

Test Benches: The Dark Side of IP Reuse

Gregg D. Lahti
gregg.d.lahti@intel.com

Intel Corporation

ABSTRACT

Reusable IP is the holy grail of engineering management; often sought but almost never fully achieved. Companies desperately want to be more efficient and produce quicker time-to-market designs with as few bugs as possible while containing engineering costs. Re-using IP for next-generation products looks great on paper but fails miserably if the proper steps to ensure that the tests, along with the actual created design source, aren't taken into account for reuse before the first line of HDL code is written. This paper will probe the often forgotten, necessary evil, and dark-side of IP reuse, the reusable test bench, and show why IP reuse cannot work unless this side of the design effort is kept in mind.

1.0 Design Reuse

The term “design reuse” is prevalent in the fast-paced, time-to-market driven technology industry. Products must be designed, tested, and manufactured faster than ever to meet a technology-driven market economy bent on having the latest technology toys. Industry has realized that reusing building blocks of intellectual property, IP, from previous designs is one way to get ahead on the project timeline.

1.1 What Really Is IP?

More than a few papers and books have been written about the need to reuse designs. Most every publication calls the design intellectual property or IP for short. But what really makes up the IP? The common theme prevalent in the ASIC design world is that the RTL description of the design is the IP. To a patent attorney that may be so, but conceptually the RTL description is just half of the total design. What helps describe the workings and functionality of the design through a process of testing is really the other half of the IP. A set of functional tests that exercise the design is as important as the design itself. If your design task was to implement an already-coded PCI-X controller in RTL form in your new project, what do you need? The reusable IP design of the PCI-X controller unit is a start, but the compliancy tests that define the functional aspects of the PCI-X controller and allow you to test your custom interface connection between the PCI-X controller and the rest of your design is mandatory to prove that your design is correct. Someone would have to write those tests to verify the design. Wouldn't it be great to have both?

1.2 Time Well Spent?

A study of total engineering time spent on a few previous ASIC projects at Intel provided a rough estimate of total engineering effort breakdown for a design project. The slices of the pie did not change much as the designs became larger and more complex.

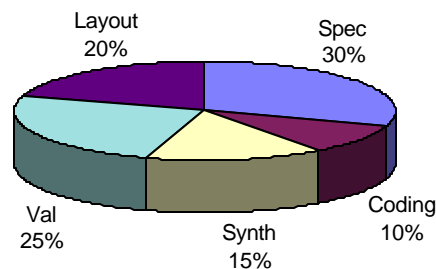


Figure 1: Project Tasks Breakdown in Total Time

The pie graph shows that roughly 50% of the total engineering time was spent coding, synthesizing and validating the RTL description of the design. The rest of the time was spent in architecture definition and back-end layout of the ASIC. Half of the logic design engineer's tasks (coding, synthesis, and validation) is the validation work, which includes functional testing, netlist or formal verification, and static/dynamic timing analysis. This translates to a minimum of a 1:1 validation engineer to logic design engineer ratio of effort for an ASIC project.

2.0 Why IP Reuse Fails: The Designer's Sandbox Rule

Engineers love to create their own designs. It's a fact. Maybe it is ego driven, maybe it is the technologically creative nature of engineers, or maybe it is the hint of doubt if the previous engineer could do the design as well as it could be done now using new technology-driven tools and different criteria. The fact still remains that engineers love to create their own IP.

There are many factors which weigh into re-creation of a design. Some of the key items that stand out are:

1. Existing design needs modifications, so doing it from scratch would be easier. It is difficult understanding a design done by other engineers and may not be well documented.
2. Existing design not setup for reuse- custom logic, instantiated cells, etc.
3. Reusing a design isn't glamorous.
4. Hint of doubt- is the existing design good, can it be improved, unknown bugs?

Designing new features are what engineers are good at. Hence, the designer's sandbox rule: a design needs recoding because it won't work in the new application without adding features.

However, a quick survey from an audience at the '99 IHDL Conference done by Cliff Cummings showed that only a handful (less than 10%) of engineers liked to write test benches and test code to verify the design. Survey your company and see how many engineers re-engineered a design but went out of their way to reuse the existing test bench even if it was not efficient, well documented, or even up to specification!

Another roadblock to reuse is found in the actual implementation. Engineers usually must spend a small percentage of effort at the beginning of the design to implement reuse elements. Making the design portable in terms of platform, process, library, tools, and environment requires extra time and effort that the engineer or the management may not be willing to spend due to schedule constraints. Re-engineering existing legacy designs to utilize a reuse strategy also prove a daunting task.

An informal study was done at Intel to determine the gains of reuse for RTL-based designs. It was determined that implementing reuse was a costly venture and gains would only be seen *after* the design was implemented a second time.

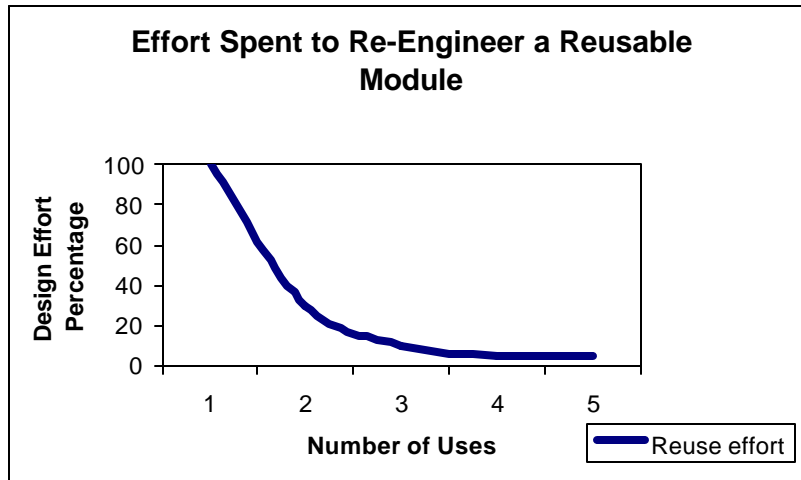


Figure 2: Engineering Work Required When Reusing a Design

The study looked at the basic headcount costs, design modifications that were required for the next implementation, and re-validation issues over several reused designs. For some design modules, such as UARTs, interrupt controllers, timers, and generic bus controllers, reuse made good sense in terms of engineering cost vs. benefit. However, large complex items that needed complete re-engineering due to library, process, and advances in architecture were not good targets of design reuse. Figure 2 points out an obvious problem: some amount of design implementation and validation time is still required to implement reuse.

3.0 The Ties That Bind IP

Reusing IP requires additional up-front work in order for the whole reuse process to work. There are generally four basic items that a design must include for reuse to be successful:

1. A document describing the design at a micro-architecture level
2. The RTL description of the design with no instantiated gates and hopefully well-commented code
3. The synthesis scripts, possibly oriented to a generic or included library, with documentation of all constraints, false paths, and clocking information
4. A self-checking, easily modifiable test bench that is well documented

What generally happens when reuse guidelines are not followed is the design function winds up oriented to a specific technology process, such as a .25 um standard cell library at a specific frequency of operation. This method subsequently introduces sub-optimal workarounds that break the reuse guidelines and limit the portability. For functionality-based test benches, the same holds true: bad test benches with poor foresight to reuse kill any hopes of future reuse.

4.0 Poor Forms of Test Benches

You too can “shoot yourself in the foot” multiple times when implementing test benches. If a well-thought, high-level testing strategy is not followed, the test bench reusability gets hindered to the point of non-reusability. Some examples of poor test benches for re-use would include any of the following, ordered from worst to workable-but-ugly methods.

4.1.1 Vector Stimulus

A vector set is a group of 1's and 0's that contain input stimulus to the input pins and usually expected output from the output pins. The ability to understand what the vectors are doing (i.e. documentation of the stimulus) as well as the ability to modify the tests to incorporate bug fixes or design improvements is lost due to the low level format. The worst possible use of vectors is to create a “golden set” of test vectors by visually verifying the waveforms, saving the stimulus vector file, and then subsequently verifying any future simulations against the “golden” vector set. This method of visually verifying the waveforms is not only time consuming, but very prone to human error. Applying stimulus with A/C timing in mind generally requires specific knowledge of the interfaces being tested. Once this happens, the test bench becomes non-portable due to frequency constraints –i.e. the tests cannot function at a faster frequency since the vector set will need to get scaled differently or the tests cannot be modified to support a different A/C timing environment.

4.1.2 Assembly Language Code

A proprietary, low-level language like assembly code to drive a Bus Functional Model within a system-like environment with a processor, bus controller and program memory is next on the worst possible usage list. Assembly code generally means a lower level of abstraction of the test and limits the engineer in easily creating a functional test description to perform large, complex testing operations. The assembly language code test bench will work, but the effort to re-use it requires more overhead in terms of tools used to compile the assembly to object code and the effort to create the test. By using an assembly-code driven test bench, reusability gets limited to a platform-specific tool for code compiling, i.e. the compiler only works on a Sun Solaris[®] 2.5.1 solution or worse, a Windows NT[®] solution. The full-system environment used (processor in BFM form, bus controller, and program memory) also limits reuse since the entire environment must be re-created as the test bench in order to reuse the tests.

Finally, assembly code isn't portable across different micro processors/controllers. If an engineer created a special function I/O block like a USB[®] controller and wrote the tests in assembly targeting an X86[®] microprocessor, the tests would need to be recoded if the block was to be reused for a System On Chip (SOC) solution using a StrongArm[®] core. The use of specific-architecture assembly code forces the whole X86[®] system architecture to be emulated in the X86[®] system test bench to test one block. Once again, test bench reuse is now limited.

4.1.3 Scripts and Environments From Hell

EDA tools are never perfect, and no testing solution will always fit the requirements. To patch problems at hand, the engineer winds up creating a script-based workaround, usually in perl. What can turn a test bench into a non-reusable nightmare is the when engineers break away from an industry standard, widely used, HDL language (VHDL, verilog, C/C++) to do the testing and create a whole environment of support scripts, test language scripts, and pre/post-processing scripts. It is difficult to create a modular test bench for a design when the test bench needs to incorporate a dozen 3000 line perl scripts relying on many environment variables, hardcoded paths, and a chain of scripts calling more scripts. It also turns into a support nightmare when the script and environment are ill documented and the engineer is no longer working in the department or company. Engineers like writing solution scripts, but commenting & documentation is usually sacrificed for quick implementation and schedule time.

5.0 What Makes a Great Reusable Test Bench?

A reusable test bench requires an “Open Test Bench” philosophy. Ideally, this means that a minimal set of tools that are industry-proven, and widely available using standard, industry known design languages should be utilized to implement and run the test bench.

5.1.1 Test Bench in Widely-Used RTL

For maximum reuse of tests, the test language should be ideally in the HDL (VHDL or verilog) of the chosen RTL simulator. The design is using a HDL to describe the RTL/behavioral code at a high-level of abstraction, why shouldn't the test bench be similar? The engineer reusing the design will generally have the same simulator as the code is written in and a good understanding of the HDL in use. The test is more portable to other same-HDL simulators, easier to understand, and faster in simulation because it is usually compiled into the simulator as a native language. There is no need for learning a new language of an expensive testbenching tool, and the simulator license that the engineer owns is all that is required. The engineer will save time and effort on the learning curve if the test bench and test code is in the RTL of choice. As always, documented and commented test code should be a top priority for reusability.

5.1.2 High Level Testing Methods

Second, create high-level procedures and functions in the HDL language to drive Bus Functional Models (BFMs) in the test bench. The use of these functions and procedures allow a greater level of abstraction to the test code and enable the test writer to perform complex operations with minimal written code. By writing at a higher level of abstraction, the test code is very portable and can be applied toward testing any abstraction of the design, such as RTL and gate-level, back-annotated-from-layout timing netlist. For examples of writing tests using procedure-driven BFMs in VHDL, see the San Jose SNUG'99 paper [Designing Procedural-Based Behavioral Bus Functional Models for High Performance Verification](#) by Gregg Lahti and Tim L. Wilson. For writing tests using functions and tasks

in verilog, see the San Jose SNUG '99 paper A BFM Simulation Strategy for Verilog by Rodney Pesavento and Michael Privett.

5.1.3 The use of C/C++

The one exception to the “same test language as RTL rule” is the use of C/C++ through PLIs. As most engineers are painfully aware of, verilog is limited in the text I/O area. Using PLIs to provide useful I/O and string facilities like scanf, sprintf, etc., greatly enhance the test benching ability. Testing SOC designs where large efforts are placed in software-based testing can be accelerated by the use of software-created C/C++ test code through the PLI interface. The plus side of using C/C++ is that software writers can supply tests very quickly or provide historical code to test the design. Most engineers do know C/C++ and C/C++ is a non-proprietary, ANSI standard language. Since all verilog simulators should by now understand the OVI 2.0 PLI standard, the integration, standardization, and license issues of using C with a verilog simulator aren't a problem.

The C/C++ issue is a bit more complicated with VHDL simulators, mainly because the 1076-1987 spec didn't standardize the C/C++ interface to the simulator. Hence, different VHDL simulators have different Foreign Language Interfacing (FLI) guidelines that tend to not be portable across all VHDL simulators. A PLI standard for VHDL needs to be set in the short-term to overcome this problem. It would also be beneficial in reuse and portability terms to provide an industry standard FLI interface that is identical to the OVI 2.0 PLI spec. With the non-standard C interface issues, it is probably a good reason to use verilog instead of VHDL for RTL simulation with any C/C++ based testing.

5.1.4 Limit Environment and Scripts

For every script included in the reuse of a design, there is an associated overhead to support the script, document the script, and deal with possible side effects of the script. Reuse gets worse if the script is in a language that engineers may not normally use, such as AWK or worse, some proprietary language that an engineer must pay for a tool license to use. Generally, all design environments have some baseline of UNIX tools available- even if the engineer is cruelly subjected to working on a Windows NT[®] platform. Utilizing UNIX-based tools that are freely available (gcc, perl, make) is quick and easy to make an environment portable to any platform.

Of course, just using UNIX-based tools alone will not help a test bench be portable. The design environment in which the test bench and simulation is run makes a huge impact on portability. Hardcoded directory paths in a simulator-specific, perl-based compile script limits reuse by the next engineer- even within the same company! The use of a few environment variables in a basic setup CSH/SH/KSH script to get the engineer “configured” and the requirement that all project related scripts and tools refer to these environment variables is one solution that works. Again, reuse can be hindered if the test bench environment requires 30 different environment variables for functionality, so don't go

overboard on the variables. Keep the environment variables concise. To minimize the effects of scripting death that affects reuse, a few key guidelines should be established:

1. Incorporate a setup script that the user sources to get basic environment variables setup. Putting all environment variables in one place makes it easy to modify by the next user. All project lib and bin files should live in in one location that the environment variables point to.
2. Organize and document the code. Simple to say, but not as easily followed or enforced. Take the time to do it right.
3. Custom scripts must have some measure of “reusability” implied in the script. Hardcoded paths to tools and the liberal use of vague environment variables aren’t easily debuggable or reusable. If tool paths or project-specific paths must be defined and can’t be setup by item #1, put them at the beginning of the script and document what the usage is for and why the items are in place.
4. If the script is patching a specific problem, the scope of problem and determined solution must be documented in the script for future reference. If a fix for an EDA tool arrives which renders a “patch script” obsolete, get rid of the script. This prevents the script from becoming a “Best Known Method” and subsequent continued historical use of the script on future projects because future project users had no documentation on why the script was originally used or what the script accomplished (item #2).

5.1.5 Keep It Stupid Simple

Test benches don’t have to be large, complex beasts with massive amounts of scripts to provide great test coverage. The simple environment is usually the best environment for getting the job done and keeping portability. The goal of the test bench is to provide a suite of tests covering the functionality and in most cases, a fault grading value of a design. Complex test benches with multiple, custom scripts are pure overhead work for an engineer to maintain.

6.0 Why Not Vera™?

Synopsys introduced the Vera™ testing environment in 1998. Vera™ itself is a good concept: setup a cycle-based testing verilog-like language that has the object-oriented constructs and add in some extra statistical control to the testing environment. Engineers who have used the product profess the fast test writing ability once the learning curve has been mitigated.

The use of Vera™ does satisfy the portability and reuse requirements for test benches, as long as the next user of the design has or will purchase the licenses required for Vera™. But does the industry need yet another language that is not an open standard freely available to the design community? Does learning yet another programming language fill in all of the holes that a pure HDL like verilog cannot do? Can the same job be done in verilog, VHDL or C/C++? Does the ASIC engineering community, which doesn’t always think like a CS major, really need object-oriented tests to implement the job?

If the same principles of a reusable test bench from section 3.2 are observed, Vera™ fails in terms of requiring yet another language syntax to write tests. It also fails in the “open test bench” philosophy, where the user is required to buy a quantity of licenses per simulation seat and locks the user into a proprietary solution. Additionally, Vera™ is a new language specification. It will have its share of bugs and annoyances just like every other compiled or interpreted language that need to be found and fleshed out over the course of users and time. One comment made in 11/1/98 EE Times article “Babel of languages competing for role in SoC” by Richard Goering and Peter Clarke outlined this drawback of Vera™:

"There are a number of competing approaches to system-level design, but the only realistic candidate in the context of SoC design is C++," said Synopsys' Brian Barrera. "Proprietary languages do not easily connect with C/C++, the de facto standard languages for systems and software development. Proprietary extensions to C are not appropriate, because they lock you into a single supplier."

At this point in time, Vera™ is not widely utilized or cost-effective for large-scale testing applications. It is my opinion that you can do all of the functions of Vera™ in verilog with a C-based PLI for text or I/O functions, and if you're crafty enough, in VHDL as well. The issues of portability to other users with regards to licensing, language learning curve, and the “newness” of the language makes the case that Vera™ either needs to be free (probably not a realistic possibility) or verilog needs the object-oriented aspects of Vera™ or Java in the next edition of an OVI-standardized language. Looking back in history with regards to verilog, Cadence took a bold step to allow the verilog language to be an open, IEEE standard. Verilog is now the predominant RTL language of the industry with many offerings of simulators, lint tools, synthesis tools, and consultants teaching verilog. Maybe Vera™ could be the next testing HDL if it was open and freely available.

7.0 Conclusions

The scale at which integration and SOC designs are ramping require design reuse as a key element in achieving realistic time-to-market. The testing aspect of the design tasks is easily over half of the total effort. Providing building blocks with a high-level, open, and reusable test bench can improve design quality and TTM.

The key items to creating a reusable test bench:

1. Tests and test bench in industry standard, widely known RTL language
2. High-level abstraction by use of procedures, functions, and tasks
3. “Open” environment: use basic UNIX tools, limit scripts, keep to industry-standard languages and tools
4. Keep it stupid simple

A design without a comprehensive and complete test is completely useless.

I wish to thank Tim Wilson for reviewing, editing, and providing useful ideas for the paper. I also wish to thank Cliff Cummings for convincing me one afternoon to write the paper.