Tried/Tested speedups for SW-driven SoC Simulation

Gordon Allan
Mentor Graphics Corporation
Fremont, California, USA
gordon_allan@mentor.com

***Abstract*—**

Simulation of today's SoC devices is not easy. A typical SoC in the 'mobile' application domain consists of one or more processors, one or more levels of bus fabric, one or more internal memories or caches, one or more off-chip memory interfaces, one or more peripheral interfaces, one or more timing/control resources, one or more specialized processing elements or data pipes, etc., etc.

The emphasis is normally on the 'more' than the 'one' as process technologies scale to allow us to integrate ever more levels of detail in one device, and mobile device manufacturers compete for functionality and performance. In 2011 the average high-end mobile SoC gate count was 104 million gates [Gary Smith 2012] and growing by 10% per year.

That's a lot of gates to simulate at once.

Adding hardware-assistance such as emulation can solve some verification problems, but for many projects there remains a need to simulate the final integrated RTL and/or gate-level SoC design in its entirety, at least to check power up, boot up from reset, and key architectural interactions. Given this need, many project teams still perform a significant portion of verification on this medium.

Apart from the main challenge of scale and performance, which we will address, there are other challenges. Those simulations take the longest to run and are also the hardest to debug, and the hardest to develop. Providing the right stimulus to exercise system-level and chip-level concerns is not easy. Stimulus has to be orchestrated so that the different peripheral features of the SoC are being exercised in concert with test firmware running on the processor or more likely on multiple processors.

In this paper we present a set of standard, freely available, easy to use techniques to accelerate all aspects of SoC simulation, allowing more rapid development, regression, and debug cycles during the integration phase of an SoC project.

The techniques draw upon familiar concepts:

Economics:
How to simulate more by simulating less

Energy Efficiency:
How to increase power by reducing power

Environmental Care:
Reducing overheads with zero-impact linkage

Keywords—SoC,Simulation,Software,Coverification,Stimulus

I. INTRODUCTION

The main challenges of today's SoC design/verification activities are:

- Integration and partitioning of complex designs
- Simulation and regression iteration time
- Time taken for debugging of failing tests
- Unnecessary debugging of incomplete testbenches
- Productivity hit of wasted simulation runs
- Explosion in reuse and configurability

We seek to make improvements in all of these, but especially to build a strong productive foundation of a high performance simulation, the benefits of which can affect several aspects of the project.

Simulator economics are well known. The design and verification structure is encoded into a huge data structure with nodes that are visited to propagate events and data values according to temporal and logical rules. This is typically an un-cacheable data structure and just traversing it contributes much to simulation time.

For general optimization of the simulation data structure, we consider approaches that can allow us to simulate fewer gates, and still achieve a particular verification objective. Approaches such as using a chassis to allow suppression of unused design content, or use of parameterization to reduce channel counts when not needed.

Having reduced the overall data structure size, and given a certain amount of gates in a design, there are other optimizations we can apply to make our simulation more 'energy efficient'. Of all the millions of connections being simulated, static nodes cost less to simulate than toggling nodes. Techniques drawing upon existing power management ideas to accelerate simulation are discussed here.

A set of Design for Verification rules is explored; these seek to incorporate extra features in the specification of the DUT which are there only to benefit accelerated verification (and maybe also production test) by avoiding needless operations that consume large chunks of simulation time when compared to the cycles spent actually performing the verification objective.

Finally, and our main topic in an SoC context, is to describe the various approaches to co-ordinated stimulus between test firmware running on the processor core or cores, and HVL stimulus on the peripherals and pins of the SoC DUT.

Two widely used approaches for linkage between the two (commonly known as the 'Executive' and the 'Trickbox' approach to software-driven stimulus) are documented and analyzed to show the overhead and impact of the linkage in each case, and a third approach is introduced - 'Zero-Overhead Optimized' (ZOO) software-driven stimulus - which reduces the linkage to zero, enables a rich API between the worlds of test firmware and HVL stimulus, and provides several other benefits to the user for debug and development.

When all the approaches described in this paper are used, SoC simulation regression run time can be cut by significant amounts, even 50%-75% depending on the stimulus and design. Each of the techniques described offers performance gains of 10% or more and are worthy of consideration.

These are tried and tested techniques that the Author has been familiar with since the early 1990s, and which still apply today, remaining as unused optimizations by many verification teams who suffer regression performance blight as a result.

They have demonstrable effect on simulation throughput and verification throughput (which is a function of simulation throughput and verification efficiency per unit of simulation). Eliminating waste is a big part of the process.

The result is simulation that is more relevant, that gives results per cycle, over fewer cycles, taking less hours, keeping regression time down to a manageable loop, and easier debug.

The optimizations proposed here are invitations for you to invest in activities and improvements that will give you a favorable return in overall time-to-money productivity.

Simulation Time is the most valuable resource in SoC Verification

A. *Useful Simulation Time*

An SoC simulation consists of activities that contribute differing amounts of value to the overall verification objective. Those activities include:

- waiting for power, clock, reset to become stable
- configuration of on-chip infrastructure
- configuration and maintenance of protocol elements
- preparing and/or fetching initial state, memory content
- fetching and preparing the testcase stimulus
- executing the testcase stimulus
- waiting for response to checkable stimulus items
- obtaining response from the SoC to do checking
- repetition of stimulus with differing parameters
- repetition of stimulus on different instances

A typical SoC simulation timeline may look like the figure below, with many of the above activities taking up runtime.

When we extract the ratio of actual verification to 'overhead', it is clear that an investment in removal or reduction in overhead will have payback in regression time and hence team productivity.

When the simulation includes an element of Software as stimulus, running on a DUT processor core, it is desirable to speed up the processor execution as well as the overall SoC simulation. The key to accelerating that software execution is to isolate it from the much slower logic simulator[1].

B. *Indirect Benefits of Optimizing Simulation Time*

In making improvements in simulation performance, we achieve a side-effect of improvements in:

- debug recording - less overhead to record waves and smaller wave databases to use in debug
- debug time - easier for the user to see problems or correlate them to their root cause, shorter time spans.

C. *Using realistic Software stimulus in verification*

A side benefit of simulation using software running on a processor core (or model) in the DUT is that the software need not only be 'test' software that is otherwise a throwaway.

Some routines from the real firmware co-development project can be incorporated here, which leads to more realistic interactions with the hardware of the SoC, improving both the quality of the hardware and of the firmware.

Diagnostic code can be incorporated here to exhaustively test hardware interfaces, and some functionality (although most functionality should be verified strictly at block-level or subsystem-level and there is no benefit in repeating that at SoC level).

For cases where more extensive functional verification is required at SoC level, for interactions that cannot be verified at a lower level, use of real driver code enables extensive functionality coverage and some interface coverage[2].

Convinced that some investment is worthwhile in this area to get a productivity return? We start with some easy fundamental improvements.

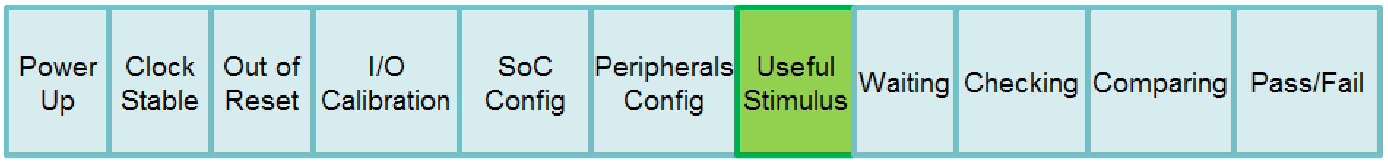

Figure: Typical SoC Simulation Timeline

II. SIMULATE MORE BY SIMULATING LESS

Can we do more with less? We discuss the economics of SoC scale simulation. Economy of the simulated DUT footprint is worthy of analysis.

A. *Chassis Approach to SoC simulation*

The ultimate approach to optimizing the DUT footprint in the simulator is to build the SoC DUT as a chassis, having configurable elements for each major subsystem in the design specification. The DUT will probably already be divided up hierarchically along those lines. One possible investment is to provide a configurable subset that is compiled or elaborated, depending on the requirements of the test suite being run. Various approaches are possible here and all are useful; some combination of them may be the best approach for a given SoC DUT:

1) *Use Stub modules for unused blocks*

For each configurable subsystem, create a stub module which can be instantiated as a proxy for that subsystem in its inactive state, i.e. it should be instantiatable just as the real module is, and have outputs appropriately tied off so that it does not have a negative effect on the remainder of the SoC. Creating stub modules is good practice for incremental SoC development and integration anyway. We are suggesting that they are not thrown away - instead maintain them as first class deliverables which are 'views' of the module that take up the least possible simulation footprint at runtime (and compile time - which may also be a significant benefit when RTL compilation time can be saved)

2) *Use Behavioral Models for lightly used blocks*

Even if stub modules are not practical, e.g. because the module has to remain partly functional or must operate for some small portion of the total simulation, it is good practice to use a behavioral model as a substitute, provided that we are not currently verifying that module, there is no impact on overall 'verification quality' while substituting it for a functional model. This is good practice for CPU or other processing elements anyway - leading to good debug capabilities as well as more instructions per second (IPS). This good practice benefits the performance because there are fewer gates to simulate, and the processor core by now is 100% verified.

B. *Optimize Multiple Instances*

When an SoC contains multiple instantiations of a module of complex functionality, it is necessary to verify each instance at SoC level to ensure that it is properly connected up to buses, interrupts, clock and low power signaling, and pin/pad connections.

However, it is only necessary to do that once, in a test suite which has that particular aspect of the verification as its goal.

All other test suites that need to use that module functionality can probably use a single instance, so there is no need to instantiate N instances for the remainder of the functional test suite at SoC level, provided we have taken care of verification of interactions at least once, and for the remainder, that we have taken care to ensure such interactions are benign.

The multiple instances may be complex peripherals, e.g. networking controllers, but one obvious manifestation of this approach for today's SoCs is in the multicore CPU area. Apart from verification of interactions, coherency, involving more than one CPU, the remainder of the SoC testing needs only one CPU, and the other or others can be removed from runtime.

C. *Configure Selective / Unused Functionality*

What we have implied is that on a per test suite basis, or even on a per test basis, we can configure the DUT build for simulation, so that modules which are not used for this test, or this batch of tests, are ignored, or removed, or optimized out.

Having a toplevel configuration profile is a useful approach, as the management of this set of DUT variants can be difficult. Above all, although there is a big runtime payback for reducing the DUT footprint, we must preserve the integrity of our verification; that would be too great a cost.

D. *Tradeoff between Compile, Elab and Runtime Control*

RTL and HVL have mechanisms in the languages to enable the chassis approach and configurability thereof, either at compile time, or elaboration time. What about runtime?

III. USE POWER MANAGEMENT SPEEDUPS

We describe here some techniques to increase the Verification Power of your simulation farm by simulating your SoC using its Low Power modes. Some may seem obvious.

If you cannot reduce the simulation footprint by the "Simulating Less" techniques mentioned in the previous section, you can still optimize the footprint's impact on SoC simulation runtime in many ways.

A. *Put Unused Peripherals to Sleep*

If your SoC has power management configuration logic built-in that lets you control which functions of the device are clocked and which are not, then use those features to optimize runtime.

At the expense of normally just one write to an SoC configuration register, many gate-transitions of simulation load can be saved by not clocking whole blocks of synchronous logic.

Each testcase can start with a minimal configuration for only essential operations on the chip being clocked, and add only those peripherals or functions that are involved in the test. If the testcase involves different functions during different phases of the test, wake them up and put them to sleep as required. Normally the 'cost' of that one write, even if repeated several times during different phases of the simulation, is repaid by keeping large amounts of DUT gates and flops static rather than toggling.

If as a result of this runtime configuration, your SoC DUT is not consuming as many milliwatts of battery power, then neither is it consuming excess simulation time.

B. *Align Clock Frequencies*

If there are multiple modules on chip with configurable clocking frequencies, set them up so that modules do not waste time waiting on a slow 'partner' module - slow them down to operate at common speed, or if that does not make sense for your design, speed them up to a common speed.

Have an overall approach to choices of operating frequencies of the various subsystems of your SoC to give benefits for simulation efficiency while preserving the integrity of your verification requirement for this test.

For example, if one peripheral is unnecessarily slow compared to the bus or CPU, then simulation cycles for all gates in the design are wasted on the slow part, while each CPU bus access to it waits for delay cycles. It is worthwhile to change the settings to minimize that effect.

Sometimes those operating frequencies are dictated by off-chip concerns e.g. a standard protocol operating at its normal frequency. Configure the whole SoC if necessary to match that, to avoid this disparity that wastes simulation-time-per-gate.

Use this optimizing approach for all tests that do not require 'real world' frequencies.

> Ask yourself "What are we verifying?" and optimize accordingly. If you are verifying 'at-speed' interactions, then setup your SoC operating frequencies for real world. Likewise, if you are validating performance at maximum specified load, you need real world interactions between SoC subsystems and off chip protocols.
> For all other verification, i.e. the majority of your functional testing at SoC level, optimize your SoC configuration for maximum simulation throughput.

IV. Use Design for Verification Speedups

Design for Verification techniques can be applied to your SoC design in order to assist the verification process. We explore there here. These are techniques which add logic or structures inside the SoC design, specified in order to benefit accelerated verification (and in some cases also to benefit production test efficiency).

The typical approach is to avoid needless operations that consume large chunks of simulation time when compared to the cycles spent actually performing the verification objective. This ratio of Simulator Efficiency depends on many factors and time taken to initialize or maintain the SoC operation in some context, is a significant factor.

Examples of SoC operations that consume significant elapsed simulation time, for good functional reasons, but no benefit to simulation, are:

- bus delays and timeouts
- hardware or software watchdog timeouts and other timers
- analog-driven reset initialization sequences, e.g. whole voltage or clock stabilizes
- analog component bringup or synchronization e.g. PLLs or PHYs
- calibration of interface delays e.g. DDR3

A. *Bus Delays and Timeouts*

The bus protocol and transfer mechanism should be verified separately at the appropriate level of module hierarchy. At SoC level there is no need for wait states or bus delays - configure them to the absolute minimum - and also bus timeouts should be set to the minimum effective value (avoiding false triggers).

B. *Watchdog Timeout Features*

Any other kind of timeout whether for software error recovery or in a hardware protocol such as Ethernet, should have a bypass mode which allows configuration out with the normal functional range of values, for effective verification.

C. *SoC Reset / Bringup Sequences*

Typically a power up sequence will await voltage stability, start any PLL/DLL functions, and then await clock stability, before releasing the majority of the SoC logic from reset allowing the main part of the simulation to proceed. Ensure both those delays can be bypassed for digital verification, where 'voltage' is instantly stable and clocks are 'instantly' active, locked and stable.

Another reason to remove any clock stability logic and rely on an artificially stable clock is if recording waveforms for debug. Wave databases typically have optimization for periodic clock signals, but that optimization can break if there is an artificial injection of 'jitter' on the clock signal. Jitter rarely adds any value to SoC verification (except perhaps for that one test which validates the clock control mechanism under load).

D. *Protocol-specific Interface Delays*

Various protocols in common use today have overheads in their implementation that are required for electrical functionality but not for digital functionality. For example, look at the SDRAM progression from DDR to DDR2 to DDR3 to DDR4 - not only refresh, but calibration - in some cases initial one-off calibration, in others, a regular repeat of the calibration process, in order to train timing-sensitive data line capture.

There is no need to spend time calibrating picosecond delay offsets for complex off chip memory arrays like DDR3 and DDR4, for the majority of SoC simulations. If it is possible to configure a DFV mode which bypasses the training phase, then do so.

Again, only one simulation test needs to have that function enabled: the one which is responsible for verifying it.

One caveat is that there may also be a set of 'performance' simulations for which accuracy may be paramount and some, not all, of the DFV modes discussed would need to be disabled.

E. *Analog / Mixed Signal Components*

Mixed signal blocks like PHYs or DLLs may be replaced with complex digital or behavioral models for SoC simulation - ensure these have DFV modes to fix clocks and reduce delays and timer count values of the type described above.

F. *Low-Power Design Features*

Even if your design does not need to run from batteries in a portable device, the optimizations described earlier may be one good reason to design in some low power features.

If your SoC does not have those capabilities - consider adding them. The environment - and your verification environment - will benefit.

V. Choose Optimal S/W Stimulus Approach

When a chip design includes one or more processor elements, they can be used to provide stimulus 'from the inside'. Processors are typically masters of activity, rather than being responsive to hardware activity. So in fact it is more true to say that they MUST provide stimulus from the inside. At least, they must respond to surrounding activity in such a way that a verification objective is met.

However, the remainder of the SoC also requires stimulus (or controlled response) - the pins of the device, all the protocol interfaces, all the surrounding protocol VIPs, need stimulus to act in concert with what is happening on the CPU core. Someone has to plan and implement that overall set of stimulus and some technique or technology has to orchestrate it, facilitate its coordination, to meet the objective.

In some cases, the software running on the CPU core is more than just test case code reading and writing a handful of registers; it may be real firmware implementing an API, indeed the verification objective may be to test the firmware layer as well as the SoC integration. In other cases, it exists only to facilitate HDL/UVM verification of integration pathways across the SoC.

Design verification teams have been wrestling with this perennial problem since the earliest embedded CPU ASICs which were SoCs before the term SoC was coined. They want stimulus to the pins and hardware of the SoC, providing protocol input on its interfaces, and they also want a sequence of software operations to be run from within, executed code on the embedded processor core(s).

In particular, they want those two kinds of stimulus to interact with each other, and assist in testing of both areas: the CPU and bus, and the peripherals and integration logic, and sometimes the firmware.

At least three interfacing and synchronization techniques exist for coordinating software stimulus and HDL stimulus, each with varying degrees of initial development required:

- the Software Executive API
- the Trickbox interface, and
- the Zero-Overhead Optimized approach

We describe each of these approaches in turn and compare their advantages and disadvantages, in particular the amount of overhead they incur in DUT simulation time and the impact that has on overall productivity and return on upfront development investment.

A. *Option 1 - the S/W Executive Approach*

The Executive API is one approach to coordinating software stimulus running on an embedded CPU core with discrete HDL stimulus on the pins or hardware of the SoC DUT.

> The Executive approach to coordinating S/W and HVL stimulus has the HVL Stimulus (e.g. UVM sequences) as 'master' which invokes 'slave' S/W routines via an I/O interface on the CPU bus.

It consists of an Executive - a software loop like an RTOS event dispatcher, running forever on the embedded processor core, which responds to available observed stimulus.

1) How it works - Inputs - Processing - Outputs

The Software loop requires an input mechanism from the Testbench, to cause Software activity and to specify which particular Software activity is required. This is typically some combination of an interrupt mechanism which exists already on the CPU / SoC integration logic, along with an I/O module allowing signaling to be read from the testbench.

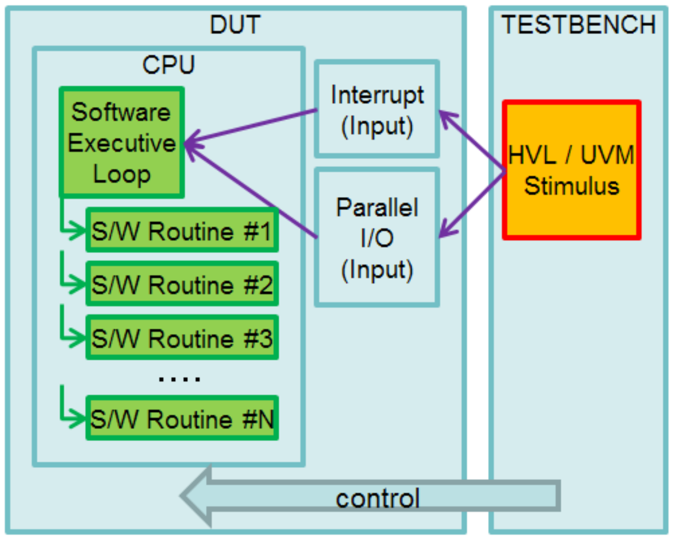

Figure: S/W Executive Architecture

When the software loop is interrupted, the I/O is read and decoded in some kind of lookup table or case statement, and a subroutine or series of software instructions is selected and executed as a result. When the routine ends, the executive loop continues, ready for the next interrupt.

2) Advantages

It is difficult to think of many advantages, but there is one: this is the only approach where HVL is the master and software is the slave. That may be an advantage in some situations.

3) Simulation Time Overheads

The main overhead is that between each valid S/W stimulus routine triggered from the master HVL stimulus, there is a period of CPU cycles which is wasted overhead, either idling or in the process of responding to interrupt, decoding the required activity, calling the required routine, eventually returning from that routine, with all the related stack push/pop operations that interrupts and subroutines entail.

4) *Other Disadvantages*

The principal disadvantage, other than performance, is that the I/O interface needs to exist, consisting normally of some data I/O signaling (e.g. a GPIO module) plus some event I/O signaling (e.g. an Interrupt input signal).

This approach necessitates either some modifications to the SoC architecture/RTL design purely to support SoC-level verification, or uses some existing peripheral I/O that happens to exist already in the DUT specification. This may be a limiting factor or more importantly prevents any verification that needs that I/O function (e.g. GPIO lines) for other operation as part of the verification - it becomes dedicated to the interfacing between HVL stimulus and the S/W executive.

As a result, this approach is often used only with a temporary SoC integration chassis which augments the specified function of the SoC with an additional I/O capability. This complexity is a significant disadvantage and so this approach may be a niche solution.

B. *Option 2 - the Trickbox Approach*

A simpler approach to software-driven stimulus is to implement an adapter within the testbench, which acts as a memory-mapped device, connected to the DUT processor bus or bus-to-memory signals. This adapter implements the linkage needed to transform the specific bus access to the required testbench stimulus or action.

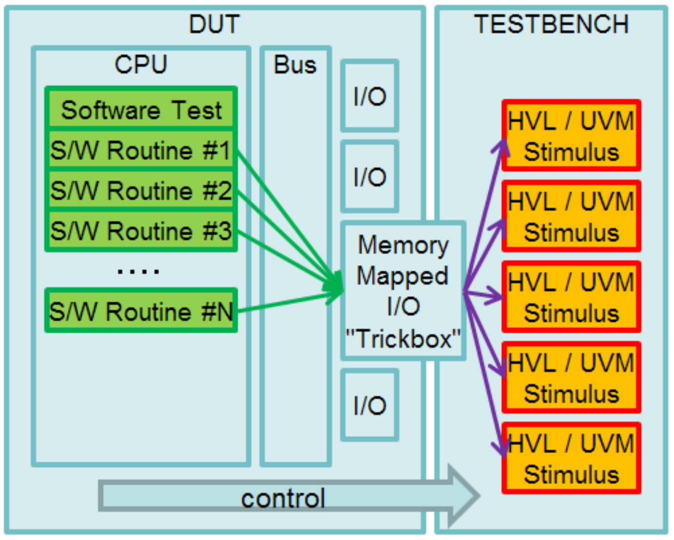

Figure: Memory-Mapped Trickbox Architecture

The adapter module is often referred to as a Trickbox. The technique is also referred to as a memory-mapped testbench interface.

The 'Trickbox' approach to coordinating S/W and HVL stimulus has the Software stimulus as master, triggering HVL stimulus (e.g. UVM sequences) via a memory mapped adapter: the Trickbox.

This linkage can be as simple as a predefined address value or range of values, and determination of which action from the set of supported testbench actions is to be triggered can be determined either by decoding some of the address bits within the decoded range, or by decoding the data value written to that address.

1) *Advantages*

The primary advantage is simplicity. Providing a self-contained adapter written in HVL that recognizes bus activity and triggers HVL stimulus is the simplest way of implementing this linkage, even simpler if existing DUT memory-mapped I/O signals are used'

Simplicity is a considerable advantage, because multiple team members may need to comprehend the code and the 'magic' at different stages of the project. Also, debug may be a frequent occurrence, and is ideally a task that requires an understanding of the code and techniques, comprehensible by the design team as well as verification/HVL experts.

The simplicity also applies to the implementation task which can be relatively straightforward, given the inflexible nature of this approach. A simple implementation can be used with multiple SW languages (C or assembly) and multiple testbench languages (SV or Verilog or VHDL) in multiple DUT/testbench architectures, without major engineering effort.

2) *Simulation Time Overheads*

The main disadvantage of this approach is the simulation time required to implement the simple linkage, to enable the software to convey instructions to the testbench to provide stimulus or other actions.

Simulation time is spent not only on the bus cycles operating the linkage, but also on the bus cycles required to fetch the instructions that are used to create those bus cycles.

[Verification teams often live with this overhead, but it can be eliminated, as we will describe shortly.]

Interactions which cause HVL activity therefore are pure overhead - at least one opcode fetch and one bus write per instance. Interactions which retrieve data from addressable locations and then do some checking have more cycles of overhead - in the worst case the cycles taken to read data, apply a logical bit mask, compare the value with expected, and jump to a pass or fail endpoint, can be added as pure overhead PLUS the cycles required to fetch all of those masking and comparing and jumping instructions from memory.

The ratio of simulation overhead to actual useful interaction is therefore at best 3:1, often 4:1 or 5:1.

3) *Other Disadvantages*

The Trickbox approach is inflexible and may be hard to extend from one project to the next, due to the fixed nature of the encodings that are passed across the linkage in address values and data values.

This disadvantage can be minimized by a symbolic approach, where both software code (C or assembler) and HVL testbench code use a common set of enumerated identifiers to designate the functions that can be triggered.

Often, an elaborate hierarchy of linkage sub-fields must be implemented in order to provide a more programmatic form of interface, e.g. using address values to select a function and data values to transmit some variable parameter to that function.

C. *Option 3 - Zero-Overhead Optimized Approach*

Both the Executive approach and Trickbox approach have overheads which consume simulation time. It is possible to eliminate those overheads, with some initial investment.

The principal overhead is the runtime linkage that allows the software stimulus being executed on the processor to trigger HVL hardware stimulus or other testbench activity.

The optimization we propose is to remove that runtime linkage overhead entirely, and replace it with a compile-time linkage that requires zero simulation time overhead.

Instead of a sequence of software instructions to prepare, trigger stimulus, measure response, compare and check pass or fail, running at a particular program counter location within the runtime software image, we use the program counter directly, with a debug breakpoint mechanism, to trigger a pre-prepared sequence of HVL activity.

Further, we enable the specification of the HVL code to be triggered in line with the remainder of the software stimulus. This requires the initial investment in customization of the compiler or assembler, or a wrapper and preprocessor for them.

With this 'ZOO' optimized software-driven verification approach, a single stimulus file contains both the embedded software portion and the triggered HVL portion of the test stimulus, inline in one file with a shared syntax.

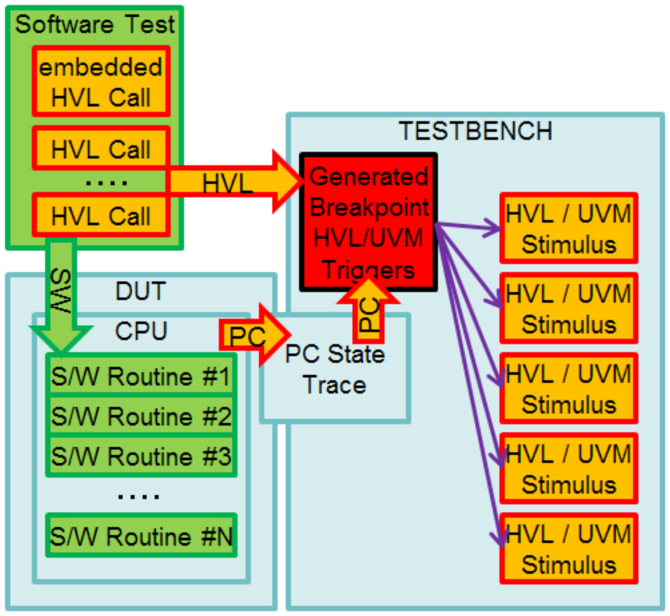

Figure: Zero-Overhead Optimized SDV Architecture

1) *Implementation: Custom compiler/assembler tool*

A modified tool flow is required in order to take a single combined source language file and split it into the two parts: software to run on the processor and HVL to run on the testbench, synchronized by instruction address breakpoints.

The instruction address for any given line of source code is not known until after compilation/assembly is done, and the ability to retrieve that information from the output of the compilation/assembly tool is required to set up the breakpoints.

There are some options here. One is to customize the tool completely to enable additional syntax to allow HVL statements or triggers to be embedded in the software at the appropriate place, and to add a back-end code generator to the tool to write out the HVL breakpoints and the actions to be performed when they are hit.

Another option is to create a wrapper around the software compiler/assembler tool which separates the two kinds of code, leaving only 'markers' in the S/W code, using existing legal syntax for pragmas or comments, that can be retrieved later from an enhanced listing output file that is already available as an output from the tool. Such an enhanced listing file would need to retain (1) the address of the instructions generated and (2) the text of the marker tag (pragma or comment) preserved in the output in the correct location relative to the addressed instruction (otherwise some slight tool mods are required here)

The wrapper preprocesses the combined source, runs the compilation or assembly tool, and then creates a file containing all the HVL breakpoint activities and linkage to trigger them retrieved from the instruction addresses in the enhanced listing output, keyed by the marker tags that passed through the tool.

2) *Implementation: Processor Breakpoint Logic*

The other half of the implementation of Zero-Overhead Optimized SDV is the instrumentation of the processor in the DUT, to interface to the testbench, so that the required HVL activity in the tool-generated breakpoint file can be triggered on an instruction-address basis during SW execution.

A processor model, whether RTL, gate-level, behavioral model, or ISS with a bus wrapper, has a Program Counter that tracks the location instructions that are fetched, decoded, executed and retired. This Program Counter (PC) value must be made available to the testbench to enable stimulus synchronized to software execution.

High level models will have a debug API to support tracing which can be used here. Behavioral, RTL, or Gate models will be successively more complex but the PC value is in there somewhere, it just has to be brought out.

The complication here is modern pipelined processors, which have speculative execution, branch prediction, rewinding. In these cases an algorithm is required to indicate that the instruction at a particular PC address X has been committed, not just fetched, or decoded, or speculatively executed in the ALU. A useful triggering algorithm is to enable triggering of events to occur AFTER an instruction has been executed, i.e. when the PC leaves that instruction address and moves on to the next instruction.

In addition, it is desirable to synchronize after any data accesses that occur as a result of that instruction, but prior to any that occur in the following instruction. Also it is necessary to take into account any effects of branches that are taken vs branches that are not taken.

If necessary, a NOP can be inserted in the instruction stream in the case of complex series of operations that are indistinguishable, but that reduces the benefit of the technique so should only be used when necessary.

Ensure that interrupts or exceptions (which can by definition change the flow of control unexpectedly) are accommodated, also the RTI or RTE statements which return from those routines.

This requirement is a significant portion of the up-front one-off development effort. Consider using a standard behavioral model for your processor core that already has a committed instruction PC or trace capability.

Once you have this signaling and/or state machine set up, encapsulate in an API which calls the generated HVL 'breakpoint definition' file at the appropriate time, passing in the PC value.

3) *Example Stimulus Code*

Depending on the tools available for your project, and for the processor core in your DUT, and depending on your design decision either to customize the tool to allow embedded HVL constructs, or add a wrapper/preprocessor to extract those and later synchronize them, there are different ways of delineating and demarcating the source file. Both assembler and C code examples could use a similar syntax, and enable both simple Verilog task calls and UVM-style sequence invocations.

```
TEST1:  MOVI.W $1234,R0
        MOV.W  R0,(DMA_CNTRL_REG_1)
        //UVM StartDmaSequence(32'hA0000000,1);
        CLR.W  (DMA_COUNT_REG)
        MOV.W  (DMA_STATUS),R1
        //UVM CheckDataRead(16'h0001,.mask(16'hAA00));
        JMP.S  TEST2
```

Example 1. HVL Code embedded in Assembler Software comments

4) *Pre-canned macros*

Named HVL subroutines or macros can be used as shorthand to trigger common activities in a concise way e.g. a function to mask and check the next bus read prior to reading a result from a volatile I/O location, may be wrapped in a task or macro called 'CheckDataRead(data,mask)' with all the masking and comparison and jumping to pass or fail being done in zero 'DUT time' in the HVL.

5) *Advantages and Disadvantages*

The advantage in simulation performance is clear. There is zero overhead at runtime, saving (N clocks x G,GGG,GGG switching gates) worth of unnecessary simulation effort.

Requiring a custom compiler / assembler flow requires an initial investment and maintenance. To minimize this upfront cost, plan to reuse this investment across multiple projects that need this kind of SW stimulus approach (a slight disadvantage).

Requiring a runtime state machine tracking the processor state and in particular the program counter for committed instruction execution requires initial investment and can be complex. This complexity can be perceived as a disadvantage. However, many other desirable benefits of debuggability are enabled by this approach (enabling a generic trace/disassembly and breakpoint mechanism for example) diluting the cost. A slight disadvantage.

Requiring the link between two kinds of stimulus to be made at compile time rather than run time seems initially inflexible. However, the stimulus (both SW and HVL) does not change from one run to the next, and neither does the synchronization between the two, so there is no 'compile tax' to pay - the stimulus only needs compiled once, just as with the other approaches. Also, the Test specification is all in one file.

Having a section of HVL code which is machine-generated as part of the automation flow rather than written by a user and checked-in, can be perceived as an added complexity. This should be minimized by adding clarity so that the file is comprehensible by the reader, including comments and naming that help to relate the triggered activity clearly to the source.

VI. Conclusions

Various standard performance improvements were described and are possible at the testbench architectural level, and they can be used and benefitted from in conjunction with (and independently of) the performance improvements of specialist EDA vendor tools e.g. CodeLink, MC2 and inFact.

Simulation Time is the most valuable resource in SoC Verification

Providing the foundation of simulation performance optimizations described here is a good start, and worthy of up front architectural design, before the inefficient testbench takes root in the project and is hard to change.

VII. Resources

Other solutions are available for SoC-level software-driven verification, providing further performance optimizations and/or abstractions for more powerful verification.

1) *Graph-based stimulus and SDV*

If a more comprehensive mapping between software and UVM stimulus is required, graph-based solutions such as Mentor's UVM Software Package used with Mentor inFact[3] can be used.

2) *Codelink: Replacing Processor Cores with models*

Replacing a processor core with a cycle-accurate model can result in speedups in software execution up to 10,000x faster, providing a ratio of overall simulation speedup of 5x to 10x depending on the ratio of SW execution to logic simulation[2].

The optimizations provided by fast models are essential, but the speedups described in this paper are independent of them - they save multiple clock cycles of elapsed DUT time - where all the gates in the DUT are contributing pure overhead. That saving scales with the size and complexity of the SoC; it is a vast improvement and worthy of some upfront investment.

References

[1] J. Kenney, "Firmware Driven OVM Testbench," Verification Horizons, June 2008

[2] P. Luszczak, "Processor Driven Verification - Use it for More Than Just Sign-off," Mentor Graphics, October 2008

[3] M. Ballance, "Boost Verification Results by Bridging the Hardware /Software Testbench Gap", DVCon 2013