Using Test-IP Based Verification Techniques in a UVM Environment

Vidya Bellippady
Microsemi Corporation
San Jose, CA

Sundar Haran
Microsemi Corporation
Hyderabad, India

Jay O'Donnell
Mentor Graphics
Seattle, WA

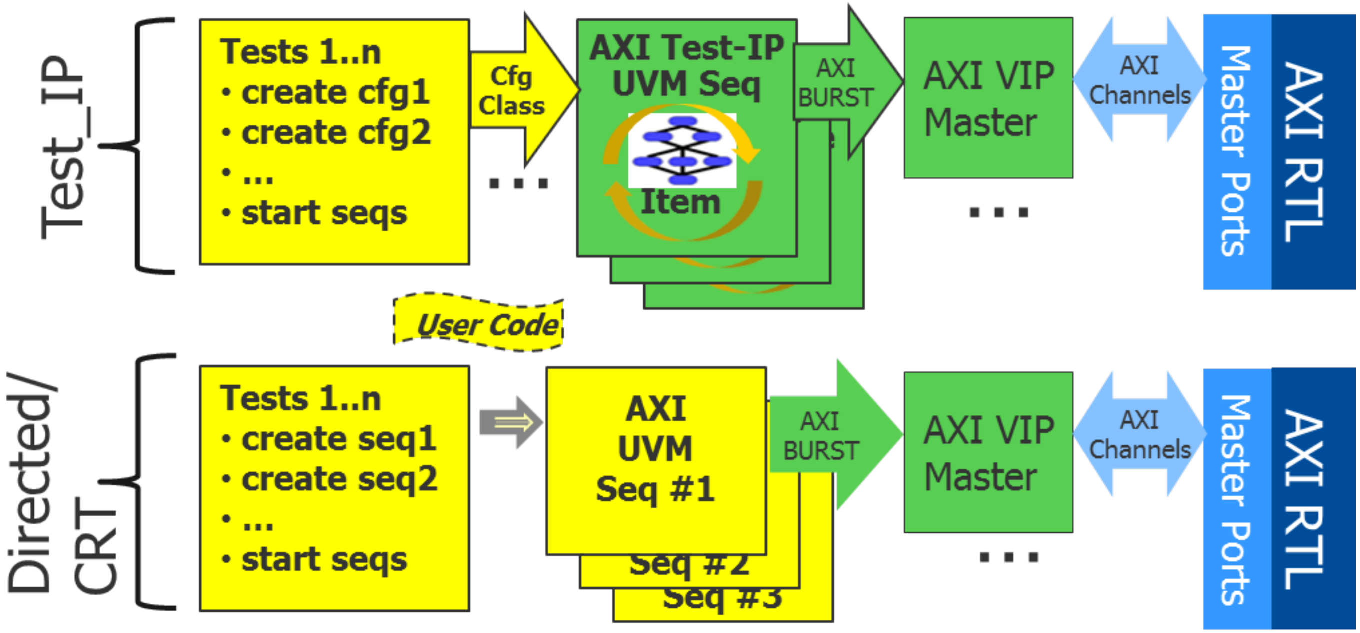
Directed & CRT Test Limitations Driving Test-IP Development

- Must know low-level VIP & bus protocol to write
- Directed sequences time-consuming to write
- CRT sequences difficult to constrain to application
  - Many seeds/sims may be req'd to hit cases
  - Lots of redundant CRT sequence code
  - Many apps not compatible w/CRT so Directed only option
- Significant user effort to write sequences
- Many simulations/tests to run to reach coverage

Test-IP Benefits compared to CRT and Directed Methods

- Shortens and simplifies test development
  - No need to know VIP & protocol details
  - No need to write sequences, simply write UVM tests
  - Test-IP tests described in a simple UVM config class
- Hits coverage goals in fewer simulations in less time
  - Leverages Test-IP graph-based stimulus targeting

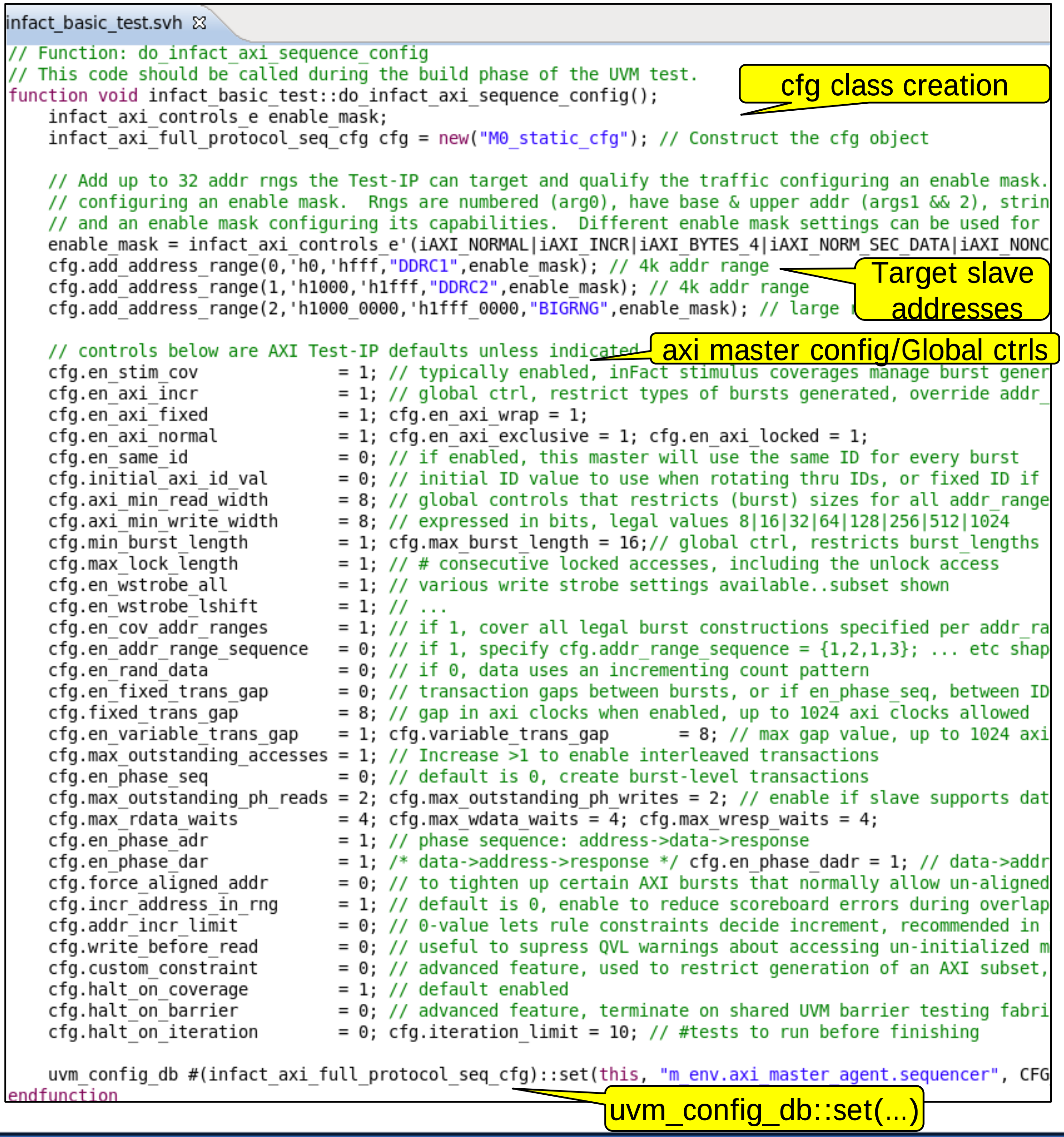
Test-IP Architecture vs DT/CRT



- Test-IP implemented as a UVM sequence
  - Sequence code never changes, simply instantiate
  - Internally leverages graph-based technology (inFact)
  - Behaves like a black box, no need to understand internals
- Test-IP behavior specified in Cfg class
  - Various integral controls (~50) and address range specifiers

Test-IP Configuration

- Create in UVM test
  - During build phase
  - Collect in a cfg class
  - Register w/config_db
- Cfg controls include:
  - Address map, 1..32 rngs
  - ~50 global controls
- Common Cfg architecture
  - Across AXI|AXI4|AHB|ACE
  - Similar features
  - AXI & AHB variant support
- User requirements
  - Learn controls, see docs
  - No inFact knowledge reqd
  - No VIP knowledge reqd



```
// Function: do_infact_axi_sequence_config
// This code should be called during the build phase of the UVM test.
function void infact_basic_test::do_infact_axi_sequence_config();
    infact_axi_controls_e enable_mask;
    infact_axi_full_protocol_seq_cfg cfg = new("M0_static_cfg"); // Construct the cfg object

    // Add up to 32 addr rngs the Test-IP can target and qualify the traffic configuring an enable mask.
    // configuring an enable mask.  Rngs are numbered (arg0), have base & upper addr (args1 && 2), strin
    // and an enable mask configuring its capabilities.  Different enable mask settings can be used for
    enable_mask = infact_axi_controls_e'(iAXI_NORMAL|iAXI_INCR|iAXI_BYTES_4|iAXI_NORM_SEC_DATA|iAXI_NONC
    cfg.add_address_range(0,'h0,'hfff,"DDRC1",enable_mask); // 4k addr range
    cfg.add_address_range(1,'h1000,'h1fff,"DDRC2",enable_mask); // 4k addr range
    cfg.add_address_range(2,'h1000_0000,'h1fff_0000,"BIGRNG",enable_mask); // large

    // controls below are AXI Test-IP defaults unless indicated
    cfg.en_stim_cov               = 1; // typically enabled, inFact stimulus coverages manage burst gener
    cfg.en_axi_incr               = 1; // global ctrl, restrict types of bursts generated, override addr_
    cfg.en_axi_fixed              = 1; cfg.en_axi_wrap = 1;
    cfg.en_axi_normal             = 1; cfg.en_axi_exclusive = 1; cfg.en_axi_locked = 1;
    cfg.en_same_id                = 0; // if enabled, this master will use the same ID for every burst
    cfg.initial_axi_id_val        = 0; // initial ID value to use when rotating thru IDs, or fixed ID if
    cfg.axi_min_read_width        = 8; // global controls that restricts (burst) sizes for all addr_range
    cfg.axi_min_write_width       = 8; // expressed in bits, legal values 8|16|32|64|128|256|512|1024
    cfg.min_burst_length          = 1; cfg.max_burst_length = 16;// global ctrl, restricts burst_lengths
    cfg.max_lock_length           = 1; // # consecutive locked accesses, including the unlock access
    cfg.en_wstrobe_all            = 1; // various write strobe settings available..subset shown
    cfg.en_wstrobe_lshift         = 1; // ...
    cfg.en_cov_addr_ranges        = 1; // if 1, cover all legal burst constructions specified per addr_ra
    cfg.en_addr_range_sequence    = 0; // if 1, specify cfg.addr_range_sequence = {1,2,1,3}; ... etc shap
    cfg.en_rand_data              = 0; // if 0, data uses an incrementing count pattern
    cfg.en_fixed_trans_gap        = 0; // transaction gaps between bursts, or if en_phase_seq, between ID
    cfg.fixed_trans_gap           = 8; // gap in axi clocks when enabled, up to 1024 axi clocks allowed
    cfg.en_variable_trans_gap     = 1; cfg.variable_trans_gap       = 8; // max gap value, up to 1024 axi
    cfg.max_outstanding_accesses = 1; // Increase >1 to enable interleaved transactions
    cfg.en_phase_seq              = 0; // default is 0, create burst-level transactions
    cfg.max_outstanding_ph_reads = 2; cfg.max_outstanding_ph_writes = 2; // enable if slave supports dat
    cfg.max_rdata_waits           = 4; cfg.max_wdata_waits = 4; cfg.max_wresp_waits = 4;
    cfg.en_phase_adr              = 1; // phase sequence: address->data->response
    cfg.en_phase_dar              = 1; /* data->address->response */ cfg.en_phase_dadr = 1; // data->addr
    cfg.force_aligned_addr        = 0; // to tighten up certain AXI bursts that normally allow un-aligned
    cfg.incr_address_in_rng       = 1; // default is 0, enable to reduce scoreboard errors during overlap
    cfg.addr_incr_limit           = 0; // 0-value lets rule constraints decide increment, recommended in
    cfg.write_before_read         = 0; // useful to supress QVL warnings about accessing un-initialized m
    cfg.custom_constraint         = 0; // advanced feature, used to restrict generation of an AXI subset,
    cfg.halt_on_coverage          = 1; // default enabled
    cfg.halt_on_barrier           = 0; // advanced feature, terminate on shared UVM barrier testing fabri
    cfg.halt_on_iteration         = 0; cfg.iteration_limit = 10; // #tests to run before finishing

    uvm_config_db #(infact_axi_full_protocol_seq_cfg)::set(this, "m_env.axi_master_agent.sequencer", CFG
endfunction
```

cfg class creation

Target slave addresses

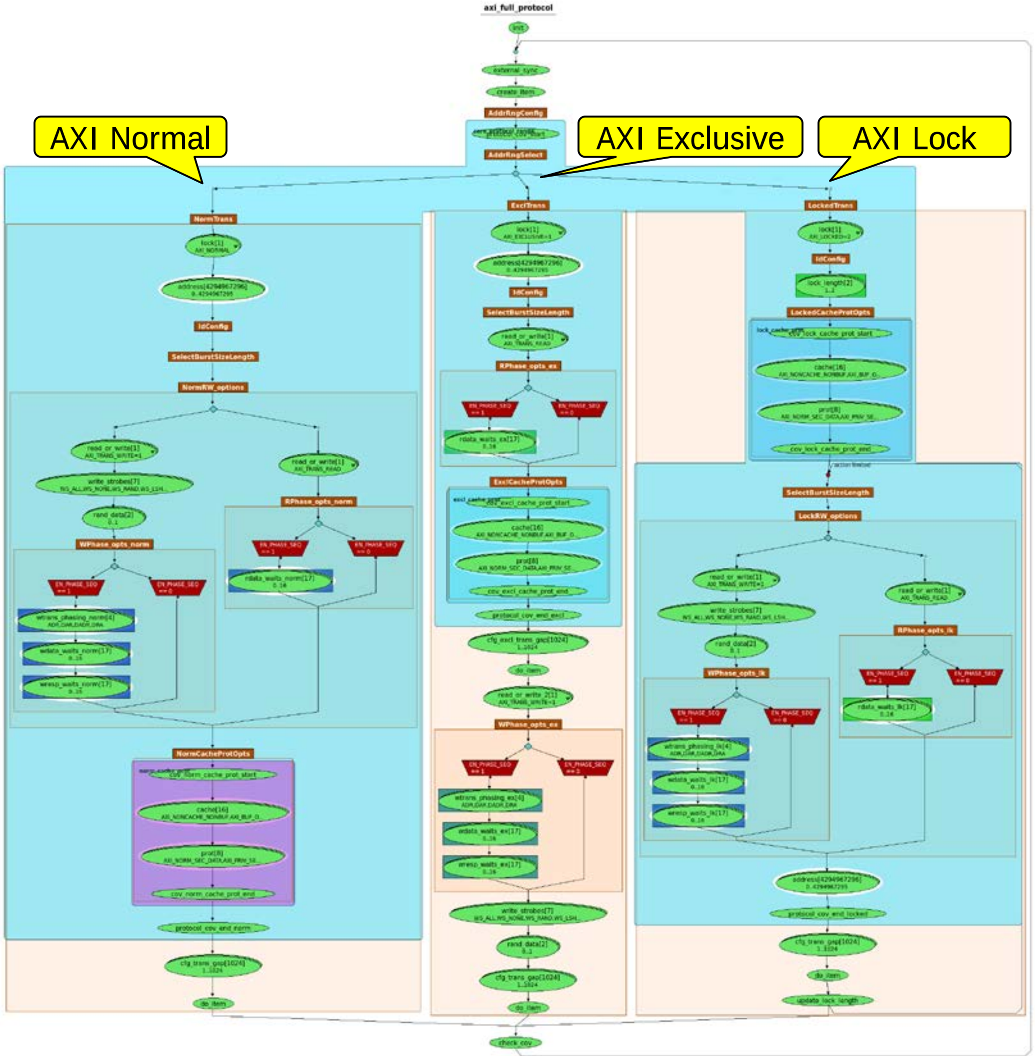
axi master config/Global ctrls

uvm_config_db::set(...)

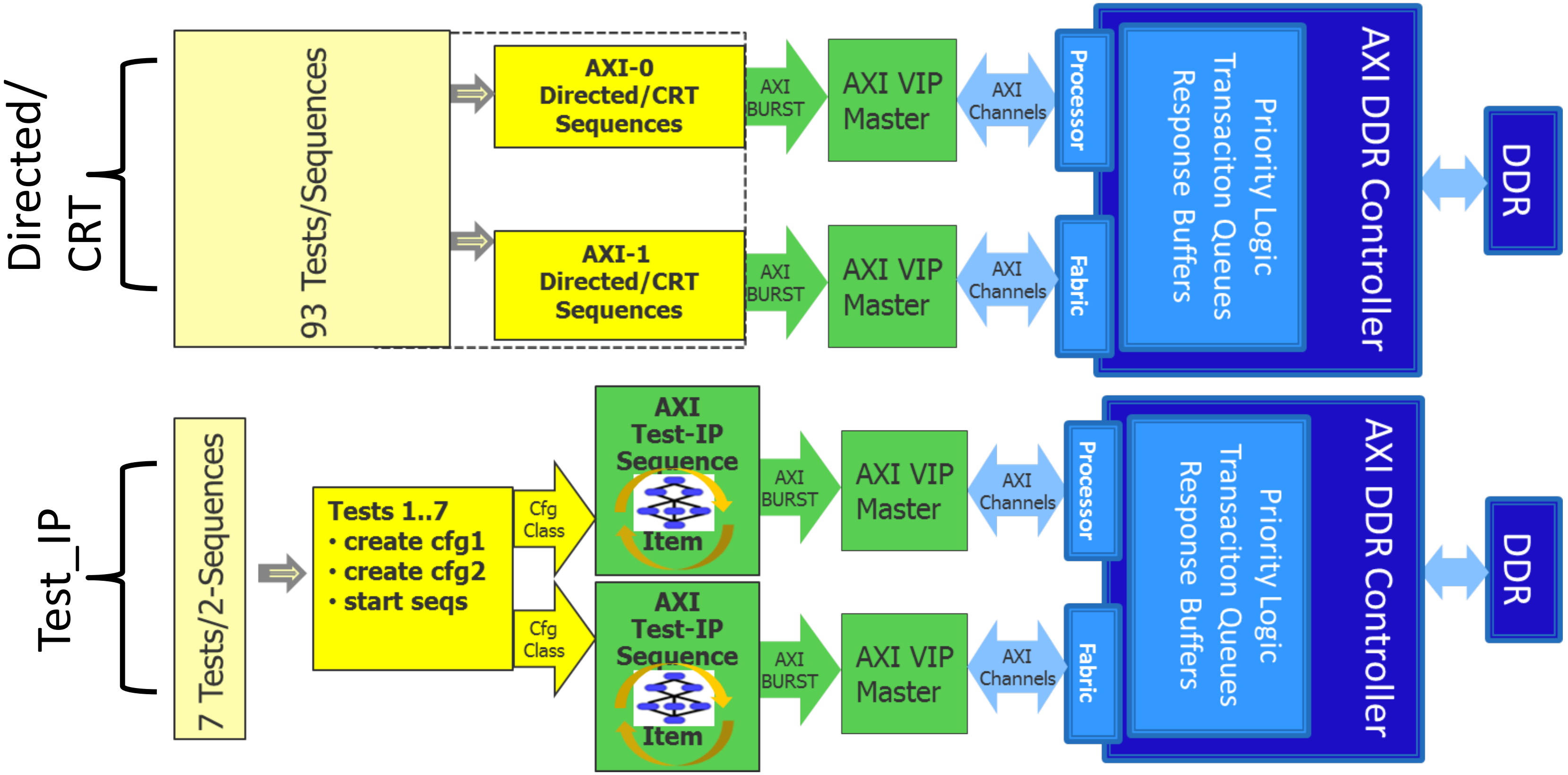
Test-IP Implementation Details

… AXI Example, if you want to know

- Graph reads cfg info
  - Adjusts graph based on cfg
- All burst options configured
  - Atomic and burst branches
    - Normal|Exclusvie|Lock
  - Burst size/length/cache/prot
  - Write strobes/ID selection
  - Addresses in range(s)
  - Data incrementing or random
- Phase-level option
  - ADR|DAR|DADR phases, waits
  - Multiple outstanding rd&wr
  - Out-of-order rd&wr
- Stimulus coverages
  - Highlighted on graph
  - Graph traversed to meet goals
  - Size adapts based on cfg



Using Test-IP to verify an AXI DDR Controller – Before/After Results



Metric	CRT/ Directed	Test-IP Approach	Benefit
#lines user testbench code	40,000	850	47x less
#OVM tests	93	7	13x fewer
Simulation time to coverage	17hrs	15min	68x faster

Test-IP Applications

- AMBA fabrics (ahb|axi|axi4|ace.. available)
- AXI DDR controllers, routers, switches
- Test-IP supporting AMBA slaves possible
- Other bus protocols where traffic important
  - Pcie, usb, …
- Application-Specific data generation
  - Currently random or incrementing payloads
  - Could be enhanced to be data-driven (TLM fifo)
  - Could be layered under higher-level sequences

Findings/Lessons Learned Developing and Using Test-IP

- Multiple successes with AMBA fabrics/devices
- Enhancements – developed
  - Concurrent interleaved bursts, same master
  - Sequential address accesses
- Enhancements -- identified
  - Multi-master accesses to same addr avoidance
  - Profile-based traffic generation
  - Test-IP implemented to model bus slaves
- Should be treated like a software product
  - Releases, bug fixes, docs, examples, support…
  - Features and applications will evolve with use