

Synergy Between VHDL & Verilog

Peet James
Senior Development Engineer
Electrical Process Group
StorageTek MS 4215
2270 South 88th Street
Louisville, CO 80028-4215
(303) 661-7669
peetj@hp7001.ecae.stortek.com

ABSTRACT

The hardware description language (HDL) wars tried to make us all pick a side and then come out fighting. Time, the true test, has shown that both VHDL and Verilog are here to stay. Both languages are being used successfully in real world designs today—sometimes even together. Where does each language shine? Where do their limitations start to show? What is it like to use them together? These and other questions are briefly answered from the perspective of a designer who has used VHDL and Verilog both separately and together.

DISCLAIMER

It is not the purpose of this paper to definitively state, once and for all, which language is better, and which to use. Rather, its purpose is to share experiences of using VHDL and Verilog, alone and together, in order to provide the reader with helpful information to assist in making informed HDL decisions. The debate over which language to use often boils down to a religious argument. It is not the point of this paper to perpetuate such argumentative discussions, but to preach co-existence and the bi-lingual use of either language depending on which is best for a given project. Both languages have strong followings and strong support and both will be around for a long time. This paper will hopefully help the reader distinguish when and where to use each language to its fullest potential.

BACKGROUND

The information presented here is based primarily on experience with a project involving the design of semi-custom ASICs containing DSP logic. The project started with a standard schematic capture-simulation-layout methodology with the normal “get-it-done-yesterday” schedule. As always, the engineers ran into some major problems and management decided to call in newly hired engineers with previous HDL and top down design experience to correct the problems and make the schedule. Full freedom was allowed in HDL selection, methodology, and tool selection. Thus both Verilog and VHDL were used on various designs and at various stages of development. What follows are some conclusions based on the outcome of this particular project, its follow-ons, and other recent work.

1. FORMAT

Experiences and findings gained while learning and using both VHDL and Verilog will be divided into five categories for comparison: the languages themselves, the language extensions, the simulators of each language, tool availability for each language, and synergy between the VHDL and Verilog.

2. LANGUAGE COMPARISON

BASICS: Both VHDL and Verilog are languages used to textually represent digital hardware. Digital logic is usually represented in three forms: behavioral, register transfer

level (RTL) or functional, and a gate or structural level. Higher forms of behavioral representations are often called system or architectural levels.

The use of HDLs in our project was primarily at the RTL and structural levels, but subsequent work has been done at the higher levels. We found that each language can represent logic at all levels. In these basic ways the languages were found to be quite similar. They both have input definition statements much like a symbol in conventional schematic entry. VHDL has the ENTITY statement, Verilog, the MODULE statement. Both languages allow the designer to define signals, wires, and registers. Each language has a host of software-type constructs (IF, CASE, etc.) that allow the definition of digital logical functions. Thus, there are lots of similarities in these basic "get-the-job-done" features, but as with virtually any new programming language, certain unique semantics must be learned.

HISTORY-VERILOG: Verilog came from the world of commercial Electrical Design Automation (EDA) tools. It was the first widely excepted HDL and many rather large pieces of hardware have been successfully designed using it. Verilog was developed by circuit designers whose purpose was to represent their logic in a clear and tangible way. It directly replaced schematic capture as a direct method for entering circuit designs. Gateway was the company that created and started selling the original language and simulator back in 1985. The use and development of Verilog was incremental. Much of the early work was more at the gate and structural levels. This is why Verilog has the very low level user-defined primitive (UDP) constructs that are great for developing libraries and other low level parts. Verilog was not a standard at first. It was proprietary and governed only by its owners (Gateway and then Cadence). The language was released into the public domain about 4 years ago and is now in the process of being standardized by the IEEE (1364 - 50% done as of Sept. 1994). Because of this, many vendors are developing Verilog simulators and tools, whereas previously only Cadence offered Verilog products.

HISTORY-VHDL: The development of VHDL (VHSIC Hardware Description Language) was funded by the Department of Defense in the late 70's to early 80's. The first proposal was in 1981, and the first release as an IEEE standard (1076) was in December of 1987. It was developed by a committee, and its preliminary purpose was to fully document digital designs. As it was being developed, the goals of the language grew to include a wider representation of digital logic (mostly up into higher levels of abstraction). Spearheaded by the defense industry contractors and required by the government, VHDL was to be a language that would clearly document what each party expected when partnering on a digital design system. Consequently VHDL is strongly typed—almost a legal document—and sometimes sacrifices readability to be more formal and precise. VHDL has never been proprietary. When it was released in 1987 it was a language without any tools. But because of its standardization and non-proprietary nature it has been fairly open for many vendors to produce tools for it. A recent revision of the VHDL specification was completed in 1993 and a new VHDL timing specification called VITAL (1076.4) is being finalized.

LEARNING: Which was easier to learn? Our engineers found Verilog (the language only—no simulator or simulator user interfaces) easier to understand and grasp. It was more intuitive. This is most likely because Verilog was developed by circuit designers. VHDL, being so strongly typed, seemed less intuitive, especially at the start up phase. The same features that make it so robust and powerful (which is a benefit to the advanced user) are a liability at the learning phase. Verilog was grasped faster and with fewer questions than VHDL. But as engineers became HDL literate, desiring higher and more robust constructs, some limitations of Verilog started to show.

As our designs grew into larger amounts of HDL that represented the entire system, our designers started to struggle with maintaining and working with this large data structure. They usually found ways to do a certain task or to represent a desired function in either of the two languages. It usually involved some hoop-jumping and PLI (programming lan-

guage interface) tricks in Verilog, while VHDL typically offered (sometimes too) many ways to do the same thing within the language constructs itself. The CONFIGURATION and PACKAGE statements in VHDL are good examples of constructs that helped manage the large HDL structure. Still, VHDL often offered so many choices that it was confusing to the new HDL user.

HIERARCHY: Our engineers found that VHDL had many higher features and functions. Some of these higher features took a long time to learn and even longer to learn to use properly, but were found to be quite useful (GENERIC and GENERATE statements are good examples). These features were primarily constructs useful as the design got bigger, the abstraction level moved up (behavioral & architectural), and when targeting models for multiple uses and re-use.

VHDL does a better job at higher design abstractions, and Verilog does a better job at the lower levels (see Figure 1).

We primarily learned through our experience that VHDL consists of a set of language primitives. From these primitives our VHDL coders could construct most necessary hardware primitives or desired descriptions. Verilog, on the other hand, consists of a set of fixed hardware primitives. This set allowed coders to describe most any RTL or gate level hardware description.

GATES/LIBRARIES: Our research showed that Verilog had quite a following in the gate and library arena. Over 170 foundries support and develop Verilog models for their libraries. More important was the fact that our foundry had very robust Verilog models that were just getting better all the time. Why all this support? It is due to Verilog's robust low level features and that Verilog has been used longer in the design community. Also when the language was put in the public domain, it already had many of the library issues ironed out. Verilog seems to be the standard in the library world. Many products have been successfully designed using Verilog libraries.

Back annotation is part of this library success story. The standard delay format (SDF) is basically a Verilog library side effect that has become a standard in itself. We used SDF and back annotation to specify more accurate timing representation with a great deal of success.

We found VHDL was having real difficulty setting library level standards. Verilog basically had its library standards set when it was proprietary and had only one simulator (Verilog-XL by Cadence). VHDL, although an IEEE standard, has had many vendors developing tools for it. Each vendor developed its own hardware primitives and library standards. No one vendors interruption has become dominant. Thus a separate library and timing standard is being developed. VITAL (1076.4) is the IEEE standards organization for VHDL library and timing. It is just now being used to validate real libraries. VHDL still has a long way to go to catch up and meet the level of interchangeability and acceptance that Ver-

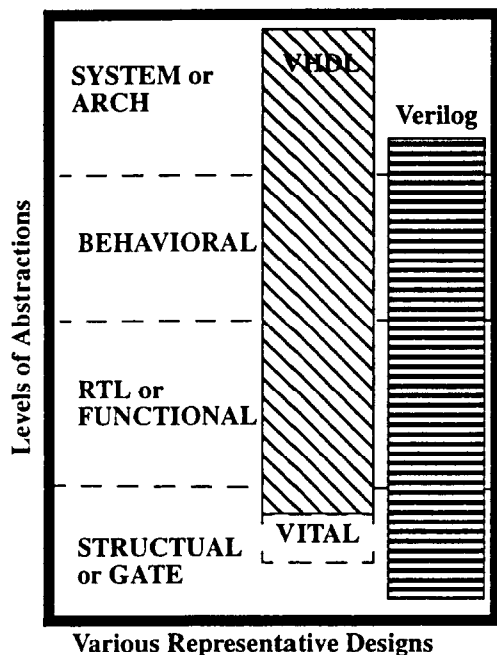


Figure 1. Hierarchical Comparison of HDLs

Verilog was found to have more features and functions (UDP, specify, etc.) at the lower levels (gates & libraries). Our Verilog libraries were much more robust than their VHDL counterparts. Overall our experience was that

ilog has in its library models. In fact, VITAL is leveraging on much of what Verilog has already developed (SDF is a good example).

3. LANGUAGE EXTENSIONS COMPARISON

VHDL does not really have any language extensions. All constructs are considered a part of the primary language. Verilog has the PLI (programming language interface) and other compiler directives to perform many of its functions. The PLI can be viewed as set of directives that take the form of either embedded comments or commands fed directly to the simulator. Our designers wrote scripts that performed PLI tasks for them automatically. The PLI is where a substantial number of repairs and enhancements are made to existing Verilog tools.

In learning to use and debug Verilog, one must learn to use the PLI. To some designers this was like having to learn a second language and a haphazard way to repair language limitations. Other designers loved hacking with the PLI and found it to be a window into the language and simulator that gave them more control.

VHDL has no PLI. It assumes the designer will use available language constructs to perform any tasks that need to be done. Because of this difference in language structure, we went about performing testbenching and simulation input and output a little differently with each. VHDL provided internal constructs that the designer put in the model or testbench code itself to perform the tasks. In Verilog testbenches, embedded commands and PLI scripts were used.

Basically, Verilog's PLI is a collection of user routines that perform various functions. For example the "\$DUMPVAR" command or the "\$MONITOR" command are PLI tools that export outputs from models during simulation. VHDL has TEXTIO and other constructs that allow the designer to output simulation data by writing various simple routines. The difference is that all the VHDL is included in the code, while the Verilog solution is a combination of internal directives (command) and compiler commands (which can be done

batch or interactively on the Verilog simulator command line).

Remembering that VHDL came from a documentation background, one can see why it chose the all in one file, all part of the language methodology. It is clean, concise, and powerful. The Verilog way seems a little less clean, but there are several ways for repairs to be made (+ compiler directives or command line commands) both by the user and the tool developer.

In learning Verilog we found that it was really a sum of Verilog the language, Verilog the compiler directive language, and the PLI. Again, some of our designers thought learning the PLI and the compiler directives was like having to learn several languages. So while the Verilog language itself might be easier to learn for the beginner, the added complexity of learning the ins and outs of the PLI might actually make it harder.

The PLI also has its own versioning, separate from the versioning of the Verilog LRM (language reference manual). Only some of the PLI constructs will be included in the IEEE (1364) standard when it comes out. We have found that a lot of our scripts (with PLI directives and command) do not transfer easily from Verilog simulator to simulator, and often times between new versions of the same simulator.

We found that VHDL transfers more easily from simulator to simulator because it has the majority of its constructs included in the testbucket code files themselves. In actual use though, many VHDL simulators have internal commands to do many PLI type functions. If a designer uses them in scripts, the same transfer problems that plague Verilog will surface. We also found some transfer problems between various VHDL tools because of language interpretation differences. Over all we found that if we used VHDL itself in our testbucket code to perform all the necessary tasks, as the language intended, we benefited with code portability and more uniform documentation later on.

In saving, archiving, and documenting designs, we found that it was very important in Verilog for the designer to archive much

more than the Verilog code itself (libraries, models, testcases). The compile command and any other run time PLI directive, logfiles, scripts, or notes also needed to be saved. In VHDL, we learned to write code with most of the simulation data and directives contained in the top level test bucket. This simplified archiving to the just code itself.

For the advanced user either language can accomplish most desired tasks, but VHDL often does it cleaner. Some advanced Verilog users who really know how to manipulate the PLI reap many benefits from tricks and workarounds that they have come up with. The problem is that often times these tricks and benefits go away when changing tools or upgrading versions of software.

4. SIMULATOR COMPARISONS

SIMULATION LIFE: Verilog and VHDL are not just languages—they are inputs and outputs to a host of software tools (simulators, synthesis, etc.). The primary tool is the HDL simulator. Figure 2 shows that as the typical simulation design cycle unfolds the number of errors found decreases and the length of simulation time needed to find an error increases. At the beginning of the design cycle, during

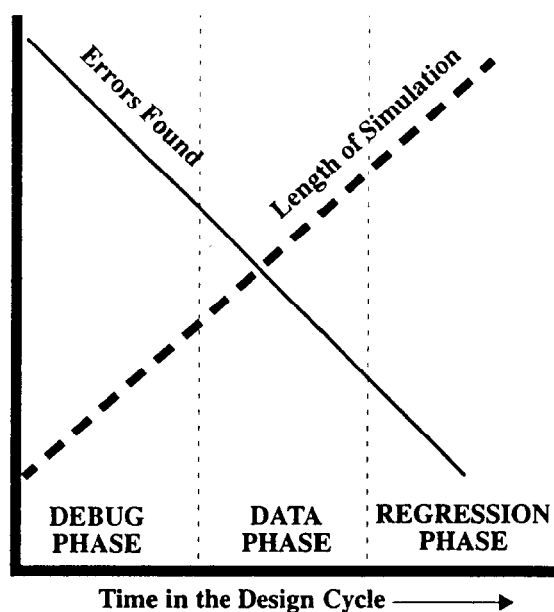


Figure 2. LIFE of SIMULATION Chart

model development, much of the designer's time is spent debugging models and testcases. During this phase it is desirable to have quick turn-around in finding bugs, correcting, and re-running a simulation. Fast turn-around time is best accomplished with an interactive environment using a simulator with a very fast and friendly user interface (UI). As the models and the testcases get fleshed out, the simulation runs grow longer. More stimulus is added to the testcases. New testcases built from slightly modified existing testcases are frequently used. More and more data, and longer and longer simulation runs are needed to find errors.

During the later phases of the design cycle the simulation model typically grows to include more and more of the system. It is desirable to have a fast simulator engine and to run simulations batch during this time. The batch testcases typically get more complicated with random data and self testing features in them to stop the simulation if an error is found. The UI is used only to recreate and rerun the error interactively once found.

VERILOG SIMULATION: After using various Verilog simulators during the course of our design we learned that the language and the simulator seemed to have developed on two fronts: the simulator engines, and the PLI user interfaces (UI). Simulator engine work was centered around speed increase and minimizing memory use. And in these areas we have seen much improvement. Verilog simulators are fast. We found them faster than our VHDL simulators when comparing at equivalent levels (benchmarks are hard to do—this is mostly a gut feeling from our designers).

Until recently improvements in the UIs of Verilog simulators have been limited. Usually smaller vendors used the PLI to develop more robust UIs with source-level debuggers, hierarchy browsers, and wave displays. These UI tools are relatively inexpensive, but were limited by the PLI and used mostly as post-processing tools. They are not well integrated or very interactive. Even today most of the Verilog simulators on the market are just simulation engines that rely on the PLI command structure to start and monitor simulation runs. The UI is still left as a separate task.

Recently, several companies have integrated the UI with their simulation engines. These new UI integrated packages are a vast improvement, but they are still buggy and quite primitive.

The result of this lack of integration of UI and simulator was often frustration. Engineers often grumbled and complained about the Verilog UI. Most hated typing long commands with lengthy variable listings and instance names, and learning all the syntax of the PLI commands. Most times they ended up just collecting these commands in scripts and running things batch. These scripts run fine during long simulations near the end of the design cycle, but often are cumbersome during the bring up and code debug stages. The entire command line PLI process is not very user friendly or very interactive. Still, some of the accomplished Verilog users were very familiar with debugging simulation runs at the command line PLI level and prefer it.

VHDL SIMULATION: VHDL simulator engines have evolved from interpretive, C-compiled, and native-compiled simulator engine cores. Each has its design trade-offs and speed advantages. We used all three kinds and saw a lot of speed ups. Again, overall we seemed to see simulation speeds slightly slower than for Verilog simulators. We guess that this is due to the increased complexities of the VHDL language itself.

VHDL simulators have not developed with separate UI and simulator engine packages like with Verilog. All the simulators that we used and tested were a single integrated, cohesive UI and engine package. When exposed to both Verilog and VHDL simulators and their UIs, most of our schematic capture circuit design engineers chose the VHDL simulator as their favorite. Why? The User Interface (UI). VHDL's all-in-one language methodology, standardization, and non-proprietary nature has caused the simulation tools to evolve with no separation of the UI and the simulation engine. To the designer it is all one package. Because of this most VHDL simulators have gone through several iterations of UI enhancements. They have easy to use browsers, waves displays, and source level debuggers. Designers can interactively,

at the press of a button, set breakpoints, start and stop the simulation, and output a signal to a list or waveform. It is all very intuitive and easy to learn. We found that the very worst UI of the very worst VHDL simulator was far better than even the latest and greatest Verilog UI. This is due to the integration of the language extensions in VHDL from the start, and to the fact that VHDL has always been in the public domain allowing for healthy competition among vendors.

So, while Verilog, as a language, might be easier to learn for a beginning HDLer, when combined with the unfriendliness of the available Verilog simulators on the market, VHDL was easier as a total package solution to learn. The better UI in the VHDL simulator made up for the harder-to-learn language constructs of VHDL. With VHDL the questions were usually language questions, with Verilog the questions were typically PLI or simulator command line questions.

5. HDL TOOLS COMPARISON

We started out using Verilog and VHDL as a digital logic entry point to get us out of a jam. Since then, with subsequent design, re-designs, and new projects, we are using them as input to and output from a variety of top-down design tools. The following is a listing of Electrical Design Automation (EDA) tools or tool areas and how both languages are used within them.

- **ESDA TOOLS:** System Design tools are one of the hottest growth areas in the EDA world. These tools go one step beyond regular top-down HDL design methodology by generating the HDL code automatically. Although there are a lot of ESDA tools that generate or use Verilog, virtually all of them generate and use VHDL code. In fact most of the ones that we have tested were designed with VHDL code generation in mind from the start. Some even run VHDL internally. Verilog generation and use was often offered in later releases. It seems from our study that Verilog sometimes has a second class status when it comes to new releases and upgrades with these tools. We want to test those that do both to see if the VHDL code is more advanced and efficient

than the Verilog code. We would theorize that a good ESDA tool would take advantage of VHDL's higher level constructs to obtain more efficient code.

- **FPGA and PLD TOOLS:** FPGA and PLD tools have various input formats. Tools in this growing segment of the EDA market take many forms of input (schematic, proprietary HDLs, VHDL, Verilog). Historically it has been un-standardized. More and more of these tools are standardizing on VHDL and Verilog. There are many vendors whose tools accept both languages. But if one looks at a list of available tools and their inputs (See ASIC & EDA magazine, February 1994, pg. 30-43), one will see that there are tools that use VHDL and not Verilog. On the other hand, every tool that uses Verilog also uses VHDL.
- **SYSTEM LEVEL SIMULATION:** As geometry sizes continue to shrink and more of our ASIC designs achieve system proportions, the need to do full system simulation with both hardware and software represented is becoming the norm. We are just now looking at System level simulation tools to get out of our current simulation bottlenecks. These new simulators should be substantially faster than event simulators. It would seem from our experience that VHDL constructs typically would work better at this system level. VHDL offers many constructs to aid in this joint simulation environment. VHDL by its original specification was supposed to be extremely viable as a high level simulation modeling language. Verilog was designed more as a lower level/gate level product. Tools are starting to become available in this crucial area and they seem to be bi-lingual. We look to evaluate them and incorporate them into our design flow.
- **SYNTHESIS:** Our testing of and experiences with several synthesis tools has shown that a majority of them are bi-lingual. In fact we used them to translate from one language to the other at times. Our synthesis tool of choice fully supports both VHDL and Verilog (input and output). New releases seem to have mutual upgrades for both VHDL and Verilog at exactly the same time. The one place where

things seemed unbalanced for us was in the area of library support and compilation. Verilog libraries are more plentiful, more robust, and more standardized. Back annotation of SDF information is also readily available for the Verilog environment. We also had ways to generate VHDL models from Verilog ones, but could not go the other direction. This phenomena is most likely due to the fact that there has not been much call to produce Verilog models because the foundries provide them. Many foundries still do not produce VHDL models causing the need to create them from existing Verilog ones.

- **GATE LEVEL SIMULATION:** For us Verilog had the fastest and most robust simulator for gate level designs. This again is primarily due to the libraries. The Verilog library models were standardized, and thus were the preferred method of signing-off on a particular design with most foundries. VHDL has been limited in its gate level simulation effectiveness due to the lack of library standards. The IEEE VITAL standard is starting to be used by simulation and library vendors to generate standardized libraries.
- **TIMING SIMULATION:** For timing verification we used Verilog because of the simulation speed advantages, the availability of good gate level models, and the SDF back annotation support. At the time we had no viable way of accomplishing the same task in VHDL. Sad to say, that condition still exists.
- **FORMAL VERIFICATION:** The mathematical comparison of a new revised model to an existing golden one is a procedure that we are finding really speeds up our regression testing. The tools that we are integrating into our current methodology at this time started as a Verilog only offering, and are just now being made available with VHDL.

6. SYNERGY OF TWO HDL'S

VHDL IN - VERILOG OUT: The final design methodology that was used in this project was one that leveraged VHDL's benefits at the front end of the design flow, and Verilog's at the back end. VHDL was chosen and used as

the entry format for behavioral and functional (RTL) simulation. This was primarily for its ease of use and clean documentation characteristics. The VHDL was heavily simulated with many test buckets and then synthesized into a Verilog gate level representation. The Verilog was then simulated with and without back annotated SDF delay information. The front end VHDL simulations were run interactively at first, benefiting from the excellent UI's of our VHDL simulators, and then in batch as the input vectors and model sizes grew. Behavioral models were used at times to speed things up. The Verilog simulation was virtually all batch. In using both languages and both simulators we gained the benefits of each.

The drawbacks to this two-language approach are quite obvious. Designers had to know two languages and two simulators; and test buckets had to be converted between the two. These disadvantages were minimized by the fact that the designers need know only basic Verilog and basic Verilog PLI and commands, because the actual structural Verilog code was generated automatically via synthesis. The resulting gate level timing simulations were in the regression phase of the simulation life cycle and thus bugs were typically few and far between. The test bucket translation was eased by adopting a methodology where the majority of the data was in the form of outside text files that could be used in both languages and simulators in the form of TEXTIO. The testbuckets themselves were thus quite simple and easily translatable between the two languages. The translation was done by the few designers that were proficient in both HDL languages.

CO-SIMULATION - BACKPLANE: As time went on and new projects arose, there came a time when we wanted to jointly simulate both Verilog and VHDL together. No simulator was available that could handle both languages so a simulation backplane methodology had to be adopted. The backplane acted as a go-between for the two simulators, keeping track of timing and adjoining signal I/O. This joint simulation proved to be acceptable in getting the job done, and successful simulations at the system level resulted. The designer could use either the VHDL or Verilog simulator as a

main simulator for initiation or debug. We found that our designers chose the VHDL simulator UI environment more often. Still, one needed to have adequate skills and experience in both languages and both simulators to debug the design successfully.

The problems encountered were twofold. The set up time for these backplanes proved to be quite difficult and time consuming; and the resulting speed of the simulation was quite reduced. The first problem is to be expected when combining three tools from three different vendors. Most of the headaches resulted from finger pointing at the other vendors whenever a problem arose. The speed issue was also expected, but the slow down was a bit more than expected. Our single-simulator simulations were long, but not unreasonable, and thus the resulting co-simulation slowdown was a hassle, but acceptable. Again, we substituted behavioral models in proven areas in order to speed up the simulation run times. If we had been pushing the limitations of a single simulator with runs measured in days instead of hours, I think that we would have been a lot more disappointed.

CO-SIMULATION - DRIVERS & GATES: As time goes on, we find that situations are coming up where we have a large model in one HDL that we want to debug. The best way to debug it is to drive and stimulate the design under test with an adjacent design that happens to be in another HDL. Another similar scenario is to have a hierarchical, structural VHDL model and Verilog gate level models. In both cases there is only one language under test, and the other language models are sort of black boxes that do not need to be probed. We are hoping (from the rumor mill) to see simulator products on the market that can compile and use models as black boxes from another HDL much like one can import a C model now.

7. CONCLUSIONS

The bottom line is that there is no bottom line. Both VHDL and Verilog are useful languages for digital logic representation. VHDL is better for some uses, Verilog for others. What follows is a short non-exhaustive sum-

mary of each language's strengths and weaknesses:

- **LEARNING:** A toss up. Verilog as a language is easier and more intuitive to the designer, but the PLI and the prehistoric UIs of its available simulators clogs the learning process. VHDL is harder to learn. It has many constructs and sometimes is so open minded (can do things ten different ways) that its brains are leaking out. But the VHDL simulator UIs are years ahead of the Verilog counterparts, and are very user friendly. Also, the headaches of learning will later be rewarded with clearer documentation and easier transferability between tools.
 - **ESDA & BEHAVIORAL DEVELOPEMENT:** VHDL is the clear winner here. Even if you argue that the languages are equally capable, the EDA tool offerings in this area are heavily VHDL. That does not look to change much in the future. Verilog will likely remain a second class citizen in these higher level tools for some time—maybe forever. As to the need for these tools, it is paramount that we include these higher level tools into our design methodology in order to meet ever-shortening design cycles.
 - **SYSTEM SIMULATION:** System simulation paradigms that should be vastly faster than regular event simulators (used in functional, gate, and timing simulators currently), have been used in large corporations for years and are starting to become commercially available. These simulation offerings should handle much larger systems and allow for joint development of software and hardware. This is primarily accomplished by the simulator checking for function and timing only at the latch I/O and regular I/O of the module, and not at each gate. It is still to early to tell, but most likely tools of this kind will become available for both Verilog and VHDL. The jury is out until actual tools are available for testing.
 - **FPGA and PLD:** The world of FPGA and PLDs is a still quite a mish-mash of data input formats. VHDL is starting to be seen as a more standard input, and is listed more often than Verilog as a viable input to
- many of these tools. Verilog is supported by many of the key vendors; however, those projects that use FPGA's or PLD's need to go with a foundry, tool set, and HDL combination that fits their particular application.
- **FUNCTIONAL SIMULATION:** Verilog simulators seem faster than VHDL counterparts. This is most likely because Verilog simulator companies have invested all of their R & D on making the simulator engines work better and faster. Verilog is also simpler as a language which might make it have a speed advantage. Verilog's speed advantage is most likely to increase because Verilog is just now starting to include some more advanced compiler technology (native-compiled engines—something VHDL simulators already exploit). But as figure 2 shows, speed of the simulation run is most advantageous towards the end of a design cycle. A good fast UI with fast turnaround capabilities is a must in the first half of a design cycle. Here the VHDL simulators are the clear winners, with the Verilog simulators just starting to clue in and catch on.
 - **GATE and TIMING SIMULATION:** Verilog is the clear winner here, due again to simulator speed superiority and library availability.
 - **FORMAL VERIFICATION:** This new simulation-saving paradigm looks to be a bi-lingual environment. Still Verilog seems to have an advantage with current offerings.
 - **CO-SIMULATION:** A two language approach will result in the need for co-simulation. Co-simulation brings a host of concerns with it; translations issues, support, multiple tool cost, higher learning cost, performance issues, and others. All these issues are solvable, and it looks like co-existence, and bi-lingual usage is becoming the wave of the future. In the future, single-simulator solutions should be available that will handle alien HDL models.