A Recipe for Multi-Million Gate ASIC Verification

By

Peet James

of

Qualis Design Corporation

ABSTRACT

The overall verification task on multi-million gate ASICs is growing exponentially. If you do not have a strategic verification plan, you will pay for it dearly down the line. This paper outlines a recipe for productive multi-million gate ASIC verification. It includes design management, top-level harnesses, bus functional modeling, behavioral modeling, regression testing, revision control, and finally scripting, which ties everything together. Methodology experience gained on several recent large ASIC designs using Verilog and VCS is incorporated.

1. Introduction

Employers and ultimately customers are screaming for larger and more complex designs in less and less time. Silicon vendors are saying a big YES with capacities that can handle just about anything one can throw at them. EDA tools are slowly catching up to handle the deep submicron world of ultra-dense designs.

But before we can ship these large multi-million gate ASICs, we have to verify that they are functionally correct. The task of verification has grown to have a life of its own. The typical verification effort is now estimated to be up to 70% of the overall effort, or as Figure 1 shows, that for each line of RTL code, 5-10 lines of verification code will need to be written.

We need a new recipe, tailored to each individual design that minimizes the effort while maximizing the degree of certainty that the ASIC is functionally accurate. This paper is an overview of how to do verification on a large multi-million gate ASIC. The bottom line is that early design discipline, setup and documentation will save many hours later in the design process. This early investment is key.

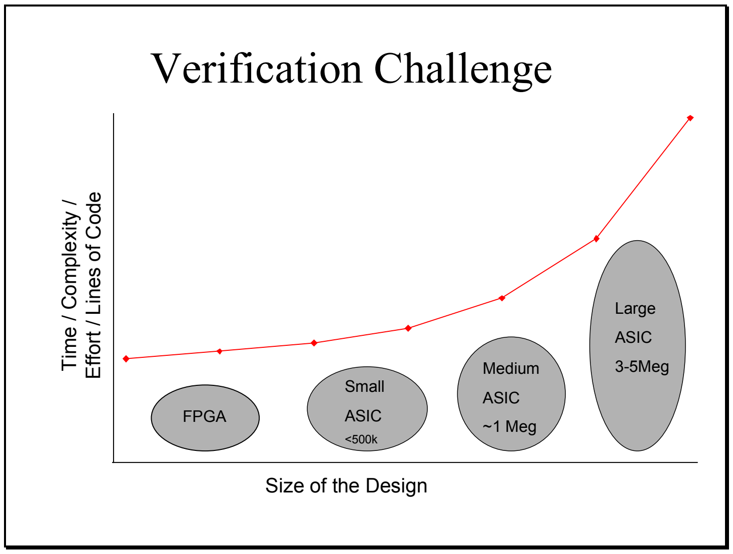


Figure 1 Each line of RTL code requires 5-10 lines of verification code

2. Key Ingredients

- Verification planning and documentation
- Design management, revision control and scripting
- Verification infrastructure

3. Verification Planning and Documentation

Just about everyone knows and uses a general ASIC specification. Any good ASIC designer will tell you that a good spec done up front and kept up to date is worth its weight in gold. An ASIC spec needs to be clear and concise.

What is also necessary today is to have a sister verification plan (VP) document. It also should be clear and concise, be done up front, be owned and kept up to date. The verification team works off this important document. It outlines the overall verification approach, structure and game plan. The VP divides the work into categories (such as basic sanity, intentional, and stress), and then details the feature tests by major interface, block, function, etc… Finally, the VP groups these feature test into testbenches, listing who will do them and an effort estimate.

A verification engineer armed with a general spec and a verification plan is primed to jump start the verification effort. With many important decisions made much earlier in the design cycle, the verification team can queue up testbenches for the RTL before the RTL is even ready. This parallelization of the verification and RTL effort is crucial in the success of large ASIC designs.

Another benefit of the VP is that team leaders can assign and gauge progress off the status generated from this document. The verification team, from top to bottom, needs to buy-in and own this approach. An investment up front to collaboratively and creatively come up with and own a verification plan saves many hours later in the design process. Skeptics to this idea are sold on it after just one project and will not do another without a good verification plan. There is also nothing to stop the team from having multiple VP docs for different parts of the design (module level). However, these specs should be all of the same format, be owned, and be kept up to date.

Remember the VP has a different audience and purpose than the general specification. Some of the content may overlap, but it is important to let the verification team write it, format it to their requirements and own it.

3.1. Example

We called the VP a conformance verification plan (CVP) on the last couple of large ASICs that I worked on. On this last project the CVP was less than 20 pages in the end, and took one week to put together. The team had to attend five one-hour meetings that were creatively and efficiently run (one per day at roughly the same time each day). The owner then combined the info and wrote the doc as the week progressed.

Section 1 spelled out the overall black-box approach of loading in processor instructions (either directly via a file, or indirectly via the real memory loading interface), and generating input and output frames (to stimulate and compare) the design under test.

Section 2 spelled out the data structure of all the data (more info below in Example 2 of the next section). Sections 1 and 2 were a result of the first two one-hour meetings.

The feature list to test was in Section 3 and grouped into three categories: basic sanity, intentional, and stress. The breakdown of each category into sub-categories came about from three creative, efficient one-hour meetings where various techniques were used to maximize and categorize features to test.

The final section (4) was a table that listed the actual testbenches and cross-referenced the list of features to test listed in Section 3. This was done off-line and for the first draft the table was mostly blank except for a good chunk of the basic sanity info that was filled in. This section was filled in all throughout the life of the project. It was important for the team to know that we were taking our best shot here, just getting a framework and a preliminary plan down on paper. Things changed later and the doc evolved alongside the changes. It was equally important that each engineer gave input and had his or her say. This input got them involved early and feeling that they were a part of the process. This involvement went a long way to getting early buy-in from everyone.

4. Design Management, Revision Control and Scripting

The days of a couple of engineers setting up an adhoc design environment and keeping things straight is over. Today's multi-million gate designs have too many files and too many people working on the design. Oftentimes the project crosses several locations around the globe. The idea is to have an easy to use automated setup that is flexible enough for everyone to use. Loan-wolf engineers who do their own thing can kill projects (and companies).

Investment up front is key. Whether you buy an off-the-shelf tool, script you own, get freeware off the net, or use a combination of some sort, the key is that the overall setup is decided on and put in place. The only mistake a team can make is to not have a revision control system. Good design management will have some sort of revision control that makes sandboxes for engineers to checkout snapshots of the locked repository, and then check things back in once they are sure the updates are valid.

The ability to tag certain file combinations is important, as is the ability to retrieve lost data. By the way, like most of these verification key ingredients, someone needs to own the database and revision control, keep it up to date and support it. It is also important to inject some creative and fun repercussions if one 'breaks' the database. We had a rotating Trophy of Shame for anyone who checked in files that broke the regression. This is the best, most fun way to ensure proper usage. None of this revision control stuff is new, our software friends have used such setups for years.

Besides revision control, a one-stop shopping place for project info is a good idea. An internal web page works great here. Hyperlinks to the general specification, the verification plan(s), cheat sheets for setting up tools (like revision control, sim scripts, simulators, synth, etc.), bug-tracking, internal news groups, and anything else that the team might need to keep working. This repository of information will be the first place any design team member will go for help.

Two more things while we are talking about project-level ideas:

- Some good bug tracking software is crucial. These systems help keep bugs from slipping through the cracks and give an overview of where the project is going. Some engineers cringe at using them, because they view it as a black mark to have a bug posted against them. Management needs to convey this as totally untrue. A creative sort of reverse-psychology solution is to set up some sort of reward system for most bugs found and fixed.

- An internal news group is great for tracking new ideas, new features, or changes that are not really bugs. One can post a concern or idea, then dialog about it until a decision is made and then implemented. A record of the process is kept automatically.

Finally, scripting to setup any and all functions goes a very long way. Not only do scripts speed things up, they enforce uniformity without much effort. If a designer knows of the script, can grab it off the internal website, and easily understand what it does and how to use it, then it will be used universally.

Scripting improves project efficiency. It is a good idea to have a script deity on the project. Someone who can crank out PERL and other scripts. A resource person who is on the design team, but whose job description leaves room for them to develop, clean up and maintain useful scripts is very helpful. A lot of scripts are out there, they just need someone to find, tweak, and locate them in an easy to find place and maintain them. One person with their finger on the scripting pulse needs to be empowered to gather and maintain these useful tidbits of technology.

The web page, the database format, the revision control, the bug tracking, the news group and the scripts, all this infrastructure needs to be set up up front, owned, and maintained. Early investment will pay off later.

4.1. Example 2

I have used modified freeware for revision control and bug tracking in the last couple of large designs. They were modified to fit our team's needs. They worked because they were set up early, easy to use (via a web page tutorial) and supported.

The verification database that we set up was also unique. It took into account the compile-saving features of VCS and broke the starting of simulations into two scripted parts: one to build the executable, and one to regress one or more simulations.

The nature of our black-box approach meant that different machine code loads and auto-generated packets drove the simulations, not the Verilog. The regression script called a file within each testbench sub directory. This script had the flexibility to run a simulation in an infinite number of ways. It could do a build if it had a unique Verilog driver, or it could (via PERL or Unix commands) generate a bunch of files and then use the previously generated executable that used generic Verilog driver files.

Most of the simulations used the later approach and had the beauty of much shorter compile times. The individual testbench script also allowed for much post-simulation processing that automatically processed and reported errors. Standard self-checking scripts and routines were available so as to keep the output uniform. The regress run script also let the user declare the run directory so that each run of the sim could save its own data. Details of how this all worked and example scripts were generally outlined in the CVP, and more thoroughly detailed in cheat sheets and examples under our internal web page. Issues were tracked via bug tracking and an internal news group.

5. Verification Infrastructure

The top of Figure 2 shows a typical verification approach where the verification is tacked onto the end of the design process after the RTL is mostly ready. Heavy reliance on individual module verification means most designers set up their own testbenches. System-level verification occurs at the end during crunch time, and often gets squeezed.

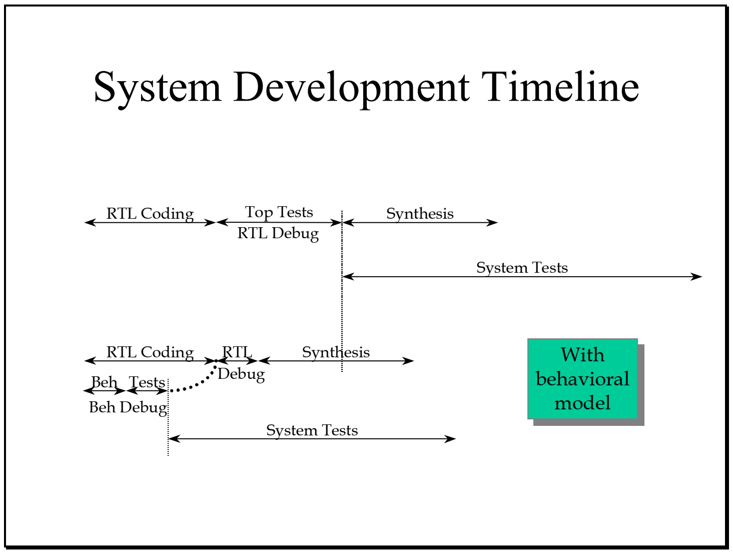


Figure 2 Using behavioral models enables system tests to begin much sooner

The bottom of Figure 2 shows a new approach where a separate verification team can queue up testbenches for the RTL team to use. This parallelization has many benefits. Module-level verification can be reduced, and bugs can be found much earlier. Several sets of eyes are reviewing the specs, so interpretation errors can be found more easily. Not only is the design cycle reduced, but also often because of the more robust verification environment, much more simulation coverage can be obtained. The keys to this jump-starting are to have invested in a VP, database, revision control, scripting, and bug tracking and then to apply behavioral modeling techniques.

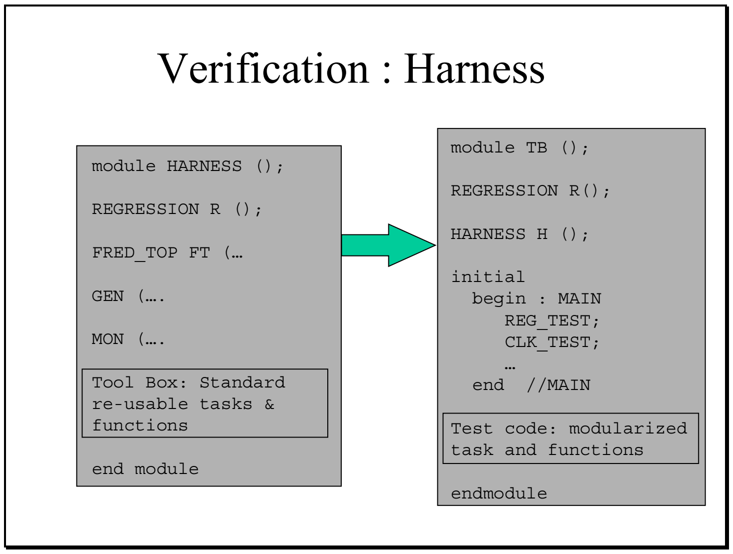
Bus functional modeling (BFM) for all major interfaces is key. These chunks of code that mimic protocols via abstracting out re-useable operations into simple tasks are easy to write and can be made self-checking. You can buy them in most cases for standard processor or buses, or you can automatically generate them off a timing diagram with available tools. Even internal proprietary buses can easily be coded up. Some engineers think this is a fairy tale, but once involved in a project with BFMs, where they can do software-type calls to tasks, they are sold to the benefits. The investment up front is key. Locating the BFMs in one spot and having one owner enables the overall project use of these valuable code snippets. Documentation cheat sheets are also useful.

Besides BFMs, having an actual behavioral model of the ASIC is key. One can combine existing BFMs and glue logic to mimic the real chip. Hierarchy can be cloned for one or two levels of the RTL (to facilitate a swapping in and out of beh, RTL and gates for speed or availability), but should be minimized. The danger here is too much detail. Full behavioral models have all the delta cycles in the world. This should be taken advantage of. Keeping things simple is key. Model RAMs and regs in the most speed efficient, easy-to-change manner. Add only timing info that is absolutely necessary. Keep it simple and remember that all the behavioral model need do is to mimic the real ASIC from a black-box point of view. If it is taking a month to develop, then it is over-complicated.

Reluctance to this approach is typical. Sometimes, management will only swallow doing behavioral models for a couple of modules. This alone proves valuable in itself, and often leads to full behavioral modeling on the next project.

Once a behavioral model and adjoining BFMs are ready, the verification team can build a harness to encapsulate the design and create a useable verification environment. Figure 3 shows the no-I/O top-level approach for the harness and testbench. With a VP, BFMs, behavioral model, and harness, the verification team can write and debug a host of testbenches in parallel with the RTL development. Bugs in the testbenches, the specs, and the harness, etc. are found and fixed early. RTL can be folded in as it becomes available. Typically, a queue of benches is ready for the RTL team. The RTL guys do not have to do as much unit-level testing and even the one they do can be streamlined with them using the harness and typical toolbox of code that has already been developed during the verification effort. The RTL team can go into debug mode instead of writing testbenches. They can direct further testing in new areas. A regress of basic sanity can be automated to run before check ins.

This parallel verification technique has important staffing implications. Parallel schedules require parallel resources!! And a parallel verification effort means more people (but for less time). You have to be willing to increase the staffing to obtain the design cycle time benefits! Trying to do behavioral modeling, a CVP, etc. without increasing staffing is of marginal benefit.


Verification : Harness

```
module HARNESS ();

REGRESSION R ();

FRED_TOP FT (…

GEN (….

MON (….

Tool Box: Standard
re-usable tasks &
functions

end module
```

```
module TB ();

REGRESSION R();

HARNESS H ();

initial
  begin : MAIN
     REG_TEST;
     CLK_TEST;
     …
  end  //MAIN

Test code: modularized
task and functions

endmodule
```

Figure 3 Example of the no-I/O top-level approach for the harness and testbench

5.1. Example 3

In one project that I worked on we used an off-the-shelf BFM for a PCI, we built our own SONETBFM (none were available at the time) receiver and transmitter and used them both inside the ASIC behavioral model and outside in the harness hook up. The SONETBFM would fill in data for the packets, and automatically set up all the up-front control frames. The sets of frames could be fast file loaded into the ASIC for speed or taken over the actual interface. We built a full behavioral model of the ASIC, had a working CVP, a harness, run scripts, and CVS database, etc. letting us queue up around 20 testbenches before the RTL was ready. Lots of bugs were found early.

6. Who Does What?

It is important to have a separate verification team when using this approach. Typically, RTL engineers do not have the behavioral coding techniques to accomplish the level of code needed to achieve success. They also have their hands full writing the RTL. However, they should be involved early on in the VP and verification environment and infrastructure. Here is a suggestion of personal distribution:

- Database and VP engineer - Develops and updates VP, maintains revision control, one-stop shopping web page, bug tracking, etc.. Typically, one person handles this function.
- Harness, toolbox and BFM engineer - Writes and maintains the harness, often the top-level is included in this, BFMs, and toolboxes that develop along the way. If it is simple, this responsibility can be handled by one person. Otherwise, it might need to be more people (say one for each BFM).
- Behavioral model engineer - Writes and maintains ASIC behavioral model. Typically, this is just one person.
- Testbench engineer - Interprets VP and GS and writes testbenches. This could be several people (seen it need up to five on large ASICS).

Another approach is to distribute the work among the whole verification team. It is a good idea to have everyone be responsible to write a couple of testbenches so that people learn to appreciate the entire verification infrastructure.

7. Conclusion

Investment up front pays off later in the design cycle. Finding the sweet spot that minimizes the effort while maximizing the degree of certainty that the ASIC is functionally correct is key. Taking the time to make a solid verification plan, obtaining and keeping ownership of the plan, and making a careful commitment to implementing that plan is the only way to speed up and increase the overall quality of a large ASIC verification process.