# Title: Using Test-IP Based Verification Techniques in a UVM Environment


| Vidya Bellippady | Sundar Haran | Jay O'Donnell |
|---|---|---|
| Microsemi Corporation | Microsemi Corporation | Mentor Graphics |
| San Jose, CA | Hyderabad, India | Seattle, WA |



## Abstract:

This paper describes a new verification technique using Test-IP, which are pre-built UVM test sequences implemented using a combination of directed, intelligent testbench (iTBA), and random methods. Test-IP converts an abstract test description defined in the UVM test into a series of protocol-specific burst sequence items passed to a standard verification-IP driver. This paper describes why the technique was first developed for AXI bus fabric applications and references a case-study where it was used to verify a 2-port AXI DDR controller.

The Test-IP approach differs from traditional approaches used with verification IP in the following ways:

- Eliminates user requirements to understand how the verification IP works when implementing tests. The user writes simple UVM tests
- Each UVM test populates a simple configuration (cfg) class specifying the type of bus traffic to be generated
- The cfg class contains user-defined address ranges/properties for peripheral addresses in the system. A set of cfg-class controls defining bus master agent capabilities are also provided
- Test-IP reads the cfg and generates traffic using intelligent testbench (iTBA) graph-based methods. iTBA graphs target stimulus combinations inferred by the cfg class which can be validated using traditional functional coverage metrics
- Supports optional generation of sequential address accesses for applications needing a more directed test approach
- Supports optional random selection of various protocol fields for use in constrained-random (CRT) applications

This technique vastly simplifies the test development process and achieves equivalent or superior functional coverage results in a fraction of the simulation time, and has applications in other verification work including bus fabric verification and performance profiling. Similar test-IP components have been implemented supporting the full set of AMBA protocols including AHB, AXI4, and ACE and used in related applications in both OVM and UVM environments.


## Introduction

Test-IP was first developed for an AMBA bus fabric application where the prior approach used a large number of directed sequences with limited functional coverage metrics. Test effectiveness was limited because it relied entirely on the user's capacity to write enough sequences without adequate feedback from functional coverage metrics. Test-IP supporting AXI was developed to address both the test capacity and test effectiveness problems.

Figure 1 shows a typical UVM fabric application using traditional directed sequences targeting AXI slaves in the system, with user-developed code highlighted in yellow. User directed sequences construct AXI bursts targeting various slaves required different addressing and burst construction depending on slave design. Unique sequences were needed for each master because different masters typically have different slave connectivity on the fabric. Each master would typically manage multiple outstanding read and write bursts which could interleave, further complicating the design of the sequences. Developing functional coverage to measure that each master accessed its target slaves generating the legal subset of AXI protocol supported by those slaves was too difficult to implement, leaving no effective means to assure tests were effective.

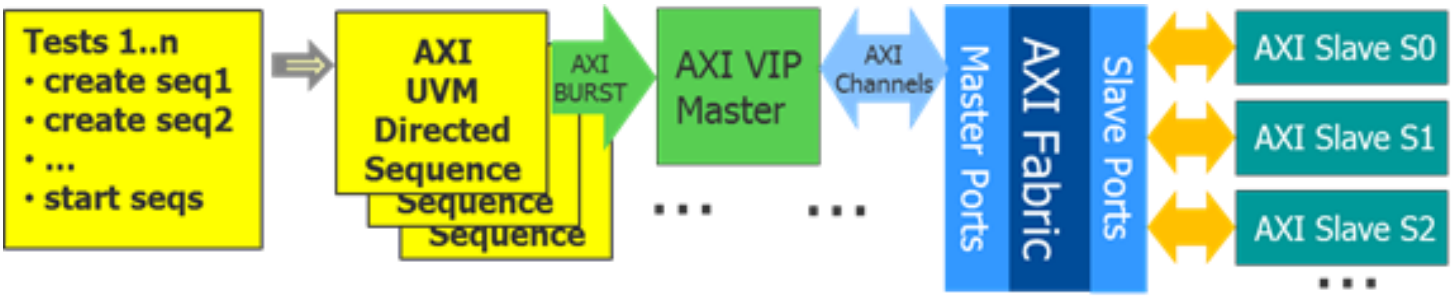


**Figure 1: Typical Directed Sequence Application where Test-IP could be applied**

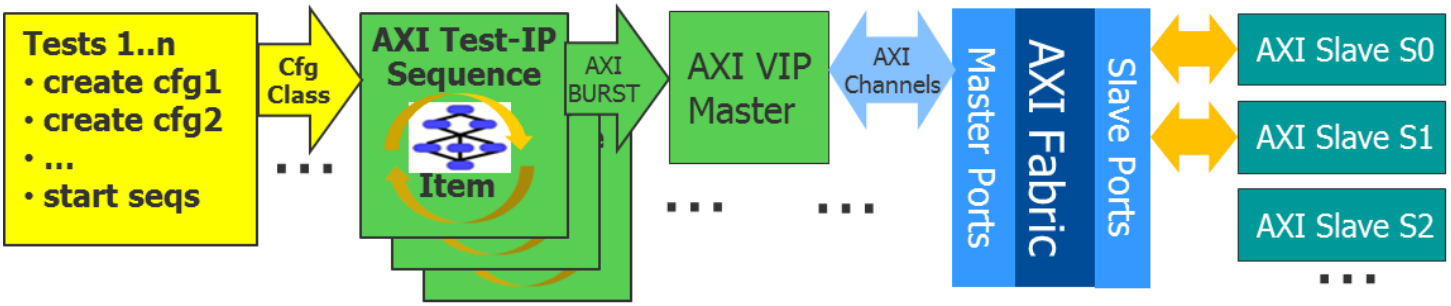


**Figure 2: Fabric Application with Test-IP applied**

Figure 2 shows the same application, but using Test-IP. The key elements of the Test-IP approach are:

- The Test-IP sequence never changes
- Behavior of each Test-IP sequence is controlled by a user-developed configuration class (cfg)
- Users specify cfg class controls identifying addresses for slaves to target including per-slave restrictions on the type of bursts generated
- Users specify cfg class controls managing how its associated master operate including burst interleave behavior, transaction gaps between bursts, stimulus coverage targeting, random or incrementing data payloads, and sequence termination controls

Benefits of the Test-IP approach in this application included:

- Significant reduction in the amount and complexity of test code to develop. The user only needed to write simple cfg classes specifying how the associated Test-IP sequence operates
- Eliminated requirements to understand VIP internals as they were handled by Test-IP
- Automatic generation of stimulus functional coverage metrics to measure test effectiveness built-in to Test-IP

The balance of the paper describes Test-IP configuration class (cfg) controls and how Test-IP is implemented. An application of Test-IP for verifying a 2-port AXI DDR controller is then described, comparing before and after results. Lessons learned from this work and prospects for applying the Test-IP approach for other bus protocols and applications are discussed in the summary.

## Test-IP Configuration Class Overview

The user-interface to control Test-IP is a configuration class (cfg) constructed during the build phase of a UVM test. The usage shown in Figure 3 is typical. Users of Test-IP only need to understand the cfg controls to use Test-IP. Each cfg class instance is registered with the uvm_config_db and looked-up by an associatedTest-IP sequence instance using a uvm_config_db “get” call.

infact_basic_test.svh

```
// 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), string name,
    // and an enable mask configuring its capabilities.  Different enable mask settings can be used for different ranges.
    enable_mask = infact_axi_controls_e'(iAXI_NORMAL|iAXI_INCR|iAXI_BYTES_4|iAXI_NORM_SEC_DATA|iAXI_NONCACHE_NONBUF);
    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 range

    // controls below are AXI Test-IP defaults unless indicated
    cfg.en_stim_cov              = 1; // typically enabled, inFact stimulus coverages manage burst generation
    cfg.en_axi_incr              = 1; // global ctrl, restrict types of bursts generated, override addr_range settings
    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 en_same_id
    cfg.axi_min_read_width       = 8; // global controls that restricts (burst) sizes for all addr_ranges
    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 for all addr_range
    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_range
    cfg.en_addr_range_sequence   = 0; // if 1, specify cfg.addr_range_sequence = {1,2,1,3}; ... etc shape burst traffic
    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 accesses
    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 clocks allowed
    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 data interleaving
    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->address->data->respons
    cfg.force_aligned_addr       = 0; // to tighten up certain AXI bursts that normally allow un-aligned bursts
    cfg.incr_address_in_rng      = 1; // default is 0, enable to reduce scoreboard errors during overlapping bursts
    cfg.addr_incr_limit          = 0; // 0-value lets rule constraints decide increment, recommended in most cases
    cfg.write_before_read        = 0; // useful to supress QVL warnings about accessing un-initialized memory
    cfg.custom_constraint        = 0; // advanced feature, used to restrict generation of an AXI subset, ref by #
    cfg.halt_on_coverage         = 1; // default enabled
    cfg.halt_on_barrier          = 0; // advanced feature, terminate on shared UVM barrier testing fabrics
    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_NAME, cfg);
endfunction
```

**Figure 3: Test-IP configuration class construction with typical fields shown**

There are three types of cfg fields which users specify when setting up a test:

- Address range specification (min 1, max 32, corresponds to AXI addresses the Master will target)
- AXI Master control fields (~50 fields, configure the overall behavior of the AXI Master)
- uvm_config_db::set() call to register the cfg class with the uvm configuration database

### Test-IP cfg Class Address Specification Controls

Cfg class address ranges identify addresses the Test-IP targets during simulation.  These address ranges are typically associated with specific slave peripherals in the system, or regions of memory to target.  A typical address range specification block is shown in Figure 4:

*infact_phase2_test.svh

```
function void do_infact_axi_sequence_config();
  infact_axi_full_protocol_seq_cfg cfg;
  infact_axi_controls_e enable_mask;
  infact_axi_controls_e iAXI_SIZES_1248 = infact_axi_controls_e'(iAXI_BYTES_1|iAXI_BYTES_2|iAXI_BYTES_4|iAXI_BYTES_8);
  cfg = new("M0_static_cfg");

  enable_mask = infact_axi_controls_e'(iAXI_NORMAL|iAXI_WRAP|iAXI_SIZES_1248|iAXI_NORM_SEC_DATA|iAXI_NONCACHE_NONBUF);
  cfg.add_address_range(1,'hf00,'h13ff,"DDRC1",enable_mask); // straddles 4k bdy
  enable_mask = infact_axi_controls_e'(iAXI_NORMAL|iAXI_INCR|iAXI_SIZE_ALL|iAXI_NORM_SEC_DATA|iAXI_NONCACHE_NONBUF);
  cfg.add_address_range(2,'hE00,'hfff, "EMAC", enable_mask);
  enable_mask = infact_axi_controls_e'(iAXI_EXCLUSIVE|iAXI_NORMAL|iAXI_INCR|iAXI_BYTES_4|iAXI_NORM_SEC_DATA|iAXI_NONCACHE_NONBUF);
  cfg.add_address_range(3,'h20000,'h20fff,"SMRAM",enable_mask);
  enable_mask = infact_axi_controls_e'(iAXI_NORMAL|iAXI_INCR|iAXI_BYTES_4|iAXI_NORM_SEC_DATA|iAXI_NONCACHE_NONBUF|iAXI_WO);
  cfg.add_address_range(4,'h30000,'h30fff,"SMRAM_WO",enable_mask);
  enable_mask = infact_axi_controls_e'(iAXI_NORMAL|iAXI_INCR|iAXI_BYTES_4|iAXI_NORM_SEC_DATA|iAXI_NONCACHE_NONBUF|iAXI_RO);
  cfg.add_address_range(5,'h40000,'h40fff,"SMRAM_RO",enable_mask);

  cfg.addr_range_sequence      = {1,2,3,4,4,4,4,5,5,5,5,4,5,4,5};
```

**Figure 4: Test-IP cfg Class Address Range Qualifiers**

In this example address ranges 1..5 are added to the cfg class, each range specifying a start and end addresses, and range-specific qualifiers encoded in an enable_mask. There are no restrictions on address overlapping across multiple ranges. Various enable_mask qualifiers may be specified to restrict the types of accesses in the respective range:

- Access type: For AXI these are in the set AXI_NORMAL|EXCLUSIVE|LOCKED
- Burst types: For AXI these are in the set AXI_INCR|FIXED|WRAP
- Bus size: For AXI, the size in bytes of the R|W channels and may allow narrow transfers
- Cache and Protection qualifiers: AXI-specific
- Read-only or Write-only qualifiers
- Optional ID associated with specific range accesses

The example also specifies an optional “addr_range_sequence” which restricts address selection to a sequence of address_ranges for applications needing sequential bus accesses. This feature enables generation of sequential reads or writes to one or more slave targets:

- cfg.addr_range_sequence= {1,2,3,4,4,4,4,5,5,5,5,4,5,4,5};

Optional control:

- cfg.incr_address_in_rng = 1;

enables an automatic address increment logic to increment up through each address range address space. This control is useful in fabric verification as it avoids overlapping in-flight accesses targeting the same address, which might occur in a CRT testbench that uses random address selection.

### Test-IP cfg Class Global Controls

Remaining cfg class controls specify how bursts are constructed by the associated bus master agent. Controls apply to all address ranges. Figure 5 shows some common controls for AXI:

*infact_phase2_test.svh

```
    // global controls that apply to all addr_ranges..some defaults shown for for clarity
    cfg.en_axi_normal           = 1; cfg.en_axi_exclusive        = 1; cfg.en_axi_locked          = 1;
    cfg.en_axi_incr             = 1; cfg.en_axi_fixed            = 1; cfg.en_axi_wrap            = 1;

    cfg.en_wstrobe_all          = 1; cfg.en_wstrobe_none         = 0; cfg.en_wstrobe_rand        = 0;
    cfg.en_wstrobe_lshift       = 1; cfg.en_wstrobe_lshift_fill  = 1; cfg.en_wstrobe_rshift      = 1;
    cfg.min_burst_length        = 1; cfg.max_burst_length        = 16;
    cfg.axi_min_read_width      = 8; cfg.axi_min_write_width     = 8;
    cfg.max_lock_length         = 1;
    cfg.en_same_id              = 0; cfg.initial_axi_id_val      = 0;

    cfg.en_stim_cov             = 1; cfg.en_wrap_bdy_cov         = 0; cfg.en_cov_addr_ranges     = 1;
    cfg.halt_on_coverage        = 1; cfg.halt_on_iteration       = 1; cfg.iteration_limit        = 500;

    cfg.en_rand_data            = 0; cfg.en_fixed_trans_gap      = 1; cfg.fixed_trans_gap        = 8;
    cfg.en_variable_trans_gap   = 0; cfg.variable_trans_gap      = 8; cfg.min_trans_gap          = 1;

    cfg.force_aligned_addr      = 0;
    cfg.incr_address_in_rng     = 1; cfg.addr_incr_limit         = 0; cfg.write_before_read      = 0;
    cfg.cov_report_interval     = 5; cfg.print_transaction       = 1;

    cfg.en_phase_seq            = 1; cfg.max_outstanding_accesses  = 4;
    cfg.max_outstanding_ph_reads = 2; cfg.max_outstanding_ph_writes = 2;
    cfg.max_wdata_waits         = 4; cfg.max_rdata_waits          = 4; cfg.max_wresp_waits        = 4;
    cfg.en_phase_adr            = 1; cfg.en_phase_dar             = 1; cfg.en_phase_dadr          = 1;
    cfg.addr_range_sequence     = {1,2,3,4,4,4,4,5,5,5,5,4,5,4,5};
```

**Figure 5: Test-IP cfg Class Global Controls for AXI Test-IP**

The set of controls include bus-protocol-specific controls, and general controls that determine how the associated UVM Test-IP sequence operates:

- Burst attributes including alignment, write strobes, data
- Low-level burst construction controls (AXI channel (i.e. phase) controls, channel interleaving)
- Termination controls and transaction gap
- Address range sequencing, write-before-read option

Current Test-IP implementation supports random or incrementing data selection using control en_rand_data. Access to external data in a TLM fifo could be implemented as an enhancement.

### Test-IP cfg class uvm_config_db assignment

The user-constructed cfg class is put into the uvm_config_db using the syntax shown in Figure 6. The associated Test-IP sequence instance will locate this cfg entry in the uvm_config_db when the sequence is started during simulation.

infact_coverage_test.svh

```
    cfg.en_infact_summary_cg     = 1; // generates a UCDB covergroup-displays inFact stimulus coverage results
    cfg.instnum_in_instname      = 0; // generates INFACT_1..INFACT_2..INFACT_3... uvm_report_info msgs, useful with fabrics
    cfg.print_transaction        = 0; // default is 0, enable for debug

    // add config to the config system so the inFact AXI full protocol sequence
    // instance can access.
    uvm_config_db #(infact_axi_full_protocol_seq_cfg)::set(this, "m_env.axi_master_agent.sequencer", infact_axi_full_protocol_seq_cfg::CFG_NAME, cfg);
endfunction
```

**Figure 6: Test-IP uvm_config_db::set call used with Test-IP cfg classes**

Test-IP also supports direct assignment of cfg class instances to their respective sequences for applications where sequences are created and started in a UVM virtual sequence.

## Test-IP Internal Implementation

Users of Test-IP need only understand the controls provided in the cfg class, while Test-IP developers will need to understand the design of the Test-IP sequence and the verification IP it controls. The main elements of Test-IP are shown in Figure 7:

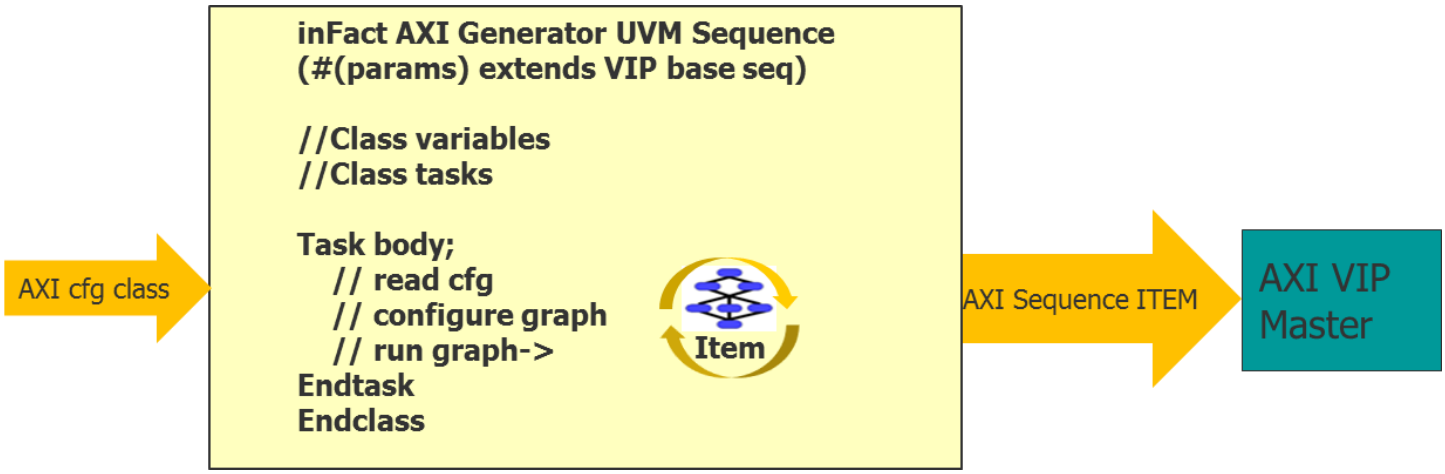


**Figure 7: Test-IP Sequence Implementation**

**Cfg-class Design:**

Each cfg class extends from uvm_object and contains cfg variables and related functions specific to the VIP bus protocol it supports. A number of common cfg variables and functions are independent of the VIP protocol and form a common set of controls to simplify usage in applications using multiple types of Test-IP. The protocol-specific cfg class contains functions to access the config_db, check functions to validate cfg class definitions are consistent, and routines to print cfg info to the transcript.

**Sequence Design:**

Test-IP Sequences are UVM sequences that internally call an inFact graph that populates VIP sequence item fields based on settings provided in the user's cfg class. The inFact graph is protocol-specific and understands the various burst construction options and fields, along with the user-defined address ranges being targeted. Figure 8 shows the inFact graph used inside AXI Test-IP:

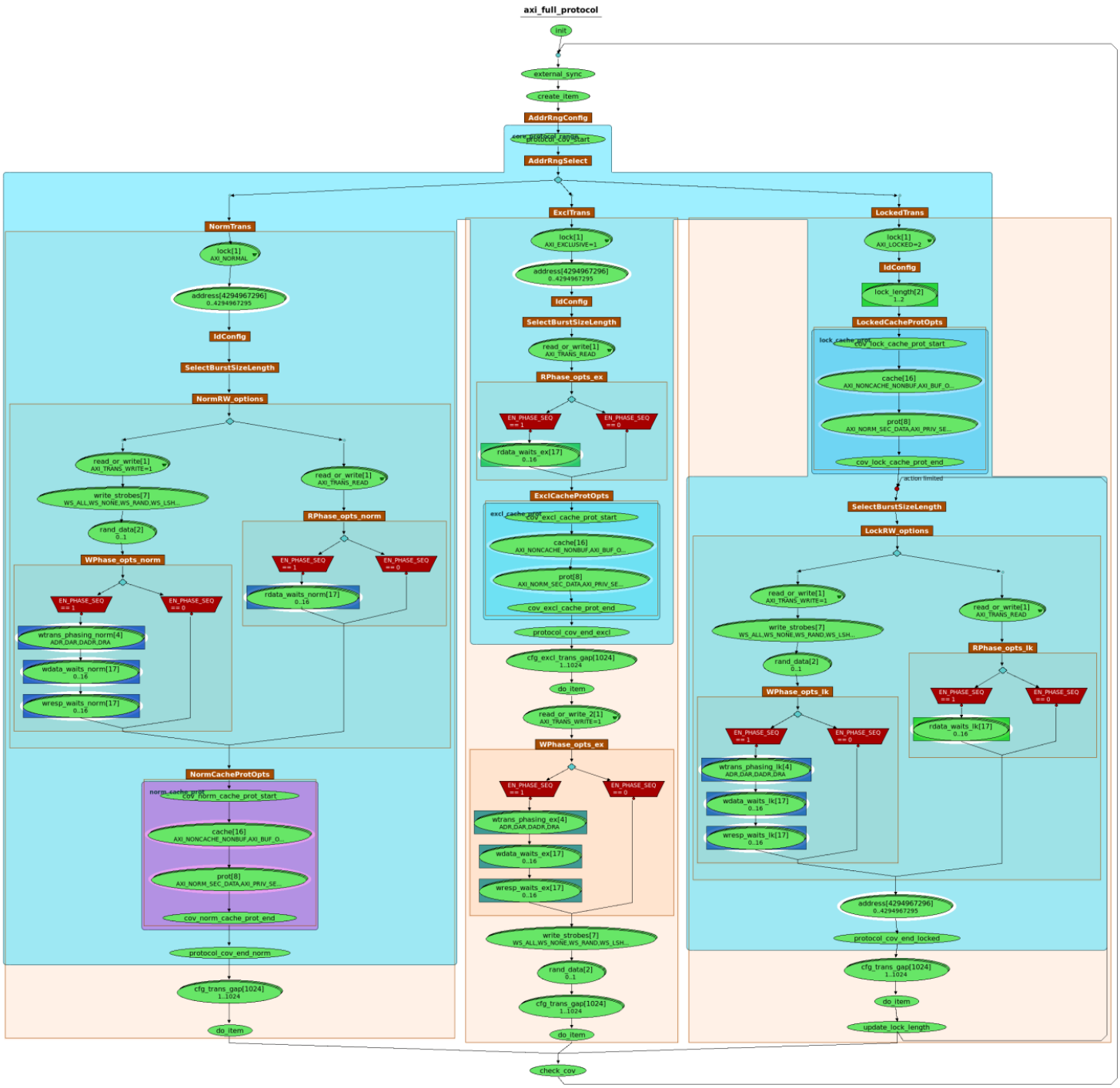


**Figure 8: AXI Test-IP Graph Example**

Some of the important aspects of this graph design include:

- Address range selection options
- Three main vertical branches corresponding to AXI Normal|Exclusive|Locked accesses
- Read or Write sub-branches
- Optional sub-branches to configure channel-level phasing and delays
- Logic to insert transaction gaps
- Various algebraic constraints (not shown) that:
    - Enable|disable different graph branches based on user cfg class settings
    - Control burst construction based on user cfg class settings

    - Implement AXI-specific address alignment based on burst type
    - Implement address increment logic if specified in the cfg
    - Restrict burst construction based on cfg class address_range enable_mask attributes
- Various color-coded stimulus coverage regions that inFact algorithms target during simulation when user cfg controls enable stimulus coverage

The inFact Test-IP sequence and underlying graph are configurable during simulation based on the user-specified cfg controls, and any VIP parameterization for (AXI) bus widths and related attributes. This means that each Test-IP sequence instance is likely to be unique even though the Test-IP sequence code never changes. Another important design attribute of the Test-IP is the built-in stimulus coverage provided by the underlying inFact technology. This stimulus coverage assures that all slave peripherals identified in the cfg class address_map are targeted with all possible burst constructions. Results from these stimulus coverages are saved in a covergroup managed by the Test-IP sequence.

## Using Test-IP to Verify an AXI DDR Controller

An existing mostly directed OVM test environment having multiple sequences and functional coverage was retrofitted to use Test-IP. This enabled comparison of the two different approaches. Figure 9 shows the original test environment:

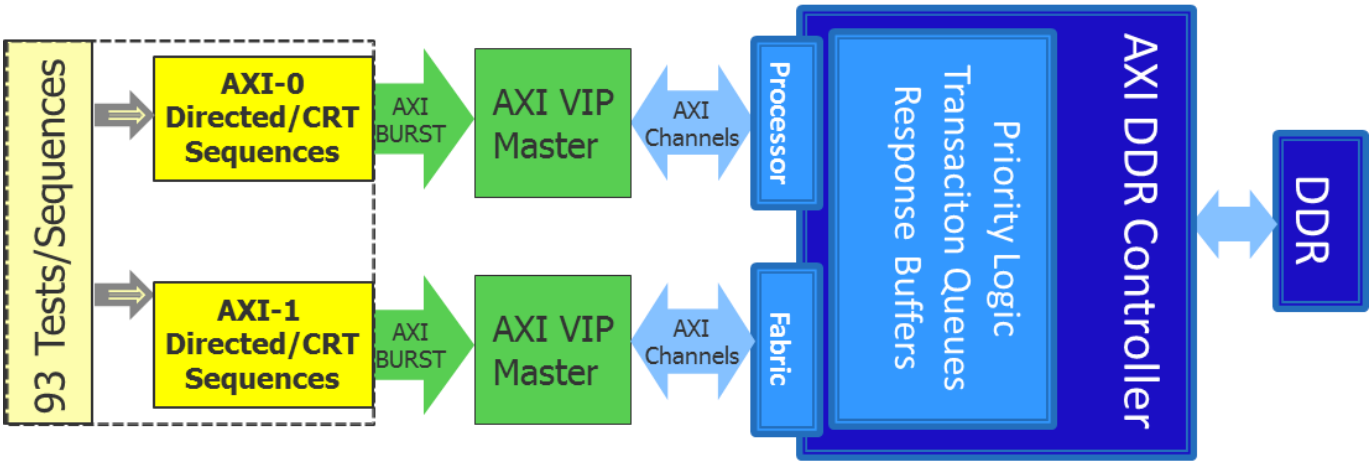


**Figure 9: Original Directed/CRT Testbench Architecture for the AXI DDR Controller**

The AXI DDR controller is used in a larger SOC design and provides DDR memory access to an internal processor on one interface, and various peripherals on the bus fabric interface. Some of the more difficult verification challenges involved testing of new logic added to a pre-existing single port controller design:

- Verifying priority-access control logic that re-orders queued memory accesses works properly, by generating sequential memory accesses on both ports that cause internal states to be hit.
- Verifying queue buffer logic is capable of handling all AXI transaction size variations presented on both ports
- Verifying the controller supports all specified AXI burst constructions and channel timing variations for each respective port, where the two ports have different capabilities

Various tests and sequences highlighted in yellow target functional coverage goals measured by functional coverage implemented as bus monitors provided by the VIP (not shown) and instrumented internally in the controller RTL (state machine transition coverage).

The original Directed/CRT testbench required a large number of tests and sequences to meet functional coverage goals due to a variety of issues:

- Requires detailed VIP knowledge to write test sequences
- Difficulty achieving functional coverage using CRT because highly-sequential bus traffic is needed, requiring many custom sequences and iteration across simulation seeds

- Difficulty implementing directed tests as complex procedural code is needed to construct sequential accesses on both controller ports

This original testbench and design were analyzed to determine the best way to deploy Test-IP and a test design identified containing 7 separate tests as shown in Figure 10:

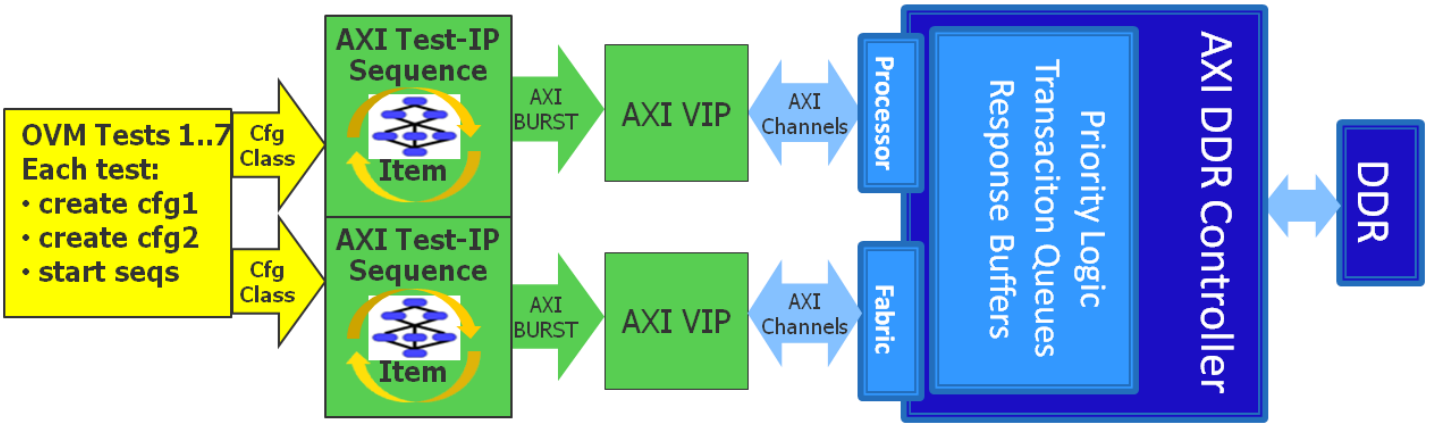


**Figure 10: Test-IP Testbench Architecture for the AXI DDR Controller**

Seven distinct tests were identified that were logically equivalent to the original set of 93 Directed/CRT tests as show in Figure 11. Each test was described by a pair of cfg classes constructed in the OVM Test that are passed to the AXI Test-IP sequence, which in-turn transforms it into a series of AXI bursts

| Test Name | Test Purpose & Design |
|---|---|
| Token Test | Verifies behavior w/multiple concurrent near-proximity reads. Implement using 5-interleaved back-to-back reads on fabric port w/varying AXI_IDs |
| Priority Test | Verifies priority given to high-priority AXI_ID using a mix of traffic on both ports, NORMAL & LOCK, reads & writes, non-adjacent addrs to avoid read combining |
| Exhaustive Test | Verifies all supported burst types , atomic modes, sizes, write strobes across a fixed set of AXI_IDs. |
| Slverr Test | Verifies controller responds to unsupported AXI_FIXED access with slverr response |
| Range Test | Verifies controller can access all memory pages generating burst that span large addr increments |
| R/W interleave Test | Verifies correct operation at the channel level for interleaved reads/writes on both ports with varying address/data/response timing |

**Figure 11: Test-IP Test Partitioning for the AXI DDR Controller**

Figure 12 shows the essential elements of the Token Test implementation to illustrate how AXI Test-IP is configured and used. All tests extend from a infact_base* class that defines common cfg settings used by all tests. The Token test constructs two cfg objects configured to generate memory accesses on the two AXI controller ports. For this test the axi0 port was kept largely inactive, while the axi1(fabric) port is accessed with an addr_range_sequence of back-to-back reads using a mix of fixed and varying AXI_IDs.

```
// inFact stand-alone token test
class axi0_1_infact_token_f extends axi0_1_infact_base_f;
    function void do_infact_axi0_sequence_config();
      infact_axi_controls_e en_mask0;
      en_mask0    = infact_axi_controls_e'(iAXI_NORMAL|iAXI_LOCKED|iAXI_INCR|iAXI_WRAP|iAXI_BYTES4_8|iAXI_COMMON_CPROT);

      // generate some activity on axi0 during token tests using address ranges that do not overlap with axi1 accesses
      m_cfg_axi0.add_address_range(1,'h6000,'h61ff,"AXI0_6000_61ff",en_mask0);

      // these cfgs keep axi0 minimally active during token tests and let the axi1 sequence
      // terminate the test when it is complete.  axi0 must generate at-least 10 bursts before releasing the barrier.
      m_cfg_axi0.en_axi_wrap             = 0; // NO wraps for token tests, to match Directed/CRT scheme
      m_cfg_axi0.en_stim_cov             = 0; // generator sequence driving AXI0 runs in random mode by default
      m_cfg_axi0.halt_on_coverage        = 0; // no, since running random
      m_cfg_axi0.halt_on_barrier         = 1;
      m_cfg_axi0.halt_on_iteration       = 1;
      m_cfg_axi0.iteration_limit         = 10;  // reduce goal since axi1 tends to get preferential access
      m_cfg_axi0.en_fixed_trans_gap      = 1;   // keep axi0 minimally active
      m_cfg_axi0.fixed_trans_gap         = 500; // keep axi0 largely inactive during token test segment
      set_config_object("env.axi_master_agent0.sequencer", cfg_t::CFG_NAME, m_cfg_axi0, 0);
    endfunction

    function void do_infact_axi1_sequence_config();
      infact_axi_controls_e en_mask1_ro,en_mask1_ro_id;

      // both ranges are RO, the *_id range uses a fixed AXI_ID while the other increments thru all possible AXI_ID values
      en_mask1_ro    = infact_axi_controls_e'(iAXI_LOCKED|iAXI_NORMAL|iAXI_INCR|iAXI_BYTES_8|iAXI_COMMON_CPROT|iAXI_RO);
      en_mask1_ro_id = infact_axi_full_protocol_seq_cfg::add_axi_id_to_mask(en_mask1_ro,4'b0010);

      // AXI1 ranges specific to token tests, maintaining (arbitrary) range numbers 7 & 8
      m_cfg_axi1.add_address_range(7,'h5030,'h51ff,"AXI1_5000_51ff_ro_id",en_mask1_ro_id); // read-only,  fixed id
      m_cfg_axi1.add_address_range(8,'h5000,'h51ff,"AXI1_5000_51ff_ro"   ,en_mask1_ro);    // read-only,  unconstrained id

      // two read-only addr regions with fixed and variable ID settings used in the sequence sent to the axi1 agent
      m_cfg_axi1.en_addr_range_sequence = 1;
      m_cfg_axi1.addr_range_sequence = {8,7,8,7,8,7,8,8,7,7}; // this one most effective, reads only, repeats 2000x

      // these cfgs keep axi1 active during token tests
      m_cfg_axi1.en_axi_wrap             = 0; // NO wraps for token tests, to Directed/CRT
      m_cfg_axi1.en_stim_cov             = 0; // token tests controlled by address range sequences, not protocol coverage
      m_cfg_axi1.halt_on_coverage        = 0;
      m_cfg_axi1.halt_on_barrier         = 1; // axi0 also shares this barrier
      m_cfg_axi1.halt_on_iteration       = 1;
      m_cfg_axi1.iteration_limit         = 2000; // run until iteration reached and barrier shared with axi0 released
      m_cfg_axi1.max_outstanding_accesses = 5; // critical to get coverage
    endfunction
endclass
```

**Figure 12: Test-IP Test Implementation for the AXI DDR Token Test**

## Results Using Test-IP to Verify an AXI DDR Controller

| 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 |

The test-IP approach offered some key benefits over CRT/Directed methods in this application

- Leverages pre-built test-IP sequences … less code to write
- Simple test design reduces the amount of user-code by >40x
- Corner cases easily hit without any requirement to re-run simulations at different seeds, resulting in an overall reduction in simulation time to reach equivalent CRT coverage by 68x
- Supports stimulus coverage targeting for variables and cross combinations using iTBA methods built-in to the test-IP
- Option to generate random bursts constrained by cfg class settings
- Supports “directed style” tests where address sequences can be specified having important attributes needed to hit coverage cases, ie: read@id1, write@id2, read@id3... and so on..

## Findings and Conclusion

Test-IP has the potential to significantly reduce the amount of work verification engineers spend generating bus traffic when verifying devices like memory controllers, bus fabrics, routers, and switches. Equivalent tests implemented using either CRT or Directed test methods have difficulty meeting coverage goals, and often require many more tests and low-level test coding to implement. Verification engineers must also understand the low-level implementation details of their verification IP to develop these tests and sequences. Test-IP eliminates the requirement to understand these details.

Test-IP leverages the built-in stimulus targeting features of the underlying inFact technology, which can automatically adapt the targeted stimulus based on instance-specific test configurations. There is no equivalent methodology using CRT or Directed techniques. While users are always encouraged to write System Verilog functional coverage models, the effort to do so can be prohibitive and often functional coverage is dropped when project timelines are critical. Test-IP adds stimulus coverage targeting for such applications, reducing the time to reach System Verilog functional coverage goals (if implemented), or increasing confidence using the built-in stimulus coverage targeting.

Protocols supported today using Test-IP include Amba ahb, axi, axi4, and ace. Other standard bus protocols could be supported. While Test-IP today is primarily intended for applications having an arbitrary data payload, the architecture could be extended to access application-specific data contained in a TLM fifo, which could be either static data, or data generated by a higher-level UVM sequence designed to generate real data in an SOC system.

Test-IP is most easily implemented using inFact graph-based UVM sequences, and could be designed to drive various types of verification IP as long as the VIP has well-architected sequence items. Non-inFact Test-IP could be developed using a combination of directed and random techniques, though the adaptive nature of the inFact-based Test-IP graphs and stimulus coverage targeting would be difficult to implement.