

MHSA 加速器功能验证报告 (UVM)

姓名：黄超凡，张博文，汪子尧
学号：124039910088, 124039910094, 124039910018

上海交通大学微纳电子学系

日期：2025 年 5 月 17 日

目录

1	UVM 介绍	2
2	UVM 验证平台详细说明	3
2.1	验证框架与验证流程	3
2.2	DUT 顶层接口设计	4
2.3	UVM 对象要素列表	6
2.4	加速器验证计划	8
2.5	断言 (Assertion) 设计	8
2.6	覆盖率 (Coverage) 设计	9
2.7	参考模型 (Reference Model) 设计	10
2.8	随机化序列设计	11
3	仿真流程及结果分析	12
3.1	基于 QuestaSim 的 UVM 仿真流程	12
3.2	Scoreboard 结果及分析	12
3.3	断言 (Assertion) 结果及分析	13
3.4	覆盖率 (Coverage) 结果及分析	15
3.4.1	功能覆盖率结果及分析	15
3.4.2	代码覆盖率结果及分析	15
4	项目验证总结	17
5	分工情况	17

1 UVM 介绍

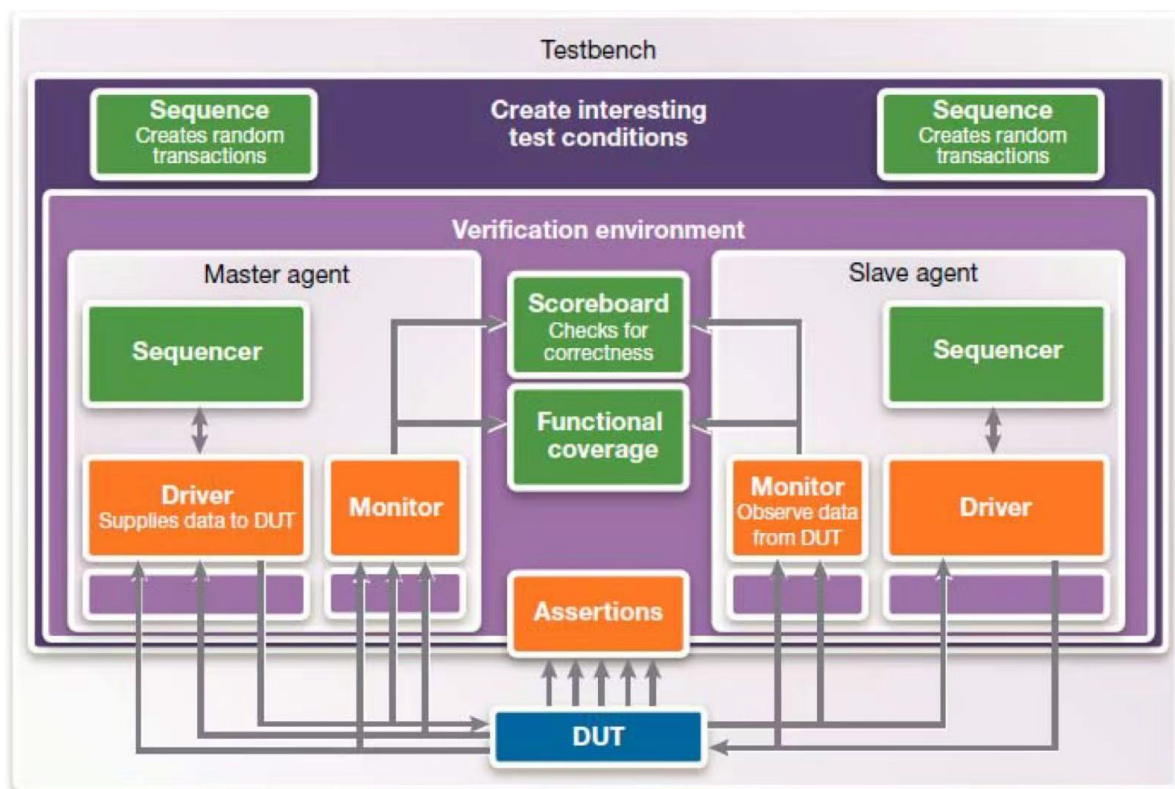


图 1: UVM 架构示意图

UVM (Universal Verification Methodology) 是一种基于 SystemVerilog 的标准化验证方法学, 由 Accellera 组织提出, 旨在提升设计验证的效率和可重用性。UVM 提供了一个统一的验证框架, 定义了标准的验证组件结构 (如 driver、monitor、scoreboard 等), 并支持高级验证功能, 如约束随机生成、功能覆盖和事务级建模。UVM 广泛应用于工业界, 是当前主流芯片设计验证流程中常用的验证方法。

UVM 基于面向对象的思想构建, 主要包括以下核心组件:

- **uvm_test:** 整个验证环境的入口, 定义测试场景。
- **uvm_env:** 验证环境的顶层容器, 封装多个 agent 和其他组件。
- **uvm_agent:** 代表一个总线或接口的验证代理, 通常包含 driver、monitor 和 sequencer。
- **uvm_driver:** 驱动 DUT 输入信号, 将 sequence item 转化为接口操作。
- **uvm_monitor:** 监控 DUT 的信号行为并采集信息供 scoreboard 或 subscriber 使用。
- **uvm_sequencer:** 与 driver 配合调度事务 (sequence)。
- **uvm_subscriber:** 接受 monitor 采样的事务, 并进行覆盖率分析。
- **uvm_scoreboard:** 进行结果的自检或对比, 用于功能正确性验证。

这些组件之间通过事务对象 (transaction) 进行数据传递, 使得验证环境高度模块化、可配置和可复用。

采用 UVM 进行验证相较于传统的手写 testbench, 具有以下优点:

- **高度模块化:** 各个组件职责明确, 方便管理和复用。

- 可扩展性强：支持面向对象的继承与多态，适应复杂系统的验证需求。
- 支持随机测试：内建约束随机激励能力，覆盖更多边界情况。
- 自动化功能覆盖分析：提升验证的完备性与收敛效率。
- 工业标准，生态完善：大量成熟 IP 及开源 UVM 验证库可以直接使用。

在本项目中，我们采用 UVM 搭建了模块的验证平台。通过构建 test、env、agent 等组件，实现了对 DUT 接口的激励、响应监控和结果校验。UVM 的约束随机生成机制帮助我们覆盖了大量 corner case，而 coverage 机制则用于评估验证的完备性。

UVM 作为业界主流验证方法学，极大提高了验证平台的开发效率与质量。在本项目中，UVM 的引入使我们验证流程更加规范、系统，且具备良好的可维护性和复用性，为后续项目积累了宝贵的经验。

2 UVM 验证平台详细说明

2.1 验证框架与验证流程

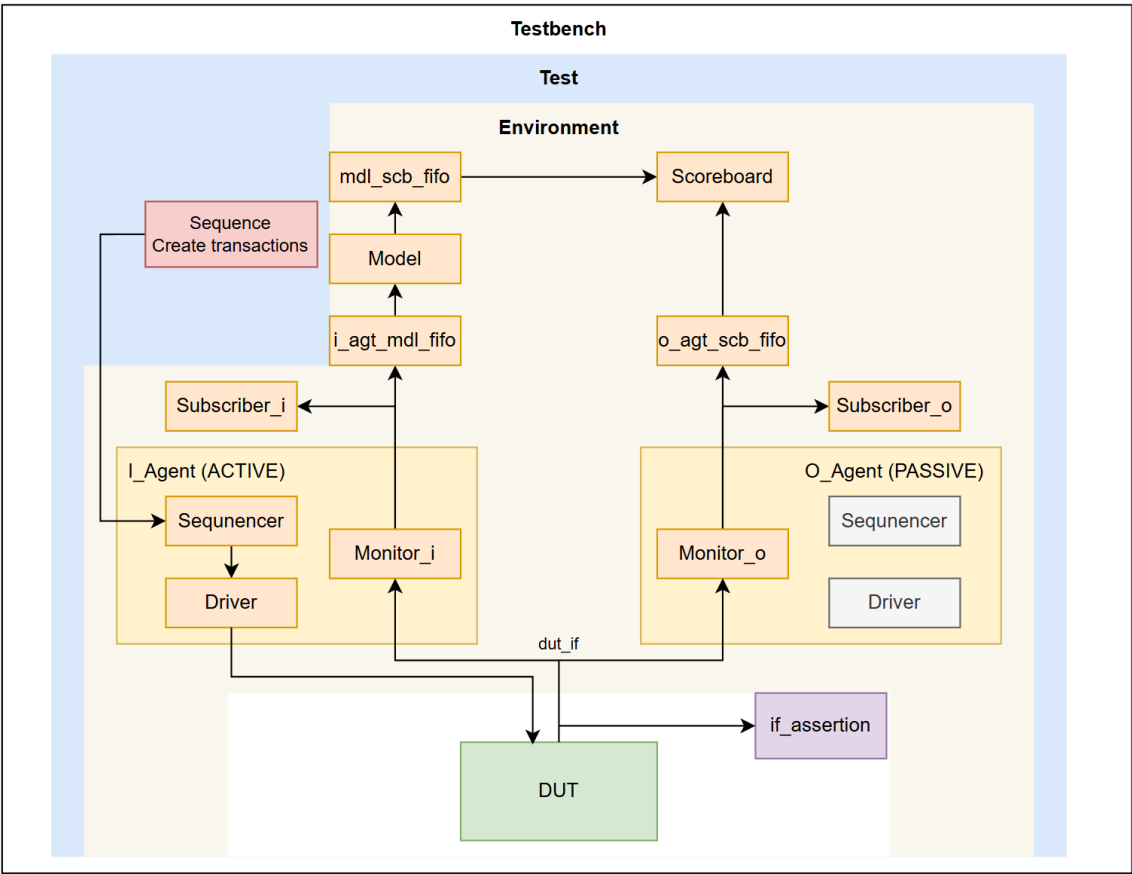


图 2: 本项目 UVM 验证平台架构示意图

本项目 UVM 验证平台架构示意图如上所示。

在本验证平台的架构设计中，我们将 Agent 模块划分为两类：主动型（Active）Agent 与被动型（Passive）Agent。其中，主动型 Agent 包含完整的验证组件结构，即 Sequencer、Driver 和

Monitor，其主要功能是主动向被测设计（DUT）施加激励，并同时采集响应信息。Sequencer 负责生成受约束的随机事务，Driver 将事务转化为具体信号驱动 DUT，而 Monitor 则负责对 DUT 的行为进行观测与数据采集。

相较之下，被动型 Agent 仅保留 Monitor 组件，用于对总线或接口的信号进行旁路观察和数据采集，其 Sequencer 和 Driver 被显式禁用（disable）。被动型 Agent 常用于不需要主动激励的场景，例如用于监听第三方模块或系统内部已有的激励路径，以实现无侵入式的协议检测与覆盖分析。

在仿真验证过程中，主动型 Agent 通过其内部的 Sequencer 生成受约束的随机事务激励，并由 Driver 将这些事务翻译为具体的信号操作，进而驱动 DUT（Design Under Test）执行功能计算。与此同时，系统中两个 Agent 的 Monitor 均对 DUT 的接口信号进行实时采集与分析，实现对 DUT 输入输出行为的全面监控。

在功能覆盖分析方面，主动型 Agent 采集的接口数据会被送入 Subscriber_i 模块，用于输入路径的覆盖率统计；同时，这些事务还会通过 i_agt_md1_fifo 传递至 Model 模块内部。在 Model 模块中，内建的参考功能模型（Golden Model）对接收到的输入事务进行仿真计算，生成对应的期望输出值，并通过 md1_scb_fifo 发送至 Scoreboard。

另一方面，被动型 Agent 监测到的 DUT 实际输出数据则被传入 Subscriber_o 模块，用于输出路径的覆盖率检查；同时，这些数据会通过 o_agt_scb_fifo 传递至 Scoreboard，作为 DUT 的实际输出值。

Scoreboard 作为整个验证平台中的关键比对单元，会将从 Model 接收到的期望值与从被动型 Agent 采集到的实际值进行一一比对。若所有比对项均符合预期，则输出验证通过信息"Check Passed!"；若存在任何不一致情况，则立即报告错误信息"Check Failed!"，并输出错误值的个数，然后触发 uvm_error，记录详细的错误日志以便后续调试与问题定位。

该验证流程实现了从激励生成、信号监测、模型预测到结果比对的完整闭环，确保了对 DUT 功能正确性的全面验证与高覆盖度评估。此类设计实现了验证环境中不同角色的合理分工与资源优化，有助于提升验证平台的灵活性和模块复用性。

2.2 DUT 顶层接口设计

在本项目的验证平台架构中，DUT（Design Under Test）的接口被抽象封装为一个名为 dut_if 的 interface 类型，以实现信号的统一建模与模块间的隔离。该接口在验证环境中被多处复用，分别用于功能驱动、行为监测及形式检查等目的。具体而言，dut_if 同时被两个 Agent 的 Monitor 模块连接，用于对 DUT 接口上的信号交互行为进行实时监测与采集；其中主动型 Agent 内部的 Driver 通过连接 dut_if，对 DUT 施加激励，驱动其输入信号。

在本验证平台中，Driver 相对于 DUT 承担主机（Master）的角色，负责主动向 DUT 发起事务请求并驱动相应信号；而 DUT 则处于从机（Slave）位置，被动响应外部输入。与此同时，Monitor 的职责是被动监听接口上的所有信号变化，以便采集用于覆盖率分析和结果比对，因此从 Monitor 的视角来看，其访问的所有端口均为输入方向（input）。为确保在不同模块访问同一个 interface 时能够正确表达信号方向，我们在 dut_if 中采用了 modport 机制进行了端口方向的区分。具体而言，为 Driver、Monitor 和 DUT 分别定义了对应的 modport，明确规定了每个模块

的信号集合及其方向属性，从而避免方向冲突，提高了接口定义的清晰性与复用性。具体而言如下所示：

```
// DUT
modport slave (
    ...
);
// Driver
modport master (
    ...
);
// Monitor
modport others (
    ...
);
```

为提升仿真建模的真实度，增强与实际硬件行为的一致性，我们在 `dut_if` 中引入了 `clocking` block 机制，对输入输出信号施加了一定的时序延迟。通过为信号指定驱动延迟（`output #delay`）与采样延迟（`input #delay`），可有效模拟时钟域下的信号变化特性，有助于捕捉潜在的时序问题并提高验证精度。如下所示：

```
// Master(Driver)
clocking mst_cb @(posedge clk);
    default input #1ns output #1ns;
    ...
endclocking : mst_cb

// Monitor
clocking mon_cb @(posedge clk);
    default input #1ns output #1ns;
    ...
endclocking : mon_cb
```

此外，为实现信号属性检查，`dut_if` 还通过 `bind` 机制与断言模块 `if_assertion` 进行绑定，使其能够在仿真过程中实时访问接口信号，并据此进行断言判断和验证规则监控。如下所示：

```
module dut #(
    parameter ID = 0
)
(
    dut_if    mhsa_if
);

dut_top u_dut_top
(
```

```

        .mhsa_if          (mhsa_if.slave          )
    );

    // assertion
    if (ID == 0)
    begin
        bind mhsa_acc_wrapper if_assertion
        if_assertion_bind_mhsa_acc_wrapper
        (
            .mhsa_if          (mhsa_if.others          )
        );
    end

endmodule : dut

```

此种接口抽象与模块绑定的设计方式，不仅增强了验证环境的可重用性和可扩展性，同时也提升了信号驱动与观察路径的清晰度，为后续验证覆盖率分析和调试定位提供了良好的支持。

2.3 UVM 对象要素列表

在本实验的 test 中，我设置了 3 个 Environment 环境进行独立测试。在 my_test 类的 connect_phase 添加如下代码可以打印出验证平台的拓扑图：

```
uvm_top.print_topology();
```

对验证平台以及 DUT 进行编译和仿真后，在 transcript 文件内，我们可以观察到验证平台的对象要素信息。以其中一个环境 env[0] 为例，对象要素列表如下所示：

```

# -----
# Name                                     Type                                     Size  Value
# -----
# uvm_test_top                             test0                                   -      @355
#   my_vsequencer_h                         my_vsequencer                           -      @425
#     rsp_export                           uvm_analysis_export                     -      @434
#     seq_item_export                       uvm_seq_item_pull_imp                   -      @552
#     arbitration_queue                     array                                   0      -
#     lock_queue                             array                                   0      -
#     num_last_reqs                         integral                                32     'd1
#     num_last_rsps                         integral                                32     'd1
#   my_env_h[0]                             my_env                                  -      @389
#     i_agt_md1_fifo                       uvm_tlm_analysis_fifo #(T)              -      @646
#     analysis_export                       uvm_analysis_imp                         -      @695
#     get_ap                               uvm_analysis_port                       -      @685
#     get_peek_export                       uvm_get_peek_imp                       -      @665
#     put_ap                               uvm_analysis_port                       -      @675

```

#	put_export	uvm_put_imp	-	@655
#	mdl_scb_fifo	uvm_tlm_analysis_fifo #(T)	-	@764
#	analysis_export	uvm_analysis_imp	-	@813
#	get_ap	uvm_analysis_port	-	@803
#	get_peek_export	uvm_get_peek_imp	-	@783
#	put_ap	uvm_analysis_port	-	@793
#	put_export	uvm_put_imp	-	@773
#	my_agent_i_h	my_agent	-	@572
#	my_driver_h	my_driver	-	@960
#	rsp_port	uvm_analysis_port	-	@979
#	seq_item_port	uvm_seq_item_pull_port	-	@969
#	my_monitor_i_h	my_monitor_i	-	@989
#	ap	uvm_analysis_port	-	@999
#	my_sequencer_h	uvm_sequencer	-	@823
#	rsp_export	uvm_analysis_export	-	@832
#	seq_item_export	uvm_seq_item_pull_imp	-	@950
#	arbitration_queue	array	0	-
#	lock_queue	array	0	-
#	num_last_reqs	integral	32	'd1
#	num_last_rsps	integral	32	'd1
#	my_agent_o_h	my_agent	-	@581
#	my_monitor_o_h	my_monitor_o	-	@1015
#	ap	uvm_analysis_port	-	@1024
#	my_model_h	my_model	-	@628
#	ap	uvm_analysis_port	-	@1045
#	bgp	uvm_blocking_get_port	-	@1035
#	my_scoreboard_h	my_scoreboard	-	@637
#	act_bgp	uvm_blocking_get_port	-	@1065
#	exp_bgp	uvm_blocking_get_port	-	@1055
#	my_subscriber_i_h	my_subscriber_i	-	@590
#	analysis_imp	uvm_analysis_imp	-	@599
#	my_subscriber_o_h	my_subscriber_o	-	@609
#	analysis_imp	uvm_analysis_imp	-	@618
#	o_agt_scb_fifo	uvm_tlm_analysis_fifo #(T)	-	@705
#	analysis_export	uvm_analysis_imp	-	@754
#	get_ap	uvm_analysis_port	-	@744
#	get_peek_export	uvm_get_peek_imp	-	@724
#	put_ap	uvm_analysis_port	-	@734
#	put_export	uvm_put_imp	-	@714

2.4 加速器验证计划

本项目的验证计划依据验证目标与检测机制的不同，系统地划分为三大类核心检查手段，分别为：断言检查（Assertion Check）、覆盖率检查（Coverage Check），以及功能正确性检查（Scoreboard Check）。各部分相辅相成，构成了覆盖全面、逻辑严密的验证体系。

- 断言检查（Assertion Check）：通过在设计接口及关键行为路径中绑定断言模块，使用 SystemVerilog Assertions（SVA）对时序关系、协议规则以及设计约束进行形式化描述，并在仿真过程中实时监控信号行为。一旦 DUT 违反预设条件，即触发断言失败，便可迅速定位潜在设计缺陷，有效提升验证的实时性和精确性。
- 覆盖率检查（Coverage Check）：覆盖率分析用于衡量验证激励的完整性和 DUT 行为的探索程度。我们在验证环境中引入了功能覆盖率（Functional Coverage），对关键事务特征、接口状态及协议场景进行建模，确保验证激励覆盖了预期功能空间；同时，启用仿真工具的代码覆盖率（Code Coverage）功能，从语句、分支、条件等多个维度评估设计代码在仿真中的触达程度，从而有效判断验证的充分性与收敛程度。
- 功能正确性检查（Scoreboard Check）：利用 Scoreboard 模块对 DUT 输出结果进行比对验证。具体流程为：参考模型（Model）根据激励数据计算出期望输出，而 DUT 的实际输出则通过被动监测路径采集；二者通过 Scoreboard 进行一一比对，如结果一致则判定为功能通过，若存在偏差则立即报告错误，保障 DUT 的逻辑功能满足设计预期。

详细的验证内容会在接下来的章节中进行详细介绍。

2.5 断言（Assertion）设计

本项目的断言模块实现了以下断言要求：

1. 每一个信号在其有效/使用时的 X 态检查。
2. 在 done 信号为高之前，start 信号的稳定性检查。
3. 在 done 信号为高之前，input_base 信号的稳定性检查。
4. 在 done 信号为高之前，output_base 信号的稳定性检查。

具体而言，实现形式如下所示：

```
// Signal X Assertion
property start_no_x_check;
    @(posedge mhsa_if.clk) disable iff (!mhsa_if.rst_n)
        not ($isunknown(mhsa_if.start));
endproperty : start_no_x_check
...
check_start_no_x: assert property (start_no_x_check) else $error($stime,
    "\t\t FATAL: 'start' exists X!\n");
...
// Signal Stable Assertion
property start_keep_check;
    @(posedge mhsa_if.clk) disable iff (!mhsa_if.rst_n)
        mhsa_if.start |-> mhsa_if.start until mhsa_if.done;
```



```

endproperty : start_keep_check

property input_base_keep_check;
  @(posedge mhsa_if.clk) disable iff (!mhsa_if.rst_n)
  (!mhsa_if.done) |-> $stable(mhsa_if.input_base);
endproperty : input_base_keep_check

property output_base_keep_check;
  @(posedge mhsa_if.clk) disable iff (!mhsa_if.rst_n)
  (!mhsa_if.done) |-> $stable(mhsa_if.output_base);
endproperty : output_base_keep_check

check_start_keep: assert property (start_keep_check) else $error($stime,
  "\t\t FATAL: 'start' is not stable!\n");
check_input_base_keep: assert property (input_base_keep_check) else
  $error($stime, "\t\t FATAL: 'input_base' is not stable!\n");
check_output_base_keep: assert property (output_base_keep_check) else
  $error($stime, "\t\t FATAL: 'output_base' is not stable!\n");

```

2.6 覆盖率 (Coverage) 设计

需要使用覆盖点 (Coverpoint) 进行测试的主要有以下内容:

1. soc_write_en 信号的状态覆盖, 代表了对加速器中 mem 的读写状态覆盖。

```

cov_soc_write_en : coverpoint soc_write_en {
  bins soc_write_en_0 = {0};
  bins soc_write_en_1 = {1};
}

```

2. start 信号的状态覆盖, 代表了加速器数据准备阶段与计算阶段的覆盖。

```

cov_start : coverpoint start {
  bins start_0 = {0};
  bins start_1 = {1};
}

```

3. soc_data_in 信号的数据类型覆盖, 代表了加速器对任意数据进行处理的能力覆盖。

```

cov_soc_data_in : coverpoint soc_data_in {
  bins all_zero = {64'b0};
  bins all_one  = {64'hFFFF_FFFF_FFFF_FFFF};
  bins inc_seq  = {[64'h1 : 64'h10]};
  bins random[] = default;
}

```

4. soc_addr 信号的范围覆盖，代表了对加速器内所有 mem 的访问覆盖。

```
cov_soc_addr : coverpoint soc_addr {  
    bins bar0 = {[32'h0000_0000 : 32'h0000_0FFF]}; // O, X  
    bins bar1 = {[32'h0000_1000 : 32'h0000_1FFF]}; // Q  
    bins bar2 = {[32'h0000_2000 : 32'h0000_2FFF]}; // K  
    bins bar3 = {[32'h0000_3000 : 32'h0000_3FFF]}; // V  
}
```

5. done 信号的状态覆盖，代表了加速器计算结束与否的状态覆盖。

```
cov_done : coverpoint done {  
    bins done_0 = {0};  
    bins done_1 = {1};  
}
```

2.7 参考模型 (Reference Model) 设计

参考模型 (Reference Model) 作为验证平台中实现期望行为的功能基准，其核心实现位于 Model 组件内部。该组件的主要职责是对 DUT 输入数据进行软件形式的功能建模，从而为后续的功能正确性比对提供准确的期望输出值。

在验证过程中，Model 组件通过与 Monitor_i 相连接的通信通道接收来自主动型 Agent 监测到的输入事务数据。这些数据被作为参考模型的输入，在 Model 内部完成对多头自注意力机制 (MHSA, Multi-Head Self-Attention) 算法的功能级模拟计算。计算完成后，模型生成的期望结果将通过结构化的接口发送至 Scoreboard，与 DUT 实际输出进行逐一比对，验证其行为是否符合设计规范。

为实现上述功能，Model 组件内部设计并实现了三个关键 function，分别为：

- store_input_data：用于接收并缓存从 Monitor_i 传入的原始输入事务数据，完成数据的预处理与格式化操作，作为后续计算的输入源。
- perform_mhsha_calculation：实现 MHSA 算法的核心计算逻辑，基于缓存的输入数据执行 MHSA 操作，输出匹配 DUT 行为的期望结果。
- write_output_data：将 MHSA 计算得到的期望输出组织为标准事务格式，并传递至 Scoreboard，供其进行与 DUT 实际输出的比对分析。

最终 Model 组件的 run_phase 被组织为如下的形式：

```
task run_phase(uvm_phase phase);  
    my_transaction tx_model_i;  
    my_transaction tx_model_o;  
  
    int transaction_count = 0;  
    logic [7:0] X[32][128];  
    logic [7:0] Wq[128][128];  
    logic [7:0] Wk[128][128];
```

```

logic [7:0] Wv[128][128];
logic [7:0] Wo[128][128];
logic [7:0] result[32][128];

forever begin
    // Modeling DUT behavior
    bgp.get(tx_model_i);
    // $display("model get data in : %h", tx_model_i.soc_data_in);
    tx_model_o = my_transaction::type_id::create("tx_model_o");
    tx_model_o.copy(tx_model_i);

    if (transaction_count < 512 + 2048 * 4)
    begin
        store_input_data(tx_model_i, transaction_count, X, Wq, Wk, Wv, Wo);
        transaction_count++;
    end

    if (transaction_count == 512 + 2048 * 4)
    begin
        perform_mhsa_calculation(X, Wq, Wk, Wv, Wo, result);
        write_output_data(result, tx_model_o, ap);
        `uvm_info(report_id, "MHSA calculation completed", UVM_LOW);
        transaction_count = 0;
    end
end
endtask : run_phase

```

2.8 随机化序列设计

在本次项目中，我们使用了三个独立的 Environment 对 DUT 进行测试，具体而言，三种验证环境分别对应如下测试场景：

1. 随机序列环境 (Random Sequence Environment)：在该环境中，激励通过受约束的随机机制生成，模拟真实应用中 mem 访问模式可能出现的不确定性和多样性。该测试场景主要用于验证 DUT 在输入数据分布无规律、地址跨越范围大等情况下的稳健性与正确性。

```

class mem_rand_write_seq extends uvm_sequence #(my_transaction);
...
endclass: mem_rand_write_seq

```

2. 自增序列环境 (Incremental Sequence Environment)：该环境中生成的事务数据呈现单调递增的顺序模式，如地址或数据依次递增。此类激励适合用于测试 DUT 在处理规律性数据访问时是否存在地址计算错误、数据偏移异常等问题，亦可验证内部缓冲与调度机制的连续性支持能力。

```
class mem_inc_write_seq extends uvm_sequence #(my_transaction);
...
endclass: mem_inc_write_seq
```

3. 全一序列环境 (All-One Sequence Environment)：在该环境下，输入数据固定为全 1 模式 (0xFFFFFFFF)，主要用于验证 DUT 在边界条件或特殊数据值（如最大值）下的功能正确性与稳定性，检测是否存在溢出、位翻转、无效掩码等潜在风险。

```
class mem_allone_write_seq extends uvm_sequence #(my_transaction);
...
endclass: mem_allone_write_seq
```

在 test package 中，test0 类的 run_phase 采用了 SystemVerilog 的 fork...join 并发语句结构，将三个独立的验证环境作为独立的并行进程进行调度与执行。

3 仿真流程及结果分析

3.1 基于 QuestaSim 的 UVM 仿真流程

首先在项目文件夹下打开 terminal，对代码文件进行编译，同时需要启用 cover 功能，sbceft 分别对应 Statement、Branch、Condition、Expression、FSM、Toggle 六种代码覆盖率，输入如下命令：

```
vlog +incdir+<questasim_dir>/verilog_src/uvm-1.2/src -cover sbceft ./
uvm_mhsa/my_top.sv
```

然后需要打开仿真，选择 test0，同时启用 assert 与 coverage 选项，输入如下命令：

```
vsim -gui work.my_top -novopt -assertdebug -coverage -sv_lib <questasim_dir
>/uvm-1.2/win64/uvm_dpi +UVM_TESTNAME=test0
```

然后输入如下命令运行仿真：

```
run -all
```

之后就可以在 Questasim 的控制台观察 Scoreboard 和 UVM reporter 的打印信息，以及可以在窗口内查看 Assertion 与 Functional Coverage 的统计信息。

最后，为了得到代码覆盖率的信息，我们可以在命令行输入如下命令将其输出为 txt 文件：

```
coverage report -output ./uvm_mhsa/report/coverage_report.txt
```

3.2 Scoreboard 结果及分析

对验证平台以及 DUT 编译并仿真后我们可以在 transcript 文件内看到 Scoreboard 以及 UVM reporter 打印的内容。如下图所示是 Scoreboard 打印的信息，从图中我们可以观察到三个验证环

境的 Scoreboard 输出了比对的结果，均为 “Check Passed!”，并且输出了接收到的事务总数为 512，比对失败的结果为 0。这里输出事务数目为 512 的原因是 DUT 设计中采用的是 4096×64 的 mem 大小，量化数据位宽为 8 bit，因此一个地址可以存放 8 个数据，而 DUT 进行 MHSA 计算后的结果是一个 32×128 的矩阵，因此事务数量为：

$$32 \times 128 \times 8 \div 64 = 512$$

```
# UVM_INFO .\uvm_mhsha\my_env_pkg.sv(323) @ 174105000: uvm_test_top.my_env_h[0].my_model_h [my_model] MHSA calculation completed
# UVM_INFO .\uvm_mhsha\my_env_pkg.sv(323) @ 208926000: uvm_test_top.my_env_h[1].my_model_h [my_model] MHSA calculation completed
# UVM_INFO .\uvm_mhsha\my_env_pkg.sv(323) @ 243747000: uvm_test_top.my_env_h[2].my_model_h [my_model] MHSA calculation completed
# UVM_INFO .\uvm_mhsha\my_test_pkg.sv(135) @ 448065000: uvm_test_top [test0] acc compute finished!
# UVM_INFO .\uvm_mhsha\my_env_pkg.sv(368) @ 453185000: uvm_test_top.my_env_h[0].my_scoreboard_h [my_scoreboard] Check Passed!
# UVM_INFO .\uvm_mhsha\my_env_pkg.sv(369) @ 453185000: uvm_test_top.my_env_h[0].my_scoreboard_h [my_scoreboard] Total number of transactions: 512
# UVM_INFO .\uvm_mhsha\my_env_pkg.sv(370) @ 453185000: uvm_test_top.my_env_h[0].my_scoreboard_h [my_scoreboard] Number of wrong transactions: 0
# UVM_INFO .\uvm_mhsha\my_test_pkg.sv(163) @ 537678000: uvm_test_top [test0] acc compute finished!
# UVM_INFO .\uvm_mhsha\my_env_pkg.sv(368) @ 543822000: uvm_test_top.my_env_h[1].my_scoreboard_h [my_scoreboard] Check Passed!
# UVM_INFO .\uvm_mhsha\my_env_pkg.sv(369) @ 543822000: uvm_test_top.my_env_h[1].my_scoreboard_h [my_scoreboard] Total number of transactions: 512
# UVM_INFO .\uvm_mhsha\my_env_pkg.sv(370) @ 543822000: uvm_test_top.my_env_h[1].my_scoreboard_h [my_scoreboard] Number of wrong transactions: 0
# UVM_INFO .\uvm_mhsha\my_test_pkg.sv(192) @ 627291000: uvm_test_top [test0] acc compute finished!
# UVM_INFO .\uvm_mhsha\my_env_pkg.sv(368) @ 634459000: uvm_test_top.my_env_h[2].my_scoreboard_h [my_scoreboard] Check Passed!
# UVM_INFO .\uvm_mhsha\my_env_pkg.sv(369) @ 634459000: uvm_test_top.my_env_h[2].my_scoreboard_h [my_scoreboard] Total number of transactions: 512
# UVM_INFO .\uvm_mhsha\my_env_pkg.sv(370) @ 634459000: uvm_test_top.my_env_h[2].my_scoreboard_h [my_scoreboard] Number of wrong transactions: 0
```

图 3: Scoreboard 打印内容

所有的验证环境计算完毕后，UVM reporter 输出 “TESS PASSED” 字样，如下图所示，可以看到 UVM ERROR 与 UVM FATAL 的数目均为 0，说明我们设置的 3 个 Environment 仿真均成功通过，DUT 功能正确。

```
##### TEST PASSED #####
#####
# UVM_INFO D:/questasim/questa_sim/verilog_src/uvm-1.2/src/base/uvm_report_server.svh(847) @ 735445000: reporter [UVM/REPORT/SERVER]
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 24
# UVM_WARNING : 0
# UVM_ERROR : 0
# UVM_FATAL : 0
# ** Report counts by id
# [RNTST] 1
# [TEST_DONE] 1
# [UVM/RELNOTES] 1
# [UVMTOP] 1
# [my_env] 4
# [my_model] 3
# [my_scoreboard] 9
# [my_test] 1
# [test0] 3
#
# ** Note: $finish : D:/questasim/questa_sim/verilog_src/uvm-1.2/src/base/uvm_root.svh(517)
# Time: 735445 ns Iteration: 67 Instance: /my_top
```

图 4: UVM reporter 打印内容

3.3 断言 (Assertion) 结果及分析

Assertion 窗口的部分截图如下所示，观察所有条目的 Failure Count 属性，均为 0，说明所有相关测试均通过，断言所测 DUT 功能正确无误。

File Name	Assertion Type	Language	Enable	Failure Count	Pass Count	Active Count	Memory	Peak Memory	Peak Memory Time	Cumulative Threads	ATV	Assertion Expression	Result
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_soc_write_en_no_x	Concurrent	SVA	on	0	61284	0	0B	80B	42000 ps	61284	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_input_base_keep	Concurrent	SVA	on	0	73540	0	0B	80B	35000 ps	73540	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_no_x	Immediate	SVA	on	0	172	0	-	-	-	172	off	assert (compiled_regexp==null)	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_soc_data_out_no_x	Concurrent	SVA	on	0	52529	0	0B	80B	49000 ps	52529	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_done_no_x	Concurrent	SVA	on	0	52529	0	0B	80B	49000 ps	52529	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_input_base_no_x	Concurrent	SVA	on	0	52529	0	0B	80B	49000 ps	52529	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_output_base_no_x	Concurrent	SVA	on	0	52529	0	0B	80B	49000 ps	52529	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_soc_write_en_no_x	Concurrent	SVA	on	0	52529	0	0B	80B	49000 ps	52529	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_no_x	Concurrent	SVA	on	0	8704	0	0B	80B	63000 ps	8704	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_soc_addr_no_x	Concurrent	SVA	on	0	8704	0	0B	80B	63000 ps	8704	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_soc_data_out_no_x	Concurrent	SVA	on	0	61284	0	0B	80B	42000 ps	61284	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_keep	Concurrent	SVA	on	0	27395	0	0B	2.2M	627291000 ps	27395	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_input_base_keep	Concurrent	SVA	on	0	52528	0	0B	80B	49000 ps	52528	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_output_base_keep	Concurrent	SVA	on	0	52528	0	0B	80B	49000 ps	52528	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_soc_data_in_no_x	Concurrent	SVA	on	0	61284	0	0B	80B	42000 ps	61284	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_soc_data_out_no_x	Concurrent	SVA	on	0	61284	0	0B	80B	42000 ps	61284	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_soc_data_in_no_x	Concurrent	SVA	on	0	61284	0	0B	80B	42000 ps	61284	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_soc_data_out_no_x	Concurrent	SVA	on	0	73540	0	0B	80B	35000 ps	73540	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_soc_data_in_no_x	Concurrent	SVA	on	0	8704	0	0B	80B	54000 ps	8704	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_soc_data_out_no_x	Concurrent	SVA	on	0	8704	0	0B	80B	54000 ps	8704	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_keep	Concurrent	SVA	on	0	1	0	0B	80B	537690000 ps	1	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_keep	Concurrent	SVA	on	0	27395	0	0B	2.2M	537678000 ps	27395	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_keep	Concurrent	SVA	on	0	61283	0	0B	80B	42000 ps	61283	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_keep	Concurrent	SVA	on	0	61283	0	0B	80B	42000 ps	61283	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_no_x	Concurrent	SVA	on	0	73541	0	0B	80B	35000 ps	73541	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_no_x	Concurrent	SVA	on	0	73541	0	0B	80B	35000 ps	73541	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_no_x	Concurrent	SVA	on	0	73541	0	0B	80B	35000 ps	73541	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_no_x	Concurrent	SVA	on	0	73541	0	0B	80B	35000 ps	73541	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_no_x	Concurrent	SVA	on	0	73541	0	0B	80B	35000 ps	73541	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_no_x	Concurrent	SVA	on	0	8704	0	0B	80B	45000 ps	8704	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_no_x	Concurrent	SVA	on	0	8704	0	0B	80B	45000 ps	8704	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_no_x	Concurrent	SVA	on	0	8704	0	0B	80B	45000 ps	8704	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_no_x	Concurrent	SVA	on	0	27395	0	0B	2.2M	448075000 ps	27395	off	assert (@posedge mhsa_f.clk) dis...	✓
/my_top/genblk1[0]u_dut/u_dut_top/mhsa_acc_wrapper/if_assertion_bind_mhsa_acc_wrapper/check_start_no_x	Concurrent	SVA	on	0	27395	0	0B	2.2M	448065000 ps	27395	off	assert (@posedge mhsa_f.clk) dis...	✓

图 5: Assertion 窗口

Assertion 波形的部分截图如下所示，可以观察到加速器的从数据准备阶段切换到计算阶段（start 信号拉高），以及从计算阶段切换到输出阶段（done 信号拉高）时各个信号的断言情况。

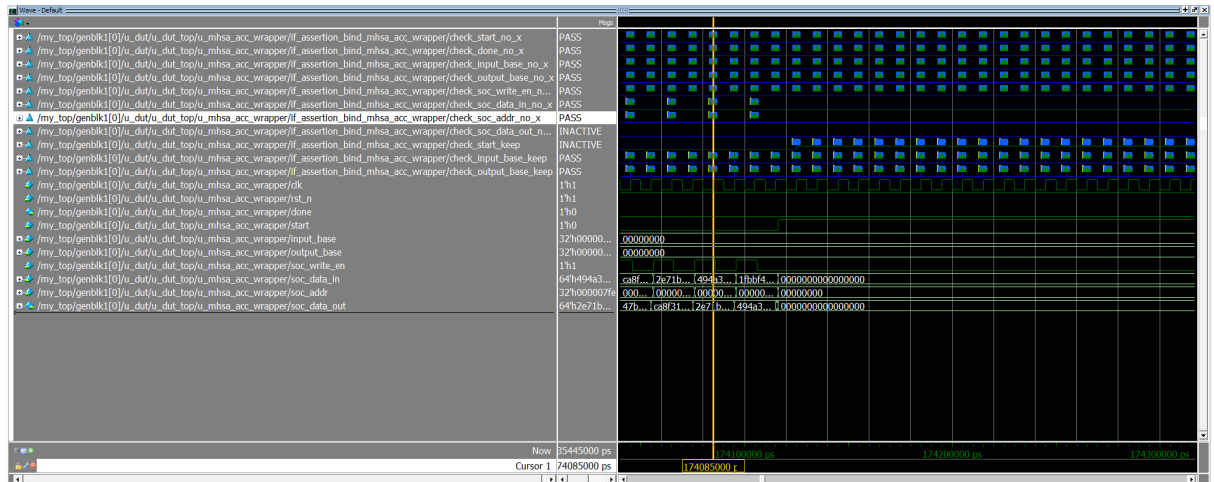


图 6: Assertion 波形（准备阶段-> 计算阶段）

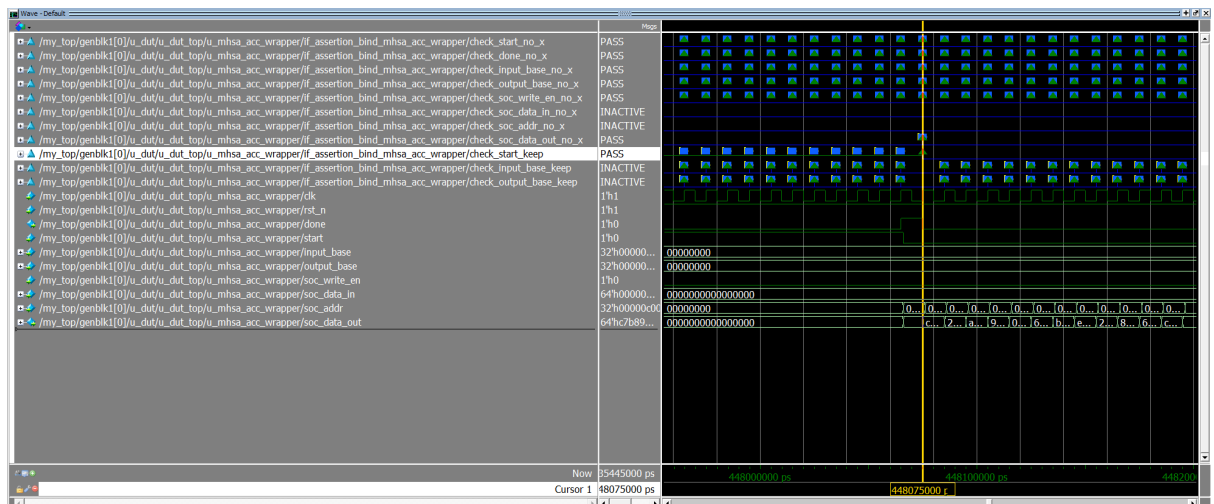
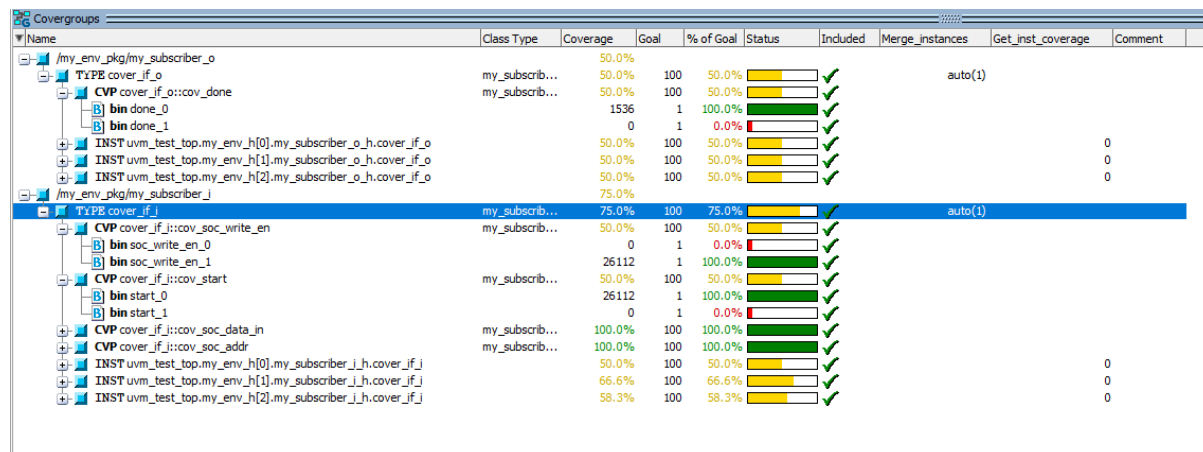


图 7: Assertion 波形（计算阶段-> 输出阶段）

3.4 覆盖率 (Coverage) 结果及分析

3.4.1 功能覆盖率结果及分析

Coverage 窗口的截图如下所示，图中带有 CVP 前缀的条目是 test0 的覆盖率，而带有 INST 前缀的条目则是各个 environment 的覆盖率。



Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_instances	Get_inst_coverage	Comment
/my_env_pkg/my_subscriber_o		50.0%							
TXPE cover_if_o	my_subscri...	50.0%	100	50.0%		✓		auto(1)	
CVP cover_if_o::cov_done	my_subscri...	50.0%	100	50.0%		✓			
bin done_0		1536	1	100.0%		✓			
bin done_1		0	1	0.0%		✓			
INST uvm_test_top.my_env_h[0].my_subscriber_o_h.cover_if_o		50.0%	100	50.0%		✓			0
INST uvm_test_top.my_env_h[1].my_subscriber_o_h.cover_if_o		50.0%	100	50.0%		✓			0
INST uvm_test_top.my_env_h[2].my_subscriber_o_h.cover_if_o		50.0%	100	50.0%		✓			0
/my_env_pkg/my_subscriber_i		75.0%							
TXPE cover_if_i	my_subscri...	75.0%	100	75.0%		✓		auto(1)	
CVP cover_if_i::cov_soc_write_en	my_subscri...	50.0%	100	50.0%		✓			
bin soc_write_en_0		0	1	0.0%		✓			
bin soc_write_en_1		26112	1	100.0%		✓			
CVP cover_if_i::cov_start	my_subscri...	50.0%	100	50.0%		✓			
bin start_0		26112	1	100.0%		✓			
bin start_1		0	1	0.0%		✓			
CVP cover_if_i::cov_soc_data_in	my_subscri...	100.0%	100	100.0%		✓			
CVP cover_if_i::cov_soc_addr	my_subscri...	100.0%	100	100.0%		✓			
INST uvm_test_top.my_env_h[0].my_subscriber_i_h.cover_if_i		50.0%	100	50.0%		✓			0
INST uvm_test_top.my_env_h[1].my_subscriber_i_h.cover_if_i		66.6%	100	66.6%		✓			0
INST uvm_test_top.my_env_h[2].my_subscriber_i_h.cover_if_i		58.3%	100	58.3%		✓			0

图 8: Coverage 窗口

通过观察带有 CVP 前缀的条目，可以发现对于 test0 而言，cov_soc_write_en，cov_start，cov_done 三个覆盖点的覆盖率为 50%，而其余覆盖点为 100%。

经过进一步分析验证环境的实现逻辑，我们发现该问题的根本原因在于 Monitor 的采样策略设计。当前版本的 Monitor 主要面向功能正确性比对的需求进行构建，其采样逻辑仅关注输入信号 soc_data_in 与输出信号 soc_data_out 之间构成的完整事务，用于驱动 Scoreboard 中的比对流程。因此，对于某些未形成完整事务的中间状态或无效信号变更（例如未完成写使能或未触发 start、done 事件）会被 Monitor 主动忽略，从而导致部分覆盖点未被采样覆盖。

尽管通过调整 Monitor 的采样逻辑可以提升覆盖率完整性，但由于 Monitor 的输出直接影响 Model、Scoreboard 等关键组件的数据路径，贸然修改将涉及较大范围的调整与修改，带来较高的开发与测试代价。

因此在这里我们通过观察波形的情况来补充没有 Subscriber 没有覆盖到的情况。我们确认仿真波形中确实存在这些覆盖点所对应的信号行为（如 soc_write_en, start, done 信号的有效翻转），但由于未被 Subscriber 捕捉，因此未反映在覆盖率报告中。该方法虽非自动化路径，但在本阶段验证工作中，足以作为补充依据，证明系统对这些覆盖点的行为已实际触达。

对于 soc_write_en, start, done 信号的翻转可以参见上一小节的 Assertion 波形图6和图7，在这两个波形图中我们可以清晰观察到这 3 个信号的翻转情况。

3.4.2 代码覆盖率结果及分析

观察 uvm_mhsa/report 文件夹下的 coverage_report.txt 文件，可以看到各个文件的代码覆盖率，每个文件有 Toggle、Statement、Branch、Condition、Expression、FSM Coverage 六个方面的覆盖率。

根据文件内的信息，我们可以得到如下表格 (Statements 和 Condition Coverage 均为 100%，故略去)：

File name	Branches (%)	Expression(%)	FSMs (%)	Toggle (%)
attmm.sv	100	100	83.3	54.9
connect.sv	100	100	83.3	60.9
linear.sv	100	100	83.3	60.9
mem_wk.sv	100	100	100	87.6
mem_wq.sv	100	100	100	87.6
mem_wv.sv	100	100	100	87.6
mem_x.sv	100	100	100	87.6
mhsa_acc_top.sv	100	100	79.1	56.3
mhsa_acc_wrapper.sv	98.4	75.0	100	84.0
mm_pe.sv	100	100	100	99.2
mm_systolic.sv	100	100	100	99.9
qkmm.sv	100	100	83.3	54.9
scale_core.sv	100	100	100	99.6
softmax.sv	100	100	100	67.2

从上表中，我们可以得到如下信息：

- 整体覆盖率表现良好：除了个别模块如 mhsa_acc_wrapper.sv 的表达式覆盖率略低（75%），其余模块在 Branches 与 Expression 两项指标上几乎都达到了 100%，表明验证激励对判断逻辑和布尔表达式的覆盖非常充分。
- FSM 覆盖整体较高但略有波动：大多数模块的 FSM 覆盖率达到或接近 100%，但 mhsa_acc_top.sv 和部分计算模块（如 attmm.sv、linear.sv、qkmm.sv）的 FSM 覆盖率约为 79% 或 83.3%，提示仍有个别状态转移路径在当前测试中未被完全触发。
- Toggle 覆盖率相对较低：相比其他指标，信号翻转（Toggle）覆盖率整体偏低，尤其是在 attmm.sv（54.9%）、qkmm.sv（54.9%）、mhsa_acc_top.sv（56.3%）等模块中较为明显，说明部分寄存器或信号在仿真过程中未充分切换，可能意味着部分路径或功能尚未被完全激活。
- mem 系列与矩阵乘核心模块表现优秀：如 mem_wk.sv、mem_x.sv、mm_systolic.sv、scale_core.sv 等模块在所有指标上均接近满覆盖，显示这些模块的验证效果较为充分，测试用例和激励策略较为完整。

同时我们观察到文件最后的统计信息，如下所示：

```
TOTAL COVERGROUP COVERAGE: 62.5%  COVERGROUP TYPES: 2
```

```
TOTAL ASSERTION COVERAGE: 100.0%  ASSERTIONS: 34
```

```
Total Coverage By File (code coverage only, filtered view): 74.1%
```

第一行 TOTAL COVERGROUP COVERAGE 为 62.5%，说明我们的设计功能覆盖率整体上

为 62.5%，具体内容在上一小节已经分析过，不再赘述。

第二行 TOTAL ASSERTION COVERAGE 为 100%，是指在 test0 中我们断言设计的覆盖率，说明我们的断言测试已经完备。

第三行 Total Coverage By File 为 74.1%，是指我们 dut 的代码覆盖率，该值理论上也可以达到 100%，说明我们的测试用例仍有改进的空间。

4 项目验证总结

本项目基于 UVM 方法学构建了模块化的验证平台，通过 Active 与 Passive 两类 Agent 配合，实现了对 DUT 输入输出接口的全面监控与驱动。验证架构中集成了参考模型、Scoreboard、Subscriber 等关键组件，并采用三种不同激励模式的 Environment 对 DUT 进行功能验证，有效覆盖了常见使用场景。

在验证过程中，我们通过断言、覆盖率和功能比对三类机制进行多维度检查。大部分覆盖点达到了 100% 的覆盖率，部分未完全覆盖的点经分析为 Monitor 采样策略所致，已通过波形验证补充确认。整体来看，验证平台结构合理、功能完备，满足项目对 DUT 功能正确性和健壮性的验证需求。

5 分工情况

姓名	负责内容	工作占比
黄超凡	验证方案制定，UVM 验证平台的搭建、调试与仿真，随机化与断言设计，验证报告撰写	1/3
张博文	加速器中 Softmax、Scale 模块的 RTL 实现，Model 组件功能模型实现，设计报告撰写	1/3
汪子尧	设计 SPEC 制定，加速器中 Linear 等其余模块的 RTL 实现及顶层设计，设计报告撰写	1/3