DesignCon 2004

VHDL-200X and the Future of VHDL

Jim Lewis, SynthWorks
jim@SynthWorks.com

Stephen Bailey, Mentor Graphic's Model Technology Group

Erich Marschner, Cadence Design Systems

J. Bhasker, eSilicon Corp.

Peter Ashenden, Ashenden Designs Pty. Ltd.

Abstract

VHDL is a critical language for RTL design and is a major component of the $200+ million RTL simulation market[1]. Many users prefer to use VHDL for RTL design as the language continues to provide desired characteristics in design safety, flexibility and maintainability. While VHDL has provided significant value for digital designers since 1987, it has had only one significant language revision in 1993. It has taken many years for design state-of-practice to catch-up to and, in some cases, surpass the capabilities that have been available in VHDL for over 15 years. Last year, the VHDL Analysis and Standardization Group (VASG), which is responsible for the VHDL standard, received clear indication from the VHDL community that it was now time to look at enhancing VHDL.

In response to the user community, VASG initiated the VHDL-200x project[2]. VHDL-200x will result in at least two revisions of the VHDL standard. The first revision is planned to be completed next year (2004) and will include a C language interface (VHPI); a collection of high user value enhancements to improve designer productivity and modeling capability and potential inclusion of assertion-based verification and testbench modeling enhancements. A second revision is planned to follow about two years later.

This paper summarizes VHDL-200X enhancements proposed for the first revision of VHDL. It also summarizes efforts being made by the IEEE 1164 (package std_logic_1164), IEEE 1076.3 (the package numeric_std), and IEEE 1076.6 (VHDL RTL Synthesis). Note, that this is work in progress and is subject to change.

Author Biographies

Jim Lewis, SynthWorks

Jim Lewis, the founder of SynthWorks, has seventeen years of design, teaching, and problem solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr. Lewis does ASIC and FPGA design, custom model development, and consulting. Mr. Lewis is an active member of VHDL Standards groups including, VHDL-200X (IEEE 1076), RTL Synthesis (IEEE 1076.6), Std_Logic (IEEE 1164), and Numeric_Std (IEEE 1076.3). Mr Lewis can be reached at jim@SynthWorks.com,

Stephen Bailey, Model Technology/Mentor Graphics

Stephen Bailey is a Technical Marketing Engineer in the Model Technology group of Mentor Graphics. Steve has been involved with digital simulation and the VHDL language definition for over 10 years and is currently chair of VHDL Analysis and Standardization Group (VASG). Prior to joining the EDA industry, Steve developed embedded real-time software systems and software development tools. He has a BS and MS in Computer Science.

Erich Marschner, Cadence Design Systems

Erich Marschner is a Senior Architect with Cadence Design Systems, responsible for advanced verification technologies, and Co-Chair of the Accellera Formal Verification Technical Committee (FVTC), which has defined the Accellera Property Specification Language (PSL). At Cadence, he has focused on transaction- and assertion-based verification involving both simulation and formal verification. Previously, as a Fellow at COMPASS Design Automation, he was responsible for VHDL equivalence checking and related technology. Prior to that, as President and co-founder of CLSI, he defined the initial (1987) version of IEEE Std 1076 VHDL and directed development of VHDL language analysis, database, and high-level synthesis technology.

J. Bhasker, eSilicon Corp.

J. Bhasker is an expert in the area of hardware description languages and RTL synthesis. He has written a number of books on these topics. He was also a distinguished member of technical staff at Bell Laboratories. He has published a number of papers in journals and conferences, mostly in the area of design automation and high-level synthesis algorithms. He is currently the chair of two working groups: the IEEE 1076.6 VHDL Synthesis working group and the IEEE 1364.1 Verilog Synthesis Working Group. He was also a major contributor to the IEEE Std 1076.3 NUMERIC packages.

Peter Ashenden, Ashenden Designs Pty. Ltd.

Dr. Ashenden received his B.Sc.(Hons) and Ph.D. from Adelaide University, South Australia. He is an independent consultant specializing in electronic design automation (EDA). He is actively involved in IEEE working groups developing VHDL standards, and is Chair of the IEEE Design Automation Standards Committee. He is the author of The Designer's Guide to VHDL, The Student's Guide to VHDL and coauthor of The Designer's Guide to VHDL-AMS. Dr. Ashenden is a Senior Member of the IEEE and a member of the ACM.

1. Introduction

This paper summarizes enhancements being made to VHDL by several IEEE working groups. You will note that the plural "groups" is correct as VHDL designers actually use several standards. A few of these are IEEE 1076 (VHPI and VHDL-200X), IEEE 1164 (the package std_logic_1164), IEEE 1076.3 (the package numeric_std plus the proposed packages for floating point), and IEEE 1076.6 (VHDL RTL Synthesis).

2. VHDL Programming Interface (VHPI)

A programming interface to VHDL has been defined and is in process of being integrated into the VHDL LRM. The VHPI will open the door for easy development and integration of 3rd party tools with any LRM compliant VHDL simulator. By lowering the cost of supporting VHDL, the VHPI will encourage developers of innovative tools to support the VHDL user community. Bleeding-edge users can exploit the VHPI to integrate their proprietary verification tools and utilities while maintaining the ability to easily switch between simulation vendors as market conditions indicate.

The VHPI also opens the door for users to integrate C language models and utilities with their VHDL designs using a standard interface portable to any VHDL simulator. IP providers can distribute IP for users of any VHDL simulator via the VHPI.

3. VHDL-200X Work

3.1 Assertion-based verification

Assertion-based verification involves adding behavioral specifications to a design in order to improve verification efficiency. These specifications can define requirements on design behavior that can be checked both statically, using formal verification techniques, and dynamically, during simulation. They can define both intended design behavior, which should be demonstrated during verification, and potential error situations, which should not occur during verification. They give added visibility into the internal state of a design.

Two types of behavior specifications are involved: “assertions”, which define correct local behavior or represent error conditions; and “coverage monitors”, which detect the occurrence of interesting behavior patterns. Assertions detect errors at or near the source, facilitating debug compared to traditional black-box simulation that detects errors if and when they propagate to a primary output. Coverage monitors detect intended conditions or behavior patterns that occur in the system and record that information for functional coverage analysis.

Assertions help ensure that interface specifications are captured along with the design. For IP delivery, embedded assertions are delivered as part of the IP and are used to verify correct integration of the IP into a larger system. The designer's assumptions and decisions about the implementation can also be captured as assertions, to improve maintainability and reusability of the design. Coverage monitors allow the designer to document the significant corner cases to be tested, and transmit design knowledge directly into the verification process improving verification engineer productivity.

3.1.1 PSL

Over the past several years, Accellera[3] developed a standard language for assertion-based verification. The Property Specification Language (PSL), is based upon the language Sugar[4], developed by IBM Haifa Research Labs. PSL version 1.01[5] became an Accellera standard in May 2003. Verification tools supporting PSL have been available for more than a year with more tool support coming.

PSL is a comprehensive language that supports various styles of behavioral specification, including Linear Time Logic (LTL) specifications, Regular Expression style specifications, and Computation Tree Logic (CTL) specifications. This variety supports formal verification and simulation tools. The language definition identifies a subset that is suited for verification flows in which both simulation and formal verification work together. This subset, the "Simple Subset", essentially consists of behavioral specifications in which time moves monotonically forward (left to right).

3.1.2 PSL in VHDL

Using PSL with VHDL is already possible today. PSL declarations and statements can be added to a VHDL design in the form of pragmas, or structured comments, that begin with the string "--psl". This convention for embedding PSL declarations and statements into HDL code is supported by many vendors today. But a much better solution would be to add PSL declarations and statements to VHDL as first-class language constructs. In VHDL-200x, the proposal is to incorporate the Simple Subset of PSL as part of VHDL.

VHDL has had assertion statements (both concurrent and sequential) since it first became a standard in 1987, so assertions are not new to VHDL. However, the existing VHDL assertion statements can only express combinational invariants, because they are defined in terms of simple Boolean expressions. PSL provides an opportunity to extend the definition of concurrent assertions in VHDL to include the ability to describe sequential behavior. This would dramatically increase the utility of VHDL concurrent assertions for both simulation and formal verification applications. Similarly, addition of the cover directive to VHDL would allow native support for coverage monitoring within VHDL designs.

PSL enhanced assertions would provide native assertion-based verification support for VHDL users. Users could specify very powerful properties and assertions:

1. Unclocked, combinational invariants that prohibit/detect glitches in asynchronous control inputs:

   ```
   assert always sel0 or sel1;
   assert never clka and clkb;
   ```

2. Clocked, combinational invariants that check relationships among synchronous control inputs:

   ```
   assert always (sel0 or sel1) @ rising(clk);
   ```

3. Singly-clocked, sequential assertions that describe paths through a state machine or interactions among state machines that have the same clock:

   ```
   assert always (req -> next(gnt before req))@ rising(clk);
   ```

4. Multiply-clocked, sequential assertions that describe interactions between clock domains:

   ```
   assert always (req@clk1) -> eventually (gnt@clk2);
   ```

5. Express sequences of control conditions that make up a behavior pattern, together with various forms of implication operators:

```
assert always {{req[*]; ack} && {rwbar[*]}} |->
   {{{rwbar[*]} &&{[*1:3]; drdy}};not req; not ack} @ rising(clk);
```

which says that, once a read transaction has started (a request followed by an acknowledge, with read/write bar high all the while), then it must complete (with data ready asserted after 1 to 3 cycles while read/write bar stays high, and then request is deasserted, and then acknowledge is deasserted).

6. Cover directives, which can use sequential regular expressions (or sequences) to describe behavior patterns that should be observed sometime during the verification flow. For example, a coverage monitor for the read transaction mentioned above might be expressed simply, as follows:

```
cover {{rwbar[*]} && {{req[*]; ack; [*1:3]; drdy}};  not req; not ack} @ rising(clk);
```

3.2 Testbench and verification

Today many users resort to high-level verification languages to access language features that are required for verification. However, the same users have clearly identified that it would be far easier for users to continue to use VHDL if these verification features were available.

Based on user feedback, a number of areas have been identified that can help improve testbench writing and consequently verification. The top priorities are:

- Associative arrays
- Fork-join
- Queues
- FIFOs
- Lists
- Synchronization and handshaking (event objects)
- Request and wait for action
- Expected value detectors
- Access to coverage data for reactive TB
- Sparse arrays
- Random value generation with optional and dynamic weighting
- Random object initialization
- Random 2-state value resolution
- Loading and dumping memories

Progress has been made in several of these areas including associative arrays, fork-join, queues, fifos, lists and event objects.

3.2.1 Associative arrays

The following is an example of how an associative array would be declared:

```
type BitAssocArrayT is  associative (INTEGER) of BIT;
variable MemoryA1: BitAssocArrayT;
type COLOR is (RED, BLUE, GREEN, YELLOW, ORANGE);
type ColorAssocArrayT is associative (COLOR, COLOR) of INTEGER;
signal ScoreBoard: ColorAssocArrayT;
```

A number of subprograms are implicitly defined for associative array objects. These subprograms allow deleting an element, checking existence of an element, returning the number of elements, getting the first, next, last or previous elements, dumping and loading an array.

3.2.2 Lists

The following is an example of list declaration and use.

```
type IntListT is list (<>) of INTEGER; -- unconstrained list
variable usb_fan: IntListT;  -- List is empty by default.
…
type BvListT is list (0 to 5) of BIT_VECTOR(0 to 3);  -- A constrained list.
type StringListT is list (3 downto 0) of STRING (0 to 5);
signal usb_data: BvListT := (“001”, “000”, “000”);  -- three element list indexed from 0 to 2
```

A number of predefined attributes (operations) are implicitly defined for a list type to allow list operations. These include ‘DELETE, ‘INSERT, ‘LENGTH, ‘SORT, ‘UNIQUE, ‘REVERSE, ‘EXISTS, and ‘INDEX.

3.2.3 Fork-Join

The fork-join construct defines the concept of a sequential block that is a sequential statement. Each sequential block within a fork-join is executed concurrently with each other. Sequential statement execution proceeds when the fork-join reaches the end condition specified (any sequential block completes, all complete, etc.).

```
 [ fj_label :]  fork
   [ [sblk_labela:] declare -- sequential block
      { declarations } ]
   begin
      { sequential statements }
   end declare [ sblk_label_a ] ;
    [ [sblk_label_b:] declare         -- another sequential block
      { declarations } ]
   begin
      { sequential statements }
   end declare [ sblk_label_b ];
. . .
join [ all | none | first |  [ condition_clause ]  [ timeout_clause ] ]  [ fj_label ];
```

Note that the testbench and verification group is considering the SUAVE project[6] proposal for dynamic instantiation of processes as an alternative to the fork-join proposal.

3.3 Data Types and Abstractions

Another team is looking at extensions to VHDL’s data types and abstractions. For object orientation, the group is considering work performed in the SUAVE project. The SUAVE proposal extends existing VHDL features (records, generics, and packages) giving VHDL features similar to template classes in

C++[7] and type generics in Ada[8]. These features give VHDL the capability to build packages that implement templates for linked lists, queues, and similar abstractions. These features are being carefully traded-off with the testbench and verification team since there is overlap between the features being worked on by both groups.

3.4 Modeling productivity and capabilities

The goal of the modeling and productivity group is to improve designer productivity through enhancing conciseness, simplifying common occurrences of code, and improving capture of intent. The following are examples of proposed enhancements.

3.4.1 Unary reduction operators

Unary reduction operators simplify the application of a logic operator to each element of an array. This proposal creates unary versions of AND, OR, NOR, NAND, XOR, and XNOR that are reduction operators. These operators will have the same precedence as the miscellaneous operators (**, ABS, and NOT).

This proposal makes the code for Parity2 a simplification of the code for Parity1:

```
parity1 <=
  (data(7) xor data(6) xor data(5) xor data(4) xor
   data(3) xor data(2) xor data(1) xor data(0)) and ParityEnable ;

Parity2 <= xor Data and ParityEnable;
```

3.4.2 Array/Scalar logical operators

These operators overload the binary logical operators. The result is a bit_vector with one operand bit and the other bit_vector. This proposal makes the code for DataOut2 a short hand for the code in DataOut1:

```
GenLoop : for I in DataOut1'Range loop
begin
   DataOut1(I) <=
       (A(I) and ASel) or (B(I) and BSel) or
       (C(I) and CSel) or (D(I) and DSel) ;
end generate;

DataOut2 <=
     (A and ASel) or (B and BSel) or
     (C and CSel) or (D and DSel) ;
```

Without these operators, a designer would likely write the above code as follows:

```
Y <=
    A when ASel = '1' else  B when BSel = '1' else
    C when CSel = '1' else   D when DSel = '1' else  (others => '0') ;
```

When the select signals (ASel, …) are mutually exclusive, the hardware functions correctly. However, it implies priority select logic and is inefficient from a hardware area and timing perspective.

3.4.3 Hierarchical signal access

A missing feature from VHDL is the ability for an entity (such as a testbench) to probe and force signals that are defined in another part of the design hierarchy (such as a tri-state enable output of the chip). Some simulator vendors have created their own proprietary package of probe and force functions. The standards group has requested donations and will evaluate these donations in creating a standard package.

3.4.4 Formatted IO: hwrite, hread, owrite, oread, …

The addtion of overloadable, standard routines to write or read formatted values such as hexadecimal or octal will reduce I/O while allowing input and output values that are easier to read and maintain.

3.4.5 String conversions: to_string, to_hstring, to_ostring, …

This proposal defines conversion functions that convert a value to a string. In concept this is similar to the existing ‘image attribute, except that with to_string, support of array values will be possible. In addition, formatted values will also be supported (such as hexadecimal and octal).

3.4.6 Sized bit string literals

Currently array bit string literals have no capability to specify a size. As a result, the size is inferred from the value. For hexadecimal values this implies that the size of the resulting literal is a multiple of four bits. Hence, X”FF”, is a bit string literal that is exactly 8 bits in size.

Adding an integer value to the front of the bit string literal to determine the size of the value allows quicker specification of bit string literals that are zero or signed extended. If the string value does not specify enough digits, by default the value will be extended by unsigned value rules. Specification of an additional prefix, “S”, indicates the value is to be extended in a signed manner. Support for X, - and Z characters is also provided. Examples of values and their interpretation:

```
3X"7"      is equivalent to "111"
32X"F"     is equivalent to "00000000000000000000000000001111"
32SX"F"    is equivalent to "11111111111111111111111111111111"
16SX"X"    is equivalent to "XXXXXXXXXXXXXXXX"
7X"8F"     is equivalent to "0001111"     -- a warning maybe issued here
```

3.4.7 Conditional and selected signal assignment in sequential code

Currently conditional and selected signal assignments are only permitted as concurrent statements. Permitting them in processes will allow the following sequential code:

```
if (FP = '1') then
   NextState  <= FLASH ;
else
   NextState  <= IDLE ;
```

```
end if ;
```

to be written more concisely as:

```
NextState <= FLASH when (FP = '1') else IDLE ;
```

3.4.8 Case/IF generate

Currently VHDL has an “IF” and a “FOR” format of generate. The if-generate does not have an else. This proposal seeks to add an “ELSE” to “IF” generate and also add a case generate for mutually exclusive generate alternatives.

3.4.9 Reading driving values of output ports

Currently VHDL does not allow the value to be read from an output port of an entity. Allowing the reading of the driving value will facilitate certain design and verification capabilities such as referencing the driving value in an assertion.

3.4.10 Permit port maps to contain expressions

Enhancing port maps to allow passing an expression to an input port eliminates the need to explicitly declare an intermediate signal and assign it the expression.

3.4.11 Process sensitivity list abbreviation

There have been numerous requests to simplify sensitivity lists. Allowing the keyword all to be used as the sensitivity list to replace the necessity of a sensitivity list and/or adding new keywords such as comb (combinational) to capture that the process is sensitive to all input signals and contains only combinational logic are some of the proposals being evaluated.

3.4.12 Case statement improvements

Currently the case statement requires the case statement expression to be a locally static type, case alternatives to be locally static, and exact matching (can’t exploit the don’t care metalogic value). The case statement will be enhanced for greater functionality and easier use.

3.5 VHDL-200X, Second Revision Plans

In the 2nd revision, the VASG will be exploring language changes that will improve tool (primarily simulation/verification) performance, continued enhancements to improve designer and verification engineer productivity and modeling capabilities – including higher levels of abstractions such as object-orientation and interface modeling, standardizing environmental features (simulation control, library mapping) for better tool portability and enhancements to target specific modeling styles such as asynchronous design.

4. New Features for Std_Logic_1164 and Numeric_Std

This section contains proposals for additions to the packages std_logic_1164, numeric_std, and numeric_bit. Although numeric_bit is not mentioned in the discussion, everything that applies to numeric_std also applies to numeric_bit. Enhancements include:

- logical reduction functions,
- array/scalar logic operations,
- array/scalar addition operators,
- TO_X01, TO_X01Z, TO_UX01, IS_X for unsigned and signed,
- shift operators for std_logic_vector and std_ulogic_vector
- unsigned arithmetic for std_logic_vector and std_ulogic_vector (new package),
- textio for types in std_logic_1164 and numeric_std (two new packages),
- floating point arithmetic (new packages)

The logic reduction functions and array/scalar logic operations are as described in the VHDL-200x topics and will not be covered in further detail here.

4.1 Array/scalar addition operators: numeric_std

Overload the addition operators to support mixing an array with a scalar for unsigned and signed. Functions for "+" and "-" will be defined in the following form:

```
function "+"(L: unsigned; R: std_ulogic) return unsigned;
function "+"(L: std_ulogic; R: unsigned) return unsigned;
```

These functions facilitate writing the following add with carry in:

```
signal Cin  : std_logic ;
signal A, B          : unsigned(7 downto 0) ;
signal Y    : unsigned(8 downto 0) ;
. . .
Y <= A + B + Cin ;
```

They also facilitate writing the following conditional incrementer:

```
signal IncEn                         : std_logic ;
signal IncCur, IncNext     : unsigned(7 downto 0) ;
. . .
IncNext <= IncCur + IncEn ;
```

4.2 TO_X01, TO_X01Z, TO_UX01, IS_X: numeric_std

Add the strength reduction and 'X' detection operators for unsigned and signed. Currently these operators are only supported for std_logic_vector and std_ulogic_vector. Functions for TO_X01, TO_X01Z, TO_UX01, and IS_X will be defined in the following form:

```
function  To_X01 ( s : unsigned ) return unsigned;
function  Is_X     ( s : unsigned ) return boolean;
```

These functions are useful for testbenches to handle 'X's and resistive strength driving levels. It is also appropriate to use these functions in input pad cells of ASIC and FPGA libraries. In an RTL design, logic should only generate the values '0', '1', '-', and 'X'.

4.3 Shift operators: std_logic_1164, numeric_std?

Overload shift operators for std_logic_vector and std_ulogic_vector. Functions for sll, srl, sla, sra, rol, and ror will be defined in the following form:

```
function "sll" ( l : std_logic_vector;   r : integer )  return   std_logic_vector;
function "sll" ( l : std_ulogic_vector; r : integer )  return   std_ulogic_vector;
```

Numeric_std currently supports sll, srl, rol, and ror. Support is being considered for sla and sra.

4.4 Unsigned arithmetic for std_logic_vector and std_ulogic_vector: new package

Create a new package that implements unsigned arithmetic operators for std_logic_vector and std_ulogic_vector. Tenatively the package is named numeric_unsigned. It will include all functions included in numeric_std minus the ones that are in std_logic_1164 (or planned for std_logic_1164).

A testbench is one of the places that will benefit most. Testbenches often need to perform a numeric algorithm on an object that is not numeric in a formal sense. For example, the following code shows data being written to consecutive bits in a RAM with exactly one bit set in each data word.

```
constant  CHIP1_RAM_BASE : std_logic_vector(31 downto 0) := X"40000000" ;
constant  ZERO_DATA : std_logic_vector(31 downto 0) := (others => '0') ;
. . .
for i in 0 to 31 loop
    CpuWrite(CpuRec, CHIP1_RAM_BASE + i , ZERO_DATA + 2**i ); -- subprogram call
end loop ;
```

For RTL design the existence of this package permits one of three methodology variations:

1) Strict: Use only types unsigned and signed. Do not use numeric_unsigned.
2) Semi-Strict: Use unsigned and signed for all math operations except counters.
3) Flexible: Use std_logic_vector for any unsigned operation.
   Use signed for all signed operations.

Note, this proposal does not include a numeric_signed package. Use ieee.numeric_std.signed for signed operations.

4.5 Textio for std_logic_1164 and numeric_std types: (two new packages)

Overload read and write procedures to support std_ulogic, std_logic, std_ulogic_vector, std_logic_vector, unsigned, and signed. Functions for read and write will be defined in the following forms:

```
procedure READ( L: inout LINE; VALUE out std_logic; GOOD: out BOOLEAN);
procedure READ( L: inout LINE; VALUE out std_logic);
procedure WRITE( L: inout LINE; VALUE in std_logic; JUSTIFIED: in SIDE:= RIGHT;
                FIELD: in WIDTH:= 0);
```

Overload read and write procedures to support base operations with std_ulogic_vector, std_logic_vector, unsigned, and signed. Functions for read and write will be defined in the following forms:

```
type REPRESENTATION is ( any, binary, octal, hexadecimal );
procedure READ( L: inout LINE; VALUE out std_logic;
                GOOD: out BOOLEAN; R: in REPRESENTATION := any );
procedure WRITE( L: inout LINE; VALUE in std_logic;
                 JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH:= 0;
                 R: in REPRESENTATION := any);
```

4.6 Summary of enhancements to std_logic_1164 and numeric_std

With the addition of the new package features, the following table summarizes the operations supported by the std_logic_1164 and numeric_std packages:

Operators	**Left**	**Right**	**Result**	**Notes:**
Logic	TypeA	TypeA	TypeA	Array = unsigned, signed, std_ulogic_vector, std_logic_vector TypeA = boolean, std_logic, std_ulogic, Array * for comparison operators the result is boolean
Numeric	Array	Array	Array, *	
	Array	Integer	Array, *	
	Integer	Array	Array, *	
Logic, Addition	Array	Std_ulogic	Array	
	Std_ulogic	Array	Array	
Logic Reduction		Array	Std_ulogic	

5. Floating point arithmetic

A family of VHDL packages are being introduced to implement fixed point and floating point arithmetic. For more information, see David Bishop's paper from DVCon 2003[9]. Also see http://www.eda.org/fphdl.

6. Enhancements to the VHDL RTL Synthesis Standard

The IEEE P1076.6-200X effort has updated the VHDL RTL synthesis standard and is currently working through the final stages of the balloting process. This standard broadens the subset of code that is considered to be synthesizable. Details of this effort are documented in the paper Lewis[10].

7. Summary

Driven by end user requests and demand, the VHDL working groups are working quickly, yet diligently, to enhance VHDL to meet the growing need to reduce design and verification times and increase design quality to meet the needs of today and tomorrow's system-on-chip and system designs. The working groups are pleased to have such a high level of participation from users and support from EDA vendors.

Working together, VHDL 200x will prove to be a significant step forward in RTL and higher-levels of design and verification.

8. Participating In Standards

VHDL standards are IEEE standards. As a VHDL community member it is both your right and responsibility to join IEEE committees and participate in VHDL standards. If you don't participate, the changes you envision and wish for (no matter how simple or obvious) will not happen. To find out more about participating in VHDL standards go the the web links, http://www.eda.org and http://www.SynthWorks.com/VhdlLinks.htm.

9. References

1. Gartner Dataquest 2002 EDA Forecast, CAE, RTL Simulation Market Forecast

2. See http://www.eda.org/vhdl-200x for more information.

3. http://www.accellera.org

4. http://www.haifa.il.ibm.com/projects/verification/sugar/index.html

5. http://www.accellera.org/pslv101.pdf

6. See SUAVE proposal (http://www.ashenden.com.au/suave.html) for such an approach.

7. Programming Languages – C++, INCITS/ISO/IEC 14882, American National Standard (INCITS-Adopted ISO/IEC Standard), 15 Sep 1998.

8. Information Technology – Programming Langauges – Ada, INCITS/ISO/IEC 8652, American National Standard (INCITS-Adopted ISO/IEC Standard), 1 May 1995.

9. Bishop, David "Floating Point for VHDL and Verilog", DVCon 2003 proceedings.

10. Lewis, Jim "IEEE 1076.6-200X: VHDL Synthesis Coding Styles for the Future", to be published in DVCon 2004 proceedings.