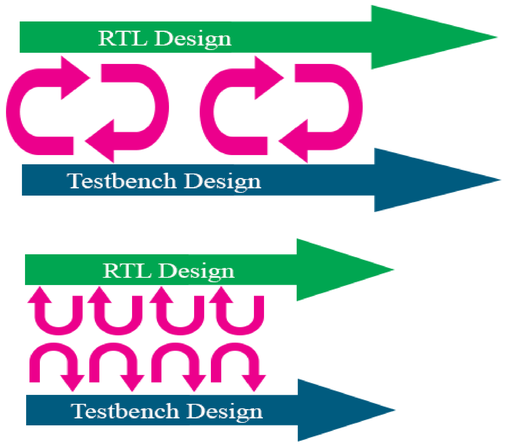
Testing the Testbench

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

The Problem

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.

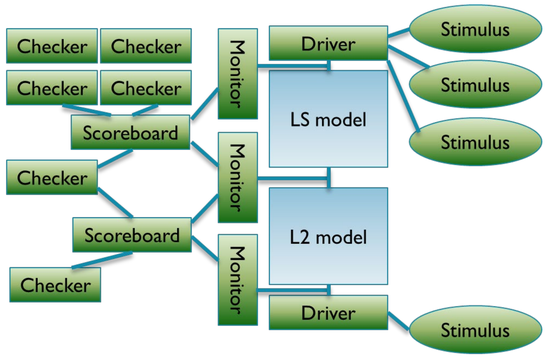
Shift Left



Decoupling testbench design from RTL design tightens the development/feedback loop, increasing efficiency, producing better results, sooner in the project schedule.

Virtual Prototyping

A variant of the Virtual Prototyping methodology 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.



All components in green can be fully developed, tested, and thoroughly stressed with models for LS and L2 blocks in place, without any need for working RTL.

Models are written to communicate over real RTL interfaces. With that in place we can build:

1. **Monitors**, to produce transactions seen on interfaces
2. **Drivers,** to wiggle the pins on interfaces
3. **Scoreboards**, to track transactions across interfaces
4. **Checkers**, to check both low-level interface and high-level block behaviour
5. **Stimulus**, to exercise all of the components above

At this point, most of the testbench is up and running, without a line of real RTL!

Unit Testing with SVUnit

Checkers are notoriously difficult to test, because false passes are very likely to go on unnoticed. 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.

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.**

```
// 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));
    api.store(cxu0, addr2, .data('haa98));

    load_id = api.load(cxu0, addr1, .size(2));

    // wrong data
    uvm_report_mock::expect_fatal();
    api.load_result(load_id, .data('h9876));

    `FAIL_IF(
        !uvm_report_mock::verify_complete());
`SVTEST_END
```

SVUnit tests are very quick to write, and easy to maintain. In just a few lines of code above, we've verified that our checker catches bad load data due to overlapping stores being mishandled.

Effort

The amount of code written for the virtual prototypes added slightly over 10% to the time spent on writing new code. However, with research [2] 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.***

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

Conclusion

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 allowed us to pull in later project milestones, completing the project earlier compared to traditional methods.**

References

[1] Steve McConnell, "Code Complete 2", 2004, p. 521
[2] Mentor Graphics, "2014 Wilson Research Group Functional Verification Study", Part 8