UVM hardware assisted acceleration with FPGA co-emulation

Alex Grove, Aldec Inc.
Tutorial Objectives

• Discuss use of FPGAs for functional verification, and explain how to harness FPGAs into a mainstream verification methodology such as UVM

• Introduce a SCE-MI based approach using the Easier UVM coding style as a reference for industry best practice

• Outline a methodology for a portable and interoperable UVM simulation environment that is acceleration ready
The Why? The Need For Speed..

• Moore’s law still keeps on going..
  – Now set to the doubling of transistors every two years

• Emulation that’s as old as EDA is in growth!
  – Significant growth in the last three years

• Verification continues to get harder and harder
  – Wilson Research Group Functional Verification Study
  – Now includes S/W (HdS – Hardware Dependent Software)

• The death of CPU scaling ~2010
  – Multi-cores are not utilized in RTL simulation

• The rise of constrained random approaches
  – Required for coverage of today’s complex designs
The Death Of CPU Scaling

Chuck Moore, "DATA PROCESSING IN EXASCALE CLASS COMPUTER SYSTEMS", The Salishan Conference on High Speed Computing, 2011
The Why? The Need For Speed..

• Moore’s law still keeps on going ..
  – Now set to the doubling of transistors every two years
• Emulation that’s as old as EDA is in growth!
  – Significant growth in the last three years
• Verification continues to get harder and harder
  – Wilson Research Group Functional Verification Study
  – Now includes S/W (HdS – Hardware Dependent Software)
• The death of CPU scaling ~2010
  – Multi-cores are not utilized in RTL simulation
• The rise of constrained random approaches
  – Required for coverage of today’s complex designs
FPGAs as a Verification Platform

- FPGAs are reprogrammable .. have replaced test chips
- Low cost as “generic” platforms
  - Large devices used by leading network companies
  - 0.25 to 0.5 cents per gate vs. 2-5 cents of big box emulators*
- Leading edge technology node e.g. UltraScale @ 20nm
  - Very large capacity with stacked silicon interconnect (SSI)
    - 2000T = ~ 14 M ASIC Gates @ 60% utilization
    - VU440 = ~ 29 M ASIC Gates @ 60% utilization
- FPGA Vendors provide tools with the silicon
  - Tools are available before silicon for lead partners
  - Have incremental build capabilities
- Only FPGAs provide the MHz performance needed for S/W

* Hogan compares Palladium, Veloce, EVE ZeBu, Aldec, Bluespec, Dini
The FPGA Co-Emulator/Accelerator

- Hardware (HES : Hardware Emulation System)
  - FPGA based system designed for verification
  - PCIe communication to host for SCE-MI
  - Built-in emulation resources (RAM, LVDS/GTX, debug traces)
- Compilers (DVM : Design Verification Manager)
  - Mix of custom compilers & FPGA vendor tools
    - Includes partitioner & automatic multiplexing of signals
  - Automate the mapping of the design to the FPGA system
- Run-time environment
  - Full control and observability
    - RTL like debug capabilities (dynamic & static probes)
  - Integration with HDL simulators (similar use model)
- VIP
  - Transactors (SCE-MI) for standard interfaces AXI, AHB, SPI, PCI, USB ..
  - Speed Adaptors for hardware interfaces (USB, Ethernet, PCIe)
10,000 Feet View Hardware Assisted

H/W RTL Debug Capability
(Controllability, Observability, & Incremental Turn time)

*SNEAK PEEK: INSIDE NVIDIA’S EMULATION LAB*
Increasing UVM throughput with FPGA-based Co-Emulation

1. HDL Simulator with SystemVerilog and UVM support
2. FPGA prototyping board with PCIe host interface
3. SCE-MI infrastructure integration tool
4. FPGA synthesis and place & route software

5. Design with UVM Testbench compliant to SCE-MI
UVM Best Practices

HVL

Class-based

HDL

Module-based

Easier UVM diagram kindly provided by Doulos
Typical UVM Simulation

BFM functionality is implemented in UVM Driver and Monitor

```verilog
task driver::do_drive();
@posedge vif.CLK);
while (vif.RST)
  @negedge vif.CLK);
vif.DI <= 'hA5A5A5A5A5;
vif.WR <= 1'b0;
//(....)
endtask
```
Typical UVM Simulation

Bottleneck: signal level acceleration only

Simulation

Acceleration
Guidelines Simulation Acceleration

• Consistent design & testbench source for simulation and acceleration in FPGA
  – Enables interoperability with simulation only and acceleration

• Transaction-level interfaces between testbench and design
  – With compact transaction messages you avoid simulator/emulator throughput bottleneck

• Separation of Timed/Untimed behavior
  – Simulate untimed transactions in UVM/HVL
  – Accelerate timed (design, transactors, clock reset generators)
  – Do not use clocks to synchronize with testbench
  – Synchronize testbench and design with transactions and events

• And one more.... on the next page ➔
SCE-MI – Standard Co-Emulation Modeling Interface

“SCE-API 2.2 speeds up electronic design verification by allowing a model developed for simulation to run in an emulation environment and vice versa”

• Why use SCE-MI?
  – Mature standard
  – Independent
  – Widely accepted

• Today we are using 4.7 function-based interface
Using Transactional Interface

**BFM Proxy**
- Using **SV Interface** to comply with UVM best practices
- Forwards transaction-level interface to UVM
- Defines TB notification transactions used by BFM Module

**BFM Module**
- Implements Bus Functional Model
- Provides transaction-level interface used by Testbench & UVM

**Test Harness and Testbench**
- Communicate using untimed transactions
- Clock generation remains in Test Harness
- Testbench does not use clock
Using Transactional Interface

What we achieve

- Transaction level interface between Testbench and Test Harness
- Untimed communication
- Same testbench for simulation and acceleration

Simulation

Acceleration

Transaction-Level Acceleration
Transactional Interface with SCE-MI

SCE-MI function based use model concept

- Function call makes a transaction
- Transaction bears a message in
  - Call arguments
  - Return value
- Function defined in HDL is called in HVL context (export)
- Function defined in HVL is called in HDL context (import)
SV DPI-C in SCE-MI Function-Based

**HVL**
SystemVerilog, UVM

- Call Functions from HDL
  - hdl_PutData()
  - hdl_GetData()

- Define Notification Functions for HDL
  - ResetDone()

**DPI-C wrapper**
C, C++

- Wrap HDL Functions
- Handle scope

- hdl_*()
- Import

- *( )
- Export

**HDL**
SystemVerilog

- Define Functions for Testbench
  - PutData()
  - GetData()

- *( )
- Import

- hvl_*( )
- Export

- hvl_ResetDone()
- Call HVL Functions

Call Chain
Tutorial Example – UUT

Design Under Test
- Register file with SRAM-like interface
- Asynchronous reset (RST)
- Synchronous write (WR=1)
- Asynchronous read (WR=0)
Typical Connection of UVM to Design

```verilog
module top_th;

logic clock = 0;
logic reset;

bus0_if bus0_if0();

mydut uut (  
  .RST(mydut_if0.RST),  
  .CLK(mydut_if0.CLK),  
  .EN(mydut_if0.EN),  
  .WR(mydut_if0.WR),  
  .SEL(mydut_if0.SEL),  
  .DI(mydut_if0.DI),  
  .DO(mydut_if0.DO)  
);
//...

interface bus0_if();
  logic RST;
  logic CLK;
  logic EN;
  logic WR;
  logic [1:0] SEL;
  logic [31:0] DI;
  logic [31:0] DO;
endinterface : bus0_if
```

Interface instance `bus0_if0` makes a hook for UVM Driver connection
Typical Connection of UVM to Design

```verilog
module top_tb;
    // (...) some other boilerplate code

    // UVM Config
    top_config env_config;

    initial begin
        // Create and populate UVM Config
        env_config = new("env_config");
        if (~env_config.randomize())
            `uvm_error("top_config", "randomize failed")

        env_config.bus0_vif = top_th.bus0_if0;

        // more config settings below ...
    end
```

Hierarchical name of `bus0_if0` interface passed to UVM components via configuration object `env_config`
class bus0_driver extends uvm_driver #(bus0_rw_tr);
  `uvm_component_utils(bus0_driver)
  virtual bus0_if vif;
  extern function new(string name, uvm_component parent);
  extern task run_phase(uvm_phase phase);
  extern function void report_phase(uvm_phase phase);
  extern task do_drive();
endclass : bus0_driver

task bus0_driver::run_phase(uvm_phase phase);
  forever
    begin
      seq_item_port.get_next_item(req);
      do_drive();
      seq_item_port.item_done();
    end
endtask : run_phase

task bus0_driver::do_drive();
  @(posedge vif.CLK);
  // Wait until reset is off
  while (vif.RST)
    @(negedge vif.CLK);
  // Set default values
  vif.DI <= 'hA5A5A5A5A5;
  vif.WR <= 1'b0;
  vif.SEL <= req.sel;
  if (req.wr) begin
    vif.DI <= req.data;
    vif.WR <= 1'b1;
  end
  // Enable operation and execute
  vif.EN <= 1'b1;
  @(posedge vif.CLK);
  vif.EN <= 1'b0;
endtask

Virtual interface used to drive and sense design ports
Changes for Acceleration Ready Test Env.

1. Replace `bus0_if` interface with BFM module
2. Move `do_drive` task to BFM module and export it using SV DPI-C
3. Create DPI-C wrapper for `do_drive`
4. Create BFM proxy interface and connect it with UVM
5. Change UVM Driver to use imported `do_drive`
Creating BFM Module (Xtor)

BFM module also called Transactor (xtor)

Changed instance under top_th
Moving do_drive to Xtor

- Export task via DPI-C
- Input arguments make transaction

```vhdl
task bus0_driver::do_drive();
  @(posedge vif.CLK);
  // Wait until reset is off
  while (vif.RST)
    @(negedge vif.CLK);
  // Set default values
  vif.DI <= 'hA5A5A5A5;
  vif.WR <= 1'b0;
  vif.SEL <= req.sel;
  if (req.wr) begin
    vif.DI <= req.data;
    vif.WR <= 1'b1;
  end
  // Enable operation and execute
  vif.EN <= 1'b1;
  @(posedge vif.CLK);
  vif.EN <= 1'b0;
endtask

export "DPI-C" task do_drive;
task do_drive(
  input byte wr_dpi,
  input byte sel_dpi,
  input int unsigned data_dpi);
  @(posedge CLK);
  // Wait until reset is off
  while (RST)
    @(posedge CLK);
  // Set default values
  di <= 'hA5A5A5A5;
  wr <= 1'b0;
  sel <= sel_dpi[1:0];
  if (wr_dpi[0]) begin
    di <= data_dpi;
    wr <= 1'b1;
  end
  // Enable operation and execute
  en <= 1'b1;
  @(posedge CLK);
  en <= 1'b0;
endtask
```
Creating DPI-C wrapper

• The wrapper is C/C++ function
• The simplest wrapper has to:
  – Set scope for called SV task
  – Call the exported SV task
• Can do additional computation or transformation of input data

```c
int hdl_do_drive ( 
    char wr, 
    char sel, 
    uint32_t data )
{
    // Set scope
    scopeutils::set_hdl_scope();
    // Call exported do_drive
    do_drive(wr, sel, data);
    return 0;
}
```

Using SystemVerilog DPI utilities:

• `svGetScope` and `svSetScope`

```c
void set_hdl_scope ()
{
    svScope my_scope = svGetScope(); //hvl scope
    svSetScope(g_scopes_map.find_hdl(my_scope));
}
```

`g_scopes_map` – a container with lookup methods to find corresponding HVL and HDL scopes
3

Scope handling helper class

class scopes {
    map<svScope, svScope> m_hvl_hdl;
    map<svScope, svScope> m_hdl_hvl;

public:
    void insert(svScope hvl, svScope hdl);
    svScope & find_hdl(svScope hvl);
    svScope & find_hvl(svScope hdl);
};

// global variable – container for scopes
extern scopes g_scopes_map;

g_scopes_map – a container with lookup methods to find corresponding HVL and HDL scopes
set_scopes – function called on SystemVerilog HVL site via DPI-C

void set_scopes(const char * hdl_path)
// Used to set HDL and HVL transactor parts (the scopes)
// This function must be called once for each transactor
// at the beginning of simulation
// This function must be called in HVL scope
{
    svScope hvl_scope = svGetScope();
    svScope hdl_scope = svGetScopeFromName(hdl_path);
    scopeutils::g_scopes_map.insert(hvl_scope, hdl_scope);
}
Creating BFM Proxy

---

interface bus0_if();

// Scope initialization
import "DPI-C" context function
    void set_scopes(input string hdl_path);

// Driver task
import "DPI-C" context task hdl_do_drive(
    input byte wr_dpi,
    input byte sel_dpi,
    input int unsigned data_dpi);

// Monitor task
export "DPI-C" task do_mon;
    task do_mon(
        input byte wr_dpi,
        input byte sel_dpi,
        input int unsigned data_dpi );
endinterface : bus0_if

---

Use SV interface

Import function for handling scopes

Import functions from BFM module
Connecting BFM Proxy

module top_tb;
    // (...) some other boilerplate code
    // BFM xtor proxy instance
    bus0_if bus0_if0_scemi_proxy();

initial
    bus0_if0_scemi_proxy.set_scopes("top_th.bus0_if0");

// UVM Config object
    top_config env_config;

initial
    begin
        // Create and populate UVM Config
        env_config = new("env_config");
        if ( !env_config.randomize() )
            `uvm_error("top_config", "Randomize failed")

        env_config.bus0_vif = top_tb.bus0_if0_scemi_proxy;
    end

- BFM Proxy instantiated under Testbench module (top_tb)
- Its handle is passed to UVM in a typical way
task mybus0_driver::do_drive();

byte wr = 8'b0 | req.wr;
byte sel = 8'b0 | req.sel;
int unsigned data = req.data;

// Call imported DPI-C task from BFM proxy
vif.hdl_do_drive(wr, sel, data);
endtask

• New implementation of UVM Driver task do_drive
• No more UVM code changed
Walking the Call Chain

UVM Driver, SystemVerilog

```systemverilog
task bus0_driver::do_drive();
  // Call imported DPI-C task
  vif.hdl_do_drive(wr, sel, data);
endtask
```

BFM Module, SystemVerilog

```systemverilog
task do_drive(
  input byte wr_dpi, sel_dpi,
  input int unsigned data_dpi);
  @(posedge CLK);
  while (RST) @(posedge CLK);
  di <= 'hA5A5A5A5;
  wr <= 1'b0;
  sel <= sel_dpi[1:0];
  if (wr_dpi[0]) begin
    di <= data_dpi;
    wr <= 1'b1;
  end
  en <= 1'b1;
  @(posedge CLK);
  en <= 1'b0;
endtask
```

BFM Proxy Interface, SystemVerilog

```systemverilog
interface bus0_if();
  // Driver task
  import "DPI-C" context
  task hdl_do_drive(
    input byte cmd_wr_nrd, sel
    input int unsigned data);
  //(...)
endinterface
```

DPI-C Wrapper, C/C++

```c
int hdl_do_drive ( char wr,
  char sel, uint32_t data )
{
  // Set scope
  scopeutils::set_hdl_scope();
  do_drive(wr, sel, data);
  return 0;
}
```
Summary Of Adaptions
Clock & Reset Generation

// Clock generator process
initial
begin
  clock = 0;
  #5;
  forever begin
    clock = 1'b1;
    #5;
    clock = 1'b0;
    #5;
  end
end

// reset generator process
initial
begin
  reset = 1;
  repeat(5) @(negedge clock);
  reset = 0;
end

Clock & Reset behavioral processes is automatically converted to FPGA resources (SCE-MI infrastructure)
SCE-MI Transactors Coding Style

• SCE-MI does not impose any coding style
• Common denominator is: Synthesizable + DPI-C
• Compilers typically accept more than RTL:
  – ISM – Implicit State Machines (used in this tutorial!)
  – System tasks (e.g. $display, $readmemh)
  – Shared variables (multiple drivers)
  – Hierarchical names
  – Named events (-->reset_done_event)
SCE-MI Constraints on the DPI-C

“SCE-MI uses a subset of DPI that is restricted in such a way as to provide a nice balance between usability, ease of adoption and implementation.”

- Data types used with DPI-C functions are limited
- 4-state logic can be converted to 2-state (1/0)
- Supported 2 levels of nesting when calling imported functions from exported and vice versa
# SCE-MI Constraints on the DPI-C

<table>
<thead>
<tr>
<th>DPI formal argument types</th>
<th>Corresponding types mapped to C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalar basic types:</strong></td>
<td></td>
</tr>
<tr>
<td>bit</td>
<td>scalar basic types:</td>
</tr>
<tr>
<td>byte</td>
<td>unsigned char</td>
</tr>
<tr>
<td>byte unsigned</td>
<td>char</td>
</tr>
<tr>
<td>shortint</td>
<td>unsigned char</td>
</tr>
<tr>
<td>shortint unsigned</td>
<td>short int</td>
</tr>
<tr>
<td>int</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>int unsigned</td>
<td>int</td>
</tr>
<tr>
<td>longint</td>
<td>unsigned int</td>
</tr>
<tr>
<td>longint unsigned</td>
<td>long long</td>
</tr>
<tr>
<td>longint unsigned</td>
<td>unsigned long long</td>
</tr>
<tr>
<td><strong>Constant string type:</strong></td>
<td>Constant string type:</td>
</tr>
<tr>
<td>string</td>
<td>const char *</td>
</tr>
<tr>
<td><strong>Packed one or multi dimensional arrays of type bit and logic</strong></td>
<td>Canonical arrays of svBitVecVal and svLogicVecVal</td>
</tr>
<tr>
<td><strong>Packed struct types</strong></td>
<td>Canonical arrays of svBitVecVal and svLogicVecVal</td>
</tr>
</tbody>
</table>
Acceleration Ready
what’s next?

Running Simulation Acceleration
Running the UVM Simulation

1. Compile UUT
2. Compile UVM Test Environment
3. Elaborate & Run Testbench
4. Compile XTORS & Test Harness
5. Compile DPI-C lib
6. DPI-C sources.list

Test Harness & RTL DUT

UVM TB
Accelerating Test Harness

Compiled Share Libs Running On Host

FPGA Co-Emulator
Creating the Acceleration Build

Test Harness & RTL DUT

Compile Design

Pre Synthesis

Synthesis

Partitioning

Generate Config

Implement (P&R)

SCE-MI 2 Compiler for UVM

Debug Instrumentation
Setup Memory Models

Guided partitions
Configure memory models
Connect to daughter-board

Generates xml config & makefiles for SCE-MI Bridge
DPI-C proxy for HVL
Running UVM Simulation Acceleration

1. UVM TB
2. Elaborate Testbench
3. Program & Run HVL
4. SCE-MI DPI Bridge
5. Compile & Link
6. DPI-C sources.list
7. Generated SCE-MI to DPI bridge (C++ code)
8. config.xml
9. FPGA bit-files
Summary

- The use of FPGAs can be extended to functional verification through the use of a co-emulation system.
- Demonstrated minor adaptations to the Easier UVM coding style that would enable acceleration with a co-emulator through the use of SCE-MI.
- Using standards, SystemVerilog & SCE-MI, provides a common interoperable testbench for both simulation and hardware-assisted verification.
Additional Reading & References

• Acceleration Solutions on Aldec’s website: www.aldec.com/solutions/acceleration

• SCE-MI: http://accellera.org/downloads/standards/sce-mi

• Easier UVM: http://www.doulos.com/content/events/easierUVM.php
Questions