UVM & Emulation

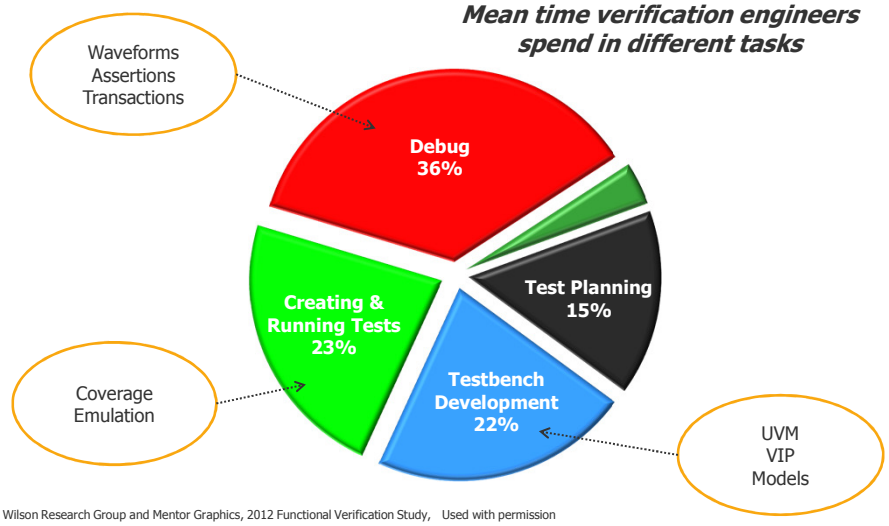
Creating SystemVerilog & UVM Testbenches for Simulation & Emulation Platform Portability to Boost Block-to-System Verification Productivity

Hans van der Schoot, Ph.D.
Emulation Technologist
Mentor Emulation Division
hans_vanderschoot@mentor.com

Agenda

- Introduction
- Fundamentals of Hardware-Assisted Testbench Acceleration
- Unified Testbench Architecture & Methodology for UVM Acceleration
- Portable Verification from Simulation to Emulation
- Wrap Up
- Q & A

Goal is to Reduce Time Spent in Verification



Wilson Research Group and Mentor Graphics, 2012 Functional Verification Study, Used with permission

4 UVM & Emulation, DVCon India 2015, HvdS

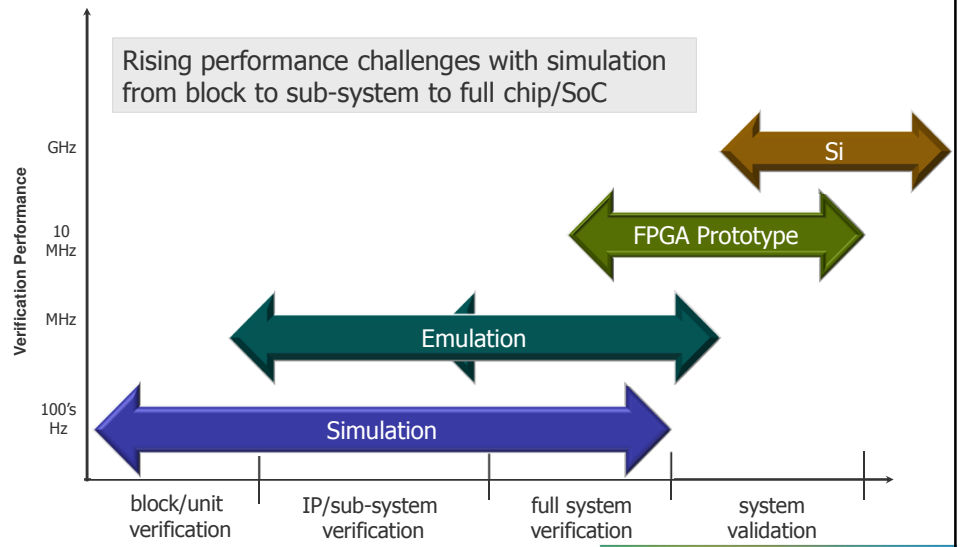
Verification Productivity

UVM

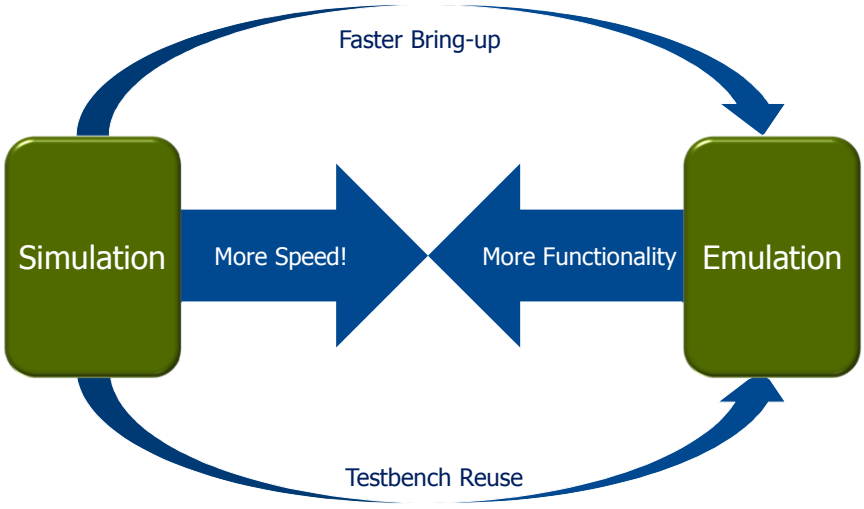
- Electronics systems companies need dramatic improvements in verification productivity
- Adoption of UVM for increased verification productivity
  - Faster to develop reusable verification components, testbenches and automated tests
- Three dimensions of reuse
  - Horizontal
    - Components, modules, libraries across projects
  - Vertical
    - Block to sub-system to system level in a single project
  - Platform
    - Testbenches, assertions, coverage across engines/tools

The Need for Speed

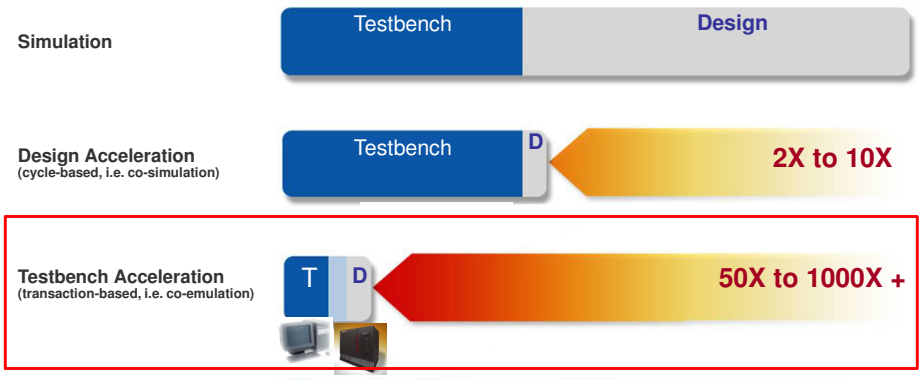
Rising performance challenges with simulation from block to sub-system to full chip/SoC



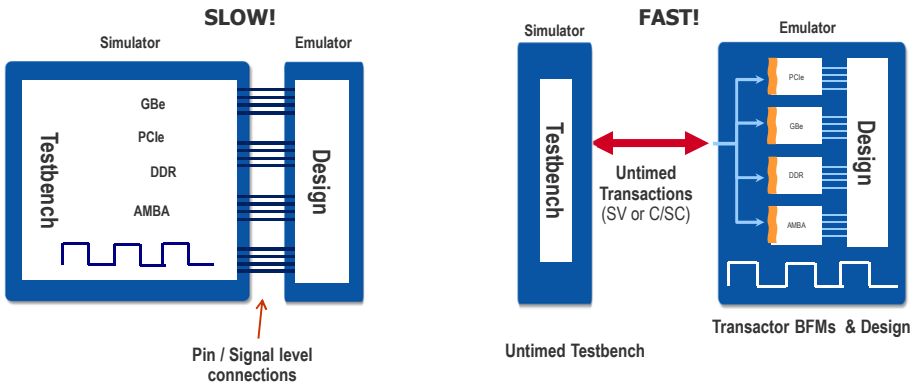
Enter Emulation – Get the Best of Both Worlds



Testbench Pivotal to Acceleration Speed-Up



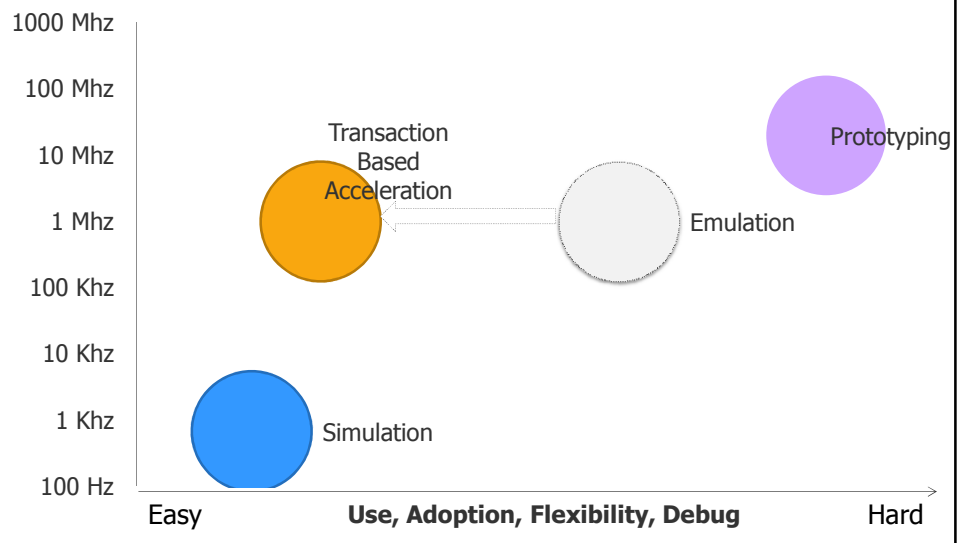
Transaction-Based Acceleration is Key to Performance



Infrequent Information-Rich Data Exchange between Emulator and Simulator is Key to Performance (Orders of Magnitude Faster)

9 UVM & Emulation, DVCon India 2015, HvdS

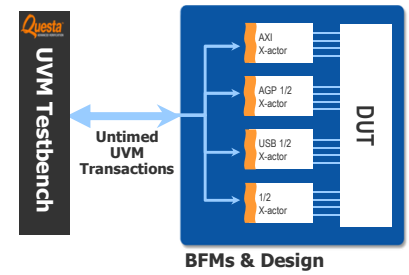
Transaction-Based Acceleration Makes Emulation Easier to Use and Adopt



10 UVM & Emulation, DVCon India 2015, HvdS

UVM Acceleration – Reuse UVM testbench

A verification architecture for simulation through acceleration

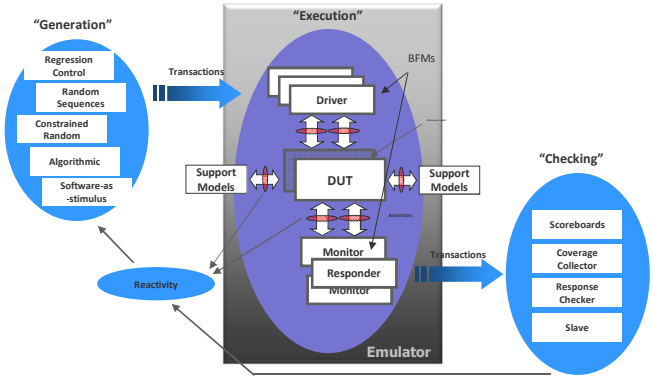


- Realize UVM vertical reuse through platform portability
  - Leverage UVM testbench into emulation
  - Leverage SV assertions & coverage into emulation
  - Getting the right testbench architecture is key
- Reduce overall development effort
  - Simulation-emulation interoperability
  - Modeling flexibility for synthesizable BFMs
  - Optimal emulation and simulation performance
- Benefits
  - More verification cycles earlier in design cycle
  - Orders of magnitude performance gain
  - Verification schedule predictability
  - Throughput for any sized design
  - Rapid turn-around of changes

Orders of Magnitude Faster - Easily Transition from Simulation

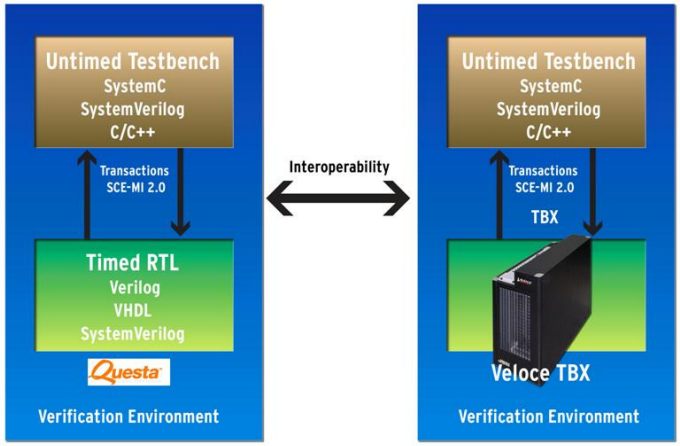
Veloce OS, Methodology & Applications

Modern Testbench & Emulation



- DUT and BFM "execution" runs in simulator or emulator
- Testbench "generation", "checking" and "coverage" runs in simulator
- Maintains simulation-based verification features and methodologies

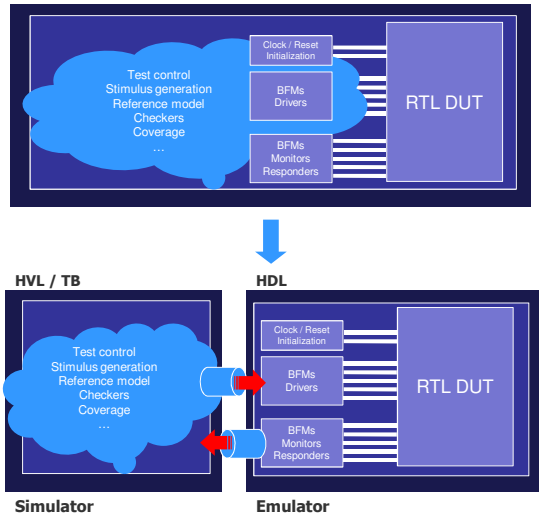
Interoperability with Software Simulator



One Testbench for Simulation and Acceleration

15 UVM & Emulation, DVCon India 2015, HvdS

Partitioning of Testbench



- Unified dual-top testbench
  - Synthesizable HDL domain with DUT, clocks and BFMs
  - Untimed HVL domain with generation and analysis
- Transaction-based cross-domain communication
  - Function/task calls between HVL and HDL domains
- Main considerations
  - Testbench architecture
  - HVL-side modeling
  - HDL-side modeling
  - HVL-HDL communication
  - Performance

16 UVM & Emulation, DVCon India 2015, HvdS

Dual Domain Testbench Architecture

HVL/TB Side	HDL Side
1. Untimed	1. Timed
2. Behavioral	**2. Synthesizable**
3. Class-based	3. Module/interface based
4. Dynamic	4. Static
5. Communication with HDL side only through transactors	**5. Communication with HVL side only through transactors**
6. Programming optimization techniques dictate performance	6. Synthesis skill and transactor design dictate performance
7. Changes don't cause emulation recompile	7. Changes may require emulation recompile
8. Standards like UVM apply	8. XRTL and synthesis standards apply
9. Verification engineer's comfort zone	9. ASIC designer's comfort zone

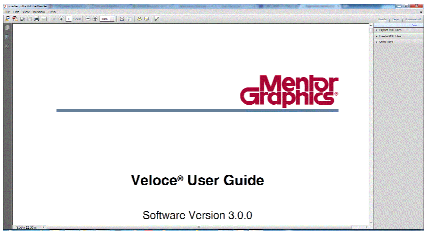
- Emulation-friendly: Separated TB-HDL domains + untimed TB
- Emlation-ready: emulation-friendly + synthesizable HDL domain

Untimed Testbench

- No # delays
- No clocks – e.g. *@(posedge clk)*
- No waits for fixed time intervals – e.g. *wait(1 ns)*
- All thread synchronization is via abstract events, not by time advance
  - Semaphore posts
  - Transactions arriving on data channels
  - Blocking reads on streaming pipes
  - Returns of blocking calls to the HDL side
- Testbench is still "time aware" and can access variables like *$time*
- Testbench can indirectly control time advancement
  - Initiating "remote" HDL task or function calls, i.e. HDL advances time while HVL threads block
  - Waiting for responses/notifications from HDL side
  - Time advance is monitored by a transactor (an HDL clock counter)

Effective HDL Modeling: Veloce XRTL

- RTL subset
- SystemVerilog interface
- SystemVerilog Virtual Interface
- SystemVerilog final block
- SVA sampling functions ($rose, $fell, $past, $stable, $changed, $sampled)
- SVA functions $countones, $onehot, $onehot0, $isunknown
- Language based force/release
- C-API based force/release
- $testplusargs and $valueplusargs
- Initial blocks
- Behavioral clock/reset generation
- System tasks; $display, $fdisplay etc.
- Implicit state machines
- Named events and waits
- DPI function/task calls
- SCE-MI 2.0 compliant transaction pipes
- Clocked tasks
- Gated clocks
- Variable delay clocks
- Multiple drivers
- Memory array
- SV cover groups



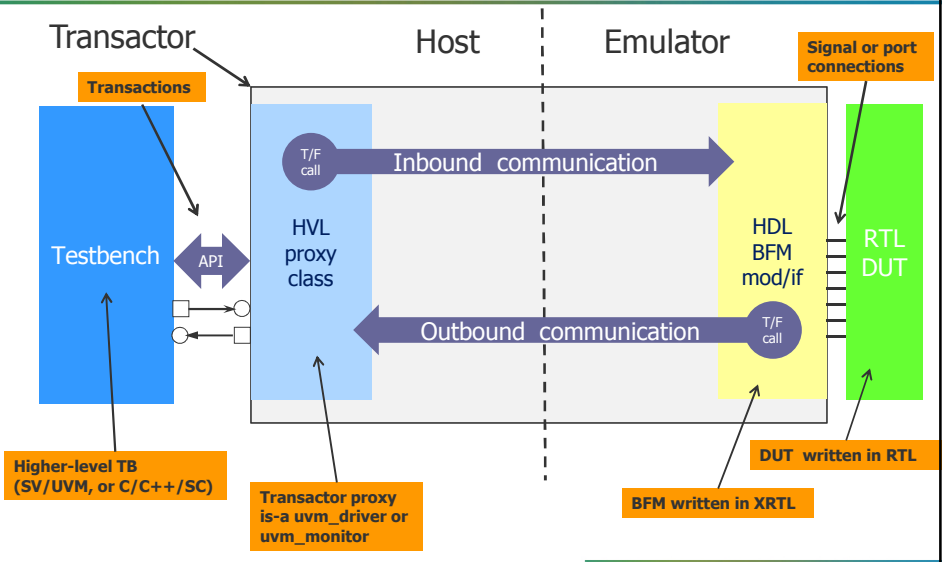
See full list and description in Veloce user guide

Development of synthesizable HDL BFMs with standards-based behavioral language constructs (IEEE P1800 and Accelera SCEMI 2)

19 UVM & Emulation, DVCon India 2015, HvdS

Transactors – Acceleration Building Blocks

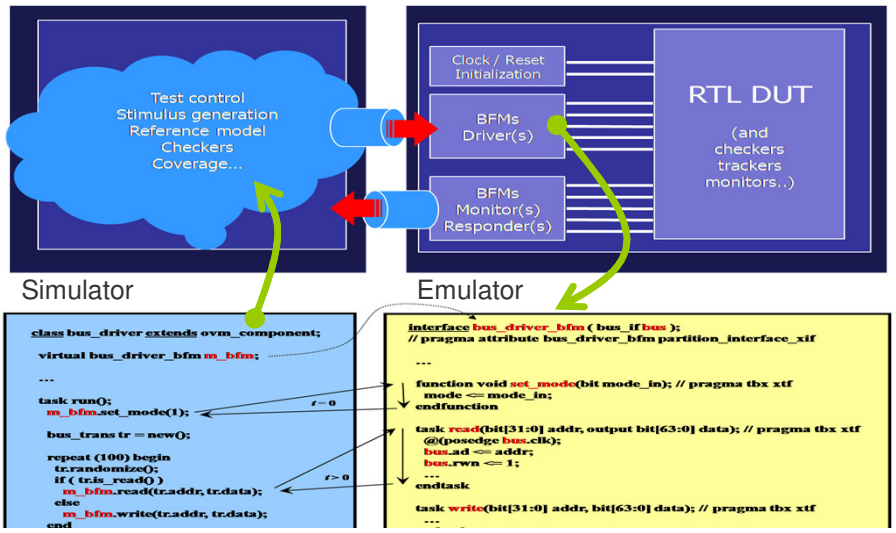
HDL BFM + HVL Proxy + HVL-HDL Channel



20 UVM & Emulation, DVCon India 2015, HvdS

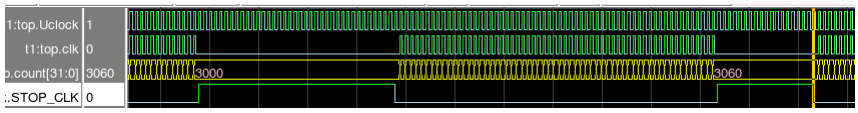
HVL-HDL Communication

Native SystemVerilog virtual interface (VIF) based reactive communication



Acceleration Performance

- Is execution H/W or S/W bound?



  - Co-emulation can start and stop the design clocks (clk)
    - Design clocks are derived from free running emulator clock (Uclock)
    - Design clocks stop during testbench and communication activity

- Total run-time = $t_{[HDL]} + t_{[HVL]} + t_{[HVL\text{-}HDL]}$

- Want H/W bound : $t_{[HDL]} / t_{[total]} >> (t_{[HvL]} + t_{[HVL\text{-}HDL]}) / t_{[total]}$
  - "Healthy" throughput
  - Design clocks active high % of time, i.e. low testbench and communication overhead

Veloce Performance Profiling

- H/W bound: 131095/131279 = 0.998
  - 99.8% time spent in H/W

```
                SIMULATION STATISTICS
Simulation finished at time 26219000
Total number of TBX clocks:                                        131279
Total number of TBX clocks spent in HDL time advancement:          131095
Total number of TBX clocks spent in HDL due to callee execution:        4
Percentage TBX clocks spent in HDL time advance:                  99.86 %
--------------------------------------------------------------------------------
Total CPU time (user mode):                                     0.10 seconds
Total time spent:                                               0.09 seconds
```

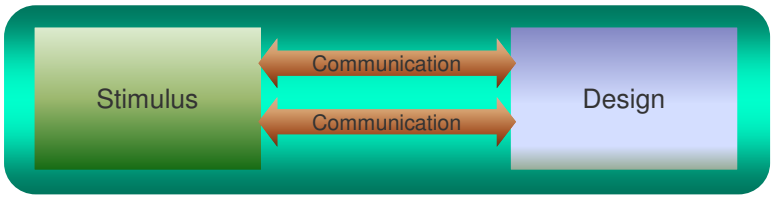
- S/W bound (HVL): 4207527/250298571=0.0168
  - 1.68% time spent in H/W

```
                    SIMULATION STATISTICS
Simulation finished at time 420752700
Total number of TBX clocks:                                     250298571
Total number of TBX clocks spent in HDL time advancement:         4207527
Total number of TBX clocks spent in HDL due to callee execution:  3016794
Percentage TBX clocks spent in HDL time advance:                   1.68 %
--------------------------------------------------------------------------------
Total CPU time (user mode):                                    35.78 seconds
Total time spent:                                             165.21 seconds
```


Optimizing Performance

- Reduce communication overhead by optimizing transaction utilization ($t_{[HVL\text{-}HDL]}$)
  - Increasing transaction sizes – larger transactions stay inside DUT longer
  - Using SCE-MI pipe-based data shaping
  - Raising abstraction to meta-transactions
  - Minimizing fine-grain scoreboarding, memory access frequencies
  - Optimizing physical packet transfers, multiple co-model channels
- Reduce testbench overhead by optimizing simulation performance ($t_{[HVL]}$)
  - Heeding file I/O, constraint solving, messaging & macro usage (UVM)
  - Compiling with optimization switches
- Enhance H/W execution by optimizing emulation frequency ($t_{[HDL]}$)
  - Improving critical paths
  - Optimizing emulator clock utilization
    - Aligning design clocks (CFR), using inactive edge optimization
  - Maximizing parallelism in BFMs
- Concurrency
  - Increase throughput by absorbing communication and testbench execution during H/W execution ($t_{[total]}$ decreases while $t_{[HDL]}$ stays the same)
- Detailed assessment through profiling, analysis, potential remodeling, etc.

Performance Bottlenecks and Choices



Stimulus

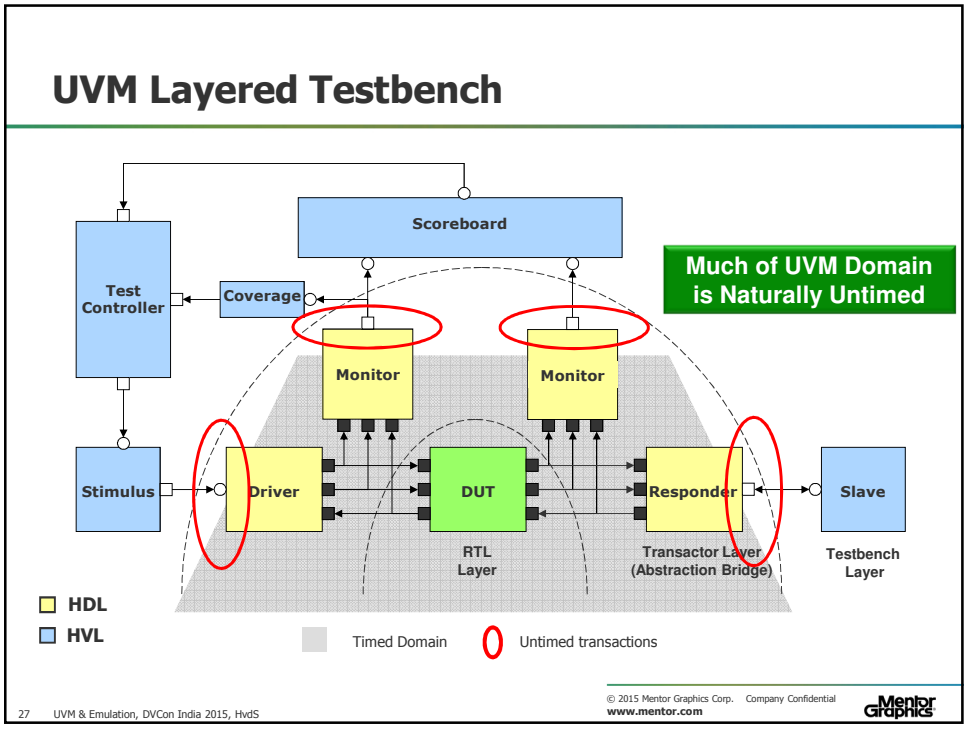
- Questa linter
- Questa profiler
- Gprof for C/C++

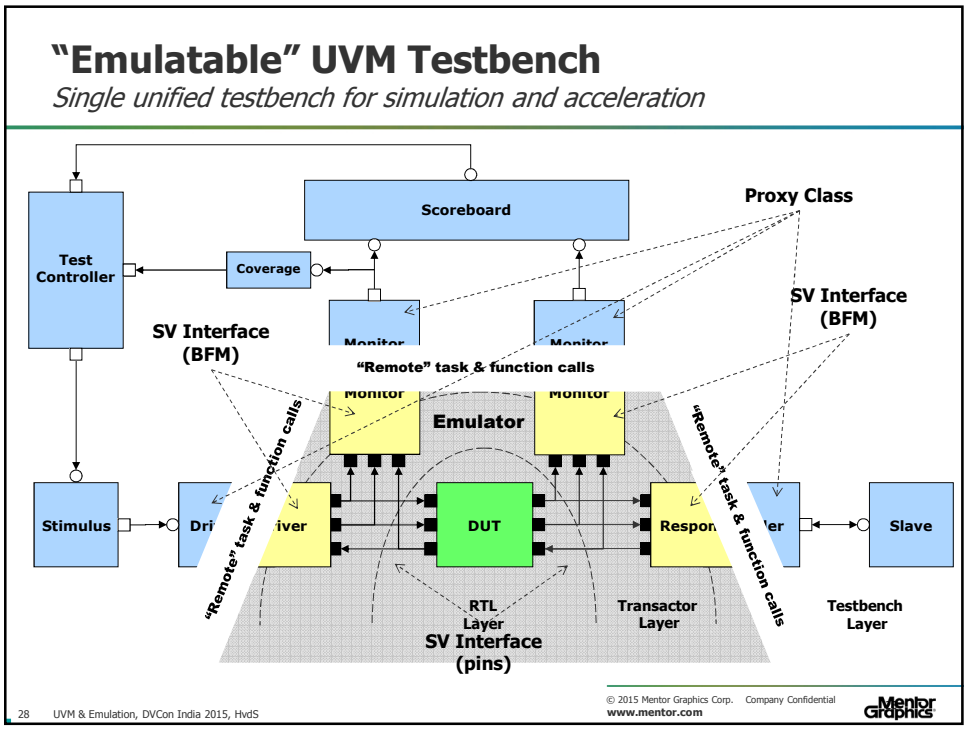
Communication

- TBX performance profiler
- Multiple co-model channels
- 3K bits TAPI packets
- HW/SW concurrency
  - Delayed return DPI
  - Outbound streaming
  - Inbound streaming DPI
  - Out of blue calls
  - Pure pipes

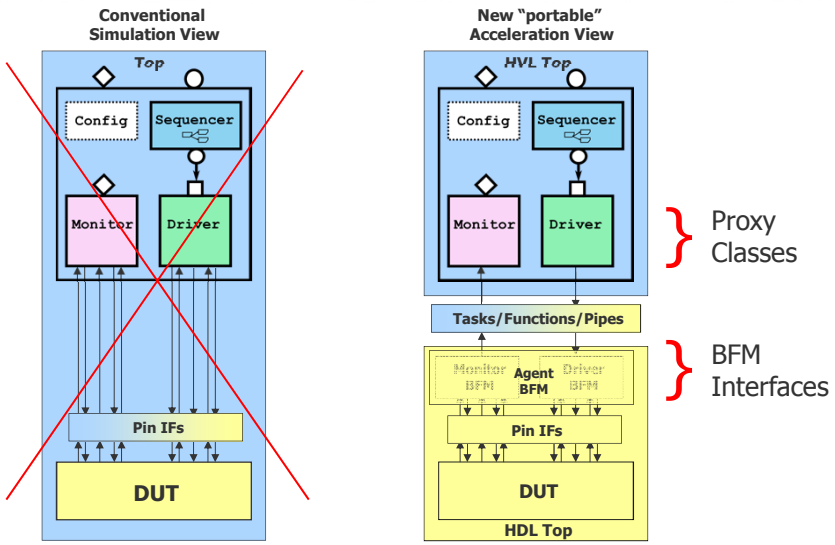
Design

- Performance advisor
- Velcp performance analysis
- CFR
- Negedge optimization
- Long paths and loops visualization tool

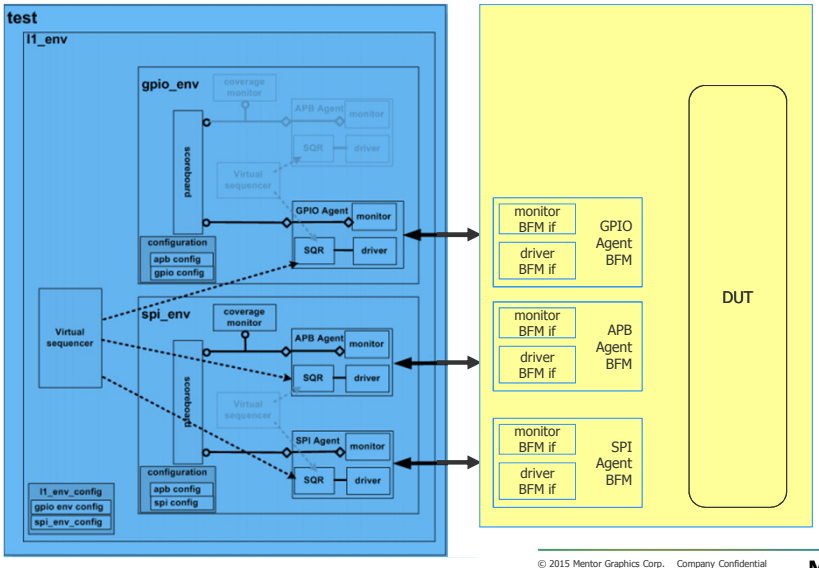




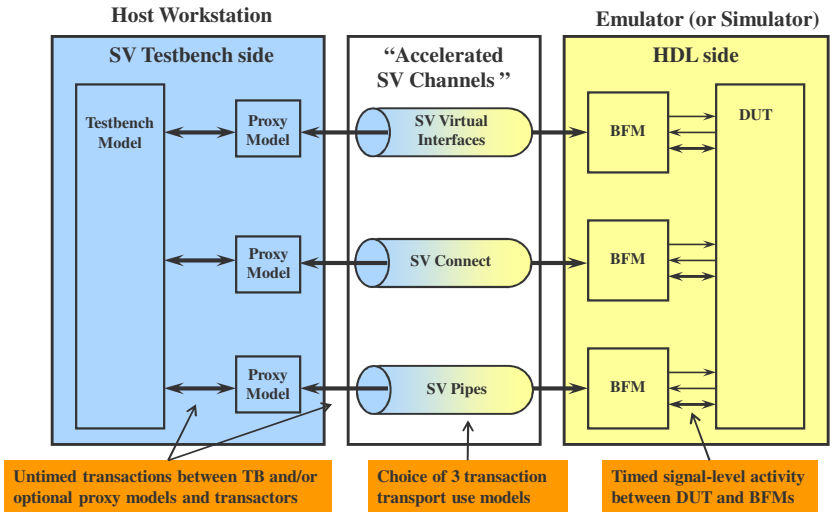
"Emulatable" UVM Agents



Vertical Reuse in Dual Domain UVM Testbench



UVM – HDL Communication Use Models



Example UVM Driver

VIF-based use model

```
class ahb_seq_item
```

```
class ahb_driver
   extends uvm_driver #(ahb_seq_item);

virtual ahb_driver_bfm bfm;

...

virtual task run_phase(uvm_phase phase);
  bfm.wait_for_reset();
  forever begin
    seq_item_port.get_next_item(req);
    bfm.drive(req.we,
              req.addr, req.data, ...);
    seq_item_port.item_done();
   end
endtask

...

endclass
```

Virtual interface pointer from testbench side to HDL-side BFM

```
interface ahb_driver_bfm(ahb_if pins);

...

task wait_for_reset();
  ...
endtask

task drive(bit we,
           bit [31:0] addr, data, ...);
  @(posedge pins.clk);
  // Drive request on pin i/f
  ...
endtask

endinterface
```

Time consuming task called directly from the testbench side to the HDL side

The UVM driver wiggles the DUT pins indirectly, no longer directly

- Flexible modeling options:
  - Separate read/write calls
  - Separate address/data transfers (possibly forked in parallel)
  - Reactive vs streaming communication

Keeping with Transaction Objects

- Classes and other non-synthesizable types should not be used as HDL BFM function/task argument types
  - Ok for simulation, not for emulation
- HVL side can explicitly convert between transaction objects and suitable packed-type representations for BFM function/task arguments
  - E.g. packed structs

Shared HVL-HDL package

```
import ahb_types_pkg::*;

task ahb_driver::run_phase(uvm_phase phase);
  bfm.wait_for_reset();
  forever begin
    ahb_seq_item_s req_s;
    seq_item_port.get_next_item(req);
    req.to_packed_struct(req_s);
    bfm.drive(req_s);
    seq_item_port.item_done();
  end
endtask
```

```
interface ahb_driver_bfm(ahb_if pins);
import ahb_types_pkg::*;

...

task drive(ahb_seq_item_s req);
  @(posedge pins.clk);
  // Drive request on pin i/f
  ...
endtask

endinterface
```

Conversion to packed struct before calling BFM task

HVL-HDL Transaction Conversion

- Recommend user-defined object conversion methods targeted for optimal HVL-HDL communication modeling and performance

Shared HVL-HDL package

```
import ahb_types_pkg::*;

class ahb_seq_item extends uvm_sequence_item;

function void to_struct(ahb_seq_item_s s);
  {s.we, s.addr, s.data, s.delay, s.error} =
    {this.we, this.addr, this.data, this.delay, this.error};
endfunction

function void from_struct(ahb_seq_item_s s);
  ...
endfunction

endclass
```

Optimization: byte vs. int

```
package ahb_types_pkg;

...

typedef struct packed {
   bit we;
   bit [31:0] addr;
   bit [31:0] data;
   bit  [7:0] delay;
   bit error;
} ahb_seq_item_s;

...

parameter int AHB_SEQ_ITEM_NUM_BITS = $bits(apb_seq_item_s);
parameter int AHB_SEQ_ITEM_NUM_BYTES = (APB_SEQ_ITEM_NUM_BITS+7)/8;
typedef bit [APB_SEQ_ITEM_NUM_BITS-1:0] ahb_seq_item_vector_t;

endpackage
```

- UVM offers virtual pack/unpack methods, though no standard way for implementing packing/unpacking transactions

Example UVM Monitor

VIF-based use model

- Same idea, but ...

```
import ahb_types_pkg::*;

task ahb_monitor::run_phase(uvm_phase phase);
  bfm.wait_for_reset();
  forever begin
    ahb_seq_item_s req_s;
    bfm.sample(req_s);
    req.from_struct(req_s);
    ap.write(req);
  end
endtask
```

```
interface ahb_monitor_bfm(ahb_if pins);
import ahb_types_pkg::*;

...

task sample(output ahb_seq_item_s req);
  @(negedge pins.clk);
  // Sample request on pin i/f
  ...
endtask

endinterface
```

- **But** more natural to have the monitor BFM "push" instead of the proxy "pull" transactions out
  - Let BFM be initiator calling proxy function through back-pointer
- Can yield much better performance for UVM analysis traffic
  - Outbound void functions are one-way non-blocking calls that do generally not impose emulator clock stoppage

Recommended UVM Monitor

VIF-based use model

```
import ahb_types_pkg::*;

class ahb_monitor extends uvm_monitor;

virtual ahb_monitor_bfm bfm;

...

function void connect_phase(uvm_phase phase);
  ...
  bfm.proxy = this;
endtask

task run_phase(uvm_phase phase);
  bfm.run();
endtask

function void write(ahb_seq_item_s req_s);
  req.from_struct(req_s);
  ap.write(req);
endfunction
...

endclass
```

Assigning the back-pointer in the build or connect phase

Function call via back pointer from BFM back to monitor proxy instance in testbench

```
interface ahb_monitor_bfm (ahb_if pins);
import ahb_types_pkg::*;

...

import ahb_pkg::ahb_monitor;
ahb_monitor proxy;

function void run();
  -> start;
endfunction

initial begin
  @(start);
  @(negedge pins.clk);
  monitor_daemon();
end

task monitor_daemon();
  forever begin
    // Sample next request on pin i/f
    ...
    proxy.write(req_s);
  end
endtask

endinterface
```

Package import of back-pointer class type

Time consuming FSM initiated from the testbench side via 0-time function call

BFM – Proxy Binding: uvm_config_db::set

- HDL-side can "register" a BFM interface handle in the UVM configuration database
  - Right where the BFM is instantiated, i.e. in HDL top or below in agent BFM if used
- Use a unique string as registration "key" to be used to access the virtual BFM interface later from the UVM testbench domain
  - E.g. the hierarchical BFM instance path

```
module hdl_top();
...

ahb_monitor_bfm ahb_mon (ahb_if);

initial begin
  import uvm_pkg::uvm_config_db;
  uvm_config_db #(virtual ahb_monitor_bfm)::
    set(null, "uvm_test_top",
              $psprintf("%m.ahb_mon"),
        ahb_mon);
  end
endmodule
```

```
module hdl_top();
...

ahb_monitor_bfm ahb_mon (ahb_if);

initial begin
  import uvm_pkg::uvm_config_db;
  uvm_config_db #(virtual ahb_monitor_bfm)::
    set(null, "VIRTUAL_INTERFACES",
              "ahb_mon_bfm_0",
        ahb_mon);
  end
endmodule
```

Registration key as combination of inst_name and field_name strings

BFM – Proxy Binding: uvm_config_db::get

- UVM domain can retrieve the virtual BFM interface from the UVM configuration database with the given registration key
- Typically done via the corresponding agent's configuration object at testbench top with a global bird's eye view of the entire environment
  - Get virtual interface from uvm_config_db and assign to a config object member
  - Register the config object in the UVM config database per usual
  - Retrieve the config object in the agent, and extract the virtual interface

```
import uvm_pkg::*;
class ahb_configuration extends
  parameterized_agent_configuration_base#(.TRANS_T(ahb_seq_item));

...

virtual interface ahb_monitor_bfm  ahb_mon_bfm;

virtual function void configure_interface(..., string bfm_interface_name);
  if (!uvm_config_db #(virtual ahb_monitor_bfm)::get(
      null, "VIRTUAL_INTERFACES", bfm_interface_name, ahb_mon_bfm))
    `uvm_error(...)
  );
```

Retrieving the virtual interface handle from uvm_config_db into the configuration object

More Examples: Time Advancement

```
class bus_driver extends uvm_component;

virtual bus_driver_bfm m_bfm;

...

task run_phase(uvm_phase phase);
  ...
  m_bfm.advance_hdl_time(100);
  ...
endtask

task advance_hdl_time(int n);
  m_bfm.advance_time(n);
endtask

...

endclass
```

```
interface bus_driver_bfm ( bus_if bus );
// pragma attribute bus_driver_bfm
    partition_interface_xif

...

task advance_hdl_time(input int n);
                          // pragma tbx xtf
  @(posedge clk);
  assert(n > 0);
  repeat (n-1) @(posedge clk);
endtask

endinterface
```


More Examples: Interrupt Monitoring

```
class bus_monitor extends uvm_component;

virtual bus_monitor_bfm m_bfm;

...

event interrupt_notification;
bit[31:0] interrupt_status;

function void notify_interrupt(
                output bit[31:0] status);
   interrupt_status = status;
   -> interrupt_notification;
endfunction

task run_phase(uvm_phase phase);
   fork
      forever begin
         @interrupt_notification;
         ...
      end
   join none
   ...
endtask

endclass
```

```
interface bus_monitor_bfm ( bus_if bus );
// pragma attribute bus_monitor_bfm
    partition_interface_xif

   bus_monitor proxy;
   // pragma tbx oneway proxy.notify_interrupt

    ...

   always @(negedge bus.intn) begin
      ...
      int_status = read_int_status_register(...);
      proxy.notify_interrupt(int_status);
      ...
   end

 endinterface
```

More Examples: Error Injection

```
class my_driver extends uvm_component;

  protected virtual my_driver_bfm bfm;

  ...

  task issue_request(REQ req);
    req.to_struct(req_s);
    bfm.issue_request(req_s);
  endtask

endclass

class my_err_driver extends my_driver;
  ...

  task issue_request(REQ req);
    bfm.set_err_type(err);
    super.issue_request(req);
  endtask

endclass
```

```
interface my_driver_bfm (my_pin_if pif);
// pragma attribute my_driver_bfm
    partition_interface_xif

  ERR_TYPE err;

  function void set_err_type (ERR_TYPE err_in);
     err = err_in;
  endfunction

  task issue_request(input request_s req);
  // pragma tbx xtf
    pif.opa <= req.a;
    pif.opb <= (err == OP_ERR) ? ~req.b : req.b;
    pif.fpu_op <= req.op;
    pif.rmode <= req.round;
    pif.start <= 1;
    @(posedge clk);
    pif.start <= 0;
  endtask

  ...
```


More Examples: Error Injection (Take 2)

```
class my_driver extends uvm_component;

   protected virtual my_driver_bfm bfm;

   ...

   task issue_request(REQ req);
      req.to_struct(req_s);
      bfm.issue_request(req_s);
   endtask

   virtual function bit is_err_request();
      return 0;
   endfunction

endclass

class err_driver extends my_driver;
   ...

   function bit is_err_request();
      return 1;
   endfunction
```

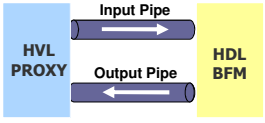
```
interface fpu_driver_bfm (fpu_pin_if fpu_if);
// pragma attribute fpu_driver_bfm
    partition_interface_xif

   fpu_driver_core #(…) proxy;

   task issue_request(input fpu_request_s
    req); // pragma tbx xtf
      fpu_if.opa <= req.a;
       if (proxy.is_error_request())
          fpu_if.opb <= ~req.b;
       else
          fpu_if.opb <= req.b;

       fpu_if.fpu_op <= req.op;
       fpu_if.rmode <= req.round;
       fpu_if.start <= 1;
       @(posedge clk);
       fpu_if.start <= 0;
    endtask

…
```

Streaming vs Reactive Transactions

- Reactive transactions (what we've seen so far):
  - Sending or receiving data "instantaneously", in one simulation delta-time
    - Caller and callee
  - May be dependent on the current state of the testbench and/or DUT
  - SV virtual interface (BFM) and class handle (proxy) based function calls
    - For SVTB/UVM only; alternative to SV-DPI imports/exports
  - Examples: register loads, interrupt responses, sending data that needs to be consumed immediately
- Streaming transactions:
  - Producer and consumer of data are largely decoupled
  - Little or no dependence on state
    - D[N+1] does not depend on result of sending D[N]
  - Examples: Audio, Video, Ethernet traffic
  - Semantics of control transfer is defined by the intermediary
    - SCEMI 2.x pipes
  - Additional notes:
    - All streaming transactions can be built from reactive transactions
    - Co-emulation solution creates buffers and other invisible infrastructure

SCEMI 2 Transaction Pipes – Overview

- Accellera SCEMI 2.x pipes specifically address transaction streaming, data-shaping and variable length messaging
  - A transaction payload is represented as a variable number of fixed-sized bit-vector elements
  - Deferred visibility semantics can give optimized performance for specific scenarios if used right
- HVL and HDL sides call APIs to read/write from/to a pipe
  - Blocking and non-blocking send/receive calls
- Pipes are unidirectional
  - Input pipes allow data flow from HVL to HDL (proxy to BFM)
  - Output pipes allow data flow from HDL to HVL (BFM to proxy)
- Pipes are deterministic
  - Produce identical results in simulation & emulation



UVM Acceleration Flow Summary

- Employ two distinct UVM and HDL top level modules
  - UVM top must be untimed; HDL top must be synthesizable for emulation
    - DUT, pin interfaces, and clock/reset logic can be largely preserved
    - Upper testbench layers should remain (largely) unaffected
  - Separate file lists for compilation required too!
- Split UVM drivers/monitors into untimed UVM proxies and timed HDL BFMs
  - BFMs are modeled as SV interfaces accessing separate SV pin interface
    - Implemented using implicit FSMs and other "RTL++" constructs
    - Used for testbench-HDL binding instead of (virtual) pin interfaces
  - Proxies encapsulate intra-transactor communication
    - Hide BFM tasks and functions which are visible only to the proxy
    - Represent interface to upper UVM testbench layers (remains unchanged)
    - Are generally light-weight, implementing basic threads to pass generated UVM stimulus to HDL side, and observed HDL responses back to UVM side
  - Transaction objects must be converted to/from synthesizable BFM task and function arguments
    - Internal to UVM proxies, e.g. using "to_struct" and "from_struct" methods
- Tune UVM-HDL communication interface for optimal performance
  - Reactive vs. streaming, inbound vs. outbound, one-way vs. two-way
  - E.g. increased transaction sizes, SCEMI data-shaping features, ...

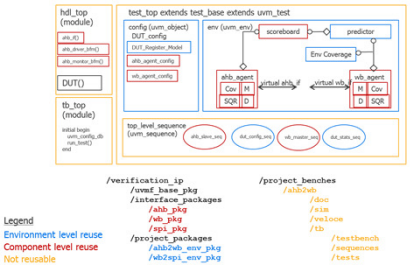
Advanced UVM Co-Emulation Considerations

- Topology of HDL BFMs cannot be elaborated dynamically
  - But HVL proxies can control (suspend, resume) model behavior dynamically, i.e. "self-starting" HDL threads can be avoided
  - Or can use shared package of static test parameters along with SV generate constructs to control common topology among both HVL and HDL sides
- HDL BFMs cannot be created using UVM factory
  - But HVL proxies can
- HDL BFMs cannot be configured and controlled by UVM configuration mechanism
  - But HVL proxies can
- HDL BFMs can contain SystemVerilog covergroups too
  - Basic data-oriented functional coverage inside BFMs to complement normal UVM domain coverage

Mentor's UVMF – Unified Simulation-Emulation Framework

UVM Framework – Unified Architecture

Enabling the benefits of both UVM and emulation with less effort



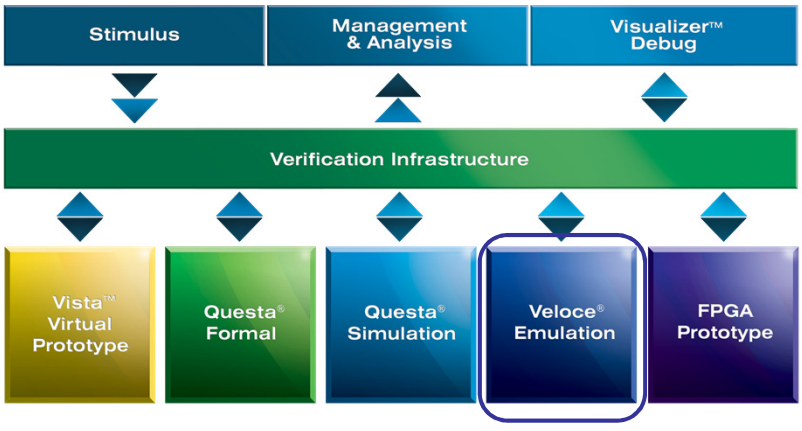
- Standards-based **UVM F**ramework that reinforces testbench acceleration from conception
- Two-top testbench methodology built on existing UVM class library
- Source code, documentation, examples, automated code generation
- Proven simulation track record on production designs at 20+ companies

Tool Support for Legacy Code Migration

- Questa #-delay tracking for all sequential delays in HVL code
  - Linting to help guard against unit time advances in untimed SV HVL
  - Generate logs capturing #delays in the testbench during Questa compile
  - Message for the #delays causing an issue or hang at runtime
- Questa hierarchical reference tracking
  - Linting to help guard against cross-boundary references

Mentor Enterprise Verification Platform

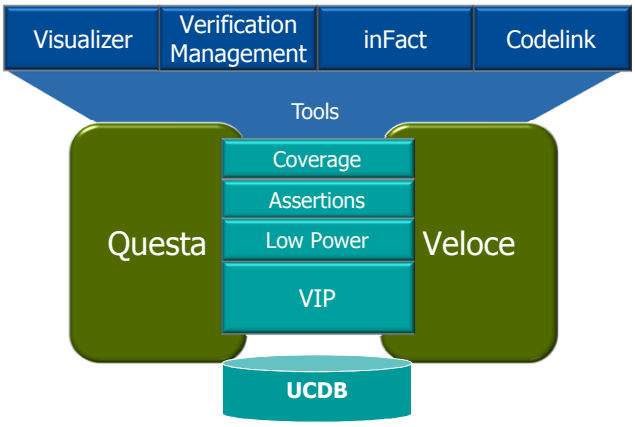
Announced in April 2014



50 UVM & Emulation, DVCon India 2015, HvdS

© 2015 Mentor Graphics Corp. Company Confidential
www.mentor.com

Questa & Veloce from Simulation to Emulation



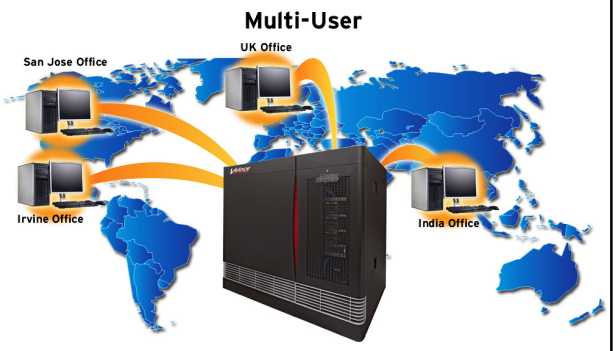
51 UVM & Emulation, DVCon India 2015, HvdS

Applications around Emulation Platform

52 UVM & Emulation, DVCon India 2015, HvdS

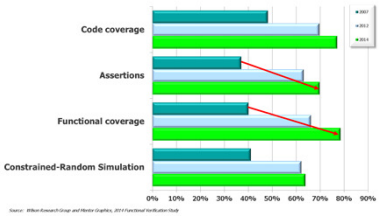
OS3/Enterprise Server Enables Multi-Users

- Multi-user access
- Active queue management for prioritizing jobs
- Maximizes investment



CDV & ABV from Simulation to Emulation

Use of Advanced Verification Techniques

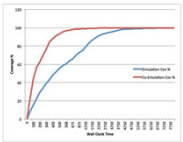


Why Emulation for Coverage?

- Increased use of SystemVerilog assertions and functional coverage inside the RTL DUT and transactors/BFMs
- Scenarios requiring long test sequences, OS boot, and real world stimulus
- Cross coverage for multicore environments or interaction among different blocks
- New methodologies like power management require coverage of power states at system level

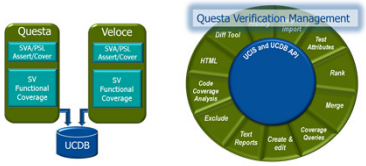
System Level Coverage Closure

- Comprehensive Veloce support for SVA assertions and SV cover groups, points, bins and crosses
- Standard UCIS support – compatible with Questa
- Capacity-wise coverage by 'skipping' covered coverpoints and saturating bin counts



Coverage/Assertions with Questa and Veloce

Unified UCDB as Single Place to Store and Analyze Metrics



- Same assertions and coverage – results go into Mentor UCDB
- Questa verification management – 100s of man years in analysis tools

Emulation for Coverage

Scalable Capacity
Handle Large SoC (RTL/Gate) with Coverage

SPEED

Performance
Logarithmic vs Exponential

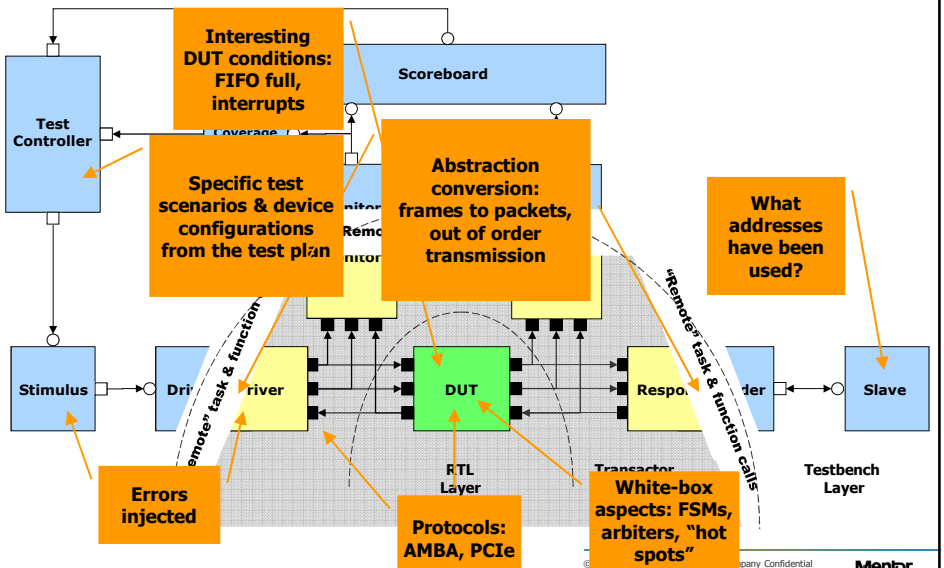
Boot the OS & Run Real Applications
Reach Corner Case Scenarios

UNIFIED POWER FORMAT

Power Management Needs Coverage

55 UVM & Emulation, DVCon India 2015, HvdS

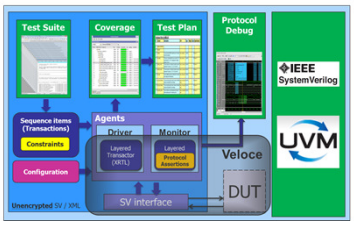
UVM Acceleration & Coverage



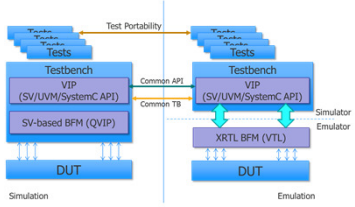
56 UVM & Emulation, DVCon India 2015, HvdS

Simulation and Emulation VIP

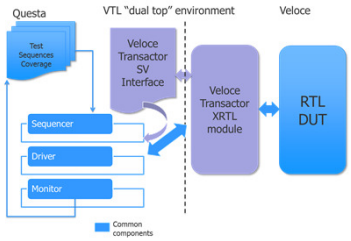
Questa and Veloce Testbench-Driven VIP



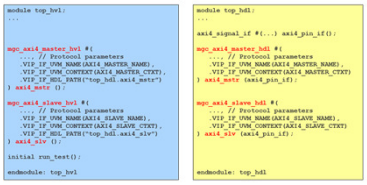
Questa and Veloce VIP Portability



Questa to Veloce VIP Migration



Dual Top Portable VIP – Code Sample

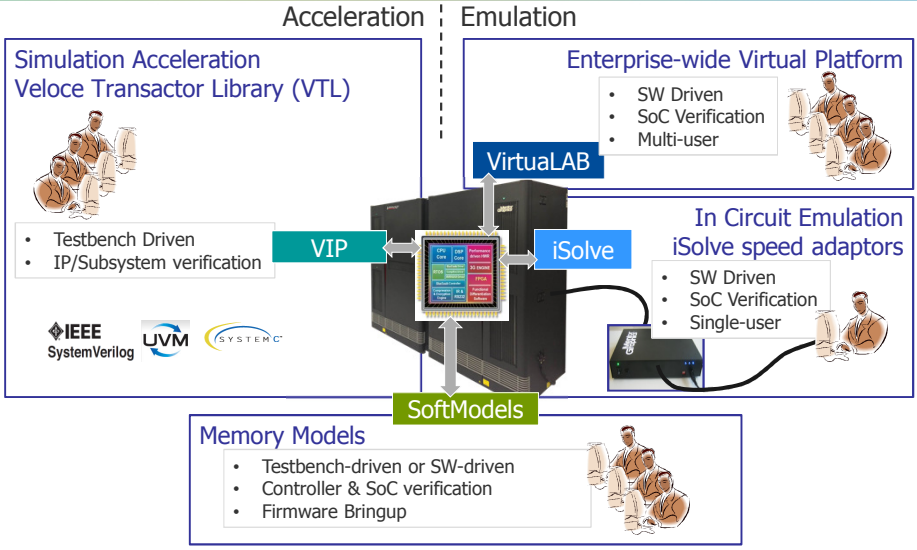


- Switch between simulation vs. emulation VIP: `+define+XRTL`

Emulation Protocol Solutions

Solutions Have Their "Sweet Spots"

Veloce2

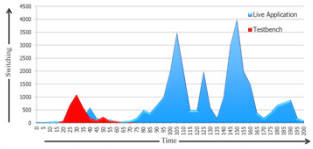


Emulation for Power Verification

Emulation for Power

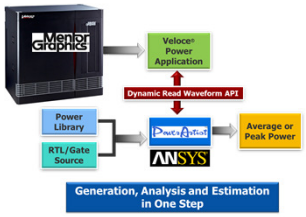
Veloce2

Testbench vs. Live Application

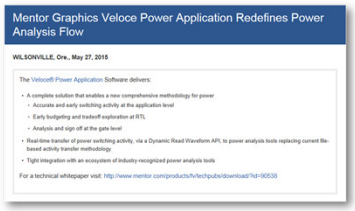


	Testbench	Live Application
Stimulus	Functional Testbench	Live Application after OS Boot
Accuracy	False Power Peaks	Real Power Peaks

Veloce Power Application Ecosystem



Mentor Announcement


Mentor Graphics Veloce Power Application Redefines Power Analysis Flow

WILSONVILLE, Ore., May 27, 2015

The Veloce® Power Application Software delivers:

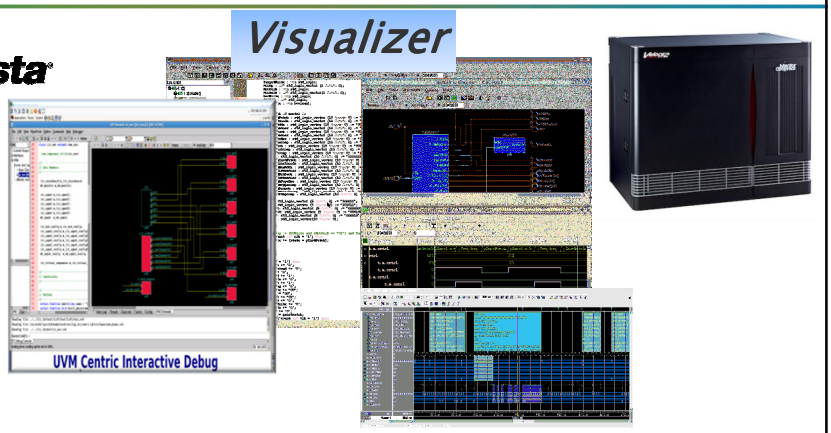
- A complete solution that enables a new comprehensive methodology for power
  - Accurate and early switching activity at the application level
  - Early budgeting and tradeoff exploration at RTL
  - Analysis and sign off at the gate level
- Real-time transfer of power switching activity, via a Dynamic Read Waveform API, to power analysis tools replacing current file-based activity transfer methodology
- Tight integration with an ecosystem of industry-recognized power analysis tools

For a technical whitepaper visit: http://www.mentor.com/products/fv/techpubs/download?id=90538

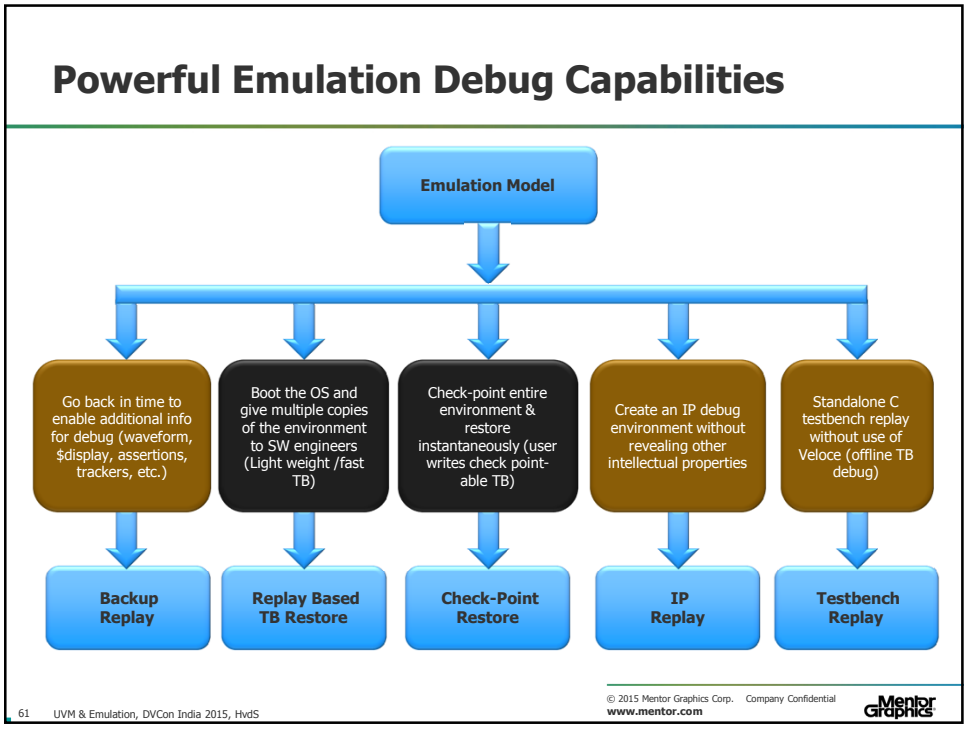
Questa and Veloce Common Debug

Maximize Performance without sacrificing ease of ese

Questa Visualizer



- Same unified debugger for pure simulation and acceleration
  - Questa running testbench and Veloce running HDL domain with DUT
- Emulator savvy, easy, powerful and FAST



UVM Acceleration Methodology Benefits

- Enables unified testbench for simulation and acceleration
  - Single source testbench to create and maintain
- Leverages power of SV and UVM testbench features
  - OOP, CRT, CDV, ABV, intelligent checking
  - UVM TLM interfaces, sequences, configuration, phasing, factory
- Is unobtrusive to established verification best practices
  - Co-emulation guidelines largely reinforce prevalent verification methodology guidelines – i.e. UVM layered architecture
- Veloce+Questa offers the opportunity for significant improvements in performance and productivity
  - 1-2MHz emulator frequency, 2-3 Gbps co-model channel bandwidth
  - Unified coverage closure
  - "Multi-platform" VIP
  - Comprehensive debug paradigm
  - Integrated tool flow and experience

Results, Collateral

Testbench Acceleration Results with Veloce

	Design Size (gates)	Simulation Time (hrs)	Acceleration Time (sec)	Speed-up
Application Processor	+200M	151	3060	177X
Network Switch	34M	16 ½	240	245X
Graphics Sub-System	8M	86 ½	635	491X
Mobile Display Processor	1.2M	5	46	399X
Memory Controller	1.1M	5	308	60X
Face Recognition Engine	1M	½	6.58	128X
Wireless Multi-Media Sub-System	1M	53	658	288X
Raid Engine Controller I	25M	13	174	268X
Raid Engine Controller II	25M	15.5	327	171X

ST Microelectronic Success Story

- Previous acceleration T.A.T
  - 3-4 weeks effort
- ST deployed Questa & Veloce
  - UVM, VIP, TBX
- Current Questa to Veloce T.A.T
  - A few hours

Co-emulation will be essential moving ahead, largely because it is a much more efficient way to approach verification. Specifically, it provides a much better way to accelerate simulation while preserving familiar testbench architecture and methodologies.

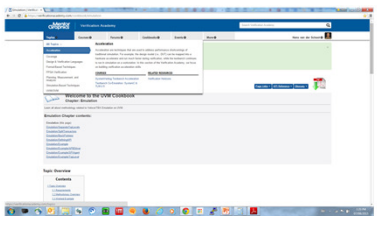
Alberto Allara, Engineering Manager, ST Microlelectronics

Collateral for Further Learning

- Verification Academy
  - Course: SystemVerilog Testbench Acceleration
  - UVM Cookbook: Testbench Acceleration through Co-Emulation
- Publications
  - MGC 2015: "From Simulation to Emulation – A Fully Reusable UVM Framework"
  - EVLSIDCS 2013: "Accelerating SystemVerilog UVM-based VIP to Improve Methodology for Verification of Image Signal Processing Designs Using HW Emulator"
  - DVCon 2013/TechOnLine: "Unifying Hardware Assisted Verification and Validation using UVM and Emulation"
  - DAC 2012: "Development of a Unified Platform for Accelerated SoC Verification and Validation"
  - MGC 2012: "Simulation + Emulation = Verification Success"
  - TechOnLine India, 2011: "Taking Verification Productivity to the Next Level"
  - DVCon 2011: "Off to the Races with Your Accelerated SystemVerilog Testbench"
  - DVCon 2008: "An Accelerateable UVM Methodology based on SCE-MI 2"

Mentor Verification Academy & Cookbooks

All About Verification Methodology and Technology Integration



Summary

- UVM offers proven verification productivity through reuse
- Creating an emulation-ready UVM testbench requires architecture considerations but performance benefits are substantial
- Your next UVM project should be planned for unified simulation and acceleration to boost block-to-system verification productivity

Thank You!

Questions?