UVM Testbench Considerations for Acceleration

Kathleen A Meade
Cadence Design Systems, Inc
Hardware-Assisted Verification Use Modes

- Software Simulation
- Block/IP Verification
- Sub-System/SoC Verification
- SoC-Level Hardware/Firmware Integration
- Full System Validation with App Software
- Field Prototype

- Hybrid
- Transaction-Based Acceleration
- Signal-Based Acceleration
- Synthesizable Testbench (STB)
- In-Circuit Emulation

Performance
Objectives of UVM Acceleration

UVM provides methodology for verifying complex designs with a focus on reuse

• Reuse of verification components and environments between simulators and hardware acceleration is gaining momentum
• This session introduces methodology techniques for creating acceleration-friendly UVM environments
  – Makes migration from simulation to acceleration much easier
  – Will not have a negative impact on performance for pure simulation

• Ultimate goal: Enhance simulation performance to run more cycles and achieve desired coverage faster
The following topics will be covered:

• Partitioning the top-level into a hardware and software top module

• Separating your UVM *monitor* into a *collector* (for signal-level information) and a *monitor* (for checking/coverage)

• Limiting access between the DUT and the testbench.

• Creating synthesizable interface tasks to:
  – Take transactions and drive signals (*driver*)
  – Reassemble transactions from signal-level details (*collector*)

• Removing timing from sequences
Is Hardware Acceleration a Good Option?

Hardware Acceleration can improve performance to 300x (or more)

- Profile your environment with a long simulation runtime
- Check that a significant portion of time is being spent in the DUT
- Acceleration is usually not a good option for environments where the testbench time is significant and the DUT time is small
- Acceleration is also not a good option if your simulation runtimes are short
  - Consider grouping short tests into one longer test with automatic checking
The main factors in determining performance potential:

- **Testbench Runtime** – Time spent in the simulator, including configuration and building of the UVM verification environment and DUT, driving stimulus, checking and sampling coverage
- **SW/HW Synchronizations** – Signals and transactions sent between the UVM Testbench and DUT
- **DUT Runtime** – Includes any aspect of the design (and testbench) that is executed on the acceleration hardware

* : SW/HW Synchronization Overhead
Analyzing Profile Results – An Example

Profiling a simulation run – for acceleration:

- Testbench takes 25% of simulation time
- DUT takes 60%
- Synchronization estimation is 15%

Maximum performance boost, if the DUT time is reduced to zero is:

\[ \text{HW\_TIME} = 60\% + 60\% \times 15\% = 69\% \]

Estimated speed-up (no opt) = \(\frac{100}{100-\text{HW\_TIME}}\) = 3.2X

Increasing DUT time to 90% and reducing sync overhead to 5%:

\[ \text{HW\_TIME} = 90\% + 90\% \times 5\% = 94.5\% \]

Estimated speed-up = \(\frac{100}{100-94.5}\) = 18.2X

Further testbench optimizations can be made for even better performance
Partitioning Your UVM Environment

- Partition your top-level module so the DUT and synthesizable components are in one module and the UVM testbench is in a separate module.
Hardware Top-Module Example

• Includes the DUT instance, a clock generator, SystemVerilog interfaces and (optional) memory

```verilog
module hw_top (); 
  // Interface Instances
  my_uvc_if   my_uvc_if0(clock, reset);
  clk_reset_if clk_reset_if0 (start_clock, reset);

  // Clock/reset generator, memory (opt)
  clkgen    clkgen(start_clock, clock);
  memory    mem_inst(...);

  // DUT instance
  top_dut    dut (clock, reset,
    .uvc_if(my_uvc_if0), ...);

endmodule
```
Testbench Top-Module Example

- Includes the UVM package, user-defined UVC packages, the top-level testbench and test files and an initial block to configure and run the test.

```verilog
testbench
module tb_top();
import uvm_pkg::*;
`include "uvm_macros.svh"
import my_uvc_pkg::*;
import clk_reset_pkg::*;
import my_tb_pkg::*;

initial begin
  uvm_config_db#(virtual my_uvc_if)::set(null,
    ".my_uvc0", "vif", hw_top.my_uvc_if0);
  uvm_config_db#(virtual clk_reset_if)::set(null,
    "*.clk_reset0", "vif", hw_top.clk_reset_if0);
  run_test();
end
endmodule
```

- **The UVM Package**
- **UVC and Testbench Packages**
- **Configure the virtual interfaces**
- **Run the test**
Implement a Collector/Monitor Pair

- A UVM **monitor** to captures transactions from the DUT interface, performs checking, coverage & sends them to other components
- Split the monitor into a **collector** class for capturing signals and forming transactions and a **monitor** for transaction-level checking and coverage
- Similar to the **sequencer/driver** pair for creation of stimuli
Monitor/Collector Interaction

class my_uvc_monitor extends uvm_monitor;
  uvm_analysis_imp#(my_uvc_trans, my_uvc_monitor) item_in_port;
  uvm_analysis_port#(my_uvc_trans) item_out_port;
...
endclass

class my_uvc_collector extends uvm_component;
  uvm_analysis_port#(my_uvc_trans) item_out_port;
...
endclass

class my_uvc_agent extends uvm_agent;
  my_uvc_monitor monitor;
  my_uvc_collector collector;
  //create in the build phase
  //connect in the connect_phase
  collector.item_out_port.connect(
      monitor.coll_in_port);
endclass
Minimize Testbench and DUT Interaction

• Every interaction between the DUT and testbench initiates a synchronization event
• Limit interaction to the collector and driver only
• All other interactions must be addressed when the testbench is migrated to run on acceleration
• Identify: Two common places where this is found:
  – Waiting for signals to change inside a sequence
  – Waiting for reset – not as critical as reset does not happen very often
• Address: Add a clock and reset agent and/or an interrupt agent to limit access to these signals and generate events related to them
Signal-Based vs Transaction-Based Acceleration

• Signal-based acceleration:
  – Partition the UVM testbench to the simulator move the DUT to the hardware side
  – Takes less time to implement and can produce results in the 3x-20x range

• Transaction-based acceleration:
  – Part of the testbench is also moved to the hardware accelerator
  – Interface between the testbench and the DUT is through task calls
  – Reduces the number of synchronizations and can produce results in the 20x-300x range!

• Goal is to develop your testbench environment to be conducive for transaction-based acceleration (TBA)
Traditional UVC Structure

- Signals driven and clocks referenced from driver and collector class through a virtual interface handle.
UVM Driver Class – drive_transfer() Task

- Signals driven and clocks referenced from driver and collector class through a virtual interface handle

```verilog
class my_uvc_driver extends uvm_driver;
...
task drive_transfer(my_trans_type_type trans);
 @(posedge vif.clock iff vif.reset)
  vif.addr <= trans.addr;
  vif.data <= trans.data;
  vif.rd_wr <= (trans.dir == READ) ? 1'b0 : 1'b1;
  vif.enable <= 1'b1;
 @(posedge vif.clock)
  if (trans.dir == READ)
    trans.data = vif.rdata;
  vif.enable <= 1'b0;
endtask
endclass
```

Virtual interface interactions cause a synchronization event
Accelerated UVC Structure

- Driver and collector call time-consuming interface tasks.
- Signals driven and clocks referenced directly in the interface (synthesizable)
UVM Driver Class – drive_transfer() Task

- The driver’s `drive_transfer` task calls a time-consuming interface task
- Data fields can be passed directly or converted to a packed struct

```vhs
class my_uvc_driver extends uvm_driver;
...
    task drive_transfer(my_trans_type trans);
        logic [31:0] rdata;
        vif.drive_transfer(trans.addr,
                             trans.data,
                             trans.dir, rdata);
        trans.data = rdata;
    endtask
endclass
```
interface my_uvc_if (clock, reset);
logic [31:0] addr;
logic [31:0] data, rdata;
logic rd_wr, enable;
...

task drive_transfer (logic [31:0] t_addr, t_data, trans_type_e t_dir, output logic [31:0] t_rdata);
wait (reset == 1);
@(posedge clock)
addr <= t_addr;
data <= t_data;
rd_wr <= (t_dir == READ) ? 1'b0 : 1'b1;
enable <= 1'b1;
@(posedge clock)
if (t_dir == READ) t_rdata = rdata;
enable <= 1'b0;
endtask
endinterface

Task should be “synthesizable” for hardware acceleration
Signals are driven/sampled directly in the interface task
Clocks and Reset

• Clocks are critical in designing for acceleration
  – They toggle frequently and cause synchronization events
• Implement a reusable agent to configure clocks, initiate resets and execute delay sequences from the testbench
• The implementation details are in a SV interface and are executed on the hardware
class clk_reset_driver extends uvm_driver#(clk_reset_item);
  virtual clk_reset_if vif;

  // `uvm_component_utils and constructor
  // Get the vif in the connect_phase

  task run_phase(uvm_phase phase);
    ...
    seq_item_port.get_next_item(req);
    if (((req.run_clk == 1) || (req.reset_delay != 0)))
      vif.start_clock(req.clk_period, req.reset_delay, ...);
    else vif.count_clocks(req.cycle_count, req.wait_count);
    seq_item_port.item_done();
    ...
  endtask
endclass

class clk_reset_item extends ...
  rand int clk_period;
  rand int reset_delay;
  rand int cycle_count;
  rand bit wait_count;
  rand bit run_clk;
  ...
endclass
Removing Timing From Sequences

- For optimum performance remove timing from the testbench
  - Includes references to clocks and #delay’s
  - Waiting on transition of a DUT signal

- Use alternate methods of specifying delays within and between sequences
  - Include a `delay_clocks` task inside your interface UVC to delay clocks within a UVC sequence
  - Use a clock and reset agent to execute sequences which starts the clocks and initiates resets (`start_clocks_seq`)
  - Also use the clock and reset agent to execute a sequence (`count_clocks_seq`) for clock delays (blocking) or timeouts
class traffic_vseq extends uvm_sequence;
  // `uvm_object_utils and constructor
  // Interface UVC Sequences
  my_uvc1_config_seq uvcl1_config_seq;
  my_uvc2_traffic_seq uvc2_traffic_seq;

  task body();
    #100 // Wait 100ns for reset to finish
    // Configure via UVC 1
    `uvm_do_on(uvc1_config_seq, p_sequencer.uvc1_master_seqr)
    // Run some traffic on UVC 2
    `uvm_do_on(uvc2_traffic_seq, p_sequencer.uvc2_master_seqr)
    // Wait 30 clocks (300ns) to end simulation
    #300;
    endtask

endclass
class traffic_vseq extends uvm_sequence;
  // `uvm_object_utils and constructor
  // Other sequences: uvcl1_config_seq, uvc2_traffic_seq
  start_clocks_seq clock_seq;
  count_clocks_seq delay_seq;
  // Other Sequences
  task body();
  // Start clocks and initiate reset
  // `uvm_do_on_with(clock_seq, p_sequencer.clk_reset_seqr,
  // { clock_seq.clock_period == 10;
  //   clock_seq.reset_delay == 10;
  //   clock_seq.run_clock == 1;    } )
  // Configure, run traffic
  // Wait 30 clocks (#300)to end
  `uvm_do_on_with(delay_seq, p_sequencer.clk_reset_seqr,
  { delay_seq.clock_cycles == 30;
    delay_seq.wait_count == 1;   } )
endtask
endclass
Summary

• Hardware acceleration of verification environments for performance is gaining momentum
• Spending time up front to construct UVM environments that are easily portable to hardware acceleration can reap big performance benefits
• The UVM recommendations introduced in this module
  – Makes migration from simulation to acceleration much easier
  – Will not have a negative impact on performance for pure simulation
• Enhance simulation performance to run more cycles and achieve desired coverage faster