Unifying Hardware-Assisted Verification and Validation Using UVM and Emulation

Hemant Sharma, Hans van der Schoot, Achutam Murarka
Mentor Graphics Corporation

***Abstract*— Successful approaches to improve verification productivity are to increase the speed of verification and begin validating software/hardware integration very early in the design process. Historically, verification and validation platforms have been developed as separate flows, preventing reuse of modules and methods between the two. As a consequence, various customized verification and validation platform features must be devised and implemented to verify complex, highly integrated System on Chip (SoC) designs. Bringing these two flows together would save an immense amount of duplicate effort and time while potentially reducing the introduction of errors, since less code needs to be developed and maintained.**

Keywords — Hardware-Assisted Verification, Emulation, Hardware/Software Co-Validation, UVM

I. INTRODUCTION

This paper describes a unified flow for both hardware-assisted RTL verification and pre-silicon validation of hardware/software integration. Deployment of this flow was accomplished in a customer setting by first combining a mainstream, transaction-level verification methodology — the Universal Verification Methodology (UVM) — with a hardware-assisted simulation acceleration (also known as co-emulation) platform. The necessary testbench modifications incorporated to enable this combination are generally nonintrusive and required no third-party class libraries; pertinent verification components from the customer environment were hence readily reusable in the pure simulation environments, across different designs using the same block, and for different verification groups.

Next to offering substantial speed-up for verification in terms of simulation cycles per second, the common-source SystemVerilog and UVM acceleration platform outlined above has subsequently been leveraged also for software validation. The hardware-assisted simulation acceleration factor of two to three (or more) orders of magnitude have made it practical for the software engineers to begin co-validating the software with the hardware far in advance of silicon. Clearly, the benefit of pre-silicon validation in terms of early detection of hardware/software integration issues boosts the efficiency of post-silicon validation as well.

The unified verification and validation flow has enabled a near seamless transition between RTL design verification and software/firmware validation. It significantly reduces the turn-around time for time consuming yet essential tasks, including debugging and regressions. It takes advantage of very fast emulator performance to handle longer and more tests to cover more design requirements and uncover more design bugs. In essence this unified flow has eliminated the productivity and quality penalties associated with creating and maintaining different verification and validation platforms.

In this paper we further detail how we established the unified design verification and pre-silicon validation flow and how we set up the associated testbench environment by leveraging verification industry standards for reuse (UVM) and co-modeling (SCE-MI 2). Several important testbench architecture and implementation decisions had to be made to maximize the reuse of tests in simulation, acceleration, and co-validation, which will be explained. We quantify the performance and overall productivity gained. We also share our recommendations for enhancements in support of such an advantageous approach. In general, the observations throughout the paper are largely applicable to most common use-case scenarios across many complex SoC designs.

II. TESTBENCH ARCHITECTURE

A. RTL design verification

A conventional verification environment has both synthesizable and non-synthesizable components instantiated in a single testbench top, as shown in Figure-1. This hinders running the testbench in a co-emulation setup, where two different physical devices are involved: a hardware emulator and a simulator.

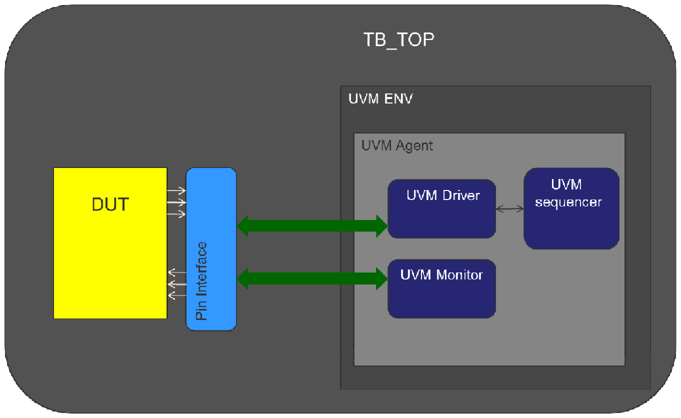

Figure-1: Traditional simulation testbench

A recommended approach is to create an acceleratable testbench that is partitioned into two tops: HDL_TOP and HVL_TOP. HDL_TOP has all the synthesizable components instantiated in it. HVL_TOP contains all untimed behavioral components as shown in Figure-2. Synthesized HDL_TOP runs on the hardware accelerator and HVL_TOP runs on the simulator.

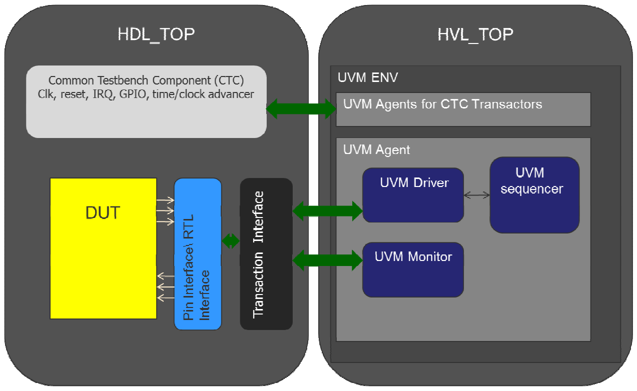
HDL_TOP
Common Testbench Component (CTC)
Clk, reset, IRQ, GPIO, time/clock advancer
DUT
Pin Interface\ RTL interface
Transaction Interface
HVL_TOP
UVM ENV
UVM Agents for CTC Transactors
UVM Agent
UVM Driver
UVM sequencer
UVM Monitor

Figure-2: Two-top testbench for HW-assisted acceleration

The HDL and HVL partitions of the model communicate at the transaction level. This communication is enabled by using a SystemVerilog virtual interface based method and/or a core-proxy based method.

A virtual interface is a SystemVerilog variable that can hold a reference to a concrete SystemVerilog interface instance. A variable of a virtual interface type can be given a value (i.e., it can be made to reference an existing interface instance) by assigning to it the hierarchical path name of the given interface instance [6]. The Mentor Graphics Veloce® emulation platform supports a synthesizable transactor interface that provides communication between emulator and testbench. Transactor interfaces encapsulate synthesizable SystemVerilog tasks and functions. A driver calls a function or task using a reference to a synthesized SystemVerilog interface to access DUT pins and signals. Similarly, a monitor waits on a function call from a transactor interface. All accesses to DUT pins and signals are thus strictly confined to the HDL partition. All such functions and tasks are restricted to synthesizable data types.

A so-called core-proxy is a C/C++ core model based on communication semantics between HDL and C, as defined by SCE-MI 2 [2]. A SystemVerilog API wrapper class connects a C-based proxy to the rest of the testbench. The proxy class maintains handles to components in the synthesizable transactors and uses DPI function calls or SCEMI pipes to communicate with these transactors [1][2]. In this approach, C-based proxy class functions provide APIs that can be used by a SystemVerilog or SystemC driver to communicate with the DUT. This paper will discuss only SystemVerilog driver and monitor implementation.

B. Extending the verification testbench for SW validation

A conventional software validation platform usually involves an FPGA prototyping platform with a JTAG connection to a software debugger. Typically, a dedicated team is responsible for partitioning the system on chip (SoC) RTL for a specific hardware platform to meet capacity and peripheral constraints. As more and more SoCs have embedded processors, there is a growing requisite for hardware/software co-validation. Earlier access to complete RTL, maintaining system bus connectivity across sub-modules and protocol peripherals, is an essential step of the verification process.

A preferred approach for hardware/software co-validation is to leverage the verification testbench to access the same RTL as used by design verification engineers. Based on the transactor modeling method described in the next section, a user can access the SoC system bus both with a UVM testbench or with a C API embedded software model to access the complete RTL. Figure-3 demonstrates a virtual machine communicating with the Mentor Graphics Veloce® emulator to validate software driver development using a proxy-based transactor. Such integration provides the complete SoC RTL in the hardware accelerator with full visibility for software and RTL debug.

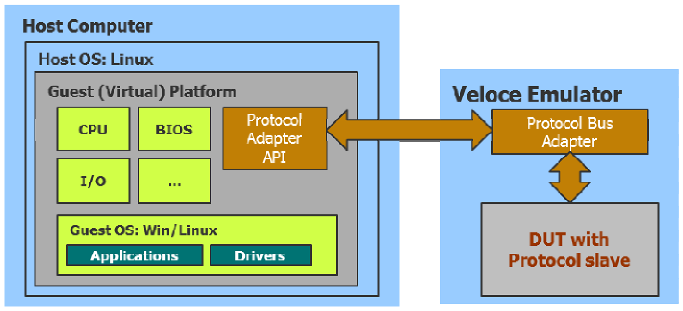

Figure-3: SW driver development with HW-assisted acceleration

III. Implementation Flow

In order to reduce the time to verification sign-off effectively, the following criteria are desirable:

- Single source design and testbench
- High performance
- Scalable testbench
- Ease of debug

To satisfy these criteria, the UVM testbench must adhere to the principles of co-emulation [1], which implies the aforementioned separation of synthesizable timed models from untimed testbench components. We bridged these partitioned domains using a SCE-MI compliant, high-performance, transaction-level communication link between the hardware and software provided by the Mentor Graphics Veloce/TBX® solution. This allowed us to accelerate the timed portions of the testbench and the DUT in the Veloce emulator without affecting the untimed UVM domain.

We further ensured that the entire environment — including especially the untimed UVM components and the timed transactors — was single source for both conventional simulation and for the co-emulation flow. Thus, any model written for the emulator can also be run in simulation alone using a SystemVerilog compliant simulator. This eliminates the need to maintain separate models and environments.

Transaction-level models that drive bus functional models maintain a transparent separation between the stimulus and the DUT pins. UVM layering concepts and transaction-level communication techniques provide the ability to swap out UVM-based directed and pseudo-random stimulus scenarios and apply, instead, real application software developed for the DUT, without sacrificing debug capabilities.

Clock and reset transactors were designed to take advantage of the variable clock delay feature of the Veloce/TBX emulation platform. This enabled the verification and validation teams to run their tests at various clock frequencies without recreating the RTL image for emulation.

The custom drivers and monitors were modeled using abstract classes and SystemVerilog interfaces. An example of such driver is shown in the subsequent transactor modeling section.

Last but not least, in order to monitor system bus activity and inter-block communication for improved debug visibility, we inserted inline SystemVerilog assertions into the DUT. Assertions and assertion coverage are supported by Veloce/TBX for synthesis into the emulator with highly efficient transfer of potentially large amounts of coverage data out of the emulator into the testbench domain. The well-known benefits of assertion-based verification coupled with the high-performance of emulation considerably reduced the total turn-around time to detect and resolve bugs.

A. Modeling transactors using SystemVerilog (virtual) interfaces

A UVM agent generally encapsulates a sequencer, driver, and monitor. A sequence item is a transaction object from the sequencer that stimulates the driver [4]. Figure-4 shows the modeling of a sequence item *Packet* and a corresponding SystemVerilog packed struct *Packet_t*. The latter is a suitable synthesizable representation of the former. To facilitate the conversion between the two representations, the sequence item provides two methods:

```
package transactions_t_pkg;
  typedef struct packed {
    int pkt_id;
    bit [15:0] address;
    bit [31:0] rd_data;
  } Packet_t;
endpackage
```

```
package transactions_pkg;
import transactions_t_pkg::*;
export transactions_t_pkg::*;
//--------------------------------------------
class Packet extends uvm_transaction;
  int pkt_id;
  rand bit [15:0] address;
  rand bit [31:0] rd_data;
  function void to_bytes(output Packet_t s);
    s = {pkt_id, address, rd_data};
  endfunction
  function void to_class(Packet_t s) ;
    {pkt_id, address, rd_data} = s ;
  endfunction
endclass // Packet
endpackage
```

Figure-4: Modeling a sequence item

- “to_class” converts a SystemVerilog packed struct to a transaction object;
- “to_bytes” converts the transaction object into a SystemVerilog packed representation.

A UVM driver gets the sequence item from the test and drives it onto the DUT pins [4]. The class-based driver receives a sequence item, converts it to a SystemVerilog struct, and passes the transaction referred by a virtual interface. In this use model, the bus functional model (BFM) to drive DUT pins is implemented in a synthesizable SystemVerilog interface. During the connect phase, the virtual interface of the UVM driver connects to a virtual interface (v_if in Figure-5) which, at the end of the elaboration step, connects to the actual transactor interface instance (driver_xif).

The synthesizable transactor interface (driver_xif) contains functions and tasks to apply transaction packets to DUT pins. It contains tasks that a UVM driver uses to write the transaction item. Figure-7 shows the connection of an actual interface to a virtual interface and its connection to the driver.

```
class driver extends uvm_component;
`uvm_component_utils(driver)
uvm_analysis_port #(Packet) stim_out_ap;

  virtual driver_xif v_if;
        task run();
          Packet stim_pkt = new;
          Packet_t stim_pkt_t;
          forever begin
                void'(stim_pkt.randomize());
                stim_pkt.pkt_id = pkt_id;
                pkt_id++;
                stim_pkt.to_bytes(stim_pkt_t);
                v_if.driver_bfm(stim_pkt_t);
          ………………
        endtask
endclass // driver
```

Figure-5: UVM driver calls functions and tasks implemented in synthesizable interface

```
interface driver_xif(pin_if pin_if_driver);
import transactions_t_pkg::*;
// pragma attribute driver_xif partition_interface_xif
  bit clk;
  assign clk = pin_if_driver.clk;
  task driver_bfm(Packet_t stim_pkt);
  // pragma tbx xtf
    endtask
endinterface
```

Figure-6: Transactor interface

The VifBindingPkg referred to in Figure-7 is a static SystemVerilog class to register unique references to concrete interface instances. This complete model is native

SystemVerilog and hence works in any SystemVerilog compliant simulator.

```
module hdl_top;
  driver_xif drvr_xif(pif.driver_mp); //Pin interface as a port to
transactor interface

    // tbx vif_binding_block
    initial begin
      import VifBindingPkg::VifBinding;
      VifBinding #(virtual driver_xif) C1;
      C1.RegisterVifBinding ($psprintf("%m.drvr_xif"), drvr_xif);
   end
```

Figure-7: Transactor interface instantiation

B. *Modeling C++ proxy-based transactors*

For monitor implementation, we leverage an analysis port and a proxy-based transactor to develop a generic monitor. A proxy-based transactor model can be interoperable between various verification languages; such as SystemVerilog, and SystemC, and even somewhat dated HVLs like Vera and *e*. The uvm_analysis_port is a specialized transaction port with a single function interface: write() [3].

Figure-8 shows the modeling of a monitor interface with a void write function similar to the analysis port. The write() function calls an import function implemented in the proxy class.

```
 interface generic_monitor
…………..
 import  "DPI-C" context function void XlMonitorConduitWrite(
       input bit [63:0] proxyChandleHolder,
       input bit [PAYLOAD_WIDTH-1:0] monitorRecord );

   function void write( input bit [PAYLOAD_WIDTH-1:0]
monitorRecord );
                    XlMonitorConduitWrite( proxyChandleHolder,
monitorRecord );
……………..
endinterface
```

Figure-8: Monitor interface

```
void XlMonitorConduitWrite(
   const svBitVecVal *proxyChandle, const svBitVecVal
*monitorRecord )
{
   XlBitVecCaster proxy( proxyChandle );

   XlMonitorConduit *me = (XlMonitorConduit *)(void
*)proxy;

   if( me  == NULL )
      // display error msg
   else
      me->write( monitorRecord );
}
```

Figure-9: Imported write() function

Figure-9 shows an import function implemented in a proxy C++ class. The monitor record is then routed via the classes that derive from this to its appropriate destination.

In our case we extended a SystemVerilog wrapper class from the proxy class. The SystemVerilog wrapper C++ proxy class and the generic_monitor constitute a generic monitor transactor. A SystemVerilog wrapper class (XlSvMonitorConduit) implemented a pure virtual write() callback function, which processes each record captured by the HDL-side monitor transactor. It is the intention for derivations of this class to override the abstract write() function with their own processing callback. Figure-10 shows one such implementation.

```
class XlSvUvmMonitorConduit
       #( type TRANSACTION_TYPE = uvm_transaction )
       extends XlSvMonitorConduit;

    local uvm_analysis_port #( TRANSACTION_TYPE )
    dAnalysisPort;
    local TRANSACTION_TYPE dMonitorRecordTransaction;

    function new( string hdlPath,
             uvm_analysis_port #(TRANSACTION_TYPE)
             analysisPort );
       super.new( hdlPath );

       dAnalysisPort = analysisPort;

       dMonitorRecordTransaction = new();
       dMonitorRecordTransaction.widthInBytes =
monitorRecordWidthInBytes();
       dMonitorRecordTransaction.hdlPath = hdlPath;
    endfunction
    function void write( const ref byte unsigned monitorRecord[]);

       dMonitorRecordTransaction.payload = monitorRecord;
       dAnalysisPort.write( dMonitorRecordTransaction );

    endfunction
endclass
```

Figure-10: Override write() callback function in derived class

The 'broadcast' transaction in the example above can be consumed by multiple subscribers, like a scoreboard or coverage collector (containing cover groups). In this case, we had a coverage agent connected to an analysis port to collect functional coverage for bus transactions.

C. *Reusing the testbench for software validation*

A typical SoC has a single processor managing multiple subsystems or multiple processors designated to perform specific tasks — all communicating via a system bus. A proxy-based transactor was developed to provide a master and slave interface to the system bus. Such an implementation provided a single RTL image that can be used with a SystemC testbench or a UVM testbench.

Software teams usually have a system model available for early code development. SystemC has emerged as the language of choice for these models. Hence, for this specific project we focused on language interoperability of the transactor; i.e., for portability between SystemC models on the one hand and SystemVerilog testbenches on the other hand. Figure-3 shows the environment setup for software validation. There are various hypervisor/virtual machines available to emulate the processor architecture. One such virtualization software package is QEMU, which can run in conjunction with the Mentor Veloce emulator. Software users can run their executable code in virtual machines that communicate with the RTL via a proxy based system bus transactor. As a result, the same RTL image is available for verification and validation with full visibility to debug.

D. *Other considerations*

To effectively implement the unified testbench for simulation and acceleration, we adhered to the following coding guidelines.

- # delays are not allowed in the testbench code. To achieve best performance, all code on the HVL testbench side should be untimed, and all timed code should be synthesized. If there is a need for unit or clock delays in the testbench (which can often be argued!), then a time advancer transactor must be used. This transactor counts the required number of clocks on the HDL side and then issues an event notification to the HVL side.
- There should not be any direct signal access from the HVL side. All communication must be transaction-based. To access individual signals indirectly, GPIO transactors were used.
- Memory models can remain as behavioral Verilog code. The Veloce compiler can infer structural memories from behavioral Verilog memories.

IV. CONCLUSION

The adopted co-emulation flow and resulting unified verification and validation platform has made it possible for SoC design verification and software validation teams to use the same RTL image for their respective test scenarios. Moreover, the software validation team can now have much earlier access to the RTL code than in previous projects of similar scope. For the case study at hand, a platform for validation was available to the software team about four months sooner.

The single-source verification environment for both simulation and emulation provided test speed ups of at least an order of magnitude. This translates to tests that took nearly a week to run in simulation now completing in just over an hour on the co-emulation platform. Key architecture and implementation decisions needed to be made in order to accomplish this unified environment and to maximize the reuse of tests in simulation, acceleration, and co-validation.

Further work is focused on integrating assertions data from the emulator in a common database to support coverage-driven verification using an emulator. With the standardization of the Unified Coverage Interoperability Standard DataBase (UCISDB) [5], engineers can collect coverage data in one common database from simulation and acceleration platforms. Thus, providing the ability to use software generated vectors to attain RTL coverage metrics.

With the speed of an emulator and a combination of directed tests as well as real-world scenarios, verification engineers will be able to validate the SoC more holistically.

As simulation acceleration techniques get adopted more broadly in the industry, advanced standards-based verification methodologies such as UVM should increasingly accommodate requirements from hardware-assisted verification testbenches.

REFERENCES

[1] H. van der Schoot, A. Saha, A. Garg, S. Krishnamurthy, "Off to the races with your accelerated SystemVerilog testbench," DVCon 2011.

[2] Accellera – Interfaces Technical Committee, "Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual", version 2.1, October 2010.

[3] Accellera – Universal Verification Methodology (UVM) 1.1 User's Guide, May 18, 2011

[4] M. Glasser, "Open Verification Methodology Cookbook", springer, 2009

[5] Accellera – Unified Coverage Interoperability Standard (UCIS), Version 1.0, June 2, 2012

[6] Rich D., Bromley, J. "Abstract BFMs Outshine Virtual Interfaces for SystemVerilog Testbenches", DVCon 2008