HDL basic training: top-down chip design using Verilog and VHDL

Douglas J Smith, VeriBest

*Increasingly complex ASIC and FPGA chips require you to shift from schematic-based design to design based on Verilog or VHDL. Understanding the basics of these languages and how you use them for designing ASICs and FPGAs will help you go with the HDL flow.*

In recent years, designers have increasingly adopted top-down design methodologies that lead them away from logic- and transistor-level design to abstract programming. The introduction of industry-standard HDLs and the existence of commercially available synthesis tools have helped establish this revolutionary design methodology. The advantages of this methodology are clear, causing a change in IC engineering-design methods.

The advantages of HDL-based design include:

- increased productivity yields, shorter development cycles, more product features, and reduced time to market;
- reduced NRE costs;
- the possibility of design reuse, which results in increased flexibility for design changes;
- faster explanation of alternative architectures;
- faster explanation of alternative technology libraries.

HDL-based design allows synthesis to rapidly sweep the design space for area, timing, and, possibly, power, and to automatically generate testable circuits. It also results in better and easier design auditing and verification.

VHDL and Verilog are industry-standard HDLs for chip design. Both languages are used worldwide and have been adopted by the IEEE. The language versions described here comply with IEEE 1076 '93 for VHDL and IEEE 1364 for Verilog.
Standard off-the-shelf ICs have fixed functions defined by chip manufacturers. On the other hand, you define the functions for both ASICs and FPGAs for each application. Unlike an FPGA, an ASIC requires a final manufacturing process to customize its operation.

A chip vendor partially manufactures an ASIC in a generic form. This initial manufacturing is the most complex, time-consuming, and expensive part of the chip-fabrication process. The results are silicon chips that have an array of unconnected transistors. When you have a design to implement in the ASIC, the vendor completes the final process of connecting the transistors. An ASIC vendor can usually complete this process in a few weeks. These weeks are the "turnaround time." There are two categories of ASIC devices: gate arrays and standard cells.

Gate arrays come with either a channeled or channelless architecture. Channeled gate arrays have either single or double rows of basic cells across the silicon. A basic cell consists of a number of transistors. You can use the channels between the rows of cells for interconnecting the basic cells during final customization. Silicon vendors manufacture a channelless gate array with a "sea" of basic cells across the silicon and without dedicated channels for interconnections. Gate arrays contain from a few thousand to several hundred thousand equivalent gates. Because of limited routing space on channeled gate arrays, you typically can use only 70 to 90% of the total number of available gates for a design.

The library of cells provided by a gate-array vendor contains primitive logic gates, registers, and hard and soft macros. Hard and soft macros are usually of MSI and LSI complexity, including cells such as multiplexers, comparators, and counters. Manufacturers define hard macros in terms of cell primitives. In comparison, you characterize soft macros, for example, by specifying the width of a counter.

Standard-cell devices are not configured with a basic-cell architecture, and no components are prefabricated on the silicon chip. The manufacturer creates custom masks for every stage of the device’s fabrication, which means standard-cell chips use silicon much more efficiently than do gate arrays. Manufacturers supply hard- and soft-macro libraries that contain elements of LSI and VLSI complexity, such as controllers, ALUs, and mPs. Additionally, soft-macro libraries contain RAM functions that you cannot efficiently implement in gate-array devices. On the other hand, you can more efficiently implement ROM functions in cell primitives.

An FPGA is a completely manufactured, design-independent device. Each FPGA vendor manufactures devices with a proprietary architecture. However, the architecture includes a number of programmable logic blocks connected to programmable switching matrices. To configure a device for a particular functional operation, you program these switching matrices to route signals between the individual logic blocks.

Choosing an ASIC or FPGA depends on many factors. The NRE costs involved with customizing an ASIC currently range from $20,000 to more than $100,000. However, after this initial outlay, the unit cost for production devices might be only about $10. This cost is much lower than the production costs of FPGA devices, typically $150 to $250 per device. The advantage of FPGAs is that they are quick and easy to program, or customize. Also, FPGAs allow pc-board CAD layout to begin while internal FPGA design is being completed. This procedure lets you perform early hardware- and software-integration testing. If system testing fails, you can modify the design and immediately program another FPGA at a relatively low cost. For these reasons, designers often initially target hardware designs to FPGA devices for system testing and small production runs. The design is then retargeted to an ASIC for larger scale production. You must consider design trade-offs when retargeting FPGAs to ASICs. For example, a long hold time may never appear in an ASIC because of its improved speed compared with an FPGA.
In an ideal world, a true top-down, system-level design methodology would enable complete systems to be described at an abstract level using an HDL and automated tools, such as partitioners and synthesizers, for implementation. This approach drives the abstract-level description to implementation on pc boards or multichip modules, which can contain standard ICs, ASICs, FPGAs, and PLD and full-custom ICs. No one has yet realized this design methodology. However, EDA tools are constantly being improved. Designers must take on new roles and learn new skills, as they are now spending more time designing HDL models and considering different architectures and system-test and testability issues. Designers using an HDL methodology spend practically no time doing gate-level design.

Technological advancements over the past six years have seen a tenfold increase in the number of gates that an ASIC can contain; 100,000 gates are now common. These advancements have increased the complexity of standard ICs and ASICs, resulting in the "system-on-a-chip" concept. A top-down design methodology is the only practical option for designing such chips. ASIC or FPGA designs in hardware-development projects must follow critical development schedules. Traditionally, you create these designs by entering them as circuit diagrams using a schematic entry tool. In rare cases, such as to reduce costs, this method may still be viable for small devices, such as PLDs. Provided the budget is available for simulation and synthesis tools, a top-down design approach using an HDL is, by far, the best design philosophy. There are many advantages of adopting a top-down design methodology. Imagine using schematics to design a 100,000-gate ASIC. A small design change can result in time-consuming changes to the schematics. Using an HDL to develop electronic hardware is similar to doing a software-development project using a high-level programming language, such as C.

The levels of hierarchical refinement of electronic hardware in a top-down design process (Figure 1) show how synthesis is the key link in the process. A top-down design methodology takes the HDL model of hardware, written at a high level of behavioral abstraction (system or algorithmic), down through intermediate levels, to a low (gate or transistor) level (Figure 2). The term "behavioral" means the behavior of intended hardware, independent of the level of abstraction at which you model it. A design represented at the gate level still represents the behavior of hardware intent. As you translate hardware models to progressively lower levels, they become more complex and contain more structural detail. The benefit of modeling hardware at higher levels of behavioral abstraction is that you are not overwhelmed with large amounts of unnecessary detail. This modeling methodology reduces design complexity. You ignore hardware structure when modeling hardware at the two highest behavioral levels. However, when modeling hardware at the RTL, it is essential to keep the hardware intent in mind at all times. Figure 3 shows how different behavioral levels of abstraction overlap the design domains of pure abstraction, structural decomposition, and physical implementation.

Figure 4 shows a typical ASIC design flow using simulation and RTL synthesis. You use the same test vectors to verify the RTL and synthesized netlist-level models. The netlist level corresponds to the gate level but may also include larger macro cells or even highly complex megacells. By comparing each level's
simulation results, you automate netlist-level testing.

**HDLs**

An HDL is a software-programming language used to model the intended operation of a piece of hardware. There are two aspects to the description of hardware facilitated by an HDL: true abstract behavioral modeling and hardware structure modeling. An HDL is declarative to facilitate the abstract description of hardware behavior for specification. Structural and design aspects of hardware intent do not prejudice this behavior. You can model hardware structure in an HDL irrespective of the design's behavior. You can also model and represent hardware behavior at various levels of abstraction during design. Higher level models describe the operation of hardware abstractly and lower level models include more detail, such as inferred hardware structure.

**VHDL vs Verilog**

Capability. You can model hardware structure equally effectively in VHDL and Verilog. To model abstract hardware in Verilog as efficiently as you can in VHDL, you need to use the programming-language interface (PLI). Designers base the choice of which HDL to use not solely on technical capability, but also on personal preferences, EDA-tool availability and commercial, business, and marketing issues. The modeling constructs of VHDL and Verilog cover slightly different spectra across behavioral abstraction levels (Figure 5).

Compilation. In VHDL, you can separately compile multiple design units (entity-architecture pairs) that are in the same system file. However, it is good design practice to keep each design unit in its own system file. In contrast, Verilog is still rooted in its native interpretive mode. Compilation is a way of speeding simulation without changing the original nature of the language. As a result, you must be careful in writing both the compilation order of code for a single file and the compilation order for multiple files. Simulation results can change by simply changing the order of compilation.

Data types. With VHDL, you can use many language or user-defined data types. This approach may require dedicated conversion functions to convert objects from one type to another. You should carefully consider the choice of which data types to use, especially with enumerated (abstract) data types. A good choice makes models easier to write and clearer to read and avoids unnecessary conversion functions that can clutter the code. You may prefer VHDL because it allows the use of multiple-language or user-defined data types.

Compared with VHDL, Verilog data types are simple, easy to use, and geared toward modeling hardware structure as opposed to abstract hardware modeling. Unlike VHDL, the Verilog language—not the user—defines all data types used in a Verilog model. There are net-data types, such as wire, and a register-data type called reg. A model with a signal whose type is one of the net-data types has a corresponding electrical wire in the implied modeled circuit. You can update objects of type reg depending upon the procedural flow of constructs that surround the objects. You may prefer Verilog because of the simplicity of its data types.

Design reusability. In VHDL, you can place procedures and functions in a package so that they are available for use by any other design unit that wishes to use them. There is no concept of packages in Verilog. You must define the functions and procedures that a model uses within the module statement itself. To make functions and procedures generally accessible from different module statements, you must place them in a separate system file and include them with the `include (pronounced tick include) compiler directive.
Ease of learning. If you have no knowledge of either language, Verilog is probably easier to grasp and understand. This statement assumes the exclusion of the Verilog compiler directive language for simulation and of the PLI. If you include these two, consider them additional languages you need to learn. Also, VHDL may seem less intuitive at first for two reasons. First, it is very strongly typed, a feature that makes it robust and powerful for an advanced user after a longer learning phase. Second, there are many ways to model the same circuit, especially one with large hierarchical structures.

Forward and back annotation. The Standard Delay Format (SDF) is a spin-off of Verilog. You use this general-purpose format to define the timing delays in a circuit. The format provides a bidirectional link between chip-layout tools and either synthesis or simulation tools, which provide more accurate timing representations. The SDF is now an industry standard.

High-level constructs. There are more constructs and features for high-level modeling in VHDL than there are in Verilog. You use abstract data types along with the following statements: model reuse; configuration statements for configuring design structure; generate statements for replicating structure; and generic statements for generic models you can individually characterize, such as bit width. These language statements are useful in synthesizable models. Verilog can parameterize models by overloading parameter constants but has no equivalent to these high-level VHDL modeling statements.

Language extensions. The use of language extensions makes a model nonstandard and, most likely, not portable to other design tools. However, extensions are sometimes necessary to achieve desired results.

VHDL has an attribute called `foreign that allows you to model architectures and subprograms in another language. For Verilog, the PLI serves as an interface mechanism between Verilog models and either models in another language or Verilog software tools. For example, you can specify user-defined tasks or functions in C and then call them from a Verilog source description. Using such tasks or functions makes a Verilog model nonstandard, and other Verilog software tools may be unable to use it. Using user-defined tasks is not a good idea.

Libraries. A library in VHDL is a storage area in the host environment for compiled entities, architectures, packages, and configurations. You can use libraries for managing multiple design projects. There is no concept of a library in Verilog, because of its origin as an interpretive language.

Low-level constructs. VHDL has simple two-input logical operators built into the language. These operators include NOT, AND, OR, NAND, NOR, XOR, and XNOR. You must specify any timing separately using the after clause. You need to use separate constructs defined in the IEEE 1076.4 VHDL Initiative toward ASIC Libraries (VITAL) standard to define the cell primitives of ASIC and FPGA libraries.

Verilog was originally developed with gate-level modeling in mind and has good constructs for modeling at this level for ASIC and FPGA library-cell primitives. Examples include user-defined primitives, truth tables, and the specify block for specifying timing delays across a module.

Managing large designs. VHDL has configuration, generate, and package statements, together with the generic clause, that help manage large design structures. There are no such statements in Verilog.

Operators. Most operators are the same in both languages. Verilog does have useful unary-reduction operators that are not predefined in VHDL. You can use a loop statement in VHDL to
perform the same operation as a Verilog unary-reduction operator. VHDL has the mod operator, which Verilog does not have.

Parameterizable models. In VHDL, a specific bit-width model can be instantiated from a generic n-bit model using the generic clause. The generic model doesn't synthesize until it is instantiated and the user specifies a value for the generic. In Verilog, you can instantiate a specific-width model from a generic n-bit model using overloaded parameter values. The generic model must have a defined default parameter. This requirement means two things. In the absence of a specified overloaded value, the model still synthesizes but uses default parameter settings. Also, the model does not need to be instantiated with a specified overloaded parameter value before it synthesizes.

Procedures and tasks. VHDL allows concurrent procedure calls; Verilog does not allow concurrent task calls.

Readability. This feature is a matter more of coding style and experience than of language features. VHDL is detailed and verbose with roots based in Ada. Verilog is more like C, with constructs based approximately 50% on C and 50% on Ada. For this reason, a C programmer may prefer Verilog over VHDL. A programmer of both C and Ada may find the construct mix somewhat confusing at first. Whichever HDL you use, when writing or reading an HDL model to be synthesized, it is important to think about hardware intent.

Structural replication. The generate statement in VHDL replicates a number of instances of the same design unit or some subpart of a design and connects them appropriately. There is no generate statement equivalent in Verilog.

Test harnesses. Designers typically spend approximately 50% of their time writing synthesizable models and the other 50% writing test harnesses to verify the synthesizable models. Test harnesses are not restricted to the synthesizable subset, so you are free to use the full potential of the language. VHDL has generic and configuration statements, not found in Verilog, that are useful in test harnesses.

Verboseness. Because VHDL is a strongly typed language, you need to precisely code models using defined and matching data types. This feature can be an advantage or a disadvantage. However, it does mean VHDL models are often more verbose and the code is often longer than its Verilog equivalent.

In Verilog, you can assign signals representing objects of different word widths to each other. Verilog automatically pads out the signal representing fewer bits to the signal with more bits, independent of whether it is the assigned signal. The language automatically optimizes (deletes) unused bits during synthesis. This approach has the advantage of not needing to model quite so explicitly as in VHDL, but it does mean a Verilog analyzer cannot always identify unintended modeling errors.

Author's biography
Douglas Smith is a member of the technical staff at VeriBest (Huntsville, AL). He has a BSc in electrical and electronic engineering from Bath University (Bath, UK) and has worked for a number of English companies doing digital design, including mP-based circuit boards and IC development. He has experience designing PLDs and FPGAs, gate arrays, and standard cell-based chips, and his EDA background includes synthesis-product-application engineering and product-marketing management. Smith is the author of HDL Chip Design—A Practical Guide for Designing, Synthesizing, and Simulating ASICs and FPGAs Using VHDL and Verilog (http://fly.hiwaay.net/~asmith). To order, phone (800) 311-3753. You can reach Smith via email at djsmith@veribest.com.