

A Methodology for Hardware-Assisted Acceleration of OVM and UVM Testbenches

by Hans van der Schoot, Anoop Saha, Ankit Garg, Krishnamurthy Suresh, Emulation Division, Mentor Graphics Corporation

[Editor's Note: This is part 2 of a two-part article on this topic. Part 1 appeared in the DVCon (February, 2011) edition of Verification Horizons. This article should serve as a great companion piece to the new Verification Academy module, Acceleration of SystemVerilog Testbenches with Co-Emulation.]

A methodology is presented for writing modern SystemVerilog testbenches that can be used not only for software simulation, but especially for hardware-assisted acceleration. The methodology is founded on a transaction-based co-emulation approach and enables truly single source, fully IEEE 1800 SystemVerilog compliant, transaction-level testbenches that work for both simulation and acceleration. Substantial run-time improvements are possible in acceleration mode and without sacrificing simulator verification capabilities and integrations including SystemVerilog coverage-driven, constrained-random and assertion-based techniques as well as prevalent verification methodologies like OVM or UVM.

IMPLEMENTING A TRANSACTION-LEVEL HVL-HDL INTERFACE

With the timed and untimed portions of a testbench fully partitioned, what remains is establishing a transaction-based communication mechanism for co-emulation. As suggested above, the use of virtual interface handles on the HVL side bound to concrete interface instances on the HDL side enables a flexible transaction transport mode for HVL-HDL communication provided thus that BFM s are implemented as SystemVerilog interfaces in the HDL hierarchy, not as modules. The flexibility stems from the fact that user-defined tasks and functions in these interfaces form the API.

Following the remote proxy design pattern discussed earlier, components on the HVL side acting as proxies to BFM interfaces can call relevant tasks and functions declared inside the BFM s via virtual interface handles to drive and sample DUT signals, initiate BFM threads, configure BFM

parameters or retrieve BFM status. By retaining specifically the original transactor layer components like driver and monitor classes as the BFM proxies (see Figure 2) – minus the extracted BFM s themselves – impact on the original SystemVerilog object-oriented testbench is minimized. The proxies form a thin layer in place of the original transactor layer, which allows all other testbench layer components to remain intact. This offers maximum leverage of existing verification capabilities and methodologies.

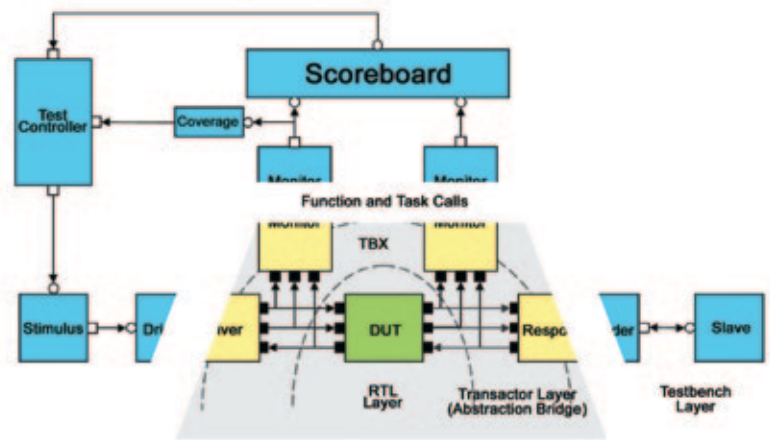


Figure 2. Transaction-based testbench with transactor/BFM proxies

The remote task/function call mechanism is based for the most part on the known Accellera SCE-MI 2 function model, and so it has the same kind of performance benefits as SCE-MI 2. In the traditional SCE-MI 2 function-based model it is the SystemVerilog DPI interface that is the natural boundary for partitioning workstation and emulator models [1], whereas the proposed methodology here uses the class object to interface instance boundary as the natural boundary for the same partitioning. Extensions specifically designed for SystemVerilog testbench modeling are added, most notably task calls in the workstation to emulator direction in which use of time-consuming/multi-cycle processing elements is allowed. This is essential to be able to model BFM s on the HDL side that are callable from the HVL side.

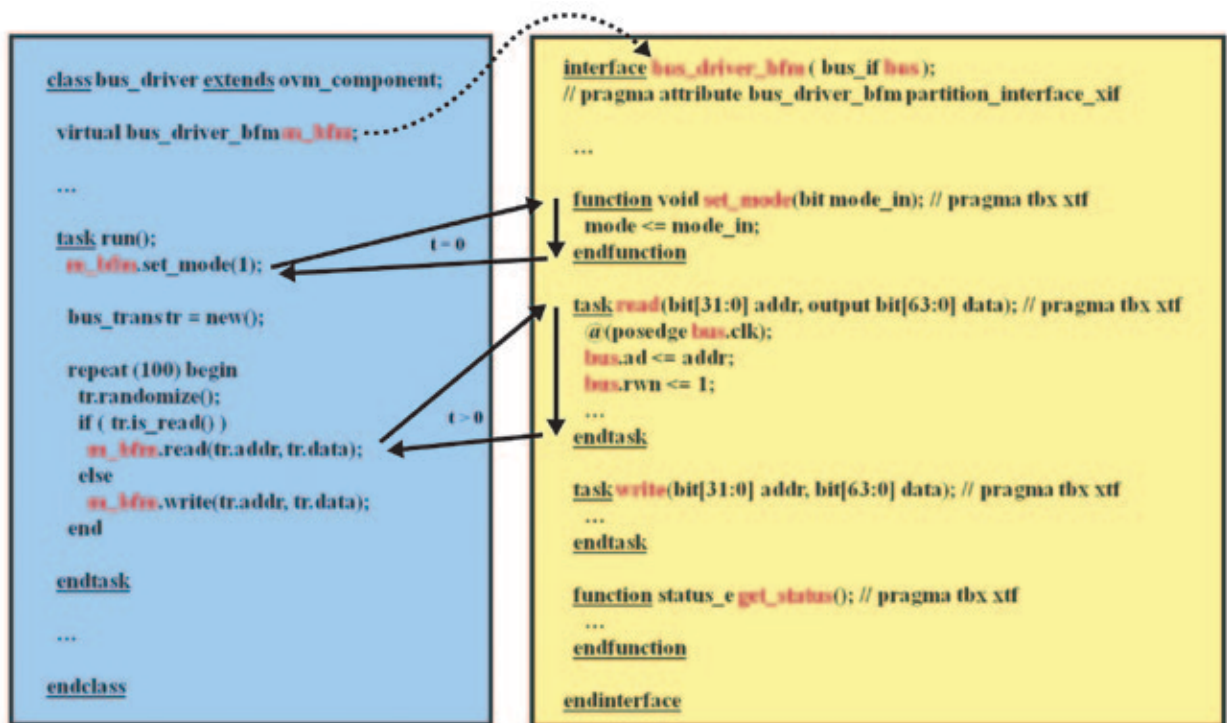


Figure 5. HDL BFM interface with HVL proxy class

The HVL-HDL co-modeling interface mechanism is depicted in Figure 5 above. A proxy class `bus_driver` has a virtual interface handle `m_bfm` to a corresponding BFM model `bus_driver_bfm` implemented as a synthesizable interface. Time-consuming tasks and non-blocking functions in the interface can be called by the driver proxy via the virtual interface to execute bus cycles, set parameters or get status information. Notice the 'bfm' suffix in the BFM interface name, which is recommended as a naming convention. Also notice the use of the bus pin interface confined to the BFM by inclusion through its port list.

TRANSACTION OBJECT CONVERSION

Classes and other dynamic or unpacked data types in SystemVerilog are generally not synthesizable and can therefore not be used as BFM function/task arguments. For SystemVerilog object-oriented testbenches that extensively use class-based transactions (e.g. those derived from the `ovm_transaction` base class in OVM) it means that these transactions cannot simply be passed as is between the BFM interfaces and their proxies. However, since BFM functions and tasks are user-defined, it may be pertinent to pass transaction class members as individual packed arguments, just as shown in the code example of Figure 5 for the address and data attributes of bus transactions.

Or one may choose to utilize special conversion routines to convert explicitly between class-based transactions and suitable packed type representations that are synthesizable such as a bit vector or packed struct. When utilized, it is recommended to standardize on `from_class(...)` and `to_class(...)` methods defined in an external converter class for each transaction type that must cross the HVL-HDL boundary. A code example is given in Figure 6.

```

1  class fpu_request extends ovm_transaction;
2
3  shortreal a;
4  shortreal b;
5  rand op_t op;
6  rand round_t round;
7
8  ...
9
10 endclass
11
12
13 package fpu_trans_util_pkg;
14 typedef struct packed {
15     bit [31:0] a;
16     bit [31:0] b;
17     op_t op;
18     round_t round;
19 } fpu_request_s;
20

```

```

21  typedef bit [$bits(fpu_request_s)-1:0]
22      fpu_request_vector_t;
23
24  ...
25
26  endpackage27
28  class fpu_request_converter;
29
30  function void to_class(
31      output fpu_request req,
32      input fpu_request_vector_t v);
33      fpu_request_s s = v;
34      req = new();
35      req.a = $bitstoshortreal(s.a)
36      req.b = $bitstoshortreal(s.b);
37      req.op = s.op;
38      req.round = s.round;
39  endfunction
40
41  function void from_class(
42      input fpu_request req,
43      output fpu_request_vector_t v);
44      fpu_request_s s;
45      s.a = $shortrealtoibits(req.a);
46      s.b = $shortrealtoibits(req.b);
47      s.op = req.op;
48      s.round = req.round;
49      v = s;
50  endfunction
51
52  endclass

```

Figure 6. Converting transaction objects for co-emulation

Figure 7 on the following page provides an example transformation of a purely class-based FPU monitor from the OVM cookbook example kit [2] into a functionally equivalent BFM / proxy pair suited for both simulation and co-emulation. The FPU monitor proxy reimplements tasks `monitor_request()` and `monitor_response()` (i.e. lines 21-30 and 32-46 in Figure 7.b) to call corresponding tasks in the BFM (i.e. lines 58-68 and 70-73 in Figure 7.b) to perform the pin-level sampling of FPU request and response transactions and output these to the BFM proxy. External

converter classes with `from_class(...)` and `to_class(...)` methods are used to convert between FPU transaction objects and convenient synthesizable packed struct representations of these transactions (i.e. lines 27 and 39 in Figure 7.b), as shown in Figure 6 for FPU requests.

For the example above it is assumed that the BFM interface is instantiated somewhere under the HDL top level hierarchy and that its corresponding proxy object on the HVL side has a virtual interface reference to the BFM. The actual binding of the virtual interface to the hierarchical HDL path of the BFM is not shown for brevity. Any such binding mechanism can be made to work also in the context of co-emulation. For OVM testbenches a recommended method described in [3] utilizes a general purpose OVM container class for wrapping any SystemVerilog type so that it can be used with the OVM configuration mechanism. It works just fine for binding BFM / proxy pairs.

HDL-TO-HVL BACK-POINTERS

For modeling flexibility and completeness a transaction-level HVL-HDL co-modeling interface can be defined in both directions. Similar to an HVL proxy class calling tasks and functions declared in an HDL interface, as discussed thus far, one can define how an HDL interface can call functions¹ declared in an HVL class. This would enable transaction-based HVL-HDL communication initiated from the HDL side. Specifically, a BFM interface may call relevant class member functions of its proxy object on the HVL side for instance to provide sampled transactions for analysis or indicate other status information. Figure 8 on the following page illustrates this. As shown, the handle of a BFM interface to the BFM proxy can be assigned simply inside the proxy itself via its virtual interface handle to the BFM. Access to any data members in the BFM proxy would not be permitted, just as cross signal references into the BFM are not allowed. Due to language restrictions on matching types, the proxy class definition together with any types it depends on must be imported inside the BFM interface via one or more packages.

Figure 7. Transforming an FPU monitor for co-emulation

```

1  class fpu_monitor extends ovm_component;
2
3  ovm_analysis_port #(fpu_pair) pair_ap;
4
5  // VIF handle to pin interface
6  local virtual fpu_pin_if #(32) m_fpu_pins;
7
8  ...
9
10 function void connect();
11     ... // Retrieve m_fpu_pins vif handle
12 endfunction
13
14 task run();
15     fork
16         monitor_request();
17         monitor_response();
18     join
19 endtask
20
21 task monitor_request();
22     forever begin
23         fpu_request req = new();
24
25         do
26             @(posedge m_fpu_pins.clk);
27             while (m_fpu_pins.start != 1);
28
29             req.a = $bitstoshortreal(m_fpu_pins.op_a);
30             req.b = $bitstoshortreal(m_fpu_pins.op_b);
31             req.op = op_ti(m_fpu_pins.fpu_op);
32             req.round = round_ti(m_fpu_pins.rmode);
33
34             $cast(m_req_in_process, req.clone());
35         end
36     endtask: monitor_request
37
38 task monitor_response();
39     forever begin
40         fpu_response rsp = new();
41         fpu_pair pair;
42
43         ... // Timed code to sample response
44
45         pair = new(m_req_in_process, rsp);
46         pair_ap.write(pair);
47     end
48 endtask: monitor_response
49
50 endclass

```

(a) Original monitor

```

1  class fpu_monitor extends ovm_component;
2
3  ovm_analysis_port #(fpu_pair) pair_ap;
4
5  // VIF handle to XRTL BFM
6  local virtual fpu_monitor_bfm m_bfm;
7
8  ...
9
10 function void connect();
11     ... // Retrieve m_bfm vif handle
12 endfunction
13
14 task run();
15     fork
16         monitor_request();
17         monitor_response();
18     join
19 endtask
20
21 task monitor_request();
22     forever begin
23         fpu_request req;
24         fpu_request_s req_s;
25
26         m_bfm.monitor_request(req_s);
27         req_converter.to_class(req, req_s);
28         $cast(m_req_in_process, req.clone());
29     end
30 endtask: monitor_request
31
32 task monitor_response();
33     forever begin
34         fpu_response rsp;
35         fpu_response_s rsp_s;
36         fpu_pair pair;
37
38         m_bfm.monitor_response(rsp_s);
39         rsp_converter.to_class(rsp, rsp_s);
40         ...
41         pair = new(m_req_in_process, rsp);
42         pair_ap.write(pair);
43     end
44 endtask: monitor_response
45
46 endclass
47
48
49 interface fpu_monitor_bfm(fpu_pin_if fpu_pins);
50 // pragma attribute fpu_monitor_bfm
51     partition_interface_xif
52
53 ...
54 wire clk = fpu_pins.clk;

```

```

55
56 task monitor_request(output
57     fpu_request_s req); // pragma tbx xtf
58     @(posedge clk);
59     while (fpu_pins.start != 1)
60         @(posedge clk);
61     req.a = fpu_pins.op_a;
62     req.b = fpu_pins.op_b;
63     req.op = op_ti(fpu_pins.fpu_op);
64     req.round = round_ti(fpu_pins.rmode);
65 endtask
66
67 task monitor_response(output
68     fpu_response_s rsp); // pragma tbx xtf
69     ... // Timed code to sample response
70 endtask
71
72 endinterface

```

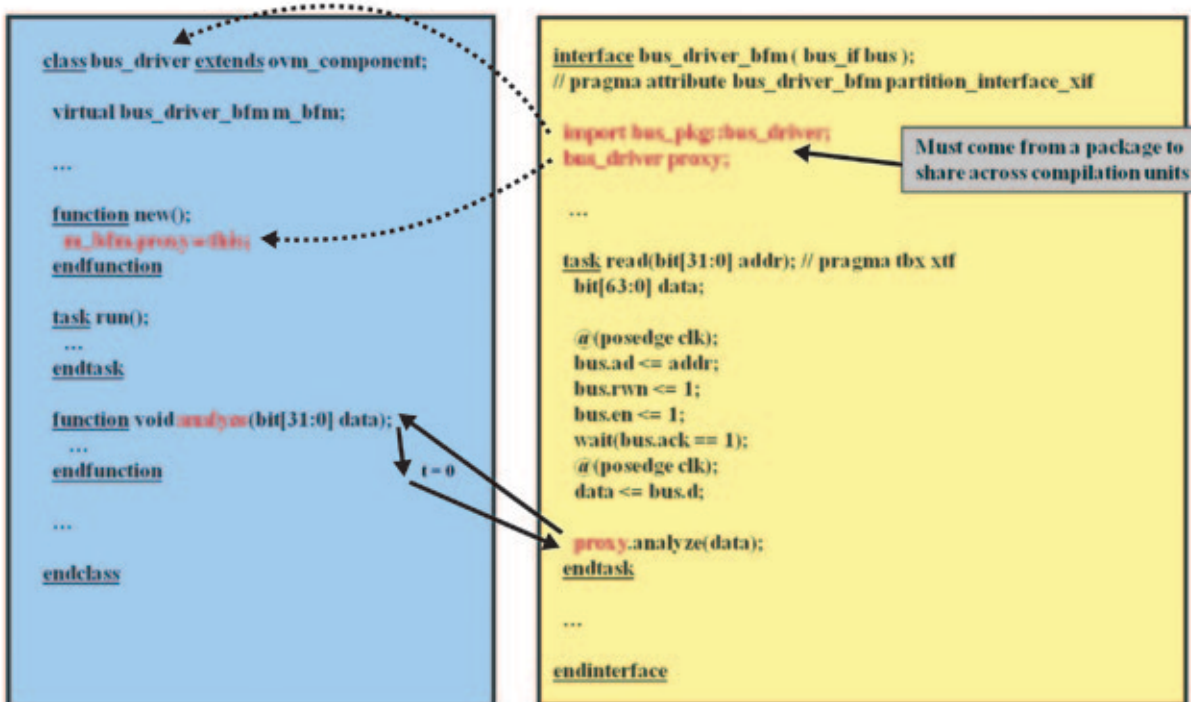
(b) XRTL monitor BFM with proxy

The use of such object handles in BFM interfaces back to their proxy classes, or 'back-pointers', is not firmly required for modeling reactive HVL-HDL communication and one can just stick to using HVL initiated 'xtf' tasks and

functions². Yet this is particularly useful for components like monitors. A typical monitor continuously listens to an interface to extract transactions and pass them out to other testbench components for analysis, just like the FPU monitor in Figure 7. It initiates communication of observed transactions to 'subscribers' like scoreboards, coverage collectors or interrupt monitors. It is in effect more natural to have a monitor BFM 'push' instead of the BFM proxy 'pull' these transactions out. More importantly, doing so presents opportunities for significant performance optimization. Observed transactions are commonly distributed for analysis using void functions (e.g. the TLM write(...) function in OVM – i.e. line 46 in Figure 7.a). Such one-way non-blocking calls can be dispatched and executed concurrently without even stopping the emulator clocks.

Figure 9 on the following page provides a second take on remodeling the OVM-based FPU monitor for co-emulation. This time the monitor BFM calls a void function write of its proxy via a back-pointer to push sampled FPU request-response pairs out to the HVL side (i.e. lines 62 and 22-26 in Figure 9.b). The reader is invited to inspect the example in more detail with respect to the one in Figure 7.

Figure 8. HDL Driver BFM interface with HVL proxy



ADDITIONAL METHODOLOGY CONSIDERATIONS

Prying apart transactor layer components into synthesizable BFM on the HDL side and untimed transaction-level proxy objects on the HVL side, as described in the previous section, has the consequence that the BFMs must be elaborated statically before run-time. At first sight some of the capabilities of a truly dynamic testbench may seem lost. Recall though that it is only the timed interface protocol that is to be implemented on the HDL side. Since the DUT interface and protocol are largely static there is no real loss of functionality. The idea is to retain the bits and pieces that must be dynamic inside the BFM proxy under the HVL top level module hierarchy. It should be apparent that a BFM interface is then in principle controllable completely through its dynamic proxy, via remote function or task calls. For instance, in terms of OVM it means that while BFMs cannot be created using the OVM factory or configured using the OVM configuration mechanism, the BFM proxies can be controlled in this way and hence indirectly the static BFMs themselves.

Thanks to the application of the remote proxy design pattern, prevalent testbench topology practices can also be facilitated without much alteration. Figure 10 depicts the normal view of an OVM agent for simulation and the adapted view for co-emulation. From the perspective of the OVM testbench on the HVL side there is no difference. Certainly, a matching topology of BFM interfaces under the HDL top can be configured only statically at elaboration-time, but as suggested by the code example in Figure 11 it is rather straightforward to employ SystemVerilog conditional or loop generate constructs on the HDL side in combination with a shared package of static test parameters imported and used by both HDL and HVL sides. The topology of a typical testbench is after all static in nature since it is expected to be fully elaborated before any testbench component starts running (e.g. the 'end-of-elaboration' phase in OVM executes before the 'run' phase). In case a truly dynamic alternative is desired it is possible to elaborate a fixed number of BFMs on the HDL side of which only a subset become active as maintained by the type and number of dynamically created BFM proxy objects.

Another methodology consideration is that current synthesis technology does not readily handle SystemVerilog coverage groups. Coverage groups are well suited for implementing

```

1  class fpu_monitor extends ovm_component;
2
3  ovm_analysis_port #(fpu_pair) pair_ap;
4
5  // VIF handle to pin interface
6  local virtual fpu_pin_if #(32) m_fpu_pins;
7
8  ...
9
10 function void build();
11     ... // Retrieve m_fpu_pins vif handle
12 endfunction
13
14 task run();
15     @(posedge m_fpu_pins.clk);
16     fork
17         monitor_request();
18         monitor_response();
19     join
20 endtask
21
22 task monitor_request();
23     forever begin
24         fpu_request req = new();
25
26         do
27             @(posedge m_fpu_pins.clk);
28             while (m_fpu_pins.start != 1);
29
30             req.a = $bitstoshortreal(m_fpu_pins.op_a);
31             req.b = $bitstoshortreal(m_fpu_pins.op_b);
32             req.op = op_ti(m_fpu_pins.fpu_op);
33             req.round = round_ti(m_fpu_pins.rmode);
34
35             $cast(m_req_in_process, req.clone());
36         end
37     endtask: monitor_request
38
39     task monitor_response();
40         forever begin
41             fpu_response rsp = new();
42             fpu_pair pair;
43
44             ... // Timed code to sample response
45
46             pair = new(m_req_in_process, rsp);
47             pair_ap.write(pair);
48         end
49     endtask: monitor_response
50
51 endclass

```

(a) Original monitor

```

1  class fpu_monitor extends ovm_component;
2
3  ovm_analysis_port #(fpu_pair) pair_ap;
4
5  // VIF handle to XRTL BFM
6  local virtual fpu_monitor_bfm m_bfm;
7
8  ...
9
10 function void connect();
11 .. // Retrieve m_bfm vif handle
12   m_bfm.proxy = this;
13 endfunction
14
15 task run();
16   fork
17     m_bfm.request_daemon();
18     m_bfm.response_daemon();
19   join
20 endtask
21
22 function void write(fpu_pair_s pair_s);
23   fpu_pair pair = new();
24   pair_converter.to_class(pair, pair_s);
25   pair_ap.write(pair);
26 endfunction
27
28 endclass
29
30
31 interface fpu_monitor_bfm(fpu_pin_if fpu_pins);
32 // pragma attribute fpu_monitor_bfm
33   partition_interface_xif
34
35   ...
36
37   import fpu_tlm_pkg::fpu_monitor;
38   fpu_monitor proxy;
39   // pragma tbx oneway proxy.write
40
41   fpu_request_s req_in_process;
42
43   task request_daemon(); // pragma tbx xtf
44   ... // Sample requests (req_in_process);
45 endtask
46
47   task response_daemon(); // pragma tbx xtf
48   fpu_pair_s pair;
49
50   @(posedge clk);
51   forever begin

```

```

52     @(posedge clk);
53     while (fpu_pins.ready != 1)
54       @(posedge clk);
55
56     ...
57
58     pair.req = req_in_process;
59     pair.rsp.result = fpu_pins.outp;
60
61     ...
62
63     proxy.write(pair);
64   end
65 endtask
66
67 endinterface

```

(b) XRTL monitor BFM with proxy

Figure 9. Transforming an FPU monitor for co-emulation—take 2

transaction-level coverage concerned with the higher level functional requirements of a design. This stands in contrast to assertion coverage which lends itself for measuring the occurrence of lower level physical events involving the sampling of DUT signals and state variables, potentially over multiple consecutive clock cycles [4]. Assertion coverage fits naturally for BFMs and is in fact supported for synthesis by TBXTM. Moreover, while surely coverage groups could be of use in BFMs as well, key to handling any genuine transaction-level coverage requirement for a BFM interface is once again the BFM's HVL proxy object, which may have coverage groups itself and forward transactions to other transaction-level coverage analysis components (e.g. see Figure 9.b).

EMPIRICAL RESULTS

Table 1 on the following page lists empirical results of applying the proposed transaction-based SystemVerilog testbench acceleration methodology. For several different designs the run-times for executing a test with pure simulation and with co-emulation are compared. The co-emulation engine used is Mentor Graphic's Veloce TBXTM. The results clearly indicate that co-emulation can be much faster than simulation

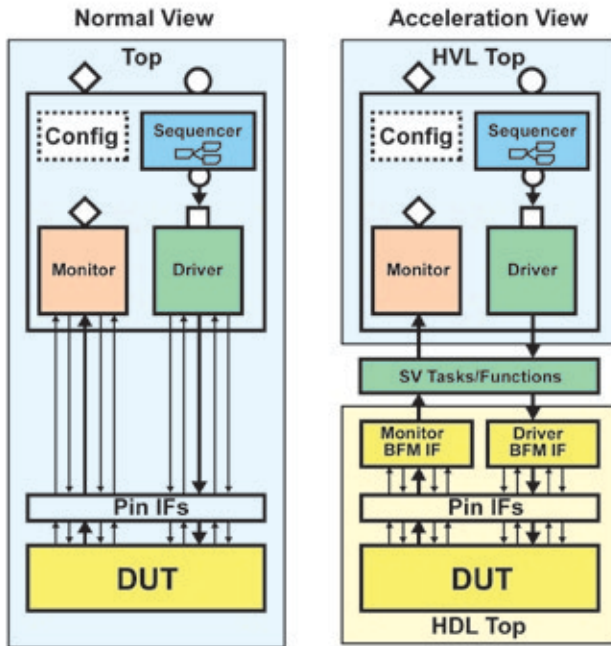


Figure 10. Normal simulation view and co-emulation view of an OVM agent

alone. Therefore, if simulation leaves you with insufficient throughput to meet your verification requirements, rather than taking calculated risks and limit the length of your simulation runs, you could greatly improve verification throughput with realistic tests using co-emulation.

SUMMARY AND CONCLUSIONS

A methodology was described for writing SystemVerilog and OVM or UVM testbenches that can be used not only for software simulation, but especially for hardware-assisted acceleration. For modern transaction-level testbenches, the pragmatic approach to hardware-assisted speedup in testbench execution is to have certain testbench

components – the lower pin-level components like drivers, monitors etc. – synthesized into real hardware and running inside the emulator together with the DUT, while other non-synthesizable testbench components – the higher transaction-level components like generators, scoreboards, coverage collectors etc. – remain in software running inside the simulator. Communication between simulator and emulator is then transaction-based, not cycle-based, reducing communication overhead and increasing performance because hardware-software data exchange is infrequent and information rich, and high frequency pin activity is confined to run in hardware at full emulator clock rates.

This so-called co-emulation or co-modeling approach is at the core of the methodology presented, which further maximizes reuse between pure simulation-based verification and hardware-assisted acceleration through the application of an object-oriented remote proxy design pattern. As a result, truly 'single source' and fully IEEE 1800 SystemVerilog compliant transaction-level testbenches can be created to work interchangeably for both simulation and acceleration. In acceleration mode substantial run-time improvements are made possible and without sacrificing simulator verification capabilities and integrations such as modern coverage-driven, constrained-random and assertion-based techniques and tools. Additionally, the acceleration methodology is independent of the SystemVerilog verification methodology used and applicable to all prevalent methodologies today including OVM or UVM, and VMM.

In technical summary, the proposed simulation and acceleration methodology stipulates that a testbench be partitioned into two completely separated hierarchies, a synthesizable HDL side and a strictly untimed HVL side. Cross module and signal references are not permitted between the two sides. Instead, only transaction-level data

Table 1. Empirical results

Design	Simulation Time	Veloce TBXTM	Speed-up Factor
Face Recognition Engine (1 MG)	½ hr.	6.58 secs.	128x
Wireless MM Sub-system (1 MG)	53 hrs.	658 secs	288x
Menory Controller (1.1 MG)	5 hrs.	308 secs	60x
Mobile Display Processor (1.2 MG)	5 hrs.	46 secs.	399x
Network Switch (34 MG)	16½ hrs.	240 secs.	245x
Graphics Sub-system (8 MG)	86½ hrs.	635 secs.	491x

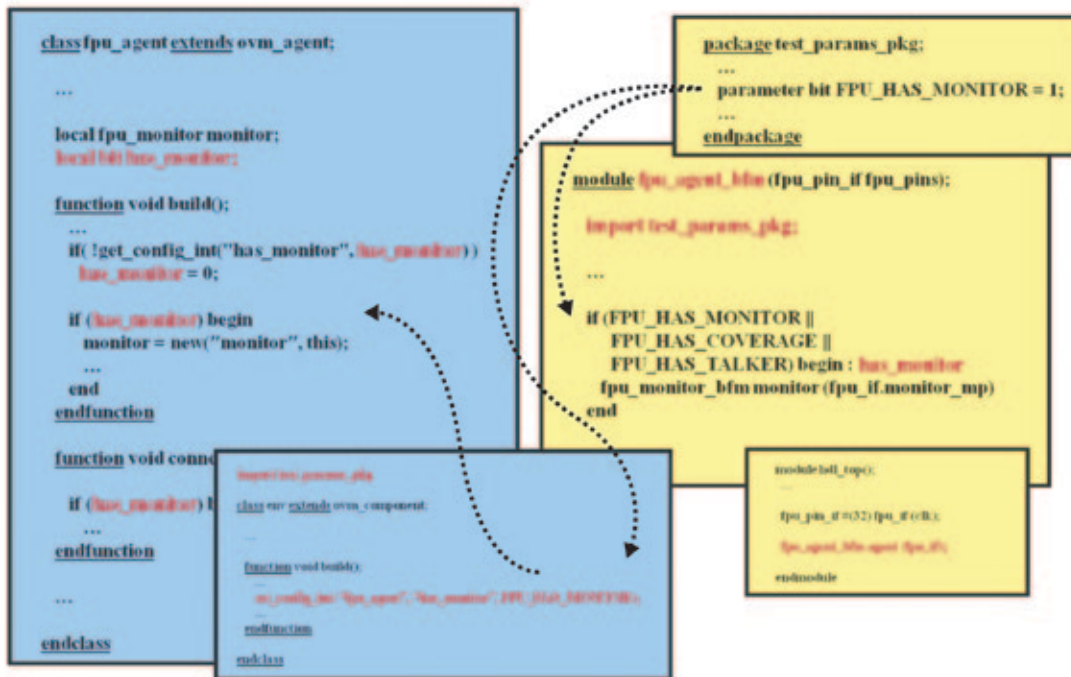


Figure 11. Topology configuration

exchange is performed via ‘remote procedure invocation’ in SystemVerilog, and with Accellera SCE-MI 2 inspired performance benefits. Specifically, each DUT interface protocol – or BFM – on the HDL side is modeled as a synthesizable SystemVerilog interface with designated tasks and functions that can be called from the HVL side through a virtual interface by a dynamic class object that acts as HVL proxy for the BFM. Transaction objects may thereby need to be converted into synthesizable arguments. Conversely, the BFM interface may also have an object handle back to its proxy to call functions defined in the proxy. Reactive transaction-based communication is thus supported across the HVL-HDL boundary in both directions with either the HVL proxy or the BFM as call initiator. Each pair of BFM and proxy is to be viewed essentially as a joint pair representing a single transactor.

REFERENCES

- [1] Accellera – Interfaces Technical Committee, “Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual,” Version 2.1 (Review Copy), October 21, 2010
- [2] M. Glasser, “Open Verification Methodology Cookbook,” Springer, 2009. (Associated example kit available at

- www.ovmworld.org/contribution-detail/24891)
- [3] A. Rose, M. Glasser, B. Osman, “OVM Configuration and Virtual Interfaces,” White Paper, Mentor Graphics, 2010.
- [4] H. van der Schoot, J. Bergeron, “Transaction-Level Functional Coverage in SystemVerilog,” DVCon, 2006.
- [5] A. Saha, K. Suresh, A. Jain, V. Kulshrestha, S. Gupta, “An Acceleratable OVM Methodology Based on SCE-MI 2,” DVCon, 2008.

NOTES

¹ Since clocks are running exclusively on the HDL side, only functions – not tasks – should be called from HDL to HVL side. Strictly speaking only time-consuming tasks are problematic, but it is recommended to avoid tasks altogether.

² Additionally, next to reactive communication it is possible to model non-reactive, streaming communication which can be viewed as an alternative where HVL and HDL sides are decoupled and run independent threads. Mentor Graphics’ Veloce TBXTM offers this alternative via the use of so-called SCEMI pipes which are a kind of ‘acceleration-friendly’ buffer with unique data-shaping features for performance, described in the Accellera SCE-MI 2 standard [1]. This is not further discussed here.