Testing the Testbench

Stan Sokorac
stan.sokorac@arm.com
ARM Inc., 5707 Southwest Pkwy, Building 1 Suite 100, Austin, TX 78735

***Abstract*- Complex software needs thorough testing to perform as desired. However, testbench development has the paradoxical requirement for working RTL code to run on, which, by definition, is not working properly. This paper shares the details of a solution to this problem implemented by the ARM® CPU verification team. By creating a testbench focused on high-level abstractions, with an architecture that allows it to run and verify itself, we were able to deliver a well tested vehicle for design verification. Through this paper, I hope to make "testing the testbench" a more widely accepted testbench development methodology.**

Keywords- Verification, Testbench, Bugs, Unit Testing, SystemVerilog

I. INTRODUCTION

Modern CPU testbenches are undeniably complex software. The memory subsystem testbench on the previous generation CPU core had 310,000 lines of code. According to research from Microsoft[1], experienced programmers make about 10-20 errors per 1,000 lines of code, which leads us to expect between 3,000 and 6,000 bugs in a modern testbench. In the software industry, using untested software of this size would be unthinkable, though in our industry, it is common practice. This puts design and verification engineers in an unnecessarily tough situation – the verification engineers are working through their bugs to get the testbench up and running to the point where it can find RTL bugs, while RTL engineers are developing increasingly complex features on top of their completely untested micro-architecture. When the RTL bugs are finally discovered, they are in the code that was written weeks or months ago. This code is now harder to change, and it is more difficult to untangle the dependencies on faulty code developed over time.

A variant of the Virtual Prototyping methodology [2] was used to tackle this problem, in which a testbench was developed that can be tested and debugged as soon as it is written, without any dependence on the RTL. Because of this decoupling, the testbench reached a higher level of quality much earlier in the project, and was available to stress-test RTL as soon as it was written, yielding additional efficiency improvements in RTL workflow.

II. VIRTUAL PROTOTYPING

A. Shift Left

In a typical product that includes both a software and a hardware component, the software is tested on the hardware it will run on. This requires functional hardware first, serializing hardware and software development. Virtual prototyping's goal is to remove this serialization, thereby reducing the overall project timeline [3]. It is a methodology in product development in which a software prototype of the hardware component is built early, facilitating development and testing of software that will eventually run on it.

In many ways, this is analogous to the relationship between RTL (hardware) and testbench (software) development. The testing of the testbench software is gated by the RTL development, serializing much of the early effort in the design project. To combat this, we have adapted this idea by building software prototypes of the RTL fully capable of exercising the majority of the features of our testbench. With this setup, we have effectively pulled-in the schedule for the verification work (Fig. 1), something that is now commonly referred to as "Shift Left".

B. Objectives

There are many different ways that functional models of RTL have been written before, all achieving different goals. They can be cycle-accurate, or very abstract, they can be written in SystemC or SystemVerilog, and so on. We have settled on several major objectives that guided our choices in the design of the prototypes and associated infrastructure.

First, the models did not require cycle accuracy, and simplicity and focus on high-level abstractions was a key requirement. While all models had to follow all of the ARM Architecture Reference Manual rules for an ARM CPU, they did not necessarily have to behave exactly like the real RTL eventually would. By focusing on transaction-level modeling, the development was quick and straightforward, while still providing correct behavior that will exercise our checkers.

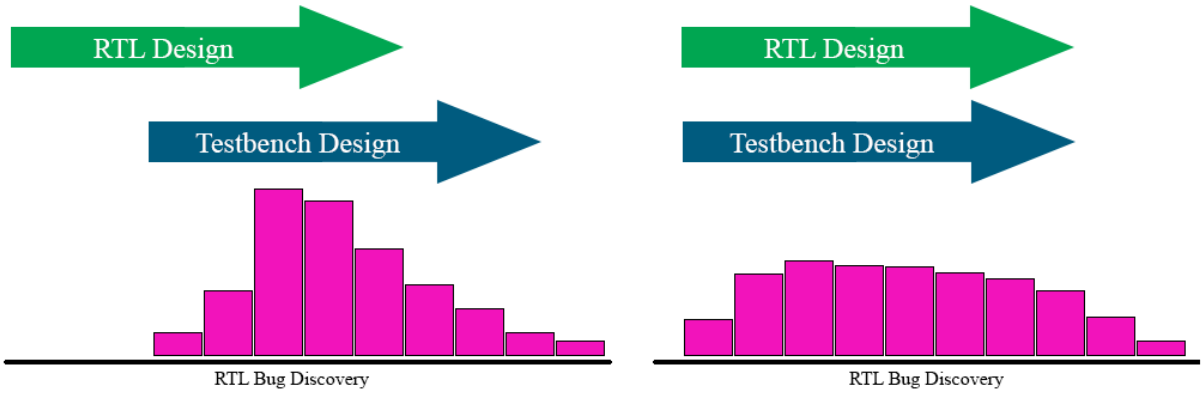

Figure 1. The traditional model on the left produces high, often unmanageable, bug discovery rates as almost-complete RTL is brought up the first time, and it takes longer for bug rate to settle down after RTL is complete. On the right, the "Shift Left" model discovers bugs at a steady rate as they are introduced into RTL, and takes much less time to settle once RTL is complete.

Second, the testbench and the prototypes had to be structured with incremental feature development in mind. Instead of waiting for the full set of features to be developed, each one must be implemented and tested independently. The resulting strong "shift left" of the verification capabilities provided a working testbench subset that is consistently a feature or two ahead of the RTL development.

Third, we required a testbench setup in which any of the prototypes (models) can be substituted with real RTL once it is ready. This meant that each model had to be able to communicate with other models and BFMs over real RTL interfaces, while also being able to connect procedurally.

Finally, the models must not limit our ability to generate stressful and meaningful stimulus to test the RTL. This was a common problem encountered in previous attempts to build testbenches in which models and RTL can be switched for each other.

C. Testbench Overview

Our setup was a multi-unit testbench including the load/store unit, level 1 and level 2 caches, and a memory management unit. SystemVerilog interfaces, including modports and clocking blocks, linked the testbench to the RTL design which was implemented using standard Verilog.

These interfaces were the key to achieving several of our major objectives: the ability to switch models with RTL required an accurate interface implementation, and interfaces were going to be the key for both driving our stimulus and source of transactions for our scoreboards and models. For that reason, we developed an infrastructure that auto-generated interfaces and many components dealing with them, which allowed us to build and test them quickly and easily, while abstracting out all low-level code from the rest of the hand-written testbench code.

When it came to developing checkers, we focused on high-level architectural checking, as well as high-level implementation that deals with design abstractions instead of low-level interface transaction monitoring. The result of this was rapid development of initial checkers and immediate ability to test significant portions of the testbench.

D. Interfaces and Interface Components

In order to maximize our ability to bring a testbench up as quickly as possible, we wanted to eliminate as much of the tedious low-level code as possible, and have engineers focus on high-level stimulus and checking right away. Our basic architecture followed the traditional UVM principles of environments, agents, monitors, and drivers: one environment per unit being tested contained interface agents for each of the interfaces, and the agents contained monitors, drivers, interface checkers, and coverage.

Where we departed somewhat from the traditional principles was in the amount of complexity in these components, or lack thereof, to be more precise. They had to be simple to achieve our goal of auto-generation, and therefore their only goal was to provide an abstraction layer between RTL signals and our transaction objects.

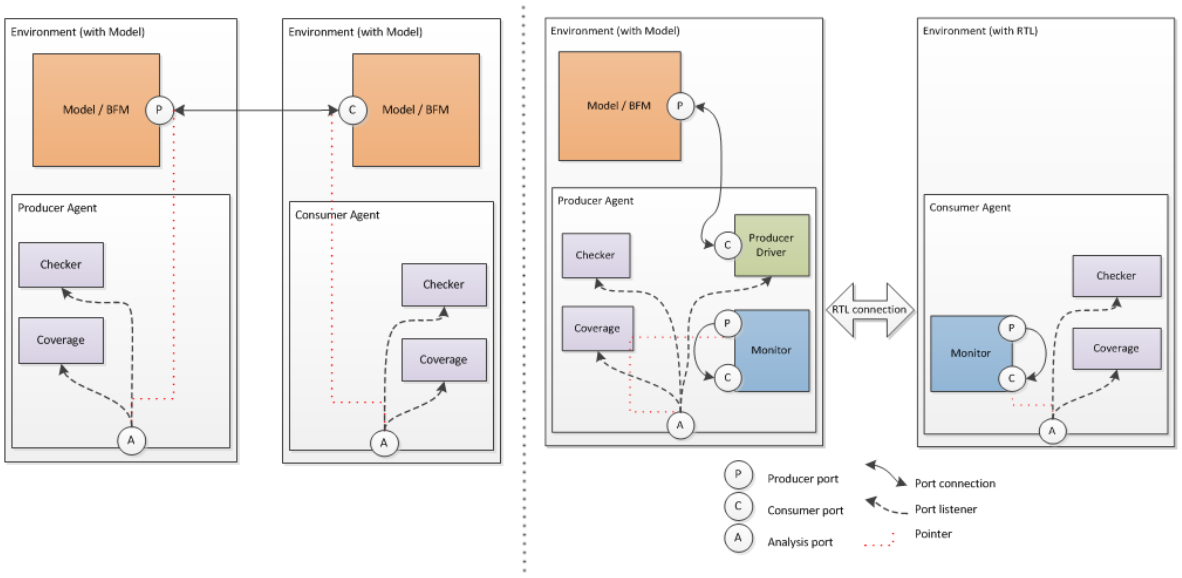

Figure 2. On the left, two agents are configured for direct model-to-model communication. On the right, a model-to-RTL configuration over RTL interface is shown. In both cases, checker and coverage are not aware of the actual configuration, and simply subscribe to transactions coming from the analysis port.

Monitors sampled signals and packaged them up into transaction objects, while drivers took those same transaction objects, and drove RTL signals. When no RTL was present, the agents bypassed drivers and monitors and passed transactions directly to each other, as shown in Fig. 2. In this setup, the other components inside the agents, as well the ones that listened to their analysis ports, were oblivious to the difference in configuration, abstracting the idea of model vs. RTL from the rest of the testbench.

The scripts we developed use a YAML[1]-based interface spec to automatically generate monitors, drivers, UVM agents with basic support for checkers and interface coverage, SystemVerilog interfaces with modports, configuration classes to hold virtual interfaces, and, finally, top-level stitching files. Many changes to signal names, widths, additions and removal of signals now became a very quick and easy task that can be performed even by RTL engineers making the RTL change. Figs. 3 and 4 illustrate a simple example of our YAML spec and one of the generated files. The YAML file in Fig. 3 lists the RTL signals involved in driving a transaction, along with transaction class accessor functions needed by the driver to retrieve the values to place on those signals. The generated driver code in Fig. 4 then makes the connection between the signals in the appropriate clocking block and the values in the transaction object.

```
intf: l2ls_rd_resp
blocks: [l2, ls]
req_type: arm_txn_l2ls_rd_resp
req_monitor: default
coverage: yes
signals:
- +%l2_ls_valid / valid()
- +l2_ls_id[4:0] / req_id()
- +l2_ls_qw_en[1:0] / qw_en()
- +l2_ls_addr[5:5] / addr5()
- +l2_ls_qvalid / q_valid()
- +l2_ls_data_beats / data_beats()
```

Figure 3. A sample of YAML code defining the read-response interface between L2 and Load/Store unit

[1] YAML is a simple data-serialization language, similar to JSON, and XML to a degree. One big advantage of YAML is that it is very human-readable, so specs written in YAML can often serve as documentation, as well.

```
// Drive a request
task arm_driver_l2_l2ls_rd_resp::drive(arm_txn_l2ls_rd_resp l2ls_rd_resp_req);
  mp.l2_cb.l2_txnid <= req.id();
  mp.l2_cb.l2_ls_valid <= 1'b1;
  mp.l2_cb.l2_ls_id[4:0] <= l2ls_rd_resp_req.req_id();
  mp.l2_cb.l2_ls_qw_en[1:0] <= l2ls_rd_resp_req.qw_en();
  mp.l2_cb.l2_ls_addr[5:5] <= l2ls_rd_resp_req.addr5();
  mp.l2_cb.l2_ls_qvalid[1:0] <= l2ls_rd_resp_req.qw_en();
  mp.l2_cb.l2_ls_addr[5:5] <= l2ls_rd_resp_req.addr5();
  mp.l2_cb.l2_ls_data_beats <= l2ls_rd_resp_req.data_beats();
endtask
```

Figure 4. A portion of the SystemVerilog driver generated from the YAML spec

Having working RTL interfaces early on allowed us to develop and test SVA assertions on all major interfaces. This provided a level of checking for our models, ensuring that they behave according to the expectations of the interface specifications. The other benefit was that some very complex assertions were tested and refined before they were used to verify RTL in both our simulation and formal environments.

The auto-generation of the interface components and other related classes was a major shift left for our verification effort. Hand-written code started at transaction-level behavior, and we rapidly developed major building blocks of the testbench.

E. Zero-Delay Mode

A common problem in testbenches designed with configurable model, rather than RTL implementation only, is that stimulus often has to travel through one or more models before reaching the DUT, which significantly reduces the ability to write high-quality reactive stimulus – i.e. scenarios in which stimulus reacted to an event inside (or on the interface to) the DUT. If that reaction took many clock cycles to reach the DUT, it was often no longer relevant. Similarly, if the model consumed a lot of clock cycles to process transactions, the throughput of stimulus reaching the DUT was negatively impacted.

In order to deal with this problem, we have come up with a novel approach to interfaces and clocking that allowed us to have both interfaces and models working in a "zero delay" mode. In this mode, communication between and within models took no real simulation time, while still having a virtual clock to all of these components to ease sequencing of events. To do this, we have implemented a central scheduler that has an inner loop which takes un-timed delta steps, and an outer loop which advances time on a normal clock when models have settled down. This is very similar to SystemC's approach to event scheduling [4] shown in Fig. 5.

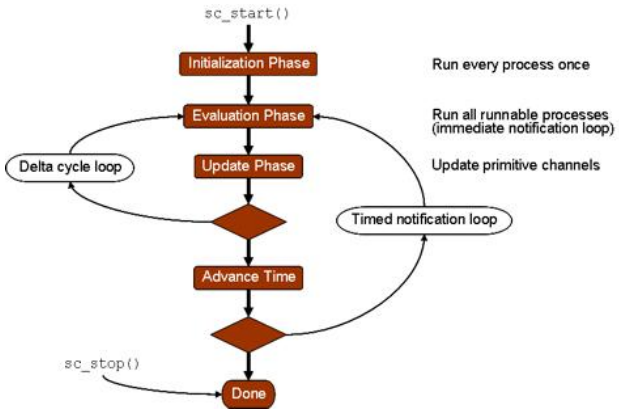

Figure 5 (Credit: Accellera Systems Initiative[4]). SystemC scheduler with an un-timed delta cycle loop inside a timed notification loop.

Our auto-generated agents can be configured to work in model-to-model zero delay mode, model-to-model communication over real RTL (used for testing RTL interface implementation, as well as SVAs that watch over those signals), and any combination of model and RTL connections, including RTL-to-RTL. This concept is self-contained within agents through agent configurations, allowing the rest of the testbench to function the same way in any of these modes. This flexibility was the key to isolating high-level code from low-level configuration of the testbench, allowing us to run the same stimulus and checks on many different combinations of models and RTL.

F. Scoreboards and High-level Checkers

In previous projects, the unit-level testbenches had very limited functionality early on as RTL was being brought up. In such a setup, most early checking was done by low-level micro-architectural checks that ensure that basic RTL features worked. The big downside of such checkers was that they were closely tied to the RTL implementation, which was bound to change dramatically at that point in the project. Such checkers start failing on most RTL changes, frustrating design engineers, and requiring significant effort from verification engineers to keep up.

To address this problem fully, we approached it from two directions. First, we initially focused on checking only the architectural rules. Since our testbench was a fully functional (and well tested) multi-core CPU memory subsystem, we could focus our efforts on writing high-level checkers, and even complex self-checking tests which verify well-defined features of the design. For example, cache coherence and consistency checkers, read-value checkers, device/strongly-ordered memory type ordering checkers, various synchronization algorithms, and so on, ensure that ARM Architecture Reference Manual rules are followed, no matter what the implementation is. These checkers were then tested by running on our prototypes, and testbench bugs were fixed well before any RTL needed to be tested.

The second part of the approach was to develop another layer of abstraction to allow checkers to be written at the highest level possible. For this, we developed components that we called scoreboards, although they performed no checking that some scoreboards do. The goal of our scoreboards was to simply "keep score" of what is going on in the design. The scoreboards would listen to analysis ports on several interfaces, and track transactions and design states across interfaces, while publishing even higher level transactions, which we called *events* for consumption by checkers. Once the scoreboards were written and well tested, they provided a vehicle for swift development of simple, yet powerful, checkers.

The benefits were significant. As RTL was written, these checkers were consistently finding RTL bugs with very few testbench bugs (i.e. false fails) reported in regressions. As RTL bugs were fixed, micro-architecture refined, and whole modules rewritten, the checkers remained the same, still checking the same things, and still finding new bugs. This instant and accurate feedback on the state of the RTL encouraged design engineers to run their own regressions and debug as they write new code, freeing up valuable verification engineer time to write new stimulus and other testbench components. The team efficiency improved dramatically, as we disconnected the common dependency between an RTL designer and a verification engineer (Fig. 6).

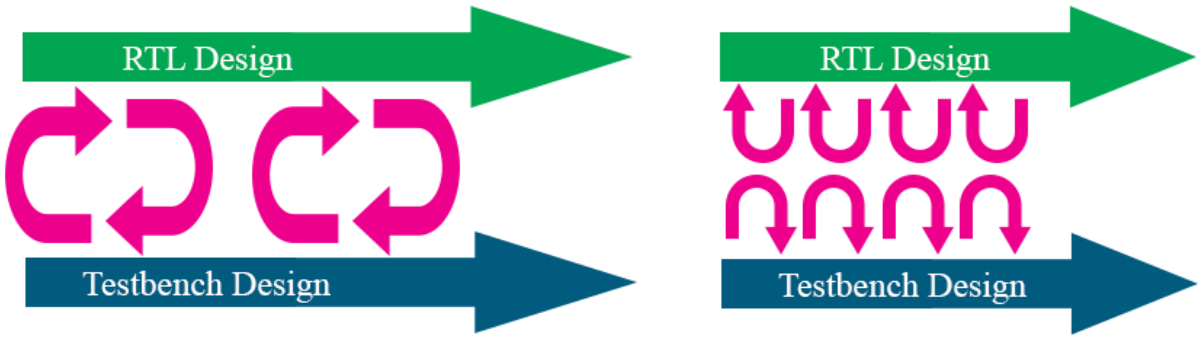

Figure 6. The common loop on the left is wide and slow, requiring interaction between designer and verification engineer on every feature. The model on the right has frequent and tight loops where, during feature bring-up phase, each engineer can test their own code as they write it.

As RTL settled down later in the project, we have started adding in micro-architectural checkers to tighten the verification process, but these high-level checkers are still in place as a "catch all" for any issues that are not detected properly by new low-level checkers.

III. UNIT TESTING

A. *SVUnit*

To further test our testbench components, particularly checkers, we used the unit testing approach using SVUnit [5]. Unit testing is a software development methodology in which individual pieces of code, i.e. units, are tested independently from the rest of the system. Small environments are created to trick the unit into thinking that it is running in the full system, and simple tests are written to directly stimulate the code in ways that are much more difficult in the full system. A checker, in this case, becomes the DUT, and tests are written to exercise the checker and compare its results to the expected values.

There are several benefits to this approach. First, it allows us to find testbench bugs early, before the full system is functional. Second, it makes it easy to test known corner cases in the code that often require hours of random simulations to reach. And, finally, the unit setup is much simpler than the full testbench, which translates into easy debugging and very quick compile and run times.

All popular programming languages have one or more unit testing frameworks available [6]. One available for SystemVerilog is called SVUnit, which is an open-source framework developed by Neil Johnson. SVUnit is a light-weight, easy-to-use framework that provides the basics needed to quickly set up a mock environment, and define, compile, and run tests. It also supports UVM, which makes testing UVM components a breeze.

Early in the project, we used SVUnit to develop unit tests for many of our base classes. For example, we have a set of classes that deal with addresses and address ranges, which have some tricky overlap and overflow end-cases that are easy to get wrong. Writing unit-tests quickly uncovered some issues, giving us more confidence that base classes will be bug-free when used in the full testbench. Fig. 7 shows a sample base-class unit test.

```
`SVTEST(overlaps)
    arm_addr_range test;
    arm_addr test_min, test_max;

    // 10 possible cases, assuming that the current range
    // doesn't start at 0 or end at max
    if (m_min_val > 0) begin

      if (m_min_val > 1) begin

        // completely before
        test = create_test_range(m_min_val-2, m_min_val-1);
        test_overlap(my_arm_addr_range, test, 0);
      end

      // meets before
      test = create_test_range(m_min_val-1, m_min_val);
      test_overlap(my_arm_addr_range, test, 1);

      // overlaps
      test = create_test_range(m_min_val-1, m_min_val+1);
      test_overlap(my_arm_addr_range, test, 1);

    end

    // starts
    test = create_test_range(m_min_val, m_min_val+1);
    test_overlap(my_arm_addr_range, test, 1);

   // and so on…
  `SVTEST_END
```

Figure 7. An example of a base-class unit test in SVUnit, which tests all possible overlap scenarios between two address ranges.

B. *Checker testing*

Checkers are notoriously difficult to test, because false passes are very likely to go on unnoticed. In my experience, most false passes are detected by accident. For example, when illegal stimulus is inadvertently created and then subsequently noticed, investigation into "why was this allowed to pass?" yields a bug in the checker. Attempts of error-injection into design have been made [7] to combat this problem, but with success in finding only certain kinds of checker bugs.

Unit testing through SVUnit allowed us to deal with this problem by feeding faulty stimulus directly into the checker, and confirming that it flags it as expected. With SVUnit's ability to intercept UVM fatal/error messages and confirm that they happen when we expected them to, we were able to assert that complex checks were failing when, and only when, expected. During a typical "why did we not catch this" bug analysis, there are few things more disappointing than the discovery that stimulus properly exposed a bug in RTL but a faulty checker never fired. This methodology allowed us to close this hole in the verification process.

Figs. 8 and 9 show the code samples of unit tests for a particularly difficult checker, a multi-core read-value checker which predicts correct read data in a coherent multi-core CPU system. The Write-After-Write (WAW) test shows the simplicity and the power of this setup. This kind of failure might not happen very often, if at all, in the full testbench, and we might never know whether the checker correctly recognizes this problem if all we ran were normal simulations. However, in only a dozen lines of code, we exercised our checker and confirmed that WAW case is correctly detected and reported.

```
// Test two stores, and bad data for overlapping load
`SVTEST(two_stores_bad_overlapping_load)
  longint unsigned addr0 = 0;
  longint unsigned addr1 = addr0 + 1;
  longint unsigned addr2 = addr0 + 2;
  int load_id;

  api.store(cxu0, addr0, .data('h5544), .size(2));
  api.store(cxu0, addr2, .data('haa98), .size(2));

  load_id = api.load(cxu0, addr1, .size(2));
  uvm_report_mock::expect_fatal(); // wrong data
  api.load_result(load_id, .data('h9876));
  `FAIL_IF(!uvm_report_mock::verify_complete());

`SVTEST_END
```

Figure 8. A simple test for load data satisfied from multiple stores.

```
// Test WAW on two cpus
`SVTEST(waw_2cxus)
  longint unsigned addr = 0;
  int load_id;


  api.store(cxu0, addr, .data('h55), .size(1));
  api.store(cxu0, addr, .data('haa), .size(1));

  load_id = api.load(cxu1, addr, .size(1));
  api.load_result(load_id, .data('haa));           // read younger data
  `FAIL_IF(!uvm_report_mock::verify_complete());   // make sure no fails reported

  api.load(cxu1, addr, .size(1));
  uvm_report_mock::expect_fatal();
  api.load_result(load_id, .data('h55));           // can't read older data
  `FAIL_IF(!uvm_report_mock::verify_complete());

`SVTEST_END
```

Figure 9. A Write-After-Write (WAW) hazard test. Once the younger data has been observed by a load, the subsequent loads can not observe the older store.

IV. EFFORT & RESULTS

The methodology described in this paper naturally requires some additional investment into early-project development, but the total increase is relatively minor and pays back many times over during the length of the full project. As Table 1 shows, the amount of code written for the virtual prototypes added slightly over 10%[2] to the time spent on writing new code. However, with research [8] showing that verification engineers spend only 22% of their time developing and debugging testbenches, we are looking at an increase of only about 2% to the overall verification effort.

As the famous quote by Brian Kernighan says, "Debugging is twice as hard as writing code in the first place", the same research shows that verification engineers will spend 61% of their time developing, running, and debugging tests. Any improvements in this area will easily offset the 2% overall increase due to the new code written.

It is hard to estimate the amount of efficiency gained by being able to debug your code as soon as it is written, but the advantage is intuitive to experienced engineers. An essay by Paul Graham [9] on the importance of being able to hold your program in your head talks about the time spent trying to "load the program into your head". This is the penalty that we have to pay every time we dig into debugging old code, and he estimates that about half-an-hour is lost every time this happens. Considering that the break-even point for the project is an efficiency gain of only 4% (as 4% of 61% of time spent on testing and debugging is over the invested 2%), my feeling is that we have exceeded that by a very large margin.

	Lines of Code
Full testbench	190 K
Virtual Prototypes	30 K
Traditional BFMs on previous projects	10 K

Table 1. The amount of code written for virtual prototypes vs. the rest of the testbench. If Virtual Prototypes were not developed, traditional BFMs would have been required to drive stimulus instead. The estimate for traditional BFMs comes from a previous project.

The debugging efficiency gains and the testing of the RTL code as soon as it is written allowed us to also set high standards for RTL quality throughout the project. While previous projects have focused on keeping basic functionality going during the development process and accepted regression pass rates in the 80-90% range until development is done, we have been able to consistently keep pass rates at 99.8% or higher, with significantly higher quality of stimulus, and without any measurable impact to the early project milestones. This will allow us to significantly pull in some of the later project milestones, as we save the months of time typically spent on getting pass rates up to the target goals.

An additional benefit of having high-quality RTL throughout the project is the ability to tweak and improve stimulus and get quick feedback on its quality through bug rates. In an environment with low pass rates, where bugs are very easy to find, no meaningful quality feedback can be extracted from wild failure rate swings. Similarly, in late stages of the project bugs are hard to find, and real improvements to stimulus can not be measured until several days of regressions are run. The sweet spot, where easy-to-find bugs are gone, yet real improvement to stimulus can yield new bugs the next morning is the perfect place to be when optimizing stimulus quality.

V. SUMMARY

Modern testbenches are complex pieces of software, and like any other software, they are full of bugs the first time they are written. In order to improve the efficiency of verification engineers, we have developed a testbench structure that allowed us to develop and test verification code without the need for working RTL. The incremental effort in development was offset many times over through efficiency gains in debugging and writing tests, which is where most of engineers' time is spent. Having high-quality testbenches and RTL earlier in the project will allow us to pull in later project milestones, completing the project earlier compared to traditional methods.

[2] (30K-10K)/190K ~= 10% of additional code over traditional BFMs which would have been needed otherwise

REFERENCES

[1] Steve McConnell, “Code Complete 2”, 2004, p. 521

[2] TWI, “FAQ: What is Virtual Prototyping?”, http://www.twi-global.com/technical-knowledge/faqs/process-faqs/faq-what-is-virtual-prototyping/

[3] Nithya Ruff, Synopsys, “The Shift Left: how virtual prototyping reduces risk”, http://www.techdesignforums.com/practice/technique/virtual-prototyping-cuts-risk/, 2012

[4] Accelera Systems Initiative, “SystemC Resources Page”, http://oopsproject.info/systemc.php

[5] Neil Johnson, “SVUnit”, http://www.agilesoc.com/open-source-projects/svunit/

[6] Wikipedia, “List of Unit Testing Frameworks”, https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

[7] Satwinder Singh, Infineon Technologies, “Using Certitude for Relative Functional Qualification of a Re-usable Testbench”, 2014

[8] Mentor Graphics, “2014 Wilson Research Group Functional Verification Study”, Part 8, https://blogs.mentor.com/verificationhorizons/blog/2015/07/13/part-8-the-2014-wilson-research-group-functional-verification-study/

[9] Paul Graham, “Holding a Program in One’s Head”, http://www.paulgraham.com/head.html, 2007