POLITECNICO DI TORINO

Master Degree Course in Electronic Engineering

Master Degree Thesis

UVM environment for RISC-V processors

Advisor
Prof. Edgar Ernesto Sanchez Sanchez
Ph.D. Annachiara Ruospo

Candidate
Leonardo Barraco

April 2021

Abstract

In the VLSI design flow, functional verification is the task of checking that the digital design is compliant with the specifications in order to find bugs in the hardware description before being mass-produced. Due to the fast growth of the design size and complexity, functional verification has become the bottleneck in the design flow. According to industry surveys, verification can take up to 70% of the total amount of time while the design phase requires around 30%. For this reason, it is necessary to develop a proper verification framework to speed up the verification phase and avoid delays in time-to-market. In this thesis, a simulation-based verification environment has been developed to verify the RISC-V RV32IMFCXpulp processor. In the UVM environment Agents are responsible for driving input test-vectors into the DUV, collecting the output transactions, and finally performing the comparison of the actual results with the expected ones. Python scripts are used to generate random constrained stimuli according to the ISA and to extract simulation results. A good verification effort must be characterized by a coverage greater than 90% as this parameter represents the confidence of the verification process. Code Coverage, with its metrics, has been used to keep track of the improvements. In order to reach a 90.1% Coverage, it was necessary to test the processor functionalities out of normal operating conditions, by injecting proper test vectors including illegal instructions (Fault Injection), interrupt requests and asynchronous resets.

Questa tesi è dedicata a Leonardo, Ignazio, Pia ed Adriana

che insieme ai miei genitori mi hanno cresciuto trasmettendomi valori e ambizioni

Contents

List of Tables

List of Figures

List of Acronyms

ALU Arithmetic Logic Unit

API Application Programming Interface

APU Auxiliary Processing Unit

AUIPC Add Upper Immediate to Program Counter

CDV Coverage Driven Verification

CSR Control and Status Registers

CSV Comma Separated Values

DUV Device Under Verification

FCC Focused Condition Coverage

FEC Focused Expression Coverage

FIFO First In First Out

FPU Floating Point Unit

FP Floating Point

FSM Finite State Machine

GPR General Purpose Register

GUI Graphic User Interface

HDL Hardware Description Language

IC Integrated Circuit

ISA Instruction Set Architecture

ISG Instruction Stream Generator

ISS Instruction Set Simulator

JALR Jump and Link Register

JAL Jump and Link

LUI Load Upper Immediate

MOS Metal-Oxide Semiconductor

OBI Open Bus Interface

OOP Object Oriented Programming

OPIMM Operand Immediate

OVM Open Verification Methodology

PC Program Count

PULP Parallel Ultra Low Power

RAM Random Access Memory

RD Register Destination

RISC Reduced Instruction Set Computer

ROM Read Only Memory

RS Register Source

RTL Register Transfer Level

SIMD Single Instruction Multiple Data

STMTS Statements

SoC System on Chip

TLM Transaction Level Modeling

UVC UVM Verification Component

UVM Universal Verification Methodology

VECOP Vectorial Operation

VHDL VHSIC Hardware Description Language

VLSI Very Large Scale Integration

eRM e Reuse Methodology

Chapter 1

Introduction

1.1 Goal of the thesis

The goal of this thesis work is to build a verification environment based on UVM methodology to verify a RISCV architecture including not only the base ISA but also extensions and proprietary extensions. It has been decided to use Code Coverage enabling all the metrics, it is important reaching a high coverage level ($> 90\%$) to ensure that both standard situation and corner cases have been verified.

1.2 Motivation

The number of transistor/IC double every 2 years

According to Moore's law[19] stated in 1965, the number of transistor per integrated circuit doubles every two years, and thanks to the technologic progress in silicon manufacturing this law is still valid. Thanks to the possibility to integrate a larger number of transistor in the same chip area, devices complexity is increasing and engineers are able to implement complex systems providing more functionalities on a single chip (SoC).
VLSI is the process of producing an IC which contain millions of MOS transistors onto a single chip. Microprocessors and memory chips are typical example of VLSI devices.
The VLSI design cycle starts with a formal specification of a VLSI chip, following a series of steps, and eventually lead to the production of a packaged chip. A typical design cycle may be represented by the flow chart shown in Fig. 1.1.

As complexity increases, the probability of having bugs in the hardware description will increase. According to "The 2020 Wilson Research Group Functional Verification Study"(Fig. 1.2), the average time spent in HDL coding represents 30%

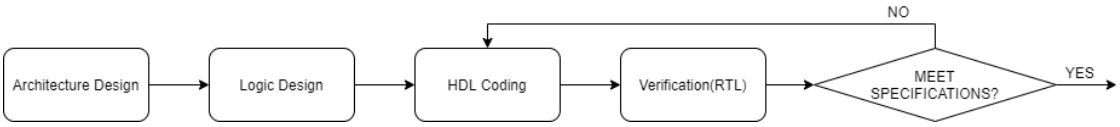


Figure 1.1: VLSI Design Flow

of the total time and around 60-70% of time is spent to verify that the architecture meets the required specification [8] . Generally, it is difficult that design meets the specification at the first verification, and delays in hardware verification lead to delays in time to market which are major issues in a company. It is clear that developing a verification framework is fundamental in order to reduce the amount of time necessary to produce a verified hardware design.

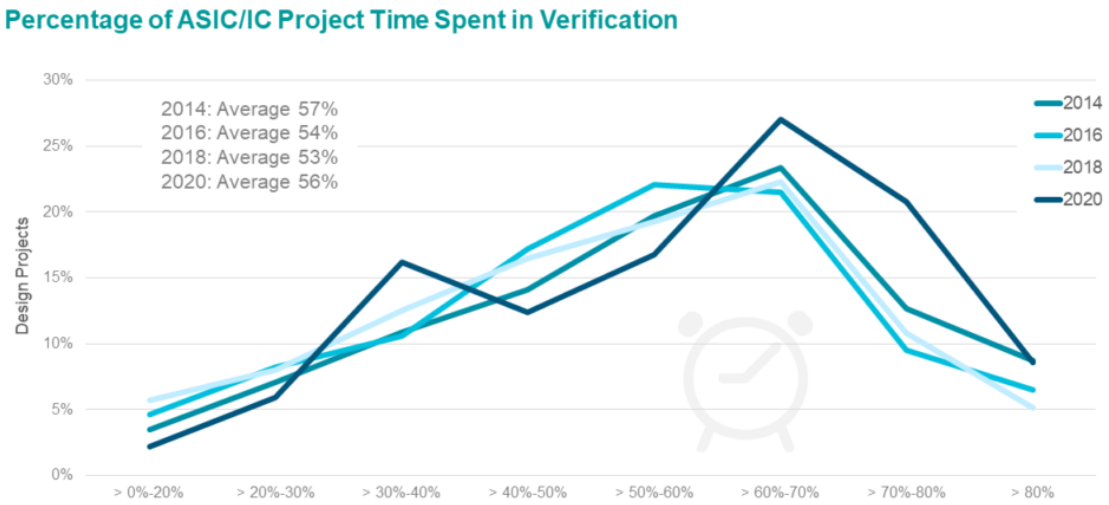


Figure 1.2: Statistics showing the increase of time spent in verification phase[8]

1.2.1 State of art

Before moving on, it would be interesting to analyze what has already been done in the field of UVM Based RISC-V Verification.

RISC-V DV

RISC-V DV is an SV/UVM based open-source RISC-V verification environment. It is available on Github[1], it has been supported by Google. The verification environment structure is reported in Fig. 1.3.

RISC-V ISG produces a constrained set of assembly programs which are then

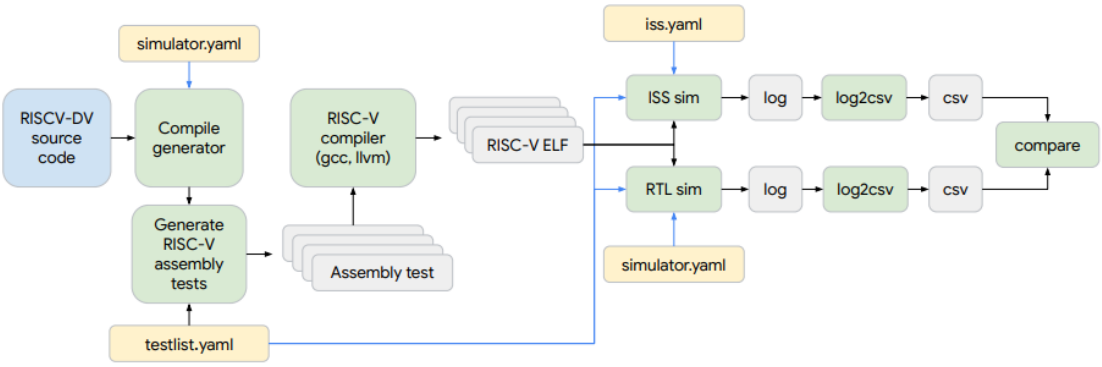


Figure 1.3: RISCV-DV framework architecture

cross-compiled and fed to the ISS and RTL. Both the DUV and the reference model write-back the log of the simulation on a `.csv` file. Finally, the log files are compared to find out any discrepancies. This Instruction Set Simulator is configurable by modifying the `ISS.yaml` file, supported ISS's are *SPIKE*,*Imperas OVPsim*, *Western Digital Whisper*, *SAIL_RISCV*.
Being UVM-based it is compatible with the major HDL simulator vendors such as *Synopsys*,*Cadence*,*Mentor Graphics*,*Metrics*. As it is an open-source project is it possible to clone it from GitHub and modify the source code to be adapted to the DUV. The Device under Verification has proprietary extensions which means customs instructions, so two modifications would be required:

- Implement custom ISA in Instruction Stream Generation;
- Implement a custom ISS capable of dealing with Pulp-proprietary extensions.

Unfortunately, the ISS developed by the RI5CY producer is not an open-source project, so a large amount of work would be necessary to develop a proper ISS.

Processor-UVM-Verification by Anish Gupta

This project, available on Github[2] has been the starting point of this thesis work.

[1]https://github.com/google/riscv-dv

[2]https://github.com/gupta409/Processor-UVM-Verification

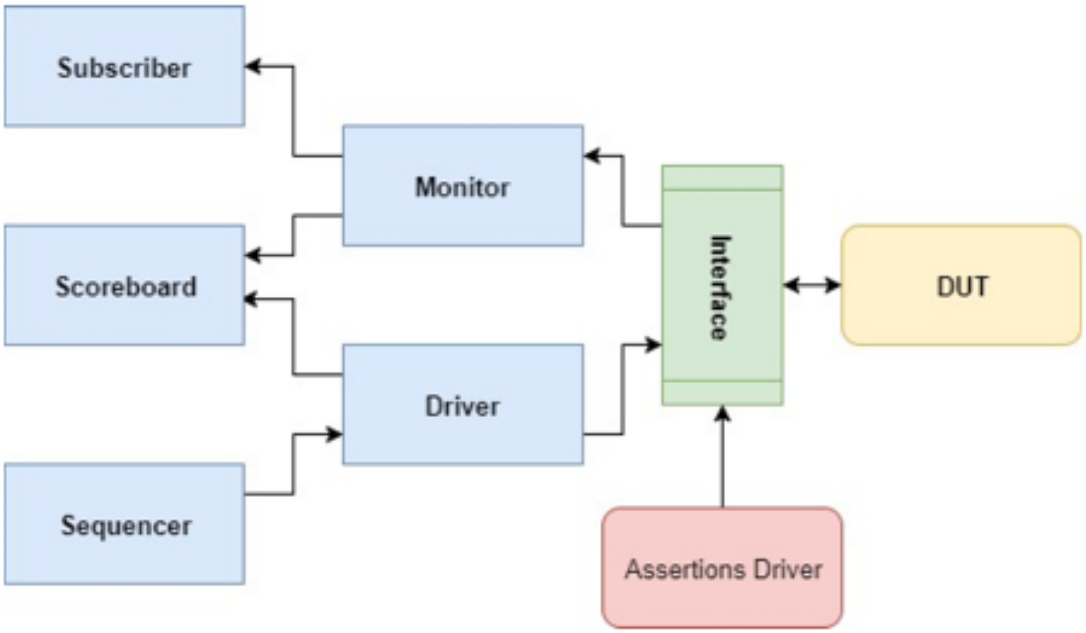


Figure 1.4: MIPS UVM framework architecture

It is a System Verilog based Verification environment for MIPS 5 staged pipelined processor. This project embeds a simple UVM environment with UVC derived from base classes. Random Instructions are generated inside the sequencer using System-Verilog constraints. In reality, this project is far different from what is needed to verify RI5CY but it represents a quite good example of a UVM framework in which the reference model is not an external ISS but is embedded in the scoreboard allowing a *run-time* check of results.

1.3 Introduction to Verification

According to Andrew Piziali, the most appropriate definition of functional verification is "Demonstrating the intent of a design is preserved in its implementation"[21]. It is important to remember that the first steps in VLSI flow have a high abstraction level, while when the design reaches the HDL coding step it is less abstract. The main consequence is that with each transformation during the design process the intent is clarified removing both ambiguity and redundancy.
The implementation is the RTL realization of the design written in an HDL such as Verilog, VHDL, SystemVerilog. Verification is a comparative process between the RTL implementation and the intent exploited to find functional logic errors. Logic errors or bugs are differences between the observed behaviour of the DUV and its

expected behaviour (intent).
This kind of errors could be caused by designers because of misinterpretation of specifications, or ambiguous specification.

1.3.1 Verification Methods

According to Andrew Piziali [21] there are 2 main methods to verify a DUV:

- Static Methods;
- Dynamic Methods.

Static Methods

Static methods are not simulation-based and use a mathematical model of the design to determine if there is any violation of the assertion. As a result, it is not required to generate and drive stimuli in the DUV to verify the design. That can be considered an advantage as the most time-consuming step in dynamic methods is the one related to the generation of the proper test vector. On the other hand, static methods have significant disadvantages related to the verification of complex architectures made up of multiple blocks.
Static methods appear to be an effective verification tool if the DUV is a small block and verification engineer is interested in its behaviour without caring about interaction with other blocks.

Dynamic Methods

Dynamic methods are simulation-based and require a simulation environment. They are characterized by simulating the DUV applying certain test vectors and comparing its response against the expected behaviour. The simulation environment should be able to record verification progress using coverage metrics.
Dynamic methods are advantageous as they allow to verify all the possible test condition. However, for a large design, it could be time-expensive as the number of test vector increase dramatically.

1.3.2 Verification Plan

The verification plan defines what must be verified in a hardware design, the verification strategy, the coverage metrics that should be set and then met to move to the next step of the design flow.
Verification plan is composed of three important aspects:

- Coverage Measurements;

- Stimulus Generation;
- Response Checking.

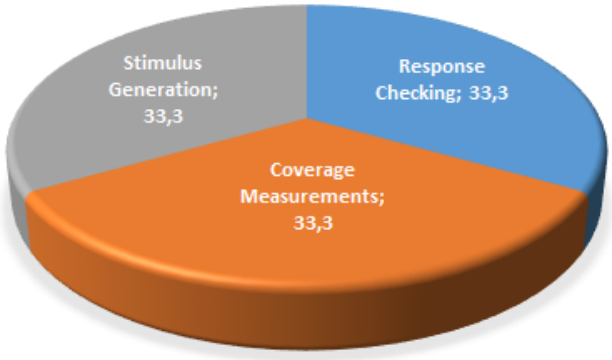


Figure 1.5: Functional Verification Aspects [21]

Coverage Measurement

The coverage measurement section of the verification plan is the one in which the verification scopes are described.
It is the most important section because determining if all bugs have been found is not possible, so metrics are required to estimate the level of coverage that has been achieved. This section includes the kinds of coverage: functional, code, assertions and eventually the metrics.

Stimulus generation

The stimulus generation part is responsible for generating the input test vector required to fully exercising the DUV and exhibiting all the possible behaviours. That means not only generating valid test vectors showing that the device is working as intended but also invalid test vectors to drive the device into corner-cases. So an important aspect of stimulus generation is verifying situation that occurs only outside of normal operating parameter in order to check the error detection logic of the DUV. The objective of stimulus generation is generating test-vectors that allow reaching a high coverage level.

Response Checking

The response checking section is responsible for verifying that DUV responses conform to the specifications. There are two different strategies:

- Reference Model Check;
- Distributed data and temporal check.

Reference Model Check

This approach requires a reference model, so a sort of implementation of the DUV at a higher abstraction level. The reference model is used alongside the DUV and receive the same input test-vectors.
The responses coming from the DUV are compared to the expected results provided by the reference model. The problem in this kind of approach is that building up a reliable reference model could lead to complex work comparable to the design process.

Distributed data and temporal check

This second strategy exploits temporal check on some monitored signals to capture device behaviour. One of the used approaches is based on Monitors and a Scoreboard in a structure like the one shown in Fig. 1.6 Input packets are captured by

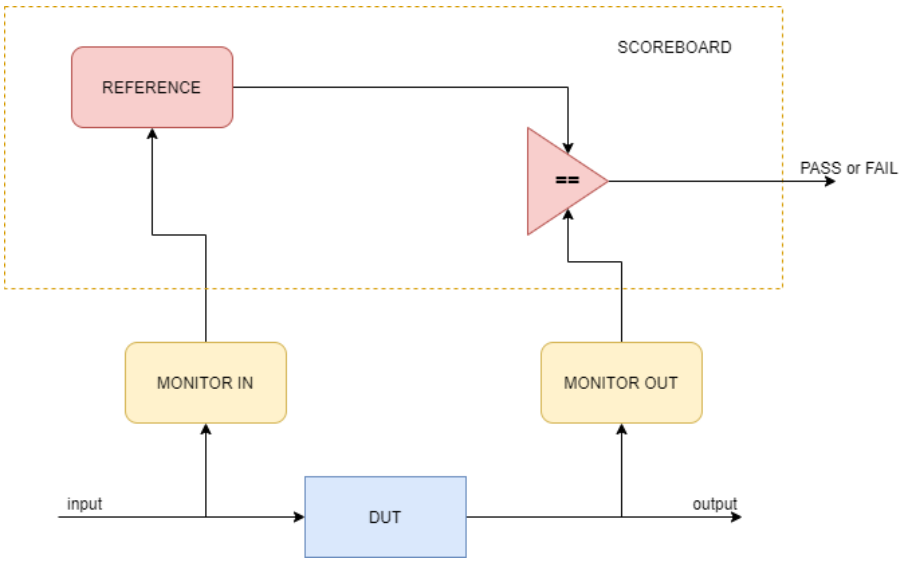


Figure 1.6: Scoreboard approach

the input monitor and sent to the reference model residing in the scoreboard while DUV outputs are collected by the output monitor and sent to the checker in the scoreboard. Inside the scoreboard, input packets are processed according to the specification to produce the expected outputs. Finally, the checker provides a Pass or Fail according to the result of the comparison.

1.4 The Universal Verification Methodology

In the previous section, the verification issue has been explained and appeared clear that there was the necessity of a universal methodology to increase the speed and the efficiency of the verification process.
UVM is a standardized methodology for verifying IC designs. UVM is derived mainly from OVM which was based on the eRM by Verisity Design. The advantages of using a universal methodology are that the best practices for an exhaustive verification are coded and UVCs are provided. It is open-source and compatible with all the major commercial simulator like Aldec, Cadence, Mentor Graphics, and Synopsys.

1.4.1 Coverage Driven Verification

UVM provides a complete framework to achieve Coverage Driven Verification combining automatic test-vector generation, self-checking testbench and coverage measurements. UVM has made it possible to create a test environment capable of exploiting "controlled randomness" of the input vectors to discover sooner design bugs. It is also possible to meet verification goals by changing testbench parameters and in this way run specific simulations to reach specific scenarios that are not easy to reach randomly (Corner cases).
Fig. 1.7 clearly shows that random tests are sufficient to reach about 50% of the

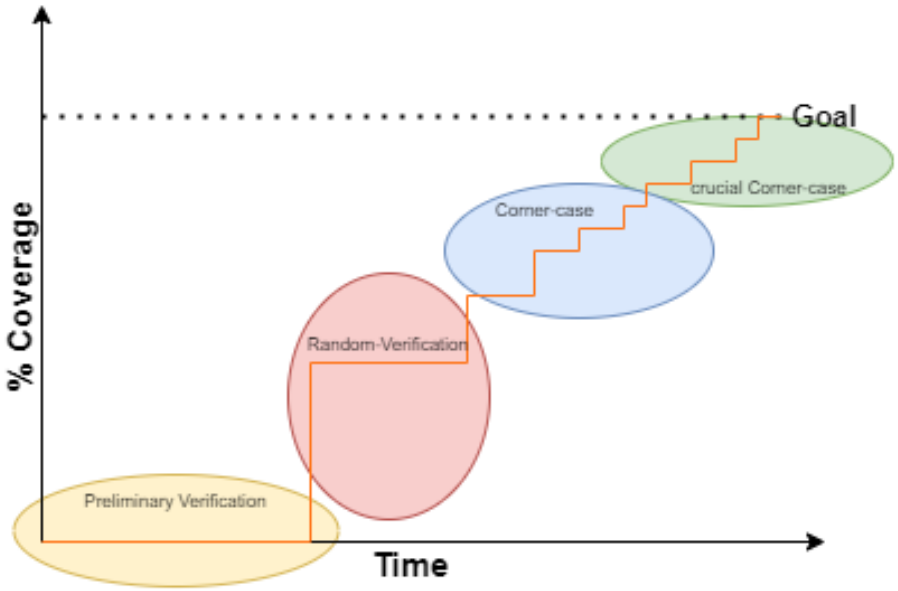


Figure 1.7: Phases of Random stimuli based verification

coverage goal. After the first random simulations, it is necessary to adjust and add

some constraint to the input sequences in order to reach corner cases.

1.4.2 UVM Components

UVM is based on OOP, this allows to increase reusability, a fundamental concept in the verification process. UVM Library provides a set of useful class from which deriving object and components, each class contains methods to deal with common operations. Thanks to OOP Verification Engineers can derive object and components from base classes and produce any modification to obtain customized classes.

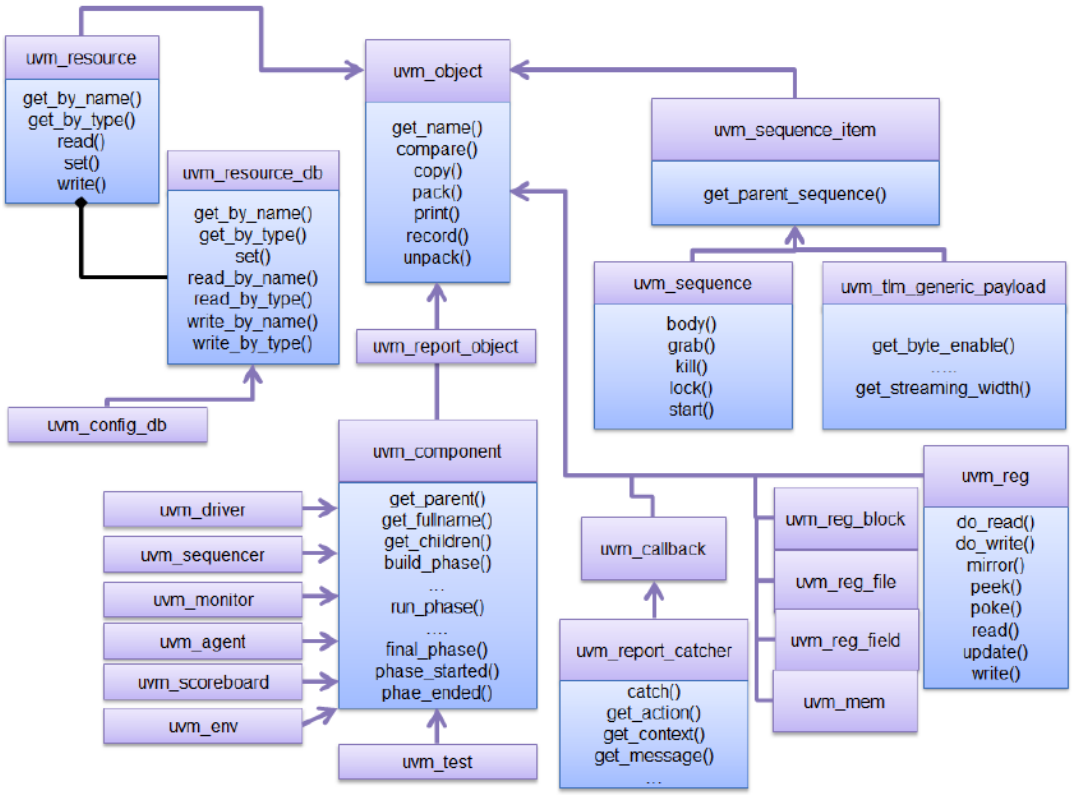


Figure 1.8: UVM Classes Diagram [13]

The `uvm_object` class is the base class for all UVM data and hierarchical classes. It contains a set of methods for common operations:

- create;
- copy;
- compare;

- print.

The `uvm_components` class contains the UVM framework components shown in Tab. 1.1.

uvm_driver	Drives signals to DUV
uvm_monitor	Monitor signals
uvm_sequencer	Create Input vectors
uvm_agent	Contains Sequencer,Driver and Monitor
uvm_env	Contain all the components of the framework
uvm_scoreboard	It represents the checker
uvm_subscriber	Receive the transaction to perform functional coverage analysis

Table 1.1: UVM components

In UVM, phases are used as a synchronization mechanism in the simulation. In this way, each component has to pass through phases and must wait for other components before moving to the next phase. The main phases are shown in Tab. 1.2

build_phase	Components build and instantiation
connect_phase	Connect components through TLM ports
end_of_elaboration_phase	Ensure that all the connection are properly set
start_of_simulation_phase	Initialization of components to avoid zero time dependencies
run_phase	During this phase time-consuming operation are performed
extract_phase	Simulation has been completed and results can be extracted
check_phase	Receive the transaction to perform functional coverage analysis
report_phase	Display result or summary of check phase

Table 1.2: UVM phases

UVM uses TLM APIs to facilitate the inter-communication between UVM components. Sequences and methods are combined to form a packet (transaction) and each UVM component can use predefined methods (put and get) to send or receive transactions.

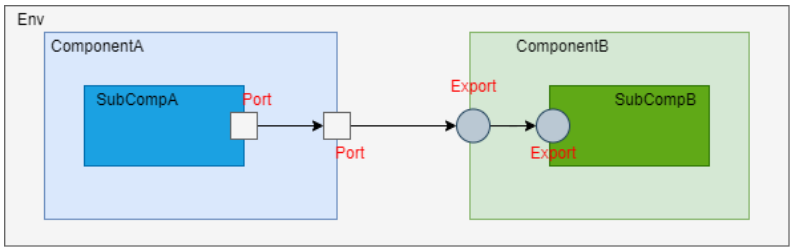


Figure 1.9: Transaction Level Modeling

1.5 Introduction to RISCV Processors

The Device Under Verification in this thesis work is RV32IMFCXpulp. It is a RISCV Processor developed by the Integrated Systems Laboratory (IIS) of ETH Zürich and Energy-efficient Embedded Systems (EEES) group of the University of Bologna in 2013.
RISC-V is an open standard ISA based on RISC principles, developed at Berkeley into the EECS Department. Originally it was developed to support computer architecture research and education but now it has become a standard architecture. The RISC-V ISA is provided under an open-source license.
The base integer ISA (RV32I) is sufficient to perform basic operation typical of a modern instruction set. It contains 40 unique instruction encoded in four different formats (R/I/S/U). Each of them has a fixed length (32 bit) and must be aligned on a four-bytes in memory. In order to simplify the decoding operations, some fields keep the same position in all formats (like opcode, source and destination register). RISC-V has 32 integer registers, with the x0 location hardwired to 0 while x1-x31 are general purpose. Except for memory access instruction, instructions operate only with registers. Load and store instructions are used to perform operations to and from memory. Apart from RV32I other extensions have been developed and are usually identified by a letter:

- "M" Standard Extension for Integer Multiplication and Division;
- "A" Standard Extension for Atomic Instructions;
- "F" Standard Extension for Single-Precision Floating-Point;
- "D" Standard Extension for Double-Precision Floating-Point;
- "Zicsr" Control and Status Register (CSR);
- "Zifencei" Instruction-Fetch Fence;
- "Q" Standard Extension for Quad-Precision Floating-Point;

- "L" Standard Extension for Decimal Floating-Point;
- "C" Standard Extension for Compressed Instructions;
- "B" Standard Extension for Bit Manipulation;
- "J" Standard Extension for Dynamically Translated Languages;
- "T" Standard Extension for Transactional Memory;
- "P" Standard Extension for Packed-SIMD Instructions;
- "V" Standard Extension for Vector Operations;
- "N" Standard Extension for User-Level Interrupts;
- "H" Standard Extension for Hypervisor;
- "Zam" Misaligned Atomics;
- "Ztso" Total Store Ordering.

Some of them has been ratified while some others are still open and subjected to change.

Chapter 2

RISC-V PULP

Before moving to the dissertation of the UVM framework, it is necessary to introduce the device under verification and its main characteristics. The DUV is RV32IMFCXpulp also known as RI5CY and it is a RISC-V processor core developed in collaboration between ETH University and the University of Bologna. RI5CY is an open-source processor provided under a permissible SolderPad open-source license. As the PULP name suggest this processor is concerned about energy efficiency avoiding power consumption when is in idle. The processor block diagram is shown in Fig. 2.1

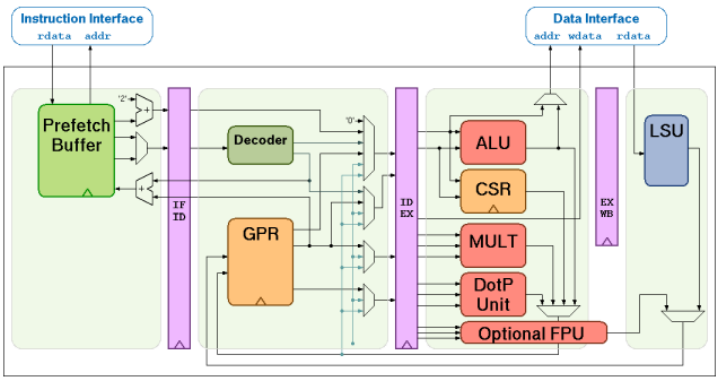


Figure 2.1: RI5CY Architecture Block Diagram

It is a 32 bit pipelined architecture with 4-stages clearly visible in the previous figure where each stage is separated by pipeline registers:

- Instruction Fetch [IF];
- Instruction Decode [ID];
- Execution [EX];

- WriteBack [WB].

It has a large Instruction set providing support for some of the RISC-V standard extensions and some proprietary extensions. As the name "RV32IMFCXpulp" suggest the supported extensions are:

- I → Base Integer Instruction Set;
- M → Integer Multiplication and Division Instruction Set;
- F → Single precision Floating point Instruction Set;
- C → Compressed Instruction Set;
- Xpulp → Pulp specific extensions including:
  - Post-incrementing load and stores;
  - Multiply and accumulate extension;
  - ALU Extensions;
  - Hardware Loops.

2.1 Complete ISA with extensions

In addition to the extensions already stated RI5CY supports also the Vectorial instructions. In this section, each of the extension will be briefly analyzed showing the available instructions.

2.1.1 Base Integer

The base integer instruction set contains:

- Integer Computational Instructions;
- Control Transfer Instructions;
- Load and Store instructions;
- Memory Ordering Instructions;
- System Instructions.

Integer Computational Instructions

Integer Computational Instructions operate on 32 bits operands stored in the integer register file. Are encoded as R-type and I-type depending on the input operands used to execute the operation. Depending on the input operands they can be furtherly divided in:

- Integer Register-Immediate Instructions (I-Type);
- Integer Register-Register Instructions (R-Type).

The destination is register `rd` for both register-immediate and register-register instructions. The R-type instructions use two operands coming from the register file according to the source addresses specified in the instruction fields while the I-type use an operand coming from the register file (`rs1`) and the other is specified in the immediate field of the instruction. Immediate must be sign-extended before being used as an operand in the execution stage. The available instructions for the register-immediate are reported in Tab. 2.1.

instr[31:20]	instr[19:15]	instr[14:12]	instr[11:7]	instr[6:0]	NAME
imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI
0000000-shamt	rs1	001	rd	0010011	SLLI
0000000-shamt	rs1	101	rd	0010011	SRLI
0100000-shamt	rs1	101	rd	0010011	SRAI
instr[31:12]		instr[11:7]	instr[6:0]	NAME	
imm[31:12]		rd	0110111	LUI	
imm[31:12]		rd	0010111	AUIPC	

Table 2.1: Base Immediate Encoding instructions

ADDI, SLTI, SLTIU, XORI, ORI, ANDI are standard operations that use the whole immediate field to obtain the immediate operand, while SLLI, SRLI and SRAI use only the 5 LSB of the immediate to define the Shift Amount (SHAMT), the other part of the immediate is still necessary for the decode operation.

The available instructions for the register-register are reported in Tab. 2.2.

instr[31:25]	instr[24:20]	instr[19:15]	instr[14:12]	instr[11:7]	instr[6:0]	NAME
0000000	rs2	rs1	000	rd	0010011	ADD
0100000	rs2	rs1	000	rd	0010011	SUB
0000000	rs2	rs1	001	rd	0010011	SLL
0000000	rs2	rs1	010	rd	0010011	SLT
0000000	rs2	rs1	011	rd	0010011	SLTU
0000000	rs2	rs1	100	rd	0010011	XOR
0000000	rs2	rs1	101	rd	0010011	SRL
0100000	rs2	rs1	101	rd	0010011	SRA
0000000	rs2	rs1	110	rd	0010011	OR
0000000	rs2	rs1	111	rd	0010011	AND

Table 2.2: Base Register Encoding instructions

Here the two operands are specified by using their register file address in fields `instr[24:20]` and `instr[19:15]` while the field `instr[31:25]` is used to decode and distinguish between ADD-SUB and SRL-SRA.

Control Transfer Instructions

RV32I provides two different types of control transfer instructions:

- Unconditional jumps;
- Conditional jumps (i.e. Branches).

The unconditional jumps instructions are JAL and JALR, these two instructions differ on the encoding (J-type for JAL and I-type for JALR) and on the behaviour. Even if both JAL and JALR stores in register rd the instruction following the jump (pc+4) the jump targets are obtained in different ways. In JAL an offset on 20 bit is explicitly provided as immediate and it is sign-extended and added to the address of the jump instruction. The jump target, in this case, is ± 1MiB range.
In JALR the target address is obtained by adding the sign-extended 12 bit immediate to the content of register rs1.
The other type of control transfer instructions is the conditional branches. In this kind of instruction the content of two registers is compared, if the resulting condition is true then the branch is taken. The branch target is obtained by sign-extending the 12-bit offset provided as immediate and added to the address of the branch instruction. There are several branch instructions that differs on the type of comparison.

- BEQ → Branch if is equal;
- BNE → Branch if not equal;
- BLT/BLTU → Branch if lower (Signed and Unsigned);

- BGE/BGEU → Branch if greater equal (Signed and Unsigned);

- BLE/BLEU → Branch if lower equal (Signed and Unsigned);

- BGT/BGTU → Branch if greater (Signed and Unsigned).

All the control transfer instructions are summarized in Tab. 2.3

instr[31:25]	instr[24:	instr[19:	instr[14:	instr[11:7]	instr[6:0	NAME
imm[12],imm[10:5]	rs2	rs1	000	imm[4:1],imm[11]	1100011	BEQ
imm[12],imm[10:5]	rs2	rs1	001	imm[4:1],imm[11]	1100011	BNE
imm[12],imm[10:5]	rs2	rs1	100	imm[4:1],imm[11]	1100011	BLT
imm[12],imm[10:5]	rs2	rs1	101	imm[4:1],imm[11]	1100011	BGE
imm[12],imm[10:5]	rs2	rs1	110	imm[4:1],imm[11]	1100011	BLTU
imm[12],imm[10:5]	rs2	rs1	111	imm[4:1],imm[11]	1100011	BGEU

instr[31:12]	instr[11:7]	instr[6:0]	NAME
imm[20],imm[10:1],imm[11],imm[19:12]	rd	0110111	JAL

instr[31:20]	instr[19:15]	instr[14:12]	instr[11:7]	instr[6:0]	NAME
imm[11:0]	rs1	000	rd	0010111	JALR

Table 2.3: Control Transfer Instructions

Load and Store Instructions

RISC-V processors are load and store architecture, standard arithmetic instructions are not allowed to read or write data memory. Only load and store instructions can access RAM to read and write data. This kind of operations is used to transfer values between the registers and memory. In particular, load instructions copy a value from the memory to register rd and stores copy the value contained in `rs2` in data memory.
The effective memory address is obtained by adding the content of register `rs1` to the 12-bit sign-extended offset. Load and store instructions can work not only with the complete 32-bit word but also with half-words or byte and in those cases, the lower part of the 32-bit data is used. Load and store instructions are summarized in Tab. 2.4

instr[31:20]	instr[19:15]	instr[14:12]	instr[11:7]	instr[6:0]	NAME
imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU
instr[31:25]	instr[24:20]	instr[19:15]	instr[14:12]	instr[11:7]	instr[6:0]	NAME
imm[11:5]	rs2	rs1	000	imm[4:0]	0000011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0000011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0000011	SB

Table 2.4: Load and Store instructions

Memory Ordering Instructions & System Instructions

Memory ordering instructions i.e. FENCE or FENCE-I are used to order device I/O and memory accesses as viewed by other hardware threads, coprocessors and external devices.
System instructions are privileged instructions that in some cases require a certain privilege level. These instructions can be divided into two main groups:

- CSR Operations;
- Privileged.

instr[31:20]	instr[19:15]	instr[14:12]	instr[11:7]	instr[6:0]	NAME
fm-pred-succ	rs1	000	rd	0001111	FENCE
imm[11:0]	rs1	001	rd	0001111	FENCE.I
000000000000	00000	000	00000	1110011	ECALL
000000000001	00000	000	00000	1110011	EBREAK
001100000010	00000	000	00000	1110011	MRET
000000000010	00000	000	00000	1110011	URET
011110110010	00000	000	00000	1110011	DRET
000100000101	00000	000	00000	1110011	WFI
csr	rs1	001	rd	1110011	CSRRW
csr	rs1	010	rd	1110011	CSRRS
csr	rs1	011	rd	1110011	CSRRC
csr	uimm	101	rd	1110011	CSRRWI
csr	uimm	110	rd	1110011	CSRRSI
csr	uimm	111	rd	1110011	CSRRCI

Table 2.5: System instructions

In Tab. 2.5 are summarized all the instructions of this type. The CSR instructions

atomically read-modify-write a single Control and Status Register. The CSR address is provided as immediate in `instr[31:20]`. There are two different versions of the same (CSRRW, CSRRS, CSRRC) instructions, the standard one in which rs1 register is used as an operand and another one in which the operand is provided as a 5-bit immediate value to be zero-extended. Privileged instructions are like ECALL, EBREAK, and so on, which are used to make a service request or to return from a service request.

2.1.2 Multiplication Extension

Multiplication extension contains instructions that are used to multiply and divide values coming from integer register file:

- MUL : 32bit x32bit multiplication, lower 32 bit are stored in rd;
- MULH : 32bit x32bit multiplication, higher 32 bit are stored in rd;
- MULHU: unsigned(32bit) x unsigned(32bit) multiplication, higher 32 bit are stored in rd;
- MULHSU: signed(32bit) x unsigned(32bit) multiplication, higher 32 bit are stored in rd;
- DIV : signed(32 bit)/signed(32 bit) division with rounding toward zero;
- DIVU : unsigned(32 bit)/unsigned(32 bit) division with rounding toward zero;
- REM : return the remainder of the signed division;
- REMU : return the remainder of the unsigned division.

instr[31:25]	instr[24:20]	instr[19:15]	instr[14:12]	instr[11:7]	instr[6:0]	NAME
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Table 2.6: Mul/Div Instructions

2.1.3 Compressed extension

The compressed extension named "C" allows a reduction of static and dynamic code size by adding short 16-bit instruction encodings for common integer operations. Exploiting compressed instructions a reduction of code size around 25% is achieved. In general, in order to keep unchanged the processor architectures and support C extension a compressed decoder is introduced in the fetch stage. Its role is to extend a 16-bit instruction to its correspondent on 32-bit, in this way is not required to change the decoder in the decode stage. Compressed instructions are reported in Tab. 2.7, Tab. 2.8 and Tab. 2.9

instr[15:13]	instr[12:9]	instr[9:7]	instr[6:5]	instr[4:2]	instr[1:0]	NAME
001	uimm[5:3]	rs1	uimm[7:6]	rd	00	C.FLD
010	uimm[5:3]	rs1	uimm[7:6]	rd	00	C.LW
011	uimm[5:3]	rs1	uimm[7:6]	rd	00	C.FLW
101	uimm[5:3]	rs1	uimm[7:6]	rs2	00	C.FSD
110	uimm[5:3]	rs1	uimm[7:6]	rs2	00	C.SW
111	uimm[5:3]	rs1	uimm[7:6]	rs2	00	C.FSW
instr[15:13]	instr[12:5]		instr[4:2]	instr[1:0]		NAME
000	nzuimm[5:4|9:6|2|3]		rd	00		C.ADDI4SPN

Table 2.7: Compressed Instructions Quadrant 0

instr[15:13]	instr[12]	instr[11:7]	instr[6:2]	instr[1:0]	NAME
000	nzimm[5]	0	nzimm[4:0]	01	C.NOP
000	nzimm[5]	rs1/rd!=0	nzimm[4:0]	01	C.ADDI
010	imm[5]	rd!=0	imm[4:0]	01	C.LI
011	nzimm[17]	rd!={0,2}	nzimm[16:12]	01	C.LUI
instr[15:13]	instr[12]	instr[11:1	instr[9:7]	instr[6:2]	instr[1:0]	NAME
100	nzuimm[5]	00	rs1/rd	nzuimm[4:0]	01	C.SRLI
100	nzuimm[5]	01	rs1/rd	nzuimm[4:0]	01	C.SRAI
100	imm[5]	10	rs1/rd	imm[4:0]	01	C.ANDI
100	0	11	rs1/rd	00-rs2	01	C.SUB
100	0	11	rs1/rd	01-rs2	01	C.XOR
100	0	11	rs1/rd	10-rs2	01	C.OR
100	0	11	rs1/rd	11-rs2	01	C.AND
instr[15:13]	instr[12:10]	instr[9:7]	instr[6:2]	instr[1:0]		NAME
110	imm[8|4:3]	rs1	imm[7:6|2:1|5]	01		C.BEQZ
111	imm[8|4:3]	rs1	imm[7:6|2:1|5]	01		C.BNEZ

Table 2.8: Compressed Instructions Quadrant 1

instr[15:13]	instr[12]	instr[11:7]	instr[6:2]	instr[1:0]	NAME
000	nzuimm[5]	rs1/rd!=0	nzuimm[4:0]	10	C.SLLI
001	uimm[5]	rd	uimm[4:3|8:6]	10	C.FLDSP
010	uimm[5]	rd	uimm[4:2|7:6]	10	C.LWSP
011	uimm[5]	rd	uimm[4:2|7:6]	10	C.FLWSP
100	0	rs1!=0	0	10	C.JR
100	0	rd!=0	rs2!=0	10	C.MV
100	1	0	0	10	C.EBREAK
100	1	rs1!=0	0	10	C.JALR
100	1	rs1/rd !=0	rs2!=0	10	C.ADD
instr[15:13]	instr[12:7]	instr[6:2]	instr[1:0]	NAME	
101	uimm[5:3|8:6]	rs2	10	C.FSDSP	
110	uimm[5:2|7:6]	rs2	10	C.SWSP	
111	uimm[5:2|7:6]	rs2	10	C.FSWSP	

Table 2.9: Compressed Instructions Quadrant 2

2.1.4 Post-incrementing Load and Store Instructions

Post-incrementing load and store instructions belong to the proprietary extension of XPulp. This kind of operation perform a load or a store and at the same time increment the address that was used for the memory access. There are two versions that differ on the offset encoding:

- Register-Register (offset come from the register file);
- Register-Immediate (offset is encoded as immediate).

In both of them, the modified address is written back in the register file (`rs1`).

imm[11:0]	rs1	funct3	rd	opcode	name
offset	base	000	dest	0001011	p.lb
offset	base	100	dest	0001011	p.lbu
offset	base	001	dest	0001011	p.lh
offset	base	101	dest	0001011	p.lhu
offset	base	010	dest	0001011	p.lw

Table 2.10: Register-Immediate loads with post increment

funct7	rs2	rs1	funct3	rd	opcode	name
0000000	offset	base	111	dest	0001011	p.lb
0100000	offset	base	111	dest	0001011	p.lbu
0001000	offset	base	111	dest	0001011	p.lh
0101000	offset	base	111	dest	0001011	p.lhu
0010000	offset	base	111	dest	0001011	p.lw

Table 2.11: Register-Register loads with post increment

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	name
offset[11:5]	src	base	000	offset[4:0]	0101011	p.sb
offset[11:5]	src	base	001	offset[4:0]	0101011	p.sh
offset[11:5]	src	base	010	offset[4:0]	0101011	p.sw

Table 2.12: Register-Immediate stores with post increment

funct7	rs2	rs1	funct3	rs3	opcode	name
0000000	src	base	100	offset	0101011	p.sb
0000000	src	base	101	offset	0101011	p.sh
0000000	src	base	110	offset	0101011	p.sw

Table 2.13: Register-Register stores with post increment

2.1.5 Hardware Loops

Hardware loops extensions aim to increase the efficiency of small loops in code. In fact, make it possible to execute a certain amount of instruction multiple times without overhead. In order to set up a hardware loop 3 information are required:

- start address;
- end address;
- counter.

These pieces of information are provided through hardware loop instructions. There are two possibilities to set up a hardware loop, the first one is using long commands in which the information is provided using three different instructions, while the second one is using a single instruction to set the three values. The main difference is that short command allows a limited range for the number of instructions contained in the loop. RI5CY support two levels of nested hardware loops so when a hardware loop is set, the level (0 or 1) must be specified in `instr[7]`. Hardware loops and their encoding is summarized in Tab. 2.14

uimmL	rs1	funct3	0000	L	opcode	name
uimmL[11:0]	00000	000	0000	L	1111011	lp.starti
uimmL[11:0]	00000	001	0000	L	1111011	lp.endi
000000000000	src1	010	0000	L	1111011	lp.count
uimmL[11:0]	00000	011	0000	L	1111011	lp.counti
uimmL[11:0]	src1	100	0000	L	1111011	lp.setup
uimmL[11:0]	uimmS[4:0]	101	0000	L	1111011	lp.setupi

Table 2.14: Hardware Loop instruction encoding

2.1.6 ALU Extension

ALU extensions belong to the Xpulp proprietary extension and aim to extend the base instruction set with:

- Bit-Manipulation instructions;
- General ALU instructions;
- Immediate Branching instructions.

As the ALU extension contains a large number of instruction their encoding tables are reported in Appendix A.

2.1.7 Vectorial

Vectorial Instructions is the extension that allows performing operations on subword elements at the same time by splitting the datapath into smaller parts (SIMD). Vectorial instructions can work either on 8-bit(byte) or 16-bit(halfword), and in addition to that three modes influences the second operand.

- 8-bit:
  - Normal mode (vector-vector operation);
  - Scalar Replication (Operand 2 is treated as a scalar and replicated 4 times to form a complete vector);
  - Immediate Scalar Replication (Operand 2 comes from immediate and has to be replicated 4 times).
- 16-bit:
  - Normal mode (vector-vector operation);
  - Scalar Replication (Operand 2 is treated as a scalar and replicated 2 times to form a complete vector);

– Immediate Scalar Replication (Operand 2 comes from immediate and has to be replicated 2 times).

Finally, Vectorial instructions are divided into ALU operations and comparison operations. Vectorial comparisons are done on bytes or on half-words and if the comparison result is true then all the bits of that byte/half-word are set to 1, otherwise to 0. As the Vectorial extension contains a large number of instruction their behaviour tables are reported in Appendix B.

2.2 PULP Architecture

Starting from the complete block diagram shown in Fig. 2.2 in this section is going to be described each of the architectural block providing a brief explanation of the functionalities.

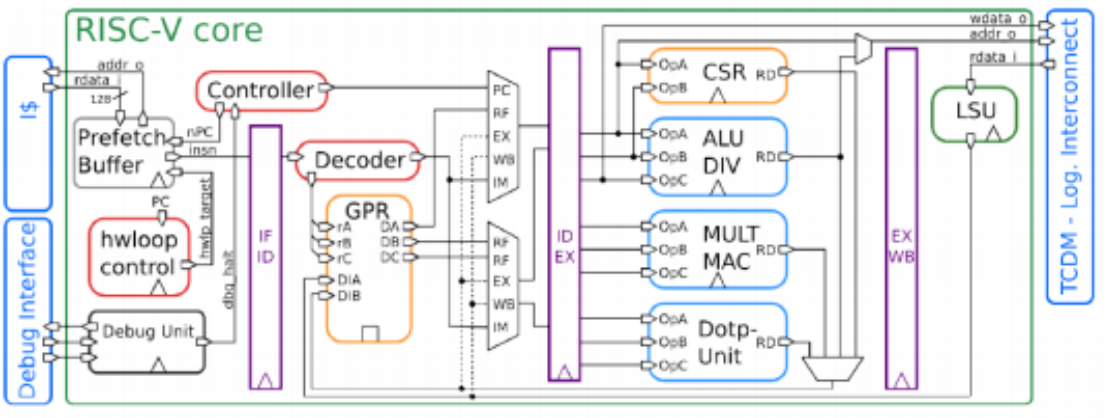


Figure 2.2: RI5CY Architecture Block Diagram

2.2.1 Instruction Fetch stage

Instruction Fetch is the first stage, it is responsible for providing `addr_o` to the instruction memory, and get the correspondent instruction stored at the given address. its main architectural blocks are:

- Prefetch Buffer;
- HwLoop Controller;
- Debug Unit;
- Controller;

Prefetch buffer is the component that actually fetches instructions from the instruction memory or instruction cache. It is available in two different versions:

- 32-bit prefetcher: It allocates a 3 entries FIFO which stores the fetched instruction words;
- 128-bit prefetcher: It stores a 128-bit wide cache line.

The usage of 128-bit or 32-bit prefetcher depends on the setting of `INSTR_RDATA_WIDTH`, and according to its value, only one prefetcher is allocated.
Hwloop Controller is the component responsible for handling hardware loops. Here the current program counter is compared to all the hw-loop end address, and jump to the right start address if the counter is equal to 0. It has a modular approach, and it is configured by setting the `N_REGS` parameter, in RI5CY, it is set to 2 because only 2 nested hardware loops are supported.
The Debug Unit is directly connected to the RI5CY Debug Interface and has a signal `debug_req_i`. That request signal makes the core jumps to a specific address where the debug ROM is mapped. This address is defined through `DM_HaltAddress` parameter.
The RISC-V controller is the main controller of the CPU, it receives signals from all the pipeline stages and according to their transition is able to handle exceptions, interrupt and normal execution.

2.2.2 Instruction Decode stage

Even if the instruction decode stage contains only two components it is an important part of the architecture as it is responsible for the de-codification of the instruction and consequently to provide proper signals to the Execution stage. In addition to that is responsible for providing correct input operands to be delivered to the ALU. Its components are the decoder and the GPR.
As explained in the previous section operands can be either stored in the register file or provided as immediate. The immediate extension is performed in this stage and all the possible combination of operands are connected to two multiplexers. According to the signals set inside the decoder, the correct operands are selected and delivered to the next stage.
GPR is a register file with 32 locations, each can contain a 32-bit word, the location x0 is hardwired to 0 and it's not possible to overwrite it. It has 3 read ports (necessary for three operands operations) and 2 write ports (Write port A is connected to the load and store unit while Write port B is connected to the Execution stage output). If FPU is used then an additional 32-bit Floating point register file is allocated.

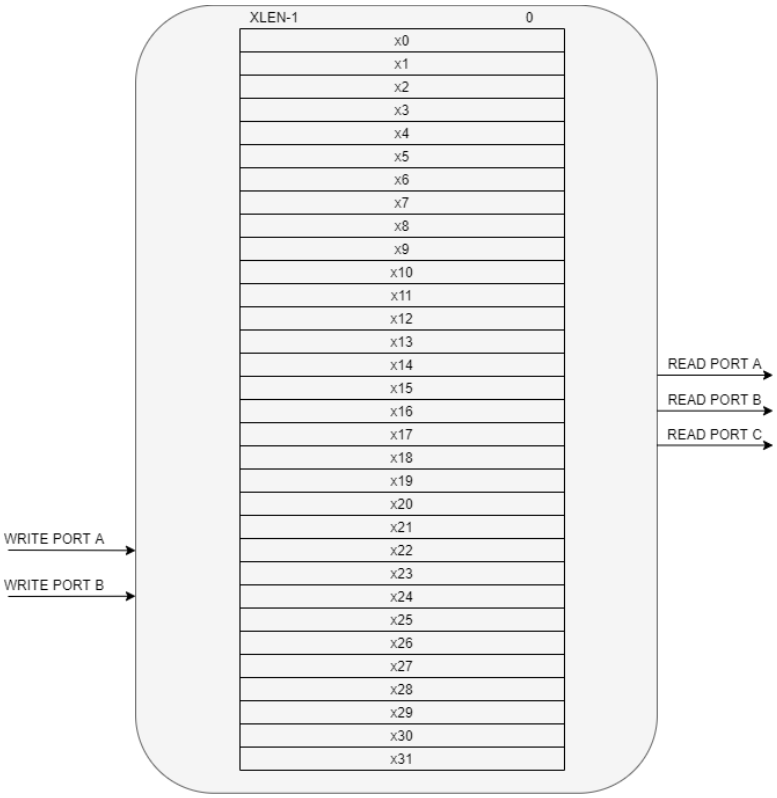


Figure 2.3: General Purpose Register File

2.2.3 Execution stage

The execution stage is responsible for the execution of the operation. It receives proper signal and operands from the decode stage. Operands and signals are then connected to the different Computational blocks allocated. As each of the computational blocks computes a result, the correct result is selected through a multiplexer. In case of load and store operations, the operands (i.e. the data to be stored and the correspondent R/W address) are forwarded to the load and store unit.
The computational blocks allocated are:

- ALU;
- Multiplier;
- FPU;
- APU;
- CSR.

ALU is responsible for the arithmetic operation and also vectorial arithmetic operation, for instance: Shift, Comparisons, Shuffle, BitManipulation and standard logic-arithmetic.
The multiplier is responsible for Integer multiplication, DotP multiplication (i.e. Multiply and Accumulate operations) and operation with complex numbers.
FPU, when enabled, is used to compute the result of operation involving single-precision floating-point operands. The APU is enabled together with the external floating-point unit (`fpnew_pkg`), exploiting an OBI-interface APU and FPU are able to communicate and results of operations computed outside of the core are available in the execution stage.
Control and Status Register are allocated in the execution stage in order to make it possible to execute atomic CSR instructions. In fact CSR instructions read the actual value stored in the control and status register and save it in rd, and at the same time, the value is overwritten.

2.2.4 WB Stage

The last stage is the write-back, in reality, as shown before the writeback is not responsible for the register file store operation that is going to be performed at the end of the execution stage. The operations done in this stage are mainly related to the load and store operation involving data memory. In case of misaligned memory access, is necessary to sign-extend data read from data memory, this operation is performed inside the load and store unit to provide the final operand to be stored in the register file.

Chapter 3

UVM Testbench

The starting point of this work is the realization of the UVM Framework. According to the reuse philosophy of UVM, all the components are derived from the UVM base classes exploiting inheritance.
Starting from the given UVM testbench, a large number of architectural modifications were required to fit the UVM framework to the device under verification. For instance, it was necessary to introduce an additional agent with all its subcomponents in order to capture input and output transaction from the DUV. The agent in charge of capturing input transition is also responsible for driving the input sequence, for this reason, it can be considered an active agent (as it includes the UVM Driver) while the other agent is a passive entity and its role is to collect internal signals of the DUV, packing them and put transactions to the scoreboard. It was also necessary to define two different interfaces, each of them represents the set of signals to be captured from the DUV at different time instant.

3.1 Overall Structure

In this section the framework structure is briefly analyzed, highlighting the main components and their role in the verification process. A schematic view is shown in Fig. 3.1. The two main components are the `tb_top` and the `uvm_test`. The wrapper included in the `tb_tob` is a useful component that has been created to encapsulate the device under-verification and the RAM. Even if the RAM is not to be verified, it is an essential component to make the processor working correctly. In fact, as the RISC-V processor includes load and stores instructions a $2^{(XLEN)}$ memory is required to be inserted alongside the device. The uvm_test instantiate the UVM environment that includes 3 main blocks:

- Agent_in;
- Agent_out;

- Scoreboard.

`Agent_in` is an active agent and is capable of driving and capturing signals. In particular, the sequencer is the component in charge of generating random test-vectors according to the processor interface, the generated inputs are then sent to the driver. Once the driver receives the input transactions from the sequencer, it dispatches them to the DUV exploiting the input interface and respecting a certain protocol. `Monitor_in`, on the other hand, is the input transaction collector, its role is to capture a set of signal inside the DUV and send them to the scoreboard.
`Agent_out` is simply a passive entity capable of catching output transaction (actual results of the operation). Output transaction is then sent to the scoreboard.
Once the scoreboard has received the input transaction (input stimuli) and the output transaction (results) it evaluates the expected results of the operation and compares them to the actual results, providing a pass or fail.

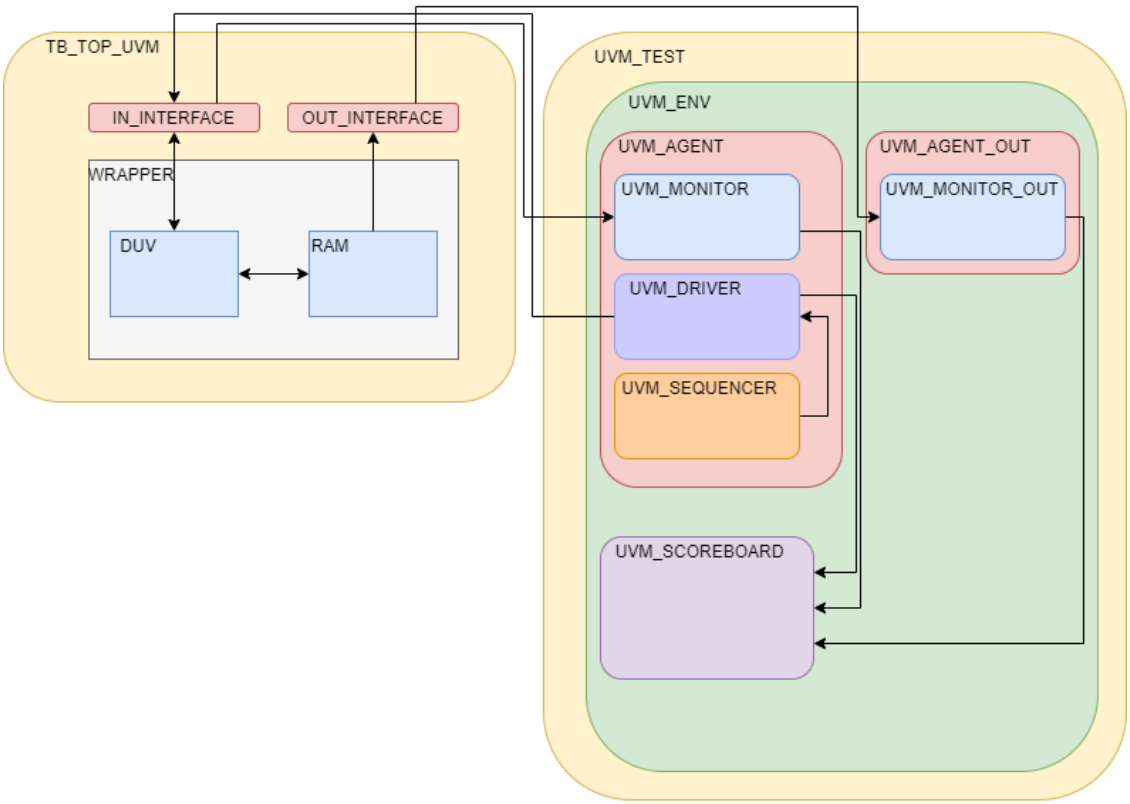


Figure 3.1: UVM Framework Structure

3.2 Top

The top hierarchy of the UVM testbench is *tb_top_uvm.sv*. The operation performed in this module are the following:

- Clock generation;
- Reset de-assertion;
- Interfaces assignments;
- Wrapper instantiation;
- test run.

The first two operations are fundamental to define the system clock and to de-assert the `rst_n` after a certain number of cycles defined in the const int `RESET_WAIT_CYCLES`. Interface assignments are the operations necessary to define the connection between the DUV and the UVM interface. All the required signals have to be written hierarchically in the interfaces declarations in order to be available in the UVM framework. Exploiting this approach any of the UVM components that has a connection with the interfaces is able to catch signals coming from the device under verification.
After that Wrapper is instantiated UVM test is launched using `run_test()`.

3.3 Wrapper

The wrapper is the component used to encapsulate the device under test i.e. `riscv_core.sv` and the RAM `data_ram.sv`. Inside the wrapper signals to read and write from and to memory are connected to the core, and some other signals are set to their default value. In this way, the device to be instantiated in the `tb_top` is the wrapper and its signals, both input and output are reduced. The wrapper allows having a clear interface in which only the signals to be driven by UVM Framework are available. Other connections are hidden inside the wrapper module.

3.4 Interfaces

In UVM the DUV is static, as a result, the communication between the testbench and the DUV cannot be done as users used to do in classic test-benches. In UVM Virtual interface feature is used, it represents a collection of signals used to drive and monitor the DUV from the testbench. The direction of the signals is decided by the `mod-ports`. In addition to that clocking-blocks are used to synchronize the

sampling instant of the signals belonging to the same block. In order to better understand how virtual interfaces work, they can be considered as a handle pointing to the interface instance. Using this approach the testbench can access the DUV signals through the virtual interface and vice versa. Interfaces are defined in `processor_interface.sv` and `processor_interface_out.sv`. Each of the UVM components that need to monitor or drive interface signals must declare a virtual interface instance of that interface and get the reference of the interface from the UVM configuration database.

3.4.1 Interface in

The input interface, described in `processor_interface.sv`, contains a set of signals to be driven and another set of signals to be monitored. According to that two `mod-ports` and their `clocking-block` are defined. The input interface block scheme is shown in Fig. 3.2.

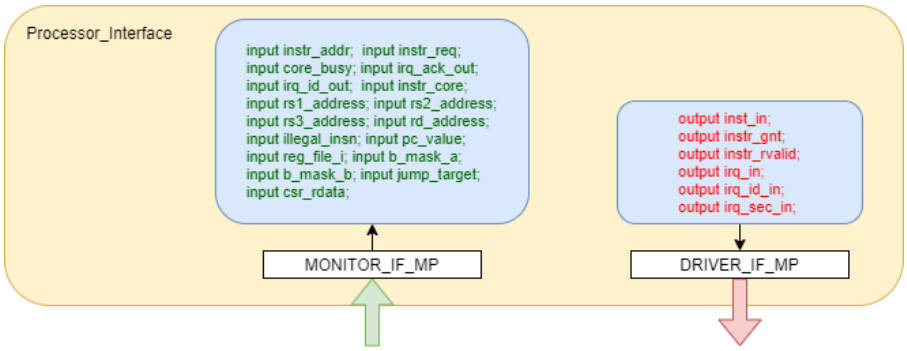


Figure 3.2: Processor interface block diagram

driver_cb

Signals to be driven belong to the `driver_cb`, are synchronized to the positive edge of the clock signal and allows the driver to push input vectors to the DUV at each clock cycle.

```
1            clocking driver_cb @ (posedge clk);
2        //Instruction signals
3        output inst_in;
4        output instr_gnt;
5    output instr_rvalid;
6        //Interrupt signals
7        output irq_in;
8        output irq_id_in;
9        output irq_sec_in;
```

```
10 endclocking : driver_cb
```

Signals in `driver_cb` are defined as output, even if it seems to be not very intuitive, it depends on the fact the interface is considered from the testbench point of view, so the signals which are modified by the driver are the output of the testbench and input of the DUV.

Monitor_cb

The signals belonging to the `monitor_cb` are synchronized to the negative edge of the clock signal to be sure that the input transaction has been delivered to the DUV. For the same reason explained before, here signals are defined as input.

```
1
2   clocking monitor_cb @ (negedge clk);
3         //Instruction signals
4         input instr_addr;
5         input instr_req;
6         input core_busy;
7         //Interrupt signals
8         input irq_ack_out;
9         input irq_id_out;
10        //Internal DUV Signals
11        input instr_core;
12        input rs1_address;
13        input rs2_address;
14        input rs3_address;
15        input rd_address;
16        input illegal_insn;
17        input pc_value;
18        input reg_file_i;
19        input b_mask_a;
20        input b_mask_b;
21        input jump_target;
22        input csr_rdata;
23  endclocking : monitor_cb
```

3.4.2 Interface out

The out interface is used to collect internal signals of the DUV after that operations have been completed. It represents the collection of the actual results provided by the DUV. Here signals are going only from the device to the testbench, for this reason, there is no need to insert a driver clocking block.The input interface block scheme is shown in Fig. 3.3.

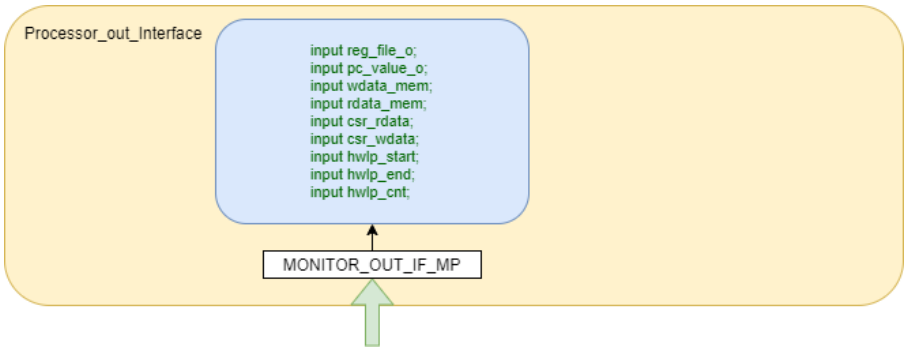


Figure 3.3: Processor out interface block diagram

All the signals defined in the out interface belong to the `monitor_out` clocking block.

```
1     clocking monitor_out_cb @ (negedge clk);
2       input reg_file_o;
3       input pc_value_o;
4       input wdata_mem;
5       input rdata_mem;
6       input csr_rdata;
7       input csr_wdata;
8       input hwlp_start;
9       input hwlp_end;
10      input hwlp_cnt;
11      endclocking : monitor_out_cb
```

3.5 Sequences

The UVM System Verilog library provides the `uvm_sequence_item` as a base class to describe data items. Data items are the transactions either to be collected or to be driven in the DUV. Transaction items in the verification framework are derived from the class `uvm_sequence_item` which provide a set of useful methods to randomize transaction fields and to compare or print transaction objects.
The sequence-item is composed of data fields required to generate the stimulus and in that case, are defined as `rand` and can have constraint ranges defined. Data fields can represents also analysis information coming from the DUV for example responses, internal signals, error signals. In the UVM framework developed for processor under-verification, there are two different sequence objects. The first one is related to the input transactions while the other one is associated with the output transactions. The definition of the two sequence object is in `processor_sequence.sv` and in `packet_out.sv`.

3.5.1 Processor Sequence

The transaction object in processor sequence is `processor_transaction`, in the following snippet of code the signals included are shown. Some of them are defined as rand to randomize input values and create input vectors. In reality, it was used as a first approach to verify that the UVM environment was working properly. As soon as it was clear that the environment was working properly an external python program was used to generate random instructions to be fed to the processor. That choice was done to simplify the randomization procedure, in fact, as the DUV has a large instruction set it was too complicated to deal with the UVM randomization feature.

```
1  class processor_transaction extends uvm_sequence_item;
2
3    `uvm_object_utils(processor_transaction)
4
5
6      bit instr_gnt;
7      bit instr_rvalid;
8      bit instr_addr;
9      bit instr_req;
10     bit irq_in;
11     bit irq_id_in;
12     bit irq_sec_in;
13     bit irq_ack_out;
14     bit irq_id_out;
15     bit core_busy;
16     bit [31:0] instr;
17     bit [31:0] instr_core;
18     bit [31:0] instrn;
19     bit [4:0] rs1_address;
20     bit [4:0] rs2_address;
21     bit [4:0] rs3_address;
22     bit [4:0] rd_address;
23     bit [31:0] pc_value;
24     bit [31:0] jump_target;
25     bit [31:0][31:0] reg_file;
26     bit [4:0] b_mask_a;
27     bit [4:0] b_mask_b;
28     bit [31:0] csr_rdata;
29     bit illegal_insn;
30     //RANDOMIZATION
31     rand bit [11:0] immediate;
32     rand bit [4:0] rs1;
33     rand bit [4:0] rs2;
```

```
34      rand bit [4:0] rs3;
35      rand bit [4:0] rd;
36
37
38
39  constraint my_range_1 {rs1 >=5'b00000; rs1<5'b11111; }
40  constraint my_range_2 {rs2 >5'b00000; rs2<5'b11111; }
41  constraint my_range_4 {rs3 >5'b00000; rs3<5'b11111; }
42  constraint my_range_3 {rd >5'b00001; rd<5'b11111; }
43  constraint myrange4 {immediate>12'b0; rd<12'b00111111111;}
44
45    function new (string name = "");
46      super.new(name);
47    endfunction
48
49  endclass: processor_transaction
```

In `processor_sequence` are also defined the sequencer operations. In the body task of `inst_sequence`, an object of type `processor transaction` is created. The random instructions that were written in `instructions.txt` by the random generator, are read line by line and associated with the transaction object. Then `finish_item()` method is used to signal that the transaction object has been completed. Now the sequencer is responsible for redirecting the created sequence to the driver. In the last part of the `processor_sequence`, the behaviour of the sequence is described through a simple for loop, in this way each time that the driver calls `get_next_item()` a new transaction is available.

3.5.2 Packet out

Packet out is simply the container of the output transaction and it is extended from the `uvm_sequence_item` base class. The signals to be captured from the DUV are shown in the following piece of code.

```
1   class packet_out extends uvm_sequence_item;
2
3     `uvm_object_utils(packet_out)
4
5           bit [31:0][31:0] reg_file;
6           bit [31:0] pc_value;
7           bit [31:0] wdata_mem;
8           bit [31:0] rdata_mem;
9           bit [31:0] csr_rdata;
10          bit [31:0] csr_wdata;
11          bit [31:0] hwlp_start;
12          bit [31:0] hwlp_end;
```

```
13          bit [31:0] hwlp_cnt;
14    function new (string name = "");
15      super.new(name);
16    endfunction
17
18  endclass: packet_out
```

3.6 Environment

In the UVM framework, the environment is the container class, in general, it contains one or more agents, and other components such as the monitor, the scoreboard, and the subscriber. The processor environment is defined in `processor_env.sv` and contains the instantiation of:

- Agent;
- Agent out;
- Scoreboard;
- Subscriber.

The environment operations are performed only during the build and connect phases. In fact, after the instantiation of the UVC's during the build phase the creator function is called to create each of them. During the connect phase, the analysis ports of Driver and Monitors are connected to the implementation of the analysis port in the Scoreboard. This is a very important step as analysis ports are the way transactions move throughout the UVM framework. A scheme of the connection is shown in Fig. 3.4.

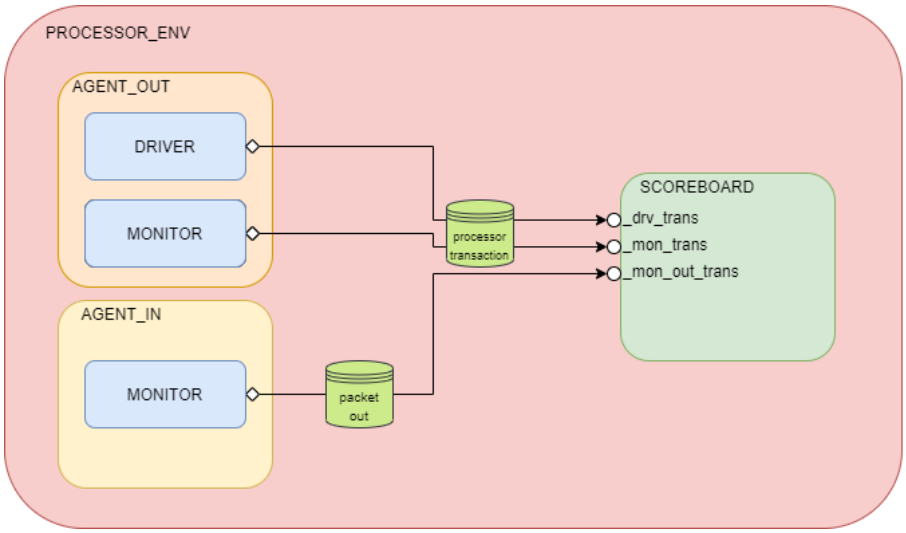


Figure 3.4: Processor Environment block diagram

Note that the dots on the scoreboard represent the implementation of the analysis ports while the diamonds represent the analysis port.

3.7 Agents

As explained in the previous section this UVM framework requires two separated agents both derived extending uvm_agent base class. The first one is the active agent and it is used to drive and capture the transaction, while the second one is a passive agent and is used only to monitor signals inside the DUV. An active agent typically contains a driver, a sequencer, and a monitor while a passive agent consists of only the monitor.

3.7.1 Agent in

Agent in is the active agent of the framework and it is defined in `processor_agent.sv`. Agent in is responsible for the creation of the required UVC's during the build phase and for the connection of the sequencer export port with the driver port during the connect phase. In addition to that during the build phase two text files are opened and their file descriptor is returned. The UVC's are created by calling the constructor function as shown in the following piece of code.

```
1 driver = processor_driver::type_id::create("driver", this);
2 mon = processor_monitor::type_id::create("mon",this);
3 sequencer = uvm_sequencer#(processor_transaction)::type_id::create("sequencer", this);
```

The two files opened here are:

- `instruction.txt`: it is the file in which the generator writes the random instructions to be sent to the processor. Opening it here make it available in sequencer which is going to read its content line by line;
- `illegal_dump.txt`: it is used inside the scoreboard. Each time that an illegal instructions has been recognized then it is written to that file. It will be useful to check if the generator is producing proper test vectors or not.

3.7.2 Agent out

Agent out is the passive agent and as a result, it contains only the declaration of the `monitor_out` component and its creation during the build phase.

3.8 Driver

The driver, derived from `uvm_driver` base class is responsible for sending the input vectors received from the sequencer to the DUV. This operation must be done accordingly to the protocol specified in the RI5CY User Manual. The signals required are:

- `instr_rdata_i`: Data read from instruction memory (i.e. the instruction to be sent);
- `instr_rvalid_i`: This signal will be high for exactly one cycle per request. Is used to signal that instr_rdata holds valid data;
- `instr_gnt_i`: The other side accepted the request.

During the build phase, the Drv2Sb (Driver to Scoreboard) port is created. Most of the driver operations are done during the `run_phase` task. In fact, if during a positive edge of the clock the request signal is high, then the driver gets a new transaction object from the sequencer using `get_next_item()` method and associate the signals contained in the transaction to the DUV signals. Once the signal has been driven to the DUV the transaction is sent to the scoreboard through the Drv2Sb port using the `write()` method.

3.9 Monitors

The monitor is used to extract signals information from the internal bus of the DUV and translate them into transactions. After that transaction has been captured it is sent to the scoreboard. Both the monitor works in the same way, the only difference is related to the type of transaction object and the signals contained in it.

3.9.1 Monitor in

In the input monitor during the run phase, a transaction object `pros_trans` is created and at the negative edge of clock, signals from the processor virtual interface are redirected to the transaction object. Then the processor transaction object is sent to the Scoreboard through the Mon2Sb port. Input Monitor has crucial importance as the scoreboard need to know if the transaction sent to the DUV has been received, and some internal signals are required to evaluate the expected results starting from the current situation (e.g. the current value of the register file).

3.9.2 Monitor out

Monitor out, on the other hand, is used to collect signals information after that operation has been completed. Using the same approach signals contained in the virtual out interface are redirected to a `packet_out` transaction object that is then sent to the Scoreboard through the `Mon2Sb_port_out`. The signals collected here represent the actual results of the instruction sent by the driver.

3.10 Scoreboard

The Scoreboard is the most complex component in the UVM framework. It is derived by extension from the `uvm_scoreboard` base class. It has the important role of verifying that everything has worked as expected by looking at input and output transaction. Up to now, we have seen that transaction objects have been moved throughout the UVM framework using analysis port. Most of them are directed to the scoreboard where their implementation is defined. Analysis port works like a callback, so each time that a UVC's write a transaction to the analysis port, the correspondent callback function is executed. It is a non-blocking mechanism that avoids time-delay in the verification framework, but it has a drawback as we need to synchronize the received transaction in order to perform meaningful comparisons. For this reason, the callback function of these analysis ports is used to put transactions into `uvm_tlm_fifo`.
The scoreboard is responsible for decoding the instruction and perform the associated operation in order to compute the expected result, compare the expected result to the actual result coming from the output transaction and provide a PASS or FAIL signal. Most of the thesis work was done on this component, in fact as the scoreboard embeds a sort of decoder and the reference model, it was very difficult to build this component.
During the run phase, the transaction objects are consumed from the FIFO using the `get()` method. Once the packets are synchronized two void functions are executed:

- `function void print_reg(packet_out pack_out);`

- `function void decode_check(processor_transaction out_trans, processor_transa exp_trans,packet_out pack_out).`

The first one is used to print out on the terminal the actual content of the register file of the DUV. The second one is the main function of the scoreboard. Before starting the decode of the instruction, a check between the driven instruction and the input one to the instruction fetch stage is performed. The result of the comparison is shown through `'uvm_info`:

- `‘uvm_info ("1_INSTRUCTION_WORD_PASS ", $sformatf("Actual Instruction=%h Expected Instruction=%h ",out_trans.instr_core,exp_trans.instrn), UVM_LOW);`

- `‘uvm_info ("1_INSTRUCTION_ERROR ", $sformatf("Actual Instruction=%h Expected Instruction=%h ",out_trans.instr_core,exp_trans.instrn), UVM_LOW).`

There are others `'uvm_info` used to signals certain conditions. As an example some results of simulation are shown in Fig. 3.5, Fig. 3.6 and Fig. 3.7.

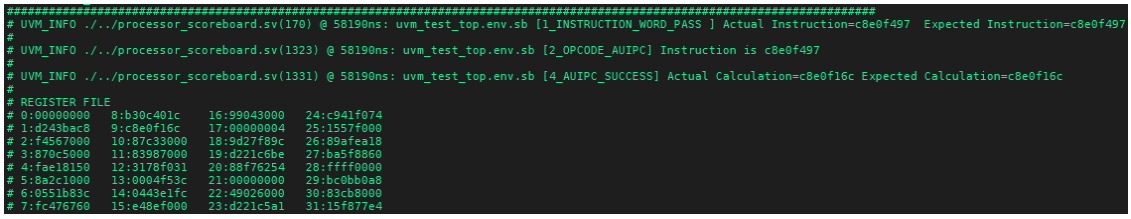


Figure 3.5: Result of AUIPC

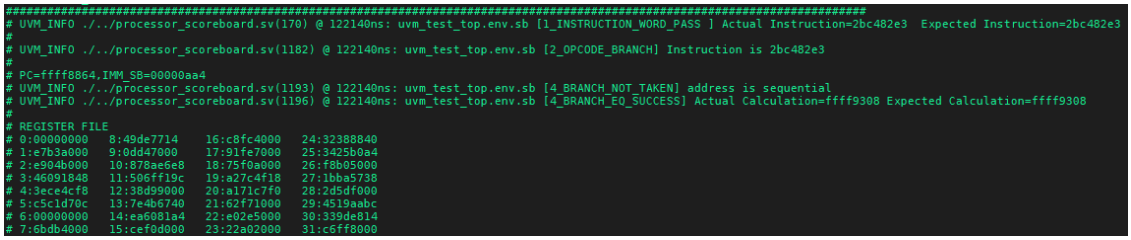


Figure 3.6: Result of Branch not taken

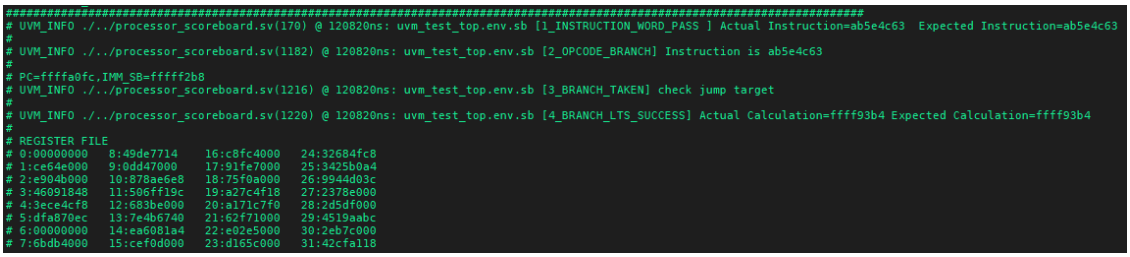


Figure 3.7: Result of branch taken

The meaning of `'uvm_info` signals are explained in the final part of this chapter. Now the decode and check function will be analyzed.

3.10.1 Decode_check

As explained before, this function receives the transaction objects and for sake of simplicity, the signals contained in transaction objects will be assigned to signals before moving to the decode and check part.

```
1       //DECLARATION FOR EXP_TRANS
2           bit [4:0] exp_rs1,exp_rs2,exp_rd;
3
4           //DECLARATION FOR OUT_TRANS
5           bit [4:0] out_rs1,out_rs2,out_rs3,out_rd;
6           bit [31:0][31:0] in_reg_file;
7           bit illegal_found;
8           bit [31:0] in_pc_value;
9           int shamt;
10          bit [31:0] out_instr;
11          bit [31:0] csr_data;
12          bit [31:0] jump_target;
13
14          //DECLARATION FOR PACK_OUT
15          bit [31:0][31:0] out_reg_file;
16          bit [31:0] out_pc_value;
17          bit [31:0] out_wdata_mem;
18          bit [31:0] out_rdata_mem;
19          bit [31:0] out_csr_wdata;
20          bit [31:0] out_csr_rdata;
21          bit [31:0] hwlp_start;
22          bit [31:0] hwlp_end;
23          bit [31:0] hwlp_cnt;
24
25          //GENERIC VARIABLES
26          bit[31:0] expected_res;
27          bit[31:0] tmp32_0,tmp32_1,tmp32_2;
```

```
28          bit[63:0] tmp64_0,tmp64_1,tmp64_2;
29          bit[15:0] tmp16_0,tmp16_1,tmp16_2;
30          bit[7:0] tmp8_0,tmp8_1,tmp8_2,tmp8_3;
31          bit[31:0] imm_i_type,imm_iz_type,imm_s_type,imm_sb_type,
32          imm_u_type,imm_uj_type,imm_z_type,imm_s2_type,imm_bi_type,
33          imm_s3_type,imm_vs_type,imm_vu_type,imm_shuffleb_type,
34          imm_shuffleh_type,imm_clip_type;
35          bit[31:0] bitmask_first ;
36      bit[31:0] bitmask_inverse ;
37      bit[31:0] bitmask ;
38          int i, j;
39          bit[31:0] count, countones, countzeros;
40          int rotamt;
41          bit branch_taken;
42          int i0,i1,i2,i3,i4,i5,i6,i7;
```

In the previous piece of code, signals are declared while in the next piece useful assignments are done to redirect transaction signals to the internal signals.

```
1           //ASSIGNMENTS EXPECTED VARIABLES
2           exp_rs1=exp_trans.instrn[20:16];
3           exp_rs2=exp_trans.instrn[24:20];
4           exp_rd=exp_trans.instrn[11:7];
5           //ASSIGNMENTS PACK_OUT VARIABLES
6           out_reg_file=pack_out.reg_file;
7           out_pc_value=pack_out.pc_value;
8           out_wdata_mem=pack_out.wdata_mem;
9           out_rdata_mem=pack_out.rdata_mem;
10          out_csr_rdata=pack_out.csr_rdata;
11          out_csr_wdata=pack_out.csr_wdata;
12          hwlp_start=pack_out.hwlp_start;
13          hwlp_end=pack_out.hwlp_end;
14          hwlp_cnt=pack_out.hwlp_cnt;
15          //ASSIGNMENTS OUT_TRANS VARIABLES
16          out_rs1=out_trans.rs1_address;
17          out_rs2=out_trans.rs2_address;
18          out_rs3=out_trans.rs3_address;
19          out_rd=out_trans.rd_address;
20          in_reg_file=out_trans.reg_file;
21          illegal_found=out_trans.illegal_insn;
22          in_pc_value=out_trans.pc_value;
23          shamt=out_rs2;
24          out_instr=out_trans.instr_core;
25          csr_data=out_trans.csr_rdata;
26          jump_target=out_trans.jump_target;
27
28          //ASSIGNMENTS IMMEDIATE VARIABLES
```

```
29          imm_i_type={{20{out_instr[31]}},out_instr[31:20]};
30          imm_iz_type={20'b0,out_instr[31:20]};
31          imm_s_type={{20{out_instr[31]}},out_instr[31:25],out_instr[11:7]};
32          imm_sb_type={{19{out_instr[31]}},out_instr[31],out_instr[7],out_instr[30:25],out_instr[11:8],1'b0};
33          imm_u_type={out_instr[31:12],12'b0};
34          imm_uj_type={ {12 {out_instr[31]}}, out_instr[19:12], out_instr[20], out_instr[30:21], 1'b0 };
35          imm_s2_type={ 27'b0, out_instr[24:20] };
36          imm_bi_type={ {27{out_instr[24]}}, out_instr[24:20] };
37          imm_s3_type={ 27'b0, out_instr[29:25] };
38          imm_vs_type={ {26 {out_instr[24]}}, out_instr[24:20], out_instr[25] };
39          imm_vu_type= { 26'b0, out_instr[24:20], out_instr[25] };
40          imm_shuffleb_type={6'b0, out_instr[28:27], 6'b0, out_instr[24:23],
41          6'b0, out_instr[22:21], 6'b0, out_instr[20], out_instr[25]};
42          imm_shuffleh_type={15'h0, out_instr[20], 15'h0, out_instr[25]};
43          imm_clip_type=(32'h1 << out_instr[24:20]) - 1;
44          //BIT MANIP OPERANDS
45          bitmask_first = 32'hFFFFFFFE << out_trans.b_mask_a ;
46      bitmask = ~(bitmask_first) << out_trans.b_mask_b ;
47      bitmask_inverse = ~(bitmask);
```

To avoid redundancy, the meaning of signals have not been explained before. Now that UVM Framework is going to use them a brief description is required.

- `exp_rs1` : It is the expected address on 5-bit of source register 1 coming from driver transaction;
- `exp_rs2` : It is the expected address on 5-bit of source register 2 coming from driver transaction;
- `exp_rd` : It is the expected address on 5-bit of destination register coming from driver transaction;
- `out_reg_file` : It is the [32bit]x[32bit] register file coming from monitor out and used to check the actual result of the computation;
- `out_pc_value` : This signal hold the program counter value coming from monitor out, useful to check if Control Transfer instruction has been executed correctly;
- `out_wdata_mem` : This signal contains the data to be written in data memory sampled by monitor out;
- `out_rdata_mem` : This signal contains the data to be read from data memory sampled by monitor out;
- `out_csr_rdata` : This signal represents the data that has been read from Control and Status Register file coming from monitor out;

- `out_csr_wdata` : This signal represents the data that has been written in Control and Status Register file coming from monitor out;
- `hwlp_start` : When a hardware loop is set, start address value is going to be written in CSR, this signal holds the value that has been written;
- `hwlp_end` : When a hardware loop is set, end address value is going to be written in CSR, this signal holds the value that has been written;
- `hwlp_cnt` : When a hardware loop is set, the counter value is going to be written in CSR, this signal holds the value that has been written;
- `out_rs1` : It is the actual address on 5-bit of source register 1 coming from input monitor;
- `out_rs2` : It is the actual address on 5-bit of source register 2 coming from input monitor;
- `out_rs3` : For three operands operation it is the actual address on 5-bit of source register 3 coming from input monitor;
- `out_rd` : It is the actual address on 5-bit of destination register coming from input monitor;
- `in_reg_file` : This signal on [32bit]x[32bit] holds the register file before that instruction has been executed, It is used to get operands and evaluate the expected result;
- `illegal_found` : This signal raise when an illegal instruction is encountered. When it happens the scoreboard signal it using an `'uvm_info` and the illegal instruction is written in "illegal_dump.txt";
- `in_pc_value` : This signals hold the program counter value sampled by input monitor, is used in some instructions to evaluate jump target (JAL, JALR) and in arithmetic instruction (AUIPC);
- `shamt` : It is an integer value specified in SRL and SRA instructions to specify the shift amount, it can be represented on 5 bit as the maximum shift amount is 32;
- `out_instr` : It is the actual instruction sampled by the input monitor that is going to be compared to the one sent by the driver;
- `csr_data` : It is the value contained in CSR before that atomic RW operation has been executed;

- `jump_target`: This signal is the Jump target sampled by input monitor, in reality, is never used but during the verification process was used to check some conditions;
- `imm_i_type` : Immediate sign extended;
- `imm_iz_type` : Immediate zero extended;
- `imm_s_type` : S-type immediate;
- `imm_u_type` : U-type immediate;
- `imm_uj_type` : UJ-type immediate;
- `imm_s2_type` : S2-type immediate;
- `imm_bi_type` : BI-type immediate;
- `imm_s3_type` : S3-type immediate;
- `imm_vs_type` : VS-type immediate (Sign-extension for vectorial type);
- `imm_vu_type` : VU-type immediate (Zero-extension for vectorial type);
- `imm_shuffleb_type` : SHUFFLEB-type immediate (SHUFFLE type for byte vectorial);
- `imm_shuffleh_type` : SHUFFLEH-type immediate (SHUFFLE type for half-word vectorial);
- `imm_clip_type` : CLIP-type immediate;
- `bitmask_first` : Signal used for bit-manipulation instructions ;
- `bitmask` : Signal used for bit-manipulation instructions, it is evaluated by left shifting bitmask of an amount coming from input monitor;
- `bitmask_inverse` : Signal used for bit-manipulation instructions, it is simply the bitwise negation of bitmask.

The decoding phase starts with a check on the expected instruction (coming from driver transaction) and actual instruction (coming from the input monitor). If this check was successful then we can move on to identify the instructions using its opcode(i.e. `instr[6:0]`) and its function fields (i.e. `instr[14:12]` function3 and `instr[31:25]` function7) according to the type of instructions. The scoreboard `decode_check` function is shown in Fig. 3.8, all the case and if are collapsed because otherwise, it is not readable.

```
//START DECODING
if(exp_trans.instrn == out_trans.instr_core)begin
$display("##########################################################################################################################################");
`uvm_info ("1_INSTRUCTION_WORD_PASS ", $sformatf("Actual Instruction=%h  Expected Instruction=%h \n",out_trans.instr_core,exp_trans.instrn), UVM_LOW)
//DEFINITION


case(out_trans.instr_core[6:0]) //OPCODE PART OF THE INSTRUCTION
            7'b0110011: begin   //OPCODE_OP
            7'b1110011:begin //OPCODE_SYSTEM
            7'b0001111:begin //OPCODE_FENCE
            7'b0010011:begin //OPCODE_OPIMM
            7'b0100011:begin //OPCODE_STORE
            7'b0000011:begin //OPCODE_LOAD
            7'b1100011:begin //OPCODE_BRANCH
            7'b1100111:begin //OPCODE_JALR
            7'b1101111:begin //OPCODE_JAL
            7'b0010111:begin //OPCODE_AUIPC
            7'b0110111:begin //OPCODE_LUI
            /* UNCOMMENT AND WORK ON IT IF YOU INCLUDE Floating Point Unit
            7'b0001011:begin //OPCODE_LOAD_POST
            7'b0101011:begin //OPCODE_STORE_POST
            7'b1011011:begin //OPCODE_PULP_OP
            7'b1010111:begin //OPCODE_VECOP
            7'b1111011:begin //OPCODE_HWLOOP
            default:begin
endcase


end
else begin
        `uvm_error("1_INSTRUCTION_ERROR", $sformatf("Actual=%h Expected=%h \n", out_trans.instr_core,exp_trans.instrn))
end
```

Figure 3.8: decode and check function collapsed

Finally, inside each of the opcode cases, the instruction is recognized and the expected value is computed. Then it is compared to the content of register rd coming from output monitor. As an example is reported in the following piece of code the complete decode and check for OPIMM Instructions.

```
7'b0010011:begin //OPCODE_OPIMM
`uvm_info ("2_OPCODE_OPIMM", $sformatf("Instruction is %h\n",out_trans.instr_core),
UVM_LOW)
 if(illegal_found==1) begin
  `uvm_error ("3_ILLEGAL_INSN", $sformatf("RAISED ILLEGAL SIGNAL IN CONTROLLER"))
  $fdisplay(processor_agent.dump_illegal,"%h",out_trans.instr_core);
end
else begin
case(out_trans.instr_core[14:12])
3'b000: begin
expected_res=imm_i_type+in_reg_file[out_rs1][31:0];
if(expected_res==out_reg_file[out_rd][31:0]) begin
`uvm_info ("4_ADDI_SUCCESS", $sformatf("Actual Calculation=%h
Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
end
else begin
`uvm_info ("4_ADDI_FAILED", $sformatf("Actual Calculation=%h
Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
end
end
3'b010: begin
expected_res = $signed(in_reg_file[out_rs1][31:0]) < $signed(imm_i_type) ? 32'h00000001 : '0;
if(expected_res==out_reg_file[out_rd][31:0]) begin
`uvm_info ("4_SLTS_SUCCESS", $sformatf("Actual Calculation=%h
```

```
Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
end
else begin
`uvm_info ("4_SLTS_FAILED", $sformatf("Actual Calculation=%h
Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
end
end
3'b011: begin
//comparison evaluation
expected_res = $unsigned(in_reg_file[out_rs1][31:0]) < $unsigned(imm_i_type) ? 32'h00000001 : '0;
if(expected_res==out_reg_file[out_rd][31:0]) begin
`uvm_info ("4_SLTU_SUCCESS", $sformatf("Actual Calculation=%h
Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
end
else begin
`uvm_info ("4_SLTU_FAILED", $sformatf("Actual Calculation=%h
Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
end
end
3'b100: begin
expected_res=imm_i_type^in_reg_file[out_rs1][31:0];
if(expected_res==out_reg_file[out_rd][31:0]) begin
`uvm_info ("4_XORI_SUCCESS", $sformatf("Actual Calculation=%h
Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
end
else begin
`uvm_info ("4_XORI_FAILED", $sformatf("Actual Calculation=%h
Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
end
end
3'b110: begin
expected_res=imm_i_type|in_reg_file[out_rs1][31:0];
if(expected_res==out_reg_file[out_rd][31:0]) begin
`uvm_info ("4_ORI_SUCCESS", $sformatf("Actual Calculation=%h
Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
end
else begin
`uvm_info ("4_ORI_FAILED", $sformatf("Actual Calculation=%h
Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
end
end
3'b111: begin
expected_res=imm_i_type&in_reg_file[out_rs1][31:0];
if(expected_res==out_reg_file[out_rd][31:0]) begin
`uvm_info ("4_ANDI_SUCCESS", $sformatf("Actual Calculation=%h
Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
```

```
71  end
72  else begin
73  `uvm_info ("4_ANDI_FAILED", $sformatf("Actual Calculation=%h
74  Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
75  end
76  end
77  3'b001: begin
78  expected_res=in_reg_file[out_rs1][31:0]<<shamt;
79  if(expected_res==out_reg_file[out_rd][31:0]) begin
80  `uvm_info ("4_SLLI_SUCCESS", $sformatf("Actual Calculation=%h
81  Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
82  end
83  else begin
84  `uvm_info ("4_SLLI_FAILED", $sformatf("Actual Calculation=%h
85  Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
86  end
87  end
88  3'b101: begin
89  if(out_trans.instr_core[31:25]==7'b0) begin
90  expected_res=$signed(in_reg_file[out_rs1][31:0]) >> $signed(shamt);
91  if(expected_res==out_reg_file[out_rd][31:0]) begin
92  `uvm_info ("4_SRLI_SUCCESS", $sformatf("Actual Calculation=%h
93  Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
94  end
95  else begin
96  `uvm_info ("4_SRLI_FAILED", $sformatf("Actual Calculation=%h
97  Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
98  end
99  end
100 else if(out_trans.instr_core[31:25]==7'b0100000) begin
101 expected_res=$signed(in_reg_file[out_rs1][31:0]) >>> $signed(shamt);
102 if(expected_res==out_reg_file[out_rd][31:0]) begin
103 `uvm_info ("4_SRAI_SUCCESS", $sformatf("Actual Calculation=%h
104 Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
105 end
106 else begin
107 `uvm_info ("4_SRAI_FAILED", $sformatf("Actual Calculation=%h
108 Expected Calculation=%h\n",out_reg_file[out_rd][31:0],expected_res), UVM_LOW)
109 end
110 end
111 end
112 endcase
113 end
114 end
```

3.10.2 Summary of simulation

In the summary of the simulation, the count of each `uvm_info` signal is reported. To simplify the extraction of the useful information from the summary a number was inserted at the begin of the string.

- `1_INSTRUCTION_WORD_PASS` or `1_INSTRUCTION_ERROR`;
- `1_BRANCH_TAKEN` or `1_BRANCH_NOT_TAKEN`;
- `2_ILLEGAL_OPCODE` or `2_OPCODE_XXX`;
- `3_ILLEGAL_INSN`;
- `4_OP_SUCCESS` or `4_OP_FAILED`.

Exploiting this approach the `plot_report()` function included in `functions.py` can extract the needed information from the simulation log and display a plot bar allowing a faster check of the performances. An example of plot is shown in Fig. 3.9.

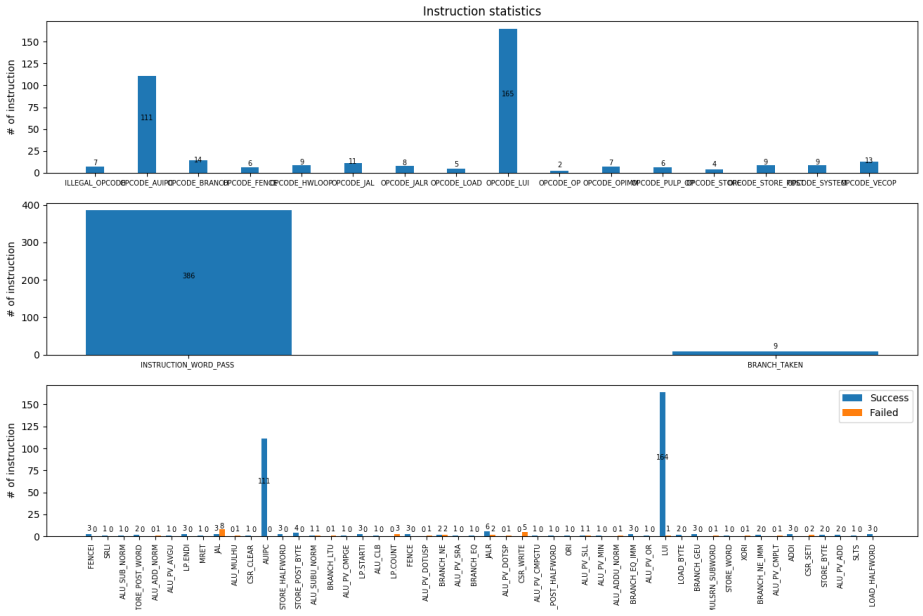


Figure 3.9: Plot of simulation summary

Chapter 4

Simulation Environment

In this chapter will be analyzed the python environment developed the support of the UVM framework. To speed up the verification procedure it was decided to develop a random instructions generator mainly based on a database of valid instructions extracted from the RTL hardware description. Finally, a graphical user interface based on Tkinter has been developed, in this way user can specify the number and type of instructions to be tested, compile and run a simulation and collect the results. In the first section will be explained how the ISA Database has been extracted and used, in the second the Random generator features are shown while in the last section the GUI functionality will be explained.

4.1 ISA Database

A large part of this work was dedicated to the extraction of the available instruction set. It was not so easy as for a certain kind of instructions there was a lack of documentation. The first approach was to identify all the valid opcodes accepted by the device under verification, which was done by simply looking at the `riscv_defines.sv`. Using an extractor script (`Extractor.py`)and a database creator (`db_opcode_32_creator.py`) the following database is obtained.

```
1    name_byopcode={
2 "1110011" :"OPCODE_SYSTEM",
3 "0001111" :"OPCODE_FENCE",
4 "0110011" :"OPCODE_OP",
5 "0010011" :"OPCODE_OPIMM",
6 "0100011" :"OPCODE_STORE",
7 "0000011" :"OPCODE_LOAD",
8 "1100011" :"OPCODE_BRANCH",
9 "1100111" :"OPCODE_JALR",
10 "1101111" :"OPCODE_JAL",
```

```
"0010111" :"OPCODE_AUIPC",
"0110111" :"OPCODE_LUI",
"1010011" :"OPCODE_OP_FP",
"1000011" :"OPCODE_OP_FMADD",
"1001111" :"OPCODE_OP_FNMADD ",
"1000111" :"OPCODE_OP_FMSUB",
"1001011" :"OPCODE_OP_FNMSUB",
"0100111" :"OPCODE_STORE_FP",
"0000111" :"OPCODE_LOAD_FP ",
"0001011" :"OPCODE_LOAD_POST",
"0101011" :"OPCODE_STORE_POST",
"1011011" :"OPCODE_PULP_OP",
"1010111" :"OPCODE_VECOP",
"1111011" :"OPCODE_HWLOOP",
}
opcode_byname={
"OPCODE_SYSTEM" :"1110011",
"OPCODE_FENCE" :"0001111",
"OPCODE_OP" :"0110011",
"OPCODE_OPIMM" :"0010011",
"OPCODE_STORE" :"0100011",
"OPCODE_LOAD" :"0000011",
"OPCODE_BRANCH" :"1100011",
"OPCODE_JALR" :"1100111",
"OPCODE_JAL" :"1101111",
"OPCODE_AUIPC" :"0010111",
"OPCODE_LUI" :"0110111",
"OPCODE_OP_FP" :"1010011",
"OPCODE_OP_FMADD" :"1000011",
"OPCODE_OP_FNMADD " :"1001111",
"OPCODE_OP_FMSUB" :"1000111",
"OPCODE_OP_FNMSUB" :"1001011",
"OPCODE_STORE_FP" :"0100111",
"OPCODE_LOAD_FP " :"0000111",
"OPCODE_LOAD_POST" :"0001011",
"OPCODE_STORE_POST" :"0101011",
"OPCODE_PULP_OP" :"1011011",
"OPCODE_VECOP" :"1010111",
"OPCODE_HWLOOP" :"1111011",
}
```

The second step was the larger time-consuming operation as it was necessary to

find all the possible instructions. A reverse-engineering operation was fundamental to extract from the `riscv_decoder.sv` the complete instruction set. The encoding of the instructions was reported in a `.csv` for sake of simplicity. As the random generator has been developed in python the best way to having a database without duplication of elements is using a set of dictionaries. In fact, python dictionaries are used to store data values in key: value pairs and does not allow duplicates.
So starting from two CSV databases:

- `ISA.csv`;
- `ISA_C.csv`.

Those CSV files have been converted in a database using respectively `db_isa_32_creator.py` and `db_isa_16_creator.py`. The resulting database is a simple structure in which instructions are divided by Encoding types (R,RI,I,S,SB,U,UJ,HL,PRIV,BM,IB,R4,V) and it is possible to search for instructions using different criterion (i.e. keys). For each type of instructions the instruction database is generated using a snippet of code like the following one:

```
f_write= open("database/db_ISA_32.py","a+")
f_write.write("#####################################\n")
f_write.write("#####################################\n")
f_write.write("    #R_type_DB\n")
f_write.write("#####################################\n")
f_write.write("#####################################\n")
f_write.write("R_TYPE_BYINSTR_NAME={\n")
for c in  range(0, len(r_instructions)):
  f_write.write("\"{}\" :\"{}\",\n". format(r_instructions[c],r_names[c]))
f_write.write("}\n")
#search by name and returns instruction complete
f_write.write("R_TYPE_BYNAMES_INSTR={\n")
for c in  range(0, len(r_instructions)):
     f_write.write("\"{}\" :\"{}\",\n". format(r_names[c],r_instructions[c]))
f_write.write("}\n")
#search by name and returns OPCODE
f_write.write("R_TYPE_BYNAMES_OPCODE={\n")
for c in  range(0, len(r_instructions)):
     f_write.write("\"{}\" :\"{}\",\n". format(r_names[c],r_opcodes[c]))
f_write.write("}\n")
 #search by name and returns instruction complete
f_write.write("R_TYPE_BYNAMES_FUNCT3={\n")
for c in  range(0, len(r_instructions)):
     f_write.write("\"{}\" :\"{}\",\n". format(r_names[c],r_funct3s[c]))
f_write.write("}\n")
```

```
26  #search by name and returns FUNCT7
27  f_write.write("R_TYPE_BYNAMES_FUNCT7={\n")
28  for c in  range(0, len(r_instructions)):
29        f_write.write("\"{}\" :\"{}\",\n". format(r_names[c],r_funct7s[c]))
30  f_write.write("}\n")
31   #search by FUNCT3 and returns instruction name
32  f_write.write("R_TYPE_BYFUNCT3_NAMES={\n")
33  for c in  range(0, len(r_instructions)):
34        f_write.write("\"{}\" :\"{}\",\n". format(r_funct3s[c],r_names[c]))
35  f_write.write("}\n")
36  #search by FUNCT7 and returns instruction name
37  f_write.write("R_TYPE_BYFUNCT7_NAMES={\n")
38  for c in  range(0, len(r_instructions)):
39        f_write.write("\"{}\" :\"{}\",\n". format(r_funct7s[c],r_names[c]))
40  f_write.write("}\n")
41  #search by OPCODE and returns instruction name
42  f_write.write("R_TYPE_BYOPCODE_NAMES={\n")
43  for c in  range(0, len(r_instructions)):
44        f_write.write("\"{}\" :\"{}\",\n". format(r_opcodes[c],r_names[c]))
45  f_write.write("}\n")
```

Some of the keys are never used, in general, the approach used in the random generator is to get all the instruction names using their opcodes, and after that get the full instructions exploiting the previously obtained name.

4.2 RV Generator

The istruction set database was the starting point for the random generator. Once the database has been created the next step is to randomize among opcodes and then among of instructions of that database.
In `rvgen2.py` database dictionaries are imported in order to be available in the current program and then are cast to list. Using lists instead of dictionaries is better as in this way we are able to use `random.choice()` method to get a random element from a list.
The main function in `rvgen2.py` is `rv_generator`, it accepts as arguments 3 parameters:

- `num`: the number of random instructions to be generated;
- `sel`: it is a string containing ones and zeros and it is used to select the subset of opcodes to be included in randomizations;
- `log`: it is a 1 or 0 and gives the possibility of avoiding printing on stdout the log on instruction randomizations.

`num` does not represent the actual number of instructions generated, in fact, to get the real number we should add prologue instructions, and consider that Control Transfer instructions lead to additional instructions. `sel` is really important in a CDV (Coverage Driven Verification) environment as allow us to select only opcodes that were not fully covered in previous simulations. Its value is set in the GUI and follow table 4.1

OP	FENCE	HWLOOP	SYSTEM	VECOP	STOREFP	PULP_OP
sel[0]	sel[1]	sel[2]	sel[3]	sel[4]	sel[5]	sel[6]

LOADFP	JAL	JALR	BRANCH	STORE	STOREPOST	LOAD
sel[7]	sel[8]	sel[9]	sel[10]	sel[11]	sel[12]	sel[13]

LOADPOST	LUI	AUIPC	OPIMM	OP_FP	FMADD	FNMADD
sel[14]	sel[15]	sel[16]	sel[17]	sel[18]	sel[19]	sel[20]

FMSUB	FNMSUB	CORNERCASES				
sel[21]	sel[22]	sel[23]				

Table 4.1: sel string encoding

According to the `sel` string provided as input, the `OPCODE_LIST` is created in a for-loop by appending `OPCODE_DB` elements to it using `sel[i]` as a masking condition. The second step to initialize the environment is to create an instruction list for each of the opcodes. The complete instruction list is obtained by combining together the list of instructions coming from the database. Then by looking at their 7 LSB's (i.e. OPCODE) instructions are appended to the correct list. Finally what we get are the following lists containing number of elements specified in parenthesis:

- OPCODE_OP (52);
- OPCODE_FENCE (2);
- OPCODE_HWLOOP (6);
- OPCODE_SYSTEM (12);
- OPCODE_VECOP (200);
- OPCODE_STORE_FP (4);
- OPCODE_PULP_OP (20);
- OPCODE_LOAD_FP (4);
- OPCODE_JAL (1);
- OPCODE_JALR (1);
- OPCODE_BRANCH (8);

- OPCODE_STORE (3);
- OPCODE_STORE_POST (6);
- OPCODE_LOAD (5);
- OPCODE_LOAD_POST (10);
- OPCODE_LUI (1);
- OPCODE_AUIPC (1);
- OPCODE_OPIMM (9);
- OPCODE_OP_FP (89);
- OPCODE_FMADD;
- OPCODE_FNMADD;
- OPCODE_FMSUB;
- OPCODE_FNMSUB.

The random generator is also capable of providing wrong instructions, illegal instructions in order to raise exception signals and cover also corner cases situations. As explained the random generator has two levels of randomization, in the first step a random opcode is chosen among the ones included in `OPCODE_LIST`, then a random instruction is chosen from the instruction list of that opcode. The random instruction got from the randomization process is a prototype containing "x" in fields that have to be filled. So depending on the opcode different operations are performed to get a complete instruction where all the fields have been properly filled.
In the following snippet of code, the randomization of `OPCODE_OP` is shown:

```
typ=random.choice(OPCODE_LIST)
      x+=1
      if(typ=="OPCODE_OP"):
          inst = random.choice(OPCODE_OP)
          instr,name =inst
          if(instr[0:2]=="11"): #IMMEDIATE BIT MANIPULATION
            LS3=random_reg()
            LS2=random_reg()
            SRC=random_reg()
            RD=random_regd()
            instr_fill="11"+LS3+LS2+SRC+instr[17:20]+RD+instr[25:]
```

```
12            elif(instr[0:2]=="10"): #REGISTER BIT MANIPULATION
13              RS1=random_reg()
14              RS2=random_reg()
15              RD=random_regd()
16              instr_fill="1000000"+RS2+RS1+instr[17:20]+RD+instr[25:]
17            else: #OTHER INSTRUCTIONS
18              RS1=random_reg()
19              RS2=random_reg()
20              RD=random_regd()
21              instr_fill=instr[0:7]+RS2+RS1+instr[17:20]+RD+instr[25:]
22            if(log==1):
23              print(colored("{}:". format(x),"red"),colored(instr_fill,"green"),
24              colored(name,"green"))
25            f.write("{}\n". format(instr_fill))
```

To generate random immediate,register address and rounding modes four different functions are used:

- `def random_reg()`: used to generate random rs1 and rs2, a random number in range [0,31] is generated and converted to its binary form;
- `def random_regd()`: used to generate random rd, a random number in range [1,31] is generated and converted to its binary form (rd $\neq$ 0);
- `def random_rounding_mode()`: used to generate random rounding mode, a random element is chosen from list "ROUNDING_MODE=["000","001","010","011","100"]";
- `def random_immediate(n)`: Used to generate random operands to be inserted as immediate, it accepts an integer argument to select the number of bits of the immediate.

After `instr_fill` has been obtained by concatenation of operands and instruction fixed fields, it is always written in the `instruction.txt` file and ready to be read by the UVM Sequencer. If log option is enabled the instructions are written to the stdout. An example of random generated program is shown in the Fig. 4.1

```
1: 00000000000000000000000110010111 PROLOGUE 1
2: 00000000000000001100000110010011 PROLOGUE 2
3: 00000000000000000000000100010111 PROLOGUE 3
4: 00000000000000001000000100010011 PROLOGUE 4
5: 00000000000000000000010100010111 PROLOGUE 5
6: 00000000000001010000010100010011 PROLOGUE 6
7: 00110000010101010001000001110011 PROLOGUE 7
8: 00000000000000000000001100010111 PROLOGUE 8
9: 00000000000000011000000000001100111 PROLOGUE 9
10: 00000000000000001000000001100111 PROLOGUE 10
11: 00000001100000000110101000100011 SW with offset from immediate
12: 10000001001000100100000110110011 ALU_BSET REGISTER BIT MANIP
13: 10111100111000110001010000001111 FENCE.I
14: 00000001001100000110010010100011 SW with offset from immediate
15: 00001111100101100000010010001111 FENCE
16: 00010001000110100001010110110011 PULP SPECIFIC P.FL1
17: 01000001000000000111110100000011 p.lbu rs2
18: 01111011000110010101100111110011 CSR_OP_WRITE_uimm
19: 00000000111111001000011100110011 RV32I ALU_ADD
20: 00010101110011110010110100110011 PULP SPECIFIC P.CLIPU
21: 00010001101100000111010110000011 p.lh rs2
22: 10000001001010010000011100110011 ALU_BEXT REGISTER BIT MANIP
23: 00001000011111100101100010110011 PULP SPECIFIC P.ROR
24: 00000001001100000100100100100011 SB with offset from immediate
25: 00101111001100111011110111101111 OPCODE_JAL
26: 10101110001000001000011010010111 OPCODE_AUIPC AFTER JAL
27: 10000110000010000101011110110111 OPCODE_LUI AFTER JAL
28: 10000110000010000101011110110111 OPCODE_LUI AFTER JAL
29: 10000110000010000101011110110111 OPCODE_LUI AFTER JAL
30: 10101001101110000101110100010111 OPCODE_AUIPC AFTER JAL
31: 11101011100011000000100100110111 OPCODE_LUI AFTER JAL
32: 11101011100011000000100100110111 OPCODE_LUI AFTER JAL
33: 11010010101101101010111010010111 OPCODE_AUIPC AFTER JAL
34: 11010010101101101010111010010111 OPCODE_AUIPC AFTER JAL
35: 11010010101101101010111010010111 OPCODE_AUIPC AFTER JAL
36: 00000001110100000000011000101011 SB_POST with offset from register
37: 10011101001111111100111011101111 OPCODE_JAL
38: 01011001100011000001100010110111 OPCODE_LUI AFTER JAL
39: 01011001100011000001100010110111 OPCODE_LUI AFTER JAL
40: 10010101011001101011111111110111 OPCODE_LUI AFTER JAL
41: 11111110000000110000111111110111 OPCODE_LUI AFTER JAL
42: 10100001111000100100010100010111 OPCODE_AUIPC AFTER JAL
43: 10100001111000100100010100010111 OPCODE_AUIPC AFTER JAL
44: 01110111001100010100000100110111 OPCODE_LUI AFTER JAL
45: 01110111001100010100000100110111 OPCODE_LUI AFTER JAL
46: 10111001000011010011011010110111 OPCODE_LUI AFTER JAL
47: 10111001000011010011011010110111 OPCODE_LUI AFTER JAL
48: 00000000010100000010101110101011 SW_POST with offset from register
49: 00111110010000100111001010010111 OPCODE_AUIPC
50: 01000001011001101111111101000101l p.lbu rs2 POST
51: 00000001000100000001010100010101l SW_POST with offset from register
52: 01000000000000101101000010010011 OPCODE_OPIMM alu_sra
53: 01010000010100000111010000000011 p.lhu rs2
54: 00000000111100000101110000101011 SH_POST with offset from immediate
55: 00000000100110011101001010010011 OPCODE_OPIMM alu_srl
56: 00010000010100000000000001110011 WFI
57: 11110000011000000000010101100111 OPCODE_JALR
58: 00101101001100101110100110110111 OPCODE_LUI
```

Figure 4.1: Random Program generated by RVGEN2.py

Finally, after the randomization process has been completed it is necessary to

modify the UVM Framework, in particular, the sequencer file must be updated to change the number of iteration required to send all the transactions read from `instruction.txt`. This operation is performed by the function `overwrite_sequencesv(num)`.

4.3 UVM Env Configurator

UVM Env Configurator is the GUI developed to configure the simulation constraint and run a complete simulation. It has been developed using the Tkinter Python library, and contains a set of elements that will be explained in the next subsections. GUI is shown in Fig. 4.2.

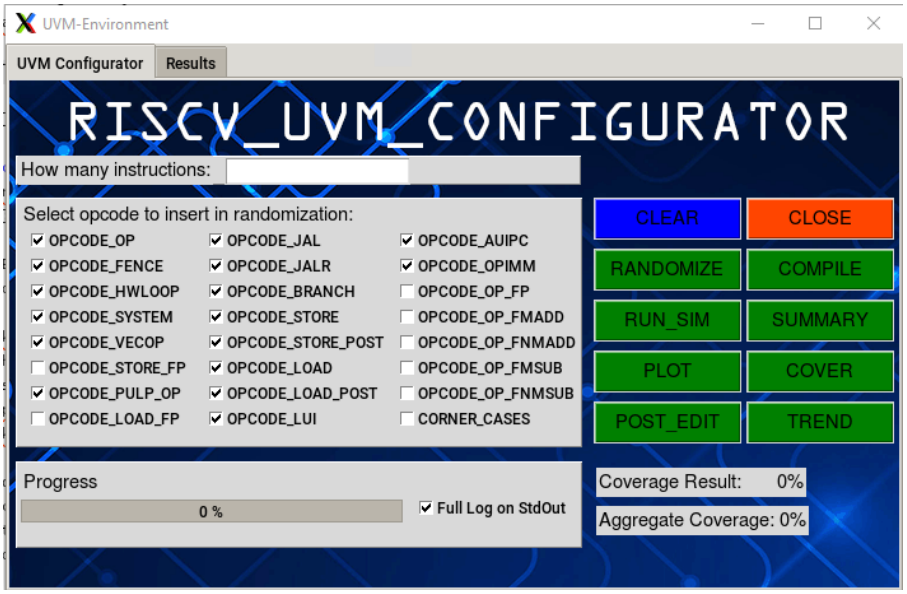


Figure 4.2: UVM Env Graphical User Interface

4.3.1 GUI Elements

In this subsection, the GUI elements are described explaining their functionalities. `tk.Entry` and `tk.CheckButton` are used to configure the simulation parameters. `tk.Buttons` are used to run simulations and interact with the UVM Framework.

tk.Notebook element

The graphical user interface is split into two tabs using `tk.Notebook`. The first frame shows the *UVM Configurator*, which will be detailed in the next paragraph, while the second frame is used to show the plots created during the simulation.

tk.Entry element

In the GUI there is a single `tk.Entry` element. It is used to specify the parameter `num` that is the number of random instruction to generate. The parameter is then passed as argument of the `rv_generator` function imported from `rvgen2.py`.

tk.CheckButton elements

There are 24 CheckButtons, each of them corresponds to each of the bit of the string `sel`. Each of the check buttons is associated with a `tk.booleanvar` whose initial value is set before initializing the interface. By changing the state of check buttons `sel` string is changed. There is an additional check button that is used to set the log parameter, if it is enabled then all the operations will print on stdout otherwise only some information will be printed.

tk.Buttons elements

There are ten `tk.buttons` used to interact with the UVM framework, in particular, each of them as a callback function that is run whenever the button is pressed:

- CLEAR: It was necessary to add this button to clean up the terminal from the result of the previous simulation. When it is pressed its callback function `clear()` is called;
- CLOSE: When it is pressed the GUI is terminated by using `root.quit()` function;
- RANDOMIZE: When it is pressed, the callback function `randomize()` is executed. All the input parameter are passed to the `rvgen2.rv_generator()` function and progress bar is updated gradually reaching 33%;
- COMPILE: After Randomization the `processor_sequence` has been modified so the UVM framework must be recompiled. When the button is pressed `compile_design()` function is executed and the progress bar is updated to 66%;
- RUN_SIM: When it is pressed `run_sim()` function is executed. At the end of the simulation if log on stdout was enabled then you should see on stdout the summary of results, otherwise, you need to press the SUMMARY button. The progress bar is updated reaching 100%;
- SUMMARY: When it is pressed a summary of the simulation is printed out on stdout;
- PLOT: This button is used to run `plot_report()` function imported from `functions.py`. When pressed a plot is created extracting information of the simulation from the summary;

- COVER: when it is pressed a coverage analysis of the last simulation is run. The result is saved in the coverage folder, a plot of the actual coverage is saved in the figure folder and the Coverage result labels are updated. Furthermore, aggregate coverage is evaluated by a combination of ".ucdb" databases of coverage;

- POST_EDIT: When this button is pressed, a post-editing operation is performed to create coverage reports divided by type and then different plots are created;

- TREND: Whenever it is pressed the result of the post-editing step is exploited to create a plot showing the percentage of coverage over the number of simulation.

4.3.2 GUI Result Frames

In this section the "Results" tab is shown and explained. Each time that a plot is created using buttons available in UVM Configurator tab, instead of showing the plot using `plt.show()` method the figure is saved in the figures folder, re-opened and shown in the results tab.

Simulation Results

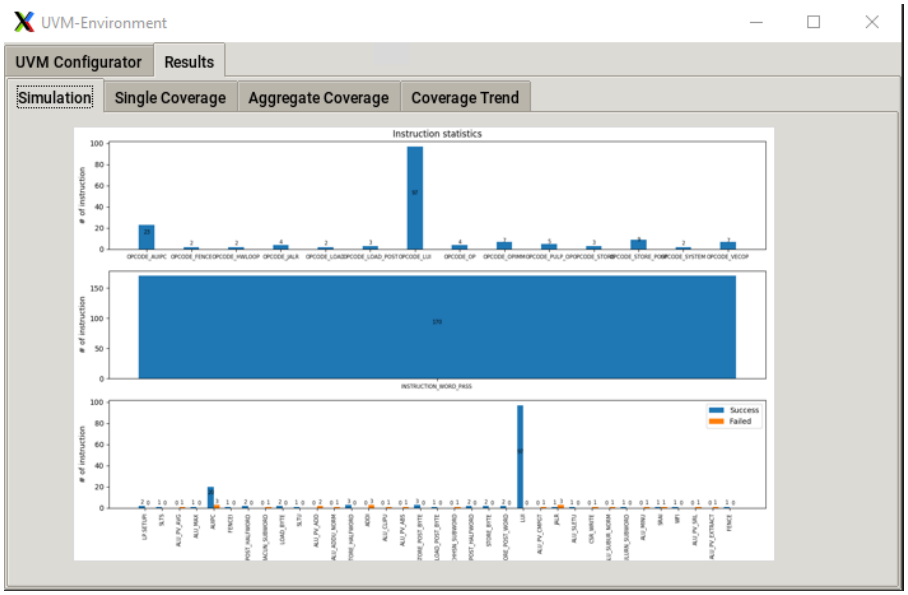


Figure 4.3: Simulation Result frame

Single Coverage Result

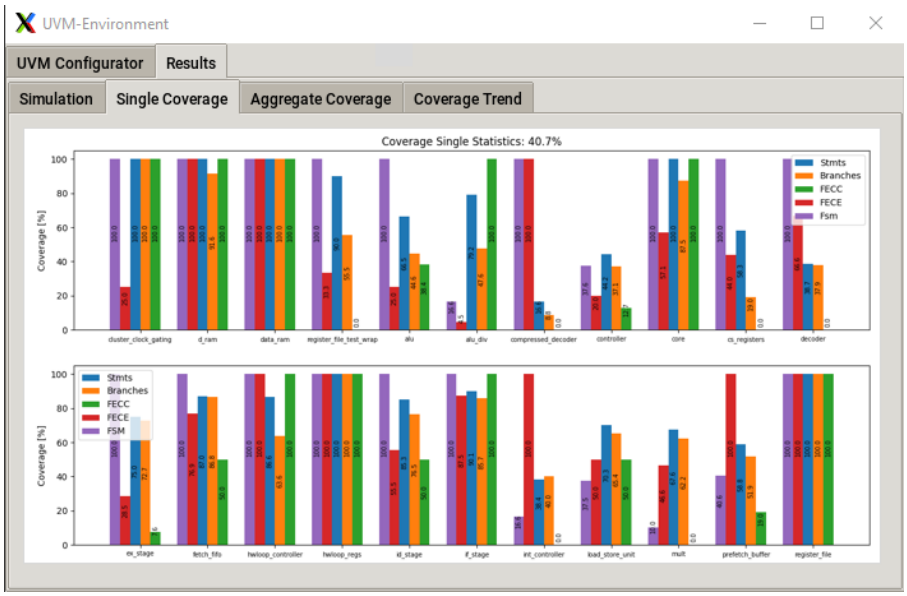


Figure 4.4: Single Coverage Result frame

Aggregate Coverage Result

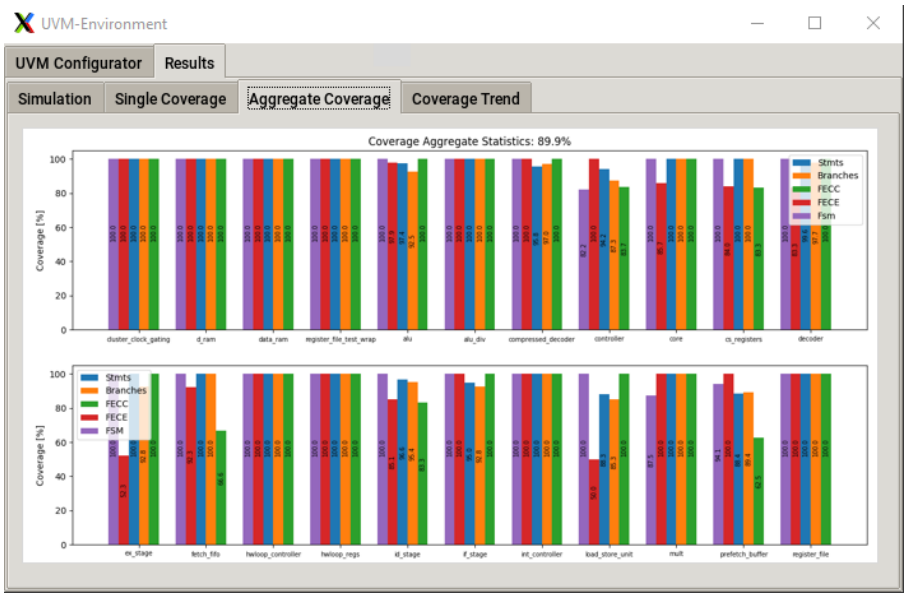


Figure 4.5: Aggregate Coverage Result frame

Coverage Trend

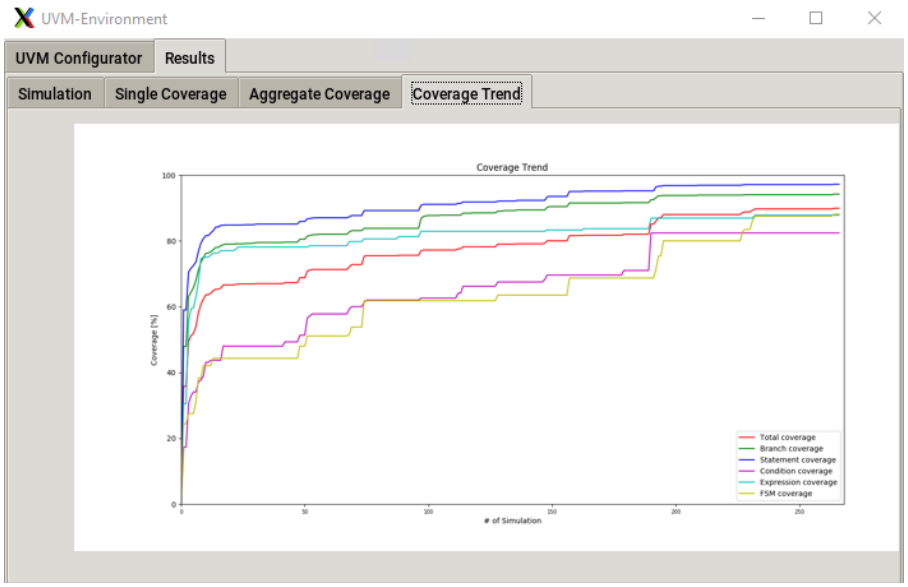


Figure 4.6: Coverage Trend frame

Chapter 5

Simulation and Results

In this chapter, the results of the verification procedure are shown and described. The first section will describe the type of coverage that has been used and the selected metrics, in the second section the results of the simulations are shown through plots to demonstrate that the DUV is working as intended (i.e. verification scope), the percentage of coverage is shown and how it has been reached is described. The last step is really important to demonstrate that the DUV has been fully exercised in most of its functionalities.

5.1 Coverage and metrics

Collecting coverage information is fundamental because improves efficiency allowing the identification of areas of the design that have not been exercised. In this project, code coverage is used to determine the level of confidence in the verification. Code coverage is a measure of the amount of code of the RTL description is executed when a simulation is run. A program with high code coverage has a lower chance of containing undetected bugs. That suggests we need to reach a high coverage level to make this verification procedure meaningful.
Enabling the analysis of code coverage is simple as it is done by inserting a command while the design is compiled and simulation is run, after the simulation, a ".ucdb" database is returned and another ModelSim command is used to extract coverage information in a user-readable form.
The available metrics are the following and will be explained in the following subsections:

- Statement Coverage;
- Branch Coverage;
- Focused Expression Coverage;

- Focused Condition Coverage;
- FSM Coverage;
- Toggle Coverage.

5.1.1 Statement Coverage

Statement coverage reports which RTL statements have been executed or not. Lines can contain multiple statements and this kind of coverage can identify more than one statement for each line of code. It is useful to identify statements that have not executed and may investigate the reasons:

- Statement could not be executed because data and control flow prevents its execution;
- Is possible to execute statement but the condition required has not been created.

$$Stmts = \frac{N\_of\_executed\_statements}{Total\_statements} * 100 \tag{5.1}$$

5.1.2 Branch Coverage

The branch coverage metric counts the amount of control flow transfer statements like *if, case, while, repeat, for, loop.* An if statement with a single condition provides 2 possible conditions, and it is necessary to cover both the condition to achieve 100%. In some cases should be checked if the expression can assume both values or not.

$$Branch = \frac{N\_of\_executed\_branches}{Total\_branches} * 100 \tag{5.2}$$

5.1.3 Focused Condition Coverage

Condition coverage checks boolean expression in conditional statements to test and evaluate the variables or sub-expressions. The goal of condition coverage is to check individual outcomes for each logical condition [12]. For instance let consider the following boolean expression:
if(x<y and a>b) there are two logical conditions, as a result the possible outcomes are

- True,True;
- True,False;
- False,True;

- False,False.

To achieve 100% coverage all the possible conditions must be covered.

$$Condition = \frac{N\_of\_executed\_operands}{Total\_number\_of\_operands} * 100 \tag{5.3}$$

5.1.4 Focused Expression Coverage

The same as condition coverage, but covers concurrent signal assignments instead of branch decisions[cit] As an example, an extract of Expression coverage is reported in Fig. 5.1.

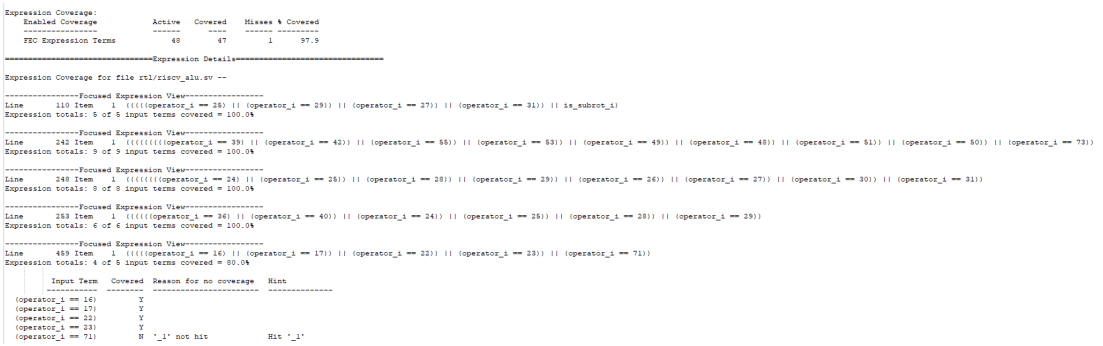


Figure 5.1: Expression coverage

5.1.5 FSM Coverage

FSM Coverage is divided into state coverage and transition coverage. In fact, even if we succeed in covering all the state of the FSM, is possible that all the transition has not covered. If we consider the FSM shown in Fig. 5.2

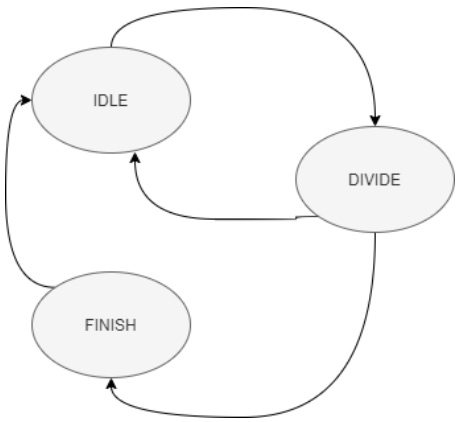


Figure 5.2: alu div FSM example

The FSM is composed of three states and four arcs. State coverage provides a table with a count of the visits for each of the state. Transition coverage will provide a table in which there are all the possible arcs to be covered. A report like the one shown in Fig. 5.3 is provided for each of the FSM recognized in the design.

```
FSM Coverage:
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    FSMs                                                        87.5
        States                       3         3         0     100.0
        Transitions                  4         3         1      75.0

================================FSM Details================================

FSM Coverage for file rtl/riscv_alu_div.sv --

FSM_ID: State_SP
    Current State Object : State_SP
    ---------------------
    State Value MapInfo :
    ---------------------
Line           State Name               Value
----           ----------               -----
 137                 IDLE                   0
 149               DIVIDE                   1
 162               FINISH                   2
    Covered States :
    ----------------
                     State           Hit_count
                     -----           ---------
                      IDLE                1077
                    DIVIDE                6515
                    FINISH                 425
    Covered Transitions :
    ---------------------
Line            Trans_ID           Hit_count          Transition
----            --------           ---------          ----------
 145                   0                 430          IDLE -> DIVIDE
 158                   1                 425          DIVIDE -> FINISH
 166                   3                 425          FINISH -> IDLE
    Uncovered Transitions :
    -----------------------
Line            Trans_ID          Transition
----            --------          ----------
 192                   2          DIVIDE -> IDLE


    Summary                     Active      Hits    Misses % Covered
    -------                     ------      ----    ------ ---------
        States                       3         3         0     100.0
        Transitions                  4         3         1      75.0
```

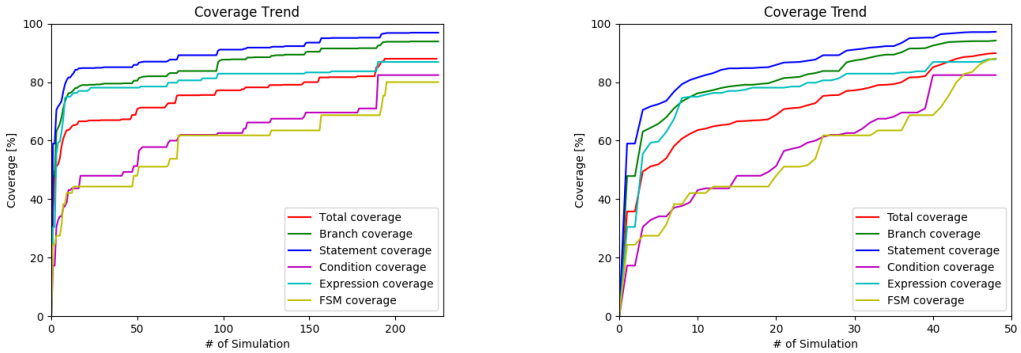
Figure 5.3: FSM coverage

5.1.6 Toggle Coverage

Toggle coverage reports the number of times each bit of signals has toggled its value. The basic toggle coverage is enabled with -t option and cover: 1→ 0 and 0→1 transition. QuestaSim provide also an extended toggle coverage (-x option) to cover also transition from and to undefined ("X"), and tristate ("Z")

5.2 Simulations

In this section, the results of the simulations are shown and discussed. As the aim of the verification is reaching a coverage value over 90% we will see step by step what was necessary to reach that value. Two different attempts have been done to reach 90%, in the first one a large number of simulation were required, while in the second attempt a reduced number was sufficient as only the best test-set has been included while the useless test-set were removed. As the test-size is an important parameter the second attempt is better and only that results will be explained, Coverage trends are reported in Fig. 5.4 for completeness.



(a) First attempt coverage trend

(b) Second attempt coverage trend

Figure 5.4: Coverage trends

5.2.1 Single Simulation

As a first approach, a single simulation with a random set of instructions has been run.
The expected result is that exploiting unconstrained randomization the value of the coverage will be of course lower than 50% because of the method employed in the random generator. In fact, as some opcode contains a large number of instructions while some other contains only one or two instructions and `random.choice()` method is not weighted, it is not possible to cover the major part of the instruction set with a single run.
The expected result has been confirmed by the results of the simulation shown in Fig. 5.5 and Fig. 5.6.

As you can see from Fig. 5.5 the number of instructions for each of the opcode is not balanced, having a great number of LUI and AUIPC. At this point, there

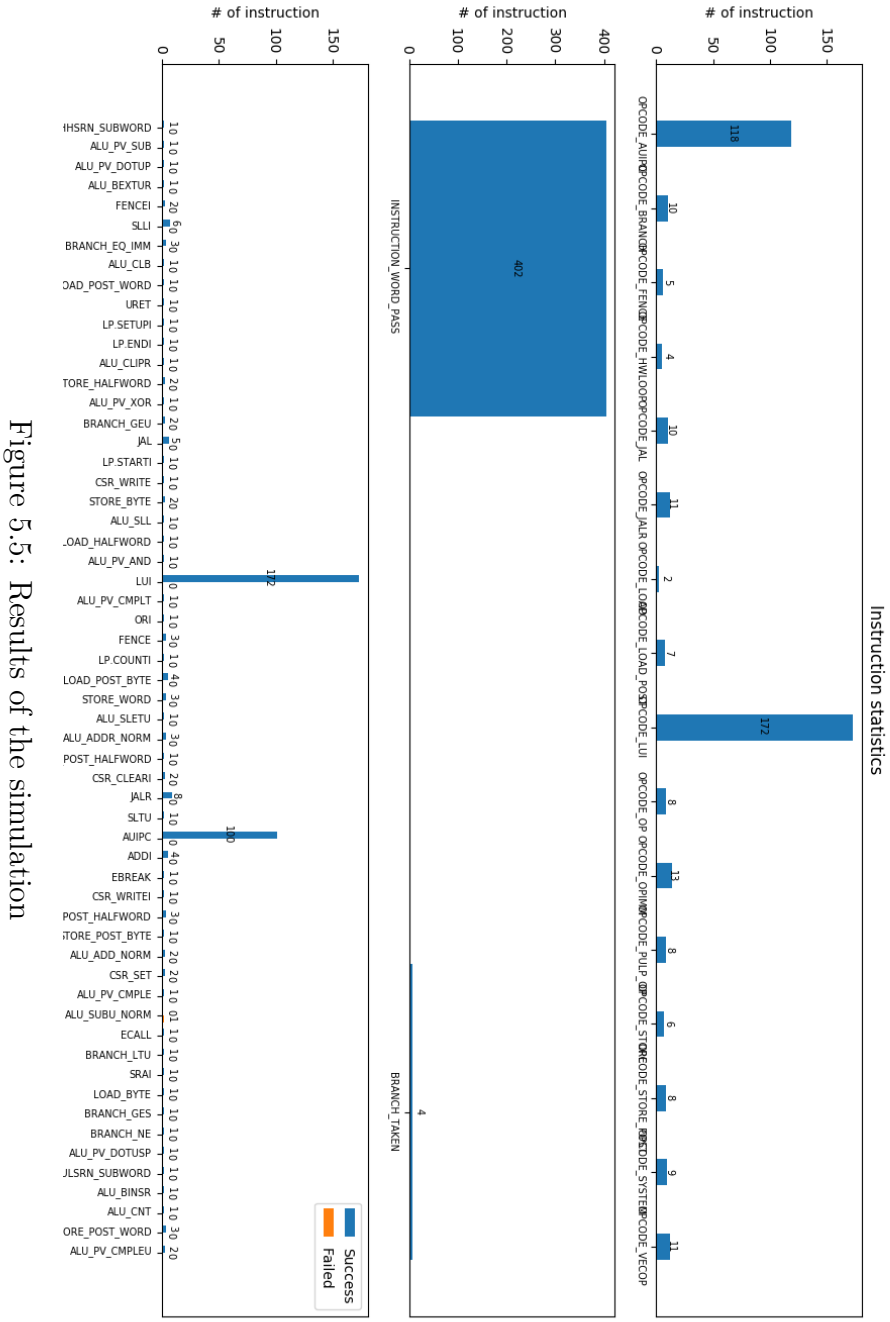


Figure 5.5: Results of the simulation

were two possible solutions:

- Modify the random generator assigning a weight to opcodes to increase the probability of executing different instructions avoiding repetition of the same instruction;
- Run multiple simulations and merge coverage databases.

Coverage report in Fig. 5.6 is a fast and convenient way to look at coverage results but is not sufficient to understand which part of the design has not covered. For this reason, together with this figure, two additional textual reports are provided:

- `CovReport.txt`: it shows the metric percentage reached for each of the components;
- `CovReportlines.txt`: It is an expanded version in which it is reported also detailed information about the uncovered parts of the design.

To increase the coverage the second technique has been exploited (i.e. Merging databases), so in the next subsections, the results of multiple runs are reported.

5.2.2 Multiple Simulations

QuestaSim provides a useful command `vcover merge` to merge the coverage results obtained in the previous simulations. This mechanism allows joining databases without repetition of previously covered parts. Now on, the coverage efforts will be explained focusing on what has been done to get a certain increase and discussing what has not been covered.

Instruction Set Coverage

After the first generic simulation, a sequence of constrained simulations has been launched to cover the entire Instruction Set. To be sure that normal situation has been properly tested we can look at the `riscv_compressed_decoder.sv` and `riscv_decoder.sv` coverage results from `CovReportlines.txt`.
For each of the simulation, the user is able to select the opcodes to be inserted according to the coverage requirements. Starting from standard Arithmetic operation up to vectorial and privileged almost all the instructions contained in ISA have been tested. Results of some simulations are shown in Fig. 5.7 and Fig. 5.8.

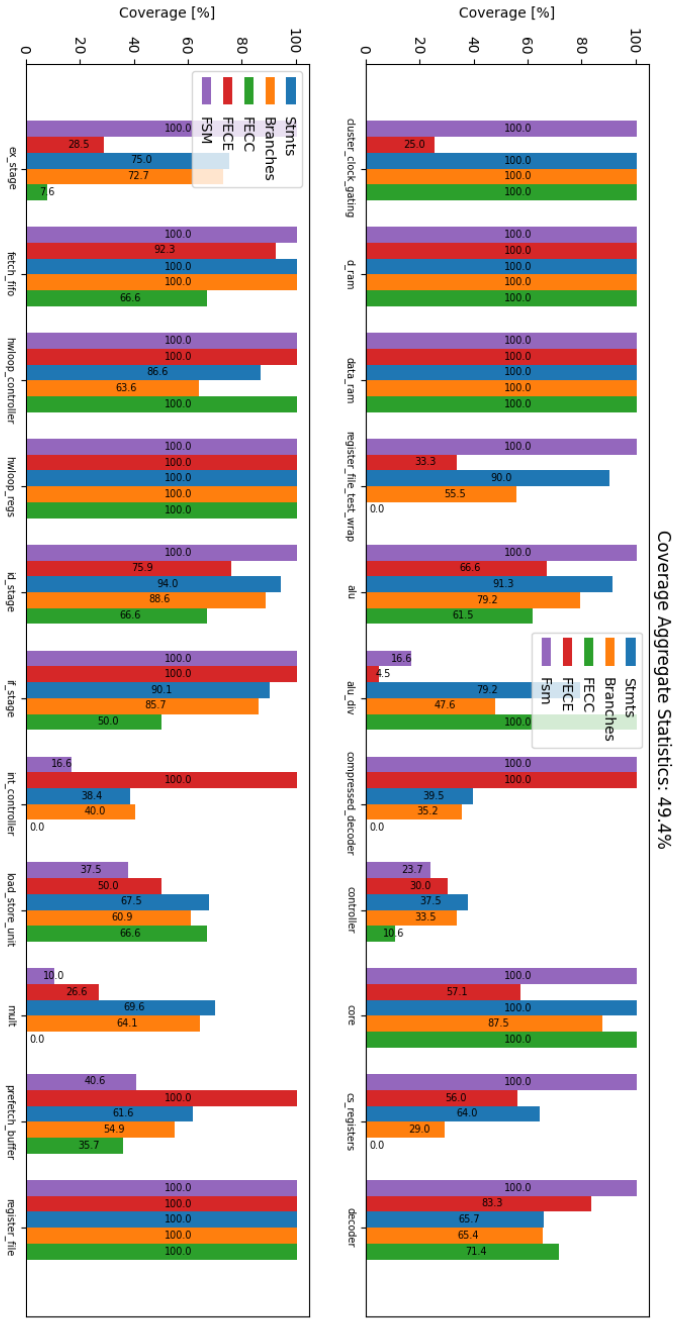
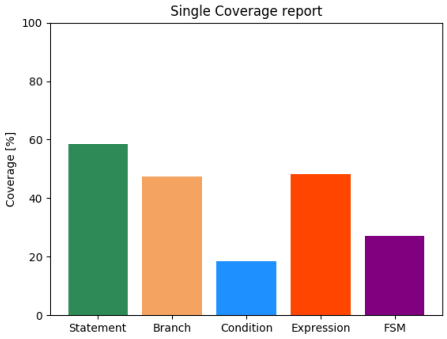
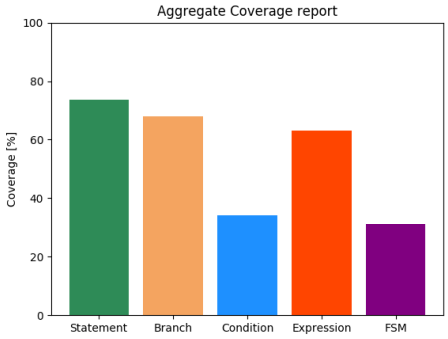


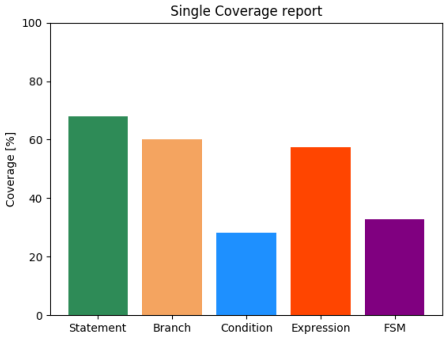
Figure 5.6: Coverage Report



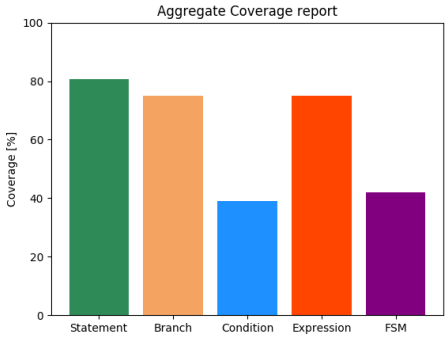
(a) Single run including AUIPC, OPCODE_OP, OPCODE_OPIMM



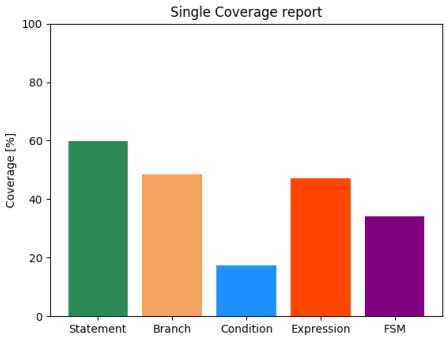
(b) Aggregate coverage after (a)



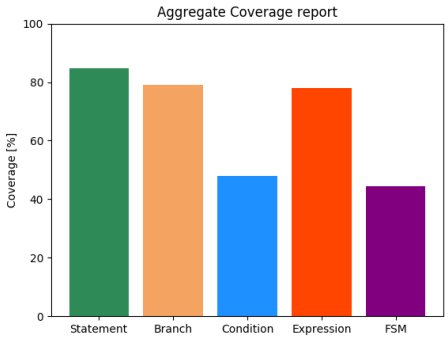
(c) Single run including OPCODE_SYSTEM



(d) Aggregate coverage after (c)



(e) Single run including PULP_OP

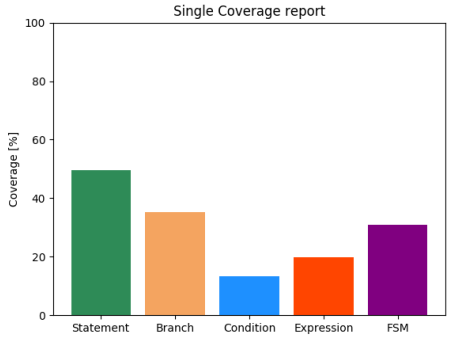


(f) Aggregate coverage after (e)

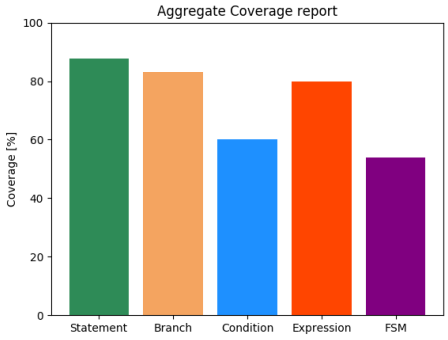
Figure 5.7: Instruction Set Coverage reports

At that point, Coverage reached 75,6% and a larger increase with this approach

was not possible as great results were already obtained for the decoder and compressed decoder. Complete coverage results are reported in Fig. 5.9.



(a) Single run including Control Transfer Instructions



(b) Aggregate coverage after (a)

Figure 5.8: Instruction Set Coverage reports

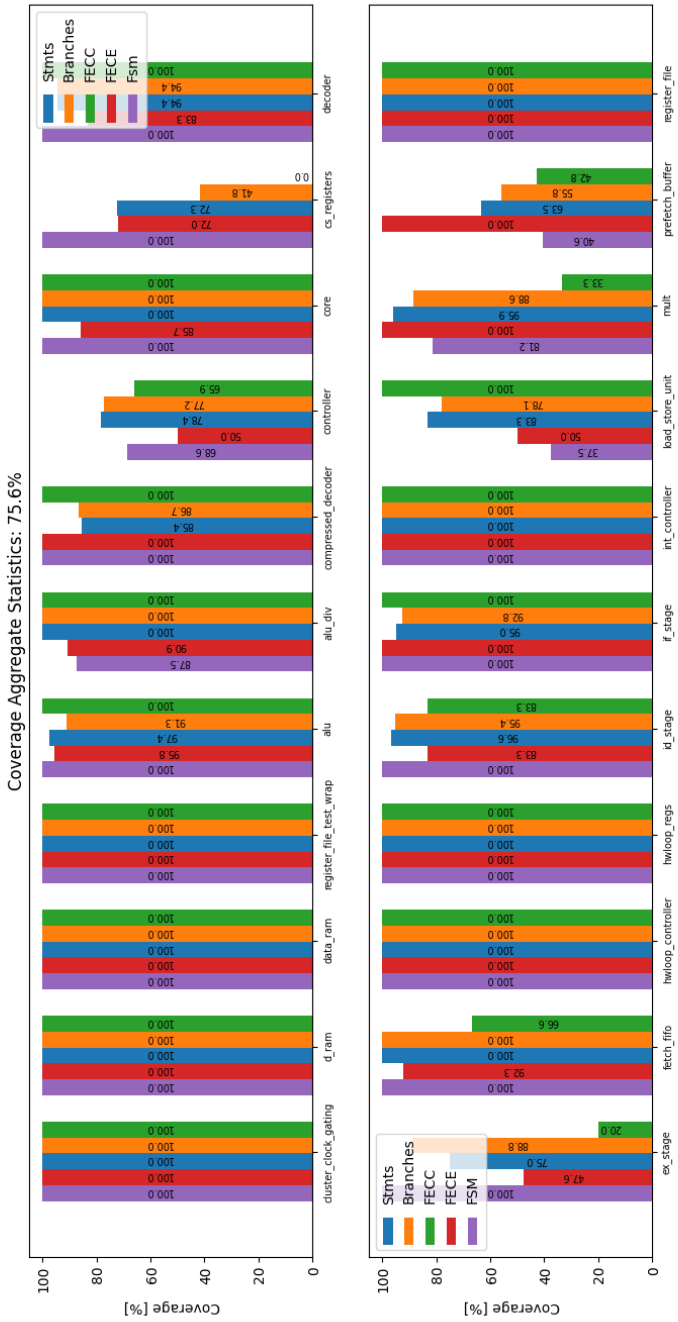


Figure 5.9: Coverage Report

Exception handling

By looking at Fig. 5.9 it appears clear that the verification effort must be directed to the `riscv_controller` and its related components. For this reason, the random generator has been modified to generate not only correct instructions but also illegal ones. Random Generator is capable of generating:

- Illegal Instructions;
- Misaligned Instructions;
- Wrong Hwloop.

In this way, the idea is to cover exceptions that raise in the device under test whenever rules are not respected. The already existing ISA database has been used to modify also the "non-modifiable" fields of the instructions. To be more clear a simple example is reported in Tab. 5.1
Correct SRAI instruction is highlighted in green while illegal SRAI is red-highlighted.

instr[31:20]	instr[19:15]	instr[14:12]	instr[11:7]	instr[6:0]	NAME
0100000-shamt	rs1	101	rd	0010011	SRAI
0100111-shamt	rs1	101	rd	0010011	SRAI-ILL

Table 5.1: Example of illegal instruction

The only difference is on the funct7 field which contains an abnormal value. When the generator picks a new instruction from the database, normally its fields are filled with register-addresses or immediate, but in this case, also the functional fields (i.e. the ones used to decode) are overwritten with random values.
Another kind of illegal instructions generated are instructions that does not respect the 32bit, or 16bit encoding. In this way misaligned access in instruction memory is simulated allowing a coverage of that corner case.
Many other types of corner cases has been inserted to cover exceptions handling of the design. Exploiting this approach a 82% of coverage has been reached. Coverage Results are reported in Fig. 5.10.

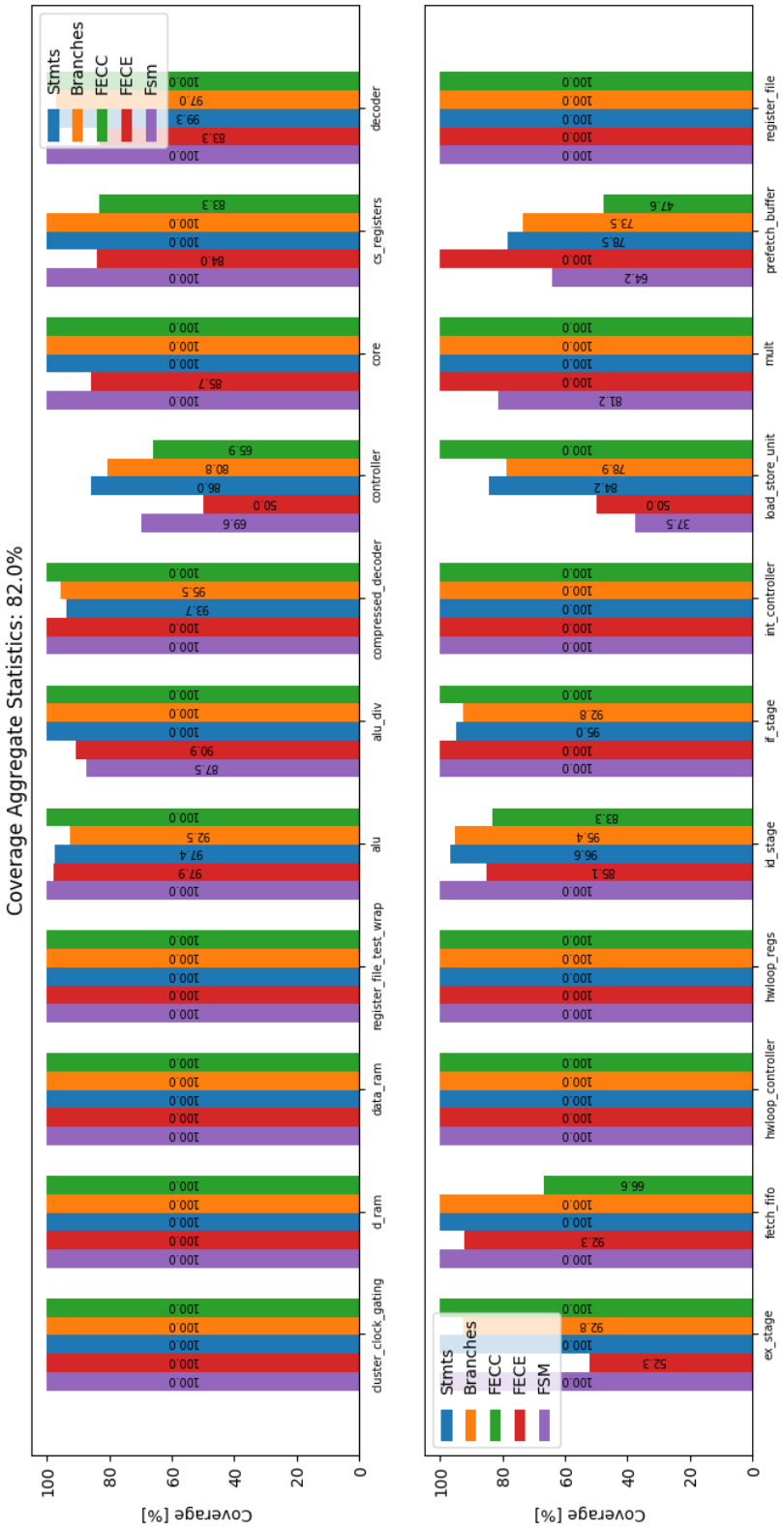


Figure 5.10: Coverage Report

Interrupt handling

The final effort to increase coverage was directed to interrupt handling. As interrupt controller's and controller's FSM contains states and transition that occurs only if interrupt request arrives and it is served, a mechanism to generate interrupts is required. This generation is done in the sequencer, signals related to interrupts were already declared in `processor_sequence.sv` so it was necessary to add some constraints on this signal to randomly send interrupt requests. In addition to that, to cover some FSM transitions in multiplier and controller it was necessary to randomly reset the DUV. After a couple of simulation 90.1% coverage has been reached (Fig. 5.12). The remaining part of the design that has not covered is due to corner cases hard to be caused or transition that never happens.

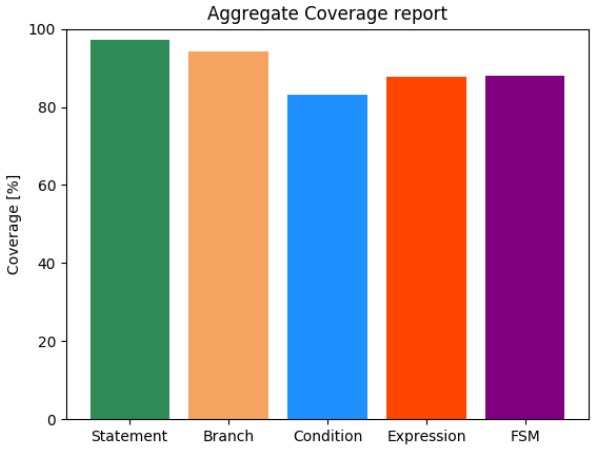


Figure 5.11: Instruction Set Coverage reports

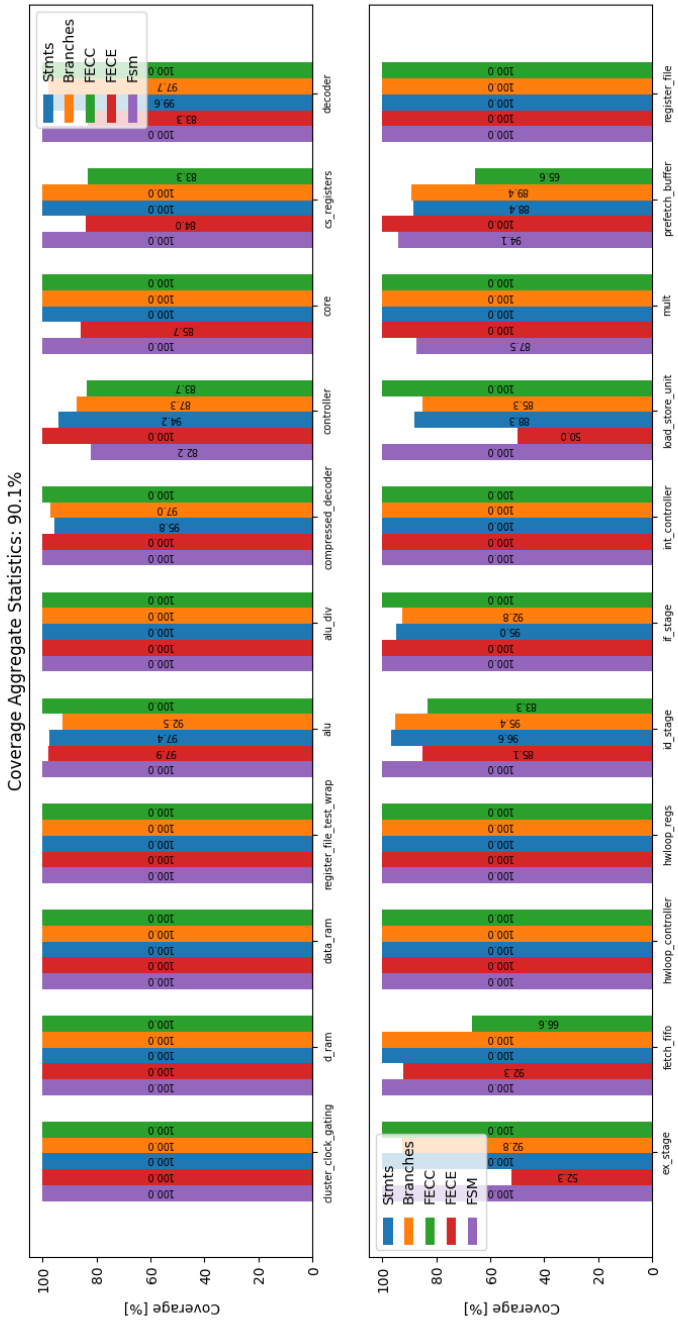


Figure 5.12: Coverage Report

Chapter 6

Conclusion and Future works

This work aimed to verify a complex processor architecture achieving a high level of confidence. The result provided in the previous chapter clearly shows that apart from some corner cases that were not possible to cover a level of coverage around 90% was achieved. It is important to specify that code coverage is a very strict way to check coverage as it is really hard to increase coverage in some cases. That result was achieved employing merge of Coverage result, otherwise, we have seen that for a single random program the coverage is around 50%. That is not good news as we discovered that randomization is not a good approach to achieve high coverage and a better approach could be introducing a smarter random generator that tries to improve its performance by looking at the previous coverage results. In addition to that, a better idea is to use an external ISS (Instruction Set Simulator) to be embedded in the scoreboard as the reference model. Introducing an ISS could give two main advantages:

- Simplify the UVM scoreboard operations;
- Increase the confidence on the pass/fail results. As ISS are well tested we avoid that errors present in DUV are repeated in the reference model.

That approach could be introduced in future works, as introducing in the currently developed UVM Framework was too complicated because the existent Instruction Set Simulator includes only certain RISC-V Extension and should be modified to include proprietary extensions.

Appendix A

ALU Extension

A.1 Bit Manipulation Operations

f2	Is3[4:0]	Is2[4:0]	rs1	funct3	rd	opcode	name
11	Luimm5[4:0]	luimm5[4:0]	src	000	dest	0110011	p.extract
11	Luimm5[4:0]	luimm5[4:0]	src	001	dest	0110011	p.extractu
11	Luimm5[4:0]	luimm5[4:0]	src	010	dest	0110011	p.insert
11	Luimm5[4:0]	luimm5[4:0]	src	011	dest	0110011	p.bclr
11	Luimm5[4:0]	luimm5[4:0]	src	100	dest	0110011	p.bset
10	00000	src2	src1	000	dest	0110011	p.extractr
10	00000	src2	src1	001	dest	0110011	p.extractur
10	00000	src2	src1	010	dest	0110011	p.insertr
10	00000	src2	src1	011	dest	0110011	p.bclrr
10	00000	src2	src1	100	dest	0110011	p.bsetr

Table A.1: Bit Manipulation Encoding

funct7	rs2	rs1	funct3	rd	opcode	name
0000100	src2	src1	101	dest	0110011	p.ror
0001000	00000	src1	000	dest	0110011	p.ff1
0001000	00000	src1	001	dest	0110011	p.fl1
0001000	00000	src1	010	dest	0110011	p.clb
0001000	00000	src1	011	dest	0110011	p.cnt

Table A.2: Bit Manipulation Encoding

A.2 General ALU Operations

funct7	rs2	rs1	funct3	rd	opcode	name
0000010	00000	src1	000	dest	0110011	p.abs
0000010	src2	src1	010	dest	0110011	p.slet
0000010	src2	src1	011	dest	0110011	p.sletu
0000010	src2	src1	100	dest	0110011	p.min
0000010	src2	src1	101	dest	0110011	p.minu
0000010	src2	src1	110	dest	0110011	p.max
0000010	src2	src1	111	dest	0110011	p.maxu
0001000	00000	src1	100	dest	0110011	p.exths
0001000	00000	src1	101	dest	0110011	p.exthz
0001000	00000	src1	110	dest	0110011	p.extbs
0001000	00000	src1	111	dest	0110011	p.extbz

Table A.3: General Alu Encoding

f2	Is3[4:0]	rs2	rs1	funct3	rd	opcode	name
00	Luimm5[4:0]	src2	src1	010	dest	1011011	p.addN
10	Luimm5[4:0]	src2	src1	010	dest	1011011	p.adduN
00	Luimm5[4:0]	src2	src1	110	dest	1011011	p.addRN
10	Luimm5[4:0]	src2	src1	110	dest	1011011	p.adduRN
00	Luimm5[4:0]	src2	src1	011	dest	1011011	p.subN
10	Luimm5[4:0]	src2	src1	011	dest	1011011	p.subuN
00	Luimm5[4:0]	src2	src1	111	dest	1011011	p.subRN
10	Luimm5[4:0]	src2	src1	111	dest	1011011	p.subuRN
01	Luimm5[4:0]	src2	src1	010	dest	1011011	p.addNr
11	00000	src2	src1	010	dest	1011011	p.adduNr
01	00000	src2	src1	110	dest	1011011	p.addRNr
11	00000	src2	src1	110	dest	1011011	p.adduRNr
01	00000	src2	src1	011	dest	1011011	p.subNr
11	00000	src2	src1	011	dest	1011011	p.subuNr
01	00000	src2	src1	111	dest	1011011	p.subRNr
11	00000	src2	src1	111	dest	1011011	p.subuRNr

Table A.4: General Alu Encoding

funct7	Is2[4:0]	rs1	funct3	rd	opcode	name
0001010	Iuimm5[4:0]	src1	001	dest	0110011	p.clip
0001010	Iuimm5[4:0]	src1	010	dest	0110011	p.clipu
0001010	src2	src1	010	dest	0110011	p.clipr
0001010	src2	src1	010	dest	0110011	p.clipur

Table A.5: General Alu Encoding

A.3 Immediate Branching

Imm12	Imm5	rs1	funct3	Imm12	opcode	name
imm[12|10:5]	imm5[4:0]	src1	010	imm[4:1|11]	1100011	p.beqimm
imm[12|10:5]	imm5[4:0]	src1	011	imm[4:1|11]	1100011	p.bneimm

Table A.6: Immediate Branching Encoding

A.4 MAC Operations

funct7	rs2	rs1	funct3	rd	opcode	name
0100001	src2	src1	000	dest	0110011	p.mac
0100001	src2	src1	001	dest	0110011	p.msu

f2	Is3[4:0]	rs2	rs1	funct3	rd	opcode	name
10	00000	src2	src1	000	dest	1011011	p.muls
11	00000	src2	src1	000	dest	1011011	p.mulhhs
10	Luimm5[4:0]	src2	src1	000	dest	1011011	p.mulsN
11	Luimm5[4:0]	src2	src1	000	dest	1011011	p.mulhhsN
10	Luimm5[4:0]	src2	src1	100	dest	1011011	p.mulsRN
11	Luimm5[4:0]	src2	src1	100	dest	1011011	p.mulhhsRN

Table A.7: MAC Encoding

f2	Is3[4:0]	rs2	rs1	funct3	rd	opcode	name
00	00000	src2	src1	000	dest	1011011	p.mulu
01	00000	src2	src1	000	dest	1011011	p.mulhhu
00	Luimm5[4:0]	src2	src1	000	dest	1011011	p.muluN
01	Luimm5[4:0]	src2	src1	000	dest	1011011	p.mulhhuN
00	Luimm5[4:0]	src2	src1	100	dest	1011011	p.muluRN
01	Luimm5[4:0]	src2	src1	100	dest	1011011	p.mulhhuRN
10	Luimm5[4:0]	src2	src1	001	dest	1011011	p.macsN
11	Luimm5[4:0]	src2	src1	001	dest	1011011	p.machhsN
10	Luimm5[4:0]	src2	src1	101	dest	1011011	p.macsRN
11	Luimm5[4:0]	src2	src1	101	dest	1011011	p.machhsRN
00	Luimm5[4:0]	src2	src1	001	dest	1011011	p.macuN
01	Luimm5[4:0]	src2	src1	001	dest	1011011	p.machhuN
00	Luimm5[4:0]	src2	src1	101	dest	1011011	p.macuRN
01	Luimm5[4:0]	src2	src1	101	dest	1011011	p.machhuRN

Table A.8: MAC Encoding

Appendix B

Vectorial Extension

For Vectorial extension, as the instruction set is replicated for sc, sci, normal and for half-words and bytes it is better to provide a table showing the behaviour instead of the encoding.

B.1 Vectorial ALU

Mnemonic	Description
pv.add[.sc,.sci]{.h,.b}	rd[i]=rs1[i]+rs2[i]
pv.sub[.sc,.sci]{.h,.b}	rd[i]=rs1[i]-rs2[i]
pv.avg[.sc,.sci]{.h,.b}	rD[i] = (rs1[i] + op2[i]) >>1
pv.avgu[.sc,.sci]{.h,.b}	rD[i] = (rs1[i] + op2[i]) >>1
pv.min[.sc,.sci]{.h,.b}	rD[i] = rs1[i] <op2[i] ? rs1[i] : op2[i]
pv.minu[.sc,.sci]{.h,.b}	rD[i] = rs1[i] <op2[i] ? rs1[i] : op2[i]
pv.max[.sc,.sci]{.h,.b}	rD[i] = rs1[i] >op2[i] ? rs1[i] : op2[i]
pv.maxu[.sc,.sci]{.h,.b}	rD[i] = rs1[i] >op2[i] ? rs1[i] : op2[i]
pv.srl[.sc,.sci]{.h,.b}	rD[i] = rs1[i] >>op2[i]
pv.sra[.sc,.sci]{.h,.b}	rD[i] = rs1[i] >>>op2[i]
pv.sll[.sc,.sci]{.h,.b}	rD[i] = rs1[i] <<op2[i]
pv.or[.sc,.sci]{.h,.b}	rD[i] = rs1[i] | op2[i]
pv.xor[.sc,.sci]{.h,.b}	rD[i] = rs1[i] ^op2[i]
pv.and[.sc,.sci]{.h,.b}	rD[i] = rs1[i] & op2[i]
pv.abs{.h,.b}	rD[i] = rs1 <0 ? – rs1 : rs1

Table B.1: Vectorial General ALU Instructions

Mnemonic	Description
pv.abs{.h,.b}	rD[i] = rs1 <0 ? – rs1 : rs1
pv.extract.h	rD = Sext(rs1[((I+1)*16)-1 : I*16])
pv.extract.b	rD = Sext(rs1[((I+1)*8)-1 : I*8])
pv.extractu.h	rD = Zext(rs1[((I+1)*16)-1 : I*16])
pv.extractu.b	rD = Zext(rs1[((I+1)*8)-1 : I*8])
pv.insert.h	rD[((I+1)*16-1:I*16] = rs1[15:0]
pv.insert,b	rD[((I+1)*8-1:I*8] = rs1[7:0]

Table B.2: Vectorial General ALU Instructions

Mnemonic	Description
pv.dotup[.sc,.sci].h	rD = rs1[0] * op2[0] + rs1[1] * op2[1]
pv.dotup[.sc,.sci].b	rD=rs1[0]*op2[0]+rs1[1]*op2[1]+rs1[2]*op2[2]+rs1[3]*op2[3]
pv.dotusp[.sc,.sci].h	rD = rs1[0] * op2[0] + rs1[1] * op2[1]
pv.dotusp[.sc,.sci].b	rD = rs1[0]*op2[0]+rs1[1]*op2[1]+rs1[2]*op2[2]+rs1[3]*op2[3]
pv.dotsp[.sc,.sci].h	rD = rs1[0] * op2[0] + rs1[1] * op2[1]
pv.dotsp[.sc,.sci].b	rD=rs1[0]*op2[0]+rs1[1]*op2[1]+rs1[2]*op2[2]+rs1[3]*op2[3]
pv.sdotup[.sc,.sci].h	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1]
pv.sdotup[.sc,.sci].b	rD=rD+rs1[0]*op2[0]+rs1[1]*op2[1]+rs1[2]*op2[2]+rs1[3]*op2[3]
pv.sdotusp[.sc,.sci].h	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1]
pv.sdotusp[.sc,.sci].b	rD=rD+rs1[0]*op2[0]+rs1[1]*op2[1]+rs1[2]*op2[2]+rs1[3]*op2[3]
pv.sdotsp[.sc,.sci].h	rD = rD + rs1[0] * op2[0] + rs1[1] * op2[1]
pv.sdotsp[.sc,.sci].b	rD=rD+rs1[0]*op2[0]+rs1[1]*op2[1]+rs1[2]*op2[2]+rs1[3]*op2[3]

Table B.3: Vectorial Dot Product Instructions

Mnemonic	Description
pv.shuffle.h	rD[31:16] = rs1[rs2[16]*16+15:rs2[16]*16] rD[15:0] = rs1[rs2[0]*16+15:rs2[0]*16]
pv.shuffle.sci.h	rD[31:16] = rs1[I1*16+15:I1*16] rD[15:0] = rs1[I0*16+15:I0*16]
pv.shuffle.b	rD[31:24] = rs1[rs2[25:24]*8+7:rs2[25:24]*8] rD[23:16] = rs1[rs2[17:16]*8+7:rs2[17:16]*8] rD[15:8] = rs1[rs2[9:8]*8+7:rs2[9:8]*8] rD[7:0] = rs1[rs2[1:0]*8+7:rs2[1:0]*8]
pv.shuffleI0.sci.b	rD[31:24] = rs1[7:0] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]

Table B.4: Vectorial Shuffle-pack Instructions

Mnemonic	Description
pv.shuffleI1.sci.b	rD[31:24] = rs1[15:8] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]
pv.shuffleI2.sci.b	rD[31:24] = rs1[23:16] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]
pv.shuffleI3.sci.b	rD[31:24] = rs1[31:24] rD[23:16] = rs1[(I5:I4)*8+7: (I5:I4)*8] rD[15:8] = rs1[(I3:I2)*8+7: (I3:I2)*8] rD[7:0] = rs1[(I1:I0)*8+7:(I1:I0)*8]
pv.shuffle2.h	rD[31:16] = ((rs2[17] == 1) ? rs1 : rD)[rs2[16]*16+15:rs2[16]*16] rD[15:0] = ((rs2[1] == 1) ? rs1 : rD)[rs2[0]*16+15:rs2[0]*16]
pv.shuffle2.b	rD[31:24] = ((rs2[26] == 1) ? rs1 : rD)[rs2[25:24]*8+7:rs2[25:24]*8] rD[23:16] = ((rs2[18] == 1) ? rs1 : rD)[rs2[17:16]*8+7:rs2[17:16]*8] rD[15:8] = ((rs2[10] == 1) ? rs1 : rD)[rs2[9:8]*8+7:rs2[9:8]*8] rD[7:0] = ((rs2[2] == 1) ? rs1 : rD)[rs2[1:0]*8+7:rs2[1:0]*8]
pv.pack.h	rD[31:16] = rs1[15:0] rD[15:0] = rs2[15:0]
pv.packhi.b	rD[31:24] = rs1[7:0] rD[23:16] = rs2[7:0]
pv.packlo.b	rD[15:8] = rs1[7:0] rD[7:0] = rs2[7:0]

Table B.5: Vectorial Shuffle-pack Instructions

B.2 Vectorial Comparison

Mnemonic	Description
pv.cmpeq[.sc,.sci].h,.b	rD[i] = rs1[i] == op2 ? ‘1 : ‘0
pv.cmpne[.sc,.sci].h,.b	rD[i] = rs1[i] != op2 ? ‘1 : ‘0
pv.cmpgt[.sc,.sci].h,.b	rD[i] = rs1[i] > op2 ? ‘1 : ‘0
pv.cmpge[.sc,.sci].h,.b	rD[i] = rs1[i] >=op2 ? ‘1 : ‘0
pv.cmplt[.sc,.sci].h,.b	rD[i] = rs1[i] < op2 ? ‘1 : ‘0
pv.cmple[.sc,.sci].h,.b	rD[i] = rs1[i] <= op2 ? ‘1 : ‘0
pv.cmpgtu[.sc,.sci].h,.b	rD[i] = rs1[i] > op2 ? ‘1 : ‘0
pv.cmpgeu[.sc,.sci].h,.b	rD[i] = rs1[i] >= op2 ? ‘1 : ‘0
pv.cmpltu[.sc,.sci].h,.b	rD[i] = rs1[i] < op2 ? ‘1 : ‘0
pv.cmpleu[.sc,.sci].h,.b	rD[i] = rs1[i] <= op2 ? ‘1 : ‘0

Table B.6: Vectorial comparison Instructions

Bibliography

[1] Pasquale Davide Schiavone Andreas Traber Micheal Gautschi. *RI5CY: User Manual*. April 2019.

[2] Andrew Waterman. *Design of the RISCV Instruction Set Architecture*. January 2016.

[3] SiFive Inc Andrew Waterman Krste Asanovic. *The RISCV Instruction Set Manual, Volume I: Unprivileged ISA*. University of California, Berkeley, July 2020.

[4] CadFlix. *ModelSim and Questa Code Coverage*. URL: `https://www.cadflix.net/video/modelsim-and-questa-code-coverage/`. (accessed: 19.01.2021).

[5] Chip-Verify. *UVM-Tutorial for Beginners*. URL: `https://www.chipverify.com/uvm/uvm-tutorial`. (accessed: 25.09.2020).

[6] Vanessa R. Cooper. *Getting Started with UVM: A Beginner's Guide*. 1st edition. Verilab Publishing, 22 May 2013.

[7] Doulos. *UVM Golden Reference Guide*. second. Doulos Ltd., 2013.

[8] Harry Foster. *Part 1: The 2020 Wilson Research Group Functional Verification Study*. URL: `https://blogs.sw.siemens.com/verificationhorizons/2020/11/05/part-1-the-2020-wilson-research-group-functional-verification-study/`. (accessed: 11.01.2021).

[9] Verification Guide. *UVM-Tutorial*. URL: `https://verificationguide.com/uvm/uvm-tutorial/`. (accessed: 21.12.2020).

[10] Guido Masera. *Integrated System Architecture notes*. 2020.

[11] Anish Gupta. *Processor-UVM-Verification*. 2018. URL: `https://github.com/gupta409/Processor-UVM-Verification/tree/master/Code`.

[12] Guru99. *Code coverage Tutorial*. URL: `https://www.guru99.com/code-coverage.html`. (accessed: 15.01.2021).

[13] Accellera Organization Inc. *System Verilog 3.1a: Language Reference Manual*. 2004.

[14] RISC-V International. *Getting Started with RISC-V Verification*. URL: https://riscv.org/blog/2020/05/getting-started-with-risc-v-verification/. (accessed: 17.09.2020).

[15] Mentor Graphics. *ModelSim Command Reference SE 6.0b*. 15.Nov.2004.

[16] Mentor Graphics. *ModelSim User's Manual SE 6.0b*. 15.Nov.2004.

[17] Andreas Meyer. *Principles of Functional Verification*. Newnes.

[18] Robert Ekendahl Mike Mintz. *Hardware Verification with System VERILOG: An Object-Oriented Framework*. Springer.

[19] Gordon E. Moore. "Cramming more components onto integrated circuits." In: (1965). DOI: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf.

[20] OpenHardware. *CV32E40P User Manual*. URL: https://core-v-docs-verif-strat.readthedocs.io/projects/cv32e40p_um/en/latest/intro.html. (accessed: 07.10.2020).

[21] Andrew Piziali. *Functional Verification Coverage Measurement and Analysis*. Springer.

[22] PLDWorld. *Code Coverage commands*. URL: http://www.pldworld.com/_hdl/2/_ref/se_html/manual_html/ce_cover.html. (accessed: 20.01.2021).

[23] Pulp-platform. *Pulp Implementation*. URL: https://pulp-platform.org/. (accessed: 27.11.2020).

[24] Ray Salemi. *The UVM Primer: An Introduction to the Universal Verification Methodology*. Boston Light Press, 2013.

[25] Kathleen Meade Sharon Rosenberg. *A Practical Guide to Adopting the Universal Verification Methodology (UVM)*. 1ST edition. Cadence Design Systems, 2010.

[26] Cadence Design Systems. *System Verilog Training bytes*. URL: https://www.youtube.com/watch?v=fBApIYoyx7E\&list=PLYdInKVfiOKZ1HMVNNcxvvWhYJMmLAq_g&index=1. (accessed: 01.09.2020).

[27] Udi Jonnalagadda Tao Liu Richard Ho. *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.

[28] Paul Wilcox. *Professional Verification : A Guide to Advanced Functional Verification*.

Ringraziamenti

Volendo fare un'analogia tra il lavoro di tesi e il funzionamento di un processore si potrebbe dire che è stato come un lungo programma eseguito istruzione dopo istruzione in attesa di una istruzione di jump che mi portasse alla conclusione. Quando ho iniziato il percorso universitario non mi sarei mai aspettato una conclusione di questo genere, con una tesi svolta completamente da casa, sforzandomi ogni giorno di fare un passo avanti nella direzione giusta. Questa situazione ha reso il lavoro più difficile, e quello che mi ha spinto ad andare avanti è stata la voglia di concludere questo lungo percorso nel migliore dei modi. Diverse persone mi sono state accanto alimentando la mia forza di volontà soprattutto quando veniva a mancare ed è a loro che voglio fare i miei ringraziamenti. In primis ci sono i miei genitori che con la loro costante presenza mi hanno sempre invogliato a fare il massimo credendo fortemente in me. Poi mio fratello maggiore Ignazio che con un percorso simile al mio, mi ha mostrato che a volte bisogna solo tenere duro ancora un attimo per raggiungere i propri obiettivi.
Un ringraziamento speciale va a Monica Monticciolo che in tutti questi anni mi ha incoraggiato ad andare avanti aiutandomi sempre a vedere il lato positivo in ogni situazione ma soprattutto quando la confusione e lo stress annebbiavano la mia vista è riuscita a mostrarmi la via di uscita.
Un grande ringraziamento va a tutti i miei amici di Trapani e quelli di Torino con i quali ho condiviso i pochi momenti di gioia di uno studente del Politecnico: Mattia La Francesca, Marco Iuculano, Francesco Biasibetti, Gianluca Monaco, Marco Saladino, Albertino Scarlata,Matteo Ripa, Michelino Baratto, Aldo Moschetti, Francesco Minosi, Filippo Sanangelantoni,Federica Nasr,Fabio Asti. Un ulteriore ringraziamento è rivolto al mio coinquilino, Giuseppe Narducci, con la nostra "ora del tè" e "birra e patatine" ha contribuito ad allegerire il momento che stavo attraversando. Infine vorrei ringraziare i due miei colleghi con i quali ho svolto la maggior parte dei progetti in questi anni: Federico Di Fazio e Manuel Capaccio senza i quali probabilmente non sarei qui a scrivere questi ringraziamenti. Con loro ho condiviso momenti difficili sapendo che prima o dopo avremmo trovato una soluzione. Inoltre vorrei ringraziare il professor Sanchez ed Annachiara Ruospo per avermi fornito l'opportunità di lavorare ad un progetto così stimolante e per avermi aiutato in ciascuna delle fasi del lavoro.