Simple & Rapid Design Verification using SystemVerilog Testbench on Intel's Next-Generation Microprocessor

Thomas Alsop
Intel Corp
503 712-9055
thomas.r.alsop@intel.com

Wayne Clift
Intel Corp
503 712-3251
wayne.clift@intel.com

Luke Hood
Intel Corp
503 712-2945
luke.hood@intel.com

Jeff Gray
Intel Corp
503 712-3529
jeff.gray@intel.com

ABSTRACT

SystemVerilog Test Bench (SVTB) is a set of language extensions to the IEEE 1800 SV LRM used to reduce the amount of time and effort required to write tests which exercise SystemVerilog (SV) RTL code. Design Verification or more correctly defined "Design Exercise" is a methodology in which pre-defined basic boundary conditions of a design must be tested before submitting code to the project's official codebase. This methodology was used on the next-generation Intel CPU project to reduce the number of bugs introduced to the RTL model **by over 50%**. SVTB for Design Exercise was the utilization of SVTB language extensions in conjunction with a test environment (TE) automatically created by the RTL tools for any level of hierarchy in the CPU project. SVTB was used predominately to fill any validation gaps where the "official" non-SVTB TE was not available for designers or where the validation TE collateral lagged behind the RTL coding milestones. SVTB enabled designers to quickly and easily write directed tests within a highly controllable and observable environment. SVTB also enabled rapid debug turnaround time, from test or RTL change, through compilation, simulation, and wave trace updates, typically 1-5 minutes. SVTB TE's also allowed for extensive randomized and constrained verification environments to be turned into models as regression test suites.

INTRODUCTION

Design RTL bugs on Intel CPU projects have increased significantly in the past decade, much more rapidly than previous trends had indicated, even accounting for project design complexity increase. Some of the more recent CPU projects reported record thousands of pre-silicon RTL bugs. Without decisive action to reduce RTL bug rates, next generation CPU teams would have faced critical RTL execution issues.

Recent CPU RTL postmortems identified that one of their most significant sources of RTL bugs were caused by the general lack of early (pre-code-release) RTL testing. These CPU teams faced a transition to a brand new design language (SystemVerilog), a new SV-based tool suite, and a new code development work model. The transition to this new work environment consumed all available bandwidth and prevented any serious focus on early "***Design Exercise***".

"Design Exercise" is not a new concept in the industry or even within the CPU design teams themselves. It is *the methodology of completing all gating exercise plan items before releasing new code to the official project released models.* It does not replace the true Validation process done by Validation teams. "***Validation"*** is an exhaustive process of testing a design to ensure that it functions precisely to specification covering all functional boundary conditions and all design usage modes. "***Exercise***" is a subset of the total Validation effort. It is an initial breadth-first testing approach applied to a new feature that provides confidence in the basic functionality of that feature, before more extensive testing is performed. In addition, exercise activities should be done at the lowest hierarchy level at which test environment support is available with sufficient controllability of the new feature. "Test your own code before you inflict it on others" is a basic principle of sound engineering practice.

Before moving to this newer SV design language and infrastructure, previous CPU designs, extensive tools and methodology were developed for local testing of new features. A proprietary PERL-based Test Environment (TE) supported local RTL testing. This proprietary TE utilized a set of shared library functions and RTL model access techniques, combined with custom user code, which allowed the user a familiar language (PERL) in which to quickly develop custom test environments at many hierarchy levels. Because PERL was well known to the Design teams at Intel, there was very little barrier to entry for an RTL coder to ramp up and do effective design exercise work.

When the CPU design teams moved to SV to model their hardware they lost the use of the PERL-base TE. Their validation team made the decision to not use SVTB for their TE's and chose instead a language that was well ahead of SVTB, at the time, in terms of complexity and usage. But because it was so significantly different from any language in the Designer's experience, it posed a tremendous barrier to entry. This, combined with other factors, led the CPU teams to use sub-standard code-release policies where designers would turn the code in without exercising the new feature at all, or they would stick it into a side model and wait for the validation team to get to it later (sometimes months) before it was turned in to a model. In both cases many bugs were uncovered causing the overhead of tracking the bug and/or having the designer re-learn the code he wrote month earlier.

There were some notable exceptions on these projects where the Validation TE's were highly effective and bucked the trend. These exceptions either gave designers the templates they would need for their exercise work or some limited use of native SV test benches (TB) were created. But at least in these more recent CPU teams, these were the exceptions to the Design Exercise paradigm. Most designers did little or no design exercise and relied on their validation team to find bugs.

Analysis of the more recent CPU teams Design Exercise experiences lead to the following mandates to be applied to next generation CPU RTL Development:

1. Release no code feature to the project repository until it has been sufficiently/quantifiably exercised.
2. Work efficiently. Take full advantage of existing Test Environment capabilities. If the required TE features are available, learn to use them to complete Design Exercise.

3. If the project TE is not yet available or does not provide sufficient controllability, consider completing Design Exercise using a custom test environment.

Within this context, several SVTB pilots were launched to experiment with SVTB and to help in the development of several supporting tools. Three units of different size and complexity were chosen for experimentation. Ultimately, the next generation CPU projects adopted SVTB for use during Design Exercise, for cases when the alternative validation environment was not ready to use or was inadequate. These CPU teams developed infrastructure tools to generate SVTB wrapper shells for any module, developed methodology and training material to be delivered to the design team, and deployed SVTB. This paper will discuss details of the tools, results from pioneering experiments, and learnings from practical usage of the SVTB flow in CPU Frontend RTL Development.

THE SVTB SOLUTION

SVTB is both an easy-to-use and powerful mechanism for writing directed tests while also being very extensible to a highly random, highly encapsulated, and fully layered verification environment. It brings many of the OOP (Object Oriented Programming) concepts of C++ into the verification world of hardware description languages (HDL), specifically targeting Verilog extensions. SVTB consists of a significant number of extensions that are now part of the IEEE 1800 SystemVerilog specification which allows for more abstract constructs to be used for the sole purpose of verification. It brings much of the 30+ years of software verification solutions like OOP into the hardware verification realm in order to reduce costs, increase productivity, and increase reuse. While it is relatively new, it is gaining acceptance in the industry as the next verification language of choice among the EDA vendors and user companies.

While there are many aspects of the language which are powerful, there are many pieces of it which still need improvement and therefore it does lag some of the EDA's solutions which have been around longer. Some of the language descriptions are vague and have left open the interpretation of various features of the language. This has lead to different solutions by different vendors' often leading to code which does not compile or behave the same from one EDA simulator to another. However, the industry is rallying around the spec and the SV community continues to pour effort into language clarification and improvement.

Once the CPU teams decided to pilot SVTB, many questions had to be answered. What would the capabilities be? How much ramp time would be required? Would there be maintenance costs? How much reuse would be available? What were the limitations? How much automation could be implemented?

Before any pilots were initiated, the overall language was reviewed along with industry standards. The two common verification methodologies in the industry were the Verification Methodology Manual (VMM)[1] and the Open Verification Methodology (OVM™)[2]. Both methodologies are similar in scope and capabilities but their complexity and comprehension levels were far beyond the needs of a simple design exercise environment. This led us to believe that all of the above questions had to be answered with a proprietary SVTB solution. Once the capabilities and limitations were understood through pilots, a working group (WG) was formed to develop a methodology around the technology, to enable automated creation of a test environment, and to deploy the system and train the design team.

THE EXTENDED ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (XAPIC) PILOT

The first pilot was conducted by Intel's Corporate Design Solutions (CDS) on the xAPIC, a sub-unit of the memory cluster. The main job of the xAPIC is to process and prioritize interrupts, initiate handler routines, and to track all interrupts to completion. The primary goal of the pilot was to see if the language constructs would meet the needs of a simple design TE. The secondary goal was to understand the capabilities and limitations in order to develop a methodology. Several key learning's about SVTB were drawn from this pilot:

Intuitive: The first key learning highlights how intuitive it was to move from SV to SVTB. Designers that have worked with SV understand the framework of the RTL environment: (a) the way hierarchy is created, (b) the concepts of template definitions and instantiation, (c) parameterization and macro usage, and (d) general syntax and semantics. This framework does not change for SVTB. Instead it builds on it, adding constructs and semantics that enable a TE to be created. In fact the vast majority of the constructs used for RTL coding can be used in the TB.

Simple: The second key learning realized how simple it was to create the TE and write your first test. Figure 1, shows the code that drives a set of interface signals over time. This is the only code a test writer needs to understand to write his first test. It consists of driving the interface with a hardcoded value, stepping one clock cycle, driving the next interface signal, and so on until a complete transaction is done.

```
     m.crpwrup    <= 1'b1;
##1; m.crwritebus <= 16'b0010_0100_1000_0000;
##1; m.crwritebus <= 16'h0000;
##1; m.crwritebus <= 16'h0000;
     m.crpwrup    <= 1'b0;
```

Figure 2 – First Test Pounding the Interface

Reusable: Third learning, taking the interface pounding example and encapsulating it into a class was very intuitive as well for SV designers. The code in Figure 2 shows how to take the code from Figure 1 and encapsulate it into a class. Within the class is a method, in the form of a task, called 'Write', which has both a port list and functionality. The port list provides a means by which we can give meaningful names to the data that we are driving. In the port list we can specify default assignments, like CRThread and CRRead, and then leave these assignments out of the

```
class ControlRegisterBus;
 task automatic WR(
        input logic [1:0]  CRCtrl,
        input logic        CRThread = 1'b1,
        input logic [10:0] CRAddr,
        input logic        CRRead = 1'b0,
        input logic [31:0] CRData);

       m.crpwrup    = 1'b1;
  ##1; m.crwritebus =  {CRCtrl, 1'b1,CRRead,
                        CRThread, CRAddr};
  ##1; m.crwritebus = CRData[15:0];
  ##1; m.crwritebus = CRData[31:16];
       m.crpwrup    = 1'b0;
 endtask
```

Figure 1 - First Class

method call shown in Figure 3.

With the class created, the test writer has to create a handle of that class, construct the object, and then call the class method via the handle within

```
CRB.WR(.CRCtrl   (2'b00),
       .CRAddr   (APIC_ID_CR_ADDR),
       .CRData   (32'h0100_0100));
```

Figure 5 - First Method

the test as shown in Figure 3**.** The 'Write' method can now be used over and over again in the test by substituting the CR address and CR data with different values. With this first level of abstraction in place, writing directed tests now becomes very strait forward and intuitive. Additional classes and methods are also created to model other transactions.

Fast: Another very important key learning with this pilot was the incredible turn-around time. Compilation AND simulation time was on average 20 seconds wall time, and simulation speed for this medium-sized DUT was 35kHz. This is compared to test times for upper level test environments which ran at 10-100Hz.

Controllability: Another pilot was conducted on the Instruction Fetch Unit (IFU) which had similar results to the xAPIC. One additional key learning from that pilot was the level of controllability given to the designer to hit specific conditions in their DUT. This designer was able to reproduce a bug found in silicon with SVTB in less than a day that was never found due to it's complexity at the upper test environment level.

METHODOLOGY DEVELOPMENT

The next step for SVTB was to formalize a methodology. The pilots gave us a direction but now the language needed to be understood in depth by a good spectrum of RTL designers to enable the design team to form the best methods to use SVTB for the entire project. The formation of a methodology would ensure that productive practices were created and taught, that an automated TE would adhere to them, and that future linting technology had a baseline set of rules and guidelines to enforce.

CDS formed an SVTB working group (WG) with CPU engineers and methodology experts, which was given the charter to develop a working methodology for the use of SVTB on the next generation CPU project. This four-month effort led to the development of CPU Design Handbook Guidelines detailing the SVTB aspects of the language and how they should be used in order to accommodate a simple and light TE for designers. Some examples of language features described in the CPU SVTB methodology include:

Program Blocks are required: Simply stated, program blocks in conjunction with the use of clocking blocks ensure that race conditions are eliminated from the TE. The SV language defines regions when code is evaluated with respect to each other and program blocks are evaluated once all the modules (DUTs) are completely relaxed. This, in combination with clocking blocks, ensures that race conditions between the RTL and the TE, a very problematic issue for Verilog users, are rarely seen. For this reason, Program Blocks are required in the SVTB TE.

Clocking Blocks are required: Another very important methodology choice is to use Clocking Blocks (CB). CB's serve three functions. First, they eliminate race conditions. Second, they abstract away timing complexities. Third, they synchronize the passing of data between the TE to the DUT on specific event edges.

Sample at #0 and Drive at #2: Another key decision taken by the WG was to model the sample and drive times through our CB's on the #0

```
clocking cb @(posedge clock);
    input  #0 Q; // Sample exactly at event edge
    output #2 D; // Drive 2 ticks after event edge
    …
endclocking
```

Figure 4 - Default #0 Sample and #2 Drive

timing edge and #2 timing edge respectively. Sampling input signals from the DUT at #0 lets the TB see all the events from the DUT after they have been triggered by the clocking edge and after the DUT has completely relaxed. The TB can then accurately respond and drive back the correct stimulus. By choosing to drive our events back to the DUT at #2 we ensure that the TB stimulus will also be the last driver of the DUT signals.

Combinational Logic Modeling: Understanding CBs made us realize that there was no way to model combinational paths through a TB within them. Most validation TE's don't have to worry about modeling combinational logic across the DUT/TB interface as they exist at very high levels of abstraction. This is not the case for Design Exercise. To model combinational logic, CB's cannot be used and the TE must explicitly drive or sample the signals directly. This removes the latency of waiting for the CB event and allows them to be immediately seen and driven back in the same time tick.

Class and Task Usage: Another methodology choice was the use of classes to encapsulate logic for reuse as seen in Figure 3**.** Classes allowed the designer to create complex structures and transactions which can be parameterized and reused. Monitors, scoreboards, transactions, generators, and drivers can all be created generically with the use of classes. While tasks would accomplish some of the same functionality as the class, they did not allow for compartmentalization and protection of class properties. Also classes have built in randomization functionality supported by the language, which cannot be done with tasks.

Logging Macros: CPU projects at Intel have very strict policies on the content conveyed in log messages. This content included which line of code was giving the message, which file it was from, what time step it occurred, what hierarchy (%m) the message came from, and of course the specific message string. Therefore different logging methods were looked at from the OVM™ and the VMM and it was decided to use a pruned-down version of what the OVM™ does, namely using macros as

```
`define ERROR(str) \
  $display("[%t] at line %0d in %0s",$time,$line,$file);\
  $display("ERROR: %m - %s", str);
```

Figure 3 - Logging Error Macro

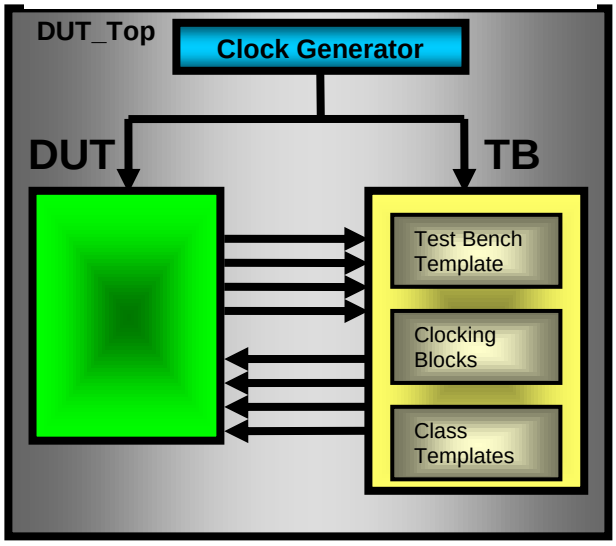
seen in Figure 5.

AUTOMATION

We wanted to create a test environment conducive for RTL designers so our focus was on these criteria:

1. Give the designer the ability to choose any hierarchy for the DUT TE
2. Enable the designer to write the test immediately
3. Allow for debug turn-around in minutes

Figure 7 - Top Level Test Environment



The automation was done within the RTL build environment of the CPU project. Two additional stages were added to the build flow to first create a TE, and then to build the TE into a simulation executable.

The create stage generates the top level TE seen in Figure 6. This includes the DUT, the TB, the clock generator, and the top level connectivity.

The create stage is only run once in the build flow. The user is encouraged to compile this new code at least once before any edits to verify that the TE compiles 'out of the box'. Once the TE has been created, the user can then begin to make changes to the collateral and start writing tests. Once the first test is written, the code is compiled, simulated, and loaded into a debugger. The debug loop then becomes a very simple process of modifying the RTL or TB, recompiling, re-simulating, and reloading the results (see Figure 7).

Automation Limitations

There were several issues that we ran into with the automation as we didn't want to burden the tools with too much complexity. This simplistic approach actually made the tools easy to maintain and use. The downside, however, led to extra effort on behalf of the RTL designers who for each of the following limitations had to work around them in one way or another.

Upper Level DUT compile required – In order to get the collateral in a cloned CPU model to create a SVTB TE, the build tools had to first compile an established block above the level of the SVTB DUT. This stemmed from the lack of grafting technology on the CPU models and from the inability to build any hierarchy level atomically.

Missing defines – Another common issue that designers ran into was not having their TE compile 'out of the box' due to either missing search paths in the simulation build configurations or missing defines which were scoped at the established upper TE levels but not as the new lower DUT level.

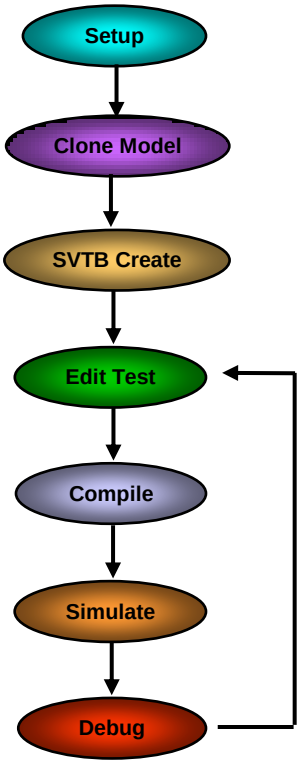


Figure 6 - Debug Flow

Reset and Configurations not automated – One of the first things needed within a TE, once the framework was created, was to create the reset and register configuration collateral which would simulate getting the TE and the DUT out of reset and into normal operation. This was not automated and therefore took effort to either drive the interfaces or drive internal registers to get to this state.

Next Generation CPU Usage

There were two usage models planned for the next generation CPU project, one in which the designer would create his TE, use it once, and throw it away once his code was exercised. The other was the opposite where the TE was created once and then forever maintained as tests were written and added to regression (test) lists.

The majority of the SVTB TE's created on the CPU project turned out to be the throw-away type. This work model is initially seen by most who hear about it as very wasteful, but the reality is just the opposite. The amount of effort to create the TE is very small, and the one time ramp cost to use it is also small. The TE automation enabled the designer to move to any model, recreate the TE, and test whatever is new on that model, sometimes by only driving the internal logic and signals they care about and observing via debug waveforms the results.

On the other end of the spectrum, there were SVTB TE's created and maintained by the designer because there was no existing TE. These fully capable TE's took full advantage of the SVTB language semantics and constructs and are discussed in the MLC and Gearbox sections below.

Finally, there were about 10 "middle of the spectrum" SVTB TE's created and turned into the RTL models. These were relatively simple TE's but still needed to ensure that new functionality did not break existing functionality. SVTB tests were created and checked into the test database, and the tests were incorporated into regression lists which were turned in with the model. Every turn-in thereafter ran these SVTB tests to make sure nothing was broken. A handful of TE's were surprisingly created at cluster and full chip (top) levels by Clock and DFX engineers who black-boxed the lower level RTL to test only their code. They used SVTB either because existing TE's did not support them and/or they wanted quick debug turn-around time.

One other very interesting SVTB implementation came about with the integration of new proprietary bus within the CPU project. The new bus had no validation environment for initial coding and mostly depended on OVM™ Verification IP provided and supported by another team within Intel. Integration issues aside, we ended up using the OVM™ code to create the bus agent and fabric TE's to validate the new RTL code. However, due to schedule, the agent code for several units did not exist yet, so SVTB was used in its place to drive the bus requests and monitor responses across the bus RTL channels. In essence, we literally had two atomic TE's, one based on OVM™ and the other on SVTB design exercise principles, working together with the RTL in the middle, resulting in over a dozen bugs being uncovered.

Mid-Level Cache

The Mid-Level Cache (MLC) functions as an intermediate cache for the CPU. It contains the queues, FIFOs and other structures to service memory requests from the lower level caches and performs cache lookups while managing data returns and protecting cache coherency.

The SVTB based TE for the MLC was created from scratch by an RTL designer familiar with the MLC Architecture and RTL as well as SV

itself, but having no SVTB experience. The main resources used to build the environment were based on learnings and discussions with the the CDS SVTB expert and System Verilog for Verification[3]. The MLC work also used some of the previous xAPIC SVTB Pilot code as templates and coding examples.

The scope of the MLC functionality was very large and startup costs were more than expected. These costs were spread over two areas: first, architecting and coding the main interface, and second, implementing the reset and configuration phases (i.e. getting the RTL to a known good post reset and power-good state). In both cases, the functionality and complexity contained within the MLC contributed to a longer than expected development time. Implementing the reset and configuration phases included driving multiple clock domains, redundancy/fuse poundings, power-good and reset sequencing, and TB cache initialization.

There were no significant startup costs ramping into SVTB or with TB development. Because the SVTB TE is built on the same SV framework used for RTL development, this meant that RTL designers very quickly engaged and started building the blocks needed for the TE. Functional TE blocks were developed in a matter of hours and focus on the actual development, rather than new language syntax and semantics. SVTB ramp time was spent understanding the new constructs, such as dynamic arrays, and how to use the language effectively by using OOP concepts.

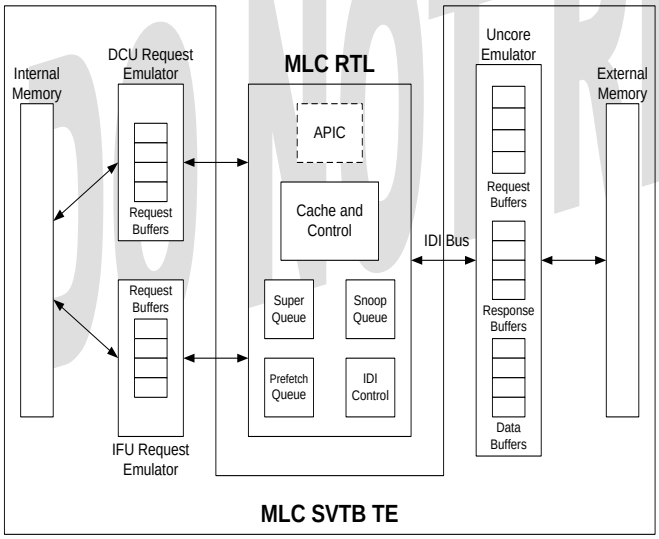


Figure 9 - MLC RTL and SVTB TE Overview

The MLC TE development effort included two main internal request emulators for the DCU and IFU, which shared access to internal memory (implemented as an associative array). An external bus emulator to represent the Uncore was also needed. The Uncore emulator had the external memory for the MLC TE, as well as Uncore request and response queues (implemented as associative arrays). Implementing an external bus emulator can be problematic if fine-grained control is

```
LLC.AddU2CRsp (.RspDelay(3),
               .RspMESI(NHM_MEU_STATE_I),
               .MatchReq(1'b1),
               .MatchSqid(4),
               .MatchAddr(40'h1234_5678));
```

Figure 10 - Response Matching Example

desired from the interface. Fortunately, this is one area in which the SVTB TE excelled. The SVTB TE allowed the TB writer to specify certain request and response matching conditions, such that individual requests could have very specific responses. The TB writer was also able to specify default cases for various response types and request/response combinations.

Once the model was up and running, it became apparent in the MLC that checker development was going to have a high cost. The higher level validation TE implemented a comprehensive data and coherency checker for the MLC, which would have taken many weeks of effort to duplicate in SVTB. Given time constraints, several lighter weight checking strategies were implemented, rather than a comprehensive validation solution.

The primary data and state checking for the MLC used internal and external memory implemented as associative arrays. Cache line data and line state were tracked with an external view and an internal view. When data was returned for core requests it was checked against the internal view. Similarly, when core write data was seen by the external bus emulator, it was checked against the external view.

A second, more lightweight form of checking was also implemented. In the lightweight check, the TB writer specified the expected data from a read-type memory transaction. The SVTB TE would compare the actual data returned with the test writer's expected data and signal an error if it did not match (see Figure 10).

```
DCU.AddRequest (.ReqType(NHM_DCU_WB_TYPE),
                .Addr(40'h1234_5678),
                .Thread(1),
                .WrData(InitData));
DCU.EmptyReqQ();
SB.FinishAllRequests();

DCU.AddRequest (.ReqType(NHM_DCU_READ_TYPE),
                .Addr(40'h1234_5678),
                .Thread(1),
                .ExpData(InitData));
```

Figure 8 - Rd/Wr Transaction with Lightweight Checking Example

One area where the SVTB TE far exceeded the higher level validation TE was in turnaround time. The extremely fast compile and run times created a work model where the designer could quickly iterate over changes (see Table 1). This mindset was good because it enabled more trial and error, but also allowed less well thought-out changes. Note that while the performance differences are significant between the upper level TE and SVTB, only a portion of this in Table 1 is the language difference. The 10 minute number includes the overhead of the build tools and recompiling the upper level TE code.

	Compile Time	Typical short test run time
SVTB TE	10 sec.	< 1 min.
Upper level TE	10 min.	5~6 min.

Table 1 – MLC TE Compile and Time Comparison

Maintaining and expanding the SVTB based TE is another area where SVTB really shined. This was due to language familiarity and the ease within SVTB to make changes. Any RTL designer could modify and add functionality to the TE, versus the upper level validation TE experience where there is only one expert for each TE.

We learned from this effort that if the RTL development begins from scratch, or with RTL that will undergo significant change, it would be much easier to develop the SVTB TE in parallel with the RTL. Such a situation allows for proportional effort to be directed into the TE and into the RTL, rather than having the TE require an order of magnitude more development effort.

Also, as seen by the MLC, one of the main benefits to using SVTB is the speed at which small pieces of TE code can become functional. Combined with the amazingly short compile and run times, and we can say that the biggest benefits to having an SVTB TE would come when the scope of the RTL is kept to a small level. The scope of the MLC required a large amount of effort to be put into TE development. If this scope could have been kept at a lower level, there would have been more benefit to having the SVTB TE.

GEARBOX

The Gearbox was a new microarchitectural unit in the original conception of the next generation CPU project. It was an entirely new design, leveraging only minor amounts of existing logic and header files from the CPU RTL databases. As such, it was a prime candidate for SVTB exercise, and until a validation resource was assigned to produce a full-fledged Validation TE, initial bring-up and exercise of the Gearbox in its early stages was performed using SVTB techniques.

However, even after the Validation environment for the Gearbox was fully developed and used for official regression testing and RTL turn-in, the SVTB approach was still used extensively for the addition of the Gearbox's more complex features.

The internal microarchitecture of the Gearbox is beyond the descriptions of this paper and considered Intel confidential material, so we only focus on the SVTB development phases of the Gearbox TE. The approach taken for RTL development was to test code in small increments as soon as it was ready, a task for which an SVTB methodology is perfectly suited. This resulted in roughly four phases of coding followed by rigorous testing, as shown in Figure 12

Phase 1: The first coding task was transcription of the fundamental logic equations that comprise the Gearbox's base function algorithm from the reference Verilog model into SV. It was decided for upper-level code readability to use SV functions instead of modules; since the algorithm fundamentally operates at the byte level using a variety of one-to-one mapping functions and their inverses, it was very straightforward to exhaustively test that passing all possible byte values through a function and then its inverse returned the original data. Testing in this way took a matter of minutes and exposed one typographical error in the low-level functions that would have been significantly harder to isolate through higher-level testing. Higher-level invertible functions that operated on a 4-byte granularity were similarly tested, though not exhaustively, on a few random data points. Note that in this phase, there was no explicit SVTB hierarchy work needed; a Verilog program that included the function library and looped through all desired data values, printing error messages when bad results was encountered, was all that was needed.

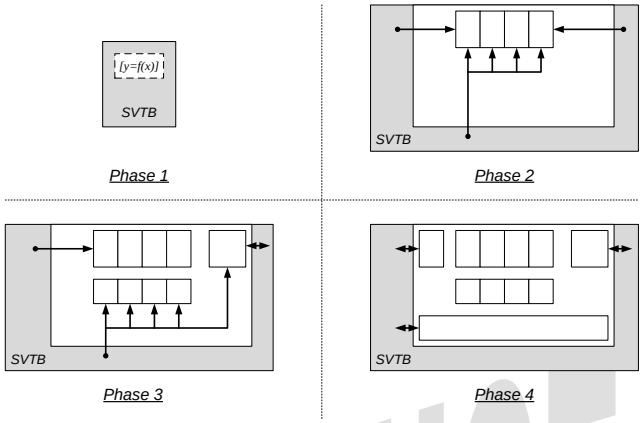


Figure 11 - Phased Approach to RTL Gearbox Development

Phase 2: With the base function library relatively well-exercised, the Gearbox sub-system functionality was then coded and tested, with the goal of proving on at least one data point that the it's main algorithm was implemented. Beginning with this phase of development, the full RTL hierarchy of the final sub-system was used as the DUT, even though not all elements were completely coded yet. All internal signals that would eventually be driven by RTL code were instead driven by TB tasks. In this phase, the data, key, and control interfaces were all driven SVTB tasks, some of which are indeed throw-away code, but some were reusable as the boundary between the TB and the real logic moved steadily outward. This approach allowed focus to be placed on flushing out bugs in the datapath, as the behavioral SVTB code for the control interface of the engine is much simpler to code correctly than the equivalent hardware FSM.

Phase 3: The next set of sub-system features were then coded in RTL, which demanded replacing current emulator logic with a much simpler emulator for the global control logic. Again, proceeding into this phase knowing that the basic datapath logic was sound allowed the focus of debugging efforts to be placed on the newly coded, and always bug-prone, control.

Phase 4: The final phase of development consisted of completing all remaining RTL code, at which point SVTB emulators were only driving and sampling top-level DUT interfaces. The focus of testing then moved from short tests, focused on correctness of single operations, to longer stress tests, where thousands of requests were sequenced into the system at both random intervals and at maximum throughput. Once this phase of exercise was complete, this subsystem was integrated into the upper level Gearbox unit code, with the very positive result of passing the higher level validation tests that targeted the Gearbox sub-system within one day of integration.

RESULTS

SVTB for Design Exercise was used in many ways on the next generation CPU project primarily filling the gaps of existing validation collateral. The usage of SVTB varied from the extreme of quick and dirty test writing to very complex TE's as described in the MLC and Gearbox examples. Even though the CPU project had a significant amount of legacy validation code, they still targeted SVTB towards exercising 13% (Table 2) of all new features (Note that a condition is something the test plan wants to hit and the checker is a piece of code that is looking for specific conditions all the time). An approximate total of 32 TE's (Table 3) were created in SVTB, most of which were considered throw away TE's.

Type	Count
Condition_TE	14118
Condition_SVTB	2149
Checker_TE	1002
Checker_SVTB	38
Total	17307

Table 2 – SVTB Conditions

SVTB Environment Type	Number
Quick & Dirty – Throw Away	20
Intermediate – In RTL Models	10
Complex Validation ENV	2
Total	32

Table 3 – SVTB TE Types

In terms of end-user productivity and usefulness, SVTB was generally seen as a big win. Ramp time was considered insignificant as the design team already understood SV. This led to quick development of tests and debugging of new features. The effort of getting the initial TE framework was free as the tools automated this. Depending on the type of TE (Table 4) the effort towards developing an initial set of transactions, getting signals into the correct CB, etc, would take anywhere from a couple hours to a week. The most praised feature of SVTB was its debug turn around time. The cost of maintaining a quick and dirty TE was very small while the more complex SVTB TE was anywhere from 5-10% of the designers time.

Activity	Quick & Dirty TE	Complex TE
Introductory Training	1 HR	1 HR
Intermediate Training	1 HR	1 HR
Advanced Training	Not needed	1 HR
Creation of TE framework	Automated	Automated
Basic transactions set dev	1-2 HR	2 days - 1 week
Test Debug Loop Time	1-2 minutes	3-6 minutes
Maintenance Cost	Less than 1%	5-10%
Return on Investment	Major Bug Reduction	Major Bug Reduction

Table 4 – SVTB Activity Breakdown

Of course there are other intangibles that have no numbers to show for them. Designers did not have to learn a new language. They had the ultimate in controllability over exercising their design which meant much faster debug. They didn't have to handoff their code to validation or wait for them to validate it before they committed it to a model. Goof bugs were caught right away before they were turned in while the code was fresh in the designers mind. There was no overhead of tracking these bugs in any tracking database. There was no overhead of clogging up the turn-in pipelines with fixes or having back end collateral be redone because the fix was made much further up in the design cycle. Additionally SVTB checkers could be shared up the design hierarchy if they were built correctly.

Results that cannot be measured:

Controllability: The ability for designers to easily toggle the functionality that matters → **Priceless**

Handoff: Doing basic validation yourself, no validation handoff → **Priceless**

Language: Existing expertise of the testing language → **Priceless**

No Goof Bugs: The absence of shame because no bugs are filed against you → **Priceless**

No re-ramp Time not spent ramping and debugging code wrote months ago → **Priceless**

No overhead No bugs filed == no tracking overhead or clogging of turn-in pipelines → **Priceless**

But of course, the most significant return on investment is the reduced bug count. The current next generation CPU bug count, at FED completion, **is trending at 35%** of what previous projects had at the same point in time (Figure 12). This is the direct result of the Design Exercise methodology and associated management tracking that was adopted and pushed by the CPU team in which **SVTB was an important factor.**

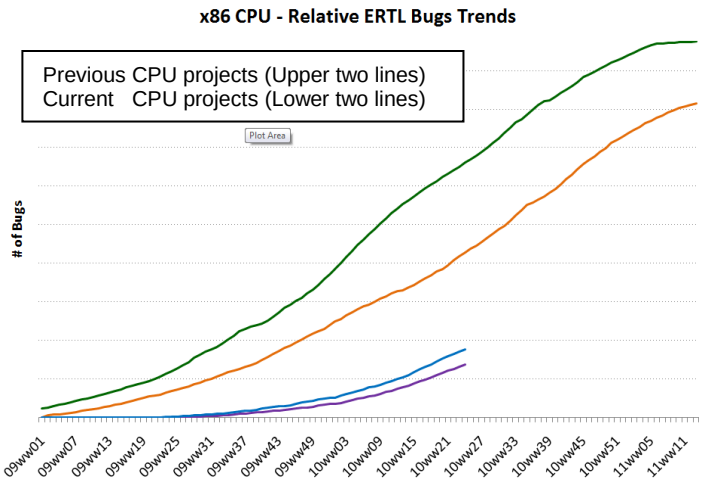


Figure 12 – x86 CPU bug count trend compared to previous projects

SUMMARY

SVTB for Design Exercise was unquestionably a success on the next generation CPU project. The consensus among those who used it is that "SVTB was great first approach for Design Exercise." Moreover, the focus on RTL stability with a heavy emphasis on the design exercise methodology is paying off in terms of historically low bug rates. SVTB gives designers an intuitive, well-known, working environment with an incredible amount of control and extremely fast debug turn-around times. Maintenance costs (0-10% of their total RTL coding time) are below what previous projects have dealt with where TE's on average cost 5-15% of designer effort.

While the results of SVTB have been beyond expectations in terms of performance and debug turn-around, they led to the discovery of needed infrastructure fixes and enhancements. This is based on direct feedback from the 20+ engineers involved in the SVTB efforts. This feedback also cited that SVTB testing is appropriate in many, but not all areas. If the legacy TE was available and supported, it was used. SVTB was considered essential in areas where existing TE's were not yet available or for cases where the ROI made sense for initial testing. Generally speaking, SVTB for design exercise was considered a "temporary" testing method.

The improvement feedback mostly centered on tool and flow maturities. Also cited was the need for more common routine files for error reporting and assertion handing. More documentation and code examples were also requested.

The bottom line is SVTB and the TE automation created on this CPU project provide the infrastructure and capabilities that allow design teams to automatically create a TE around any block or DUT in their design and quickly turn around the debug of basic features through directed test writing, which enables bugs to be caught before they reach the official project release models.

In creating this first pass automated TE and methodology; the CPU team implemented a stop-gap mechanism for filling validation holes on the project. This was understood to be a rudimentary beginner implementation with an eye towards learning from the experience and thus focus on those enhancements which would take SVTB and Design Exercise to the next level. Currently several other projects are now using SVTB for Design Exercise.

BIBLIOGRAPHY/REFERENCES

[1] Verification Methodology Manual (VMM) for SystemVerilog, Janick Bergeron, Eduard Cerny, Alan Hunter, Any Nightingale, Springer 2005

[2] Open Verification Methodology (OVM™) – (http://www.OVMworld.org/)

[3] SystemVerilog for Verification, Chris Spears. Springer 2006