UVM & Emulation

Architecting SystemVerilog UVM Testbenches for Simulation-Emulation Portability to Boost Block-to-System Verification Productivity

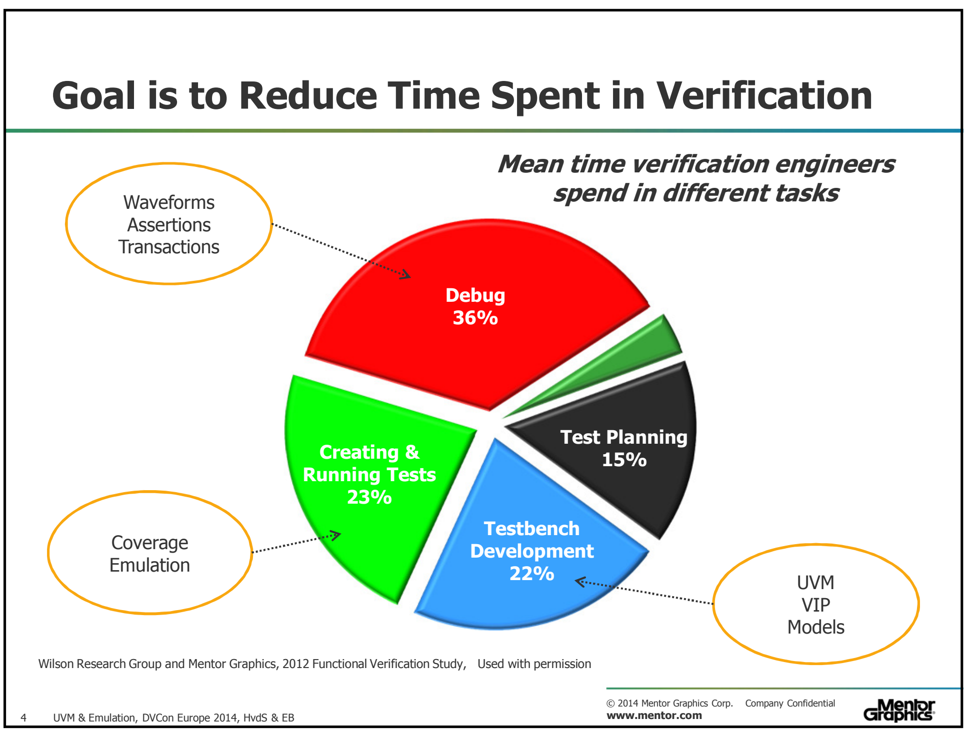
Hans van der Schoot, Ph.D.
Emulation Technologist
Mentor Emulation Division
hans_vanderschoot@mentor.com

Ellie Burns-Brookens
Simulation Product Manager
Design Verification Technology
ellie_burns@mentor.com

Agenda

- Introduction
- Fundamentals of Hardware-Assisted Testbench Acceleration
- Unified Testbench Architecture & Methodology for UVM Acceleration
- Unified Paradigm for Transaction-Based Coverage, Assertions & Debug
- Results & Wrap Up
- Q & A


INTRODUCTION

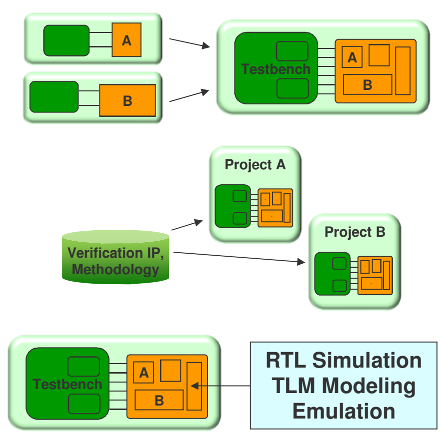

Goal is to Reduce Time Spent in Verification
Mean time verification engineers spend in different tasks
Waveforms
Assertions
Transactions
Debug
36%
Test Planning
15%
Creating & Running Tests
23%
Testbench Development
22%
Coverage
Emulation
UVM
VIP
Models
Wilson Research Group and Mentor Graphics, 2012 Functional Verification Study, Used with permission
4 UVM & Emulation, DVCon Europe 2014, HvdS & EB
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com

Verification Productivity

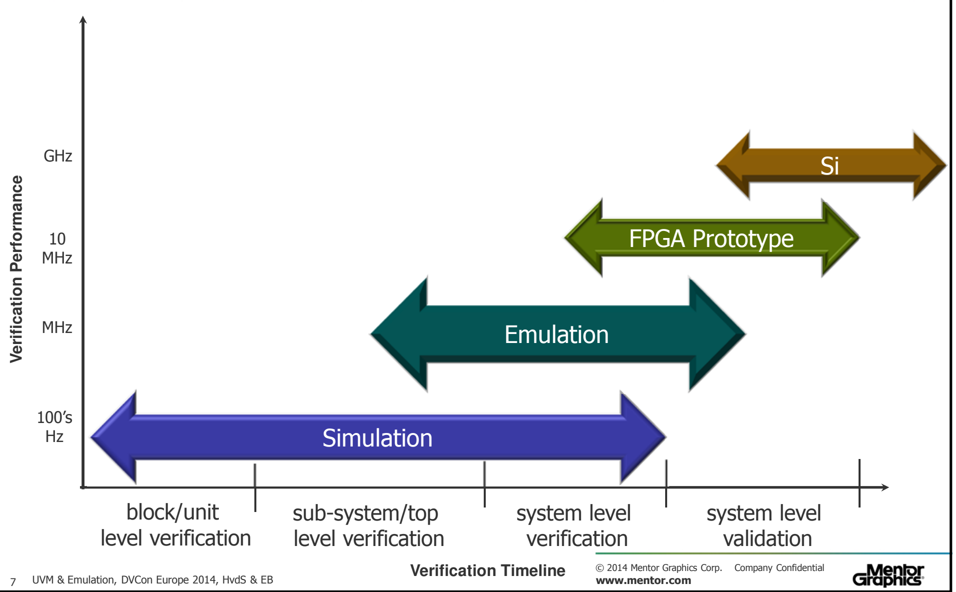
- Electronics systems companies need dramatic improvements in verification productivity
- Adoption of UVM for increased verification productivity
  - Faster to develop reusable testbenches and automated tests
- UVM-based verification reuse from block to sub-system to system level

UVM Harnesses SystemVerilog and TLM into a Reuse Methodology

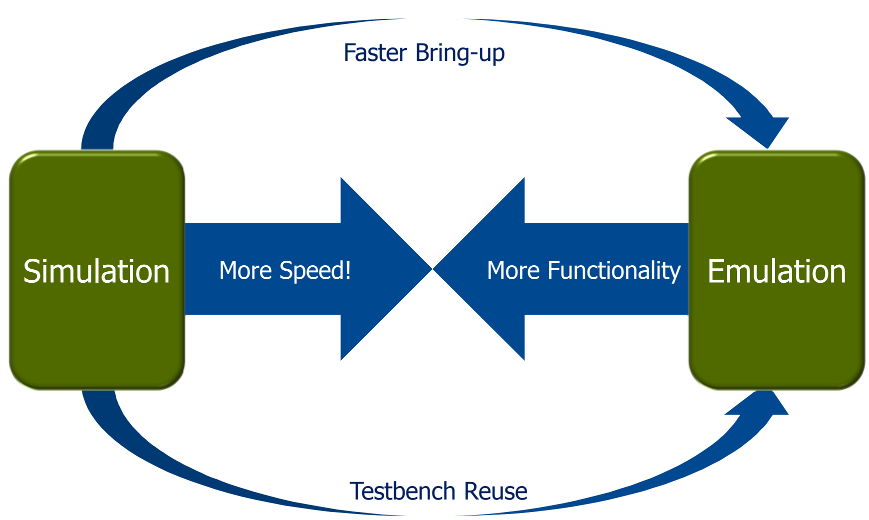
- Vertical Reuse
  - From block to system in a single project
- Horizontal Reuse
  - Reuse of modules, libraries across projects
- Platform Reuse
  - Reuse of testbenches, assertions and coverage across tools
  - **Must be able to reuse on emulation platform**



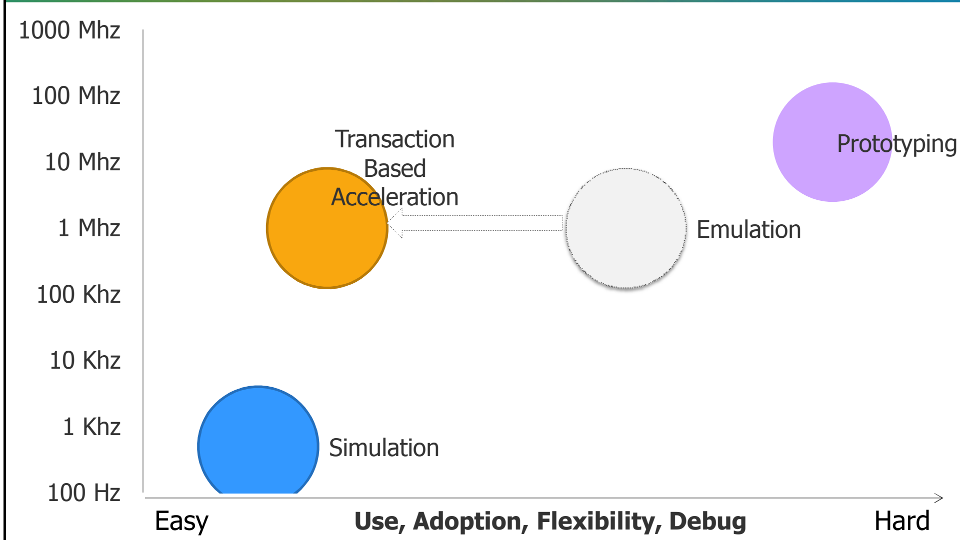
Typical Performance and Development/Verification Cycle



Users Demand the Best of Both Worlds



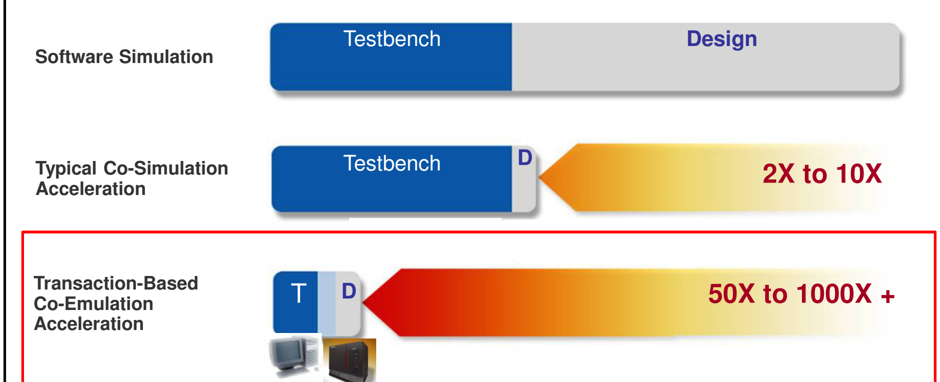
Transaction-Based Acceleration Makes Emulation Easier to Use and Adopt



9 UVM & Emulation, DVCon Europe 2014, HvdS & EB

© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com

Testbench Pivotal to Acceleration

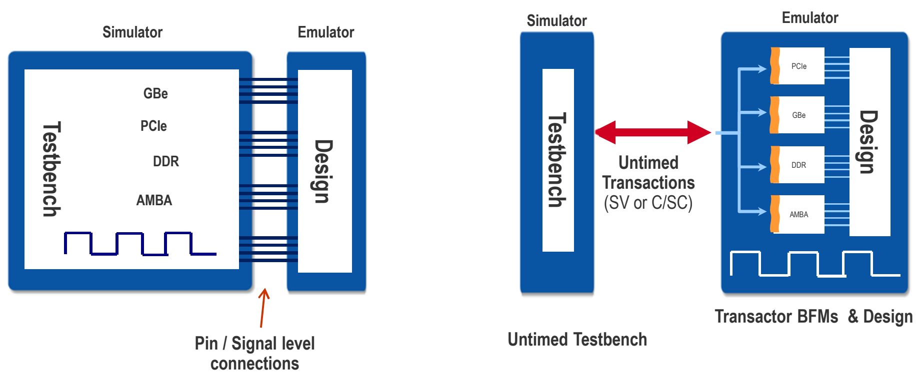


10 UVM & Emulation, DVCon Europe 2014, HvdS & EB

© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com

Key to Co-Emulation Performance

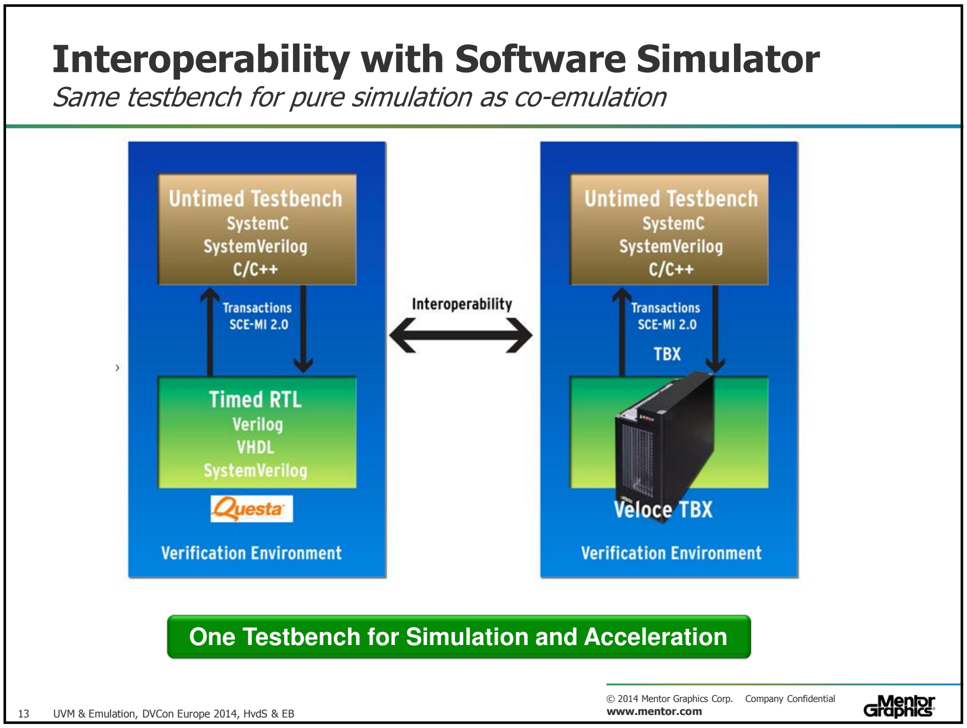
Transaction-Based Communication

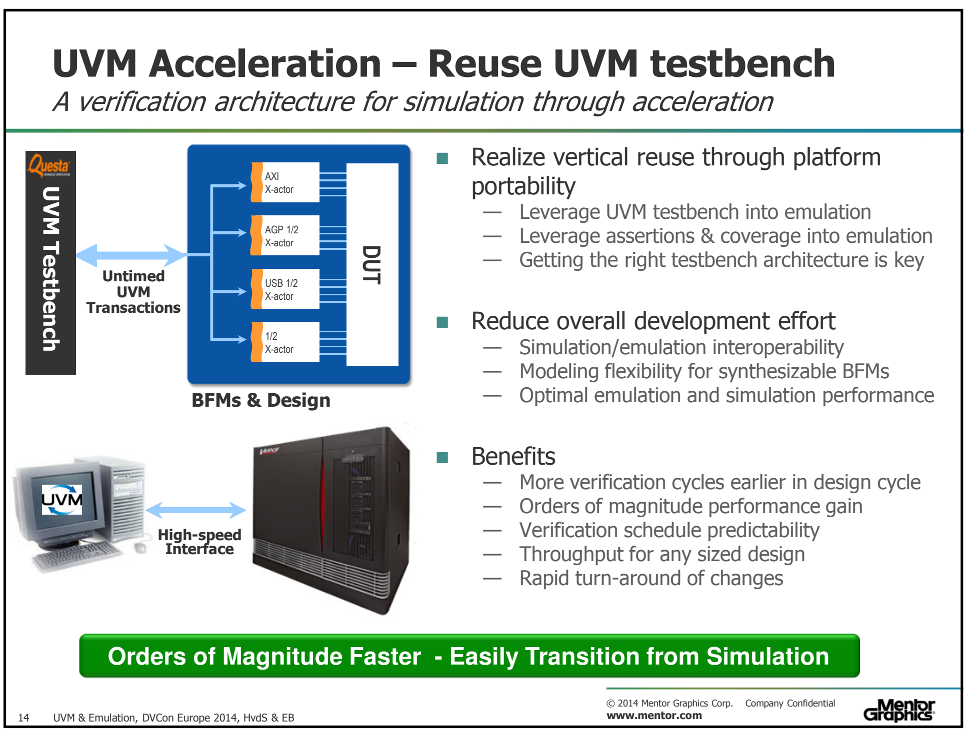


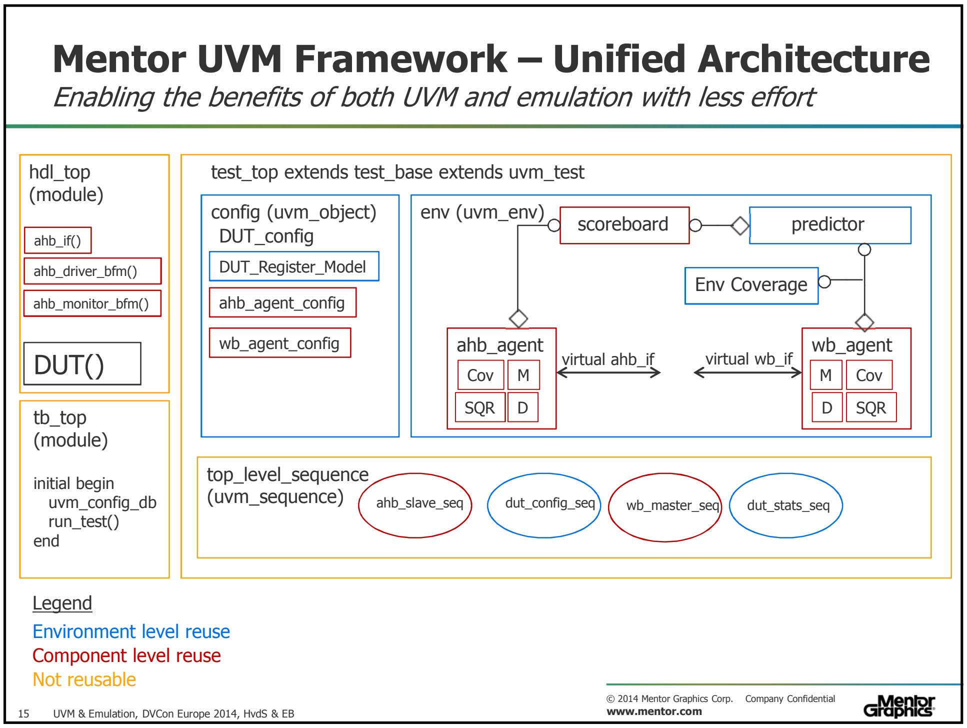
Infrequent Information-Rich Data Exchange between Emulator and Simulator is Key to Performance

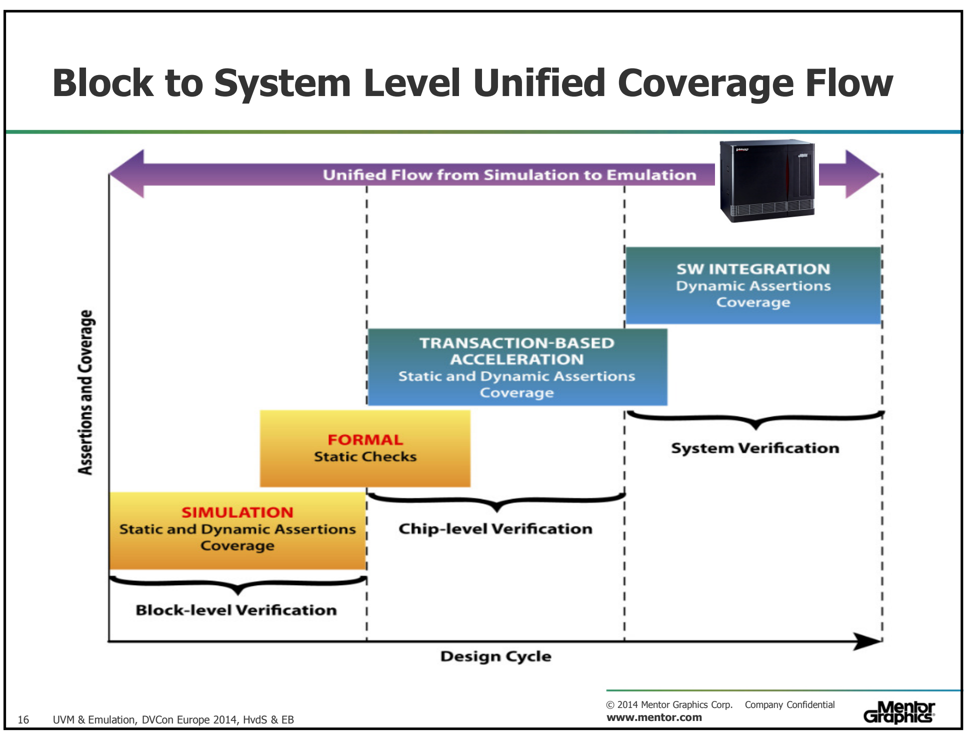
Co-Emulation 101

- All components mapped into emulator must be written in a language subset synthesizable by a logic synthesis tool
- Simulation testbench must be untimed with all operations event-driven
- Interaction between simulator and emulator must be at transaction level to prevent simulation environment from being a bottleneck
  - I.e. not co-simulation
  - Transactions passed each way must be made up of simple integral types though – e.g. cannot be classes
- Clear split must exist between testbench running in simulator and logic running in emulator
  - Separate "top level" hierarchies to allow them to be processed separately

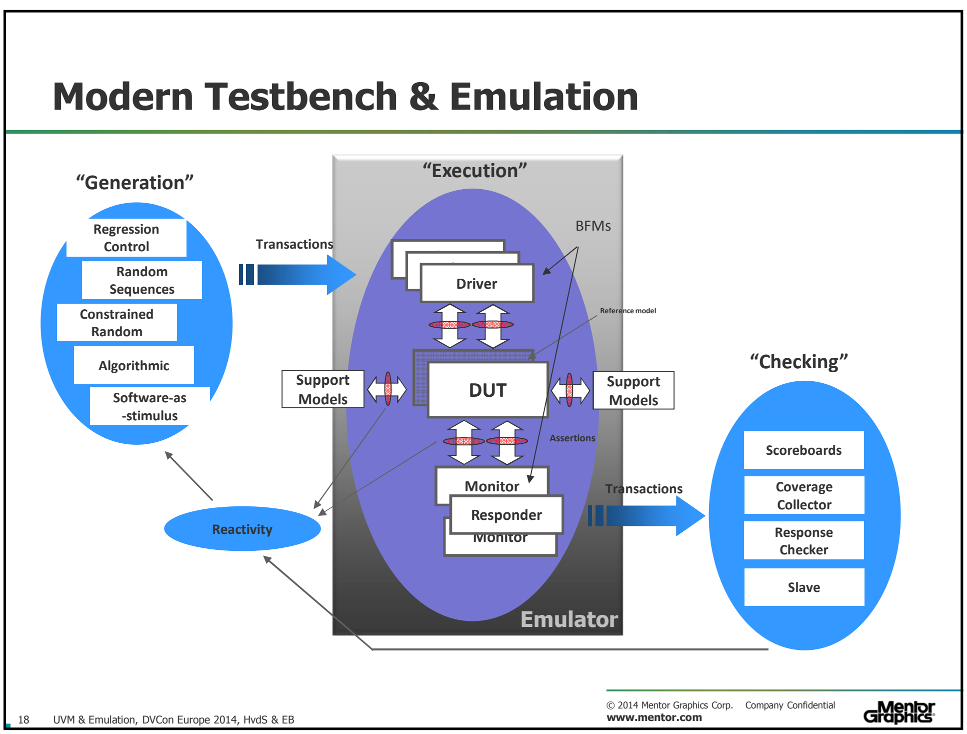

Interoperability with Software Simulator
Same testbench for pure simulation as co-emulation
Untimed Testbench
SystemC
SystemVerilog
C/C++
Transactions
SCE-MI 2.0
Timed RTL
Verilog
VHDL
SystemVerilog
Questa
Verification Environment
Interoperability
Untimed Testbench
SystemC
SystemVerilog
C/C++
Transactions
SCE-MI 2.0
TBX
Veloce TBX
Verification Environment
One Testbench for Simulation and Acceleration
13
UVM & Emulation, DVCon Europe 2014, HvdS & EB
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com


UVM Acceleration – Reuse UVM testbench
A verification architecture for simulation through acceleration
Questa
UVM Testbench
Untimed
UVM
Transactions
AXI
X-actor
AGP 1/2
X-actor
USB 1/2
X-actor
1/2
X-actor
DUT
BFMs & Design
UVM
High-speed
Interface
Realize vertical reuse through platform portability
— Leverage UVM testbench into emulation
— Leverage 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
14
UVM & Emulation, DVCon Europe 2014, HvdS & EB
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com


Mentor UVM Framework – Unified Architecture
Enabling the benefits of both UVM and emulation with less effort
hdl_top
(module)
ahb_if()
ahb_driver_bfm()
ahb_monitor_bfm()
DUT()
tb_top
(module)
initial begin
uvm_config_db
run_test()
end
test_top extends test_base extends uvm_test
config (uvm_object)
DUT_config
DUT_Register_Model
ahb_agent_config
wb_agent_config
env (uvm_env)
scoreboard
predictor
Env Coverage
ahb_agent
Cov
M
SQR
D
virtual ahb_if
virtual wb_if
wb_agent
M
Cov
D
SQR
top_level_sequence
(uvm_sequence)
ahb_slave_seq
dut_config_seq
wb_master_seq
dut_stats_seq
Legend
Environment level reuse
Component level reuse
Not reusable
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com
15 UVM & Emulation, DVCon Europe 2014, HvdS & EB


Block to System Level Unified Coverage Flow
Unified Flow from Simulation to Emulation
Assertions and Coverage
SW INTEGRATION
Dynamic Assertions
Coverage
TRANSACTION-BASED
ACCELERATION
Static and Dynamic Assertions
Coverage
FORMAL
Static Checks
SIMULATION
Static and Dynamic Assertions
Coverage
System Verification
Chip-level Verification
Block-level Verification
Design Cycle
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com
16 UVM & Emulation, DVCon Europe 2014, HvdS & EB

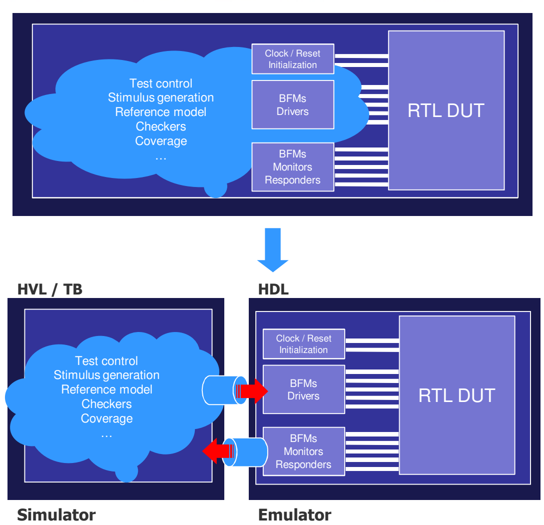

CO-EMULATION
FUNDAMENTALS
17


Modern Testbench & Emulation
"Generation"
Regression Control
Random Sequences
Constrained Random
Algorithmic
Software-as -stimulus
Transactions
"Execution"
BFMs
Driver
Reference model
Support Models
DUT
Support Models
Assertions
Monitor
Responder
Monitor
Transactions
Reactivity
Emulator
"Checking"
Scoreboards
Coverage Collector
Response Checker
Slave
18 UVM & Emulation, DVCon Europe 2014, HvdS & EB
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com

Co-Emulation: Key Concepts

- Single testbench for simulation and acceleration
  - DUT and BFM "execution" runs in simulator or emulator
  - Testbench "generation", "checking" and "coverage" runs in simulator
  - Maintains simulation-based verification features and methodologies
- Testbench partitioned into two separated domains – 2 tops
  - Timed/synthesizable DUT + BFMs, and clk/rst generation (HDL side)
  - Untimed testbench generation and analysis code (HVL/TB side)
- Transaction-based communication between two domains
  - Infrequent information-rich transactions between domains let emulator run at full speed with fewer interruptions
    - As opposed to cycle-based signal-level exchanges
  - Transactions are task/function calls
    - Reactive communication via cross-domain function/task calls
    - Buffered communication via SCE-MI 2 pipes for streaming applications
  - Domains bound together using SV virtual interfaces or SV-DPI

Partitioning of Testbench



Main considerations
- Testbench architecture
- HVL-side modeling
- HDL-side modeling
- HVL-HDL communication
- Performance

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

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 thread blocks
  - Waiting for responses/notifications from HDL side
  - Time advance is monitored by a transactor (an HDL clock counter)

Inside the Emulator

- It's actual hardware
- Must get synthesized into gates
  - Static, i.e. functionality cannot be added or changed at runtime
- SystemVerilog classes are not synthesizable
- Most advanced SystemVerilog testbench constructs are not synthesizable
  - Classes, processes, program blocks
  - Clocking blocks, fork-join
  - Dynamically-sized arrays (dynamic, associative, or queues)
- Processes actually run concurrently in the hardware
- Memories have limited number of ports
- Runs many times faster (MHz vs kHz speeds)

Effective HDL Modeling

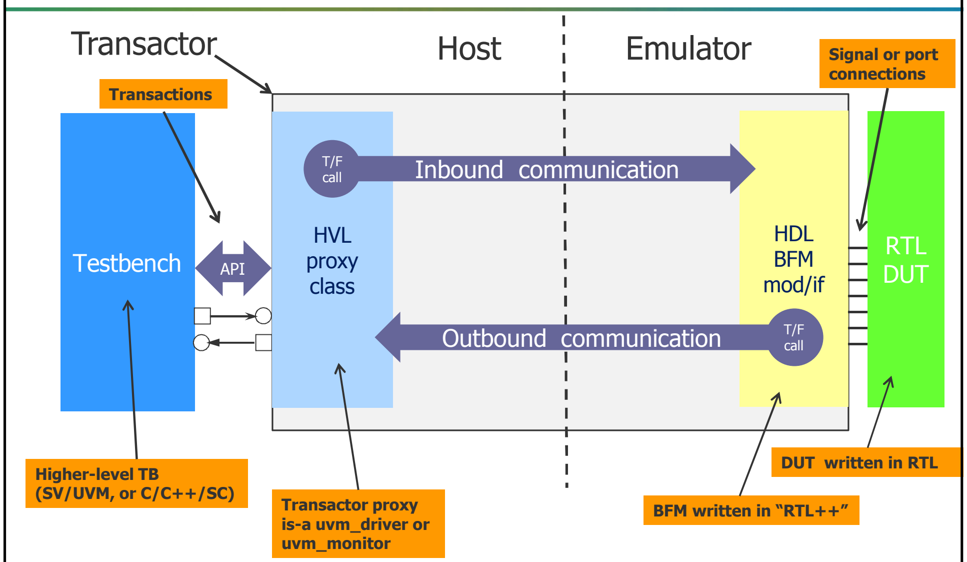
- Development of synthesizable HDL BFMs facilitated thru familiar modeling with behavioral language constructs
  - "RTL++" (i.e. XRTL)
    - Implicit FSMs, initial code blocks, named events/waits, behavioral clock & reset, force/release, system tasks, memory arrays, (virtual) interfaces, assertions, coverage
  - SCE-MI 2 based reactive function calls and streaming pipes
- Fully standards-based modeling with IEEE P1800 SystemVerilog and Accelera SCEMI 2.x
  - BFMs, checkers and monitors run unmodified in any standard compliant EDA tool
- Synthesizable HDL models must run at full emulator clock rate for high performance

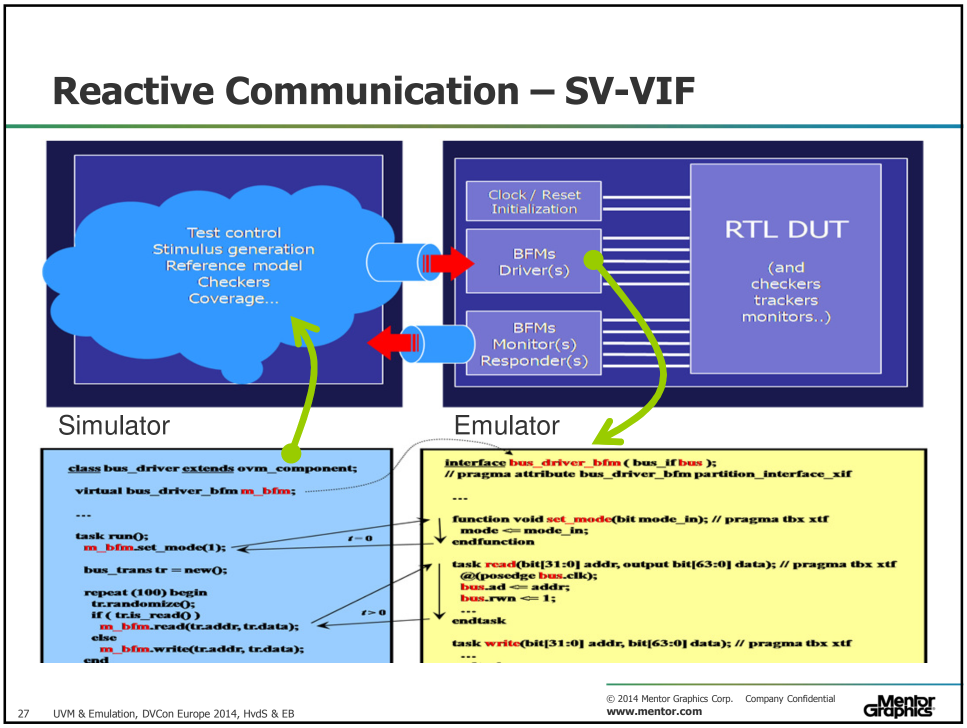
Transaction-Based Communication

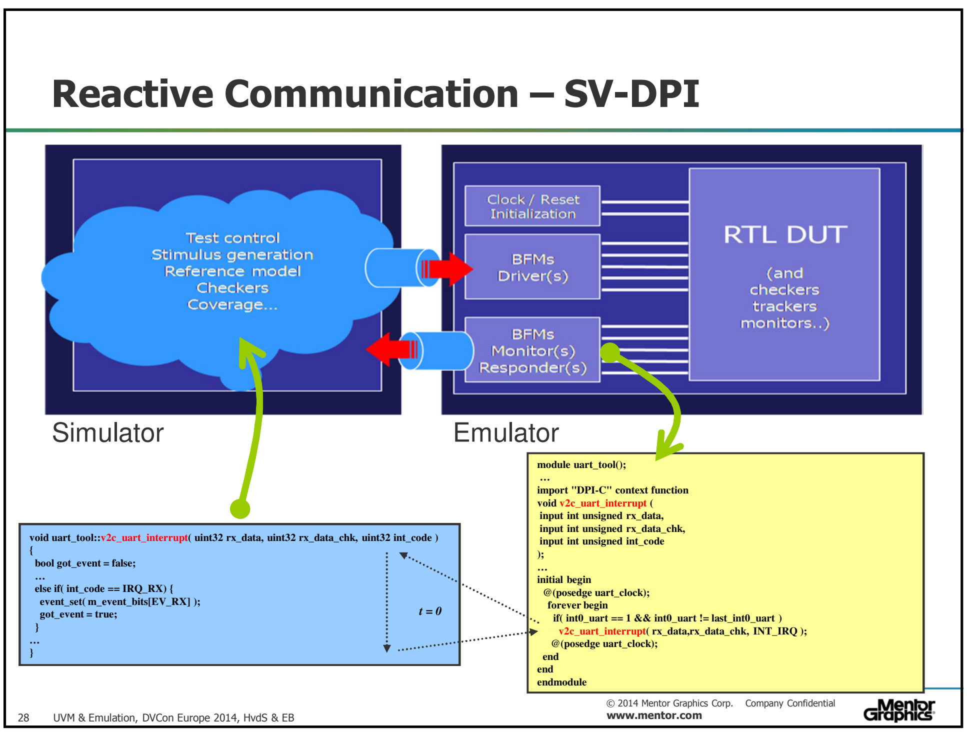
- Transaction level communication between HVL and HDL side is by function/task calls that represent transactions
  - Like traditional BFM-style
- BFM functions/tasks provided by SV interface or module on HDL side and invoked from HVL side
  - No direct access to HDL-side signals/pins from HVL side
    - Only within HDL side
  - No direct access to HVL-side data variables from HDL side
  - No shared variables across the HVL-HDL boundary
  - Argument types must be synthesizable data types
- Note: SV interface for BFM encapsulation enables familiar access from SV HVL side using virtual interface
  - Do not merge BFM with SV pin interface for reuse purposes

Transactors – Co-Emulation Building Blocks

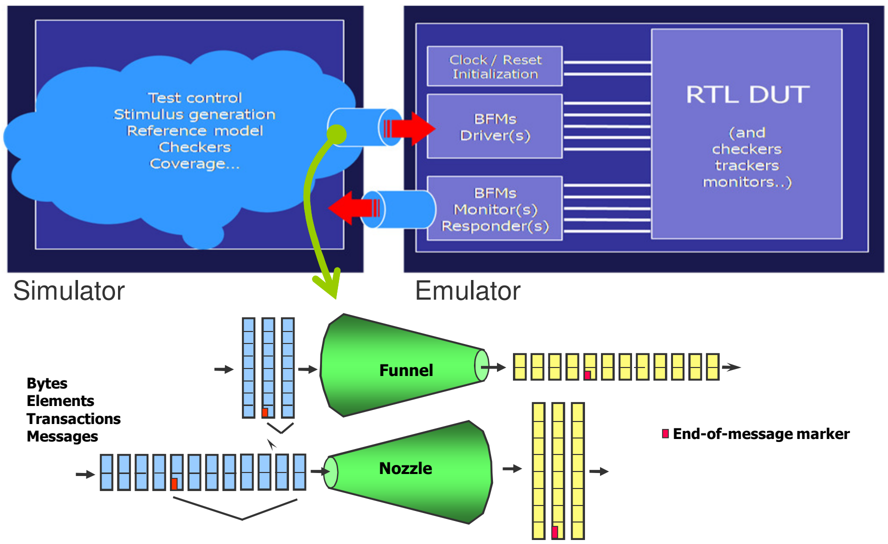
HDL BFM + HVL Proxy + HVL-HDL Channel




Reactive Communication – SV-VIF
Test control
Stimulus generation
Reference model
Checkers
Coverage…
Clock / Reset
Initialization
BFMs
Driver(s)
BFMs
Monitor(s)
Responder(s)
RTL DUT
(and
checkers
trackers
monitors..)
Simulator
Emulator
class bus_driver extends ovm_component;
virtual bus_driver_bfm m_bfm;
...
task run();
m_bfm.set_mode(1);
bus_trans tr = new();
repeat (100) begin
tr.randomize();
if (tr.is_read())
m_bfm.read(tr.addr, tr.data);
else
m_bfm.write(tr.addr, tr.data);
end
t = 0
t > 0
interface bus_driver_bfm (bus_if bus);
// pragma attribute bus_driver_bfm partition_interface_xif
...
function void set_mode(bit mode_in); // pragma tbx xtf
mode <= mode_in;
endfunction
task read(bit[31:0] addr, output bit[63:0] data); // pragma tbx xtf
@(posedge bus.clk);
bus.ad <= addr;
bus.rwn <= 1;
...
endtask
task write(bit[31:0] addr, bit[63:0] data); // pragma tbx xtf
...
27 UVM & Emulation, DVCon Europe 2014, HvdS & EB
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com
Mentor Graphics


Reactive Communication – SV-DPI
Test control
Stimulus generation
Reference model
Checkers
Coverage…
Clock / Reset
Initialization
BFMs
Driver(s)
BFMs
Monitor(s)
Responder(s)
RTL DUT
(and
checkers
trackers
monitors..)
Simulator
Emulator
void uart_tool::v2c_uart_interrupt(uint32 rx_data, uint32 rx_data_chk, uint32 int_code)
{
bool got_event = false;
…
else if(int_code == IRQ_RX) {
event_set(m_event_bits[EV_RX]);
got_event = true;
}
…
}
t = 0
module uart_tool();
…
import "DPI-C" context function
void v2c_uart_interrupt (
input int unsigned rx_data,
input int unsigned rx_data_chk,
input int unsigned int_code
);
…
initial begin
@(posedge uart_clock);
forever begin
if(int0_uart == 1 && int0_uart != last_int0_uart)
v2c_uart_interrupt(rx_data,rx_data_chk, INT_IRQ);
@(posedge uart_clock);
end
end
endmodule
28 UVM & Emulation, DVCon Europe 2014, HvdS & EB
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com
Mentor Graphics

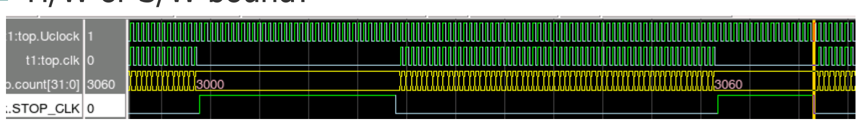
Streaming Communication – SCEMI Pipes



29 UVM & Emulation, DVCon Europe 2014, HvdS & EB

Co-Emulation Performance

- Total run-time = $t_{[HDL]} + t_{[HVL]} + t_{[HVL\text{-}HDL]}$
- 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
- Want H/W bound – "healthy" throughput
  - Design clocks active high % of time, i.e. low testbench and communication overhead

$$t_{[HDL]} / t_{[total]} >> (t_{[HvL]} + t_{[HVL\text{-}HDL]}) / t_{[total]}$$

30 UVM & Emulation, DVCon Europe 2014, HvdS & EB

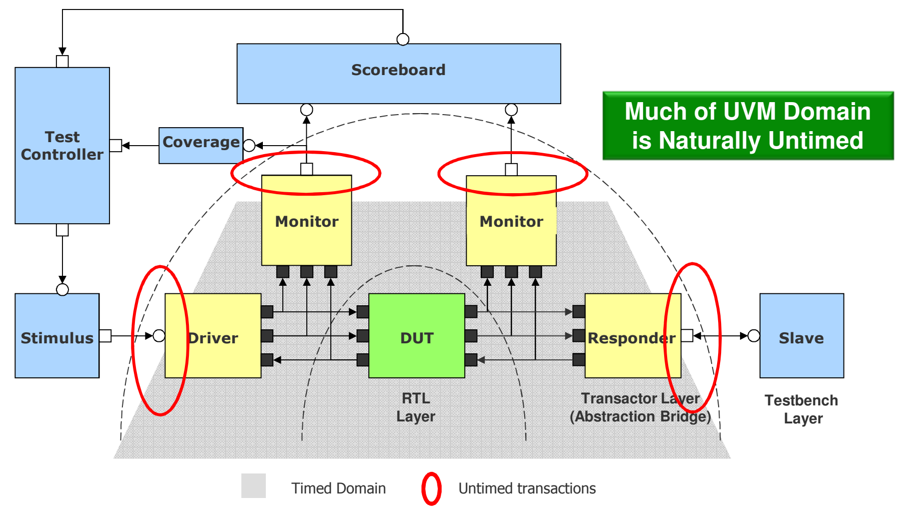
Optimizing Performance

- Reduce communication overhead by optimizing transaction utilization
  - Increasing transaction sizes – larger transactions stay inside DUT longer
  - Using SCE-MI pipe-based data shaping
  - Raising abstraction to meta-transactions
  - Maximizing concurrency between simulator and emulator
  - Minimizing fine-grain scoreboarding and memory access frequencies
- Reduce testbench overhead by optimizing simulation performance
  - Heeding file I/O, constraint solving, messaging & macro usage (UVM)
  - Compiling with optimization switches
- Enhance H/W execution by optimizing emulation frequency
  - Improving critical paths
  - Optimizing emulator clock utilization
    - Aligning design clocks (CFR), using inactive edge optimization
  - Maximizing parallelism in BFMs
- Detailed analysis through profiling, linting, etc.

31 UVM & Emulation, DVCon Europe 2014, HvdS & EB

UVM & CO-EMULATION

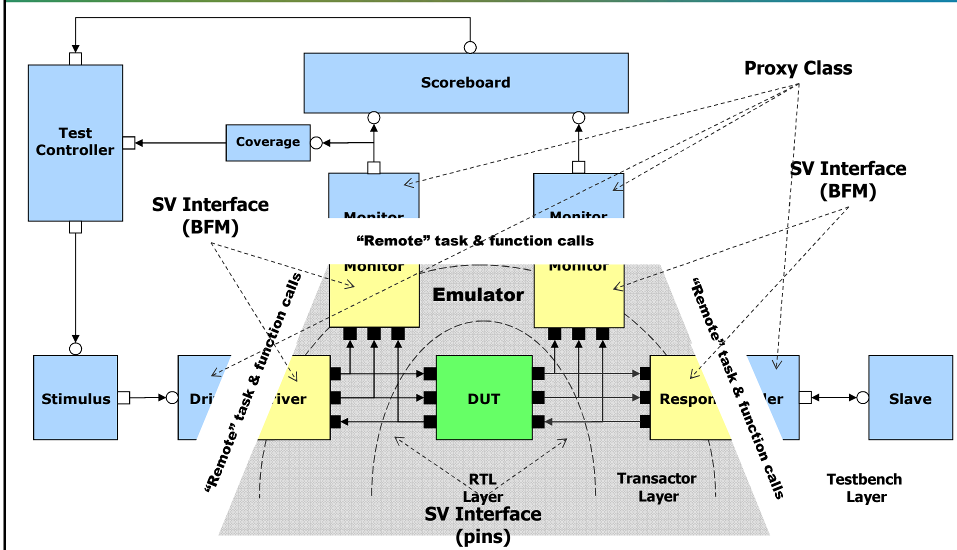
UVM Layered Testbench



UVM Orthogonal to Co-Emulation

- Abstraction and reuse principles of UVM should and do apply independent of execution platform
  - UVM already advocates absence of timing control and hierarchical accesses (XMRs) for upper "testbench layer" components
    - No clock and (especially) unit delays, XMRs
  - UVM already advocates delegation of timing control to lower "transactor layer" components
    - UVM agents, drivers, monitors, responders, masters, slaves
- UVM layering facilitates adherence to co-emulation requirements
  - UVM usage can continue largely per established modeling best practices
    - Some notable advanced considerations discussed later
  - Some of the recommendations merely become mandated
    - "You shall [not]" instead of "You should [not]"
- Execution platform dependence should be a private transactor matter
  - Front-end untimed transaction-level transactor API need not change
  - Splitting UVM drivers and monitors into proxy + channel + BFM is a localized affair and hence a manageable and sensible added practice

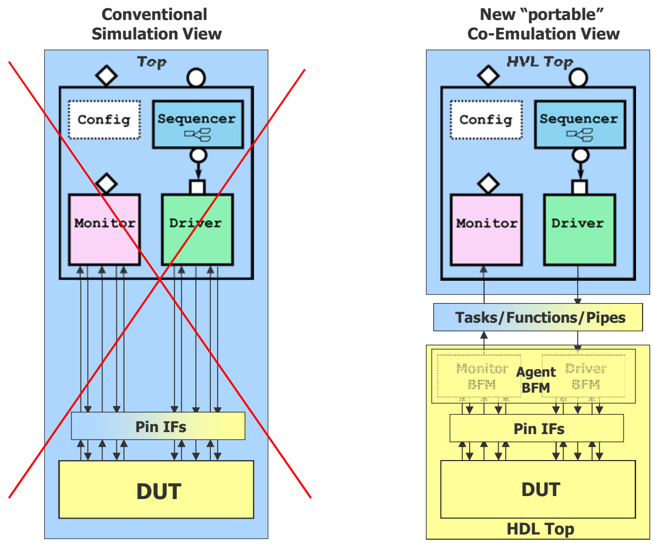
Unified UVM Simulation/Emulation Testbench



"Emulatable" UVM Transactors

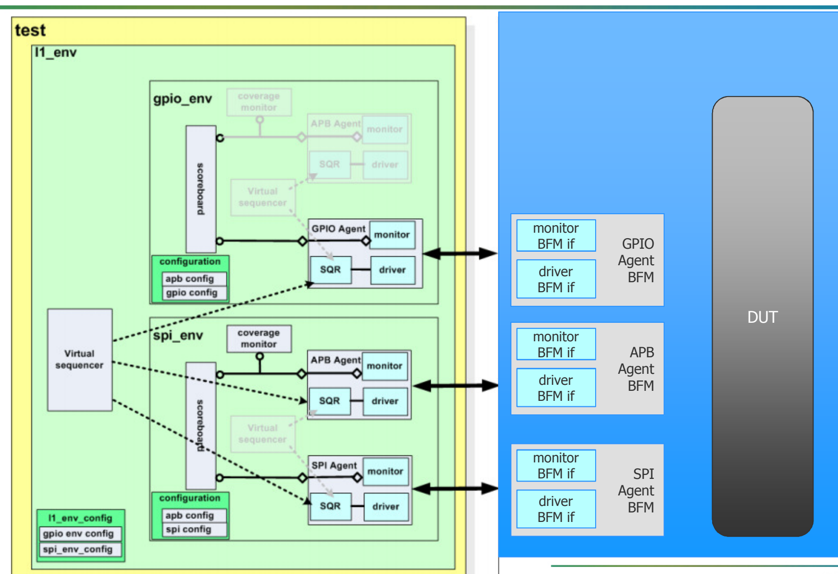
- HDL BFM is an SV interface
  - Avoid non-synthesizable modeling constructs
- UVM driver/monitor is the class proxy for the BFM
- UVM proxy can access internal tasks and functions (only) of the BFM via virtual interface – inbound
  - To drive and sample DUT signals
  - To trigger HDL FSM initiation
  - To set HDL configuration parameters
- HDL BFM can access functions (only) of the UVM proxy via "backpointer" class object handle – outbound
  - To provide control and data notifications
- Standard UVM block-to-top reuse continues to apply
  - UVM agent and environment encapsulations are preserved

"Emulatable" UVM Agents



37 UVM & Emulation, DVCon Europe 2014, HvdS & EB

Vertical Reuse in Dual Top UVM Framework

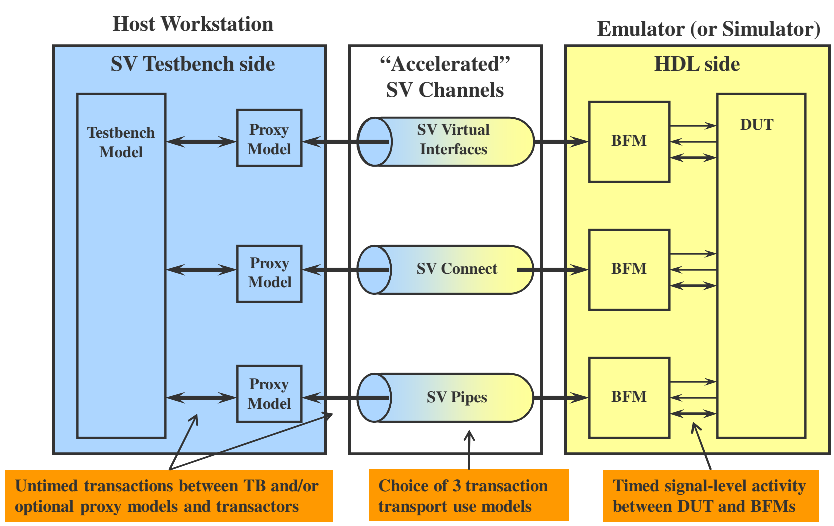


38 UVM & Emulation, DVCon Europe 2014, HvdS & EB

UVM – HDL Interface Modeling

- Communication between untimed UVM and synthesizable HDL partitions must be transaction-based
  - Not cycle-based
- Flexible transaction transport interfaces
  - Reactive: "Remote" function calls between proxy and BFM as discussed for instantaneous configuration, FSM initiation, control, and status
  - Streaming (non-reactive): SCE-MI 2 transaction pipes for highly optimized transfers of large amounts of one-way transaction data
- Fully standards-based HVL-HDL interface modeling
  - IEEE SystemVerilog along with Accellera SCE-MI 2 function model and associated performance benefits

UVM – HDL Transaction Transport Use Models



Example UVM Driver

```
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();
  @(posedge pins.rst_n);
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)

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

- UVM offers virtual pack/unpack methods, though no standard way for implementing packing/unpacking transactions
- Recommend user-defined object conversion methods targeted for optimal HVL-HDL communication modeling and performance

Shared HVL-HDL package

Optimization: byte vs. int

```
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
```

```
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
```


Example UVM Monitor

- 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 not require emulator clock stoppage

Preferred UVM Monitor

```
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 driver 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_daeom();
  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

45 UVM & Emulation, DVCon Europe 2014, HvdS & EB

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

46 UVM & Emulation, DVCon Europe 2014, HvdS & EB

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(...)
  );
```

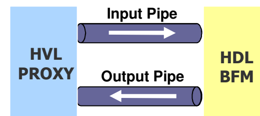
Retrieving the virtual interface handle from uvm_config_db into the configuration object

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

- Accelera 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



SCEMI 2 Transaction Pipes – Basic Usage

```
import scemi_pipes_pkg::*;

class some_driver
   extends uvm_driver #(some_seq_item);

scemi_static_input_pipe #(16,1,400) req_pipe;

...

function connect_phase(uvm_phase phase);
  ...
  req_pipe = new({cfg.hdl_bfm_path, ".in_pipe"});
endfunction

virtual task run_phase(uvm_phase phase);
  fork bfm.run(); join_none
  forever begin
    seq_item_port.get_next_item(req);
    put(req);
    seq_item_port.item_done();
   end
endtask

virtual protected task put(REQ req);
  ...
  req.to_struct(req_s);
  req_pipe.send_bits(.num_elements(1),req_s,.eom(1));
endtask

endcla
```

Input pipe handle in driver proxy on testbench side bound to SCEMI input pipe in HDL-side BFM with large depth

```
scemi_input_pipe
        #(.BYTES_PER_ELEMENT(<1>),
        .PAYLOAD_MAX_ELEMENTS(<1>),
        .BUFFER_MAX_ELEMENTS(< 32>),
in_pipe (<user_clk>);
```

```
interface some_driver_bfm(some_if pins);

scemi_input_pipe
  #(16,1,400) in_pipe(pins.clk);

...

task run();
  bit[127:0] data;
  int ne_valid;
  bit eom;
  forever begin
    in_pipe.receive(1,ne_valid,data,eom);
    assert(ne_valid == 1);
    // Process data
    ...
    if (eom == 1)
      ...
  end
endtask

endinterface
```

Each call reads 1 element of 16 bytes; 16 bytes can be processed in one cycle

eom evaluates to true for last element in a message

Can also use `scemi_dynamic_input_pipe::send_bytes` with open array `ref byte unsigned data[]` instead of fixed size vector `bit [STATIC_PAYLOAD_MAX_BYTES*8-1:0] data`

SCEMI 2 Transaction Pipes – Basic Usage

```
import scemi_pipes_pkg::*;

class some_monitor extends uvm_monitor

scemi_static_output_pipe #(16,1,400) rsp_pipe;

...

function connect_phase(uvm_phase phase);
  ...
  rsp_pipe = new({cfg.hdl_bfm_path, ".out_pipe"});
endfunction

virtual task run_phase(uvm_phase phase);
  fork bfm.run(); join_none
  forever begin
    get(rsp);
    ap.write(rsp)
   end
endtask

virtual protected task get(output RSP rsp);
  ...
  rsp_pipe.receive_bits(1,ne_valid,rsp_s,eom);
  assert(ne_valid == 1);
  assert(eom == 1);
  rsp.from_struct(rsp_s);
endtask

endcla
```

Output pipe handle in monitor proxy on testbench side bound by hierarchical path to SCEMI output pipe in HDL-side BFM with large depth

```
scemi_output_pipe
     #(.BYTES_PER_ELEMENT(<1>),
     .PAYLOAD_MAX_ELEMENTS(<1>),
     .BUFFER_MAX_ELEMENTS(< 32>),
out_pipe (<user_clk>);
```

```
interface some_monitor_bfm(some_if pins);

scemi_output_pipe
  #(16,1,400) out_pipe(pins.clk);

...

task run();
  bit[127:0] data;
  forever begin
    // Compute response data
    ...
    out_pipe.send(.num_elements(1),
                  data,
                  .eom(1));
    num_responses++;
    if (num_responses == 10)
      out_pipe.flush();
end
endtask

ndinterface
```

Each call writes 1 element of 16 bytes; 16 bytes can be processed in one cycle

eom set to true for last element in a message

Can also use `scemi_dynamic_output_pipe::receive_bytes` with open array `ref byte unsigned data[]` instead of fixed size vector `bit [STATIC_PAYLOAD_MAX_BYTES*8-1:0] data`

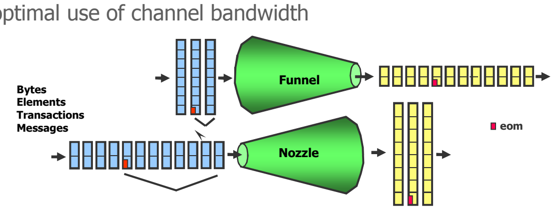
SCEMI 2 Transaction Pipes – Additional Usage

- Pipes are particularly useful in case of large or variable-length transactions
  - How else to transfer a dynamic payload modeled as SV queue or dynamic array?
  - Pipes break payload into smaller fixed-size elements for transfer **Open Array**
  - BFMs consume/produce payload a specified number of elements at a time

```
byte open_array [];
pipe.send_bytes(.num_elements(open_array.size()/16),
                .data(open_array),
                .eom(1));
```

wide array send in a single call using a "dynamic" pipe

- Pipes can have a different width at one end than the other for data-shaping
  - Wide send end - narrow receive end
  - Narrow send end - wide receive end
  - Makes optimal use of channel bandwidth



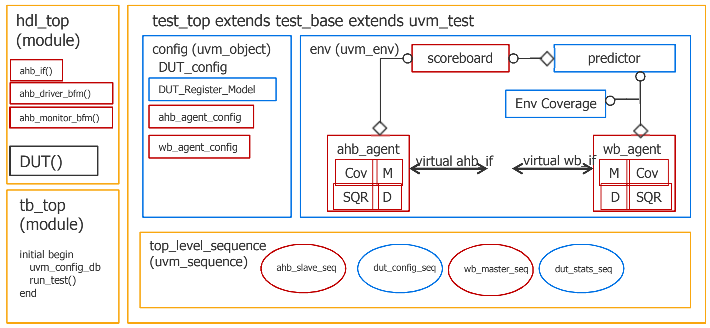
UVM & Emulation Flow Summary

- 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 cover groups too
  - Basic data-oriented functional coverage inside BFMs to complement normal UVM domain coverage

Mentor UVM Framework – Directory Structure




```
/verification_ip
    /uvmf_base_pkg
    /interface_packages
        /ahb_pkg
        /wb_pkg
        /spi_pkg
    /project_packages
        /ahb2wb_env_pkg
        /wb2spi_env_pkg

/project_benches
    /ahb2wb
        /doc
        /sim
        /veloce
        /tb
            /testbench
            /sequences
            /tests
```

Legend
Environment level reuse
Component level reuse
Not reusable

55 UVM & Emulation, DVCon Europe 2014, HvdS & EB

Mentor UVM Framework – Interface Packages

```
interface_packages
|-- ahb_pkg.sv
|-- ahb_pkg_hdl.sv
|-- src
    |-- ahb_configuration.svh
    |-- ahb_driver.svh
    |-- ahb_driver_bfm.sv
    |-- ahb_if.sv
    |-- ahb_monitor.svh
    |-- ahb_monitor_bfm.sv
    |-- ahb_sequence_lib.svh
    |-- ahb_transaction.svh
    |-- ahb_transaction_coverage.svh
    |-- ahb_typedefs.svh
    |-- ahb_typedefs_hdl.svh
    |-- reg2ahb_adapter.svh
```

***.svh files are generally HVL files implementing classes, or shared HVL-HDL files implementing common parameters or typedefs.**

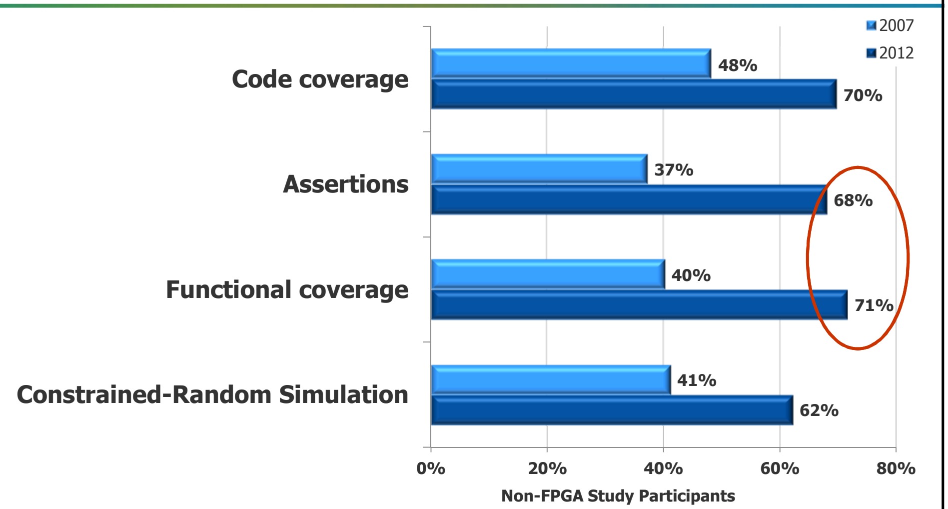
These are included inside packages (hence the .svh extension).

***_if.sv, *_bfm.sv and *_hdl.sv(h) files are HDL files respectively implementing pin interfaces, BFM interfaces, and synthesizable typedefs with packages etc. (shown in red)**

56 UVM & Emulation, DVCon Europe 2014, HvdS & EB

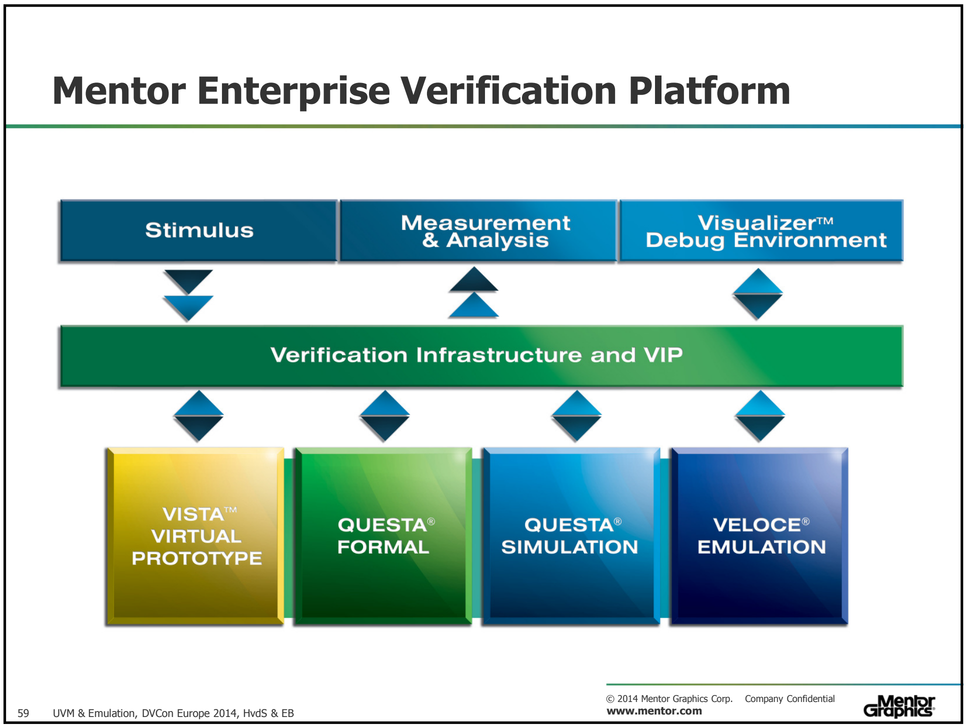
UNIFIED ANALYSIS & DEBUG

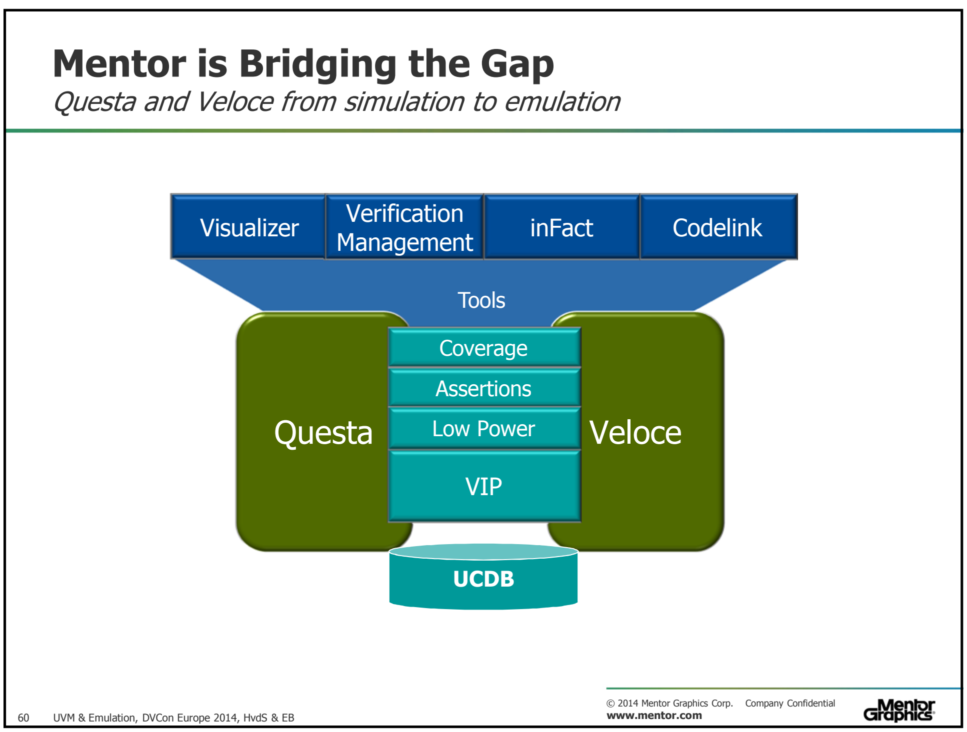
Use of Advanced Verification Techniques



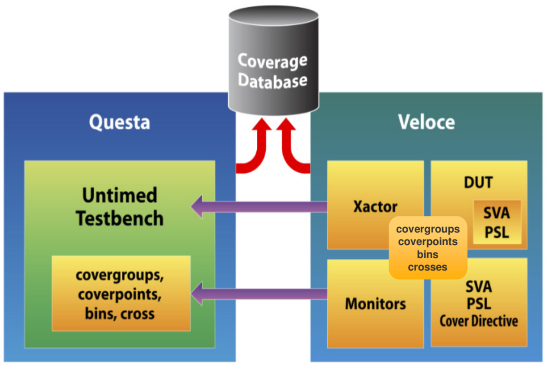
Source: Wilson Research Group and Mentor Graphics, 2012 Functional Verification Study

58 UVM & Emulation, DVCon Europe 2014, HvdS & EB


Mentor Enterprise Verification Platform
Stimulus
Measurement & Analysis
Visualizer™ Debug Environment
Verification Infrastructure and VIP
VISTA™ VIRTUAL PROTOTYPE
QUESTA® FORMAL
QUESTA® SIMULATION
VELOCE® EMULATION
59 UVM & Emulation, DVCon Europe 2014, HvdS & EB
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com


Mentor is Bridging the Gap
Questa and Veloce from simulation to emulation
Visualizer
Verification Management
inFact
Codelink
Tools
Questa
Coverage
Assertions
Low Power
VIP
Veloce
UCDB
60 UVM & Emulation, DVCon Europe 2014, HvdS & EB
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com

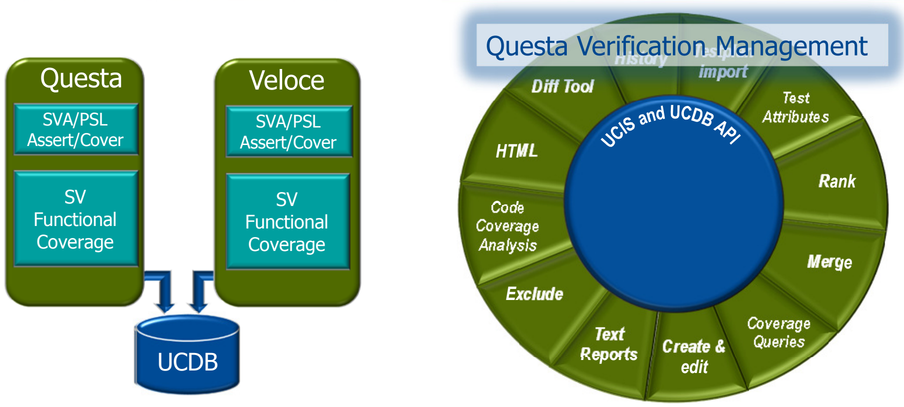
Functional Coverage and Assertions



- Assertions
  - Broad language and constructs
  - SVA, PSL, OVL
- Functional coverage
  - Cover groups, points, bins, and crosses
  - Cover properties
- Single UCDB file for testbench, transactors and DUT coverage

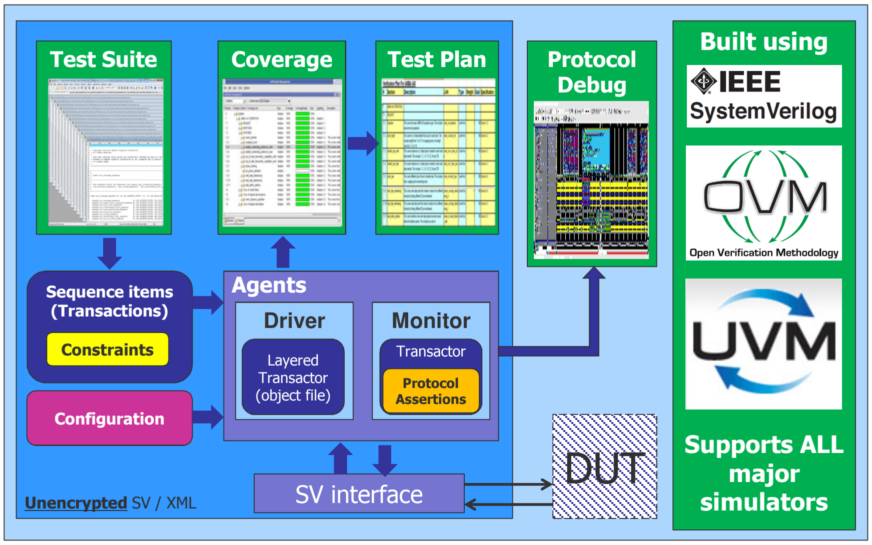
Using Assertions/Coverage with Questa and Veloce

Unified UCDB gives one place to store and analyze metrics



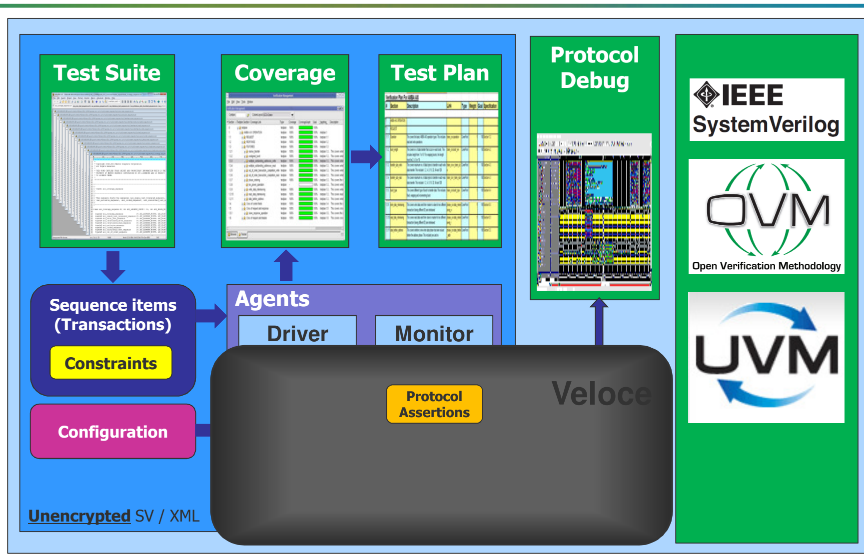
- Same assertions and coverage – results go into Mentor UCDB
  - Capacity-wise coverage in Veloce – instruments only covergroups not at 100%
- Questa verification management – 100s of man years in analysis tools

Questa Verification IP



63 UVM & Emulation, DVCon Europe 2014, HvdS & EB
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com

Veloce Accelerated Verification IP

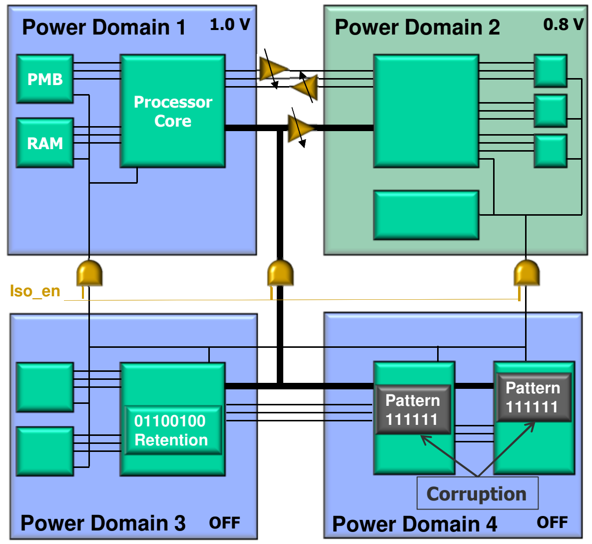


64 UVM & Emulation, DVCon Europe 2014, HvdS & EB
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com

Low Power Verification

Unified from block to system

- Leakage power reduction is key in low-power design
- Chief aspects of power management
  - Power shut-off
  - Isolation
  - Retention
  - Corruption
  - Multiple voltages
  - Level shifters
- Unified Power Format (UPF)
  - Power management defined independent of design
  - UPF 2.1

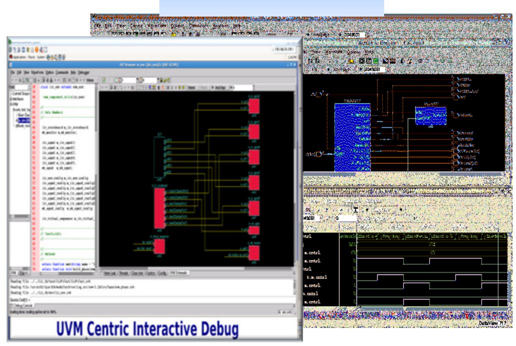


Unified Debug for Questa and Veloce

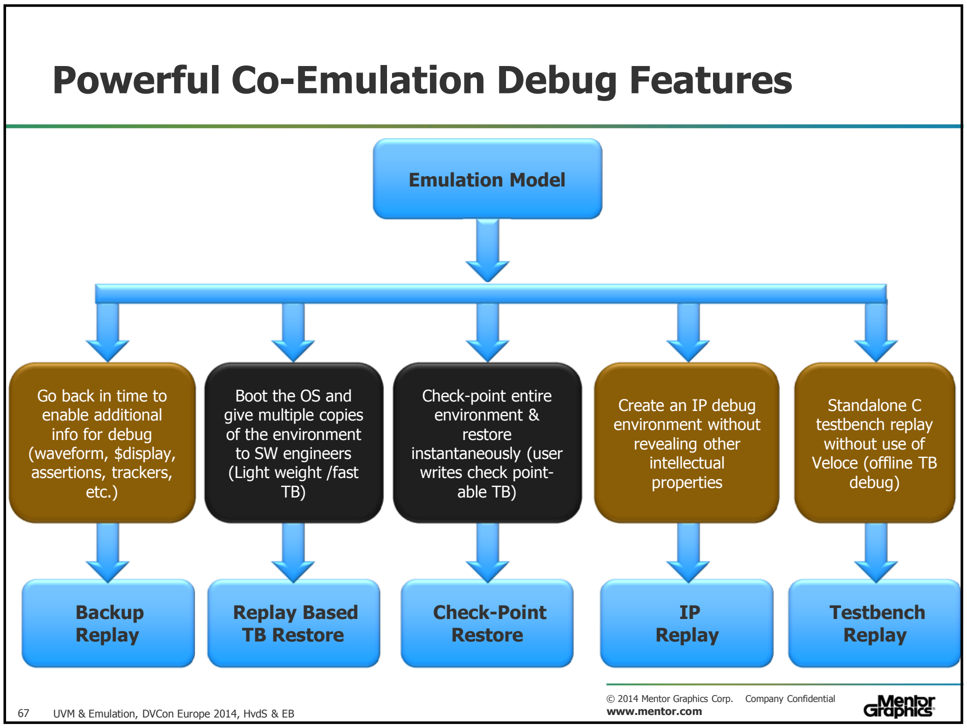
Maximize performance and still have ease of use

Visualizer

Questa



- Same debugger in simulation or emulation
- Unified debugger for simulation acceleration – Questa running testbench and Veloce running DUT
- Emulator savvy, easy, powerful and FAST


Powerful Co-Emulation Debug Features
Emulation Model
Go back in time to enable additional info for debug (waveform, $display, assertions, trackers, etc.)
Boot the OS and give multiple copies of the environment to SW engineers (Light weight /fast TB)
Check-point entire environment & restore instantaneously (user writes check point-able TB)
Create an IP debug environment without revealing other intellectual properties
Standalone C testbench replay without use of Veloce (offline TB debug)
Backup Replay
Replay Based TB Restore
Check-Point Restore
IP Replay
Testbench Replay
67 UVM & Emulation, DVCon Europe 2014, HvdS & EB
© 2014 Mentor Graphics Corp. Company Confidential
www.mentor.com


RESULTS & WRAP-UP

Co-Emulation Performance for Network Chip

Pure simulation time vs. Veloce runtime			
Number of Packets	Simulation ime (sec)	Veloce Wall Clock Time (sec)	Speed Up (X factor)
1	280	5.7	49
5	1473	16.7	88
10	2572	26.4	97
100	25720*	231.6	111*
1000	257200*	2321.2	111*

* Extrapolated

		Comments
Compiled frequency	1.2 Mhz	Frequency for fastest user clock
Capacity	5 Crystals	16 Crystals per AVB

Accelerating Multimedia SoC Sub-System of Wireless Design

Simulation Phase	Original Configuration native Questa			Veloce Configuration TBX/Veloce		Speedup Over Original
	(hours)	(seconds)	(CPS)	(seconds)	(CPS)	(X factor)
Phase I	2.93	10542	0	24	0	439
Phase II	**1.21**	**4352**	**742**	**5**	**522795**	**870**
Phase III	0.96	3472	0	17	0	204
Overall	**5.10**	**18366**	**176**	**46**	**56839**	**399**

5 hour simulation

46 second emulation

400X speedup

One testbench for simulation and acceleration

Further Results/Examples

- High Performance Networking
  - Full UVM/VIP/etc

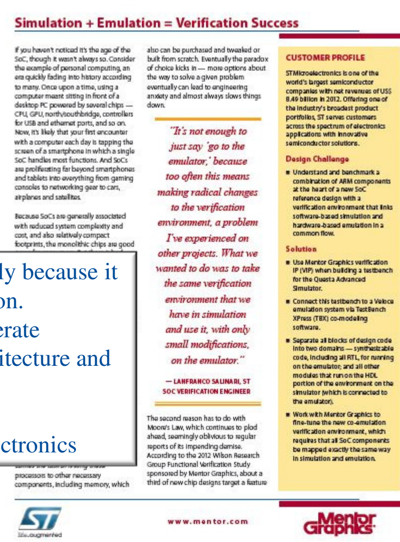
Design 1 (suite of tests)	171X - 268X
Design 2 (suite of tests)	250x – 317X

- Multi-CPU subsystem
  - ReUsed in multiple designs
  - 51X

71 UVM & Emulation, DVCon Europe 2014, HvdS & EB

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



Simulation + Emulation = Verification Success

If you haven't noticed it's the age of the SoC, though it wasn't always so. Consider the example of personal computing, an era quickly fading into history according to many. Once upon a time, using a computer meant sitting in front of a desktop PC powered by several chips — CPU, GPU, north/southbridge, controllers for USB and ethernet ports, and so on. Now, it's likely that your first encounter with a computer each day is tapping the screen of a smartphone in which a single SoC handles most functions. And SoCs are proliferating far beyond smartphones and tablets into everything from gaming consoles to networking gear to cars, airplanes and satellites.

Because SoCs are generally associated with reduced system complexity and cost, and also relatively compact footprints, the monolithic chips are good

processors to other necessary components, including memory, which also can be purchased and tweaked or built from scratch. Eventually the paradox of choice kicks in — more options about the way to solve a given problem eventually can lead to engineering anxiety and almost always slows things down.

"It's not enough to just say 'go to the emulator,' because too often this means making radical changes to the verification environment, a problem I've experienced on other projects. What we wanted to do was to take the same verification environment that we have in simulation and use it, with only small modifications, on the emulator."

— LANFRANCO SALINARI, ST SOC VERIFICATION ENGINEER

The second reason has to do with Moore's Law, which continues to plod ahead, seemingly oblivious to regular reports of its impending demise. According to the 2012 Wilson Research Group Functional Verification Study sponsored by Mentor Graphics, about a third of new chip designs target a feature

CUSTOMER PROFILE

STMicroelectronics is one of the world's largest semiconductor companies with net revenues of US$ 8.49 billion in 2012. Offering one of the industry's broadest product portfolios, ST serves customers across the spectrum of electronics applications with innovative semiconductor solutions.

Design Challenge

- Understand and benchmark a combination of ARM components at the heart of a new SoC reference design with a verification environment that links software-based simulation and hardware-based emulation in a common flow.

Solution

- Use Mentor Graphics verification IP (VIP) when building a testbench for the Questa Advanced Simulator.
- Connect this testbench to a Veloce emulation system via TestBench XPress (TBX) co-modeling software.
- Separate all blocks of design code into two domains — synthesizable code, including all RTL, for running on the emulator; and all other modules that run on the HDL portion of the environment on the simulator (which is connected to the emulator).
- Work with Mentor Graphics to fine-tune the new co-emulation verification environment, which requires that all SoC components be mapped exactly the same way in simulation and emulation.

life.augmented

www.mentor.com

> 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

72 UVM & Emulation, DVCon Europe 2014, HvdS & EB

Collateral for Further Learning

- Verification Academy
  - Course: SystemVerilog Testbench Acceleration
    https://verificationacademy.com/courses/systemverilog-testbench-acceleration
  - UVM Cookbook: Testbench Acceleration through Co-Emulation
    https://verificationacademy.com/cookbook/emulation
- Publications
  - MGC 2014: "From Simulation to Emulation – A Fully Reusable UVM Framework"
    www.mentor.com/products/fv/resources/overview/from-simulation-to-emulation-a-fully-reusable-uvm-framework-0def891c-ab7a-453d-b079-2c99f584650e
  - IJVLSIDCS 2013: "Accelerating SystemVerilog UVM-based VIP to Improve Methodology for Verification of Image Signal Processing Designs Using HW Emulator"
    airccse.org/journal/vlsi/papers/4613vlsi02.pdf
  - DVCon 2013/TechOnLine: "Unifying Hardware Assisted Verification and Validation using UVM and Emulation"
    www.techonline.com/electrical-engineers/education-training/tech-papers/4425340/Unifying-Hardware-Assisted-Verification-and-Validation-Using-UVM-and-Emulation
  - DAC 2012: "Development of a Unified Platform for Accelerated SoC Verification and Validation"
    https://s3.amazonaws.com/verificationhorizons.verificationacademy.com/volume-9_issue-1/articles/stream/bringing-verification-and-validation-under-one-umbrella_vh-v9-i3.pdf
  - MGC 2012: "Simulation + Emulation = Verification Success"
    www.mentor.com/products/fv/success/stmicroelectronics_simulation-emulation
  - TechOnLine India, 2011: "Taking Verification Productivity to the Next Level"
    www.techonlineindia.com/techonline/news_and_analysis/169218/taking-verification-productivity-level
  - DVCon 2011: "Off to the Races with Your Accelerated SystemVerilog Testbench"
    events.dvcon.org/2011/proceedings/papers/05_3.pdf
    verificationhorizons.verificationacademy.com/volume-7_issue-2/articles/stream/a-methodology-for-hardware-assisted-acceleration-of-ovm-and-uvm-testbenches_vh-v7-i2.pdf
  - DVCon 2008: "An Acceleratable OVM Methodology based on SCE-MI 2"
    www.mentor.com/products/fv/resources/overview/an-acceleratable-ovm-methodology-based-on-sce-mi-2-ae7634ed-5672-4d8a-aa6a-3542451778d8

73 UVM & Emulation, DVCon Europe 2014, HvdS & EB

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 architected for simulation and emulation portability to boost block-to-system verification productivity

Thank You!

Questions?