



Simulator Savvy

Getting the Most From Your HDL

Most of us picked up HDL design the way we discover a new city. We land at the airport, and a taxicab drops us off at some arbitrary location - a hotel perhaps. From that point, we branch out in an ad-hoc manner, discovering things mostly by accident, and gradually building a mental picture of the place for ourselves. Different people may develop entirely differing views of a place depending on when and how they landed there and how their luck went in this semi-random discovery process. Much of what you discover first is based on what you were trying to accomplish when you arrived - attending a conference, visiting relatives, or maybe hitting the tourist attractions.

In HDL-based electronic design, this ad-hoc learning process can lead to dramatic differences in the style and efficiency of the HDL we create. Because both the verification and implementation tools are automated, we also get dramatically different results. These differences often determine the success or failure of our project. We may be efficient and lucky and end up with a design that verifies smoothly and implements correctly the first time, or we may find ourselves in a quagmire of debugging, modifying and completely re-writing with a design that won't synthesize cleanly, gives unsatisfactory results, or isn't even suited to efficient debug and analysis.

There are, however, a number of things you can do to give yourself the best chance of success. After consulting with a number of experts, here is our list of recommendations and best practices for getting your HDL-based design off on the right foot.

Equip your team right

Designing in HDL is usually an iterative loop between your editor and HDL simulator. You will be using these two things a lot. Be sure you have the right ones to start with. Your time is valuable, and the cost of even the most exotic HDL tools pales by comparison. Think of this as optimizing the

innermost execution loop of your design process. Even a small gain here pays huge dividends.

There are several things you'll want to think about in setting yourself up. While there are a number of excellent HDL products on the market, Mentor Graphics's ModelSim or Aldec's Active HDL are the simulators of choice for most FPGA designers (according to our FPGA design survey). Any good HDL simulator will work, but there's certainly safety in numbers, and you can greatly improve your chances of success by walking the most traveled paths.

There is also, usually, a decision to be made between using the version supplied by your favorite FPGA company in their design kit or buying the more robust version directly from an EDA company. While it is often the same basic simulator (many FPGA vendors OEM Mentor's ModelSim, for example) there are usually features and capabilities missing in the OEM version that can make the purchase of the full version worthwhile. Greater capacity, faster execution, more robust debugging environments, better multiple-language and IP support, and direct customer service are among the advantages typically touted by EDA companies for designers upgrading to the full version from the OEM. ATTENTION BEAN COUNTERS: THIS IS NOT THE PLACE TO SAVE YOUR COMPANY SOME MONEY. Did we mention above that the edit-simulate-debug process is in the innermost loop of the design flow? Have your accounting department multiply the cost of the average engineer by the number of engineers and some reasonable percentage of the project duration, then have them add in the cost of going to market a few weeks later than, say, your chief competitor. Now, compare that number with the few thousand you might spend upgrading to an industrial-strength HDL environment.

Now that you've got the go-ahead from management, here's what you should look for: First, you don't typically use HDL simulation the same way the ASIC folks do. They need to do a lot of long simulations with large designs and millions of vectors in order to verify for double-dog-sure that the design has no chance of a functional error before they tape out and wait for their first round of bad silicon to come back. In FPGA, you just need enough performance to debug effectively with the biggest number of models you might load at one time. Furthermore, you probably care a lot more about interactive versus batch performance of the simulator (again, compared with those poor, suffering ASIC folk), so be sure you clarify before you buy.

Second, many companies will offer you a substantial break on a single-language version of the product. Don't fall for it. "Oh, but we only use Verilog. It's a company standard and everyone here is trained in it..." Interestingly,

we've heard that one before. That's true right up until you need some key piece of soft-IP that's only available in VHDL, or you merge with some company that's a VHDL-only shop (I know, it'll never happen to you), or you decide to create high-level models for some of your blocks in C or C++. These unexpected events are becoming routine in more and more design teams, and it pays to just start out in the 21 st century. Get a multi-language simulator.

Third, there is still a large contingent of designers (and you know who you are) who absolutely must strike each and every keystroke in their HDL files with their own fingers. They are close relatives of the programmers who still write only in assembly (because, hey – it's more efficient) and the masons who still hand-form and bake each brick. This is not, however, the kind of thinking that got us through the industrial age, and it pays to keep up with the times. There are a number of very good products, from templating, context-aware editors to structural and state-diagram tools that can help you create your HDL faster, with more consistent style, and with fewer errors than even the best vi savant can manage on a good day.

If you tried these tools in the past and got less than impressive results, it's time to try them again. Remember when you were a kid and hated beets, but then you grew up and tried beets again, and ... they were even worse? OK, it's not like that. Tools like state-diagram editors have improved dramatically over the past few years with more robust features, better language generation, and more predictable and controllable results. State machines in particular are key sources of errors in HDL design, and a power tool for editing, viewing, and de-bugging can save a lot of headaches, particularly when it comes time to modify an existing design.

Become an expert

Because many of us learned HDL in our spare time, and usually on an "as-needed" basis while working on some actual design project, few have taken the time to go back and really learn the languages thoroughly in a top-down, structured manner. Furthermore, once we've hacked our way through a few projects with some degree of success, it's embarrassing to go to our manager and ask to sign up for VHDL 101. We're afraid it might jinx our je ne sais quois. Some designers go on faking it their entire career, spending a lot of time on support lines asking synthesis companies why their avant-garde style of HDL coding isn't yet supported. Here's an alternative: slip on one of those Groucho Marx noses and go register for an extension class under an assumed name. You'll learn a lot that will truly help you in your career. You might even

run into a few other Grouchos that sound a lot like some people you know from work.

Some of the most under-utilized and helpful tools available for learning HDL style are reference designs. If you find the right one to match your project, you can save a lot of development time compared with starting from scratch, learn a lot of HDL technique, and pick up the all-important style ideas from other designers. Don't assume that just because a design is called "reference" it is well written, however. Double-check it based on what you know about the application, the language, and the original designer. You don't want to get too far down the path with your project and discover that the "reference" design that you used for IP was actually an academic exercise that assumed away some of the real-world considerations.

Beyond learning your language, nothing pays more dividends than becoming an expert in the actual EDA tools that you're going to use. EDA companies have spent a lot in recent years making their tools "friendlier," which means that the average teenager who's mastered computers well enough to use online chat can probably also fire up an HDL simulator and get some VHDL code running in debug mode. The hazard of user friendliness is that it's so easy to get up and running that many designers just stop there, in the front lobby of the technology, using only five percent of their tool's capability. "We have to work hard to educate customers with documentation, newsletters, and training," says Jarek Kaczynski, research engineer at Aldec. "People always ask for new features, then get very surprised to learn that what they're requesting has already been there for two years. It's a constant battle for technical support, sales, and marketing to keep customers informed."

Pass it on

Once you've acquired the right tools and trained yourself properly, you find yourself as project lead on a complex FPGA design with a fresh-out-of-college engineer and three high-school interns. Even if most of your team is experienced, it only takes one young random HDL generator to inject a lot of chaos into your project. This is where a tool like Stellar Tools's recently announced HDL Explorer can help. HDL Explorer allows you to capture "best known methods" of HDL coding and style and run automated checks to see if your design conforms. If you've got a team engaged in multiple design projects, and you want to capture and propagate best practices, an organized method like Stellar's could be invaluable.

Up your game

Today's HDLs are complex, robust languages with rich feature sets. In almost every case, the tools you'll use with that HDL put some restrictions on the subset and style that can be used. It pays to keep that in mind when creating your design. Typically, the synthesis tool is the pickiest about what language constructs are supported and what coding style yields best results. If you write from the beginning with synthesis in mind, you won't get too far down the path happily verifying a design that you can't simulate, or that will give you inferior or unusable results.

Also, if you've read this far and haven't donned the Groucho mask yet, think about picking up assertion-based design. The judicious use of assertions, particularly in and around IP blocks, can dramatically increase your verification efficiency by flagging potential problems at the source. In today's complex designs, an error can take millions of cycles to propagate to an I/O pin where it's visible to your testbench. Mentor Graphics's recently acquired 0-in CheckerWare and Assertion Synthesis provide support for assertion insertion and checking to simplify the process. Assertions haven't found their way into mainstream FPGA design yet, but their potential for instrumenting designs to catch and correct bugs early is huge.

Stay in step

Do you feel like you've mastered these skills already? Don't get complacent. New design and verification techniques are emerging on a regular basis that can help you get your project working, de-bugged, and off to market in record time. Higher-level languages, transaction-based simulation techniques, and formal verification technologies are all making inroads and have the potential to streamline the FPGA design process as our devices and our designs grow more complex. The next time you walk around verification town, keep your eyes open. You may notice a few interesting things you've never seen before.

Kevin Morris, FPGA and Programmable Logic Journal

February 15, 2005