ACE’ing the Verification of a Coherent System Using UVM

by

Parag Goel; Amit Sharma;
Ray Varghese; Romondy Luo; Satyapriya Acharya
Peer Mohammed

Synopsys & Mindspeed Technologies
Agenda

• Overview of the AXI-ACE protocol
• Challenges in the verification of a cache coherent system
• Addressing the challenges using UVM
  – Stimulus Generation
  – System Level Checks
  – Reusability & Debug
• Summary
Motivation & Overview for ACE Protocol

- Extends AXI to support multi-core
  - More processing power
  - Optimal Power consumption
  - Low latency systems
- Support for hardware-coherent caches.
  - All masters observe the correct data value at any address
  - 5 state cache model
- Additional signaling
  - Conveys updated information to locations that require hardware coherency support.
Overview (Continued)

- Allows for Complex Configuration of Masters, Slaves and Interconnect
- Additional channels
  - Enable communication with a cached master
- SNOOPs, Cache state transitions etc: new paradigms
Verification Challenges : ACE System

Generation of Complex Stimulus

• Large permutations of transfer attributes
  – Cache states, Transaction types, Burst lengths/sizes, Snoop mechanisms/responses
    • coherent transactions/ Responses to Snoop transactions
    • Cache line allocation & cache state transitions
    • speculative fetches, snoop filtering, User-specified scheduling

• All combinations of concurrent accesses

• Effective System level sequence generation
  – Across Initiating Masters, Snooped Masters, Slave Main Memory
Verification Challenges : ACE System

**Coding System Level Checks**
- Coherent/Snoop transactions accessing the same location
- Outstanding, interleaved and out-of-order transactions
- System Level Cache Coherency Validation
- Cache State Transition Validation
- Data Integrity checks

**Building Reusable Components**
- Vertical Reuse Challenges
  - Mix of RTL + testbench components
  - Should factor RTL behavior
- Horizontal Reuse Challenges
  - Compliance of components across projects
  - More cores-more address overlapping.
  - Complicated snoop scheduling/sequences
Cache Coherency Validation

VERIFICATION STRATEGY

• Integration testing

• Basic testing
  – Stimulus generation at single interfaces
  – Validate all transaction types correctly for all combinations of valid cache line states.

• Intermediate testing
  – Specific scenarios involving multi-master communication:
    • Overlapping writes
    • Snoop during memory update

• Advanced testing
  – System level Stimulus mapped to traffic profiles
Leveraging UVM capabilities to tackle ACE verification Challenges

- Building a reusable component - ACE UVM VIP
  - Factory enabled,
  - Callbacks,
  - Configuration mgmt,
  - TLM communication

- Stimulus Generation
  - Sequence Library: Atomic & Nested sequences
  - Virtual sequencer
Using UVM Based VIP to address challenges

- Slave models memory and responds to requests.
- Infrastructure for System Level Checks
  - Port Monitor for port-level checks.
  - System Monitor: system-level checks, coherency checks and data integrity checks.
- Configurable coverage model
  - Used in conjunction with ACE Verification Planner
UVM Stimulus Generation Infrastructure

Base class for transactions

Normal sequence on normal sequencer

callback tasks

Virtual sequence on virtual sequencer

Separates the stimulus generation from the test bench
class axi_basic_readclean extends uvm_sequence;

`uvm_object_utils(axi_basic_readclean)
  //Constructor

virtual task pre_start();
  if(strating_phase != null)
    starting_phase.raise_objection(this);
endtask

virtual task body();
  `uvm_do_with(req, {
    xact_type == svt_axi_transaction::COHERENT;
    coherent_xact_type == axi_transaction::READCLEAN;
  })
endtask

virtual task post_start();
  if(strating_phase != null)
    starting_phase.drop_objection(this);
endtask

endclass: axi_basic_readclean
Examples of a Typical Sequence: ReadClean Coherent Command

Grouping of sequences and nested scenarios

Statements:
- `uvm_do(seq1)`
- `uvm_do(seq2)`
- `uvm_do(seq3)`

```
extends uvm_sequence

cache_init_sequence
  cache_init_sequence seq1;
  cache_invalidate_sequence seq2;
  basic_readclean_sequence seq3;

task body();
  `uvm_do(seq1)
  `uvm_do(seq2)
  `uvm_do(seq3)
endtask
```
Virtual Sequencer
– Enables Orchestration

- Stimulus control across multiple masters in an environment.
  - In specific order, using individual sequencers

- Actual Sequencers ⇔ Virtual Sequencers
  - Instantiated & connected in agent/env

```
axi_env extends uvm_env;

axi_master0_sequencer
axi_master1_sequencer
axi_master_virtual_sequencer

basic_MakeUnique_sequence
basic_ReadShared_sequence

virtual_sequence runs on virtual_sequencer
```
Creating Configurable Sequences

virtual task body();
    bit status;
    axi_operation_type op_type;
    status = uvm_config_db#(axi_operation_type)::get(null,
        get_full_name(), "op_type", op_type);

if (op_type == axi_transaction::ACE_EVICT)
    `uvm_do(axi_basic_evict_seq)
else
    `uvm_do(axi_basic_writeback_seq)
endtask

• UVM configuration mechanism for added configurability
• Allows sequences to reconfigure themselves
  – Help Model Dynamic scenarios
• More granular control for virtual sequences
**Sequence Library Infrastructure**

- Grouping of sequences & virtual sequences
  - complex hierarchical scenarios from atomic sequences

```verilog
class uvm_sequence_library #(type REQ=int,RSP=REQ) extends uvm_sequence #(REQ,RSP);

class my_seq_lib extends uvm_sequence_library
    #(my_item);
    `uvm_object_utils(my_seq_lib);
    `uvm_sequence_library_utils(my_seq_lib);
    function new(string name="");
        super.new(name);
        init_sequence_library();
    endfunction
endclass

class my_seq1 extends my_seq;
    `uvm_object_utils(my_seq1);
    `uvm_add_to_seq_lib(my_seq1,my_seq_lib);
endclass
```
System Level Checks

- Building system monitor
  - UVM event pool
  - UVM Resource DB
- TLM-1.0/2.0 ports help redirect transaction.
- Enables checks related to
  - Data Integrity
  - Coherency
  - Scheduling
  - Correctness of System
  - Performance

<table>
<thead>
<tr>
<th>Check</th>
<th>Specification Ref</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coherent_resp_isshared_check</td>
<td>C6.4 Transaction responses from the interconnect 10. If WasUnique was not asserted for any snoop response received by the interconnect then, - If any snoop responses had IsShared asserted then, IsShared must be asserted in the transaction response to the initiating master</td>
<td>Checks that the IsShared response to initiating master is correct</td>
</tr>
<tr>
<td>snoop_resp_wasunique_check</td>
<td>C1.2.3 Cache State Rules: A line in a Unique State must be only in one cache</td>
<td>Checks that no two responses to a snoop transaction have the WasUnique(CRRESP[4]) bit asserted.</td>
</tr>
</tbody>
</table>
Reusability Aspects

• Hierarchical and Distributed Phasing
  ▪ Allows different ACE components to go out of phase
    ▪ Use of UVM domains enables to mimic independent flows in a single simulation
  ▪ Relevant for Power aware components: allows ‘catching up’ when Restored from a Powered down state
    ▪ UVM phase jumps, sync/unsync mechanism aids

• Coverage Model shaped by Current Configuration

• Enables Debug at different levels of abstraction
  ▪ Logging, recording etc....
UVM Enabled Debug of a Cache Coherent System

- Browse Protocol Transaction Activity
- View transcript files
- View Docs and Class Reference Hierarchy
- Quickly identify problems and causes
SUMMARY

• Protocol extensions like ACE help meet the increasing processing requirements
• Added Complexity brings advanced challenges in Verification
• Verification Methodologies provide the framework to design environments to address these challenges
• Effective Verification Planning Methodology and native System Verilog VIP key for success in such efforts
• Stimulus Generation Infrastructure, UVM resource Mechanism, Distributed Phasing can be harnessed for best results