This is always a busy time of year for me. In addition to my Verification Horizon editing duties, I’m also currently the General Chair of the annual Design and Verification Conference (DVCon), at which this issue of the newsletter will make its debut. Both of these roles provide me with a valuable perspective on the advancements being made in our industry and, coming as they do near the beginning of the calendar year, afford me a wonderful opportunity to recharge my batteries and renew my enthusiasm for the coming year. I hope I can share some of that enthusiasm with you – and maybe even get you excited as well.

I always enjoy seeing examples of innovation in our industry, especially those cases where novel solutions grow from unique combinations of new and/or existing technologies. The choice to use a widely adopted technology as a starting point, rather than starting from scratch, often makes it possible for new solutions to gain quick traction and credibility. When done right, a good offering not only solves a particular problem but also provides the necessary foundation on which future success can be built. I think you’ll see some great examples of what I mean in this issue.

Our first article this month, “Using inFact with OVM Sequences,” shows how Mentor’s inFact intelligent testbench automation tool can be deployed seamlessly with an existing OVM-based environment to provide more effective and efficient stimulus generation...more

“One of my favorite things about this week (DVCon) is the opportunity to meet new friends (and catch up with old friends, too), and I’d love to hear in person what you think about the conference, Verification Horizons, or the industry in general.”

—Tom Fitzpatrick

For more information or to download the full newsletter, visit [Verification Horizons](http://www.mentor.com/verificationhorizons).
part of the system remains sluggish, will appreciate the benefit of such an approach. The article also discusses how modularity and configurability allow you to reuse most of your existing OVM environment as you take advantage of the increased emulation performance.

Staying with our emulation colleagues, we next learn of their experiences in using iDesignSpec (IDS) from Agnisys, one of our Questa Vanguard Partners, to automatically generate the code and documentation for all of their registers. The newly-released OVM Register Package v1.0 provides a convenient abstraction layer for handling different register types and bus transactions in an OVM environment, and IDS provides a useful platform to specify the registers and memories in a system. The automation not only produces OVM code, but also makes it easy to keep the verification and design code in sync with the specification, which inevitably changes throughout the development process.

Our next article, “Verification Management Eases Those Re-Spin Worries,” describes important aspects of managing the verification process from an initial verification plan through to the ultimate goal of electronic closure. The article also covers the automated management of verification runs, results and trend analysis, and test plan tracking. All of this is made possible by the underlying architecture of the Unified Coverage Database (UCDB), which forms the basis of the Accellera Unified Coverage Interface Standard.

We turn once again in this issue to our good friend and colleague Harry Foster, whose “What’s New With the Verification Academy?” article describes how we’ve expanded our FPGA Verification module and released a new module on OVM basics. If you haven’t visited the Verification Academy yet, I encourage you to do so.

In the “I love it when a plan comes together” department, our next article shows the value of the VIP Interoperability Standard recently released by the Accellera VIP Technical Subcommittee (of which I’m a member). Mentor has established an impressive portfolio of Multi-View Verification Components (MVCs) that support verification standard protocols in OVM. To make these MVCs available to a wider range of users, the authors used the interoperability standard to integrate an MVC into a VMM environment. The article gives a great overview of the interoperability library and concrete guidance on how to apply it in a real-world situation.

On a related note, for those of you who realize that OVM is the better way to go, our friends at Chipright show in this issue how to convert a VMM testbench to OVM. The article presents a nice step-by-step guide to everything from environment structure to converting transactors, communication semantics and stimulus generation. The key “take-away” is that it’s a lot easier than you might think, so there’s no reason not to do it.

Last but not least, our partners at XtremeEDA give great insight into transforming your IC development process and becoming more agile in handling the issues that always crop up during a project. Their offered list of best practices is informed by their wealth of consulting experience with many different teams and processes. I’m sure you’ll find these ideas to be extremely (pardon the pun) useful.

If you’re receiving this newsletter at DVCon, I hope you enjoy the show. One of my favorite things about this week is the opportunity to meet new friends (and catch up with old friends, too), and I’d love to hear in person what you think about the conference, Verification Horizons, or the industry in general. Please stop by and say hello.

Respectfully submitted,

Tom Fitzpatrick
Verification Technologist
Mentor Graphics
Hear from
the Verification
Horizons team
weekly online at,
VerificationHorizonsBlog.com
TABLE OF CONTENTS

Page 6...Using inFact with OVM Sequences
by Jay O’Donnell, SLE Division, Mentor Graphics

Page 10...A Strong Foundation for Design, Verification, and Firmware Requires an Automation Tool for Register Management
by Francois Xavier Duthu @ Mentor Emulation Division, Mentor Graphics, France and Anupam Bakshi, Agnisys

Page 14...Transaction-Based Testbench Methods Speed Veloce Hardware Acceleration
by Jim Kenney, Marketing Director, Emulation Division Mentor Graphics

Page 18...Verification Management Eases Those Re-spin Worries
by Darron May, Product Manager, DVT Division, Mentor Graphics

Page 22...What’s New with the Verification Academy?
by Harry Foster, Chief Verification Scientist, Design Verification Technology, Mentor Graphics
Page 24...Reuse in the Real World—Proving the Accellera VIP Interoperability Kit

by Manoj Kumar Manu – LMTS, Ashish Kumar, Lead Application Engineer, and Adam Erickson, Verification Technologist, Mentor Graphics

Page 27...Converting from VMM to OVM: A Case Study

by Damien Melville, Chipright Ltd

Page 33...Agile Transformation in IC Development

by Bryan Morris and Neil Johnson, Xtreme EDA
 Using inFact with OVM Sequences

by Jay O’Donnell, SLE Division, Mentor Graphics

OVERVIEW
Most OVM users understand how OVM sequences can be used for stimulus generation, implemented as either a directed or constrained random test. However, they may not be aware that this same OVM sequence construct is supported by inFact intelligent testbench automation. This article examines how inFact’s coverage driven stimulus generation solution can be deployed in an OVM sequence environment and compares simulation performance between sequences developed using a constrained random methodology and inFact.

BASELINE OVM CONSTRAINED RANDOM SEQUENCE
A simple floating point unit example is used to illustrate the OVM sequence application. Figure 1 shows the main elements used when verifying FPU operations:

An OVM sequence generates a series of fpu_request items which the sequencer delivers to the FPU. The FPU processes floating point operations having the following input operands:

- (1) “A” real-number input operand driven with values falling into (7) ranges of <-1, -1, >-1<0, 0, >0 <1, 1, >1
- (1) “B” real-number input operand driven with values falling into (7) ranges of <-1, -1, >-1<0, 0, >0 <1, 1, >1
- (5) OperationTypes: +, -, *, /, square-root
- (4) RoundingMode: round_up, round_down, round_even, round_0

The code structure for this simple constrained random OVM sequence is shown in Figure 2.

The duration of the sequence during simulation is determined by the implementation of the body() task. This example uses a procedural loop in the body() task to generate a series of random FPU operations calling task random_calc() which in turn calls sequence task random_operand() to select A and B values. The randomly constructed fpu_request is then passed to the sequencer for execution. The forever loop in body() task runs until coverage is obtained by an external coverage block which evaluates each request’s construction against user-defined coverage. The coverage block implements a cross-product of the operands above containing 980 unique cross-bins (7 * 7 * 5 * 4).

In the more general case, the body task might implement procedural code of arbitrary design necessary to construct sequence items for the application. Often the sequence items are described in a separate randomized class having various rand data members and associated constraints describing legal values for the data fields. For more
complex applications, the sequence item randomization may use multiple calls to randomize() sub-fields in the item controlled by procedural code in the body() task.

**TESTBENCH DESIGN AND PERFORMANCE CONSIDERATIONS**

The complexity of the sequence and related sequence item constraints, and the amount of simulation time required to obtain coverage, depends on the size of the coverage, the size of the sequence_item rand variable state space, and the constraints affecting random variable selection.

If the coverage and rand variable state spaces are fairly small, users will get good results using a constrained random methodology. As the coverage state space becomes larger containing crosses of many variables, coverage closure becomes more difficult requiring multiple simulation runs with different seeds to arrive at a set of simulations that achieve coverage goals. Sometimes coverage closure becomes very difficult because of a combination of a large coverage state space and random variable constraints. In these cases, users may spend considerable time analyzing why certain cover points are never reached, and may resort to writing directed tests to make up for the shortfall. Considerable simulation time and engineering effort may be expended, design schedules may be impacted, and confidence in the verification result diminished if decisions are made to accept sub-optimal coverage due to time constraints.

Applications that have these coverage closure problems are good candidates for inFact.

**OMV SEQUENCE TESTBENCH IMPLEMENTED WITH INFACT**

Figure 3 shows the same application implemented using inFact. The only structural difference is the replacement of the constrained random OVM sequence block with an inFact OVM sequence block.

The inFact sequence architecture uses the same OVM plumbing as its constrained random counterpart and fits into the existing OVM environment without modification.

**INFACT OVM SEQUENCE STRUCTURE**

The main differences between constrained random and inFact OVM sequences involve the design of the OVM sequence body() task. Procedural code and random calls normally found in the body() task are replaced with calls to an inFact rule graph. Graphs typically implement a loop where each iteration assigns values to the sequence item by calling “Action Tasks” in the sequence class. The relationship of the graph to the body() task and OVM Sequence is shown in Figure 4.

In the FPU application, the graph replaces procedural code in the OVM sequence body() task, creating the fpu_request, populating its fields, and delivering it to the OVM sequencer. Most of the action tasks assign values to the sequence item fields, and can be implemented in a single line of Verilog code.

The first step in developing an inFact OVM testbench is rule creation. Figure 5 shows the user-developed rule text and its inFact-generated graph. The lowest-level graph object, known as an action (green), is mapped to a task in the OVM sequence that implements its
function. Combinations of actions can be associated with rule symbols (brown) expressing hierarchy. An example of this is symbol “Rounding” which is assigned a choice of four possible FPU rounding modes symbolically defined and selected using the “|” logical or operator. Rules are composed using combinations of actions, symbols and rule operators which include alternative choices and repeats. Symbol “Test” expresses all of the possible ways an fpu_request could be assembled and delivered. The highest level symbol, “instr_gen_engine”, specifies the procedural execution of the OVM sequence including a testbench initialization phase (action init_tb) and an infinite repeat of symbol “Test”.

At that point, users have the option of terminating simulation, terminating the sequence, or reconfiguring the graph to run in “graph random” mode which is similar to constrained random operation.

Three inFact coverage algorithms can be incorporated in any inFact rule graph and may execute concurrently or in-sequence:

- Verification that each action was entered at least one time, under control of an inFact “Action Coverage” algorithm. Users often configure action coverages corresponding to testbench coverpoints.
- Verification that every path through the graph was traversed at least one time, under control of an inFact “Path Coverage” algorithm. Users often configure path coverages corresponding to testbench cross coverpoints.
- Selection of regions not defined in a path or action coverage region to be traversed randomly, which is the default inFact behavior.

Figure 6 shows the same graph implemented with a path coverage region (blue) and a typical path displayed. This particular graph contains a total of 980 possible paths which the path coverage algorithm selects during the lifetime of the OVM sequence. Sequence logic terminates when all 980 paths are complete for this particular implementation.

EXECUTION OF AN INFACT OVM SEQUENCE DURING SIMULATION

An inFact OVM sequence, like any OVM sequence, is invoked by its associated sequencer. In an inFact OVM sequence, the body task starts execution of the inFact graph. In most cases, the inFact graph executes for the duration of the simulation. Most inFact graphs contain an outer loop which is repeatedly traversed under control of inFact coverage algorithms until inFact stimulus coverage(s) are satisfied.

Once rules have been entered, the inFact IDE “generate” step creates OVM sequence template code. Most of the text in Figure 4 is generated by the IDE. Users typically customize the internals of the action tasks and augment the sequence class for their application using native SystemVerilog and OVM constructs.

BENEFITS OF USING INFAC T TO IMPLEMENT AN OVM SEQUENCE

Coverage closure issues found in many constrained random applications are resolved using inFact, due to a combination of factors:

- inFact uses a graph to systematically construct sequence_items thereby avoiding redundant generation of sequence_item data fields inherent in random methodologies.
- The inFact graph can be configured to mirror the covergroup architecture to eliminate test duplication. The result is that each inFact graph traversal hits new coverpoints or crosses without redundancy.

Figure 5 – inFact Rule Text to Graph Relationship

www.mentor.com
inFact typically achieves coverage closure 10x faster than constrained random by avoiding the redundancy of randomize(), and targeting specific coverage combinations specified in the rule graph.

inFact provides an option to simulate on multiple simulation nodes, non-redundantly, using its distributed simulation feature. This gives users the option to get simulation results upwards of 1000x faster for time-critical applications.

- No requirement to analyze coverage holes associated with blocks under inFact control. inFact’s stimulus coverage algorithms traverse the stimulus state space completely without redundancy.
- No need to write directed tests to improve coverage for inFact controlled blocks.

The significant coverage closure benefit is apparent when simulation results running the FPU constrained random and inFact OVM sequences are compared:

**CONCLUSION**

OVM applications having coverage closure problems can realize significant benefits using inFact. Orders of magnitude improvement in time-to-coverage closure is possible due to inFact’s non-repeating, stimulus generation algorithms that target user-defined covergroups.

OVM users can realize these benefits by selectively incorporating inFact-generated OVM sequences into their existing OVM environments. inFact provides an IDE that supports the automatic OVM sequence generation using code templates. Users customize the generated template code for their application by writing SystemVerilog code just like they do today.

These inFact benefits can be realized in other environments including SystemC, Verilog, SystemVerilog, and environments using “e” and Vera test components. The underlying inFact graph-based algorithms for stimulus generation can be applied in any of these environments and are best appreciated in environments where coverage is being measured and coverage closure an issue.
A Strong Foundation for Design, Verification, and Firmware Requires an Automation Tool for Register Management

by Francois Xavier Duthu @ Mentor Emulation Division, Mentor Graphics, France and Anupam Bakshi, Agnisys

INTRODUCTION

Register and Memory map management has become very important especially for complex ASICs and FPGAs. Gone are the days of describing each register by hand in various different places in the design cycle. Such an error prone and time consuming approach is now, thankfully, replaced by off-the-shelf automation. We automated our register management process using IDesignSpec (IDS). It has enabled us to describe the registers along with the functional specification in a document and using an editor plug-in automatically generated a variety of outputs. Further we were able to generate new outputs that fit precisely in our existing flows using TCL scripts, thereby completely automating our processes.

This article details experiences of using IDS for building a strong, error-free foundation for design, verification, firmware and not spending time and resources in doing it.

EMULATION SYSTEMS

Emulation systems have certain features that present unique opportunities as well as challenges. Our design is predominately software configurable, so we need to share a lot of data with the firmware team, software application teams, and keep all this data in sync with the design. Due to the scalable nature of these designs, they typically have a large number of configurable repeating structures.

Automation is absolutely critical for such designs. Earlier, each team had to write its own “data description” obtained from different specification, depending on their needs. We needed a way to not just describe our register and memory map but do so in a concise and effective way.

Our design group uses ‘generate’ constructs in Verilog to describe the repeating nature of our design components. We needed a system to automatically create code that looks similar to hand generated code such that the output is concise and not overly verbose.

Our firmware and software groups have a unique way to describe their register maps. We needed to automatically generate these file formats from a common source.

Our verification group has built their environment using OVM 2.0 and specifically used the recently released OVM register package 1.0. We needed a way to automatically generate those files as well.

In addition, we needed a way to share the documentation in our team as well as on our Intranet. Having a common repository was important as it meant that all teams were “on the same page”.

HARDWARE SPECIFICATION

We describe our hardware in a specification document. In this document, instead of describing the registers using normal document editor constructs such as formatted text and tables, we use the templates provided by IDS. Data from these templates is automatically picked up by the generator and appropriate outputs are produced.

Registers are described using a single template. A repeating structure is created with the RegGroup construct. The RegGroup construct consists of two templates, a starting and an ending template. The starting template contains properties such as name, address offset, external, size, and repeat. The contents of the RegGroup are repeated as many times as the value of repeat property. The contents can be either Registers or other RegGroups. Using these constructs, complex, multi-dimensional registers can be created quite naturally.

The inherent hierarchy of digital systems is captured using additional templates such as block, chip, board and system. These templates convert a linear document into a structured, hierarchical data model of the hardware suitable for downstream automation.

We used the ability to add custom register types using the “external” property on the templates. This enabled us to stitch in our own implementation of the register or group of registers and not take one generated automatically. This was also used to put memories in the register maps for which no RTL implementation was required.

GENERATING OUTPUTS

Having described the hardware registers in the form of a structured, hierarchical data model, we generated a variety of outputs for the various teams suitable for our existing tool flows.
RTL DESIGN

We use a proprietary bus protocol to access the registers. IDS automatically generated a slave. We created a simple bus bridge to interface with the slave. The automatically generated Verilog code uses “generate” constructs for repetitive structures. This makes it easier for humans to parse and use. The registers and RegGroups marked “external” got transformed into external ports on the slave. This enabled us to connect the appropriate memories and special register implementations that were too complex for auto-generation. This flexibility was really useful since we were not limited by any shortcoming in the tool.

OVM REGDEF

OVM register definition files generated by IDS are compatible with 1.0 release of the Register Package from Mentor Graphics and run on our Questa based verification environment. Enumerations and Constraints on the register fields are picked up from the document and automatically converted into OVM constructs.

Multi-dimensional registers are transformed into appropriate SystemVerilog and OVM register package constructs without much verbosity which has better run time performance.
CUSTOM C/C++ HEADER

We used a TCL script to completely re-design the outputs to conform exactly to our needs. IDS treats TCL transformations at par with other native transformations. Field enumerations, or the possible values that the register field can take, is very important to the firmware and device-driver groups. These are automatically picked up from the document and are available as C/C++ enums in the header files.

OUTPUT DOCUMENTATION

HTML and PDF documentation generated from the source was created for sharing on the Intranet. All teams have access to the same documents right from the start of the project and through the design cycle.

IMPROVED PROCESS

The much improved process now allows design, verification, firmware teams to collaborate where earlier they had their own process for dealing with the hardware specifications. Now they can all share a single data model for the hardware which gets transformed into a view that they precisely need. This has significantly reduced the chances of misinformation and omissions. It has removed the painstaking process of manually synchronizing the various files containing very similar register map data.

Figure 2: OVM Register Definition file

```verbatim
import ovm_pkg::*;
import ovm_register_pkg::*;

// block-name: CMEM

typedef struct packed {
    bit [31:23] padding23;
    bit Loop_pause;
    bit Trace_enable;
    bit [20:19] ECC_mode;
    bit [18:15] Byte_select;
    bit [14:13] Select_data;
    bit [18:15] Select_control;
    bit [10:8] Select_address;
    bit [5:0] Wait_states;
} CMEM_Scheduler_RAM_Scheduler_RAM_Reg_t;

class CMEM_Scheduler_RAM_Scheduler_RAM_Reg extends ovm_register #( CMEM_Scheduler_RAM_Scheduler_RAM_Reg_t);

covergroup CMEM_Scheduler_RAM_Scheduler_RAM_Reg_cg;
    Wait_states : coverpoint data.Wait_states;
    Operation : coverpoint data.Operation;
    Select_address : coverpoint data.Select_address;
    Select_control : coverpoint data.Select_control;
    Select_data : coverpoint data.Select_data;
    Byte_select : coverpoint data.Byte_select;
    ECC_mode : coverpoint data.ECC_mode;
    Trace_enable : coverpoint data.Trace_enable;
    Loop_pause : coverpoint data.Loop_pause;
endgroup

function void sample();
    CMEM_Scheduler_RAM_Scheduler_RAM_Reg_cg.sample();
endfunction

function new(string name, ovm_component p);
    super.new(name, p);
    CMEM_Scheduler_RAM_Scheduler_RAM_Reg_cg = new();
    WMASK = 'b0;
    RMASK = 'b0;
    WMASK.Wait_states = 'b111111;
    RMASK.Wait_states = 'b111111;
endfunction

class CMEM_register_file extends ovm_register_file;

CMEM_Scheduler_RAM_Scheduler_RAM_Reg CMEM_Scheduler_RAM_Scheduler_RAM_Reg_reg[2][256];
CMEM_Scheduler_Seq_ctrl CMEM_Scheduler_Seq_ctrl_reg[2];
CMEM_Scheduler_Seq_rst_epoch CMEM_Scheduler_Seq_rst_epoch_reg[2];
```
typedef enum __CHIPX_t_CMEM_map_area_offsets__ {
    CHIPX_CMEM_MAP_AREA_MAP_AREA_SCHEDULER_D0_OFFSET   = 0x1,
    CHIPX_CMEM_MAP_AREA_MAP_AREA_SCHEDULER_D1_OFFSET   = 0x2,
    CHIPX_CMEM_MAP_AREA_MAP_AREA_MEM_OFFSET                        = 0x4,
    CHIPX_CMEM_MAP_AREA_MAP_AREA_ERROR_OFFSET                   = 0x100,
    CHIPX_CMEM_MAP_AREA_MAP_AREA_MEM_OFFSET                        = 0x4,
    CHIPX_CMEM_MAP_AREA_MAP_AREA_TEST_OFFSET                        = 0x200,
    CHIPX_CMEM_MAP_AREA_MAP_AREA_ERROR_OFFSET                   = 0x100,
    CHIPX_CMEM_MAP_AREA_MAP_AREA_CONTROL_OFFSET              = 0x400,
    CHIPX_CMEM_MAP_AREA_MAP_AREA_DEBUG_OFFSET                   = 0x800,
    CHIPX_CMEM_MAP_AREA_NUMBER_OF_AREA_OFFSETS
} CHIPX_t_CMEM_map_area_offsets;

// Register: Filler_status
enum CHIPX_CONTROL_mapping_ADDRESS_register_FILLER_STATUS_bits_position {
    CHIPX_CONTROL_MAP_ADDRESS_REG_FILLER_STATUS_FILLER_SEQ_DONE_BIT_PO   = 0,
    CHIPX_CONTROL_MAP_ADDRESS_REG_FILLER_STATUS_FILLER_SEQ_RUN_BIT_POS   = 1,
    CHIPX_CONTROL_MAP_ADDRESS_REG_FILLER_STATUS_NB_OF_BIT_POS
};

Figure 3: Generated C/C++ header file

AREAS FOR FURTHER IMPROVEMENT

IDesignSpec automatically generates code for coverage, but, in the future, we hope to do more and have finer control over what is covered. We would like to have more control over the class inheritance in the files generated for OVM register definitions. We intend to make use of the IDSBatch program which will make the entire process make-based.

CONCLUSION

We were able to raise the abstraction for describing our hardware registers using IDesignSpec. This higher level of abstraction and automation allows us to keep various views of the hardware/software interface automatically synchronized. This has improved the time it takes to make a change in the specification and for that change to propagate through the system. It has also helped us improve the QoR by eliminating a whole class of bugs that could have potentially crept in, had we not used any such automation. One of the main advantages is output file customization using TCL script. It allowed a lot of flexibility as we have full control over what files get generated.

Modeling the core data in a target neutral and output neutral format has enabled us to “future proof” our design data. This has enabled us to build upon a strong foundation in our design, verification and firmware environments.

REFERENCES

Modern emulation systems: http://www.mentor.com/products/fv/emulation-systems/
IDesignSpec product details: www.agnisys.us/product/idesignspec.aspx
Support for OVM Register Package in IDesignSpec: http://www.businesswire.com/portal/site/home/permalink/?ndmViewId=news_view&newsId=20100128006745
A BRIEF HISTORY

In the early 1990s, simulation accelerators from IKOS and Zycad were prevalent on the verification landscape. Offering significantly better performance than software simulators, both companies carved out a profitable existence speeding simulation on large designs. But the rapid advance in workstation performance and simulation efficiency closed the gap, and by the early 2000, the accelerators had vanished. IKOS made the transition to hardware emulation with its popular VStation series. Zycad was unable to muster the financial resource to make the same leap.

Hardware emulators from Quickturn, IKOS and Mentor Graphics flourished at the turn of the century, but what really distinguished them from simulation accelerators? Two major factors come to mind. First, emulators didn’t try to be simulators which freed them to run much faster. They were 2-state machines that shunned compute intensive features like timing and fault analysis. Second, they severed ties to workstation-hosted testbenches in favor of running stand-alone (target-less) or wired to physical hardware.

EMULATION VS. ACCELERATION

Let’s begin with a clarification of terms. Hardware acceleration is custom hardware which executes a compiled RTL design and is interfaced to a workstation executing a testbench. Hardware emulation is custom hardware which executes a compiled RTL design and is cabled to external hardware. Mentor Graphic’s Veloce is a single product that performs both functions. The labels emulation and acceleration describe Veloce operating modes, not unique products or technology.

TESTBENCH LIMITS PERFORMANCE

In the past decade, hardware emulator performance has advanced from 100KHz to multiple MHz, so why has this not driven a resurgence in simulation acceleration. After all, what’s wrong with two-state unit-delay simulation? Isn’t this the basis of cycle-based simulation? And isn’t most timing analysis today done statically? So why not use emulation hardware to accelerate simulation? The remaining limiting factor is the workstation-hosted testbench. Emulators want to run free, unfettered by handshaking with a testbench on every edge of the clock. IKOS (subsequently acquired by Mentor Graphics) never abandoned their acceleration roots and have invested 100+ staff years in optimizing the testbench-emulator interface.

Figure 1: Accelerating just the design yields modest speedup. To justify the cost of emulation HW, the testbench must also be accelerated.

The two primary factors limiting simulation acceleration are testbench execution speed, and the workstation, emulator interface. The simulation profile in figure-1 shows the testbench consumes 30% of the CPU resource, reducing design evaluation to zero, implying an infinitely fast accelerator, yielding a mere 3X speedup. This is insufficient to justify the purchase price of a hardware emulator. It’s evident from figure-1 that the testbench must also be accelerated. The round trip latency of the PCIe workstation-emulator interface is ~7 us. A testbench driving the clock high and low across this interface encounters a 14us latency per clock cycle throttling a 1 MHz emulator down to 70 KHz. Considering the combined impact of testbench compute load and communication latency, the only viable solution is...
to shift the testbench-emulator boundary such that more of the load is carried by the emulator and less frequent communication is required.

Figure-2: Pin-level handshaking between the workstation and emulator throttles emulation performance by a factor of 10X or more.

THE EVOLUTION OF TESTBENCH XPRESS™

The first effort in this space sought to emulate the entire testbench, allowing the emulator to free run and eliminate most handshaking with the workstation. Mentor launched a project to synthesize the complete testbench onto the emulation hardware. This concept eventually faltered as some behavioral constructs could not be synthesized and users were unwilling to limit their testbench to those constructs that could be accelerated. The effort was not in vain however. While the entire testbench could not be synthesized, the project did result in shifting the boundary deep into the testbench, permitting acceleration of a significant portion of it. The resulting technology TestBench Xpress is the genesis of the transaction-based testbench upon which advanced methods like OVM and VMM are based. By synthesizing and thereby accelerating the generation of pin-level activity (read transactor), the testbench-emulator boundary was moved to a near optimal position. A testbench can be written in advanced constructs of VHDL and SystemVerilog, and the data they generate is passed to accelerated transactors in the emulator for expansion into pin-level activity.

Figure-3: Passing high-level, untimed transactions across the boundary to accelerated transactors shifts the burden of pin-level bus-cycle generation to the emulation hardware.

OVM TO THE RESCUE

While TestBench Xpress was a boon to simulation acceleration, its adoption was limited by user reluctance to duplicate testbench development effort. In the age of design and verification reuse, developing a second testbench to comply with TestBench Xpress met with limited acceptance. Enter the Accellera Committee and its drive towards the transaction-based Open Verification Methodology (OVM) and the sister effort on Standard Co-Emulation Model Interface (SCE-MI). With some minor enhancements, Testbench XPress is now OVM compliant and the reuse of a simulation testbench for emulation is a reality.

Figure-4: The Accelerated OVM Configuration can run in either native software simulation or emulation.
The accelerated OVM configuration (Figure-4) has two distinct roots: the testbench root for the OVM testbench components and TLM channels, and the top root containing the RTL DUT and accelerated transactors. The TLM channel (xi_tlm_fifo) object pointer is passed directly to testbench model (Stimulus), but the opposite endpoint of the TLM channel is actually a SCE-MI 2.0 transaction pipe instantiated in accelerated transactor (Driver). The accelerated OVM configuration can be run in either simulation or emulation without modification. This eliminates the need to maintain two versions of the testbench, one for simulation and another for emulation. It also reduces behavior discrepancies between the simulation and emulation environments minimizing the effort required to achieve testbench reuse. An added benefit is the freedom to develop and debug the testbench and transactors in simulation, a readily available and less expensive resource than the hardware emulator which is typically in high demand.

**ACCELERATED TRANACTORS**

An example of the division between the accelerated and simulated components of a transactor is depicted in figure-4 where an AXI bus transactor is partitioned across the two domains. The Accellera Standard SCE-MI-2 interface is used to bridge simulation and emulation. The pin-level bus-cycle generators are functional state machines which act to expand a single SCE-MI-2 transfer into a sequence of pin events. Had these state machines not been modeled in emulator, it would increase the simulator workload and multiply the amount of cross-domain communication. PCIe can efficiently move large blocks of data in a single transfer, but the latency of each transfer is high, thus the goal is to minimize the number of transfers while maximizing the content of each transfer. Ethernet represents the extreme of transactor expansion. An Ethernet packet can cross domains in a single SCE-MI-2 transfer, yet the accelerated functional state machine in the emulator will expand the packet header and payload into thousands of DUT clocks. With accelerated transactors, an OVM testbench can feed transactions to the emulator at a rate that keeps it running a full speed and eliminates the need to pause waiting for stimulus.

**SCE-MI-2 PIPES**

The Accellera SCE-MI subcommittee continues their focus on an efficient co-emulation interface, with the latest invention being SCE-MI-2 Pipes. Pipes, which are built on top of the SystemVerilog DPI standard and therefore run equally well in simulation, permit high-volume data transfer between the testbench and emulator, while minimizing performance-robbing handshaking. Many standard interfaces found in designs today support data streaming where many bytes of data can be burst through the interface to and from the DUT. While it does occur, interfaces which require you to write a byte then read a byte are in the minority. Pipes are efficient at bursting data to and from the design and support advanced concepts like variable-length messages and data shaping. In essence, Pipes present a simple interface to the testbench developer while shielding the complexity of optimizing the transfer between the testbench and hardware emulator. With pipes you don’t have to be an expert in emulation to write expert testbenches.
CONCLUSIONS

Hardware emulators have continued their march toward higher speed and greater capacity, but mating them with a traditional testbench resulted in unimpressive overall performance. To address this, Mentor Graphics invested heavily in TestBench Xpress, a transaction-based methodology that delivers excellent simulation acceleration on the Veloce emulator. Until recently, few verification groups embraced this method, but all that has changed with the arrival of OVM. Now this industry standard transaction-based method is being heavily promoted by the EDA industry and has gained broad acceptance, enabling a single testbench to be applied effectively to both simulation and emulation. OVM and TestBench Xpress unlock the power of Veloce hardware emulators and dramatically lower the cost of deploying hardware acceleration for functional verification.
Verification Management Eases Those Re-spin Worries

by Darron May, Product Manager, DVT Division, Mentor Graphics

When verification is not under control, project schedules slip, quality is jeopardized and the risk of re-spins soars. These less-than-stellar outcomes seem to be happening more and more often. First-time success with silicon is waning, down from nearly 40% in 2002 to less than 30% in 2007, according to an independent verification survey funded by Mentor Graphics. Re-spins are mainly due to functional or logical flaws in the design, which suggests an increasing number of problems in the overall verification management process. Among such problems: a dearth of tools that can allow the specification to drive the process and can manage the volumes of data generated during verification. What’s needed is a common platform and environment that provides all parties – system architects, software engineers, designers and verification specialists – with real-time visibility into the project. And not just to the verification plan, but also to the specifications and the design, both of which tend to change through time. There are three dimensions to any IC design project: the process, the tools and the data. Any comprehensive approach to verification management needs to handle them all.

DATA MANAGEMENT

Very roughly, the amount of verification data grows in proportion to the square of the gate count. So given the rise in design complexity, it’s no surprise that data management is increasingly the foundation of verification management activities. Questa’s verification management capabilities are built upon the Unified Coverage Database (UCDB), an API donated to Accellera by Mentor Graphics in 2008 and chosen as the basis of an industry standard. UCDB captures nearly any source of coverage data generated by an array verification tools and processes; Questa and ModelSim use this format natively to store code coverage, functionality coverage and assertion data in all supported languages. UCDB also allows for a tagging system linking test or verification plan objects to the broader coverage model, which makes it easier to report on coverage and even achieve
electronic closure. A small collection of utilities help handle data associated with a range of verification activities: merging, ranking, analyzing and reporting. And because of the open API, users not only have complete access to the data but also the ability to customize any aspect of the analysis and storage of such data.

PROCESS MANAGEMENT
Verification is driven by requirements concerning both the functionality of the final product and the intended methods of testing this functionality. By providing tools to import verification or test plans and then guide the overall process, Questa verification management helps deal with this complexity and shepherd a project toward electronic closure. It also provides the ability to store snapshots of data across the lifetime of a project, which helps to concentrate efforts where they are most needed.

TEST PLAN TRACKING
Projects are tracked in spreadsheets or documents created by a range of applications, from Microsoft Excel and Word to Openoffice Calc and Write. So it’s critical that a verification management tool be open to a range of file formats, a basic feature of Questa, which is built on the premise that a user should be able to use any capture tool to record and manage the plan. This document becomes the guide for the verification process and within Questa’s user interface the plan’s data can be sorted, filtered and subjected to complex queries such as which tests are most effective at testing this particular feature or which set of tests needs to be run to get the best coverage for a modified instance of the design. The tool allows the plan to be annotated with nearly any category of metadata, which can then be sorted (by engineer, project group, verification method, milestone, and so on) and tracked. This sorted data can then be shared in text or HTML reports, making it easier to allocate scarce resources and more accurately hit and manage deadlines.

TREND ANALYSIS
Understanding the progress of a dynamic verification process requires an ability to view coverage data. Accordingly, a verification management tool needs to provide the means to manage, view and analyze this data, whether it’s generated from a single test or the combination of a complete regression run. Just producing and managing individual snapshots of coverage data can be difficult due to the huge amounts of data involved. And these snapshots generally fail to give a good idea of progress being made over time, which depends on the ability to view and analyze data from multiple regression runs. UCDB affords this ability, reducing to a single database the regression data from multiple snapshots and then querying this database for trends. The merging capabilities are significant as these reduce the data stored by two orders of magnitude while at the same time giving full visibility on progress of all relevant aspects of the verification or test plan. The types of data recorded include that related to coverage of design units, instances of design units, functional coverage and the overall verification plan. Trend analysis is critical when using constrained-random verification techniques in which it’s important to be able to view coverpoints, crosses and covergroups in
real time to pinpoint where and when coverage has been hit. Seeds
and constraints that hit particular coverage targets at one point in
the project often prove ineffective later as the project progresses, a
frustration that can be at least somewhat mitigated by fine-grained
trend analysis.

TOOL MANAGEMENT

Verification management means balancing various tools and
techniques to get to closure, often with an infrastructure built on
home-grown scripting and lots of manual maintenance. And as
verification complexity ascends, so too does the need for a more
flexible automated solution.

VERIFICATION RUN MANAGEMENT

Questa’s verification run manager is one such solution, bringing
consistency to a project through heavy doses of automation. Among
the benefits of automation: improvements in time to coverage and
time to next bug, and also increased ability of dispersed project teams
to accurately estimate the time to completion. The run management
system records the definition of the tasks needed for verification, the
dependency between these tasks, and parameterization of the tasks,
all of which helps to automate decisions and proves to be especially
useful within a constrained random environment. Other features
that help gauge the effectiveness of the verification environment
include easy monitoring of regression completeness, simulation times
and test failures. Questa can control and monitor the launching of
regressions locally, on a network or further integrated with a load
sharing system such as LSF or Sungrid. Many regressions are run
overnight and the verification team does not want to waste time re-
running tests in the morning before being able to start debugging.
This is why any tool’s run system needs to ensure that all jobs are
complete and that jobs that fail as part of the regression are re-run
with full visibility, one of the many features of Questa’s verification
run manager. The tool also helps handle the tedious post-processing
chores of merging coverage, analyzing tests, managing the verification
plan and optimizing and ranking the regression tests. One example:
as simulations are complete and coverage results are available, they
are added to a merge queue where merge tasks are automatically
generated and deployed to combine the results from multiple jobs.
This increases efficiency by spreading the merging process, especially
when combining multiple coverage sets, and also makes coverage
data available sooner than the alternative of waiting for all tests to be
completed before beginning analysis.

VERIFICATION RESULTS ANALYSIS

Questa’s verification results analysis speeds the ability to address
failures identified during a regression, which helps a verification
project to stay on schedule. The technology brings together the results
of multiple verification runs, assisting in the grouping, sorting, triaging
and filtering of messages over the complete set of regression tests.
The results analysis can be triggered automatically and used by the
run manager, allowing the results of a given test to control if and
what should be saved in a triage database to allow further analysis.
Questa’s results analysis technology is flexible, enabling the user to
both identify parts of a debug or assertion message for storage and
also easily extract variable data from the messages into the triage
database. This extracted data can be used for grouping and filtering. For example: information about a protocol, various ports, transactions, and events occurring in the running of the verification tool can be sorted according to time, type of test or severity of message. This is a boon to failure analysis and debugging activities. Questa’s results analysis and triage capabilities can be used as standalone technology, though of course there are benefits to integrating with other Questa capabilities. One such benefit: speeding along the debugging process by providing the tool’s verification run manager with incremental data that can be analyzed to fix problems during regression testing instead of at the end of the regression cycle.

CONCLUSION

An effective verification management system requires technology to manage the process, tools and data. Such a system should be anchored in the verification plan, which in turn is closely linked to the design specification. And the verification management tool should enable electronic closure of the verification plan by providing tools that both reduce the volume of data and give deeper visibility and control. Questa’s verification management delivers all this in an environment that is modular, flexible and open. But perhaps its most attractive feature is that its various capabilities are assembled with one overarching goal in mind: to give IC design teams the best chance of producing silicon right the first time.
What’s New with the Verification Academy?
by Harry Foster, Chief Verification Scientist, Design Verification Technology, Mentor Graphics

These are exciting times for verification—and in particular, these are exciting times for the Verification Academy. This month we are launching our most ambitious release to date, which consists of the following new content:

1. We are augmenting our existing FPGA Verification module with three new sessions.
2. We are releasing a new module titled OVM Basics.

Now, as shown in Table 1, the Verification Academy covers a wide variety of topics, which enables you to start evolving your advanced functional verification skills.

What’s particularly exciting about this release is that we are addressing the numerous requests from you for more SystemVerilog testbench examples. The requests we are hearing for more SystemVerilog should not be a surprise, given the significant industry interest in SystemVerilog, as shown in the figure on the following page.

This month we are delivering on your request, starting with three new sessions that we added to the FPGA Verification Module. These sessions are presented by the subject matter expert, Ray Salemi, and they provide numerous SystemVerilog working examples. For those of you who don’t know Ray, he is author of the book FPGA Simulation, which forms the technical foundation for this module.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Number of Sessions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evolving Capabilities</td>
<td>1</td>
<td>Provides a framework for all the Verification Academy modules, while introducing a tool for assessing and improving an organization’s advanced functional verification capability</td>
</tr>
<tr>
<td>Assertion-Based Verification</td>
<td>11</td>
<td>Provides a comprehensive introduction to ABV techniques, including an introduction to SystemVerilog Assertions</td>
</tr>
<tr>
<td>CDC Verification</td>
<td>7</td>
<td>Provides an understanding of the clock-domain crossing problems in terms of metastability and reconvergence, and then introduces verification solutions</td>
</tr>
<tr>
<td>FPGA Verification</td>
<td>8</td>
<td>Although targeted at FPGA engineers, this module provides an excellent introduction to anyone interested in learning various functional verification techniques</td>
</tr>
<tr>
<td>OVM Basics</td>
<td>8</td>
<td>Provides a step-by-step introduction to the basics of OVM</td>
</tr>
</tbody>
</table>

Table 1. Verification Academy Modules

The new FPGA Verification sessions include:
1. Test Planning—an easy, low-investment approach to creating a basic functional verification test plan
2. Transactions—an introduction to creating a more efficient testbench by using transactions
3. Functional coverage—a step-by-step introduction to building coverage models using SystemVerilog covergroups

In addition to more SystemVerilog examples, we also have had numerous requests for basic training on the Open Verification Methodology (OVM). So we are excited to announce our new module titled OVM Basics. This module is delivered by the subject matter and training expert John Aynsley of Doulos. If you’ve never heard John present, then you are in for a real treat. John is both entertaining and informative.
The new OVM Basics module consists of 2.5 hours of content, and is divided into eight 20-minute sessions. The module is primarily aimed at existing VHDL and Verilog engineers or managers who recognize they have functional verification problems but have little or no experience with constrained-random verification or object-oriented programming. In releasing the OVM Basics module, our goal is to raise your skill level to the point where you have sufficient confidence in your own technical understanding. In turn, this confidence will position you to start the process of adopting advanced functional verification techniques.

I encourage you to check out all our new and existing content at the Verification Academy by visiting www.verificationacademy.com.
INTRODUCTION:

The SystemVerilog functional verification community has two open-source libraries and accompanying methodologies: the Open Verification Methodology (OVM) and Verification Methodology Manual (VMM). Both OVM and VMM are intended to help designers more quickly develop verification IP (VIP) that is inherently interoperable with other IP developed in a homogenous environment. Even with these libraries’ benefit, developing a typical VIP from scratch is still time-consuming and error-prone, and integration of immature components injects significant risks to a project. To mitigate those risks as much as possible, the verification architect is compelled to find and integrate existing VIP. The choices can be very limiting if one is forced to work in homogenous environments—OVM or VMM, but not both. So, when Accellera VIP Technical Steering Committee released a beta version of the OVM-VMM Interoperability Kit [1], it theoretically expanded the pool of VIP available for integration in a functional verification environment. This paper describes the effort and results of testing that hypothesis in the real world.

Integrating OVM VIP in a VMM testbench presented several challenges, some of which were not directly addressed in the Accellera VIP Interoperability Kit.

• instantiating and building the component within the testbench
• coordinating different simulation phases
• configuring components to operate properly in the desired context
• orchestrating and coordinating stimulus and other traffic between components
• passing data types between components
• distributing notifications across the testbench
• issuing and controlling messages

This paper will show how we used the VIP Interoperability Library to integrate our OVM-based Multiview Verification Components (MVCs), such as AHB, USB, I2C, and UART, into existing VMM environments. Here, we utilized interoperability adapters such as channel-to-TLM, analysis-to-notify, analysis-to-channel, and others. We also found it essential to group the foreign VIP and adapter(s) into a single container component such that we exposed only the interfaces and data-types that were native to enclosing environment. All data-type and interface semantic conversions are performed locally inside the container by the interoperability adapters and application-specific “glue code.” We have first encapsulated all lower components and created a top OVM component, and then this OVM component along with VMM channels and avt analysis channel are instantiated inside one top level container. Figure 1 and 2 illustrates using this technique for integrating foreign VIP.
Figure 2. Container class that instantiates the wrapper class, VMM channel and avt_analysis_channel. To connect the OVM analysis port to the VMM channel, the OVM transactions will be converted to a VMM transaction class and then placed in the VMM channel through the avt_analysis_channel so that it can be sent to the rest of the VMM environment.

And finally this component will be instantiated in VMM sub-environment and its VMM channel will be connected with outer worlds VMM channel for outer communication.

Figure 3. Communication with the outer world. The OVM component described in Figure 2 can then be instantiated like any vmm_xactor and its vmm_channel can be connected to outer world’s vmm_channel.

Below are the major steps for making use of the kit to integrate OVM-based VIP into a VMM environment. Among these steps are:

- Create static functions to convert OVM-based transaction classes to VMM-based transaction classes and vice versa.

```verilog
class my_i2c_converter_vmm2mvc;
typedef my_vip_i2c_slave_transaction my_vip_i2c_slave_transaction_t;
......
static function i2c_master_i2c_data_transfer_t convert_master (my_vip_i2c_master_transaction_t from, i2c_master_i2c_data_transfer_t to=null);
......
case (from.type_enum)
READ: convert_master.RnW = 0;
WRITE: convert_master.RnW = 1;
GENERAL_CALL: convert_master.request_type = I2C_GENERAL_CALL ;
endcase
convert_master.wdata = from.m_bvvData;
convert_master.nAck_byte = from.m_nNumAcks;
if (from.m_enStopEnable = VMT_BOOLEAN_TRUE)
convert_master.stop_tran = 1;
else
convert_master.stop_tran = 0;
endfunction : i2c_master_i2c_data_transfer_t
endclass : my_i2c_converter_vmm2mvc
```

- Create VMM channels and use the adapter to communicate VMM data instead of OVM transaction.

```verilog
class DmaAhbXactor extends ovm_component;
typedef vip_wrapper #(1,1,1,1,32,64,64) vip_wrapper_t;
vip_wrapper_t vip_top;
MemModel sysMem;
typedef avt_analysis_channel
#(ahb_single_transfer_t,dw_vip_ahb_transaction_t,
ahb_rw_convert_ovm2vmm)
) ahb_analysis_channel;
ovm_analysis_port #(ahb_single_transfer_t)
top_wrdata_ahb_single_transfer_analysis_port;
```

www.mentor.com
ahb_analysis_channel ap_adapter;
analysis_fifo #(ahb_single_transfer_t) ahb_single_transfer_analysis_fifo;
vmm_channel_typed #(my_vip_ahb_transaction_t) monOutChan;

virtual function void build();
ap_adapter = new("ap_adapter",this,monOutChan);
if (monOutChan == null)
  monOutChan = ap_adapter.chan;
endfunction

endclass

• if the OVM VIP requires configuration, extract the configuration from the VMM environment during the configuration phase of VMM and pass the configuration to the OVM VIP
• translate analysis ports to VMM notifiers as required
• translate analysis to the VMM channel as required
• if OVM configuration is used in multiple OVM-based components, use set_config_* mechanism

set_config_object("i2c_master_agent",mvc_config_base_id,i2c_config_0,0);
set_config_object("i2c_slave_agent[0]",mvc_config_base_id,i2c_config_1,0);

• call ovm_build method from the build method of VMM env.

function void build();
super.build();
  vmm_note(log," Entering into the build() method of environment");
...
ovm_build;
endfunction

Figure 4 provides a block level view of the VMM testbench environment which uses an OVM-based MVC and VMM-based verification components.

SUMMARY OF OUR RESULTS:

This paper describes how we applied and tested the Accellera VMM-OVM interoperability kit on a real-world, mixed-methodology verification environment. We found that by using the library we could indeed bridge the gap between two independently developed verification methodologies: OVM and VMM. The verification architect can now easily and efficiently mix methodology testbench environments, thus maximizing the pool of VIP available for integration, which helps mitigate the risks and costs associated with creating new verification components.

REFERENCES:

2. OVM Users Guide, available: www.ovmworld.org/resources.php (registration required)
Converting from VMM to OVM: A Case Study
by Damien Melville, Chipright Ltd

ABSTRACT:
Following several requests from our customers for an OVM type testbench it was decided to put a case study together to highlight the steps we took to convert our existing VMM testbench solution to OVM. This would give engineers a good understanding of the conversion process.

However, the conversion raised a couple of initial questions:

• Could the same VMM Testbench and directory structure be reused in an OVM environment?
• How easy would this be given that they are two different verification methodologies?
• How easy would it be to develop an OVM testbench solution from scratch if reuse of our VMM structure was not an option?
• Couldn’t we just use an OVM wrapper and run VMM below the surface?
• How compliant would OVM be with Synopsys VCS users?

In this article, we discuss how these questions were answered and the final approach that we took to develop our OVM solution. We conclude by highlighting our results from the conversion and describing the benefits from having an OVM testbench solution in place. For the purposes of explanation we have used the UART as an example in the following code snippets.

VMM TO OVM CONVERSION OPTIONS:
After initial discussion it was realized that we had 3 options when looking to create an OVM environment solution. These were:

• To utilize the VMM kit provided by Mentor Graphics so that VMM designs could run on Questa when under an OVM wrapper. Conclusion: seemed like a good strategy but we wanted a complete OVM solution
• Build the OVM model from scratch which meant a complete rethink of the structure of the environment. Conclusion: Good solution but we wanted to reuse our current structure
• Use the original VMM structure which had been successful and would require little or no ramp up in coming to terms with the verification environment and testcase creation

In the end it was decided that the best approach was to use the original verification structure that was in place for our VMM architecture. Of course this led to more technical issues.

• Could the VMM components be replaced by corresponding OVM components?
• Could we re-use channels for passing data between sub-blocks?
• Would the VMM data class and direct testcase generation be affected?
• Could configurability be maintained?
• What about Randomization?

CURRENT VMM TESTBENCH STRUCTURE:

Figure 1 VMM Block Level Testbench

Figure 1 illustrates our current VMM environment structure containing:

• Transactors with a standardized architecture
• An interface with a standardized implementation with clocking blocks and modports
• A sub environment level for use in system level testbenches
• A local testbench environment encapsulating test case infrastructure and mechanisms including a regression flow
• A base data class for creating data members for use inside the transactor
• A peripheral class (Bus Functional Model) for describing the transactor functionality
• A configuration class for configurability e.g. Master / Slave options
• A directed test case constructs class for dedicated test case commands / functions
• An environment class which contains the Interface and is a target for re-use in other VMM testbenches

Our aim was to maintain the above verification environment and port this to OVM

Step 1: Environment conversion

The first step of this conversion was to recreate the VMM top level environment in OVM. The interface and environment files were easily interchangeable between VMM and OVM therefore allowing us to keep the same top level architecture as previously. With a couple of base component changes (e.g. VMM_env -> OVM_env) and using methodology specific macros and functions it was quite straightforward to recreate the top level OVM environment files. The next step was to look at the VMM transactors and see if a corresponding OVM component could replace these seamlessly.

Step 2: Using the OVM Agent

We have seen how our original VMM environment and interface files were easily ported to OVM giving an environment structure as can be seen in Figure 2 above. So what about the other components of the environment, specifically within the local environment?

OVM recommends that environment developers create a more abstract container called an agent to encapsulate a driver (an active entity that emulates logic that drives the DUT), sequencer (an advanced stimulus generator that controls the items that are provided to the driver for execution) and monitor (a passive entity that samples DUT signals but does not drive them). Open Verification Components (OVCs) can contain more than one agent and can be split, for example, into a slave and master agent with one being set to active and the other set to passive as required. As stated in the OVM user guide ‘active agents emulate devices and drive transactions according to test directives. Passive agents only monitor DUT activity’. In VMM you may also create this extra layer of hierarchy but in our case this wasn’t so.

For our particular VMM testbench we implemented both master and slave functionality in one BFM. For OVM this was split between the two agents with each being configured as required by the verification environment. Below is a typical OVM master agent. We can see how the driver, sequencer and monitor are referenced.

```
class uart_master_agent extends ovm_agent;
protected ovm_active_passive_enum is_active = OVM_ACTIVE;
protected int master_id;
uart_master_driver driver;
uart_master_sequencer sequencer;
uart_master_monitor monitor;
`ovm_component_utils_begin(uart_master_agent)
`ovm_field_enum(ovm_active_passive_enum, is_active, OVM_ALL_ON)
`ovm_field_int(master_id, OVM_ALL_ON)
`ovm_component_utils_end
endclass : uart_master_agent
```

Figure 3 OVM Master Agent

Step 3: VMM_xactor to OVM_component conversion

The OVM_component, from definition, forms the foundation of the OVM. It encapsulates the behaviour of drivers, scoreboards, and other objects that are needed in a testbench. Similarly the VMM_xactor encapsulates these same objects. By definition, it encapsulates models that act on the data transactions that travel through the verification environment.

As was seen from Step 2 the BFM (VMM_xactor) was split into master and slave driver components of an OVM environment (OVM_driver). These would then be encapsulated by a master or slave agent
depending on the type of interaction with the DUT. See previous description of the OVM_agent.

**Step 4: VMM_channel Vs OVM port connections**

Data transmission and handling on the other hand was something that needed to change a bit more. In VMM we were used to using channel implementations for data handling. We needed to declare the channel type within the class definition, construct it, and determine if data was available to be read from the channel or written. For example:

```plaintext
declaration:   uart_data_channel in_chan;
construction: if (in_chan == null) in_chan = new("Transaction", name);
Write:            this.wait_if_stopped_or_empty(this.in_chan);
this.in_chan.activate(current_transaction);
this.in_chan.start();
// Write data
this.in_chan.complete();
this.in_chan.remove();
```

**Figure 4 VMM Channel implementation**

For an OVM implementation this was treated differently. At the agent level, for both a master and slave, the driver and sequencer are connected, as can be seen in Figure 5, within a connect() function.

```plaintext
// Connection at agent level
driver.seq_item_port.connect(sequencer.seq_item_export);
```

**Figure 5 OVM driver and sequencer connection**

Derived driver classes use 'seq_item_port' to request items from a sequencer and 'seq_item_export' provides access to the sequencer's implementation of the sequencer interface as can be seen from the code snippet in Figure 6. In essence this implementation operates in a producer consumer format. The sequencer produces data and the driver consumes this data and sends it on to the DUT. The library of sequences produced by the sequencer and consumed by the driver is detailed within the sequence library class which is detailed in the next section.

**Step 5: VMM data class Vs OVM sequencer library**

Our base data class (VMM_data) as seen in Figure 1 VMM Block Level Testbench, would now become a sequence item in OVM. This sequence item could then be used within the driver, scoreboard or monitor components of the OVM environment. Figure 7 and Figure 8 show the original VMM data base class and new OVM data class (sequence item) respectively. The directed testcase generator (dtst) in our VMM environment was replaced by a sequence library of classes used by the sequencer to generate data for the driver to pass to the DUT in directed testcase mode. A code snippet from the VMM directed testcase generator is shown in Figure 9 along with one sequence from the OVM sequencer library in Figure 10.

```plaintext
class uart_data extends vmm_data;
    rand bit [63:0] data_tx;
    rand bit [47:0] data_rx;
    // Optional constraints on data
    constraint test {}
    `vmm_data_member_begin(uart_data)
    `vmm_data_member_scalar(data_tx,DO_ALL)
    `vmm_data_member_scalar(data_rx,DO_ALL)
    `vmm_data_member_end(uart_data)
endclass : uart_data
```

**Figure 7 VMM data class**
class uart_data extends ovm_sequence_item;
    rand bit [7:0] data_tx;
    rand bit [7:0] data_rx;

    // Optional constraints on data
    constraint test {}

    `ovm_object_utils_begin(uart_data)
    `ovm_field_int (data_tx, OVM_ALL_ON)
    `ovm_field_int (data_rx, OVM_ALL_ON)
    `ovm_object_utils_end

    extern function new(string name="uart_data");
endclass : uart_data

Figure 8 OVM Sequence item

class cl_uart_dtst extends vmm_xactor;

......

......

// Create the method to facilitate directed data stimulus

function void uart_send_tx_data ( logic [7:0] i_data1,
    int transaction_id = -1);

int status;
status = randomized_obj.randomize() with
{data_tx == i_data1;};
randomized_obj.data_id = transaction_id;
send_obj(randomized_obj);
endfunction

endclass

Figure 9 VMM directed testcase generator class

class uart_send_tx_data extends ovm_sequence #(uart_data);
    rand int unsigned count;
    `ovm_sequence_utils_begin(uart_send_tx_data, uart_master_sequencer);
    `ovm_field_int(count, OVM_ALL_ON)
    `ovm_sequence_utils_end

    function new(string name="UART send tx data");
        super.new(name);
    endfunction

    constraint count_ct {count==10;}
    rand bit [7:0] data_tx;

    virtual task body();
    repeat (count) begin: repeat_block
        `ovm_do_with(req,{req.data_tx>0;req.data_tx<112;})
        get_response(rsp);
    end: repeat_block
endtask: body

endclass : uart_send_tx_data

Figure 10 OVM sequence library

Step 6: VMM Configuration class Vs OVM set_config_int, get_config_int

Calling set_config_* in OVM causes configuration settings to be created and placed in a table internal to a specific component. There are similar global methods that store settings in a global table. Each setting stores the supplied inst_name, field_name, and value for later use by descendent components during their construction. (The global table applies to all components and takes precedence over the component tables.) When a descendant component calls a get_config_* method, the inst_name and field_name provided in the get call are matched against all the configuration settings stored in the global table and then in each component in the parent hierarchy, topdown. Upon the first match, the value stored in the configuration setting is returned. Thus, precedence is global, following by the top-level component, and so on down to the descendent component’s parent. These methods work in conjunction with the get_config_* methods to provide a configuration setting mechanism for integral, string, and ovm_object-based types.

Therefore from the description above, in OVM, during the build phase a component uses the get_config interface to obtain its configuration settings and can also use the set_config interface to set any configuration for its children.
In terms of our VMM to OVM conversion this was just a matter of utilizing the `set_config_*` and `get_config_*` constructs instead of the configuration class instantiation as done in VMM.

Figure 11 and Figure 12 below show how the `set_config_*` and `get_config_*` were used for our testbench configuration in OVM.

```cpp
// Code snippet from direct_te0 which is an extension of the local environment
function void direct_te0::build();
    super.build();
    set_config_int("*", "recording detail", OVM_FULL);
    set_config_int("*", "duplex_mode", uart_cfg::HALF_TX);
    set_config_int("*", "rx_mode", uart_cfg::RX_NONE);
    set_config_int("*", "char_size", uart_cfg::EIGHT_BIT);
    set_config_int("*", "nbr_stop_bits", uart_cfg::ONE);
    set_config_int("*", "uart_rx_enable", uart_cfg::MBOX);
    set_config_int("*", "tx_mode", uart_cfg::TX_MASTER);
    set_config_int("*", "uart_sync", uart_cfg::OFF);
    set_config_string("*", "default_sequence",
        "uart_send_tx_data");
endfunction: build
```

**Figure 11 Setting configuration options, OVM**

```cpp
// Code snippet from master driver
task automatic uart_master_driver::run();
    get_config_int("tx_mode", tx_mode);
    get_config_int("duplex_mode", duplex_mode);
    get_config_int("rx_mode", rx_mode);
    get_config_int("parity_mode", parity_mode);
    get_config_int("char_size", char_size);
    get_config_int("nbr_stop_bits", nbr_stop_bits);
    get_config_int("uart_rx_enable", uart_rx_enable);
    get_config_int("uart_sync", uart_sync);
endtask: main.
```

**Figure 12 Getting Configuration options, OVM**

So far we have replaced the VMM environment files with their OVM equivalents. We have also replaced our VMM base data class (VMM_data) with the ovm_sequence_item. Our directed testcase transactor class (dtst) has also been replaced by an OVM sequence library of data classes. The BFM (peripheral class) has been split into a master / slave combination and has been encapsulated by an agent as described previously.
**Step 7: VMM atomic generator Vs OVM random sequence:**

Within the VMM data class a macro exists to create an atomic generator channel of type specified by the data class for random transactions, as shown in Figure 15. In OVM this can be achieved by setting the default sequence to ‘ovm_random_sequence’. This sequence randomly selects and executes a sequence from the sequencer’s sequence library. The ‘ovm_random_sequence’ class is a built-in sequence that is preloaded into every sequencer’s sequence library with the name ‘ovm_random_sequence’. This is shown in Figure 16.

```
class cl_uart_data extends vmm_data;

------------------
endclass: cl_uart_data

`vmm_atomic_gen(cl_uart_data, “UART Atomic Generator”)
```

*Figure 15 VMM Atomic generator creation*

```
class direct_te0 extends uart_local_env;

...
endclass: direct_te0

function void direct_te0::build();
  super.build();
  set_config_string("*", "default_sequence", "ovm_random_sequence");
endfunction: build
```

*Figure 16 Setting OVM default random sequence*

**Step 8: Regression / Coverage**

The regression mechanism that we used for our VMM environment was very portable to the new OVM environment. We utilized a method of ‘tcl’ scripting to regress all of the direct and random testcases with a PASS / FAIL result for each testcase and a PASS / FAIL result for the overall regression. As ‘tcl’ is environment independent our regression mechanism worked for both VMM and OVM.

The coverage point and coverage group setup is independent of verification environment so was ported to OVM seamlessly from VMM and will not be discussed in this article.

**TOOL ISSUES: SYNOPSYS VCS, MENTOR QUESTA**

When initially looking to work with OVM we were concerned that the methodology may not be usable across all tools. We have successfully compiled and simulated the OVM (version 2.0.2) conversion on Mentor Graphics Questa version 2009.110 and Synopsys VCS C.2009.06-1

**CONCLUSIONS AND RESULTS**

The objective of this article was to outline the steps involved in converting a tried and tested VMM testbench environment to OVM. From these 7 simple steps we can conclude that the conversion task is not as daunting as initially thought. The main areas of change were in the data handling (step 4), the use of the OVM agent (step 2), configuration (step 6), and the sequence library (step 5). Once these steps were taken we were well on our way to completing a successful conversion from VMM to OVM.

**ABOUT CHIPRIGHT:**

Chipright is an innovative engineering organization and industry leader in the provision of electronic design and verification consultancy. The company provides best in class products and services to the global electronics industry. We have been involved with the latest verification trends and methodologies since inception, including VMM and OVM, and have successfully helped clients to adopt new environments and migrate between environments.

We are experts in developing rapid prototype verification flows using technologies based upon SystemVerilog (OVM, VMM). Our highly skilled engineering team specializes in developing solutions with constrained random verification, verification IP library modeling, bus functional models (BFM’s), standardized regression flows, functional / code coverage and assertion writing. We pride ourselves in educating client engineers to be OVM and/or VMM enabled with a standardized test bench development infrastructure. More information on Chipright can be found at www.chipright.com
INTRODUCTION

There are opportunities for positive transformation in any IC development team. While people and process related initiatives can have a tremendous complementary impact to the technical advancements already occurring in IC development, their value is often overlooked. Technology driven advancements such as coverage driven verification and the proliferation of OVM as a functional verification platform have been critical to keeping pace with exploding design size and complexity. However the difference between success and failure will frequently depend on how effectively people can embrace these new technologies, and what processes are put in place to ensure their optimal use. The technology itself is rarely the decisive factor for a successful or failed product delivery.

In the world of Agile software development delivering value to a customer is the primary objective, and teamwork and efficient processes are the essential components. This article identifies the characteristics common to effective agile development teams, and how a team that embraces these characteristics—in combination with using cutting edge technology—can accelerate their schedule, improve their product quality, and increase customer and employee satisfaction.

PROVIDING VALUE TO A CUSTOMER

Providing value to a customer early and often is the primary objective of an Agile team.

The CEO and sales force probably say them but how often are the words “value” and “customer” used together in ASIC development? While a team might realize who the customer is, the concept of value is probably not well understood nor used as a basis for discussion during development.

Too often, teams are concerned with planning and executing according to a requirements specification. That is important, but creating a plan and then executing the plan cloud the purpose of a project. The purpose of any project is to provide value to a customer. If—or more likely is when—your plan has to change to provide value to a customer, change it. A team that understands the needs of a customer and are concerned with meeting those needs will have happy customers. And as much as it hurts sometimes to make customers happy, realize that a company cannot survive without them.

The entire team needs to understand the thoughts and objectives of its customer. Everyone needs to understand what they are building and why they are building it. If a team is constantly asking itself, “Is this good for the customer?”, the resulting product is more likely to satisfy your customer’s needs.

Most Agile methodologies advocate regular interaction with the customer to ensure a product vision is relayed directly to the development team. If possible, it is even recommended that the customer be onsite and available throughout development, so that the feedback loop is as short as possible.

VISIBILITY AND EFFECTIVE COMMUNICATION

IC development teams, in general, rely heavily on written documentation, bug tracking systems and other formal means for communication. Most ignore the merits of communication in other forms; and unfortunately, teams choose to ignore its most effective form: verbal communication.

A Picture or 1000 Words?

Documentation is necessary. However, it should have three key elements to be truly effective: correct, complete, and concise. While written document can be correct and complete, it is sometimes difficult to be concise.

Software engineers have evolved diagrams that can concisely and accurately design intent using graphical capture. The most advanced design capture method is using Unified Modeling Language (UML). Using UML engineers can capture requirements, architecture, synchronization, and timing. From the saying “a picture speaks a thousand words”, a UML diagram can significantly reduce your project’s documentation requirements. The underlying rules, enforce a UML to be precise—reducing the likelihood of ambiguity common in the written text.

As “Agile” also implies ability to adapt quickly, it is generally faster to update a set of UML diagrams for any major architectural changes over a written text.
Visibility and effective communication are also characteristics of an Agile team. Verbal communication is most effective for conveying intent so an Agile flow creates frequent opportunities to ask directly for information or clarification. Arrange your workspace so that conversation between team members is easy and convenient. When cube walls get in the way, try removing them! Co-locating a team in a single room is also very common for Agile teams (Beck 2004) to ensure people and information are always close at hand. Cockburn 2007 recommends a one bus length rule between team members. Any further than that and meaningful communication degrades drastically.

When a conversation is not possible or appropriate, the next best thing is communication that is as interactive as possible in order to keep response times short (Cockburn 2007). Keep in mind that telephone, Instant Messenger and the like are generally more effective for communicating intent than email, written documentation or bug tracking systems. Where information is required by the entire team, use forms that are as visible as possible (e.g. Intranet wikis).

**REGULAR REFLECTION**

The end-of-project post mortem is an ineffective technique for critiquing the qualities and actions of the team that have either made it successful, lead it to failure, or left it somewhere in between. A post-mortem may lead to corrective action, but it cannot be applied until the next project.

A better approach is to establish a review process at regular intervals throughout the project to measure and critique a team's progress. If required, a team can take corrective action to the benefit of their current project, as well as improving the process for future projects. Agile teams continuously reflect on their product quality and process efficiency.

---

**THE CROSS-FUNCTIONAL IC DEVELOPMENT TEAM**

IC development teams are typically divided into functional sub-teams. Design, verification, physical design and software are the most common functional splits. Between them, there are usually spikes of cooperation during times of bug fixing and crisis management. Cooperation also tends to increase as a project nears conclusion. These functional sub-teams, however, usually set their own priorities and work toward their own goals on a daily and weekly basis. And while they do not operate in complete seclusion from each other, it is fair to say that meaningful cooperation is relatively limited during normal development.

A critical feature of an Agile team is its cross-functional composition. While each team member has an area of expertise, there are no formal lines drawn based on function. The entire team works toward common goals on a daily basis and collectively the team assumes a “get it done” attitude. The team’s common goal of delivery at regular intervals brings steady motivation for regular cooperation. It also creates a penchant for crisis aversion that is not present between independent sub-teams. Agile teams use the most effective means of communication possible to convey and clarify intent—which is often verbal communication—and reach beyond their area of expertise when necessary.

In a time where specialization and organization into functional sub-teams is the norm, IC development teams stand to benefit from working as a cross-functional team with a renewed emphasis on shared goals and cooperative problem solving.

---

Many agile teams use UML to capture design intent. While there many free tools that allow UML capture, they lack the “flow” connected software to hardware. Mentor’s BridgePoint xtUML provides a consistent approach that can be used by an IC development for embedded systems. BridgePoint goes beyond that static capture of UML, and provides the ability for your software team to accurately model and test your system and provide the ability to dynamically analyze the system.

Software + Hardware = System

In SoC development, embedded software is a functional team that often operates independently of design and verification. However, because the software team typically acts as proxy for the end user, it is better to have them involved in specification and implementation of the hardware throughout development. One way to involve the software team earlier is to build an ESL model.

While one benefit of ESL modeling is to enable parallel development of software and hardware, in many cases it only encourages the formation of yet another independent team: the modeling team. We believe that simply adding another team to enable parallel development without changing how teams interact provides little benefit.
INCREMENTAL DEVELOPMENT AND THE STEEL THREAD

Incremental development is an approach where functionality is built and tested as a thin slice through the entire development flow, start to finish. A product starts as a small yet functional subset and steadily grows as features are completed. Progress is based on code that works rather than code that’s written. Working code is a great metric for measuring real progress and demonstrating and clarifying implementation decisions.

One barrier to incremental development is there is often a considerable amount of effort required to identify, implement and test a feature that performs some significant functionality and provides some recognizable value to a customer (Andrea 2001). Such a feature is sometimes called a “steel thread” because it often represents a very thin but significant end-to-end path through a device.

It is likely that a team new to incremental development not only lacks the skill to scope and plan a first increment, but that the sheer number of tasks spawned in the first increment can be difficult to coordinate effectively. It is important, therefore, that the development team does what it can to prepare for the challenge.

Most IC development teams are familiar with the concept of “basic sanity”. Generally speaking, the basic sanity milestone is the first time an IC performs some simple function of recognizable value. Therefore, basic sanity has similar functionality to that of a first increment—or the first “steel thread” for the project. That value in this “steel thread” is apparent not only to the development team, but quite likely to other stakeholders, including the customer.

An effective way to mentally prepare for a first increment is to identify the simplest set of functions through a retrospective exercise where the criteria for basic sanity of a legacy product are analyzed. Similarly, for new products decide on the minimum set of functions that would demonstrate “basic sanity”. Any logic that is not specifically required for basic sanity should be ignored. In cases where unnecessary functionality cannot be completely ignored, the team should understand where and how it may be simplified (Schwaber 2004). After identifying the minimum required for the basic sanity, the team can go through a planning exercise to understand how they could implement and test basic sanity functionality as fast as possible. This defines the goal and tasks for steel thread for the first iteration.

With the completion of the first steel thread, incrementally adding functionality should be a more straightforward assignment because the steel thread gives the team a functioning starting point from which they can confidently add new features. No doubt though, the first and all subsequent incremental improvements will require practice, planning and great communication from a cross-functional team to be successful!

Incremental Development with OVM

Verification in incremental development—verifying a steel thread in particular—requires two criteria for success:

- a flexible environment that is easily deployed
- functionality that is easy to build and integrate

A verification environment based on OVM can satisfy both these criteria.

To start, the ovm_component class provides a defined, but extensible run-flow. Very little effort is required to have an operational—albeit non-functional—verification environment.

In terms of functionality, OVM provides mature, reusable verification components allowing a team to quickly build functionality. Verification component base classes for monitors (ovm_monitor), scoreboards (ovm_scoreboard) and sequence

We are suggesting a different approach where software, modeling and hardware experts work together to incrementally build a system. As the system architect defines features, the modeling expert incrementally populates the ESL model with the logic required for each feature. The model can then be leveraged by the software experts—for driver or application development—and the design and verification experts—for implementation and test of the hardware.

Leveraging the model twice is only a byproduct of this approach. The real benefit is it creates a tight feedback loop between each group of experts and regular opportunities to confirm design intent. The entire team runs in lockstep throughout development; guided by the system architect and the ESL model. There is constant collaboration between hardware and software where cooperatively they implement and test the vision of the system architect. The architect can see the end product incrementally grow in functionality and use the functioning system to refine his/her vision.

These feedback loops and cooperation cannot reasonably exist without a cross-functional team armed with tools that move focus toward the system and away from its pieces.
generators (ovm_sequence) provide the means to quickly create the active stimulus and checking components needed to verify a steel thread.

In addition, the OVM library contains a robust and fully functional set of components for reporting, object synchronization and object generators—to name a few.

From an integration stand-point, the tight coupling between the run-flow and OVM-based verification components provides considerable benefits when it comes to integrating new components. The common virtual interface of each component enables the run-flow to assume control of each component by virtue of a simple parent-child relationship established at instantiation. Furthermore, the fact that OVM leverages Transaction Level Modeling (TLM) and a common transaction class (ovm_data) to standardize communication between components makes inter-component, communication and synchronization relatively straightforward.

Lastly, OVM has an active user community providing a wealth of additional resources that can leverage in a functional verification environment.

The well defined yet flexible OVM library makes it very well suited to building the framework and functionality required for verification in incremental development.

**REGULAR REGRESSION TESTING**

Regression testing is critical for teams to maintain a working code base. It is also a vital prerequisite for incremental development. A regression suite should be run regularly to ensure the incremental addition of new features have not inadvertently broken existing functionality. While this is common practice for many IC development teams, what might not be common is regression testing that starts on day one of development. Regression testing on day one is an effort to prevent bugs from even entering the code base instead of resolving to find them later.

If a team wants to prevent the accumulation of bugs in the code base—and use working code as an objective progress metric—early and regular regression testing should be considered mandatory.

**SUMMARY**

This article highlights just a few ways an IC development team can make an Agile transformation. Agile software development teams have been using—and benefiting from— these and many others for more than a decade.

Agile methods, are relatively unknown and untried in IC development. The time has come, however, for organizations to recognize the success of Agile methods in other fields and to consider the merits of Agile development approaches in IC development.

And while Agile methods can provide great new frameworks and tools... the details of when and how they are used are still up to us!

**REFERENCES**
