

Chapter 4

UVM Verification Ecosystem Implementation

In the previous chapter, we laid out the core components and philosophies behind UVM. In this chapter, we show how these concepts were applied in the design of our verification environment. Cross-references to previous sections appear as (see a.b.c) throughout this chapter e.g., (see 3.2).

4.1 UVM Challenges & Scaffolding Approach

Although UVM has significant benefits, it also introduces substantial challenges. The library exposes ~300 classes with highly versatile but potentially confusing methodology—the same task often has multiple valid approaches, creating ambiguity for new users. The official documentation lacks practical, end-to-end examples. Beyond documentation gaps, UVM presents practical hurdles: a steep learning curve, complex debugging due to layered OOP architecture, and difficult failure tracking since issues in layered components may surface far from their origin.

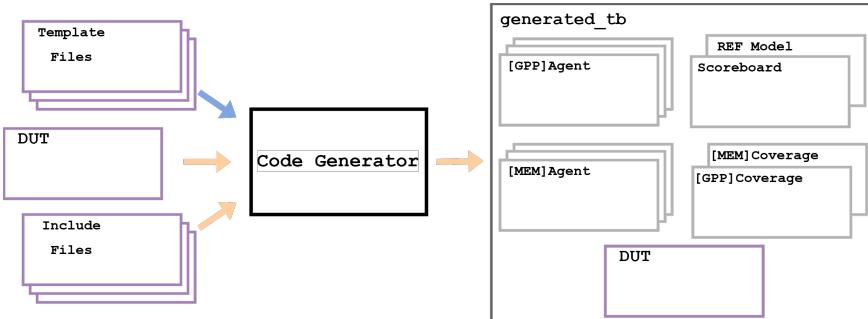


Fig 4.1: Basic UVM generated testbench by the code generator

To address this challenge, we used a code generator developed by Duolos to create a well-structured test bench foundation. The generator requires our DUT combined with template files written in the easierUVM template language, along with user-defined code files. The code generator offers three approaches: (i) learning the basics through working examples, (ii) scaffolding—generating initial code versions as a foundation, then discontinuing the generator and building manually upon the base, and (iii) continuous regeneration throughout development.

Project Implementation Methodology. Building on this foundation, we chose the scaffolding approach, utilizing the well-structured test bench architecture and basic boilerplate patterns generated by the code generator. We then manually developed our complete verification ecosystem by adhering to UVM's best practices and maintaining consistent coding patterns.

4.2 Architecture & Design Decisions

As shown in Fig 1.1, the verification ecosystem employs a dual-agent architecture developed through systematic component integration. Development progressed from basic Sequencer-Driver handshakes and sampling monitors to a complete Reference Model and Scoreboard implementation.

A critical design milestone occurred during UVM RAL evaluation. Initial integration attempts revealed fundamental misalignment with our requirements, leading to deeper understanding of RAL's appropriate use cases. This analysis informed development of a Memory Service driver with comprehensive memory model, while preserving integration-ready RAL files for future physical memory interface applications.

This architectural evolution prompted systematic evaluation of core verification platform requirements and verification planning methodologies. The resulting approach emphasizes flexibility and systematic component patterning, establishing a scalable foundation that adapts to complex system-on-chip verification challenges.

4.2.1 Multi-Layer Abstraction Framework

Semantic Abstraction. To address readability challenges with control signals where numerical indices created unnecessary complexity throughout the codebase, we implemented verbose parameters inspired by enumeration techniques commonly used for CSR signals within the DUT, establishing semantic clarity at the interface level.

This approach extended across multiple domains: (1) Memory identifiers using descriptive names (MEM1 instead of numerical indices), (2) Timing control parameters for race condition management, (3) Coverage-specific parameters abstracting complex packed array hexadecimal values, and (4) Functionality mode enumerations supporting comprehensive memory feature validation.

```
localparam logic [31:0] START_MATMUL = 32'h1;
localparam logic [31:0] START_CALCOPY = 32'h2;
parameter real RACE_CTRL = 0.01;

typedef enum logic [1:0] {
    START_IDX_REG, //0
    BUSY_IDX_REG, //1
    DONE_IDX_REG //2
} idx_regs;

localparam logic [NUM_MEMS-1:0][31:0] MEM_BE_ALL_0_1 = 64'hFFFF_FFFF_FFFF_FFFF;

covergroup cg_out; // MEM Cov
cp_mem_be: coverpoint tx_out.mem_be {
    bins mem_be_vals = {MEM_BE_ALL_0,
                        MEM_BE_ALL_0_1};
}
...
endgroup

covergroup cg_in; // GPP Cov
...
cp_start: coverpoint tx_in.host_regs[START_IDX_REG] {
    bins start_vals = {START_MATMUL, START_CALCOPY};
}
...
endgroup
```

Fig 4.2: Semantic Abstraction Implementation - From numerical parameters to descriptive enumerations

Physical Timing Abstraction. Through this initial mental process, preliminary planning rules established a common global contract across all components: standardized behavior according to positive clock edges. Drivers exclusively operate on positive edges when manipulating pins as the front-door interface to the DUT, while monitors sample accordingly. This approach creates a unified mental model for addressing synchronization across all verification components.

```

function void xlr_mem_tx::set_e_mode(string s);
    assert (s=="def"||s=="rst_i"||s=="rst_o"||s=="rd"||s=="wr")
    else begin
        `uvm_fatal(get_type_name(),
        $sformatf("invalid e_mode '%s'", s)) return;
    end
    e_mode = s;
endfunction // Defines "Which event?"

```

Fig 4.3: Transaction Data Abstraction Example

Transaction Data Abstraction. The simulation involves GPP-DUT control signals and DUT-MEM read/write requests (see 2.2). To manage complexity, an "Event Mode" transaction property was implemented to communicate transaction types across components effectively.

The Event Mode approach extends beyond simple categorization by incorporating field-specific method operations. When a mode is set, overridden transaction methods operate exclusively on event-related transaction fields. For example, a read event restricts operations to `mem_rdata`, `mem_rd`, and `mem_addr` signals, ensuring precise field manipulation. To maintain architectural integrity, a strict component responsibility rule governs Event Mode usage: only transaction creator components (such as Monitors) may set event modes, while consumer components (such as the Scoreboard) remain restricted from mode modification.

This design ensures that once a transaction broadcasts, consumers can apply methods confidently, knowing they operate on the correct transaction fields for the specified event type.

Interface Signal Access (ABC Methods). Implementation through Abstract Base Class technique replacing Virtual Interfaces (see 3.5) enables running multiple parameterized DUT versions simultaneously. This approach requires method-based signal access since interface signals in an ABC implementation belong to the interface rather than the ABC class itself. While initially appearing complex, these methods prove straightforward to implement through descriptive, meaningful naming conventions (detailed method implementations available in Appendix A).

The resulting signal abstraction creates self-documenting code with minimal implementation overhead. Combined with the previous abstraction layers, this approach achieves abstraction levels where components focus exclusively on their core purpose while code complexity remains minimal.

```

function void xlr_mem_tx::do_copy(
    uvm_object rhs);
    xlr_mem_tx rhs_;
    if (!$cast(rhs_, rhs))
        `uvm_fatal("Event Set","Cast Fail")
    super.do_copy(rhs);
    mem = rhs_.mem; //Communicates Which:
    e_mode = rhs_.e_mode; //MEM?/Event?
    if (mem==MEMA) begin
        if (e_mode=="def") begin
            // ... all I/O Data Signals
        end else if(e_mode=="rst_i") begin
            mem_rdata = rhs_.mem_rdata;
        end else if(e_mode=="rst_o") begin
            // ... all OP Data Signals
        end else if(e_mode=="rd" ) begin
            mem_rd   = rhs_.mem_rd;
            mem_addr = rhs_.mem_addr;
            mem_rdata = rhs_.mem_rdata;
        end else if(e_mode=="wr" ) begin
            mem_wr   = rhs_.mem_wr;
            mem_addr = rhs_.mem_addr;
            mem_be   = rhs_.mem_be;
            mem_wdata = rhs_.mem_wdata;
        end
    end else begin ...
endfunction

```

Fig 4.4: Field-selective copying

4.2.2 Structural Organization & Phase Methods

UVM's boilerplate nature enables shared structural patterns across verification components. Although components may vary in specific elements, we maintained consistent architectural frameworks. This structural uniformity simplifies the verification environment, requiring attention to specific implementations only. In the following examples we wish to demonstrate the structure and the typical patterns within the phase methods.

In Fig 4.5, a global pattern for each component adhering to the same template framework is showcased with consistent internal organization, varying only in their component-specific instantiation details. This organizational consistency reduced navigation overhead in our project while maintaining flexibility.

The essential UVM infrastructure consists of macro inclusions, package imports, class declarations, factory registrations via ‘`uvm_component_utils`’ for overriding and analysis port / export declarations, followed by phase method signatures with the `extern` keyword and additional helper methods. Implementing the methods outside of the class creates an improved separation aimed for better readability which was important to us through continuous scaling of the verification environment.

```
'include "uvm_macros.svh"
import uvm_pkg::*;
import honeyb_pkg::*;
// for subscribers uvm_analysis_imp_decl...
class user_defined_class extends uvm_base_class;
  `uvm_component_utils(user_defined_class)

  // uvm analysis/export handles...
  // VIF, component, variable declarations...

  extern function new(string name, uvm_component parent);
  extern function void build_phase(uvm_phase phase);
  extern function void connect_phase(uvm_phase phase);
  extern task run_phase(uvm_phase phase);
  // additional helper methods (do_mon(), write() ...)
endclass

/* constructor
// build_phase..
  (config retrieval, component instantiation)
// connect_phase..
  (port/export connections)
// run_phase..
  (for time consuming components e.g., drivers)

// helper methods...
```

Fig 4.5: Standardized Component Template

Constructors & Build Phases

In Fig 4.6, constructors are kept as simple boilerplate, receiving names as strings and parent references with a superclass call to the component’s `new()` method. The build phase is where most things happen: port instantiations, configuration retrieval using `get()` with error handling and factory creation receiving two arguments: a string of the component’s handle in the form of a repeatedly used “`m_*`” naming convention and the parent referencing to (“`this`”) for proper hierarchical relationships.

At the bottom, a condition statement specific to agents is seen checking a config knob for setting the agent’s mode as active; if true the driver and the sequencer are created, otherwise the agent will remain passive with just the monitor being instantiated.

```
function xlr_gpp_agent::new(string name, uvm_component parent);
  super.new(name, parent);
endfunction // Boilerplate

function void xlr_gpp_agent::build_phase(uvm_phase phase);
  analysis_port_in = new("analysis_port_in", this);
  analysis_port_out = new("analysis_port_out",
    this);

  if (!uvm_config_db #(xlr_gpp_config)::get(this,
    "", "config", m_config))
    `uvm_error("", "xlr_gpp config not found")

  m_monitor = xlr_gpp_monitor::type_id::create("m_monitor",
    this);

  if (get_is_active() == UVM_ACTIVE) begin
    m_driver = xlr_gpp_driver::type_id::create("m_driver",
      this);
    m_sequencer = xlr_gpp_sequencer_t::type_id::create("m_sequencer",
      this);
  end
endfunction
```

Fig 4.6: Constructors & Build Phases

Connect Phases

Fig 4.7 shows the GPP Agent's connect phase representing the general structure of how connect phases typically look like with the exception of the agent mode if() statement being unique to the agent along with the virtual interface, where we make sure it is not null and proceed to pass it down from the top config to the monitor & driver.

In addition, a child-parent connection is seen between the monitor & agent, demonstrating consistent connection syntax with variations in port types and component hierarchy (see 3.3.1).

```
function void xlr_gpp_agent::connect_phase(uvm_phase phase);
  if (m_config.vif == null)
    `uvm_warning("", "xlr_gpp virtual interface is not set!")

  m_monitor.vif = m_config.vif;
  m_monitor.analysis_port_in.connect(analysis_port_in); // Child - Parent (Agent) connection
  m_monitor.analysis_port_out.connect(analysis_port_out);

  if (get_is_active() == UVM_ACTIVE) begin
    // Peer to Peer connection
    m_driver.seq_item_port.connect(m_sequencer.
      seq_item_export);
    m_driver.vif = m_config.vif;
  end
endfunction
```

Fig 4.7: Connect Phases

Run Phases

We implement three different run_phase patterns, where components of the same type share common implementations:

(i) **Monitors.** The run phase in monitors are simple: They include the factory method for the transaction objects created to broadcast the sampled pin-level action, followed by calling the user-defined do_mon() method (see 4.3.1).

(ii) **Drivers.** Fig 4.9 shows the run phase of our MEM driver. At the top, ABC methods realizing a simple boot sequence reset handling are seen (ABC list in Appendix A), followed by the Driver's handshake protocol with the Sequencer (see 3.3.3). Within that handshake, the driver raises an objection indicating it is busy and calls the do_drive() user-defined method. The objection drop is implemented within a fork...join_none block to avoid risking early termination (see 3.7.2).

(iii) **Env.** The last run phase is unique being a boilerplate implementing the virtual top sequence within the top_env (see 3.3.3). The figure shows the important mechanism of accessing uvm_phase, calling the set_starting_phase method to pass the access further to the child sequences. It is important to note that virtual sequences don't run on sequencers, which is why it's being started with null.

```
task xlr_mem_monitor::run_phase(uvm_phase phase);
  m_trans_in = xlr_mem_tx::type_id::create("m_trans_in");
  m_trans_out = xlr_mem_tx::type_id::create("m_trans_out");
  do_mon();
endtask // Boilerplate + Report
```

Fig 4.8: MEM Monitor's Run Phase

```
task xlr_mem_driver::run_phase(uvm_phase phase);
  m_xlr_mem_if.rst_n_negedge_wait();
  m_xlr_mem_if.pin_wig_rst();
  m_xlr_mem_if.rst_n_posedge_wait();
  forever begin
    seq_item_port.get_next_item(req);
    phase.raise_objection(this);
    do_drive();

    fork // drop delay
      begin // avoiding accidental early terminate
        repeat (10) m_xlr_mem_if.clk_posedge_wait();
        phase.drop_objection(this);
      end
    join_none
    seq_item_port.item_done();
  end
endtask
```

Fig 4.9: MEM Driver's Run Phase

```
task top_env::run_phase(uvm_phase phase);
  top_default_seq vseq;

  vseq = top_default_seq::type_id::create("vseq");
  vseq.set_item_context(null, null); //No Sequencer!
  if (!vseq.randomize())
    `uvm_fatal("", "Failed to randomize virtual sequence")
  vseq.m_xlr_mem_agent = m_xlr_mem_agent;
  vseq.m_xlr_gpp_agent = m_xlr_gpp_agent;
  vseq.m_seq_count = m_config.m_seq_count;
  vseq.set_starting_phase(phase);
  vseq.start(null);
endtask // Boilerplate
```

Fig 4.10: Top Env's Run Phase

4.3 Key Method Implementations

This section presents the core methods across four critical verification components, illustrating the design patterns for each through practical application of our abstraction layers (see 4.2.1), enabling a cohesive system for effective transaction processing.

4.3.1 Monitors & Drivers

Fig 4.11 shows the `do_drive()` method called from `run_phase` (see Fig 4.9). The driver waits for posedge `clk` before driving pins according to the global timing contract. Like various GPP Agent methods, `clk_posedge_wait()` replicates the naming convention and semantic style of the MEM Agent's ABC Methods (see 4.2.1), creating consistent language across both agents despite their different interface mechanisms (VIF vs. ABC). Additional GPP package methods prioritize semantic abstraction over explicit comparisons and comments (e.g., `is_start_asserted()`). The driver checks for START or CALCOPY operations encoded in the GPR, matching available DUT functionalities: START instructs the DUT to perform Matrix Multiplication; CALCOPY performs the same operation plus copies the result to additional memory. The signal is held for 1 clk cycle, then the driver awaits the DUT's DONE signal.

Fig 4.12 establishes consistent patterns between reset samplers (`IN&OUT`) broadcasting through different channels (see 1.2). While functionally simple, these samplers highlight two race condition scenarios that commonly occur when monitors use BAs and drivers use NBAs¹. This conflict stems from LRM Event Ordering, where blocking assignments execute in the Active Region while non-blocking assignments execute in the NBA Region. Our solution uses intentionally placed `#RACE_CTRL` delays (see 4.2.1) near specific methods requiring timing management.

This pattern mirrors the reset samplers in MEM Monitor.

Fig 4.13 shows the read sampler in MEM monitor. The monitor flushes the input transaction channel, then waits until read is requested. It creates multiple transactions aligned with platform functionality, holds 1 clk cycle and reads the Driver's response, mimicking MEM Driver behavior.

This pattern mirrors the write sampler.

```
task xlr_gpp_driver::do_drive();
    clk_posedge_wait();
    for(int cs_idx=0; cs_idx < 32; cs_idx++) begin
        vif.host_regs[cs_idx]<req.host_regs[cs_idx];
        vif.host_regs_valid[cs_idx] <= 1'bl; end
    if( is_start_asserted()
        req.host_regs[START_IDX_REG],
        req.host_regs_valid[START_IDX_REG]) ||
        is_calcop asserted(
        req.host_regs[START_IDX_REG],
        req.host_regs_valid[START_IDX_REG])) begin
        clk_posedge_wait(); // Hold for 1 clk
        vif.host_regs[START_IDX_REG] <= '0;// deassert
        vif.host_regs_valid[START_IDX_REG] <= '0;
        done_wait_until_assert(); end
    endtask
```

Fig 4.11: GPP Driver: Do-Drive Method

```
forever begin // IN
    rst_n_nedge_wait(); #RACE_CTRL;
    m_trans_in.host_regs      = vif.host_regs;
    m_trans_in.host_regs_valid = vif.host_regs_valid;
    m_trans_in.set_e_mode("rst_i"); //sample&event set
    'honeyb("GPP Monitor", "RESET(STIMULUS) detected")
    m_trans_in.print();
    analysis_port_in.write(m_trans_in);
    rst_n_posedge_wait();
end ...
forever begin // OUT
    rst_n_nedge_wait(); #RACE_CTRL;
    // Sample DUT OP Signals..
    m_trans_out.set_e_mode("rst_o");
    'honeyb("GPP Monitor", "RESET(DUT RSP) detected")
    m_trans_out.print();
    #RACE_CTRL; analysis_port_out.write(m_trans_out);
    rst_n_posedge_wait();
end
```

Fig 4.12: GPP Monitor: I/O Reset Sampler

```
forever begin //Read REQ & RSP Sampling
    flush_trans_in();
    m_xlr_mem_if.wait_for_rd_sampling();
    rd_mems = m_xlr_mem_if.get_active_rd_mems();
    for (int m=0; m < NUM_MEMS; m++) if (rd_mems[m])
        create_trans_in(m); // DUT's REQ
    m_trans_in.set_e_mode("rd");
    m_xlr_mem_if.clk_posedge_wait();
    // Driver's RSP
    for (int m=0; m < NUM_MEMS; m++) if (rd_mems[m])
        m_trans_in.mem_rdata[m] =
            m_xlr_mem_if.get_rdata(x_mem'(m));
    'honeyb("MEM Monitor", "READ detected")
    m_trans_in.print(); // Report
    analysis_port_in.write(m_trans_in); end
```

Fig 4.13: MEM Monitor: Read Sampler

¹Blocking & Non-Blocking Assignments

4.3.2 REF Model & Scoreboard

The reference model implements multi-port architecture through paired methods with consistent patterns varying only in functionality per agent (see 3.4.1): `write_*`() and `send_*`(). This subscriber-broadcaster pattern enables transaction processing where `write()` methods receive monitored transactions while `send()` methods generate expected results, establishing explicit role distinctions that support clear mental modeling and align with UVM best practices.

Transactions require careful reference management on the receiving end to prevent data corruption. To address that, we formalized this common UVM practice as the **DFC Rule**; a systematic approach to emphasize its critical importance. The initialism stands for "Declare it, Factorize it, Copy it" representing the three crucial steps for producing the expected results safely. The REF follows the producer-sets-mode rule(see 4.2.1), resetting `e_mode` based on expected DUT behavior. Another special field for functionality modes enables cross-agent coordination, where GPP provides functionality intelligence that MEM operations later utilize through the `calcopy()` method abstraction. Lastly, the reset case follows the same pattern: the REF sets the event and resets all fields with `op_flush()`.

Centralized on checking, the scoreboard consists of 4 `write()` methods, 2 DUT-REF pairs for the GPP & MEM, and a transaction comparison method. The write methods are identically patterned, a DFC followed by a `push_back()` (see 3.4.1). Once the DUT's transaction arrives, this method is called to: check queue sizes, use `pop_front()`, `compare()` & report.

The examples of this section demonstrate our methodology for handling entire ecosystems with the key intent of maximizing flexibility through parameterized interfaces and project-specific semantics with multiple layered abstractions while maintaining complexity at bare minimum to enable scalability for easy physical memory integration and advancement to SoC-level simulation. By applying these systematic approaches, we significantly reduced commenting and repetitive logic, which in turn reduced the heavy mental load correlated with maintaining the environment.

```
function void reference::write_reference_mem(xlr_mem_tx t);
    send_xlr_mem_input(t);
endfunction// Boilerplate
...
function void reference::send_xlr_mem_input(xlr_mem_tx t);
    xlr_mem_tx tx; // DFC Rule
    tx = xlr_mem_tx::type_id::create("tx");
    tx.copy(t);
    if (tx.e_mode == "rd") begin
        tx.set_e_mode("wr");
        // the expected signals for writing back
        tx.mem_be [MEMO] = 32'hFFFFFF; // Write Gate En
        tx.mem_wr [MEMO] = 1'b1;
        tx.mem_addr [MEMO] = 8'h01; // Write res into addr[1]
        // Set Irrelevant fields to '0, e.g.:
        tx.mem_be [MEM1] = '0;
    ...
    // Expected DUT Output Computation
    ...
    // CALCOPY's Func: Copying result into MEM1
    if (f_mode == CALCOPY) tx.calcopy(MEM1, MEM0);
    analysis_port_mem.write(tx);
end else if (tx.e_mode == "rst_i") begin
    tx.set_e_mode("rst_o");
    tx.op_flush();
    analysis_port_mem.write(tx);
end else `honeyb("REF Model", "[WARNING]Event Mismatch!");
endfunction// Chain of Events: rd -> wr & rst_i -> rst_o
```

Fig 4.14: REF Model: Key Methods & DFC Rule

```
function void xlr_scoreboard::write_mem_ref(xlr_mem_tx tx);
    //The DFC Rule: Declare it, Factorize it, Copy it
    xlr_mem_tx stored_tx; // D
    stored_tx = xlr_mem_tx::type_id::create("stored_tx"); // F
    stored_tx.copy(tx); // C
    // First-In Insertion
    mem_ref_queue.push_back(stored_tx);
endfunction
...
function void xlr_scoreboard::compare_mem_transactions();
    xlr_mem_tx ref_tx; xlr_mem_tx dut_tx; bit result;
    ref_tx = xlr_mem_tx::type_id::create("ref_tx");
    dut_tx = xlr_mem_tx::type_id::create("dut_tx");
    // Check if both queues have transactions
    if(mem_ref_queue.size()>0&&mem_dut_queue.size()>0) begin
        // First-Out Extraction
        ref_tx = mem_ref_queue.pop_front();
        dut_tx = mem_dut_queue.pop_front();
        result = ref_tx.compare(dut_tx);
        if (result) begin//Event-specific success reporting
            ...
            mem_match_count++;
        end else begin // Event-specific failure reporting
            ...
            mem_mismatch_count++;
            dut_tx.print(); ref_tx.print();
        end
    end
endfunction
```

Fig 4.15: Scoreboard: Key Methods & DFC Rule

enable scalability for easy physical memory integration and advancement to SoC-level simulation. By applying these systematic approaches, we significantly reduced commenting and repetitive logic, which in turn reduced the heavy mental load correlated with maintaining the environment.

4.4 Memory Model & Service Driver Development

During project development, UVM RAL was explored as a potential solution for memory modeling. The initial understanding was that RAL capabilities could be applied without physical memories in the DUT, leading to creating a complete RAL setup for the SRAM components.

However, this approach revealed two critical limitations. First, using `uvm_reg` & `uvm_reg_block` classes with mirror & desired value spaces for large memories proved impractical; with configurations reaching 8 SRAMs and 1024 address lines at 256-bit width, simulation performance becomes prohibitive, highlighting the need for `uvm_mem` classes instead designed for such cases.

The second issue was more fundamental: RAL requires physical memory components to function effectively, which were not included in the scope of this project. This realization clarified that RAL use cases were misaligned with the project's simulation-based approach.

4.4.1 Dual-Mode Architecture: Configuration Framework

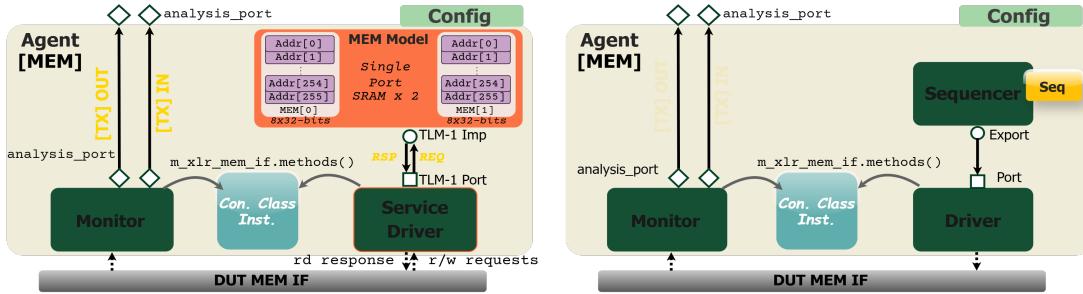


Fig 4.16: MEM Agent: Dual-Mode Architecture

The solution emerged from recognizing that drivers can operate in responsive mode. Instead of the traditional Sequencer + Sequence architecture, a memory model could be implemented where the service driver communicates through a generic TLM-1 interface using REQ-RSP mechanisms. In this setting, the service driver responds to DUT requests while the GPP driver manages the overall simulation flow.

Integration requires a config parameter. `mem_is_used` was added to the memory config file for mode switching. The modular design of the ecosystem allowed this integration with minimal code changes through two key mechanisms: (i) a type override mechanism in the `top_test` replaces components system-wide. (ii) within the agent class, additional handles are declared with conditionally controlled factory instantiations and connections, requiring down-casting to resolve compilation-elaboration timing conflicts (Fig 4.17).

```

function void top_test::build_phase(..);
    top_config m_config;
    ...
    // 0 For Sequ'r(+Sequence) Mode
    m_config.m_xlr_mem_config.mem_is_used = 1;
    if (m_config.m_xlr_mem_config.mem_is_used)
        xlr_mem_driver::type_id::set_type_override(
            xlr_mem_service_driver::get_type());
endfunction

class xlr_mem_agent extends uvm_agent;
    xlr_mem_model m_model; // New Handles
    xlr_mem_service_driver m_service_driver;
    xlr_mem_driver m_driver; // Existing Handle
    ...
    // Method Signatures
endclass

function void xlr_mem_agent::build_phase(..);
    ...
    // AP Create, cfg db retrieval
    if (m_config.mem_is_used == 1'b0) begin
        m_sequencer = xlr_mem_sequencer_t::type_id::create(
            "m_sequencer", this);
    end else begin // if 1 -> Build mem model
        m_model = xlr_mem_model::type_id::create(
            "m_model", this);
        uvm_config_db #(xlr_mem_config)::set(this,
            "m_model", "config", m_config);
    end ...
endfunction

function void xlr_mem_agent::connect_phase(..);
    if (m_config.mem_is_used == 1'b0)
        m_driver.seq_item_port.connect(m_sequencer,
            seq_item_export);
    else begin // If 1 -> Connect mem model & Driver
        if (m_model == null)
            `uvm_fatal("", "m_model not created")
        if (!$cast(m_service_driver, m_driver))
            `uvm_fatal("", "Driver override failed");
        m_service_driver.model_bt.connect(m_model.bt_imp);
        // Connect: service_driver <-> mem_if !
        m_service_driver.m_xlr_mem_if = m_xlr_mem_if;
    end ...
endfunction

```

Fig 4.17: Dual-Mode Configurations

4.4.2 Service Driver Mechanics

```

class xlr_mem_service_driver extends xlr_mem_driver;
  uvm_blocking_transport_port#(xlr_mem_tx, xlr_mem_tx)
    ) model_bt; // req, rsp

  xlr_mem_tx req; // Mem-Service TXs
  xlr_mem_tx rsp;
  // indicators for active memories
  bit [NUM_MEMS-1:0] rd_mems, wr_mems;
  ... // Method Signatures
endclass

```

```

function void xlr_mem_service_driver::build_phase(..);
  model_bt = new("model_bt", this); // Port Create
endfunction

task xlr_mem_service_driver::run_phase(..);
  req = xlr_mem_tx::type_id::create("req"); // Factory
  rsp = xlr_mem_tx::type_id::create("rsp");
  ... // Reset Boot Sequence
  do_drive();
endtask

```

The service driver class declares TLM generic transport ports, needed when standard TLM connections (see 3.3.1) are insufficient, along with indicators for active memory operations.

Memory requests are monitored through `wait_for_dut_request()`² within `do_drive()`, detected via `get_active_*` methods and stored in indicators due to possible concurrent reads & writes across different SRAMs, constrained only by their single-port nature.

Driver then creates requests & calls `mem_model` via `transport(..)`, responding in zero-time to read requests through `pin_wig_rdata()` held for 1 cycle. Write requests lack responses due to absent ACK signal, resulting in a special report method for debugging. This demonstrates effective role separation; driver handles timing without concern for memory operation conflicts, delegating all business logic to `mem_model`, simplifying the code while keeping clear functionality.

4.4.3 Memory Model Design

Our model was developed to support Matrix ops (see 2.3) with features designed for verification flexibility. Functionality centers around memory initialization policies covering various scenarios.

INIT_* Policies. Default: memories are randomized through a custom randomizer class, capping values within specified ranges per word, enabled via Xcelium's `-seed random` flag in our `xrun_options.rtl` script with a default seed knob useful for debugging.

Optionally, file-based loading using a convenient format is available, or for scenarios where some memories serve as write targets the `INIT_NONE` is included. To complement these, `UNINIT_LAST` manages read attempts from unwritten addresses issuing warnings, returning the most recent read value, whereas `UNINIT_ZERO` provides the zero matrix. These policies are also controlled from our control center in `top_test`.

```

task xlr_mem_service_driver::do_drive();
  forever begin
    flush_all(); // Refreshing Req + Indicators
    m_xlr_mem_if.wait_for_dut_request(); // DUT Polling
    rd_mems = m_xlr_mem_if.get_active_rd_mems();
    wr_mems = m_xlr_mem_if.get_active_wr_mems();

    for (int m = 0; m < NUM_MEMS; m++)
      if (rd_mems[m] || wr_mems[m]) create_req(m);
    model_bt.transport(req, rsp); // Send Req
    // Drive response for READ requests
    for (int m = 0; m < NUM_MEMS; m++)
      if (rd_mems[m]) m_xlr_mem_if.pin_wig_rdata(
        x_mem'(m), rsp.mem_rdata[m]);
    create_req_report(); // RD Valid & Mem ACK report
    m_xlr_mem_if.clk_posedge_wait(); // Hold 1 cycle!
  end
endtask

```

Fig 4.18: *Memory Service Driver Implementation*

```

class mem_line_randomizer; // in "xlr_mem_model.sv"
  rand logic [NUM_WORDS-1:0][WORD_WIDTH-1:0] data;
  // NUM_WORDS = 8, WORD_WIDTH = 32
  constraint valid_range {
    foreach (data[i]) { data[i] inside {[0:20]}; }
  }
endclass // Utility class for randomization

```

```

class top_config extends uvm_object;
  rand xlr_mem_config m_xlr_mem_config;
  ... // Handles, Params, Signatures
endclass
// global cfg defaulting center: Set once only
function top_config::new(..); // DONT CHANGE
  m_xlr_mem_config.mem_is_used = 1;
  m_xlr_mem_config.rand_seed = 0;
  m_xlr_mem_config.en_write_dumps = 0;
  m_xlr_mem_config.sim_dump_limit = 1;
  m_xlr_mem_config.dump_directory = "./mem_dumps/";
  m_xlr_mem_config.init_policy = INIT_RANDOM;
  m_xlr_mem_config.uninit_policy = UNINIT_LAST;
  ... // Other default knobs
endfunction

```

```

function void top_test::build_phase(..);
  ... // Declare & Retrieve top config
  m_config.m_xlr_mem_config.mem_is_used=1; // 0:Off
  if(m_config.m_xlr_mem_config.mem_is_used)begin
    m_config.m_xlr_mem_config.en_write_dumps=1;
    *sim_dump_limit = 2;
    *.init_policy = INIT_RANDOM;
    *.uninit_policy = UNINIT_LAST;
  end
  ... // top_env factory
endfunction

```

Fig 4.19: *Memory Model Configurations*

²See Appendix A for complete ABC Method list

Memory Model: Core Methods

```

class xlr_mem_model extends uvm_component;
  uvm_blocking_transport_imp#(//req, rsp, imp src
    xlr_mem_tx, xlr_mem_tx, xlr_mem_model) bt_imp;
  typedef logic [7:0][31:0] mem_line;
  mem_line mem [NUM_MEMS][int]; //UP TO 8 MEMS
  mem_line last_mem_rdata[NUM_MEMS];
  bit has_last [NUM_MEMS]; //indicator
  typedef struct packed {
    mem_line data; bit was_uninit; } read_result_t;
  ... // Handles + Method Signatures

```

Being responsible for memory ops, the model implements a generic TLM transport imp. We declare memory structures supporting up to 8 SRAMs through a hashmap, with `last_mem_rdata` and `has_last` supporting the UNINIT_* read policies (see Fig 4.19). `build_phase` creates the imp and retrieves `xlr_mem_config`, followed by a setup via selected policy.

As the core functionality, `transport` is realized as a zero-time task method. Responsible for validating requests adhering to the platform's SP-SRAM constraints while leveraging Cross-bar interconnection capabilities, rejecting null, out-of-bound addresses, & read/write requests both attempted on the same memory, exiting simulation as these indicate testbench or DUT errors requiring immediate attention.

There's a clear separation of concerns between driver and memory model. Model handles mem ops unaware of timing, which is handled by driver. Valid requests proceed through concurrent read & write execution.

```

function void do_write(int mem_idx, int addr,
  mem_line wdata, logic [BE_WIDTH-1:0] be);
  mem_line old_line = mem[mem_idx][addr];
  mem[mem_idx][addr]=be_merge(old_line, wdata, be);
  if(m_config.en_write_dumps)
    dump_mem(mem_idx,sim_fname);
endfunction // be handling + write dump feature

```

```

function void build_phase(..);
  bt_imp = new("bt_imp", this);
  if (!uvm_config_db#(xlr_mem_config)::get(..))
    `uvm_error("", "xlr_mem_config not found")
  init_all(); // Based on Init Policy
endfunction // Mem Init + port build

```

```

task transport( // RSP to Driver's Call
  input xlr_mem_tx req, output xlr_mem_tx rsp);
  if (req == null) `uvm_fatal("", "NULL REQ")
  for (int m = 0; m < NUM_MEMS; m++) // Illegal for
    if ((req.mem_rd[m]&&req.mem_wr[m]) // Single-Port SRAM
      `uvm_fatal("", "RD & WR ON SAME MEM")
  for (int m = 0; m < NUM_MEMS; m++) // Addr Handle
    if ((req.mem_addr[m] < 0) || // If Out of Bounds
      (req.mem_addr[m] > LINES_PER_MEM - 1))
      `uvm_fatal("", $sformat("Addr OOB: mem_addr[%0d] = %0h", m, req.mem_addr[m], LINES_PER_MEM-1))
  // rsp created only if req is valid:
  rsp = xlr_mem_tx::type_id::create("rsp");
  rsp.copy(req);

  for (int m = 0; m < NUM_MEMS; m++) begin
    if (req.mem_wr[m]) do_write(m, req.mem_addr[m], req.
      mem_wdata[m], req.mem_be[m]);
  end // Perform Concurrent Writes

  for (int m = 0; m < NUM_MEMS; m++) begin
    if (req.mem_rd[m]) begin
      read_result_t result = do_read(m, req.mem_addr[m]);
      mem_line R = result.data;
      rsp.mem_rdata[m] = R; last_mem_rdata[m] = R;
      has_last[m] = 1'b1; end
  end // Perform Concurrent Reads
endtask

```

Fig 4.20: Memory Model Implementations

DUT errors requiring immediate attention.

```

function mem_line be_merge(old_line, wdata, be);
  mem_line new_line = old_line; int be_idx;
  for (int w = 0; w < 8; w++)
    for (int b = 0; b < 4; b++) begin
      be_idx = w*4 + b;
      if(be[be_idx])new_line[w][b*8 +:8]=wdata[w][b*8 +:8];
    end
  return new_line;
endfunction

```

Fig 4.21: Write + Byte Enable Methods

Fig 4.21 shows `do_write`, exemplifying mem ops logic by managing data storage with optional debugging infrastructure (see Fig 4.19). By setting `en_write_dumps`, the ecosystem creates a well-organized filing system containing mem snapshots demonstrated in Fig 4.22: Created in workspace, the directories contain the initialized memories and timestamped write tracking snapshots limited with `sim_dump_limit` knob. A format of `<addr_hex> <data_256bits_hex>` was defined, allowing various uses such as taking a written result and easily feeding it back to the ecosystem for continuous operations. We believe these features altogether create a true memory sandbox offering flexible simulations necessary for MATMUL ops.

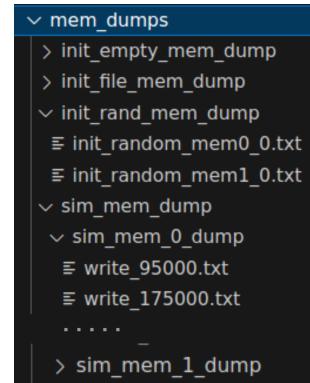


Fig 4.22: Write Dump File Sys

4.5 Debug Infrastructure & Validation

Recalling Section 4.1's scaffolding methodology, the early stages of manual development presented significant debugging challenges. While the code generator provided a structural foundation, the transition to manual implementation revealed fundamental issues with the standard UVM debugging workflow. The verification environment exhibited widespread functionality failures, and tracking these issues became increasingly difficult.

```

1 UVM_INFO ../src/tb/xlr_gpp/sv/xlr_gpp_driver.sv(95) @ 480000: uvm_test_top.m_env.m_xlr_gpp_agent.m_driver [] gpp
2     driver got 'done signal' moving on...
3 UVM_INFO ../src/tb/xlr_mem/sv/xlr_mem_monitor.sv(147) @ 484000: uvm_test_top.m_env.m_xlr_mem_agent.m_monitor [MEM
4     MON]
5 Write Request! Addr SRC : mem_addr[0] = 1
6 UVM_INFO ../src/tb/top/sv/xlr_scoreboard.sv(141) @ 484000: uvm_test_top.m_env.m_xlr_scoreboard [MEM DUT SCRBD]
7     Received Write Request!
8 UVM_INFO ../src/tb/top/sv/xlr_scoreboard.sv(142) @ 484000: uvm_test_top.m_env.m_xlr_scoreboard [MEM DUT SCRBD]
9 mem_addr[0] = 1
10 mem_wr = 1
mem_be = 4294967295
UVM_INFO ../src/tb/xlr_mem/sv/xlr_mem_xlr_mem_tx.sv(71) @ 484000: reporter@0xstored_tx [] Setting mode to: wr
UVM_INFO ../src/tb/top/sv/xlr_scoreboard.sv(203) @ 484000: uvm_test_top.m_env.m_xlr_scoreboard [] MEM TXs match!
```

Fig 4.23: Standard UVM_INFO CLI Output Demonstrating Visual Complexity

The HoneyB utility package emerged from this debugging crisis, evolving into the cornerstone for addressing the ecosystem's most pressing development bottleneck: inefficient debugging tools.

```

HoneyB Message | xlr_mem_monitor(67) @[0] (MEM Monitor) run_phase initialized...
HoneyB Message | xlr_gpp_driver(46) @[0] (GPP Driver)   run_phase initialized...
HoneyB Message | top_seq_lib(55) @[0] (TOP Sequence)    New sequence starting...

[EVENT] (OUTPUT_RESET):
host_regs = 'h0
host_regs_valid = 'h2

HoneyB Message | xlr_scoreboard(275) @[10000] {Scoreboard} Comparing... | Queue Sizes : GPP_REF = 1 GPP_DUT = 1
HoneyB Message | xlr_scoreboard(291) @[10000] {Scoreboard} RESET Status | MATCHED SUCCESSFULLY!
HoneyB Message | xlr_scoreboard(233) @[105000] {Scoreboard} COMPUTATION | MATCHED SUCCESSFULLY!
```

Fig 4.24: HoneyB CLI Output Demonstrating Clean Formatting

Evidently, the verbose `uvm_info` output, with **complete** file paths and **inconsistent** formatting seen in Fig 4.23, created mental fatigue during extended debugging sessions with unnecessary friction of repeatedly typing `uvm_info()`. By prioritizing debugging efficiency through simplified syntax (`honeyb()`) and focused output formatting, we recognized that debugging tools must minimize cognitive load, particularly when development velocity depends on rapid issue identification and resolution.

Fig 4.25 shows the printer's core. The macro interface defines three defaulted parameters: component name, status indicator (e.g., Broadcasting, Detection, Warning), and message content. While these parameters follow semantic conventions to establish consistent printer language, they remain flexible without rigid enforcement serving as guidance.

The `honeyb_printer` function accepts five arguments enabling precise file path tracking and line number identification. Through `$display`, the function implements the consistent formatting pattern seen in Fig 4.24, ensuring uniform output styling across the verification ecosystem.

The package extended beyond printing with enums, memory policies, and global parameters, all featured across the preceding sections.

```

function void honeyb_cls::honeyb_printer(
    string component = "", string status = "", string msg = "",
    string file, int line);
    file = honeyb_filename_extract(file);
    $display("HoneyB Message | %s(%0d) @[%0t] %s) %s%s", file,
            line, $time, component, status, msg);
endfunction : honeyb_printer
... // honeyb_filename_extract implementation
`define honeyb(component = "", status = "", msg = "")
honeyb_pkg::honeyb_cls::honeyb_printer(component, status,
msg, `__FILE__, `__LINE__);
```

Fig 4.25: HoneyB Printer's Core Method

4.5.1 Validation Scenarios

This closing section validates the ecosystem by showcasing scenarios that demonstrate each component performing its intended role.

Ecosystem Initialization: Driver & Sequencer(+Sequence) Mode

Each test run begins with the following detailed initialization log:

The initialization begins with `top_env` declaring the execution of elaboration phases, coverage activation, and completion of the elaboration phase. A detailed topology then lists all created components by name and type, with hierarchical indentation including the TLM ports for each component, validating that everything has been properly built and connected. This is how we validated successful elaborations during the project.

It is then followed by factory details confirming the parameterized interface override (`NUM_MEMS = 2, LOG2_LINES_PER_MEM = 8`). The sequence concludes with monitors, drivers, and sequences entering their run-phase initialization.

Startup Reset and MATMUL Operation Start

Fig 4.27 illustrates the system flow from startup reset through operation initiation, demonstrated with a MATMUL operation. The HoneyB messaging architecture creates logical CLI flow by having monitors report transactions while other components provide status updates upon receiving them. This design prevents CLI overflow with unnecessary details, though comprehensive debugging prints remain available throughout the ecosystem for troubleshooting when needed.

Scoreboard updates, performs immediate comparison and confirms successful reset status (see 4.3.2). When a sequence begins, components report their status and specify the DUT functionality being executed. Memory driver reports read operations and pin activity, while the monitor provides sampled activity reports.

The printing system explicitly highlights events, emphasizing their importance for system context. This event-driven reporting is essential for debugging, as it clearly shows which events triggered component operations - crucial for identifying incorrect event configurations.

```
HoneyB Message | top_env(48) @[0] (TOP Environment) Starting build_phase...
HoneyB Message | top_env(80) @[0] (TOP Environment) Starting connect_phase...
HoneyB Message | top_env(87) @[0] (Top Environment) MEM Coverage Enabled, Connecting...
HoneyB Message | top_env(97) @[0] (Top Environment) GPF Coverage Enabled, Connecting...
UVM_INFO @ 0: reporter [UVMTOP] UVM testbench topology:
Name Type Size Value
uvm_test_top top_test - 82734
m_env top_env - 82799
m_refrence reference - 83002
analysis_export_mem uvm_analysis_imp_reference_mem - 83085
analysis_port_mem uvm_analysis_port - 83187
...
m_xlr_gpp_agent xlr_gpp_agent - 82908
...
#####
Factory Configuration (*)
Instance Overrides:
Requested Type Override Path Override Type
xlr_mem_if_base *.xlr_mem_if_2_8 <unknown>
HoneyB Message | xlr_mem_monitor(97) @[0] (Mem Monitor) run_phase initialized...
HoneyB Message | xlr_mem_driver(43) @[0] (MEM Driver) run_phase initialized...
HoneyB Message | xlr_gpp_monitor(66) @[0] (GPF Monitor) run_phase initialized...
...
HoneyB Message | top_seq_lib(55) @[0] (TOP Sequence) New sequence starting...
```

Fig 4.26: Ecosystem Initializing Top Env, Coverage, TB Topology, Factory, Type & Components

```
HoneyB Message | xlr_gpp_monitor(92) @[10000] (GPP Monitor) RESET(STIMULUS) detected..
[EVENT] (INPUT_RESET):
host_regsi = 'h0
host_regs_valid = 'h0

HoneyB Message | xlr_reference_model(135) @[10000] (REF Model) RESET(GPP) Received..
HoneyB Message | xlr_scoreboard(102) @[10000] (Scoreboard) WRITE Result Received!
HoneyB Message | xlr_scoreboard(275) @[10000] (Scoreboard) Comparing...
HoneyB Message | xlr_scoreboard(275) @[10000] (Scoreboard) Queue Sizes | GPP_REF = 1 GPP_DUT = 1
HoneyB Message | xlr_scoreboard(291) @[10000] (Scoreboard) RESET Status| MATCHED SUCCESSFULLY!
...
HoneyB Message | xlr_gpp_monitor(112) @[55000] (GPP Monitor) Start(MATMUL) detected..
[EVENT] (START):
host_regsi [START_IDX_REG] = 'h1
host_regs_valid[START_IDX_REG] = 'h1

HoneyB Message | xlr_reference_model(114) @[55000] (REF Model) START(MATMUL) Received..
HoneyB Message | xlr_reference_model(122) @[55000] (REF Model) Sent busy..
HoneyB Message | xlr_scoreboard(124) @[55000] (Scoreboard) BUSY status Received!
HoneyB Message | xlr_scoreboard(124) @[55000] (Scoreboard) Not Model1 sent done..
HoneyB Message | xlr_gpp_driver(99) @[65000] (GPF Driver) DONE status Received!
HoneyB Message | xlr_mem_driver(104) @[65000] (MEM Driver) waiting for DONE status...
HoneyB Message | xlr_mem_monitor(113) @[75000] (MEM Monitor) Read from | MEM0 |
HoneyB Message | xlr_mem_monitor(113) @[75000] (MEM Monitor) READ detected..

[EVENT] (OUT_READ):
=====
mem_rdata [0] = 'hd00000009000000d00000013000000d00000011000000100000012
mem_addr [0] = 'h0
mem_rd [0] = 'h1
```

Fig 4.27: Startup Reset and MATMUL OP

MATMUL Operation Completion and Results

The second half demonstrates status signal verification through repetitive patterns consistent with the design philosophies discussed earlier in the chapter. Write operations show expected MATMUL results with mem1 remaining unmodified, confirmed by scoreboard validation. The sequence concludes when the done signal is received, indicating successful handshake completion between driver and sequence (see 3.3.3).

This test executed 20 sequence iterations to validate system reliability across multiple runs. The scoreboard provides a comprehensive final report with coverage metrics based on covergroups that track event occurrences. The coverage percentage indicates completion status by verifying that each event was triggered the expected number of times relative to the sequence count, with results shown below:

The report shows 21 memory matches representing the startup reset plus 20 sequence iterations, with zero mismatches confirming successful operations and coverage goals achieved. The CSR signals are evaluated through cross-coverage metrics that verify control signals and their corresponding valid signals occur together, adhering to the established protocol requirements.

Ecosystem Initialization: Memory Model & Service Driver

The memory model configuration demonstrates key architectural differences during initialization, while maintaining the same operational framework shown previously:

The control center configures memory models with different initialization policies - MEM0 uses random values while MEM1 remains empty, demonstrating flexible memory management. The topology confirms successful service driver integration with proper down-casting (`xlr_mem_service_driver` override) and valid TLM-1 connections, distinguishing this architecture from the simple agent mode.

```
HoneyB Message :: xlr_scoreboard(173) @[175000] [Scoreboard] BUSY status :: Received!
HoneyB Message | xlr_scoreboard(212) @[175000] [Scoreboard] Comparing...
HoneyB Message | xlr_scoreboard(275) @[175000] [Scoreboard] Queue Sizes | GPP_REF = 2 GPP_DUT = 1
HoneyB Message | xlr_scoreboard(293) @[175000] [Scoreboard] BUSY Status | MATCHED SUCCESSFULLY!
HoneyB Message | xlr_mem_monitor(149) @[105000] (MEM Monitor) WRITE detected.

[EVENT] (DUT_WRITE):
=====
mem_wr [0] = 'h1
mem_addr [0] = 'h1
mem_be [0] = 'hfffffff
mem_wdata [0] = 'h21000000a000012000000082
=====

mem_wr [1] = 'h0
mem_addr [1] = 'h0
mem_be [1] = 'h0
mem_wdata [1] = 'h0

=====
HoneyB Message | xlr_scoreboard(155) @[105000] (Scoreboard) WRITE Result Received
HoneyB Message | xlr_scoreboard(216) @[105000] (Scoreboard) Comparing...
HoneyB Message | xlr_scoreboard(216) @[105000] (Scoreboard) Queue Sizes | MEM_REF = 1 MEM_DUT = 1
HoneyB Message | xlr_scoreboard(293) @[105000] (Scoreboard) COMPOSITION | MATCHED SUCCESSFULLY!
HoneyB Message | xlr_gpp_driver(17) @[105000] (GPP Driver) DONE status received, moving on...
HoneyB Message | xlr_gpp_seq_lib(66) @[115000] (GPP Sequence) Sequence completed! [MATMUL]
... // Additional Sequences
```

Fig 4.28: MATMUL Execution and Completion

coverage metrics based on covergroups that track event occurrences. The coverage percentage indicates completion status by verifying that each event was triggered the expected number of times relative to the sequence count, with results shown below:

```
HoneyB Message :: xlr_scoreboard(333) @[1935000] (Scoreboard) Final Report:
HoneyB Message :: xlr_scoreboard(334) @[1935000] (Scoreboard) -----
HoneyB Message :: xlr_scoreboard(335) @[1935000] (Scoreboard) MEM Matches : 21
HoneyB Message :: xlr_scoreboard(336) @[1935000] (Scoreboard) MEM Mismatches : 0
HoneyB Message :: xlr_scoreboard(337) @[1935000] (Scoreboard) GPP Matches : 41
HoneyB Message :: xlr_scoreboard(338) @[1935000] (Scoreboard) GPP Mismatches : 0

HoneyB Message | xlr_gpp_coverage(138) @[1935000] (Coverage) Coverage Report:
HoneyB Message | xlr_gpp_coverage(139) @[1935000] (Coverage) -----
HoneyB Message | xlr_gpp_coverage(140) @[1935000] (Coverage) [IN] Cross_start_X_Valid = 100.0%
HoneyB Message | xlr_gpp_coverage(141) @[1935000] (Coverage) [OUT] Cross_busy_X_Valid = 100.0%
HoneyB Message | xlr_gpp_coverage(142) @[1935000] (Coverage) [OUT] Cross_done_X_Valid = 100.0%
HoneyB Message | xlr_gpp_coverage(143) @[1935000] (Coverage) [IN] Coverage score = 100.0%
HoneyB Message | xlr_gpp_coverage(136) @[1935000] (Coverage) [OUT] Coverage score = 100.0%
```

Fig 4.29: MATMUL Operation Report & Coverage

Coverage score = 100.0%

```
HoneyB Message | top_test(71) @[0] (CFG DR Control Center) Setting Mem Model Cfns...
HoneyB Message | xlr_mem_model(291) @[0] (MEM Model) DUMP_DIR_CREATED
Created dump directory structure at './mem_dumps/'
HoneyB Message | xlr_mem_model(228) @[0] (MEM Model) INIT_RANDOM = ORI->mem0, 256 lines
HoneyB Message | xlr_mem_model(376) @[0] (MEM Model) DUMP_STATUS = OK! MEM0
HoneyB Message | xlr_mem_model(377) @[0] (MEM Model) DUMP_STATUS = OK! MEM1
file@'./mem_dumps/init_empty_mem.dump/init_empty_mem1_0.txt' dumped 0 addresses
UVM_INFO @ 0: reporter [UVMTOP] UVM testbench topology:
...
m_driver xlr_mem_service_driver - 84537
m_ipb_upt uvm_blocking_transport_port - 84762
rsp_port uvm_analysis_port - 84637
seq_item_port uvm_seq_item_pulldown_port - 84586
m_model xlr_mem_model - 84616
ht_imp uvm_blocking_transport_imp - 84815
...
Type Overrides:
Requested Type Override Type
xlr_mem_driver xlr_mem_service_driver
```

Fig 4.30: Ecosystem Initializing Memory Model, Service Driver with Factory

CALCOPY Operation Complete Cycle

Key differences begin with confirmation that the sequencer remains uninitialized in this agent configuration, followed by GPP sequence and monitor reports indicating CALCOPY operation initiation. Two key CLI differences demonstrate the memory model capabilities: the dump limit configuration successfully restricts output to 2 entries as configured, and the write results show CALCOPY's distinct behavior with both memories being modified

The validation concludes with complete coverage metrics demonstrating 100% achievement across all defined goals, with zero mismatches confirming reliable system operation.

Memory Dump Analysis: Initial and Post-Operation

For validation completeness, the memory dump files confirm initialization states as configured in Figure 4.30. Each dump begins with header information followed by memory contents in the specified format. MEM0 shows random initialization with populated addresses and data values, while MEM1 remains empty with no initialized addresses, demonstrating successful implementation of the different memory policies.

The post-operation dump files validate successful CALCOPY execution (referencing Figure 4.31). Both memories now contain matching data values that correspond to the CLI activity reports, confirming accurate operation of the CALCOPY functionality and proper memory handling. This completes comprehensive validation of the entire ecosystem across both agent architectures and operational modes.

```

HoneyS Message | x1c_gpp_monitor(110) @{[245000]} (GPP Monitor) UNINITIALIZED, STATIC, PM
HoneyS Message | x1c_gpp_seqctrl(47) @{[245000]} (GPP Sequence) New sequence starting... [GALCOPY]
HoneyS Message | x1c_gpp_monitor(110) @{[245000]} (GPP Monitor) Start(CALCOPY) ctrl detected.

[EVENT] (START):
host_regs@{[START_IDX_REG]} = 'h2
host_regs@{[START_IDX_REG+1]} = 'h1
HoneyS Message | x1c_gpp_monitor(110) @{[255000]} (GPP Monitor) (UPDATE) Generated Max Dumps: 2
HoneyS Message | x1c_gpp_seqctrl(47) @{[255000]} (GPP Sequence) (CALCOPY) state=0, Next=0, HMI=0
HoneyS Message | x1c_mem_monitor(149) @{[295000]} (MEM Monitor) WRITE detected, Broadcasting...

[EVENT] (DUT_WRITE):
=====
mem_wr [0] = 'hl
mem_addr [0] = 'hl
mem_be [0] = 'hfffffff
mem_wdata [0] = 'h630000004b000000b20000007b

=====
mem_wr [1] = 'hl
mem_addr [1] = 'hl
mem_be [1] = 'hfffffff
mem_wdata [1] = 'h630000004b000000b20000007b

```

Fig 4.31: CALCOPY & Mem Model Additions

```

unlike MATMUL which left MEM1 unchanged.

[Homey8 Message] xlr_scoreboard(333) @ [1935000] {Scoreboard} Final Report:
[Homey8 Message] xlr_scoreboard(334) @ [1935000] {Scoreboard} -----
[Homey8 Message] xlr_scoreboard(335) @ [1935000] {Scoreboard} MEM Matches : 21
[Homey8 Message] xlr_scoreboard(336) @ [1935000] {Scoreboard} MEM Mismatches : 0
[Homey8 Message] xlr_scoreboard(337) @ [1935000] {Scoreboard} GFP Matches : 41
[Homey8 Message] xlr_scoreboard(338) @ [1935000] {Scoreboard} GFP Mismatches : 0

[Homey8 Message] xlr_gpp_coverage(139) @ [1935000] {Coverage} Coverage Report:
[Homey8 Message] xlr_gpp_coverage(139) @ [1935000] {Coverage} -----
[Homey8 Message] xlr_gpp_coverage(140) @ [1935000] {Coverage} [IN ] Cross start_X_valid = 100.0%
[Homey8 Message] xlr_gpp_coverage(143) @ [1935000] {Coverage} [OUT] Cross busy_X_valid = 100.0%
[Homey8 Message] xlr_gpp_coverage(144) @ [1935000] {Coverage} [OUT] Cross done_X_valid = 100.0%

[Homey8 Message] xlr_mem_coverage(136) @ [1935000] {Coverage} [IN ] Coverage score = 100.0%
[Homey8 Message] xlr_mem_coverage(138) @ [1935000] {Coverage} [OUT] Coverage score = 100.0%

```

Fig 4.32: MATMUL Operation Report & Coverage

Fig 4.33: Write Dump INITs

Fig 4.34: Write Dump Post-Op