# GENERIC UVM FOR SOFT PROCESSORS

By

Kholoud Mahmoud Ebrahim Abdulrahman

Randa Ahmed Hussein Aboudeif

Karim Ayman Heweidy Gad

Mostafa Ayman Ibrahim Sarhan

Waleed Samy Abdelhameed Taie

Yasser Ibrahim Saleh Mohamed

A Graduation Project Thesis Submitted to
the Faculty of Engineering at Cairo University
in Partial Fulfillment of the Requirements for the
Degree of Bachelor of Science
in
Electronics and Communications Engineering

Under the supervision of

Associative Prof. Hassan Mustafa
Dr. Khaled Salah

Faculty of Engineering, Cairo University

Giza, Egypt

July 2020

## ACKNOWLEDGEMENTS

We would like to thank our supervisors: Dr. Hassan Mostafa, for his great support, supervision and encouragement. Moreover, we are very grateful to Dr. Khaled Salah for his valuable technical support, patient guidance, enthusiastic encouragement and useful critiques throughout the project's phases. Special thanks to our Fund resources: ONE Lab. (Cairo University), ASRT (the Academy of Scientific Research and Technology) and Mentor Graphics Egypt.

# TABLE OF CONTENTS

## LIST OF TABLES

# LIST OF FIGURES

# LIST OF LISTINGS

# ABSTRACT

In the modern era of CPU complexity advancements, Processor verification has always been an ever-increasing challenge. The gap between what a verification plan can offer nowadays and the current technology requirements is constantly widened. Despite of varies efforts on perfecting "Golden-verification-models" during the design phase, and adoption of object-oriented programming into the whole process; numerous industry experts still consider solo verification test benches as an extreme, time-consuming barricade that leads to a longer time-to-market and a questionable continuity of the current verification process. The Universal Verification Methodology (UVM), has come in action as a literal savior to the whole verification community, by offering a merge between SystemVerilog and SystemC into one environment that is completely standardized, constrained, and reusable, allowing a powerful verification methodology to a wide range of design sizes and types. The main contribution that this work introduces is implementing a generic UVM. In other words, building one verification environment that can be used to accommodate many RTL designs (Soft Processors), having not only different Instruction Set Architectures (ISAs) -of the same categories-, however additionally different techniques, and mechanisms handling the pipeline infrastructures. The proposed generic UVM (GUVM) structure permits the targeted user to attach any soft processor (core) having nearly the same micro-architecture to the proposed test bench, and to monitor both: CPU internal behavior and the complete flow of all supported instructions.

Keywords—UVM, SystemVerilog, Generic, Soft Processors, Cores, Verification Environment, Functional Verification, Simulation-based Verification, Instruction-based

Verification, Core-independent Verification.

# CHPATER 1: INTRODUCTION

Design Verification is simply the process of checking that a given design parameter correctly implements the target, specifications, and the required functionality [1]. Traditionally, 70% of a chip development cycle is dedicated to design verification. Based on that, the verification to design team ratio ranges from 2:1 to 3:1; that major overhead in the whole process lead to a new trend in the verification arena: striving towards *standardized*, and *reusable* test benches. That is when newer methodologies have been introduced like the Open Verification Methodology (OVM) and the Verification Methodology Manual (VMM), both have been actually fine for a while, however the industry is in need of a non-proprietary approach; thus finally in 2011, major technology giants joined together through Accellera and created the Universal Verification Methodology (UVM) [2].

UVM has opened new horizons in the verification world, and its highly configurable features have made the *generic* proposal really possible; as the old saying goes: "You can never go wrong with an object-oriented-based line of code". Normally a System on Chip (SOC) can be verified effectively using a simulation testbench that provides data to the SoC inputs and checks resulting data at the SoC outputs. The problem is that running all the possible testing scenarios is computationally impossible. On the other hand, a modern testbench-based verification environment automatically generates randomized stimulus for the SoC inputs under control of user-specified constraints and checks the results of each test automatically. UVM is the best verification approach that has been created to develop constrained-random testbenches in a uniform fashion and to permit limited reuse of testbench components [3].

In this work, an attempt to extend this reusability concept furthermore, by applying the same UVM environment to a plethora of different Designs Under Test (DUTs) implementation, with completely different architectures and some limitations on the common functions among different DUTs.

## 1.1 Problem Definition

It becomes possible for electronic system designers to assemble complete systems-on-chips due to the ever-increasing advances in the integrated circuit technology. At the same time, shrinking time-to-market leaves a small room for errors in the design development. Therefore, the verification process with all its stages (pre-silicon verification, post-silicon validation and runtime verification) has become one of the main tasks during the design-to-fabrication process, Thus, bugs are detected and fixed at early design flow stages [3].

Both design and verification nowadays are pushing towards reusable environments, and correspondingly the need of a standalone, and pre-verified verification infrastructure has arisen, to ensure that the verification step is not the bottleneck of the design flow.

UVM is being used as it has improved verification quality due to the constrained random verification. Test benches are reusable as verification components or agents get instantiated within a verification environment inside a project, and they may require some modification to suit the requirements of this verification environment. In addition to that, the overall verification environment could be used and modified according to the requirements of a certain test [2].

The problem of verifying soft processors using separate modified verification environments or different approaches still exists, and does cost the product cycle a long computational time and effort to get into the market. As a result, there is a great demand for a reusable generic environment used to verify soft processors with as minimum modification as possible; in order to save effort, time and cost.

## 1.2 Related Work and Contributions

For the best knowledge of the authors, this is the first Contribution in verifying different soft processors (cores) using only one generic and reusable UVM (the same UVM test bench without any tweaking); all the related works are based on the idea of implementing a UVM test bench to verify only one DUT (it can be reused however after tweaking the main components of the test bench itself).

In this work, the objective is to design and implement only one generic UVM that is used to verify the functionality of different soft processors (cores). These different cores are based on different instruction set architectures (ISAs), and they have different infrastructures and specifications such as, the number of the pipeline stages, and the behavior of the cache memories.

The proposed generic UVM attempts to prove this objective are on three open-source cores:

- RI5CY core, based on RISC-V ISA with four pipeline stages.
- LEON 2.4 core, based on SPARC-V8 ISA with five pipeline stages.

- Amber a23 core, based on ARM-V2a ISA with three pipeline stages.

The generic UVM proposed, can additionally be used to verify any similar soft processor based on one of the three previously mentioned ISAs, and has nearly the same specifications.

## 1.3 Organization of the remaining chapters

In Chapter 2, the history and the background about the UVM in general and the three proposed open-source cores are discussed. In Chapter 3, the proposed solution and methodology are explained. In Chapter 4, the proposed implementation is investigated in details, followed by an illustrative example in Chapter 5. in Chapter 6, the validation and the results of the proposed work are presented.  Finally, in Chapter 7, conclusion and future work directions are drawn.

# CHPATER 2: HISTORY AND BACKGROUND

## 2.1 UVM [4]-[12]

The most common term in verification is known as *functionality testing* or *functional verification* which is the process of demonstrating functional correctness of a processor design with respect to its specifications, this process is preceded by creating a verification plan that defines: which properties and functionalities need to be verified, different methods and approaches that will be used in processor testing, the expected behaviour of an appropriate design, defining functional coverage models and functional specifications of the verification, and finally the testing strategy; such major decisions must be taken in the verification planning phase [4].

Due to complex aspects of an IC design, these processes tend to be very challenging; during the past decade alone, average time spent by verification engineers to verily the complete functionality of their designs wasted more than 60 percent of the total design time. Even developers of smaller chips and FPGAs designs are facing difficulties with former verification approaches. The wanted goal of verification is becoming more difficult to achieve using conventional verification techniques, and hence solving this issue requires a detailed review of common testing methodology.

Directed testing was more convenient for testing single functionalities, however, it is hard to hit more complex scenarios using only directed testing. On the other hand, constrained-random verification (CRV) can be very efficient in tackling processor verification challenges, such as: complex instruction sets, multiple pipeline-stages, in order or out of order execution strategies, instruction parallelism, and multi precision operations. The most important module of a CRV environment is the test-case generator, which plays a very significant role in most of the recent approaches towards developing automated processor verification environments. A test-case generator generates a large set of valid test cases in a pseudo-random way, controlled and guided by constrained randomness. The development of such test generators has started to get the attention of functional verification engineers, and researchers since the early 2000s. However, the development of these generators has been categorized as a software problem due to poor and weak features of Hardware Description Languages (HDLs) available back then in terms of verification and software, like Verilog and VHDL [4].

However, recent efforts have been spent towards the utilization of SystemVerilog features as a Hardware Verification Language (HVL) to improve stimulus generation quality; and then UVM gradually dominated the verification world, as it covers these needs. UVM is a powerful verification methodology that was designed to be able to verify a variety of different design sizes and design types that could be in Verilog, SystemVerilog, VHDL, and SystemC code. It is an open source SystemVerilog library allowing creation of flexible, reusable verification components and assembling powerful test environments utilizing constrained random stimulus generation and functional coverage models [8]. It is based upon the three C’s of random verification: [12]

1- Checkers: As long as the stimulus is automated, in addition you have to write self-checking test benches in SystemVerilog.

2- Coverage: The question “Are we done yet?” have to be answered, as in addition is known as “Functional Coverage”, it is about recording the progress during a verification run and identifying how thoroughly the design have been exercised.

3- Constraints: What if holes have been covered, or if the design have not been exercised thoroughly enough. That is where constraints come into play, the constraints have to be increased on those random vectors in order to increase test coverage.

A UVM test bench is composed of verification components that are encapsulated, reusable, ready-to-use, and configurable elements; checking an interface protocol, a design sub-module, or a full system. The architecture of each component is logical. It includes a complete set of elements enabling the stimulation, check and collection of coverage information related to the specific protocol or design. This test bench instantiates the Design under Test module and the UVM Test class, then configures all connections between them. Module-based components are instantiated under the UVM test bench as well. The UVM Test is dynamically instantiated at run-time, allowing the UVM test bench to be compiled once and run with many different tests. All complex test benches may be architected as shown in Fig. 1 with little or more modification depending on the design complexity. The proposed implementation will be explained in details during Chapter 3.

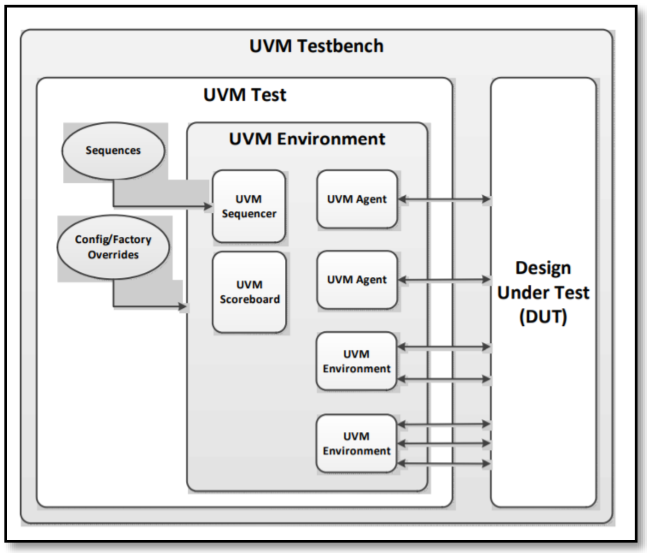


**Fig. 1 UVM block diagram.**

The overall verification environment can be tweaked by individual separately-written tests. It is typically a small piece of UVM code and that test would then customize the behavior of the complete verification environment in order to direct it in some way to try to test some particular feature of the DUT (interesting case).

## 2.2 Soft Processors [13]-[24]

A soft microprocessor (or a soft core) is considered Register transfer logic (RTL) code that describes a specific design and capable of executing some sort of an instruction set. This code can then be synthesized into a net list and mapped onto a programmable logic such as FPGA. Unlike hard processors which are physically implemented as a structure in silicon. The main advantage of soft processors is the higher configurability, and adjustability, as all features are written in code and thus its instruction set could be easily extended, modified and altered; on the other hand, increasing those capabilities will result in much more waste of resources and FPGA area, in addition to consuming more power while running at lower speeds as they are limited by fabric speed.

Nonetheless, some soft processors RTL is open source for research and development purposes, and that aligns perfectly with the used DUTs' choices as follows:

**1-** RISCY Core: A 4-stage 32-bit in order processor core based on RISC-V ISA under Solderpad license. In addition, some extensions were added to support embedded processing, hardware loops and advanced ALU instructions that are not included in standard RISC-V. It was first introduced in 2013 by Luca Benini as a collaboration between the University of Bologna and Zurich; this research work involved a team of 50-60 members with concentration on programmable systems that need to be flexible, scalable and does not waste energy. This core is fully written in Verilog and uses GCC as a compiler, with native support for interrupt and hardware synthesis. The core under consideration runs at 500 MHz, can interface with all basic peripherals like SPI, I2C, UART & JTAG Debug interface, and interfaces with both AXI/APF for high speed and low speed peripherals respectively [20], [21], and [22].

**2-** Leon 2.4: A 5-stage 32-bit processor core based on SPARC V8 ISA under GPL license. This project was initiated by the European Space Agency (ESA) in 1997 to develop a high-performance processor for their projects and future requirements. Later, it was developed even more for embedded applications with some features like: support for multi-core systems, separate data and instruction caches, hardware-based multiplier and divider, a memory management unit, interrupt controller, and finally an on-chip AMBA bus. This core is entirely written in VHDL, and in addition support both FPGA and ASIC synthesis [23].

**3-** Amber a23: A 3-stage 32-bit processor core based on ARM V2a ISA under LGPL license and hosted on the OpenCores website. It was developed in Verilog2001 and optimized for FPGA synthesis. Amber 23 is capable of 0.75 DMIPS per MHz, has a unified instruction and data cache, and communicates its data through wishbone bus interface. It can boot a Linux 2.4 kernel and its project does include a pre-made FPGA initialization, as well as peripheral support for UART, timer controller, interrupt controller, a test module and an Ethernet MAC [24].

# CHPATER 3: THE PROPOSED SOLUTION AND METHODOLOGY

Building a generic environment to verify different soft processors becomes a great demand nowadays, however to build an efficient and professional test bench, proper verification steps must be done, and followed in an effective manner to get the required results. Therefore, the proposed solution steps are as following:

1) **Understanding the design specifications:** studying the architecture of the desired DUT to be verified using the available data sheets and documentations. The case study here is on the three different open-source cores; those are previously mentioned before [25].

2) **Creating a verification plan:** based on the study done on the DUTs, a general verification plan is set, which consists of four main layers as shown Fig. 2:

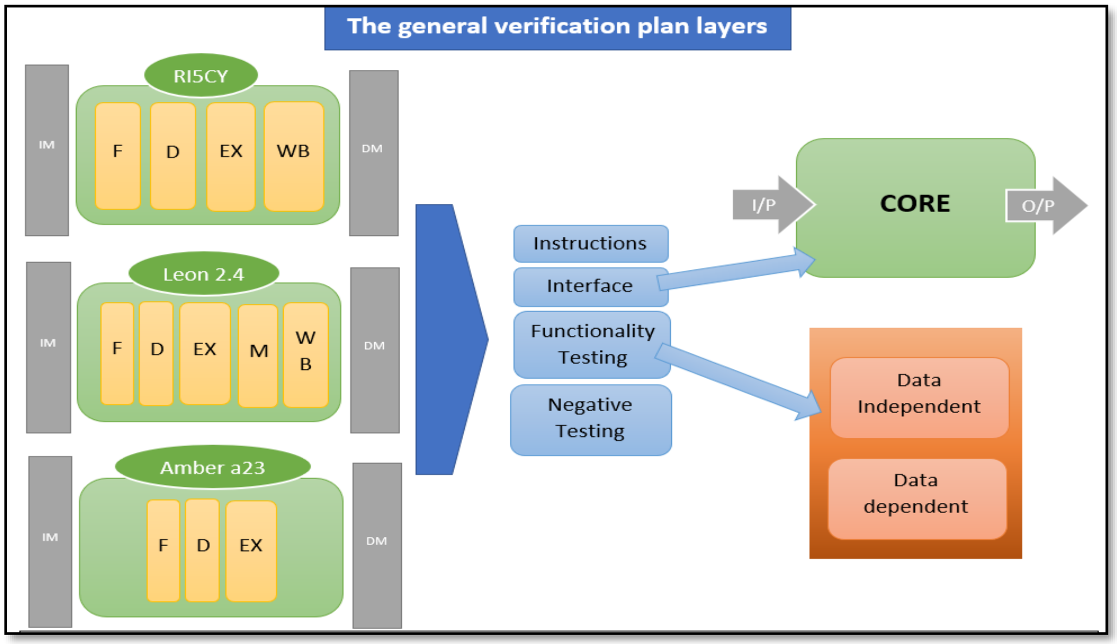


**Fig. 2 The general verification plan layers**

- ***The instructions layer:*** Where the specifying of instructions needed to be verified (functional verification) for each core (What to verify?).
- ***The interface layer:*** Where the determination of the I/O ports of the core in a black-box approach (I/O ports of the top-level module of the core).
- ***The functionality testing layer:*** Where the instructions needed to be verified are divided in a table into functions, then each function is divided into a group of instructions with different types, followed by a description of the test applied on each instruction in this group with the coverage bits detailed division. An example of the functionality testing table for the "add" instruction is shown in TABLE I (How to verify?) [26], [27].
- ***The Negative testing layer:*** In which the core is fed with invalid input data to check if it is behaving as expected or not, for invalid input data; whether the test bench shows any error message when it is supposed to and does not show any error message when it is not supposed to. In other words, checking that to what extent the core keeps itself stable in different situations for invalid input data for which it is not designed.

3) **Identifying the verification methods required:** In the proposed solution, the Simulation Based Verification is used, where constrained-random stimuli are generated for the design under test, then a golden reference (the scoreboard) is used to generate the expected output, which is then compared to the actual output determining the validity of the design functionality. (What should be?)
4) **Building the verification environment** (this part will be covered in an abstraction level in the next section of this chapter, and in great details in Chapter 4).

5) **Executing the plan**: Developing and running the tests, finding bugs, and debugging to make sure that the bug is a flaw in the DUT, not a problem with the verification environment itself.

**Table 1 Functional Testing Description.**

| **Func.** | **Inst.** | **Cores** | | |
|---|---|---|---|---|
| | | **Leon 2.4** | **Ri5cy** | **Amber a23** |
| Arith. | Add | Testing that the output register x[rd] has the result of adding register x[rs2] and x[rs1]. | Testing that the output register x[rd] has the result of adding the contents of register x[rs2] and register x[rs1]. Arithmetic overflow is ignored. | Testing that the output register x[rd] has the result of adding register x[rn] and x[rm]. |
| | | *Inst[31:30]→<br>[2'b10]<br>*Inst[29:25]→<br>[5'bxxxxx]<br>*Inst[24:19]→<br>[6'b'000000]<br>*Inst[18:14]→<br>[5'b'xxxxx]<br>*Inst[13]→<br>[1'b0]<br>*Inst[12:5]→<br>[8'b00000000]<br>*Inst[4:0]→<br>[5'bxxxxx] | * Inst[31:25]→<br>[7b'0000000]<br>*Inst[24:20]→<br>[5'bxxxxx]<br>*Inst[19:15]→<br>[5'bxxxxx]<br>*Inst[14:12]→<br>[3'b000]<br>*Inst[11:7]→<br>[5'bxxxxx]<br>*Inst[6:0]→<br>[7'b0110011] | * Inst[31:28]→<br>[4'bxxxx]<br>*Inst[27:20]→<br>[8'b00001000]<br>*Inst[19:16]→<br>[4'bxxxx]<br>*Inst[15:12]→<br>[4'bxxxx]<br>*Inst[11:4]→<br>[8'b00000000]<br>*Inst[3:0]→<br>[4'bxxxx] |

The main goal of the proposed methodology is to obtain a generic verification environment using a generic UVM test bench to verify different soft processors. Fig. 3 shows the hierarchy of the proposed generic UVM test bench (this hierarchy will be explained and covered in details in Chapter 4).

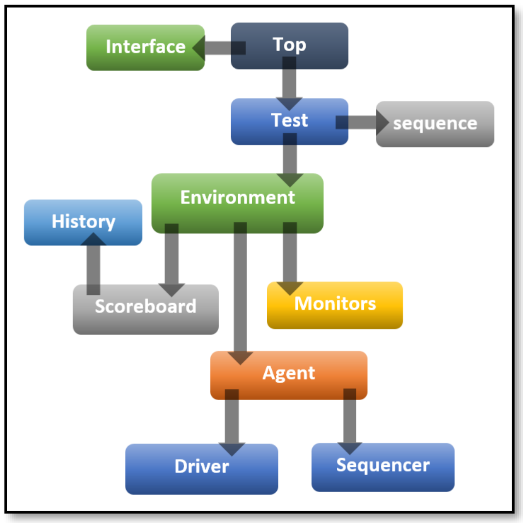


**Fig. 3 The generic UVM test bench hierarchy.**

The flow of the hierarchy works as following:

- The top module instantiates the test and the interface.
- The test instantiates the environment and the sequence (which is extended from the parent sequence)
- The environment instantiates the agent, the scoreboard, and the monitors.
- The scoreboard instantiates the history.
- The agent instantiates the sequencer and the driver.

The proposed verification approach allows the designated user to use the proposed generic UVM with any soft processor of the three previously mentioned open-source cores (or any available core with the same instructions and with a similar mechanism) after attaching, and connecting a few things to the test bench:

1. **The DUT itself** (RTL implementation code).
2. **The Core Package** (and its related files)**:** Includes all the core instructions, the format mapping of each one of the instructions, and some core specific defines.

3. **The Core Interface:** Includes input and output ports for the top-level-module of the core, in addition to pre-defined functions that the driver uses to drive instructions or data to the DUT.
4. **The Sequence:** Where the determination of the sequence, and behavior of how to verify the instructions. This child sequence inherits a parent shared, and pre-programmed sequence file (in the sequencer).
5. **Running the Test file**: Every child sequence has its own test to be instantiated, and triggered.
6. Header files connected to the scoreboard contain the test cases of the instructions needed to be verified that are included in the core package.

# CHPATER 4: IMPLEMENTATION

In this chapter, the implementation of the proposed generic UVM (GUVM) environment will be discussed in details.

Fig. 4 shows the hierarchy of the proposed implementation.

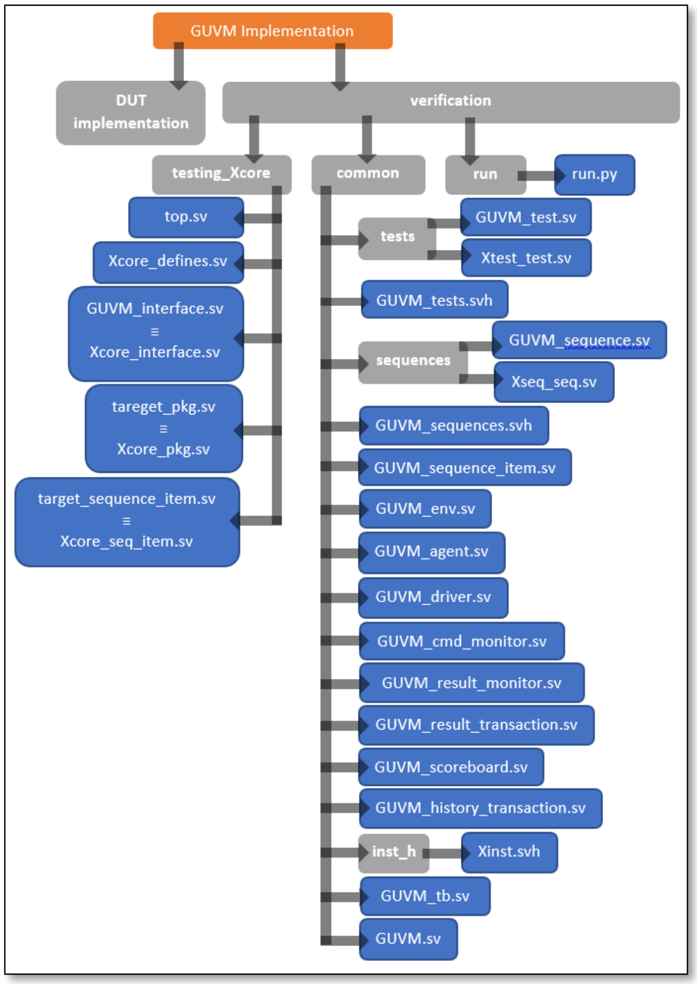


**Fig. 4 The generic UVM implementation.**

As the hierarchy consists of two main folders as following:

- **DUT implementation:** The RTL implementation code of the DUT needed to be verified (Xcore).

- **Verification:** Where the actual code of the test bench is organized into three main sub-folders as following:
    - ***common:*** Where the main generic components of the test bench are instantiated and triggered.
    - ***testing_X:*** Where the DUT specific files are instantiated and triggered.
    - ***run:*** Where running the test bench using a python scripts take place with a great flexibility.

Following, the contents of each sub-folder will be discussed in details:

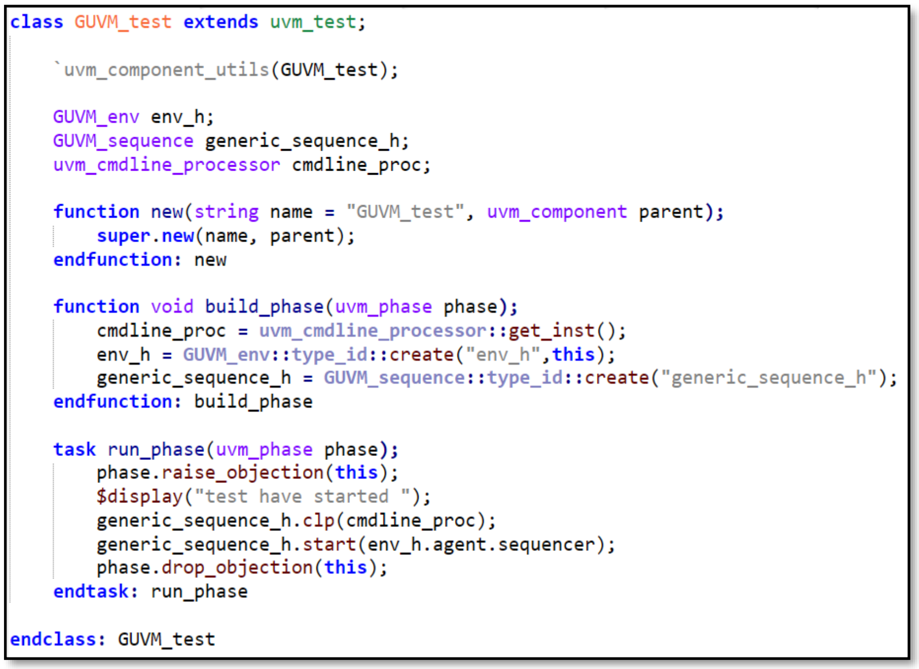
## 4.1 common:

It consists of a number of files and sub-sub-folders connected to each other to form the generic components of the proposed test bench:

**a.** ***tests:*** a sub-sub-folder contains a number of files (child tests) to run with the proposed test bench such as, "add_test.sv", all these child tests are extended from a parent test: "GUVM_test.sv":

**1.** *"GUVM_test.sv":* Where the parent test (GUVM_test) is extended from the (uvm_test) base class, it is registered with the UVM factory, then the constructor of the new class is declared. The generic environment (GUVM_env), the generic parent sequence (GUVM_sequence), and the UVM command line processor class (uvm_cmdline_processor) are declared to be instantiated and created at the build phase as shown in Listing 1. Then the objection has been raised at the run phase to let (uvm_cmdline_processor) get the required arguments, and start the sequencer responsible for sending the transactions or the sequence items to the driver (GUVM_driver). More details related to the arguments of (uvm_cmdline_processor) are in Section 4.1.b.1, and more details related to the sequencer, and the generic driver (GUVM_driver) are in Section 4.1.e, and Section 4.1.f.

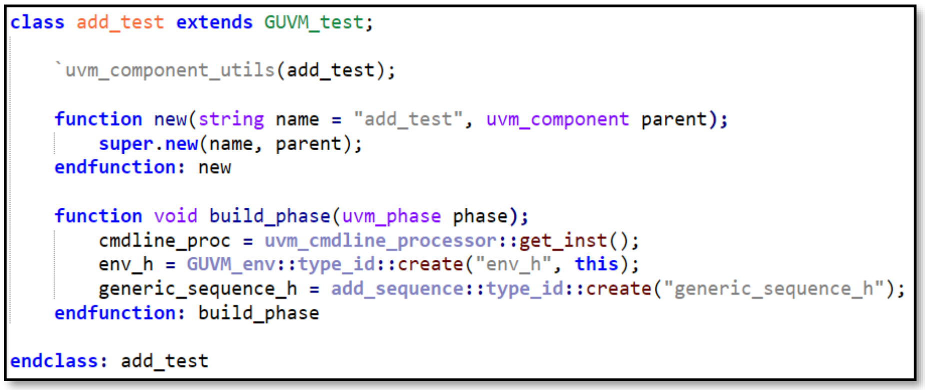
**Listing 1 The parent test file (GUVM_test).**



```
class GUVM_test extends uvm_test;

    `uvm_component_utils(GUVM_test);

    GUVM_env env_h;
    GUVM_sequence generic_sequence_h;
    uvm_cmdline_processor cmdline_proc;

    function new(string name = "GUVM_test", uvm_component parent);
        super.new(name, parent);
    endfunction: new

    function void build_phase(uvm_phase phase);
        cmdline_proc = uvm_cmdline_processor::get_inst();
        env_h = GUVM_env::type_id::create("env_h",this);
        generic_sequence_h = GUVM_sequence::type_id::create("generic_sequence_h");
    endfunction: build_phase

    task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        $display("test have started ");
        generic_sequence_h.clp(cmdline_proc);
        generic_sequence_h.start(env_h.agent.sequencer);
        phase.drop_objection(this);
    endtask: run_phase

endclass: GUVM_test
```

**2.** *<u>"Xtest_test.sv":</u>* Where this child test is extended from the generic parent test (GUVM_test), it is registered with the UVM factory, then the constructor of the new class is declared. The generic environment (GUVM_env), the child sequence (GUVM_sequence) and the UVM command line processor class (uvm_cmdline_processor) at the build phase are instantiated and created. Listing 2 shows the child test (add_test) used with the child sequence (add_sequence). More details related to the arguments of (uvm_cmdline_processor) are in Section 4.1. b.1.

**Listing 2        A child test file (add_test).**



```
class add_test extends GUVM_test;

    `uvm_component_utils(add_test);

    function new(string name = "add_test", uvm_component parent);
        super.new(name, parent);
    endfunction: new

    function void build_phase(uvm_phase phase);
        cmdline_proc = uvm_cmdline_processor::get_inst();
        env_h = GUVM_env::type_id::create("env_h", this);
        generic_sequence_h = add_sequence::type_id::create("generic_sequence_h");
    endfunction: build_phase

endclass: add_test
```

3. *"GUVM_tests.svh":* Where the files of all the parent and child tests are included.

**b.** ***sequences:*** A sub-sub-folder contains a number of files (child sequences) that get instantiated by the child tests previously such as, "add_seq.sv", all these child sequences are extended from a parent sequence: "GUVM_sequence.sv":

1. *"GUVM_sequenec.sv":* Where (GUVM_sequence) is extended from the (uvm_sequence) base class, it is registered with the UVM factory, then the constructor of the new class is declared. As shown in Listing 3, it contains a group of functions and tasks as following that can be called in the child sequences:

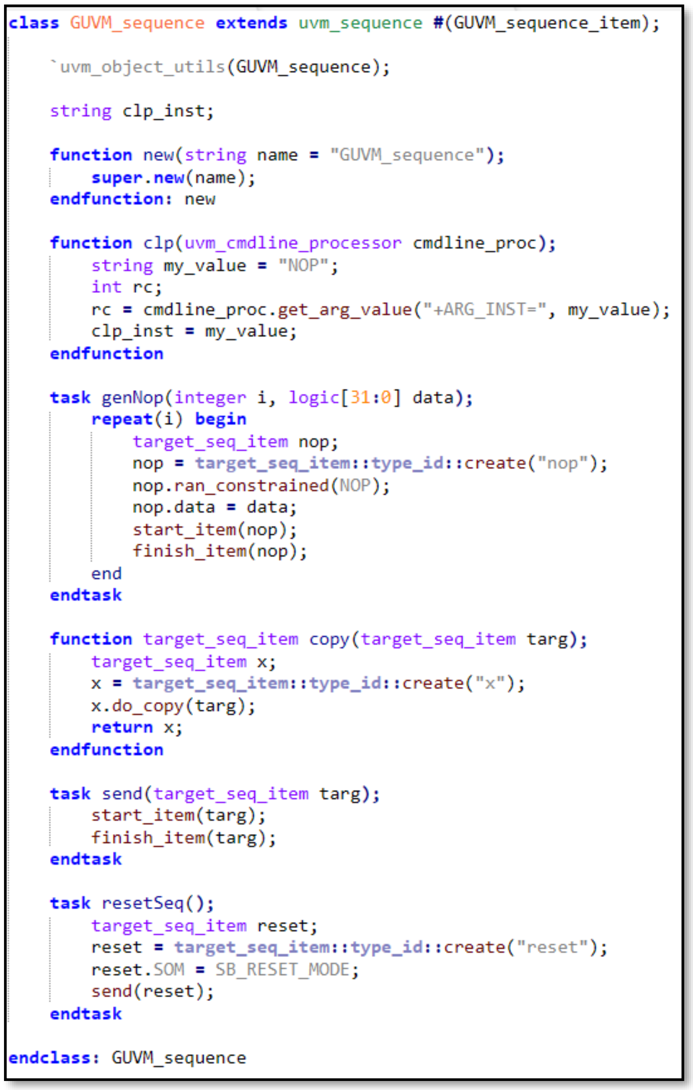
- ***clp():*** After getting the argument using (uvm_cmdline_processor), this argument such as, "A" (for the addition instruction) will be stored in a string type variable to be used with the child sequences later. More details related to the arguments of (uvm_cmdline_processor) are in Section 4.3.

- ***getNop():*** Function used by the child sequences to send (i) number of no operation instruction (NOP) with the sequence item (nop), which is instantiated, and created by the child sequence item (target_seq_item).

More details related to the target sequence item (target_seq_item) are in Section 4.2.e.

- ***send():*** Function used by the parent sequence (called inside another function), or by the child sequences to send any sequence item as (target) sequence item which is instantiated, and created by *the target sequence item* (target_seq_item). More details related to the target sequence item (target_seq_item) are in Section 4.2.e.
- ***copy():*** Function *used by the child sequences to make a* copy of the sequence item (target) to a new sequence item (x), which is instantiated, and created by (target_seq_item). More details related to the target sequence item (target_seq_item) are in Section 4.2.e.
- ***resetSeq():*** Function used by the child sequences to reset the generic history transaction (GUVM_history_transaction). More details related to the generic history transaction (GUVM_history_transaction) are in Section 4.1.k.

**Listing 3 The parent sequence file (GUVM_sequence)**



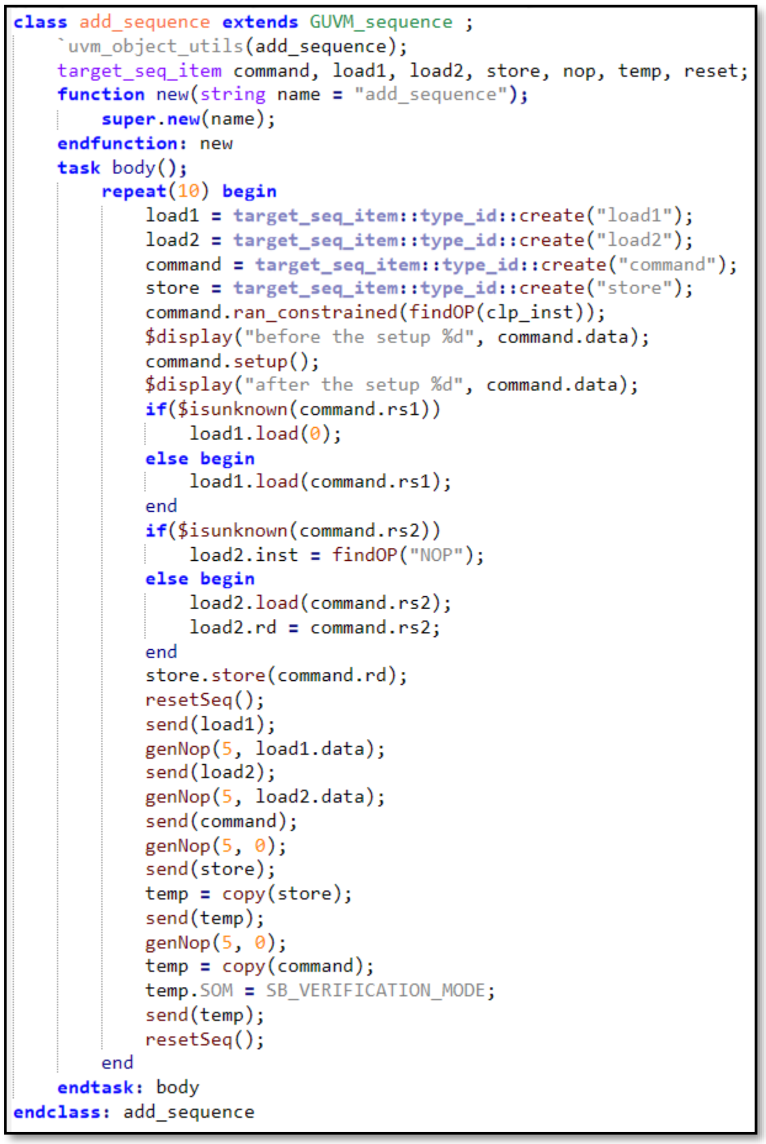
```
class GUVM_sequence extends uvm_sequence #(GUVM_sequence_item);

    `uvm_object_utils(GUVM_sequence);

    string clp_inst;

    function new(string name = "GUVM_sequence");
        super.new(name);
    endfunction: new

    function clp(uvm_cmdline_processor cmdline_proc);
        string my_value = "NOP";
        int rc;
        rc = cmdline_proc.get_arg_value("+ARG_INST=", my_value);
        clp_inst = my_value;
    endfunction

    task genNop(integer i, logic[31:0] data);
        repeat(i) begin
            target_seq_item nop;
            nop = target_seq_item::type_id::create("nop");
            nop.ran_constrained(NOP);
            nop.data = data;
            start_item(nop);
            finish_item(nop);
        end
    endtask

    function target_seq_item copy(target_seq_item targ);
        target_seq_item x;
        x = target_seq_item::type_id::create("x");
        x.do_copy(targ);
        return x;
    endfunction

    task send(target_seq_item targ);
        start_item(targ);
        finish_item(targ);
    endtask

    task resetSeq();
        target_seq_item reset;
        reset = target_seq_item::type_id::create("reset");
        reset.SOM = SB_RESET_MODE;
        send(reset);
    endtask

endclass: GUVM_sequence
```

2. *“Xseq_seq.sv”:* This child sequence is extended from the generic parent sequence (GUVM_sequence), the constructor of the new class is declared and, it is registered with the UVM factory. The sequence items are defined as (target_seq_item) needed for this child sequence specifically, then a task is instantiated with a finite number of iterations where the sequence items defined previously are created, and are used with the pre-defined functions inside the parent sequence to build the behavior, and sequence to test the instruction.

Inside the task, the child sequences can call other pre-defined functions, or tasks -other than the functions, or tasks inside the parent sequence, inside the target sequence item (target_seq_item), or the target package (target_pkg), for example:

- ***ran_constrained():*** Function used to randomize the non op-code fields of a certain instruction. More details related to this function are in Section 4.1.c.
- ***load():*** Function used to specify the input (load) register address. More details related to this function are in Section 4.2.d.
- ***store():*** Function used to specify the output (store) register address. More details related to this function are in Section 4.2.e.
- **setup():** Function used to set-up the instruction format fields. More details related to this function are in Section 4.2.e.
- ***findOP():*** Function used to return the instruction corresponding to a string input parameter  from the package. More details related to this function are in Section 4.2.c.

**Listing 4 A child sequence file (add_seq).**



```
class add_sequence extends GUVM_sequence ;
    `uvm_object_utils(add_sequence);
    target_seq_item command, load1, load2, store, nop, temp, reset;
    function new(string name = "add_sequence");
        super.new(name);
    endfunction: new
    task body();
        repeat(10) begin
            load1 = target_seq_item::type_id::create("load1");
            load2 = target_seq_item::type_id::create("load2");
            command = target_seq_item::type_id::create("command");
            store = target_seq_item::type_id::create("store");
            command.ran_constrained(findOP(clp_inst));
            $display("before the setup %d", command.data);
            command.setup();
            $display("after the setup %d", command.data);
            if($isunknown(command.rs1))
                load1.load(0);
            else begin
                load1.load(command.rs1);
            end
            if($isunknown(command.rs2))
                load2.inst = findOP("NOP");
            else begin
                load2.load(command.rs2);
                load2.rd = command.rs2;
            end
            store.store(command.rd);
            resetSeq();
            send(load1);
            genNop(5, load1.data);
            send(load2);
            genNop(5, load2.data);
            send(command);
            genNop(5, 0);
            send(store);
            temp = copy(store);
            send(temp);
            genNop(5, 0);
            temp = copy(command);
            temp.SOM = SB_VERIFICATION_MODE;
            send(temp);
            resetSeq();
        end
    endtask: body
endclass: add_sequence
```

The child sequences can control the mode of the generic scoreboard (GUVM_scoreboard) with the following mechanism: The default mode is the history mode (SB_HISTORY_MODE), where the generic history transaction (GUVM_history_transaction) is updated with the created sequence items and the transaction of outputs of the DUT using the generic command monitor (GUVM_cmd_monitor), and the generic result monitor (GUVM_result_monitor) after broadcasting these sequence items by the generic driver (GUVM_driver) to the interface of the DUT, then the mode is changed to the verification mode (SB_VERIFICATION_MODE) where the scoreboard checks the output of the DUT with a calculated golden reference. More details related to the generic scoreboard (GUVM_scoreboard) and the generic result transaction (GUVM_result_transaction) are in Section 4.1.j, and Section 4.1.i, and more details related to the generic command monitor (GUVM_cmd_monitor), the generic result monitor (GUVM_result_monitor), and the generic driver (GUCM_driver) are in Section 4.1.g, Section 4.1.h, and Section 4.1.k.

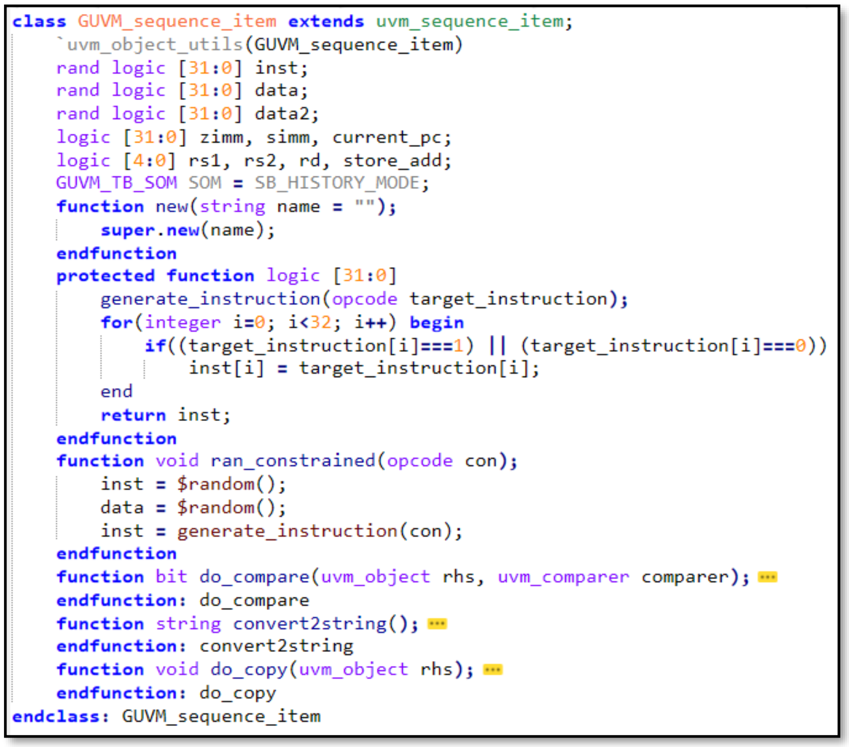
Note that the instruction can have its own child sequence, or the same sequence can be shared with more than one instruction. Listing 4 shows the child sequence (add_seq) that can be used as the child sequence of the arithmetic, shift and logic groups of instructions.

**3.** *“GUVM_tests”:* Where the files of all the parent and child sequences are included.

**c.** ***“GUVM_sequence_item.sv”:*** A file where the parent sequence item (GUVM_sequence_item) is extended from (uvm_sequence_item) base class, it is registered with the UVM factory, some logic variables as shown in Listing 5, and Listing 6 are defined, then the constructor of the new class is declared. The parent sequence item contains a group of functions as following, that can be called in the child sequences, as well as the child sequence items such as target_seq_item:

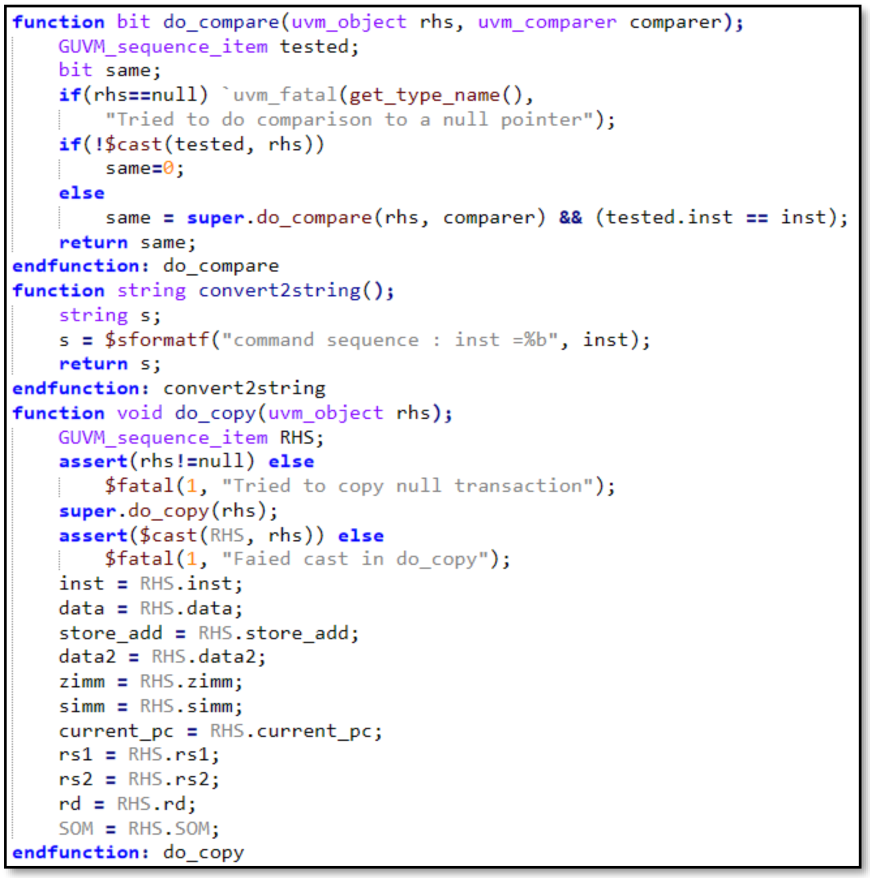
- ***ran_constrained():*** Function used to randomize the non-opcode fields of a certain instruction. It receives the 32 bits of the instruction coming from the package, then it calls another function (***generate_instruction()***) where the non-opcode fields are constrained-randomized, then it is assigned to the previously defined (inst) variable along with the randomized data assigned to (data).
- ***generate_instruction():*** Function used to constrained-randomize the non-opcode fields of the instruction with the following mechanism: it checks each bit of the 32 bits of the instruction starting from the least significant bit (LSB), if it is not "1" or "0" (it is "x"), the function will convert it to "1" or "0" randomly, then it return the constrained-randomized instruction back to be assigned to the (inst) variable.
- ***do_compare():*** Function used if desired to compare the contents, and objects of the given two sequence items; if they are the same, it returns "1".
- ***do_copy():*** Function used if desired to copy the contents, and objects of the sequence item to another sequence item.
- ***convert2string():*** Function used if desired to convert the 32 bits of the instruction of the sequence item to its name, or argument.

**Listing 5 The parent sequence item (GUVM_sequence_item).**



```
class GUVM_sequence_item extends uvm_sequence_item;
    `uvm_object_utils(GUVM_sequence_item)
    rand logic [31:0] inst;
    rand logic [31:0] data;
    rand logic [31:0] data2;
    logic [31:0] zimm, simm, current_pc;
    logic [4:0] rs1, rs2, rd, store_add;
    GUVM_TB_SOM SOM = SB_HISTORY_MODE;
    function new(string name = "");
        super.new(name);
    endfunction
    protected function logic [31:0]
        generate_instruction(opcode target_instruction);
        for(integer i=0; i<32; i++) begin
            if((target_instruction[i]===1) || (target_instruction[i]===0))
                inst[i] = target_instruction[i];
        end
        return inst;
    endfunction
    function void ran_constrained(opcode con);
        inst = $random();
        data = $random();
        inst = generate_instruction(con);
    endfunction
    function bit do_compare(uvm_object rhs, uvm_comparer comparer); ...
    endfunction: do_compare
    function string convert2string(); ...
    endfunction: convert2string
    function void do_copy(uvm_object rhs); ...
    endfunction: do_copy
endclass: GUVM_sequence_item
```

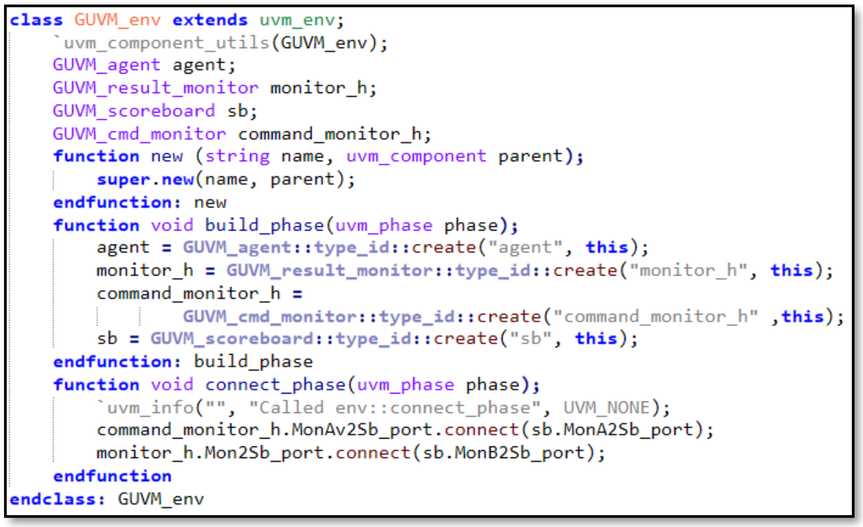
**Listing 6 The parent sequence item: do_compare(), do_copy() and, convert2string() functions.**



```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    GUVM_sequence_item tested;
    bit same;
    if(rhs==null) `uvm_fatal(get_type_name(),
        "Tried to do comparison to a null pointer");
    if(!$cast(tested, rhs))
        same=0;
    else
        same = super.do_compare(rhs, comparer) && (tested.inst == inst);
    return same;
endfunction: do_compare
function string convert2string();
    string s;
    s = $sformatf("command sequence : inst =%b", inst);
    return s;
endfunction: convert2string
function void do_copy(uvm_object rhs);
    GUVM_sequence_item RHS;
    assert(rhs!=null) else
        $fatal(1, "Tried to copy null transaction");
    super.do_copy(rhs);
    assert($cast(RHS, rhs)) else
        $fatal(1, "Faied cast in do_copy");
    inst = RHS.inst;
    data = RHS.data;
    store_add = RHS.store_add;
    data2 = RHS.data2;
    zimm = RHS.zimm;
    simm = RHS.simm;
    current_pc = RHS.current_pc;
    rs1 = RHS.rs1;
    rs2 = RHS.rs2;
    rd = RHS.rd;
    SOM = RHS.SOM;
endfunction: do_copy
```

**d.** ***<u>"GUVM_env.sv":</u>*** A file where (GUVM_env) is extended from (uvm_env) base class, it is registered with the UVM factory, then the constructor of this new class is declared. The generic agent (GUVM_agent) is defined, the generic result monitor (GUVM_result_monitor), the generic command monitor (GUVM_cmd_monitor), and the generic scoreboard (GUVM_scoreboard) are instantiated, and created at the build phase as shown in Listing 7, then (GUVM_cmd_monitor) is connected to (GUVM_scoreboard) through (MonA2Sb_port), and (GUVM_result_monitor) to (GUVM_scoreboard) through (MonB2Sb_port) at the connect phase. More details related to these ports are in Section 4.1.g, and Section 4.1.h.

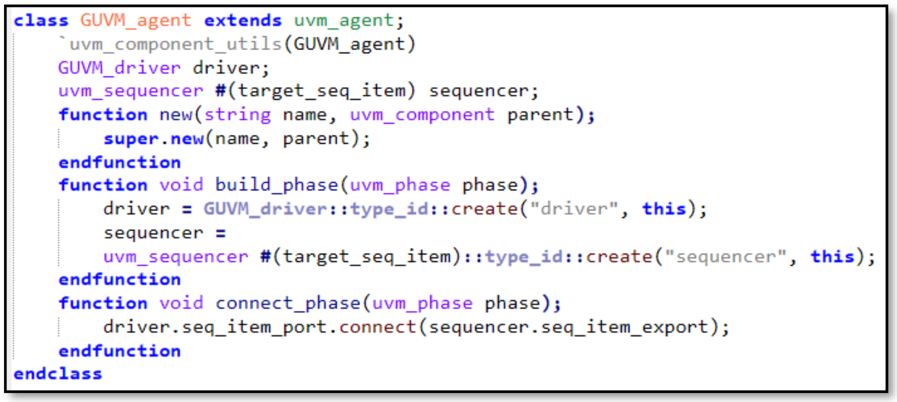
**Listing 7        The generic environment (GUVM_env).**



```
class GUVM_env extends uvm_env;
    `uvm_component_utils(GUVM_env);
    GUVM_agent agent;
    GUVM_result_monitor monitor_h;
    GUVM_scoreboard sb;
    GUVM_cmd_monitor command_monitor_h;
    function new (string name, uvm_component parent);
        super.new(name, parent);
    endfunction: new
    function void build_phase(uvm_phase phase);
        agent = GUVM_agent::type_id::create("agent", this);
        monitor_h = GUVM_result_monitor::type_id::create("monitor_h", this);
        command_monitor_h =
                GUVM_cmd_monitor::type_id::create("command_monitor_h" ,this);
        sb = GUVM_scoreboard::type_id::create("sb", this);
    endfunction: build_phase
    function void connect_phase(uvm_phase phase);
        `uvm_info("", "Called env::connect_phase", UVM_NONE);
        command_monitor_h.MonAv2Sb_port.connect(sb.MonA2Sb_port);
        monitor_h.Mon2Sb_port.connect(sb.MonB2Sb_port);
    endfunction
endclass: GUVM_env
```

**e.** ***<u>"GUVM_agent.sv":</u>*** A file where (GUVM_agent) is extended from (uvm_agent) base class, it is registered with the UVM factory, then the constructor of this new class is declared. The generic driver (GUVM_driver), the sequencer -used to generate data transactions as class objects (target_seq_item) and send it to the driver (GUVM_driver) for excution are defined. The sequencer is exended from (uvm_sequencer) base class- to be instantiated, and created at the build phase as shown in Listing 8. Then the (seq_item_port) in the driver is connected with (seq_item_export) in the sequencer, so that the driver can use the TLM functions to get the next item from the sequencer when required. More details related to the generic driver (GUVM_driver) are in Section 4.1.f.

**Listing 8 The generic agent (GUVM_agent).**



```
class GUVM_agent extends uvm_agent;
    `uvm_component_utils(GUVM_agent)
    GUVM_driver driver;
    uvm_sequencer #(target_seq_item) sequencer;
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction
    function void build_phase(uvm_phase phase);
        driver = GUVM_driver::type_id::create("driver", this);
        sequencer =
        uvm_sequencer #(target_seq_item)::type_id::create("sequencer", this);
    endfunction
    function void connect_phase(uvm_phase phase);
        driver.seq_item_port.connect(sequencer.seq_item_export);
    endfunction
endclass
```

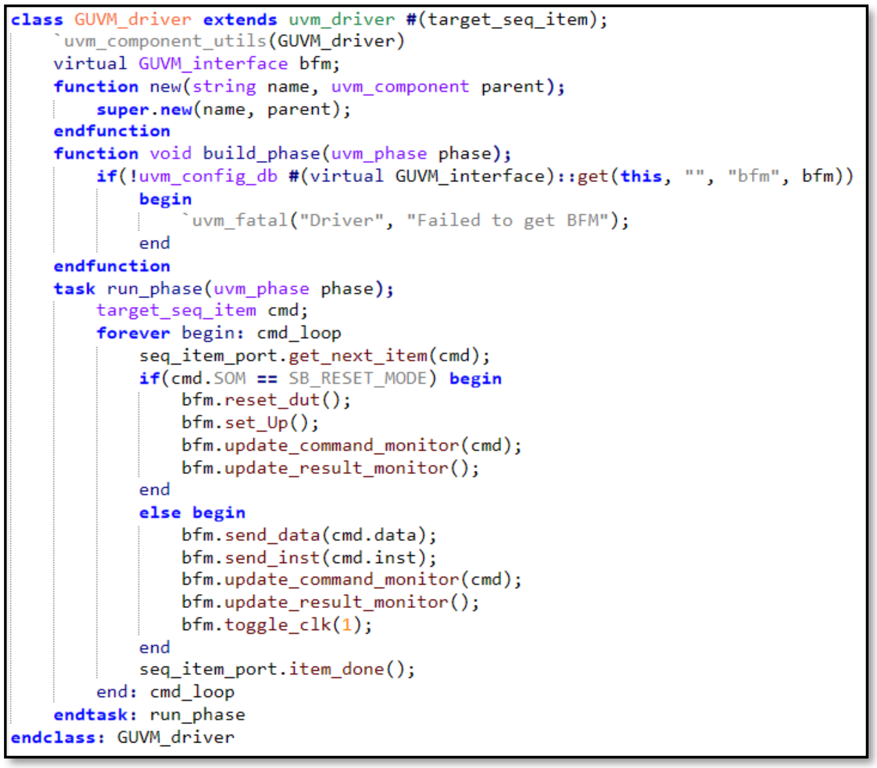
**f.** ***<u>“GUVM_driver.sv”:</u>*** A file where (GUVM_driver) is extended from (uvm_driver) base class, it is registered with the UVM factory, then the constructor of this new class is declared. The interface (GUVM_interface) is defined as a virtual one (bfm), then the UVM configuration data base (uvm_config_db) is used to pass objects between the defined virtual interface (bfm) and (GUVM_driver) at the build phase as shown in Listing 9. Then the run phase is started in a forever loop to trigger the interface and let it accesses the DUT, passes the generated sequence items (stimuli) that are coming from the sequencer (from the child sequences) through the driver as (target_seq_item) to the DUT, receives the outputs of the DUT, and passes them to the monitors by calling a group of fucntions, and tasks inside this interface as following:

- ***reset_dut() and set_Up():*** Tasks used to trigger the interface to reset and setup the DUT with the proper signals. More details related to these tasks are in Section 4.2.x.

- ***send_data() and send_inst():*** Tasks used to trigger the interface to send the coming sequence items and its contents (the instruction and the data) to the DUT. More details related to these tasks are in Section 4.2.x.
- ***update_result_monitor():*** Task used to trigger the interface to send the outputs from the DUT to the generic result monitor (GUVM_result_monitor). More details related to this task are in Section 4.1.h, and Section 4.2.x.
- ***update_command_monitor():*** Function used to trigger the interface to send the coming sequence item from the  generic driver driver (GUVM_driver) to the DUT, to the generic command monitor (GUVM_cmd_monitor). More details related to this function are in Section 4.1.g, Section 4.2.x.
- **toggle_clk():** Task used to trigger the interface to toggle the DUT clock (i) given times. More details related to this task are in Section 2.2.x.

The generic driver (GUVM_driver) is additionally used to reset the history of the scoreboard if the scoreboard mode set by the child sequences is the reset mode (SB_RESET_MODE). More details related to the generic scoreboard (GUVM_scoreboard) are in Section 4.1.j.

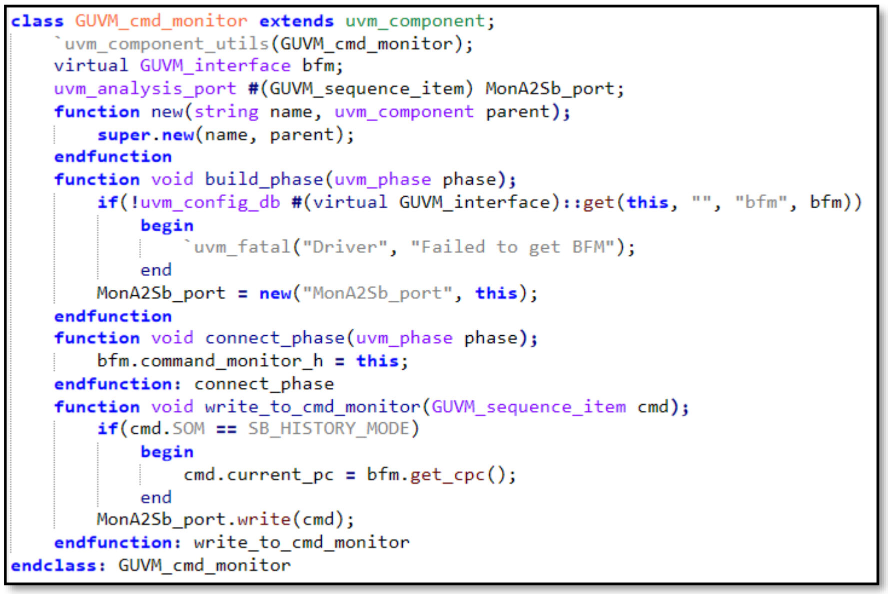
**Listing 9 The generic driver (GUVM_driver).**



```
class GUVM_driver extends uvm_driver #(target_seq_item);
    `uvm_component_utils(GUVM_driver)
    virtual GUVM_interface bfm;
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction
    function void build_phase(uvm_phase phase);
        if(!uvm_config_db #(virtual GUVM_interface)::get(this, "", "bfm", bfm))
            begin
                `uvm_fatal("Driver", "Failed to get BFM");
            end
    endfunction
    task run_phase(uvm_phase phase);
        target_seq_item cmd;
        forever begin: cmd_loop
            seq_item_port.get_next_item(cmd);
            if(cmd.SOM == SB_RESET_MODE) begin
                bfm.reset_dut();
                bfm.set_Up();
                bfm.update_command_monitor(cmd);
                bfm.update_result_monitor();
            end
            else begin
                bfm.send_data(cmd.data);
                bfm.send_inst(cmd.inst);
                bfm.update_command_monitor(cmd);
                bfm.update_result_monitor();
                bfm.toggle_clk(1);
            end
            seq_item_port.item_done();
        end: cmd_loop
    endtask: run_phase
endclass: GUVM_driver
```

g. ***<u>"GUVM_cmd_monitor.sv":</u>*** A file where the generic command monitor (GUVM_cmd_monitor) is extended from (uvm_component) base class, then it is registered with the UVM factory. (MonA2Sb_port) is defined as an extension from (uvm_analysis_port) that can deal with the parent sequence item and its extended child sequence items, then the constructor of this new class is declared. Then the interface (GUVM_interface) is defined as a virtual interface (bfm), then the UVM configuration data base (uvm_config_db) is used to pass objects between the defined virtual interface (bfm) and (GUVM_cmd_monitor) at the build phase as shown in Listing 10. Additionally, (MonA2Sb_port) is created between (GUVM_cmd_monitor) and (GUVM_scoreboard) at the build phase after it is already defined. Then (GUVM_cmd_monitor) is connected with the virtual interface (bfm) at the connect phase, then a void function (*write_to_cmd_monitor()*) is created to receive the sequence items as (target_seq_item) that are coming from the generic driver (GUVM_driver) to the DUT, pass them to the generic scoreboard (GUVM_scoreboard) through (***write()***) function of (MonA2Sb_port), then the generic history transaction (GUVM_history_transaction) is updated with the new command sequence item. The generic command monitor (GUVM_cmd_monitor) can update this sequence item with a new content (current_pc) by calling a pre-defined function (*get_cpc()*) in the DUT interface (GUVM_interface), only if the mode of the generic scoreboard (GUVM_scoreboard) is the history mode (SB_HISTOTY_MODE). More details related to (get_cpc()) and (write_to_cmd_monitor()) functions are in Section 4.2.x, and more details related to the generic scoreboard (GUVM_scoreboard), and the generic history transaction (GUVM_history_transaction) are in Section 4.1.j , and Section 4.1.k.

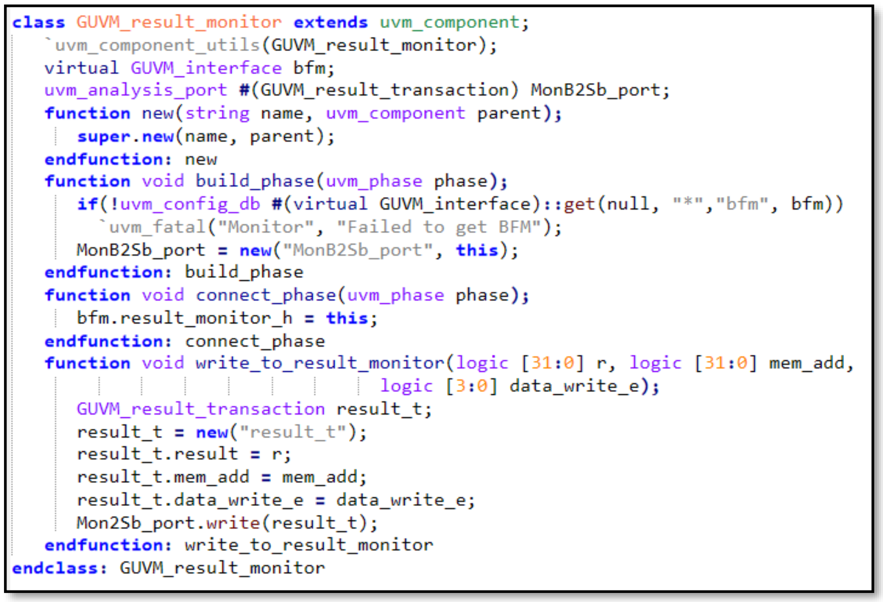
**Listing 10 The generic command monitor (GUVM_cmd_monitor).**



```
class GUVM_cmd_monitor extends uvm_component;
    `uvm_component_utils(GUVM_cmd_monitor);
    virtual GUVM_interface bfm;
    uvm_analysis_port #(GUVM_sequence_item) MonA2Sb_port;
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction
    function void build_phase(uvm_phase phase);
        if(!uvm_config_db #(virtual GUVM_interface)::get(this, "", "bfm", bfm))
            begin
                `uvm_fatal("Driver", "Failed to get BFM");
            end
        MonA2Sb_port = new("MonA2Sb_port", this);
    endfunction
    function void connect_phase(uvm_phase phase);
        bfm.command_monitor_h = this;
    endfunction: connect_phase
    function void write_to_cmd_monitor(GUVM_sequence_item cmd);
        if(cmd.SOM == SB_HISTORY_MODE)
            begin
                cmd.current_pc = bfm.get_cpc();
            end
        MonA2Sb_port.write(cmd);
    endfunction: write_to_cmd_monitor
endclass: GUVM_cmd_monitor
```

**h.** ***"GUVM_result_monitor.sv":*** A file where the generic result monitor (GUVM_result_monitor) is extended from (uvm_component) base class, it is registered with the UVM factory. (MonB2Sb_port) is defined as an extension from (uvm_analysis_port) that can deal with the generic result transaction (GUVM_result_transaction) responsible of moving the outputs, and results from the DUT to the generic scoreboard (GUVM_scoreboard) through (GUVM_result_monitor), then the constructor of this new class is declared. Then the interface (GUVM_interface) is defined as a virtual interface (bfm), then the UVM configuration data base (uvm_config_db) is used to pass objects between the defined virtual interface (bfm) and (GUVM_cmd_monitor) at the build phase, additionally, (MonB2Sb_port) is created between (GUVM_cmd_monitor) and (GUVM_scoreboard) at the build phase as shown in Listing 11. Then (GUVM_cmd_monitor) is connected with the virtual interface (bfm) at the connect phase, then a a void function (*write_to_result_monitor()*) is created, that receives the desired outputs from the DUT such that, output data, output address, and output data write enable signal, and assigns them to a new defined and created transaction, then passing it to the generic scoreboard (GUVM_scoreboard) through (*write()*) function of (MonB2Sb_port), then the generic history transaction (GUVM_history_transaction) is updated with the transaction. More details related to (write_to_cmd_monitor()) fucntion are in Section 4.1.x, more details related to (GUVM_result_transaction) are in Section 4.1.i, and more details related to the generic scoreboard (GUVM_scoreboard), and the generic history transaction (GUVM_history_transaction) are in Section 4.1.j , and Section 4.1.k.

**Listing 11 The generic result monitor (GUVM_result_monitor).**



```
class GUVM_result_monitor extends uvm_component;
   `uvm_component_utils(GUVM_result_monitor);
   virtual GUVM_interface bfm;
   uvm_analysis_port #(GUVM_result_transaction) MonB2Sb_port;
   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction: new
   function void build_phase(uvm_phase phase);
      if(!uvm_config_db #(virtual GUVM_interface)::get(null, "*","bfm", bfm))
         `uvm_fatal("Monitor", "Failed to get BFM");
      MonB2Sb_port = new("MonB2Sb_port", this);
   endfunction: build_phase
   function void connect_phase(uvm_phase phase);
      bfm.result_monitor_h = this;
   endfunction: connect_phase
   function void write_to_result_monitor(logic [31:0] r, logic [31:0] mem_add,
                                          logic [3:0] data_write_e);
      GUVM_result_transaction result_t;
      result_t = new("result_t");
      result_t.result = r;
      result_t.mem_add = mem_add;
      result_t.data_write_e = data_write_e;
      Mon2Sb_port.write(result_t);
   endfunction: write_to_result_monitor
endclass: GUVM_result_monitor
```

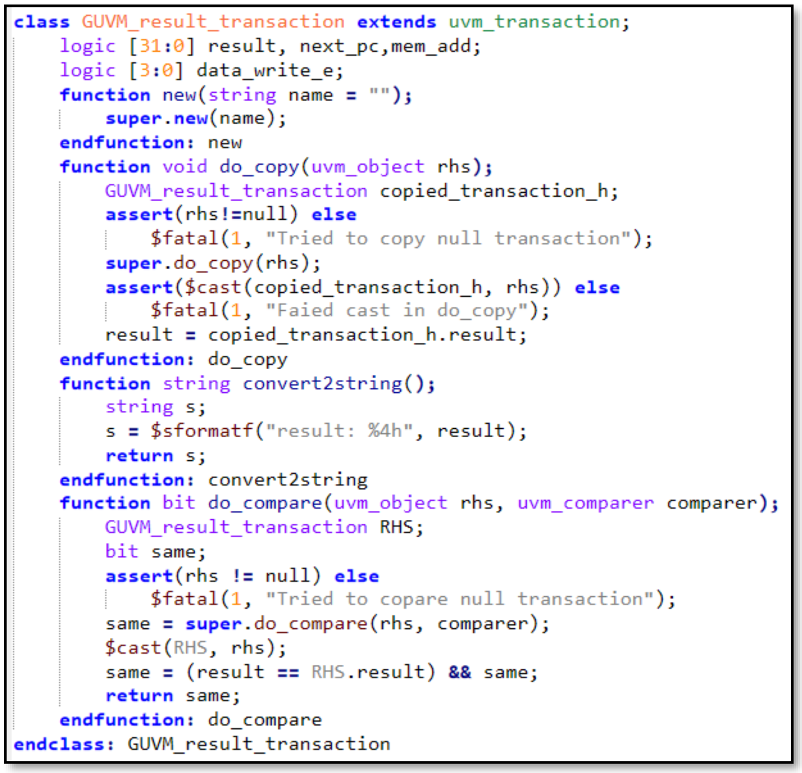
i. ***“GUVM_rersult_transaction.sv”:*** A file where the generic result transaction (GUVM_result_transaction) is extended from (uvm_transaction) base class, then the constructor of this new class is declared. The items needed to be transported by (GUVM_result_transaction) from the interface DUT (GUVM_interface) to the generic result monitor (GUVM_result_monitor) all the way the generic scoreboard (GUVM_scoreboard) are defined as shown in Listing 12. More details related to the generic scoreboard (GUVM_scoreboard) are in Section 4.1.j, and more details related to the generic result monitor (GUVM_result_monitor) are in Section 4.1.h.

Additionally, a group of functions are created as following those can be used if desired:

- ***do_compare():*** Function used if desired to compare the contents, and objects of the given two transactions; if they are the same, it returns “1”.

- ***do_copy():*** Function used if desired to copy the contents, and objects of the transaction to another transaction.
- ***convert2string():*** Function used if desired to convert the 32 bits of the instruction of the sequence item to its name, or argument.

**Listing 12 The generic result monitor (GUVM_result_monitor).**

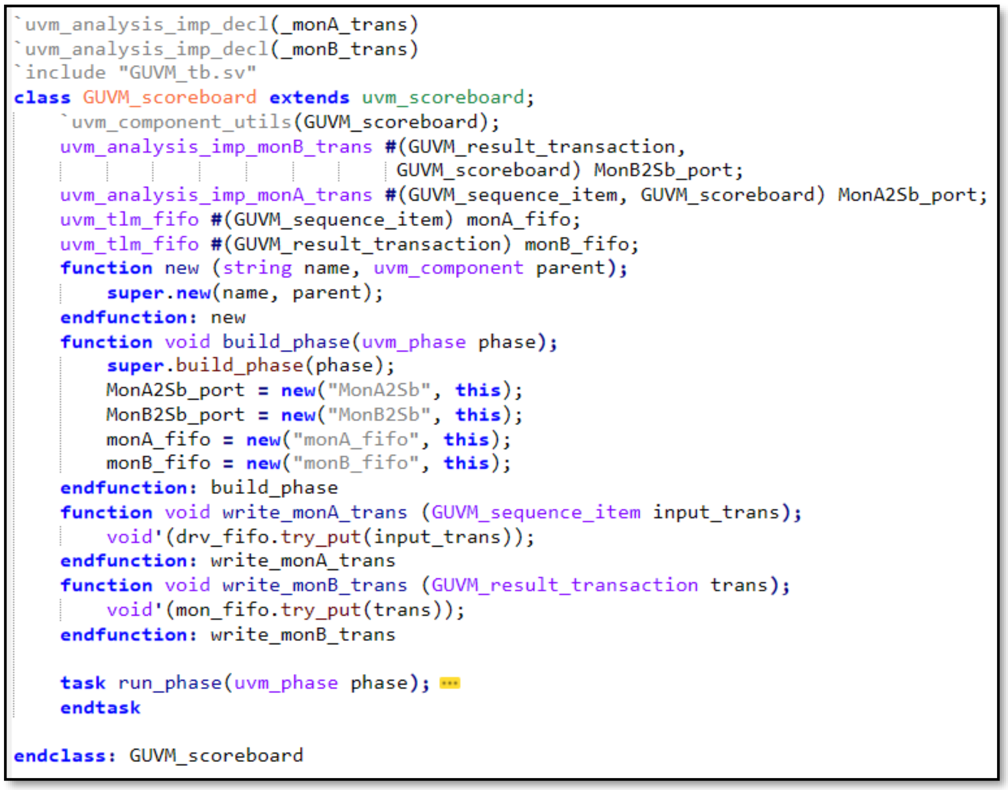


```
class GUVM_result_transaction extends uvm_transaction;
    logic [31:0] result, next_pc,mem_add;
    logic [3:0] data_write_e;
    function new(string name = "");
        super.new(name);
    endfunction: new
    function void do_copy(uvm_object rhs);
        GUVM_result_transaction copied_transaction_h;
        assert(rhs!=null) else
            $fatal(1, "Tried to copy null transaction");
        super.do_copy(rhs);
        assert($cast(copied_transaction_h, rhs)) else
            $fatal(1, "Faied cast in do_copy");
        result = copied_transaction_h.result;
    endfunction: do_copy
    function string convert2string();
        string s;
        s = $sformatf("result: %4h", result);
        return s;
    endfunction: convert2string
    function bit do_compare(uvm_object rhs, uvm_comparer comparer);
        GUVM_result_transaction RHS;
        bit same;
        assert(rhs != null) else
            $fatal(1, "Tried to copare null transaction");
        same = super.do_compare(rhs, comparer);
        $cast(RHS, rhs);
        same = (result == RHS.result) && same;
        return same;
    endfunction: do_compare
endclass: GUVM_result_transaction
```

**j.** ***<u>"GUVM_scoreboard.sv":</u>*** A file where the generic scoreboard (GUVM_scoreboard) is extended from (uvm_scoreboard) base class, the analysis implementation ports are defined with names after defining them with macros, then the TLM FIFOs is defined, which is used to store the received sequence items from the generic command monitor (monA_fifo), and the transactions from the generic result monitor (monB_fifo), then the constructor of this new class is declared. The defined ports and FIFOs are instantiated, and created at the build phase as shown in

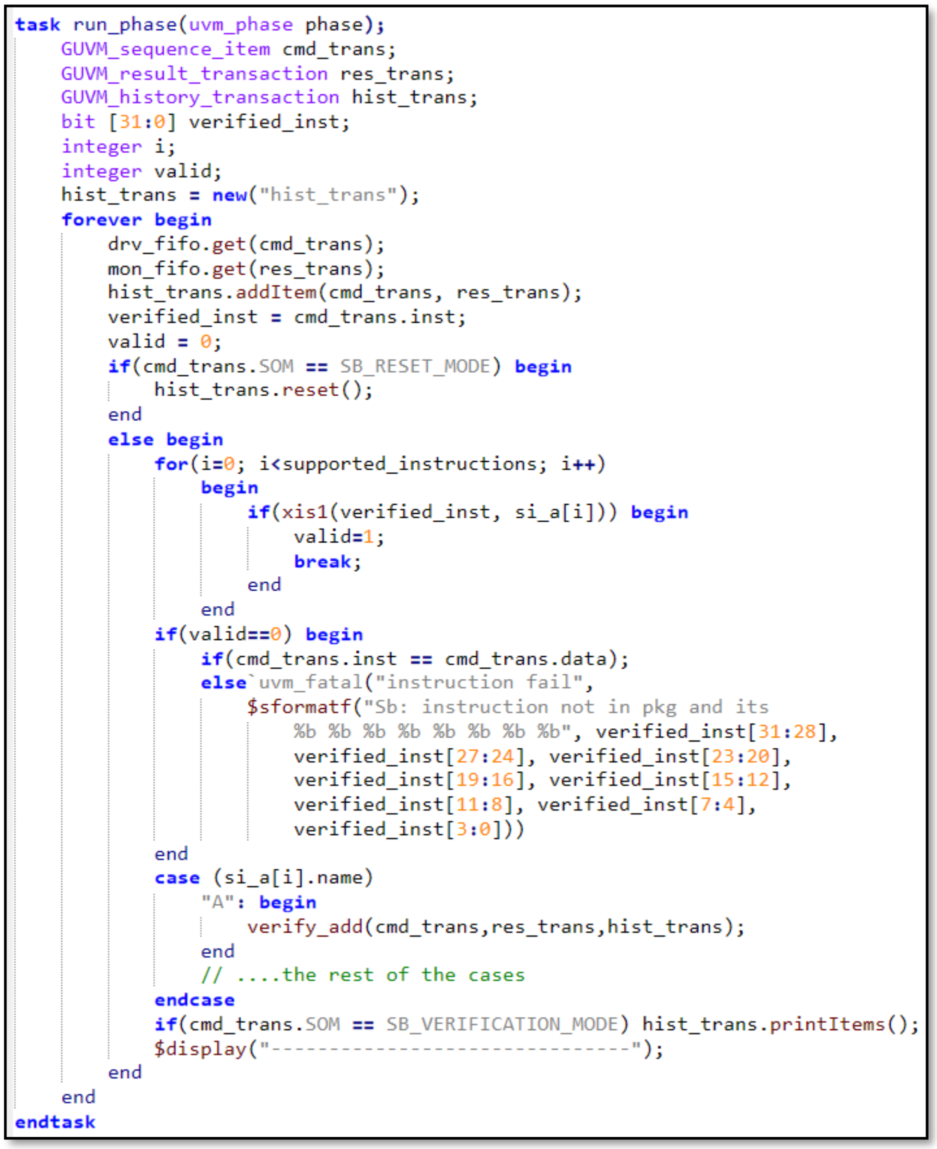
Listing 13, then two new void functions are created: (write_monA_trans) function, which is called when the generic command monitor (GUVM_cmd_monitor) broadcasts a new sequence item to the scoreboard and add it to (monA_fifo), and (write_monB_trans) function, which is called when the generic result transaction (GUVM_result_transaction) broadcasts the DUT results in a new transaction to the scoreboard and add it to (monB_fifo). Then (cmd_trans) is defined as ( GUVM_sequence_item) to be able to deal with the stored sequence items inside (monA_fifo), and (result_trans) to be able to deal with the stored transactions inside the (monB_fifo) at the run phase as shown in Listing 14, additionally, (verified_inst) is defined, which is used to store the instruction of the new derived sequence item, and create after defining a new history transaction (history_tans) to store the received sequence item and the received transaction with the following mechanism: A new iteration inside the forever loop occurs, then the FIFOs release the sequence item and the transaction related to the (verified_inst), then these  are stored inside the history transaction using (***addItem()***) function, this function is pre-defined inside the generic history transaction (GUVM_histoty_transaction), if the mode of the scoreboard set by the child sequence is the reset mode (SB_RESET_MODE), the generic history transaction will be reset, however, if the mood is different, The verification is continued and checking if (verified_inst)  exists or not inside the instruction array filled by the core package (*si_a[]*) using (***xist1()***) function, if it does not exist, an error message is displayed, if it exists, a case condition checks the name of the instruction such as, “A”, “M”, and “Load” to choose the proper test case to execute -from a group of header files included to the scoreboard by the included file “GUVM_tb.sv”- by calling the pre-defined function of the chosen header file, for example, “verify_add()”, this function takes -as input arguments- the generic command monitor sequence item, the result generic monitor transaction and the generic history transaction. These header files use the history to check if the outputs of the DUT equal to the calculated results or not. At the end, if the mode of the scoreboard is changed to the verification mode (SB_VERIFICATION_MODE), it prints out all the contents of the history along with the calculated result defining if the instruction successfully passes the functionality test or not. More details related to the generic history transaction (GUVM_history_transaction) are in Section 4.1.k, and more details related to the header files of the test cases are in Section 4.1.l.

**Listing 13 The generic scoreboard (GUVM_scoreboard).**



```
`uvm_analysis_imp_decl(_monA_trans)
`uvm_analysis_imp_decl(_monB_trans)
`include "GUVM_tb.sv"
class GUVM_scoreboard extends uvm_scoreboard;
    `uvm_component_utils(GUVM_scoreboard);
    uvm_analysis_imp_monB_trans #(GUVM_result_transaction,
                                  GUVM_scoreboard) MonB2Sb_port;
    uvm_analysis_imp_monA_trans #(GUVM_sequence_item, GUVM_scoreboard) MonA2Sb_port;
    uvm_tlm_fifo #(GUVM_sequence_item) monA_fifo;
    uvm_tlm_fifo #(GUVM_result_transaction) monB_fifo;
    function new (string name, uvm_component parent);
        super.new(name, parent);
    endfunction: new
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        MonA2Sb_port = new("MonA2Sb", this);
        MonB2Sb_port = new("MonB2Sb", this);
        monA_fifo = new("monA_fifo", this);
        monB_fifo = new("monB_fifo", this);
    endfunction: build_phase
    function void write_monA_trans (GUVM_sequence_item input_trans);
        void'(drv_fifo.try_put(input_trans));
    endfunction: write_monA_trans
    function void write_monB_trans (GUVM_result_transaction trans);
        void'(mon_fifo.try_put(trans));
    endfunction: write_monB_trans

    task run_phase(uvm_phase phase); ...
    endtask

endclass: GUVM_scoreboard
```

**Listing 14	The run phase of the generic scoreboard (GUVM_scoreboard).**

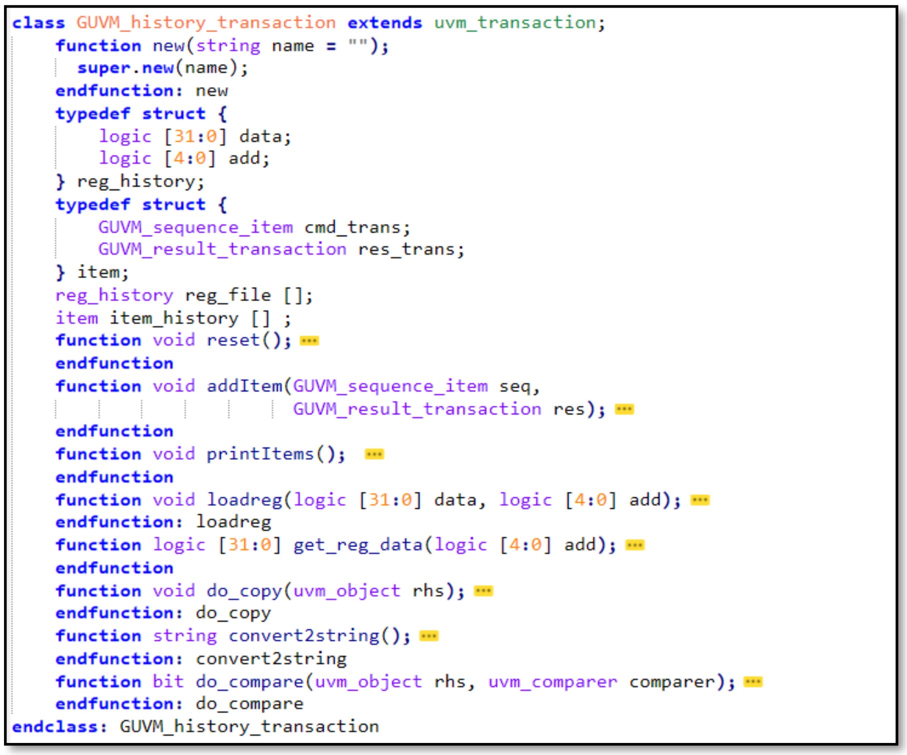


```
task run_phase(uvm_phase phase);
    GUVM_sequence_item cmd_trans;
    GUVM_result_transaction res_trans;
    GUVM_history_transaction hist_trans;
    bit [31:0] verified_inst;
    integer i;
    integer valid;
    hist_trans = new("hist_trans");
    forever begin
        drv_fifo.get(cmd_trans);
        mon_fifo.get(res_trans);
        hist_trans.addItem(cmd_trans, res_trans);
        verified_inst = cmd_trans.inst;
        valid = 0;
        if(cmd_trans.SOM == SB_RESET_MODE) begin
            hist_trans.reset();
        end
        else begin
            for(i=0; i<supported_instructions; i++)
                begin
                    if(xis1(verified_inst, si_a[i])) begin
                        valid=1;
                        break;
                    end
                end
            if(valid==0) begin
                if(cmd_trans.inst == cmd_trans.data);
                else`uvm_fatal("instruction fail",
                    $sformatf("Sb: instruction not in pkg and its
                        %b %b %b %b %b %b %b %b", verified_inst[31:28],
                        verified_inst[27:24], verified_inst[23:20],
                        verified_inst[19:16], verified_inst[15:12],
                        verified_inst[11:8], verified_inst[7:4],
                        verified_inst[3:0]))
            end
            case (si_a[i].name)
                "A": begin
                    verify_add(cmd_trans,res_trans,hist_trans);
                end
                // ....the rest of the cases
            endcase
            if(cmd_trans.SOM == SB_VERIFICATION_MODE) hist_trans.printItems();
            $display("------------------------------");
        end
    end
endtask
```

**k.** ***<u>"GUVM_history_transaction.sv":</u>*** A file where the generic history transaction (GUVM_history_transaction) is extended from (uvm_transaction) base class, then the constructor of this new class is declared. Then two new structures are declared: one used to store the sequence items and the transaction coming from the monitors (item), and another one used to store any data of a certain address (reg_history) as shown in Listing 15, then the structure (*reg_file[]*) -a virtual register file- as (reg_history) and the structure (*item_history[]*) as (item) are created. The generic history transaction (GUVM_history_transaction) contains a group of functions that can be called by the generic scoreboard (GUVM_scoreboard), or the header files of the test cases as following:

- ***reset():*** Function used by the generic scoreboard (GUVM_scoreboard) to clear the contents of the  structures explained above.
- ***addItem():*** Function used by the generic scoreboard (GUVM_scoreboard) to add the received transactions and sequence items from the monitors to the *(item_history[]*) structure.
- ***printItems():*** Function used by the generic scoreboard (GUVM_scoreboard) to print contents of the (*item_history[]*) structure.
- ***Loadreg():*** Function used by the test cases header files to load, and save given data of a specific given address -as inputs to the functions- inside the virtual register file (*reg_history[]*) to be used when desired.
- ***get_reg_data():*** Function used by the test cases header files to get, and summon the stored data inside the virtual register file (*reg_history[]*) related to an address given as an input.

**Listing 15 The generic history transaction (GUVM_history_transaction).**

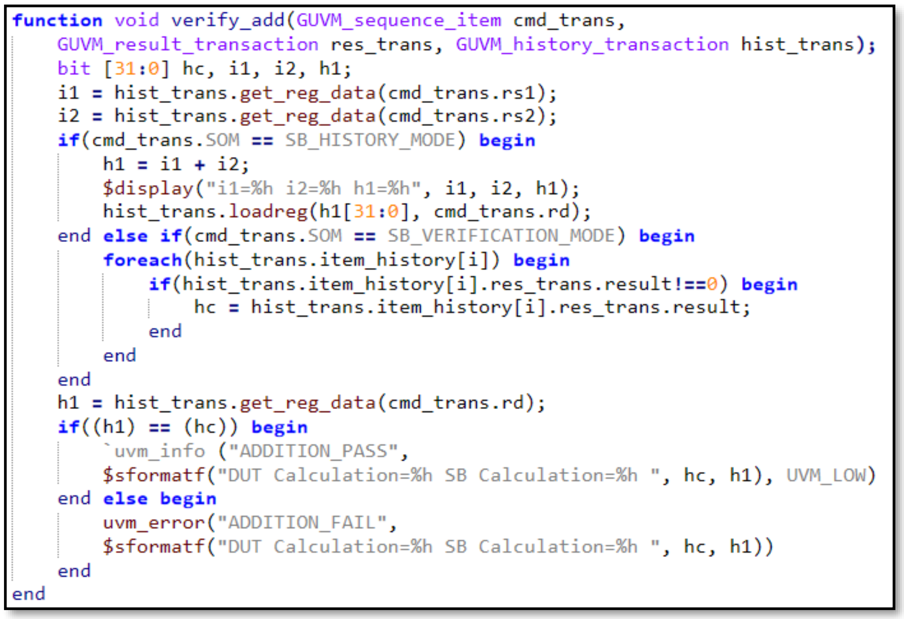


```
class GUVM_history_transaction extends uvm_transaction;
    function new(string name = "");
      super.new(name);
    endfunction: new
    typedef struct {
        logic [31:0] data;
        logic [4:0] add;
    } reg_history;
    typedef struct {
        GUVM_sequence_item cmd_trans;
        GUVM_result_transaction res_trans;
    } item;
    reg_history reg_file [];
    item item_history [] ;
    function void reset(); ...
    endfunction
    function void addItem(GUVM_sequence_item seq,
                          GUVM_result_transaction res); ...
    endfunction
    function void printItems();  ...
    endfunction
    function void loadreg(logic [31:0] data, logic [4:0] add); ...
    endfunction: loadreg
    function logic [31:0] get_reg_data(logic [4:0] add); ...
    endfunction
    function void do_copy(uvm_object rhs); ...
    endfunction: do_copy
    function string convert2string(); ...
    endfunction: convert2string
    function bit do_compare(uvm_object rhs, uvm_comparer comparer); ...
    endfunction: do_compare
endclass: GUVM_history_transaction
```

**l.** ***inst_h:*** A sub-sub-folder contains the header files of all the test cases of the instructions of the desired core to be functionally tested, and verified. All these header files are connected to the generic scoreboard (GUVM_scoreboard) and the generic history transaction (GUVM_hsitory_transaction) to check the outputs of the DUT, and compare them with the calculated results, for example, “add.svh”.

- *“Xinst.svh”:* A file where the created function that is called at the run phase in the generic scoreboard (GUVM_scoreboard) to handle the test case of the desired instruction, where this function receives the command monitor sequence item (cmd_trans), the result monitor transaction (result_trans) and the history transaction (history_trans). Listing 16 shows the testing function of the add instruction, first (i1) and (i2) are defined as the first and the second operands of the add instruction, then they are assigned with the actual randomized data by using a pre-defined function inside the generic history transaction (*get_reg_data()*), this function takes the addresses of the source registers of the addition instruction, then it returns the register stored data related to each address from a virtual register file, these data are previously stored inside the virtual register file  of the generic history transaction by the (load) sequence items sent by the child sequence (add_seq) before the addition sequence item. Next the mode of the generic scoreboard is checked, if it is the default history mode (SB_HISTORY_MODE), A defined golden reference (h1) is set by calculating the summation of the two operands, then the virtual register file is updated with calculated golden reference (h1) using the destination register address -to be used later when desired such as, for comparison- using the pre-defined (*loadreg()*) function of the generic history transaction. Noting that this (*loadreg()*) function is used by any sequence item to store its contained data in the virtual register file, only if the mode of the scoreboard is the default history mode). However, if the mode is the verification mode (SB_VERIFICATION_MODE), the comparison between the golden reference (h1) and the stored output of the DUT (inside the result transaction) begins with the following mechanism: the items of the history transaction are searched for the DUT data result of the addition instruction included, and contained by the stored result transaction and assign it to a new defined variable (hc), then the calculated golden reference is obtained back from the virtual register file (h1), then checking if  (h1) and (hc) are equal or not, if they are equal, the addition instruction successes and passes the functionality testing, if not, an error message printed out referring to the failure of this instruction in the core. More details related to the generic history transaction (GUVM_history_transaction) are in Section 4.1.k, and more details related to the generic scoreboard (GUVM_scoreboard) are in Section 4.1.j.

**Listing 16  The header file of the addition instruction.**

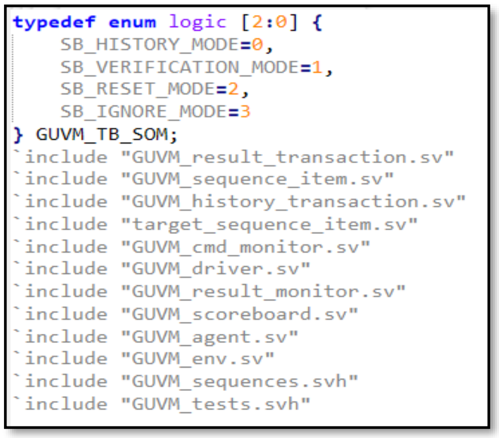


```
function void verify_add(GUVM_sequence_item cmd_trans,
    GUVM_result_transaction res_trans, GUVM_history_transaction hist_trans);
    bit [31:0] hc, i1, i2, h1;
    i1 = hist_trans.get_reg_data(cmd_trans.rs1);
    i2 = hist_trans.get_reg_data(cmd_trans.rs2);
    if(cmd_trans.SOM == SB_HISTORY_MODE) begin
        h1 = i1 + i2;
        $display("i1=%h i2=%h h1=%h", i1, i2, h1);
        hist_trans.loadreg(h1[31:0], cmd_trans.rd);
    end else if(cmd_trans.SOM == SB_VERIFICATION_MODE) begin
        foreach(hist_trans.item_history[i]) begin
            if(hist_trans.item_history[i].res_trans.result!==0) begin
                hc = hist_trans.item_history[i].res_trans.result;
            end
        end
    end
    h1 = hist_trans.get_reg_data(cmd_trans.rd);
    if((h1) == (hc)) begin
        `uvm_info ("ADDITION_PASS",
        $sformatf("DUT Calculation=%h SB Calculation=%h ", hc, h1), UVM_LOW)
    end else begin
        uvm_error("ADDITION_FAIL",
        $sformatf("DUT Calculation=%h SB Calculation=%h ", hc, h1))
    end
end
```

**m.** ***“GUVM_tb.sv”:*** where the header files of all the test cases of the instructions of the core for functionality testing ware included.

**n.** ***“GUVM.sv”:*** where the modes of the generic scoreboard (GUVM_scoreboard) are defined, additionally includes all the files and sub-sub-folders of the common sub-folder of the test bench to be included with the (testing_Xcore) sub-folder as shown in Listing 17.

**Listing 17 Contents of "GUVM.sv" file.**

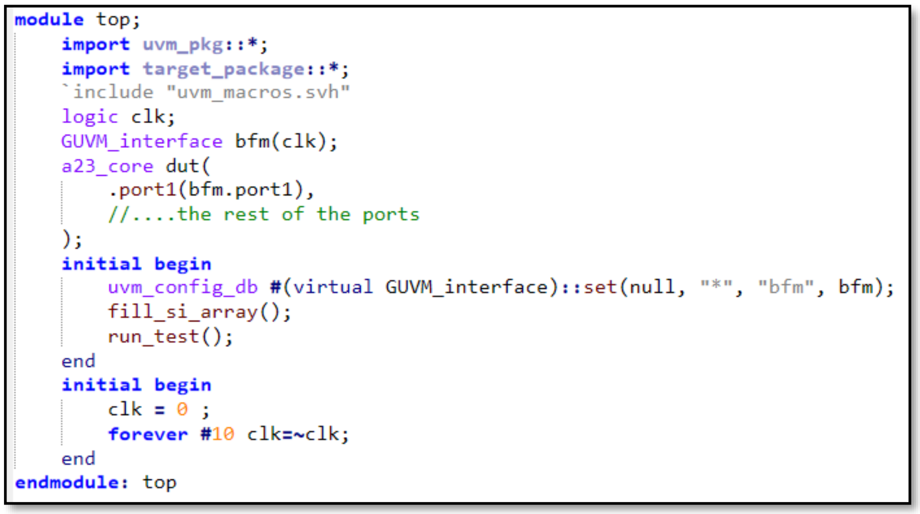

```
typedef enum logic [2:0] {
    SB_HISTORY_MODE=0,
    SB_VERIFICATION_MODE=1,
    SB_RESET_MODE=2,
    SB_IGNORE_MODE=3
} GUVM_TB_SOM;
`include "GUVM_result_transaction.sv"
`include "GUVM_sequence_item.sv"
`include "GUVM_history_transaction.sv"
`include "target_sequence_item.sv"
`include "GUVM_cmd_monitor.sv"
`include "GUVM_driver.sv"
`include "GUVM_result_monitor.sv"
`include "GUVM_scoreboard.sv"
`include "GUVM_agent.sv"
`include "GUVM_env.sv"
`include "GUVM_sequences.svh"
`include "GUVM_tests.svh"
```

## 4.2 testing_Xcore:

It consists of a number of files connected to each other and to the common folder to form the complete generic UVM environment.

**a.** ***"top.sv":*** A file where the top module of the test bench is declared, where (uvm_pkg), (uvm_macros) and the target package (target_pkg) of the core containing the core specific instructions needed to be verified are imported and included. Addionally, the DUT is instantiated by port mapping the ports of the top module (the inputs and the outputs of the DUT) and they are instantiated inside the interface (GUVM_interface) that is defined as a virtual interface (bfm), then the UVM configuration data base (uvm_config_db) is used to pass objects between the virtual interface (bfm) and the top module at the build phase as shown in Listing 18. (*fill_si_array()*) function defined in the target package (target_pkg) is called, this function fills an array with the instructions of the core to be used later by the generic (GUVM_scoreboard). In addition to that, the top module used to trigger, and run the test that is defined as an input argument in (run.py), additionally, the top module is used to instantiate the clock signal sent by the interface to the DUT. More details related to the target package (target_pkg) are in Section 4.2.c, and more details related to the generic scoreboard (GUVM_scoreboard) are in Section 4.1.j.

**Listing 18 The top module of the test bench.**

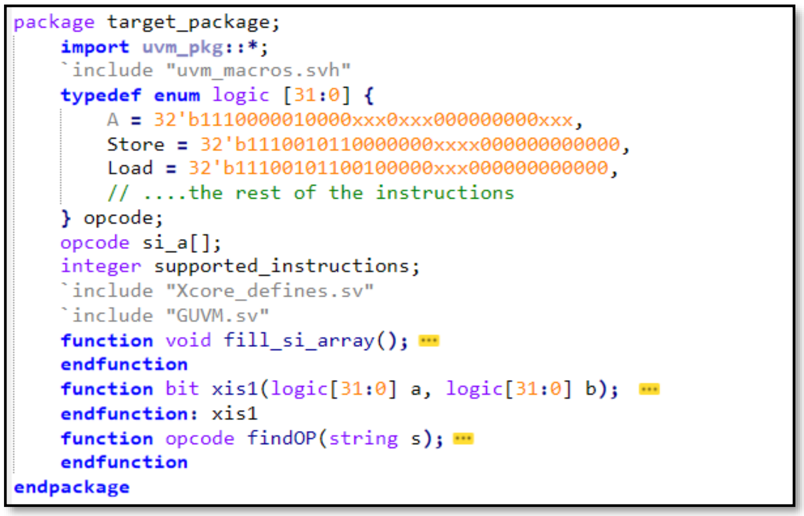


```
module top;
    import uvm_pkg::*;
    import target_package::*;
    `include "uvm_macros.svh"
    logic clk;
    GUVM_interface bfm(clk);
    a23_core dut(
        .port1(bfm.port1),
        //....the rest of the ports
    );
    initial begin
        uvm_config_db #(virtual GUVM_interface)::set(null, "*", "bfm", bfm);
        fill_si_array();
        run_test();
    end
    initial begin
        clk = 0 ;
        forever #10 clk=~clk;
    end
endmodule: top
```

**b.** ***<u>"Xcore_defines.sv":</u>*** A file where some core specific parameters that can be used through the test bench are defined.

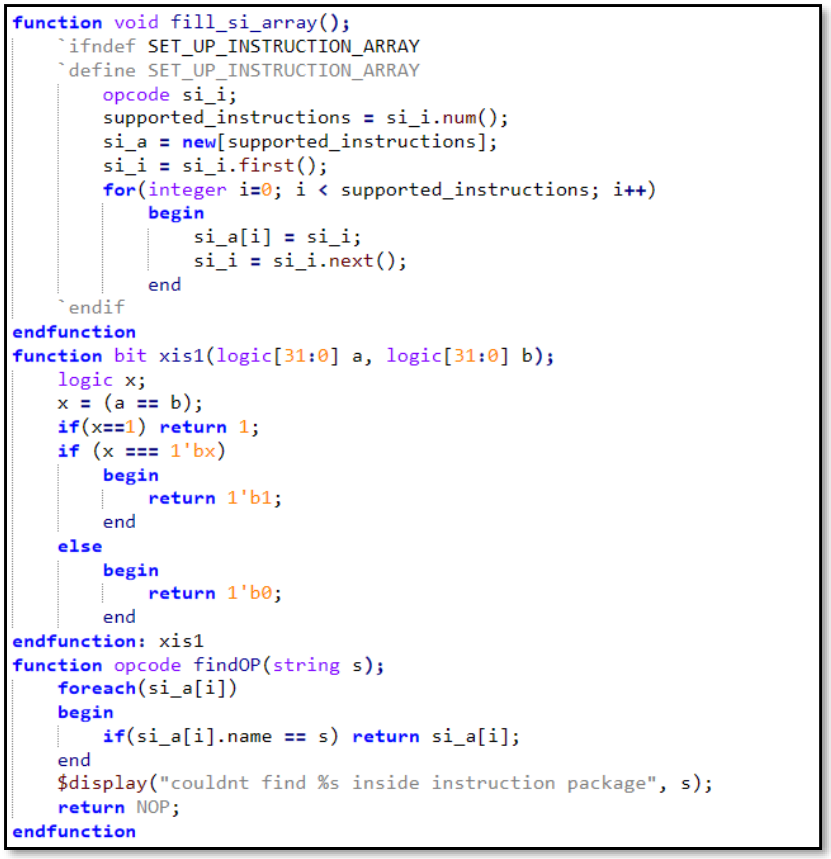
**c.** ***<u>"target_pkg.sv" (≡ "Xcore_pkg.sv"):</u>*** A file where (uvm_pkg), (uvm_macros), (Xcore_defines) and (GUVM) -that includes the main components of the test bench- are included as shown in Listing 19. Then a user-defined enumeration logic data type (opcode) is defined that contains the 32 bits of all the core specific instructions needed to be verified, Then a new array (*si_a[]*) with the user-defined data type (opcode) used to store these opcodes is defined, to be used later with the generic scoreboard (GUVM_scoreboard), this array is filled by the void new function (*fill_si_array()*) that is called by the top module of the test bench (top), where the pre-defined function (*first()*) is used to get the first opcode (instruction), then the pre-defined function (*second()*) is used with the number of the supported instructions (supported_instructions) in a for loop to fill the array with the supported opcodes as shown in Listing 20. In addition to that, a new void function (*xis1()*) is created, that is called by the generic scoreboard (GUVM_scoreboard) to check if the previously explained (verified_inst) -of the coming sequence item- already exists and supported inside the enumeration data type (opcode) or not by comparing the fixed bits (that the constrained randomization don't affect, or change) of (verified_inst) with the bits of all the elements (instructions) of the array (*si_a[]*), and return "1" if they are the same. Then another function (*findOP()*) is created with the (opcode) user-defined data type used by the child sequences to return the opcode of an instruction after getting its name, or argument. More details related to the generic scoreboard (GUVM_scoreboard) are in Section 4.1.j, and more details related to the top module of the test bench are in Section 4.2.a.

**Listing 19 The Xcore/target package (target_pkg).**



```
package target_package;
    import uvm_pkg::*;
    `include "uvm_macros.svh"
    typedef enum logic [31:0] {
        A = 32'b1110000010000xxx0xxx000000000xxx,
        Store = 32'b1110010110000000xxxx000000000000,
        Load = 32'b11100101100100000xxx000000000000,
        // ....the rest of the instructions
    } opcode;
    opcode si_a[];
    integer supported_instructions;
    `include "Xcore_defines.sv"
    `include "GUVM.sv"
    function void fill_si_array();
    endfunction
    function bit xis1(logic[31:0] a, logic[31:0] b);
    endfunction: xis1
    function opcode findOP(string s);
    endfunction
endpackage
```

**Listing 20 The functions of the target package (target_pkg).**

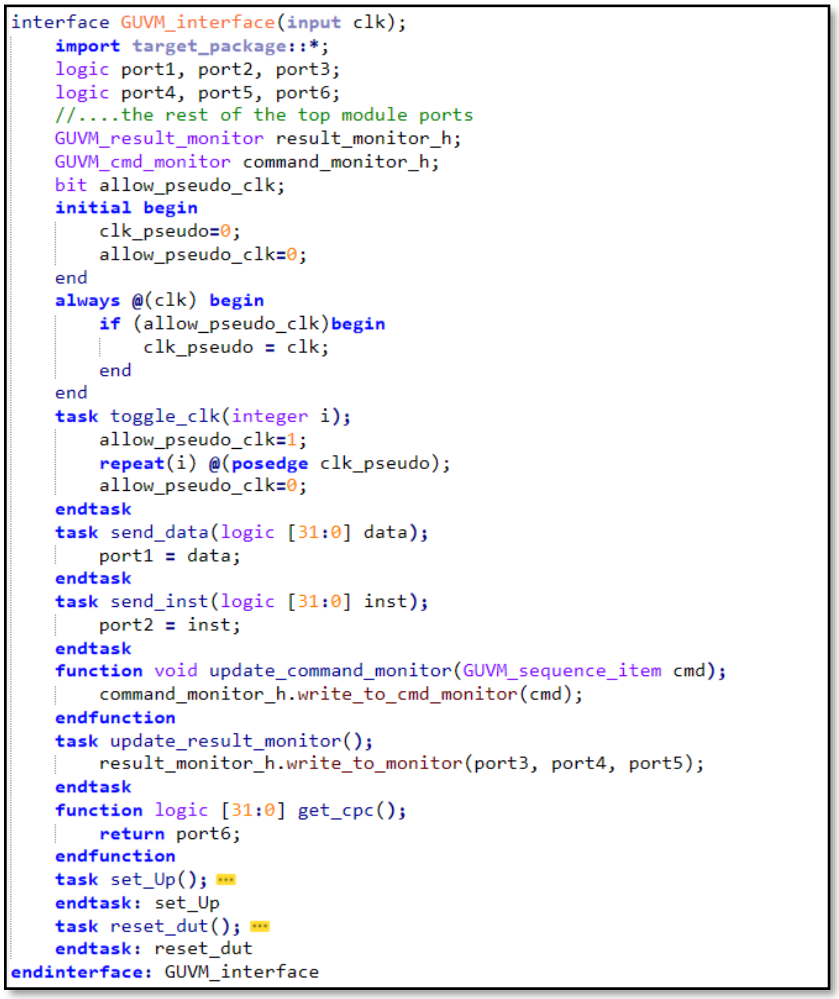


```
function void fill_si_array();
    `ifndef SET_UP_INSTRUCTION_ARRAY
    `define SET_UP_INSTRUCTION_ARRAY
        opcode si_i;
        supported_instructions = si_i.num();
        si_a = new[supported_instructions];
        si_i = si_i.first();
        for(integer i=0; i < supported_instructions; i++)
            begin
                si_a[i] = si_i;
                si_i = si_i.next();
            end
    `endif
endfunction
function bit xis1(logic[31:0] a, logic[31:0] b);
    logic x;
    x = (a == b);
    if(x==1) return 1;
    if (x === 1'bx)
        begin
            return 1'b1;
        end
    else
        begin
            return 1'b0;
        end
endfunction: xis1
function opcode findOP(string s);
    foreach(si_a[i])
    begin
        if(si_a[i].name == s) return si_a[i];
    end
    $display("couldnt find %s inside instruction package", s);
    return NOP;
endfunction
```

**d.** ***<u>"GUVM_interface.sv" (≡ "Xcore_interface.sv"):</u>*** A file where a new interface is created to deal directly with the DUT. Firstly, the target package (target_pkg) is imported, then the ports of the DUT instantiated by the top module (top) are defined, then the generic command monitor (GUVM_cmd_monitor) and the generic result monitor (GUVM_result_monitor) are defined to be used with the contained functions and tasks. A clocking mechanism is created, which is used to stop the clock of the core after the pipeline is completed to reduce the simulation time by creating a pseudu clock (clk_pseudo) that is instantiated in the top module (top) as the clock of the DUT, this (clk_pseudo) follows and copies the top module generated input clock (clk); if toggling the clock is needed, (allow_pseudo_clk) is set to "1", then (clk_pseudo) is toggled for a given number of cycles, then (allow_pseudo_clk) is set back to "0". More details related to the target package (target_pkg) are in Section 4.2.c, and more details related to the generic scoreboard (GUVM_scoreboard) are in Section 4.1.j.

The interface of the DUT (GUVM_interface) as shown in Listing 21, contains a group of functions, and tasks used by the generic driver (GUVM_driver) to control the input and the outputs of the DUT as following:

**Listing 21 The core (DUT) interface (GUVM_interface).**



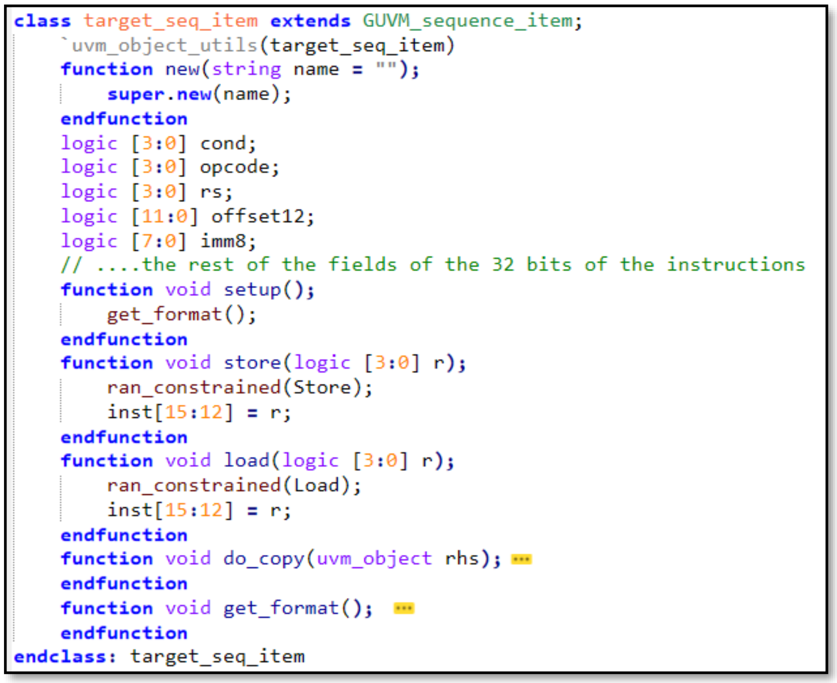
```
interface GUVM_interface(input clk);
    import target_package::*;
    logic port1, port2, port3;
    logic port4, port5, port6;
    //....the rest of the top module ports
    GUVM_result_monitor result_monitor_h;
    GUVM_cmd_monitor command_monitor_h;
    bit allow_pseudo_clk;
    initial begin
        clk_pseudo=0;
        allow_pseudo_clk=0;
    end
    always @(clk) begin
        if (allow_pseudo_clk)begin
            clk_pseudo = clk;
        end
    end
    task toggle_clk(integer i);
        allow_pseudo_clk=1;
        repeat(i) @(posedge clk_pseudo);
        allow_pseudo_clk=0;
    endtask
    task send_data(logic [31:0] data);
        port1 = data;
    endtask
    task send_inst(logic [31:0] inst);
        port2 = inst;
    endtask
    function void update_command_monitor(GUVM_sequence_item cmd);
        command_monitor_h.write_to_cmd_monitor(cmd);
    endfunction
    task update_result_monitor();
        result_monitor_h.write_to_monitor(port3, port4, port5);
    endtask
    function logic [31:0] get_cpc();
        return port6;
    endfunction
    task set_Up(); ...
    endtask: set_Up
    task reset_dut(); ...
    endtask: reset_dut
endinterface: GUVM_interface
```

- ***toggle_clk():*** Task used to toggle the DUT clock for (i) given times with the (clk_pseudo) clocking mechanism.
- ***reset_dut():*** Task used to reset the DUT (the pipeline of the core).
- ***set_Up():*** Task used to setup the DUT with the proper inputs and signals, these signals control the modes of operation of the DUT, the interrupt status, etc.
- **send_data():** Task used to send the data contents of the incoming sequence items to the DUT through the proper port.

- ***send_inst():*** Task used to send the instruction content (inst) of the coming sequence items to the DUT through the proper port.
- ***update_result_monitor():*** Task used to send the outputs from the DUT to the generic result monitor (GUVM_result_monitor).
- ***update_command_monitor():*** Function used to send the incoming sequence item -from the  generic driver (GUVM_driver) to the DUT- to the generic command monitor (GUVM_cmd_monitor).

**e.** ***"target_sequence_item.sv" (≡ "Xcore_sequ_item"):*** A file where the target sequence item (target_seq_item) is extended from the parent sequence item (GUVM_sequence_item), it is registered with the UVM factory as shown in Listing 22, then the constructor of this new class is declared. Then the fields of instructions of the core (the formats of the instructions) are defined, as each core has a different way of dividing the fields of the 32-bit instructions.

**Listing 22 The target sequence item (target_seq_item).**



```
class target_seq_item extends GUVM_sequence_item;
    `uvm_object_utils(target_seq_item)
    function new(string name = "");
        super.new(name);
    endfunction
    logic [3:0] cond;
    logic [3:0] opcode;
    logic [3:0] rs;
    logic [11:0] offset12;
    logic [7:0] imm8;
    // ....the rest of the fields of the 32 bits of the instructions
    function void setup();
        get_format();
    endfunction
    function void store(logic [3:0] r);
        ran_constrained(Store);
        inst[15:12] = r;
    endfunction
    function void load(logic [3:0] r);
        ran_constrained(Load);
        inst[15:12] = r;
    endfunction
    function void do_copy(uvm_object rhs);
    endfunction
    function void get_format();
    endfunction
endclass: target_seq_item
```

The target sequence item (target_seq_item) contains a group of functions that can be called by the child sequences as following:

- ***setup():*** Function used by the child sequences to get the formats of the instructions needed to be verified by calling (*get_format()*) function.
- ***get_format():*** Function used to specify the bits of every defined field from the 32 bits of the instruction.
- ***store():*** Function used to randomize the store instruction (store) using (*rand_constrained()*) function of the parent sequence item (GUVM_sequence_item) and specify the address of the destination register.
- ***load():*** Function used to randomize the load instruction (load) using (*rand_constrained()*) function of the parent sequence item

(GUVM_sequence_item) and specify the address of the source and load register.

- **do_copy():** Function used if desired to copy the contents of the sequence item (the fields of the formats of the instructions in this case).

## 4.3 run:

This sub-folder contains a python script file (run) connected to a number of TCL files to run the test bench and verify the desired DUT (from the proposed three open-source cores), these TCL files used to compile the DUT and the test bench files.

- ***“run.py”:*** A python script file used with a great flexibility to control the test bench, it uses the TCL files to compile the DUT and the test bench as desired: it provides compiling only the DUT, compiling only the test bench, or compiling both of them. After compiling, the simulation is started by choosing the suitable test to simulate the top module of the test bench as shown in Fig. 5.

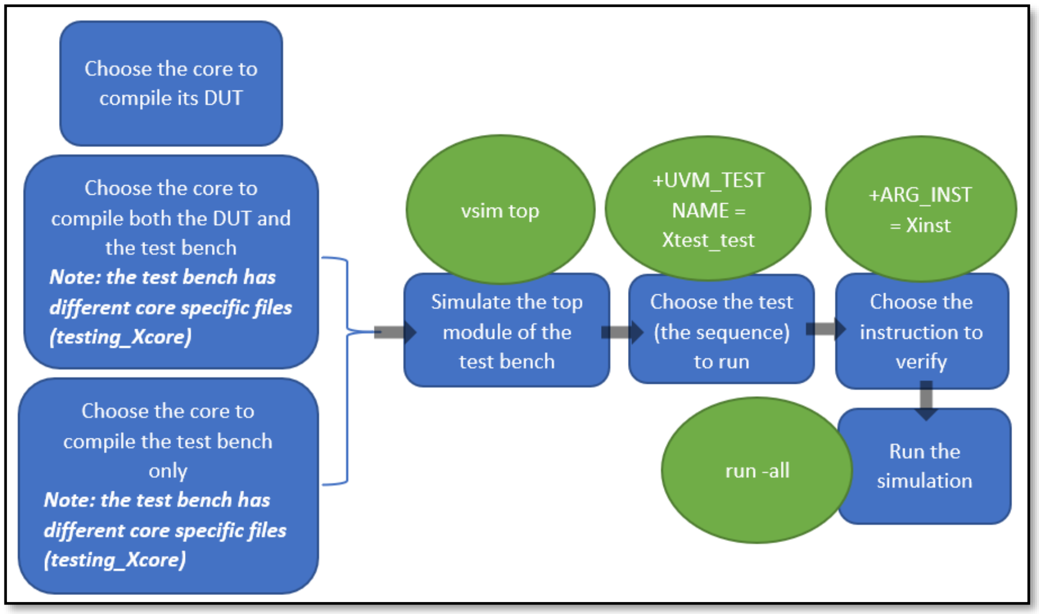

Choose the core to compile its DUT
Choose the core to compile both the DUT and the test bench
Note: the test bench has different core specific files (testing_Xcore)
Choose the core to compile the test bench only
Note: the test bench has different core specific files (testing_Xcore)
vsim top
Simulate the top module of the test bench
+UVM_TEST NAME = Xtest_test
Choose the test (the sequence) to run
+ARG_INST = Xinst
Choose the instruction to verify
run -all
Run the simulation

**Fig. 5 The flow of the python script (run file).**

For example, add_test, this test triggers the related child sequence such as add_seq, then the needed instruction to be verified is chosen, by choosing the argument related to this instruction, like “A” for the addition instruction, then the simulation is started.

# CHPATER 5: AN ILLUSTRATIVE EXAMPLE

In this chapter an illustrative example is shown in steps how the functionality of the addition instruction (add) is tested, starting from the python script all the way to the outputs, and results transcript as following:

1. The DUT is compiled and the test bench too.
2. The top module of the test bench is simulated, where the DUT and the interface of the DUT (Xcore_interface) are instantiated, and then choosing the test to run.
3. The child test -extended from the parent test (GUVM_test)- of the addition instruction (add_test) is chosen from the python script (run) file that will trigger the child sequence (add_seq) related to this instruction, or choosing the instruction first, to simulate it, if the chosen sequence is used for more than one instruction. This child test additionally triggers the generic environment (GUVM_env).
4. The generic environment (GUVM_env) triggers the generic scoreboard (GUVM_scoreboard), the monitors with their ports, and the generic agent (GUVM_agent).
5. The generic agent (GUVM_agent) triggers the sequencer and the generic driver (GUVM_driver).
6. For the sequencer, the parent sequence item (GUVM_sequence_item), and the child (target) sequence item (target_seq_item) are used to generate the constrained-randomized stimuli with respect to the child sequence of the addition instruction.
7. The driver receives the sequence items generated with respect to the behavior of the sequence, and send them to the interface of the DUT as shown in Fig 6.

8. The interface sends the received sequence item to the DUT for processing, and to the generic command monitor (GUVM_cmd_monitor) where they are delivered to the generic history transaction (GUVM_history_transaction) and saved to be used later with the scoreboard.
9. The DUT sends the results to the interface, then the interface delivers them to the generic result transaction (GUVM) where they will be received by the generic history transaction (GUVM_history_transaction) and saved to be used later with the scoreboard.
10. The generic scoreboard (GUVM_scoreboard) calls the header file related to the addition instruction where it generates golden references from the saved sequence items and compares them with the actual outputs of the DUT defining if the instruction successfully passes the functionality test or not.

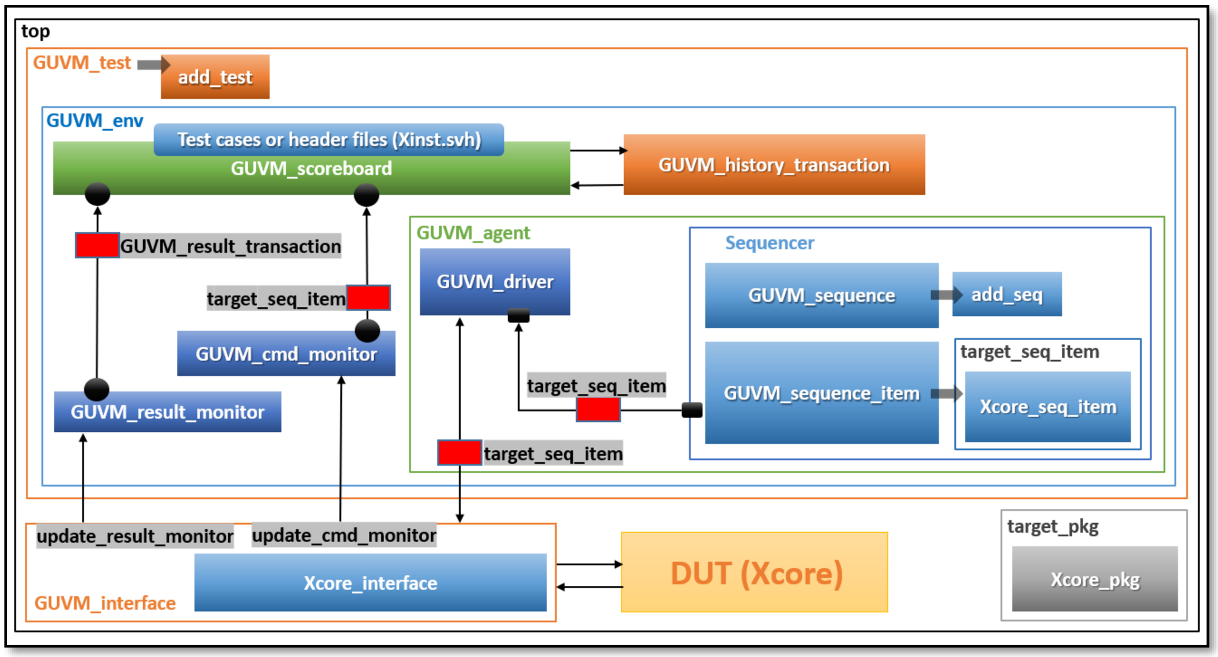


**Fig. 6 Generic UVM for Soft Processors Test Bench Architecture.**

# CHPATER 6: VALIDATION AND RESULTS

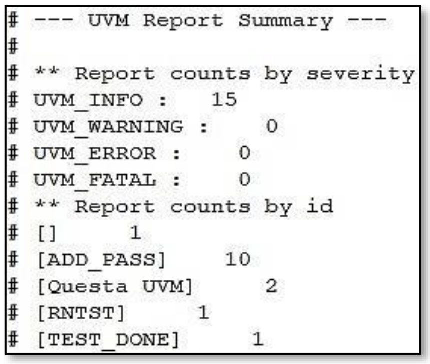
This chapter discusses the test bench experimental results including the UVM Report Summary. Moreover, the studied and the implemented cases including the testing scenarios like the verification of ALU instructions, load and store instructions, and jump instructions. In addition to that, the performance evaluation, comparison with related work, and limitations.

## 6.1 Hardware Experimental Results

After running the required test through python as discussed before, results of the test are logged in transcript text file in folder “trans” in the running file directory.

First, there is a summary at the end of the results the UVM report that illustrates types and number of UVM reporting statements that are triggered in simulation and the number of passed and failed test cases as shown in Listing 23.

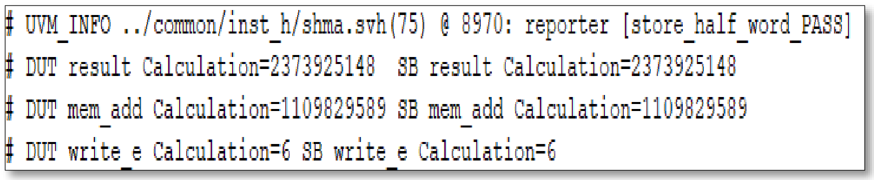
**Listing 23 UVM report summary for add test**



```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   15
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# []     1
# [ADD_PASS]    10
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
```

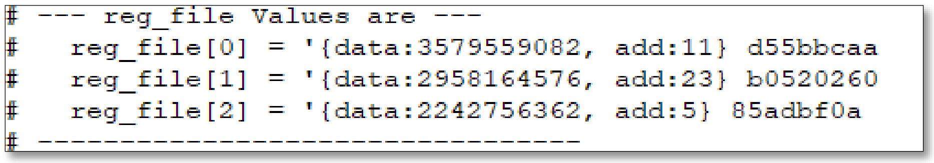
Secondly, there is a detailed report for each sequence iteration. UVM report consists of three parts: first part is displayed by the scoreboard after comparing expected results calculated by the scoreboard with actual results that come from the DUT by explaining these results containing  the main result of the test case whether it is pass or fail as shown in Listing 24, second part is the dynamic register file which is displayed by the history transaction that contains the final values that should be saved in the core's register file as shown in Listing 25, the last part is a table which is furthermore displayed by the history transaction that contains the values of the most important ports saved after sending each sequence item to the core as shown in Listing 26 and Listing 27, those illustrate add test case instruction.

**Listing 24 First part of the detailed report for store half word test.**



```
# UVM_INFO ../common/inst_h/shma.svh(75) @ 8970: reporter [store_half_word_PASS]
# DUT result Calculation=2373925148  SB result Calculation=2373925148
# DUT mem_add Calculation=1109829589 SB mem_add Calculation=1109829589
# DUT write_e Calculation=6 SB write_e Calculation=6
```

**Listing 25 The dynamic register file saved.**



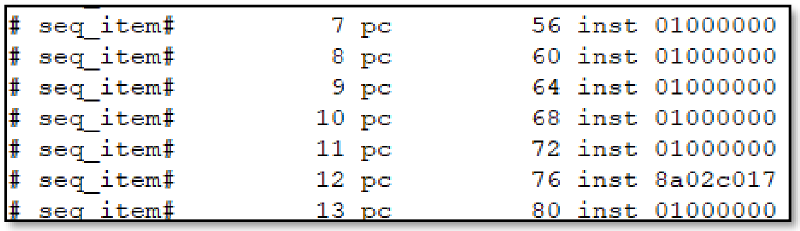
```
# --- reg_file Values are ---
#   reg_file[0] = '{data:3579559082, add:11} d55bbcaa
#   reg_file[1] = '{data:2958164576, add:23} b0520260
#   reg_file[2] = '{data:2242756362, add:5} 85adbf0a
# ---------------------------------
```

A brief description of these ports is as follows:

- seq_item#: Is the number of the sent sequence item.
- pc: Contains the address of the instruction currently being executed by the Integer Unit.
- inst: Is the port that instruction cache supposed to use for sending instruction to the core (in case of common instruction and data cache this port is common too as in

amber23 core when data is sent through this port to the core its saved and displayed at inst).

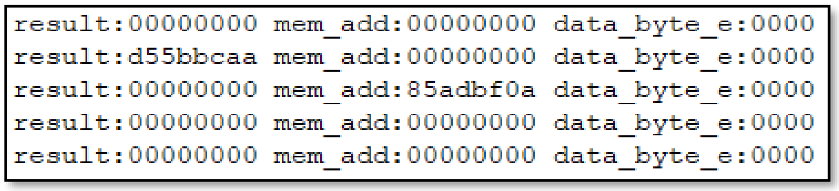
**Listing 26 History table contents for add test.**



```
# seq_item#          7 pc         56 inst 01000000
# seq_item#          8 pc         60 inst 01000000
# seq_item#          9 pc         64 inst 01000000
# seq_item#         10 pc         68 inst 01000000
# seq_item#         11 pc         72 inst 01000000
# seq_item#         12 pc         76 inst 8a02c017
# seq_item#         13 pc         80 inst 01000000
```

- <u>result:</u> Is the port that core uses for sending data to data cache.
- <u>mem_add:</u> Is the port that the core uses for sending address to data cache.
- <u>data_byte_e:</u> Is the port that the core uses for sending data byte enable to data cache which is important for verifying the type of instructions that makes transaction to one or two bytes of data.

**Listing 27 The rest of history table content.**



```
result:00000000 mem_add:00000000 data_byte_e:0000
result:d55bbcaa mem_add:00000000 data_byte_e:0000
result:00000000 mem_add:85adbf0a data_byte_e:0000
result:00000000 mem_add:00000000 data_byte_e:0000
result:00000000 mem_add:00000000 data_byte_e:0000
```

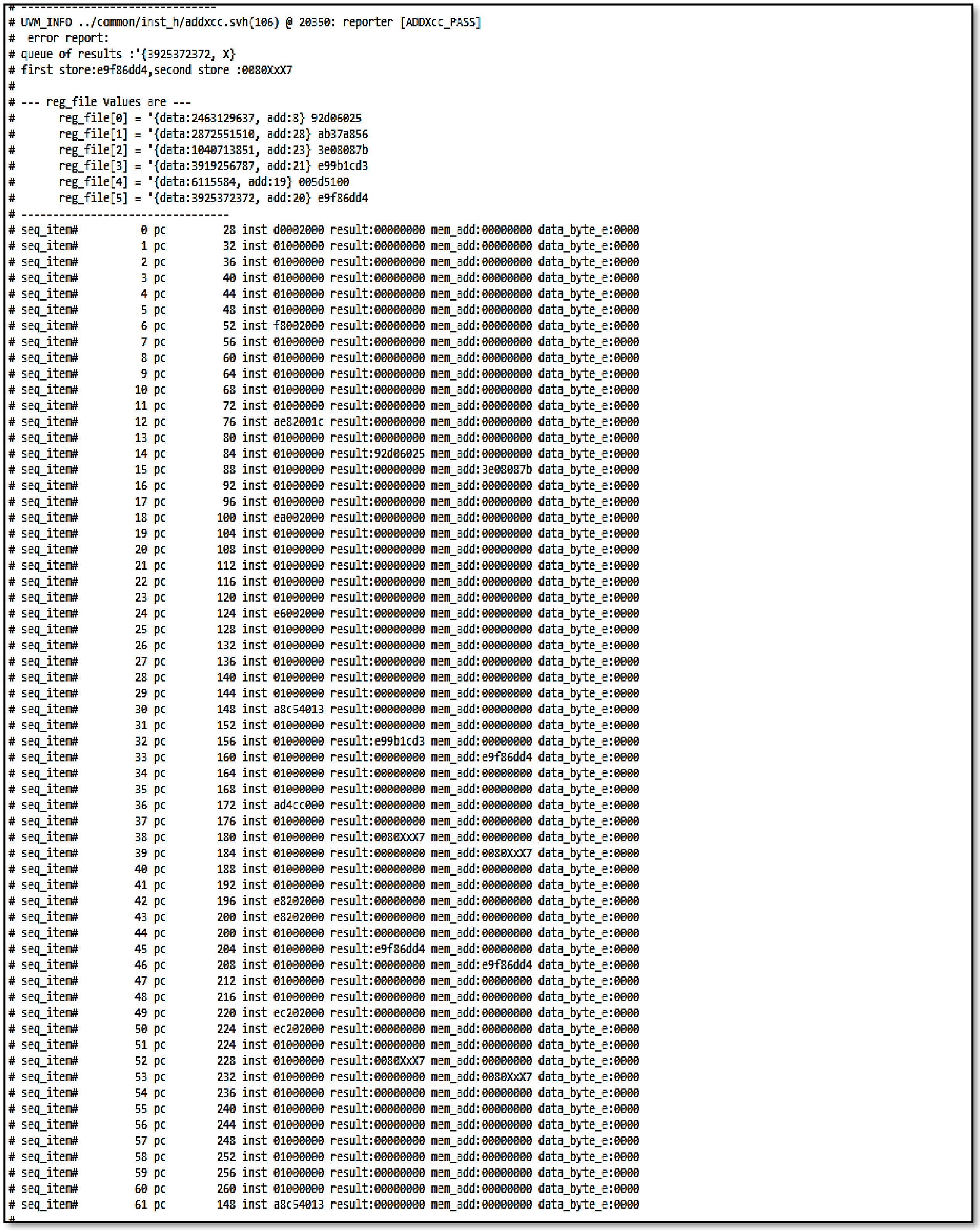
## 6.2 Case Studies

The methodology that is used for the verification of the three soft processors (RI5CY, LEON 2.4, and Amber a23) focuses on proving the concept of a generic UVM architecture that is able to verify various soft cores with the minimum effort of modifications and adjusting in the code of this UVM test bench. Therefore, the highest priority is given to the cases of instructions that are common in the three cores which are used to prove that concept. There are different types of instructions that are common in these three cores and it is covered most of them such that:

### 6.2.1 **ALU Instructions:**

Most of them are common in the test case that is used in their verification, they only differ in the used operators (e.g., +, -, *, /, XOR, OR). Checking the result of the arithmetic operation is enough to verify this type of functionality. In addition to that, there are arithmetic instructions which use flags and modify them such as (add with carry, sub and modify flags) which are verified by checking the result of the arithmetic operation considering carry flag before operation and arithmetic operation effect on the core flags. Listing 28 shows the test bench results for add with carry and modify flags instruction in LEON 2.4 core.

**Listing 28 Output of add with carry test**

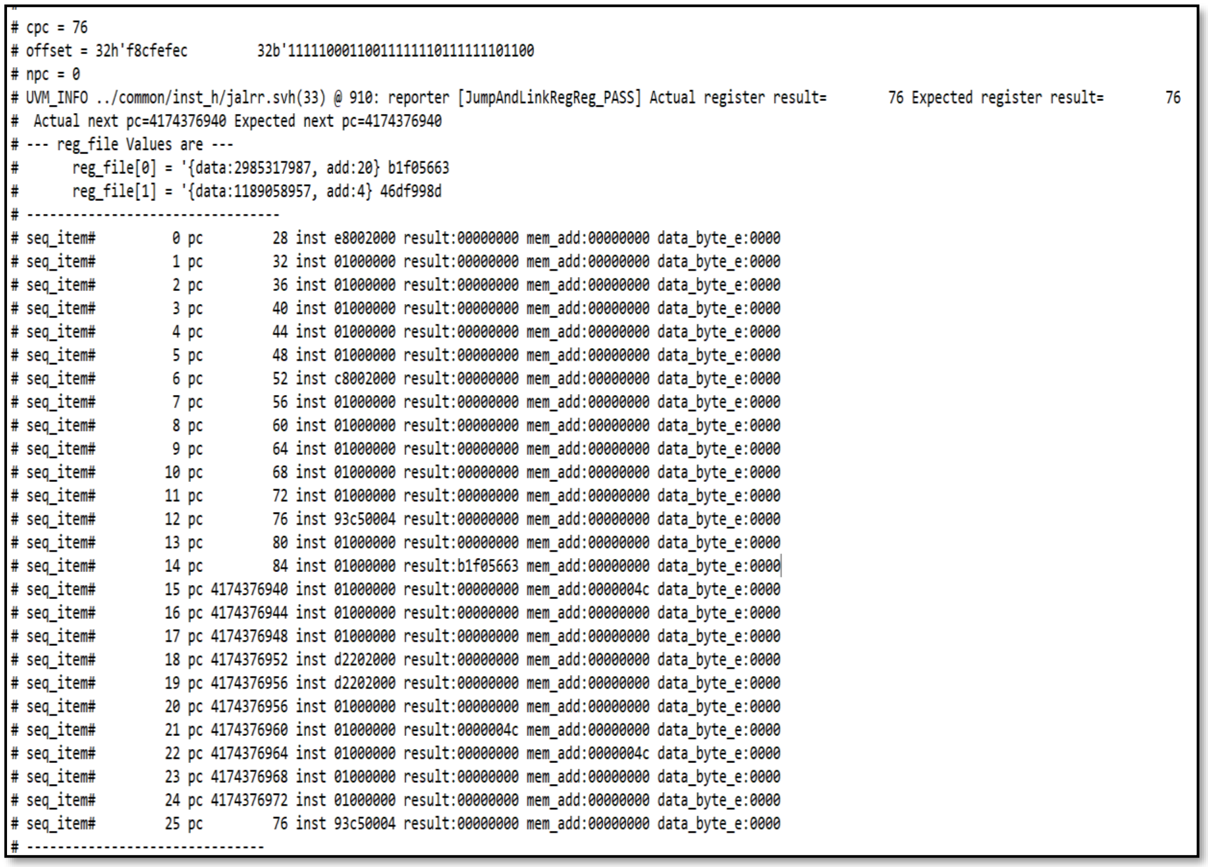


```
# --------------------------------
# UVM_INFO ../common/inst_h/addxcc.svh(106) @ 20350: reporter [ADDXcc_PASS]
#  error report:
# queue of results :'{3925372372, X}
# first store:e9f86dd4,second store :0080XXX7
#
# --- reg_file Values are ---
#        reg_file[0] = '{data:2463129637, add:8} 92d06025
#        reg_file[1] = '{data:2872551510, add:28} ab37a856
#        reg_file[2] = '{data:1040713851, add:23} 3e08087b
#        reg_file[3] = '{data:3919256787, add:21} e99b1cd3
#        reg_file[4] = '{data:6115584, add:19} 005d5100
#        reg_file[5] = '{data:3925372372, add:20} e9f86dd4
# ---------------------------------
# seq_item#         0 pc        28 inst d0002000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         1 pc        32 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         2 pc        36 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         3 pc        40 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         4 pc        44 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         5 pc        48 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         6 pc        52 inst f8002000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         7 pc        56 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         8 pc        60 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         9 pc        64 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        10 pc        68 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        11 pc        72 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        12 pc        76 inst ae82001c result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        13 pc        80 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        14 pc        84 inst 01000000 result:92d06025 mem_add:00000000 data_byte_e:0000
# seq_item#        15 pc        88 inst 01000000 result:00000000 mem_add:3e08087b data_byte_e:0000
# seq_item#        16 pc        92 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        17 pc        96 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        18 pc       100 inst ea002000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        19 pc       104 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        20 pc       108 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        21 pc       112 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        22 pc       116 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        23 pc       120 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        24 pc       124 inst e6002000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        25 pc       128 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        26 pc       132 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        27 pc       136 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        28 pc       140 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        29 pc       144 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        30 pc       148 inst a8c54013 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        31 pc       152 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        32 pc       156 inst 01000000 result:e99b1cd3 mem_add:00000000 data_byte_e:0000
# seq_item#        33 pc       160 inst 01000000 result:00000000 mem_add:e9f86dd4 data_byte_e:0000
# seq_item#        34 pc       164 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        35 pc       168 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        36 pc       172 inst ad4cc000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        37 pc       176 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        38 pc       180 inst 01000000 result:0080XxX7 mem_add:00000000 data_byte_e:0000
# seq_item#        39 pc       184 inst 01000000 result:00000000 mem_add:0080XxX7 data_byte_e:0000
# seq_item#        40 pc       188 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        41 pc       192 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        42 pc       196 inst e8202000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        43 pc       200 inst e8202000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        44 pc       200 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        45 pc       204 inst 01000000 result:e9f86dd4 mem_add:00000000 data_byte_e:0000
# seq_item#        46 pc       208 inst 01000000 result:00000000 mem_add:e9f86dd4 data_byte_e:0000
# seq_item#        47 pc       212 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        48 pc       216 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        49 pc       220 inst ec202000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        50 pc       224 inst ec202000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        51 pc       224 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        52 pc       228 inst 01000000 result:0080XxX7 mem_add:00000000 data_byte_e:0000
# seq_item#        53 pc       232 inst 01000000 result:00000000 mem_add:0080XxX7 data_byte_e:0000
# seq_item#        54 pc       236 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        55 pc       240 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        56 pc       244 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        57 pc       248 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        58 pc       252 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        59 pc       256 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        60 pc       260 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        61 pc       148 inst a8c54013 result:00000000 mem_add:00000000 data_byte_e:0000
#
```

### 6.2.2 <u>**Jump Instructions:**</u>

Verified by checking pc value to be changed to the required address. In addition to that, there is Jump and Link instruction, which needs to check pc value and value in the register which saves ‘current pc’ and in another case the register saves ‘current pc + 1’. After studying jump cases in three cores it is noticed that cores behave differently when instruction cache address misalignment happens where LEON core crashes when misalignment happens and Amber core floors the address to the nearest multiple of four. Listing 29 shows the test bench results for Jump and Link instruction in LEON 2.4 core.

**Listing 29 Test bench results for jump and link instruction.**

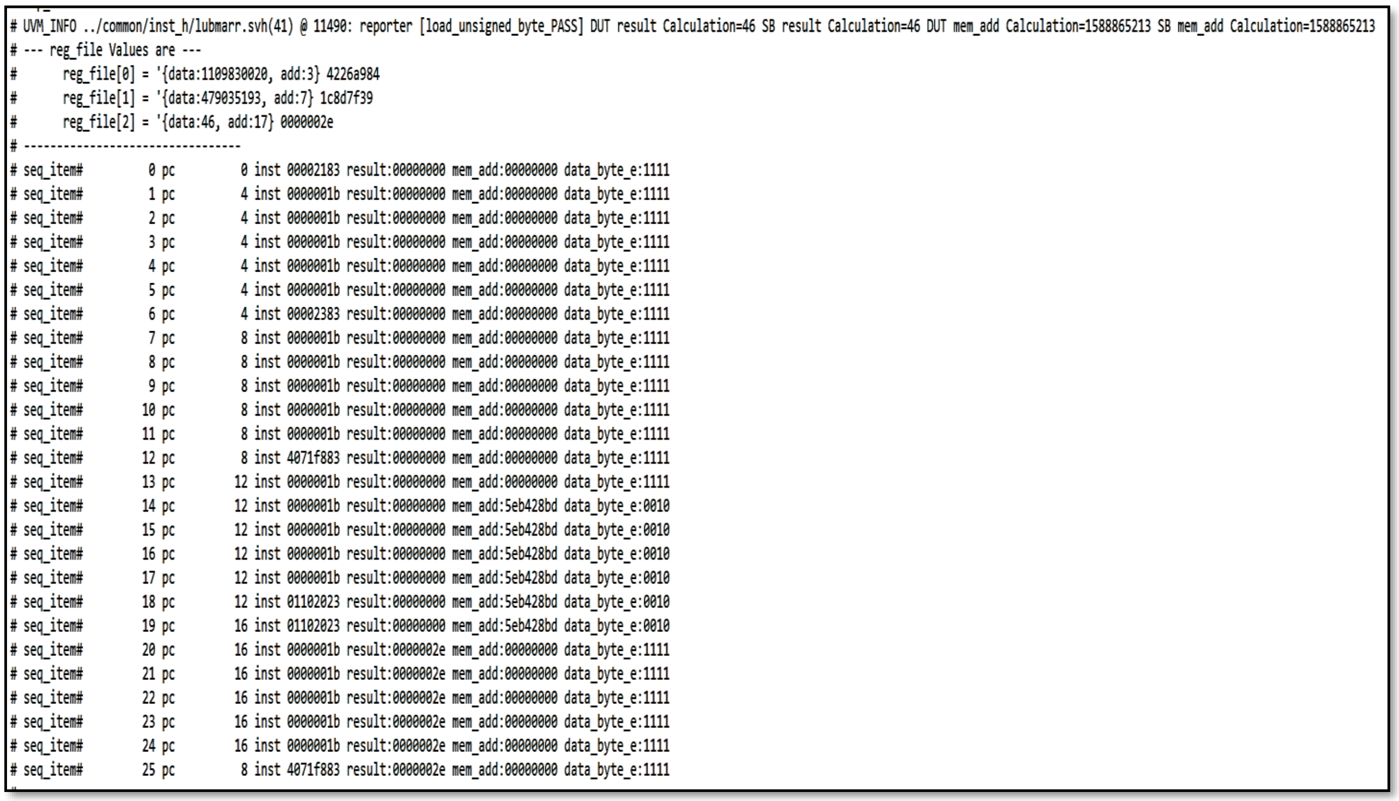


```
# cpc = 76
# offset = 32h'f8cfefec        32b'11111000110011111110111111101100
# npc = 0
# UVM_INFO ../common/inst_h/jalrr.svh(33) @ 910: reporter [JumpAndLinkRegReg_PASS] Actual register result=       76 Expected register result=       76
#  Actual next pc=4174376940 Expected next pc=4174376940
# --- reg_file Values are ---
#      reg_file[0] = '{data:2985317987, add:20} b1f05663
#      reg_file[1] = '{data:1189058957, add:4} 46df998d
# ---------------------------------
# seq_item#         0 pc        28 inst e8002000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         1 pc        32 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         2 pc        36 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         3 pc        40 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         4 pc        44 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         5 pc        48 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         6 pc        52 inst c8002000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         7 pc        56 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         8 pc        60 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#         9 pc        64 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        10 pc        68 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        11 pc        72 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        12 pc        76 inst 93c50004 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        13 pc        80 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        14 pc        84 inst 01000000 result:b1f05663 mem_add:00000000 data_byte_e:0000
# seq_item#        15 pc 4174376940 inst 01000000 result:00000000 mem_add:0000004c data_byte_e:0000
# seq_item#        16 pc 4174376944 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        17 pc 4174376948 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        18 pc 4174376952 inst d2202000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        19 pc 4174376956 inst d2202000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        20 pc 4174376956 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        21 pc 4174376960 inst 01000000 result:0000004c mem_add:00000000 data_byte_e:0000
# seq_item#        22 pc 4174376964 inst 01000000 result:00000000 mem_add:0000004c data_byte_e:0000
# seq_item#        23 pc 4174376968 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        24 pc 4174376972 inst 01000000 result:00000000 mem_add:00000000 data_byte_e:0000
# seq_item#        25 pc        76 inst 93c50004 result:00000000 mem_add:00000000 data_byte_e:0000
# -------------------------------
```

### 6.2.3 Transaction and Memory Interaction Instructions:

They are instructions (load, store, and swap) which make transactions of data between cache memory and core. After studying transaction cases in three cores it has been noticed that the cores behave differently when data cache address misalignment happens (e.g., RISCY core has misalignment feature and redistributes the word depending on least significant two bits in the address and LEON core crashes when misalignment happens). Although, RI5CY core and Amber core have misalignment feature, however, they redistributed the word differently which required different verifying cases to verify each case of those. Listing 30 shows the test bench results for Load Unsigned Byte instruction in RI5CY core.

**Listing 30 Test bench results for load unsigned byte instruction.**

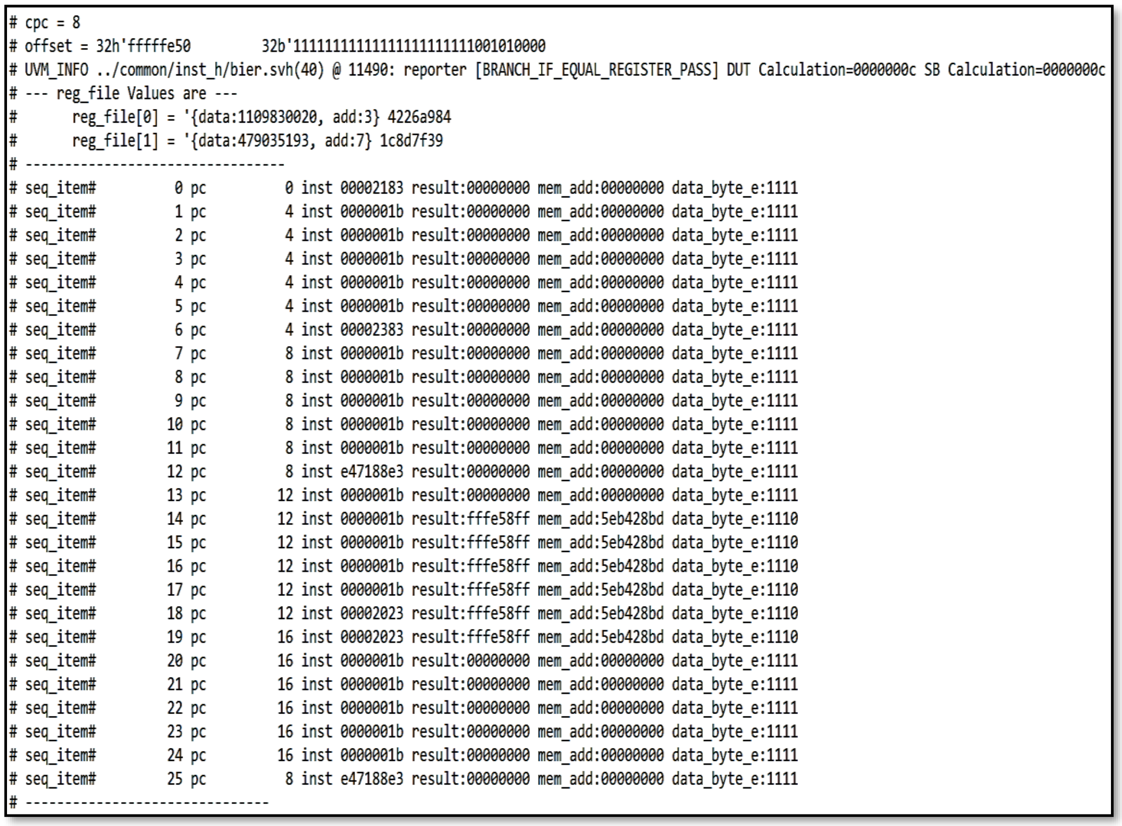


```
# UVM_INFO ../common/inst_h/lubmarr.svh(41) @ 11490: reporter [load_unsigned_byte_PASS] DUT result Calculation=46 SB result Calculation=46 DUT mem_add Calculation=1588865213 SB mem_add Calculation=1588865213
# --- reg_file Values are ---
#       reg_file[0] = '{data:1109830020, add:3} 4226a984
#       reg_file[1] = '{data:479035193, add:7} 1c8d7f39
#       reg_file[2] = '{data:46, add:17} 0000002e
# ---------------------------------
# seq_item#         0 pc         0 inst 00002183 result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         1 pc         4 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         2 pc         4 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         3 pc         4 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         4 pc         4 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         5 pc         4 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         6 pc         4 inst 00002383 result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         7 pc         8 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         8 pc         8 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         9 pc         8 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        10 pc         8 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        11 pc         8 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        12 pc         8 inst 4071f883 result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        13 pc        12 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        14 pc        12 inst 0000001b result:00000000 mem_add:5eb428bd data_byte_e:0010
# seq_item#        15 pc        12 inst 0000001b result:00000000 mem_add:5eb428bd data_byte_e:0010
# seq_item#        16 pc        12 inst 0000001b result:00000000 mem_add:5eb428bd data_byte_e:0010
# seq_item#        17 pc        12 inst 0000001b result:00000000 mem_add:5eb428bd data_byte_e:0010
# seq_item#        18 pc        12 inst 01102023 result:00000000 mem_add:5eb428bd data_byte_e:0010
# seq_item#        19 pc        16 inst 01102023 result:00000000 mem_add:5eb428bd data_byte_e:0010
# seq_item#        20 pc        16 inst 0000001b result:0000002e mem_add:00000000 data_byte_e:1111
# seq_item#        21 pc        16 inst 0000001b result:0000002e mem_add:00000000 data_byte_e:1111
# seq_item#        22 pc        16 inst 0000001b result:0000002e mem_add:00000000 data_byte_e:1111
# seq_item#        23 pc        16 inst 0000001b result:0000002e mem_add:00000000 data_byte_e:1111
# seq_item#        24 pc        16 inst 0000001b result:0000002e mem_add:00000000 data_byte_e:1111
# seq_item#        25 pc         8 inst 4071f883 result:0000002e mem_add:00000000 data_byte_e:1111
```

### 6.2.4 **Branch Instructions:**

This type of instructions needs to check condition to execute the branch operation, flag or result of operation, then check if the branch is taken or not and then check the address that core has to jump to. At this point, the difference of the cores' behavior to address misalignment is clear, in addition to the difference in the flags or operations that are used for the condition. Listing 31 shows the test bench results for Branch if equal instruction in RI5CY core.

**Listing 31 Test bench results for branch if equal**



```
# cpc = 8
# offset = 32h'fffffe50        32b'11111111111111111111111001010000
# UVM_INFO ../common/inst_h/bier.svh(40) @ 11490: reporter [BRANCH_IF_EQUAL_REGISTER_PASS] DUT Calculation=0000000c SB Calculation=0000000c
# --- reg_file Values are ---
#       reg_file[0] = '{data:1109830020, add:3} 4226a984
#       reg_file[1] = '{data:479035193, add:7} 1c8d7f39
# ---------------------------------
# seq_item#         0 pc         0 inst 00002183 result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         1 pc         4 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         2 pc         4 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         3 pc         4 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         4 pc         4 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         5 pc         4 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         6 pc         4 inst 00002383 result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         7 pc         8 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         8 pc         8 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#         9 pc         8 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        10 pc         8 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        11 pc         8 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        12 pc         8 inst e47188e3 result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        13 pc        12 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        14 pc        12 inst 0000001b result:fffe58ff mem_add:5eb428bd data_byte_e:1110
# seq_item#        15 pc        12 inst 0000001b result:fffe58ff mem_add:5eb428bd data_byte_e:1110
# seq_item#        16 pc        12 inst 0000001b result:fffe58ff mem_add:5eb428bd data_byte_e:1110
# seq_item#        17 pc        12 inst 0000001b result:fffe58ff mem_add:5eb428bd data_byte_e:1110
# seq_item#        18 pc        12 inst 00002023 result:fffe58ff mem_add:5eb428bd data_byte_e:1110
# seq_item#        19 pc        16 inst 00002023 result:fffe58ff mem_add:5eb428bd data_byte_e:1110
# seq_item#        20 pc        16 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        21 pc        16 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        22 pc        16 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        23 pc        16 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        24 pc        16 inst 0000001b result:00000000 mem_add:00000000 data_byte_e:1111
# seq_item#        25 pc         8 inst e47188e3 result:00000000 mem_add:00000000 data_byte_e:1111
# -------------------------------
```

## 6.3 Performance Evaluation

This section will discuss the performance metrics of the GUVM test bench and what are its strength points.

- Reusability**:** The test bench is written in a way that has been achieved and discussed throughout the chapters such that it can be reused on different processors.
- Test plan coverage***:*** It is to verify instructions that are common in the three ISAs and some of the uniquely owned for each ISA to prove the flexibility of the test bench.
- Flexibility**:** Since each processor has its own way to handle the data and its interface with the memory, the way of implementing the register file and its dealing with the flags and their storage. Therefore, it has been a must to implement one method to deal with the differences and get the needed data without changing the core of the test bench.
- Bug detection**:** no bug was found in the instructions that has been tested yet. They all act according to their respective ISA except the failed test cases that are expected and are considered as a part of the negative testing layer that has been discussed in the verification plan.

## 6.4 Comparison with Related Work

Some features in the conventional test bench had to be edited and modified in order to support different ISAs, they will be discussed in this section.

- Randomization**:** Each ISA has its own instructions and instruction format with different fields in the instructions to be randomized, to overcome this problem a package was made for each processor in this package exists an enumerator (enum data type) table that tells the test bench what are the instructions supported by this ISA and what are the fields that need to be randomized.

- <u>Core dependent code</u>**:** The test bench was split in two categories; common code and core dependent code. The code dependent code was kept to a minimum and for the things that can vary vastly across the different implementations of the same ISA. This core dependent code contains the interface, package of instructions and instruction format.
- Communication protocols: In this test bench all communication protocols are being handled by the interface itself since each processor has its own signals and protocols. The driver communicated with the interface in a form of methods that all interfaces has in common, however applying the method itself varies vastly even in processors with same ISA.
- <u>Scoreboard:</u> Scoreboard has an adaptive history in the form of a dynamic array that can adapt to different register files with different sizes.

## 6.5 Limitations

Thus far, a limitation to this implementation is whether the instruction is already supported by this test bench or not. The instructions that are supported by the previously mentioned ISAs or any similar instructions based on similar ISA of the same category should be easily implemented with minimum effort. While the instructions that are completely different and are based on different ISAs (different categories) might face challenges with the proposed implementation, however it is believed that they can be tested with the same method. In addition to that, the other limitation is that Load, and Store instructions have to be executed correctly without errors.

# CHPATER 7: CONCLUSION AND FUTURE WORK

## 7.1 Conclusion

The huge increase in the complexity of processors’ microarchitectures and the wide variety of extended algorithms that are executed by different ISA’s leads to creating a gap between verification requirements and the implementation of a generic/reusable test bench for testing different soft processors. This paper has presented a general end to end verification environment using UVM for verifying soft processors. This work has explored the capabilities of UVM to be used efficiently in processor functional verification and a new verification methodology has been proposed such that it can be easily utilized for different instruction set architectures or micro-architectures. The proposed work has involved a practical example of three different soft processors based on three different instruction set architectures (RI5CY, LEON 2.4, and Amber 23), each one has its own specifications, behavior. The main goal is to try to solve the problem of microprocessors' verification which cycles a long computational effort and time as discussed before. Therefore, the solution is purely based on open source methodology UVM and standard System Verilog to build a generic and reusable environment for verifying different soft cores. In addition to that, a well-structured bottom-up stimuli generation solution have been implemented to make a good use of the object-oriented capabilities available in UVM base classes. These stimuli are used to test the functionality of the mentioned cores during memory access and handling instructions. Clearly this process needs some accurate steps to be followed to achieve the best results, this is what is called a verification plan as discussed in the previous sections.

## 7.2 Future Work

After the concept of making a unified, generic and reusable test bench verification environment has been proven, the future work includes development of this concept to verify a complete System-on-Chip (SoC). Moreover, Code and functionality coverage can be furthermore considered to make sure that no corner cases or verification holes are left unchecked. Finally, it's planned to use machine learning algorithms to generate the required stimulus and test the design entirely.

# REFERENCES

[1] What's the Deal with SoC Verification?", Electronic Design, 2020. [Online]. Available: https://www.electronicdesign.com/technologies/dsps/article/21795646/whatsthe-deal-with-soc-verification.

[2] "VLSI design: Free online UVM training from Aldec", Eeherald.com, 2020. [Online]. Available: http://www.eeherald.com/section/news/onws2013020689.html

[3] Techopedia.com. (2019). What is a System on a Chip (SoC)? - Definition from Techopedia. [online] Available at: https://www.techopedia.com/definition/702/system-on-a-chip-soc.

[4] Khairallah, Mustafa and Ghoneima, Maged. (2014). Reusable Processor Verification Methodology Based on UVM. 10.13140/2.1.3936.9926.

[5] System Verilog Reference Guide. [offline] available at: http://svref.renerta.com.

[6] Introduction to UVM at John Aynsley YouTube channel: https://youtu.be/imH4CFmVGWE?list=PLLn6Zp_o9-jSI_HXqN9bkvE70YY4dDfub.

[7] System Verilog versus System C. [online] available at: https://www.doulos.com/content/events/VHDL_vs_SV_vs_SC.php.

[8] Aldec.com. (2019). UVM Spells Relief - Blog - Company - Aldec. [online] Available at: https://www.aldec.com/en/company/blog/110--uvm-spells-relief.

[9] Fiergolski, A. (2019). Simulation environment based on the Universal Verification Methodology.

[10] Accellera.org. (2019). [online] Available at: https://accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf .

[11] Verificationacademy.com. (2019). UVM Factory. [online] Available at: https://verificationacademy.com/verification-methodologyreference/uvm/docs_1.1a/html/files/base/uvm_factory-svh.html.

[12] Doulos.com. (2019). UVM Verification Primer. [online] Available at: https://www.doulos.com/knowhow/sysverilog/uvm/tutorial_0/.

[13] electronics.stackexchange.com. (2019). Soft core Processors VS Hard core Processors. [online] Available at: https://electronics.stackexchange.com/questions/55377/soft-coreprocessors-vs-hard-core-processors.

[14] En.wikipedia.org. (2019). Soft microprocessor. [online] Available at: https://en.wikipedia.org/wiki/Soft_microprocessor.

[15] Edaboard.com. (2019). What's the difference among soft processor and hard processor? [online] Available at: https://www.edaboard.com/showthread.php?44943-whats-thedifference-among-soft-processor-and-hard-processor.

[16] Digilentinc.com. (2019). The usefulness of Soft CPU Cores in FPGA Devices. [online] Available at: https://forum.digilentinc.com/topic/4735-the-usefullness-of-soft-cpu-coresin-fpga-devices/ .

[17] Stackoverflow.com. (2019). What is difference between soft core on NIOS and hard core? [online] Available at: https://stackoverflow.com/questions/41255732/what-isdifference-between-soft-core-on-nios-and-hard-core.

[18] Slideshare.net. (2019). Implementation of Soft-core processor on FPGA (Final Presentation). [online] Available at: https://www.slideshare.net/deepakpdks/implementation-of-softcore-processor-on-fpgafinal-presentation.

[19] Embeddedrelated.com. (2019). FPGA based processor vs. "hard" processor. [online] Available at: https://www.embeddedrelated.com/showthread/comp.arch.embedded/53193-1.php.

[20] Pulp-platform.org. (2019). User Manual – PULP platform. [online] Available at: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf.

[21] Riscv.org. (2019). KISS PULPino. Updates on PULPino. [online] Available at: https://riscv.org/wp-content/uploads/2016/12/Tue0915-RISC-V-PULPino-updateZaruba-ETH-Zurich.pdf.

[22] Pulp-platform.org. (2019). RISC-V Tutorial - PULP platform. [online] Available at: https://pulp-platform.org/docs/hipeac/pulp_intro_kgf.pdf. Sparc.org. (2019).

[23] 210.212.205.26. [Online]. Available: http://210.212.205.26/sudarshan/Main/ca/leon-doc.pdf.

[24] Opencores.org. (2020). Technical Documents (Amber 2x core manual). [online] Available at: https://opencores.org/websvn/filedetails?repname=amber&path=%2Famber%2Ftrunk%2Fdoc%2Famber-core.pdf.

[25] A. Munir, M. Magdy, S. Ahmed, S. Nasr, S. El-Ashry and A. Shalaby, "Fast Reliable Verification Methodology for RISC-V Without a Reference Model," 2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV), Austin, TX, USA, 2018, pp. 12-17, doi: 10.1109/MTV.2018.00012.

[26] Dvcon-europe.org. [Online]. Available: https://dvcon-europe.org/sites/dvcon-europe.org/files/archive/2015/proceedings/DVCon_Europe_2015_TA1_4_Paper.pdf?fbclid=IwAR3nc4Et_uKj6EDbwSz_mLcgfcOFng1Y-JxCRlbE0ID9rNiJyQdIKxxD05E.

[27] "gupta409/Processor-UVM-Verification", GitHub. [Online]. Available: https://github.com/gupta409/Processor-UVM-Verification/tree/master/Documentation?fbclid=IwAR_WIw0Kfolx7WAh2vE_fGZOp3tcmugvtes5CPS1ji6h5eXJLAK8XXTbhs.

[28] Gaisler.com. [Online]. Available: https://gaisler.com/doc/sparcv8.pdf.

[29] Technical Documents (SPARC Architecture Manual Version 8.) | SPARC International, Inc. [online] Available at: https://sparc.org/technical-documents/#V8.

[30] David Patterson and Andrew Waterman. 2017. The RISC-V Reader: An Open Architecture Atlas (1st. ed.). Strawberry Canyon.

[31] Opencores.org. [Online]. Available: https://opencores.org/websvn/filedetails?repname=amber&path=%2Famber%2Ftrunk%2Fdoc%2Famber-core.pdf.

[32] Fiergolski, A. (2019). Simulation environment based on the Universal Verification Methodology.

[33] Accellera.org. (2019). [online] Available at: https://accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf.

[34] R. Salemi, The UVM Primer: A Step-by-Step Introduction to the Universal Verification Methodology, 1st ed. Boston Light Press, 2013.

[35] M. Horn, M. Peryer, T. Fitzpatrick and J. Stickley, Universal Verification Methodology UVM Cookbook.