UNIVERSITÄT BREMEN

BACHELORARBEIT

Co-Simulation-Based Verification of a Register-Transfer Level RISC-V Implementation in Reference to a Virtual Prototype

---

Autor:
Luca Müller

Erstgutachter:
Prof. Dr. Rolf Drechsler
Zweitgutachter:
Dr. Thomas Röfer

Abgabedatum: 29.10.2021

Abstract. In this thesis, a co-simulation-based framework for the verification of a RISC-V implementation on the register-transfer level is presented. The co-simulation flow provided by the RISCV-DV verification tool is implemented for the register-transfer level core MicroRV32 with the virtual prototype RISCV-VP as its reference instruction set simulator. A variety of tests are executed with the flow and the discovered bugs are documented. Functional coverage information of the tests is collected to assess the quality of the verification. Finally, the implementation is evaluated in the context of how it can serve as a reference for similar verification efforts and what improvements and additions may be conducted in the future.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Verification is an essential part in every hardware design process. It aims to ensure the correctness of the design according to its specification in order to avoid any bugs from occurring in the eventual hardware realization. One accessible and efficient technique that can be used for this purpose is simulation, in which the hardware is run for specific inputs and its outputs are compared to the expected results in order to verify whether the hardware works correctly [CS03].
As a means to avoid having to manually calculate the expected results of a simulation run, the so-called co-simulation method can be used. Its approach is to run a simulation with the same inputs on two hardware implementations for the same specification, preferably on different levels of abstraction. The system under test is run alongside a reference system, for instance an *Instruction Set Simulator* (ISS) or a *Virtual Prototype* (VP), which has been verified itself and provides the expected results for the execution of the simulation. When a mismatch between the outputs of the simulation occurs, its cause can be traced and bugs in the implementation of the *Device Under Test* (DUT) can be fixed. It might also be the case that a mismatch is caused by the reference system, but as long as the DUT and the reference system do not have the same bug in their implementation, all errors can be found by mismatches in the co-simulation.

In this thesis, a framework for the co-simulation-based verification of RISC-V hardware designs at the *Register-Transfer Level* (RTL) is presented. RISC-V [WA19a] is an *Instruction Set Architecture* (ISA) which has gained great popularity in recent times due to its open-source nature and has sparked interest in many projects like implementations on different abstraction levels, as well as tools supporting the design and verification process, creating a large ecosystem surrounding the ISA. The framework uses a virtual prototype as its reference system and makes use of the RISCV-DV instruction generator, which offers a co-simulation flow with trace comparison, as well as a functional coverage model. It consists of multiple extensions on RISCV-DV, as well as the RTL core MicroRV32 [AP20] and the RISC-V Virtual Prototype [Her+18],

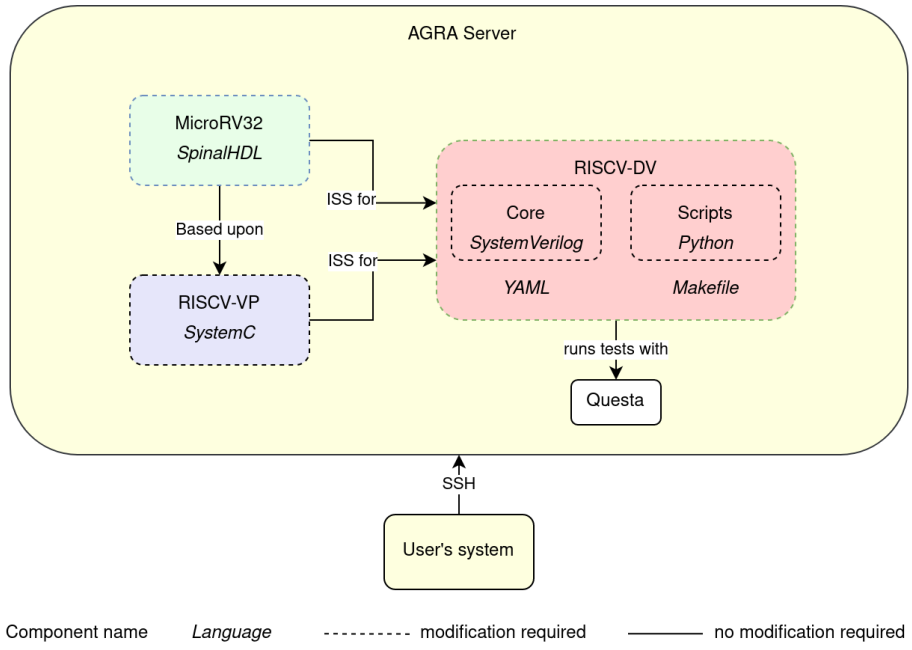

Figure 1.1: Existing components and their relations

which both are RISC-V implementations on different abstraction layers currently being developed at the *Group of Computer Architecture* (AGRA) [Agr] at the university of Bremen [Uni]. Their development is conducted as a co-design, meaning the functionality is first implemented on the virtual prototype alongside application specific software and then adopted for the RTL core. This makes the co-simulation approach adequate for the verification of MicroRV32, as it is meant to have the exact same functionality as RISCV-VP. Although this specific framework can only be used for the the RTL core and the VP, its implementation can serve as a blueprint and show the feasibility for similar efforts.

1.1 Existing Components

Figure 1.1 shows the already existing components and their relations to one another. The central piece is MicroRV32, which is implemented in SpinalHDL [Spib]. The modern hardware design language uses the *Scala Build Tool* (SBT) [Sbt] to build the project and simulate the execution of the core. The functionality of MicroRV32 is based upon RISCV-VP, which itself is written in SystemC [Sysa], a class library for C++, meaning it can be built using a standard C++ compiler and the resulting binaries can be executed.

These two RISC-V implementations, despite being complete RISC-V platforms and not just instruction set simulators, can be integrated into the RISCV-DV toolflow like ISSs are. The core implementation of RISCV-DV is written in the hardware verification language SystemVerilog [Sysb] with *Universal Verification Methodology* (UVM) and is supplemented by various scripts implemented in Python [Pyt]. Additionally, the generator is configured with the help of YAML [Yam] files and several Makefiles have to be extended to automate the execution and integrate both ISSs.
RISCV-DV's test generator is executed using a simulator supporting SystemVerilog and UVM, which for this thesis is the Questa Advanced Simulator [Que]. Since Questa is a commercially licensed tool, the access to the tool is provided through the AGRA servers. For this reason, the complete framework runs on one of these servers, which can be accessed via SSH.

The amount of interconnected components requiring different kinds of adjustments give rise to the challenge of determining where exactly extensions need to be made. Only then can it be assessed how they are realized, which is further complicated by the various programming and description languages used in their implementations.

1.2 Goals of this Thesis

The main objective of this thesis is the implementation and evaluation of a co-simulation-based verification framework.
In order to make use of RISCV-DV's co-simulation flow, several adjustments need to be made on the tool itself, MicroRV32 and RISCV-VP. They are presented in detail in chapter 3.1. Once the flow is implemented, RISCV-DV will be able to compare the outputs of a verification run. Each iteration will report any mismatches that occur between the RTL core and the VP, indicating that their results for the execution of a generated program differ. When a mismatch is found, it can be investigated where it stems from and whether there is a bug in the implementation of MicroRV32 or even RISCV-VP. In addition to this trace comparison, RISCV-DV can also extract coverage information while running the co-simulation.

With those capabilities of RISCV-DV in mind, the goals of this thesis are as follows:

I Realize the co-simulation of RISCV-DV for MicroRV32 and RISCV-VP by implementing the necessary extensions in all three components.

II Run multiple iterations of the tests provided by RISCV-DV in order to find bugs in MicroRV32 and potentially RISCV-VP.

III Collect information on the coverage of the tests in order to make assumptions on the correctness of the RTL core if no mismatches are reported.

IV Explore what the results mean for the feasibility of efforts to use co-simulation for verification on the register-transfer level and how this approach can be improved further.

1.3 Structure

The remainder of this thesis is structured as follows:
Chapter 2 provides an overview of the RISC-V instruction set architecture and the components used in the presented framework.
Chapter 3 elaborates on the implementation, presenting some extracts of the code and giving rationale on why some aspects are realized the way they are.
Chapter 4 illustrates the bugs which were found in both MicroRV32 and RISCV-VP with the framework and the functional coverage that was achieved.
Chapter 5 gives an outlook on how the presented framework may be extended and what modifications may be necessary in case of changes to the underlying components.
Chapter 6 concludes this thesis with an overview of what was achieved and what could be improved.

Chapter 2

Background

A variety of frameworks and tools were used during work on this thesis. With the RISC-V ISA, MicroRV32, RISCV-VP and RISCV-DV, some of them are briefly introduced in this chapter.

2.1 RISC-V: The Instruction Set Architecture

RISC-V [WA19a] is an open-source instruction set architecture, originally developed for educational purposes at the University of California, Berkley. Today it is managed by the RISC-V Foundation [Risa] which has found industry partners such as Huawei, Western Digital and Google [Risb] that are working to create commercial solutions based on the RISC-V ISA.
While RISC-V offers 32-bit, 64-bit and 128-bit address space variants, this thesis will focus on a 32-bit address space, as it is the only one supported by MicroRV32 as of now.

The ISA is built with a small base integer ISA in favor of powerful modification options and extensibility. This is realized via optional instruction-set extensions which add functionality to the base instruction set. The standard extensions defined in the RISC-V manual range from the "M" extension for multiplication and division, over the "C" extension for compressed instructions to the "F" extension for single-precision floating-point operations. Developers may also define their own non-standard extensions for highly specialized applications. All RISC-V processor implementations must however support the RV32I base instruction set architecture. As with the address space variants, the remainder of this thesis will deal with the RV32I base ISA, as, apart from the CSR extension, no extensions have been implemented in MicroRV32 as of now.

Figure 2.1 shows the four base instruction formats defined for RV32I. The `opcode`, `funct3` and `funct7` fields are used to identify the specified instruction. The `imm` fields are used to construct the immediate value if needed for the instruction. The `rs1`, `rs2` and `rd` fields specify the first source register,

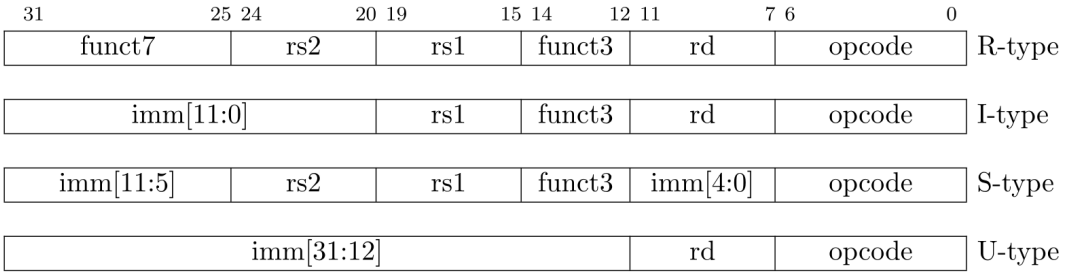

Figure 2.1: The four base instruction formats of the base RV32I ISA (reprinted from [WA19a])

second source register and destination register of the instruction respectively.

- The **R-format** is used by register-register instructions where both source operands are registers. These include arithmetic operations like `ADD` and `SUB`, logical operations like `AND`, `OR` and `XOR` and bit-shift operations like `SLL` and `SRL`.
- The **I-format** is used by register-immediate instructions where one source operand is a register and the other is an immediate value. These are the same as for the R-format except that the second operand is an immediate value instead of a register. The I-format is also used for the `LOAD` instruction.
- The **S-format** used by the `STORE` instruction as well as all branch instructions.
- The **U-format** is used by the `LUI` and `AUIPC` instructions as well as for the unconditional jump instructions `JAL` and `JALR`.

In addition to these instructions, the RV32I ISA also supports access to system functionality with the `ECALL` and `EBREAK` instructions, which both use the `SYSTEM` opcode, as well as the `FENCE` operation.

The base ISA supports 32 *General Purpose Registers* (GPRs) named x0-x31, with a width of 32-bit each. Apart from the `x0` register, which represents a hardwired 0, no register serves a specific purpose, although there are conventions for the usage of registers specified in the manual. There is one additional register called `pc`, which holds the value of the current *Program Counter* (PC). Besides the standard register name, where the decimal number of the register is simply preceded by an "x", there is the *Application Binary Interface* (ABI) name, where each register has its unique name, e.g. "zero" for the register number 0.
In addition to these general purpose registers, the RISC-V manual on the

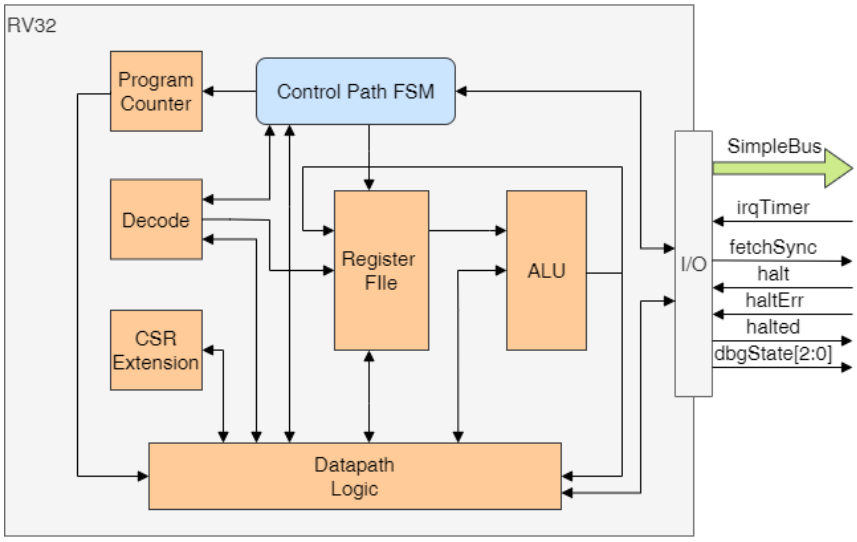

Figure 2.2: Diagram on the composition of MicroRV32 (reprinted from [AP20], p. 16)

privileged architecture [WA19b] specifies some *Control and Status Registers* (CSRs). CSRs are read-only or read-write registers containing information such as the capabilities of the processor, like `misa` and `marchid`, or the current state of operation, like `mstatus`. The control and status registers can be manipulated by the six CSR instructions. They all use the `SYSTEM` opcode, like the `ECALL` and `EBREAK` instructions do and are defined in the *Zicsr* extension.

2.2 MicroRV32: The RTL Platform

MicroRV32 [AP20] [APHD21b] is a SpinalHDL-based implementation of an RV32I_Zicsr RISC-V core on the register-transfer level and was created for a master thesis by Sallar Ahmadi-Pour. It is suitable to be used on an FPGA and comes with some peripheral implementations and software that can run on the core.

Figure 2.2 shows of which components MicroRV32 consists and how these components are connected.

- The **ALU** is responsible for the execution of instructions by providing arithmetic and logical operations.
- The **CSR extension** provides implementations for some control and status registers. These include `mhartid`, `mstatus` and `mie`.
- The **Datapath Logic** describes where data needs to be allocated in order to execute an instruction.

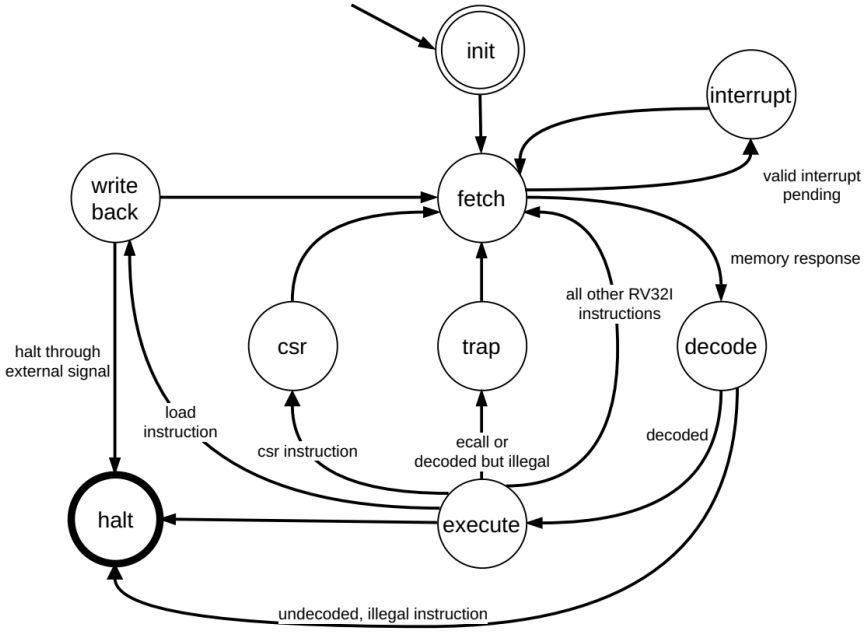

Figure 2.3: The control path of MicroRV32 (reprinted from [AP20], p. 17)

- The **Decode** module is used for decoding the instruction and providing signals to be used for its execution.
- The **Program Counter** takes care of all the logic regarding the current PC.
- The **Register File** implements the 32 general purpose registers defined by the RV32I ISA.

Instructions generally run through three phases in the core: `fetch`, `decode` and `execute`. This three-phase cycle takes four clock cycles, as fetching an instruction with the associated memory access requires two clock cycles, unlike decoding and executing which only take one clock cycle. The complete Control Path *FSM (Finite State Machine)* of MicroRV32 is shown in figure 2.3. Note that the `halt` state indicates that the core shall stop running.

The simulation loop for MicroRV32 as implemented in SpinalHDL [Spib] is outlined as pseudo-code in listing 2.1. The loop condition is set to false when the control path of the core has reached the `halt` state. The simulation loop then waits for a rising edge of the clock, causing all components of the core to update their signals according to their implementation.

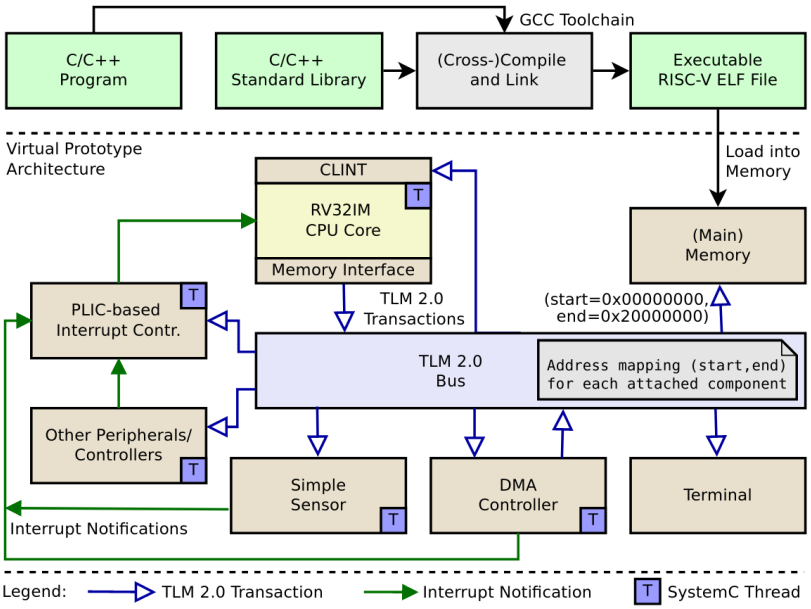

Figure 2.4: Architecture overview of the RISCV-VP (reprinted from [Her+18], p. 2)

Listing 2.1: Pseudo-code of MicroRV32 simulation loop in TopSim module

```
1  while(dutRunning) {
2      dutRunning = !dut.cpu.io.halted
3      dut.clockDomain.waitRisingEdge()
4  }
```

2.3 RISCV-VP: The Virtual Prototype

RISCV-VP [Her+18] is a SystemC [Sysa] virtual prototype supporting both RV32IMAFDC and RV64IMAFDC RISC-V cores. It was first presented in 2018 by Herdt et. al. and is still constantly being worked and improved upon. Like MicroRV32, the VP comes with peripheral support and software examples which can be run on it.

The original architecture of the VP is shown in figure 2.4. For the purposes of this thesis, the most important component will be the **CPU Core**, which implements all instructions MicroRV32 does, as well as the extensions supported by the VP. One detail to note is that the **Main Memory** of the VP is initialized with a RISC-V *ELF* (executable and linkable format) file, which makes the integration with RISCV-DV easier, as will be elaborated later.

For the RISCV-VP, the simulation loop includes more steps than for MicroRV32, as the ISS module takes care of instruction fetching, decoding and execution. Listing 2.2 shows the main aspects of the RISCV-VP simulation loop implemented in SystemC. While the VP's core is runnable, another step is executed, handling any errors that might occur. During each execution step, the instruction is loaded from the memory at the address of the current PC. The instruction is then decoded, yielding its opcode, and the PC is preemptively incremented by 4, as each instruction takes up 4 bytes in memory. Finally, the instruction is executed according to its opcode.

Listing 2.2: Pseudo-code of RISCV-VP simulation loop in ISS module

```
1  void ISS::exec_step() {
2      instr = Instruction(load_instr(pc));
3      op = instr.decode();
4      pc += 4;
5      switch (op) {/* Execute instruction according to opcode */}
6  }
7
8  void ISS:run_step() {
9      try {
10         exec_step();
11     } catch (SimulationTrap &e) {/* Handle error in execution step */}
12 }
13
14 void ISS::run() {
15     do {
16         run_step()
17     } while(status == Runnable)
18 }
```

2.4 RISCV-DV: The Verification Tool

RISCV-DV [Risc] is an open-source instruction generator which offers tools for the verification RISC-V processors. It supports the RV32IMAFDC and RV64IMAFDC instruction sets and is maintained by Google.

The main feature of RISCV-DV is its instruction generation. The tool offers a variety of tests that can be performed on RISC-V implementations on different levels of abstraction, ranging from ISSs like Spike [Spia], over virtual prototypes like RISCV-VP, to actual RTL implementations like MicroRV32.

The base test strategies included in RISCV-DV all serve varying purposes by focusing on a specific group of instructions.

- The **riscv_arithmetic_basic_test** is the simplest test provided by RISCV-DV. It only includes arithmetic and logical instructions, without any memory access via load/store operations or branching. This makes it

suited to detect general problems of a hardware design as only the instructions that are easiest to implement are tested.

- The **riscv_jump_stress_test** focuses on jumps and branches by performing back-to-back unconditional and conditional jump operations. This may reveal any potential errors in the design of features like pipelining or address calculations for the program counter.
- The **riscv_loop_test** works with similar instructions as the **riscv_jump_stress_test** but with more realistic scenarios by running a program with multiple loops, giving it more versatility.
- The **riscv_unaligned_load_store_test** is a very specific test, focusing on unaligned load and store operations, especially `LH` and `LB` as well as `SH` and `SB`. These instructions are of special interest, since despite the freedom of RISC-V implementations to choose the endianness of words in memory ([WA19a], p. 9), these unaligned memory instructions still need to remain deterministic and conform to the specification.

By extending the instruction generator or implementing a custom one, these tests may be adjusted and new tests may be created.

In order to verify the results these tests yield, RISCV-DV offers a co-simulation flow. Its idea is to run two ISSs, which need to be made usable by RISCV-DV, with the same program and have them both generate logs on information like the executed instructions and their results. These logs are then converted to a standardized CSV format and the outputs are compared. The details of this co-simulation flow are explained in chapter 3.1, while the way MicroRV32 and RISCV-VP are made available as ISSs for RISCV-DV is presented in chapter 3.2.

Another feature of RISCV-DV which is currently still work in progress is a functional coverage model. It builds on the information required for the co-simulation flow to extract the coverage of all tests that were conducted. The covergroup currently includes information on categories like all operand registers and positive/negative immediate values and may also be extended due to RISCV-DV's open-source nature.

Chapter 3

Co-Simulation Framework

This chapter elaborates on the implementation of the co-simulation framework by outlining the required steps before presenting details on how the realization looks like. Additionally, the functional coverage functionality of RISCV-DV is presented. The implementation accumulates to 1043 lines of code in 21 files.

3.1 Co-Simulation Flow in RISCV-DV

Before the verification framework can be extended, RISCV-DV's co-simulation flow and the steps required to realize it need to be understood. The flow is explained in RISCV-DV's documentation [Risd] and is shown for MicroRV32 and RISCV-VP in figure 3.1. The first step is to simulate them both with the same program, which is generated as a test ELF file by RISCV-DV. The instructions it contains are determined by the test strategy chosen for a given iteration. During simulation, MicroRV32 and RISCV-VP each generate a log file containing relevant trace information. This log file is then translated to a trace CSV file via the ISS-specific `log2csv` script. Once this trace CSV file can be generated during test execution, RISCV-DV can compare the results of the core and the VP and coverage information can be collected.

In order to realize the co-simulation for MicroRV32 and RISCV-VP, all of these steps need to be extended. MicroRV32 and RISCV-VP have to be modified to be usable as ISSs for RISCV-DV, which requires changes on all three components. Once tests can be run, both ISSs have to be enabled to collect logging

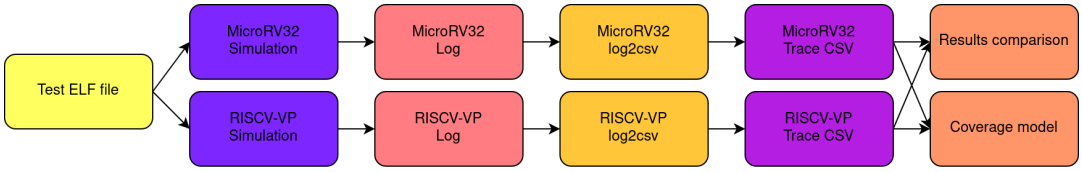

Figure 3.1: The co-simulation flow of RISCV-DV (assembled from [Risd], section "Appendix")

information relevant to the trace CSV format. In order to translate the log file to the aforementioned CSV format, the `log2csv` script needs to be implemented for both MicroRV32 and RISCV-VP.

3.2 MicroRV32 & RISCV-VP as ISSs for RISCV-DV

In order to use MicroRV32 and the RISCV-VP as an ISS for RISCV-DV, they first are added to the list of available ISSs. RISCV-DV uses the `iss.yaml` configuration file for this purpose. The integration requires minimal additional modifications for RISCV-VP, as it is already available in the version of RISCV-DV used for this thesis thanks to [APHD21a]. RISC-V comes with various platform implementations with different sets of memory and peripheral configurations. From these the `test32-vp` is suited for the co-simulation flow and is run with the `dv_mode` argument, which enables the conditional execution of code only relevant for RISCV-DV, as well as the specification at which address the memory should start. Lastly, listing 3.1 shows how the iss.yaml is modified and how RISCV-VP is executed with a generated ELF binary and other parameters.

Listing 3.1: RISCV-VP as an ISS for RISCV-DV

```
1 # Custom ISS: riscv virtual platform
2 - iss: riscv-vp
3 path_var: RISCV_VP_PATH
4 cmd: >
5 <path_var>/test32-vp --dv-mode --memory-start=2147483648 <elf>
```

Integrating MicroRV32 into RISCV-DV requires more steps than with RISC-VP because its RTL core is based on SpinalHDL. As the hardware description in this language is meant for both simulation and synthesis into FPGA hardware, the toolchain is set up differently. Since SBT is required to be executed in the directory in which the SpinalHDL project resides, RISCV-DV uses a Makefile to call a Makefile in the MicroRV32 path in order to execute SBT. Listing 3.2 shows how in the `iss.yaml` file, another make command (`mrv_make`) is called, which is specified in a Makefile at the root of the RISCV-DV project directory. This in turn calls another make target in the directory of MicroRV32, which then simulates the RTL core with the specified ELF file and the `dv_mode` argument.

Listing 3.2: MicroRV32 as an ISS for RISCV-DV

```
1 # microrv32
2 - iss: mrv32
3 path_var: MRV32
4 cmd: >
5 make -s mrv_make elf="<elf>"
```

After MicroRV32 and RISCV-VP are integrated as ISSs, they have to be able to execute the ELF file which RISCV-DV generates for its tests. For RISCV-VP, executing ELF binaries is already supported, so the ELF file can be passed directly.
MicroRV32 on the other hand natively runs with a so-called hexfile instead. A hexfile contains the instructions words as eight-digit hexadecimal numbers, with each instruction word on a separate line. Upon execution of the core, the memory is created with enough space for all instruction words from the hexfile. The total number of instruction words is hard-coded into the module and executing a different hexfile requires modification of the source code. In order to convert any ELF file provided to run on the core to a hexfile, MicroRV32 comes with the `elf2bin.py` script which takes care of the conversion. Listing 3.3 shows how the testbench module MicroRV32TopSim is extended in order to automate the ELF to hexfile conversion. This allows RISCV-DV to call the MicroRV32 simulation with a given test in the ELF file format.

Listing 3.3: Automatic conversion from ELF file to hexfile for MicroRV32

```
1   // Convert elf-file to hex-file
2   if (args.size < 1) {
3     println("Simulation error: name of elf-file not provided")
4     System.exit(1)
5   }
6   val elfFileName = args(0)
7   val hexFileName = "./mrv32_hexfile"
8   val mrv32_home = sys.env("MRV32")
9   val e2b_ret = Seq(mrv32_home + "/microrv32/sw/elf2bin.py", elfFileName,
      hexFileName).!
10  val memSize = Source.fromFile(hexFileName).getLines.size
11  if(memSize == 0 || e2b_ret != 0) { // Either empty memory or error during
      elf2bin execution
12    println("Simulation error: elf-file conversion failed.")
13    new File(hexFileName).delete()
14    System.exit(1)
15  }
```

The conversion script is executed in line 9 and the output hexfile is stored as `mrv32_hexfile`, while the size of the memory is determined by the number of lines of the hexfile in line 10. With these changes, MicroRV32 can now take the ELF file created by RISCV-DV as an argument, convert it to the required hexfile, and dynamically instantiate the memory with that hexfile and the calculated memory size.

Even though the ELF file can now be used, there is still a problem that is encountered when trying to run MicroRV32 as an ISS. The core is unable to use the interrupt handler integrated into the code of the tests which is used for termination, meaning it runs indefinitely. The explanation for this is a combination of the way RISCV-DV terminates its tests and how termination

works in MicroRV32.
Listing 3.4 shows how RISCV-DV's tests are terminated by executing an ECALL instruction after loading the value `1` into the gp register. This prompts the execution of the trap handler code provided by RISCV-DV. It is important to note however that none of the executed code is of interest to the tests themselves, as it mostly just regards register contents being written into memory. After the trap handler is finished, execution continues at the `write_tohost` label, where essentially an endless loop is run with the instructions in lines 6, 7 and 10 respectively.

Listing 3.4: Example of termination of tests in RISCV-DV

```
1  800012dc <test_done>:
2  800012dc:   00100193          li  gp,1
3  800012e0:   00000073          ecall
4
5  800012e4 <write_tohost>:
6  800012e4:   00001f17          auipc   t5,0x1
7  800012e8:   d03f2e23          sw  gp,-740(t5) # 80002000 <tohost>
8
9  800012ec <_exit>:
10 800012ec:   ff9ff06f          j   800012e4 <write_tohost>
11
12 800012f0 <instr_end>:
13 800012f0:   00000013          nop
```

The purpose of this behavior is to use a *Host-Target Interface* (HTIF), which can be used by the target, the core or the VP respectively, to communicate with the host, the system which simulates the core. It is used by the Spike simulator for termination if a value is written to the `tohost` address. Spike periodically checks if this is the case in order to initiate the termination, breaking the endless loop explained above. An emulation of this behavior implemented for RISCV-VP, meaning termination is ensured when running RISCV-DV's tests. For a core at the register-transfer level like MicroRV32 however, this behavior is not intended and thus not implemented.
This raises the need to add a mechanism to automatically terminate the core in a different way after the tests are finished. The intended way to terminate the platform for MicroRV32 is to use its shutdown peripheral by using a similar method, but this would require modifying the code of the tests before every iteration. A more convenient way is to modify the state machine of MicroRV32 according to the behavior shown in listing 3.4. As there are no other `ECALL` instructions present in the generated tests and all regular instructions after the `ECALL` are not relevant to the tests, the `ECALL` instruction can be intercepted and used to halt the core.
To implement this for MicroRV32, the transition of the state machine is adjusted as shown in listing 3.5. Normally the state machine transitions from the `stateTrap` state to the `stateFetch` state as shown in figure 2.3. If the core is run in `dv_mode` however, execution is halted, as the first `ECALL` instruction indicates that the tests are done. This is also one of the uses for the `dv_mode`

argument provided to MicroRV32.

Listing 3.5: Trap handler of MicroRV32 suited for RISCV-DV

```
1        // traphandler and ecall state
2        val stateTrap : State = new State{
3            whenIsActive{
4                if(dv_mode)
5                goto(stateHalt)
6                else
7                goto(stateFetch)
8            }
9        }
```

These modifications allow RISCV-DV's tests to run on the RTL core in co-simulation with the VP. However, RISCV-VP now executes more instructions on the same tests, because compared to the modifications on the RTL core, the VP does not stop execution after the `ECALL` instruction. To resolve this mismatch, the same behavior is implemented for `ECALL`s in the VP. The modification is similar, but implemented differently, namely when the `SimulationTrap` is caught in the `ISS::run_step` method, which is raised when an `ECALL` instruction is encountered (compare listing 2.2). Upon executing an `ECALL` instruction, a flag indicates the termination of the program after the instruction.

With all of these changes implemented, the first step of the RISCV-DV co-simulation flow now works as intended, as both MicroRV32 and RISCV-VP can run the test ELF files provided by the instruction generator. Additionally, both the RTL core and the VP will execute the exact same number of instructions for the same test program, meaning there will only be mismatches in the tests if there is a bug in either MicroRV32 or RISCV-VP.

3.3 Logging and Translation to Trace Format

In order to realize the remaining three steps of the co-simulation flow (compare figure 3.1), the logging of information and its translation to the RISCV-DV trace CSV format needs to be implemented.

3.3.1 Required Information

As shown in chapter 3.1, RISCV-DV uses a trace CSV format to compare the outputs of the RTL core with the ISS. It contains the following fields which need to be logged from both MicroRV32 and RISCV-VP (adapted from [Risd], section "Appendix"):

- **Program counter:** The current PC at the time the instruction is fetched
- **Instruction name:** A string representing the name of the instruction, e.g. `ADDI`

- **GPR:** The general purpose register which is updated by the instruction and its new values; if no register is updated, this field is empty
- **CSR:** The control and status register which is updated by the instruction and its new values; if no register is updated, this field is empty
- **Binary:** The instruction represented as a hexadecimal number
- **Instruction string:** A string representing the instruction, e.g. `ADDI x3 x0 0x10`
- **Operand:** The operands of the instruction; for the above example this would be `x3, x0, 0x10`

3.3.2 Logging

To collect the information in MicroRV32, the simulation loop needs to be extended to output the information presented in chapter 3.3.1. The data can simply be printed to the standard console output, as it will be redirected to a file specified by RISCV-DV.

Information on the current instruction is set in the `Decode` module MicroRV32's CPU during the `decode` phase. It contains signals for all relevant values of the instruction, like its opcode, source registers and destination registers, as shown in listing 3.6.

Listing 3.6: MicroRV32 `Decode` module data signals

```
1   // data signals
2   val instruction = Reg(Bits(32 bits)) init(0)
3   val instrSwapped = Bits(32 bits)
4   val opcode = Bits(7 bits)
5   val source1 = Bits(5 bits)
6   val source2 = Bits(5 bits)
7   val destination = Bits(5 bits)
8   val immediate = Bits(32 bits)
9   val funct3 = Bits(3 bits)
10  val funct7 = Bits(7 bits)
11  val funct12 = Bits(12 bits)
12  val shamt = Bits(5 bits)
```

By logging information in each iteration of the simulation loop however, the same logging output is generated multiple times, as instructions typically need four clock cycles to be executed. Instead, logging has to take place exactly once for every instruction. To realize this, the `dbgState` signal of the CPU is used, which is only used for debugging purposes and has no relevance to the actual execution. Logging is performed when the `dbgState` has a value of 3, which it is set to when the control path of the core is in the `execute` phase, right after the `decode` phase (compare figure 2.3).

Not all information can be logged in this state however. As the instruction has just been decoded, it is not executed yet, which means that the updated general-purpose register and control and status register values have neither been calculated nor written yet. In total, the structure of the logging in the simulation loop is outlined in listing 3.7. All logging only happens if MicroRV32 is simulated in `dv_mode`. The general information on the instruction is collected when the `dbgState` is 3, while the values of the updated gpr and csr are read two steps later. The exception here is step 2, where no instruction has been executed yet.

Listing 3.7: Structure of logging in MicroRV32 simulation loop

```
1  if (dv_mode && state == 3) {
2      // Collect information from debug module and program counter
3  }
4  dut.clockDomain.waitRisingEdge()
5  sim_steps += 1
6  if (dv_mode && sim_steps != 2 && trace_step+2 == sim_steps) {
7      // Conditionally read values of updated GPR and CSR
8      // Write information to standard console output
9  }
```

The difficulty of collecting information in MicroRV32 varies greatly for the required data. For some information, a single signal simply has to be read, while sometimes a differentiation needs to be made for every possible instruction. To account for this, first the signals from the decode logic presented in listing 3.6 are read and stored. With those values, a distinction of all possible instructions is made in multiple levels of depth.

Listing 3.8 introduces the structure of information collection in MicroRV32 as well as the different levels of depth at which information is collected. The first matching is done with the `opcode` signal, which in some cases already is unique to the instruction, like for `OP_AUIPC`, while in other cases it only specifies the instruction type, e.g. `OP_REGIMM` stands for register-immediate instructions. The next level is usually the `funct3` signal, which differentiates between a group of instructions like `ADDI` and `XORI` for `OP_REGIMM`. Sometimes that is still not enough to uniquely identify an instruction. For instance, `ADD` and `SUB` have the same `funct3` value, so in these cases instructions need to be differentiated by their `funct7` value. An exception to this are the `ECALL`, `EBREAK` and `MRET` instructions, which can be told apart by their `funct12` value instead of their `funct3` value.

Listing 3.8: Structure of information collection in MicroRV32

```
1  // Depth 0: Information which is the same for all instructions
2  opcode match{
3      case OP_REGREG =>
4      // Depth 1: Information specific to register-register instructions
5      funct3 match{
6          case F3_XOR =>
7          // Depth 2: Information specific to XOR instruction
8          case F3_ADD =>
9              if (funct7 == 0)
10             // Depth 3: Information specific to ADD instruction
11             else
12             // Depth 3: Information specific to SUB instruction
13         ...
14     }
15     ...
16 }
```

With this structure defined, the required information is collected as follows:

- The **program counter** is unspecific to the instruction and thus stored at depth 0. One thing to consider here however is that the program counter is automatically incremented by 4 when a new instruction is fetched, meaning the correct PC of the current instruction is present in the PC register at the end of the previous instruction. In order to account for this behavior, the program counter of the last instruction needs to be stored to be output together with the rest of the logged data.

- The **instruction name** is always collected at the deepest level of matching, as it is specific to a single instruction only.

- The number representing the destination **GPR** can simply be stored at depth 0, while the updated value of the GPR is read from the core's register file two simulation steps later as mentioned before.
  Regarding the register name, the standard representation can easily be constructed and the ABI name is realized as a function with a match/case expression.
  Not every instruction updates a GPR, causing no output for the GPR field. Thus the collection of information needs to track if a GPR was updated for an instruction by using a flag `gpr_update` in the code. The flag is set at depth 1, as it relates to the type of instruction, for example register-register instructions always update a GPR, while store instructions never update a GPR.

- For the **CSR** field, the procedure is similar to the GPR field, with a flag `csr_update` to indicate whether a control and status register is updated by the instruction. However, the string representation of the CSR has no standard format, so a match/case expression is necessary for the

name. This also applies to the updated value which is logged later, as each CSR has its own signal.

- The **binary** representation of the instruction can also simply be read at depth 0 from the `instruction` signal of the `Decode` module and only needs to be converted to hexadecimal when output.
- The **instruction** string is not explicitly logged by MicroRV32, as it can simply be constructed from the instruction name and the operands.
- The **operands** are collected step by step using string concatenation at different levels of depth. They can mostly be determined at depth 1. For instance, all register-immediate instructions have the `rd` and `rs1` registers as their operands. In this example, the immediate value differs however, as some instructions use the `immediate` value, while others use `shamt`, the shift amount, as their third operand, meaning this operand needs to be considered at depth 2.

A sample logging output is shown in listing 3.9. The names of the fields are denoted before a colon separates them from their contents. Although not shown here, each line is also preceded by the string "[trace]", which allows the identification of the lines relevant to tracing, as SBT also generates other output on its own.

Listing 3.9: Sample logging output of MicroRV32

```
1 pc:80000298,instr:sub,gpr:a6:80000000,csr:,binary:404f0833,operands:a6,t5,tp
2 pc:8000029c,instr:sub,gpr:a6:80000000,csr:,binary:404f0833,operands:a6,t5,tp
3 pc:800002a0,instr:add,gpr:t3:00000169,csr:,binary:00890e33,operands:t3,s2,s0
4 pc:800002a4,instr:sub,gpr:a6:80000000,csr:,binary:404f0833,operands:a6,t5,tp
```

In the VP, collecting information is much more convenient, as the abstraction to a SystemC model grants the ability to reference signal values directly in C++. Nonetheless there are considerations to be made and challenges that arise.

The collection of information is done at the end of the `ISS::exec_step` method (compare listing 2.2), after the instruction has been completely executed. This has the advantage that the updated register values have been written already, meaning complications like in MicroRV32 cannot occur. RISCV-VP also provides a method to retrieve an instruction's string, therefore no distinction for all instructions is needed, the instruction type is sufficient most of the time. In general, the logging process is very similar to MicroRV32, but the matching does not have to go to the instruction level, but can mostly stop at the instruction type level. One aspect which is a bit subtle is reading the value of the potentially updated CSR. This can only be done once in every call of

`ISS::exec_step`, as the method to read the CSR will throw an exception otherwise. To prevent this, the CSR value needs to be stored in a class member variable the first time it is read. The problem with this is that code related to the functionality of RISCV-VP itself has to be adjusted, so these changes need to be made carefully.
Another challenge is the `ECALL` instruction, as it breaks the control flow of the method and raises an exception, meaning the code for logging is never reached. To circumvent this, the output for the `ECALL` instruction is generated when the exception is caught in the `ISS::run_step` method, where the execution also terminates as explained in chapter 3.2.

The logging output for RISCV-VP looks very similar to that of MicroRV32. The differences are the different register names, as the responsible method returns it in the format `ABI name (standard name)` and the instruction strings which are all uppercase. Additionally, the output has a field for the full instruction string, which MicroRV32 does not.

3.3.3 Translation from Log File to Trace CSV

After the log files are properly generated for core and VP during simulation, they need to be translated to the actual tracing CSV format. This is done via Python scripts in RISCV-DV, which are specific to the ISS and its logging output. They contain two regular expressions which are used to match the logging output, one for the line prefix and one to identify all fields which contain the relevant information. Ultimately, additions only need to be done to one function for each ISS, `read_mrv32_instr` and `read_vp_instr` respectively, which processes a single line of logging output in case it matches the regular expression.

Listing 3.10 shows the implementation of the translation function. The regular expression in line 1 matches with the prefix of the logged lines, while the one in line 2 corresponds to all fields and the information in each logging line. For all groups of the regular expression, a split is made to disregard the prefix, e.g. "pc:". For the GPR and CSR fields, it needs to be found out if there were any updates, which depends on whether there is one colon or there are two, as the register name and value are separated by a colon too. The instruction string, which is not explicitly logged by MicroRV32, is constructed in lines 18 to 22 where the operands are added to the instruction name retrieved earlier in line 8.

Listing 3.10: Translation of MicroRV32 log to trace csv

```
1 LINE_RE = re.compile(r"\[info\] \[trace\]")
2 FIELDS_RE = re.compile(r"(pc:[a-f0-9]+?),(instr:\w*),(gpr:\w*:?[a-f0-9]*?),(csr:\w
      *:?[a-f0-9]*?),(binary:[a-f0-9]+?),(operands:(?:\w*\s?,?)*)")
3 LOGGER = logging.getLogger()
```

```
4
5  def read_mrv32_instr(match_t, full_trace):
6      instr = RiscvInstructionTraceEntry()
7      instr.pc = fields.group(1).split(":")[1].strip()
8      instr_s = fields.group(2).split(":")[1].strip()
9      instr.instr = instr_s
10     gprList = fields.group(3).split(":")
11     csrList = fields.group(4).split(":")
12     if len(gprList)!=2: # gpr update
13         instr.gpr.append(gprList[1].strip() + ":" + gprList[2].strip())
14     if(len(csrList)!=2): # csr update
15         instr.csr.append(csrList[1] + ":" + csrList[2])
16     instr.binary = fields.group(5).split(":")[1].strip()
17     instr.mode = "3"
18     operands = fields.group(6).split(":")[1].strip()
19     if(operands != ""): # operands exist
20         instr_s += " " + operands
21     instr.instr_str = instr_s
22     instr.operand = operands
```

With the script implemented, RISCV-DV will translate every logging line produced by MicroRV32 and convert it to a line in the trace CSV format.

The script for RISCV-VP looks very similar, so it is not further elaborated here. One difference is that the register names in the VP are a bit more difficult to handle, since they are formatted as explained in chapter 3.3.2.
In order to use the scripts, the corresponding functions have to be added to RISCV-DV's `run.py` script, which is executed when running tests and takes care of the translation too.

Now that the logging and its translation to the trace CSV format is implemented, almost all steps of RISCV-DV's co-simulation flow (compare figure 3.1) can be executed, as both MicroRV32 and RISCV-VP can run tests so that trace CSV files are generated. This enables the result comparison, as the tests must simply be run with both ISSs for it to be carried out automatically. The coverage generation however needs to be set up manually.

3.4 Coverage Report for Co-Simulation

RISCV-DV offers the possibility to create a coverage report from the trace CSV of any ISS or its logging output by translating it to the trace CSV format beforehand. More precisely, the coverage information is actually collected by the RTL simulator Questa and output to a binary `ucdb` file. The information is then converted to a human-readable coverage report via the `vcover report` command provided by Questa. To automate this process, the Makefile of RISCV-DV is extended by a target `mrv_cov` to report the coverage of experiments conducted for MicroRV32 and `vp_cov` for RISCV-VP respectively.
An example of a coverage report is outlined in listing 3.11. Although the

actual output is more verbose and presented as a table, this example is sufficient to demonstrate the kind of information it contains. Chapter 4.3 also goes into greater detail on the information present in coverage report. The coverage is generally reported separately for each instruction, the example shows the `ADD` instruction. For each entry there are several coverpoints regarding their operands. A coverpoint is an expression which can take multiple values and has to be covered, for instance the destination register of an instruction. Each coverpoint groups these possible values into bins, which can range from single values like general purpose registers to a collection of values like numbers with a positive sign.
All operations report the occurrence of every register as the first source, second source and destination register if applicable. The number of occurrences for a register to be considered covered can also be defined and defaults to 1. For arithmetic operations, the sign of the values in the operand registers are reported and for logical operations the relation of the two source operands are also grouped in one of four bins; `identical`, `opposite`, `similar` and `different`. Branch operations report whether a branch was taken or not and memory operations report whether their memory address is aligned or not. This information, along with many more coverpoints for different kinds of instructions, enables the analysis of a large range of characteristics of the executed instructions and the quality of the verification process.

Listing 3.11: Outline of a coverage report

```
1  TYPE /riscv_instr_pkg/riscv_instr_cover_group/add_cg
2  covered/total bins: 114/114
3  missing/total bins: 0/114
4  % Hit:              100%/100%
5  Coverpoint cp_rs1
6  covered/total bins: 32/32
7  missing/total bins: 0/32
8  % Hit:              100/100%
9  bin auto[ZERO]   242 Covered
10 bin auto[RA]     271 Covered
11 ...
12 Coverpoint cp_rs2
13 ...
14 Coverpoint cp_rd
15 ...
16 Coverpoint cp_rs1_sign
17 ...
18 ...
```

Since all steps of RISCV-DV's co-simulation flow are now implemented, the first goal of this thesis is completed. The flow can be used to verify MicroRV32 with RISCV-VP as the reference ISS. Additionally, a coverage report can be created that provides large amounts of data and grants insight into the quality of the conducted testing.

Chapter 4

Experimental Evaluation

The second and third goal of this thesis are explored in this chapter. The co-simulation flow is executed in order to verify MicroRV32. Although RISCV-VP's purpose is to serve as the reference ISS, there still might be cases in which a mismatch reported by a RISCV-DV test can be ascribed to an error in the implementation of the VP. During the verification process, coverage reports are generated that are assessed in this chapter.

For the **riscv_arithmetic_basic_test**, **riscv_jump_stress_test** and **riscv_loop_test** 100 tests are performed each, with an average of approximately 1250, 850 and 500 instructions respectively. The **riscv_unaligned_load_store_test** consistently reports multiple mismatches due to a bug in MicroRV32 explained below, while the other three tests do not report mismatches in any of their runs.

4.1 Bugs in MicroRV32

The following bugs in MicroRV32 were discovered during test execution:

Unaligned load and store operations

RISCV-DV reports mismatches in every run of the `riscv_unaligned_load_store_test`, which implies that these unaligned memory instructions are not implemented correctly in the core.
Pages 24 and 25 of the RISC-V specification [WA19a] define the ways unaligned load and store instructions may be handled by a RISC-V implementation. The simplest way of handling these instructions is to raise an address-misaligned exception, which is then dealt with in the trap handler. A different approach is to support unaligned load and store instructions by handling them invisibly, i.e. in the hardware implementation. This may be connected to a longer execution time of the instruction however.

In its current state, MicroRV32 does not support unaligned load and store instruction at all, which explains the mismatches reported by RISCV-DV. The unaligned memory instructions eventually lead to different values in a register, as RISCV-VP supports the invisible handling of these instructions.

To fix this bug, unaligned memory instructions need to be supported correctly for MicroRV32. As the specification offers several ways to do this, the same method as in RISCV-VP should be used in order to be able to keep the VP as a reference model for the verification of the core.

Control and status register instructions

Another instruction class which causes issues across all tests if not handled correctly are the control and status register instructions. Although these can be disabled in the generator options, the header which is executed before the actual testing starts already contains CSR instructions and reads the values of several control and status registers.

Upon investigating this bug, it becomes clear that although RISCV-DV reports mismatches between the ISSs, both are working correctly. This can be explained by the purpose the respective CSRs have in RISC-V. One of the registers which is read at the beginning of each test is the `misa` CSR for instance. It specifies which ISA is supported by the ISS. These obviously differ for core and VP, as MicroRV32 only implements the base RV32I ISA, while RISCV-VP supports many ISA extensions.

An easy way to avoid mismatches is simply ignoring outputs of the CSR instructions, as they don't contribute to the tests themselves. This can be done by always writing a constant value, e.g. 0, as the GPR value in the trace CSV, which can be implemented by simply ignoring the actual GPR value and replacing it in the translation script of each ISS. A more thorough approach is to adjust either ISS to hold the CSR value the other has. The easiest way to realize this is to write different values to the respective CSRs when initializing RISCV-VP in DV mode. The `misa` CSR value could then be set to actually reflect the capabilities of MicroRV32, which avoids any mismatches when running the tests while not requiring any "hacks" in the translation script.

4.2 Bugs in RISCV-VP

The following bugs in RISCV-VP were discovered during test execution:

Classification of immediate shift operations

In the module belonging to the instructions of the VP, a method is provided which returns the type of an instruction, e.g. R-Type for the `ADD` instruction. This method incorrectly classifies the `SLLI`, `SLRI` and `SRAI` instructions as being R-Type operations. This is inaccurate, as the immediate instructions use the I-Type instead. Although this bug does not directly affect RISCV-VP's functionality, it could cause problems both if used for verification purposes like this one or if it may be needed for the VP's logic for any purpose in the future.

4.3 Coverage of Conducted Experiments

A collective coverage report was generated for all conducted experiments explained at the beginning of this chapter. In this section, an overview of the functional coverage will be given before the collected information is described in more detail based on the example of the `ADD` instruction.

4.3.1 Overview of coverage for all instructions

An overview of the functional coverage of all conducted experiments is provided in table 4.1. The coverage is presented separately for all instructions, as it is done in the coverage report itself. The exception here are the CSR instructions, as they were explicitly not part of the tests apart from some instructions mandatory in RISCV-DV's generated tests.
For each instruction, the total coverage is listed, as well as some of the coverpoints present for most instructions. These include the two source registers, `rs1` and `rs2`, the destination register `rd`. For logical operations, the `logical` coverpoint is of importance, which puts the values of the two source operands into relation. Each instruction which uses the `rd` register also has a covergroup `gpr_hazards`, which reports hazards on the `rd` register that might be relevant to architectures which use pipelining. If a coverpoint is not relevant to a given instruction, its value is left out as indicated by a hyphen.

Looking at the complete coverage report, the functional coverage is very high in all metrics due to the large volume of the tests. As can be seen in table 4.1, the total coverage amounts to 100% for many instructions. Note that even though the required number of hits for most metrics is only one, the actual numbers are considerably higher. The instructions that show a coverage of 0% are the load and store instructions which do not operate on complete data words. This is due to the fact that these instructions were disabled for all conducted tests because of MicroRV32's bug in dealing with unaligned memory

Table 4.1: Overview of coverage for the conducted tests

Instruction	Total	rs1	rs2	rd	logical	gpr_hazard
ADD	100%	100%	100%	100%	-	100%
SUB	100%	100%	100%	100%	-	100%
ADDI	100%	100%	-	100%	-	100%
LUI	91.66%	-	-	100%	-	75%
AUIPC	91.66%	-	-	100%	-	75%
SRA	100%	100%	100%	100	-	100%
SLL	100%	100%	100%	100%	-	100%
SRL	100%	100%	100%	100%	-	100%
SRAI	100%	100%	-	100%	-	100%
SLLI	100%	100%	-	100%	-	100%
SRLI	100%	100%	-	100%	-	100%
XOR	100%	100%	100%	100%	100%	100%
OR	100%	100%	100%	100%	100%	100%
AND	100%	100%	100%	100%	100%	100%
XORI	100%	100%	-	100%	100%	100%
ORI	96.87%	100%	-	100%	75%	100%
ANDI	96.87%	100%	-	100%	75%	100%
SLT	100%	100%	100%	100%	-	100%
SLTU	100%	100%	100%	100%	-	100%
SLTI	100%	100%	-	100%	-	100%
SLTIU	100%	100%	-	100%	-	100%
BEQ	100%	100%	-	100%	-	100%
BNE	100%	100%	-	100%	-	100%
BLT	100%	100%	-	100%	-	100%
BGE	100%	100%	-	100%	-	100%
BLTU	100%	100%	-	100%	-	100%
BGEU	100%	100%	-	100%	-	100%
LB	0%	0%	-	0%	-	0%
LH	0%	0%	-	0%	-	0%
LW	66.63%	93.54%	-	81.25%	-	75%
LBU	0%	0%	-	0%	-	0%
LHU	0%	0%	-	0%	-	0%
SB	0%	0%	0%	-	-	0%
SH	0%	0%	0%	-	-	0%
SW	68.02%	93.54%	81.25%	-	-	100%
JAL	71.87%	-	-	87.50%	-	-
JALR	88.88%	100%	-	100%	-	-

instructions explained above.

The instructions with a total coverage between 0% and 100% do not reach a complete coverage for different reasons.
For the **LUI** and **AUIPC** instructions, the read-after-write hazard did not occur, which can be explained by the fact that they do not read the value of a register.
The **ORI** and **ANDI** instructions do not cover the `logical` coverpoint since none of their occurrences had two source operands with opposite values.
As none of the conducted tests explicitly target the **LW** and **SW** instructions, their coverage values lack due to a variety of factors. For instance, not all general purpose registers are used as source or target registers respectively. Some coverpoints also regard the alignment of the memory addresses, where unaligned addresses are left out again.
The **JAL** and **JALR** also show the same characteristics with the unaligned memory instructions, as well as not all general purpose registers being covered.

Overall, the high functional coverage of the generated tests shows that this flow and the chosen test strategies support a high number of possible bugs in a RISC-V implementation.

4.3.2 Detailed coverage for the `ADD` instruction

Table 4.2 shows an extract of the coverage for the `ADD` instruction from the report. As the complete coverage information for this instruction alone already spans 160 lines in the coverage report, only parts are shown and explained here. The complete coverage for the `ADD` instruction in Table A.1 in the Appendix.

For the `rs1`, `rs2` and `rd` registers, the coverage is above 100 hits for all 32 general purpose registers. However, the exact numbers partly differ greatly. As can be seen in the table, for the `rs1` coverpoint, i.e. the first source register, the `GP` register was hit 689 times, while the `S0` was only hit 111 times, which are the highest and lowest number of hits respectively. The values for most registers range from 200 to 300 hits, a pattern which can also be observed for `rs2` and `rd`. Despite some larger differences in the number of hits, the coverage shows that the operand registers chosen in RISCV-DV's tests are distributed evenly and no register is neglected or even left out in the conducted experiments.

Table 4.2: Extract of coverage for the `ADD` instruction

Covergroup	Metric	Goal
TYPE /riscv_instr_pkg/riscv_instr_cover_group/add_cg 100.00%	100	
Coverpoint cp_rs1	100.00%	100
bin auto[ZERO]	413	1
bin auto[RA]	302	1
bin auto[SP]	483	1
bin auto[GP]	689	1
bin auto[TP]	203	1
bin auto[T0]	183	1
bin auto[T1]	199	1
bin auto[T2]	445	1
bin auto[S0]	111	1
bin auto[S1]	234	1
...	...	...
Coverpoint cp_rs2	100.00%	100
...	...	...
bin auto[A0]	341	1
bin auto[A1]	325	1
bin auto[A2]	282	1
bin auto[A3]	297	1
bin auto[A4]	227	1
bin auto[A5]	258	1
bin auto[A6]	240	1
bin auto[A7]	207	1
bin auto[S2]	389	1
bin auto[S3]	332	1
bin auto[S4]	159	1
...	...	...
Coverpoint cp_rd	100.00%	100
...	...	...
bin auto[S5]	202	1
bin auto[S6]	295	1
bin auto[S7]	238	1
bin auto[S8]	321	1
bin auto[S9]	290	1
bin auto[S10]	226	1
bin auto[S11]	226	1
bin auto[T3]	172	1
bin auto[T4]	173	1
bin auto[T5]	191	1
bin auto[T6]	518	1
Coverpoint cp_rs1_sign	100.00%	100
bin auto[POSITIVE]	4830	1
bin auto[NEGATIVE]	4276	1
...	...	...
Coverpoint cp_gpr_hazard	100.00%	100
bin auto[NO_HAZARD]	7451	1
bin auto[RAW_HAZARD]	630	1
bin auto[WAR_HAZARD]	263	1
bin auto[WAW_HAZARD]	762	1
Cross cp_sign_cross	100.00%	100
bin ⟨auto[NEGATIVE],auto[NEGATIVE],auto[NEGATIVE]⟩	754	1
bin ⟨auto[POSITIVE],auto[NEGATIVE],auto[NEGATIVE]⟩	1898	1
bin ⟨auto[NEGATIVE],auto[POSITIVE],auto[NEGATIVE]⟩	1917	1
...	...	...

RISCV-DV also distributes the signs of all operands evenly. Table 4.2 shows that the value in the `rs1` register was positive in 4830 and negative in 4276 cases across all `ADD` instructions. Similar values can be observed for `rs2`, while the values in the `rd` register, namely the results of the instructions, were negative in more cases.

Another coverpoint are the hazards occurring for general purpose registers between instructions. The report shows that most of the time there are no hazards, which is true for other instructions as well. However, all potential hazards that could appear; read-after-write, write-after-read and write-after-write, do in fact occur in the tests, meaning potential bugs regarding those hazards should have been reported.

As seen in table 4.2, the coverage report also contains information on the combination of hits, for example the sign cross. It illustrates the combinations of different signs that the values of the `rs1`, `rs2` and `rd` registers can take are covered. As an example from the table, for 1917 `ADD` instructions, the sign of the value in the `rs1` register was negative, the sign of the value in the `rs2` register was positive and the sign of the value in the `rd` register was negative as well. This shows that the tests not only cover all signs of the different operand registers, but also cover all different combinations, which gives insight into whether the hardware is able to deal with these sign combinations in their arithmetic units.

Looking at the coverage of instructions in more detail based on the example of the `ADD` instruction shows that not only the relative coverage of the conducted experiments is very high, with most instructions reaching 100%. The absolute values are also very large, although they differ greatly for some coverpoints. This gives even more confidence that the co-simulation flow provides a high quality of verification.

Chapter 5

Discussion & Future Work

With the co-simulation flow completely implemented and put to use, this chapter focuses on the final goal of this thesis. First, the co-simulation is assessed in the context of how it serves as a guideline for similar efforts. Then, possible extensions and necessary additions are discussed.

5.1 Assessment of the Co-Simulation Flow

The additions and modifications presented in this thesis show that the steps required to integrate a RISC-V implementation into the co-simulation flow are similar for all ISSs. To integrate another ISS with RISCV-DV, it first needs the ability to run the provided ELF files, if that is not already the case. Then, the most elaborate task is collecting all the required information and output it at the correct times. In the end, the CSV translation script has to be set up, which can however easily be adapted from other ISSs in case a similar logging format is used. In this sense, chapter 3 of this thesis can serve as a guide to adding an ISS to RISCV-DV.

Chapter 4 shows that the co-simulation flow is effective in finding bugs in the implementation of an RTL core. Although the volume of bugs is not as large, the flow serves as a good verification option, since RISCV-DV takes care of the complete verification process without the need to come up with an own strategy. The functional coverage report which can be generated without much extra effort ensures the quality of the verification and enables the adjustment of the testing strategy to enhance the process. The framework is very universal, as all RISC-V implementations that can be simulated can also be verified with the co-simulation flow of RISCV-DV.

5.2 Coverage-Based Instruction Generation

As mentioned before, the ability to generate coverage reports enables the assessment of the quality of the verification. In order to increase this quality, coverage metrics could be used to influence the instruction generation done

by RISCV-DV itself. Similar efforts have been discussed before, for instance the so-called *Coverage-Based Fuzzing Method* (CGF), which has also been used for the verification of RISCV-VP in [Her+19].
The technique originates from the software world, where it is surprisingly effective for its relatively simple approach [Kle+18]. Translated to the hardware world, its idea is to create random inputs to the device under test, which are the instructions in the case of the co-simulation. Instead of completely randomizing the inputs however, as RISCV-DV can do with its `riscv_rand_instr_test`, there are fuzzing approaches which mutate the randomly generated inputs, for instance to make them valid instructions and only make the operands random. In coverage-based fuzzing, these mutations are applied according to the coverage the verification process has reached so far, for instance to increase the probability for registers to occur as operands which have not been adequately covered yet.

In order to realize such an approach for the co-simulation flow, the instruction generation of RISCV-DV would have to be adjusted. The tool already offers a variety of instruction streams which are responsible for the generation of different kinds of instructions, so they would have to be modified, as is also mentioned in RISCV-DV's documentation [Risd].
Without going into too much detail on such efforts, as this is just meant to be a proposition of what can be done with the co-simulation flow, the feasibility of using coverage-based fuzzing with RISCV-DV would have to be assessed first. The main advantage would be the re-usability, as once the fuzzing is implemented, all that needs to be done to support another ISS is to realize the co-simulation flow for that RISC-V implementation. However, the efforts connected to adjusting the instruction generator would also have to be evaluated, as it might be easier to just use established fuzzing tools like libFuzzer [Lib] to get the same results.

5.3 Extension of MicroRV32

Another consideration which might even make adjustments to the flow necessary are possible extensions of MicroRV32. For instance, if anything other than the base ISA is implemented, this needs to be accounted for when collecting the information during the logging process. Thanks to the robustness of the implementation however, possible extensions of MicroRV32 itself can be incorporated into the verification flow without much effort.

The structure of the logging requires no change, unless the control path of the3 RTL core is altered. If new instructions are added, the information collection has to be extended to account for these instructions. As this is already demonstrated for the base ISA, it should also not be a great challenge. Other

changes to the way MicroRV32 operates should not affect the logging.

The co-simulation flow even helps if ISA extensions are implemented, as with relatively low effort it offers a robust way for verification of said implementation. If such extensions are made, it will also be interesting to assess the quality of this flow in terms of both identified bugs and the coverage that can be reached.

Chapter 6

Conclusion

In this thesis, the co-simulation flow of RISCV-DV was proposed to verify a RISC-V implementation on the register-transfer level. It was implemented for the MicroRV32 RTL core, with RISCV-VP as its reference ISS. Testing the flow showed that it is effective in finding bugs in the core and the VP as well, while a high functional coverage of the tests indicates that apart from these bugs MicroRV32 can be considered to work correctly. Additionally, future extensions were proposed in order to extend and improve the solution further.

The first goal was the realization of the co-simulation flow. This was explained in detail in chapter 3, where the process was presented step by step. The realization is successful, as the co-simulation flow is now completely functional and can be used to run any tests. The manner the implementation is structured also allows for easy extension, meaning this goal is fulfilled completely.

With the flow implemented, the second goal was to use it to find bugs in MicroRV32 and RISCV-VP. Even though not many bugs were found, this goal was also achieved and all those that were found were presented and explained in chapter 4.

Another goal discussed and accomplished in that chapter was the collection of functional coverage. It was shown that a high functional coverage is reached by running the tests provided by RISCV-DV and the generated coverage reports contain lots of valuable information to be evaluated and used to improve the quality of the verification process.

Finally, the fourth goal was fulfilled in chapter 5, where the flow was assessed and further improvements and potentially necessary extensions were discussed. It was shown that such future adjustments are possible, making the flow suitable for any efforts that might build upon this thesis.

Overall, this thesis accomplished all the goals that were proposed. The co-simulation flow is suited for the verification on the register-transfer level and is effective at finding bugs while providing a high functional coverage. It can serve as a blueprint for similar efforts and offers a variety of options for additions and improvements in the future.

Appendix A

Complete coverage

Table A.1: Complete coverage report for the `ADD` instruction

Covergroup	Metric	Goal
TYPE /riscv_instr_pkg/riscv_instr_cover_group/add_cg 100.00%	100	
covered/total bins:	114	114
missing/total bins:	0	114
% Hit:	100.00%	100
Coverpoint cp_rs1	100.00%	100
covered/total bins:	32	32
missing/total bins:	0	32
% Hit:	100.00%	100
bin auto[ZERO]	413	1
bin auto[RA]	302	1
bin auto[SP]	483	1
bin auto[GP]	689	1
bin auto[TP]	203	1
bin auto[T0]	183	1
bin auto[T1]	199	1
bin auto[T2]	445	1
bin auto[S0]	111	1
bin auto[S1]	234	1
bin auto[A0]	169	1
bin auto[A1]	199	1
bin auto[A2]	346	1
bin auto[A3]	228	1
bin auto[A4]	292	1
bin auto[A5]	200	1
bin auto[A6]	479	1
bin auto[A7]	234	1
bin auto[S2]	216	1
bin auto[S3]	76	1
bin auto[S4]	243	1
bin auto[S5]	355	1
bin auto[S6]	237	1
bin auto[S7]	251	1
bin auto[S8]	337	1
bin auto[S9]	236	1
bin auto[S10]	158	1

bin auto[S11]	424	1
bin auto[T3]	312	1
bin auto[T4]	289	1
bin auto[T5]	358	1
bin auto[T6]	205	1
Coverpoint cp_rs2	100.00%	100
covered/total bins:	32	32
missing/total bins:	0	32
% Hit:	100.00%	100
bin auto[ZERO]	473	1
bin auto[RA]	423	1
bin auto[SP]	318	1
bin auto[GP]	328	1
bin auto[TP]	304	1
bin auto[T0]	355	1
bin auto[T1]	293	1
bin auto[T2]	259	1
bin auto[S0]	143	1
bin auto[S1]	215	1
bin auto[A0]	341	1
bin auto[A1]	325	1
bin auto[A2]	282	1
bin auto[A3]	297	1
bin auto[A4]	227	1
bin auto[A5]	258	1
bin auto[A6]	240	1
bin auto[A7]	207	1
bin auto[S2]	389	1
bin auto[S3]	332	1
bin auto[S4]	159	1
bin auto[S5]	319	1
bin auto[S6]	334	1
bin auto[S7]	169	1
bin auto[S8]	225	1
bin auto[S9]	373	1
bin auto[S10]	158	1
bin auto[S11]	204	1
bin auto[T3]	328	1
bin auto[T4]	246	1
bin auto[T5]	290	1
bin auto[T6]	292	1
Coverpoint cp_rd	100.00%	100
covered/total bins:	32	32
missing/total bins:	0	32
% Hit:	100.00%	100
bin auto[ZERO]	260	1
bin auto[RA]	324	1
bin auto[SP]	362	1
bin auto[GP]	220	1
bin auto[TP]	241	1
bin auto[T0]	236	1
bin auto[T1]	300	1

bin auto[T2]	239	1
bin auto[S0]	290	1
bin auto[S1]	340	1
bin auto[A0]	323	1
bin auto[A1]	427	1
bin auto[A2]	528	1
bin auto[A3]	181	1
bin auto[A4]	222	1
bin auto[A5]	411	1
bin auto[A6]	163	1
bin auto[A7]	348	1
bin auto[S2]	303	1
bin auto[S3]	180	1
bin auto[S4]	356	1
bin auto[S5]	202	1
bin auto[S6]	295	1
bin auto[S7]	238	1
bin auto[S8]	321	1
bin auto[S9]	290	1
bin auto[S10]	226	1
bin auto[S11]	226	1
bin auto[T3]	172	1
bin auto[T4]	173	1
bin auto[T5]	191	1
bin auto[T6]	518	1
Coverpoint cp_rs1_sign	100.00%	100
covered/total bins:	2	2
missing/total bins:	0	2
% Hit:	100.00%	100
bin auto[POSITIVE]	4830	1
bin auto[NEGATIVE]	4276	1
Coverpoint cp_rs2_sign	100.00%	100
covered/total bins:	2	2
missing/total bins:	0	2
% Hit:	100.00%	100
bin auto[POSITIVE]	4841	1
bin auto[NEGATIVE]	4265	1
Coverpoint cp_rd_sign	100.00%	100
covered/total bins:	2	2
missing/total bins:	0	2
% Hit:	100.00%	100
bin auto[POSITIVE]	4351	1
bin auto[NEGATIVE]	4755	1
Coverpoint cp_gpr_hazard	100.00%	100
covered/total bins:	4	4
missing/total bins:	0	4
% Hit:	100.00%	100
bin auto[NO_HAZARD]	7451	1
bin auto[RAW_HAZARD]	630	1
bin auto[WAR_HAZARD]	263	1
bin auto[WAW_HAZARD]	762	1

Cross cp_sign_cross	100.00%	100
covered/total bins:	8	8
missing/total bins:	0	8
% Hit:	100.00%	100
Auto, Default and User Defined Bins:		
bin ⟨auto[NEGATIVE],auto[NEGATIVE],auto[NEGATIVE]⟩	754	1
bin ⟨auto[POSITIVE],auto[NEGATIVE],auto[NEGATIVE]⟩	1898	1
bin ⟨auto[NEGATIVE],auto[POSITIVE],auto[NEGATIVE]⟩	1917	1
bin ⟨auto[POSITIVE],auto[POSITIVE],auto[NEGATIVE]⟩	186	1
bin ⟨auto[NEGATIVE],auto[NEGATIVE],auto[POSITIVE]⟩	1373	1
bin ⟨auto[POSITIVE],auto[NEGATIVE],auto[POSITIVE]⟩	240	1
bin ⟨auto[NEGATIVE],auto[POSITIVE],auto[POSITIVE]⟩	232	1
bin ⟨auto[POSITIVE],auto[POSITIVE],auto[POSITIVE]⟩	2506	1

Bibliography

[Agr] *Group of Computer Architecture at the University of Bremen*. http://www.informatik.uni-bremen.de/agra/eng/index.php. Accessed 09-09-21.

[AP20] Sallar Ahmadi-Pour. *MicroRV32 - A SpinalHDL based RISC-V RV32I_Zicsr implementation for the use on a Field Programmable Gate Array*. 2020.

[APHD21a] Sallar Ahmadi-Pour, Vladimir Herdt, and Rolf Drechsler. "Constrained Random Verification for RISC-V: Overview, Evaluation and Discussion". In: *MBMV 2021; 24th Workshop*. 2021, pp. 1–8.

[APHD21b] Sallar Ahmadi-Pour, Vladimir Herdt, and Rolf Drechsler. "MicroRV32: An Open Source RISC-V Cross-Level Platform for Education and Research". In: *Proceedings of the Workshop on Design Automation for CPS and IoT*. Destion '21. Nashville, Tennessee: Association for Computing Machinery, 2021, 30–35. ISBN: 9781450383165. DOI: 10.1145/3445034.3460508. URL: https://doi.org/10.1145/3445034.3460508.

[CS03] Edmund M. Clarke and P. A. Subrahmanyam. "Hardware Verification". In: *Encyclopedia of Computer Science*. GBR: John Wiley and Sons Ltd., 2003, 777–780. ISBN: 0470864125.

[Her+18] Vladimir Herdt et al. "Extensible and Configurable RISC-V Based Virtual Prototype". In: *2018 Forum on Specification Design Languages (FDL)*. 2018, pp. 5–16. DOI: 10.1109/FDL.2018.8524047.

[Her+19] Vladimir Herdt et al. "Verifying Instruction Set Simulators using Coverage-guided Fuzzing". In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019.

[Kle+18] George Klees et al. "Evaluating Fuzz Testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, 2123–2138. ISBN: 9781450356930. DOI: 10.1145/3243734.3243804. URL: https://doi.org/10.1145/3243734.3243804.

[Lib] *libFuzzer – a library for coverage-guided fuzz testing.* https://llvm.org/docs/LibFuzzer.html. Accessed 22-08-21.

[Pyt] *Python*. https://www.python.org/. Accessed 10-09-21.

[Que] *Questa Advanced Simulator*. https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/. Accessed 19-08-21.

[Risa] *RISC-V Foundation*. https://riscv.org/. Accessed 09-08-21.

[Risb] *RISC-V Members*. https://riscv.org/members/. Accessed 09-08-21.

[Risc] *RISCV-DV*. https://github.com/google/riscv-dv. Accessed 11-08-21.

[Risd] *RISCV-DV Documentation*. https://htmlpreview.github.io/?https://github.com/google/riscv-dv/blob/master/docs/build/singlehtml/index.html. Accessed 12-08-21.

[Sbt] *Scala Build Tool*. https://www.scala-sbt.org/. Accessed 10-09-21.

[Spia] *Spike RISC-V ISA Simulator*. https://github.com/riscv/riscv-isa-sim. Accessed 11-08-21.

[Spib] *SpinalHDL*. https://github.com/SpinalHDL/SpinalHDL. Accessed 10-08-21.

[Sysa] "IEEE Standard for Standard SystemC Language Reference Manual". In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (2012). DOI: 10.1109/IEEESTD.2012.6134619.

[Sysb] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018). DOI: 10.1109/IEEESTD.2018.8299595.

[Uni] *University of Bremen*. http://www.uni-bremen.de. Accessed 17-10-21.

[WA19a] Andrew Waterman and Kriste Asanović. *The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA*. 2.2. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley. 2019.

[WA19b] Andrew Waterman and Kriste Asanović. *The RISC-V Instruction Set Manual. Volume II: Privileged Architecture*. 1.10. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley. 2019.

[Yam] *YAML Ain't Markup Language*. https://www.syaml.org/. Accessed 10-09-21.