Assignment of master's thesis

Title:	Automatic verification methods in the SystemVerilog register access layer
Student:	Bc. Timur Ganeev
Supervisor:	Ing. Martin Kohlík, Ph.D.
Study program:	Informatics
Branch / specialization:	Design and Programming of Embedded Systems
Department:	Department of Digital Design
Validity:	until the end of summer semester 2024/2025

Instructions

- Study the constructs and components of the register access layer (RAL) of the SystemVerilog UVM library.
- Focus on RAL methods and constructs designed for automatic coverage calculations and built-in checking sequences.
- Create examples for individual RAL components with detailed comments - focus on the configuration of the RAL elements and methods from the previous point.
- Create a readme text containing a brief instructions for each example.

Electronically approved by prof. Ing. Hana Kubátová, CSc. on 6 January 2024 in Prague.

Master's thesis

AUTOMATIC VERIFICATION METHODS IN THE SYSTEMVERILOG REGISTER ACCESS LAYER

Bc. Timur Ganeev

Faculty of Information Technology
Department of Digital Design
Supervisor: Ing. Martin Kohlík, Ph.D.
May 9, 2024

Contents

List of Figures

List of Tables

List of code listings

I would like to give my thanks to my supervisor – Ing. Martin Kohlík, Ph.D. for his willingness and readiness to help me along the way of writing this thesis. Without his assistance this thesis and its practical part would not be finished.

Declaration

In Prague on May 9, 2024

Abstract

The purpose of this thesis is to study the Universal Verification Methodology (UVM) for digital circuit verification and its Register Abstraction Layer (RAL) in particular. This thesis describes the process of testbench implementation for registers and memories using UVM RAL. Next, it is explained how the user can set up automatic coverage collection and how the user can execute built-in UVM RAL sequences for checking functionality of registers and memories.

Keywords SystemVerilog, Verilog, Universal Verification Methodology, UVM, Register Abstraction Layer, RAL, Verification, Simulation, Register model, Register, Memory, RAM, ROM, Coverage, Coverage collection, Built-in sequences

Abstrakt

Tato práce se zabývá metodologií pro verifikaci digitálních integrovaných obvodů (Universal Verification Methodology – UVM), zejména její registrovou vrstvou (Register Abstraction Layer – RAL). V dané práci je popsán proces implementace prostředí pro testování registrů a pamětí s využitím komponent a metod UVM RAL. Dále je zde detailně popsáno, jak uživatel může nastavit automatickou kontrolu pokrytí a jak se dá spustit vestavěné UVM RAL sekvence pro verifikaci funkcionality testovaných registrů a pamětí.

Klíčová slova SystemVerilog, Verilog, Universal Verification Methodology, UVM, Register Abstraction Layer, RAL, Registrová vrstva, Verifikace, Simulace, Registrový model, Registr, Paměť, RAM, ROM, Pokrytí, Kontrola pokrytí, Vestavěné sekvence

List of abbreviations

UVM	Universal Verification Methodology
RAL	Register Abstraction Layer
DUT	Design/Device Under Test
RAM	Random-access Memory
ROM	Read-only Memory
TLM	Transaction Level Modeling
AMBA	Advanced Microcontroller Bus Architecture
XML	Extensible Markup Language
LSB	Least Significant Bit
MSB	Most Significant Bit
HDL	Hardware Description Language
IDE	Integrated Development Environment
NI-SIM	Digital Circuit Simulation and Verification
CTU	Czech technical university in Prague
FIT	Faculty of Information Technology
MIPS	Microprocessor without Interlocked Pipelined Stages

Introduction

In this day and age embedded systems have become a part of our normal lives and are present all around us – from computers, mobile phones and tablets to TVs, dishwashers and credit cards. Such systems are very robust and complex and can contains billions of transistors. Because of that, it is not impossible to have a bug in such system. If a bug is not discovered in time, it can cost a lot of money to deal with the consequences, since the earlier the bug is discovered – the less it will cost. Textbook example of such a thing is an Intel Pentium FDIV bug in 1994. It affected the division of certain pairs of high-precision numbers and, as a result, processor would return incorrect binary floating-point numbers. In total in costed Intel $475 million pre-tax to recover replacement and write-off these microprocessors.

Verification of such digital designs is one of the crucial tasks during design process and it cant take up to 70% of the total time spent working on a chip. The end goal of verification is a confirmation, that produced design meets all requirements and specifications. Just for that reason Universal Verification Methodology (UVM) was created with initial version releasing in 2011. It is a standardized methodology, which purpose is to verify integrated circuits. It utilizes SystemVerilog language and follows the principles of object-oriented programming. UVM was derived from the Open Verification Methodology (OVM), which is an open-source verification methodology for digital designs and systems-on-chip and was released in 2008. UVM is more flexible and reusable methodology that can be easily applied to different digital designs.

One of the crucial parts of such designs are registers and memories. Registers can be used for setting desired configurations while memories contain instructions and data. For design to behave without errors they also need to be verified. For that reason UVM contains base class library for registers and memory structures, which is called Register Abstraction Layer (RAL). It is not necessary to use UVM RAL to verify them, but without it, programmer has to take care of each register individually, which can prove to be a tedious task, since typical design can contain hundreds of them. This thesis describes how to create and configure UVM RAL register model to verify registers and memories in design under test (DUT).

Structure of this thesis is as follows: chapter 1 further describes the goals of this work. Chapter 2 contains the research regarding current existing solutions, i.e. websites, articles or theses that present the process of DUT integration into the UVM RAL register model. Chapter 3 talks about UVM in general and shows the usage of individual UVM and RAL components. Chapter 4 describes the process of DUT verification using UVM register abstraction layer. Finally, chapter 5 validates, that DUT was integrated correctly and register model is capable of catching artificially inserted bugs in the design.

Chapter 1

Specification of the assignment

This chapter further describes goals of the assignment of this thesis.

The purpose of this thesis is to showcase the process of integration of registers, RAM and ROM memories into UVM register model and their subsequent verification using UVM RAL. This includes coverage collection and built-in UVM RAL sequences for testing registers and memories.

Before that, this thesis also describes UVM and RAL themselves, so that anyone who is not familiar with it can still understand the essence of this work.

Another purpose of this work is to describe the configuration of individual UVM and RAL components to successfully integrate DUT into the register model.

Implemented testbench will be located in a GitLab repository [1].

Chapter 2
Research

This chapter describes existing solutions, i.e. websites, theses and articles, which describe the process of verification of DUT using UVM RAL.

2.1 ChipVerify

Website [2] is an extensive guide to UVM. It describes every basic component of this methodology and shows how a programmer can use it for verification of DUT of their choice. This website goes into great detail about a lot of aspects of UVM and is a great starting point for everyone, who wants to learn about it from scratch.

It also contains a section about register abstraction layer, where the author describes individual components of this layer and provides complete example of integrated registers in RAL environment.

What this website does not describe, however, is how the user can integrate memories (RAM/ROM) into the RAL. Therefore, it does not provide any information about memory operations that can be used to verify their behavior.

There is also no information about the differences between implicit and explicit prediction in register model (prediction modes are explained in detail later in this work). In the complete example author actually has implemented predictor, but does not provide any explanation as to why and when we should actually use it in our testbenches.

Another thing that this website is lacking is the process of automatic coverage collection during verification using UVM RAL. The only thing, that is related to coverage, is a general description of it using standard SystemVerilog language.

Finally, there is no information at all about built-in UVM RAL sequences.

2.2 Verification Guide

Website [3] is also another place, where one can learn more about UVM and RAL in general. In comparison with ChipVerify [2] this website contains even less information and can be a little difficult to understand.

The author briefly explains functionality of implicit and explicit predictions, but does not use any predictors in the complete example, i.e. there is no practical example as to how the user can implement implicit/explicit predictions in their testbenches.

Similarly to [2], there is no information about integrating memories in register model, automatic coverage collection and built-in UVM RAL sequences.

2.3 VLSI Verify

VLSI Verify website [4] is another place containing useful information about UVM itself and about register abstraction layer.

There are plenty of pictures and schemes describing different aspects of UVM. In comparison with previous websites, this website has a section about implicit, explicit and passive predictions. Author presents advantages and disadvantages of each prediction mode and even provides an example as to how one can turn on desired prediction in their own testbench. But author does so without going into too much detail.

What website does not describe, however, are how the user can integrate memories into register abstraction layer, how to enable automatic coverage collection and how to execute built-in RAL sequences.

2.4 Vojtěch Jílek's Master Thesis

The goal of Vojtěch Jílek's master's thesis [5] was to showcase the usage of UVM and RAL in general and to verify the functionality of a simple single cycle and pipelined processor. Another goal of that thesis was to describe common mistakes and problems that a novice developer may encounter while using register abstraction layer.

Vojtěch Jílek was able to accomplish initial goal with his thesis. It contains useful information about UVM and register layer and can be used to quickly get to know all its general components and functions.

The code of the thesis [6] contains testbenches for single cycle and pipelined processors and utilizes a wide variety of constructions, that are supported by UVM to make the verification process as simple as possible. The only problem is that the code barely contains any comments and it can be quite difficult to comprehend what each individual component or function is supposed to do.

Another thing that this thesis is lacking in is the usage of RAL automatic coverage collection and built-in register abstraction layer sequences. The author also does not describe the differences between implicit and explicit predictions.

Chapter 3
Analysis

The purpose of this chapter is to describe basic concepts of UVM and its register abstraction layer.

3.1 Universal Verification Methodology

As it was already stated in the introduction, UVM is used to verify digital designs. It was created with a purpose of standardizing the structures of verification programs and testing methodologies. With UVM the user can create verification environment for complex designs with the help of built-in classes and methods.

Main features of UVM are:

Testbench Components UVM contains a set of classes, that can be used in verification process. They can be extended and modified as the user sees fit.

Transactions Communication between the DUT and testbench happens with a help of the so-called transactions. The user can extend the existing transaction class, which is used to transfer information between DUT and testbench.

Communication between individual components inside a testbench happens on a Transaction Level Modeling (TLM) level.

Messaging and Reporting With a help of UVM the user can printout relevant information about the simulation runs, such as warnings and errors, which can be used to further improve verification environment and for debugging.

Configuration With a help of configuration database the user can store and retrieve appropriate information for testbench components.

Functional Coverage Functional coverage is a process, that evaluates how thoroughly were the functionalities of the design tested. It is a user-defined metric, and, with a help of UVM, the user can ensure, that the DUT was exhaustively verified.

Register Abstraction Layer RAL simplifies the process of verification of DUT registers and memories. For more in-depth information about RAL refer to section 3.2.

Basic information about UVM is taken from [7].

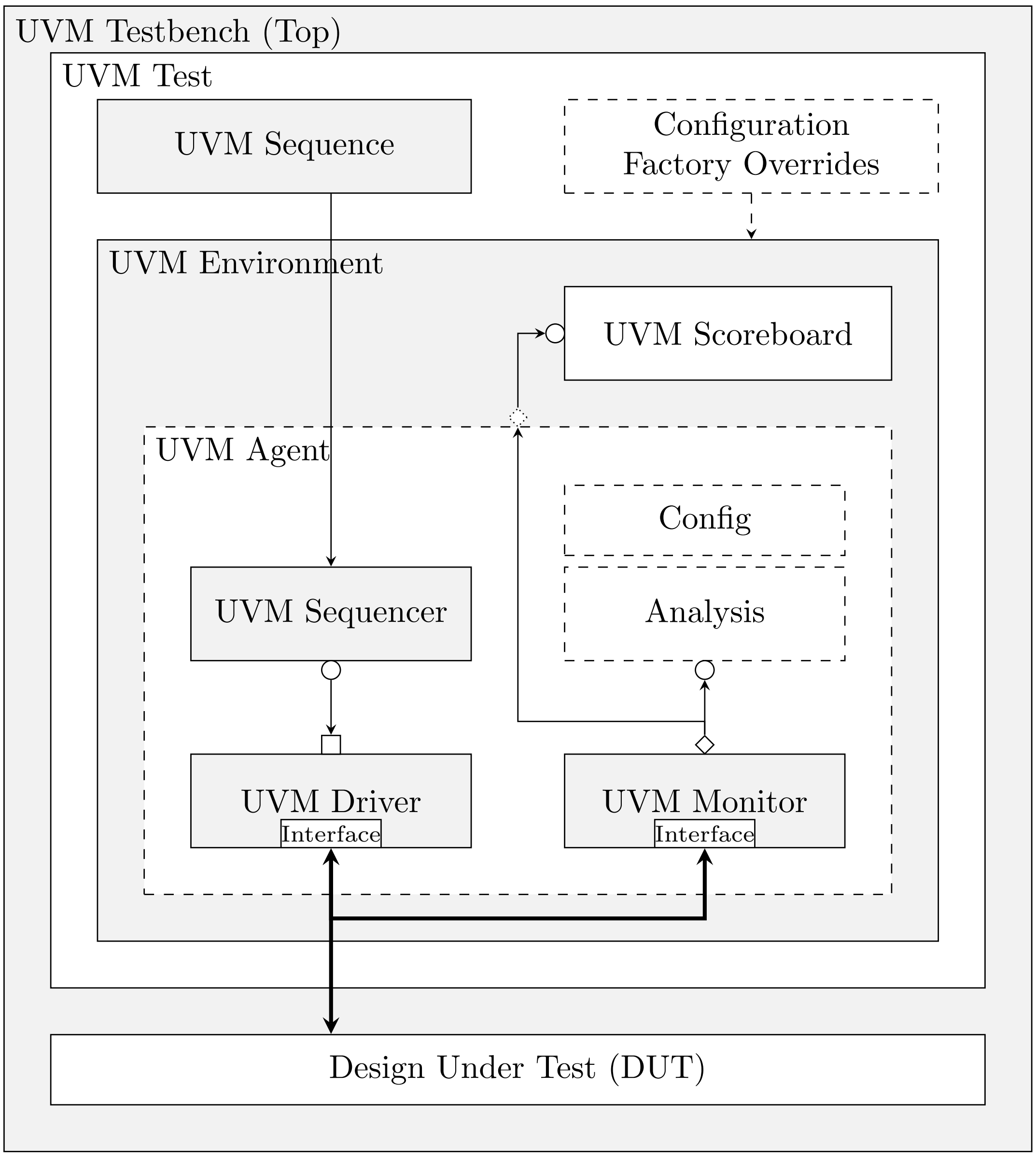


Figure 3.1 Basic structure of UVM testbench. Source: [8, s. 4]

3.1.1 UVM Testbench Components

Here we will describe every major component of UVM. Basic structure of a typical testbench, that utilizes UVM, is shown in figure 3.1.

UVM Testbench (Top)

Testbench top is used to statically store the instance of DUT and corresponding interfaces. Interfaces are used for DUT and UVM test to communicate with each other. Testbench top is also responsible for generating clock and reset signals, and for choosing a test that will be executed on a DUT block [9].

UVM Test

Before the simulation starts, UVM test dynamically generates and configures verification environment (UVM Environment). UVM test is also responsible for connecting this environment to DUT using interfaces and it is achieved with a help of configuration database. UVM test can start the simulation itself by executing UVM sequence on UVM sequencers [8, s. 5].

UVM Environment

UVM environment is a container for multiple, reusable verification components, such as agents, drivers, monitors, scoreboards and other environments. These components communicate with each other using TLM ports [8, s. 5].

UVM Sequence Item

UVM sequence item is a data packet, that encapsulates all relevant signals that should be sent to DUT in order to verify its desired functionality. For example, sequence item that would be used to send some data to some register can contain write enable bit, register address and data.

Having these signals in a structure allows user to perform useful operation on them, such as randomizing, modifying, printing out, copying etc.

UVM Sequence

UVM sequence contains data items (`uvm_sequence_item`), which are later sent to the DUT with a help of a driver via sequencer. Sequence can randomize and send these sequence items in a specified order, therefore we can create interesting scenarios for DUT to process. Sequence is not a part of the UVM environment and can be connected to only one UVM Sequencer [8, s. 6].

UVM Sequencer

UVM sequencer is a mediator between sequence and driver. It chooses what data items coming from a sequence should be passed to a driver so that they can be later sent to a DUT. Sequencer is also able to receive response data items from a DUT via driver. Single sequencer can have multiple sequences connected to it [8, s. 6].

UVM Driver

UVM driver communicates with a DUT. It retrieves randomized `uvm_sequence_item` from a sequencer and converts it to a logical level signals for interface. Corresponding interface handle can be acquired from configuration database. Driver class contains REQ (request) and RSP (response) parameterized types, which are of type `uvm_sequence_item` by default. Request is a sequence item, that will be sent to DUT via interface, whereas response is a sequence item, that contains data sent from the DUT to the testbench [10].

UVM Monitor

UVM Monitor is used to observe signal activity from the interface (logic level) and converts it into transaction level objects (`uvm_sequence_item`) that can be sent to other verification components via TLM analysis port. Monitor can be configured for basic protocol checking and coverage collection [11].

UVM Agent

UVM agent encapsulates sequencer, driver and monitor into a single block. Agent is responsible for their instantiation and for establishing connection between them. Agent can be configured for basic protocol checking and for coverage collection. Agent can be active or passive [12]:

Active agent instantiates all three components (sequencer, driver and monitor). Data can be driven to DUT.

Passive agent only instantiates a monitor. Data can not be driven to DUT. This way agent can only observe signals from the interface and send them to other components. Useful when we do not need to send data to DUT.

UVM Scoreboard

UVM scoreboard is a verification component that checks functionality of the DUT. It receives data items from monitor through TLM port and compares it with a reference (golden) DUT model [13].

3.1.2 UVM Phases

In UVM based SystemVerilog testbench class object can be created at any time during simulation. That means that we need to somehow check if object we are about to call is already created, i.e. we need some kind of synchronization mechanism. For that reason UVM introduces the concept of so-called phases.

Each verification component is derived from `uvm_component` class and hence supports the usage of phases. Phases are implemented as virtual callback functions/methods and can be defined by the user. Each individual component can not proceed to the next phase, unless all other components finish their operations in the current phase. That way proper synchronization between all components is guaranteed.

Order of execution of these phases is depicted in figure 3.2. Each individual phase is described in detail in the following sections.

All information regarding UVM phases is taken from [15], [14], [16] and [17].

3.1.2.1 build_phase

Build phase is responsible for creating and configuring testbench structure. It calls `build_phase()` of every `uvm_component` that is currently present in a testbench in a top-down order. This phase does not consume any simulation time and therefore is implemented as a function.

Typical uses of `build_phase()` include instantiation of sub-components, instantiation of register model, getting configuration values for the components being built and setting configuration values for sub-components.

Phase ends when all `uvm_component` components have been instantiated.

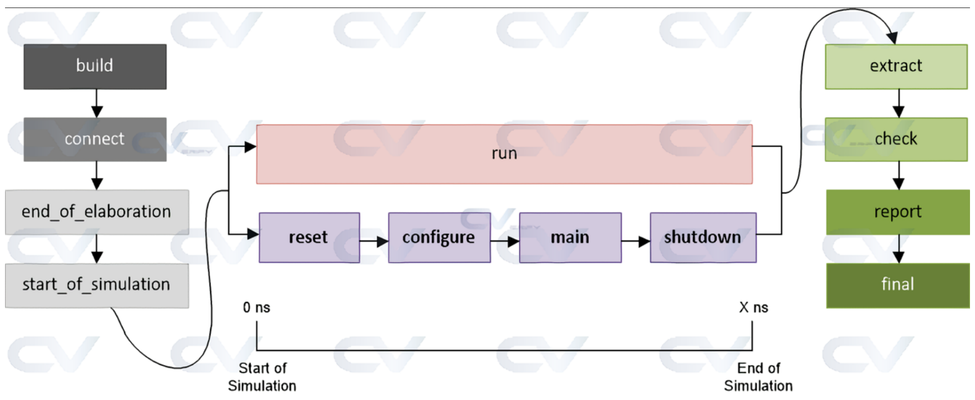


Figure 3.2 Universal Verification Methodology Phases. Source: [14]

3.1.2.2 connect_phase

Connect phase is used to connect all verification components with each other. `connect_phase()` of every `uvm_component` that is currently present in a testbench is called in a bottom-up order. This phase does not consume any simulation time and therefore is implemented as a function.

Typical uses of `connect_phase` include connecting TLM ports and connecting register model to adapter components.

Phase ends when the connection between all `uvm_components` has been established.

3.1.2.3 end_of_elaboration_phase

End of elaboration phase is responsible for potential parametrization of verification components. It calls `end_of_elaboration_phase()` of every `uvm_component` that is currently present in a testbench in a bottom-up order. This phase does not consume any simulation time and therefore is implemented as a function.

Typical uses of `end_of_elaboration_phase` include displaying environment topology, opening files and defining additional configuration settings for components.

Phase ends when all defined operations in it have been executed.

3.1.2.4 start_of_simulation_phase

Start of simulation phase is used to print out additional information about verification topology. It calls `start_of_simulation_phase()` of every `uvm_component` that is currently present in a testbench in a bottom-up order. This phase does not consume any simulation time and therefore is implemented as a function.

Typical uses of `start_of_simulation_phase` include displaying environment topology, setting debugger breakpoints and setting initial run-time configuration values.

Phase ends when all defined operations in it have been executed.

3.1.2.5 run_phase

Run phase starts the simulation of DUT. It calls `run_phase()` of every `uvm_component` that is currently present in a testbench. Run phase runs in parallel to the other runtime phases, as shown in figure 3.3. All components are synchronized with the respect to the current run phase. This phase (i.e. the simulation itself) consumes actual simulation time and therefore is implemented as a task.

Other runtime phases (apart from `run_phase`) are:

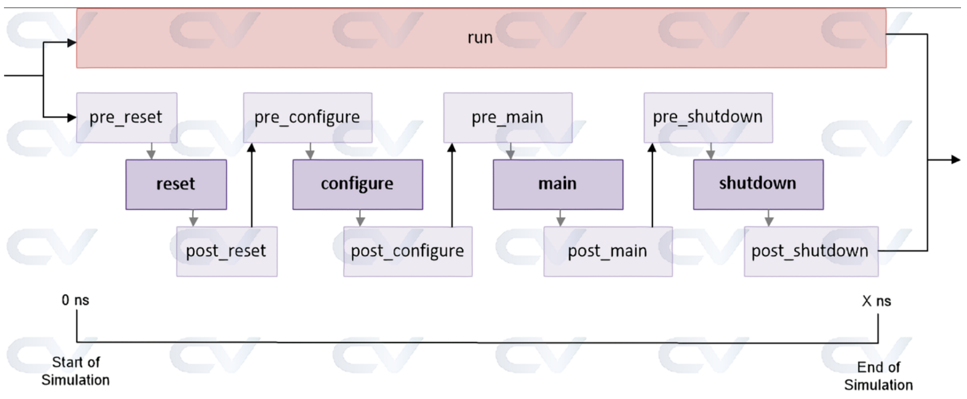


Figure 3.3 Universal Verification Methodology Run Phases. Source: [14]

reset_phase This phase is responsible for generating reset and clock signals for verification components that are connected to an interface.

configure_phase This phase is used to configure DUT and its memories to be compatible with the verification environment.

main_phase This phase is used for primary test stimulus, i.e. its purpose is to properly start stimulus sequences.

shutdown_phase This phase waits for all data to be drained out of the DUT after simulation ends. Buffered data from the DUT is extracted with a help of read/write operations or sequences.

Phases with a name `pre_*` or `post_*`, where `*` is a name of a runtime phase, are used to further fine-tune corresponding runtime phase.

Phase `run_phase` ends when two conditions are satisfied:

1. There is no need for DUT to be simulated further.
2. `post_shutdown_phase` is ready to end.

In that case `run_phase` terminates in one of two ways:

1. Every verification component can raise and drop so-called “objections” on the specified phase. Raised objection indicates, that this component is not ready to end and is still executing some operations. If all objections on `run_phase` are dropped – phase ends.
2. Even if there are raised objections on `run_phase`, it can still end if the timeout expires. By default, the timeout is set to 9200 seconds, but can be overridden with `set_timeout()` method.

3.1.2.6 extract_phase

This phase is used to extract data from different points of the verification environment. It calls `extract_phase()` of every `uvm_component` that is currently present in a testbench in a bottom-up order. This phase does not consume any simulation time and therefore is implemented as a function.

Typical uses of `extract_phase` include extracting remaining data and final state information from verification components, computing and displaying statistics and closing files.

Phase ends when all desired information has been collected and summarized.

3.1.2.7 check_phase

Check phase is used for checking for any unexpected events in the verification environment. It calls `check_phase()` of every `uvm_component` that is currently present in a testbench in a bottom-up order. This phase does not consume any simulation time and therefore is implemented as a function.

Main purpose of this phase is to check if there is no unaccounted-for data remains in the testbench.

Phase ends when current test is known to have passed or failed.

3.1.2.8 report_phase

Report phase is responsible for reporting results of the test. It calls `report_phase()` of every `uvm_component` that is currently present in a testbench in a bottom-up order. This phase does not consume any simulation time and therefore is implemented as a function.

Phase ends when the current test ends.

3.1.2.9 final_phase

Final phase is used for clearing up various things that are related to current simulation run. It calls `final_phase()` of every `uvm_component` that is currently present in a testbench in a top-down order. This phase does not consume any simulation time and therefore is implemented as a function.

Phase ends when the simulator tool is ready to exit.

3.2 UVM – Register Abstraction Layer

Registers are indispensable parts of every hardware design. They can be used to store configuration data and can be written to with a help of software. Such software can use some kind of bus protocol, for example AMBA (Advanced Microcontroller Bus Architecture), to set registers to certain values.

Verifying registers can be tedious, as the user would need to manually create sequence items with desired data, create `uvm_sequence` and execute this sequence on an appropriate sequencer. For that reason UVM contains so-called Register Abstraction Layer (RAL).

This register abstraction layer not only makes the process of register verification easier, but is also capable of verifying various memory blocks.

UVM RAL is not necessary in order to verify functionality of the registers and memories, but it contains useful abstract classes and methods, which make verification engineer's job much easier.

All information about RAL is taken from [2] and [18].

3.2.1 Frontdoor and Backdoor

Reading and writing registers and memory words through RAL can be done via so-called frontdoor or backdoor.

Communication between register model and DUT via frontdoor means converting register model transaction to a protocol specific bus transaction and driving this transaction into the DUT through a physical interface. In other words, it is the same as when some component in the actual digital design is communicating with registers or memories in the same design. It means that all logic surrounding such communication is active and clock cycles (i.e. simulation time) are consumed.

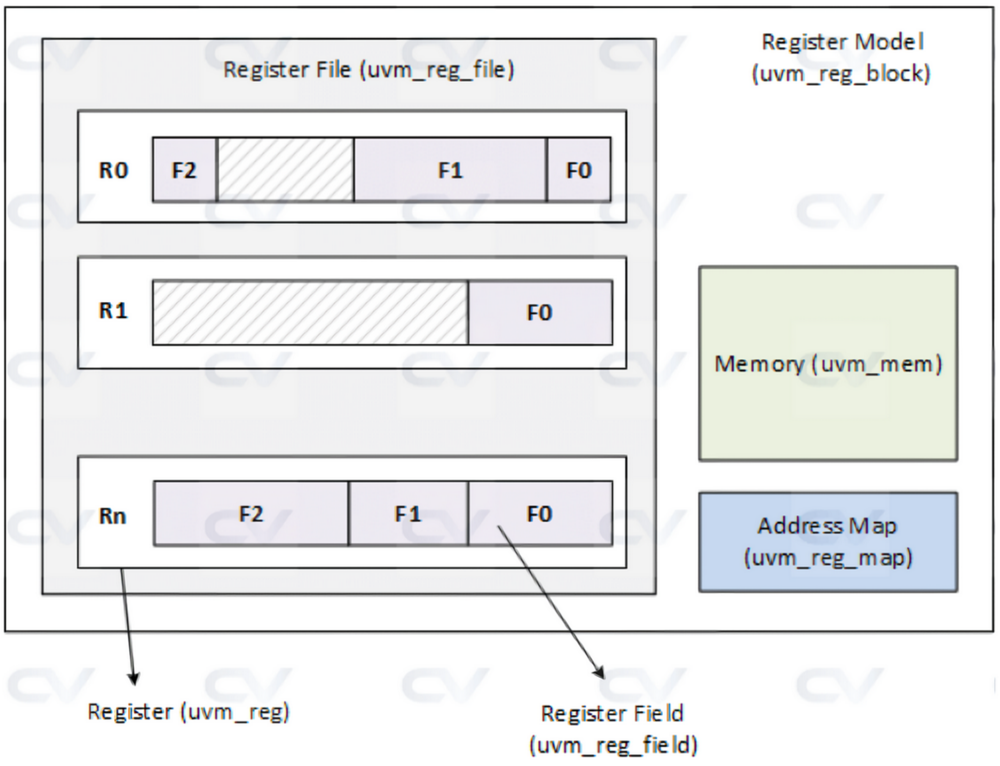


Figure 3.4 UVM Register Abstraction Layer – Register Model Structure. Source: [18]

Communication via backdoor is done via simulator database. The value of signal is written or read based on its name in the design. Transactions that use backdoor to access registers and memories are executed instantaneously and do not consume any clock cycles (i.e. simulation time). The drawback of backdoor access is the absence of a control logic, that is typically active during "normal" operations involving bus communication.

In order to utilize backdoor access the user must specify the hardware description language (HDL) path for a desired register or memory.

Information about frontdoor and backdoor is taken from [19].

3.2.2 UVM RAL – Register Model Structure

"Register Model" is the main block of the register abstraction layer. It contains everything that is needed to successfully create testbench that would be capable of verifying registers and memories within a digital design.

Since typical digital design can consist of hundreds of registers and memories, the task of manually creating a register model can be time-consuming. That is why it is a common practice to use some kind of script or software, that takes the register specification as an input, for example in IP-XACT XML format, and generates corresponding register model in SystemVerilog language using RAL base classes [18].

Typical structure of a UVM RAL register model is depicted in figure 3.4. In the following sections each component of the register model will be described in detail.

3.2.2.1 uvm_reg

This base class mimics the actual register in the design. User can extend `uvm_reg` class to create and configure their own register, so that it would correspond (functionality wise) to the register in the design. The user can `read()` from or `write()` the specified value in the register.

Each UVM RAL register contains so-called `desired` and `mirrored` values:

Desired value is the value, that we want register to have. We can set and get this value using `set()` and `get()` methods. We can also `update()` the corresponding register in the DUT to have this desired value.

Mirrored value is the value, that should be in the actual register in the DUT. If implicit (auto) prediction is turned on, each time a `read()` or `write()` transaction occurs, corresponding register is updated with a value from the DUT. Mirrored values can be modified manually with `predict()` method. Method called `mirror()` updates, and optionally compares, the current mirrored value in the register model with the actual register value from the DUT. Current mirrored value can be read using `get_mirrored_value()` method.

More about register model prediction modes is described in section 3.2.3.

Registers can have multiple "fields", i.e. blocks of continuous bits. Each field represents a certain feature in the design and every field can have different access policies. For example one 32 bits register can have first 16 bits (`[31:16]`) with read-write access, while the remaining 16 bits (`[15:0]`) can be read-only. In the RAL fields are represented by `uvm_reg_field` base class.

Relevant `uvm_reg` methods are described in detail below.

Code listing 3.1 uvm_reg::new()

```
function new ( string name = "",
               int unsigned n_bits,
               int has_coverage )
```

Creates a new instance of RAL register with a specified name, total number of bits and functional coverage models.

Code listing 3.2 uvm_reg::configure()

```
function void configure ( uvm_reg_block blk_parent,
                          uvm_reg_file regfile_parent = null,
                          string hdl_path = "" )
```

Configures the existing RAL register. Specify the parent register block of this register, parent register file (optional) and register's HDL path.

Code listing 3.3 uvm_reg::needs_update()

```
virtual function bit needs_update()
```

Returns '1' if `desired` and `mirrored` values of the fields in this registers are not equal.

Code listing 3.4 uvm_reg::write()

```
virtual task write ( output uvm_status_e status,
                     input uvm_reg_data_t value,
                     input uvm_path_e path = UVM_DEFAULT_PATH,
                     input uvm_reg_map map = null,
                     input uvm_sequence_base parent = null,
                     input int prior = -1,
                     input uvm_object extension = null,
                     input string fname = "",
                     input int lineno = 0 )
```

Writes the specified **value** in this register using specified **path**. Path can have values UVM_FRONTDOOR or UVM_BACKDOOR. Both of these paths respect access policies of the registers, i.e. writing to a read-only register will not change the register value.

■ **Code listing 3.5** uvm_reg::read()

```
virtual task read ( output uvm_status_e status ,
                    input uvm_reg_data_t value ,
                    input uvm_path_e path = UVM_DEFAULT_PATH ,
                    input uvm_reg_map map = null ,
                    input uvm_sequence_base parent = null ,
                    input int prior = -1 ,
                    input uvm_object extension = null ,
                    input string fname = "" ,
                    input int lineno = 0 )
```

Reads the current register and stores its value in **value** variable using specified **path**. Path can have values UVM_FRONTDOOR or UVM_BACKDOOR. Both of these paths respect access policies of the registers, i.e. reading read-clear register will set this register value to zeroes.

■ **Code listing 3.6** uvm_reg::poke()

```
virtual task poke ( output uvm_status_e status ,
                    input uvm_reg_data_t value ,
                    input string kind = "" ,
                    input uvm_sequence_base parent = null ,
                    input uvm_object extension = null ,
                    input string fname = "" ,
                    input int lineno = 0 )
```

Deposits the specified **value** into this register using its HDL path (backdoor access). Poke method does not respect access policies of the registers, i.e. read-only registers can be written to.

■ **Code listing 3.7** uvm_reg::peek()

```
virtual task peek ( output uvm_status_e status ,
                    input uvm_reg_data_t value ,
                    input string kind = "" ,
                    input uvm_sequence_base parent = null ,
                    input uvm_object extension = null ,
                    input string fname = "" ,
                    input int lineno = 0 )
```

Reads current register and stores its value in **value** variable using its HDL path (backdoor access). Peek method does not respect access policies of the registers, i.e. read-clear register will not change its value after **peek()** method is executed.

■ **Code listing 3.8** uvm_reg::update()

```
virtual task update ( output uvm_status_e status ,
                      input uvm_path_e path = UVM_DEFAULT_PATH ,
                      input uvm_reg_map map = null ,
                      input uvm_sequence_base parent = null ,
                      input int prior =  -1 ,
                      input uvm_object extension = null ,
                      input string fname = "" ,
                      input int lineno = 0 )
```

Updates the DUT register value with its **desired** RAL value. It can be done using frontdoor (**write()**) or backdoor (**poke()**) access.

■ **Code listing 3.9** uvm_reg::mirror()

```
virtual task mirror ( output uvm_status_e status,
                      input uvm_check_e check = UVM_NO_CHECK,
                      input uvm_path_e path = UVM_DEFAULT_PATH,
                      input uvm_reg_map map = null,
                      input uvm_sequence_base parent = null,
                      input int prior = -1,
                      input uvm_object extension = null,
                      input string fname = "",
                      input int lineno = 0 )
```

Updates and optionally compares register's `mirrored` value with an actual value from the DUT. Using arguments `UVM_NO_CHECK` and `UVM_CHECK` current mirrored value can be compared with an actual DUT register value and an error message can be displayed in case of these values not being equal. This method can be executed with frontdoor (`write()`) or backdoor (`poke()`) accesses.

■ **Code listing 3.10** uvm_reg::predict()

```
virtual function bit predict ( uvm_reg_data_t value,
                               uvm_reg_byte_en_t be = -1,
                               uvm_predict_e kind = UVM_PREDICT_DIRECT,
                               uvm_path_e path = UVM_FRONTDOOR,
                               uvm_reg_map map = null,
                               string fname = "",
                               int lineno = 0 )
```

Manually changes the `mirrored` value of this register. Returns `TRUE` if the prediction was successful for every field in the register.

All information about `uvm_reg` class is taken from [18] and [20].

3.2.2.2 uvm_reg_field

This class represents individual bits in a register. They can be configured to have specified size, least significant bit (LSB), reset value and access policy using `configure()` method. RAL supports wide variety of access policies as shown in the table 3.1 (information about them is taken from [21]).

Relevant `uvm_reg_field` methods are described in detail below.

■ **Code listing 3.11** uvm_reg_field::new()

```
function new ( string name = "uvm_reg_field" )
```

Creates a new field instance. This method should not be called directly, instead,
`uvm_reg_field::type_id::create()`
factory method should be invoked.

■ **Code listing 3.12** uvm_reg_field::configure()

```
function void configure ( uvm_reg parent,
                          int unsigned size,
                          int unsigned lsb_pos,
                          string access,
                          bit volatile,
                          uvm_reg_data_t reset,
                          bit has_reset,
                          bit is_rand,
                          bit individually_accessible )
```

Table 3.1 Register field access policies supported by UVM RAL

Access policy	Description
RO (Read-only)	Writing has no effect, reading returns current register value
RW (Read-write)	Writing updates current register value, reading returns current register value
RC (Read-clear)	Writing has no effect, reading clears all register bits
RS (Read-set)	Writing has no effect, reading sets all register bits
WRC (Write-read-clear)	Writing updates current register value, reading clears all register bits
WRS (Write-read-set)	Writing updates current register value, reading sets all register bits
WC (Write-clear)	Writing clears all register bits, reading returns current register value
WS (Write-set)	Writing sets all register bits, reading returns current register value
WSRC (Write-set-read-clear)	Writing sets all register bits, reading clears all register bits
WCRS (Write-clear-read-set)	Writing clears all register bits, reading sets all register bits
W1C (Write-1-clear)	Writing ’1’ clears matching bit, reading returns current register value
W1S (Write-1-set)	Writing ’1’ sets matching bit, reading returns current register value
W1T (Write-1-toggle)	Writing ’1’ toggles matching bit, reading returns current register value
W0C (Write-0-clear)	Writing ’0’ clears matching bit, reading returns current register value
W0S (Write-0-set)	Writing ’0’ sets matching bit, reading returns current register value
W0T (Write-0-toggle)	Writing ’0’ toggles matching bit, reading returns current register value
W1SRC (Write-1-set-read-clear)	Writing ’1’ sets matching bit, reading clears all register bits
W1CRS (Write-1-clear-read-set)	Writing ’1’ clears matching bit, reading sets all register bits
W0SRC (Write-0-set-read-clear)	Writing ’0’ sets matching bit, reading clears all register bits
W0CRS (Write-0-clear-read-set)	Writing ’0’ clears matching bit, reading sets all register bits
WO (Write-only)	Writing updates current register value, reading results in error
WOC (Write-only-clear)	Writing clears all register bits, reading results in error
WOS (Write-only-set)	Writing sets all register bits, reading results in error
W1 (Write-once)	First write operation updates current register value, other write operations have no effect, reading returns current register value
WO1 (Write-only-once)	First write operation updates current register value, other write operations have no effect, reading results in error

Configures the instance of current register field. Specify the `parent` register of this field, field `size` in bits, its position of the LSB relative to the LSB of the register, field's access policy, its `reset` value, whether the field is actually reset, whether the field may be randomized and whether the field is the only one to occupy a byte lane.

Other relevant methods have the same structure as the `uvm_reg` ones.

3.2.2.3 uvm_reg_file

This class encapsulates a collection of registers and other register files.

Relevant methods include:

Code listing 3.13 uvm_reg_file::new()

```
function new ( string name = "" )
```

Creates a new instance of a register file with a specified name.

Code listing 3.14 uvm_reg_file::configure()

```
function void configure ( uvm_reg_block blk_parent ,
                          uvm_reg_file regfile_parent ,
                          string hdl_path = "" )
```

Configures this register file. Specify the parent register block and parent register file. If current register file is instantiated in a register block, `regfile_parent` should be set to `NULL`. If current register file is instantiated in another register file, `blk_parent` should be set to that register file's parent block and `regfile_parent` should be set to that register file. We can also specify the register file's HDL path [22].

3.2.2.4 uvm_mem

This class represents memory block in the actual design. It can be extended and configured to have specified memory depth, width and access policy. Supported access policies are read-write for RAM and read-only for ROM memories.

Just like with registers, RAL memories can be read and written with `read()`/`peek()` and `write()`/`poke()` methods. These methods are structurally the same as in the `uvm_reg` base class with the exception of `offset` parameter, which can be used to read/write data from/to specified offset in the memory.

Unlike registers, however, memories do not have `desired` and `mirrored` values, since memories are usually much larger than any register file and duplicating all memory words will be counterproductive.

In case of memories, we are able to use `burst_read()` and `burst_write()` methods. With these methods we can read and write multiple memory words at once, without the need to execute `read()` and `write()` on every individual word.

All relevant methods are described below.

Code listing 3.15 uvm_mem::new()

```
function new ( string name ,
               longint unsigned size ,
               int unsigned n_bits ,
               string access = "RW",
               int has_coverage = UVM_NO_COVERAGE )
```

Creates new RAL memory with a specified name. Parameter `size` specifies the number of memory locations (memory depth), while `n_bits` is a number of bits in every memory location (memory width). Access policy and functional coverage models can also be specified.

Code listing 3.16 uvm_mem::configure()

```
function void configure ( uvm_reg_block parent,
                          string hdl_path = "" )
```

Configures current instance of RAL memory with specified parent register block and HDL path.

Code listing 3.17 uvm_mem::burst_write()

```
virtual task burst_write ( output uvm_status_e status,
                           input uvm_reg_addr_t offset,
                           input uvm_reg_data_t value[],
                           input uvm_path_e path = UVM_DEFAULT_PATH,
                           input uvm_reg_map map = null,
                           input uvm_sequence_base parent = null,
                           input int prior = -1,
                           input uvm_object extension = null,
                           input string fname = "",
                           input int lineno = 0 )
```

Writes the values from the `value` dynamic array into the memory starting at the specified `offset`. Frontdoor (`UVM_FRONTDOOR`) or backdoor (`UVM_BACKDOOR`) accesses can be used. This method respects the access policy of a memory, i.e. values inside of a ROM memory will not be overwritten.

Code listing 3.18 uvm_mem::burst_read()

```
virtual task burst_read ( output uvm_status_e status,
                          input uvm_reg_addr_t offset,
                          output uvm_reg_data_t value[],
                          input uvm_path_e path = UVM_DEFAULT_PATH,
                          input uvm_reg_map map = null,
                          input uvm_sequence_base parent = null,
                          input int prior = -1,
                          input uvm_object extension = null,
                          input string fname = "",
                          input int lineno = 0 )
```

Reads and stores data from the memory starting at the specified offset into the `value[]` dynamic array. The number of read memory locations is defined by the size of `value[]` array.

Frontdoor (`UVM_FRONTDOOR`) or backdoor (`UVM_BACKDOOR`) accesses can be used. This method respects the access policy of a memory.

All information about `uvm_mem` base class is taken from [23].

3.2.2.5 uvm_reg_map

Every register and memory block is mapped to specific addresses in the register block. This `uvm_reg_map` base class represents an address map, i.e. a collection of registers and memories that are accessible via specific physical interface. Register block can have multiple address maps, each corresponding to a physical interface that is connected to a DUT.

Registers and memories can be added to an address map with `add_reg()` and `add_mem()` methods respectively. By using these methods we can specify register/memory access policy and offset in current address map. Valid policies are read-write, read-only and write-only. These policies refer to the registers' or memories' accessibility via this address map, not the accessibility of registers and memories themselves.

For example, register can have read-write policy, while not begin accessible for writing via this specific register map. That same register can be mapped to another address map with read-write accessibility and in that case it can be successfully written to.

Address map of another register sub-block can be added to current register block at a specified offset with a `add_submap()` method.

Before executing any read and write transaction on registers and memories a sequencer and an adapter must be specified for current address map. It can be achieved using `set_sequencer()` method.

All relevant `uvmr_reg_map` methods are described below.

Code listing 3.19 uvm_reg_map::new()

```
function new ( string name = "uvm_reg_map" )
```

Creates a new instance of register address map with a specified name.

Code listing 3.20 uvm_reg_map::configure()

```
function void configure ( uvm_reg_block parent,
                          uvm_reg_addr_t base_addr,
                          int unsigned n_bytes,
                          uvm_endianness_e endian,
                          bit byte_addressing = 1 )
```

Configures the current address map of `parent` register block. Every register, memory or sub-block will be offset relative to the base address (`base_addr`) of this address map. Byte width of the bus, on which this map is used, can be specified with `n_bytes` parameter. Parameter `byte_addressing` specifies whether or not consecutive memory addresses refer to the data that is one byte apart.

Code listing 3.21 uvm_reg_map::add_reg()

```
virtual function void add_reg ( uvm_reg rg,
                                uvm_reg_addr_t offset,
                                string rights = "RW",
                                bit unmapped = 0,
                                uvm_reg_frontdoor frontdoor = null )
```

Adds the `rg` register to this address map at the specified `offset` relative to the base address of the map. Register's accessibility is defined by the `rights` parameter. Valid options are "RW", "RO" and "WO". A register can only be added to an address map whose parent block is the same as the register's parent block.

Code listing 3.22 uvm_reg_map::add_mem()

```
virtual function void add_mem ( uvm_mem mem,
                                uvm_reg_addr_t offset,
                                string rights = "RW",
                                bit unmapped = 0,
                                uvm_reg_frontdoor frontdoor = null )
```

Adds the `mem` memory to this address map at the specified `offset` relative to the base address of the map. Memory's accessibility is defined by the `rights` parameter. Valid options are "RW", "RO" and "WO". A memory can only be added to an address map whose parent block is the same as the memory's parent block.

Code listing 3.23 uvm_reg_map::add_submap()

```
virtual function void add_submap ( uvm_reg_map child_map,
                                   uvm_reg_addr_t offset )
```

Adds the `child_map` address sub-map to this address map at the specified `offset` relative to the base address of this map. An address sub-map can only be added to an address map in the grandparent register block of the address sub-map.

Code listing 3.24 uvm_reg_map::set_sequencer()

```
virtual function void set_sequencer ( uvm_sequencer_base sequencer,
                                      uvm_reg_adapter adapter = null )
```

Sets the sequencer and adapter for this register address map. Before executing any methods that communicate with the DUT registers and memories via this address map, `set_sequencer()` method for this map must be called.

Code listing 3.25 uvm_reg_map::set_base_addr()

```
virtual function void set_base_addr ( uvm_reg_addr_t offset )
```

Sets the base address for this this map.

Code listing 3.26 uvm_reg_map::set_auto_predict()

```
function void set_auto_predict ( bit on = 1 )
```

Enables the implicit (auto) prediction for this map. When implicit prediction is turned on, register's `mirorred` value is updated whenever any read or write operation is executed on the corresponding register in the register model. It is done with via `uvm_reg::predict()` method.

If implicit prediction is turned off, `mirrored` values of the registers are not updated after any read or write operations.

Code listing 3.27 uvm_reg_map::get_auto_predict()

```
function bit get_auto_predict ()
```

Returns '1' if implicit (auto) prediction mode is turned on for this map. Returns '0' otherwise.

All information about `uvm_reg_map` base class is taken from [24].

3.2.2.6 uvm_reg_adapter

Register model communicates with DUT by executing various RAL methods, such as

`uvm_reg::read() and uvm_reg::write()`

or

`uvm_mem::read()/burst_read() and uvm_mem::write()/burst_write()`

During the execution of these methods register model components communicate with each other using generic structure of type `uvm_reg_bus_op`:

Code listing 3.28 Structure uvm_reg_bus_op

```
typedef struct {
  uvm_access_e kind;
  uvm_reg_addr_t addr;
  uvm_reg_data_t data;
  int n_bits;
  uvm_reg_byte_en_t byte_en;
  uvm_status_e status;
} uvm_reg_bus_op;
```

This structure consists of

kind Kind of current transaction – read or write.

addr Target address in address map.

data Data to read or write.

n_bits Number of bits being transferred.

byte_en Enables for the byte lanes on the bus. Meaningful only when the bus supports byte enables and the operation originates from a field write/read, i.e. when the width of the bus is shorter than the width of a register/memory word.

status Result of the transaction.

In order for driver to understand, what register or memory word should be written to or read from, we need to convert this register model transaction to a protocol specific bus transaction. This can be achieved using `uvm_reg_adapter` base class.

In order for register model to successfully communicate with DUT, the user should extend adapter base class and implement two methods:

Code listing 3.29 uvm_reg_adapter::reg2bus()

```
pure virtual function uvm_sequence_item reg2bus (
  const ref uvm_reg_bus_op rw
)
```

and

Code listing 3.30 uvm_reg_adapter::bus2reg()

```
pure virtual function void bus2reg (
  uvm_sequence_item bus_item,
  ref uvm_reg_bus_op rw
)
```

Method `reg2bus()` is responsible for converting `uvm_reg_bus_op` transaction to sequence item. Inside it the user must create new bus-specific sequence item, set its members to corresponding values based on a `rw` register model transaction, and return it.

Method `bus2reg()` does the opposite. The user must set the members of the `rw` register transaction to the corresponding values based on a `bus_item` bus transaction. Unlike in the `reg2bus()`, bus transaction is already created.

All information about `uvm_reg_adapter` base class is taken from [25].

3.2.2.7 uvm_reg_block

Register block can contain registers, register files, memories, address maps and other register blocks, and is represented by `uvm_reg_block` base class.

Address map can be created in this block using `create_map()` method. With this method we can also specify base address for this map. All registers, memories and sub-blocks in current map will be offset with regard to this base address.

Before executing any read and write transaction on registers and memories, top-level register block should be locked. It can be done with a help of `lock_model()` method. This method recursively locks an entire register block and builds its address map. After this, no more registers and memories can be added to current register block. Once the block is locked, it can not be unlocked.

Relevant `uvm_reg_block` methods are described in detail below.

Code listing 3.31 uvm_reg_block::new()

```
function new (
  string name = "",
  int has_coverage = UVM_NO_COVERAGE
)
```

Creates an instance of a new register block with a specified name and functional coverage models.

Code listing 3.32 uvm_reg_block::configure()

```
function void configure (
  uvm_reg_block parent = null,
  string hdl_path = ""
)
```

Configure current register block's parent block and HDL path.

Code listing 3.33 uvm_reg_block::create_map()

```
virtual function uvm_reg_map create_map (
  string name,
  uvm_reg_addr_t base_addr,
  int unsigned n_bytes,
  uvm_endianness_e endian,
  bit byte_addressing = 1
)
```

Creates and returns a new address map for this register block. Base address of the map, width of the bus (that will be associated with this map) in bytes, endianness and byte addressing can be configured. Every register block already has instantiated `default_map` of type `uvm_reg_map`.

Code listing 3.34 uvm_reg_block::lock_model()

```
virtual function void lock_model ()
```

Recursively locks current register model.

Code listing 3.35 uvm_reg_block::is_locked()

```
function bit is_locked ()
```

Returns '1' if current register block is locked, returns '0' otherwise.
All information about `uvm_reg_block` base class is taken from [26].

3.2.2.8 uvm_reg_predictor

Base class `uvm_reg_predictor` is capable of updating `mirrored` values of RAL registers based on observed transactions on the interface. Detailed description of RAL predictor is presented in section 3.2.3.2.

3.2.3 Prediction modes

UVM RAL supports three prediction modes for register models: implicit (auto), explicit and passive.
All information about RAL prediction modes is taken from [27].

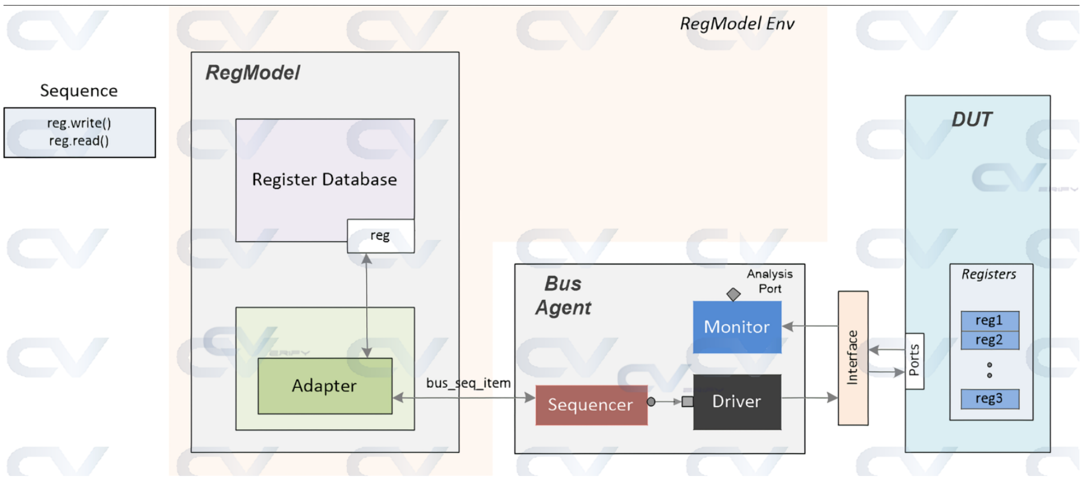


Figure 3.5 Testbench Structure of a Register Model with Implicit Prediction. Unedited picture is taken from: [28]

3.2.3.1 Implicit (auto) prediction

Register model with implicit prediction updates corresponding registers' `mirrored` values (with `uvm_reg::predict()` method) after any read or write operation is executed in current register model. Typical structure of such register model is depicted in figure 3.5.

To enable implicit mode all the user needs to do is to call

```
uvm_reg_map::set_auto_predict(1)
```

for the address map of the top-level register block. With this mode enabled, all registers, that are mapped to that address map, will be updated after any read or write operation is executed via this map.

The sole advantage of implicit prediction is that it is easy to set up – it can be enabled with one method. Big disadvantage of such prediction mode is not being able to update `mirrored` values if the registers are read or written with sequences that are executed directly on the agent's sequencer. To be more specific – register model is blind to any operations with DUT registers if they are not executed through a register model.

3.2.3.2 Explicit prediction

Register model with explicit prediction enabled is able to monitor all transactions (including those outside of register model) that are occurring on the interfaces. To achieve that, register model utilizes so-called predictor, which is represented by `uvm_reg_predictor` base class. Typical structure of such register model is depicted in 3.6.

Explicit prediction is enabled by default, but in order for it to work properly we need to take following steps:

1. Monitor (`uvm_monitor`) needs to be set up on the bus and should be able to catch every transaction that is executed on the registers in the DUT.
2. The predictor (`uvm_reg_predictor`) should be configured:
   a. Corresponding register model address map should be assigned to `map` variable of the `uvm_reg_predictor`.

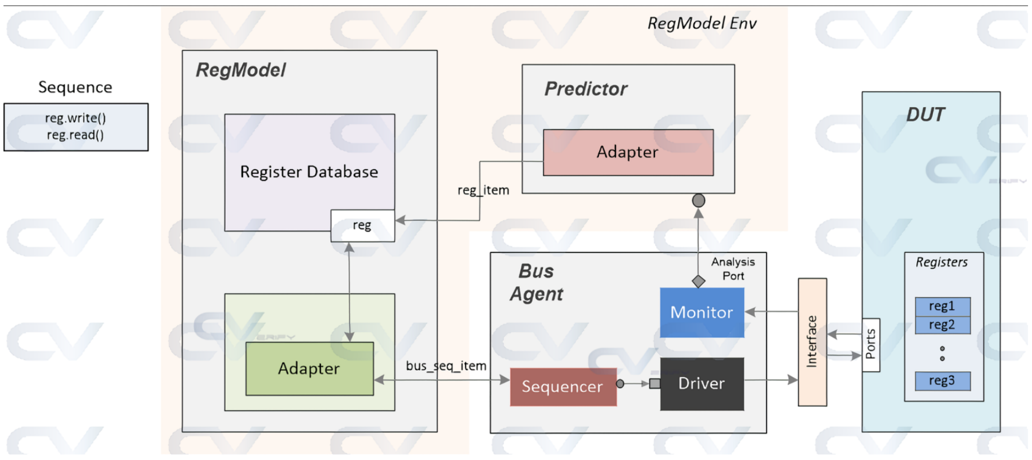


Figure 3.6 Testbench Structure of a Register Model with Explicit Prediction. Source: [28]

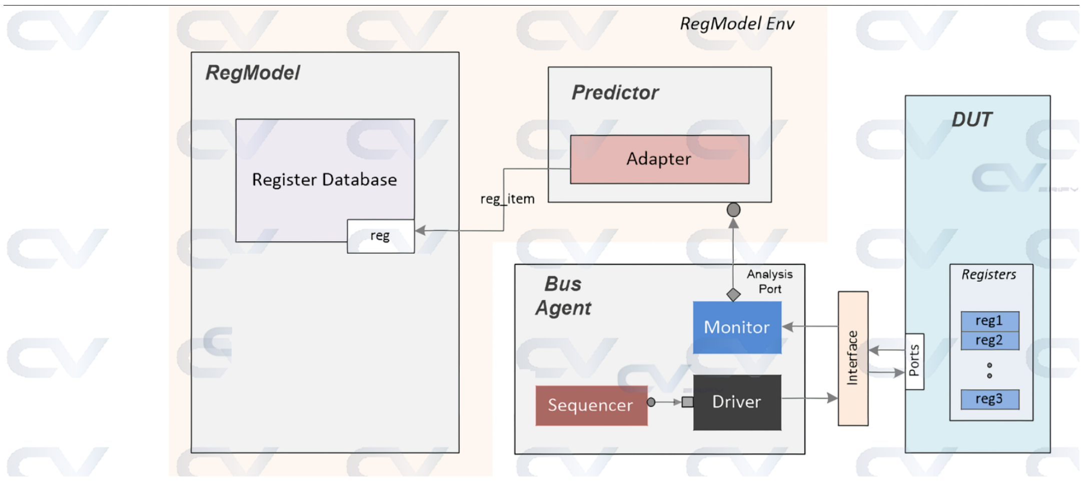


Figure 3.7 Testbench Structure of a Register Model with Passive Prediction. Unedited picture is taken from: [28]

b. Register model adapter should be assigned to `adapter` variable of the `uvm_reg_predictor`. Predictor should be able to convert bus transaction that it gets from the monitor, to register model transactions.
c. Monitor's analysis port should be connected to the `bus_in` analysis port of the predictor.

With predictor properly set up, register model will be able to monitor every occurring transaction on the bus and will be able to update corresponding register's `mirrored` value.

Disadvantage of explicit prediction is that two extra components (monitor and predictor) need to be created and configured.

3.2.3.3 Passive prediction

Register model with passive prediction is able to monitor all transactions on the bus, but is unable to execute any frontdoor operations on the registers itself. Typical structure of register model with passive prediction is presented in figure 3.7.

The sole purpose of such register model is to keep registers up to date with the actual DUT registers.

The only way for such register model to read or write values in DUT registers is to execute methods using backdoor access.

3.2.4 Coverage collection

UVM RAL supports automatic coverage collection for registers and memories. If implemented correctly, coverage can be used to check whether all registers or memory words were read or written, and whether registers and memories had specified values at some point during the simulation.

All information about coverage collection for registers, register blocks and memories are taken from [20], [26] and [23] respectively.

3.2.4.1 include_coverage()

With static method

Code listing 3.36 uvm_reg::include_coverage()

```
function void uvm_reg::include_coverage ( string scope,
                                          uvm_reg_cvr_t models,
                                          uvm_object accessor = null );
  uvm_reg_cvr_rsrc_db::set({"uvm_reg::", scope},
                           "include_coverage",
                           models, accessor);
endfunction
```

we can specify which functional coverage model should be used in corresponding register, register block or memory. This method creates a new database resource, writes `models` value to it, and sets it into the database using scope `{"uvm_reg::", scope}` and name “include_coverage” as the lookup parameters.

Values of `models` variable can be user-defined or predefined by RAL, since `uvm_reg_cvr_t` is just a 32-bit wide variable:

Code listing 3.37 uvm_reg_cvr_t

```
typedef bit [`UVM_REG_CVR_WIDTH-1:0] uvm_reg_cvr_t;
```

`` `UVM_REG_CVR_WIDTH `` has a value of 32 by default.

Predefined functional coverage models are:

Code listing 3.38 Enumeration uvm_coverage_model_e

```
typedef enum uvm_reg_cvr_t {
  UVM_NO_COVERAGE    = 'h0000,
  UVM_CVR_REG_BITS   = 'h0001,
  UVM_CVR_ADDR_MAP   = 'h0002,
  UVM_CVR_FIELD_VALS = 'h0004,
  UVM_CVR_ALL        = -1
} uvm_coverage_model_e;
```

Because these values are the powers of two, desired functional coverage model can be created by bit-wise OR operation between individual model identifiers. Each model represents the following [29]:

UVM_NO_COVERAGE Coverage collection is turned off.

UVM_CVR_REG_BITS Individual register bits.

UVM_CVR_ADDR_MAP Individual register and memory addresses.

UVM_CVR_FIELD_VALS Field values.

UVM_CVR_ALL All coverage models.

3.2.4.2 build_coverage()

The `new()` method of base classes `uvm_reg`, `uvm_reg_block` and `uvm_mem` has a functional coverage model as one of the parameters, which can be set for this register/register block/memory. To properly do so we need to utilize `build_coverage()` method:

Code listing 3.39 uvm_reg::build_coverage()

```
function uvm_reg_cvr_t uvm_reg::build_coverage (uvm_reg_cvr_t models);
  build_coverage = UVM_NO_COVERAGE;
  void'(uvm_reg_cvr_rsrc_db::read_by_name({"uvm_reg::", get_full_name()},
                                          "include_coverage",
                                          build_coverage , this));
  return build_coverage & models;
endfunction: build_coverage
```

Register blocks and memories also have this method with the same structure. This method controls what functional coverage model should be built in current register/register block/memory. It locates a resource by scope

`{"uvm_reg::", get_full_name()}`

and name “include_coverage” and compares its value against `models` coverage model. This method then returns bit-wise AND between `models` and coverage model from database set by `uvm_reg::include_coverage()`.

Build coverage method then should be used when the new instance of register/register block/memory is created. For example, if we want to create 32 bit register with coverage model `UVM_CVR_ALL` we would call `new()` method like that:

```
class my_register extends uvm_reg;
  function new (string name = "my_register");
    super.new(name, 32, build_coverage(UVM_CVR_ALL));
  endfunction
endclass : my_register
```

By using `build_coverage()` method we ensure, that only the desired coverage model is built for current register/register block/memory. For example, if coverage model for a whole register model was set by `uvm_reg::include_coverage()` to `UVM_NO_COVERAGE` and `new()` method was the same as in the code above, no coverage model will be built for this register.

3.2.4.3 add_coverage()

We can expand current functional coverage model with method

Code listing 3.40 uvm_reg::add_coverage()

```
function void uvm_reg::add_coverage (uvm_reg_cvr_t models);
  m_has_cover |= models;
endfunction: add_coverage
```

Notice that there is no way to “delete” an already built coverage model, we can only add new ones (`m_has_cover` is of type `int`). This method should only be called in the constructor of subsequently derived classes.

Register blocks and memories also have this method with the same structure.

3.2.4.4 has_coverage()

We can check if current register/register block/memory has specified functional coverage models.

Code listing 3.41 uvm_reg::has_coverage()

```
function bit uvm_reg::has_coverage (uvm_reg_cvr_t models);
  return ((m_has_cover & models) == models);
endfunction: has_coverage
```

This method returns '1' even when there are other coverage models enabled besides coverage models specified in `models`.

Register blocks and memories also have this method with the same structure.

3.2.4.5 set_coverage()

Coverage sampling can be turned on or off for specified coverage models.

Code listing 3.42 uvm_reg::set_coverage()

```
function uvm_reg_cvr_t uvm_reg::set_coverage (uvm_reg_cvr_t is_on);
  if (is_on == uvm_reg_cvr_t'(UVM_NO_COVERAGE)) begin
    m_cover_on = is_on;
    return m_cover_on;
  end

  m_cover_on = m_has_cover & is_on;

  return m_cover_on;
endfunction: set_coverage
```

This method returns sum of coverage models with enabled coverage sampling (`m_cover_on` is of type `int`). Coverage sampling is not enabled by default.

Memories also have this method with the same structure.

By executing this method on a register block, we enable sampling for specified coverage models for this register block and recursively for all register blocks, registers and memories within it.

3.2.4.6 get_coverage()

To check whether the coverage sampling is turned on for specified coverage models we can use method

Code listing 3.43 uvm_reg::get_coverage()

```
function bit uvm_reg::get_coverage (uvm_reg_cvr_t is_on);
  if (has_coverage(is_on) == 0)
    return 0;
  return ((m_cover_on & is_on) == is_on);
endfunction: get_coverage
```

This method returns '1' even if coverage sampling is enabled for other coverage models besides coverage models specified in `is_on`.

Register blocks and memories also have this method with the same structure.

3.2.4.7 sample()

Sample method for RAL registers

■ **Code listing 3.44** uvm_reg::sample()

```
protected virtual function void sample ( uvm_reg_data_t data,
                                         uvm_reg_data_t byte_en,
                                         bit is_read,
                                         uvm_reg_map map )
```

is empty by default and is executed every time current RAL register is read or written with specified `data` via specified `map` (only if implicit or explicit prediction is correctly set up for this map).

Sample method for RAL register blocks

■ **Code listing 3.45** uvm_reg_block::sample()

```
protected virtual function void sample ( uvm_reg_addr_t offset,
                                         bit is_read,
                                         uvm_reg_map map )
```

is also empty by default and is executed every time an address within one of its address maps in current register block is read or written (only if implicit or explicit prediction is correctly set up for these maps). Offset refers to the offset within current register block, not the absolute address.

Sample method for RAL memories

■ **Code listing 3.46** uvm_mem::sample()

```
protected virtual function void sample ( uvm_reg_addr_t offset,
                                         bit is_read,
                                         uvm_reg_map map )
```

is also empty by default and is executed every time memory word at a specified offset in current memory is read or written. Offset refers to the offset within current memory, not the absolute address.

It is user's responsibility to correctly implement these sample methods for correct coverage collection.

3.2.4.8 sample_values()

This method is almost identical to the `sample()` methods above. The main difference is that it is completely up to user to correctly implement this method, since it is not automatically called whenever RAL register is read or written in register model. Only register blocks and registers support the usage of this method, while memories do not.

If this method is implemented correctly, user can sample values at any point in time during simulation, not necessarily only after read and write operations on registers.

3.2.4.9 Setting up coverage collection

In order to properly set up automatic coverage collection for register block/register/memory we need to:

1. Specify which functional coverage models should be built in register blocks/registers/memories. It should be done before register model is built by utilizing

   ```
   uvm_reg::include_coverage()
   ```

2. Register block/register/memory should be constructed (`new()`) with a desired coverage model by utilizing `build_coverage()` method.

3. Cover groups should be instantiated for specified coverage models based on a `has_coverage()` method.

4. Sample method(s) should sample values based on a `get_coverage()` method.

5. Before starting any sequence/test, sampling for desired coverage models should be enabled for desired register blocks/registers/memories by utilizing `set_coverage()` method.

3.2.5 Built-in RAL sequences

UVM register abstraction layer provides a set of built-in sequences for register model testing. These sequences can be used to check basic functionality of RAL registers and memories. For example, reset values, reading and writing, HDL paths and access policies can be checked by executing corresponding sequence.

To start the desired sequence all the user needs to to is to specify what register/register block/memory should be tested. Than method `start()` of the sequence should be called.

Before starting any RAL built-in sequence DUT should be reset.

Each register/register block/memory can be left out from testing by current sequence. This can be done by setting a bit-type resource into database:

```
uvm_resource_db#(bit)::set (
  {"REG::",regmodel.blk.get_full_name(),".*"},
  "NO_REG_TESTS", 1
);
```

where `regmodel.blk` is either register, register block or memory. Each sequence has their own set of disable resources, which are listed in the following sections.

3.2.5.1 Register Sequences

Built-in RAL registers sequences are:

uvm_reg_hw_reset_seq tests the reset values of registers in specified register block. This sequence reads all registers in register block through all available address maps and compares these read back values with expected reset values [30].

uvm_reg_single_bit_bash_seq tests specified register by writing '1' and '0' values in every bit via all available address maps. With register's mirrored value sequence checks if these values were successfully written.

Registers containing fields with unknown access policies can not be tested [31].

uvm_reg_bit_bash_seq sequence executes `uvm_reg_single_bit_bash_seq` sequence on every register in a provided register block [31].

uvm_reg_single_access_seq verifies, that a specified register can be written through its default address map (via frontdoor) and then can be read using backdoor access and vice versa. Sequence checks, that the result of an operation is equal to register's mirrored value.

This sequence:

1. Writes reverted reset value via frontdoor.
2. Reads that value via backdoor and compares it against mirrored value.
3. Writes reset value via backdoor.
4. Reads that value via frontdoor and compares it against mirrored value.

Table 3.2 Built-in RAL register sequences

Sequence	Disable Resource	Can be executed on
`uvm_reg_hw_reset_seq`	`NO_REG_TESTS` `NO_REG_HW_RESET_TEST`	Register Block
`uvm_reg_single_bit_bash_seq`	`NO_REG_TESTS` `NO_REG_BIT_BASH_TEST`	Register
`uvm_reg_bit_bash_seq`	`NO_REG_TESTS` `NO_REG_BIT_BASH_TEST`	Register Block
`uvm_reg_single_access_seq`	`NO_REG_TESTS` `NO_REG_ACCESS_TEST`	Register
`uvm_reg_access_seq`	`NO_REG_TESTS` `NO_REG_ACCESS_TEST`	Register Block
`uvm_reg_shared_access_seq`	`NO_REG_TESTS` `NO_REG_SHARED_ACCESS_TEST`	Register

Registers that can not be accessed via backdoor, registers with read-only fields or registers containing fields with unknown access policies can not be tested [32].

uvm_reg_access_seq sequence executes `uvm_reg_single_access_seq` sequence on every register in a provided register block [32].

uvm_reg_shared_access_seq verifies the accessibility of a specified register through multiple address maps by writing values to it via all available address maps and then reading it back from all other address maps. Mirrored value of a register is used to check the results.

Registers containing fields with unknown access policies can not be tested [33].

If specified register is only mapped in one address map – sequence does nothing.

Table 3.2 describes the disable resources for these sequences and RAL components, on which they can be executed. Sequence is not executed on a register or register block, if at least one of the corresponding disable resources was set for that register or register block.

3.2.5.2 Memory Sequences

Built-in RAL memory sequences are:

uvm_mem_single_walk_seq executes walking-ones algorithm on a specified memory [34]. Algorithm can be described using pseudocode:

```
// The walking process is, for address k:
// - Write ~k
// - Read k-1 and expect ~(k-1) if k > 0
// - Write k-1 at k-1
// - Read k and expect ~k if k == last address
```

uvm_mem_walk_seq executes `uvm_mem_single_walk_seq` on every memory in specified register block [34].

uvm_mem_single_access_seq writes data to a specified memory using its default address map (via frontdoor) and then reads it back using backdoor access and vice versa. Sequence checks if written and read data are equal.

This sequence:

Table 3.3 Built-in RAL memory sequences

Sequence	Disable Resource	Can be executed on

`uvm_mem_single_walk_seq`	`NO_REG_TESTS`
	`NO_MEM_TESTS`
	`NO_MEM_WALK_TEST`
		Memory

`uvm_mem_walk_seq`	`NO_REG_TESTS`
	`NO_MEM_TESTS`
	`NO_MEM_WALK_TEST`
		Register Block

`uvm_mem_single_access_seq`	`NO_REG_TESTS`
	`NO_MEM_TESTS`
	`NO_MEM_ACCESS_TEST`
		Memory

`uvm_mem_access_seq`	`NO_REG_TESTS`
	`NO_MEM_TESTS`
	`NO_MEM_ACCESS_TEST`
		Register Block

`uvm_mem_shared_access_seq`	`NO_REG_TESTS`
	`NO_MEM_TESTS`
	`NO_REG_SHARED_ACCESS_TEST`
	`NO_MEM_SHARED_ACCESS_TEST`
		Memory

1. Writes random value via frontdoor.
2. If memory can be read, reads that value via backdoor and compares it against written value.
3. Writes the value from step 1 with inverted bits via backdoor.
4. Reads that value via frontdoor and compares it against written value.

Memories without specified backdoor HDL paths can not be tested [35].

uvm_mem_access_seq executes `uvm_mem_single_access_seq` on every memory in specified register block [35].

uvm_mem_shared_access_seq verifies the accessibility of a memory through multiple address maps. This sequence writes data to a specified memory through all available address maps and then reads it back through all other address maps. Sequence checks if written and read data are equal [33].

If specified memory is only mapped in one address map – sequence does nothing.

Table 3.3 describes the disable resources for these sequences and RAL components, on which they can be executed. Sequence is not executed on a memory or register block, if at least one of the corresponding disable resources was set for that memory or register block.

3.2.5.3 Aggregated Sequences

Aggregated sequences execute aforementioned sequences on a specified register block. These sequences are:

uvm_reg_mem_shared_access_seq executes

- `uvm_reg_shared_access_seq` on every register in specified register block [33].
- `uvm_mem_shared_access_seq` on every memory in specified register block [33].

uvm_reg_mem_built_in_seq executes user-defined set of built-in RAL sequences depending on `tests` variable of this sequence [36]:

```
bit [63:0] tests = UVM_DO_ALL_REG_MEM_TESTS;
```

`UVM_DO_ALL_REG_MEM_TESTS` is a default value of `tests` and is a part of the enumeration [29]:

Code listing 3.47 Enumeration uvm_reg_mem_tests_e

```
typedef enum bit [63:0] {
  // Executes uvm_reg_hw_reset_seq
  UVM_DO_REG_HW_RESET      = 64'h0000_0000_0000_0001,

  // Executes uvm_reg_bit_bash_seq
  UVM_DO_REG_BIT_BASH      = 64'h0000_0000_0000_0002,

  // Executes uvm_reg_access_seq
  UVM_DO_REG_ACCESS        = 64'h0000_0000_0000_0004,

  // Executes uvm_mem_access_seq
  UVM_DO_MEM_ACCESS        = 64'h0000_0000_0000_0008,

  // Executes uvm_reg_mem_shared_access_seq
  UVM_DO_SHARED_ACCESS     = 64'h0000_0000_0000_0010,

  // Executes uvm_mem_walk_seq
  UVM_DO_MEM_WALK          = 64'h0000_0000_0000_0020,

  // Executes all of the above
  UVM_DO_ALL_REG_MEM_TESTS = 64'hffff_ffff_ffff_ffff
} uvm_reg_mem_tests_e;
```

By looking at the source code of this sequence [37] we can see that if this sequence is set to execute `uvm_reg_mem_shared_access_seq`, register blocks that have disable resource set to

`NO_MEM_TESTS` or `NO_MEM_SHARED_ACCESS_TEST`

would still have `uvm_reg_mem_shared_access_seq` executed on them, contrary to the source code of `uvm_reg_mem_shared_access_seq` [38] and what is specified in [33].

Same thing happens in case of `uvm_mem_walk_seq` sequence. Register blocks that have disable resource set to

`NO_MEM_TESTS`

would still have `uvm_mem_walk_seq` executed oh them, contrary to the source code of the sequence `uvm_mem_walk_seq` [39] and what is specified in [34].

uvm_reg_mem_hdl_paths_seq verifies that HDL paths for the default design abstraction are accessible by the simulator via backdoor access. If register or memory does not have defined HDL path – it is not checked. This sequence is performed in zero simulation time and does not read/write from/to the DUT [40].

Table 3.4 describes the disable resources for these sequences and RAL components, on which they can be executed. Sequence is not executed on a register block, if at least one of the corresponding disable resources was set for that register block.

Table 3.4 Built-in RAL aggregated sequences

Sequence	**Disable Resource**	**Can be executed on**

`uvm_reg_mem_shared_access_seq`	`NO_REG_TESTS`
	`NO_MEM_TESTS`
	`NO_REG_SHARED_ACCESS_TEST`
	`NO_MEM_SHARED_ACCESS_TEST`
		Register Block

`uvm_reg_mem_built_in_seq`	Depends on the specified sequences	Register Block
`uvm_reg_mem_hdl_paths_seq`	None	Default Design Abstraction

Chapter 4

Implementation

This chapter describes the process of creating testbench and RAL register model for DUT, which consists of block of registers and RAM and ROM memories. After that, the process of integrating this register model into the testbench is explained.

4.1 Used Software and Tools

DVT Eclipse was used as an IDE for this project and Questa simulator was used for simulations. These tools were chosen because the author of this thesis was already familiar with these tools while studying the subject "Digital Circuit Simulation and Verification" (NI-SIM), which is taught at Faculty of Information Technology at Czech technical university in Prague (CTU FIT).

4.2 Initial Testbench

At the start of this project, the goal was to create simple DUT blocks and register model to check if they would work correctly. Initial testbench is depicted in figure 4.1 and:

1. Has DUT block of 16 read-write and 16 read-only registers each 32-bit wide.
2. Has DUT block of RAM.
3. Has implemented register model, which consists of:
   a. 16 read-write and 16 read-only RAL registers.
   b. RAL RAM.
   c. One (default) address map with implicit prediction.
   d. Adapter.
4. Has passive agent which is connected to a default address map.
5. Does not support coverage collection in register model.
6. Is capable of reading and writing registers and RAM using `read()` and `write()` RAL methods, i.e. address map of register model correctly communicates with agent's sequencer and is able to employ adapter component to convert RAL transactions to bus transaction and vice versa.

After that, various improvements were added to that testbench, which are described in section about final testbench 4.3.

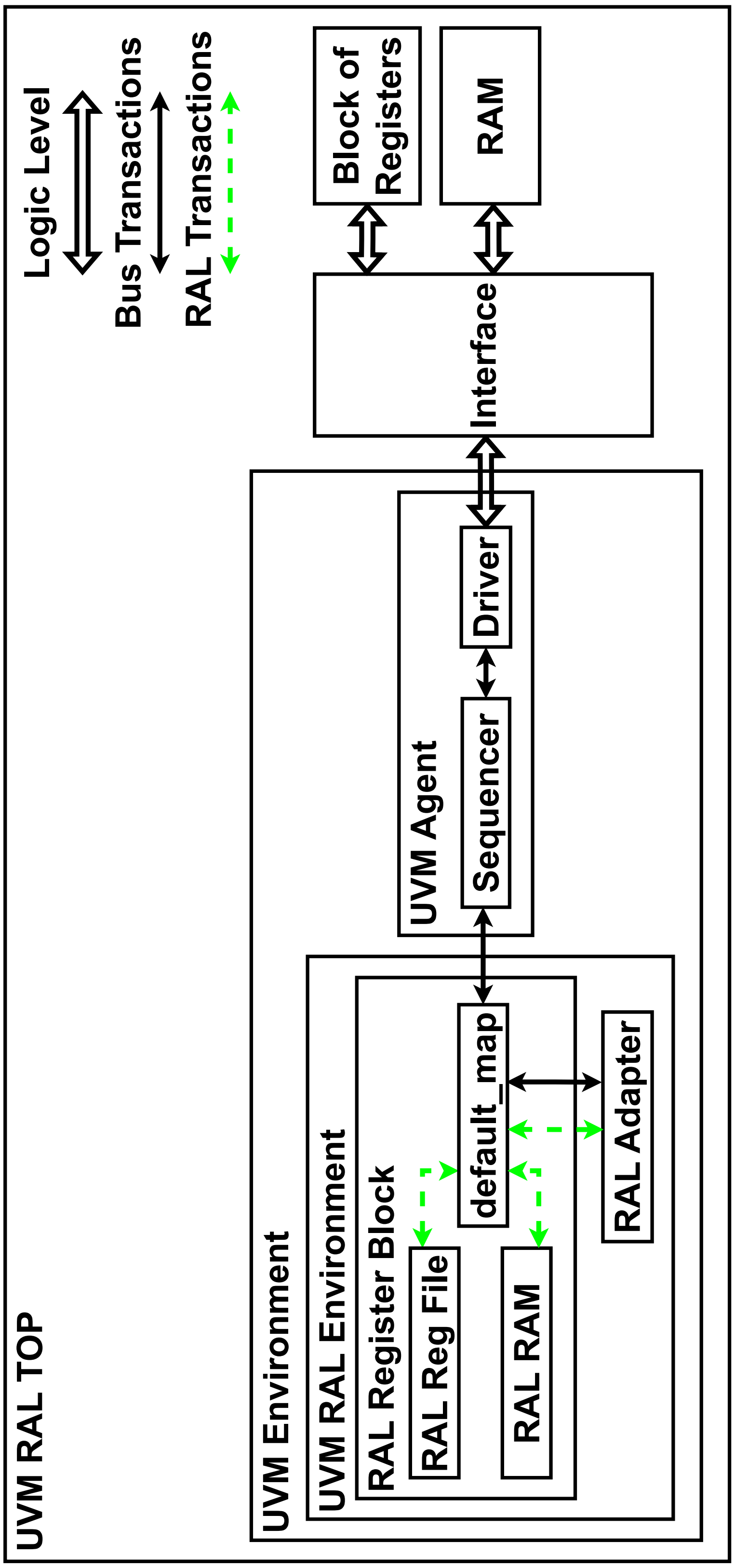


Figure 4.1 Structure of Initial Testbench

4.2.1 Design Under Test

4.2.1.1 Block of Registers

The structure of block of registers is the same as in the Vojtěch Jílek's master's thesis [5], i.e. it contains 32 registers each 32 bits wide. Such structure of general purpose registers is used in the MIPS processors with 32-bit long instructions.

Initial block of registers consists of 16 read-write and 16 read-only registers:

Code listing 4.1 Initial block of registers

```
module my_regs
(
  input  logic clk,
  input  logic reset,
  input  logic [31:0] reg_addr,
  input  logic write_enable,
  input  logic [31:0] reg_in,
  output logic [31:0] reg_out
);

logic [31:0] regs [0:31];

assign reg_out = regs[reg_addr];

always @(posedge clk) begin
  if (reset === 1'b0) begin
    foreach (regs[i]) begin
      regs[i] <= 32'h0000_0000;
    end
  end else begin
    if (write_enable === 1'b1) begin
      // First 16 registers are read-write (regs[0:15])
      // Last 16 registers are read-only (regs[16:31])
      if (reg_addr >= 32'd0 && reg_addr <= 32'd15) begin
        regs[reg_addr] <= reg_in;
      end
    end
  end
end
```

4.2.1.2 RAM

Initial RAM memory consists of 1024 words each 32 bits wide. Such width and depth of the memory are the same as in the Vojtěch Jílek's master's thesis [5]:

Code listing 4.2 Initial RAM

```
module my_mem
(
  input  logic clk,
  input  logic reset,
  input  logic [31:0] mem_addr,
  input  logic write_enable,
  input  logic [31:0] mem_in,
  output logic [31:0] mem_out
);
```

```
logic [31:0] mem [0:1023];

assign mem_out = mem[mem_addr];

always @(posedge clk) begin
  if (reset === 1'b0) begin
    foreach (mem[i]) begin
      mem[i] <= 32'h0000_0000;
    end
  end else begin
    if (write_enable === 1'b1) begin
      mem[mem_addr] <= mem_in;
    end
  end
end
```

4.2.2 Interface

Interface is needed for driver (and later for monitor) to successfully communicate with DUT:

Code listing 4.3 Initial interface

```
interface my_interface;
  logic clk;
  logic reset;

  // Signals that communicate with registers
  logic [31:0] reg_addr;
  logic reg_write_enable;
  logic [31:0] reg_in;
  logic [31:0] reg_out;

  // Signals that communicate with memory
  logic [31:0] mem_addr;
  logic mem_write_enable;
  logic [31:0] mem_in;
  logic [31:0] mem_out;
endinterface
```

4.2.3 Bus Transaction Sequence Item

In order for driver (and later for monitor) to drive (observe) logic level signals on the interface, we need to specify the structure of bus transaction that register model would pass to the driver (would receive from the monitor). We can do that by extending base `uvm_sequence_item` class, where we declare all variables that are needed for communication between DUT and testbench:

```
class my_reg_model_item extends uvm_sequence_item;
  rand logic mem_op;
  rand logic [31:0] addr;
  rand logic write_enable;
  rand logic [31:0] data;

  `uvm_object_utils_begin(my_reg_model_item)
    `uvm_field_int(mem_op, UVM_ALL_ON)
    `uvm_field_int(addr, UVM_ALL_ON)
    `uvm_field_int(write_enable, UVM_ALL_ON)
    `uvm_field_int(data, UVM_ALL_ON)
  `uvm_object_utils_end

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

endclass
```

We need to differentiate between memory and register block operations, that is why 1-bit variable `mem_op` is instantiated (`mem_op == 1'b1` means that current operation is executed in RAM).

Variable `data` can be used as an input and as an output, since `write_enable` variable tells us whether current transaction is of type READ or WRITE.

4.2.4 Register Model

4.2.4.1 RAL Read-write Registers

Each register needs to have register field that can be instantiated and set up later:

```
rand uvm_reg_field reg_data;
```

Function `new()` is as follows:

```
function new (string name = "my_ral_reg_rw");
  super.new(name, 32, UVM_NO_COVERAGE);
endfunction
```

Total number of bits in this register is 32 and no functional coverage models are supported in it.

Function `build()` is as follows:

```
function void build ();
  reg_data = uvm_reg_field::type_id::create("reg_data");
  reg_data.configure(this, 32, 0, "RW", 0, 0, 1, 1, 1);
endfunction
```

Instance of register field is created in build method and then configured to:

1. Have this register as a parent.
2. Have 32 bits.
3. Have the position of LSB to be set to 0 (relative to the LSB of the whole register).
4. Have read-write access.
5. Have volatility bit set to 0.
6. Have reset values set to 0.
7. Have "is bit actually reset?" bit set to 1.
8. Have "can value be randomized?" bit set to 1.
9. Have "is field the only one to occupy a byte lane in the register?" bit set to 1.

4.2.4.2 RAL Read-only Registers

Read-only registers are created in the same way as read-write registers, the only difference is the "RO" access policy in `uvm_reg_field::configure()` method.

4.2.4.3 RAL Register File

Register file consists of aforementioned 16 read-write registers and 16 read-only registers. Contrary to this section's name, these registers were instantiated in `uvm_reg_block` type block, not the `uvm_reg_file`. The reason for this is that when `configure()` method of a register is called, expected arguments are this register's register block parent, register file parent and register's HDL path (refer to 3.2). Register block argument can not have NULL value (otherwise compilation error occurs), therefore in order to properly configure any register we need to have its parent register block already instantiated. That requirement violates the structure of the register model and that is why the decision was made to have all registers in `uvm_reg_block`.

Another reason to use `uvm_reg_block` instead of `uvm_reg_file` is that register file does not support enabling of a desired functional coverage model for registers in it (refer to 3.13 and 3.31).

From this point forward this collection of 32 registers would still be referred to as "register file", whereas "register block" would refer to a RAL component with register file, RAM and register map.

Registers in this register file are instantiated like this:

```
rand my_ral_reg_rw ral_reg_rw_inst [0:15];
rand my_ral_reg_ro ral_reg_ro_inst [0:15];
```

Method `new()`:

```
function new (string name = "my_ral_reg_file");
  super.new(name, UVM_NO_COVERAGE);
endfunction
```

Build method consists of:

```
default_map = create_map("ral_reg_file_map", 0, 4, UVM_LITTLE_ENDIAN);
```

where we create and configure the default address map of this register file to have:

1. Specified name.
2. Specified base address.
3. Specified byte width of the bus on which this map is used.
4. Specified endian format.

Then we need to create, build, configure and set HDL path for each register in this register file:

```
foreach (ral_reg_rw_inst[i]) begin
  ral_reg_rw_inst[i] = my_ral_reg_rw::type_id::create(
    $sformatf("ral_reg_rw_inst_%0d", i)
  );

  ral_reg_rw_inst[i].build();

  ral_reg_rw_inst[i].configure(this);

  ral_reg_rw_inst[i].add_hdl_path_slice(
    $sformatf("regs[%0d]", i), 0, ral_reg_rw_inst[i].get_n_bits()
  );

  default_map.add_reg(ral_reg_rw_inst[i], 4 * i, "RW");
end
```

Configure method sets this register “file” as parent register block for specified register. Using `add_hdl_path_slice()` we can set HDL path for this register to be “regs[i]”, where “i” is register index (from 0 to 15). HDL paths for the registers are set to that string value because registers’ values in the DUT are stored using signal with a name “regs”. These HDL paths would refer to the register field from zero bit (relative to the LSB of the register) with length of 32 bits (`get_n_bits()` returns the width of the register in bits).

After that, current register is added to the default address map of this register file with specified offset and access policy. Offset is set like that because it points to a single byte and each register consists of 4 bytes.

The same process is applied to 16 read-only registers.

4.2.4.4 RAL RAM

Method `new()` of RAM:

```
function new (string name = "my_ral_mem");
  super.new(name, 1024, 32, "RW", UVM_NO_COVERAGE);
endfunction
```

creates RAL instance of memory with specified name, number of memory locations, number of bits in each memory location, access policy and functional coverage model.

Build function:

```
function void build ();
  this.add_hdl_path_slice("mem_inst.mem", 0, 1024 * 32);
endfunction
```

sets HDL path for this memory to a specified string. That HDL path will refer to this memory’s data from specified offset (bit) with a specified size (in bits). HDL path for the memory is set to that string value because memory words in the DUT are stored using signal with a name “mem”. Notice that we do not need to specify HDL path as “mem_inst.mem[i]”, where “i” is an index of a memory word. Instance of DUT RAM in the testbench would need to be called “mem_inst”.

4.2.4.5 RAL Register Block

RAL component regarded as “register block” in the context of this thesis would contain register file and RAM (and ROM in the final testbench):

```
rand my_ral_reg_file ral_reg_file_inst;
rand my_ral_mem ral_mem_inst;
```

Function `new()`:

```
function new (string name = "my_ral_reg_block");
  super.new(name, UVM_NO_COVERAGE);
endfunction
```

Function `build()` consists of:

```
default_map = create_map("ral_reg_block_map", 0, 4, UVM_LITTLE_ENDIAN);
```

Here we configure default map of this register block with specified name, base address, bus width in bytes and endian format.

After that we need to create, build and configure register file:

```
ral_reg_file_inst = my_ral_reg_file::type_id::create(
  "ral_reg_file_inst"
);

ral_reg_file_inst.build();

ral_reg_file_inst.configure(this, "regs_inst");

default_map.add_submap(ral_reg_file_inst.default_map, 0);
```

Configure method sets parent register block and HDL path for this register file. After setting HDL path for this register file to a specified value, all HDL paths of every register in it would start with “regs_inst”. For example, HDL path for a first register would be “regs_inst.regs[0]”. Instance of DUT block of registers in the testbench would need to be called “regs_inst”.

With `add_submap()` method we add a default address map of the register file to a default address map of the register block with a zero offset.

Lastly, we need to create, build and configure RAM:

```
ral_mem_inst = my_ral_mem::type_id::create("ral_mem_inst");
ral_mem_inst.build();
ral_mem_inst.configure(this);
default_map.add_mem(ral_mem_inst, 64'h0000_0000_FF00_0000, "RW");
```

Configure method sets the parent register block of the memory and optionally its HDL path. We do not need to specify HDL path of the memory, because if was already set in the RAL memory itself.

Finally, we add memory to this register block’s default address map with specified offset and access policy. Offset is set to that value to avoid overlapping between register file addresses and memory word addresses.

Address map of the register file was added with zero offset. That means that address
`64'h0000_0000_0000_0000` in default map of the register block would refer to the register `regs[0]`,
`64'h0000_0000_0000_0004` to the register `regs[1]`,
`64'h0000_0000_0000_0008` to the register `regs[2]`
and so on. Last register would have an address
`64'h0000_0000_0000_007C` or 124 in decimal numeral system, since we have 32 registers each 4 bytes wide.

With memory offset being as it is specified, address
`64'h0000_0000_FF00_0000` in default map of the register block would refer to a first memory word – `mem[0]` and
`64'h0000_0000_FF00_0FFC` or 4092 in decimal numeral system would refer to the last memory word – `mem[1023]`, since we have 1024 memory words each 4 bytes wide.

4.2.4.6 RAL Adapter

Adapter needs to have implemented `reg2bus()` and `bus2reg()` methods:

```
function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
  tx = my_pkg::my_reg_model_item::type_id::create("tx");

  tx.addr = rw.addr;
  tx.write_enable = (rw.kind == UVM_WRITE) ? 1 : 0;
  tx.data = rw.data;

  if (rw.addr[31:24] == 8'hFF) begin
    tx.mem_op = 1'b1;
    tx.addr[31:24] = 8'h00;
  end else begin
    tx.mem_op = 1'b0;
  end

  tx.addr = tx.addr >> 2;
  return tx;
endfunction
```

Firstly, we need to create bus transaction (sequence item) and assign proper values to its variables based on a current RAL transaction. Variable `mem_op` is set that way, because when we added memory to the register block we set its address offset to the value

`64'h0000_0000_FF00_0000`

If the current transaction is executed in memory, we set the address byte `[[31:24]]` of bus transaction to zeroes, since in the beginning of this method we just copied the whole address of RAL transaction.

Lastly, we divide that address by 4 (or logically shift it by 2 bits to the right). We need to do that, because in RAL transaction each address points to a single byte of data and therefore address `64'd0` would refer to the first register, address `64'd4` to the second register and so on. But in the actual design each address points to a single register or memory word. For example address `64'd0` in the actual block of registers would refer to the first register, address `64'd1` to the second register and so on.

In `bus2reg()` method:

```
function void bus2reg(uvm_sequence_item bus_item, ref uvm_reg_bus_op rw);
  assert( $cast(tx, bus_item) ) else begin
    `uvm_fatal("my_adapter", "Failed␣to␣cast")
  end

  rw.addr = tx.addr << 2;
  rw.kind = tx.write_enable ? UVM_WRITE : UVM_READ;
  rw.data = tx.data;

  if (tx.mem_op === 1'b1) begin
    rw.addr[31:24] = 8'hFF;
  end

  rw.status = UVM_IS_OK;
endfunction
```

RAL transaction does not need to be created, since it is passed to the method by reference. Firstly, we cast `bus_item` of `uvm_sequence_item` type to our specified bus transaction, i.e. to `my_reg_model_item`. Then we set its variables to proper values. Address needs to be logically shift to the left by 2 bits. If current bus transaction was executed in the memory, we set `[31:24]` address byte of RAL transaction to `8'hFF`, since we set memory address offset to that value in the register model.

4.2.4.7 UVM RAL Environment

UVM RAL environment consists of register block and adapter. In the build phase of UVM RAL environment:

```
function void build_phase (uvm_phase phase);
  super.build_phase(phase);

  ral_reg_block_inst = my_ral_reg_block::type_id::create(
    "ral_reg_block_inst", this
  );

  adapter_inst = my_adapter::type_id::create("adapter_inst", this);

  ral_reg_block_inst.build();
  ral_reg_block_inst.lock_model();
  ral_reg_block_inst.default_map.set_auto_predict(1);
  ral_reg_block_inst.set_hdl_path_root("uvm_ral_top");
endfunction
```

we create instances of register block and adapter, build the register model and then lock it, set implicit prediction mode for default address map of the register block and set HDL path for the register block to a specified string. With such HDL path we can access every register or memory word in it with
"uvm_ral_top.regs_inst.regs[i]" path, where "i" is register index
and
"uvm_ral_top.mem_inst.mem[i]" path , where "i" is memory word index.

4.2.5 UVM Sequencer

Sequencer is parameterized with bus transaction that it will receive from register model. The sole purpose of sequencer is to send that transaction to the driver.

4.2.6 UVM Driver

Driver is parameterized with bus transaction that it will receive from the sequencer. Build phase is used to get interface handle from the database:

```
function void build_phase (uvm_phase phase);
  super.build_phase (phase);

  if (!uvm_config_db#(virtual my_interface)::get(
    null, "", "interface_inst", interface_inst
  ))

    `uvm_fatal("NOVIF", {
      "Missing␣interface_inst:␣", get_full_name(), ".interface_inst"
    })
endfunction
```

In the run phase we:

1. Set interface signals to zeroes.
2. Wait for reset signal to be deasserted.
3. Get new bus transaction from the sequencer.
4. Based on whether current transaction is executed on registers or memory, and whether it is of type READ or WRITE, we set relevant interface signals to proper values. In case of

WRITE transactions we write data on the negative edge of the clock and in case of READ transactions we read data on the positive edge of the clock.

We can get transaction from the sequencer by using

```
seq_item_port.get_next_item(reg_model_item_inst);
```

method, where `seq_item_port` is driver's port to request items from the sequencer and `reg_model_item_inst` is an instance of bus transaction.

With method

```
seq_item_port.item_done();
```

we let the sequencer know, that current bus transaction was written/contains read information from the DUT and is ready to be received.

4.2.7 UVM Agent

UVM agent contains sequencer, driver and no monitor (passive agent). In its build phase we create instance of driver and sequencer and in the connect phase we connect sequencer's port to the driver's port:

```
function void connect_phase (uvm_phase phase);
  driver_inst.seq_item_port.connect(sequencer_inst.seq_item_export);
endfunction
```

4.2.8 UVM Environment

UVM environment consists of UVM agent and UVM RAL environment. In its build phase we create instances of agent and RAL environment and in its connect phase we assign sequencer and adapter for register block's default address map:

```
virtual function void connect_phase (uvm_phase phase);

  ral_env_inst.ral_reg_block_inst.default_map.set_sequencer(
    .sequencer(agent_inst.sequencer_inst),
    .adapter(ral_env_inst.adapter_inst)
  );

endfunction
```

4.2.9 UVM Sequences

All sequences are parameterized with bus transaction sequence item.

4.2.9.1 my_simple_regs_sequence_1

This sequence tests read-write and read-only registers currently present in register model.

Firstly, this sequence gets all registers from the register model with

```
uvm_reg_block::get_registers()
```

method and stores them in a queue. Then, these registers are divided into read-write and read-only register queues using `uvm_reg::get_rights()` method. This method returns the accessibility of a register in a specified map, not the access policy of the register itself. Since we set the accessibility of read-write and read-only registers for default address map of the register block to "RW" and "RO" respectively, we can use `uvm_reg::get_rights()` method to differentiate between the registers.

Next, for each read-write register this sequence executes following methods:

```
foreach (rwregs[i]) begin
  wdata = $urandom();
  rwregs[i].write(status, wdata, UVM_FRONTDOOR);
  rwregs[i].peek(status, rdata);
  assert (wdata === rdata);

  wdata = $urandom();
  rwregs[i].poke(status, wdata);
  rwregs[i].read(status, rdata, UVM_FRONTDOOR);
  assert (wdata === rdata);
end
```

1. Writes random data to a register using `write()` method with frontdoor access.
2. Uses `peek()` method (backdoor access) and checks if this peeked data is equal to the written data.
3. Pokes random data into the register using `poke()` method (backdoor access).
4. Reads the register using `read()` method with frontdoor access and compares poked and read data.

And for each read-only register, sequence:

```
foreach (roregs[i]) begin
  wdata = $urandom();
  roregs[i].poke(status, wdata);
  roregs[i].read(status, rdata, UVM_FRONTDOOR);
  assert (wdata === rdata);

  poked_wdata = wdata;
  wdata = $urandom();
  roregs[i].write(status, wdata, UVM_FRONTDOOR);
  roregs[i].peek(status, rdata);
  assert (poked_wdata === rdata);
end
```

1. Pokes random data into the register using `poke()` method (backdoor access).
2. Reads the register using `read()` with frontdoor access and checks that poked and read data are equal.
3. Writes new random data into the register using `write()` method with frontdoor access.
4. Peeks into the register using `peek()` method (backdoor access) and checks that peeked and poked value from the first step are equal, i.e. register value did not change after frontdoor write.

By using frontdoor and backdoor accesses we ensure that the registers can be successfully read/written and that their HDL paths are set correctly.

4.2.9.2 my_simple_regs_sequence_2

This sequence checks the functionality of read-write and read-only registers the same way as in the `my_simple_regs_sequence_1` sequence, with the exception of comparison between written/poked and read/peeked values. In `my_simple_regs_sequence_2` sequence comparing is done via `uvm_reg::mirror()` method with `UVM_CHECK` argument using frontdoor access. That way there is no need to manually compare written and read values of the register. Code for checking read-write registers:

```
foreach (rwregs[i]) begin
  wdata = $urandom();
  rwregs[i].write(status, wdata, UVM_FRONTDOOR);
  rwregs[i].peek(status, rdata);
  assert (wdata === rdata);

  rwregs[i].poke(status, wdata);
  rwregs[i].mirror(status, UVM_CHECK, UVM_FRONTDOOR);
end
```

Code for checking read-only registers:

```
foreach (roregs[i]) begin
  wdata = $urandom();
  roregs[i].poke(status, wdata);
  roregs[i].mirror(status, UVM_CHECK, UVM_FRONTDOOR);

  poked_wdata = wdata;
  wdata = $urandom();
  roregs[i].write(status, wdata, UVM_FRONTDOOR);
  roregs[i].peek(status, rdata);
  assert (poked_wdata === rdata);
end
```

Comparison still needs to be done manually if we use `peek()` method.

4.2.9.3 my_simple_mem_sequence_1

This sequence tries to write to/read from memory word at offset 512 in all memories in register model (there is only one RAM memory in our register model). By using

```
uvm_reg_block::get_memories()
```

we store all memories into a queue and then write/read data to/from the memory word with offset 512 using frontdoor and backdoor accesses:

```
foreach (mems[i]) begin
  mems[i].read(status, 512, rdata, UVM_FRONTDOOR);
  wdata = $urandom();
  mems[i].write(status, 512, wdata, UVM_FRONTDOOR);
  mems[i].read(status, 512, rdata, UVM_FRONTDOOR);

  //----------------------------------------------------------//

  mems[i].read(status, 512, rdata, UVM_BACKDOOR);
  wdata = $urandom();
  mems[i].write(status, 512, wdata, UVM_BACKDOOR);
  mems[i].read(status, 512, rdata, UVM_BACKDOOR);
```

```
    //----------------------------------------------------------//

    mems[i].read(status, 512, rdata, UVM_FRONTDOOR);
    wdata = $urandom();
    mems[i].write(status, 512, wdata, UVM_FRONTDOOR);
    mems[i].read(status, 512, rdata, UVM_FRONTDOOR);
  end
```

Queue `mem[i]` cycles through all memories in register block (there is only one in our register block). After each `read()` and `write()` methods we display the written/read value using `$display()` methods, which are not present in the code snippet above for the sake of brevity. Comparison between written and read values were done only visually.

4.2.10 UVM Tests

4.2.10.1 my_base_test

Every test is executed on the register model, therefore we would need to instantiate that register model every time we create a new test. To avoid that, this base test was created, where we instantiate whole environment in its build phase. Every subsequent test is extended from this base test.

4.2.10.2 my_custom_sequences_test

This test executes all sequences described in "UVM Sequences" section. In its build phase we instantiate these sequences and in its run phase we pass the instance of the register model to the sequences and then we start them:

```
task run_phase (uvm_phase phase);
  phase.raise_objection(this);

  sequence_inst_1.ral_reg_block_inst =
  env_inst.ral_env_inst.ral_reg_block_inst;

  sequence_inst_2.ral_reg_block_inst =
  env_inst.ral_env_inst.ral_reg_block_inst;

  sequence_inst_3.ral_reg_block_inst =
  env_inst.ral_env_inst.ral_reg_block_inst;

  sequence_inst_1.start(null);
  sequence_inst_2.start(null);
  sequence_inst_3.start(null);

  phase.drop_objection(this);
endtask
```

Normally we would need to pass the handle of a sequencer, on which corresponding sequence needs to be executed, to a `start()` method. In case of sequences, that operate on register model, there is no need for that, since the sequencer of the default address map of the register block is used (that was set with `uvm_reg_map::set_sequencer()`).

Table 4.1 Structure of block of registers in final testbench

Number of registers	Type of register	Indices of registers in DUT
1	READ-WRITE	[0]
1	READ-CLEAR	[1]
1	READ-SET	[2]
1	WRITE-READ-CLEAR	[3]
1	WRITE-READ-SET	[4]
1	WRITE-CLEAR	[5]
1	WRITE-SET	[6]
1	WRITE-1-TOGGLE	[7]
1	WRITE-0-TOGGLE	[8]
7	READ-WRITE	[9:15]
1	READ-ONLY	[16]
1	RW/RO (16b/16b)	[17]
1	RC/RS (16b/16b)	[18]
1	WRC/WRS (16b/16b)	[19]
1	WC/WS (16b/16b)	[20]
1	W1T/W0T (16b/16b)	[21]
10	READ-WRITE	[22:31]

4.2.11 UVM RAL Top Module (Testbench)

Top module is responsible for generating clock signal, invoking active-low reset, instantiating block of registers (with instance name “regs_inst”) and RAM (with instance name “mem_inst”), and for starting specified UVM test.

Before the specified test is started, interface handle needs to be deposited into the UVM database:

```
initial begin
  uvm_config_db#(virtual my_interface)::set(
    uvm_root::get(), "*", "interface_inst", interface_inst
  );

  run_test("my_custom_sequences_test");
end
```

4.2.12 Results of Simulation of Initial Testbench

After executing test `my_custom_sequences_test` no errors were reported. Results of executing `my_simple_mem_sequence_1` were checked visually in console and in waveform viewer, but no inconsistencies were found.

Register model was successfully integrated into the testbench and basic read and write functions were working properly on registers and memory. Now current testbench is ready to be improved further.

4.3 Final Testbench

The structure of final testbench is depicted in figure 4.2 and this testbench:

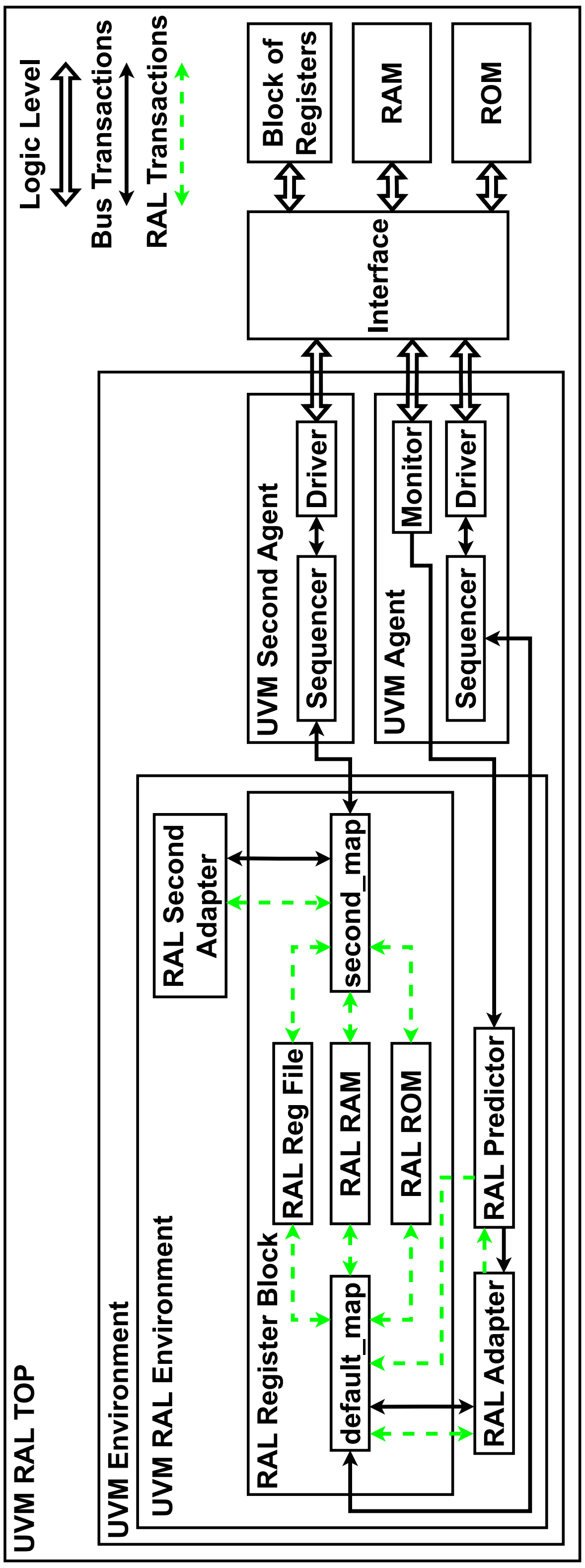


Figure 4.2 Structure of Final Testbench

1. Has DUT block of registers described in the table 4.1, each 32-bit wide.

2. Has DUT blocks of RAM and ROM.

3. Has implemented register model, which consists of:

   a. RAL registers that correspond to the registers in the DUT block of registers.
   b. RAL RAM and ROM.
   c. One or two address maps based on a `TWO_MAPS` macro. If two address maps are used, both of them have implicit prediction turned on. In case of only one (default) address map present – that map is set to have explicit prediction enabled.
   d. One or two adapters based on a `TWO_MAPS` macro.
   e. Has predictor which is connected to default address map.

4. Has active agent, which is connected to default address map. If macro `TWO_MAPS` is not defined, its monitor sends observed transactions to the predictor.

5. Has passive agent, which is connected to the second address map, and is instantiated if `TWO_MAPS` is defined.

6. Fully supports coverage collection based on a specified functional coverage model.

7. Is capable of verifying the functionality of DUT blocks using custom and built-in RAL sequences.

The reason for instantiating components and setting prediction modes based on a `TWO_MAPS` macro is described in 4.3.10.

4.3.1 Design Under Test

4.3.1.1 Block of Registers

RTL code of block of registers was re-written to reflect its new structure described in the table 4.1.

4.3.1.2 RAM

Structure of RAM remained the same as in the initial testbench with the exception of its size. Now it is possible to define RAM size with `RAM_BYTES` macro, which can be specified in the package `my_pkg.sv`. Size of RAM should be at least 4 bytes. Default value of `RAM_BYTES` is 2^{12}.

```
logic [31:0] mem [0:(‘RAM_BYTES/4)-1];
```

4.3.1.3 ROM

ROM has the same structure as RAM minus the possibility of outputting any data. Its size can be configured using `ROM_BYTES` macro in `my_pkg.sv` package. Default value of `ROM_BYTES` is 2^{12}.

4.3.2 Interface

Interface was updated to support communication with ROM:

```
interface my_interface;
  logic clk;
  logic reset;

  // Signals that communicate with registers
  logic [31:0] reg_addr;
  logic reg_write_enable;
  logic [31:0] reg_in;
  logic [31:0] reg_out;

  // Signals that communicate with RAM
  logic [31:0] ram_addr;
  logic ram_write_enable;
  logic [31:0] ram_in;
  logic [31:0] ram_out;

  // Signals that communicate with ROM
  logic [31:0] rom_addr;
  logic [31:0] rom_out;
endinterface
```

4.3.3 Bus Transaction Sequence Item

Bus transaction sequence item was updated to support transaction that would be executed on ROM. Instead of one-bit variable new enumeration was created in package `my_pkg.sv`:

```
typedef enum bit [1:0] {REGS = 2'b00,
                        RAM  = 2'b01,
                        ROM  = 2'b10} target_type;
```

Variables in the bus sequence item were changed as follows:

```
rand my_pkg::target_type target;
rand logic [31:0] addr;
rand logic write_enable;
rand logic [31:0] data;
logic reset = 1'b1;
```

With `reset` variable this bus sequence item can be used to invoke reset in the DUT at any given time – there is no reason to invoke reset from the top testbench module anymore.

4.3.4 Register Model

Registers, register file and memories are configured in the same way as in the initial testbench with the exception of coverage collection settings, which are described in detail in the following sections.

Because RAL provides functional coverage models for individual register bits, field values and addresses 3.38, cover groups for written/read bits/values/addresses were created for registers/register file/memories in this register model.

4.3.4.1 RAL Read-write Registers

To check, whether current register had specified value during read operation or specified value was written to it at some time during simulation, we need to create a cover group for it.

Cover group for checking if all individual bits were written to RW register:

```
covergroup wr_bits;
  option.per_instance = 1;

  wr_bits : coverpoint reg_data.value[31:0] {
    bins wr_bits[] = { [0:(33'd2**33'd32)-1] }
    with ( $onehot(item) == 1 );
  }
endgroup
```

Variable `per_instance` is responsible for saving statistics of current cover group individually for every instance of this register. Variable `reg_data` is a register field of current register and variable `value` contains its mirrored value.

Then, we check if current field value is within a specified range – it should be greater or equal than 0 and lesser or equal than $2^{32} - 1$. Those are all possible values of a 32-bit field. Since default width of integers in SystemVerilog is 32 bits, we need to extend operands to 33 bits first.

Lastly, separate cover point bin is created for every one-hot value.

Identical cover group that checks whether every individual bit of the register was read is created with a name `rd_bits`.

Cover group for written field values is than created and is as follows:

```
covergroup wr_vals;
  option.per_instance = 1;

  wr_vals : coverpoint reg_data.value[31:0] {
    bins wr_vals[4] = { [0:(33'd2**33'd32)-1] };
  }
endgroup
```

where all possible values are equally divided between four cover point bins.

Identical cover group for read values is created with a name `rd_vals`.

Function `new()` of the register was updated to create current register with potentially all functional coverage models. What models are actually created depends on:

```
uvm_reg::include_coverage()
```

Then, aforementioned cover groups are instantiated based on with what coverage models current register was created:

```
function new (string name = "my_ral_reg_rw");
  super.new(name, 32, build_coverage(UVM_CVR_ALL));

  if (has_coverage(UVM_CVR_REG_BITS)) begin
    wr_bits = new();
    rd_bits = new();
  end

  if (has_coverage(UVM_CVR_FIELD_VALS)) begin
    wr_vals = new();
    rd_vals = new();
  end
endfunction
```

Sampling method is than implemented. Values are sampled based on whether sampling was enabled (`set_coverage()`) for specified functional coverage models in current register:

```
virtual function void sample (uvm_reg_data_t data,
                              uvm_reg_data_t byte_en,
                              bit            is_read,
                              uvm_reg_map    map);

  if (get_coverage(UVM_CVR_REG_BITS)) begin
    if (is_read) begin
      rd_bits.sample();
    end else if (!is_read) begin
      wr_bits.sample();
    end
  end

  if (get_coverage(UVM_CVR_FIELD_VALS)) begin
    if (is_read) begin
      rd_vals.sample();
    end else if (!is_read) begin
      wr_vals.sample();
    end
  end
endfunction
```

Sampling for read and write cover groups occurs based on a type of current operation.

4.3.4.2 Remaining RAL Registers

Cover groups for all remaining registers are configured the same way as for read-write registers and according to the table 4.2.

Registers, that contain two 16-bits fields can be configured the same way as registers, that contain one 32-bits field. The user only needs to instantiate and configure two separate fields. For example, to create W1T/W0T register, we need to instantiate two register fields:

```
rand uvm_reg_field reg_data_w1t;
rand uvm_reg_field reg_data_w0t;
```

And then properly configure them in the build phase:

```
function void build ();
  reg_data_w1t = uvm_reg_field::type_id::create("reg_data_w1t");
  reg_data_w0t = uvm_reg_field::type_id::create("reg_data_w0t");

  reg_data_w0t.configure(this, 16, 0, "W0T", 0, 0, 1, 1, 1);
  reg_data_w1t.configure(this, 16, 16, "W1T", 0, 0, 1, 1, 1);
endfunction
```

Second parameter is width of this field and third parameter is position of its LSB relative to the LSB of the whole register.

In case of write-1-toggle and write-0-toggle registers we do not sample mirrored `value` of the field, but auxiliary variable. The reason for that is because when we write values to these registers they will not contain these exact values after write operations. That is why we need to sample values that is about to be written, not the value the register will contain after the operation.

Table 4.2 Data sampling settings for RAL registers

Register	Individual register bits		Field values	
	During WRITE	**During READ**	**During WRITE**	**During READ**
RW	One-hot values	One-hot values	All possible values divided into four bins	All possible values divided into four bins
RC	×	One-hot values	×	All possible values divided into four bins and all zeroes value
RS	×	One-cold values	×	All possible values divided into four bins and all ones value
WRC	One-hot values	One-hot values	All possible values divided into four bins	All possible values divided into four bins and all zeroes value
WRS	One-cold values	One-cold values	All possible values divided into four bins	All possible values divided into four bins and all ones value
WC	×	One-hot values	×	All possible values divided into four bins and all zeroes value
WS	×	One-cold values	×	All possible values divided into four bins and all ones value
W1T	One-hot values to be written	One-hot values	×	All possible values divided into four bins
W0T	One-cold values to be written	One-cold values	×	All possible values divided into four bins
RO	×	One-hot values	×	All possible values divided into four bins
RW/RO RC/RS WRC/WRS WC/WS W1T/W0T	Coverage settings for individual fields are the same as for corresponding access policy above			

4.3.4.3 RAL Register File

Register file is configured the same way as in the initial testbench. The only difference is coverage settings and presence of the second address map.

Second address map is instantiated and registers are added to it only if macro `TWO_MAPS` is defined.

Cover groups for individual register addresses are created to check if all registers have been read or written. Cover group for write operations is as follows:

```
wr_addr : coverpoint cov_offset {
  bins wr_addr[] = { [0:(31*4)] } with ( (item % 4) == 0 );

  ignore_bins ignore_addr = { 1*4,  2*4, 16*4, 18*4, 22*4, 23*4, 24*4,
                              25*4, 26*4, 27*4, 28*4, 29*4, 30*4, 31*4};
}
```

Auxiliary variable `cov_offset` contains an offset of a register that is a target of a write operation. Aforementioned offsets are excluded from coverage collection, since these they refer to the read-only registers: RC (regs[1]), RS (regs[2]), RO (regs[16]), RC/RS (regs[18]) and 10 RO registers (regs[22-31]).

Cover group for register addresses during read operations is:

```
rd_addr : coverpoint cov_offset {
  bins rd_addr[] = { [0:(31*4)] } with ( (item % 4) == 0 );
}
```

Since all registers can be read, we need to check all possible register offsets.

Register file is then created with potentially all coverage models. Cover groups for register addresses are instantiated in a `new()` function based on a `UVM_CVR_ADDR_MAP` coverage model and are sampled if sampling for that model is enabled.

4.3.4.4 RAL RAM

RAM is configured the same way as in the initial testbench with the exception of coverage collection that checks if all memory words were read and written:

```
wr_addr : coverpoint cov_offset {
  bins wr_addr[] = { [0:( ((‘RAM_BYTES/4)-1) * 4 )] } with
                   ( (item % 4) == 0 );
}
```

Identical cover group with a name `rd_addr` for memory word offsets during read operations is also created.

RAM is then created with potentially all coverage models. Cover groups are instantiated and data are sampled based on a `UVM_CVR_ADDR_MAP` coverage model.

4.3.4.5 RAL ROM

ROM is identical to the RAL RAM with the exception of cover group for memory word offsets during write operations. Since ROM can not be written, cover group for write operations does not exist.

4.3.4.6 RAL Register Block

Register block is configured the same way as in the initial testbench with the exception of second address map and offsets of register file and memories.

Register block is created with potentially all coverage models. Second address map is instantiated, created and register file and memories are added to it if macro `TWO_MAPS` is defined.

Offsets of register file, RAM and ROM for default address map are as follows:

```
default_map.add_submap(ral_reg_file_inst.default_map, 0);
default_map.add_mem(ral_ram_inst, 64'h4000_0000_0000_0000, "RW");
default_map.add_mem(ral_rom_inst, 64'h8000_0000_0000_0000, "RO");
```

Offsets are set like that to correspond to the enumeration specified in 4.3.3 (we can differentiate between register file and memories based on the two most significant bits of the address).

Offsets for second address map are set differently:

```
`ifdef TWO_MAPS
  second_map.add_mem(ral_rom_inst, 0, "RO");
  second_map.add_submap(ral_reg_file_inst.second_map,
                        64'h4000_0000_0000_0000);
  second_map.add_mem(ral_ram_inst,
                     64'h8000_0000_0000_0000, "RW");
`endif
```

The reason for that is to showcase that both of the address maps would refer to the same register/RAM/ROM even if their offsets for them are different.

4.3.4.7 RAL Adapter

Adapter for default address map of the register block was changed to reflect new offsets for registers/RAM/ROM:

```
function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
  tx = my_pkg::my_reg_model_item::type_id::create("tx");
  tx.addr = rw.addr;
  tx.write_enable = ((rw.kind == UVM_WRITE) ||
                     (rw.kind == UVM_BURST_WRITE)) ? 1 : 0;
  tx.data = rw.data;

  case (rw.addr[63:62])
    2'b00 : begin
      tx.target = my_pkg::REGS;
    end

    2'b01 : begin
      tx.target = my_pkg::RAM;
    end

    2'b10 : begin
      tx.target = my_pkg::ROM;
    end
  endcase

  tx.addr = tx.addr >> 2;
  return tx;
endfunction
```

`UVM_BURST_WRITE` kind of operation is used when memory `burst_write()` method is executed. Function `bus2reg()` is:

```
function void bus2reg(uvm_sequence_item bus_item, ref uvm_reg_bus_op rw);
  assert( $cast(tx, bus_item) ) else begin
```

```
    `uvm_fatal("my_adapter", "Failed␣to␣cast")
  end

  rw.addr = tx.addr << 2;
  rw.kind = tx.write_enable ? UVM_WRITE : UVM_READ;
  rw.data = tx.data;

  case (tx.target)
    my_pkg::REGS : begin
      rw.addr[63:62] = 2'b00;
    end

    my_pkg::RAM : begin
      rw.addr[63:62] = 2'b01;
    end

    my_pkg::ROM : begin
      rw.addr[63:62] = 2'b10;
    end
  endcase

  rw.status = UVM_IS_OK;
endfunction
```

While converting bus transaction to a register layer transaction there is no need to differentiate between `UVM_BURST_READ/WRITE` and `UVM_READ/WRITE` kinds of transactions, since burst read/write operation is just an array of individual read/write operations.

4.3.4.8 RAL Second Adapter

Second adapter is associated with the second address map in the register model and is identical to the adapter of the default address map. The only difference is that it converts transaction based on the offsets for registers/RAM/ROM in second address map.

4.3.4.9 RAL Predictor

In order to properly connect predictor to an address map there is no need to implement custom class which would be extended from the base class of the predictor – everything can be done inside RAL environment.

4.3.4.10 UVM RAL Environment

UVM RAL environment consists of register block, adapter for the default address map, predictor and adapter for the second address map if macro `TWO_MAPS` is defined. Predictor is instantiated and parameterized with bus transaction sequence item:

```
uvm_reg_predictor #(my_pkg::my_reg_model_item) predictor_inst;
```

In the build phase these components are created (based on a `TWO_MAPS`), register model is built, locked, its HDL root path is set and prediction modes are configured for default and second address maps. Predictor is created as follows:

```
predictor_inst = uvm_reg_predictor#(my_pkg::my_reg_model_item)::
  type_id::create("predictor_inst", this);
```

If we are using two address maps in our register model – both of them should be set to have implicit prediction enabled, otherwise only one (default) address map is used and is set to have explicit prediction enabled:

```
`ifdef TWO_MAPS
  ral_reg_block_inst.default_map.set_auto_predict(1);
  ral_reg_block_inst.second_map.set_auto_predict(1);
`else
  ral_reg_block_inst.default_map.set_auto_predict(0);
`endif
```

In the connect phase we specify what address map should be associated with predictor and what adapter it should use to convert bus transactions to RAL transactions:

```
function void connect_phase (uvm_phase phase);
  super.connect_phase(phase);

  predictor_inst.map = ral_reg_block_inst.default_map;
  predictor_inst.adapter = default_map_adapter_inst;
endfunction
```

The only thing left to do for predictor to work properly is to connect monitor to it – that will be done in the UVM environment.

4.3.5 UVM Sequencer

This sequencer is identical to the sequencer in the initial testbench, it serves only one purpose – to send bus transactions from the sequence to the driver.

4.3.6 UVM Driver

Driver was changed to support communication with ROM and to be able to invoke reset based on a corresponding bit in the bus transaction.

In case of transactions, that are executed on a block of registers, we change the address of a target register to `32'hFFFF_FFFF` on the next negative edge of the clock after reading/writing data from/to that register. The reason for that change is the presence of write-1/0-toggle type registers. Initially, we would write data to the register and wait for a couple of positive edges of the clock, all the while register address on the interface stays the same. But if we write to write-1/0-toggle register that way – the value of the register would change multiple times, even if we initially intended to only write to this register once. That is why we need to change the address to a value, that does not correspond to any existing register in the DUT.

4.3.7 UVM Monitor

Monitor is used to observe transactions on the interface and is needed for predictor. It contains UVM analysis port which is parameterized with bus transaction sequence item. In its build phase, analysis port is created and interface handle is received from the UVM database.

Run phase consists of one forever loop, where the interface is observed for register read and write transactions on every positive edge of the clock. Valid transaction is the one that does not have its address set to `32'hFFFF_FFFF`. Caught register transaction is then send to the analysis port only if macro `TWO_MAPS` is not defined. That way we can ensure, that predictor would actually receive any transactions from the monitor only if one (default) address map is present and it has explicit prediction enabled:

```
`ifndef TWO_MAPS
  monitor_port.write(reg_model_item_inst);
`endif
```

Since RAL memories does not support `desired` and `mirrored` values, there is no need to monitor interface for memory transactions.

4.3.8 UVM Agent and Second Agent

Agent and second agent are the instances of the same class and they consist of aforementioned driver, sequencer and monitor.

Sequencer's port and driver's port are connected in the connect phase.

4.3.9 UVM Environment

Environment consists of agent, second agent if macro `TWO_MAPS` is defined, and RAL environment. These components are instantiated and then created in the build phase (depending on `TWO_MAPS` macro).

Connect phase is used to

1. Set default map's sequencer and adapter to agent's sequencer and adapter.
2. Set second map's sequencer and adapter to second agent's sequencer and second adapter if macro `TWO_MAPS` is defined.
3. Connect analysis ports of the agent's monitor and predictor with each other.

```
virtual function void connect_phase (uvm_phase phase);
  ral_env_inst.ral_reg_block_inst.default_map.set_sequencer(
    .sequencer(agent_inst.sequencer_inst),
    .adapter(ral_env_inst.default_map_adapter_inst)
  );

  `ifdef TWO_MAPS
    ral_env_inst.ral_reg_block_inst.second_map.set_sequencer(
      .sequencer(second_agent_inst.sequencer_inst),
      .adapter(ral_env_inst.second_map_adapter_inst)
    );
  `endif

  agent_inst.monitor_inst.monitor_port.connect(
    ral_env_inst.predictor_inst.bus_in
  );
endfunction
```

Contrary to what was depicted in figure 4.2, second agent actually has a monitor, but it is not connected to any other UVM component, and if second agent is instantiated, its monitor does not send any data at all.

4.3.10 Reason for utilization of TWO_MAPS macro

After the testbench was implemented some simple methods that read/write data from/to the registers/memories and methods that manipulate with desired/mirrored values were executed to test if testbench is working at all. At this point TWO_MAPS macro did not exist and register model had two address maps – default map with explicit prediction and second map with implicit prediction enabled. After the execution of some RAL methods it was discovered, that mirrored values of the registers did not contain the correct values after write and read operation, and the reason for that is the presence of write-1/0-toggle registers.

Let's say we want to write to write-1-toggle register via the second map. Because it is configured to have implicit prediction, it updates the mirrored value of that register in the register model with value, that this register will have after the operation. But because we also have a default map with explicit prediction, agent's monitor catches that write transaction on the interface, sends it to the predictor, and predictor sends it to the default map, which, in turn, updates mirrored value of the same register again, thus reverting its value back to the previous one. In the end, register model thinks that two write operations occurred on that register.

The reason for such behavior is because every address map in the register model is supposed to be connected to a different physical interface – multiple address maps should not be connected to the same interface, otherwise errors like described above will occur. In our testbench we have two address maps on the same interface mainly to showcase the usage of built-in `uvm_reg_mem_shared_access_seq` sequence, which tests the accessibility of each register and memory in the register model via multiple address maps. If we want to have multiple maps on the same interface we need to make them blind to the transactions, that are occurring on that same interface – i.e. we need them to have implicit prediction enabled.

To summarize the effects of the TWO_MAPS macro – if this macro is defined:

1. Second address map, its corresponding adapter and second agent are instantiated, created and configured.
2. Both default and second maps are configured to have implicit prediction enabled.
3. Monitors do not send any values to their analysis ports.

If TWO_MAPS macro is not defined:

1. Second address map, its corresponding adapter and second agent are not instantiated.
2. Only one (default) address map is present in the register model and it is configured to have explicit prediction enabled.
3. Monitor sends caught transactions on the interface to the predictor.

4.3.11 UVM Sequences

my_reg_rw_manual_seq

This sequence is similar to the `my_simple_regs_sequence_1` in the initial testbench. The difference is that here we execute frontdoor write, peek, poke, frontdoor read methods to verify the functionality of only read-write registers in the register model.

The other difference is the manner of detecting which registers in the register model is of type read-write, since we can not use `uvm_reg::get_rights()` method anymore, because it returns the accessibility of a register in a given address map. For example, read-write and write-1-toggle registers would both have their accessibility set to "RW" in address map(s).

That is why we utilize methods

1. `uvm_reg_block::get_registers()` to get all registers in specified register block.
2. `uvm_reg::get_fields()` to get all register fields in specified register.
3. `uvm_reg_field::get_access()` to get the actual access policy of specified register field.

my_reg_rw_mirror_seq

This sequence is identical to `my_reg_rw_manual_seq` with the exception of using mirror method for checking written and poked values instead of doing it manually. For every read-write register in the register model we execute frontdoor write, backdoor mirror, poke, frontdoor mirror methods to verify their functionality.

my_reg_ro_manual_seq

This sequence is similar to `my_reg_rw_manual_seq` with the exception of verifying read-only registers. For every read-only register in the register block we execute poke, frontdoor read, frontdoor write, peek methods to verify their functionality.

my_reg_ro_mirror_seq

This sequence is identical to `my_reg_ro_manual_seq` with the exception of using mirror method for checking poked and written values instead of doing it manually. For every read-only register in the register model we execute poke, frontdoor mirror, frontdoor write, backdoor mirror methods to verify their functionality.

my_reg_desired_seq

This sequence does not actually test any registers but only showcases the usage of `set()`, `get()` and `update()` methods by executing them on read-only and read-write registers. These methods are related to the desired values of the registers and this sequence is used to present how the users can utilize such methods in their testbenches.

my_reg_mirrored_seq

This sequence does not actually test any registers but only showcases the usage of `predict()` and `mirror()` methods by executing them on read-only and read-write registers. These methods are related to the mirrored values of the registers and this sequence is used to present how the users can utilize such methods in their testbenches.

my_reg_exp_prediction

This sequence shows the difference between implicit and explicit prediction. Random value is written to write-read-set register without using RAL and mirrored value of that register is then displayed. If implicit prediction is enabled – mirrored value is not updated and in case of explicit prediction – mirrored value is updated.

my_all_regs_0_1_seq

This sequence writes and pokes all possible one-hot and one-cold values to all registers present in the current register model. For every register:

1. One-hot value is written using frontdoor.
2. Two frontdoor mirror methods are executed. It is done to verify functionality of (write)-read-clear/set registers.
3. Same one-hot value is poked into the register.
4. Two frontdoor mirror methods are executed.

5. Steps 1 through 4 are executed for every possible one-hot value.

6. Step 5 is executed but one-cold values are used instead.

my_mem_ram_read_write_seq

This sequence checks the functionality of RAM. For every memory word:

1. Random data is written via frontdoor.

2. Data is peeked and compared with written data.

3. Random data is poked.

4. Data is read via frontdoor and compared with poked data.

my_mem_rom_read_write_seq

This sequence is identical to `my_mem_ram_read_write_seq`, but instead, it verifies functionality of ROM. For every memory word:

1. Random data is poked.

2. Data is read via frontdoor and compared with poked data.

3. Random data is written via frontdoor.

4. Data is peeked and compared with poked data.

my_mem_burst_seq

This sequence uses burst methods to verify functionality of RAM:

1. Dynamic arrays for writing and reading with the size of RAM are created and randomized.

2. Write array is written to RAM using frontdoor `burst_write()`.

3. By using backdoor `burst_read()` we store all of the RAM data into the read array and compare it with array containing written values.

4. We repeat steps 2 and 3 with backdoor `burst_write()` and frontdoor `burst_read()`.

Since backdoor burst methods respect access policies of memories, we can not use burst methods to verify ROM.

my_reset_seq

This sequence resets all DUT blocks and register model. DUT blocks are reset by sending bus transaction sequence item with reset bit directly to the sequencer without RAL. After that, all mirrored and desired values of the registers in the register model are reset using `uvm_reg_block::reset()` method.

4.3.12 UVM Tests

4.3.12.1 my_base_test

This base test is identical to the one in the initial testbench with the exception of coverage setting:

```
uvm_reg::include_coverage("*", UVM_CVR_ALL);
```

This way all coverage models are set for all RAL components (for their `build_coverage()` methods) in the register model.

4.3.12.2 my_built_in_sequences_test

This sequence executes built-in RAL sequences on current register model. These sequences are:

1. `uvm_reg_hw_reset_seq`
2. `uvm_reg_bit_bash_seq`
3. `uvm_reg_access_seq`
4. `uvm_mem_walk_seq`
5. `uvm_mem_access_seq`
6. `uvm_reg_mem_shared_access_seq`
7. `uvm_reg_mem_hdl_paths_seq`

After instantiating and creating these sequences, register model that these sequences should be executed on is passed to a `model` variable of these sequences. Sequences then can be started by calling their `start(null)` method.

Before starting any built-in sequences, DUT and register model should be reset. It is done via `my_reset_seq` sequence, which is executed before starting any built-in sequence. In contrast to built-in RAL sequences, we need to specify reset sequence's sequencer, since it is executed outside of RAL:

```
reset_seq_inst.start(env_inst.agent_inst.sequencer_inst);
```

Coverage sampling is also enabled for all coverage models for all RAL components (for their `get_coverage()` methods) in register model:

```
env_inst.ral_env_inst.ral_reg_block_inst.set_coverage(UVM_CVR_ALL);
```

4.3.12.3 my_custom_sequences_test

This sequence resets the DUT and register model and then executes following sequences:

1. `my_reg_rw_manual_seq`
2. `my_reg_rw_mirror_seq`
3. `my_reg_ro_manual_seq`
4. `my_reg_ro_mirror_seq`
5. `my_reg_desired_seq`

6. `my_reg_mirrored_seq`
7. `my_reg_exp_prediction`
8. `my_all_regs_0_1_seq`
9. `my_mem_ram_read_write_seq`
10. `my_mem_rom_read_write_seq`
11. `my_mem_burst_seq`

Coverage sampling is also enabled for all coverage models for all RAL components (for their `get_coverage()` methods) in register model.

4.3.12.4 my_full_test

This sequence executes all sequences described in

`my_built_in_sequences_test`

and

`my_custom_sequences_test`

with the exception of sequences that do not verify any functionality of DUT blocks:

1. `my_reg_desired_seq`
2. `my_reg_mirrored_seq`
3. `my_reg_exp_prediction`

4.3.13 UVM RAL Top Module (Testbench)

Top testbench module is identical to the one in the initial testbench. The only difference is the absence of active-low reset generation, since sequence `my_reset_seq` is now used to trigger reset in the DUT blocks.

Chapter 5

Testing

The purpose of this chapter is to run aforementioned tests with different coverage models and sampling settings to ensure, that coverage collection for register model was built correctly. A number of bugs were also artificially inserted into the DUT blocks to showcase that custom and built-in sequences would be capable of catching them.

5.1 Inserted Bugs

Bugs are artificially inserted into the block of registers, RAM and ROM if macros `REG_BUGS`, `RAM_BUGS` and `ROM_BUGS` respectively are defined. What kind of bugs are inserted is described in the table 5.1.

5.2 Execution of my_custom_sequences_test

5.2.1 Identifying Bugs

This test was executed with all bugs related macros defined. Test was able to discover all inserted bugs with the exception of bug regarding reset value of WC register, since none of the sequences in this test check reset values of the registers.

All log messages with caught bugs is too long to list here, that is why the whole log file after executing this test is available on the GitLab page of this project. Some of these log messages are:

Read-write register (regs[0]):

```
# Frontdoor write 32’h40026e11 to register ral_reg_rw_inst_0. (@ 80.00 ns)
# Peeked value in register ral_reg_rw_inst_0 is 32’h40026e12. (@ 80.00 ns)
# UVM_ERROR ./UVM/Sequences/Register_Sequences/my_reg_rw_manual_seq.sv(91)
@ 80.00 ns: reporter@@reg_rw_manual_seq_inst [peek_after_write]
Peeked value is not the same as written value in register ral_reg_rw_inst_0.
Written value = 32’h40026e11, peeked value = 32’h40026e12
```

Read-clear/read-set register:

```
# Frontdoor write 32’h00000001 to register ral_reg_rc_rs_inst_18.
(@ 42000.00 ns)
# Read register ral_reg_rc_rs_inst_18 and compare its value with the current
mirrored value in register model. (@ 42000.00 ns)
```

Table 5.1 Inserted bugs in the DUT blocks

Name of the register/RAM/ROM	**Description of the bug**
RW (regs[0])	Written value is incremented by 1
RC	After every read operation register value is set to `32'h8000_0000`
RS	After every read operation register value is set to `32'hFFFF_FFFE`
WRC	Only 16 LSB bits can be written, register is not cleared after read operations
WRS	Only 16 MSB bits can be written, register value after read operations is `32'h1111_1111`
WC	After write operations register value is set to `32'h8000_0000`, reset value is set to `32'h0010_0000`
WS	Register can be written (behavior of a RW register)
W1T	Value is toggled on '0' value (behavior of W0T register)
W0T	After toggling, value is incremented by 1
7 RW (regs[9:15])	Values are not written (behavior of RO register)
RO (regs[16])	Values can be written (behavior of RW register)
RW/RO	RW field value is set to `16'hCAFE` after every write operation
RC/RS	[31:17] bits are RC while [16:0] bits of the register are RS
WRC/WRS	WRC field value is set to `16'h0000` after every write operation, WRS field value is set to `16'hF7FF` after every read operation
WC/WS	16 MSB bits are WS while 16 LSB bits are WC
W1T/W0T	Whole register behaves like W1T register
RAM	Word with index '0' is always written with value incremented by 1, word with index `(`RAM_BYTES/4)/2` is always written with value `32'hABCD_DCBA`, word with index `(`RAM_BYTES/4)-1` can not be written
ROM	On every positive edge of the clock, word with index '0' is written with zeroes, word with index `(`ROM_BYTES/4)/2` is written with value `32'hABCD_DCBA`, word with index `(`ROM_BYTES/4)-1` is written with value `32'h1234_5678`

```
# Read register ral_reg_rc_rs_inst_18 one more time
(check value after read operation) and compare its value with the current
mirrored value in register model. (@ 42040.00 ns)
# UVM_ERROR verilog_src/uvm-1.1d/src/reg/uvm_reg.svh(2889) @ 42080.00 ns:
reporter [RegModel] Register "ral_reg_block_inst.ral_reg_file_inst.
ral_reg_rc_rs_inst_18" value read from DUT (0x000000000001ffff) does not
match mirrored value (0x000000000000ffff)
# UVM_INFO verilog_src/uvm-1.1d/src/reg/uvm_reg.svh(2902) @ 42080.00 ns:
reporter [RegModel] Field reg_data_rc (ral_reg_block_inst.ral_reg_file_inst.
ral_reg_rc_rs_inst_18[31:16]) mismatch read=16'h1 mirrored=16'h0
```

Write-clear/write-set register:

```
# Frontdoor write 32'h00000001 to register ral_reg_wc_ws_inst_20.
(@ 16400.00 ns)
# Read register ral_reg_wc_ws_inst_20 and compare its value with
the current mirrored value in register model. (@ 16400.00 ns)
# UVM_ERROR verilog_src/uvm-1.1d/src/reg/uvm_reg.svh(2889) @ 16440.00 ns:
reporter [RegModel] Register "ral_reg_block_inst.ral_reg_file_inst.
ral_reg_wc_ws_inst_20" value read from DUT (0x00000000ffff0000)
does not match mirrored value (0x000000000000ffff)
# UVM_INFO verilog_src/uvm-1.1d/src/reg/uvm_reg.svh(2902) @ 16440.00 ns:
reporter [RegModel] Field reg_data_ws (ral_reg_block_inst.ral_reg_file_inst.
ral_reg_wc_ws_inst_20[15:0]) mismatch read=16'h0 mirrored=16'hffff
# UVM_INFO verilog_src/uvm-1.1d/src/reg/uvm_reg.svh(2902) @ 16440.00 ns:
reporter [RegModel] Field reg_data_wc (ral_reg_block_inst.ral_reg_file_inst.
ral_reg_wc_ws_inst_20[31:16]) mismatch read=16'hffff mirrored=16'h0
```

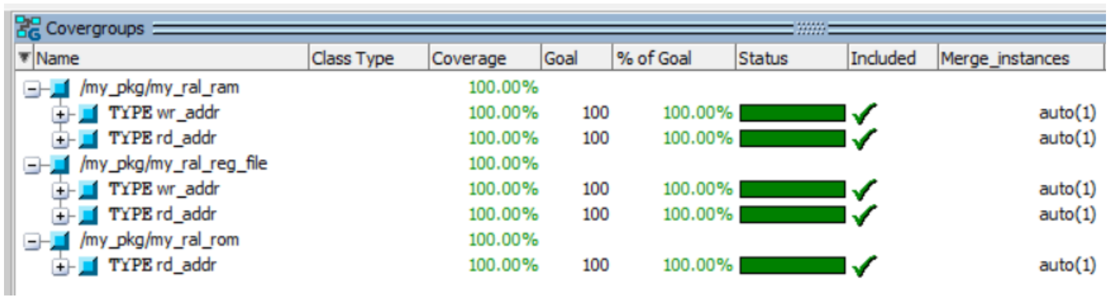
RAM:

```
# UVM_ERROR ./UVM/Sequences/Memory_Sequences/my_mem_burst_seq.sv(72)
@ 597480.00 ns: reporter@@mem_burst_seq_inst [Written and read values
are not equal] Written memory word wdata[0] = 32'he5c0e086 is not equal
to read memory word rdata[0] = 32'he5c0e087
# UVM_ERROR ./UVM/Sequences/Memory_Sequences/my_mem_burst_seq.sv(72)
@ 597480.00 ns: reporter@@mem_burst_seq_inst [Written and read values
are not equal] Written memory word wdata[512] = 32'h425717fb is not equal
to read memory word rdata[512] = 32'habcddcba
# UVM_ERROR ./UVM/Sequences/Memory_Sequences/my_mem_burst_seq.sv(72)
@ 597480.00 ns: reporter@@mem_burst_seq_inst [Written and read values
are not equal] Written memory word wdata[1023] = 32'h10654181 is not equal
to read memory word rdata[1023] = 32'hc89a216a
```

ROM:

```
# UVM_ERROR ./UVM/Sequences/Memory_Sequences/my_mem_rom_read_write_seq.sv(97)
@ 484920.00 ns: reporter@@mem_rom_read_write_seq_inst [peek_after_poke]
Peeked value is not the same as the initially poked value in memory
ral_rom_inst at offset 0.
Poked value = 32'h50c66d2a, peeked value = 32'h00000000

# UVM_ERROR ./UVM/Sequences/Memory_Sequences/my_mem_rom_read_write_seq.sv(97)
@ 525880.00 ns: reporter@@mem_rom_read_write_seq_inst [peek_after_poke]
Peeked value is not the same as the initially poked value in memory
ral_rom_inst at offset 512.
Poked value = 32'h65598b51, peeked value = 32'habcddcba
```



Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_instances
/my_pkg/my_ral_ram		100.00%					
TYPE wr_addr		100.00%	100	100.00%	🟩	✓	auto(1)
TYPE rd_addr		100.00%	100	100.00%	🟩	✓	auto(1)
/my_pkg/my_ral_reg_file		100.00%					
TYPE wr_addr		100.00%	100	100.00%	🟩	✓	auto(1)
TYPE rd_addr		100.00%	100	100.00%	🟩	✓	auto(1)
/my_pkg/my_ral_rom		100.00%					
TYPE rd_addr		100.00%	100	100.00%	🟩	✓	auto(1)

Figure 5.1 Custom sequences test – collected coverage for individual register and memory addresses models only

```
# UVM_ERROR ./UVM/Sequences/Memory_Sequences/my_mem_rom_read_write_seq.sv(97)
@ 566760.00 ns: reporter@@mem_rom_read_write_seq_inst [peek_after_poke]
Peeked value is not the same as the initially poked value in memory
ral_rom_inst at offset 1023.
Poked value = 32'h547ba9bc, peeked value = 32'h12345678
```

Test was also executed with defined macro `TWO_MAPS` – test produced the same results, with the exact number of UVM info and UVM error messages as in the previous run.

If test is executed without inserted bugs (with or without `TWO_MAPS`), no UVM error is reported besides the two expected error messages from `my_reg_mirrored_seq` sequence, where we manually change mirror value of the register and then execute `mirror()` method with `UVM_CHECK`.

5.2.2 Coverage Collection

Test was also executed with different coverage model and sampling settings. If we turn on only specific coverage models in the base test, only these models will be built. If we have all coverage models enabled, but only some of them have enabled sampling, only these specified models will be sampled.

For example, if we only enable coverage model for individual register and memory addresses (`UVM_CVR_ADDR_MAP`) in the base test (using `include_coverage()`) and enable sampling on all possible coverage models – only cover groups for register and memory word addresses will be built and sampled 5.1.

If we build all possible coverage models but enable sampling only for individual register bits and for register and memory word addresses (`set_coverage()` with `UVM_CVR_REG_BITS` + `UVM_CVR_ADDR_MAP` in the test before executing any sequences) – all coverage models will be built, but only cover groups for register bits and for register and memory word addresses will be sampled 5.2.

If all coverage models are built and sampled – coverage collection is at 100% 5.3.

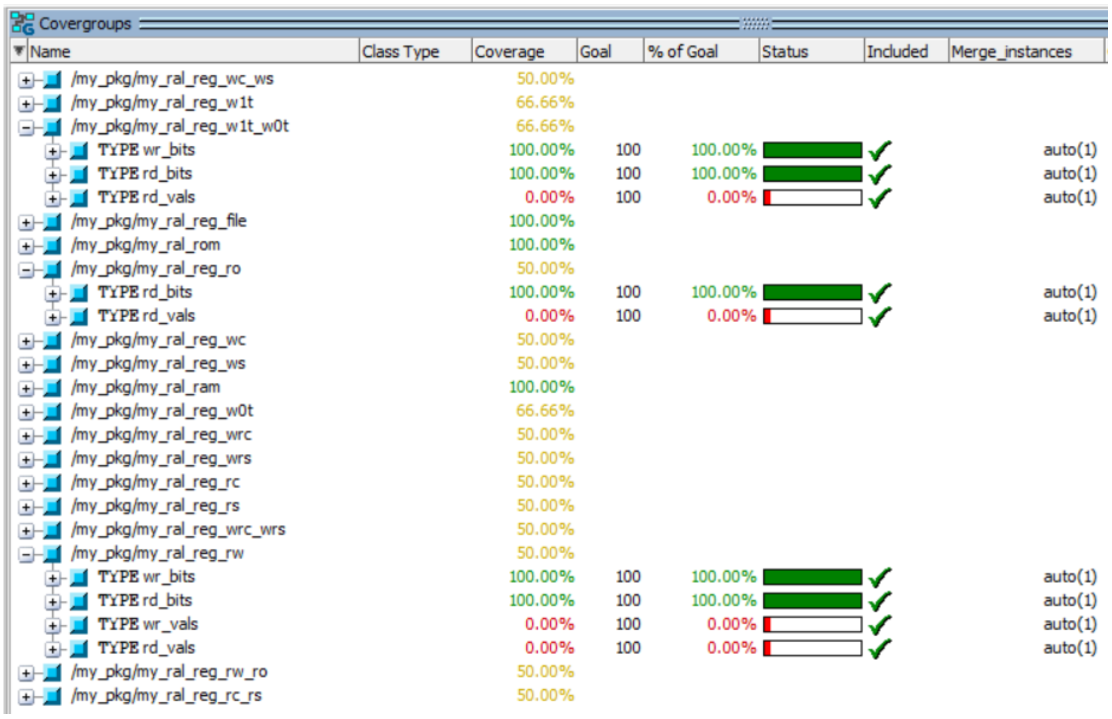
Aforementioned coverage collections are the same with or without inserted bugs or `TWO_MAPS` macro defined.

5.3 Execution of my_built_in_sequences_test

5.3.1 Identifying Bugs

Test was executed with all bugs related macros defined and without `TWO_MAPS` macro. Built-in sequences were not able to detect following bugs:

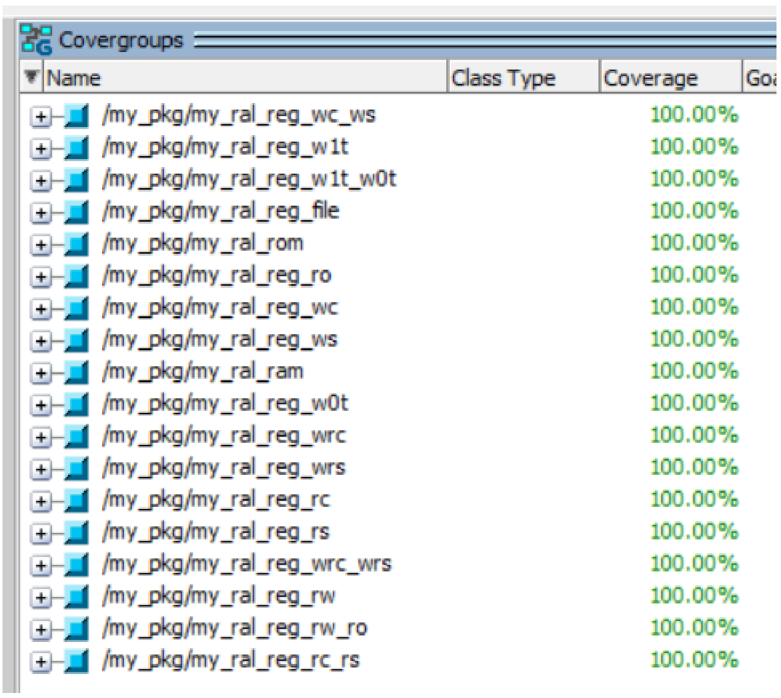
1. Write-read-clear register is not cleared after read operations.



Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_instances
/my_pkg/my_ral_reg_wc_ws		50.00%					
/my_pkg/my_ral_reg_w1t		66.66%					
/my_pkg/my_ral_reg_w1t_w0t		66.66%					
TYPE wr_bits		100.00%	100	100.00%	🟩	✓	auto(1)
TYPE rd_bits		100.00%	100	100.00%	🟩	✓	auto(1)
TYPE rd_vals		0.00%	100	0.00%	🟥	✓	auto(1)
/my_pkg/my_ral_reg_file		100.00%					
/my_pkg/my_ral_rom		100.00%					
/my_pkg/my_ral_reg_ro		50.00%					
TYPE rd_bits		100.00%	100	100.00%	🟩	✓	auto(1)
TYPE rd_vals		0.00%	100	0.00%	🟥	✓	auto(1)
/my_pkg/my_ral_reg_wc		50.00%					
/my_pkg/my_ral_reg_ws		50.00%					
/my_pkg/my_ral_ram		100.00%					
/my_pkg/my_ral_reg_w0t		66.66%					
/my_pkg/my_ral_reg_wrc		50.00%					
/my_pkg/my_ral_reg_wrs		50.00%					
/my_pkg/my_ral_reg_rc		50.00%					
/my_pkg/my_ral_reg_rs		50.00%					
/my_pkg/my_ral_reg_wrc_wrs		50.00%					
/my_pkg/my_ral_reg_rw		50.00%					
TYPE wr_bits		100.00%	100	100.00%	🟩	✓	auto(1)
TYPE rd_bits		100.00%	100	100.00%	🟩	✓	auto(1)
TYPE wr_vals		0.00%	100	0.00%	🟥	✓	auto(1)
TYPE rd_vals		0.00%	100	0.00%	🟥	✓	auto(1)
/my_pkg/my_ral_reg_rw_ro		50.00%					
/my_pkg/my_ral_reg_rc_rs		50.00%					

Figure 5.2 Custom sequences test – collected coverage for all coverage models built, but with enabled sampling only for individual register bits and for register and memory word addresses



Covergroups

Name	Class Type	Coverage	Goa
/my_pkg/my_ral_reg_wc_ws		100.00%	
/my_pkg/my_ral_reg_w1t		100.00%	
/my_pkg/my_ral_reg_w1t_w0t		100.00%	
/my_pkg/my_ral_reg_file		100.00%	
/my_pkg/my_ral_rom		100.00%	
/my_pkg/my_ral_reg_ro		100.00%	
/my_pkg/my_ral_reg_wc		100.00%	
/my_pkg/my_ral_reg_ws		100.00%	
/my_pkg/my_ral_ram		100.00%	
/my_pkg/my_ral_reg_w0t		100.00%	
/my_pkg/my_ral_reg_wrc		100.00%	
/my_pkg/my_ral_reg_wrs		100.00%	
/my_pkg/my_ral_reg_rc		100.00%	
/my_pkg/my_ral_reg_rs		100.00%	
/my_pkg/my_ral_reg_wrc_wrs		100.00%	
/my_pkg/my_ral_reg_rw		100.00%	
/my_pkg/my_ral_reg_rw_ro		100.00%	
/my_pkg/my_ral_reg_rc_rs		100.00%	

Figure 5.3 Custom sequences test – collected coverage with all coverage models built and sampled

2. Write-read-set register has a value of `32'h1111_1111` after read operations.

3. All ROM bugs.

All other bugs were successfully discovered. List of all error messages is too long to describe here, log file after executing this test is saved on GitLab page of this project. Some of these error messages are:

Reset value of WC register:

```
# Starting uvm_reg_hw_reset_seq_inst sequence...
# UVM_ERROR verilog_src/uvm-1.1d/src/reg/uvm_reg.svh(2889) @ 440.00 ns:
reporter [RegModel] Register "ral_reg_block_inst.ral_reg_file_inst.
ral_reg_wc_inst_5" value read from DUT (0x0000000000100000) does not
match mirrored value (0x0000000000000000)
```

Write-read-clear register (bug regarding the inability to write to 16 MSB bits of the register):

```
# Starting uvm_reg_bit_bash_seq_inst sequence...
# UVM_ERROR verilog_src/uvm-1.1d/src/reg/sequences/
uvm_reg_bit_bash_seq.svh(174) @ 60320.00 ns:
reporter@@uvm_reg_bit_bash_seq_inst.reg_single_bit_bash_seq
[uvm_reg_bit_bash_seq] Writing a 1 in bit #16 of register
"ral_reg_block_inst.ral_reg_file_inst.ral_reg_wrc_inst_3" with
initial value 'h0000000000000000 yielded 'h0000000000000000
instead of 'h0000000000010000
```

Write-1-toggle/write-0-toggle register:

```
# Starting uvm_reg_bit_bash_seq_inst sequence...
# UVM_ERROR verilog_src/uvm-1.1d/src/reg/sequences/
uvm_reg_bit_bash_seq.svh(174) @ 1440.00 ns:
reporter@@uvm_reg_bit_bash_seq_inst.
reg_single_bit_bash_seq [uvm_reg_bit_bash_seq] Writing a 1
in bit #0 of register "ral_reg_block_inst.ral_reg_file_inst.
ral_reg_w1t_w0t_inst_21" with initial value 'h0000000000000000
yielded 'h0000000000000001 instead of 'h000000000000fffe
```

RAM:

```
# Starting uvm_mem_walk_seq_inst sequence...
# UVM_ERROR verilog_src/uvm-1.1d/src/reg/sequences/uvm_mem_walk_seq.svh(143)
@ 166980.00 ns: reporter@@uvm_mem_walk_seq_inst.single_mem_walk_seq
[uvm_mem_walk_seq] "ral_reg_block_inst.ral_ram_inst[1-1]" read back as
'h0000000000000000 instead of 'h00000000fffffff.
# UVM_ERROR verilog_src/uvm-1.1d/src/reg/sequences/uvm_mem_walk_seq.svh(143)
@ 218180.00 ns: reporter@@uvm_mem_walk_seq_inst.single_mem_walk_seq
[uvm_mem_walk_seq] "ral_reg_block_inst.ral_ram_inst[513-1]" read back as
'h00000000abcddcba instead of 'h00000000fffffdff.
# UVM_ERROR verilog_src/uvm-1.1d/src/reg/sequences/uvm_mem_walk_seq.svh(165)
@ 269250.00 ns: reporter@@uvm_mem_walk_seq_inst.single_mem_walk_seq
[uvm_mem_walk_seq] "ral_reg_block_inst.ral_ram_inst[1023]" read back as
'h0000000000000000 instead of 'h00000000fffffc00.
```

Described register bugs were not discovered because built-in sequences write some value to these registers and, after that, read them only once. That way register correctly returns that written value, but fails to clear/set its bits after read operation. After that, built-in sequences write another value to these registers, without reading it one more time to check if the bits were cleared/set correctly after previous read operation. That was the exact reason for two `mirror()` methods in our `my_all_regs_0_1_seq` sequence.

ROM bugs were not discovered, because `uvm_mem_access_seq` sequence writes some value to the ROM via backdoor and reads it on the very next positive edge of the clock, at precisely the same time when error data is about to be written. Because of that, sequence receives correct data that was just written to a memory word and no error is reported. It is also stated, that ROM should be idle and not modify its data during the execution of sequence [35].

If test is executed with all inserted bugs and with `TWO_MAPS` macro, all bugs besides the ROM bugs are identified. Previously undiscovered bugs in write-read-clear and write-read-set registers were caught thanks to the `uvm_reg_mem_shared_access_seq` sequence, which does nothing in case of only one (default) address map in register model:

```
# Starting uvm_reg_mem_shared_access_seq_inst sequence...
# UVM_ERROR verilog_src/uvm-1.1d/src/reg/sequences/
uvm_reg_mem_shared_access_seq.svh(172) @ 1483260.00 ns: reporter@@
uvm_reg_mem_shared_access_seq_inst.reg_shared_access_seq
[uvm_reg_shared_access_seq] Register "ral_reg_block_inst.ral_reg_file_inst.
ral_reg_wrc_inst_3" through map "ral_reg_block_inst.ral_reg_file_inst.
ral_reg_file_second_map" is 'h000000000000c29e instead of 'h0000000000000000
after writing 'hb1fc7863cf54c29e via map "ral_reg_block_inst.ral_reg_file_inst.
ral_reg_file_default_map" over 'h0000000000000000.
```

```
# Starting uvm_reg_mem_shared_access_seq_inst sequence...
# UVM_ERROR verilog_src/uvm-1.1d/src/reg/sequences/
uvm_reg_mem_shared_access_seq.svh(172) @ 1483020.00 ns: reporter@@
uvm_reg_mem_shared_access_seq_inst.reg_shared_access_seq
[uvm_reg_shared_access_seq] Register "ral_reg_block_inst.ral_reg_file_inst.
ral_reg_wrs_inst_4" through map "ral_reg_block_inst.ral_reg_file_inst.
ral_reg_file_second_map" is 'h0000000011111111 instead of 'h00000000ffffffff
after writing 'hc8fb6c9121487b42 via map "ral_reg_block_inst.ral_reg_file_inst.
ral_reg_file_default_map" over 'h0000000000000000.
```

If the test is executed without any bugs inserted and with or without `TWO_MAPS` defined, no errors are reported.

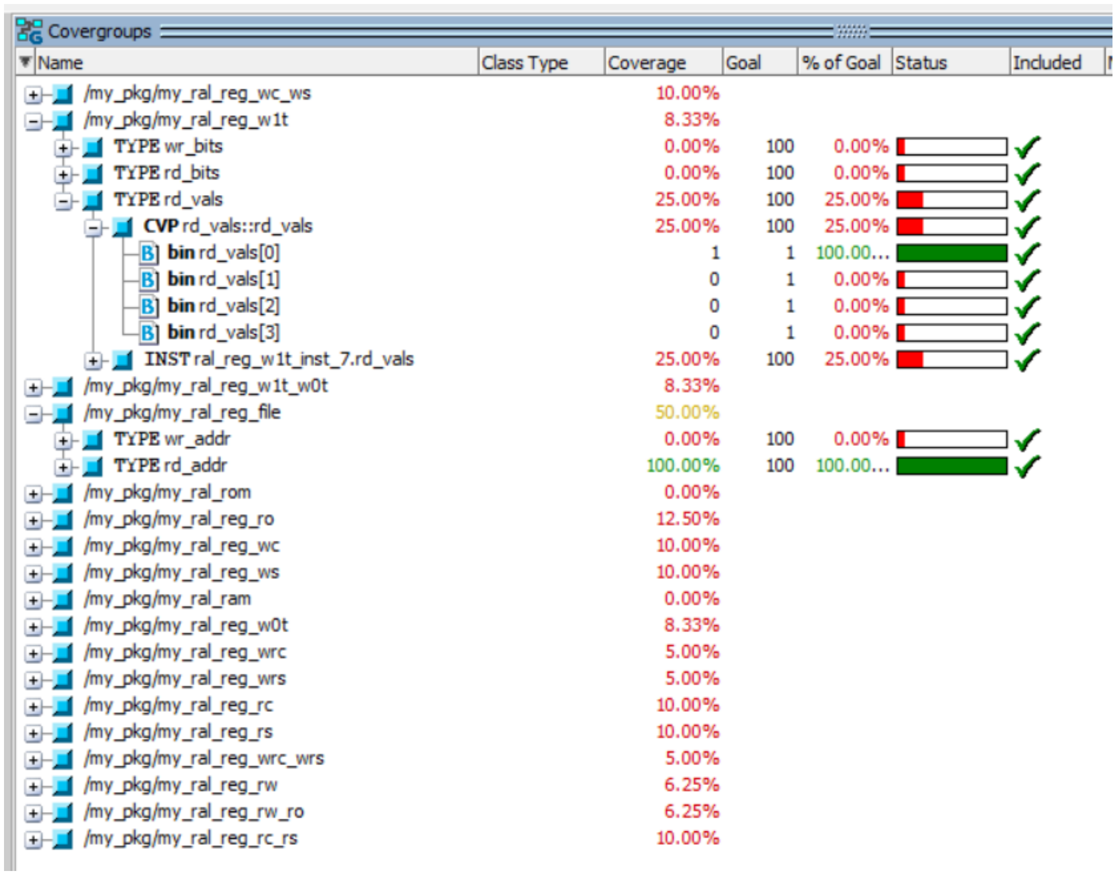
5.3.2 Coverage Collection

Test was executed with various functional coverage models enabled and with sampling enabled for various coverage models. Coverage models were successfully (not) built and (not) sampled based on a `include_coverage()` and `set_coverage()` methods.

If test is executed with all possible coverage models enabled and with sampling enabled on all of them, total coverage is not at 100%. That is the reason why this test was executed a couple more times with only one built-in sequence executed at a time to check coverage collection of every individual sequence. No bugs were inserted during that process.

5.3.2.1 uvm_reg_hw_reset_seq

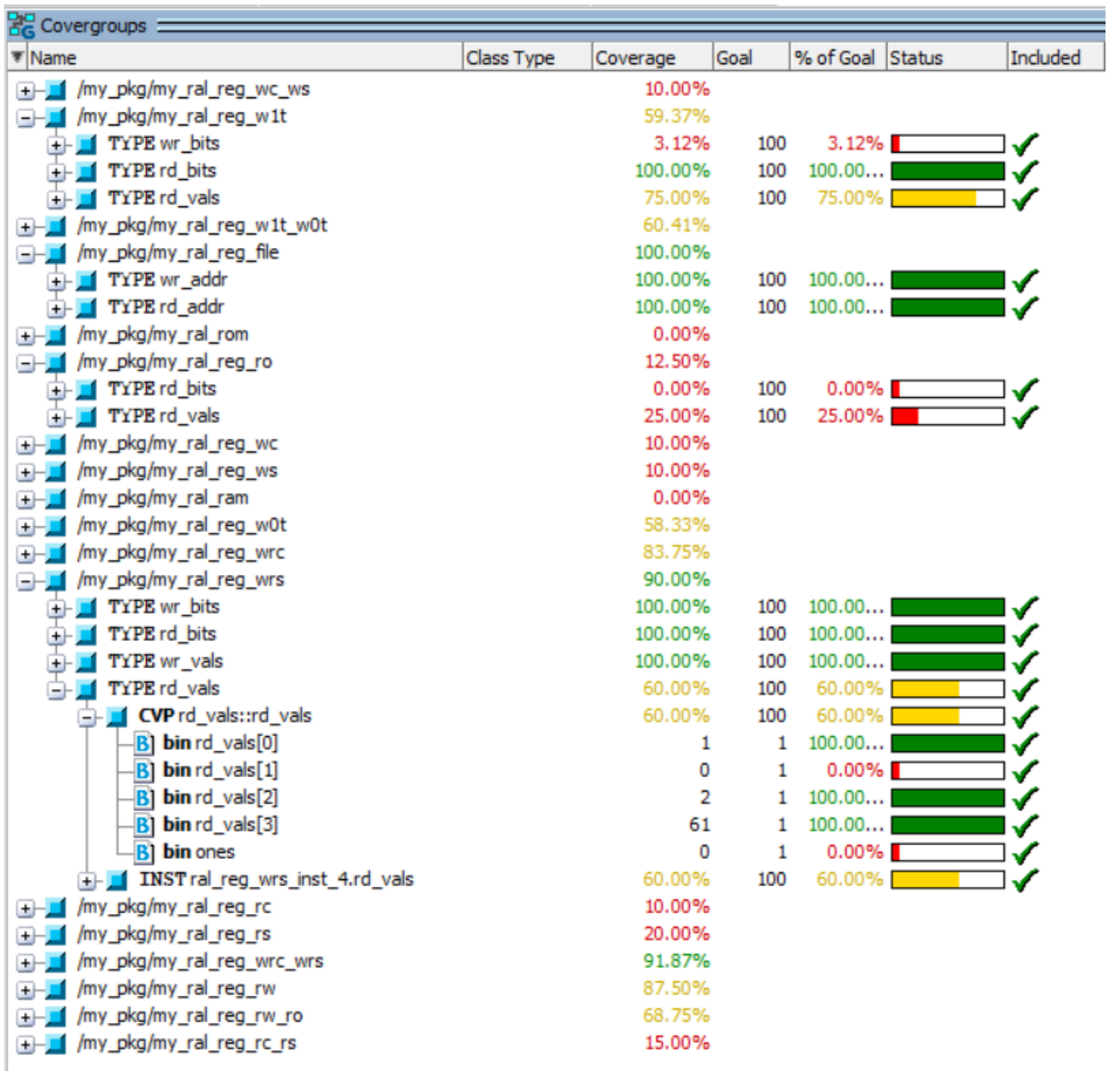
This sequence only reads registers and nothing more. We can see 5.4 that all registers were actually read at least once and that they all have their first bin (bin with zero value) sampled. If



Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included
/my_pkg/my_ral_reg_wc_ws		10.00%				
/my_pkg/my_ral_reg_w1t		8.33%				
TYPE wr_bits		0.00%	100	0.00%	🟥	✓
TYPE rd_bits		0.00%	100	0.00%	🟥	✓
TYPE rd_vals		25.00%	100	25.00%	🟥	✓
CVP rd_vals::rd_vals		25.00%	100	25.00%	🟥	✓
bin rd_vals[0]		1	1	100.00...	🟩	✓
bin rd_vals[1]		0	1	0.00%	🟥	✓
bin rd_vals[2]		0	1	0.00%	🟥	✓
bin rd_vals[3]		0	1	0.00%	🟥	✓
INST ral_reg_w1t_inst_7.rd_vals		25.00%	100	25.00%	🟥	✓
/my_pkg/my_ral_reg_w1t_w0t		8.33%				
/my_pkg/my_ral_reg_file		50.00%				
TYPE wr_addr		0.00%	100	0.00%	🟥	✓
TYPE rd_addr		100.00%	100	100.00...	🟩	✓
/my_pkg/my_ral_rom		0.00%				
/my_pkg/my_ral_reg_ro		12.50%				
/my_pkg/my_ral_reg_wc		10.00%				
/my_pkg/my_ral_reg_ws		10.00%				
/my_pkg/my_ral_ram		0.00%				
/my_pkg/my_ral_reg_w0t		8.33%				
/my_pkg/my_ral_reg_wrc		5.00%				
/my_pkg/my_ral_reg_wrs		5.00%				
/my_pkg/my_ral_reg_rc		10.00%				
/my_pkg/my_ral_reg_rs		10.00%				
/my_pkg/my_ral_reg_wrc_wrs		5.00%				
/my_pkg/my_ral_reg_rw		6.25%				
/my_pkg/my_ral_reg_rw_ro		6.25%				
/my_pkg/my_ral_reg_rc_rs		10.00%				

Figure 5.4 Collected coverage for uvm_reg_hw_reset_seq sequence



Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included
/my_pkg/my_ral_reg_wc_ws		10.00%				
/my_pkg/my_ral_reg_w1t		59.37%				
TYPE wr_bits		3.12%	100	3.12%	🟥	✓
TYPE rd_bits		100.00%	100	100.00...	🟩	✓
TYPE rd_vals		75.00%	100	75.00%	🟨	✓
/my_pkg/my_ral_reg_w1t_w0t		60.41%				
/my_pkg/my_ral_reg_file		100.00%				
TYPE wr_addr		100.00%	100	100.00...	🟩	✓
TYPE rd_addr		100.00%	100	100.00...	🟩	✓
/my_pkg/my_ral_rom		0.00%				
/my_pkg/my_ral_reg_ro		12.50%				
TYPE rd_bits		0.00%	100	0.00%	🟥	✓
TYPE rd_vals		25.00%	100	25.00%	🟥	✓
/my_pkg/my_ral_reg_wc		10.00%				
/my_pkg/my_ral_reg_ws		10.00%				
/my_pkg/my_ral_ram		0.00%				
/my_pkg/my_ral_reg_w0t		58.33%				
/my_pkg/my_ral_reg_wrc		83.75%				
/my_pkg/my_ral_reg_wrs		90.00%				
TYPE wr_bits		100.00%	100	100.00...	🟩	✓
TYPE rd_bits		100.00%	100	100.00...	🟩	✓
TYPE wr_vals		100.00%	100	100.00...	🟩	✓
TYPE rd_vals		60.00%	100	60.00%	🟨	✓
CVP rd_vals::rd_vals		60.00%	100	60.00%	🟨	✓
bin rd_vals[0]		1	1	100.00...	🟩	✓
bin rd_vals[1]		0	1	0.00%	🟥	✓
bin rd_vals[2]		2	1	100.00...	🟩	✓
bin rd_vals[3]		61	1	100.00...	🟩	✓
bin ones		0	1	0.00%	🟥	✓
INST ral_reg_wrs_inst_4.rd_vals		60.00%	100	60.00%	🟨	✓
/my_pkg/my_ral_reg_rc		10.00%				
/my_pkg/my_ral_reg_rs		20.00%				
/my_pkg/my_ral_reg_wrc_wrs		91.87%				
/my_pkg/my_ral_reg_rw		87.50%				
/my_pkg/my_ral_reg_rw_ro		68.75%				
/my_pkg/my_ral_reg_rc_rs		15.00%				

Figure 5.5 Collected coverage for uvm_reg_bit_bash_seq sequence

`TWO_MAPS` macro is defined, coverage collection percentage is a little higher, because registers are read twice (through both maps) and after the first read operation, registers of type write-read-set and read-set have their values changed to all ones.

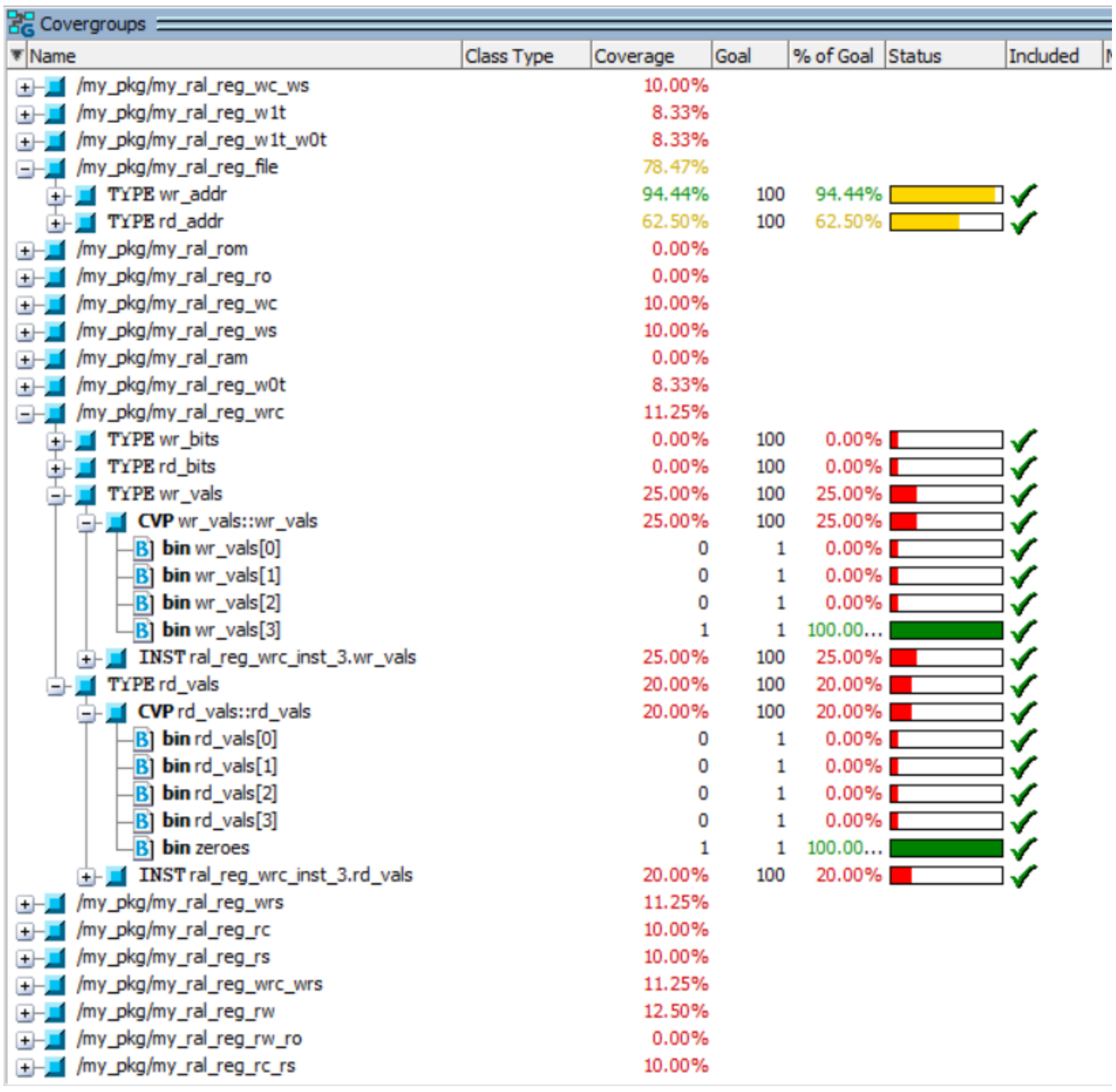
5.3.2.2 uvm_reg_bit_bash_seq

While looking at the collected coverage 5.5 we can see that in case of write-1/0-toggle registers, sequence does not actually write one-hot and one-cold values to them, but such a value, so that after write operations registers would contain one-hot (W1T) and one-cold (W0T) values.

Registers, that can be only be read, and registers of type write-clear and write-set have such a low coverage percentage because in order to sample their read values, data needs to be poked into them via backdoor, which this sequence does not do.

In case of write-read-clear and write-read-set registers, we can see, that they are never read twice in a row, i.e. all ones (WRS) or all zeroes (WRC) values are not sampled.

In case of registers that are written with one-hot (RW and WRC) and one-cold (WRS) values we can see that their write and read bins for field values are not at 100%. That happened because



Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included
/my_pkg/my_ral_reg_wc_ws		10.00%				
/my_pkg/my_ral_reg_w1t		8.33%				
/my_pkg/my_ral_reg_w1t_w0t		8.33%				
/my_pkg/my_ral_reg_file		78.47%				
TYPE wr_addr		94.44%	100	94.44%	🟨	✓
TYPE rd_addr		62.50%	100	62.50%	🟨	✓
/my_pkg/my_ral_rom		0.00%				
/my_pkg/my_ral_reg_ro		0.00%				
/my_pkg/my_ral_reg_wc		10.00%				
/my_pkg/my_ral_reg_ws		10.00%				
/my_pkg/my_ral_ram		0.00%				
/my_pkg/my_ral_reg_w0t		8.33%				
/my_pkg/my_ral_reg_wrc		11.25%				
TYPE wr_bits		0.00%	100	0.00%	🟥	✓
TYPE rd_bits		0.00%	100	0.00%	🟥	✓
TYPE wr_vals		25.00%	100	25.00%	🟥	✓
CVP wr_vals::wr_vals		25.00%	100	25.00%	🟥	✓
bin wr_vals[0]		0	1	0.00%	🟥	✓
bin wr_vals[1]		0	1	0.00%	🟥	✓
bin wr_vals[2]		0	1	0.00%	🟥	✓
bin wr_vals[3]		1	1	100.00...	🟩	✓
INST ral_reg_wrc_inst_3.wr_vals		25.00%	100	25.00%	🟥	✓
TYPE rd_vals		20.00%	100	20.00%	🟥	✓
CVP rd_vals::rd_vals		20.00%	100	20.00%	🟥	✓
bin rd_vals[0]		0	1	0.00%	🟥	✓
bin rd_vals[1]		0	1	0.00%	🟥	✓
bin rd_vals[2]		0	1	0.00%	🟥	✓
bin rd_vals[3]		0	1	0.00%	🟥	✓
bin zeroes		1	1	100.00...	🟩	✓
INST ral_reg_wrc_inst_3.rd_vals		20.00%	100	20.00%	🟥	✓
/my_pkg/my_ral_reg_wrs		11.25%				
/my_pkg/my_ral_reg_rc		10.00%				
/my_pkg/my_ral_reg_rs		10.00%				
/my_pkg/my_ral_reg_wrc_wrs		11.25%				
/my_pkg/my_ral_reg_rw		12.50%				
/my_pkg/my_ral_reg_rw_ro		0.00%				
/my_pkg/my_ral_reg_rc_rs		10.00%				

Figure 5.6 Collected coverage for uvm_reg_access_seq sequence

one-hot and one-cold values do not cover all possible values, which are divided into four cover bins.

All registers were written or read at least once.

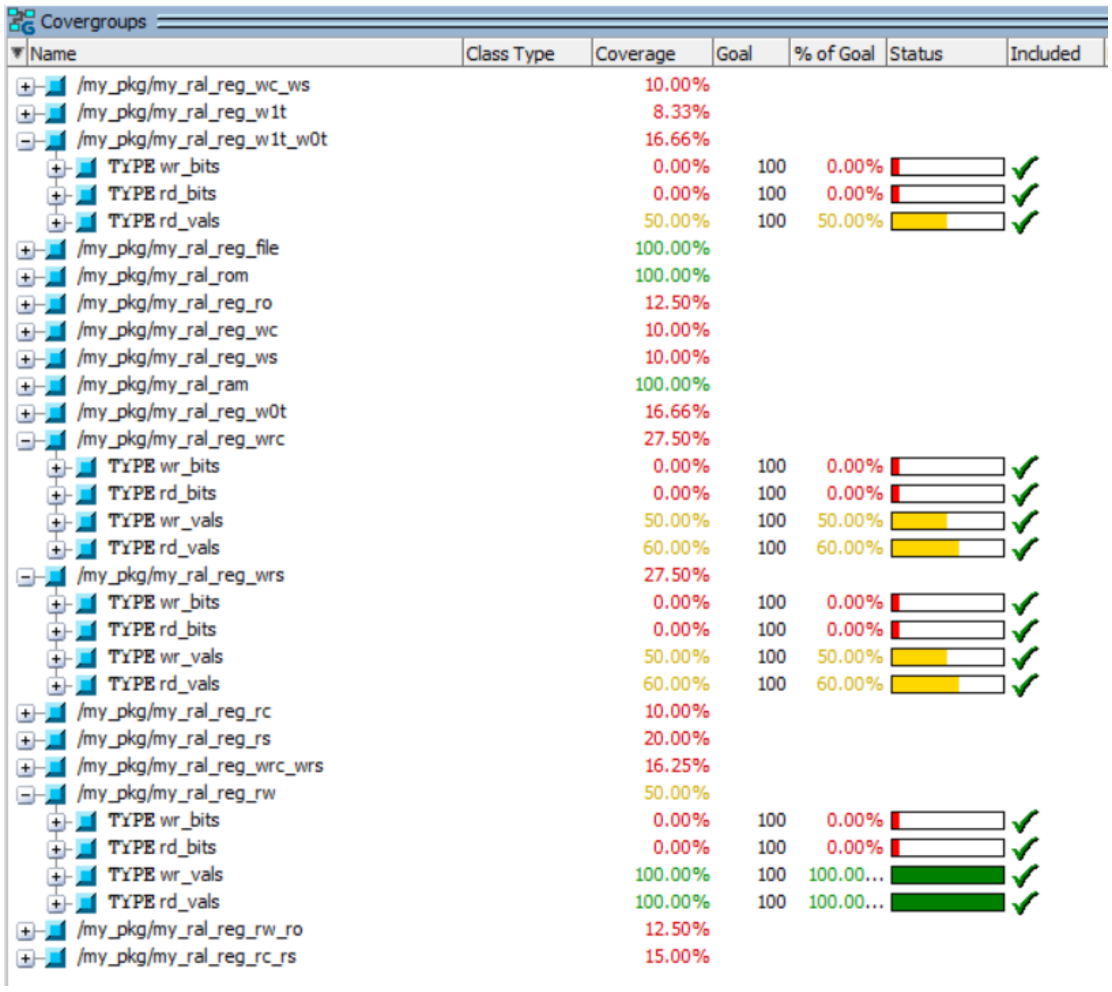
With `TWO_MAPS` defined resulting coverage is identical.

5.3.2.3 uvm_reg_access_seq

By looking at the coverage 5.6 we can see that only written inverted reset values (all ones) and read reset values (all zeroes) are sampled. Values that were written via backdoor are not sampled.

Registers containing read-only fields are not tested at all (RW/RO register in cover group `wr_addr`, RW/RO and all of RO registers in cover group `rd_addr`).

If executed with `TWO_MAPS` macro, coverage of some of the registers were increased by exactly two times. That is to be expected, since this sequence reads and writes values via all currently available address maps.



Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included
/my_pkg/my_ral_reg_wc_ws		10.00%				
/my_pkg/my_ral_reg_w1t		8.33%				
/my_pkg/my_ral_reg_w1t_w0t		16.66%				
TYPE wr_bits		0.00%	100	0.00%	🟥	✓
TYPE rd_bits		0.00%	100	0.00%	🟥	✓
TYPE rd_vals		50.00%	100	50.00%	🟨	✓
/my_pkg/my_ral_reg_file		100.00%				
/my_pkg/my_ral_rom		100.00%				
/my_pkg/my_ral_reg_ro		12.50%				
/my_pkg/my_ral_reg_wc		10.00%				
/my_pkg/my_ral_reg_ws		10.00%				
/my_pkg/my_ral_ram		100.00%				
/my_pkg/my_ral_reg_w0t		16.66%				
/my_pkg/my_ral_reg_wrc		27.50%				
TYPE wr_bits		0.00%	100	0.00%	🟥	✓
TYPE rd_bits		0.00%	100	0.00%	🟥	✓
TYPE wr_vals		50.00%	100	50.00%	🟨	✓
TYPE rd_vals		60.00%	100	60.00%	🟨	✓
/my_pkg/my_ral_reg_wrs		27.50%				
TYPE wr_bits		0.00%	100	0.00%	🟥	✓
TYPE rd_bits		0.00%	100	0.00%	🟥	✓
TYPE wr_vals		50.00%	100	50.00%	🟨	✓
TYPE rd_vals		60.00%	100	60.00%	🟨	✓
/my_pkg/my_ral_reg_rc		10.00%				
/my_pkg/my_ral_reg_rs		20.00%				
/my_pkg/my_ral_reg_wrc_wrs		16.25%				
/my_pkg/my_ral_reg_rw		50.00%				
TYPE wr_bits		0.00%	100	0.00%	🟥	✓
TYPE rd_bits		0.00%	100	0.00%	🟥	✓
TYPE wr_vals		100.00%	100	100.00...	🟩	✓
TYPE rd_vals		100.00%	100	100.00...	🟩	✓
/my_pkg/my_ral_reg_rw_ro		12.50%				
/my_pkg/my_ral_reg_rc_rs		15.00%				

Figure 5.7 Collected coverage for uvm_reg_mem_shared_access_seq sequence

5.3.2.4 uvm_mem_walk_seq

If executed with or without `TWO_MAPS` macro, coverage of this sequence is at 100% for RAM, i.e. every memory word was written and read at least once. Coverage for all registers is at 0%.

5.3.2.5 uvm_mem_access_seq

If executed with or without `TWO_MAPS` macro, coverage of this sequence is at 100% for RAM and ROM, i.e. every memory word in RAM was written and read at least once and every memory word in ROM was read at least once. Coverage for all registers is at 0%.

5.3.2.6 uvm_reg_mem_shared_access_seq

If sequence is executed with only one address map, this sequence does nothing and no coverage is collected. When executed with two address maps we can see 5.7 that no bin that covers individual register bits is sampled, since this sequence only writes and reads random values via all possible address maps.

All registers, RAM and ROM memory words were written and read at least once.

5.3.2.7 uvm_reg_mem_hdl_paths_seq

This sequence checks if specified backdoor HDL paths are accessible by the simulator and does not interact with an actual DUT blocks.

5.4 Execution of my_full_test

Since this test contains same sequences as in the previous two tests, describing the execution of this test in detail is redundant.

All inserted bugs were identified and collected coverage is at 100% (if all coverage models are enabled and sampling is enabled on all of them).

5.5 Execution of uvm_reg_mem_built_in_seq sequence

This sequence is used to execute a user-defined selection of built-in register block sequences. This sequence is not a part of any of our tests, but it was executed individually to check, if it will successfully start defined sequences.

If it is executed with only one sequence, that tests registers, or with only sequences, that test memories, `uvm_reg_mem_built_in_seq` sequence starts them without any problems. But if it executes at least two sequences, that test registers, unexpected errors occur. For example, if we have no inserted errors, one or two address maps, and if we set this sequence to execute sequences, that check reset values and do bit-bashing:

```
UVM_DO_REG_HW_RESET + UVM_DO_REG_BIT_BASH
```

we get these two errors:

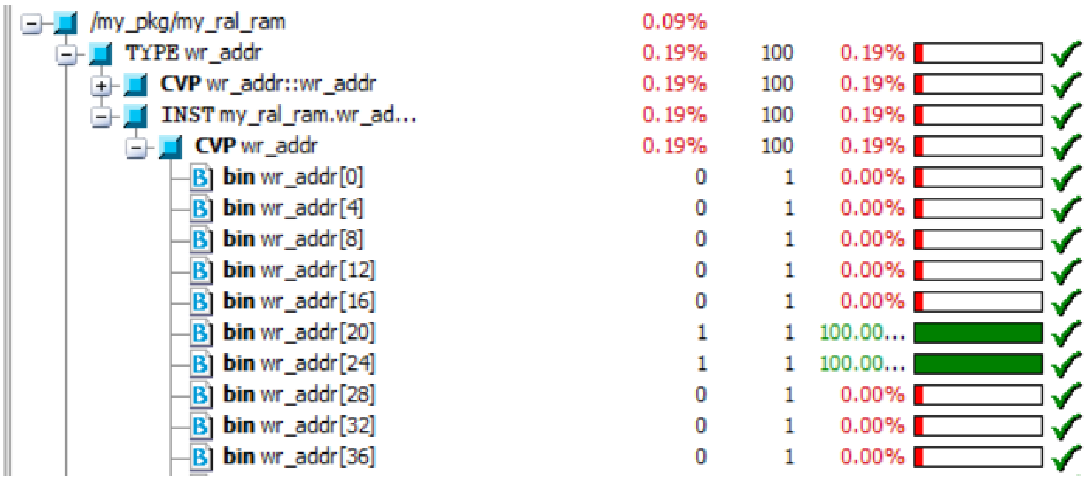
```
# UVM_ERROR verilog_src/uvm-1.1d/src/reg/sequences/
uvm_reg_bit_bash_seq.svh(174) @ 16760.00 ns:
reporter@@uvm_reg_mem_built_in_seq_inst.reg_bit_bash_seq.
reg_single_bit_bash_seq [uvm_reg_bit_bash_seq] Writing a 1 in bit #0 of
register "ral_reg_block_inst.ral_reg_file_inst.ral_reg_rc_rs_inst_18"
with initial value 'h0000000000000000 yielded 'h000000000000ffff instead of
'h0000000000000000

# UVM_ERROR verilog_src/uvm-1.1d/src/reg/sequences/
uvm_reg_bit_bash_seq.svh(174) @ 62840.00 ns:
reporter@@uvm_reg_mem_built_in_seq_inst.reg_bit_bash_seq.
reg_single_bit_bash_seq [uvm_reg_bit_bash_seq] Writing a 1 in bit #0 of
register "ral_reg_block_inst.ral_reg_file_inst.ral_reg_rs_inst_2"
with initial value 'h0000000000000000 yielded 'h00000000ffffffff instead of
'h0000000000000000
```

That happened, because values of these registers were set after read operation invoked by `uvm_reg_hw_reset_seq`.

After examining source code for `uvm_reg_mem_built_in_seq` sequence [37], it was discovered, that there is no reset between any of the sequences in it, which is in contradiction to what is specified in the source codes of register block built-in sequences (for example [31]). Each one of them has a task called `reset_blk()` which is used to reset DUT, but it is empty by default. Comments of this task state, that “DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT”.

Since `uvm_reg_mem_built_in_seq` uses UVM factory to `create()` instances of built-in sequences, we can:



/my_pkg/my_ral_ram	0.09%			
TYPE wr_addr	0.19%	100	0.19%	✓
CVP wr_addr::wr_addr	0.19%	100	0.19%	✓
INST my_ral_ram.wr_ad...	0.19%	100	0.19%	✓
CVP wr_addr	0.19%	100	0.19%	✓
bin wr_addr[0]	0	1	0.00%	✓
bin wr_addr[4]	0	1	0.00%	✓
bin wr_addr[8]	0	1	0.00%	✓
bin wr_addr[12]	0	1	0.00%	✓
bin wr_addr[16]	0	1	0.00%	✓
bin wr_addr[20]	1	1	100.00...	✓
bin wr_addr[24]	1	1	100.00...	✓
bin wr_addr[28]	0	1	0.00%	✓
bin wr_addr[32]	0	1	0.00%	✓
bin wr_addr[36]	0	1	0.00%	✓

Figure 5.8 Collected coverage after writing to RAM memory word with offset 5

1. Create new classes that will be extended from the built-in sequences.
2. Implement their `reset_blk()` task to reset DUT.
3. Override built-in sequences with our classes using UVM factory `set_inst_override()` or `set_type_override()`.

After that we can successfully execute `uvm_reg_mem_built_in_seq` with all built-in sequences.

But this process can be quite tedious, since it is easier to simply create new UVM test, where we would reset DUT manually after execution of every built-in sequence. That is exactly what our `my_built_in_sequences_test` test does.

5.6 Potential memory sampling bug

While examining coverage collection for memories a potential memory sampling bug in UVM RAL library was discovered.

Let's say we want to write some value to RAM memory word with offset 5 (while all coverage models are enabled and sampling is enabled on all of them):

```
env_inst.ral_env_inst.ral_reg_block_inst.ral_ram_inst.write(
  status, 5, 32'hFFFF_FFFF, UVM_FRONTDOOR
);
```

If we look at the collected coverage after this operation 5.8, we can see that memory word with offset 6 was also sampled. If we then write to words with offsets 5 and 6 – memory words with offsets 5, 6 and 7 are sampled. Same thing occurs if we `read()`, `burst_write()` or `burst_read()` data from/to memory.

After looking at the source code for UVM RAL memory (`uvm_mem.svh` [41]) we can see that `write()` method calls `do_write()` method:

```
task uvm_mem::write(output uvm_status_e      status,
                    input  uvm_reg_addr_t    offset,
                    input  uvm_reg_data_t    value,
                    input  uvm_path_e        path = UVM_DEFAULT_PATH,
                    input  uvm_reg_map       map = null,
                     input  uvm_sequence_base parent = null,
                     input  int               prior = -1,
                     input  uvm_object        extension = null,
                     input  string            fname = "",
                     input  int               lineno = 0);

  // create an abstract transaction for this operation
  uvm_reg_item rw = uvm_reg_item::type_id::create(
    "mem_write",,get_full_name()
  );
  rw.element      = this;
  rw.element_kind = UVM_MEM;
  rw.kind         = UVM_WRITE;
  rw.offset       = offset;
  rw.value[0]     = value;
  rw.path         = path;
  rw.map          = map;
  rw.parent       = parent;
  rw.prior        = prior;
  rw.extension    = extension;
  rw.fname        = fname;
  rw.lineno       = lineno;

  do_write(rw);

  status = rw.status;

endtask: write
```

Method do_write(), in turn, calls XsampleX():

```
task uvm_mem::do_write(uvm_reg_item rw);
  // ...
  // FRONTDOOR
  if (rw.path == UVM_FRONTDOOR) begin
    // ...
    if (rw.status != UVM_NOT_OK)
      for (int idx = rw.offset;
           idx <= rw.offset + rw.value.size();
           idx++) begin
        XsampleX(map_info.mem_range.stride * idx, 0, rw.map);
        m_parent.XsampleX(map_info.offset +
                          (map_info.mem_range.stride * idx),
                          0, rw.map);
      end
  end
  // ...
endtask: do_read
```

And XsampleX() calls sample() method, that is implemented in our RAL memories:

```
/*local*/ function void XsampleX(uvm_reg_addr_t addr,
                                 bit            is_read,
                                 uvm_reg_map    map);
  sample(addr, is_read, map);
endfunction
```

Table 5.2 Basic information about UVM RAL memory methods

RAL Memory method	**Triggers sampling?**	**Respects DUT memory access policy?**
Frontdoor write()	Yes	Yes
Backdoor write()	No	Yes
Frontdoor read()	Yes	Yes
Backdoor read()	No	Yes
Frontdoor burst_write()	Yes	Yes
Backdoor burst_write()	No	Yes
Frontdoor burst_read()	Yes	Yes
Backdoor burst_read()	No	Yes
poke()	No	No
peek()	No	No

When we write to memory word with offset 5, `rw.offset` is set to 5, `rw.value.size()` is 1, and that is why `XsampleX()` method is executed twice – with `idx` having values 5 and 6. Methods `read()`, `burst_write()` and `burst_read()` have the exact same issue.

A post was created on a Siemens forum about UVM methodology, where this problem was described, but there were no answers provided yet.

5.7 Simulation Results

All tests were successfully executed and all bugs were identified with the exception of test consisting of built-in RAL sequences with one address map. Coverage collection was correctly implemented – after every read and write operation on a register or memory word, data are properly sampled. Functional coverage models and their sampling can be turned on and off and corresponding cover groups are created and sampled as expected.

Since it can be difficult to understand which RAL methods respect access policy of a register, which of them actually modify desired and mirrored values and which of them trigger sampling, tables describing these methods were created (5.3 and 5.2). Information provided in these tables was gathered based on the results of execution of our tests and by executing these RAL methods individually.

Table 5.3 Basic information about UVM RAL register methods

RAL Register method	**Modifies desired value?**	**Modifies mirrored value?**	**Triggers sampling?**	**Respects DUT register access policy?**
set()	Only if register access policy allows it	No	No	Does not interact with DUT registers
reset()	Yes	Yes	No	Does not interact with DUT registers
Frontdoor write()	Yes	Only if implicit/explicit prediction is set up		Yes
Backdoor write()	Yes	Yes	No	Yes
Frontdoor read()	Only if implicit/explicit prediction is set up			Yes
Backdoor read()	Yes	Yes	No	Yes
poke()	Yes	Yes	No	No
peek()	Yes	Yes	No	No
Frontdoor update()	No	Only if implicit/explicit prediction is set up		Yes (see set() method)
Backdoor update()	No	Yes	No	Yes (see set() method)
Frontdoor mirror()	Only if implicit/explicit prediction is set up			Yes
Backdoor mirror()	Yes	Yes	No	Yes
Frontdoor predict() with UVM_PREDICT_WRITE or UVM_PREDICT_READ	Only if register access policy allows it		No	Does not interact with DUT registers
Frontdoor predict() with UVM_PREDICT_DIRECT	Yes	Yes	No	Does not interact with DUT registers
Backdoor predict()	Yes	Yes	No	Does not interact with DUT registers

Conclusion

The goal of this thesis was to study the constructs of UVM and its register abstraction layer (RAL). Another purpose of this work was to present, how RAL can be used for automatic coverage collection and how its built-in sequences can be utilized to verify functionality of registers and memories.

As a result of this thesis, testbench for a block of registers and RAM and ROM memories was successfully created. This testbench utilizes UVM RAL components and methods and contains descriptive comments as to why and how they can be configured and used for user's needs.

Number of sequences and tests, that showcase the use of RAL methods for verification, were also created. Each of the built-in RAL sequence was described and used in our testbench. By artificially inserting bugs in our design, we proved that our and built-in sequences are able to identify them.

Automatic coverage collection was implemented for every register and memory. Functional coverage models can be configured for desired register block/register/memory as we see fit. By studying the collected coverage after each test and built-in RAL sequence, we made sure, that coverage was configured and collected properly.

Couple of potential bugs in UVM register abstraction layer were also identified: aggregated sequence can execute sequences, even if their disable attributes are set 3.2.5.3, built-in RAL sequences are not able to detect that WRS and WRC are not set/cleared after read operations 5.3.1, and that one more memory word is sampled every time memory is read or written 5.6.

To summarize, this thesis and corresponding source codes can be used to showcase, how the UVM RAL testbench can be created and configured, and how it can be used to verify functionalities of registers and memories in digital design.

Bibliography

1. GANEEV, Timur. *Master thesis - UVM RAL* [online]. 2024. [visited on 2024-05-09]. Available from: `https://gitlab.fit.cvut.cz/ganeetim/master_thesis_uvm_ral`.
2. *ChipVerify: UVM Register Abstraction Layer* [online]. 2023. [visited on 2024-03-15]. Available from: `https://www.chipverify.com/uvm/uvm-register-layer`.
3. *Verification Guide: Introduction to UVM RAL* [online]. 2024. [visited on 2024-03-15]. Available from: `https://verificationguide.com/uvm-ral/introduction-to-uvm-ral/`.
4. *VLSI Verify: Register Abstraction Layer (RAL) Model* [online]. 2024. [visited on 2024-03-16]. Available from: `https://vlsiverify.com/uvm/ral/ral-model/`.
5. JÍLEK, Vojtěch. *Simulace procesorů v jazyce SystemVerilog*. Prague, 2022. Available also from: `http://hdl.handle.net/10467/101053`. Master's thesis. Czech technical university in Prague, Faculty of Information Technology, Department of Digital Design. Supervisor: Martin Kohlík.
6. JÍLEK, Vojtěch. *SCE - Simulace procesoru* [online]. 2022. [visited on 2024-03-16]. Available from: `https://gitlab.fit.cvut.cz/jilekvoj/sce---simulace-procesoru`.
7. *ChipVerify: UVM Tutorial* [online]. 2023. [visited on 2024-03-17]. Available from: `https://www.chipverify.com/tutorials/uvm`.
8. KOHLÍK, Martin. *Digital Circuit Simulation and Verification: Universal Verification Methodology* [online]. 2021. [visited on 2024-03-17]. Available from: `https://courses.fit.cvut.cz/NI-SIM/media/lectures/11-UVM.pdf`. [File is accessible after logging in to CTU network].
9. *VLSI Verify: UVM testbench Top* [online]. 2024. [visited on 2024-03-17]. Available from: `https://vlsiverify.com/uvm/uvm-testbench-top/`.
10. *VLSI Verify: UVM Driver* [online]. 2024. [visited on 2024-03-21]. Available from: `https://vlsiverify.com/uvm/uvm-driver/`.
11. *ChipVerify: UVM Monitor [uvm_monitor]* [online]. 2023. [visited on 2024-03-21]. Available from: `https://www.chipverify.com/uvm/uvm-monitor`.
12. *ChipVerify: UVM Agent — uvm_agent* [online]. 2023. [visited on 2024-03-21]. Available from: `https://www.chipverify.com/uvm/uvm-agent`.
13. *ChipVerify: UVM Scoreboard* [online]. 2023. [visited on 2024-03-22]. Available from: `https://www.chipverify.com/uvm/uvm-scoreboard`.
14. *ChipVerify: UVM Phases* [online]. 2023. [visited on 2024-03-22]. Available from: `https://www.chipverify.com/uvm/uvm-phases`.

15. *VLSI Verify: UVM Phases* [online]. 2024. [visited on 2024-03-23]. Available from: https://vlsiverify.com/uvm/uvm-phases/.
16. *Verification Academy: UVM Common Phases* [online]. 2024. [visited on 2024-03-23]. Available from: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/base/uvm_common_phases-svh.html#uvm_build_phase.
17. *Verification Academy: UVM Run-Time Phases* [online]. 2024. [visited on 2024-03-23]. Available from: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/base/uvm_runtime_phases-svh.html.
18. *ChipVerify: UVM Register Model Classes* [online]. 2023. [visited on 2024-03-28]. Available from: https://www.chipverify.com/uvm/uvm-register-model.
19. *ChipVerify: UVM Register Backdoor Access* [online]. 2023. [visited on 2024-03-31]. Available from: https://www.chipverify.com/uvm/uvm-register-backdoor-access.
20. *Verification Academy: uvm_reg* [online]. 2024. [visited on 2024-03-28]. Available from: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1b/html/files/reg/uvm_reg-svh.html.
21. *Verification Academy: uvm_reg_field* [online]. 2024. [visited on 2024-03-29]. Available from: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1a/html/files/reg/uvm_reg_field-svh.html.
22. *Verification Academy: uvm_reg_file* [online]. 2024. [visited on 2024-03-31]. Available from: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1b/html/files/reg/uvm_reg_file-svh.html.
23. *Verification Academy: uvm_mem* [online]. 2024. [visited on 2024-03-31]. Available from: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1b/html/files/reg/uvm_mem-svh.html.
24. *Verification Academy: uvm_reg_map* [online]. 2024. [visited on 2024-03-31]. Available from: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1b/html/files/reg/uvm_reg_map-svh.html.
25. *Verification Academy: Classes for Adapting Between Register and Bus Operations* [online]. 2024. [visited on 2024-04-01]. Available from: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/reg/uvm_reg_adapter-svh.html.
26. *Verification Academy: uvm_reg_block* [online]. 2024. [visited on 2024-04-01]. Available from: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/reg/uvm_reg_block-svh.html.
27. *VLSI Verify: RAL Predictor* [online]. 2024. [visited on 2024-04-01]. Available from: https://vlsiverify.com/uvm/ral/ral-predictor/.
28. *ChipVerify: UVM Register Environment* [online]. 2023. [visited on 2024-04-01]. Available from: https://www.chipverify.com/uvm/uvm-register-environment.
29. *Verification Academy: Global Declarations for the Register Layer* [online]. 2024. [visited on 2024-04-01]. Available from: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/reg/uvm_reg_model-svh.html.
30. *Verification Academy: uvm_reg_hw_reset_seq* [online]. 2024. [visited on 2024-04-05]. Available from: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/reg/sequences/uvm_reg_hw_reset_seq-svh.html.
31. *Verification Academy: Bit Bashing Test Sequences* [online]. 2024. [visited on 2024-04-05]. Available from: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1a/html/files/reg/sequences/uvm_reg_bit_bash_seq-svh.html.

32. *Verification Academy: Register Access Test Sequences* [online]. 2024. [visited on 2024-04-05]. Available from: `https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/reg/sequences/uvm_reg_access_seq-svh.html`.

33. *Verification Academy: Shared Register and Memory Access Test Sequences* [online]. 2024. [visited on 2024-04-05]. Available from: `https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/reg/sequences/uvm_reg_mem_shared_access_seq-svh.html`.

34. *Verification Academy: Memory Walking-Ones Test Sequences* [online]. 2024. [visited on 2024-04-05]. Available from: `https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/reg/sequences/uvm_mem_walk_seq-svh.html`.

35. *Verification Academy: Memory Access Test Sequence* [online]. 2024. [visited on 2024-04-05]. Available from: `https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/reg/sequences/uvm_mem_access_seq-svh.html`.

36. *Verification Academy: uvm_reg_mem_built_in_seq* [online]. 2024. [visited on 2024-04-05]. Available from: `https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/reg/sequences/uvm_reg_mem_built_in_seq-svh.html`.

37. *Verification Academy: uvm_reg_mem_built_in_seq.svh* [online]. 2024. [visited on 2024-04-07]. Available from: `https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/src/reg/sequences/uvm_reg_mem_built_in_seq.svh`.

38. *Verification Academy: uvm_reg_mem_shared_access_seq.svh* [online]. 2024. [visited on 2024-04-07]. Available from: `https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/src/reg/sequences/uvm_reg_mem_shared_access_seq.svh`.

39. *Verification Academy: uvm_mem_walk_seq.svh* [online]. 2024. [visited on 2024-04-07]. Available from: `https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/src/reg/sequences/uvm_mem_walk_seq.svh`.

40. *Verification Academy: HDL Paths Checking Test Sequence* [online]. 2024. [visited on 2024-04-05]. Available from: `https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/reg/sequences/uvm_reg_mem_hdl_paths_seq-svh.html`.

41. *Verification Academy: uvm_mem.svh* [online]. 2024. [visited on 2024-05-09]. Available from: `https://verificationacademy.com/verification-methodology-reference/uvm/src/reg/uvm_mem.svh`.

Contents of the attachment

```
readme.txt ........................................ brief description of the attachment
src
  Masters_Thesis_UVM_RAL ........................... source codes (GitLab repository)
  Masters_Thesis_UVM_RAL_Latex ............. source code of the thesis in LaTeX format
text ............................................................... text of the thesis
  Masters_Thesis_UVM_RAL.pdf ................................... thesis in PDF format
```