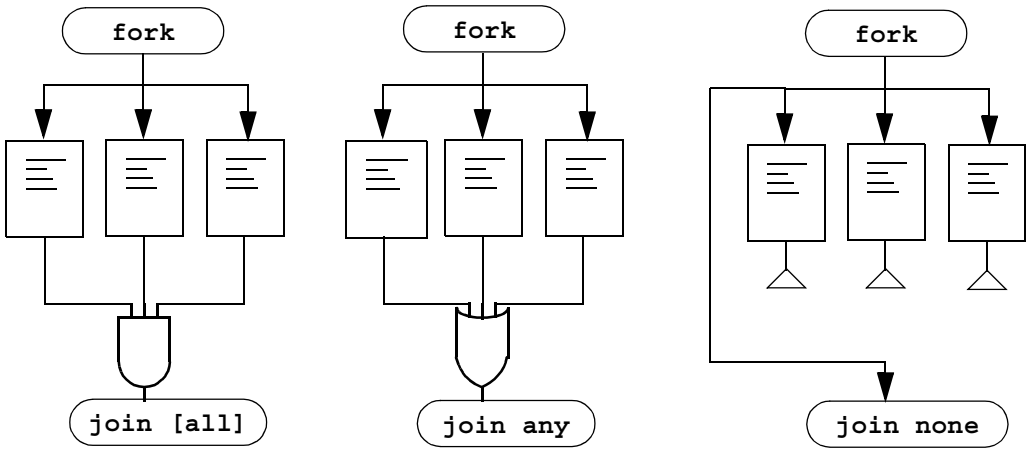
Table 4-1 all, any, none

all	The default option is **all**. Code after the block executes after all of the concurrent processes are completed
any	When **any** is used, code after the block executes after any single concurrent process is completed
none	When **none** is used, code after the block executes immediately, without waiting for any of the processes to complete

You do not need to specify more than one forked thread. If only a single thread is specified in a fork/join block and that thread consists of a single VeraLite statement, the thread does not need to be encapsulated with braces ({}).

The flow for a fork/join block is shown in Figure 4-1.

Figure 4-1 fork/join Flow Diagram



Note:

When defining a fork/join block, encapsulating the entire fork inside braces ({}) results in the entire block being treated as a single thread, and the code executes consecutively.

For example, do not use this construct:

```
fork
{
      statement1
      statement2
}
join                //becomes simply a sequential process
```

Example of a basic fork/join construct:

(Default is **all**)

```
fork
{
      @1,100 bus.ack == 1'b0;
      printf("First Block: bus.ack is driven\n");
}

{
      @5 bus.req = 1'b0;
      @1 bus.req <= 1'b1;
      printf("Second Block: bus.req is driven\n");
}
join
```

The concurrent block executes all the statements in parallel. The beginning of each statement is executed at the same point in time. Subsequent statements are executed based on any timing considerations within the process.

fork and join Control

VeraLite provides several constructs and a system task to control fork/join blocks.

- **wait_child()**
- **wait_var()**

- **terminate**
- **suspend_thread()**

The constructs, **wait_child()** and **wait_var()**, wait for the completion of processes. The **terminate** construct stops the execution of processes. The **suspend_thread()** system task temporarily suspends threads.

wait_child()

The **wait_child()** system task is used to ensure that all child processes are executed before the VeraLite program terminates.

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

```
task wait_child();
```

By default, a simulation is terminated when the end of the program is reached, regardless of the status of any child processes. Using the **wait_child()** task causes the simulation to wait until all the child processes in the current context are completed before executing the next line of code.

This is an example of a program using the **wait_child()** construct:

```
program test
{
      start_monitors(); /*Starts monitors that loop
                          forever in background*/
      do_test(); //Performs the actual test
}

task start_monitors()
{
      fork
      {...}
      join none
}
```

```
task do_test()
{
      //Code to do testing

      fork
      {...}
      join none /*Creates child processes that take an
            indeterminate amount of time to complete*/
      wait_child();
}
```

This example calls two separate tasks. The **do_test** task forks off several child processes that take an indeterminate amount of time to complete. The **wait_child()** call waits for the threads called in the do_test task to complete before executing subsequent VeraLite code. Note that the **wait_child()** call does not wait for any child processes created outside of its context.

The definition of **context** assumed here is:

> A context is a node in the simulator's call Stack. VeraLite constructs that create a new context are:
>
> - the program block
> - task
> - function
> - each process inside the fork/join

To see how fork/join-all and **wait_child()** differ, consider the following code.

fork/join-all:

```
fork
{statement3};
{statement4};
join none

fork
{statement1};
{statement2};
join all
```

wait_child():

```
fork
{statement3};
{statement4};
join none

fork
{statement1};
{statement2};
join none

wait_child();
```

In the fork/join-all example, code following the block executes after the concurrent processes, statements1 and 2 are completed. However, code after the block executes immediately, without waiting for statements 3 and 4 to complete.

In the **wait_child()** example, statements 3 and 4 *are* waited for.

wait_var()

The **wait_var()** system task blocks the calling process until one of the variables in its arguments list changes values.

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

```
task wait_var(integer|bit|string|enum variable_list);
```

variable_list:

The *variable_list* consists of one or more variables (separated by commas) of type **integer**, **bit**, **string**, **array**, or **enumerated** type.

The **wait_var()** task blocks the current process until one of the specified variables changes value. Only true value changes unblock the process. Reassigning the same value does not unblock. If more than one variable is specified, a change to any of the variables unblocks the process.

This is an example of the **wait_var()** task:

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

fork
{
      wait_var(data[2]);
      printf("Data[2] has changed to: %d\n", data[2]);

}
{
      for (i=0;i<100;i++)
      {
            data[i]=$random();
            @(posedge CLOCK);

      }
}
join
```

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

terminate

The terminate statement terminates all active descendants of the process in which it was called.

The syntax for **terminate** is:

```
terminate;
```

If any of the child processes have other descendants, the **terminate** command terminates them as well. If used at the top level, **terminate** terminates all child processes. When the main program is completed, the VeraLite simulator executes an implicit terminate statement.

This is an example of how **terminate** is used within a simple fork/join block:

```
task do_test()
{
      // Code to do testing
      fork
      {...}
      join any/* Creates child processes that take an
                        indeterminate amount of time to complete
                        Code to do more testing*/
      terminate;
}
```

This example forks off several child processes within a task. After any of the child processes are complete, the code continues to execute. Before the task is completed, all remaining child processes are terminated.

suspend_thread()

The **suspend_thread()** system task is used to temporarily suspend the current thread.

The syntax **suspend_thread()** is:

```
task suspend_thread();
```

The **suspend_thread()** system task temporarily suspends the current thread and allows other ready concurrent threads to run. When all ready threads have had one chance to block, the suspended thread resumes execution.

For example:

```
for (i=0;i<10;i++)
{
      fork
      my_task(i);
      join none
      suspend_thread();
}
```

This example forks multiple threads calling **my_task()**. The thread is forked, the task is called, and then the calling thread is suspended. The forked thread calling **my_task(0)** completes and passes control back to the for loop. The next iteration of the loop occurs and forks the next thread. That thread begins and completes execution. All 10 threads are created and execute in sequence.

Note:

Suspended threads execute after all other current threads execute. However, relative to simulation time, the thread is still executed concurrently with the other threads.

Maximum Threads

To limit memory consumption set the VERA_MAX_CONTEXTS environment variable:

```
setenv VERA_MAX_CONTEXTS number
```

If more than *number* threads are created at the same time, a warning message is printed.

Semaphores

A semaphore is an operation used for mutual exclusion and synchronization.

- Conceptual Overview
- Allocating a Semaphore
- Checking Key Availability
- Returning Keys

Conceptual Overview

Conceptually, semaphores can be viewed as a bucket. When you allocate a semaphore, you create a virtual bucket. Inside the bucket are a number of keys. No process can be executed without first having a key. So, if a specific process requires a key, only a finite number of occurrences of that process can be in progress simultaneously. All others must wait until a key is returned to the virtual bucket.

The semaphore system functions are:

```
function integer alloc(SEMAPHORE,integer semaphore_id,
     integer semaphore_count, integer key_count);

function integer semaphore_get(WAIT | NO_WAIT,
     integer semaphore_id, integer key_count);

task semaphore_put(integer semaphore_id, integer
     key_count);
```

Allocating a Semaphore

To allocate a semaphore, you must use the **alloc()** system function.

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

```
function integer alloc(SEMAPHORE, integer semaphore_id,
     integer semaphore_count, integer key_count);
```

semaphore_id:

The *semaphore_id* is the ID number of the particular semaphore being created. It must be an integer value. You should generally use 0. When you use 0, a semaphore ID is automatically generated by the simulator. Using any other number explicitly assigns an ID to the semaphore being created.

semaphore_count:

The *semaphore_count* specifies how many semaphore “buckets” you want to create. It must be an integer value.

key_count:

The *key_count* specifies the number of keys initially allocated to each semaphore “bucket” you are creating.

Note:

The number of keys in the bucket can increase if more keys are put into the bucket than are removed. Therefore, *key_count* is not necessarily the maximum number of keys in the bucket.

The **alloc()** function returns the base semaphore ID if the semaphores are successfully created. Otherwise, it returns `0`.

Checking Key Availability

To check that there are enough keys left in the semaphore, you must use the **semaphore_get()** system function.

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

```
function integer semaphore_get(NO_WAIT | WAIT,
     integer semaphore_id, integer key_count);
```

Predefined Macros:

NO_WAIT

The NO_WAIT option continues code execution even if there are not enough keys available.

WAIT

The WAIT option suspends the process until there are enough keys available, at which time execution continues.

semaphore_id:

The *semaphore_id* specifies which semaphore to get keys from.

key_count:

The *key_count* specifies the number of keys being taken from the semaphore.

When the **semaphore_get()** function is called, it checks the specified semaphore for the number of required keys.

- If there are enough keys available, a 1 is returned and execution continues.
- If there are not enough keys available, a 0 is returned and the process is suspended depending on the wait option.

The semaphore waiting queue is FIFO based. By default, a process will wait at a semaphore without timing out. Users can set a time limit with the **timeout()** system task. See Timeout Limit on page 4-20.

If multiple semaphores are allocated, you can access the Nth semaphore using this method:

```
semID=alloc(SEMAPHORE, 0, 4, 2);
if (semaphore_get(WAIT, semID+2, 1))
     printf("The semaphore was successful.");
```

This example allocates four semaphores with IDs 0 to 3, each with two keys. Then it checks to see if there is a key in the third semaphore. If there is, a message is printed.

Returning Keys

To put keys back into a semaphore, you must use the **semaphore_put()** system task.

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

```
task semaphore_put(integer semaphore_id, integer key_count);
```

semaphore_id:

The *semaphore_id* specifies which semaphore to return the keys to.

key_count:

The *key_count* specifies the number of keys being returned to the semaphore.

When the **semaphore_put()** system task is called, the specified number of keys is returned to the semaphore. If a process has been suspended to wait for a key, that process executes when enough keys have been returned.

Mailboxes

A mailbox is a mechanism to exchange messages between processes. Data can be sent to a mailbox by one process and retrieved by another.

Conceptual Overview

Conceptually, mailboxes behave like real mailboxes. When a letter is delivered and put into the mailbox, you can retrieve the letter (and any data stored within). However, if the letter has not been delivered when you check the mailbox, you must choose whether to wait for the letter or retrieve the letter on subsequent trips to the mailbox. Similarly, VeraLite's mailboxes allow you to transfer and retrieve data in a very controlled manner.

The mailbox system functions are:

Allocating a Mailbox

To allocate a mailbox, you must use the **alloc()** system function.

The syntax for allocating a Mailbox is:

```
function integer alloc(MAILBOX, integer  mailbox_id, integer
     mailbox_count);
```

mailbox_id:

The *mailbox_id* is the ID number of the particular mailbox being created. It must be an integer value. You should generally use 0. A mailbox ID is automatically generated when 0 is used.

mailbox_count:

The *mailbox_count* specifies how many mailboxes you want to create. It must be an integer value.

The **alloc()** function returns the base mailbox ID if the mailboxes are successfully created. Otherwise, it returns 0.

The maximum number of mailboxes that can be created is determined by **vera_mailbox_size**.

Sending Data to the Mailbox

The **mailbox_put()** system task sends data to the mailbox.

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

```
task mailbox_put(integer mailbox_id, scalar data)
```

mailbox_id:

The *mailbox_id* specifies which mailbox receives the data.

data:

The *data* can be any general expression that evaluates to a scalar.

The **mailbox_put()** system task stores data in a mailbox in a FIFO manner. Note that when passing objects, only object handles are passed through the mailbox.

Returning Data

The **mailbox_get()** system function returns data stored in a mailbox.

The syntax for mailbox_get is:

```
function integer mailbox_get(NO_WAIT | WAIT | COPY_NO_WAIT |
     COPY_WAIT, integer mailbox_id [, scalar dest_var [, CHECK]]);
```

Predefined Macros:

Table 4-2 provides the definitions of the various wait options.

Table 4-2 Wait Option definitions

WAIT OPTIONS	Description
NO_WAIT	Dequeues mailbox data if it is available. Otherwise, it returns an empty status (0).
WAIT	Suspends the calling thread until data is available in the mailbox, and then dequeues the data.
COPY_NO_WAIT	Copies mailbox data without dequeuing it if it is available. Otherwise, it returns an empty status (0).
COPY_WAIT	Suspends the calling thread until data is available in the mailbox, and then copies the data without dequeuing it.

mailbox_id:

The *mailbox_id* specifies which mailbox data is being retrieved from.

dest_var:

The *dest_var* is the destination variable of the mailbox data.

CHECK:

CHECK specifies whether type checking occurs between the mailbox data and the destination variable. CHECK is optional.

The **mailbox_get()** system function assigns any data stored in the mailbox to the destination variable and returns the number of entries in the mailbox, including the entry just received.

- If there is a type mismatch between the data sent to the mailbox and the destination variable, a runtime error occurs unless the CHECK option is used.

- If the CHECK option is active, a -1 is returned, and the message is left in the mailbox and is dequeued on the next **mailbox_get()** function call.
- If the mailbox is empty, the function waits for a message to be sent, depending on the wait option. If the wait option is NO_WAIT, the function returns a 0.
- If no destination variable is specified, the function returns the number of entries in the mailbox, but it does not dequeue an item from the mailbox.

For example, this can be used to continue generating mailbox entries until a specified number are generated:

```
mboxID=alloc(MAILBOX, 0, 1);
while (mailbox_count <11)
{
      mb_data=$random();
      mailbox_put(mboxID, mb_data);
      mailbox_count=mailbox_get(NO_WAIT, mboxID);
}
```

This example generates random numbers and puts them in the mailbox. The loop continues while the number of entries is less than 11.

The mailbox waiting queue is FIFO based. By default, a process will wait at a mailbox without timing out. Users can set a time limit with the **timeout()** system task. See Timeout Limit on page 4-20.

Timeout Limit

A process will wait forever in semaphore and mailbox if the waiting resources are not available. However, the system task **timeout()** can be used to set a time limit.

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

```
task timeout(EVENT, integer cycle_limit);
```

or

```
task timeout(event event_name, integer cycle_limit]);
```

or

```
task timeout(SEMAPHORE | MAILBOX | WAIT_VAR | WAIT_CHILD,
     integer cycle_limit [, integer object_id]);
```

Predefined Macros:

EVENT, SEMAPHORE, MAILBOX, WAIT_VAR and WAIT_CHILD specify the type of object for which the timeout is defined.

cycle_limit:

The *cycle_limit* specifies the maximum number of cycles any request will wait.

event_name:

The *event_name* is an event variable.

object_id:

The *object_id* specifies an individual resource for which the timeout is set. If it is not specified, the timeout exists for all objects of the type specified.

When the **timeout()** system task is used, it sets the maximum number of cycles that an object will wait for a request. The cycles are based on the SystemClock. If the cycle limit is set to 0 cycles, the timeout is disabled. You can specify a timeout for a specific event or for all events of a certain type.

When a semaphore or event times out, a verification error occurs.

These are examples of timeout statements:

```
timeout(SEMAPHORE, 100);
timeout(EVENT, 20);
timeout(myevent, 300);
```

Note:

Specific timeouts take precedence over global timeouts.

5

Interfacing to the Device Under Test

This chapter covers the properties of the interface specifications, and signal declarations.

The interface specification can group signals by clock domains for multiple clock designs. There is no limit to the number of interface declarations that can be created.

Interface Declaration

The syntax of the VeraLite interface declaration is depicted in Figure 5-1. Included are the syntax for port-connected interface signals and for direct-connected (HDL) signals.

Figure 5-1 The VeraLite Interface

```
interface interface_name
{
 signal_direction [signal_width] signal_name signal_type [signal_type]
       [hdl_node "hdl_path"];
}
```

The limit on the number of signals that can be declared per interface is 4096.

Interface Signal Declarations

This section deals with the properties of:

- Port-connected Interface Signals
- Direct-connect Interface Signals
- Interface Signal of type CLOCK

Port-connected Interface Signals

A port-connected interface signal involves connecting port level signal on an VeraLite testbench.

The syntax to declare a port-connected interface signal is:

```
signal_direction [signal_width] signal_name signal_type ;
```

```
inout[signal_width] signal_name input_signal_type
     output_signal_type;
```

signal_direction:

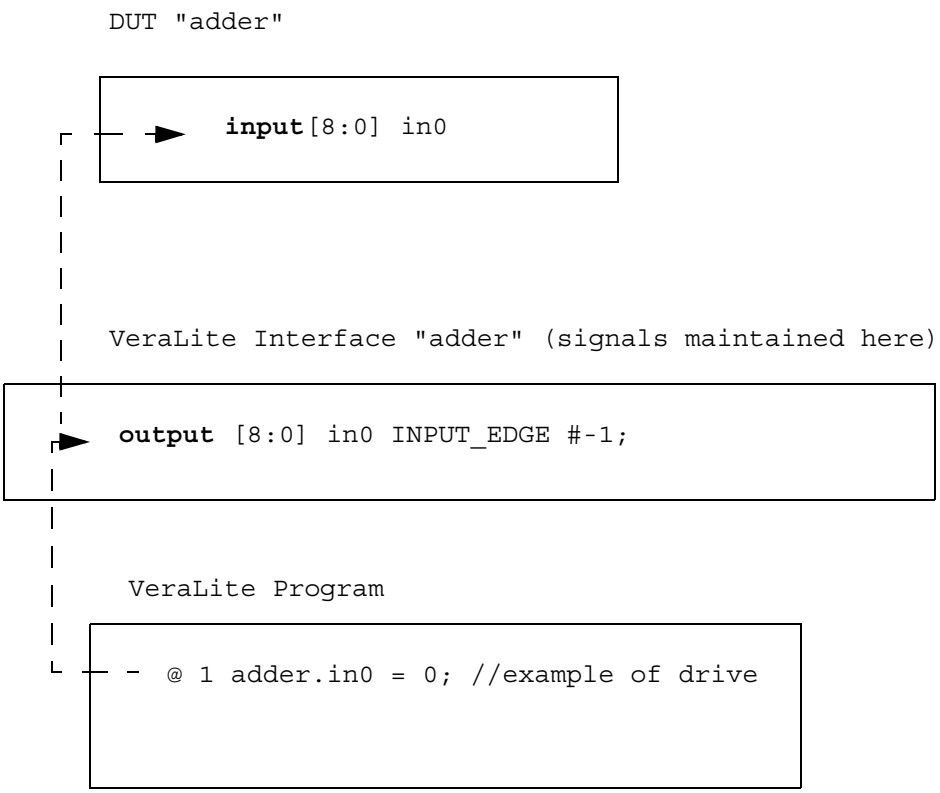
The *signal_direction* specifies the direction of the signal **with respect to VeraLite and not to the DUT**. See Figure 5-2 on page 5-4

- **input** indicates that the signal goes from the DUT to VeraLite.
- **output** indicates that the signal goes from VeraLite to the DUT.
- **inout** specifies a bi-directional signal.

inout:

The **inout** specifies the bi-direction of the signal. A bi-directional signal has two signal_types for each signal_type.

Figure 5-2 Representing the signal in0 in the DUT, Interface and VeraLite Program



signal_width:

The *signal_width* is a bit vector specifying the width of the signal. It must be in the form [*msb*:0].

signal_name:

The *signal_name* identifies the signal being defined. It is the VeraLite name of the HDL signal being connected at the top-level. It is also the name of one of the DUT instance's ports.

signal_type:

The valid signal types and their definitions are listed in Table 5-1.

Table 5-1 Signal types

Signal type	Operation
NHOLD	Output is driven on the negative edge of the interface clock.
PHOLD	Output is driven on the positive edge of the interface clock.
NSAMPLE	Input is sampled (evaluated) at the negative edge of the interface clock.
PSAMPLE	Input is sampled (evaluated) at the positive edge of the interface clock.
CLOCK	Specifies the clock to which the interface signals synchronizes.

For a unidirectional signal, only one signal type can be used. Furthermore, input signals are only sampled, and output signals are only driven. A bidirectional signal can be both sampled and driven.

Note:

In order to sample an output signal, declare it as signal type, inout.

Direct-connect Interface Signals

A direct-connect interface signal involves connecting to an internal signal within the hierarchy of the DUT.

The syntax to declare a direct-connect interface signal is:

```
signal_direction [signal_width] signal_name signal_type
          hdl_node "hdl_path";
```

signal_direction:

The *signal_direction* specifies the direction of the signal with respect to VeraLite.

- **input** indicates that the signal goes from the DUT to VeraLite.
- **output** indicates that the signal goes from VeraLite to the DUT.
- **inout** specifies a bi-directional signal.

signal_width:

The *signal_width* is a bit vector specifying the width of the signal. It must be in the form [*msb*:0].

signal_name:

The *signal_name* identifies the signal being defined. It is the top-level name of the DUT signal being connected.

signal_type:

The valid *signal_types* and their definitions are listed in Table 5-1 on page page 5-5.

hdl_path:

The *hdl_path* is the HDL path to the specified signal. It must be surrounded by double quotes.

Below are samples of HDL node declarations in a Verilog:

```
input[31:0] grant PSAMPLE #-2 hdl_node
     "sys.cpu2.p0_d1";
```

```
/* "sys" is the top level Verilog module, and "cpu" is the
DUT */
```

Notice, that the path always starts from the top level HDL module. What can be included in the string is determined by what the simulator supports. For example, when using a Verilog simulator, you can concatenate multiple Verilog nodes.

```
Example:
      module top()
            reg[7:0]datH;
            reg[7:0]data:;
      endmodule

      interface myint
      {
            input CLOCK...
            input[15:0]data...hdl_node"{top.dataH,top.dataL}";
      }
```

Interface Signal of type CLOCK

Each interface may include, at most, one input signal of type CLOCK. If an input signal of type CLOCK is not designated then the interface signals are synchronized using SystemClock.

The syntax for declaration is:

```
input clock_name CLOCK;
```

The other signals defined in a given interface are governed by this clock. VeraLite samples and drives interface signals on the specified edge of this clock.

A signal of type CLOCK can be either a port-connected or a direct-connect interface signal.

Note:

Clock domains can be overlapped. The same signal can be associated with multiple clocks via multiple interface definitions. However, despite multiple interfaces, a single signal cannot be driven to two values at the same time.

6

Signal Operations

This chapter covers the four primitive statements provided by VeraLite that operate on interface signals; synchronization, drive, sample, and expect. This chapter also includes discussion of implicitly synchronized, explicitly synchronized and asynchronous signal operations. These topics are covered in the following sections:

- Synchronization
- Driving a signal
- Sampling a Signal
- The expect Event
- Implicit Synchronization
- Asynchronous Signal Operations
- Sub-Cycle Delays

Synchronization

"Implicit" synchronization involves a signal in an interface being synchronized to the interface clock. The synchronization operator (@) is used to perform explicit synchronization. That is, you are explicitly synchronizing to the signal changing value.

The syntax is:

```
@([specified_edge] interface_signal);
```

specified_edge:

The *specified_edge* identifies the edge at which the synchronization occurs. Either **negedge**, which specifies a negative or falling edge of the interface signal, or **posedge**, which specifies a positive or rising edge of the interface signal, can be designated. If no edge is specified, the synchronization occurs on the next change in the specified signal.

interface_signal:

The *interface_signal* specifies the signal to which the synchronization is linked. It can be any signal in an interface declaration or **CLOCK**. The interface signal can be any subfield of a signal as well. If **CLOCK** is specified, the synchronization operation is performed on SystemClock.

If the interface signal is a subfield of a signal, the synchronization occurs on the first change of the signal subfield. If the subfield is a 1-bit subfield, you can synchronize on clock edges. If you specify variables in the subfield, they are evaluated at runtime.

You can use the **or** keyword to specify multiple interface signals. If you specify more than one signal, the synchronization occurs on the next change of any of the listed signals.

These are some example synchronization statements:

- In the first example, the synchronization occurs on the next change in the signal, `ack_1`.

```
@(ram_bus.ack_l);
```

- The second example synchronizes to the SystemClock.

```
@(CLOCK);
```

- The third example synchronizes to the positive edge of the interface clock, `ram_bus.clock`.

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

- The fourth example synchronizes to the falling edge of the specified subfield, `intf.sign[a]`. Note that the specified subfield must be a 1-bit subfield and `a` is evaluated at runtime.

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

- The final example specifies multiple interface signals. The synchronization occurs on the next positive edge of either `intf.sig1` or `intf.sig2`, whichever changes first.

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

At initialization, HDLs can create edges at time = 0 (for example, going from X to the initialized value). This means that synchronization conditions can be set before initialization of the signal.

Driving a signal

The drive operator sets the value of output interface signals.

The syntax to drive a signal is:

```
[delay] signal_name range drive_operator expression;
```

delay:

The *delay* optionally specifies the number of cycles that pass before the signal is driven. It is in the form `@n` , where *n* is the number of cycles. When `delay` is not specified, the default is `@0`.

Note:

Drive delays are specified in VeraLite as integers.

signal_name:

The *signal_name* is the name of the interface signal being driven.

range:

The *range* specifies which bits of the signal are driven. If no range is specified, the entire signal is driven.

drive_operator:

The *drive_operator* must be either `=`, which specifies a blocking drive, or `<=`, which specifies a non-blocking drive.

expression:

The *expression* can be any valid VeraLite expression.

These are some drive examples:

```
foo_bus.data[3:0] = 4'h5; // blocking drive
@1 foo_bus.data <= 8'hz; // non-blocking drive
```

Blocking and Non-Blocking Drives

There are two types of drives specified by the drive operator: **blocking** and **non-blocking**.

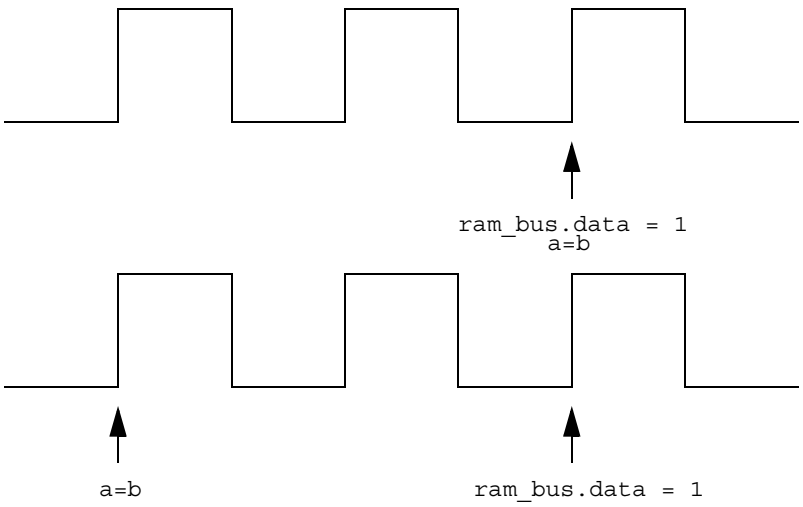
Blocking drives suspend execution until the statement completes. Note that the clock edge (NHOLD or PHOLD) that the drive signal is associated with is used for counting the HDL cycles during suspension. Once the statement completes, execution resumes.

Non-blocking drives schedule the drive at a future cycle and execution continues. When the specified cycle occurs, the drive is executed.

These are examples of blocking and non-blocking drives:

```
@3 ram_bus.data = 1; // blocking drive
     a = b;

@3 ram_bus.data <= 1; // non-blocking drive
     a = b;
```



The first block is a blocking drive. Three cycles must pass before both lines are executed. The second block is a non-blocking drive. The first line is scheduled to be executed 3 cycles in the future, then the second line is executed.

Drives

A given signal should only be driven by a single drive at any given time. Multiple drives at the same time result in conflicting drives. Conflicting drives drive the signal to X and result in a simulation error.

Sampling a Signal

Sample assigns the value of a signal to a variable.

The syntax is:

```
variable = signal_name;
```

The *signal_name* is an interface signal. It must be an **input** or **inout** signal. It is sampled at the next sampling point (specified in the interface definition) and the value is assigned to the variable. The delay attribute (@ in drive signals) cannot be used. Also remember that you can sample subfields within the signal by specifying a specific subfield in the signal width.

Note:

When sampling a signal in an expression, it is done *immediately* (i.e., asynchronously). Output interface signals cannot be used in any right hand side part of the expression since it cannot be sampled. In particular cannot be used in **sscanf(), fprintf(), sprintf()** or **printf()**.

Implicit Synchronization

The drive, sample, and expect primitives perform implicit synchronization to the interface CLOCK. That means that the clock is advanced only when it is necessary to perform the next signal operation.

Consider the following interface definition as an example:

```
interface foobus
{
      output reset_l NHOLD;
      input strobe_l PSAMPLE;
      output ack_l NHOLD;
      inout data PSAMPLE NHOLD;
      input clock CLOCK;
}
```

In this interface, output signals are driven at the negative edge of the interface clock, and input signals are sampled at the positive edge of the interface clock. Thus, the following code advances the simulation cycle a half cycle per statement even though a delay is not specified.

Consider these examples:

- The first signal is driven on the negative clock edge:

```
foobus.reset_l = 1'b1;
```

- The second signal is sampled on the positive clock edge:

```
foobus.strobe_l == 1'b1;
```

- The third signal is driven on the negative clock edge:

```
foobus.ack_l = 1'b0;
```

- The fourth signal is sampled on the positive clock edge:

```
foobus.strobe_l == 1'b0;
```

The description gets more complicated when delay values are used to generate proper timing with respect to different edges. To avoid this, use the same edge for inputs and outputs, with appropriate output skews.

For example:

```
interface foobus
{
   output reset_l PHOLD #2;
   input strobe_l PSAMPLE;
   output ack_l PHOLD #2;
   inout data PSAMPLE PHOLD #2;
   input clock CLOCK;
}
```

Asynchronous Signal Operations

By default, drives, samples, and expects are relative to a clock edge (specified in the interface specification). However, the HDL side of the simulation may be using very detailed timing constructs. VeraLite provides the async and delay constructs to allow detailed timing down to the HDL timestep.

async Modifier

The async optional modifier specifies that the operation happen immediately, without waiting for the edge specified in the interface. It can be used with synchronization operators, drives, samples, and expects.

The syntax for the async modifier is:

synchronization	`@(`*signal_name* **async**`);`
Drive	*signal_name range drive_operator expression* **async**`;`
Sample	*variable* `=` *signal_name* **async**`;`
Expect	*expect_list* **async**`;`

The synchronization construct allows you to act exactly on the current edge rather than waiting for the corresponding sampling edge.

The drive, sample, and expect constructs force the operation immediately instead of waiting for the edge specified in the interface.

These are examples of async statements:

```
@(posedge main_bus.request async);
memsys.data[3:0] = 4'b1010 async;
data[2:0] = main_bus.data[2:0] async;
main_bus.data[7:4] == 4'b0101 async;
```

Sub-Cycle Delays

VeraLite provides the **delay()** system task to block the VeraLite side of the simulation while a specified amount of time elapses on the HDL side of the simulation.

The syntax for the **delay()** system task is:

```
task delay(integer time);
```

time:

The *time* specifies the length of the delay. It is in the same timing units being used by the HDL.

This is an example of the **delay()** system task:

```
@(posedge CLOCK);
delay(5);
function1();
...
```

This example synchronizes to the positive edge of CLOCK. Then it advances the simulation time 5 time ticks. Function1 executes 5 time ticks after the clock edge.

7

Class and Methods

This chapter discusses the VeraLite implementation of Object Oriented Programming. OOP forms the basis of the data structures, encapsulation.

Classes and Objects

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*, subroutines are called *methods*, and we will refer to both as *members* of the class. The properties and methods, taken together, usually define the contents and capabilities of some kind of *object*.

For example, a packet is 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 we can do with a packet: initializing the packet, setting the command, reading the

packet's status, checking the sequence number. Each `Packet` is different, but as a **class**, packets have certain intrinsic properties that we 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;

      task new() / initialization
      {
            command = IDLE;
            address = 41'b0;
            master_id = 5'bx;

      task clean()
      {
            command = 0; address = 0; master_id = 5'bx;
      }

// public access entry points

      task issue_request( integer delay )
      {
            // send request to bus
            // ...
      }

      function integer current_status()
      {
            current_status = status;
      }
}
```

Note that a common convention is to capitalize the first letter of the class name, so that it is easy to recognize class declarations.

Objects and Instance of Classes

So far, we only have the definition of the class Packet. We have created a new, complex data type but we can't do anything with the class itself. We need to create an *instance* of the class, a single `Packet` object. The first step is to create a variable that can hold an object's name (or handle):

```
Packet p;
```

Nothing has been created yet. We have just declared that `p` is a variable that can hold the handle of a Packet object. In VeraLite, for `p` to refer to something, we need to explicitly create an instance of the class using the **new** keyword.

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

You can detect uninitialized object handles by comparing them with null.

For example:

```
class obj_example
{
      ...
}

task task1 (integer a, (obj_example myexample = null))
{
      if (myexample == null) myexample = new;
}
```

This example checks if `myexample` is initialized. If it is not, it initializes it with the new command.

Accessing Object Properties

Now that we have created an object, we can use its data fields by qualifying property names with an instance name. Looking at the earlier example, we can use the commands for our Packet p as follows:

```
Packet p = new;
p.command = INIT;

time = p.time_requested;
```

Using Object Methods

To access an object's methods, we use the same syntax we 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 we don't have to pass arguments to **current_status()**. The properties of a class are freely and broadly available to the methods of the class, but each method only accesses the properties associated with its object, its instance.

Constructors

VeraLite does not require the complex memory allocation and de-allocation 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 you create an object, for example

```
Packet p = new;
```

The system executes the `new` task associated with the class:

```
class Packet
{
      integer command;

      task new()
      {
            command = IDLE;
      }
```

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` subroutine is invoked, if it exists, allowing you to do any initialization or start-up functions you require. The new task is also called the constructor of a class.

It is also possible to pass arguments to the constructor, to allow for run-time customization of the object:

```
Packet p = new(STARTUP,  get_time(LO));
```

where the `new` initialization task in `Packet` might now look like

```
task new (integer in_command=IDLE, bit[40:0] in_address=0, integer
time_stamp=0)
{
      command = in_command; address = in_address;
      time_requested = time_stamp;
}
```

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

Class Properties

So far, we have only declared *instance* properties. Each instance of the class, each **Packet**, has its own copy of each of its six variables. There are also cases where we only want one copy of the variable, to be shared by all instances. These *class properties* are created using the `static` keyword. Thus, for example, in a case where all instances of a class need access to a semaphore id, we might have

```
class Packet
{
      static integer semId = alloc(SEMAPHORE, 0, 1, 1);
```

Now, `semId` will be created and initialized the first time an object of the `Packet` class is created. Thereafter, every packet object can access the semaphore in the usual way:

```
Packet p;
semaphore_get(WAIT, p.semId);
```

this

There are times when you need to unambiguously refer to properties or methods in the current instance. For example, the following declaration is a common, clean way to write an initialization routine:

```
class Demo
{
      integer x;

      task new (integer x)
      {
            this.x = x;
      }
```

The `x` is now both a property of the class and an argument to the task `new`. In the task `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, you can always qualify members with `this` to refer to the current instance, but it is usually unnecessary.

Assignment, Re-naming and Copying

When we declare a class variable, we have only created a name for an object.

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. It is not until we create an instance of type Packet that the object exists, and that p1 contains an actual handle.

```
p1 = new;
```

Thus, if we create another variable

```
Packet p2;
```

and assign `p1` to `p2`

```
p2 = p1;
```

then we still have only one object, which we can refer to with either the name p1 or p2. Note, we have only executed new once, so we have only created one object.

If we rewrite the last expression slightly differently, however, we make a copy of `p1`:

```
p2 = new p1;
```

Now we have executed new twice, so we have created two objects. 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, we have created two names for the same object. This is true even if the class declaration includes the instantiation operator new:

```
class A
     {
          integer j = 5;
     }

class B
     {
          integer i = 1;
```

```
            A a = new;
      }

program test
{
      integer test;
      B b1 = new;      // Create an object of class B
      B b2 = new b1;   // Create an object that is a copy of b1
      b2.i = 10;       // i is changed in b2, but not in b1
      b2.a.j = 50;     // change object a, shared by both b1 and b2
      test = b1.i;     // test will be set to 1 (b1.i has not changed)
      test = b1.a.j;   // test will be set to 50 (a.j has changed)
```

Note several things. We can initialize properties and instantiate objects directly in a class declaration. Second, the shallow copy does not copy objects. Third, we can chain instance qualifications 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.next.val  /* would chain through a sequence of
                                    handles to get to val.*/
```

To do a full (deep) copy, where everything (including nested objects) are copied, you need to write custom code. Thus, we might have

```
Packet p1 = new;
Packet p2 = new;
p2.copy(p1);
```

where copy(Packet p) is a method written to copy the object specified as its argument into its instance.

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. You do this in two steps. Within the class body, you declare the method prototype - whether it is a function or task, any attributes (protected, public, and/or virtual), and the full specification of its arguments. Then, outside of the class, you declare the full method - including the prototype but without the attributes - and, to tie the method back to its class, you 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;
      }
      protected virtual function integer send (integer value);
}
function integer Packet::send(integer value)
{     // dropped protected virtual, added Packet::
      // body of method
      ...
}
```

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

External Classes

As with subroutines, the class declaration can be in a separate file from the code that instantiates and invokes the class; you need to provide an external declaration of the class to support the tight type-checking required by VeraLite.

The attributes and the method prototypes need to be re-declared:

```
extern class packet
{
      bit [3:0] command;
      bit [40:0] address;
      bit [4:0] master_id;
      task issue_request( integer delay );
      function integer current_status();
}
```

An extern class declaration only requires the inclusion of class properties and methods referenced in the files that include the declaration.

Typedef

Sometimes you need to declare a class variable 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.

The way around this is to use typedef to provide an interim declaration for the second class:

```
typedef class C2;     // C2 is declared to be of type class
class C1
```

```
{
      C2 c;
}
class C2
{
      C1 c;
}
```

So, C2 is of type class, a fact that is re-enforced later in the source code.

8

Linked Lists

VeraLite supports any type of list (for example, integer, string, and class object), with the exception of bit vectors. To use a particular type of linked list, you must create it before the main program and before any list declarations:

```
MakeVeraList (data_type)
```

Note:

There are no terminating semi-colons. You should only enable a particular type of linked list one time, regardless of the of lists you use. You must enable a list type before you declare or use a list of that type.

To use linked lists in VeraLiteVeraLite testbenches containing multiple source files, you must:

- call **MakeVeraList(*type*)** .

- include the VeraListProgram.vrh file only in your source code containing main/program.
- call **ExternVeraList(*type*)** if you want to use a list created in some other file.

In addition to including the file **ListMacros.vrh** in the files where VeraLite linked lists are used, you must include the file **VeraListProgram.vrh** in the main program:

```
#include <VeraListProgram.vrh>
#include   <ListMacros.vrh>
```

If you want to use lists across multiple files, call the VeraLite list macro, **ExternVeraList(*type*),** in the file where you want to use the list.

Multiple includes of ListMacros.vrh are allowed.

The values TRUE and FALSE used with linked lists have been redefined as _VERA_TRUE and _VERA_FALSE respectively.

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 (elements) 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.

List Declaration

Linked lists are supported via a package that is shipped with VeraLite. Alternatively, users can write their own linked list package. To use the VeraLite linked list package, you must:

- enable the list type,
- declare the lists,
- declare the iterators
- include the ListMacros.vrh header file in the file using the list:

```
#include <ListMacros.vrh>
```

- include the file VeraListProgram.vrh in the main program:

```
#include <VeraListProgram.vrh>
```

Creating Lists

VeraLite supports any type of list (for example, integer, string, and packet). To use a particular type of linked list, you must create it before the main program and before any list declarations:

```
MakeVeraList(data_type)
```

Note that there are no terminating semi-colons. You should only enable a particular type of linked list one time, regardless of the of lists you use. You must enable a list type before you declare or use a list of that type.

Declaring Lists

You must declare all lists before using them via the VeraList construct:

```
VeraList_data_type list1, list2, ..., listN;
```

The VeraList construct declares lists of the indicated type. List declaration must occur before the main VeraLite program and after the list enabling statements. Data stored in the list elements must be of the same type as the list declaration.

Declaring List Iterators

You must declare all list iterators before using them via the VeraListIterator construct:

```
VeraListIterator_data_type iterator1, iterator2, ..., iteratorN;
```

The VeraListIterator construct declares list iterators of the indicated type. You must declare iterators as you would any other variable declaration.

Size Methods

This section describes the list methods that analyze list sizes.

size()

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

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

empty()

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

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

Element Access Methods

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

front()

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

list1.front();

back()

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

list1.back();

Iteration Methods

This section describes the list methods used for iteration.

start()

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

list1.start();

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). To access the last element in the list, use **list.finish()** followed by **iterator.prev()**.

Modifying Methods

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

assign()

list1.assign(start_iterator, finish_iterator);

The **assign()** method assigns elements of one list to another:

The method assigns the elements 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.

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.

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();

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, you must **new()** the list. This creates a new list header.

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.

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). When you call 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.

erase_ra0nge()

list1.erase_range(start_iterator, finish_iterator);

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.

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 integer, packet, or string, depending on the VeraList type.

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 must be of type integer, packet, or string, depending on the VeraList type.

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.

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.

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 integer, packet, or string, depending on the VeraList type.

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 indicated 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.

Iterator Methods

This section describes the methods used by iterators.

next()

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

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

prev()

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

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

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.

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.

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 **MakeVeraList(*type*)**.