POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Reinforcement Learning Framework for RISC-V Functional Verification

Supervisors
Prof.ssa Mariagrazia Graziano
Prof. Maurizio Martina
Dr. Andrea Marchesin
Dr. Michele Caon

Candidate
Marco Rosa Gobbo
ID: s309578

Academic year 2023-2024

Summary

Verification in the context of digital design is the process of testing and validating the behavior of a system before it gets released or deployed. This is a fundamental part of the design process, often taking more than half of the development time due to the complexity of reaching complete coverage. Traditional verification techniques, such as directed testing and constrained random testing, often fail to capture critical edge cases in complex systems. To address this gap, this thesis explores the application of Reinforcement Learning (RL) for the functional verification of RISC-V cores, which are becoming increasingly popular, specifically through the automatic generation of assembly code to enhance test coverage. This investigation begins by building a test-bench for RISC-V cores intended to be as implementation-independent as possible using the Universal Verification Methodology (UVM) in SystemVerilog (SV) and Spike instruction set simulator as the gold model. The test-bench is then translated into a Python-based environment using the *PyUVM* library and Verilator as the simulator to enable an open-source setup. This facilitates the integration with the rest of the components needed in the flow, such as the custom instruction generator and the coverage collection, providing a flexible framework for closed-loop instruction generation and core state observation.

We introduce at this point the RL agent to guide the instruction generator based on coverage metrics and Central Processing Unit (CPU) state (e.g., register file and program counter). Because of the action space being so vast and never tackled before by other research works, the first agent implementation involves a custom-built RL agent, relying on Gymnasium to have a standard API towards the environment. It uses a deep Q-learning agent based on Neural Networks as the function approximators, divided in a state encoder and specialized child Neural Network (NN) to avoid the explosion of the Action space size. The second approach uses StableBaseline 3 (SB3) library that provides established RL algorithms, including Proximal Policy Optimization and Multi-Input Policy. For both cases different state vectors and reward functions are experimented with.

Finally, we compare the post-training results obtained by the RL agent to the average coverages obtained by requesting random instructions to the instruction generator. The first agent approach does not show any improvements due to the NN not converging, caused by a naive implementation of the neural networks which

leads to exploding weights and the loss values not decreasing. The second, SB3 approach, shows encouraging results. For instance, with 100 requests to the instruction generator (ca. 200 assembly instructions), an average coverage increase of 4.2% is observed compared to the random generation. The RL agent is able to generate diverse instruction sequences that stress different areas of the processor and show the presence of data dependencies in the generated code, thanks to the reward function promoting these behaviors.

The work done provides a solid foundation for future research while already having tackled some of the implementation options that showed a more efficient approach. The fully open-source nature of the framework represents a performant and versatile basis to further explore other machine learning approaches compared to proprietary solutions.

Acknowledgements

First and foremost, I would like to thank my parents, the two people who have always been by my side and offered support at every moment. I also extend my gratitude to my aunt for her constant interest in both my academic and personal career. I thank all my friends who patiently endured my ever-changing moods and pessimistic outlooks on a daily basis.

A heartfelt thank you to the PhD students without whom this work would not have been possible: Vincenzo, Flavia, Andrea, and Michele. I am also grateful to Professor Graziano and Professor Martina, through whom I had the opportunity to undertake this thesis.

Contents

List of Tables

List of Figures

Acronyms

MLP	Multi-Layer Perceptron
SB3	StableBaseline 3
TB	Test-Bench
SV	SystemVerilog
ML	Machine Learning
UVM	Universal Verification Methodology
OVM	Open Verification Methodology
RL	Reinforcement Learning
MDP	Markov Decision Process
SDF	Standard Delay Format
HDL	Hardware Description Language
RTL	Register Transfer Level
IP	Intellectual Properties
TLM	Transaction-Level Modeling
CSR	Control Status Registers
DUT	Device Under Test
ISS	Instruction Set Simulator
ISA	Instruction Set Architecture
NN	Neural Network
OOP	Object Oriented Programming

DB	DataBase
CER	Combined Experience Replay
PRB	Prioritized Replay Buffer
TD	Temporal Difference
API	Application Programming Interface
PPO	Proximal Policy Optimization
FIFO	First In First Out
FSM	Finite State Machine
PC	Program Counter
UCDB	Unified Coverage DataBase
GUI	Graphic User Interface
ALU	Arithmetic logic unit
SW	Software
ASM	Assembly
CPU	Central Processing Unit
OBI	Open Bus Interface
GPU	Graphics Processing Unit
EDA	Electronic design automation
TCL	Tool Command Language
PSL	Property Specification Language
SVA	SystemVerilog Assertions
WB	Weights and Biases
ReLU	Rectified Linear Unit

Chapter 1

Introduction

The growing number of extensions and the modular nature of RISC-V cores brought to light the limitations of traditional verification methods when the addition of new instructions requires a rapid re-verification of the core. This thesis, starting from a more traditional UVM test-bench wants to explore the use of machine learning and in particular reinforcement learning to aid in the creation of test programs for the functional verification of the cores. In this introductory chapter the motivation behind this work as well as some notes on the history that led up to the current state of the art in the verification field.

1.1 Motivation

Verification plays a crucial role in the design of digital systems, often consuming more than half of the total design time [9] as it can be seen in fig. 1.1.

The complexity of modern digital systems has grown exponentially over the years, with many designs now incorporating billions of gates, Moore's remains true for the number of transistors used in a chip [10], in fig. 1.2 we can see how this is true not only for CPUs but for the other designs too. This increased complexity has made the verification process more challenging and time-consuming. Engineers must meticulously test every aspect of the system to ensure that all components work together seamlessly and that the system as a whole performs as expected under all possible operating conditions. The substantial time devoted to verification is justified by the potentially catastrophic consequences of errors in the design or microarchitecture. Fixing issues after an Integrated Circuit (IC) has been manufactured or deployed can be extremely costly, both in terms of financial resources and damage to a company's reputation. One of the most famous cases was the bug that afflicted the Pentium 4 divisor and that highlighted issues in the verification process [11]. In some critical applications, such as medical devices or automotive systems, errors could even lead to life-threatening situations. Therefore, thorough

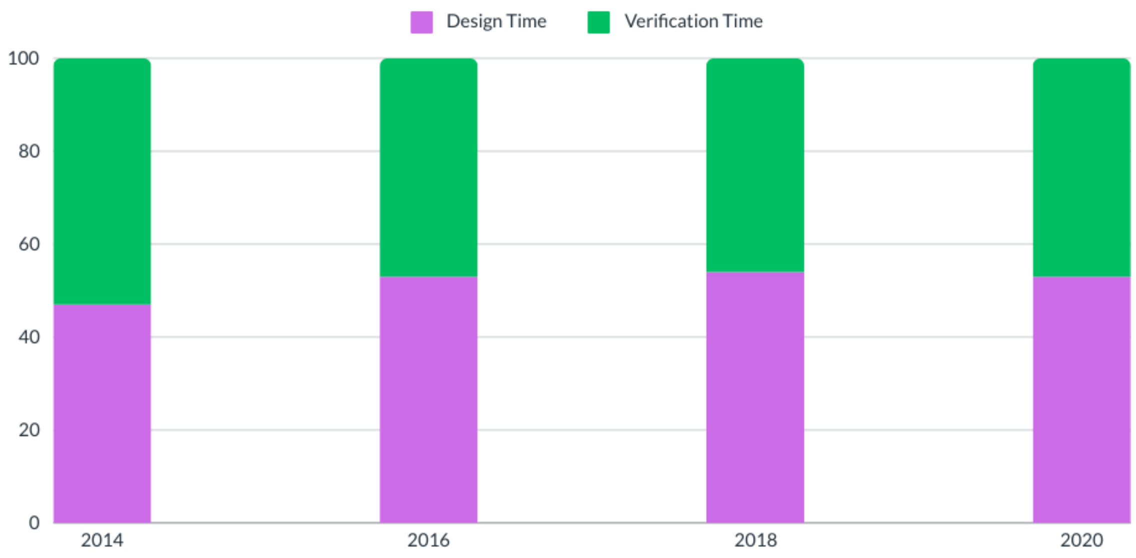

Figure 1.1: Mean time spent for verification compared to design.
Source: Wilson Research Group and Mentor.

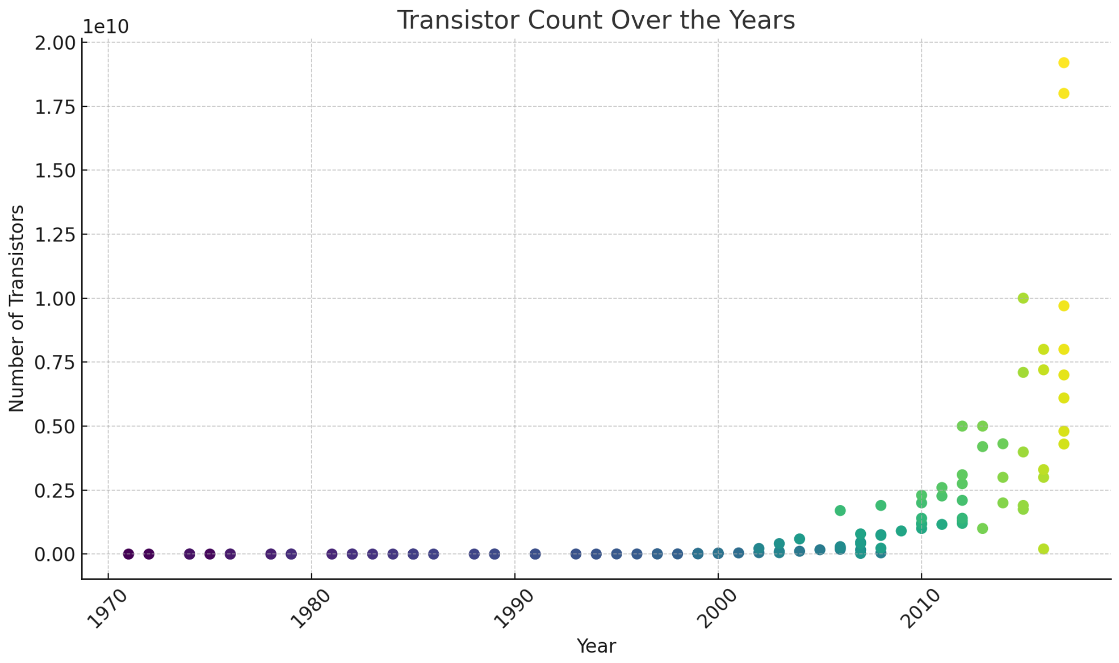

Figure 1.2: Transistor count for CPUs over the years. Data source: Wikipedia [1]

verification is essential to catch and correct any issues before they can cause harm or financial loss. Verification occurs at multiple levels throughout the design process. It begins with the verification of individual components and modules, progresses to the integration of these components, and ends with system-level verification. This multi-level approach ensures that not only do individual parts work correctly, but they also function properly when integrated into the larger system, this is becoming more and more important as Intellectual Propertiess (IPs) get reused for multiple projects. Traditional verification methods, such as directed testing and constrained random testing, have been the workhorses of digital system verification for a long time. Directed testing involves creating specific test cases to verify particular functionalities or corner cases of a design. This approach is precise and targeted but requires significant manual effort to create comprehensive test suites. Verification engineers must anticipate potential issues and create tests to expose them, which becomes increasingly challenging as system complexity grows. Random testing, given enough time, is the most complete verification method, the problem being that the time required explodes with size of the device to test, in this case, constrained random testing still generates a large number of random test cases but within specified constraints that limit the number to the interesting scenarios. However, it may still miss rare corner cases and can be inefficient in terms of the number of tests required to achieve adequate coverage. While these methods have served the industry well, they are showing their limitations with bigger and more sophisticated designs. In this thesis, we are working with a CPU core, a prime example of a complex digital design, more specifically, with a small RISC-V core. The emergence of the RISC-V architecture has added a new dimension to the verification challenge. RISC-V has gained significant traction in the processor design landscape, largely due to its open-source nature and modular design. These characteristics allow for rapid innovation and customization, but they also introduce new verification complexities. RISC-V's modular approach means that cores can be customized with various optional extensions and custom instructions, while this flexibility is a strength of the architecture it also multiplies the number of possible configurations that need to be verified. Each combination of extensions and custom features potentially introduces new interactions and edge cases that must be thoroughly tested. As RISC-V cores find applications across a wide spectrum, from tiny embedded systems to high-performance computing, the verification requirements become even more diverse. The open-source nature of RISC-V also means that many different organizations and individuals are developing cores and extensions, as opposed to the standard cores provided by ARM or other IP producers. These cores often incorporate advanced features such as out-of-order execution, speculative processing, and complex cache hierarchies. These features create a vast state space of possible behaviors, making it extremely difficult for traditional methods to achieve exhaustive verification. This variety and modularity together with increasing time-to-market importance, fast but reliable verification gains more importance, as shown in fig. 1.3 all the time that can be saved translates to possible earnings [12].

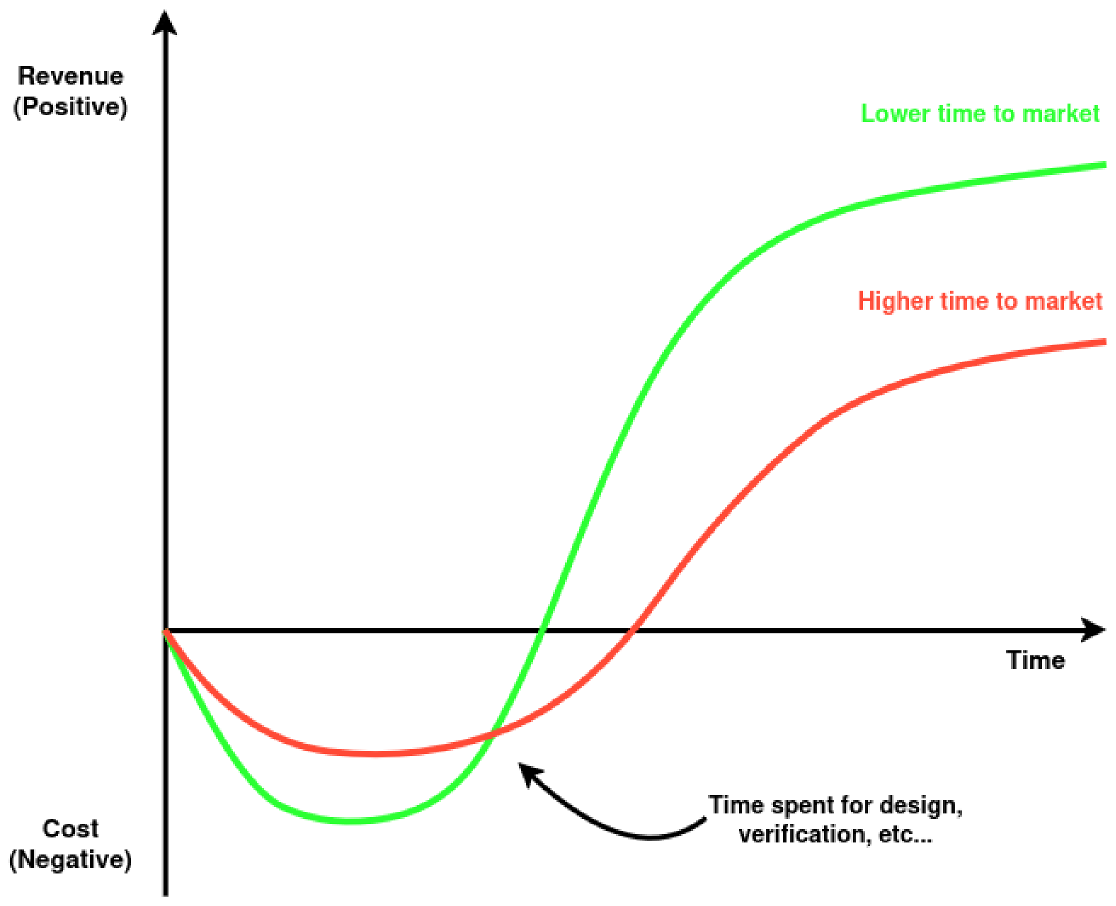

Figure 1.3: Relationship between time to market revenue and cost. Source: Tcgen [2]

Perhaps most critically, the classic methods mentioned before may fail to uncover subtle corner cases or rare scenarios that could lead to critical failures in the field. Such scenarios might only occur under very specific conditions that are hard to anticipate or reproduce in a test environment. While this promotes innovation, it also raises the importance of robust, standardized verification methodologies to ensure interoperability and reliability across the ecosystem.

1.2 Historical notes on verification

Originally, for simple circuits, the verification process was relatively straightforward, consisting in engineers manually writing the input vectors for the test. As designs became more complex, this manual approach quickly proved inadequate [13]. The advent of Hardware Description Language (HDL) like Verilog and VHDL in the 1980s marked a significant shift in both design and verification methodologies. These languages allowed engineers to describe hardware at a higher level of abstraction and enabled the use of simulation for verification. Engineers could now write test-benches in the same language as the design, providing a more integrated and scalable approach to verification. The 1990s saw the introduction of **coverage-driven** verification, a concept that emphasized measuring the completeness of verification efforts. Tools emerged that could analyze code coverage, providing metrics on how thoroughly a design had been tested. This shift from a purely functional verification approach to one that also considered coverage marked a significant advancement in the field. As we entered the 2000s, the exponential increase in design complexity led to the development of constrained random testing. This technique allowed engineers to generate large numbers of random test cases within specified constraints, helping to uncover corner cases that might have been missed by manually created tests. The ability to automatically generate diverse test scenarios greatly expanded the scope of verification efforts. In the mid-2000s assertion-based verification was introduced, where engineers could embed checks directly into their designs or test benches. Languages like Property Specification Language (PSL) [14] and SystemVerilog Assertions (SVA) [15] were developed to support this methodology, allowing for the automatic detection of violations during simulation. At this point, an additional level of abstraction was introduced through the collaborative efforts of Mentor Graphics and Cadence Design Systems, leading to the development of the Open Verification Methodology (OVM), an interoperable SystemVerilog-based verification framework. OVM provides a standardized library of base classes, enabling users to build modular and reusable verification environments. In these environments, components communicate using Transaction-Level Modeling (TLM) interfaces. This approach promotes both intra- and inter-company reuse by offering a common methodology, with classes that support the development of stimulus sequences and facilitate the reuse of components from block-level to system-level verification. The next step and current industry standard is the UVM that has been introduced and standardized by IEEE, it's an extension of OVM. UVM [16] that further refines the concepts of modularity and reuse, offering enhanced features for stimulus generation, coverage, and transaction-level modeling, making it the industry-standard methodology for creating scalable and reusable verification environments.

1.2.1 State of the art for verification

As UVM has already been consolidated in the verification panorama new tools and methodology are being built as an addition of substitution to it. This can be considered the state of the art for verification in the case of complex digital systems such as RISC-V cores, these tools do not overlap completely with what is then going to be proposed in the thesis but are what is being used in the industry at the moment

The officially used tools for Open-HW cores, of particular interest since in the thesis the device tested is a core from group (a non-profit organization developing open-source cores), employ "smart" random instruction generators capable of creating coherent instruction sets, but the generation parameters are still manually set. The following frameworks are the most used and that have support for a number of cores.

RISCV-DV

RISCV-DV is an open-source instruction stream generator for RISC-V processors, part of Google's DV project aimed at providing a complete verification infrastructure for RISC-V cores. Built on an SV/UVM infrastructure, it supports a wide range of RISCV extensions that are going to be explained more in detail in the background chapter section 2.4, including RV32IMAFDC and RV64IMAFDC instruction sets, multiple privileged modes (Machine, Supervisor, and User), page table randomization, privileged Control Status Registers (CSR) setup randomization, trap/interrupt handling, and more. The system can be easily customized via YAML files, as shown in fig. 1.4. Generated instructions are fed to both an Instruction Set Simulator (ISS) and the Device Under Test (DUT), with outputs compared at the end of the simulation. Functional coverage can be collected directly from the ISS, simplifying DUT test-bench development. The project supports various ISS and Register Transfer Level (RTL) simulators, configurable via YAML files. At its core, RISCV-DV uses a Python-based random instruction generator built on PyVSC, a library for random stimulus generation and coverage collection.

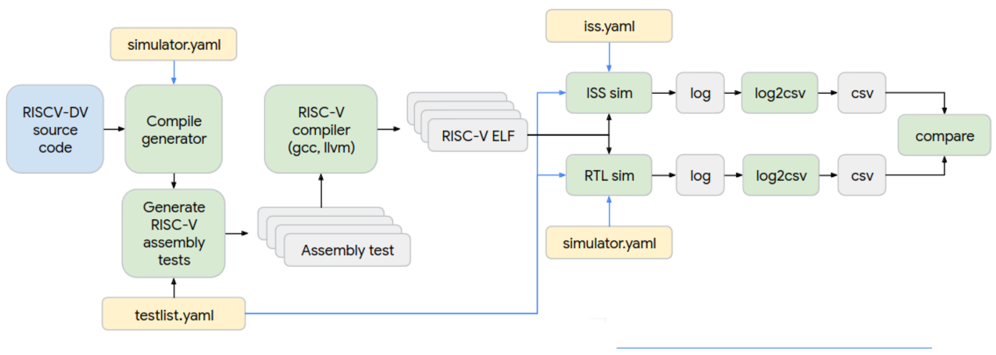

Figure 1.4: RISCV-DV architecture. [3]

FORCE-RISCV

FORCE-RISCV an open-source random instruction generator for RISC-V processors, is part of the OpenHW Group. It uses randomization to generate valid test sequences by selecting instructions, registers, and addresses. The generation flow is controlled through a set of python APIs, with output provided as standard *.elf* and assembly files. FORCE-RISCV supports numerous RISC-V Instruction Set Architecture (ISA) features, including RV64G, RV32G, V extension 1.0, RISC-V privileged ISA (U, S, and M privilege levels), and various virtual memory systems (Sv48, Sv39, Sv32) [17]. It also offers fast exception handling, non-trivial exception handlers, full privilege mode switching support, and multiprocess/multithread instruction generation. The generator incorporates an ISS to model instruction behavior. After generating each instruction, it's executed on the ISS, and FORCE-RISCV updates the appropriate architectural state based on the output.

riscvISACOV

The riscvISACOV [18] project aims to provide a common infrastructure for functional coverage of RISC-V cores. As the ISA is consistent across cores, the coverage model can be shared, although interrupt handling, exceptions, and CSR registers may vary between cores. Led by Imperas and following the OpenHW ARVM-Functional Coverage project, riscvISACOV is not a complete verification environment but significantly aids test-bench development by providing the coverage component.

1.2.2 Beyond UVM

In recent years, Python and Cocotb (coroutine-based co-simulation test-bench) have emerged as a valid alternative to the standard SV based test-benches in hardware verification, offering flexibility and ease of use compared to traditional methodologies like UVM. Cocotb is a Python-based framework that allows users to write test benches in Python, while co-simulating with HDLs simulators. This approach makes use of Python's simplicity and rich ecosystem, enabling faster test development and higher productivity. Cocotb also facilitates the integration of high-level software testing practices into hardware verification, allowing for more efficient testing, faster iterations, and greater accessibility for software engineers involved in hardware verification.

1.3 Historical notes on Machine Learning

Thanks to the enormous quantity of data and computing power available in recent years, machine learning has evolved exponentially [19] transforming from a theoretical concept to a cornerstone of modern computing. The real research in machine learning began in the 1980s, as researchers looked for better ways to model learning algorithms. Early work, such as neural networks had difficulties in achieving practical success due to limited computational power and the lack of sufficient data.

However, the 1990s saw a shift towards more robust statistical methods like decision trees and support vector machines, which provided more reliable performance across various tasks.

The 2000s marked a turning point with the rise of big data and improvements in hardware, which made it feasible to train complex models on vast datasets. The resurgence of neural networks, particularly deep learning, played a pivotal role. Innovations in training techniques, such as back-propagation, allowed neural networks to solve more intricate problems. Deep neural networks are used nowadays to solve complex problems as predicting the structure of proteins, autonomous driving, image recognition and natural language processing.

A particular branch that is then going to be explored more in detail in the background and implementation sections is **reinforcement learning**, which, in recent years, has gained significant attention, especially with the integration of deep learning techniques. The success of algorithms like Deep Q-Networks and models such as AlphaGo [20] demonstrated the potential of reinforcement learning to solve complex, decision-making problems in dynamic environments. These developments, along with advancements in areas like unsupervised learning and transfer learning, have established machine learning as a critical tool in various fields ranging from healthcare to autonomous systems and of particular interest for us in verification where for example can be used to cluster the violations discovered in a design and get to the root cause [21].

1.4 Thesis organization

Having now introduced the arguments that are going to be developed here is how the thesis is going to be organized: in the background chapter an overview of the theory leveraged to construct the framework that is needed to have a complete vision of the thesis can be found, more specifically we are going to look at UVM, the open source scene in verification, the RISCV ISA, what reference model we are going to use and finally some reinforcement learning and in general machine learning bases. The implementation chapter is divided into 5 sections, going over the coding of the base test bench as a standalone unit, how we collected the coverage values, the different implementations of the RL agent and environment, our custom instruction generator, and finally how the whole is glued together in the simulation environment. As for the results chapter, we take a look at 3 main points: how the base test benched performed, if and why the RL agent converged to meaningful results, and lastly how well it performed compared to random instructions. Finally, in the conclusions, we are going to look at the problems, solutions and possible future work regarding this project.

Chapter 2

Background

In this chapter an introduction to all the major concepts and technologies needed to understand what the thesis is built on is illustrated, ranging from the traditional UVM methodology to the Machine Learning (ML) models then used in the implementation.

2.1 UVM

2.1.1 UVM motivations

As mentioned in the introduction UVM is an IEEE standard to implement a methodology for verifying digital circuit designs. This standard is implemented via different libraries, the most used one being the Accellera System Verilog one, supported in the latest versions of most HDL simulators. This implementation consists in a set of base classes with methods defined in it, from which the System Verilog verification environment can be developed by extending the base classes.

2.1.2 UVM structure

Figure 2.1 shows the most important components of a typical UVM test-bench. Going from the bottom up we have the DUT, which is the entity that is actually being tested throughout, usually written in in Verilog or other HDLs. The I/O of this device is described at the signal level and for this reason the *Interfaces* are introduced, a way to translate the signals into more abstract transactions. From the test-bench side ideally only `sequence_items` are seen, they will represent a set of inputs or outputs of the DUT. The sequence used depends on the test and could be a fixed one as well as a sequence able to adapt and respond to signals read by the *Monitor* coming from the device under test. These packets are sent to the DUT via the *Driver*, which asks the *Sequencer* for the next sequence item to be driven. The driver translates the information from the more abstract Transaction Level

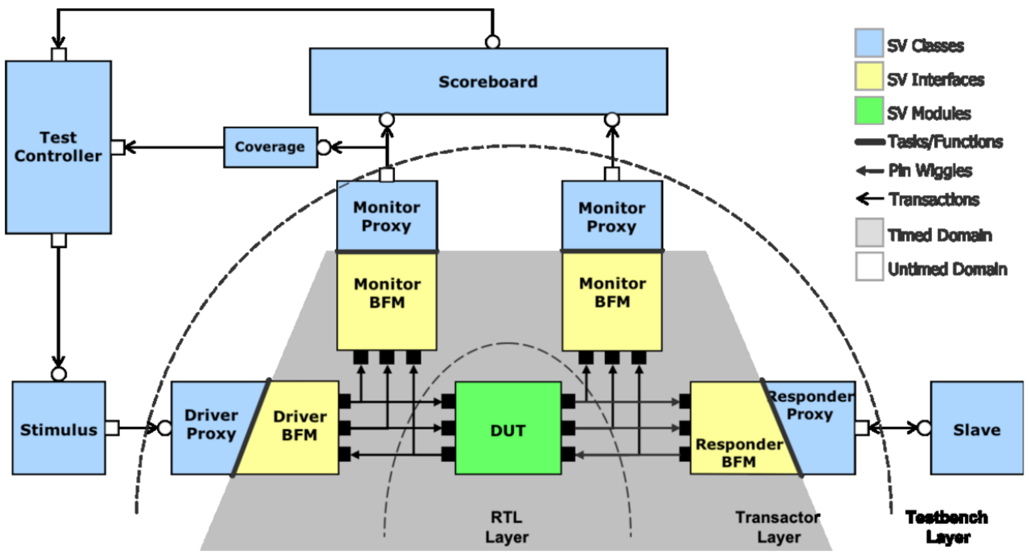

Figure 2.1: UVM TB structure, source: UVM cookbook [4].

used in the packets to the actual signals to drive the DUT. The *Sequencer* is the component that is in charge of the scheduling of the sequence items, it can be seen as a sort of arbiter that decides which sequence item to send to the driver. The *Monitor* is a passive component that samples the signals of the DUT and creates the relative transaction items, as opposed to the driver here it's going from the lower signal level of the interface to a more abstract transaction level of the items, that are then used by the coverage and scoreboard components. A reference model to compare our DUT to is required in the framework as well, this is called the *predictor* and can be implemented in various ways, for example writing it directly in SytemVerilog or using an external function in C [22] or another source like a simulator that emulates the desired behavior of the DUT. The *Scoreboard* is the component that compares the results of the DUT with the ones obtained from the reference model.

Moving to a different view of the test-bench in fig. 2.2 a hierarchy of the UVM classes can be seen.

The *Agent* is a collection of the sequencer, driver and monitor and it is used to simplify the instantiation of the test-bench. The *Environment* is the top-level component that instantiates all the agents and the scoreboard and it is the component that is instantiated in the test-bench, this can be seen more clearly in fig. 2.2. The *Test* is the component that drives the environment and it is the one that is created in the test-bench. This hierarchy allows having for each "Top" test-bench a set of Tests with their relative sequences and different environments as to maximize the reuse of code.

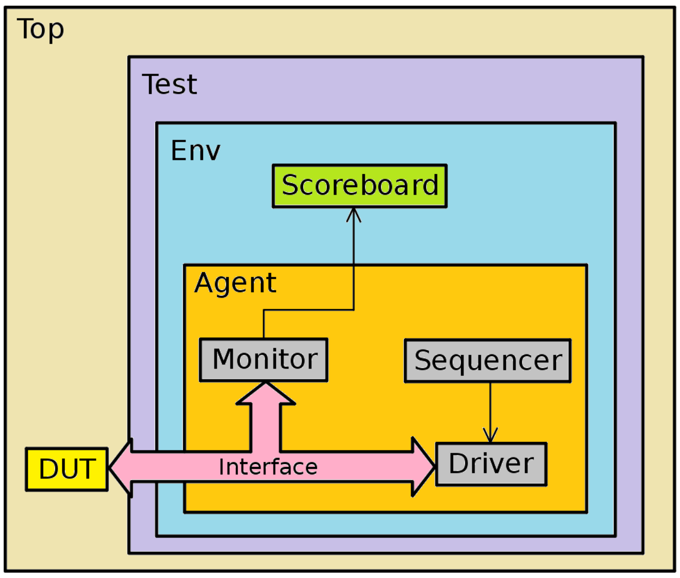

Figure 2.2: UVM hierarchy [5].

2.1.3 UVM Communication

Communication is handled through a combination of hierarchical structuring, standard interfaces and transaction-based communication. The standard interfaces define a set of methods, signals and transactions that allow components to exchange information in a consistent manner. Components connect to each other through these ports and exports which in TLM act as interfaces between modules and handle communication at a higher abstraction level, typically in terms of transactions rather than individual signals declared within the components themselves. Transactions are the fundamental units of communication in UVM, each represents a piece of data that is sent from one component to another through the interface communication channels. UVM components can also use event notification mechanisms and callbacks to communicate, for instance, a monitor might raise an event when a certain condition is observed and a scoreboard might register a callback on that event to process the data.

2.1.4 UVM Phases

All the components derived from *uvm_component* go through a predefined set of phases, that act as a synchronization mechanism during the simulation. The phases are defined as callbacks so the components can do work in the callback phase period. Methods that do not consume simulation time are the functions and the methods that do are tasks. The phases can be grouped into three main categories:

1. **Build and connect phases**: in this phase the components are created and

the connections between them are established. The build phase is the first phase that is executed and it is executed in a top-down order. The connect phase is used to set up the connections between the components, this is done bottom-up.

2. **Run phase**: actual time-consuming simulations, it runs in parallel with the other run-time UVM phases.

3. **Clean-up phases**: here the data from the scoreboard is extracted and computed, the check on the outputs is performed and the results are printed out.

2.1.5 Additional UVM features

On top (or in parallel) to the already described classes there are additional features provided by the UVM library to help the verification process. The **UVM Factory** mirrors the Object Oriented Programming (OOP) concept of a factory, it can be used to create and configure objects at run-time to be able to override the default configuration of the test-bench. The **UVM Database** provides an interface for the resources facility, it simplifies the passing of data between components and it is used to store the configuration of the testbench. All operations related to the database are static, in the DataBase (DB) the resources are stored by a given name (or type) and they can be retrieved by the same name (or type) in the correct scope.

2.1.6 UVM good practices

Being a mature methodology UVM has a set of best practices that are recommended to be followed to maximize the efficiency of the verification process. Some may feel like over-complications but they are generally useful to avoid common pitfalls and to make the code more readable and maintainable.

2.2 Spike and reference model

A general problem in verification is what to consider "known good" especially in not trivial cases such as a CPU core. Usually there are different possibilities such as comparing to an already existing and proven design, writing a model at a higher abstraction level or using formal methods to verify the correct result. In the case of a core writing a model to emulate the whole behavior of it becomes a challenging task and prone to error, for this reason one can use an ISS to abstract the correct behavior the core should have when provided with code. ISSes are not cycle accurate but work with the minimum granularity of a single instruction (commit) so they are not usable in case of extensive verification of the sub-modules of a core but in case of functional verification (as is going to be our case) they are the ideal choice.

2.2.1 Spike

Spike [23] is one of the most widely used, open-source ISS that can be found. It acts as a reference model for the RISC-V ISA (Instruction Set Architecture) and is developed by the RISC-V Foundation, it's mainly used for testing, simulating, and verifying the functionality of RISC-V processors, and it provides a platform for running RISC-V programs without requiring actual hardware. It supports all the official ISA extensions mentioned in the previous sections and over that the exceptions and interrupts. It comes with a debugging mode for step-by-step simulation of a program and new instructions can be easily added to the set of supported ones.

2.3 Open source verification

For software, open source means that the source code is free for anyone to see, modify and improve, in the last years such software has started to appear in hardware design and verification too, comprehensive guides and resources can easily be found, for example this list that groups together verification frameworks and tools available currently. As mentioned in the introduction Python as a verification language is gaining traction in the verification community and along with it open-source alternatives for RTL simulators are getting more popular.

2.3.1 Verilator

Verilator is one of the aforementioned powerful, widely-used free and open-source tools designed for high-performance simulation of digital circuits described in SystemVerilog and Verilog. It translates HDLs designs into C++ or SystemC, creating models that can be executed faster than traditional interpreted simulators. This speed is especially useful in large, complex designs or when performing extensive regression testing, as the generated models can take advantage of multi-threading and optimized execution paths [24]. One of the key advantages of Verilator is that it often rivals, or even surpasses, the performance of proprietary simulators. This makes it highly attractive for projects where licensing costs and scalability are concerns. Verilator cycle-based simulation methodology is ideal for certain use cases, though it does fall short in features typically found in event-driven simulators. For example Standard Delay Format (SDF) annotation, which is used to back-annotate timing information from synthesis or place-and-route tools into the simulation, which is crucial for timing-aware verification. This happens to not be a problem for our scope while the issues emerge for two different aspects:

- Incomplete UVM support: since Verilator's support for UVM is still under development, it means one needs to create the test bench outside of SystemVerilog.

- Limited coverage capabilities: while Verilator offers basic coverage support, it doesn't match the options provided by commercial tools like QuestaSim. In scenarios where achieving and tracking functional coverage is a primary requirement, Verilator might require additional manual effort or external tools to fill this gap.

2.3.2 Fusesoc, Edalize and the build tools

Historically and still today, the development of hardware design flows heavily relies on the use of custom scripts, often written in Tool Command Language (TCL) or bash, to manage the compilation, simulation, and synthesis. TCL scripts provide designers with flexibility and control over tool flows. However, they also come with challenges. Handwritten scripts are often difficult to maintain, error-prone, and tightly coupled to specific tools, limiting portability and reuse across different projects and environments. As projects grew in complexity, the need for more standardized, scalable, and modular approaches became evident. TCL and other scripting languages lacked native support for package management and dependency tracking, making it hard to manage IP blocks and external libraries in larger designs. Furthermore, the manual nature of script-based workflows can introduce inconsistencies, as different teams or individuals might use slightly different scripts or configurations for the same project. This led to the development of more sophisticated build systems and automation frameworks, which aim to provide a higher-level abstraction over the low-level tool invocations that TCL scripts typically handled. *FuseSoC* [25] and *Edalize* emerged as part of this evolution. FuseSoC, as a package manager and build system, abstracts much of the scripting *FuseSoC* in setting up toolchains and managing dependencies. *Edalize*, as the backend tool driver, replaces custom scripts with a uniform interface for various Electronic design automation (EDA) tools. Together, they offer a solution that not only automates the flow but also ensures portability, reusability, and ease of integration across multiple tool environments. In the context of this thesis *Fusesoc* is going to be used for the compilation, simulation and handling of the Verilator outputs simplifying the overall flow compared to the combination of bash scripts used initially.

2.3.3 Python verification

Python as a verification language alleviates the lacking support of UVM in Verilator. As mentioned in the state of the art in the last years Python verification has started to gain traction, this is due to multiple reasons, that can be summarized in:

- Being Python a highlevel language means that it is easier to write and read than other languages like SystemVerilog or VHDL.

- Python is a generalpurpose language, which translates to a wider range of applications where it can be used, allowing additional features to the Test-Bench and the model.

- Writing reference model for complex DUTs is easier in Python thanks to the highlevel constructs and the vast number of libraries available.

But how can the Python test bench interact with the DUT? Here is where the CocoTB library comes in. Cocotb is a co-routine-based co-simulation library for writing Python test-benches for VHDL and Verilog. It is a wrapper around the simulator that allows the Python code to interact with the device we are testing; it's completely free and open-source (BSD license) and it offers built-in support for integration with continuous integration tools like Jenkins, GitLab, etc.

Let's briefly see how a Cocotb test-bench works: as it can be seen in fig. 2.3 there is no need for any user-written wrapper code on the HDL side, Cocotb drives stimulus onto the inputs of the DUT and monitors the outputs directly from python. The HDL code instantiation has to be handled externally by the simulator. Then the test is just a Python function, the *await* keyword is used to pass the control back to the simulator to advance the simulation time when the python code is not executed. A single test can spawn multiple co-routines that can run concurrently and independently thanks to the *start* and *start_soon* function calls.

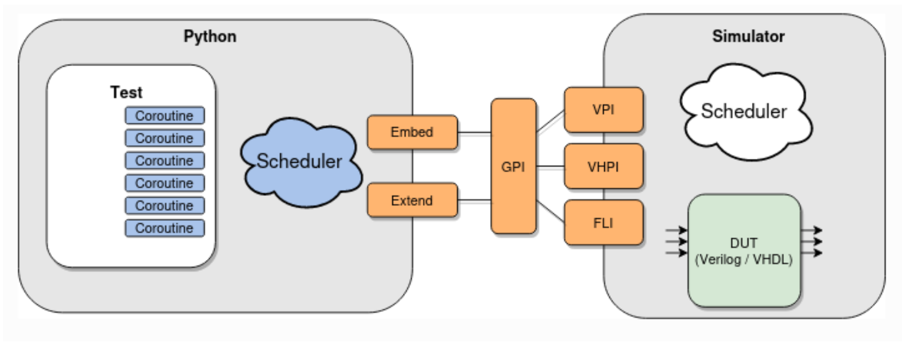

Figure 2.3: Cocotb DUT interaction, source: CocoTB [6].

2.4 RISC-V

RISC-V is an open standard of instructions, ISA, based on the Reduced Instruction Set principle. The project, born in 2010 at Berkeley University [26], is now a significant player in the hardware industry. It has grown into a robust and versatile architecture, supported by a global community of researchers, developers, and companies. RISC-V offers several advantages, such as simplicity, modularity, and extensibility, making it suitable for a wide range of applications from embedded systems to high-performance computing. The open nature of RISC-V allows for innovation and customization, promoting a collaborative environment where advancements in processor technology can be shared and utilized widely.

2.4.1 CVE2 core

This is the core we mainly used during the development of the test-bench, it's one of the smallest OpenHW cores, forked from the Ibex core. It targets low-cost embedded applications and is based on a two-stage pipeline. The cores we are verifying come from the OpenHW group, a non-profit organization where both hardware and software designers work together to develop open-source cores, the related software and tools. The core is heavily parametrizable and can support Integer (I), Embedded (E), Integer Multiplication and Division (M) and Compressed (C) extensions. In fig. 2.4 the block diagram of the core is shown, the source code for it can be

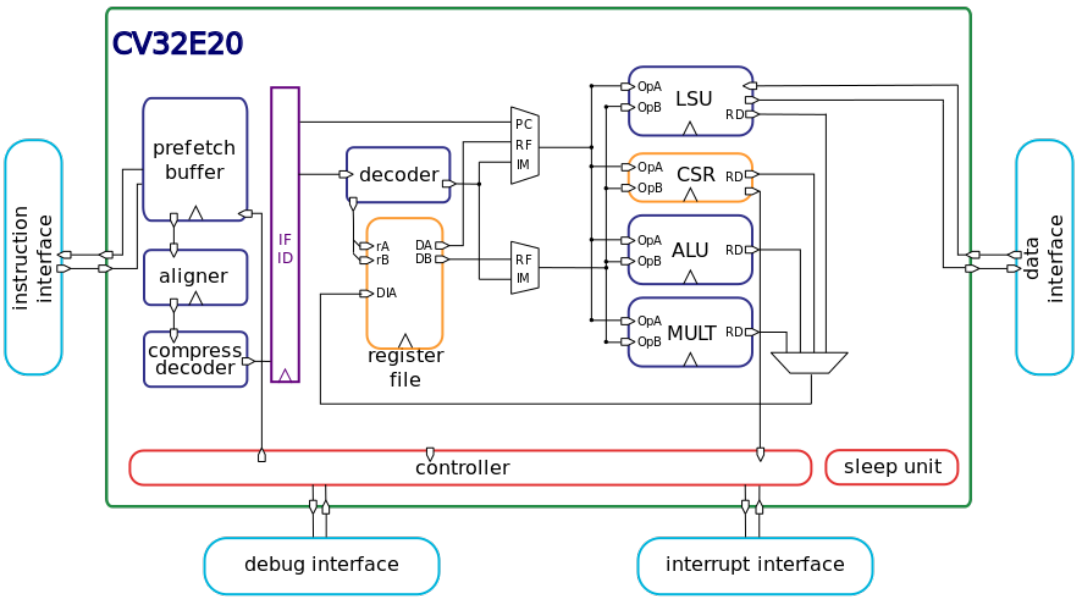

Figure 2.4: CVE32E20 Core, source: OpenHW [7]

found in the relative Github page.

2.4.2 Instruction extensions

The RISV isa includes the ability to add in a modular way different sets of instructions, depending on the needs of the core. The base fundamental instruction is the "I" (integer) set, then we have the most and officially defined extensions:

- M (integer Multiplication and Division)

- A (Atomic instructions)
- F (Single Precision Floating Point)
- D (Double Precision Floating Point)
- C (Compressed Instructions)
- G (General purpose)
- V (Vector Extension)
- B (Bit Manipulation)
- Zifecei (Instruction Fetch Fence)
- Zicsr (Control Status register Instructions)

This modularity is a great strength of RISCV but on the other hand poses a problem for the test-bench construction, since if we want an agnostic test bench able to test different cores it has to be able to support these different types of instructions without major changes in the flow and configuration.

2.5 Reinforcement learning

RL is a branch of machine learning where an agent learns by interacting with its environment [27]. Unlike supervised learning, where the model learns from a dataset of labeled examples, or unsupervised learning, which seeks to uncover patterns in unlabeled data, reinforcement learning involves an agent that must make decisions and learn from the consequences of its actions. The key characteristics that distinguish RL from other learning paradigms are:

- The system operates in a closed loop, where the agent's actions influence the environment and vice versa.
- The agent receives no direct instructions on which actions to take, learning instead from trial and error.
- The delayed consequences of actions, where the effects may unfold over extended periods, complicate the learning process.

The system operates in a closed loop, where the agent's actions influence the environment and vice versa. The agent receives no direct instructions on which actions to take, learning instead from trial and error. The delayed consequences of actions, where the effects may unfold over extended periods, further complicate the learning process.

At the core of RL, a learning agent interacts with the environment in pursuit of a goal. The agent observes the current state of the environment, selects actions that affect the environment, and receives feedback in the form of rewards. The challenge lies in determining which actions lead to the best long-term outcomes.

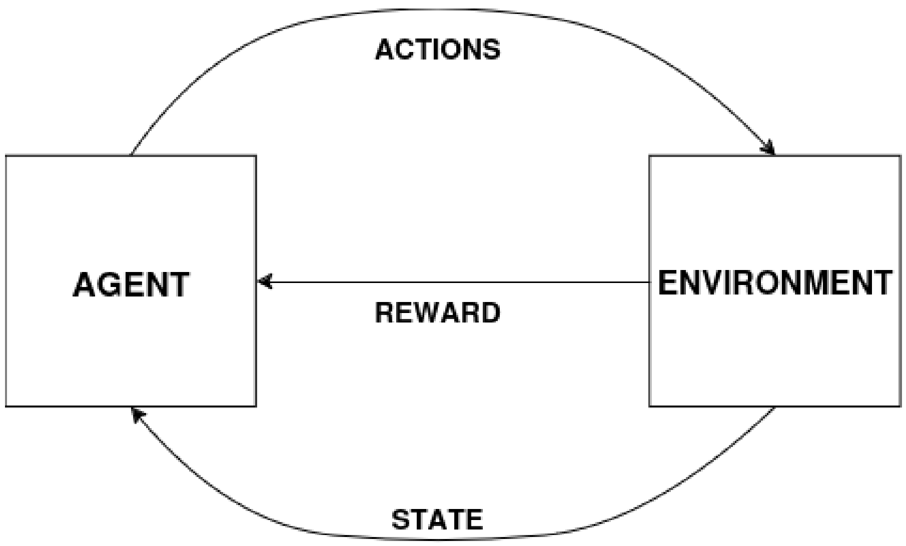

Figure 2.5: Agent-Env interaction.

This paradigm is particularly distinct because, unlike supervised learning where the agent is provided with correct outputs for given inputs, in RL, there is no such supervisor to guide every action. Similarly, it differs from unsupervised learning, which focuses on finding hidden structures in data. Instead, RL focuses on maximizing a reward signal, a numerical value that indicates success.

One of the central challenges in RL is balancing exploration and exploitation. The agent must exploit actions that have proven effective in the past, but it must also explore new actions that could potentially yield better rewards. Various strategies, such as the ϵ-greedy policy, Upper Confidence Bound (UCB), and gradient-based methods using the softmax function, address this balance by encouraging exploration in different ways. In reinforcement learning, the unpredictability of actions means that the agent must continuously monitor the environment and adjust its strategy accordingly. The main elements of RL beyond the agent and environment can be summarized as follows:

- Policy: the strategy that the agent uses to determine the next action based on the current state, ranging from a simple look up table to a complex algorithm

- Reward signal: a scalar feedback signal that the agent tries to maximize received from the environment

- Value function: a function that estimates the expected cumulative reward of a state, what is "good" in the long run so we seek states that have high value over a high immediate reward
- Model: a model of the environment that allows the agent to predict the next state and reward, some RL algorithms don't need a model of the environment and just follow the trial and error approach

More in detail at each time step (t = 0,1,... or some arbitrary successive stages) agent and environment interact and the agent receives some representation of the state of the environment and based on that chooses the next action from the set of actions possible from that state. In the next time step the agent receives a numerical reward as a consequence. In each time step the agent has a mapping from states to the probabilities of selecting one of the possible actions. The general rule is that anything outside the agent is considered part of the environment, so the agent is the only part that can be modified to improve the performance of the system.

The reward signal is the way to communicate to the agent which is the final objective so one must take care of not rewarding intermediate situations that may lead the agent to focus on the latter rather than the final goal. The rewards are computed in the environment since the boundary between the agent and environment can be put in such a way that the rewards are always external in some way. This avoids the situation where the agent simply decrees that the reward has been received and the problem is solved. While this is true the agent is also free to have some kind of internal reward signal defined by itself.

The agent goal is to maximize the cumulative reward it receives in the long term, in general we want to maximize the expected return where the return G_t is a specific function of the sequence of rewards received. The simplest case is when the return is just the sum of the rewards. This makes sense when there is an actual final step, so each episode ends in its terminal state after which there is a reset. But if the interaction between the agent and environment goes on indefinitely we need to introduce the concept of discounting: the agent tries to choose action so to maximize the sum of the rewards it will receive in the future.

Keeping track of all the state information from the past is not feasible in most cases, for this reason we look for signals that are able to inform the agent about the current state in a comprehensive way so as to take the correct action. This type of property is the Markov property. A state is Markov if based on the information of only the current state the agent can take the same decision as having the full history of the past states. Even if this is not always possible it is appropriate to think of the state as an approximation of a Markov state. A Markov Decision Process (MDP) is RL task that satisfies the Markov property [28]. Solving a RL task translates to finding a policy that is able to achieve a high reward in the long run. For finite MDPs we can define an optimal policy as the one that achieves the highest expected return for all states.

2.5.1 Deep Reinforcement Learning

Finding the action-reward function is the main goal of reinforcement learning, but in many cases the state space is too large to be able to store all the values in a table. In this case we can use function approximation to estimate the value of the state. Neural networks are a good choice for this task since they are able to approximate any function. [29] . We now have a brief look at neural networks in the particular case of Multi-Layer Perceptron (MLP), a type of NN consisting of fully connected neurons that utilize a nonlinear activation function. One of its main characteristics is being able to distinguish data that is not linearly separable when using at least three layers. The choice of the activation function mentioned before is fundamental in the behavior of the network and can improve the gradient vanishing problems common to deep neural networks. As concluded in [30] usually there is no single answer to which is best but the process requires lots of trial and error and observation of the training. Two of the most common thanks to their low computational cost are the Relu and Leaky Relu These are defined in the following way:

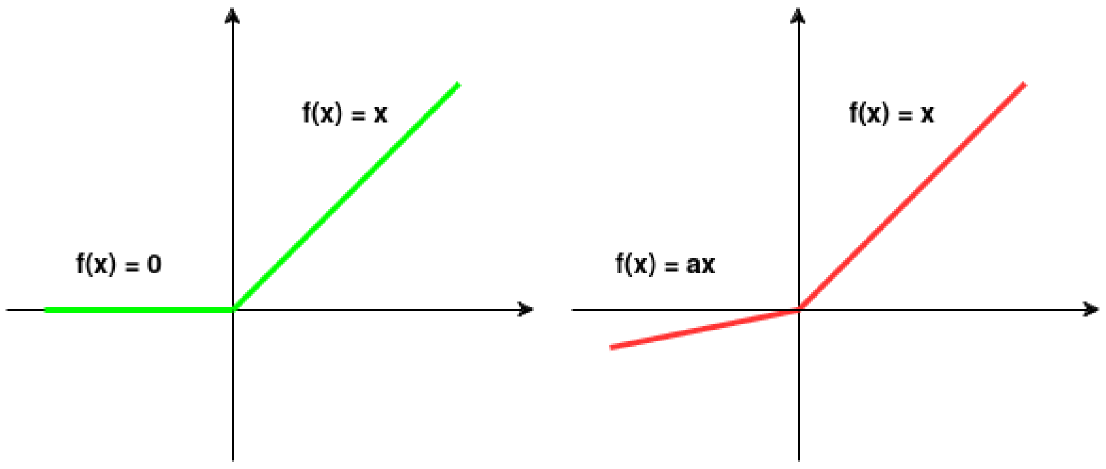

Figure 2.6: ReLU (left) and leaky rely (right) activation functions.

Another important problem in MLPs is the initialization of weights, which can significantly affect the learning process. Poor weight initialization can lead to issues like vanishing or exploding gradients, slowing down convergence or causing instability during training. One common approach is Xavier [31] initialization, where weights are scaled according to the size of the input layer to keep the variance of activations similar across layers. Another is Kaiming initialization [32], which is particularly useful when using ReLU activation and scaling weights to prevent gradient issues in deep networks. Proper weight initialization helps ensure that gradients flow efficiently during back-propagation, improving the stability and speed of learning in deep MLPs. For our context, reinforcement learning, using activation functions like ReLU and Leaky ReLU can help stabilize the learning process, especially when training deep networks for function approximation. As neural networks are used to approximate the state-action value function, these activation functions allow for more efficient back-propagation and avoid the stagnation of learning that can happen with the gradient vanishing issue.

But how are these Multi-Layer Perceptrons used in the RL agent? In reinforcement learning, MLPs are commonly used within **Actor-Critic** and Deep Q-learning frameworks. In the latter architectures, two distinct networks operate together: the **actor** network, which determines the policy by mapping states to actions, and the **critic** network, which evaluates the action taken by estimating the value of the state. As for the latter (deep Q-learning), the agent seeks to learn the Q-function Q(s,a), which represents the expected future reward of taking action a_a in state s_s, and then acting optimally from that point onward The Q-function approximates the optimal action-value function using a deep neural network. As seen in *Playing Atari with Deep Reinforcement Learning* [33] a common use is the use of MLPs to approximate Q-values in Q-learning algorithms. The Bellman equation is typically used to derive the optimal Q-values, and gradient-based methods such as Adam optimization [34] are applied to minimize the loss between predicted and target values. One problem that emerges with deep reinforcement learning is the sampling inefficiency of it [35], a possible solution is the one we are going to see next.

Replay buffer

The main idea behind replay buffers is to train the agent with the transitions sampled randomly from a buffer, where a transition is defined as the quadruple (s,a,r,s'), so the current state, action, reward and next state. At each step the current transaction is inserted in the buffer to be later sampled. so that the same experience can be used multiple times during the training, requiring fewer samples. Behind the replay buffer we have a quite big field of research since its size and way of storing the relevant experiences are crucial for the correct training of the network. Large replay buffers are usually involved in RL problems but this can hinder the performance, a number of solutions for this problem have been proposed like the Combined Experience Replay (CER) [36] where when sampling a batch of transitions we add the latest experience too to the batch. One other notable improvement is the **Prioritized Replay Buffer (PRB)** [37], which extends the replay buffer by prioritizing experiences based on their importance, typically determined by the Temporal Difference (TD) error. This is done by assigning each experience e in the replay buffer a priority p. If this is based on the TD error, we have $p = |\delta| + \epsilon$, where δ is the TD error and ϵ is a small positive constant to ensure no experience has zero priority. The probability of sampling an experience is then given by $p^\alpha / \sum p^\alpha$, where α is a hyper-parameter that controls the degree of prioritization. The idea is that experiences with high priority are sampled more frequently, allowing the agent to learn more from them. The TD error is calculated as $\delta = r + \gamma V(s') - V(s)$, where r is the reward, γ is the discount factor, $V(s)$ is the value of state s, and $V(s')$ is the value of the next state s'.

2.5.2 Stable baseline 3 and Proximal Policy Optimization

Stable baseline 3

SB3 is a popular open-source library for reinforcement learning in Python. It provides a set of reliable implementations of state-of-the-art reinforcement learning algorithms, built on top of the PyTorch deep learning framework. SB3 is designed to be user-friendly and modular, making it accessible for both beginners and experienced researchers in the field of reinforcement learning. It offers a clean and consistent API across different algorithms, allowing users to easily experiment with and compare various RL methods [38].

Proximal Policy Optimization

Proximal Policy Optimization (PPO) involves two neural networks: the Policy ($\pi_\theta(a|s)$), which outputs the probability distribution of the actions given a state, and the Value network ($V_\phi(s)$), which estimates the expected return of a state [39].

The objective in PPO is to maximize the expected reward while ensuring that the new policy doesn't deviate too much from the old one. This is achieved via a surrogate objective function that includes either clipping or reduction of the term. For the clipped one, letting θ old be the parameters of the policy before the update and θ be the parameters after the update, we have that the ratio $r(\theta) = \pi\theta(a|s)/\pi\theta_{old}(a|s)$ is used to compare the new and old policy. The clipped surrogate objective is then: $(clip(r(\theta), 1-\epsilon, 1+\epsilon)A_t)])$, where A_t is the advantage function at time t, and ϵ is a hyper-parameter that controls the clipping range.

The training process proceeds in the following loop:

1. Data collection: Run the current policy on the environment and collect a set of trajectories (each trajectory is a sequence of states, actions, rewards, and next states).

2. Advantage Estimation: Compute the advantage estimates A_t for each time step.

3. Policy update: Use the collected data to update the policy network. The PPO algorithm updates the policy by maximizing the clipped surrogate objective function using stochastic gradient ascent.

4. Value Function update: Simultaneously, update the value network by minimizing the loss between the predicted value and the observed return. The value loss function is the mean squared error between the predicted value and the target value.

In the implementation chapter the PPO model is going to be used in combination with the *MultiInputPolicy*, which is designed to handle multiple input spaces. In our case, this is needed because of the mix of one-hot encoded and float values that form the observation space.

Chapter 3

Implementation

3.1 UVM Test-bench

The initial implementation of the core test-bench was written in traditional SystemVerilog. However, when it became necessary to integrate the complete flow with repeated simulations to facilitate the training of the RL component, it became apparent that using QuestaSim for simulation was not feasible due to its long initialization times and difficulties in interacting with it from external scripts. Consequently, the decision was made to transition to Verilator, which offered faster initialization but lacked UVM support. As a result, Python, and in particular the CocoTb library, was adopted as the preferred language for the test-bench.

Using only CocoTb would have meant discarding UVM, along with all the standardized methods widely used in the verification ecosystem. Fortunately, there are open-source projects available that implement UVM classes and interfaces as Python libraries. In this context, *pyuvm*, introduced in the background chapter, was chosen. It leverages *CocoTb* for simulator interaction and implements the most commonly used parts of UVM, while benefiting from Python's looser typing and non-parameterized classes.

Pyuvm introduces several quality-of-life improvements, such as the automatic registration of components with the factory and the use of Python's logging system instead of the UVM reporting system. The documentation explicitly states that less frequently used parts of UVM, such as the resource database and some of the more uncommon ports, were not implemented.

In the code provided in the appendix listing 6.3, the *Driver* class demonstrates the primary features of *PyUVM* and *CocoTb*. The interface singleton is retrieved and used to communicate with the DUT, and, as in SystemVerilog, clock edges can be awaited (in this case using a custom function in the virtual interface to wait for a specified number of clock cycles). *PyUVM* provides predefined phases that can be overridden. The driver responds to core requests via two functions: one for data and one for instructions. The data function is more complex, as it must both store and load data, while the instruction function merely provides requested instructions. Both functions adhere to the Open Bus Interface (OBI) standard, illustrated in fig. 3.1, and follow the request/response/valid sequence. Integrating these components is not straightforward. While *Fusesoc* (and indirectly Edalize) supports *CocoTb*, it does so only in the latest development version of *Edalize*. Additionally, a modification to the *sim.py* script in *Edalize* is required, as *CocoTb* libraries that should be loaded at runtime are instead checked at compile time by the linker. This issue is resolved by adding the *-Wl,–unresolved-symbols=ignore-all* flag to the *LDFLAGS* in the *sim.py* script. More details on the solution can be found in this Github issue. Once configured, the flow operates similarly to a classic SystemVerilog test-bench, and from the user's perspective, the use of Python and Verilator for the test-bench is transparent.

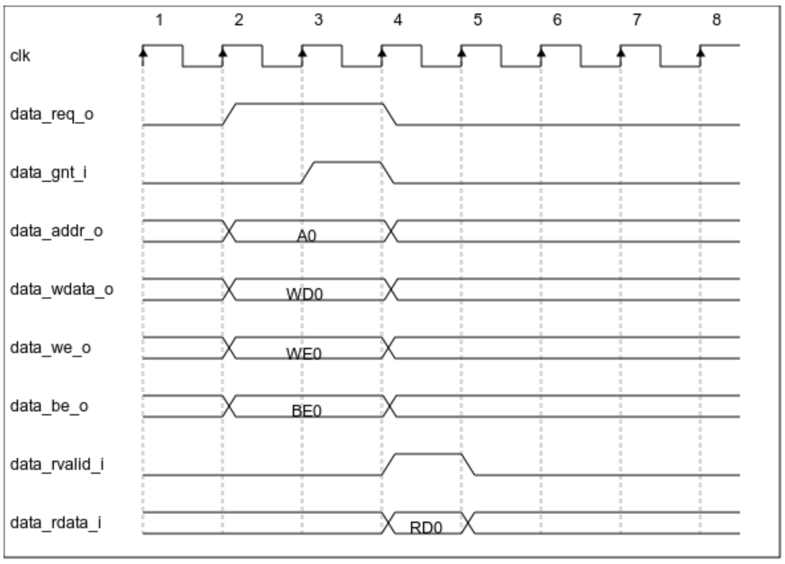

Figure 3.1: OBI standard for the LSU, source: OBI manual [8].

3.1.1 UVM Environment

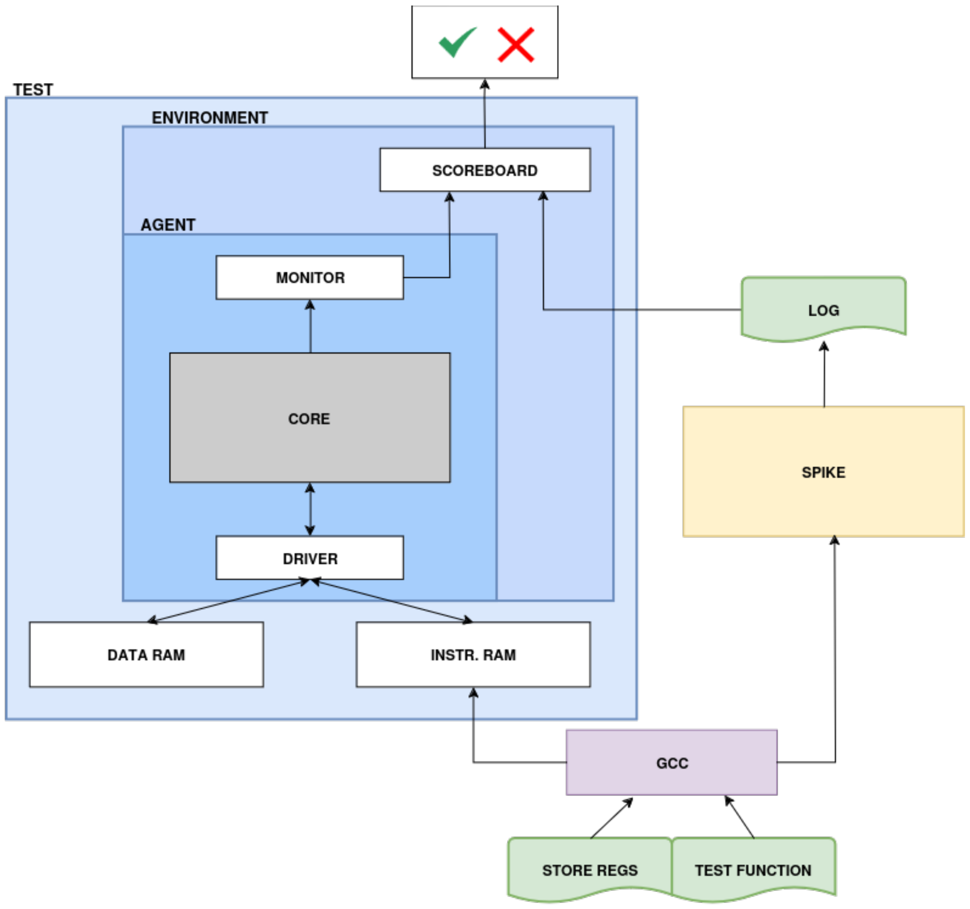

Figure 3.2: Core test-bench.

The structure of the UVM test-bench is nearly identical between Python and the SystemVerilog implementations, as depicted in fig. 3.2.

Unlike a traditional UVM test-bench with predefined sequences and a sequencer, an interactive approach is employed. The driver responds directly to core requests, sourcing from two distinct models: one for data memory and one for instruction memory. By utilizing UVM transaction-level modeling [40], transport ports facilitate communication between the driver and the memory components, enabling data or instruction requests and waiting for corresponding responses. An example of a data memory model is provided below:

Memory models

```python
async def transport(self, data_req):
    s = core_data_item("s")
    data_address = int(data_req.data_addr.value)
    data_bytenable = int(data_req.data_be.value)
    data_writedata = int(data_req.data_wdata.value)
    data_readdata = 0

    self.verbose_test = ConfigDB().get(None, "", "VERBOSE_TEST")

    if data_req.data_we.value == 1:
        for i in range(0, int(self.DataWidth / 8)):
            if data_bytenable & (1 << i):
                wdata = (data_writedata >> (8 * i)) & 0xff
                self.memory[data_address + i] = wdata
            else:
                if data_address + i in self.memory:
                    wdata = self.memory[data_address + i]
                else:
                    wdata = 0
    else:
        # Read operation
        for i in range(0, int(self.DataWidth / 8)):
            if data_address + i in self.memory:
                data_readdata |= (self.memory[data_address + i] << (8 * i))
            else:
                data_readdata |= (0 << (8 * i))
    if self.verbose_test:
        self.logger.info(f"Data request: data we: {data_req.data_we.value}
         address:
         {hex(data_address)}, bytenable: {hex(data_bytenable)}, writedata:
          S{hex(data_writedata)}, readdata: {hex(data_readdata)}")
    s.data_rdata = data_readdata
    return s
```

Source Code 3.1: Source code of the memory model.

Initially, the instruction memory is loaded from a *.vmem* file generated by *objdump* from the ELF file. For data memory, byte-addressability is required, based on the *byte enable* signal from the core. While this feature is easier to implement in SystemVerilog due to built-in bit-oriented variables, it can be achieved in Python through bit-wise masks. As seen in the code, the class implements the *transport* method, which handles bi-directional data communication between two components.

Monitor

To maintain the black-box approach, the core is monitored exclusively through writes and reads to the data memory. The monitor observes only the interface signals related to the data memory and generates packets, which are then sent to the scoreboard via the analysis port. On the other hand, the Spike output is parsed to extract only the data memory transactions, which are written to a file. This file is subsequently read by the predictor components, which push the data packets to the scoreboard's First In First Out (FIFO) queue. Additionally, the monitor detects writes to a particular address, the one predefined for Spike to signal the end of the program. This is utilized to drop the objection in the run phase of the monitor, transitioning to the UVM check phase, where the scoreboard compares the packets from the model and the DUT.

```
..
if await self.vif.read_data_req() and int(await self.vif.read_data_we()) == 1:
    seq = core_item("seq")
    seq.data_addr, seq.data_we, seq.data_wdata, seq.data_be,
    seq.data_rvalid, seq.data_req = await self.vif.read_data_interface()
    seq.instr_addr, seq.instr_req, seq.instr_rdata =
    await self.vif.read_instr_interface()
    self.ap.write(seq)
    if await self.vif.read_data_addr() == 0x20008
     and await self.vif.read_data_wdata() == 1:
        self.logger.info(
        f"Simulation stopped as the data write to address 0x20008 with value 1"
        )
        self.drop_objection()
        break
...
```

Source Code 3.2: End of simulation check.

Items

Three distinct items are utilized: two handle data and instruction requests and responses, each containing all the necessary OBI signals. The third item, *fsm_item*, is used in the monitor to store the current and next states observed from the control unit Finite State Machine (FSM), which are then sent to the coverage class.

Interface

In the PyUVM implementation of the UVM test-bench, an interface is employed to define methods for reading signals from the DUT. These methods group together signals related to the data or instruction sections. Similarly, write functions are defined for use by the driver. A useful method defined in the interface is the one that waits for a predetermined number of clock cycles. Additionally, the reset function is implemented ere and invoked at the start of the simulation.

Predictor

The predictor ingests the output of the Spike simulation to create data items, which are then transmitted to the scoreboard through a dedicated port so to be compared to the items coming from the DUT.

Driver

The driver module was previously discussed as an example for *pyUVM*. The notable point is that it monitors the interface to determine when to send data or instructions, in accordance with the OBI protocol mentioned earlier.

Coverage Class

The coverage class, an *uvm_subscriber*, implements the write port required to receive instruction and FSM items from the monitor. The instructions are decoded by differentiating between compressed or non-compressed types. By analyzing OPCODE and FUNC fields, the original instruction can be reconstructed, along with the registers involved. Although this decoding could be performed on the instruction generator side, the current implementation is more general and can work with any code provided. The coverage is checked by comparing sets of observed instructions against a reference set that contains all instructions of the given type.

Scoreboard

The scoreboard is a conventional one, implementing two write FIFOs: one for the monitor and the other for the reference model (the predictor). Before comparing the address and content of the data packets, the scoreboard applies the appropriate mask to the data. This is necessary because the core only transmits the complete word when writing to memory, even for half-word or byte writes. The mask is applied based on the *byte enable* bits sent along with the data.

```
for i in range (0,4):# range is
    temp = 0
    if (int(dut_data.data_be.value)) & (1 << i):
        temp = (int(dut_data.data_wdata.value) >> (8 * i)) & 0xff
    else:
        temp = 0
        dut_wdata = (temp << (8 * i)) | dut_wdata
```

Source Code 3.3: Scoreboard masking.

Agent and Environment

The agent and environment classes are grouped together, as their primary function is to build the sub-components and connect them in the relevant phases.

Test

The test is the top-level entity of the test-bench, where the environment is instantiated, and the verbose mode variable is set. During the run phase, the clock is initiated using the Cocotb method call.

3.1.2 Software, Compilation, and Handling

As mentioned earlier, the instruction memory model in the test-bench needs to be loaded with the software to be run for the test. This software, whether written in assembly or compiled from C, must be formatted in a binary format compatible with the core's instruction memory. The following *GCC-copy* command is used in the Makefile, in combination with the *sed* command, to add the correct starting address for the code at the beginning of the file:

```
(OBJCOPY) --reverse-bytes=4 --verilog-data-width=4 -I binary -O verilog $^ $@
sed -i '1s/^.*$$/@00100000/' $@
```

Source Code 3.4: GCC commands for instruction .vmem file creation.

The bytes are reversed because the core expects the data in a Little-Endian format. The data width specifies the number of instructions per line in the output file, which is a *.vmem* file that will be read by the test-bench to load the memory model.

For the base test-bench, there is also an option to pre-load the data memory with content. The following Makefile recipe handles this process:

```
$(OBJCOPY) --dump-section .data=$(@:.data.vmem=.data.bin) $^
$(OBJCOPY) --dump-section .rodata=$(@:.data.vmem=.rodata.bin) $^
cat $(@:.data.vmem=.rodata.bin) >> $(@:.data.vmem=.data.bin)
$(OBJCOPY) --reverse-bytes=4 --verilog-data-width=4 -I binary -O verilog
$(@:.data.vmem=.data.bin) $@
rm -f $(@:.data.vmem=.data.bin) $(@:.data.vmem=.rodata.bin)
sed -i '1s/^.*$$/@00140000/' $@
```

Source Code 3.5: GCC commands for data .vmem file creation.

This recipe dumps both the .data and .rodata sections into separate binary files, concatenates them, and converts the file to a *.vmem* format, cleans up temporary files and inserts the starting address for the data memory. The starting addresses are defined in the linker file, which is used by GCC during compilation:

```
OUTPUT_ARCH(riscv)
MEMORY
{
    ram         : ORIGIN = 0x00100000, LENGTH = 0x30000 /* 192 kB */
    stack       : ORIGIN = 0x00130000, LENGTH = 0x8000  /* 32 kB */
    data        : ORIGIN = 0x00140000, LENGTH = 0x20000 /* 128 kB */
}
/* Stack information variables */
_min_stack      = 0x2000;   /* 8K - minimum stack space to reserve */
_stack_len     = LENGTH(stack);
_stack_start   = ORIGIN(stack) + LENGTH(stack);
_entry_point = _vectors_start;
ENTRY(_entry_point)
```

Source Code 3.6: Section of the linker script.

As shown above, the *_entry_point* is defined, which is where the processor is expected to fetch the first instruction. The *_vectors_start* is located in the vector section of the linker script, which points to a location in the *crt0.S* file containing the following code:

```
    .section .vectors, "ax"
    .option norvc
    .org 0x00
    j _start
    .org 0x100
    .rept 15
    j trap_handler
    .endr
```

Source Code 3.7: Vector section of the crt0.

This section disables compressed instructions and jumps to the start of the *crt0 routine*, which resets all registers to zero and sets the core in vector mode for trap handling. This is needed since the Spike simulation adds some initial instructions that modify the content of the registers. After that sets the core in vector mode instead of the default direct for trap handling and loads into the *mtvec* register the *trap_handler* address, where the trap handler is just a routine that calls the *dump_register* function and then terminates the simulation:

```
li t0, 0x00000000
csrw mstatus, t0
li t2, 0x100100
csrw mtvec, t2
```

Source Code 3.8: Setting the core to vector mode.

At the end of the test program's assembly file, a call to the register dumping function is included. This is a crucial aspect of the methodology used to compare the DUT with the reference model. By writing all register contents into memory, the data can be accessed and observed via the chip interface.

3.1.3 Spike ISS

The Spike ISS is employed as the reference model for the core. The simulator accepts the input *elf* file generated from the assembly code and runs using the debug and *log-commits* options to produce an output file that logs all committed instructions and memory transactions. The latter is especially important, as it serves as the basis for comparison with the results gathered in the scoreboard. The simulator is launched prior to the actual DUT simulation and does not operate in lock-step with the DUT, as such synchronization would require probing internal core signals, violating the principle of maintaining an RTL-independent test environment.

Memory regions and the initial Program Counter (PC) value can be passed to Spike through arguments, which is necessary to synchronize the behavior of Spike with that of the DUT. Additionally, the assembly code must zero out all core registers via software because Spike, which does not run the code bare-metal, introduces extra instructions prior to executing the user program. This can potentially lead to behavior discrepancies between Spike and the DUT.

Spike also recognizes the end of a user program through a specific write to a designated address. This write operation must be included in all test assembly programs and is used by the test-bench to determine when the simulation can be stopped.

As Spike is open-source, modifications to its source code are possible. Such modifications were made to alter the simulator's behavior, specifically to enable the printing of all register file contents at each committed step. This feature is particularly useful for creating RL states later in the process.

```
3 0x0010034e (0xcb610113) x2  0x00138000
(spike) core   0: 0x00100352 (0x022000ef) jal     pc + 0x22
core   0:
zero: 0x00000000 ra: 0x00100356 sp: 0x00138000 gp: 0x00000000
tp: 0x00000000 t0: 0x00000000 t1: 0x00000000 t2: 0x00100100
s0: 0x00000000 s1: 0x00000000 a0: 0x00000000 a1: 0x00000000
a2: 0x00000000 a3: 0x00000000 a4: 0x00000000 a5: 0x00000000
a6: 0x00000000 a7: 0x00000000 s2: 0x00000000 s3: 0x00000000
s4: 0x00000000 s5: 0x00000000 s6: 0x00000000 s7: 0x00000000
s8: 0x00000000 s9: 0x00000000 s10: 0x00000000 s11: 0x00000000
t3: 0x00000000 t4: 0x00000000 t5: 0x00000000 t6: 0x00000000
```

Source Code 3.9: Spike log example.

3.2 Coverage and metrics collection

A fundamental part of the flow is the creating of a coverage report so to have a metric of how effective the test program being run is. This is done in different ways, depending on the open or closed approach.

3.2.1 First, closed source, approach

The primary metric provided by QuestaSim to assess the effectiveness of testing is code coverage. While code coverage does not directly indicate the correctness of the design, it provides insights into how thoroughly the code is exercised during test execution. A high level of code coverage, often approaching 100%, is typically required, and this metric is generally achievable with a well-constructed test suite. The code coverage provided by QuestaSim is divided into several categories:

- **Statement coverage**: counts the execution of each statement on a line individually, even if there are multiple statements on a single line.
- **Branch coverage**: it measures the percentage of branches in the code that have been executed. This can be an "if/then/else" or a "case" statement.
- **Condition coverage**: can be considered an extension of branch coverage, analyzes the decision made in "if" and ternary statements.
- **Expression coverage**: similar to condition coverage, it analyzes the expression on the right-hand side of assignment statements.
- **Toggle coverage**: it measures the percentage of toggles in the code that have been executed. A toggle is a signal that changes its value. This can be very expensive in terms of simulation time.
- **FSM coverage**: it measures the percentage of states, transitions and paths in the FSMs present in the code that has been executed.
- **SystemVerilog class coverage**: it measures the percentage of classes, methods and statements in the SystemVerilog classes that have been executed.

The flow to collect the coverage is the following:

1. Compile the design with the coverage option enabled, actually the *+cover* argument is added to the optimization phase (vopt).
2. Enable the coverage collection during the simulation via the *coverage opttop* argument.
3. Save the collected data in a *.ucdb* file on simulation exit.
4. Run the simulation.

Questa can be configured to collect coverage for only a subset of the source code by specifying, during the compilation phase, the coverage argument for the modules of interest. A subset of modules can similarly be specified via the *+<selection>* modifier. Coverage data can be dynamically saved during different test runs, not only at the end of the simulation.

The Unified Coverage DataBase (UCDB) file serves as the single repository where all the coverage data is stored. Once saved, the coverage can be analyzed using the Graphic User Interface (GUI), either during an active simulation or through post-processing.

During the optimization process, non-essential parts of the design (e.g., code that is never called) can be removed, which can lead to faster simulation times but may cause an apparent decrease in coverage. The GUI provides the ability to observe which lines of code are excluded from coverage. By default, Questa applies a balanced level of optimization, though this can be customized via the *coveropt* option. Code that is optimized out does not count towards possible hits and thus does not impact the coverage calculation.

3.2.2 Final open source based coverage collection

As highlighted in the test-bench section, the transition to open-source tools like Verilator introduces certain limitations, particularly regarding coverage capabilities. Although Verilator supports toggle and line coverage, it lacks a critical feature: FSM coverage, which is essential for thorough verification.

To address this limitation, it is necessary to break the black-box approach previously maintained and observe some internal signals within the core. For example, it becomes essential to monitor the registers within the Control Unit, which represent the internal state. This information is integrated into the corresponding test-bench module and reported at the end of the simulation.

Verilator provides line coverage data in the form of a text-based output, specifying each code line and the number of times it has been activated. This format is easily parsed, and by using a bash script, a report is generated. The report is organized by the core's main units and serves as input for the ML component of the flow. For example, this process produces the following results for the CVE2 core:

- Arithmetic logic unit (ALU) line coverage
- compressed decoder line coverage
- controller line coverage
- core line coverage
- counter line coverage
- control status registers line coverage
- CSR handling line coverage
- decoder line coverage

- fetch FIFO line coverage
- decode stage line coverage
- fetch stage line coverage
- load store unit line coverage
- multiplier/divider line coverage
- prefetch buffer line coverage
- register file line coverage

Additionally to these coverages, the functional ones are included too, looking at the type of instructions that are sent to the core. This is done as mentioned in the test-bench section via the monitor and the corresponding coverage class, moreover, here the dependencies are checked:

- R type functional coverage
- I type functional coverage
- RM type functional coverage
- S type functional coverage
- B type functional coverage
- U type functional coverage
- J type functional coverage
- C type functional coverage
- Register dependencies functional coverage

All the coverages at the end of the simulation cycle are collected and organized in a single file so to be then easily processed by the ML part of the flow.

3.3 Reinforcement Learning agent and environment

In the following sections the Agent and Environment structures will be described as well as their interactions, which can be summarized as in fig. 3.3.

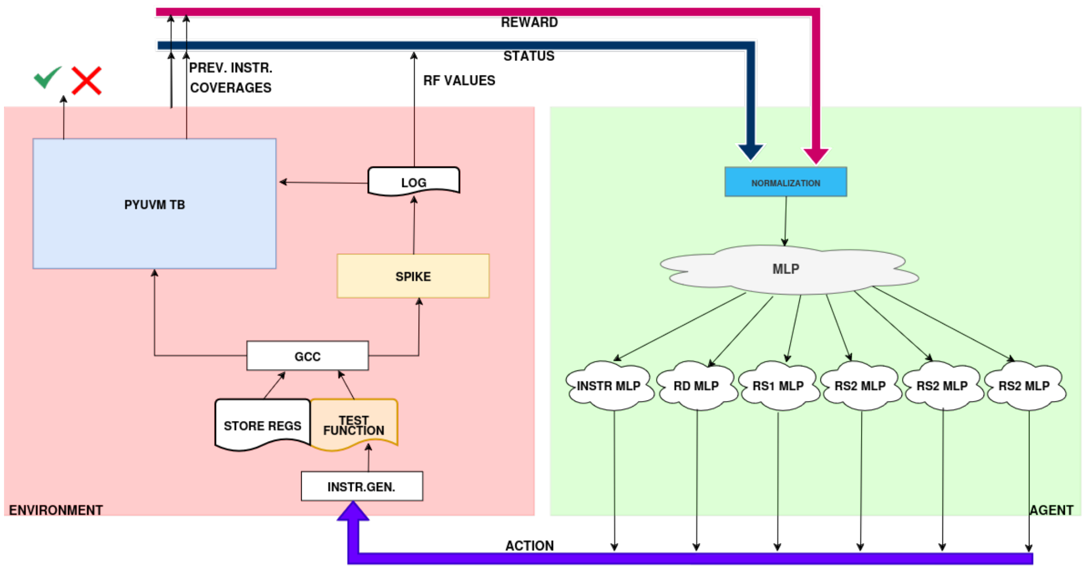

Figure 3.3: First version of the RL environment with a custom agent.

3.3.1 Environment

As discussed in the introduction to Reinforcement Learning, the two key components in this framework are the Agent and the Environment. In the initial implementation, this division was followed using the Python package *Gymnasium*, which provides standardized templates for classes and methods to ensure a structured approach.

For the Environment class, the core methods are *step()* and *reset()*. The *reset()* method initializes and cleans the environment at the start of each simulation, while the *step()* method defines the environment's dynamics, determining how it responds to actions taken by the agent. Additionally, the class includes attributes like *action_space* and *observation_space*, representing the range of possible actions and observations that the agent can execute and perceive.

In the initial tests, the observation space consisted of the following data:

- The difference between the current value of the RF and the values from the previous step.
- All the separate coverage values, both in absolute terms and their differences from the previous step.
- The differences between the last five instruction actions and the previous five.

All these values are either normalized between 0 and 1 or, in the case of the last instructions, one-hot encoded into Numpy vectors. As for the expected actions (action space they are displayed in table 3.1.

Action name	Description	Number of choices
instr_type	The instruction type (group).	14
dst_reg	The destination register.	32
src_reg1	The first source registers.	32
src_reg2	The second source registers	32
imm_val	The immediate value (group)	5
imm_use	Whether the instruction is intended as an immediate type or not.	2

Table 3.1: Action types.

Initially, the actions were one-hot encoded, as the Q-values from the MLPs were already represented in a standard encoded format for simplicity. In the latest version, the PPO model outputs actions directly decoded from this one-hot format. At the beginning of the process, the reward function was a weighted average of the coverage values, with extra rewards for detecting failures (such as mismatches between the gold model and the DUT), since bug discovery is a key objective of the test-bench. Over time, this reward function was enhanced to include rewards for specific events, resulting in the final one that is summarized in table 3.2.

Lastly, for the *state* the information found in table 3.3 is provided to the agent, in the third column the standard *Gymnasium* data structure used is specified. This forms the state vector that can be seen in fig. 3.4.

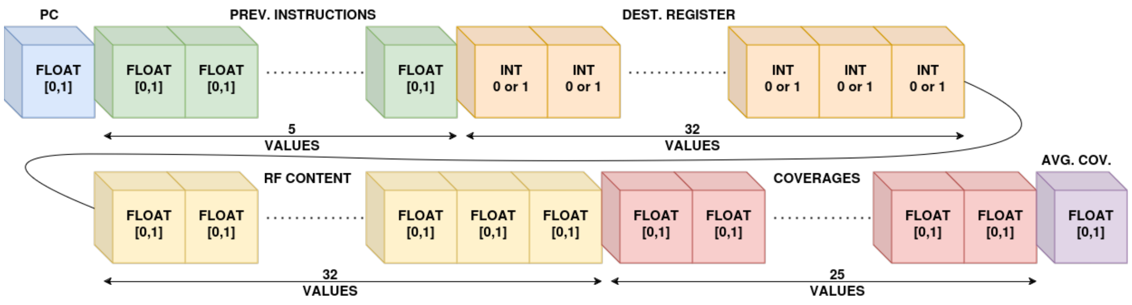

Figure 3.4: Composition of the state vector

In the *step()* method, instruction generation is the first task, accomplished by

Type of reward	Explanation
Additional reward in case of reaching predetermined coverage goals (0.70, 0.71... and so on.).	To provide additional incentives in case of higher coverages.
For register dependencies detected between the previous and current instruction issued.	Since this might not directly effect the overall coverage.
No identical instruction in the last 5.	To try and maximize the instruction coverage.
If the content of the registers used is zero, there is a negative reward.	Repeated operations using zero value are less interesting from the verification standpoint.
The average coverage is used as a base for all the previous to be added/subtracted.	The basic metric that we are most interested in.

Table 3.2: Reward calculation

invoking the relevant function and passing the action values. The returned instruction(s) are then inserted into the *test_function.S* file via another method. During this process, the current instruction count is updated based on the number of instructions returned by the generator. The previous instruction is then updated, and the list containing the last five instruction types is shifted accordingly.

Once the setup is complete, the actual simulation is initiated using a script that handles software and RTL compilation, the Spike run, and the RTL simulation while parsing the coverage values. Exception handling is critical at this stage, particularly for scenarios where Spike or the simulation does not conclude properly. The Python sub-process package is used to set timeout values for both processes. If an exception occurs—whether due to timeout, GCC compilation errors, or a mismatch between the DUT and Spike—the output is logged in an error file. Depending on the situation, training is either halted, or the current step is completed, and the done flag is set.

After the simulation, the Spike log is parsed to extract register file values, which are normalized between 0 and 1 from their original 32-bit format. Coverage values are also extracted from the coverage report file, and previous values are updated after rescaling. At this point, termination conditions for the episode are evaluated. These include reaching an average coverage value greater than 95%, hitting the

State component	Description	Shape (Data type)
program_counter	The PC of the core after the last generated instruction.	Box(low=0, high=1, shape=(1,), dtype=float)
last_instructions	Set of the last 5 instruction groups	Box(low=0, high=1, shape=(5,), dtype=float)
destination_register	The destination register of the last instruction, to help with the dependencies.	Box(low=0, high=1, shape=(32,), dtype=int)
registers	All the registers content .	Box(low=0, high=1, shape=(32,), dtype=float)
coverages	all the different covareges	Box(low=0, high=1, shape=(25,), dtype=float)
avg_coverage	The average coverage, the same one we use for the reward.	Box(low=0, high=1, shape=(1,), dtype=float)

Table 3.3: State vector components

maximum instruction count for the episode, or encountering a mismatch between the DUT and the gold model. Another condition for termination is the execution of an illegal instruction or memory access. In such cases, the exception handler is triggered, and no further instructions are executed beyond the illegal one.

Finally, the differences between the previous and current state variable values are computed and concatenated with the coverage values to form the state vector. This state vector, along with the reward, the done flag, and additional information (such as the normalized average coverage for logging), is returned to the agent.

3.3.2 Agent

Custom implementation

The focus is now shifted to the Agent, where the actual learning part of the process occurs. This class receives the observation space and action sizes from the environment and constructs networks with the appropriate input and output sizes.

The initial implementation of the agent was a custom design utilizing a deep Q-learning approach. A deep neural network, built using the *pytorch* library in Python, was employed to predict the Q-values of the potential actions. This MLP was structured into a primary larger network and six smaller networks dedicated to generating the Q-values for the six distinct actions. This was done so to not make the size of the Q-values space explode.

The various neural networks were implemented using the sequential mode offered by *pytorch*, facilitating the definition of the interconnected layers and their activation functions. This structure enabled the writing of the forward method, which connects the main encoder network to the specialized child networks.

For reference in constructing the agent, an existing implementation was utilized as an example. This code was a modified version of the implementation described in the paper Playing Atari with Deep Reinforcement Learning [33], where RL is applied to learn how to successfully play seven Atari games using screen captures of the gameplay as input. While the environment in this context is significantly different, the general structure of the agent provided a solid foundation. Key components borrowed from the paper included the optimal action-value function adhering to the Bellman equation and the loss function, defined as the mean squared error between the predicted Q-values and the target Q-values.

The agent was trained using the Adam optimizer, and loss was back-propagated through the network. Managing the combined loss from the six sub-networks during back-propagation to the main network posed challenges. Ultimately, a custom LeakyReLU class was defined to scale down the gradients from the sub-networks by a factor of six, preventing them from overshadowing the main network.

The following snippet illustrates the custom LeakyReLU implementation:

```python
class LeakyReLUCustom(nn.LeakyReLU):
    def __init__(self, negative_slope=0.01, inplace=False):
        super(LeakyReLUCustom, self).__init__(negative_slope=negative_slope,
         inplace=inplace)

    def backward(self, grad_output):
        return grad_output / 6
```

Source Code 3.10: LeakyRelu implementation

And how the loss is calculated:

```
def compute_loss(model, replay_buffer, batch_size, gamma):
    state, action, reward, next_state, done = replay_buffer.sample(batch_size)
    state = torch.FloatTensor(np.float32(state)).to(device)
    next_state = torch.FloatTensor(np.float32(next_state)).to(device)
    action = torch.LongTensor(action).to(device)
    reward = torch.FloatTensor(reward).to(device)
    done = torch.FloatTensor(done).to(device)

    main_output_old, *q_values_old = model(state)
    main_output_new, *q_values_new = model(next_state)

    # losses is a list of losses for each sub-net
    losses = []
    for i, (q_values_old_sub, q_values_new_sub)
     in enumerate(zip(q_values_old, q_values_new)):
        sub_action = action[:, i]
        q_value_old = q_values_old_sub.gather(1,
         sub_action.unsqueeze(1)).squeeze(1)
        q_value_new = q_values_new_sub.max(1)[0]
        expected_q_value = reward + gamma * q_value_new * (1 - done)
        loss = (q_value_old - expected_q_value.detach()).pow(2).mean()
        losses.append(loss)
    return losses
```

Source Code 3.11: Loss calculation.

As for the neural networks themselves, the main one was composed of three fully connected layers expanding the state space, while the sub-networks were composed of five fully connected layers, gradually reducing the size down to the different action sizes. Again, from the paper, the idea of the replay buffer was taken. This is a memory that stores the last N transitions to have a more diverse training set. The replay buffer was implemented as a *deque*, and sampling was done randomly. The replay buffer aimed to break the correlation between the samples to have more stable training. Moreover, the buffer increases sample efficiency since the same sample can be used multiple times. In our case, a memory size of 1024 and a batch size of 32 were used. The epsilon-greedy policy was used to balance the exploration and exploitation of the agent, with epsilon decreasing over time. During the various iterations, different activation functions, as well as weight initializations, were tried, resulting at the end with the use of LeakyRelu since the networks were growing in depth and so the problem of the vanishing gradients appeared, and the Kaiming initialization was used for the weights. In the table 6.1 the final parameters tried are summarized.

The last addition to the custom agent was the Prioritized Replay buffer, already coverd in the background chapter, Unfortunately, this did not improve the training and was discarded, highlighting the probability in the construction of the agent estimator itself.

3.3.3 Stable Baseline 3 implementation

The initial custom implementation of the agent did not yield meaningful improvements in coverage when compared to random generation. This lack of success can be attributed to insufficient experience in tuning a complex agent capable of managing a vast observation space and, particularly, a large action space. Consequently, the decision was made to transition to a well-established library like Stable Baselines 3.

While the overall flow, as illustrated in Figure fig. 3.5, remains the same, the implementation of the agent undergoes significant changes. The library offers a range of agent implementations, with the primary requirement being adherence to the *Gymnasium* standard interface for the environment. This condition imposes that the agent can call the following functions within the environment:

Method	Description
init(self, arg1, arg2, ...)	Defines the action_space and observation space as *gym.spaces* objects.
step(self, action)	Used for advancing the episode step by step, has to return: observation, reward, terminated, truncated, info.
reset(self, seed=None, options=None)	Used to reset the environment at the beginning of the episode, has to return observation, info.
render(self)	used only in the case of visual representation of the environment.
close(self)	closes the environment at the end of the episode.

Table 3.4: Custom agent parameters.

With an environment that adheres to the required methods, the custom environment can be instantiated and wrapped in a *Monitor* object. This enables the logging and recording of rewards and other metrics using Weights and Biases (WB) for tracking purposes. The code for environment instantiation is as follows:

```
def make_env(log_dir):
    env = core_sim_env(instr_number = config["instr_number"],
     step_number = config["step_number"])
    env = Monitor(env, log_dir)
    return env
```

Source Code 3.12: Environment instantiation in the agent.

The PPO model, in combination with the MultiInputPolicy, is used to handle the multiple input spaces. This is necessary due to the observation space comprising both one-hot encoded and float values.

Stable Baselines provides custom callbacks, which are passed to the training or evaluation function. These callbacks serve multiple purposes, such as saving the model and logging the training data via the Weights and Biases API, which provides tracking and visualization for machine learning projects. An example of custom callbacks for saving the model is included in the appendix listing 6.1. The info data structure contains the additional information returned by the environment at each step.

3.4 Instruction Generation and Test Function

A fundamental part of the environment flow is translating agent actions into actual instructions to append to the test function program. To accomplish this, existing instruction generators were initially explored, specifically Force RISC-V [17]. This generator features a C++ back-end for instruction generation and a Python front-end, where the user configures the generation process. The Python file requires the user to define the sequence(s) class with the appropriate generation method. Instructions are predefined based on the architecture and can be retrieved through predefined sets or custom dictionaries. These dictionaries contain the instructions and their associated weights, which determine the probability of selecting the next instruction during generation. The dictionaries themselves can be placed into a higher-level map, introducing an additional layer of weighting to select different instruction subgroups. The main sequence can consist of multiple sub-sequences, which can be either user-defined or sourced from a pre-defined library. This design allows for flexible switching between different generator behaviors during the program generation process.

Ultimately, a custom Python script was developed to handle this process because of the complete control required over the generated instructions. The script receives six actions from the agent: *instruction type, rd, rs1, rs2, immediate, choose_immediate, and number_instructions.* The core of the script involves a switch-case structure that organizes the instructions into groups corresponding to the first action provided. These groups are not a one-to-one match with standard RISC-V instruction sets but represent simplified and sometimes overlapping categories. As of the latest version, the groups include: *LOGIC, SHIFT_LEFT, SHIFT_RIGHT, ADD_SUB, MUL, DIV, REM, LOAD, STORE, JUMP, BRANCH_EQ_NE, BRANCH_LT, BRANCH_GE, LOAD_IMM, and ILLEGAL_INSTR.* These groups can be adjusted based on how granular the control needs to be. Bias can be introduced towards specific instructions by including them in multiple groups (e.g., LOAD_IMM is also part of the ALU group as ADDI).

Instruction generation begins with a random selection from the chosen group, followed by filling in the fields with the provided action values. Different instruction types come with specific constraints, such as register limits or immediate ranges, which are managed through specialized classes for each RISC-V instruction group. Each class ensures that the generated instruction adheres to the constraints based on the input values. For example, I-type instructions are restricted to a specific immediate range. The agent does not provide the exact value for the immediate, but instead indicates a general group (e.g., large, small, or zero), which must be accounted for during generation.

For Jump, Branch, and Load/Store instructions, additional considerations are made to ensure valid memory access or address jumping. For these instructions, auxiliary instructions are generated to preload valid values into the relevant registers. Constraints are then applied to the requested action to ensure that the preloaded registers and immediate values are used.

For the Jump instruction, a dynamically generated label is needed, along with a small random number of *NOP* instructions between the jump and the label. The same process applies to Branch instructions, which behave similarly to Jump but do not save the return address in the link register. For Jump and Link instructions, a single dummy target can be fixed in the test code, as the return address is saved in the link register, allowing the flow to resume from there.

The source code in listing 6.2 demonstrates this behavior for Jump, Branch, and memory instructions.

If the agent requests an illegal instruction, the generator returns a *.word 0x00000000* instruction. This instruction will be accepted by the compiler but will cause an illegal instruction fault during execution. To generate a misaligned instruction, a more complex sequence is required. This involves creating a code area misaligned to 2 bytes (or in other words, aligned to 1 byte), followed by a jump to this misaligned region. The instruction set returned includes the necessary compiler directives for alignment and the jump to the misaligned area.

3.5 Complete simulation environment

To ensure the correct operation of the simulation and the RL flow, a combination of Python scripts, Makefiles, and core files are employed. The RL environment directly interacts with a Python script that resembles a Makefile in structure, orchestrating the RTL compilation, Software (SW) compilation, and the Spike and RTL simulations. Since the primary difference between runs is the software loaded into memory, the RL environment utilizes functions from the instruction generator (as described in the previous section) to generate valid instructions from the agent. These instructions are then inserted into the test program using another script, placing them before the register dumping function and after the last inserted instruction. In the case of environment resets, instructions are inserted in a way that cleans the environment.

The coverage results provided by Verilator are parsed within the aforementioned Python Makefile using a series of scripts that also combine the functional coverage results from the UVM test bench. Additionally, another script is used to parse the Spike log, extracting only the register contents representing the core state. Simultaneously, memory writes are isolated for the UVM test bench to verify the correctness of the simulation.

The Python Makefile, through the subprocess library, handles the output parsing from various commands, allowing the detection of errors or timeouts that may occur during simulation. Custom exceptions, defined in a separate file, manage error reporting. If an exception occurs—whether due to a timeout, an error in the SW or RTL compilation, or a core misbehavior compared to the reference model during the simulation—the exception, the last instruction appended, and the *stdout/stderr* are logged into a file and reported in the terminal.

In fig. 3.5 the overall division of the scripts is depicted, following the numbering on the scheme we have:

1. Agent and Environment interaction, already discussed via the step function.
2. The call to the simulation handler inside the step function of the Environment.
3. The exception system, all the exceptions raised are passed up the level until they reach this point.
4. Again the environment calling the instruction generator and inserter to append the new instruction to the test code.
5. The auto-generated code that already includes the dump register function and the CRT0.
6. The code can be compiled creating the *vmem* and *elf* files for Spike and the actual simulation.

7. Spike simulation occurs guided by the simulation handler.
8. RTL simulation is handled via Fusesoc.
9. Coverage reports from the test-bench and core status from Spike are parsed and fed back to the environment.

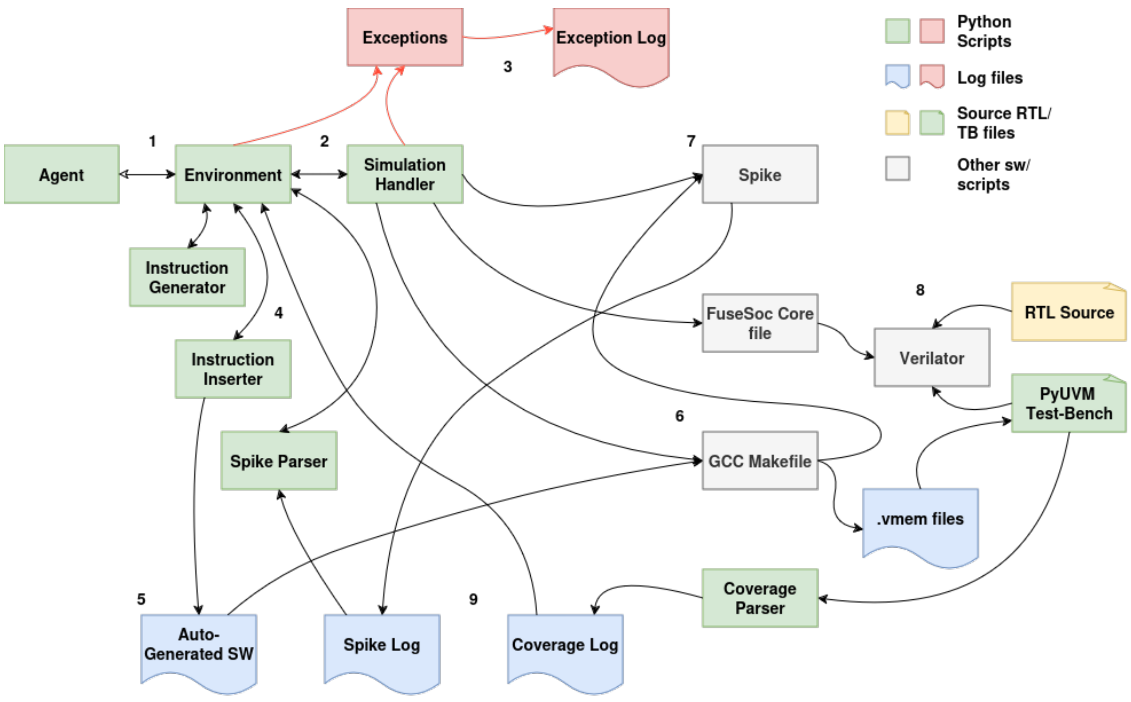

Figure 3.5: Complete program flow.

Chapter 4

Results

4.1 RISCV test bench

The first result to analyze is the contribution of the test bench itself. As discussed in the introduction, building a verification framework is inherently complex, and doing so in Python with the limited online resources available for *pyUVM* added an extra layer of difficulty. However, the results have been successful. The current test bench allows for writing code in either C or assembly, compiling it, and generating the necessary .vmem files to load into the test bench memory emulators. From there, the test bench runs the tests, comparing the results with those from the Spike simulation. This flow fully supports C programs, including those that allocate constants and other variables in data memory.

```
157.00ns INFO      [uvm_test_top.core_env.core_fsm_coverage]: Coverage percentage: 17.14%
157.00ns INFO      [uvm_test_top.core_env.core_func_coverage]: R type coverage: 0%
157.00ns INFO      [uvm_test_top.core_env.core_func_coverage]: I type coverage: 22%
157.00ns INFO      [uvm_test_top.core_env.core_func_coverage]: RM type coverage: 0%
157.00ns INFO      [uvm_test_top.core_env.core_func_coverage]: S type coverage: 0%
157.00ns INFO      [uvm_test_top.core_env.core_func_coverage]: B type coverage: 0%
157.00ns INFO      [uvm_test_top.core_env.core_func_coverage]: U type coverage: 100%
157.00ns INFO      [uvm_test_top.core_env.core_func_coverage]: J type coverage: 50%
157.00ns INFO      [uvm_test_top.core_env.core_func_coverage]: C type coverage: 29%
157.00ns INFO      [uvm_test_top.core_env.core_func_coverage]: Dependency coverage: 42%
157.00ns INFO     cocotb.regression                  core_test passed
157.00ns INFO     cocotb.regression
*************************************************************************************
** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
*************************************************************************************
** core_test.core_test            PASS         157.00           0.10       1566.08  **
*************************************************************************************
** TESTS=1 PASS=1 FAIL=0 SKIP=0                157.00           0.31        510.54  **
*************************************************************************************
```

Source Code 4.1: PyUVM output.

In listing 4.1 the *pyUVM* test bench generates an output similar to a standard

System Verilog implementation, providing comparable information even though it is using the python logging system and not the UVM reporting one. The coverage class reports results through the reporting system, and the outcomes are verified against the predictor (Spike), with the passing of the test asserted. The *pyUVM* library allowed for a more efficient writing process of the test bench core compared to the verbose nature of System Verilog, while still maintaining the UVM infrastructure. Although simplified in some aspects, as discussed in the background and implementation chapters, it remains effective. On the simulation side, compared to the traditional closed-source approach (e.g., QuestaSim), a notable speed-up was observed in simulation iterations after the initial compilation of the Verilog files, thanks to Verilator. This enhancement was crucial for integrating the RL environment, which demands thousands of simulation restarts during training. Additionally, the reporting and scripting infrastructure built around the test bench efficiently logs exceptions and tracks the status of the RL agent model during repeated simulation cycles.

4.2 Reinforcement learning training

4.2.1 First custom agent approach

As discussed in the RL implementation chapter, the initial custom agent approach failed to demonstrate any consistent improvement in the reward, and more critically, the neural network weights either exploded or remained constant. This behavior indicates a fundamental flaw in the implementation or the approach to the problem itself. In fig. 4.1, the training of the main neural network diverges rather than converging to smaller values, which is typical in a correct learning process for a NN. The intensity of the blue color is directly proportional to the number of weight values found in that range, and as it can be seen as the number of episodes increases, the number values diverge. A similar pattern is observed in the child MLPs and their biases, highlighting a significant issue with the training process. This divergence likely stems from a combination of improper weight initialization, an excessively high learning rate, insufficient regularization, or even an incorrect problem formulation.

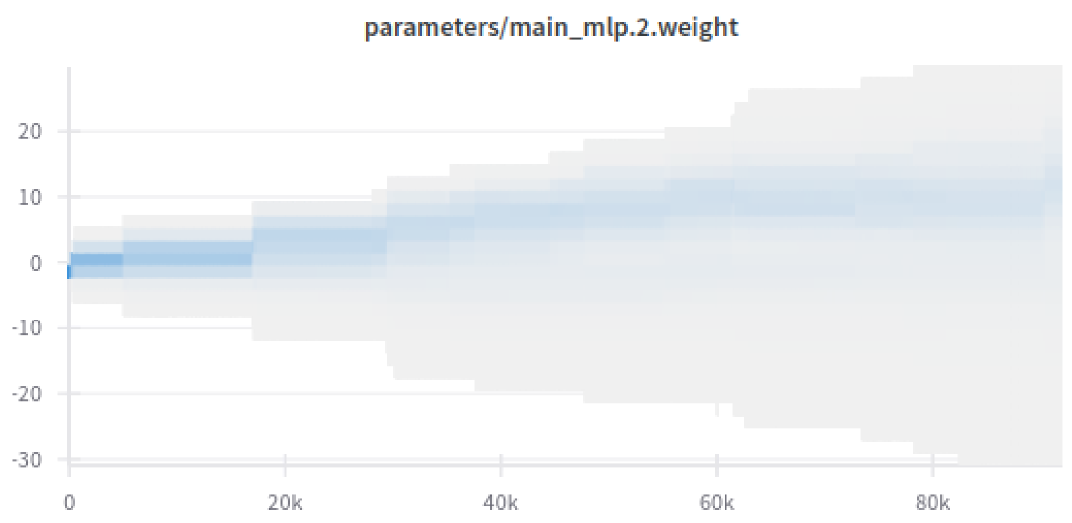

Figure 4.1: Main MLP weights.

The result of this faulty learning can be seen in the reward (fig. 4.2) that comes from the environment, not increasing but moving randomly during the episodes.

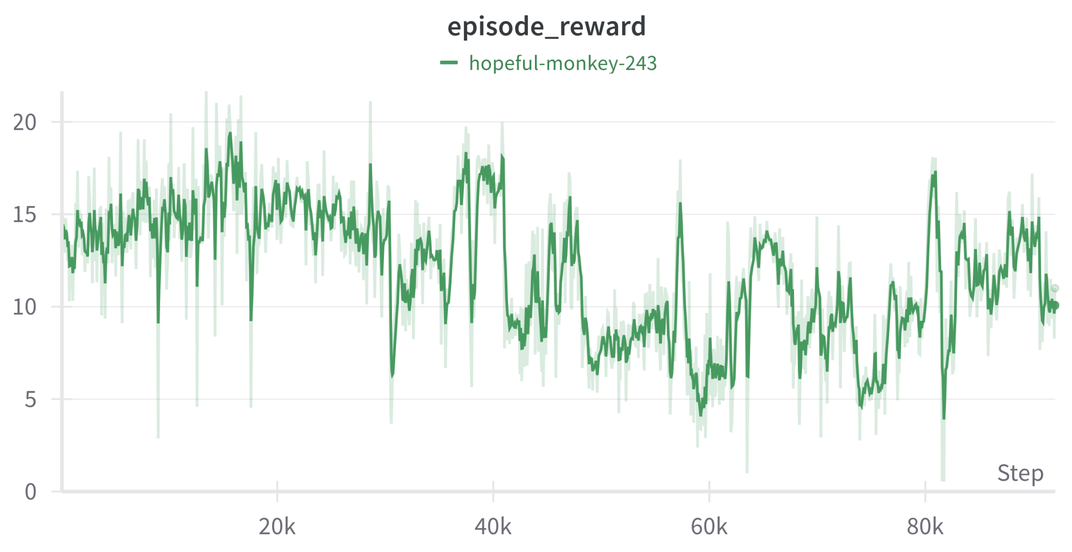

Figure 4.2: Reward of the custom agent.

4.2.2 Stable baseline 3 approach

We now move to the SB3 implementation of the agent,

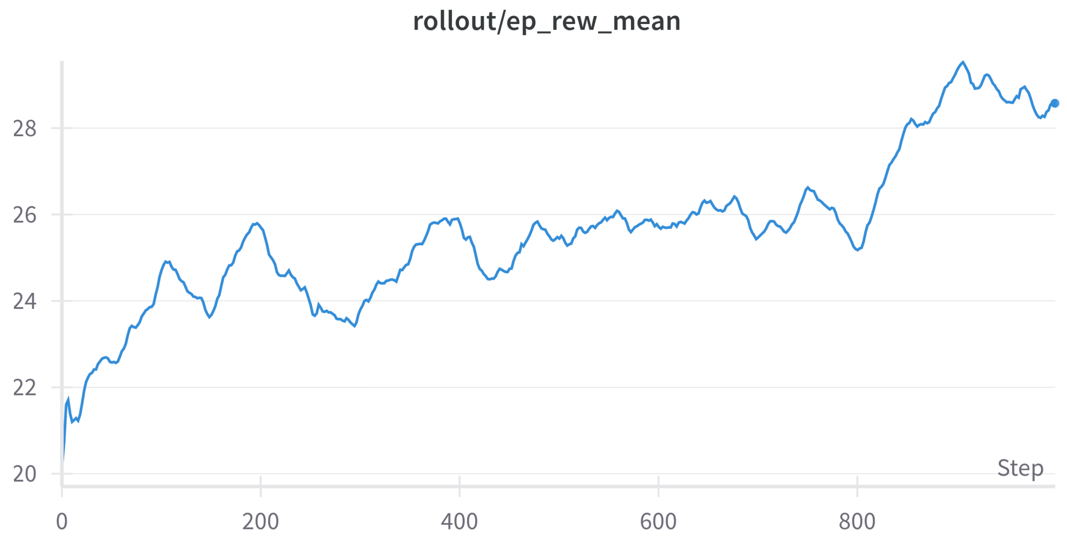

Figure 4.3: Training reward.

in fig. 4.3 the zoom-in of the first 1000 episodes (WB mistakenly defines them as "steps") of training of the RL neural network for a 50 instruction code generation is shown. As it can be seen there is a growth of the average reward over the course of the episodes. Weights&dBiases provides other metrics such as the explained variance, measured as:

$$\frac{1 - [Var]([empirical\_return] - [predicted\_value])}{[Var]([empirical\_return])}$$

that shows how well the prediction of the model explains the variance of the actual data fig. 4.4, a value of 1 is the best possible result.

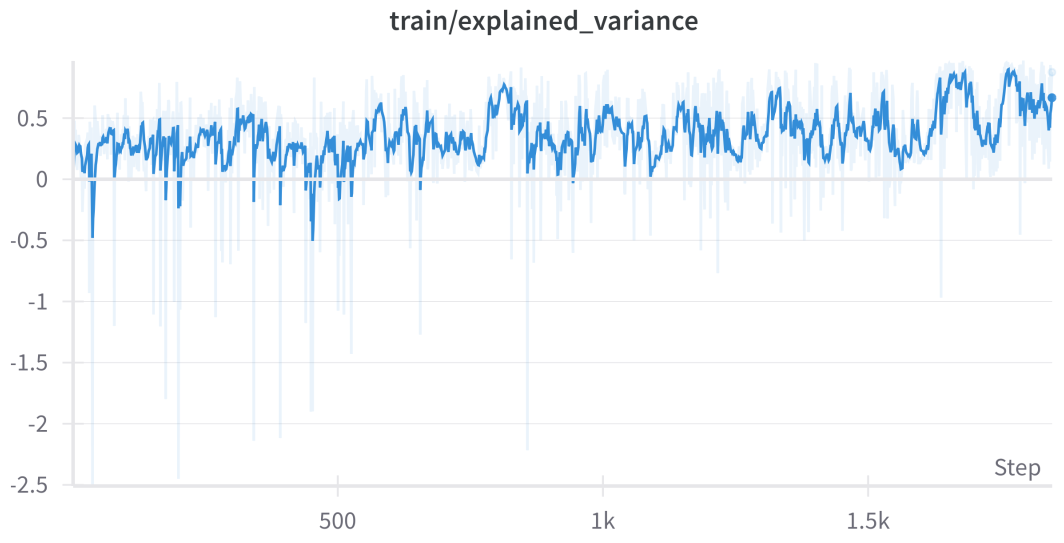

Figure 4.4: Explained variance over episodes.

All the above point to a correct training of the NN on which the agent is based, which in turn results in an increasing coverage as it is going to be discussed in the next section. A more common metric that can be observed is the entropy loss of the PPO in the network, fig. 4.5, this is going towards lower (absolute) values as the training progresses as opposed to the loss function in the custom agent where it would fluctuate randomly.

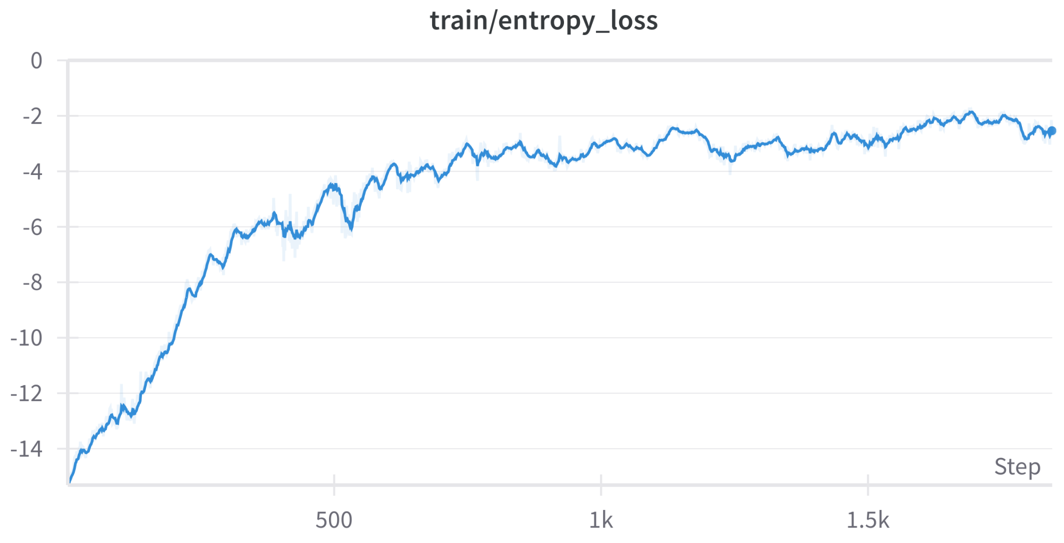

Figure 4.5: Entropy loss over episodes.

4.3 Coverage results

Now that it has been confirmed that the underlying NN is learning during training, it is necessary to compare the results to determine if there was an actual improvement in the coverage of the core. The obvious baseline for comparison is the instruction generator that was developed, used in a completely random manner, i.e., simply requesting random instruction/sets of instructions up to the same amount requested by the RL algorithm.

In the following table, both the average value of 20 different randomly generated codes and the maximum value are shown to display both a best-case scenario, where the random generation chooses a diverse set of instructions and registers, and a mean case, which can be expected when running the random instruction generator a single time. For the mean, the standard deviation is also reported, providing an idea of how much the coverage changes from one random run to the next.

In this first table, only the average coverage value between all the different functional and code coverages is considered. A deeper analysis of the individual coverages will follow.

# of Agent actions.	Avg. coverage of random approach	Std. Dev of random approach	Best cov. of random approach	RL result after training
50	65.62%	1.46	68.53%	69.49%
100	70.00%	0.70	71.64%	74.24%
10000 (1 run)	—	—	74.42%	—

Table 4.1: Average coverage results.

The number of actions corresponds to the number of steps in one episode, and one action may generate more than one instruction due to jump, branch, and memory actions, which consist of multiple instructions, as explained in the instruction generator section. For context, requesting 10,000 actions (resulting in 18,000 instructions) from the instruction generator produced coverage almost identical to that of the RL method with just 100 actions. The number reported is referred to a single run since the large number of requested instructions smoothed the results.

Another interesting metric is the progression of coverage as new instructions are appended to the test program in both the random and RL cases. This demonstrates how the random approach, after reaching around 70% coverage, almost plateaus taking a large number of instruction (as mentioned in the case of the 10000 requests) to grow near the RL result. Since randomly selecting instructions that specifically target uncovered cases becomes more difficult, further improvements in coverage rely solely on chance, as expected from a random generation method.

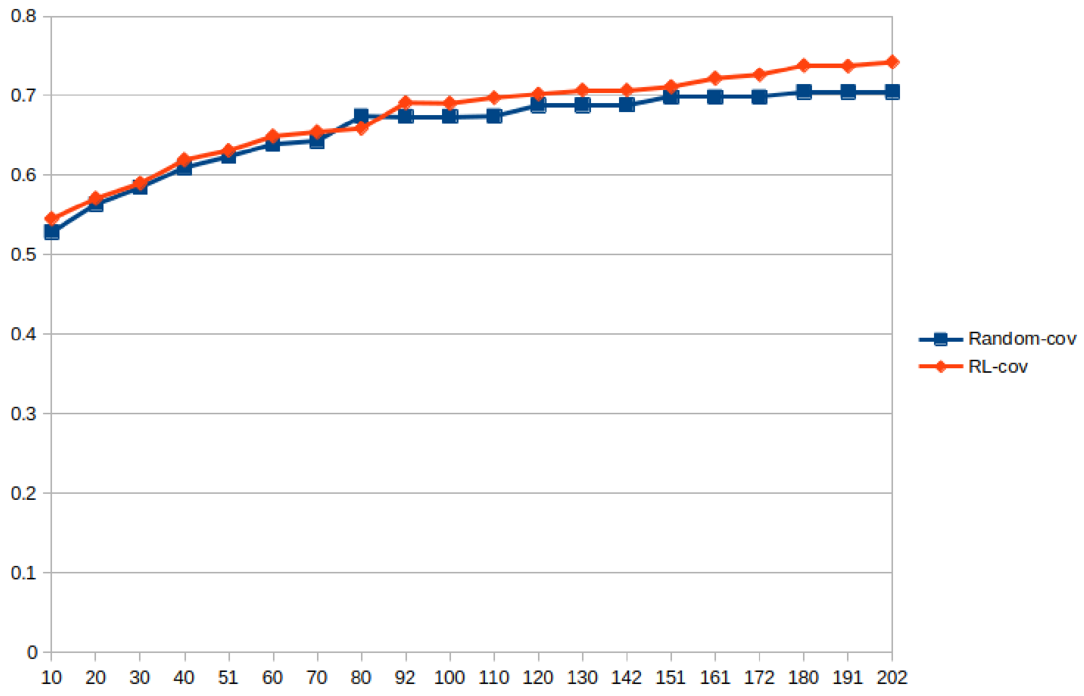

Figure 4.6: Coverage progression in the two cases.

The analysis of individual coverages presented in table 4.2 (specifically for the best RL result) reveals that even with the best-performing generated software some coverage values remain relatively low. Starting with the coverage metrics that exhibit higher values, functional coverage is predominantly influenced by the agent's ability to request instructions, as its coverage values directly reflect the instruction type. In contrast, register dependencies present a more significant challenge, as the algorithm must recognize that subsequent instructions using the same destination and source registers contribute to an increase in this coverage metric. The situation is further complicated for compressed-type instructions, as the ability to compress an instruction by the compiler depends on its specific construction (e.g., registers, immediate values), leading to lower coverage.

In terms of FSM coverage, the issue arises because the status of the core remains unchanged after a reset and the initiation of instruction fetching; since all instructions are valid, there is no way for the core to transition out of the "DECODE" state. When examining line coverage, the low values primarily result from code not executed by the current instruction set configuration (*IMC*) and previously mentioned cases not covered, such as those in state machine code. A significant portion of the total lines of code is dedicated to managing exceptions and corner cases that typically do not appear in normal, valid code. Functional coverage can

Coverage Name	**Cov Value**
ALU line coverage	40.00%
Compressed decoder line coverage	45.87%
Controller line coverage	36.80%
Core line coverage	83.10%
Counter line coverage	77.78%
Control status registers line coverage	35.34%
CSR handling line coverage	80.00%
Decoder line coverage	51.81%
Fetch FIFO line coverage	100.00%
ID stage line coverage	85.29%
IF stage line coverage	80.49%
Load store unit line coverage	42.00%
Multiplier/divider line coverage	97.30%
Prefetch buffer line coverage	95.24%
Register file line coverage	100.00%
FSM coverage of the controller unit	17.14%
R type functional coverage	90.00%
I type functional coverage	88.89%
RM type functional coverage	100.00%
S type functional coverage	100.00%
B type functional coverage	100.00%
U type functional coverage	100.00%
J type functional coverage	100.00%
C type functional coverage	37.50%
Register dependencies functional cov.	71.43%

Table 4.2: Coverage table with respective values

achieve high coverage in simpler modules or those performing specific operations, such as the multiplier module.

Attention now turns to the actual assembly code generated by the algorithm.

```
..
LA x8, _stack_start
LB x12, 4(x8)
BGE x8, x2, target_branch67
target_branch67:
XORI x19, x8, -676
BGEU x8, x3, target_branch70
ADDI x0, x0, 0
```

```
ADDI x0, x0, 0
target_branch70:
LA x8, _stack_start
SB x8, 4(x8)
LA x8, _stack_start
LW x23, 4(x8)
LA x8, _stack_start
SH x21, 4(x8)
```

```
REM x12, x8, x8
LA x8, _stack_start
SH x8, 4(x8)
LA x8, _stack_start
SW x3, 4(x8)
REMU x3, x26, x2
BLT x8, x5, target_branch86
ADDI x0, x0, 0
ADDI x0, x0, 0
ADDI x0, x0, 0
ADDI x0, x0, 0
target_branch86:
BLT x8, x2, target_branch92
ADDI x0, x0, 0
target_branch92:
LA x8, _stack_start
SH x3, 4(x8)
SRAI x3, x8, 24
LA x8, _stack_start
SH x3, 4(x8)
BEQ x8, x2, target_branch100
...
```

Source Code 4.2: ASM automatically generated code.

As it is apparent from the previous Assembly (ASM) code generated the RL agent has effectively learned to have a very differentiated set of instructions and, thanks to the custom rewards, a number of register dependencies are present too. An example of how the instruction generator handles branches and jumps can be seen, the *addi x0,x0,0* instructions inserted are the *nop* instructions used to randomly offset the destination from the jump branch or jump instruction. Moreover, the dynamically generated labels for the branches are present.

Chapter 5

Conclusions and future work

The Python test bench for the RISC-V architecture has been a successful implementation. Python demonstrated itself as a viable alternative to System Verilog while still keeping the UVM standard, offering a more streamlined coding experience and a range of features that prove its flexibility as a programming language. Furthermore, the structure of the Test-Bench (TB) enabled the testing of the core using only the standard interface provided, without necessitating direct engagement with the source RTL (aside from the FSM coverage). During the development of the test bench, a collateral benefit was the exploration of automation in the compilation and simulation processes, which pointed out interesting edge cases when using *Fusesoc* in conjunction with *PyUVM* and *Cocotb*.

In the context of Reinforcement Learning, a modest improvement in code quality was observed such as the 4.2% higher coverage for the same number of requests and the fact that with those 100 requests the RL generated code is able to match the very long (10000 requests) case. The most significant achievement was the establishment of an integrated environment and simulation workflow capable of executing requested actions, generating valid instructions, simulating their execution, and verifying the correctness of results. This process also included the collection of necessary information to inform the RL environment and provide relevant core-state data to the agent. This accomplishment required extensive scripting and experimentation and was aided by the addition of a reporting system for any exception in the flow.

However, certain challenges were encountered during the project. One major issue was the size of the state space; even when restricting the analysis to a subset of instructions, the potential combinations of instructions, registers, and immediate values expanded dramatically, complicating the training process. Additionally, to achieve comprehensive coverage of the RTL code, it would be essential to incorporate exceptions and interrupts into the generated instructions. This raises the question of how to manage episodes that would inevitably be truncated as a result of such inclusions. A potential solution could involve generating distinct sections of code separately, ensuring that those likely to trigger exceptions are placed at the end of the test program. This would facilitate the normal execution of other instructions before encountering the exceptions.

The reward function plays a crucial role in any RL problem and requires careful calibration to highlight specific aspects of the generated code. It was observed that additions to the reward function, which incorporated additional bonuses or penalties not directly related to the core coverage, do not necessarily correlate with improvements in coverage metrics so need to be chosen with care depending on what aspects of the core one is interested in testing more. The training was accelerated utilizing a Graphics Processing Unit (GPU); however, further acceleration could be achieved through the parallelization of multiple training sessions initialized with different random seeds.

The work conducted establishes a solid foundation for future research, having already addressed several implementation strategies that demonstrate a more efficient approach. The fully open-source nature of the framework offers a performant and versatile basis for exploring various ML methodologies. This flexibility will facilitate continued advancements in the field, allowing for the investigation of additional techniques and innovations.

Chapter 6

Appendix: Code

```
class save_coverages_values(BaseCallback):
def __init__(self, verbose=1):
super(save_coverages_values, self).__init__(verbose)
# clean the file
with open("coverages.txt", "w") as f:
f.write("")
def _on_step(self) -> bool:
if self.locals['dones'][0]:
info = self.locals['infos'][0]
with open("coverages.txt", "a") as f:
f.write(f"Coverages at step {self.num_timesteps}:")
f.write(f"{info['coverages']}\n")
print(f"Savings coverages at step {self.num_timesteps}")

return True
```

Source Code 6.1: Example of custom callback for stable baseline

```
...
elif instruction_type == STORE:
return_instrs = []
rs1 = generate_rd(rs1_act)
if rs1 == 0:
rs1 = 1
la_instr = f"LA x{rs1}, _stack_start"
return_instrs.append(la_instr)
s_instr = random.choice(list(STORE_bin))
s_instr = S_type_instruction(instr=s_instr, rd_act = rd_act,
 rs1_act=rs1, imm_act=imm_act, force_imm=4)
return_instrs.append(s_instr)
return return_instrs
elif instruction_type == JUMP:
instr = random.choice(list(J_type_bin))
if instr == 'JAL' and rd_act == 0:
return_instrs = []
jal_instr = J_type_instruction(instr, rd_act, rs1_act, imm_act)
return_instrs.append(jal_instr)
for i in range(0, random.randint(0, 8)):
nop_instr = I_type_instruction('ADDI', 0, 0, 0, None)
return_instrs.append(nop_instr)
return_instrs.append("target:")
return return_instrs
elif instr == 'JAL' and rd_act != 0:
jal_instr = J_type_instruction(instr, 1, rs1_act, 0)
return jal_instr
elif instr == 'JALR':
return_instrs = []
la = "LA x2, target_jal"
return_instrs.append(la)
jalr_instr = J_type_instruction(instr, 1, 2, 0)
return_instrs.append(jalr_instr)
return return_instrs
elif instruction_type == BRANCH_EQ_NE:
return_instrs = []
instr = random.choice(list(BRANCH_EQ_NE_bin))
b_instr = B_type_instruction(instr, rs1_act, rs2_act, instr_count)
return_instrs.append(b_instr)
for i in range(0, random.randint(0, 4)):
nop_instr = I_type_instruction('ADDI', 0, 0, 0, None)
return_instrs.append(nop_instr)
return_instrs.append(f"target_branch{instr_count}:")
return return_instrs
```

Source Code 6.2: Section of the instruction generator source code

```
class core_driver(uvm_driver):
def __init__(self, name, parent):
```

```
super().__init__(name, parent)

def build_phase(self):
self.transport_data = uvm_blocking_transport_port("transport_data", self)
self.transport_instr = uvm_blocking_transport_port("transport_instr", self)
self.vif = core_interface()

async def start_of_simulation_phase(self):
self.vif.dut.rst_ni.value = 0
await cocotb.start_soon(self.vif.reset())
await self.vif.reset()

async def run_phase(self):
await self.start_of_simulation_phase()
cocotb.start_soon(self.drive_data())
cocotb.start_soon(self.drive_instr())
await self.drive_data()
await self.drive_instr()

async def drive_data(self):
await self.vif.wait_clock(1)
while True:
await self.vif.write_data_interface(data_rvalid=0, data_rdata=None,
data_gnt=1)
if await self.vif.read_data_req():
data_req = core_data_item("data_req")
data_req.data_addr, data_req.data_we, data_req.data_wdata,
data_req.data_be, _, _ = await self.vif.read_data_interface()
data_resp = await self.transport_data.transport(data_req)
if data_req.data_we.value == 0:
await self.vif.write_data_interface(data_rvalid=1,
data_rdata=data_resp.data_rdata, data_gnt=1)
else:
await self.vif.write_data_interface(data_rvalid=1,
data_rdata=0, data_gnt=1)
else:
await self.vif.write_data_interface(data_rvalid=0,
data_rdata=None, data_gnt=None)

await self.vif.wait_clock(1)

async def drive_instr(self):
```

```
while True:
await self.vif.write_instr_interface(fetch_enable=1,
instr_gnt=1, instr_rvalid=0, instr_rdata=None)
if await self.vif.read_instr_req():
await self.vif.write_instr_interface(fetch_enable=1,
instr_gnt=1, instr_rvalid=1, instr_rdata=0)
instr_req = core_instr_item("instr_req")
instr_resp = core_instr_item("instr_resp")
instr_req.instr_addr, _, _ = await self.vif.read_instr_interface()
instr_resp = await self.transport_instr.transport(instr_req)
await self.vif.write_instr_interface(fetch_enable=1,
instr_gnt=1, instr_rvalid=1, instr_rdata=instr_resp.instr_rdata)
else:
await self.vif.write_instr_interface(fetch_enable=1,
instr_gnt=1, instr_rvalid=0, instr_rdata=None)
await self.vif.wait_clock(1)
```

Source Code 6.3: Source for the driver class.

Parameter	**Description**
EPISODES = 1500	Number of episodes that compose the training process.
BATCH_SIZE = 32	Number of samples taken from the replay buffer at each iteration.
GAMMA = 0.99	The factor that determines how much future rewards are taken into account.
EPS_START = 0.99	Starting value of epsilon, the factor that determines the exploration-exploitation trade-off.
EPS_END = 0.01	Minimum value that epsilon can reach. The lowest exploration rate achievable.
EPS_DECAY = 15000	Decay factor for epsilon, a higher value means slower decay.
INITIAL_MEMORY = 40	Minimum sample size stored in the replay buffer before starting to sample and back-propagate.
MEMORY_SIZE = 1024	Size of the replay buffer.
MLP parameters	
MAIN_HIDDEN_SIZE_0 = 256	First hidden layer of the main MLP, expanding the state space.
MAIN_HIDDEN_SIZE_1 = 512	Second hidden layer of the main MLP, continuing the expansion of the state.
MAIN_OUTPUT_SIZE = 1024	Third and last layer of the main MLP.
SUB_NET_SIZE_0 = 512	Output size of the first layer of the multiple sub-networks, from here we start decreasing the size.
SUB_NET_SIZE_1 = 256	As in the main MLP we are expanding the size in these layers we are gradually reducing the size again to the action size.
SUB_NET_SIZE_2 = 128	As before, still reducing the size.
SUB_NET_SIZE_3 = 64	Fourth layer of the sub-networs, still reducing the size.
SUB_NET_SIZE_4 = 32	Last layer before the output one, size almost matching the Q-values then needed.
Optimizer parameters	
MAIN_LR = 0.03	Learning rate for the optimizer (Adam in this case). This controls how much the model's weights are updated during each training step. A high learning rate can lead to faster convergence but may also cause instability, while a lower learning rate ensures smoother updates.

Table 6.1: Gymnasium API

Bibliography

[1] Wikipedia, Transistor count.
URL https://en.wikipedia.org/wiki/Transistor_count

[2]

[3] T. Liu, R. Ho, U. Jonnalagadda, Open source risc-v processor verification platform (2019).
URL https://riscv.org/wp-content/uploads/2019/12/12.10-16.10b-Open-Source-Verification-Platform-for-RISC-V-Processors.pdf

[4] Siemens, Uvm cookbook (2018).
URL https://verificationacademy.com/cookbook/uvm-universal-verification-methodology/

[5] O. Group, Uvm hierarchy (2024).
URL http://vlsi4freshers.com

[6] Cocotb, Cocotb documentation (2024).
URL https://docs.cocotb.org/en/stable/

[7] openhwgroup, Cve2 (2024).
URL https://github.com/openhwgroup/cve2

[8] OpenHW, Obi lsu manual (2024).
URL https://cv32e40p.readthedocs.io/en/latest/load_store_unit.html#load-store-unit

[9] Siemens, The 2020 wilson research group functional verification study (2021).
URL https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verification-study/

[10] DesignReuse, Transistor count trends continue to track with moore's law (2020).
URL https://www.design-reuse.com/news/47652/transistor-count-trends.html

[11] D. Price, Pentium fdiv flaw-lessons learned, IEEE Micro 15 (2) (1995) 86–88.

[12] K. S. Pawar, U. Menon, J. c. k. h. Riedel, Time to market, Integrated Manufacturing Systems 5 (1994) 14–22.
URL `https://api.semanticscholar.org/CorpusID:110293977`

[13] Y. Caspi, Hardware functional verification – present and future (2013).
URL `https://research.ibm.com/haifa/conferences/hvc2013/present/YuvalCaspi_HVC-2013-tutorial.pdf`

[14] SemiconductorEngineering, Property specification language (2024).
URL `https://semiengineering.com/knowledge_centers/languages/property-specification-language/`

[15] chipverify, System verilog assertions (2024).
URL `https://www.chipverify.com/systemverilog/systemverilog-assertions`

[16] S. Qamar, W. H. Butt, M. W. Anwar, F. Azam, M. Q. Khan, A comprehensive investigation of universal verification methodology (uvm) standard for design verification, Proceedings of the 2020 9th International Conference on Software and Computer Applications (2020).
URL `https://api.semanticscholar.org/CorpusID:219132594`

[17] OpenHWGroup, Open source risc-v processor verification platform (2023).
URL `https://github.com/openhwgroup/force-riscv`

[18] Imperas, riscvisacov.
URL `https://github.com/riscv-verification/riscvISACOV`

[19] M. I. Jordan, T. Mitchell, Machine learning: Trends, perspectives, and prospects, Science 349 (2015) 255 – 260.

[20] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, D. Hassabis, Mastering the game of go without human knowledge, Nature 550 (7676) (2017) 354–359.
URL `https://doi.org/10.1038/nature24270`

[21] Synopsys, Enhancing chip verification with ai and machine learning (2022).
URL `https://www.synopsys.com/blogs/chip-design/enhance-chip-verification-with-ai-and-machine-learning.html`

[22] S. Konale, N. Rao, C-based predictor for scoreboard in universal verification methodology, in: 2014 International Conference on Advances in Engineering & Technology Research (ICAETR - 2014), 2014, pp. 1–5.

[23] Spike (2024).
URL `https://github.com/riscv-software-src/riscv-isa-sim`

[24] E. Ham, Y. Jeon, J. Lim, J.-H. Kim, Verilator-based fast verification methodology for ble mac hardware, in: 2023 International Conference on Electronics, Information, and Communication (ICEIC), 2023, pp. 1–3.

[25] O. Kindgren, Fusesoc.
URL `https://fusesoc.readthedocs.io/en/stable/user/overview.html`

[26] A. Waterman, Design of the risc-v instruction set architecture, 2016.
URL `https://api.semanticscholar.org/CorpusID:63861396`

[27] F. Wörgötter, B. Porr, Reinforcement learning, Scholarpedia 3 (2019) 1448.

[28] C. C. White, D. J. White, Markov decision processes, European Journal of Operational Research 39 (1) (1989) 1–16.
URL `https://www.sciencedirect.com/science/article/pii/0377221789903482`

[29] K. Arulkumaran, M. P. Deisenroth, M. Brundage, A. A. Bharath, Deep reinforcement learning: A brief survey, IEEE Signal Processing Magazine 34 (2017) 26–38.
URL `https://api.semanticscholar.org/CorpusID:4884302`

[30] Bio-inspired Neurocomputing, Springer Singapore, 2021.
URL `http://dx.doi.org/10.1007/978-981-15-5495-7`

[31] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: Y. W. Teh, M. Titterington (Eds.), Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, Vol. 9 of Proceedings of Machine Learning Research, PMLR, Chia Laguna Resort, Sardinia, Italy, 2010, pp. 249–256.
URL `https://proceedings.mlr.press/v9/glorot10a.html`

[32] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification (2015). `arXiv:1502.01852`.
URL `https://arxiv.org/abs/1502.01852`

[33] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing atari with deep reinforcement learning (2013). `arXiv:1312.5602`.
URL `https://arxiv.org/abs/1312.5602`

[34] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization (2017). `arXiv:1412.6980`.
URL `https://arxiv.org/abs/1412.6980`

[35] M. M. Botvinick, S. Ritter, J. X. Wang, Z. Kurth-Nelson, C. Blundell, D. Hassabis, Reinforcement learning, fast and slow, Trends in Cognitive Sciences 23 (2019) 408–422.
URL `https://api.semanticscholar.org/CorpusID:122539846`

[36] S. Zhang, R. S. Sutton, A deeper look at experience replay (2018). `arXiv:1712.01275`.
URL `https://arxiv.org/abs/1712.01275`

[37] T. Schaul, J. Quan, I. Antonoglou, D. Silver, Prioritized experience replay (2016). `arXiv:1511.05952`.
URL `https://arxiv.org/abs/1511.05952`

[38] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, N. Dormann, Stable-baselines3: Reliable reinforcement learning implementations, Journal of Machine Learning Research 22 (268) (2021) 1–8.
URL `http://jmlr.org/papers/v22/20-1364.html`

[39] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms (2017). `arXiv:1707.06347`.
URL `https://arxiv.org/abs/1707.06347`

[40] Transaction level modeling (2024).
URL `https://verificationguide.com/uvm/uvm-tlm/`