Master RADMEP

Master Radiation and its Effects on MicroElectronics and Photonics Technologies (RADMEP)

Template Code Generator for Design Verification based on Universal Verification Methodology

Master Thesis Report

Presented by

Theresa Kahale

And defended at

University Jean Monnet

11-12 September 2023

Academic Supervisor: Dr. Arto Javanainen, University of Jyväskylä

Host Company: Apple Technology Services B.V. & Co. KG

Jury Committee:

Dr. Arto Javanainen, University of Jyväskylä

Prof. Sylvain Girard, University Jean Monnet

Prof. Dr. Ir. Paul Leroux, Katholieke Universiteit Leuven

Prof. Frédéric Saigné, University of Montpellier

Template Code Generator for Design Verification based on Universal Verification Methodology

Abstract

In recent years, the semiconductor industry has been showing advanced growth driven by the increasing demand for electronic devices such as smartphones, laptops, tablets and other consumer electronics. Given their extensive applications, the need for higher performance and efficiency requirements has led to smart and innovative development of complex designs, resulting in a more challenging verification process. With the highest workload (around 70%) on verification, adopting tools and methodologies such as Universal Verification Methodology (UVM) is critical to enhance the quality of the design and increase the time-to-market with no defects. UVM is a SystemVerilog based architecture that provides a library to establish robust verification environments. It enables the use of customizable modular and reusable components and testbenches. To reduce the verification time and effort, this master thesis proposes a UVM code generator which instantiates the essential infrastructure for UVM verification components and creates the necessary directory structure for the codes and the files. Given the design input and UVM template files, the code generator will compose the building blocks such as the interfaces and UVM verification components (UVC), environments and testbenches that will connect the Design Under Test (DUT) to the UVCs. The UVM standard structure and proper encapsulation will be adopted to allow for flexible modification and reusability in top level verification environments. To ensure its correct functionality, the tool will be tested on a simple AHB SRAM Controller that manages access to a single port SRAM interface. Constrained random tests will be provided including the necessary checks to get full coverage on the design. Overall, the work on the template code generator enables the automation of block level verification using UVM by providing the verification engineers with the completed testbench and environment to test the DUT.

Keywords: UVM, template code generator, verification environment and testbenches, chip design.

Preface

Before you lies my master thesis titled “Template Code Generator for Design Verification based on Universal Verification Methodology”.

It was completed as a requirement for graduation from the Erasmus Mundus RADMEP program. During my 6-month internship spanning from February to August 2023, I had the privilege of working at Apple Inc. in Munich, Germany, a renowned global technological company. I served as a Design Verification (DV) Engineering Intern in the Power Management Unit (PMU) DV Team.

Under the guidance of an outstanding team, I gained exposure to numerous DV practices. I would like to express my sincere gratitude to my supervisor, the Design Verification engineering manager, for his unwavering support, understanding, and warm welcome into his group during the past few months. Being a part of his team has provided me with invaluable learning opportunities. Indeed, I was given several chances to step out of my comfort zone, allowing me to improve my soft skills and engage with experts within the company.

I would like to extend a heartfelt thanks to my buddy and DV engineer for his excellent guidance and consistent assistance in completing this work. I deeply appreciate his genuine contributions, flexibility, and willingness to help. During the internship, he shared his technical expertise with me and encouraged me to broaden my horizons, and for that, I am grateful.

I would also like to acknowledge the entire team I was working with. Thank you all for taking the time to address my questions and making this experience even more enjoyable. I appreciate our exchange of mindsets and our daily fun encounters.

A special thanks to my academic supervisor, Arto Javanainen, for checking up on my work, reviewing my progress, and providing valuable insights.

Working at Apple means being exposed to experienced engineers in the field and being challenged to grow as an individual. I am grateful for the support I have received in this journey from everyone at the company and from the partner universities.

Table of Contents

Table of Figures

Table of Tables

Introduction

The semiconductor industry encompasses substantial importance in the world resulting from its influence on today's modern life [1]. Increasing demand in the performance requirements of digital integrated circuits (ICs) has driven the industry to innovate and push the boundaries of electronic chips [2]. From consumer electronics, healthcare and biotechnology to communication networks, the growth trend of this technology with its miniaturization is nowhere to stop [3]. Undoubtably, the market is significantly valuable, estimated at US$440 billion in 2020, two times fold its worth 20 years ago, and it's foreseen to reach a trillion dollar in 2030 [4]. Following the pandemic breakout, the industry faced widespread shortage, strengthening its economic value [5]. In the list of traded products, semiconductors rank the fourth after crude oil, refined oil, and cars. However, as the basis of digital economy, the industry is constantly being challenged and must meet the market demands as they quickly evolve.

Apple Inc. is a U.S. company that provides computer and consumer electronics with a market cap of over 3 trillion USD [6]. It's the prime technological company with the highest revenue in the world. It is mostly famous for the development of the Macintosh in 1984 and the iPhone in 2007 presented by Steve Jobs, the co-founder and the former CEO of Apple. In the past year, the company dominated more than half the market share of smartphone sales in the U.S., progressively increasing to 52% in the 4th quarter of 2022 [7]. Apple products range from hardware to operating systems (OS) to services and accessories. Among the current hardware products are the iPhone, iPad, Mac, Apple Watch, AirTag. While for OS, the most popular ones are iOS, macOS and watchOS. Apple supports many software and services such as App Store, FaceTime, iTunes, Apple Fitness+, and accessories like the AirPods and chargers [8]. Nowadays, the company is at the forefront of unleashing new technologies, recently with Apple Vision Pro [9]. It's a Virtual Reality (VR) headset that enables spatial computing through interacting with applications, notifications, meetings, work, movies and more through lenses. The company perseveres in challenging itself as innovation, creativity, excellence, and confidentiality are core principles of its employees.

Given its importance, developing chips is a complex process that was formed 60 years ago. Despite the need for higher flexibility and performance, better power consumption and rapid chip production, the core steps in the chip development cycle have remained the same [10]. The process is initiated by stating the specifications and the chip architecture, then the Register Transfer Level (RTL) design is developed using hardware description languages (HDL), and later will be synthesized into gate netlist, allowing the physical layout displaying all connections on the actual chip to be built [11]. Meanwhile, the design must be verified to behave as desired, as it's crucial to ensure no faults exist before it is sent for manufacturing. Lastly, the actual chip will undergo post-silicon validation, testing and, in case any issue arises, debugging.

A critical step in the development cycle briefly described above is design verification (DV). It consists of around 60-70% of the workload and project resources while a significant part of the other processes is automated. In this phase, the engineer works on developing verification environment where the outcome of the product is checked against the design input. This step aims to confirm the correctness of the design ensuring it complies with the specified specs before production.

With the increasing design complexity, the burden is imposed on design verification engineers to develop a more robust approach [11]. Companies struggle mostly to meet the restricting time-to-market while also gaining confidence that the chip will pass all tests after production. A common motto adopted by DV teams states that “if it’s not verified, it will not work”. Not being 100% sure that the complete design space is exploited is an added obstacle as DV engineers might expect the presence of unaddressed bugs. With these challenges among others later described in subsection 2.2, the need to automate parts of the design verification work is desired and is worth all the efforts.

Many technologies have been developed to improve verifying the design, notably, Universal Verification Methodology (UVM) [12], SystemVerilog Assertions (SVA), Constrained Random Verification (CRV), Unified Power Format for Low Power (UPF) and many more [11]. As UVM is one of the most adopted methodologies, one way of speeding up the development phase is to benefit from UVM’s reusability and adopt generators that can automatically build up the infrastructure for the verification environment. There are few ones available commercially: UVMGen allows for the creation of individual verification components for simple blocks following UVM’s basic structure and is still at an early stage of development [13]. Another one is EasierUVM developed by Doulos [14]. This generator has more advanced features as it can create many agents in an environment and can take the pin list file as input for the interface generation. On GitHub repositories, few developers have committed their testbench template generator with a graphical user interface (GUI) that allows the collection of information regarding naming, signals, configurations [15] [16]. However, the generated structure and environment in these tools are to a certain extent rigid and theoretical, making them not likely to be adopted in the industry by engineers. In addition to that, the tools are not as developed to meet the demand of the DV team in big companies such as Apple.

Creating an effective environment from scratch is burdensome, hence the thesis presents a UVM-based template generator that creates the complete environment for block level verification. The aim is to omit the trivial steps the DV engineer must perform to initiate the testing. By layering the infrastructure, more time will be available for developing test cases, writing checkers, and verifying the functionality of the design. First, a simple DUT is adopted to verify the working principle of the generator, then a more thorough verification testbench is developed for a block design. Section 1 depicts the theoretical background and the detailed description of design verification concepts. In turn, Section 2 presents the approach adopted to build the generator, paving the way for Section 3 that describes more closely the implementation. Afterwards, Section 4 portrays use cases of the template generator where the verification of two design blocks of the SRAM controller is exercised.

Section 1: Background Information

1.1. Chip Design Cycle

Many teams are included in the design flow of a chip. The process is initiated by establishing the high-level architectural design and detailing the functional requirements which leads to the design of the specifications, including block diagrams and chip architecture [12]. The next step consists of implementing the logic functionality describing the structure of the device using HDL. The design is performed at a high level, allowing more abstraction, thus ease of implementation of complex blocks. The verification flow starts at the beginning of the project, and in some cases it continues even after the shipment of the design for manufacturing. More details about this step are explained in 2.2. Once a significant portion of the features have been verified, the design code undergoes synthesis with the defined constraints, and static timing analysis (STA) is performed to check against timing violations. If any, fixes are implemented, and the design is resynthesized until no more violations are detected. Later, the Standard Delay File (SDF) along with the gate netlist complying with to the technology adopted are created. Delays described in the SDF are now introduced for the logic gates. This allows to run gate level simulations (GLS) to check against mainly functional problems present in the design and possible undetectable problems in the constraints. In the case where the design is unsynchronous or multiple clock domains are present, STA is unable to identify timing violations. Proper verification of the block is needed and GLS may be able to pinpoint them. Moving on to post-Synthesis where layout is generated, this step includes planning the placement of the cells, routing all blocks together and generating the clock tree. If all timing checks are met, the product can be signed off and is released into production for later validation and testing of the physical design [17].

1.2. Verification Flow and Challenges

As previously introduced, verification plays a crucial role in the chip design cycle. Starting off with the testbenches that must be created, the verification environment is defined and setup to execute tests. Stimuli is inputted through the environment and applied to the Design Under Test (DUT), while the response is monitored against the expected outcome [11]. In addition, tests listing all possible cases are generated ensuring all potential scenarios and corner cases are covered.

Verification cycle consists of four major steps:

- Development/Identification phase that contains making the verification plan (vPlan), building the architecture including the testbench and the tests.

The verification plan is the meeting point between the DV team and other teams like design, system, architects, platform architecture. It clarifies what needs to be verified. The vPlan is formed out of the chip specs where the DV engineer has to understand the design and extract all verification items, also known as features or attributes, in addition to review the scenarios to be covered in the test planning. In case of no available specs, it’s imperative to go through the HDL code and perform bug hunting. Test strategy and tools are identified at this stage, and testbench gets created. Not only that, but also coverage metrics and goals are clearly defined as the target from later stages is to meet them. The main challenge is to cut the time to develop through adopting constraints, building verification components that are adjustable, using CRV and SVA.

- Simulation phase including software, emulation, acceleration, etc.

The simulations run initially are simple in hope of revealing basic working of the design and aiming to bring up the device for more complex verification. The goal is to hit the coverage goals defined in the verification planning. Tests run can either be constrained or directed. The former mainly randomizes the input stimuli with defined constraints to limit the space and speed up simulations. It can cover a wider amount of test scenarios and is usually first adopted to quickly ramp up the metrics and cover the most features. The latter implies that the sequences generated cover a particular scenario and it is used to hit a known feature. It is adopted to push the coverage metrics higher and to assist the random simulations as some features are harder to attain and need more simulation time if not specifically targeted. Depending on the design block and the DV engineer's preferences, directed tests can be initiated first targeting specific test cases and then the scope is widened with more randomization introduced. UVM is the most popular verification methodology and is in fact adopted by the DV team I worked with during my internship. Using Transaction Level Models (TLM) enabled through UVM, further explained in subsections 1.4 and 1.5, reduces the efforts to simulate, in addition to hardware acceleration, emulation, etc. The latter consists of placing the design on the Field Programmable Gate Arrays (FPGA), allowing the testbench to run on a general-purpose machine [18]. The former, also FPGA-based, is slower yet offers more debug possibilities and full observability. The main idea is that the DUT is exposed to a hardware environment that resembles where it's expected to function. In case the simulations are long and hard to manage, hardware-assisted verification is preferable. Unfortunately, it only works with digital circuits, not applicable in the blocks the PMU team has to verify since analog signals are also present. Aiming to stretch for the coverage goals, number of tests to simulate could also be decreased by adopting Coverage-driven verification (CDV) [19], through re-allocation of resources so that coverage can be attained on the required timeline, as explained before. An example of this optimization is the use of test case ranking [20]. Machine Learning algorithms (ML) are developed to check the coverages described in subsection 1.3 to identify which tests contribute the most so they can be run more often to hit higher metrics.

- Debug phase that consists of finding the source of the bugs.

Re-running sets of simulations to figure out if the device functionality has regressed or reverted is known as regression. Many randomized versions of the verification tests cases are launched generally in the evening each with a different random seed and results are reviewed the next day. The purpose is to identify where the bugs have begun, in order to be caught at their initial point. For the issue to be noticed, checkers are needed. Each attribute has an associated checker and is a collection of assertions with the role to compare resulting outcome of the DUT against what is expected. Assertions look for design rules and can quickly pinpoint where the errors are generated. Debugging effort should be minimized as possible, hence why in addition to SVA, CRV allows for less tests to debug.

- Cover phase where code and functional coverage are checked and reported.

The reporting process allows to check how good is the verification planning and execution. The goal is to check whether the environment built verify the purpose of the design through the test results. Coverage is defined as the percentage of verification intentions that have been met. If the coverages are 100%, the DV team would feel more confident about the correctness of the design, even though they can never be sure it's fault-proof. Note that the phases loop as the project progress moves forward, in other words, simulations are launched, coverages are checked, and as errors are generated, debugging is done, and the cycle reiterates.

1.3. Coverage Metrics

There are two types of coverages: code and functional coverage [21]. The former is a percentage measure of the amount of HDL code that has been executed. Typically, this figure of merit is extracted from the simulator and reported to view the structural coverage of the code. In this metric, the intention of the design is not checked, however, it follows the code control graph and is expected to reach 100%. Even then, it remains uncertain whether the design works as intended.

It incorporates four types of coverages [22]:

- Block – checks the number of line codes executed.
- Expression – checks if all right-hand side (RHS) expressions results have been applied.
- Finite state machine (FSM) – verifies all possible states are visited and transitions have been exercised.
- Toggle – looks if all assignments and possible bits in code variables have their state modified.

Whereas the latter is a metric defined by the user and it measures the design features mentioned initially in the vPlan that have been addressed in the tests' simulation and the verification environment. After running the simulations, analysis to the report is done and more test planning is needed to hit the verification holes, making sure all the code is exercised and the vPlan attributes are hit.

There are two types of functional coverage [23]:

- Data-oriented Coverage – checks the variations of data values that have occurred. We can get data-oriented coverage by writing coverage groups (cg), coverage points and by cross coverage. In other words, it is possible to define in SystemVerilog the specs of the coverage model, and the cg can be sampled at a defined clocking event and can collect the coverage points.
- Control-oriented Coverage – checks whether behavioural sequences have occurred. It can be obtained through assertion coverage by writing SVA.

Since the metric is related to the attributes defined by the DV engineers, the coverage can be 100% even if some features are missing and not accounted for in the functional coverage model [24].

1.4. UVM Methodology

UVM is developed to address the efforts in enhancing the verification process of integrated circuits and in meeting a more demanding market. It is a class library enabling automation for the SystemVerilog language to develop testbenches that are easy to write, configurable and reusable [25]. It also provides a set of standards and best ways for tackling verification, along with how to run simulations and handle the results. Pre-defined classes and structures are offered by the methodology to create the UVM testbench that gets compiled once, following that, many tests can be initiated.

Verification components are created, and a defined hierarchy is built. More explanations will follow as the generator will be presented. As an example, a UVM typical environment can be shown below in Fig. 1.

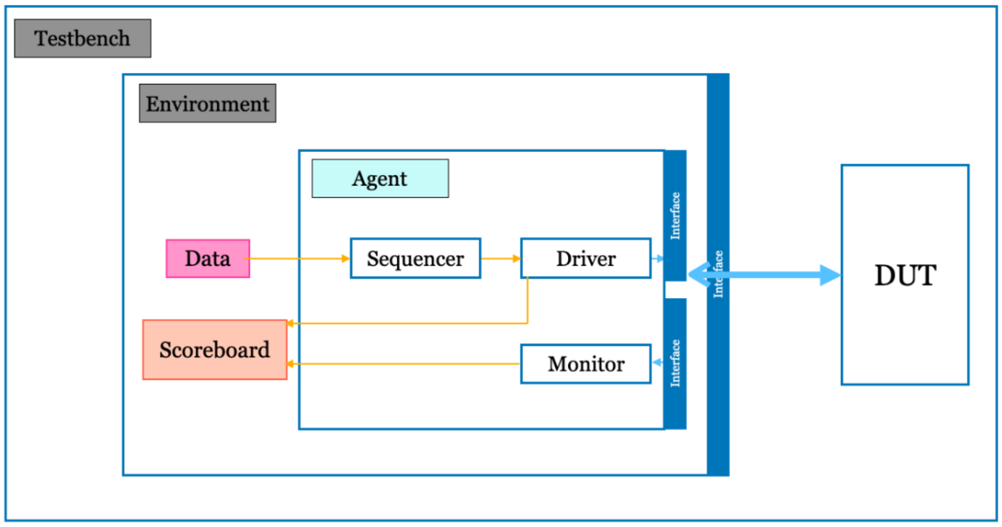


Figure 1: Typical UVM Structure

From top of the hierarchy down, a testbench is instantiated that includes the verification environment and the Design Under Test module. The DUT and the environment interact through the interface. The environment is formed out of agents and a scoreboard. The agents can either be active or passive. The former implies that the agent includes a driver, a sequencer, and a monitor, whereas the latter eliminates the driver and the sequencer. An active agent receives stimulus input or a UVM sequence that gets managed by the sequencer, converted into signal level, and applied by the driver to the DUT through its interface, and the response is collected by the monitor. The driver will inform the scoreboard about the transaction intended to be driven on the bus. UVM adds features such as the generation of sequences, data automation and manipulation. On a higher-level abstraction, the transaction is a UVM sequence item, consisting of data variables and constraints that can be randomized, making UVM implement TLM by also using specific communication ports to send and receive data. Going back to the scoreboard, its role comprises of comparing the resulting output from the DUT and the expected response. It receives its input from the agent's monitor and from the reference model that mimics the DUT functionality or from the driver. In addition to the flexible testbench, UVM allows access to a centralized configuration database that stores objects, interfaces, class handles and others. With each holding an identifier, all components can pass and receive items.

Regarding simulation, UVM offers phases whereby it's possible to manage the order in which the components and objects are created and executed, shown in Fig. 2. First, the build phases are executed, where the testbench is constructed and configured hierarchically. Sequentially, the run phases consume simulation time as they consist of SV tasks, and they generate and apply the stimulus. Lastly, in the last phase, the results of the tests are reported and analyzed. In the figure, it can be noted that each phase contains many sub-phases, and they are configurable up to the user. For debugging purposes, UVM macros are enabled for data processing, such as messaging and reporting warnings and errors.

As can be inferred previously, one of the advantages of adopting UVM methodology is the separation of the stimulus or sequences from the testbench which makes them both reusable. Another feature is the ability to control components through their configuration objects, no matter the hierarchical level of the component itself [26].

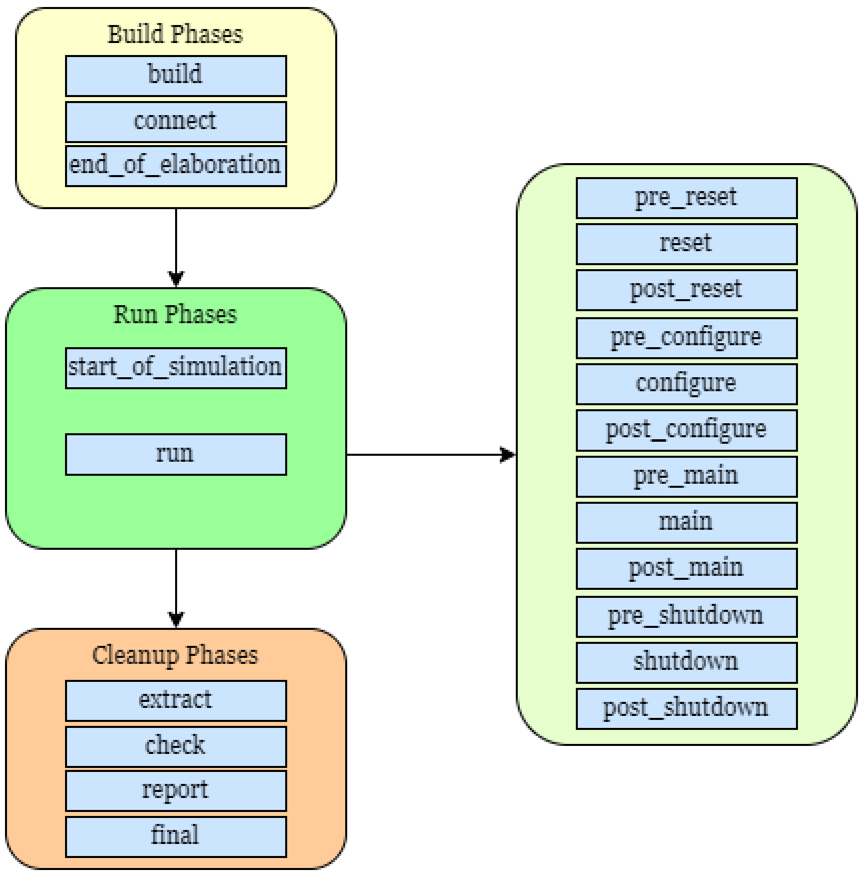


Figure 2: UVM Phases

1.5. Transaction Level Modeling (TLM)

TLM describes a system whereby transactions are used to transfer information across the channels. It simulates faster than RTL as it can be used to build highly abstract models at a system level. The concept can be implemented in any description language. The transaction is a class object that defines specific information regarding the system protocols. It represents the data transmitted through ports, disregarding the specifications of the components themselves. Many transactions can model event sequences, allowing engineers to disregard the signal-level details and focus on the interaction of the system. Adopting TLM is crucial to target the rising design complexities and make verification faster [18].

Section 2: Design of the Generator

Before conducting the development of code generator, it is imperative to design and define the building blocks of the UVM testbenches that will be created for each design intended to be verified. The phases of DV are conserved, yet more systematized. The DV engineers receive the blocks along with the specification and the documentation of the actual implementation. Subsequently, they prepare its verification plan as expected, as they move on to adopting the UVM generator to create the testbench instead of doing all the repetitive work manually. This step enables the user to invest more efforts into implementing the test scenarios and the checkers. In order to create the necessary files, UVM templates have to be defined for each component needed in the environment allowing the generator to rely on them and on the information it parses from the RTL code for proper functioning.

In this section, a detailed description will be presented firstly about the possible testbench structure the generator can output, and secondly, about the steps needed to extract information from the design in an automated way.

2.1. General description

Briefly, the code generator will receive the SystemVerilog design files that describe all functionality of the block and use the Python templates to create the complete UVM environment for testing of this specific block. The generated files are suitable to perform block level verification and can be also vertically reused for top level verification at a higher hierarchical point. The generation of the interfaces along with the complete structure is automated, what remains to be customized as desired is the development of the checkers, along with the test scenarios. The structure generated will allow the DV engineers to tweak the UVM verification components (UVC) and make their own modifications easily.

2.2. UVM Structure

Different testbench architectures are proposed depending on the nature of the block, whether it should receive data packets or implement a protocol. The latter is simpler and allows the structure to have less components.

2.2.1. For data signals

In this scenario, it is required to send in packets of data as inputs and then keep track of the stimulus through the monitor of the driving agent. Thus, an active agent is absolutely needed sharing the DUT's input signals with its own interface. The response of the DUT has to be collected and monitored by another agent that interfaces with the output signals. In this case, the environment requires a scoreboard to compare what the agent has collected to the expected outcome from the input stimulus. In order to minimize the efforts that the user has to perform to adapt the files into top level simulations, the monitoring agent must be contained inside another environment ready to be inserted in a more complex structure.

The reasoning mentioned leads to the structure shown in Figure 3 for block level testing that conforms with the standard UVM methodology. In the top testbench module *tb_top*, the DUT is instantiated in the same hierarchy with all the tests exercising their different objective. Each *test* contains an *active environment* that englobes the agents driving the stimuli (set as *active*) and a *passive* environment that itself contains agents collecting responses (set also as *passive*), along with the dedicated sequence library *seq_lib* with all sequences to initiate on the design block. The number of agents in each environment is the same and will be dependent on the actual DUT and can be customized by the DV engineer. Diving deeper into the active one, the agents are instantiated containing all UVM components (driver, sequencer, monitor) and in the other environment only the monitor is present, for the purpose of checking and coverage only. The stimuli are coded by the user as UVM sequences in the library and passed down to the corresponding sequencer of the active agents through the virtual sequencer in the environment. In addition, a common scoreboard is added to capture data from the monitors of all agents and perform the comparison, keeping track of the results. Focusing on the active agent, the transactions received are executed by its sequencer which plays the role of the moderator sending the transaction to the driver. In turn, this entity passes data to the DUT through the interface handle complying with the protocol of the block. The monitor within the active agent will capture the stimuli sent, convert it to a translation level data object and broadcast it to the scoreboard to calculate the expected value. Similarly, the monitor within the output agent will collect the output signals from the DUT, translate it, exercise the checkers, and send it to the scoreboard. As can be inferred, at this level, each block is verified individually and the UVC are in control of toggling the pins of the block.

In contrast, vertically integrating the UVC with other agents and environments at a higher level means no control of the pins, as stimulus originates from other blocks and checkers are already implemented. Figure 4 displays the testbench architecture when the simulations are run in top level. The main difference is only the passive environment implying that only the passive agents are present. The same files generated can be used in top level without any other modifications. The UVC will be connected with other ones, using the same checkers to verify constantly the signal integrity at its interfaces.

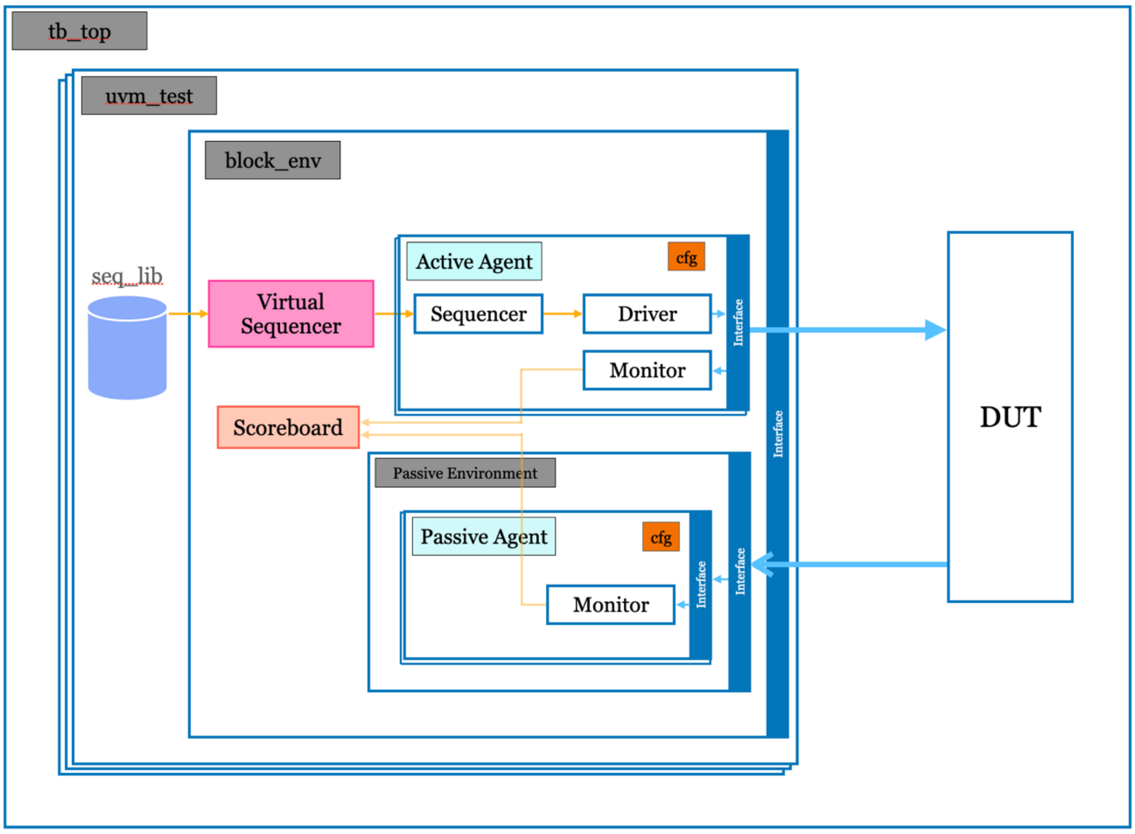


Figure 3: UVM testbench architecture for block level verification in case of data signal flow

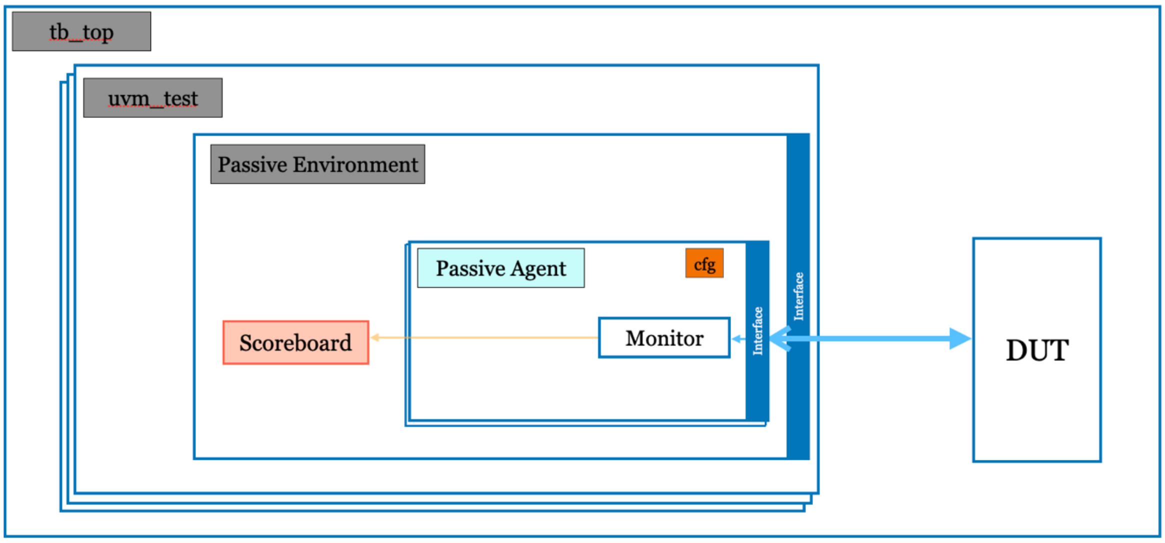


Figure 4: UVM Testbench architecture for top level verification

2.2.2. For control signals

In the second structure that can be formed, in the case where the DUT implements some protocols and manages control signals, the generated architecture for block level verification would result into the one shown in Figure 5. Such scenarios can be for example implementation of Advanced High-performance Bus (AHB) [27] or Advanced eXtensible Interface (AXI) [28] protocols. Looking from afar, the hierarchy is the same; the top testbench is instantiated containing the tests, with all UVM components present. As expected, each test includes the active environment along with the sequence library, as stimulus still needs to be driven. However, the active agents are only needed inside the environment as no data packets are actually sent, thus no scoreboard is initialized.

On the other hand, converting the structure to be used in top level entails that the active environment must become passive, thus, passive agents with only monitors. The virtual sequencer inside this environment is needless as the agents do instantiate neither their sequencer nor their driver. The UVC, as mentioned before, does not drive any signal, however, the monitor is indeed required with all its checkers in use. From this reasoning, the architecture will be the same as the one presented in Figure 4 without the scoreboard, hence, its integration into the top level with other blocks will be the same for any type of DUT.

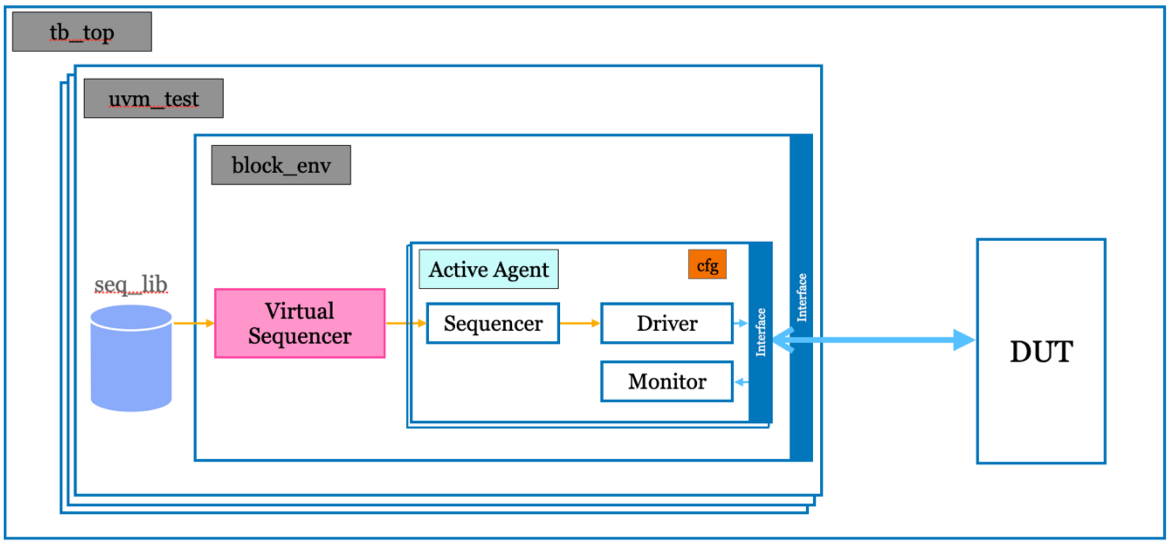


Figure 5: UVM testbench architecture for block level verification in case of control signal flow

2.3. Interface generation

The user can choose how many agents are to be created inside the environment and what signals to be included in each agent. A list of agents' names and their port lists along with the type of the agent can be inputted to the generator. Aiming for better reusability and efficiency across the DV team throughout different projects, not all agents get their verification component (VC) files generated. The last information about the agent's type is required as some of them already exist and can be referenced in this VC without reinventing the wheel.

Upon receiving the list, information about the block can be automatically extracted from the RTL files, to be able to relate the given description with the actual signals and parameters. In other words, from the modules that the generator parses from the design, a more detailed list will be formed containing the following:

- direction of each signal, whether input, output or bidirectional
- comments related to each signal
- dependencies of the module
- module parameters with their values and their comments

Even if this step fails, the testbench can still be generated, as only the interfaces are related to the block parsing.

Section 3: Implementation

3.1. Generator Flowchart

The working mechanism of the code generator can best be described in Fig. 6.

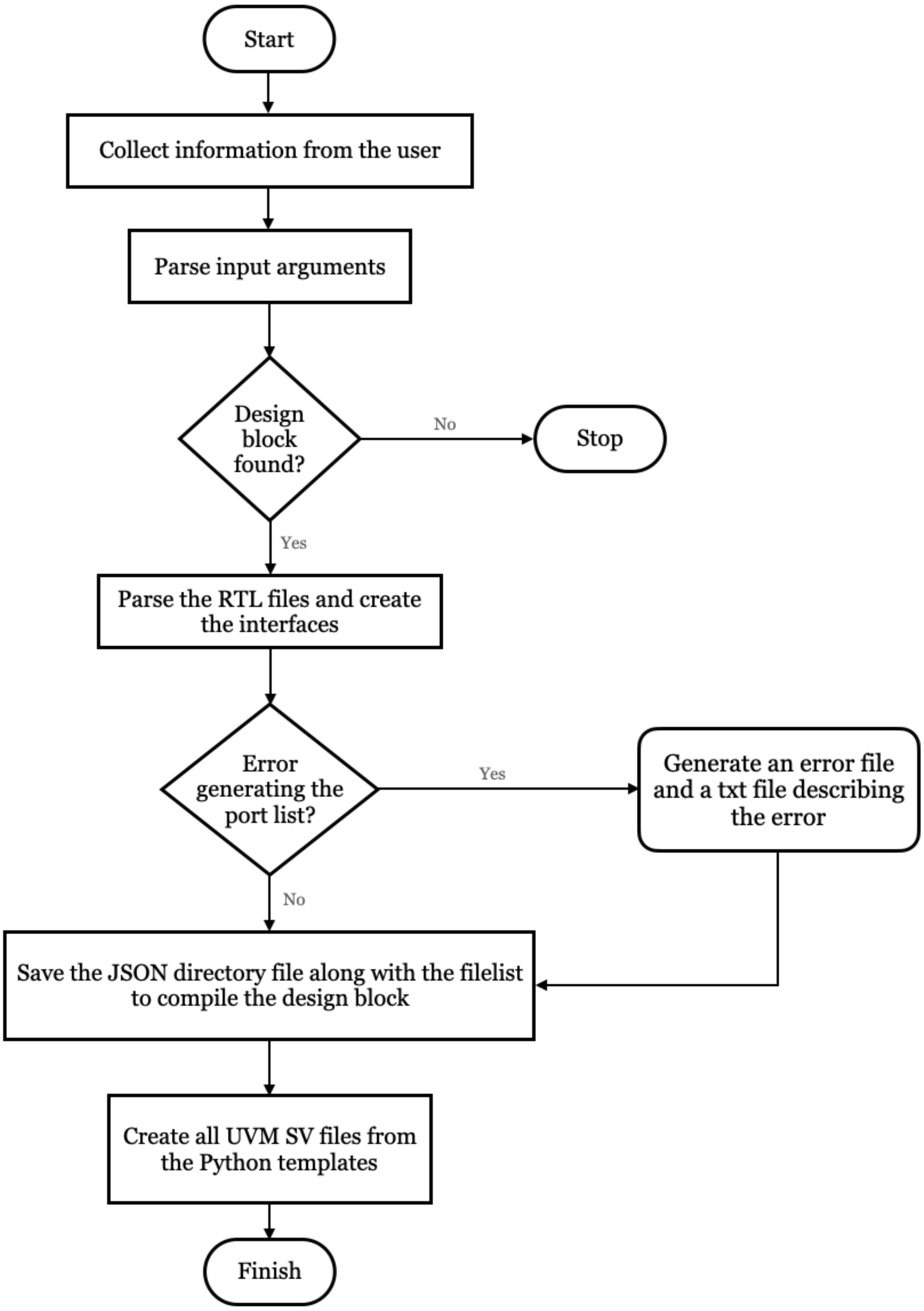

Start
Collect information from the user
Parse input arguments
Design block found?
No
Stop
Yes
Parse the RTL files and create the interfaces
Error generating the port list?
Yes
Generate an error file and a txt file describing the error
No
Save the JSON directory file along with the filelist to compile the design block
Create all UVM SV files from the Python templates
Finish

Figure 6: Flowchart showing the execution steps of the code generator

3.2. Execution Workflow

3.2.1. Pre-processing

The main Python script is run from the command line and collects the following information from the user:

- `design_name`: name of the block design file that will be verified.
- `change_folder_name`: change the name of the verification folder (y/n).
- `folder_name`: name of the folder if to be changed, default `vc_{design_name}`.
- `dest_dir`: destination directory where the verification folder will be created.
- `override`: whether to override the verification environment if it exists (y/n).
- `mult_agents`: whether to instantiate many agents in the environment (y/n).
- `agents_json_path`: if `mult_agents` argument is (y), the full path is needed for the JSON file detailing all agents' name with their signals.

Checkers are implemented to make sure that the user is inputting a valid answer. The script will go through all possible design directories to find the complete design path of the block, making sure it exists. If the block is nowhere to be found, the execution of the code will end. The top RTL file matches the design name, and it is fetched in order to process information about the block. The block folder must be compiled standalone through generating its filelist, which is the list of files that it depends on. Otherwise, some dependencies or errors in the code need to be tackled and feedback is shared with the corresponding designer. This step generates a dictionary later converted into a JSON file describing the connections of the DUT. The filelist and the JSON output along with errors in design parsing if any are saved in the VC folder for the user to reference back to in case any mistake is made.

Afterwards, each signal in the generated port lists has to match with the signal name given in `agents_json_path`. For each agent, the following dictionaries are created where each contain most importantly the comments, size, name, value of each module data:

- `parameters_dict`
- `input_signals`
- `output_signals`
- `internal_signals`

In the case where only one agent is instantiated, the step above is automatically performed after the interface parsing from the output dictionary. The single agent's name will be the design one and it will contain all the signals connected to the DUT. In the other case, a meshed dictionary is created as the dictionaries need to be compatible with the JSON file that the user has written.

The next step is the generation of the top VC folder along with its subdirectories described in Table 1 and shown in bold in Fig. 7. All paths to the created directories are stored in a dictionary for subsequent functions to retrieve them easily. Subsequently, after all this effort, the generator has all inputs needed to create the verification testbench.

3.2.2. VC Directories and Files Generated

Table 1 shows the description and the contents of each subdirectory inside the main VC folder. As mentioned previously, each entry gets created and the full path pointing to it is stored in a look-up table for easier retrieval in the following steps.

Subdirectory	**Purpose**
`conf`	Configuration files required to compile the environment, testbench on block level and top level
`diaglist`	List controlling the test names to be run
`docs`	Documentation for the verification folder
`docs/specs`	Files generated from parsing the design RTL code and creating the interface: - Filelist needed to compile the design block - JSON dictionary containing list of ports and connections - Error and text file in case there was any error in the interface generation
`src/sv/block_tb`	Contains the Top-Level module. In the future, this directory will also contain the driving interface
`src/sv/bundles`	Bundle interfaces in case of additional analog signals in the block
`src/sv/chip_tb`	All base test files in case of top-level simulation
`src/sv/components`	For each agent, a directory for UVM SV components is created
`src/sv/hookups`	Includes the passive interface making all connections to the DUT signals
`src/sv/reg_mirror`	Contains the register package files
`src/sv/sva`	List of all SV assertions
`src/sv/tests`	Directory for all tests to be exercised
`src/sv/tests/seqlib`	Contains scenarios and stimuli specific to the DUT
`src/sv`	Includes all SV files for the UVM environment

Table 1: Explanation of subdirectories inside the generated verification folder

Figure 7 displays all the subdirectories already mentioned along with the files listed more clearly with their naming convention. The structure is well organized and adopted already for all the VCs in the team, making it easily readable and manageable by the user, in case more modifications are desired. In case of a single agent containing all DUT signals, the agent's files will be generated directly under components as shown in the figure. Then again, in the other case, another subdirectory will be allocated for each agent with all files included in it. In this work, the register mirror `reg_mirror` and the `bundles` directories will be kept empty, as it's accorded secondary priority for the time being. The blocks used to verify the implementation of the generator do not contain these features, however, it will be added as a task for the future. The listed sva entry is fully programmed by the user, so it’s also empty.

```
|-vc_<block_name>
|  |-conf
|  |  |-vc_<block_name>_chip.config.pl
|  |  |-vc_<block_name>.config.pl
|  |  |-vc_<block_name>_tb.config.pl
|  |-diaglist
|  |  |-vc_<block_name>_diaglist
|  |-docs
|  |  |-specs
|  |  |  |-<block_name>_err.txt
|  |  |  |-<block_name>.f
|  |  |  |-<block_name>.json
|  |  |  |-<block_name>.txt
|  |-src
|  |  |-sv
|  |  |  |-block_tb
|  |  |  |  |-chipTb.sv
|  |  |  |  |-<block_name>_block_env.sv
|  |  |  |  |-<block_name>_block_pkg.sv
|  |  |  |  |-<block_name>_tb_if.sv
|  |  |  |-bundles
|  |  |  |-chip_tb
|  |  |  |-components
|  |  |  |  |-vc_<agent_name>
|  |  |  |  |  |-vc_<block_name>_agent_cfg.sv
|  |  |  |  |  |-vc_<block_name>_agent_defines.sv
|  |  |  |  |  |-vc_<block_name>_agent_if.sv
|  |  |  |  |  |-vc_<block_name>_agent_pkg.sv
|  |  |  |  |  |-vc_<block_name>_agent.sv
|  |  |  |  |  |-vc_<block_name>_driver.sv
|  |  |  |  |  |-vc_<block_name>_monitor.sv
|  |  |  |  |  |-vc_<block_name>_seq_item.sv
|  |  |  |  |  |-vc_<block_name>_seq_lib.sv
|  |  |  |  |  |-vc_<block_name>_sequencer.sv
|  |  |  |-hookups
|  |  |  |  |-vc_<block_name>_hookups.sv
|  |  |  |-reg_mirror
|  |  |  |-sva
|  |  |  |-tests
|  |  |  |  |-<block_name>_base_test.sv
|  |  |  |  |-<block_name>_test_pkg.sv
|  |  |  |  |-seqlib
|  |  |  |  |  |-vc_<block_name>_seq.sv
|  |  |  |-vc_<block_name>_env.sv
|  |  |  |-vc_<block_name>_env_cfg.sv
|  |  |  |-vc_<block_name>_env_defines.sv
|  |  |  |-vc_<block_name>_env_pkg.sv
|  |  |  |-vc_<block_name>_env_if.sv
|  |  |  |-vc_<block_name>_env_scoreboard.sv
|  |  |  |-vc_<block_name>_env_vsequencer.sv
```

Figure 7: Generated file structure for the verification component

3.2.3. Generate UVM Architecture

As mentioned lastly in subsection 3.1, the next step is to generate the SystemVerilog (SV) files from the templates in Fig. 8 and render them according to their correct placement in Fig. 7. The base files are written in Python using Jinja library, which is an open-source Python templating engine that renders user-customized documents [29]. An excerpt of the template to create the environment package file listing dependencies for the UVM environment is displayed in Fig. 9. Each template will be converted into an SV file given the needed inputs arguments to fill in the placeholders. The field names, for example `<block_name>`, are defined within double brackets, and they can be customized, filtered and processed. The argument for the template is a formatted string that corresponds to the SV code. Comments are inserted to assist the DV Directly Responsible Individual (DRI) in editing, adding their own instructions and customizing the files after complete generation.

To ensure maximum reusability and minimal rewriting implemented among the DV team, the VC files for blocks used across blocks and projects should not be rewritten. In the `agents_json_path` parsed from the user, each agent entry has an attribute `cell` where it's required to insert the name of the VC folder if it already exists. In this case, no separate directory under `components` is instantiated for this agent, and simply the dependency is added in the configuration files. For example, such instances can be clocking agents that connect clock and reset signals to the blocks, or Advanced High-performance Buses (AHB) [27] that is a protocol for high-speed communication between interconnects in a design.

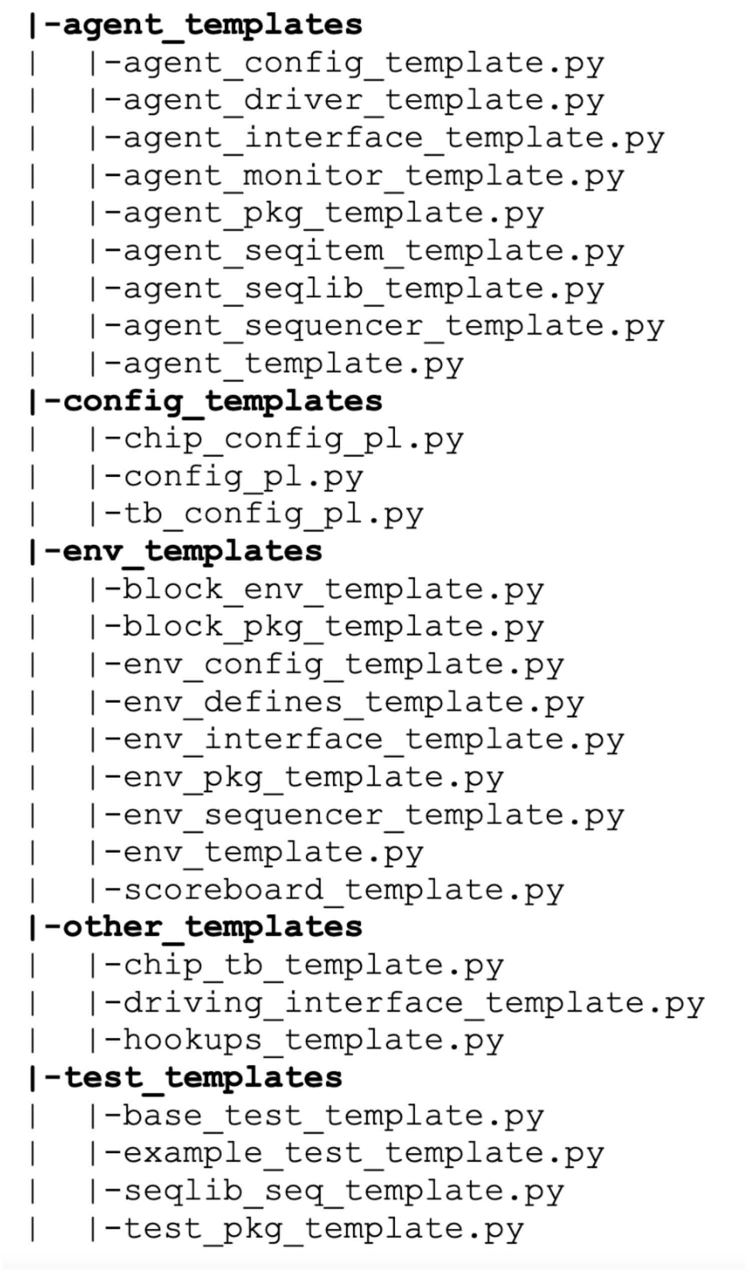
The adopted approach to create the files is bottom-up:

- The agents are filtered whether an SV directory is to be created or not. The files listed beneath `agents_templates` in Fig. 8 are created in the `components` subdirectory. Most importantly, each agent contains its interface file that lists the desired signals by the user.
- One level above in the hierarchy, the environment files listed in Fig. 8 are created under the `sv` directory. All necessary imports are passed down so the package file can reference them. The environment along with its interface also instantiates the other instances and creates the connections. The active block environment is created with its sequencer connected to the agents'.
- An important step must be done at this stage, in the case where AHB agents are required, a mapping function is called to connect the theoretical signals from the existing VC AHB agent to the ports defined in the block design.
- Following that, the generator can create the hookups file under the `hookups` directory. The purpose of the file is to connect the signals from the interface of each environment agent to the signal in the DUT.
- Next, the testbench interface, also known as the driving interface, is generated for the block environment. The main difference here is that the clock signals from the clocking block are declared and passed to the agents.
- Moreover, the base test file that resets the blocks is created along with its package under `tests` and its sequence item yet to be completed under `seqlib`.

The user will inherit the test, develop its one and list it in the package to be compiled.

- At the top of the hierarchy, the top level testbench is generated. It contains the DUT instance connected with the driving interface.
- Finally, the configuration files are formed with the necessary components to be compiled in order to run the initial base test.

The environment can then be initially built and run without any error. After the execution of the main generator, more detailed tests need to be added along with assertions and different scenarios to be exercised under the sequence library directory.



```
|-agent_templates
|   |-agent_config_template.py
|   |-agent_driver_template.py
|   |-agent_interface_template.py
|   |-agent_monitor_template.py
|   |-agent_pkg_template.py
|   |-agent_seqitem_template.py
|   |-agent_seqlib_template.py
|   |-agent_sequencer_template.py
|   |-agent_template.py
|-config_templates
|   |-chip_config_pl.py
|   |-config_pl.py
|   |-tb_config_pl.py
|-env_templates
|   |-block_env_template.py
|   |-block_pkg_template.py
|   |-env_config_template.py
|   |-env_defines_template.py
|   |-env_interface_template.py
|   |-env_pkg_template.py
|   |-env_sequencer_template.py
|   |-env_template.py
|   |-scoreboard_template.py
|-other_templates
|   |-chip_tb_template.py
|   |-driving_interface_template.py
|   |-hookups_template.py
|-test_templates
|   |-base_test_template.py
|   |-example_test_template.py
|   |-seqlib_seq_template.py
|   |-test_pkg_template.py
```

Figure 8: UVM SystemVerilog Templates

```
import jinja2
ENV_PKG_TEMPLATE = jinja2.Template("""
//==============================================================================
//
// Description:  UVM Environment Package for {{block_name}}
//
//==============================================================================
// Author: {{author_email}}
//==============================================================================

`ifndef   __VC_{{block_name | upper}}_ENV_PKG__
`define   __VC_{{block_name | upper}}_ENV_PKG__

package vc_{{block_name}}_env_pkg;
  import uvm_pkg::*;
  `include "uvm_macros.svh"
  import vc_X_pkg::*;

  // Import all agent files in the env
  {% for agent in active_agents -%}
  import vc_{{agent}}_agent_pkg::*;
  {% endfor %}

  `include "vc_{{block_name}}_env_cfg.sv"
  `include "vc_{{block_name}}_vsequencer.sv"
  `include "vc_{{block_name}}_env_defines.sv"
  `include "vc_{{block_name}}_scoreboard.sv"
  `include "vc_{{block_name}}_env.sv"
  // TODO: TO BE COMPLETED BY THE DV ENGINEER
  `include "tests/seqlib/vc_{{block_name}}_seq.sv"

endpackage : vc_{{block_name}}_env_pkg

`endif //__VC_{{block_name | upper}}_ENV_PKG__

""")
```

Figure 9: Python template rendered into the SystemVerilog environment package file

Section 4: Results

After presenting the UVM generator, this section gives the results of testing its functionality on some design blocks. Simulations were performed using SimVision through the Xcelium™ software by Cadence. Two blocks are adopted: the first one is a typical synchronous 8-bit positive-edge triggered flipflop and the other one is the Dual-Port SRAM controller. In the first part, the aim is to run some simulations on the generated testbench highlighting the generator's valid output. While the second part consists of adopting the tool to create the VC directory for verification purposes.

4.1. Simulations on Flipflop design

4.1.1. DUT Topology

The Flipflop is a fundamental block in digital circuits used as a data storage element. The written SV code and the diagram are shown in Fig. 10 and 11. The design consists of a clock, reset and input data and an output signal of the same width as the input. The block is positive edge triggered, which means that at the rising edge of the clock signal the output will follow the input if the reset is not asserted. In other times, the output does not respond to the variations of the input, yet it preserves its previous state until the next rising clock edge. Whenever the reset signal goes low, the flipflop will change its state and set the output to low, regardless of the clock signal.

```
module apc_DUT #(parameter WIDTH = 8)(

  input  wire        clk,
  input  wire        rst_n,

  //Inputs
  input  logic [WIDTH-1:0]  data_D,

  //Outputs
  output wire [WIDTH-1:0]  data_Q
);

logic [WIDTH-1:0]  s_data_o;

assign data_Q = s_data_o;

always_ff @(posedge clk or negedge rst_n) begin
  if(~rst_n) begin
    s_data_o <= '0;
  end //Reset
  else begin
    s_data_o <= data_D;
  end
end //always_ff

endmodule : apc_DUT
```

Figure 10: RTL code for the flipflop used for testing

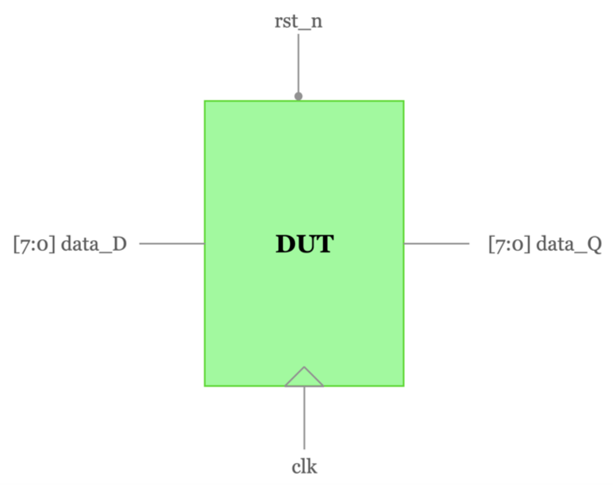
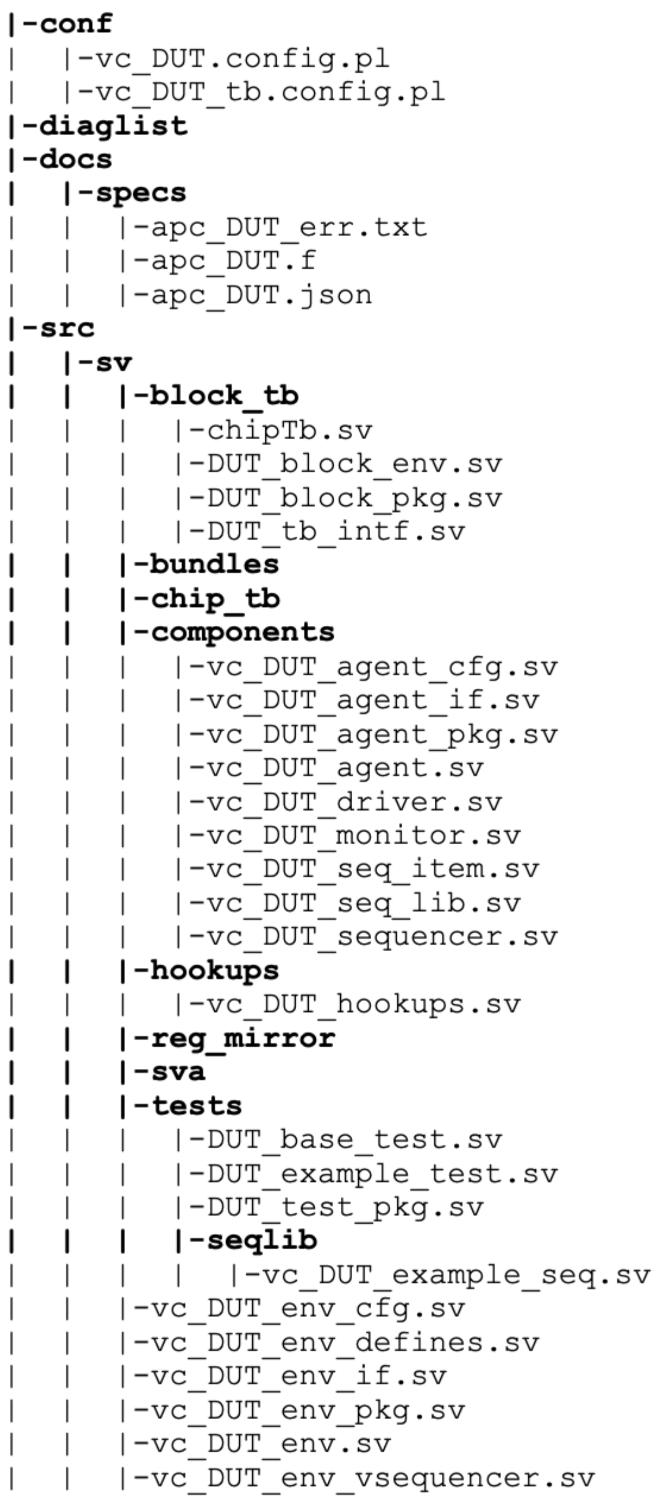


Figure 11: Flipflop diagram used for testing

4.1.2. Testbench creation using the generator

As described above, the point of using a simple block design is to shift focus on the functionality of the generator. The tool was given the path of the block along with the other required inputs without the JSON description of the agents since only one agent is desired. After finding the codes, the port list is generated from the main SV module and the testbench can now be created. As described in the previous subsections, the UVM templates are used for creating the files shown in Fig. 12. For simplicity, the `chip_tb_config` is not created as only block level testing is required. In this scenario, no further subdirectory is created under `components`, only one agent is created with the default name corresponding to the design along with its files. All the signals are combined in the agent interface which is instantiated in the environment one and connected to the DUT signals in the hookups and the driving interface. The VC folder is compiled to ensure no errors or dependencies are present. An excerpt of the generated JSON file `apc_DUT.json` is presented in Fig. 13, using an inside tool developed in the company. Out of parsing the design files to get a list of the connections of the block, dictionary-like entries are formed as follows.

- Busses: in this entry, all internal signals declared in the module are added. Each one has the following attributes: name, declaration type, most-significant bit (msb) and least-significant bit (lsb) of the bus length and line comments.
- Parameters: contains information about each module parameter including the declaration type, value, comment, and its bus width.
- Ports: all inputs and outputs of the block are defined in this entry with their relevant characteristics. For each signal, the declaration type is added along with its length, direction (input or output to the DUT), and cell name (design block).



```
|-conf
|   |-vc_DUT.config.pl
|   |-vc_DUT_tb.config.pl
|-diaglist
|-docs
|   |-specs
|   |   |-apc_DUT_err.txt
|   |   |-apc_DUT.f
|   |   |-apc_DUT.json
|-src
|   |-sv
|   |   |-block_tb
|   |   |   |-chipTb.sv
|   |   |   |-DUT_block_env.sv
|   |   |   |-DUT_block_pkg.sv
|   |   |   |-DUT_tb_intf.sv
|   |   |-bundles
|   |   |-chip_tb
|   |   |-components
|   |   |   |-vc_DUT_agent_cfg.sv
|   |   |   |-vc_DUT_agent_if.sv
|   |   |   |-vc_DUT_agent_pkg.sv
|   |   |   |-vc_DUT_agent.sv
|   |   |   |-vc_DUT_driver.sv
|   |   |   |-vc_DUT_monitor.sv
|   |   |   |-vc_DUT_seq_item.sv
|   |   |   |-vc_DUT_seq_lib.sv
|   |   |   |-vc_DUT_sequencer.sv
|   |   |-hookups
|   |   |   |-vc_DUT_hookups.sv
|   |   |-reg_mirror
|   |   |-sva
|   |   |-tests
|   |   |   |-DUT_base_test.sv
|   |   |   |-DUT_example_test.sv
|   |   |   |-DUT_test_pkg.sv
|   |   |   |-seqlib
|   |   |   |   |-vc_DUT_example_seq.sv
|   |   |-vc_DUT_env_cfg.sv
|   |   |-vc_DUT_env_defines.sv
|   |   |-vc_DUT_env_if.sv
|   |   |-vc_DUT_env_pkg.sv
|   |   |-vc_DUT_env.sv
|   |   |-vc_DUT_env_vsequencer.sv
```

Figure 12: VC directory for the Flipflop DUT

```
{
    "apc_DUT": {
        "busses": [
            {
                "s_data_o": {
                    "bus": "[WIDTH-1:0]",
                    "comment": [
                        ""
                    ],
                    "declaration": "var",
                    "lsb": "0",
                    "msb": "WIDTH-1",
                    "name": "s_data_o"
                }
            }
        ],
        "parameters": {
            "WIDTH": {
                "bus": "",
                "comment": [],
                "type": "",
                "value": "8"
            }
        },
        "ports": {
            "clk": {
                "array": "",
                "bus": "",
                "cell": "apc_DUT",
                "comment": [],
                "connections": {},
                "direction": "input",
                "type": ""
            },
            "data_D": {
                "array": "",
                "bus": "[WIDTH-1:0]",
                "cell": "apc_DUT",
                "comment": [
                    "Outputs"
                ],
                "connections": {},
                "direction": "input",
                "type": "logic"
            },
            "data_Q": {
                "array": "",
                "bus": "[WIDTH-1:0]",
                "cell": "apc_DUT",
                "comment": [],
                "connections": {},
                "direction": "output",
                "type": "[WIDTH-1:0]"
            },
```

Figure 13: Generated JSON file from design parsing for the FF DUT

4.1.3. Development of the tests

At this stage, several tests can be developed under the `tests` directory with scenarios written under `seqlib`. For this DUT, two tests are mainly covered. The base test serves as a reset and an initializer for the DUT. This is considered as the initial step before actually running any test. As for the example test, it starts by executing base test followed by the example sequence. The scenario in this case is straightforward, the stimuli sent by the agent’s driver for the DUT should be collected as output from block. Hence, consecutive sequence items containing the data attribute are randomized and sent to the DUT from the driver, and they are observed in the monitor.

4.1.4. Resulting Simulations

After running the example test, some of the resulting waveforms displayed in SimVision are shown in Fig. 14. The first division of the waveforms show the clock and the reset signal in the top level along with the input data to the DUT and its output. The second and third one display information regarding the signals of the Agent Driver and Monitor respectively. It can be observed that the same data sent from the driver side is collected in the monitor. A new sequence item is declared each time a new packet is to be sent. Only one packet where data is `'h75` is shown in the waveform, sent to the DUT, and collected back.

Looking at the waveforms, connectivity was proven from the proper propagation of all signals. The clock was delivered to all blocks, along with the reset signals. The transaction sequenced into the active agent was translated to pin-level and driven across the agent's interface as the D-input of the Flipflop. The value is also seen on the interface signals connected to the monitor.

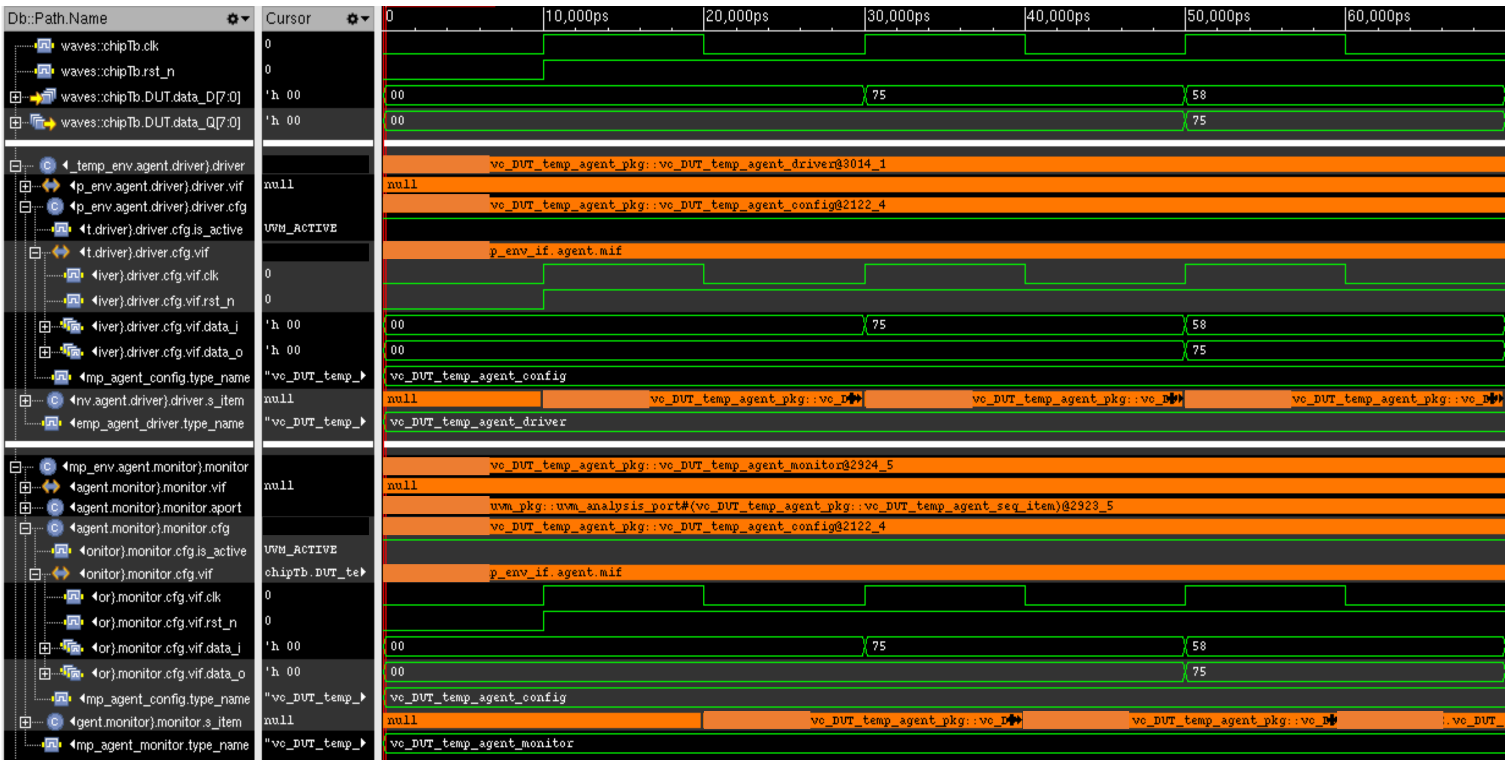

Db::Path.Name
Cursor
0
10,000ps
20,000ps
30,000ps
40,000ps
50,000ps
60,000ps
waves::chipTb.clk
waves::chipTb.rst_n
waves::chipTb.DUT.data_D[7:0]
waves::chipTb.DUT.data_Q[7:0]
_temp_env.agent.driver}.driver
p_env.agent.driver}.driver.vif
p_env.agent.driver}.driver.cfg
t.driver}.driver.cfg.is_active
t.driver}.driver.cfg.vif
iver}.driver.cfg.vif.clk
iver}.driver.cfg.vif.rst_n
iver}.driver.cfg.vif.data_i
iver}.driver.cfg.vif.data_o
mp_agent_config.type_name
nv.agent.driver}.driver.s_item
emp_agent_driver.type_name
mp_env.agent.monitor}.monitor
agent.monitor}.monitor.vif
agent.monitor}.monitor.aport
agent.monitor}.monitor.cfg
onitor}.monitor.cfg.is_active
onitor}.monitor.cfg.vif
or}.monitor.cfg.vif.clk
or}.monitor.cfg.vif.rst_n
or}.monitor.cfg.vif.data_i
or}.monitor.cfg.vif.data_o
mp_agent_config.type_name
gent.monitor}.monitor.s_item
mp_agent_monitor.type_name
'h 00
null
UVM_ACTIVE
chipTb.DUT_te
"vc_DUT_temp_
00
75
58
vc_DUT_temp_agent_pkg::vc_DUT_temp_agent_driver@3014_1
vc_DUT_temp_agent_pkg::vc_DUT_temp_agent_config@2122_4
p_env_if.agent.mif
vc_DUT_temp_agent_config
vc_DUT_temp_agent_driver
vc_DUT_temp_agent_pkg::vc_DUT_temp_agent_monitor@2924_5
uvm_pkg::uvm_analysis_port#(vc_DUT_temp_agent_pkg::vc_DUT_temp_agent_seq_item)@2923_5
vc_DUT_temp_agent_monitor

Figure 14: Simulation waveform showing random packets sent to the DUT from the driver and collected in the monitor

4.2. Verification of Dual Port SRAM Controller

4.2.1. SRAM Controller Topology

The other block to test the generator on is shown in Figure 15 [30]. The high-level diagram mainly consists of 3 instances, 2 AHB to SRAM bridges (Path A and B) and an arbiter. The purpose of the block is to multiplex both bridges to transfer data from these two separate ports that are connected each to an AHB bus. The arbiter chooses the order of execution of the data transactions and allows access to a common SRAM memory using the round-robin scheduling approach [31]. In other words, both paths get equal share of time to run their tasks in a cyclic way. The request for memory gets processed by the interface that initiates it first. However, in the case where a transaction is made independently in parallel from both interfaces, one of them gets priority and is allowed access before. As explained before, the working mechanism of each block to be verified needs to be comprehended to create the attributes planning. For this block, the AHB transaction sent from each path has to be correctly transferred to the SRAM Memory in the case where sequential sequences are inputted. Moreover, the arbitration feature needs to be verified. Parallel requests need to be formulated and the correct access order should be exercised by the DUT.

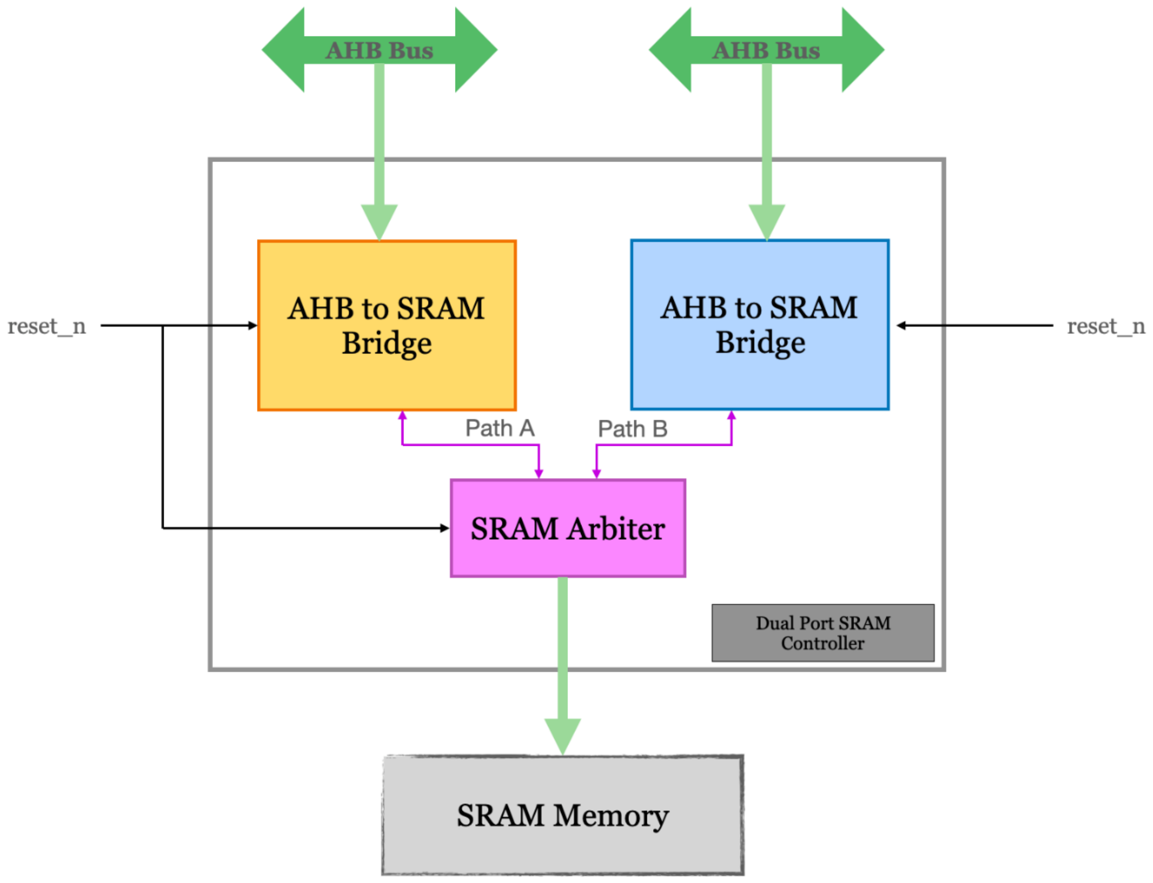


Figure 15. Dual Port SRAM Controller topology

4.2.2. Testbench creation using the generator

A systematic division of creating the UVM environment for this DUT is by creating multiple agents responsible for each interface of the DUT. Five agents are needed: one that provides the clock and reset signals for all components, one for each of the two driving AHB interfaces, another handling the SRAM signals connected to the memory and a last one with the remaining miscellaneous signals. Multiple desired agents need to be described for the generator in the form of a JSON file as displayed in Fig. 16. All signals connecting to the DUT need to be divided and listed according to which agent will control it. An excerpt of the `agents_args` is shown for one agent where the name is the dictionary key. In the case where the cell attribute is allocated, no agent directory is instantiated, and the correct general VC is imported and adopted in the design. Out of the five agents, the generator needs to create the files for only two which correspond to the SRAM interface and the side signals. The others will reference the general clocking and AHB VCs available in the development directory.

```
"X_ahb": {
   "cell": "vc_master_ahb",
   "signals": [
            "X_haddr_i"    ,
            "X_hprot_i"    ,
            "X_hsize_i"    ,
            "X_htrans_i"   ,
            "X_hwdata_i"   ,
            "X_hwrite_i"    ,
            "X_hsel_i"   ,
            "X_hready_i"   ,
            "X_hrdata_o"   ,
            "X_hreadyout_o" ,
            "X_hresp_o"
   ]
},
```

Figure 16: Excerpt from the agents_args JSON file mentioning details about one agent to be created

The complete JSON file is written, and the generator is called to create the VC folder with the resulting structure shown in Fig. 17. Since the DUT requires the transmission of transactions, the complete structure with a passive environment along with a scoreboard is required. The main difference between the output files of this block and the previous one is the subsequent division under `components` for the multiple agents. After successful compilation of the generated VC directory, the tests, the UVM sequences and the checkers have to be added in their corresponding files.

```
|-conf
|  |-vc_X.config.pl
|  |-vc_X_tb.config.pl
|-diaglist
|-docs
|  |-specs
|  |  |-apc_X.json
|-src
|  |-sv
|  |  |-block_tb
|  |  |  |-chipTb.sv
|  |  |  |-X_block_env.sv
|  |  |  |-X_block_pkg.sv
|  |  |  |-X_tb_if.sv
|  |  |-bundles
|  |  |-chip_tb
|  |  |-components
|  |  |  |-vc_side_signals
|  |  |  |  |-vc_side_signals_agent_cfg.sv
|  |  |  |  |-vc_side_signals_agent_if.sv
|  |  |  |  |-vc_side_signals_agent_pkg.sv
|  |  |  |  |-vc_side_signals_agent.sv
|  |  |  |  |-vc_side_signals_driver.sv
|  |  |  |  |-vc_side_signals_monitor.sv
|  |  |  |  |-vc_side_signals_seq_item.sv
|  |  |  |  |-vc_side_signals_seq_lib.sv
|  |  |  |  |-vc_side_signals_sequencer.sv
|  |  |  |-vc_sram
|  |  |  |  |-vc_sram_agent_cfg.sv
|  |  |  |  |-vc_sram_agent_if.sv
|  |  |  |  |-vc_sram_agent_pkg.sv
|  |  |  |  |-vc_sram_agent.sv
|  |  |  |  |-vc_sram_driver.sv
|  |  |  |  |-vc_sram_monitor.sv
|  |  |  |  |-vc_sram_seq_item.sv
|  |  |  |  |-vc_sram_seq_lib.sv
|  |  |  |  |-vc_sram_sequencer.sv
|  |  |-hookups
|  |  |  |-vc_X_hookups.sv
|  |  |-reg_mirror
|  |  |-sva
|  |  |-tests
|  |  |  |-X_arb_test.sv
|  |  |  |-X_base_test.sv
|  |  |  |-X_example_test.sv
|  |  |  |-X_test_pkg.sv
|  |  |  |-seqlib
|  |  |  |  |-vc_X_arb_seq.sv
|  |  |  |  |-vc_X_example_seq.sv
|  |  |-vc_X_env_cfg.sv
|  |  |-vc_X_env_defines.sv
|  |  |-vc_X_env_pkg.sv
|  |  |-vc_X_env.sv
|  |  |-vc_X_if.sv
|  |  |-vc_X_scoreboard.sv
|  |  |-vc_X_vsequencer.sv
```

Figure 17: Resulting VC directory for the SRAM Controller

4.2.3. Further Testbench Development

Stimuli have to be generated from both input AHB agents, driven through the DUT and collected at SRAM interface. No separate sequence must be defined for the AHB agents; however, the constraints are defined for the general random AHB signals regarding the width of the address and the data to be stored in the memory. The constraints are applied to the AHB sequencer as it randomizes these fields and passes them through the driver to the DUT.

Three tests have been written:

- A base test, asserting the reset signals and initializing all input connections to the design block. The first one will always be executed first before any other test to ensure proper functioning of the DUT.
- The second example test consists of generating consecutive random sequences originating from only one AHB bus at a time. The purpose of the test is to ensure that the AHB transaction randomized is properly sent over to the SRAM Arbiter output.
- And the last one tests the arbitration feature by simultaneously starting several UVM sequences from both interfaces, ensuring the prioritized one (Path A) gets handled first.

Further, the scoreboard is developed to collect the transaction items from the monitors of all agents except the clocking one as it is irrelevant.

4.3. Resulting Simulations

The base test initializing all signals is exercised first before the actual test scenario. In all the waveforms in this part, it can be noticed that the signals that vary across the simulations start from 0, indicating that they were reset correctly. Some nonchanging signals are set to a constant value, such as the address and data widths. The results of the sequential test generating sequences from each Path and monitoring them in the testbench are shown below.

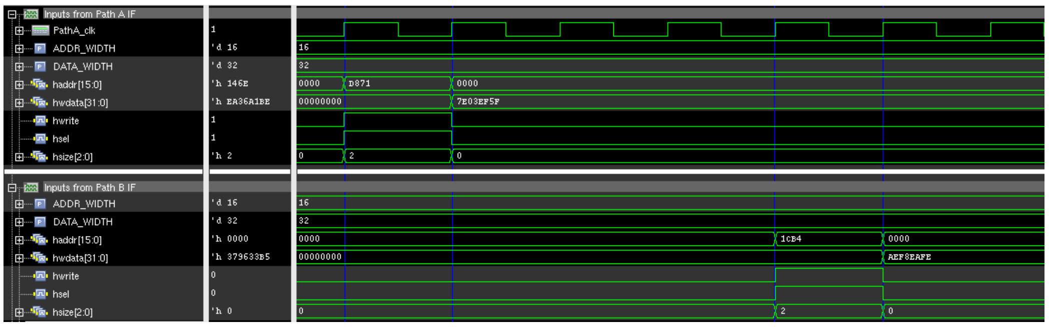


Figure 18: Sequential stimulus generated from each input path in the AHB agent interfaces (IF)

Figure 18 displays a sliced version of the input waveforms that were generated in the UVM environment. The 16 bits address of the AHB transaction from Path A is randomized at the first blue marker, and its 32 bits data one clock cycle later at the second blue marker. Following some delay, the same scenario occurs again from Path B. The size of both transfers is set to 2 indicating 2 data sets of 16 bits each. Each driven transaction is also sent to the scoreboard that will compare the controller's outcome with the expected value from the input.

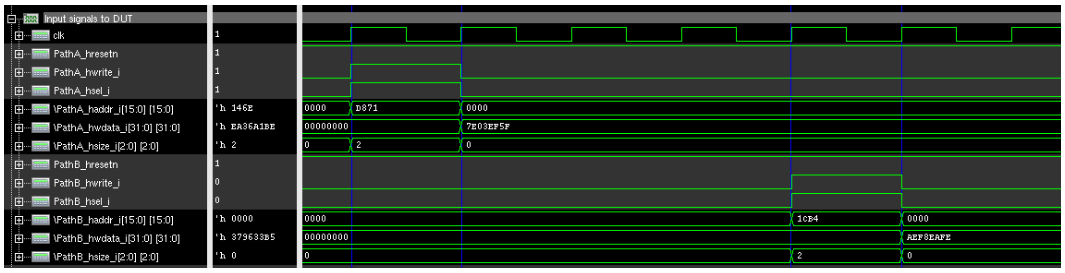


Figure 20: Sequential stimulus received at the DUT input interface from the env

The randomized transaction items are driven from each corresponding agent into the DUT. Figure 19 highlights the correct values of the signals into the DUT. After confirming that the input stimuli are correctly driven, the outputs of the DUT back into the UVM environment are to be checked.

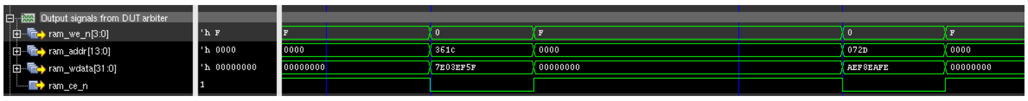


Figure 19: SRAM Controller output of a sequential driven test

Figure 20 highlights the resulting outcome from driving the stimuli onto the DUT. An excerpt of the output signals is presented. Following the address generated at the first blue marker in the environment, the translated address and the data are triggered from the SRAM Arbiter towards the memory in the next clock cycle. The same happens for the second packet from Path B. The flags such as the write enable work properly, as it goes low when the packet is to be written to the memory.

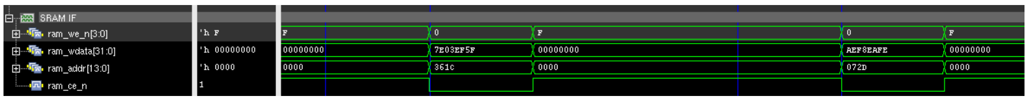
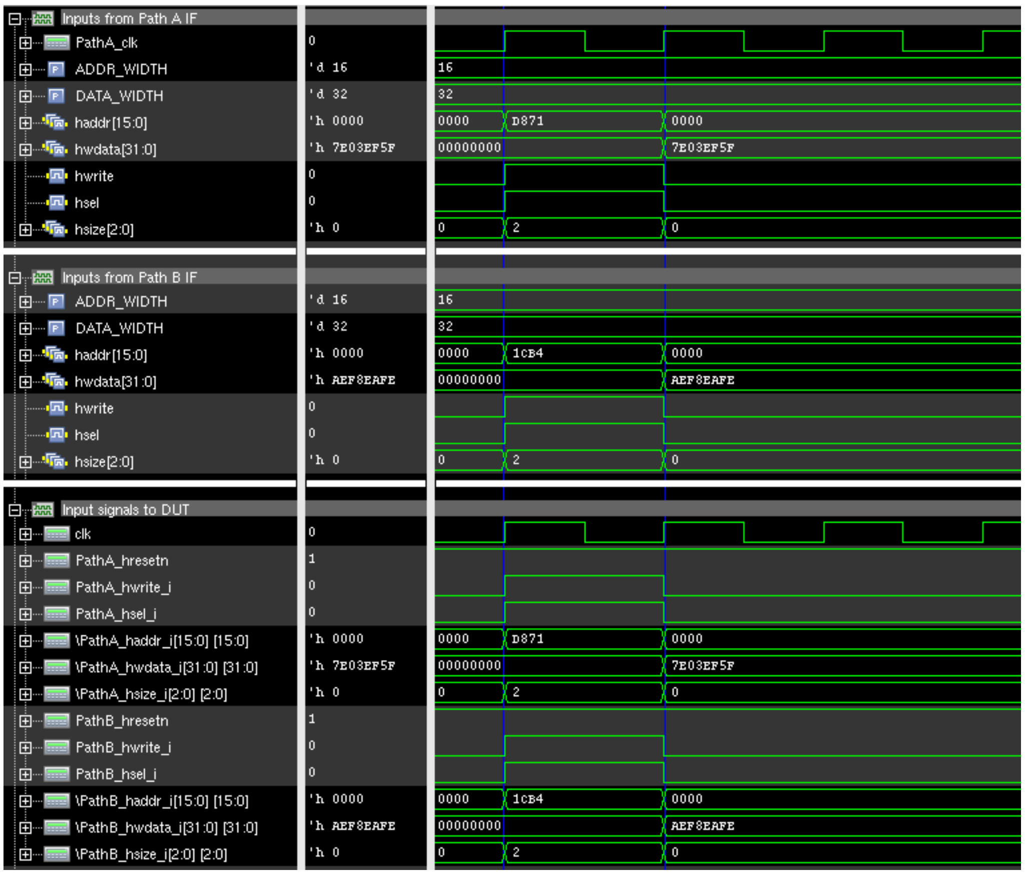


Figure 21: Signals in the SRAM Agent's IF going into the monitor to collect the DUT's outputs

To verify the correct response of the controller, the SRAM output signals are connected to the passive environment to the SRAM Agent. Figure 21 displays the signals collected by the agent's interface into its monitor. The outcome matching confirms the correct structure of the testbench. Each received transaction gets forwarded to the scoreboard. The comparison performed in this UVM component consists of identifying from which path the input was driven, predicting the expected packet, and comparing it with the output collected from the SRAM IF.

Several sequential packets have been driven randomly from both interfaces. The evaluation of the DUT's response confirms that the SRAM arbiter is able to process individual memory accesses from both interfaces.

As the previous test successfully completed, the arbitration feature needs to be tested. For that, another test is developed and a new sequence defined in the library is initiated to the DUT.



Inputs from Path A IF
PathA_clk 0
ADDR_WIDTH 'd 16 16
DATA_WIDTH 'd 32 32
haddr[15:0] 'h 0000 0000 D871 0000
hwdata[31:0] 'h 7E03EF5F 00000000 7E03EF5F
hwrite 0
hsel 0
hsize[2:0] 'h 0 0 2 0
Inputs from Path B IF
ADDR_WIDTH 'd 16 16
DATA_WIDTH 'd 32 32
haddr[15:0] 'h 0000 0000 1CB4 0000
hwdata[31:0] 'h AEF8EAFE 00000000 AEF8EAFE
hwrite 0
hsel 0
hsize[2:0] 'h 0 0 2 0
Input signals to DUT
clk 0
PathA_hresetn 1
PathA_hwrite_i 0
PathA_hsel_i 0
\PathA_haddr_i[15:0] [15:0] 'h 0000 0000 D871 0000
\PathA_hwdata_i[31:0] [31:0] 'h 7E03EF5F 00000000 7E03EF5F
\PathA_hsize_i[2:0] [2:0] 'h 0 0 2 0
PathB_hresetn 1
PathB_hwrite_i 0
PathB_hsel_i 0
\PathB_haddr_i[15:0] [15:0] 'h 0000 0000 1CB4 0000
\PathB_hwdata_i[31:0] [31:0] 'h AEF8EAFE 00000000 AEF8EAFE
\PathB_hsize_i[2:0] [2:0] 'h 0 0 2 0

Figure 22: Parallel stimuli driven into the SRAM Controller for memory access

As can be seen in Fig. 22, different AHB transactions are generated at the same time in both paths and sent to the SRAM controller. Figure 23 shows the signals coming out from the SRAM Arbiter and collected by the SRAM Monitor. The resulting waveforms show that one clock cycle later to the request, the arbiter gives priority to the data packet from the first path as per the design specs. At the second blue marker, the translated address and the packet originating from Path A are forwarded to the scoreboard. After the write enable control signal goes high, the SRAM arbiter can send to the memory the second transaction from Path B. The scoreboard will receive the second packet and perform correct comparison.

The purpose of the test scenarios described above is to validate the UVM verification environment generated from the tool presented in this thesis. Indeed, further tests are needed to ensure the adopted design blocks are fully verified.

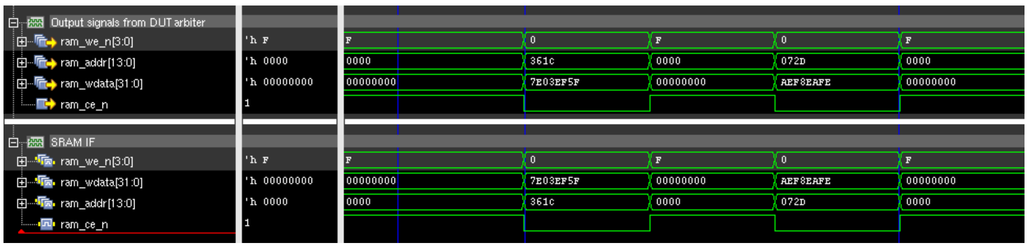


Figure 23: SRAM Controller output and collected signals from the arbitration test

Conclusion

Nowadays, ICs are increasing in complexity as the market demands for continuous design improvements. A more reliable product and a faster design is favorable by manufacturers and by consumers. However, given the arisen complications from the product, companies have to ensure that no bugs are present as early as possible while meeting the time-to-market of the product. This has led DV engineers to take on more responsibility and have a bigger impact in projects, consuming the most resources. Therefore, any possible automation in the verification flow is favorable.

In this thesis, a UVM-based template generator is presented that creates the building blocks of the verification testbench. First, the description of the problem is introduced along with the solution presented in this work. While Section 1 highlights key background information, explaining the product flow, verification of the design and its key figures, and UVM theory. Later, Section 2 covers the design concept of the generator. Moving on to Section 3 that focusses on the actual implementation of the tool and how it will function. It is developed in Python and will rely on UVM SV templates yet to be completed from design parsing. The last part of the thesis presents test cases of using the UVM Generator on some design blocks. Section 4 displays the tests developed to exercise simulations on the DUTs and the outputted verified testbench hierarchy with all its proper connectivity.

The testbench created to verify the blocks allows to speed up the verification flow as less time is invested in creating the environment infrastructure. Therefore, more focus can be done on developing tests, stimuli, and checkers to make sure no verification holes are present, and the design is fault-proof.

Future improvements can indeed be implemented to increase the tool's horizons and make it more flexible.

Future Work

Further development is required for the UVM Template Generator to adopt more advanced features. A Graphical User Interface (GUI) is planned to be implemented. This will allow the DV engineer to interact with more visual design rather than the command line, input agents name and customize more the generated testbench. As the design block grow in complexity and in number of inputs/outputs, it will become tiresome for the user to create the JSON file, copy all signals, and ensure proper division is implemented. Hence, a more user-friendly experience will be developed in which the signals can be dragged or visually selected and separated as desired.

As for the generator's features, currently, the complete testbench is outputted following the UVM standards with all components present. However, in some practical cases, some changes to the structure are anticipated. The aim is to allow the user to choose between the standard structure described in this thesis or a flexible one. In this case, extensive questions will follow inquiring about the desired arrangement with naming and placement.

In addition to that, the development of tests that exercise basic scenarios such as resetting and initializing is planned for.

Bibliography

[1] R. Casanova, “Despite short-term cyclical downturn, global semiconductor market's long-term outlook is strong,” *Semiconductor Industry Association*, 08-Feb-2023. [Online]. Available: https://www.semiconductors.org/despite-short-term-cyclical-downturn-global-semiconductor-markets-long-term-outlook-is-strong/.

[2] K. Salah, “A UVM-based smart functional verification platform: Concepts, pros, cons, and opportunities,” *2014 9th International Design and Test Symposium (IDT)*, Algeries, Algeria, 2014, pp. 94-99, doi: 10.1109/IDT.2014.7038594.

[3] C. A. Mack, “The end of the semiconductor industry as we know it,” SPIE Proceedings, 2003. doi:10.1117/12.506867.

[4] O. Burkacky, J. Dragon, and N. Lehmann, “The Semiconductor Decade: A Trillion-dollar industry,” McKinsey &amp; Company, https://www.mckinsey.com/industries/semiconductors/our-insights/the-semiconductor-decade-a-trillion-dollar-industry.

[5] A. Varias, R. Varadarajan, J. Goodrich, and F. Yinug, “Strengthening the Global Semiconductor Value Chain in an Uncertain Way,” Semiconductors, https://www.semiconductors.org/wp-content/uploads/2021/05/BCG-x-SIA-Strengthening-the-Global-Semiconductor-Value-Chain-April-2021_1.pdf.

[6] G. Wright, “What is Apple? an products and history overview,” WhatIs.com, https://www.techtarget.com/whatis/definition/Apple.

[7] F. Laricchia, “US smartphone market share by vendor 2016-2022,” Statista, https://www.statista.com/statistics/620805/smartphone-sales-market-share-in-the-us-by-vendor/.

[8] “List of Apple products,” Apple Wiki, https://apple.fandom.com/wiki/List_of_Apple_products.

[9] “Apple Vision pro,” Apple, https://www.apple.com/apple-vision-pro/.

[10] Y.-K. Chen and S. Y. Kung, “Trend and challenge on system-on-a-chip designs,” Journal of Signal Processing Systems, vol. 53, no. 1–2, pp. 217–229, 2007. doi:10.1007/s11265-007-0129-7

[11] A. B. Mehta, ASIC/SOC Functional Design Verification a Comprehensive Guide to Technologies and Methodologies. Cham: Springer, 2018.

[12] Y.-N. Yun, J.-B. Kim, N.-D. Kim, and B. Min, “Beyond UVM for practical SOC verification,” *2011 International SoC Design Conference*, Jeju, Korea (South), 2011, pp. 158-162, doi: 10.1109/ISOCC.2011.6138671.

[13] “Methodology Focused Code Generator for UVM,” UVMGen, https://www.uvmgen.com/.

[14] “Introduction to the Easier UVM Coding Guidelines,” Doulos, https://www.doulos.com/knowhow/systemverilog/uvm/easier-uvm/easier-uvm-coding-guidelines/introduction-to-the-easier-uvm-coding-guidelines/.

[15] Hellovimo, “The novel GUI based UVM template generator,” GitHub, https://github.com/hellovimo/uvm_testbench_gen/wiki/The-Novel-GUI-Based-UVM-Template-Generator.

[16] Hjking, “Hjking/uvm_gen: UVM Generator,” GitHub, https://github.com/hjking/uvm_gen.

[17] A. Piziali, Functional Verification Coverage Measurement and Analysis. Berlin: Springer, 2008.

[18] S. Asaf, E. Marcus, and A. Ziv, “Defining coverage views to improve functional coverage analysis,” Proceedings of the 41st annual Design Automation Conference, 2004. doi:10.1145/996566.996579

[19] S. Kumar, S. Shanbhag, M. Mongia and G. Verma, "A systems approach to verification using hardware acceleration," 2nd Asia Symposium on Quality Electronic Design (ASQED), Penang, Malaysia, 2010, pp. 189-193, doi: 10.1109/ASQED.2010.5548241.

[20] P. Tonella, P. Avesani and A. Susi, "Using the Case-Based Ranking Methodology for Test Case Prioritization," 2006 22nd IEEE International Conference on Software Maintenance, Philadelphia, PA, USA, 2006, pp. 123-133, doi: 10.1109/ICSM.2006.74.

[21] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," in IEEE Design & Test of Computers, vol. 18, no. 4, pp. 36-45, July-Aug. 2001, doi: 10.1109/54.936247.

[22] Vlsiverify, “Functional coverage,” VLSI Verify, https://vlsiverify.com/system-verilog/functional-coverage/functional-coverage?expand_article=1.

[23] J. Bhadra, M. S. Abadir, L. -C. Wang and S. Ray, "A Survey of Hybrid Techniques for Functional Verification," in IEEE Design & Test of Computers, vol. 24, no. 2, pp. 112-122, March-April 2007, doi: 10.1109/MDT.2007.30.

[24] “Coverage-Driven Verification Methodology,” Doulos, https://www.doulos.com/knowhow/systemverilog/uvm/easier-uvm/easier-uvm-deeper-explanations/coverage-driven-verification-methodology/.

[25] N. B. Harshitha, Y. G. Praveen Kumar and M. Z. Kurian, "An Introduction to Universal Verification Methodology for the digital design of Integrated circuits (IC’s): A Review," 2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS), Coimbatore, India, 2021, pp. 1710-1713, doi: 10.1109/ICAIS50930.2021.9396034.

[26] "IEEE Standard for Universal Verification Methodology Language Reference Manual," in IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017), vol., no., pp.1-458, 14 Sept. 2020, doi: 10.1109/IEEESTD.2020.9195920.

[27] "ARM AMBA 5 AHB Protocol Specification", ARM Limited, 2015.

[28] "AMBA AXI Protocol Specification", ARM, 2023.

[29] Jinja, https://jinja.palletsprojects.com/en/3.1.x/

[30] K. Truong, "A simple built-in self test for dual ported SRAMs," *Records of the IEEE International Workshop on Memory Technology, Design and Testing*, San Jose, CA, USA, 2000, pp. 79-84, doi: 10.1109/MTDT.2000.868619.

[31] A. Sirohi, A. Pratap, and M. Aggarwal, “Improvised round robin (CPU) scheduling algorithm,” *International Journal of Computer Applications*, vol. 99, no. 18, pp. 40–43, 2014. doi:10.5120/17475-8361