Hardware description language

From Wikipedia, the free encyclopedia

In electronics, a **hardware description language** or **HDL** is any language from a class of computer languages, specification languages, or modeling languages for formal description and design of electronic circuits, and most-commonly, digital logic. It can describe the circuit's operation, its design and organization, and tests to verify its operation by means of simulation.[*citation needed*]

HDLs are standard text-based expressions of the spatial and temporal structure and behaviour of electronic systems. Like concurrent programming languages, HDL syntax and semantics includes 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 on electric computer-aided design (CAD).

HDLs are used to write executable specifications of some piece of hardware. A simulation program, designed to implement the underlying semantics of the language statements, coupled with simulating 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 classed 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.

It is certainly possible to represent hardware semantics using traditional programming languages such as C++, although to function such programs must be augmented with extensive and unwieldy class libraries. Primarily, however, software programming languages do not include any capability for explicitly expressing time, and this is why they do not function as a hardware description language. Before the recent introduction of SystemVerilog, C++ integration with a logic simulator was one of the few ways to use OOP in hardware verification. SystemVerilog is the first major HDL to offer object orientation and garbage collection.

Using the proper subset of virtually any (hardware description or software programming) language, a program called a synthesizer (or 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.[*citation needed*] 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.

Contents

History

The first hardware description languages were ISP (Instruction Set Processor),[1] developed at Carnegie Mellon University, and KARL, developed at University of Kaiserslautern, both around 1977. ISP was, however, more like a software programming language used to describe relations between the inputs and the outputs of the design. Therefore, it could be used to simulate the design, but not to synthesize it. KARL included design calculus language features supporting VLSI chip floorplanning and structured hardware design, which was also the basis of KARL's interactive graphic sister language ABL, implemented in the early 1980s as the ABLED graphic VLSI design editor, by the telecommunication research center CSELT at Torino, Italy. In the mid 80's, a VLSI design framework was implemented around KARL and ABL by an international consortium funded by the commission of the European Union (chapter in [2]). In 1983 Data-I/O introduced ABEL. It was targeted for describing programmable logical devices and was basically used to design finite state machines.

The first modern HDL, Verilog, was introduced by Gateway Design Automation in 1985. Cadence Design Systems later acquired the rights to Verilog-XL, the HDL-simulator that would become the de-facto standard (of Verilog simulators) for the next decade. In 1987, a request from the U.S. Department of Defense led to the development of VHDL (VHSIC Hardware Description Language, where VHSIC is Very High Speed Integrated Circuit). VHDL was based on the Ada programming language. Initially, Verilog and VHDL were used to document and simulate circuit-designs already captured and described in another form (such as a schematic file.) HDL-simulation enabled engineers to work at a higher level of abstraction than simulation at the schematic-level, and thus increased design capacity from hundreds of transistors to thousands[*citation needed*].

The introduction of logic-synthesis for HDLs pushed HDLs from the background into the foreground of digital-design. Synthesis tools compiled HDL-source files (written in a constrained format called RTL) into a manufacturable gate/transistor-level netlist description. Writing synthesizeable RTL files required practice and discipline on the part of the designer; compared to a traditional schematic-layout, synthesized-RTL netlists were almost always larger in area and slower in performance[*citation needed*]. A circuit design from a skilled engineer, using labor-intensive schematic-capture/hand-layout, would almost always outperform its logically-synthesized equivalent, but synthesis' productivity advantage soon displaced digital schematic-capture to exactly those areas that were problematic for RTL-synthesis: extremely high-speed, low-power, or asynchronous circuitry. In short, logic synthesis propelled HDL technology into a central role for digital design.

Within a few years, both VHDL and Verilog emerged as the dominant HDLs in the electronics industry, while older and less-capable HDLs gradually disappeared from use. But VHDL and Verilog share many of the same limitations: neither HDL is suitable for analog/mixed-signal circuit simulation. Neither possesses language constructs to describe recursively-generated logic structures. Specialized HDLs (such as Confluence) were introduced with the explicit goal of fixing a specific Verilog/VHDL limitation, though none were ever intended to replace VHDL/Verilog.

Over the years, a lot of effort has gone into improving HDLs. The latest iteration of Verilog, formally known as IEEE 1800-2005 SystemVerilog, introduces many new features (classes, random variables, and properties/assertions) to address the growing need for better testbench randomization, design hierarchy, and reuse. A future revision of VHDL is also in development, and is expected to match SystemVerilog's improvements.

Design using HDL

Efficiency gains realized using HDL means 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, this 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.

Simulating and debugging HDL code

Main article: Logic simulation

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.

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[*citation needed*] 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.

Design Verification with HDLs

Main article: Functional verification

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 EDA 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) more properties, over the narrowed 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.

HDL and programming languages

A HDL is analogous to a software programming language, but with major differences. Many programming languages are inherently procedural (single-threaded), with limited syntactical and semantic support to handle concurrency. HDLs, on the other hand, resemble concurrent programming languages in their ability to model multiple parallel processes (such as flipflops, adders, etc.) that automatically execute independently of one another. Any change to the process's input automatically triggers an update in the simulator's process stack. Both programming languages and HDLs are processed by a compiler (usually called a synthesizer in the HDL case), but with different goals. For HDLs, 'compiler' refers to synthesis, a process of transforming the HDL code listing into a physically realizable gate netlist. The netlist output can take any of many forms: a "simulation" netlist with gate-delay information, a "handoff" netlist for post-synthesis place and route, or a generic industry-standard EDIF format (for subsequent conversion to a JEDEC-format file).

On the other hand, a software compiler converts the source-code listing into a microprocessor-specific object-code, for execution on the target microprocessor. As HDLs and programming languages borrow concepts and features from each other, the boundary between them is becoming less distinct. However, pure HDLs are unsuitable for general purpose software application development, just as general-purpose programming languages are undesirable for modeling hardware. Yet as electronic systems grow increasingly complex, and reconfigurable systems become increasingly mainstream, there is

growing desire in the industry for a single language that can perform some tasks of both hardware design and software programming. SystemC is an example of such—embedded system hardware can be modeled as non-detailed architectural blocks (blackboxes with modeled signal inputs and output drivers). The target application is written in C/C++, and natively compiled for the host-development system (as opposed to targeting the embedded CPU, which requires host-simulation of the embedded CPU). The high level of abstraction of SystemC models is well suited to early architecture exploration, as architectural modifications can be easily evaluated with little concern for signal-level implementation issues. However, the threading model used in SystemC and its reliance on shared memory mean that it does not handle parallel execution or lower level models well.

In an attempt to reduce the complexity of designing in HDLs, which have been compared to the equivalent of assembly languages, there are moves to raise the abstraction level of the design. 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 (http://jacquardcomputing.com/roccc/) tools from Jacquard Computing Inc (http://jacquardcomputing.com) . Annapolis Micro Systems, Inc.'s CoreFire Design Suite and National Instruments LabVIEW FPGA provide a graphical dataflow approach to high-level design entry. Languages such as SystemVerilog, SystemVHDL, and Handel-C seek to accomplish the same goal, but are aimed at making existing hardware engineers more productive versus making FPGAs more accessible to existing software engineers. There is more information on C to HDL and Flow to HDL in their respective articles.

Languages

Analogue circuit design

Abbreviation	**Name**	**Use**
AHDL	Analog Hardware Descriptive Language (HDL)	an open analog hardware description language
SpectreHDL	SpectreHDL	a proprietary analogue hardware description language
Verilog-AMS	Verilog for Analog and Mixed-Signal	an open standard extending Verilog for analog and mixed analog/digital simulation
HDL-A™	HDL-A	a proprietary analogue hardware description language

Digital circuit design

The two most widely-used and well-supported HDL varieties used in industry are Verilog and VHDL.

Abbreviation	**Name**	**Notice**
ABEL	Advanced Boolean Expression Language	
AHDL	Altera HDL	a proprietary language from Altera
AHPL	A Hardware Programing language	
Bluespec		high-level HDL originally based on Haskell, now with a SystemVerilog syntax
C-to-Verilog (http://www.c-to-verilog.com)		Converter from C to Verilog
Confluence (http://www.tomahawkins.org)		a functional HDL; has been discontinued
CoWareC		a C-based HDL by CoWare. Now discontinued in favor of SystemC
CUPL	Universal Compiler for Programmable Logic [3]	a proprietary language from Logical Devices, Inc.
ELLA		no longer in common use
ESys.net		.net framework written in C#
Handel-C		a C-like design language
HJJ	Hardware Join Java (http://rcl.unisa.edu.au/h_join_java.html)	based on Join Java
HML (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=820756)		based on SML
Hydra (http://www.dcs.gla.ac.uk/~jtod/Hydra/)		based on Haskell
Impulse C	another C-like HDL	
ParC (http://parallel.cc)	Parallel C++	C++ extended with HDL style threading and

		communication for task-parallel programming
JHDL		based on Java
Lava (http://raintown.org/lava/)		based on Haskell
Lola		a simple language used for teaching
M		A HDL from Mentor Graphics
MyHDL		based on Python
PALASM		for Programmable Array Logic (PAL) devices
ROCCC 2.0 (http://jacquardcomputing.com/roccc/)	Riverside Optimizing Compiler for Configurable Computing	Free and open-source C to HDL tool
RHDL (http://rhdl.rubyforge.org)		based on the Ruby programming language
Ruby (hardware description language)		
SystemC		a standardized class of C++ libraries for high-level behavioral and transaction modeling of digital hardware at a high level of abstraction, i.e. system-level
SystemVerilog		a superset of Verilog, with enhancements to address system-level design and verification
SystemTCL		SDL based on Tcl.

THDL++ (http://visualhdl.sysprogs.org/)	Templated HDL inspired by C++	An extension of VHDL with inheritance, advanced templates and policy classes
Verilog		most widely-used and well-supported HDL
VHDL	VHSIC HDL	most widely-used and well-supported HDL

Printed Circuit Board design

Several projects exist for defining printed circuit board connectivity using language based, textual, entry methods.

Abbreviation	**Name**	**Notice**
PHDL	PCB HDL	a free and open source HDL for defining printed circuit board connectivity

See also

- Specification language
- Modeling language
- Hardware Verification Language
- SystemC
- SystemVerilog
- Property Specification Language
- OpenVera
- Bluespec

References

1. ^ Barbacci, M. "The ISPS Computer Description Language," Carnegie-Mellon Univ., Dept. of Computer Science, 1977
2. ^ J. Mermet (editor): Fundamentals and Standards in Hardware Description Languages (Springer Verlag, 1993)
3. ^ Eurich, J.P. and Roth, G. (1990): "EDIF grows up". IEEE Spectrum, Vol. 27, Issue 11, pp. 68 - 72.

External links

- Verilog-AMS Technical Subcommittee (http://www.eda.org/verilog-ams/)
- HCT (http://hct.sourceforge.net/) - The HDL Complexity tool, used to determine design complexity.