

# HDL vs HVL

Jyotirmoy Pathak

School of Electronics and Electrical Engineering  
Lovely Professional University

## Abstract

Hardware description language (HDL) describes the behavior of the module. Two preferred HDL languages are VHDL and Verilog. Functionality verification is carried out with testbench module. As design sizes have increased, however, the amount of verification required has escalated dramatically. While writing the test bench and verification routines in pure Verilog HDL is still possible, the amount of coding far exceeds what can be accomplished in a reasonable amount of time. So along came proprietary Hardware Verification Languages (HVLs) such as VERA to the rescue. These languages specialized in giving verification engineers powerful constructs to describe stimulus and verify functionality in a much more concise manner. These proprietary languages solve a need, but at the costs of requiring engineers to learn and work with multiple languages, and often at the expense of simulation performance. Having different languages for the hardware modeling and the hardware verification has also become a barrier between those engineers doing the design work and those doing the verification.

**Keywords** HDL, HVL, System-C, Verilog, Verification

## 1. Introduction

In electronics, a hardware description language or HDL is a specialized computer language used to program the structure, design and operation of electronic circuits, and most commonly, digital logic circuits. A hardware description language enables a precise, formal description of an electronic circuit that allows for the automated analysis, simulation, and simulated testing of an electronic circuit. It also allows for the compilation of an HDL program into a lower level specification of physical electronic components, such as the set of masks used to create an integrated circuit. A hardware description language looks much like a programming language such as C; it is a textual description consisting of expressions, statements and control structures. One important difference between most programming languages and HDLs is that HDLs explicitly include the notion of time. HDLs form an integral part of electronic design automation systems, especially for complex circuits, such as microprocessors.

A hardware verification language, or HVL, is a programming language used to verify the designs of electronic circuits written in a hardware description language. HVLs typically include features of a high-level programming language like C++ or Java as well as features for easy bit-level manipulation similar to those found in HDLs. Many HVLs will provide constrained random stimulus generation, and functional coverage constructs to assist with complex hardware verification.

## 2. Control-Flow Languages

It is certainly possible to represent hardware semantics using traditional programming languages such as C++, which operate on control flow semantics as opposed to data flow, although to function as such, programs must be augmented with extensive and unwieldy class libraries. Generally, however, software programming languages do not include any capability for explicitly expressing time, and thus cannot function as hardware description languages. Before the recent introduction of SystemVerilog, C++ integration with a logic simulator was one of the few ways to use object-oriented programming in hardware verification. SystemVerilog is the first major HDL to offer object orientation and garbage collection.

Using the proper subset of hardware description language, a program called a synthesizer, or logic synthesis tool, can infer hardware logic operations from the language statements and produce an equivalent netlist of generic hardware primitives to implement the specified behaviour. Synthesizers generally ignore the expression of any timing constructs in the text. Digital logic synthesizers, for example, generally use clock edges as the way to time the circuit, ignoring any timing constructs. The ability to have a synthesizable subset of the language does not itself make a hardware description language.

## 3. Hardware Design Language

### 3.1 Description of HDL

HDLs are standard text-based expressions of the structure of electronic systems and their behavior over time. Like concurrent programming languages, HDL syntax and semantics include explicit notations for expressing concurrency. However, in contrast to most software programming languages, HDLs also include an explicit notion of time, which is a primary attribute of hardware. Languages whose only characteristic is to express circuit

connectivity between a hierarchy of blocks are properly classified as netlist languages used in electric computer-aided design (CAD). HDL can be used to express designs in structural, behavioral or register-transfer-level architectures for the same circuit functionality; in the latter two cases the synthesizer decides the architecture and logic gate layout.

HDLs are used to write executable specifications for hardware. A program designed to implement the underlying semantics of the language statements and simulate the progress of time provides the hardware designer with the ability to model a piece of hardware before it is created physically. It is this executability that gives HDLs the illusion of being programming languages, when they are more precisely classified as specification languages or modeling languages. Simulators capable of supporting discrete-event (digital) and continuous-time (analog) modeling exist, and HDLs targeted for each are available.

### 3.2. Design using HDL



As a result of the efficiency gains realized using HDL, a majority of modern digital circuit design revolves around it. Most designs begin as a set of requirements or a high-level architectural diagram. Control and decision structures are often prototyped in flowchart applications, or entered in a state diagram editor. The process of writing the HDL description is highly dependent on the nature of the circuit and the designer's preference for coding style. The HDL is merely the 'capture language', often beginning with a high-level algorithmic description such as a C++ mathematical model. Designers often use scripting languages such as Perl to automatically generate repetitive circuit structures in the HDL language. Special text editors offer features for automatic indentation, syntax-dependent coloration, and macro-based expansion of entity/architecture/signal declaration.



The HDL code then undergoes a code review, or auditing. In preparation for synthesis, the HDL description is subject to an array of automated checkers. The checkers report deviations from standardized code guidelines, identify potential ambiguous code constructs before they can cause misinterpretation, and check for common logical coding errors, such as dangling ports or shorted outputs. This process aids in resolving errors before the code is synthesized. In industry parlance, HDL design generally ends at the synthesis stage. Once the synthesis tool has mapped the HDL description into a gate netlist, the netlist is passed off to the back-end stage. Depending on the

physical technology (FPGA, ASIC gate array, ASIC standard cell), HDLs may or may not play a significant role in the back-end flow. In general, as the design flow progresses toward a physically realizable form, the design database becomes progressively more laden with technology-specific information, which cannot be stored in a generic HDL description. Finally, an integrated circuit is manufactured or programmed for use.

### 3.3 Simulating and Debugging HDL code

Essential to HDL design is the ability to simulate HDL programs. Simulation allows an HDL description of a design (called a model) to pass design verification, an important milestone that validates the design's intended function (specification) against the code implementation in the HDL description. It also permits architectural exploration. The engineer can experiment with design choices by writing multiple variations of a base design, then comparing their behavior in simulation. Thus, simulation is critical for successful HDL design.

To simulate an HDL model, an engineer writes a top-level simulation environment (called a testbench). At minimum, a testbench contains an instantiation of the model (called the device under test or DUT), pin/signal declarations for the model's I/O, and a clock waveform. The testbench code is event driven: the engineer writes HDL statements to implement the (testbench-generated) reset-signal, to model interface transactions (such as a host-bus read/write), and to monitor the DUT's output. An HDL simulator the program that executes the testbench maintains the simulator clock, which is the master reference for all events in the testbench simulation. Events occur only at the instants dictated by the testbench HDL (such as a reset-toggle coded into the testbench), or in reaction (by the model) to stimulus and triggering events. Modern HDL simulators have full-featured graphical user interfaces, complete with a suite of debug tools. These allow the user to stop and restart the simulation at any time, insert simulator breakpoints (independent of the HDL code), and monitor or modify any element in the HDL model hierarchy. Modern simulators can also link the HDL environment to user-compiled libraries, through a defined PLI/VHPI interface. Linking is system-dependent (Win32/Linux/SPARC), as the HDL simulator and user libraries are compiled and linked outside the HDL environment.

### 3.4 high Level Synthesis

In their level of abstraction, HDLs have been compared to assembly languages. There are moves to raise the abstraction level of the design in order to reduce the complexity of programming in HDLs, creating a sub-field called *high-level synthesis*. Companies such as Cadence, Synopsys and Agility Design Solutions are promoting SystemC as a way to combine high level languages with concurrency models to allow faster design cycles for FPGAs than is possible using traditional HDLs. Approaches based on standard C or C++ (with libraries or other extensions allowing parallel programming) are found in the Catapult C tools from Mentor Graphics, the Impulse C tools from Impulse Accelerated Technologies, and the free and open-source ROCCC 2.0 tools from Jacquard Computing Inc. Annapolis Micro Systems, Inc.'s CoreFire Design Suite and National Instruments LabVIEW FPGA provide a graphical dataflow approach to high-level design entry and languages such as SystemVerilog, SystemVHDL, and Handel-C seek to accomplish the same goal, but are aimed at making existing hardware engineers more productive, rather than making FPGAs more accessible to existing software engineers. It is also possible to design hardware modules using MATLAB and Simulink using the Mathworks tool HDL Coder or Xilinx System Generator (XSG) from Xilinx (formerly *Accel DSP*).



## 4. Hardware Verification Language

Design verification is often the most time-consuming portion of the design process, due to the disconnect between a device's functional specification, the designer's interpretation of the specification, and the imprecision of the HDL language. The majority of the initial test/debug cycle is conducted in the HDL simulator environment, as the early stage of the design is subject to frequent and major circuit changes. An HDL description can also be prototyped and tested in hardware — programmable logic devices are often used for this purpose. Hardware prototyping is comparatively more expensive than HDL simulation, but offers a real-world view of the design. Prototyping is the best way to check interfacing against other hardware devices and hardware prototypes. Even those running on slow FPGAs offer much shorter simulation times than pure HDL simulation.

#### 4.1 Design Verifications with HDL

Historically, design verification was a laborious, repetitive loop of writing and running simulation test cases against the design under test. As chip designs have grown larger and more complex, the task of design verification has grown to the point where it now dominates the schedule of a design team. Looking for ways to improve design productivity, the electronic design automation industry developed the Property Specification Language.

In formal verification terms, a property is a factual statement about the expected or assumed behavior of another object. Ideally, for a given HDL description, a property or properties can be proven true or false using formal mathematical methods. In practical terms, many properties cannot be proven because they occupy an unbounded solution space. However, if provided a set of operating assumptions or constraints, a property checker can prove (or disprove) certain properties by narrowing the solution space.

The assertions do not model circuit activity, but capture and document the designer's intent in the HDL code. In a simulation environment, the simulator evaluates all specified assertions, reporting the location and severity of any violations. In a synthesis environment, the synthesis tool usually operates with the policy of halting synthesis upon any violation. Assertion-based verification is still in its infancy, but is expected to become an integral part of the HDL design toolset.



#### 4.2 Description of HVL

A hardware verification language, or HVL, is a programming language used to verify the designs of electronic circuits written in a hardware description language. HVLs typically include features of a high-level programming language like C++ or Java as well as features for easy bit-level manipulation similar to those found in HDLs. Many HVLs will provide constrained random stimulus generation, and functional coverage constructs to assist with complex hardware verification. [SystemVerilog](#), [OpenVera](#), e, and [SystemC](#) are the most commonly used HVLs. [SystemVerilog](#) attempts to combine HDL and HVL constructs into a single standard.

SystemC is defined and promoted by the Open SystemC Initiative (OSCI), and has been approved by the IEEE Standards Association as IEEE 1666-2005 - the SystemC Language Reference Manual (LRM). The LRM provides the definitive statement of the semantics of SystemC. OSCI also provide an open-source proof-of-concept simulator (sometimes incorrectly referred to as the reference simulator), which can be downloaded from the OSCI website. Although it was the intent of OSCI that commercial vendors and academia could create original software compliant to IEEE 1666, in practice most SystemC implementations have been at least partly based on the OSCI proof-of-concept simulator.

SystemC has semantic similarities to VHDL and Verilog, but may be said to have a syntactical overhead compared to these when used as a hardware description language. On the other hand, it offers a greater range of expression, similar to object-oriented design partitioning and template classes. Although strictly a C++ class library, SystemC is sometimes viewed as being a language in its own right. Source code can be compiled with the SystemC library (which includes a simulation kernel) to give an executable. The performance of the OSCI open-source implementation is typically less optimal than commercial VHDL/Verilog simulators when used for register transfer level simulation.

### 4.3 SYSTEM C

SystemC is a set of C++ classes and macros which provide an event-driven simulation interface in C++ (see also discrete event simulation). These facilities enable a designer to *simulate* concurrent processes, each described using plain C++ syntax. SystemC processes can communicate in a *simulated* real-time environment, using signals of all the datatypes offered by C++, some additional ones offered by the SystemC library, as well as user defined. In certain respects, SystemC deliberately mimics the hardware description languages VHDL and Verilog, but is more aptly described as a system-level modeling language.

SystemC is applied to system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis. SystemC is often associated with electronic system-level (ESL) design, and with transaction-level modeling (TLM).

## 5. System-C Language Features

Modules are the basic building blocks of a SystemC design hierarchy. A SystemC model usually consists of several modules which communicate via ports. The modules can be thought of as a building block of SystemC.

Ports allow communication from inside a module to the outside (usually to other modules) via channels.

Exports incorporate channels and allow communication from inside a module to the outside (usually to other modules).

Processes are the main computation elements. They are concurrent.

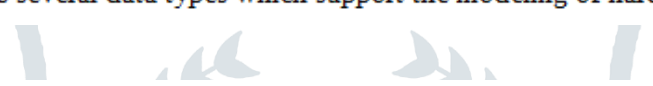
Channels are the communication elements of SystemC. They can be either simple wires or complex communication mechanisms like FIFOs or bus channels.

Elementary channels are signal (equivalent to wire), buffer, fifo, mutex, semaphore

Interfaces Ports use interfaces to communicate with channels.


Events allow synchronization between processes and must be defined during initialization.

Data Types SystemC introduces several data types which support the modeling of hardware.



SystemC version 1 included common hardware-description language features such as structural hierarchy and connectivity, clock-cycle accuracy, delta cycles, four-valued logic (0, 1, X, Z), and bus-resolution functions. From version 2 onward, the focus of SystemC has moved to communication abstraction, transaction-level modeling, and virtual-platform modeling. SystemC version 2 added abstract ports, dynamic processes, and timed event notifications.

## 6. Conclusion



So, finally we get to know that both HDL and HVL are very different from each other. Difference are highlighted between these. Major differences are HDL is used to design digital logic whereas HVL is used to Functionally verify the digital logic designed using a HDL. HDL is used for RTL design whereas HVL is used for RTL verification.

## References

- 1) Sutherland, S., & Mills, D. (2003). HDVL+=(HDL & HVL) SystemVerilog 3.1 The Hardware Description AND Verification Language. *Dodatok A Obsah CD*.
- 2) Bergeron, J. (2012). *Writing testbenches: functional verification of HDL models*. Springer Science & Business Media.
- 3) van der Schoot, H., & Yehia, A. (2015). UVM and emulation: How to get your ultimate testbench acceleration speed-up. In *Proceedings of Design and Verification Conference (DVCON), Munich, Germany*.
- 4) Jerinic, V., Langer, J., Heinkel, U., & Muller, D. (2006, March). New methods and coverage metrics for functional verification. In *Proceedings of the Design Automation & Test in Europe Conference* (Vol. 1, pp. 1-6). IEEE.



- 5) Datta, K., & Das, P. P. (2004, January). Assertion based verification using HDVL. In *17th International Conference on VLSI Design. Proceedings.* (pp. 319-325). IEEE.
- 6) van der Schoot, H., Anoop, S., Ankit, G., & Krishnamurthy, S. (2011). A Methodology for Hardware-Assisted Acceleration of OVM and UVM Testbenches.
- 7) James, P. (2004). *VERIFICATION PLANS* (Vol. 18, p. 25).
- 8) van der Schoot, H., Saha, A., Garg, A., & Suresh, K. (2011). Off To The Races With Your Accelerated SystemVerilog Testbench.
- 9) Langer, J., Heinkel, U., Jerinic, V., & Muller, D. (2006, October). Improved Coverage Driven Verification and Corner Case Analysis using Decision Diagrams. In *2006 International Symposium on Communications and Information Technologies* (pp. 1179-1184). IEEE.
- 10) Thompson, K., & Williamson, L. (2002). Hardware verification with the unified modeling language and vera. *Proceedings of Synopsys Users Group.*

