2. Lexical conventions

EDITORS NOTE: 10th May 2011. I am not going to correct reference hyperlinks. It is easier to do a pass through the document once all the sections are in place to fix these.
The grammar for the Numbers section needs to be updated to reflect the merged grammar.
I don't mention casting in the Real literal constants section
I don't discuss array assignment patterns or how to determine the type (implicit or explicit) of an array literal
2.6.3 - syntax for system tasks and system functions needs to be updated to reflect current merged grammar

2.1 Overview

This clause describes the lexical tokens used in SystemVerilog-AMS HDVL source text and their conventions.

2.2 Lexical tokens

SystemVerilog-AMS HDVL source text files shall be a stream of lexical tokens. A *lexical token* shall consist of one or more characters. The layout of tokens in a source file is free format — that is, spaces and newlines shall not be syntactically significant other than being token separators, except escaped identifiers (see 2.6.1).

The types of lexical tokens in the language are as follows:

— White space
— Comment
— Operator
— Number
— String
— Identifier
— Keyword

2.3 White space

White space shall contain the characters for spaces, tabs, newlines, and formfeeds. These characters shall be ignored except when they serve to separate other lexical tokens. However, blanks and tabs shall be considered significant characters in string literals(see 2.9).

2.4 Comments

The SystemVerilog-AMS HDVL has two forms to introduce comments. A *one-line comment* shall start with the two characters `//` and ends with a newline. *Block comments* shall start with `/*` and end with `*/`. Block comments shall not be nested. The one-line comment token `//` shall not have any special meaning in a block comment.

```
comment ::=                                                    // from A.9.2
        one_line_comment
      | block_comment
one_line_comment ::= // comment_text \n
block_comment ::= /* comment_text */
```

comment_text ::= { Any_ASCII_character }

Syntax 2-1—Syntax for comments

2.5 Operators

Operators are single, double, or triple character sequences and are used in expressions. Clause 4 discusses the use of operators in expressions.

Unary operators shall appear to the left of their operand. *Binary operators* shall appear between their operands. A *conditional operator* shall have two operator characters which separate three operands.

2.6 Identifiers, keywords, and system names

An *identifier* shall be used to give an object a unique name so it can be referenced. An *identifier* shall either be a *simple identifier* or an *escaped identifier* (see 2.6.1). A *simple identifier* shall be any sequence of letters, digits, dollar signs (**$**), and the underscore characters (_).

The first character of a simple identifier shall not be a digit or `$`; it can be a letter or an underscore. Identifiers shall be case sensitive.

Examples:

```
shiftreg_a
busa_index
error_condition
merge_ab
_bus3
n$657
```

Implementations may set a limit on the maximum length of identifiers, but they shall be at least 1024 characters. If an identifier exceeds the implementation-specified length limit, an error shall be reported.

2.6.1 Escaped identifiers

Escaped identifiers shall start with the backslash character (\) and end with white space (space, tab, newline, or formfeed). They provide a means of including any of the printable ASCII characters in an identifier (the decimal values `33` through `126` or `21` through `7E` in hexadecimal).

Neither the leading backslash character nor the terminating white space is considered to be part of the identifier. Therefore, an escaped identifier `\cpu3` is treated the same as a non-escaped identifier `cpu3`.

Examples:

```
\busa+index
\-clock
\***error-condition***
\net1/\net2
\{a,b}
\a*(b+c)
```

2.6.2 Keywords

Keywords are predefined simple identifiers which are used to define the language constructs. A SystemVerilog-AMS HDVL keyword preceded by an escape character is not interpreted as a keyword.

All keywords are defined in lowercase only. Annex B lists all defined SystemVerilog-AMS HDVL keywords.

2.6.3 System tasks and functions

The `$` character introduces a language construct which enables development of user-defined tasks and functions. System constructs are not design semantics, but refer to simulator functionality. A name following the `$` is interpreted as a *system task* or a *system function*.

The syntax for a system task or function is given in Syntax 2-2.

---

analog_system_task_enable ::= *// from A.6.9*
    analog_system_task_identifier [([analog_expression] { , [analog_expression] })] ;

system_task_enable ::= system_task_identifier [([expression] { , [expression] })] ;
    task_enable ::= hierarchical_task_identifier [(expression { , expression })] ;

analog_system_function_call ::= *// from A.8.2*
    analog_system_function_identifier [([analog_expression] { , [analog_expression] })]

system_function_call ::= system_function_identifier
    [(expression { , expression })]

system_function_identifier ::= $ [a-zA-Z0-9_$] { [a-zA-Z0-9_$] } *// from A.9.3*

system_task_identifier ::= $ [a-zA-Z0-9_$] { [a-zA-Z0-9_$] }

Syntax 2-2—Syntax for system tasks and functions

SystemVerilog defines a standard set of system tasks and system functions in this document (see FIX_REF Clause 20 and FIX_REF Clause 21). Additional *user-defined system tasks* and *system functions* can be defined using the PLI, as described in FIX_REF Clause 36. Software implementations can also specify additional system tasks and system functions, which may be tool-specific (see FIX_REF Annex D for some common additional system tasks and system functions). Additional system tasks and system functions are not part of this standard.

Examples:

```
$display ("display a message");
$finish;
```

2.6.4 Compiler directives

The ` character (the ASCII value 0x60, called *grave accent*) introduces a language construct used to implement compiler directives. The compiler behavior dictated by a compiler directive shall take effect as soon as the compiler reads the directive. The directive shall remain in effect for the rest of the compilation unless a different compiler directive specifies otherwise. A compiler directive in one description file can therefore control compilation behavior in multiple description files.

Example:

```
`define WORDSIZE 8
```

SystemVerilog-AMS defines a standard set of compiler directives in this document (see FIX_REF Clause 22). Software implementations can also specify additional compiler directives, which may be tool-specific (see FIX_REF Annex E for some common additional compiler directives). Additional compiler directives are not part of this standard.

2.7 Numbers

Constant numbers can be specified as integer constants (see 2.7.1) or real constants (see 2.7.2). The formal syntax for numbers is listed in Syntax 2-3.

---

number ::= *// from A.8.7*
    decimal_number
  | octal_number
  | binary_number
  | hex_number
  | real_number

real_number ::=
    unsigned_number . unsigned_number
  | unsigned_number [. unsigned_number] exp [sign] unsigned_number
  | unsigned_number [. unsigned_number] scale_factor

exp ::= **e** | **E**

scale_factor ::= **T** | **G** | **M** | **K** | **k** | **m** | **u** | **n** | **p** | **f** | **a**

decimal_number ::=
    unsigned_number
  | [size] decimal_base unsigned_number
  | [size] decimal_base x_digit { _ }
  | [size] decimal_base z_digit { _ }

binary_number ::= [size] binary_base binary_value

octal_number ::= [size] octal_base octal_value

hex_number ::= [size] hex_base hex_value

sign ::= **+** | **-**

size ::= non_zero_unsigned_number

non_zero_unsigned_number ::= non_zero_decimal_digit { _ | decimal_digit}

unsigned_number[2] ::= decimal_digit { _ | decimal_digit }

binary_value[2] ::= binary_digit { _ | binary_digit }

octal_value[2] ::= octal_digit { _ | octal_digit }

hex_value[2] ::= hex_digit { _ | hex_digit }

decimal_base[2] ::= **'**[**s**|**S**]**d** | **'**[**s**|**S**]**D**

binary_base[2] ::= **'**[**s**|**S**]**b** | **'**[**s**|**S**]**B**

octal_base[2] ::= **'**[**s**|**S**]**o** | **'**[**s**|**S**]**O**

hex_base[2] ::= **'**[**s**|**S**]**h** | **'**[**s**|**S**]**H**

non_zero_decimal_digit ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

decimal_digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

binary_digit ::= x_digit | z_digit | **0** | **1**

octal_digit ::= x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

hex_digit ::=
x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
| **a** | **b** | **c** | **d** | **e** | **f** | **A** | **B** | **C** | **D** | **E** | **F**

x_digit ::= **x** | **X**

z_digit ::= **z** | **Z** | **?**

2) Embedded spaces are illegal.

Syntax 2-3—Syntax for integer and real constants

2.7.1 Integer literal constants

Integer literal constants can be specified in decimal, hexadecimal, octal, or binary format.

There are two forms to express integer constants. The first form is a simple decimal number, which shall be specified as a sequence of digits 0 through 9, optionally starting with a plus or minus unary operator. The second form specifies a *based literal constant*, which shall be composed of up to three tokens—an optional size constant, an apostrophe character (', ASCII 0x27) followed by a base format character, and the digits representing the value of the number. It shall be legal to macro substitute these three tokens.

The first token, a size constant, shall specify the size of the integer literal contant in terms of its exact number of bits. It shall be specified as a non-zero unsigned decimal number. For example, the size specification for two hexadecimal digits is 8, because one hexadecimal digit requires 4 bits.

The second token, a `base_format`, shall consist of a case insensitive letter specifying the base for the number, optionally preceded by the single character `s` (or `S`) to indicate a signed quantity, preceded by the apostrophe character. Legal base specifications are `d`, `D`, `h`, `H`, `o`, `O`, `b`, or `B`, for the bases decimal, hexadecimal, octal, and binary respectively.

The apostrophe character and the base format character shall not be separated by any white space.

The third token, an unsigned number, shall consist of digits that are legal for the specified base format. The unsigned number token shall immediately follow the base format, optionally preceded by white space. The hexadecimal digits a to f shall be case insensitive.

Simple decimal numbers without the size and the base format shall be treated as *signed integers*, whereas the numbers specified with the base format shall be treated as signed integers if the s designator is included or as *unsigned integers* if the base format only is used. The s designator does not affect the bit pattern specified, only its interpretation.

A plus or minus operator preceding the size constant is a unary plus or minus operator. A plus or minus operator between the base format and the number is an illegal syntax.

Negative numbers shall be represented in 2’s complement form.

An `x` represents the unknown value in hexadecimal, octal, and binary constants. A `z` represents the high-impedance value. See 6.3 of IEEE Std 1800-2009 SystemVerilog HDVL for a discussion of the SystemVerilog value set. An `x` shall set 4 bits to unknown in the hexadecimal base, 3 bits in the octal base, and 1 bit in the binary base. Similarly, a `z` shall set 4 bits, 3 bits, and 1 bit, respectively, to the high-impedance value.

If the size of the unsigned number is smaller than the size specified for the literal constant, the unsigned number shall be padded to the left with zeros. If the left-most bit in the unsigned number is an `x` or a `z`, then

an x or a z shall be used to pad to the left respectively. If the size of the unsigned number is larger than the size specified for the literal constant, the unsigned number shall be truncated from the left.

The number of bits that make up an unsized number (which is a simple decimal number or a number without the size specification) shall be at least 32. Unsized unsigned literal constants where the high order bit is unknown (X or x) or three-state (Z or z) shall be extended to the size of the expression containing the literal constant.

NOTE—In IEEE Std 1364-1995, in unsized constants where the high order bit is unknown or three-state, the x or z was only extended to 32 bits.

An unsized single-bit value can be specified by preceding the single-bit value with an apostrophe ('), but without the base specifier. All bits of the unsized value shall be set to the value of the specified bit. In a self-determined context, an unsized single-bit value shall have a width of 1 bit, and the value shall be treated as unsigned.

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

The use of x and z in defining the value of a number is case insensitive.

When used in a number, the question-mark (?) character is a SystemVerilog-AMS HDVL alternative for the z character. It sets 4 bits to the high-impedance value in hexadecimal numbers, 3 bits in octal, and 1 bit in binary. The question mark can be used to enhance readability in cases where the high-impedance value is a don't-care condition. See the discussion of **casez** and **casex** in 12.5.1 of IEEE Std 1800-2009 System-Verilog HDVL. The question-mark character is also used in user-defined primitive state tables. See Table 29-1 in 29.3.6 of IEEE Std 1800-2009 SystemVerilog HDVL.

In a decimal literal constant, the unsigned number token shall not contain any x, z, or ? digits, unless there is exactly one digit in the token, indicating that every bit in the decimal constant is x or z.

The underscore character (_) shall be legal anywhere in a number except as the first character. The underscore character is ignored. This feature can be used to break up long numbers for readability purposes.

Several examples of specifying literal integer numbers are as follows:

Example 1 — Unsized literal constant numbers

```
659      // is a decimal number
'h 837FF // is a hexadecimal number
'o7460   // is an octal number
4af      // is illegal (hexadecimal format requires 'h)
```

Example 2 — Sized literal constant numbers

```
4'b1001  // is a 4-bit binary number
'D 3     // is a 5-bit decimal number
3'b01x   // is a 3-bit number with the least
         // significant bit unknown
12'hx    // is a 12-bit unknown number
16'hz    // is a 16-bit high-impedance number
```

Example 3 — Using sign with literal constant numbers

```
8 'd -6  // this is illegal syntax
-8 'd 6  // this defines the two's complement of 6,
         // held in 8 bits—equivalent to -(8'd 6)
4 'shf   // this denotes the 4-bit number '1111', to
```

```
          // be interpreted as a 2's complement number,
          // or '-1'. This is equivalent to -4'h 1
-4 'sd15 // this is equivalent to -(-4'd 1), or '0001'
16'sd?   // the same as 16'sbz
```

Example 4 — Automatic left padding of literal constant numbers

```
logic [11:0] a, b, c, d;
logic [84:0] e, f, g;

initial begin
   a = 'h x;      // yields xxx
   b = 'h 3x;     // yields 03x
   c = 'h z3;     // yields zz3
   d = 'h 0z3;    // yields 0z3

   e = 'h5;       // yields {82{1'b0},3'b101}
   f = 'hx;       // yields {85{1'hx}}
   g = 'hz;       // yields {85{1'hz}}
end
```

Example 5 — Automatic left padding of constant literal numbers using a single-bit value

```
logic [15:0] a, b, c, d;
a = '0;     // sets all 16 bits to 0
b = '1;     // sets all 16 bits to 1
c = 'x;     // sets all 16 bits to x
d = 'z;     // sets all 16 bits to z
```

Example 6— Underscores in literal constant numbers

```
27_195_000                // unsized decimal 27195000
16'b0011_0101_0001_1111   // 16-bit binary number
32 'h 12ab_f001           // 32-bit hexadecimal number
```

Sized negative literal constant numbers and sized signed literal constant numbers are sign-extended when assigned to a **`logic`** data object type logic, regardless of whether or not the **`logic`** itself is signed.

The default length of `x` and `z` is the same as the default length of an integer.

2.7.2 Real literal constants

The *real literal constant numbers* are represented as described by IEEE std 754, an IEEE standard for double precision floating point numbers.

Real numbers shall be specified in either decimal notation (e.g., `14.72`), in scientific notation (e.g., `39e8`, which indicates 39 multiplied by 10 to the 8^{th} power) or in scaled notation (e.g., `24.7K`, which indicates 24.7 multiplied by 10 to the third power). Real numbers expressed with a decimal point shall have at least one digit on each side of the decimal point. The underscore character is legal anywhere in a real literal constant except as the first character of the constant or the first character after the decimal point. The underscore character is ignored.

For example:

```
1.2
0.1
```

```
2394.26331
1.2E12   // the exponent symbol can be e or E
1.30e-2
0.1e-0
23E10
29E-2
236.123_763_e-12  // underscores are ignored
1.3u
7k
```

The following are invalid forms of real numbers because they do not have at least one digit on each side of the decimal point:

```
.12
9.
4.E3
.2e-7
.1p
34.M
```

Table 2-1 describes each symbol and their value used in scaled notation of a real literal constant.

Table 2-1—Scaled Symbols and notation

Symbol	Value
T	1e12
G	1e9
M	1e6
K, k	1e3
m	1e-3
u	1e-6
n	1e-9
p	1e-12
f	1e-15
a	1e-18

No space is permitted between the number and the symbol. Scale factors are not allowed to be used in defining digital delays (e.g., `#5u`).

2.7.3 Implicit conversion of numbers

Real numbers shall be converted to integers by rounding the real number to the nearest integer, rather than by truncating it. Implicit conversion shall take place when a real number is assigned to an integer. The ties shall be rounded away from zero. For example:

— The real numbers 35.7 and 35.5 both become 36 when converted to an integer and 35.2 becomes 35.
— Converting –1.5 to integer yields –2, converting 1.5 to integer yields 2.

2.8 Time Literals

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

```
2.1ns
40ps
```

The time literal is interpreted as a **realtime** value scaled to the current time unit and rounded to the current time precision.

2.9 String literals

A *string literal* is a sequence of characters enclosed by double quotes ("").

Nonprinting and other special characters are preceded with a backslash.

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

Example 1:

```
$display("Humpty Dumpty sat on a wall. \
Humpty Dumpty had a great fall.");
```

prints

```
Humpty Dumpty sat on a wall. Humpty Dumpty had a great fall.
```

Example 2:

```
$display("Humpty Dumpty sat on a wall.\n\
Humpty Dumpty had a great fall.");
```

prints

```
Humpty Dumpty sat on a wall.
Humpty Dumpty had a great fall.
```

SystemVerilog-AMS also includes a **`string`** data type to which a string literal can be assigned. Variables of type **`string`** have arbitrary length; they are dynamically resized to hold any string. String literals are packed arrays (of a width that is a multiple of 8 bits), and they are implicitly converted to the **`string`** type when assigned to a **`string`** type or used in an expression involving **`string`** type operands (see 3.3). String parameters are treated differently and are described in 3.4.6.

For details regarding the assignment of string literals to an integral type refer 5.9 in IEEE Std 1800-2009 SystemVerilog HDVL

2.9.1 Special characters in strings

Certain characters can only be used in string literals when preceded by an introductory character called an *escape character*. Table 2-2 lists these characters in the right-hand column, with the escape sequence that represents the character in the left-hand column.

Table 2-2—Specifying special characters in string

Escape string	Character produced by escape string
\n	New line character
\t	Tab character
\\	\ character
\"	" character
\v	vertical tab
\f	form feed
\a	bell
\ddd	A character specified in 1–3 octal digits ($0 \le d \le 7$) If less than three characters are used, the following character shall not be an octal digit. Implementations may issue an error if the character represented is greater than \377. It shall be illegal for an octal_digit in an escape sequence to be an x_digit or a z_digit.
\xdd	A character specified in 1 to 2 hex_digits. If only one digit is used, the following character shall not be a hex_digit. It shall be illegal for a hex_digit in an escape sequence to be an x_digit or a z_digit.

2.10 Array literals

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

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

The nesting of braces shall follow the number of dimensions, unlike in C. However, replication operators can be nested. The inner pair of braces in a replication is removed. A replication expression only operates within one dimension.

```
integer n[1:2][1:6] = '{2{'{3{4, 5}}}}; // same as
'{'{4,5,4,5,4,5},'{4,5,4,5,4,5}}
```

2.11 Attributes

A mechanism is included for specifying properties about objects, statements, and groups of statements in the SystemVerilog-AMS source that can be used by various tools, including simulators, to control the operation

or behavior of the tool. These properties are referred to as *attributes*. This subclause specifies the syntactic mechanism used for specifying attributes, without standardizing on any particular attributes.

The syntax is found in Syntax 2-4.

---

attribute_instance ::= **(*** attr_spec { **,** attr_spec } ***)** *// from A.9.1*

attr_spec ::= attr_name [**=** constant_expression]

attr_name ::= identifier

Syntax 2-4—Syntax for attributes

An *attribute_instance* can appear in the SystemVerilog-AMS description as a prefix attached to a declaration, a module item, a statement, or a port connection. It can appear as a suffix to an operator or a function name in an expression.

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

If the same attribute name is defined more than once for the same language element, the last attribute value shall be used; and a tool can issue a warning that a duplicate attribute specification has occurred.

Nesting of attribute instances is disallowed. It shall be illegal to specify the value of an attribute with a constant expression that contains an attribute instance.

Refer to REF:Annex A for the syntax of specifying an attribute instance on specific language elements. Several examples are illustrated below.

Example 1 — The following example shows how to attach attributes to a case statement:

```
(* full_case, parallel_case *)
case (foo)
   <rest_of_case_statement>
```

or

```
(* full_case=1 *)
(* parallel_case=1 *) // Multiple attribute instances also OK
case (foo)
   <rest_of_case_statement>
```

or

```
(* full_case, // no value assigned
parallel_case=1 *)
case (foo)
   <rest_of_case_statement>
```

Example 2 — To attach the full_case attribute, but NOT the parallel_case attribute:

```
(* full_case *) // parallel_case not specified
case (foo)
   <rest_of_case_statement>
```

or

```
(* full_case=1, parallel_case = 0 *)
case (foo)
   <rest_of_case_statement>
```

Example 3 — To attach an attribute to a module definition:

```
(* optimize_power *)
module mod1 (<port_list>);
```

or

```
(* optimize_power=1 *)
module mod1 (<port_list>);
```

Example 4 — To attach an attribute to a module instantiation:

```
(* optimize_power=0 *)
mod1 synth1 (<port_list>);
```

Example 5 — To attach an attribute to a **reg** declaration:

```
(* fsm_state *) logic [7:0] state1;
(* fsm_state=1 *) logic [3:0] state2, state3;
logic [3:0] reg1; // reg1 does NOT have fsm_state set
(* fsm_state=0 *) logic [3:0] reg2; // nor does this one
```

Example 6 — To attach an attribute to an operator:

```
a = b + (* mode = "cla" *) c; // sets the value for the attribute mode
                              // to be the string cla
```

Example 7 — To attach an attribute to a function call:

```
a = add (* mode = "cla" *) (b, c);
```

Example 8 — To attach an attribute to a conditional operator:

```
a = b ? (* no_glitch *) c : d;
```

2.11.1 Standard attributes

The SystemVerilog-AMS HDVL standardizes the following attributes:

— The `desc` attribute is used to generate help messages when attached to parameter, variable and net declarations within a module. The attribute must be assigned a string. See 3.4.3.

— The `units` attribute is used to describe the units of the parameter or variable which it is attached to within a module. The attribute must be assigned a string. See 3.4.3.