UVM Acceleration
The Most Efficient Way to Shorten Your Verification Cycle

December, 2014
Justin Zhang
Technical Manager
Mentor Emulation Division
Goal is to Reduce Time Spent in Verification

Mean time verification engineers spend in different tasks

- Debug: 36%
- Creating & Running Tests: 23%
- Testbench Development: 22%
- Test Planning: 15%
- Coverage Emulation
- Waveforms
- Assertions
- Transactions
- UVM
- VIP
- Models

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

A Complete Solution Leveraging Multiple Verification Engines
Verification Productivity

- Electronics systems companies need dramatic improvements in verification productivity

- SV and UVM make verification engineers more productive
  - Vertical and horizontal reuse
  - Faster to develop reusable testbenches and automated tests

- Veloce accelerates SV/UVM
  - Platform reuse
  - Dramatic speed up of execution of verification
Users Demand the Best of Both Worlds

- Faster Bring-up
- More Speed!
- More Functionality
- Testbench Reuse

Simulation → More Speed! → Emulation

Emulation → More Functionality → Simulation
Testbench Pivotal to Acceleration

Software Simulation

Testbench  

Design

Simulation Acceleration (Co-Simulation)

Testbench

2X to 10X

Transaction-Based Acceleration (Co-Emulation)

T  

D

50X to 1000X +
Key to Co-Emulation Performance

Transaction-Based Communication

Infrequent Information-Rich Data Exchange between Emulator and Simulator is Key to Performance
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
Co-Emulation: Key Concepts

- Testbench partitioned into **two separated domains** – 2 tops
  - **Timed/synthesizable HDL side**: DUT + BFMs and clk/rst generation
  - **Untimed HVL/TB side**: testbench generation and analysis code

- **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 for instantaneous configuration, FSM initiation, control and status
    - Buffered communication via SCE-MI 2 pipes for streaming applications
  - Domains bound together using SV virtual interfaces or SV-DPI

- **One unified testbench** for simulation and acceleration
  - Interoperable between pure simulation and co-emulation
How To Accelerate Your SV/UVM TB?

Main considerations

- Testbench architecture
- HVL-side modeling
- HDL-side modeling
- HVL-HDL communication
- Performance
# Dual Domain Testbench Architecture

<table>
<thead>
<tr>
<th>HVL/TB Side</th>
<th>HDL Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Untimed</td>
<td>1. Timed</td>
</tr>
<tr>
<td>2. Behavioral</td>
<td>2. Synthesizable</td>
</tr>
<tr>
<td>3. Class-based</td>
<td>3. Module/interface based</td>
</tr>
<tr>
<td>4. Dynamic</td>
<td>4. Static</td>
</tr>
<tr>
<td>5. Communication with HDL side only through transactors</td>
<td>5. Communication with HVL side only through transactors</td>
</tr>
<tr>
<td>7. Changes don’t cause emulation recompile</td>
<td>7. Changes may require emulation recompile</td>
</tr>
<tr>
<td>8. Standards like UVM apply</td>
<td>8. XRTL and synthesis standards apply</td>
</tr>
<tr>
<td>9. Verification engineer’s comfort zone</td>
<td>9. ASIC designer’s comfort zone</td>
</tr>
</tbody>
</table>

- "Untimed" and "Timed" refer to the timing characteristics of the testbench.
- "Behavioral" and "Synthesizable" indicate whether the design is intended for behavioral simulation or for synthesis.
- "Class-based" and "Module/interface based" indicate the structure of the design.
- "Dynamic" and "Static" refer to the predictability and stability of the design.
- "Communication with HDL side only through transactors" and "Communication with HVL side only through transactors" describe the communication mechanisms.
- "Programming optimization techniques dictate performance" and "Synthesis skill and transactor design dictate performance" highlight the key factors affecting performance.
- "Changes don’t cause emulation recompile" and "Changes may require emulation recompile" indicate the flexibility of the design.
- "Standards like UVM apply" and "XRTL and synthesis standards apply" mention the compliance with industry standards.
- "Verification engineer’s comfort zone" and "ASIC designer’s comfort zone" reflect the ease of use and the designer's expertise.
UVM Layered Testbench

Much of UVM Domain is Naturally Untimed
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

- 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 DPI 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

- HDL models synthesized with Veloce run at full emulator clock rate for high performance
Veloce XRTL Language Features

- RTL subset
- SystemVerilog interface
- SystemVerilog Virtual Interface
- SystemVerilog final block
- SystemVerilog nested interface
- 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
XRTL Modeling: Clock Generation

// tbx clkgen
initial begin
  clk = 0;
  forever
    #5 clk = ~clk;
  end
end

// tbx clkgen
initial begin
  clk = 0;
  #3;
  forever begin
    clk = ~clk;
    #5;
  end
  end
end

// tbx clkgen
initial begin
  clk = 0;
  #3;
  forever begin
    clk = 1;
    #6 clk = 0;
    #4 clk = 1;
    #4 clk = 0;
  end
  end
end

// tbx clkgen
initial begin
  clk = 0;
  #3;
  forever begin
    clk = 1;
    #4;
    clk = 0;
    #6;
  end
  end
end
XRTL Modeling: Initialization

- Simple initial blocks for all signal initialization

```verilog
parameter ALL_OK = 1;
initial begin
    receivedCount <= 0;
    receivedStatus <= ALL_OK;
end
```

- Initialization at declaration

```verilog
logic [1:0] state = 2'b00;
```

- Initialization in initial implicit FSMs

```verilog
initial begin
    reset <= 0;
    count = 1;
    @(posedge clk);
    while(count < reset_cycles)
    begin
        count = count + 1;
        @(posedge clk);
    end
end
```
XRTL Modeling: Implicit FSMs

Explicit

```verilog
initial state = STATE_0;
always @(posedge clock)
begin
  case( state )
    STATE_0: begin
      port0 <= data;
      state <= STATE_1;
    end
    STATE_1: begin
      port1 <= data;
      state <= DONE;
    end
  DONE: begin end
endcase
end
```

Implicit

```verilog
always
begin
  @(posedge clock ); // STATE_0
  port0 <= data;
  @(posedge clock ); // STATE_1
  port1 <= data;
  while(1)
    @(posedge clock ); // DONE
  end
end
```
Transactors – Co-Emulation Building Blocks
HDL BFM + HVL Proxy + HVL-HDL Channel

Testbench

Transactions

API

Higher-level TB
(SV/UVM, or C/C++/SC)

Transactor proxy is-a
uvm_driver or
uvm_monitor

Inbound communication

T/F call

HVL proxy class

Outbound communication

Signal or port
connections

RTL DUT

HVL proxy class

T/F call

HDL BFM
mod/if

DUT written in RTL

BFM written in “RTL++”

Transactions

Host

Signal or port
connections

RTL DUT

HVL proxy class

T/F call

HDL BFM
mod/if

DUT written in RTL

BFM written in “RTL++”
SVTB – HDL Transaction Transport Use Models

Host Workstation

SV Testbench side

- Testbench Model
- Proxy Model
- Proxy Model
- Proxy Model

“Accelerated SV Channels”

- SV Virtual Interfaces
- SV Connect
- SV SCEMI Pipes

Choice of 3 transaction transport use models

Emulator (or Simulator)

HDL side

- BFM
- DUT
- BFM
- BFM

Timed signal-level activity between DUT and BFMs

Untimed transactions between TB and/or optional proxy models and transactors
Unified UVM Simulation/Emulation Testbench

Diagram showing the various components of a UVM-based simulation/emulation testbench, including:

- Test Controller
- Coverage
- Scoreboard
- Monitor
- Emulator
- DUT
- Monitor
- Proxy Class
- SV Interface (BFM)
- Stimulus
- Driver
- Monitor
- Responder
- Slave
- SV Interface (pins)
- "Remote" task & function calls
- RTL Layer
- Transactor Layer
- Testbench Layer

The diagram illustrates the flow of data and control signals between these components, highlighting the integration of various testing layers and interfaces.
“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

View

New portable" Co-Emulation View

Top

Config

Sequencer

Monitor

Driver

Pin IFs

DUT

HVL Top

Config

Sequencer

Monitor

Driver

Tasks/Functions/Pipes

Agent BFM

Driver BFM

Pin IFs

DUT

HDL Top
Vertical Reuse in Dual Top UVM Framework
Example UVM Driver

```
class ahb_seq_item
extends uvm_sequence_item;
... rand bit we; rand bit [31:0] addr; rand bit [31:0] data; rand
int delay;
rand bit error;
constraint ahb_seq_item_delay_c {
    delay < 100;
}
...
endclass

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

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
derenityface
```

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

Virtual interface pointer from testbench side to HDL-side BFM

Time consuming task called directly from the testbench side to the HDL side
The UVM driver wiggles the DUT pins indirectly, no longer directly
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

```vhdl
import ahb_types_pkg::*;

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

Shared HVL-HDL package
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

```verbatim
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;
...
endpackage

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:0] ahb_seq_item_vector_t;

Optimization: byte vs. int
```

Shared HVL-HDL package
Example UVM Monitor

- Same idea, but ...

```verilog
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
```

- **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
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

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

... Package import of back-pointer class type

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

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

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

Assigning the back-pointer in the build or connect phase
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

```verilog
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

```天堂
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 – Basic Usage

```plaintext
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

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

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

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

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

comi evaluates to true for last element in a message
```
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

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
endinterface

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

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

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
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
Collateral for Further Learning

- **Verification Academy**
  - Course: SystemVerilog Testbench Acceleration
    [https://verificationacademy.com/courses/systemverilog-testbench-acceleration](https://verificationacademy.com/courses/systemverilog-testbench-acceleration)
  - UVM Cookbook: Testbench Acceleration through Co-Emulation
    [https://verificationacademy.com/cookbook/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-b079-2c99f584650a](www.mentor.com/products/fv/resources/overview/from-simulation-to-emulation-a-fully-reusable-uvm-framework-0def891c-ab7a-453d-b079-2c99f584650a)
  - **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](airccse.org/journal/vlsi/papers/4613vlsi02.pdf)
  - **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”
    [events.dvcon.org/2011/proceedings/papers/05_3.pdf](events.dvcon.org/2011/proceedings/papers/05_3.pdf)
  - **DVCon 2008**: “An Acceleratable OVM Methodology based on SCE-MI 2”
UVM Acceleration Methodology Benefits

- Enables truly single source testbench for simulation and co-emulation

- Leverages power of SV and UVM testbench features
  - Object-oriented modeling, constrained-random generation, coverage-driven verification, intelligent checking
  - TLM interfaces, sequences, configuration, phasing, factory,

- Is unobtrusive to established verification best practices
  - Co-emulation guidelines are largely orthogonal to prevalent verification methodology features and guidelines – i.e. UVM

- Veloce offers the opportunity for dramatic improvements in productivity and performance
Summary

- UVM offers proven verification productivity through reuse

- Creating an emulation-ready UVM testbench requires architecture considerations but performance benefits can be substantial

- Your next UVM projects should be architected for unified simulation and co-emulation to boost block-to-system verification productivity
Thank You

Justin Zhang
December 2014