CORE-V Verification Strategy

Author: **Michael Thompson** mike@openhwgroup.org

1 Introduction

This document captures the methods, verification environment architectures and tools used to verify the first two members CORE-V family of RISC-V cores, the CV32E and CV64A.

The OpenHW Group will, together with its Member Companies, execute a complete, industrial grade pre-silicon verification of the first generation of CORE-V IP, the CV32E and CV64A cores, including their execution environment[1]. Experience has shown that “complete” verification requires the application of both dynamic (simulation, FPGA prototyping, emulation) and static (formal) verification techniques. All of these techniques will be applied to both CV32E and CV64A.

1.1 License

1.2 Definition of Terms

CORE-V: A family of RISC-V cores developed by the OpenHW Group. The CV32E and the CV64A are the first two members of that family. The CV32E has two planned variants, the CV32E40P and CV32E40.

Member Company: Also **MemberCo**. A company or organization that signs-on with the OpenHW Group and contributes resources (capital, people, infrastructure, software tools

1 Memory interfaces, Debug&Trace capability, Interrupts, etc.

etc.) to the CORE-V verification project.

AC: Active Contributor. An employee of a Member Company that has been assigned to work on an OpenHW Group project.

ISS: Instruction Set Simulator. A behavioural model of a CPU. An ISS can execute the same code as a real CPU and will produce the same logical results as the real thing. Typically only "ISA visible" state, such as GPRs and CSRs are modelled, and any internal pipelines of the CPU are abstracted away.

ELF: Executable and Linkable Format, is a common standard file format for executable files. The RISC-V GCC toolchain compiles C and/or RISC-V Assembly source files into ELF files.

SDK: Software Developers Toolkit. Also, **riscv-gcc-tool-chain**. A set of software tools used to compile C and/or RISC-V assembler code into an executable format. In the case of the CV32E and CV64A, this includes the supported RISC-V ISA compliant instructions, plus a set of XPULP extended instructions.

Verification Environment: Code, scripts, configuration files and Makefiles used in pre-silicon verification. Typically a testbench is a component of the verification environment, but the terms are often used interchangeably.

Testbench: In UVM verification environments, a testbench is a SystemVerilog module that instantiates the device under test plus the SystemVerilog Interfaces that connect to the environment object. In common usage "testbench" can also have the same meaning as verification environment.

$PROJ_ROOT: Local path of a cloned copy of a GitHub repository. An example to illustrate:

[prompt]$ cd /wrk/greg/openhw
[prompt]$ git clone https://github.com/openhwgroup/core-v-verif

Here $PROJ_ROOT is **/wrk/greg/openhw/core-v-verif**. Note that this is not a required shell variable – its use in this document is merely as a reference point for an absolute path to your working copy.

1.3 Conventions Used in this Document

Prose in this document is captured in Liberation Serif, 12-point font. **Bold** type in that font is used for emphasis. Filenames and filepaths are in Liberation Mono 10-point font coloured indigo as in

`./cv32/README.md`. The shell variable $PROJ_ROOT shall be used to denote the root of the absolute path of your clone of a GitHub repository. For example `$PROJ_ROOT/cv32/tb/core/tb_top.sv`. SystemVerilog class and module names are presented in italics using Liberation Serif, 12-point font, coloured indigo, such as *uvmt_cv32_base_test_c*. All URLs will be underlined, using blue coloured text.

1.4 CORE-V Genealogy

The first two members the OpenHW Group's CORE-V family of RISC-V cores are the CV32E and CV64A. Currently, two variants of the CV32E are defined: the CV32E40P and CV32E40. The OpenHW Group's work builds on several RISC-V open-source projects, particularly the RI5CY and Ariane projects from PULP-Platform. CV32E is a derived of the RI5CY project[2], and CV64A is derived from Ariane[3]. In addition, the verification environment for CORE-V leverages previous work done by lowRISC and others for the Ibex project, which is a fork of the PULP-Platform's zero-riscy core.

This is germane to this discussion because the architecture and implement of the verification environments for both CV32E and CV64A are strongly influenced by the development history of these cores. This is discussed in more detailed in Section 3.

Unless otherwise noted, the "previous generation" verification environments discussed in this document come from one of the following master branches in GitHub:

RI5CY: https://github.com/pulp-platform/riscv/tree/master/tb/core
Ariane: https://github.com/pulp-platform/ariane/tree/master/tb
Ibex: https://github.com/lowRISC/ibex/tree/master/dv

1.5 A Note About EDA Tools

The CORE-V family of cores are open-source, under the terms of the Solderpad Hardware License, Version 2.0. This does not imply that the tools required to develop, verify and implement CORE-V cores are themselves open-source. This applies to both the EDA tools such as simulators, and specific verification components, such as Instruction Set Simulators.

Often asked questions are "which tools does OpenHW support?", or "can I use an open-source simulator to compile/run a CORE-V testbench?". The short answer is that the CORE-V testbenches require the use of IEEE-1800 (2017) or newer SystemVerilog tools and that this almost certainly means

2 Note that CV32E is not a fork of RI5CY. Rather, the GitHub repository https://github.com/pulp-platform/riscv was moved to https://github.com/openhwgroup/core-v-cores.

3 CV64A is not forks of the Ariane. The GitHub repository https://github.com/pulp-platform/ariane was moved to https://github.com/openhwgroup/core-v-cores.

that non-commercial, open-source Verilog and SystemVerilog compiler/simulators will not be able to compile/run a CORE-V testbench.

CORE-V verification projects are intended to meet the needs of Industrial users and will therefore use the tools and methodologies currently in wide-spread industrial use, such as the full SystemVerilog language, UVM-1.2, SVA, plus code, functional and assertion coverage. For these reasons users of CORE-V verification environments will need to have access to commercial simulation and/or formal verification tools.

For historical reasons, the "core" testbench of the CV32E40P does run using Verilator, an open-source software tool which translates a subset of the SystemVerilog language to a C++ or SystemC cycle-accurate behavioural model. Continued support for Verilator will be on a best-effort basis.

The specific EDA SystemVerilog simulators used by OpenHW are Metrics ***dsim*** and Cadence ***Xcelium***, so its a very safe bet that the Makefiles will always support rules to compile/simulate with these tools. Use of other commercial tools is predicated on member interest and support.

2 Verification Planning and Requirements

A key activity of any verification effort is to capture a Verification Plan (aka Test Plan or just testplan). This document is not that. The purpose of a verification plan is to identify what features need to be verified; the success criteria of the feature and the coverage metrics for testing the feature. At the time of this writing the verification plan for the CV32E40P is under active development. It is located in the core-v-verif GitHub repository at https://github.com/openhwgroup/core-v-docs/tree/master/verif/CV32E40P/VerificationPlan.

The Verification Strategy (this document) exists to support the Verification Plan. A trivial example illustrates this point: the CV32E40P verification plan requires that all RV32I instructions be generated and their results checked. Obviously, the testbench needs to have these capabilities and its the purpose of the Verification Strategy document to explain how that is done. Further, an AC will be required to implement the testbench code that supports generation of RV32I instructions and checking of results, and this document defines how testbench and testcase development is done for the OpenHW projects.

The subsections below summarize the specific features of the CV32E40* verification environment as identified in the Verification Plan. It will be updated as the verification plan is completed.

2.1 Base Instruction Set

1. Capability to generate all legal RV32I instructions using all operands.

2. Ability to check status of GPRs after instruction execution.
3. Ability to check side-effects, most notably underflow/overflow after instruction execution.

2.2 Privileged Spec

2.3 XPULP Instruction Extensions

2.4 Custom Circuitry

2.5 Interrupts

2.6 Debug

2.7 RVI-Compliant Interface

3 PULP-Platform Simulation Verification

Before discussing the verification strategy of the CV32E and CV64A, we need to consider the starting point provided to OpenHW by the RI5CY (CV32E) and Ariane (CV64A) cores from PULP-Platform. It is also informative to consider the on-going Ibex project, another open-source RISC-V project derived from the ‘zero-riscy’ PULP-Platform core.

For those without the need or interest to delve into history of these projects, Executive Summary below provides a (very) quick summary. Sub-sections 3.2 and 3.3 review the status of RI5CY and Ariane testbenches in sufficient detail to provide the necessary context for sub-section 4 and 5, which details how the RI5CY and Ariane simulation environments will be migrated to CV32E and CV64A simulation environments.

3.1 Executive Summary

In the case of the CV32E, we have an existing testbench developed for RI5CY. This testbench is useful, but insufficient to execute a complete, industrial grade pre-silicon verification and achieve the goal of 'production ready' RTL. Therefore, a two-pronged approach will be followed whereby the existing RI5CY testbench will be updated to create a CV32E40P "core" testbench. New testcases will be developed for this core testbench in parallel with the development of a single UVM environment capable of supporting the existing RI5CY testcases and fully verifying the CV32E cores. The UVM environment will be loosely based on the verification environment developed for the Ibex core and will also be able to run hand-coded code-segments (programs) such as those developed by the RISC-V Compliance Task Group.

In the case of CV64A, the existing verification environment developed for Ariane is not yet mature enough for OpenHW to use. The recommendation here is to build a UVM environment from scratch for the CV64A. This environment will re-use many of the components developed for the CV32E verification environment, and will have the same ability to run the RISC-V Compliance test-suite.

3.2 RI5CY

The following is a discussion of the verification environment, testbench and testcases developed for RI5CY.

3.2.1 RI5CY Testbench

The verification environment (testbench) for RI5CY is shown in Illustration 1. It is coded entirely in SystemVerilog. The core is instantiated in a wrapper that connects it to a memory model. A set of assertions embedded in the RTL[4] catch things like out-of-range vectors and unknown values on control data. The testbench memory model supports I and D address spaces plus a memory mapped address space for a set of virtual peripherals. The most useful of these is a virtual printer that provides something akin to a "hardware printf" capability such that when the core writes ASCII data to a specific memory location it is written to stdout. In this way, programs running on the core can write human readable messages to terminals and logfiles. Other virtual peripherals include external interrupt generators, a 'perturbation' capability that injects random (legal) cycle delays on the memory bus and test completion flags for the testbench.

4 These assertions are embedded directly in the RTL source code. That is, they are not bound into the RTL from the TB using cross-module references. There does not appear to be an automated mechanism that causes a testcase or regression to fail if one or more of these assertions fire.

3.2.2 RI5CY Testcases

Testcases are written as C and/or RISC-V assembly-language programs which are compiled/linked using a light SDK developed to support these test[5]. The SDK is often referred to as the "toolchain". These testcases are all self-checking. That is, the pass/fail determination is made by the testcase itself as the testbench lacks any real intelligence to find errors. The goal of each testcase is to demonstrate correct functionality of a specific instruction in the ISA. There are no specific testcases targeting features of the core's micro-architecture.

A typical testcase is written using a set of macros similar to `TEST_IMM_OP`[6] as shown below:

```
# instruction under test: addi
#                   result       op1       op2
TEST_IMM_OP(addi, 0x0000000a, 0x00000003, 0x007);
```

This macro expands to:

```
li    x1,   0x00000003;       # x1  = 0x3
addi  x14,  x1,   0x007;      # x14 = x1 + 0x7
li    x29,  0x0000000a;       # x29 = 0xA
bne   x14,  x29,  fail;       # if ([x14] != [x29]) fail
```

Note that the GPRs used by a given macro are fixed. That is, the `TEST_IMM_OP` macro will always use x1, x14 and x29 as destination registers.

5 Derived from the PULP platform SDK.

6 The macro and assembly code shown is for illustrative purposes. The actual macros and testcases are slightly more complex and support debug aids not shown here.

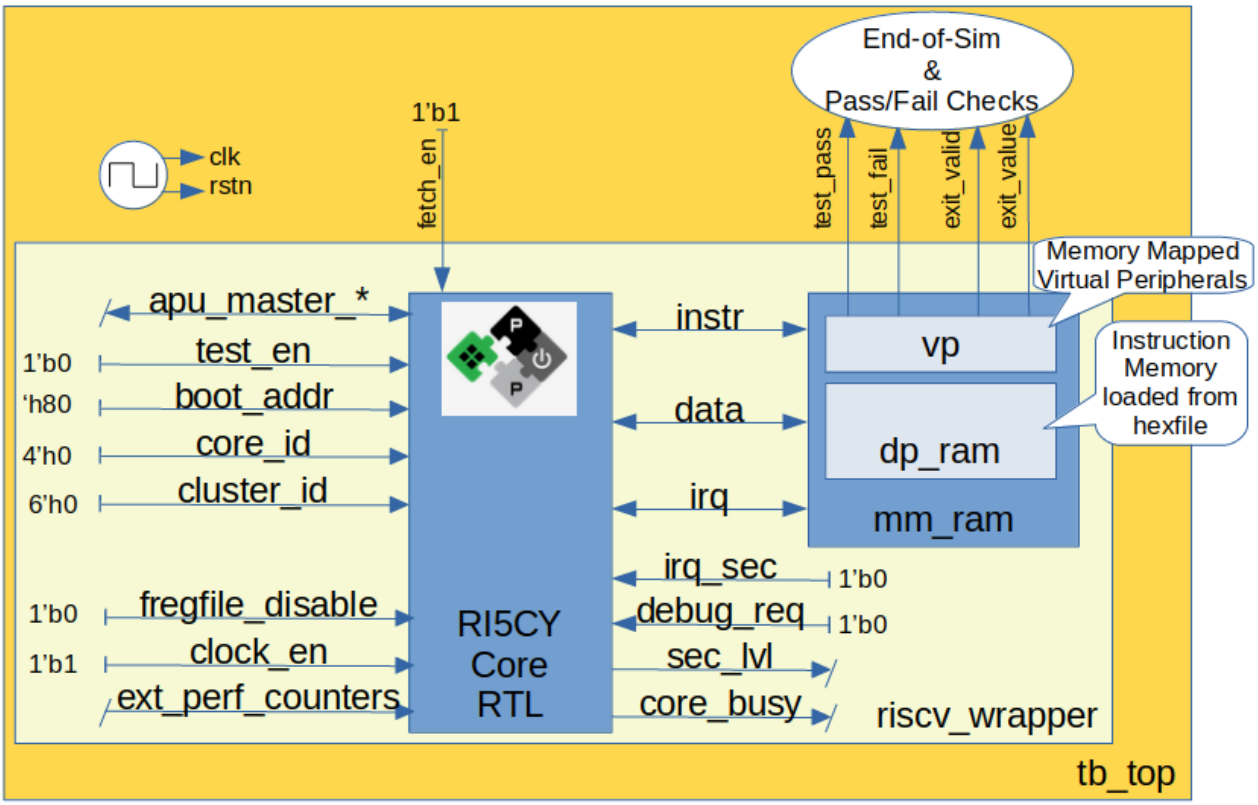

Illustration 1: RI5CY Testbench

The testcases are broadly divided into two categories, riscv_tests and riscv_compliance_tests. In the RI5CY repository these were located in the **tb/core/ riscv_tests** and **tb/core/ riscv_compliance_tests** respectively. In the core-v-verif repository, these can be found at **$PROJ_ROOT/cv32/tests/core/riscv_tests** and **$PROJ_ROOT/cv32/tests/core/riscv_compliance_tests**.

3.2.2.1 RISC-V Tests

This directory has sub-directories for many of the instruction types supported by RISC-V cores. According to the README, only those testcases for integer instructions, compressed instructions and multiple/divide instructions are in active development. It is not clear how much coverage the PULP defined ISA extensions have received.

Each of the sub-directories contains one or more assembly source programs to exercise a given instruction. For example the code segments above were drawn from the **addi.S**[7], a program that

7 **$PROJ_ROOT/cv32/tests/core/riscv_tests/rv64ui/addi.S** in your local copy of the core-v-verif repository.

exercises the *add immediate* instruction. The testcase exercises the addi instruction with a set of 24 calls to `TEST_*` macros as shown above.

There are 217 such tests in the repository. Of these the integer, compressed and multiple/divide instructions total 65 unique tests.

3.2.2.2 RISC-V Compliance Tests

There are 56 assembly language tests in the **riscv_compliance_tests** directory. It appears that that these are a clone of a past version of the RISC-V compliance test-suite.

3.2.2.3 Firmware Tests

There are a small set of C programs in the **firmware** directory. The ability to compile small stand-alone programs in C and run them on a RTL model of the core is a valuable demonstration capability, and will be supported by the CORE-V verification environments. These tests will not be used for actual RTL verification as it is difficult to attribute specific goals such as feature, functional or code coverage to such tests.

3.2.3 Comments and Recommendations for CV32E Verification

The RI5CY verification environment has several attractive attributes:

1. It exists and it runs. The value of a working environment is significant as they all require many person-months of effort to create.
2. It is simple and straightforward.
3. The ‘perturbation’ virtual peripheral is a clever idea that will significantly increase coverage and increase the probability of finding corner-case bugs.
4. Software developers that are familiar with RISC-V assembler and its associated tool-chain can develop testcases for it with little or no ramp-up time.
5. Any testcase developed for the RI5CY verification environment can run on real hardware with only minor modification (maybe none).
6. It runs with Verilator, an open-source SystemVerilog simulator. This is not a requirement for the OpenHW Group or its member companies, but it may be an attractive feature nonetheless.

Having said that the RI5CY verification environment has several shortcomings:

i. All of the intelligence is in the testcases. A consequence of this is that achieving full coverage of the core will require a significant amount of testcase writing.
ii. All testcase are directed-tests. That is, they are the same every time they run. By definition only the stimulus we think about will be run and only the bugs we can imagine will be found. Experience shows that this is a high-risk approach to functional verification.

iii. Testcases focuses on only ISA with no attention paid to micro-architecture features and non-core features such as interrupts and debug.
iv. Stimulus generation and response checking is 100% manual.
v. The performance counters are not verified.
vi. The FPU is not instantiated, so it is not clear if it was ever tested in the context of the core.
vii. All testing is success-based – there are no tests for things such as illegal instructions or incorrectly formatted instructions.
viii. There is no functional coverage model, and code coverage data has not been collected.
ix. Some of the features of the testbench, such as the 'perturbation' virtual peripheral on the memory interface are not used by Verilator as the perturbation model uses SystemVerilog constructs that Verilator does not support.
x. Randomization of the 'perturbation' virtual peripheral on the memory interface is not controllable by a testcase.

So, much work remains to be done, and the effort to scale the existing RI5CY verification environment and testcases to 'production ready' CV32E RTL is not warranted given the shortcomings of the approach taken. It is therefore recommended to replace this verification environment with a UVM compliant environment with the following attributes:

a) Structure modelled after the verification environment used for the low-RISC Ibex core (see Section 3.4 in this document).
b) UVM environment class supporting the complete UVM run-flow and messaging service (logger).
c) Constrained-random stimulus of inst ructions using a UVM sequence-item generator. An example is the Google RISC-V instruction generator.
d) Prediction of execution results using a reference model built into the environment, not the individual testcases. Imperas has an open-source ISS that could be used for this component.
e) Scoreboarding to compare results from both the reference model and the RTL.
f) Functional coverage and code coverage to ensure complete verification of the core.

Its important to emphasize here that the the goal is to have a single verification environment capable of both compliance testing, using the model developed for the RI5CY verification environment, and constrained-random tests as per a typical UVM environment. Once this capability is in place, the existing RI5CY verification environment will be retired altogether.

Developing such a UVM environment is a significant task that can be expected to require up to six engineer-months of effort to complete. This need not be done by a single AC, so the calendar time to get a UVM environment up and running for the core will be in the order of two to three months. This document outlines a strategy for developing and deploying the UVM environment for CV32E in sub-section 4.

The rationale for undertaking such a task is twofold:

1) A full UVM environment is the shortest path to achieving the goals of the OpenHW Group. A UVM based constrained-stimulus, coverage driven environment is scale-able and will have measurable goals which can be easily tracked so that all member companies can see the effort's status in real-time[8]. The overall effort will be reduced via testcase automation and the probability of finding corner-case bugs will be greatly enhanced.

2) The ability to run processor-driven, self-checking testcases written in assembly or C, maintains the ability to run the compliance test-suite. Also, this scheme is common practice within the RISC-V community and such support will be expected by many users of the verification environment, particularly software developers. Note that such tests can be difficult to debug if the self check indicates an error, but, for a more "mature" core design, such as the CV32E (RI5CY) and CV64A (Ariane) they can provide a useful way to run 'quick-and-dirty' checks of specific core features.

Waiting for two to three months for RI5CY core verification to re-start is not practical given the OpenHW Group goals. Instead, a two-pronged approach which sees new testcases developed for the existing testbench in parallel with the development of the UVM environment is recommended. This is a good approach because it allows CORE-V verification to make early progress. When the CV32E UVM environment exceeds the capability of the RI5CY environment, the bulk of the verification effort will transition to the UVM environment. The RI5CY environment can be maintained as a tool for software developers to try things out, a tool for quick-and-easy bug reproduction and a platform for members of the open-source community restricted to the use of open-source tools.

3.3 Ariane

The verification environment for Ariane is shown in Illustration 2. It is coded entirely in SystemVerilog, using more modern syntax than the RI5CY environment. As such, it is not possible to use an open source SystemVerilog simulator such as Icarus Verilog or Verilator with this core.

The Ariane testbench is much more complex than the RI5CY testbench. It appears that the Ariane project targets an FPGA implementation with several open and closed source peripherals and the testbench supports a verification environment that can be used to exercise the FPGA implementation, including peripherals as well as the Ariane core itself.

8 Anyone with access to GitHub will be able to see the coverage results of CORE-V regressions.

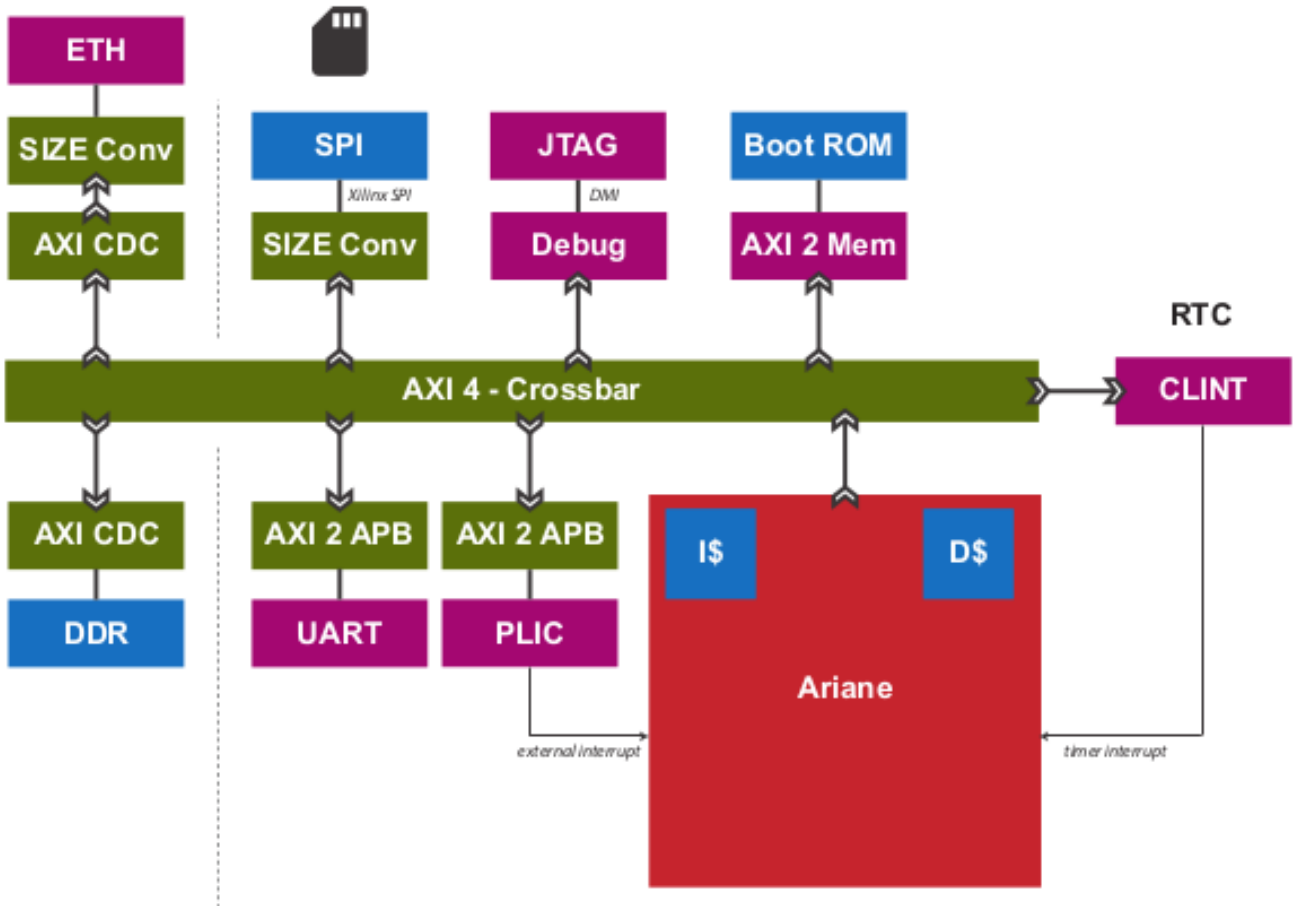

Illustration 2: Ariane Testbench

3.3.1 Ariane Testcases

A quick review of the Ariane development tree in GitHub shows that there are no testcases for the Ariane core. In response to a query to Davide Schiavone, the following information was provided by Florian Zaruba, the current maintainer of Ariane:

> *There are no specific testcases for Ariane. The Ariane environment runs cloned versions of the official RISC-V test-suite in simulation. In addition, Ariane boots Linux on FPGA prototype and also in a multi core configuration.*

So, the (very) good news is that the Ariane core has been subjected to basic verification and extensive exercising in the FPGA prototype. The not-so-good news is that CV64A lacks a good starting point for its verification efforts.

3.3.2 Comments and Recommendations for CV64A Verification

Given that the focus of the Ariane verification environment is based on a specific FPGA implementation that the OpenHW Group is unlikely to use and the lack of a library of existing testcases, it is recommended that a new UVM-based verification environment be developed for CV64A. This would be a core-based verification environment as is envisioned for CV32E and not the mini-SoC environment currently used by Ariane.

At the time of this writing it is not known if the UVM environment envisioned for CV32E can be easily extended for CV64A, thereby allowing a single environment to support both, or completely independent environments for CV32E and CV64A will be required.

3.4 IBEX

Strictly speaking, the Ibex is not a PULP-Platform project. According to the README.md at the Ibex GitHub page, this core was initially developed as part of the PULP platform under the name "Zero-riscy", and was contributed to lowRISC who now maintains and develops it. As of this writing, Ibex is under active development, with on-going code cleanups, feature additions, and verification planned for the future. From a verification perspective, the Ibex core is the most mature of the three cores discussed in this section.

Ibex is not a member of the CORE-V family of cores, and as such the OpenHW Group is not planning to verify this core on its own. However, the Ibex verification environment is the most mature of the three cores discussed here and its structure and implementation is the closest to the UVM constrained-random, coverage driven environment envisioned for CV32E and CV64A.

The documentation associated with the Ibex core is the most mature of the three cores discussed and this is also true for the Ibex verification environment, so it need not be repeated here.

3.4.1 IBEX Impact on CV32E and CV64A Verification

The Ibex verification environment, shown in Illustration 3, is almost, but not quite, a complete end-to-end UVM-based constrained-random, coverage-driven verification environment. The flow of the Ibex environment is very close to what you'd expect: constraints define the instructions in the generated program which is fed to both the device-under-test (Ibex core RTL model) and a reference model (in this case an Instruction Set Simulator provided by Imperas). The resultant output of the RTL and ISS are compared to produce a pass/fail result. Functional coverage (not shown in the Illustration) is applied to measure whether or not the verification goals have been achieved.

As shown in the Illustration, the Ibex verification environment is a set of five distinct processes which are combined together by script-ware to produce the flow above:

1. An SV/UVM simulation of the Instruction Set Generator. This produces a RISC-V assembly program in source format. The program is produced according to a set of input constraints.
2. A compiler that translates the source into an ELF and then to a binary memory image that can be executed directly by the Core and/or ISS.
3. An ISS simulation.
4. A second SV/UVM simulation, this time of the core itself.
5. Once the ISS and RTL complete their simulations, a comparison script is run to check for differences.

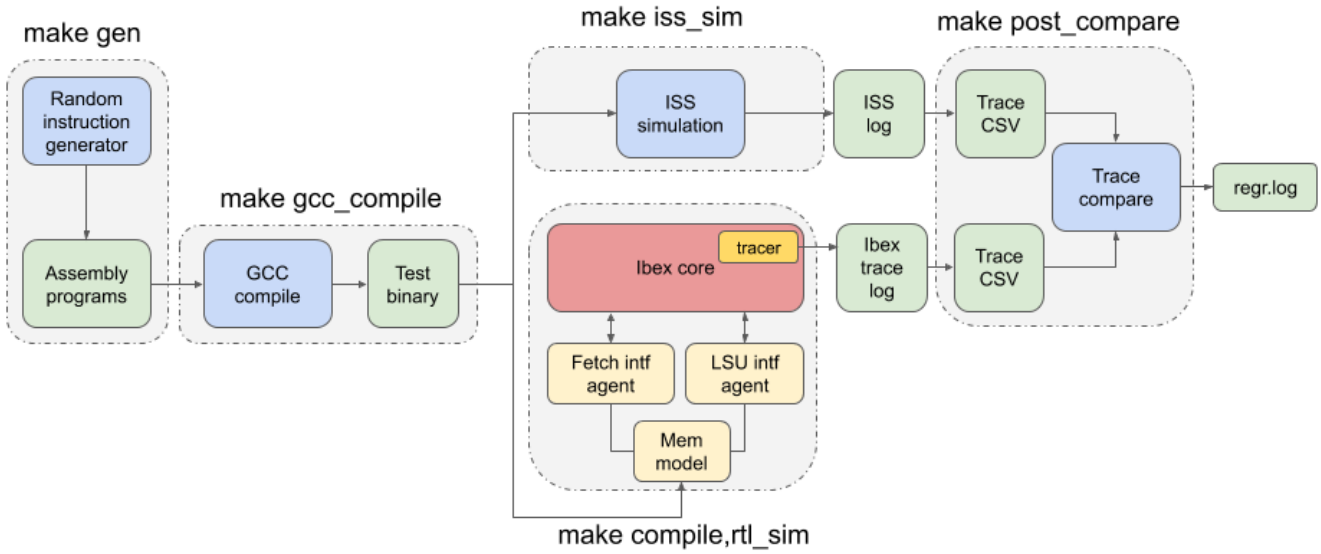

Illustration 3: Ibex Verification Environment

This is an excellent starting point for the CV32E verification environment and our first step shall be to clone the Ibex environment and get it running against the CV32E[9]. Immediately following, an effort will be undertaken to integrate the existing generator, compiler, ISS and RTL into a single UVM verification environment. It is known that the compiler and ISS are coded in C/C++ so these components will be integrated using the SystemVerilog DPI. A new scoreboarding component to compare results from the ISS and RTL models will be required. It is expected that the *uvm_scoreboard*

9 This does not change the recommendation made earlier in this document to continue developing new testcases on the existing RI5CY testbench in parallel.

base class from the UVM library will be sufficient to meet the requirements of the CV32E and CV64A environments with little or no extension.

Refactoring the existing Ibex environment into a single UVM environment as above has many benefits:

- Run-time efficiency. Testcases running in the existing Ibex environment must run to completion, regardless of the pass/fail outcome and regardless of when an error occurs. A typical simulation will terminate after only a few errors (maybe only one) because once the environment has detected a failure it does not need to keep running. This is particularly true for large regressions with lots of long tests and develop/debug cycles. In both cases simulation time is wasted on a simulation that has already failed.
- Easier to debug failing simulations:
  - Informational and error messages can be added in-place and will react at the time an event or error occurs in the simulation.
  - Simulations can be configured to terminate immediately after an error.
- Easier to maintain.
- Integrated testcases with single-point-of-control for all aspects of the simulation.
- Ability to add functional coverage to any point of the simulation, not just instruction generation.
- Ability to add checks/scoreboarding to any point of the RTL, not just the trace output.

4 CV32E40P Simulation Testbench and Environment

As stated in sub-section 3.1, CV32E40P verification will follow a two-pronged approach using an updated RI5CY testbench, hereafter referred to as the core testbench in parallel with the development of a UVM environment. The UVM environment will be developed in a step-wise fashion adding ever more capabilities, and will always maintain the ability to run testcases and regressions.

The UVM environment will be based on the verification environment developed for the Ibex core, using the Google random-instruction generator for stimulus creation, the Imperas ISS for results prediction and will also be able to run hand-coded code-segments (programs) such as those developed by the RISC-V Compliance Task Group.

The end-goal is to have a single UVM-based verification environment capable of complete CV32E40P and CV32E40 verification. This environment will be rolled out in three phases are detailed in sub-section 4.2.

4.1 Core Testbench

The “core” testbench, shown in Illustration 1, on page 18, is essentially the RI5CY testbench with some slight modifications. It is named after the directory is it located in. This testbench has the ability to run the directed, self-checking RISC-V Compliance and XPULP test programs (mostly written in Assembler) used by RISC-V and will be used to update the RISC-V Compliance and add XPULP Compliance testing for the CV32E40P. These tests are the foundation of the Base Instruction Set and XPULP Instruction Extensions captured in the CV32E40P verification plan.

The testbench has been (or will be) modified in the following ways:

1. Fix several Lint errors (Metrics dsim strictly enforces the IEEE-1800 type-checking rules).
2. Update parameters as appropriate.
3. Some RTL files were placed in the core director – these have been moved out.
4. Support UVM error messages.
5. (TBD) Updates to the end-of-simulation flags in the Virtual Peripherals.

As mentioned in A Note About EDA Tools, above, currently this testbench compiles and runs under Verilator. Continued support for Verilator is not assured.

4.2 The CV32E40* UVM Verification Environment

This sub-section discusses the structure and development of the UVM verification environment under development for CV32E40*. This environment is intended to be able to verify the CV32E40P and CV32E40 devices with only minimal modification to the environment itself.

4.2.1 Phase 1 Environment

The goal of the phase 1 environment are to able to execute all of the compliance tests from the RISC-V Foundation, PULP-Platform and OpenHW Group, plus a set of manually written C and assembler testcases in a minimal UVM environment. Essentially, it will have the same functionality as the core testbench, but will all the overhead of the UVM.

Recall from Illustration 1 the structure of the core testbench. Swapping out the RI5CY RTL model for the CV32E40P RTL model, and adding SystemVerilog interfaces yields the testbench components for the phase 1 environment. Rounding out the environment is a minimal UVM environment and UVM base test. This is shown in Illustration 4.

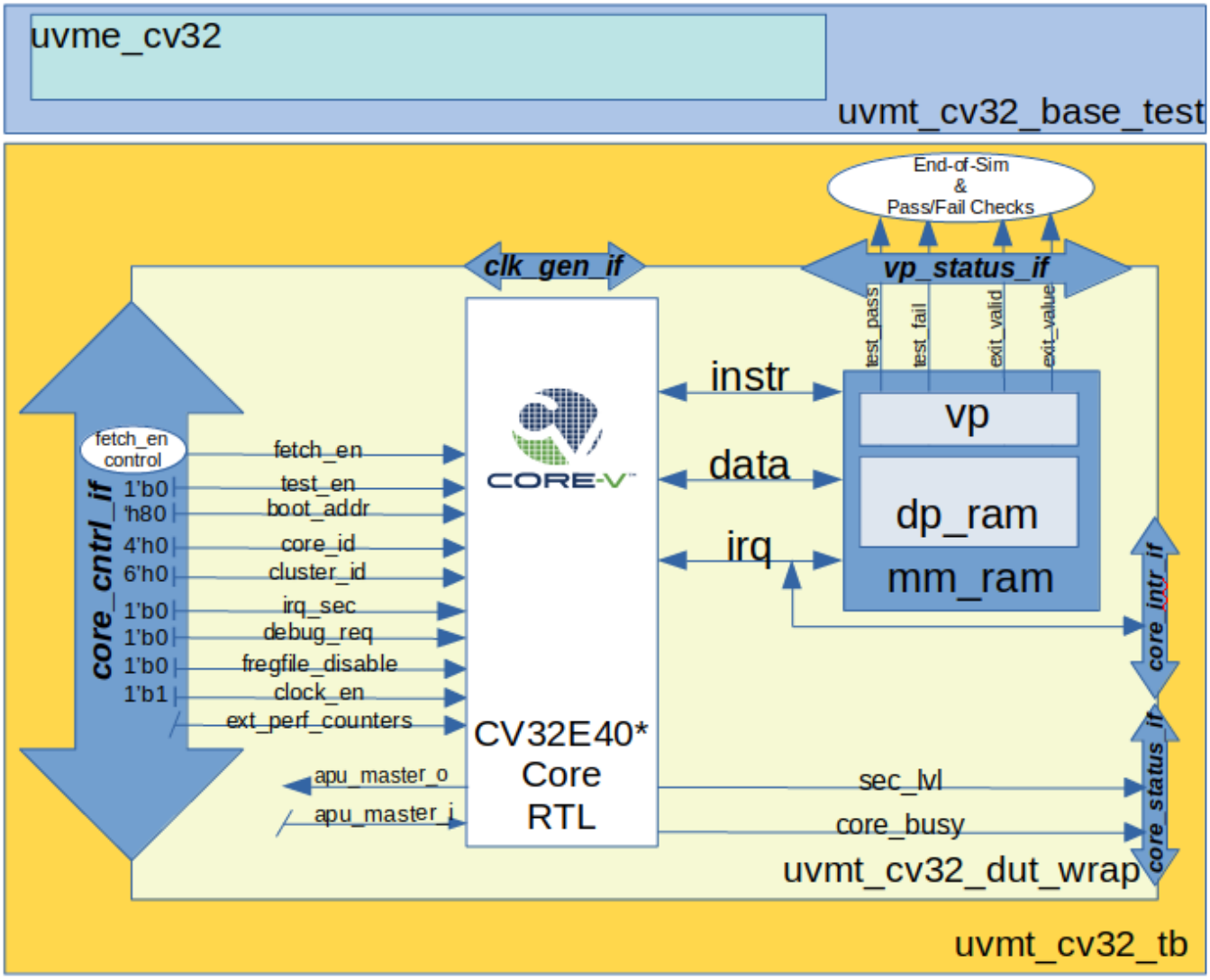

Illustration 4: Phase 1 CV32E40P UVM Environment

The testbench components of the phase 1 environment are the so-called “DUT wrapper” (module uvmt_cv32_dut_wrap) which is a modification of the riscv_wrapper in core testbench, and the “testbench” (module uvmt_cv32_tb) which is a replacement of the tb_top module from the core testbench. This structure provides the UVM environment with access to all of the CV32E40P top-level control and status ports via SystemVerilog interfaces. Note that for phase 1, most of the control inputs are static, just as they are in the core testbench. The phase 2 environment will have dedicated UVM

agents for each of the interfaces shown in Illustration 4, allowing testcases to control these interfaces using UVM test sequences.

The phase 1 environment will also control the function of the riscv-gcc toolchain directly as part of the UVM run-flow, simplifying the Makefiles used to control compilation and execution of testcases.

4.2.2 Phase 2 Environment

The phase two environment is shown in Illustration 5. Phase 2 introduces the Google Random Instruction Generator and the Imperas ISS as a stand-alone components. The most significant capabilities of the phase 2 environment are:

- Ability to use SystemVerilog class constraints to automatically generate testcases.
- Results checking is built into the environment, so that testcases do not need to determine and check their own pass/fail criteria.
- Simple UVM Agents for both the Interrupt and Debug interfaces. ToDo: show this in the Illustration.
- Ability to run any/all testcases developed for the Phase 1 environment.
- Support either of the CV32E40P or CV32E40 with only minor modifications.

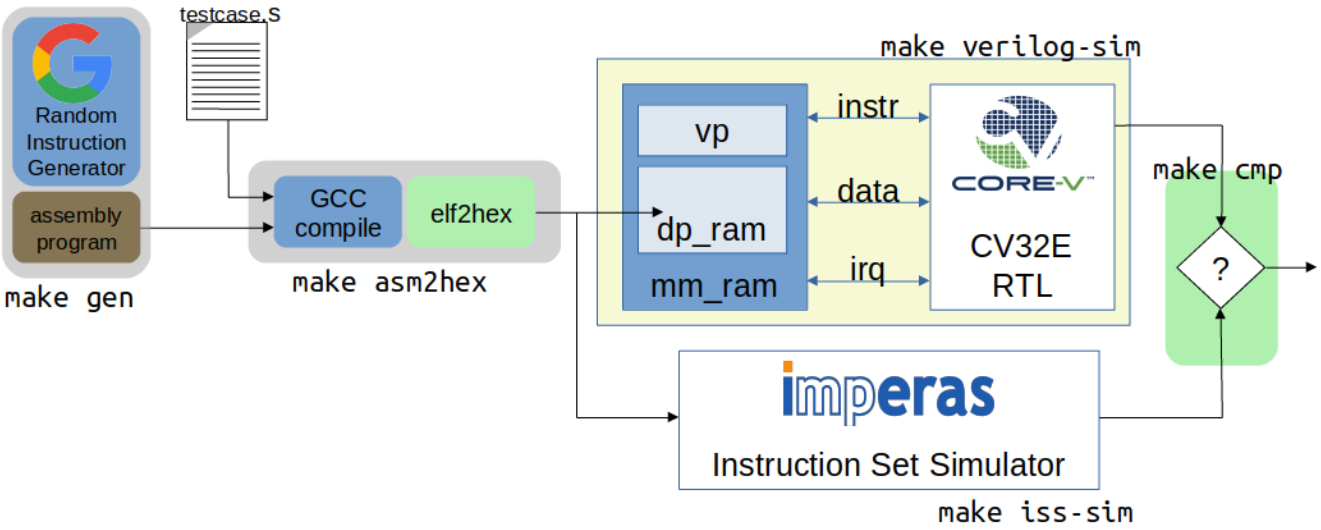

*Illustration 5: Phase 2 Verification Environment for CV32E40**

As shown in the Illustration, the environment is not a single entity. Rather, it is a collection of disjoint components, held together by script-ware to make it appear as a single environment. When the user invokes a command to run a testcase, for example, `make xrun-firmware`[10], a set of scripts and/or Makefile rules are invoked to compile the environment and test(s), run the simulation(s) and check results. The illustration shows the most significant of these:

- **make gen**: this is an optional step for those tests that run stimulus generated by the Google random instruction generator. Tests that use manually generated or externally sourced tests will skip this test. The generator produces an assembly-language file which is used as input to **asm2hex**.
- **make asm2hex**: this step invokes the SDK (riscv-gcc tool-chain) to compile/assemble/link the input program into an ELF file. The input program is either from the **make gen** step or a previously written assembler program. The ELF is translated to a hexfile, in verilog "memh" format, that can be loaded into a SystemVerilog memory.
- **make sv-sim:** this step runs a SystemVerilog simulator that compiles the CV32E and its associated testbench. As with the RI5CY testbench, the asm2hex generated hexfile is loaded into Instruction memory and the core starts to execute the code it finds there. Results are written to an *actual* results output file.
- **make iss-sim**: this step compiles and runs the Instruction Set Simulator simulator, using the same ELF produced in the make asm2hex step. The ISS thereby runs the same program as the RTL model of the core and produces an *expected* result output file.
- **make cmp**: here a simple compare script is run that matches the actual results produced by the RTL with the expected results produced by the ISS. Any mismatch results in a testcase failure.

4.2.3 Phase 2 Development Strategy

The disjoint-component nature of the phase two environment simplifies its development, as almost any component of the environment can be developed, unit-tested and deployed separately, without a significant impact on the other components or on the phase one environment. In addition, the Ibex environment provides a working example for much of the phase two work.

The first step will be to introduce the random-instruction generator into the script-ware. This is seen as a relatively simple task as the generator has been developed as a stand-alone UVM component and has previously been vetted by OpenHW. Once the generator is integrated, user's of the environment will

10 See the README at https://github.com/openhwgroup/core-v-verif/tree/master/cv32/tests/core to see what this does. Note that the User Manual for the Verification Environment, which explains how to write and run testcases, will be maintained there, not in the core-v-docs project which is home for this document.

have the ability to run existing or new testcases for the phase one environment, as well has run generated programs on the RTL. The programs generated by the Google random-instruction generator are not self-checking, so tests run with the generator will not produce a useful pass/fail indication, although they may be used to measure coverage.

In order to get a self-checking environment, the ISS needs to be integrated into the flow. This is explicitly supported by the Google generator, so this is seen as low-risk work. An open issue is to extract execution trace information both the RTL simulation and ISS simulation in such a way as to make the comparison script simple. Ideally, the comparison script would be implemented using ***diff***. This is a significant ToDo.

4.2.4 Phase 3 Environment

ToDo

4.2.5 Phase 3 Development Strategy

ToDo

4.3 File Structure and Organization

ToDo

4.3.1 Naming Convention

4.3.2 Directory and File Structure

4.3.3 Compiling the Environment

5 The CV64A Simulation Verification Environment

ToDo

6 Simulation Tests in the UVM Environments

With the exception of the “core testbench” for CV32E40P, the CORE-V environments are all UVM environments and the overall structure should be familiar to anyone with UVM experience. This section discusses the CORE-V-specific implementation details that affect test execution, and that are important to test writers. It attempts to be generic enough to apply to both the CV32E and CV64A environments.

A unique feature of the CORE-V UVM environments is that a primary source of stimulus, and sometimes the only source of stimulus, comes in the form of a “test program” that is loaded into the testbench’s memory model and then executed by the core itself. The UVM test, environment and agents are often secondary sources of stimulus and sometimes do not provide any stimulus at all. This means it is important to draw a distinction between the “**test program**” which is a set of instructions executed by the core, and the “**UVM test**”, which is a testcase in the UVM sense of the word.

6.1 Test Program

In this context a “test program” is set of RISC-V instructions that are loaded into the testbench memory. The core will starting fetching and executing these instructions when *fetch_en* is asserted. Test programs may be manually produced by a human or by a tool such as the UVM random instructor generator component of the environment. Test programs are coded either in RISC-V assembler or C. All of the randomly generated programs are RISC-V assembler[11].

The environment can support test programs regardless of how they are created. However, the environment needs to know two things about a test program:

- Is the program pre-existing, or does it need to be generated at run-time?
- Is the test program self-checking? That is, can it determine, on its own, the pass/fail criteria of a test program and can it signal this to the testbench?

Section 3.2.2 details how many of the test programs inherited from the RI5CY project are both pre-existing and self-checking. It is expected, but not required, that most of the pre-existing test programs will be self-checking.

Section ToDo introduces the operation of the random instruction generator and how it generates test programs. Here, the situation regarding to self-checking tests is inverted. That is, it is expected, but not required, that most of the generated test programs will **not** be self-checking.

The UVM environment is equipped to support four distinct types of test programs:

1. **Pre-existing, self-checking**
   The environment requires a memory image for the program to exist in the expected location, and will check the “status flags[12]” virtual peripheral for pass/fail information.

11 Those familiar with the RI5CY testbench may recall that random generation of C programs using csmith was supported. Csmith was developed to exercise C compilers, not processors, it is not supported in the CORE-V environments.

12 See Section 6.1.1, below.

2. **Pre-existing, not self-checking**
   The environment requires a memory image for the program to exist in the expected location, and will **not** check the "status flags" virtual peripheral for pass/fail information.
3. **Generated, self-checking**
   The environment will use its random instruction generator to create a test program, and will check the "status flags" virtual peripheral for pass/fail information.
4. **Generated, not self-checking**
   The environment will use its random instruction generator to create a test program, and will **not** check the "status flags" virtual peripheral for pass/fail information.
5. **None**
   It is possible to run a UVM test without running a test program. An example might be a test to access CSRs via the debug module interface interface in debug mode.

Although five types are supported, it is expected that types 1 and 4 will predominate.

Simulations pass/fail outcomes will also be affected by other checkers/monitors that are not part of the status flags virtual peripheral. It is required that any such checkers/monitors shall signal an error condition with `uvm_error(), and these will cause a simulation test to fail, independent of what the test program may or may not write to the status flags virtual peripheral.

It is possible to use an instruction generator to write out a set of test programs, self checking or not, and run these as if they were pre-existing test programs. From the environment's perspective, this indistinguishable from type 1 or type 2.

The programs can be written to execute any legal instruction supported by the core[13]. Programs have access to the full address range supported by the memory model in the testbench plus a small set of memory-mapped "virtual peripherals", see below.

6.1.1 Virtual Peripherals

A SystemVerilog module called *mm_ram* is located at `$PROJ_ROOT/cv32/tb/core/mm_ram.sv`. It connects to the core as shown in Illustration 5. In additional to supporting the instruction and data memory (*dp_ram*) this module implements a set of virtual peripherals by responding to write cycles at specific addresses on the data bus. These virtual peripherals provides the features listed in Table 1.

The printer and status flags virtual peripherals are used in almost every assembler testcase provided by the RISC-V foundation for their ISA compliance test-suite. As such, these virtual peripherals will be

13 Generation of illegal or malformed instructions is also supported, and will be discussed in a later version of this document.

maintained throughout the entire CORE-V verification effort. It is also believed, but not known for certain, that the signature writer is used by several existing testcases, so this peripheral may also be maintained over the long term.

The use of the interrupt timer control and instruction memory stall controller are not well understood and it is possible that none of the testscases inherited from the RISC-V foundation or the PULP-Platform team use them. As such they are likely to be deprecated and their use by new test programs developed for CORE-V is strongly discouraged.

Virtual Peripheral	VP Address (data_addr_i)	Action on Write
Address Range Check	>= 2**16, but not one of the valid VP addresses below.	**Terminate simulation** TODO: make this a `uvm_fatal()
Virtual Printer	32'h1000_0000	$write("%c", wdata[7:0]);
Interrupt Timer Control	32'h1500_0000	timer_irg_mask <= wdata;
	32'h1500_0004	timer_count <= wdata; This starts a timer that counts down each clk cycle. When timer hits 0, an interrupt (irq_o) is asserted.
Virtual Peripheral Status Flags	32'h2000_0000	Assert **test_passed** if wdata==’d123456789 Assert **test_failed** if wdata==’d1 **Note**: asserted for one clk cycle only.
	32'h2000_0004	Assert **exit_valid**; **exit_value** <= wdata; **Note**: asserted for one clk cycle only.
Signature Writer	32'h2000_0008	signature_start_address <= wdata;
	32'h2000_000C	signature_end_address <= wdata;
	32'h2000_0010	Write contents of dp_ram from sig_start_addr to sig_end_addr to the signature file. Signature filename must be provided at run-time using a **+signature=<*sig_file*>** plusarg. Note: this will also asset **exit_valid** with **exit_value** <= 0.
Instruction Memory Interface Stall Control	32'h1600_XXXX	Program a table that introduces “random” stalls on IMEM I/F.

Table 1: List of Virtual Peripherals

6.2 UVM Test

A UVM Test is the top-level object in every UVM environment. That is, the environment object(s) are members of the testcase object, not the other way around. As such, UVM requires that all tests extend from *uvm_test* and the CV32E environment defines a “base test”, *uvmt_cv32_base_test_c*, that is a

direct extension of *uvm_test*. All testcases developed for CV32E should extend from the base test, as doing so ensures that the proper test flow discussed here is maintained (it also frees the test writer from much mundane effort and code duplication). The comment headers in the base test (attempt to) provide sufficient information for the test writer to understand how to extend it for their needs.

A typical UVM test for CORE-V will extend three time consuming tasks:

1. **reset_phase():** often, nothing is done here except to call *super.reset_phase()* which will invoke the default reset sequence (which is a random sequence). Should the test writer wish to, this is where a test-specific reset virtual sequence could be invoked.
2. **configure_phase():** in a typical UVM environment, this is a busy task. However, assuming the program executed the core does so, the core's CSRs do not require any configuration before execution begins. Any test that requires pre-compiled programs to be loaded into instruction memory should do that here.
3. **run_phase():** for most tests, this is where the procedural code for the test will reside. A typical example of the run-flow here would be:
    - Raise an objection;
    - Assert the core's fetch_en input;
    - Wait for the core and/or environment(s) to signal completion;
    - Drop the objection.

6.2.1 Workarounds

The CV32E base test, *uvmt_cv32_base_test_c*, in-lines code (using **`include)** from `uvmt_cv32_base_test_workaround.sv`. This file is a convenient place to put workarounds for defects or incomplete code in either the environment or RTL that will affect all tests. This file must be reviewed before the RTL is frozen, and ideally it will be empty at that time.

6.3 Run-flow in a CORE-V Test

The test program in the CORE-V environment directly impacts the usual run-flow that is familiar to UVM developers. Programs running on the core are completely self-contained within their extremely simple execution environment that is wholly defined by the ISA, memory map supported by the *dp_mem* and the virtual peripherals supported by *mm_mem*[14]. This execution environment knows nothing about the UVM environment, so the CORE-V UVM environments are implemented to be aware of the test program and to respond accordingly as part of the run-flow.

14 This is termed Execution Environment Interface or EEI by the RISC-V ISA.

Section 6.1 introduced the five types of core test programs supported by the CORE UVM environment and section 6.2 showed how the configure_phase() and run_phase() of a CORE-V UVM run-flow implement the interaction between the UVM environment and the test program. This interaction is depends on the type of test program. Illustration 6 shows how the CORE-V UVM base test supports a type 1 test program.

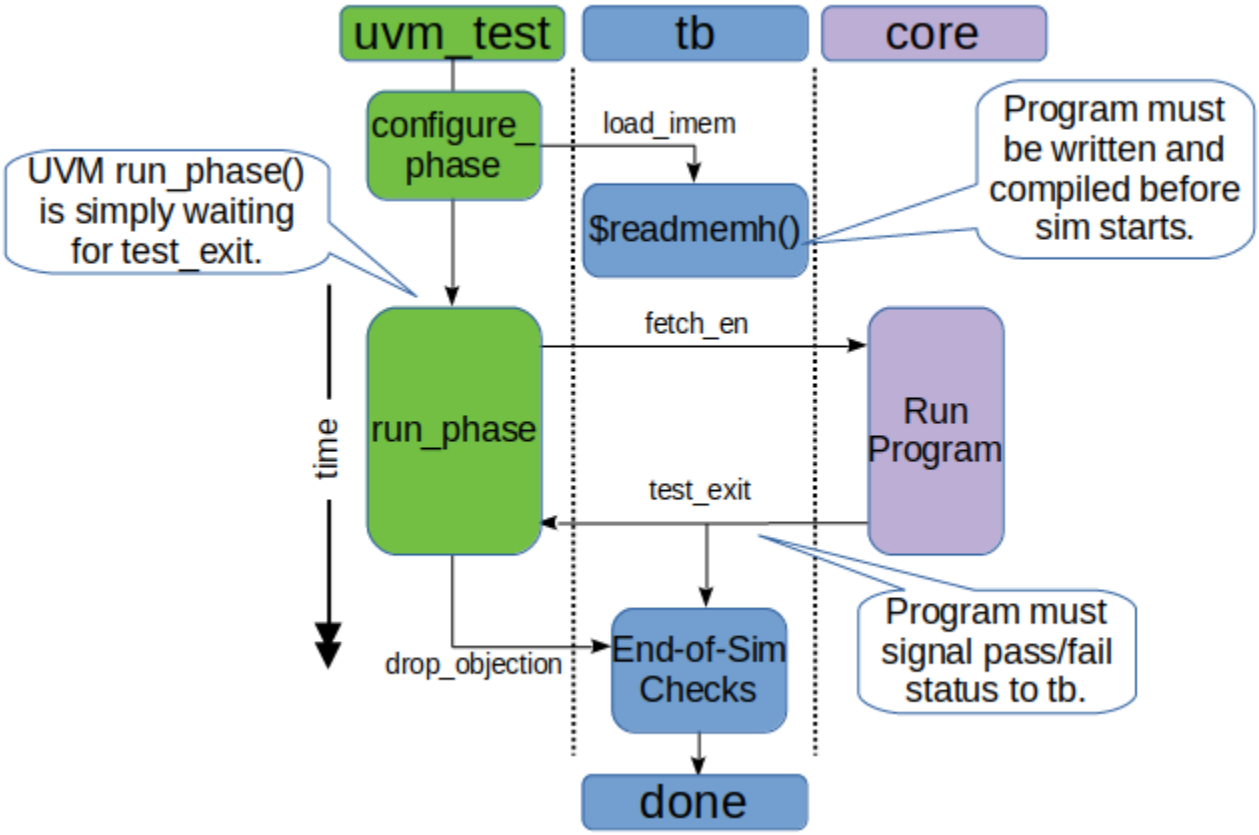

Illustration 6: Preexisting, Self-checking Test Program (type 1) in a CORE-V UVM test

In the self-checking scenario, the testcase is pre-compiled into machine code and loaded into the *dp_ram* using the **$readmemh()** DPI call. The next sub-section explains how to select which test program to run from the command-line. During the configuration phase the test signals the TB to load the memory. The TB assumes the test file already exists and will terminate the simulation if it does not.

In the run phase the base test will assert the fetch_en input to the core which signals it to start running. The timing of this is randomized but keep in mind that it will always happen after reset is de-asserted (because resets are done in the reset phase, which always executes before the run phase).

At this point the run flow will simply wait for the test program to flag that it is done via the status flags virtual peripheral (see Table 1). The test program is also expected to properly assert the test pass or test fail flags. Note that the environment will wait for the test flags to asserts or until the environment's watch dog timer fires. A watch-dog firing will terminate the simulation and is, by definition, a failure.

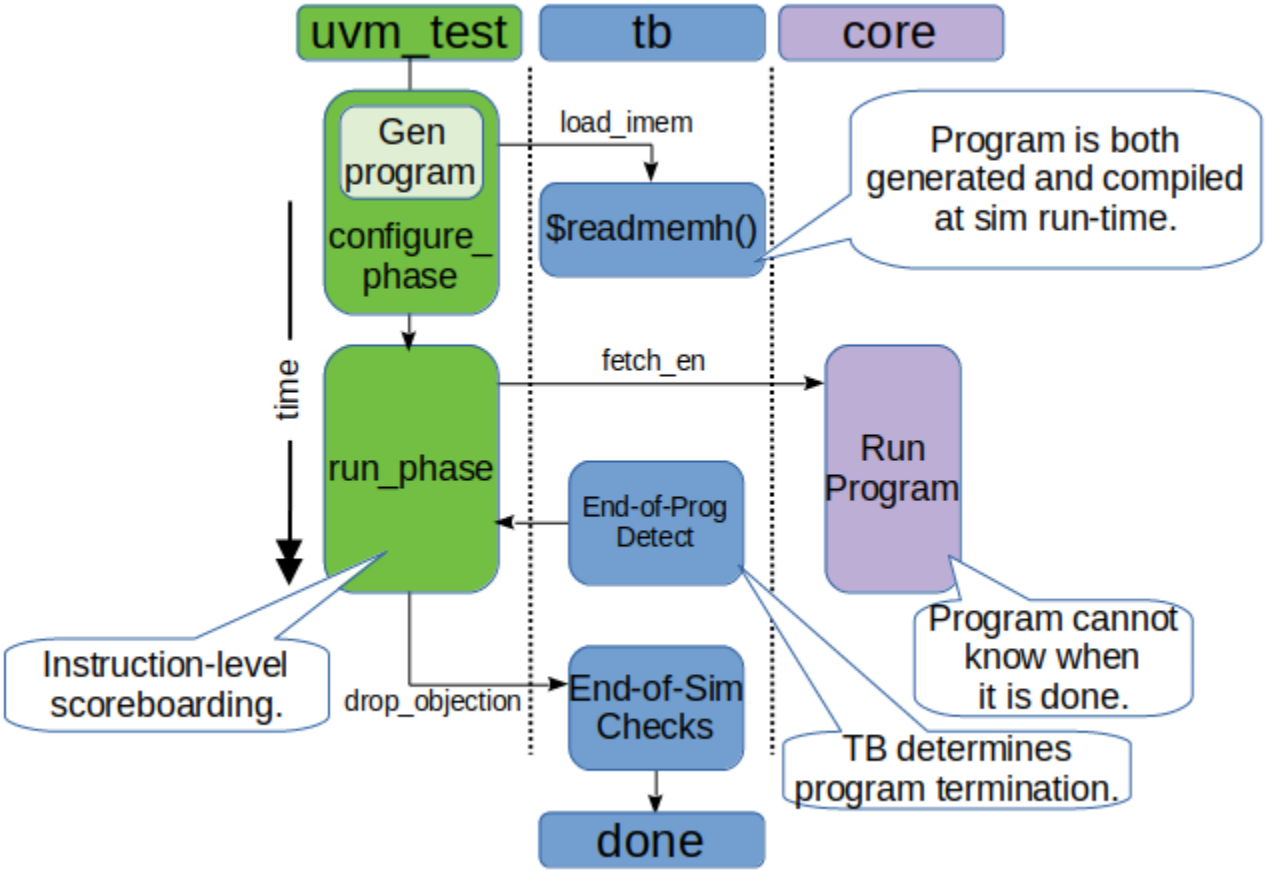

Illustration 7: Generated, non-self-checking (type 4) Test Program in a CORE-V UVM test

The flow for a type 4 (generated, non-self checking) test program is only slightly different as shown in Illustration 7. In these tests the configure phase will invoke the generator to produce a test program and the toolchain to compile it before signalling the TB to load the machine code into *dp_mem*. As before, the run phase will assert fetch_en to the core and the program begins execution.

Recall that a type 4 test program will not use the status flags virtual peripheral to signal test completion. It is therefore up to the UVM environment to detect end of test. This is done when the

various agents in the environment detect a lack of activity on their respective interfaces. The primary way to detect this is via the Instruction-Retire agent (TODO: describe this agent).

In a non-self-checking test program the intelligence to determine pass/fail must come from the environment. In the CORE-V UVM environments this is done by scoreboarding the results of the core execution and those predicted by the ISS as shown in Illustration 5. Note that most UVM tests that run self-checking test programs will also use the ISS as part of its pass/fail determination.

6.4 CORE-V Testcase Writer's Guide

6.4.1 File Structure of the Test Programs and UVM Tests

Below is a somewhat simplified view of the CV32 tests directory tree. The test programs are in cv32/tests/core. (This should probably be cv32/tests/programs, but is named "core" for historical reasons.) Sub-directories below core contain a number of type 1 test programs.

The UVM tests are located at cv32/tests/uvmt_cv32. It is a very good idea to review the code in the base-tests sub-directory. In "core-program-tests" is the type 1 and type 4 testcases (types 2 and 3 may be added at a later date). These ca be used as examples and are also production level tests for either type 1 or type 4 test programs. An up to date description of the testcases under uvmt_cv32 can be found in the associated README.

Lastly, the cv32/tests/vseq directory is where you will be (and should add) virtual sequences for any new testcases you develop.

```
$PROJ_ROOT/
└── cv32/
    └── tests/
        ├── core/
        │   ├── README.md
        │   ├── custom/
        │   │   ├── hello_world.c
        │   │   └── <etc>
        │   ├── riscv_compliance_tests_firmware/
        │   │   ├── addi.S
        │   │   └── <etc>
        │   ├── riscv_tests_firmware/
        │   │   └── <etc>
        │   └── firmware/
        │       └── <etc>
        └── uvmt_cv32/
            ├── base-tests/
            │   ├── uvmt_cv32_base_test.sv
            │   ├── uvmt_cv32_base_test_workarounds.sv
```

```
│   └── uvmt_cv32_test_cfg.sv
├── core-program-tests/
│   ├── README.md
│   ├── uvmt_cv32_type1_test.sv
│   └── uvmt_cv32_type4_test.sv
└── vseq/
    └── uvmt_cv32_vseq_lib.sv
```

6.4.2 Writing a Test Program

This document will probably never include a detailed description for writing a test program. The core's ISA is well documented and the execution environment supported by the testbench is trivial. The best thing to do is check out the examples at **`$PROJ_ROOT/cv32/tests/core`**.

6.4.3 Writing a UVM Test to run a Test Program

The CV32 base test, *uvmt_cv32_base_test_c*, has been written to support all five of the test program types discussed in Section 6.1.

There are pre-existing UVM tests for type 1 (pre-existing, self-checking) and type 4 (generated, not-self-checking) tests for CV32E40P in the core-v-verif repository. If you need a type 2 or type 3 test, have a look at these and it should be obvious what to do.

6.4.3.1 Testcase Scriptware

At **$PROJ_ROOT/cv32/tests/uvmt_cv32/bin/test_template** you will find a shell script that will generate the shell of a testcase that is compatible with the base test. This will save you a bit of typing.

6.4.4 Running the testcase

Testcases are intended to be launched from **$PROJ_ROOT/cv32/sim/uvmt_cv32**. The README at this location is intended to provide you with everything you need to know to run an existing testcase or a new testcase. If this is not the case, please create a GitHub issue and assign it to @mikeopenhwgroup.

7 CORE-V Formal Verification

Formal verification of the CV32E and CV64A cores is a joint effort of the OpenHW Group and OneSpin Solutions with the support of multiple Active Contributors (AC) from other OpenHW Group member companies. This section specifies the goals, work items, workflow and expected outcomes of CV32E and CV64A formal verification.

7.1 Goals

Completeness of formal verification is measured in a way similar to simulation verification. That is, a Verification Plan (Testplan) will be captured that specifies all features of the cores, and assertions will be either automatically generated or manually written to cover all items of the plan. Formal verification is said to be complete when proofs for all assertions have been run and passed. Code coverage and/or cone-of-influence coverage will be reviewed to ensure that all logic is properly covered by at least one assertion in the formal testbench.

Note that proofs may be either bounded or unbounded. Where it is not practical to achieve an unbounded proof a human analysis is performed to determine the minimum proof depth required to sign off the assertion in question. For these bounded proofs, the assertion is considered covered when the required proof depth has been achieved without detecting a counterexample (failure).

7.2 Formal CORE-V ISA Specifications

It is believed that the RISC-V Foundation has plans to create formal, machine readable, versions of the RISC-V ISA and that the implementation language for this machine readable ISA is Sail. Once complete and ratified, the formal model(s) will be *the* ISA and the human language versions of the ISA will be demoted to reference documents. ToDo: find a reference to confirm this.

Sail is a product of the REMS group, an academic group in the UK, which has also created partial Sail models of the RV32IMAC and RV64IMAC ISAs. These model are maintained in GitHub at https://github.com/rems-project/sail-riscv and the project is in active development.

7.2.1 Use of Sail Models in CORE-V Verification

Three considerations are driving the OpenHW Group's interest in formal ISA (Sail) models:

- Assuming the RISC-V Foundation develops and supports complete ISA specification in Sail, the RISC-V community may expect the same of OpenHW. Developing, maintaining and supporting formal specifications of the CORE-V ISAs will lend credibility to the CORE-V family.
- A formal model of the ISA supports the creation of a tool-flow that can produce "correct-by-construction" software emulators, compilers, compliance tests and reference models. This capability will generate interest in CORE-V IP from both Industry and Academia.
- The primary interest in Sail is the **possibility of using a Sail model as a reference model for the formal testbench assertions.** The assertions will verify that a certain micro-architecture implements the ISA from the Sail spec. Essentially, the assertions together with the OneSpin

GapFree technology perform an equivalence check between Sail model and the RTL to ensure that:

- everything behaves according to the ISA (Sail model),
- nothing on top of what is specified in the ISA (Sail model) is implemented in the RTL.

OneSpin is currently investigating how to best make use of the Sail model. This will be captured in a future release of this document.

7.2.2 Development of Sail Models for CORE-V Cores

At the time of this writing[15], the completeness of the RV32/64IMAC Sail models is not known, but is believed to be complete. Extensions of the models will be required to support Zifencei, Zicsr, Counters and the XPULP extensions. OpenHW may also wish to include User Mode and PMP support as well, especially for the CV64A. Its a given that much or all of the work to create these extensions to the Sail models will need to be done by the OpenHW Group.

Given that CV32E and CV64A projects are leveraging pre-existing specifications and models, it should be possible for the micro-architecture and Sail models to be developed in parallel and by different ACs.

7.3 Work Items

This sub-section details a set of work items (or deliverables) to be produced by either the OpenHW Group and/or OneSpin Solutions. Note that deliverables assigned to OpenHW may be produced solely or jointly by an employee or contractor of the OpenHW Group, or by an Active Contributor (AC) provided by another member company.

Table 2: CORE-V Formal Verification Work Items

#	Work Items	Provided By	Comment
1	Micro-architecture Specifications (one per core)	OpenHW Group	Based on design documentation developed by PULP-Platform
2	ISA Sail Models (one per core)	OpenHW Group	Based on the RV64IMAC Sail model developed by the RISC-V Foundation
3	Define the use of Sail ISA specification/model in a formal verification flow.	OneSpin Solutions	OneSpin is currently investigating how to best make

15 First week of January, 2020.

			use of the Sail model. See Section 7.2 for a discussion of this topic.
4	Compute Infrastructure	OpenHW Group	OpenHW will create one or more VMs on the IBM Cloud to support formal verification of both Cores.
5	Tool Licenses	OneSpin Solutions	OneSpin provides tool licenses in sufficient numbers to allow for "reasonable" regression turn-around time.
6	Formal Testplans (one per core)	OpenHW Group and OneSpin Solutions	ToDo: work with OneSpin to define template.
7	Formal Testbenches (one per core)	OneSpin Solutions	OneSpin is not responsible for the complete formal testbench, see sub-section 7.3.4.
8	Formal Verification of Cores	OpenHW Group and OneSpin Solutions	See the sub-section 7.4.

7.3.1 Specifications

See rows #1 and #2 in Table 2, above. The first step of the process is for the OpenHW Group to develop and deliver:

- **Micro-architecture specifications** for both cores. This activity has started and is proceeding under the direction of Davide Schiavone, Director of Engineering for the Cores Task Group.
- **Sail models** of each core's ISA. This activity will be managed by the Verification Task Group. The expectation is that this pre-existing Sail model can be extended for both the CV32E and CV64A cores, including the PULP ISA extensions.

7.3.2 Compute and Tool Resources

This is rows #4 and #5 in Table 2, above. Tool licenses in sufficient numbers to allow for "reasonable" regression turn-around time on CV64A RTL. These tools will be installed on VMs on the IBM Cloud and will only be accessible by employees/contractors of the OpenHW Group or select ACs actively involved in formal verification work.

7.3.3 Formal Testplans

OpenHW and OneSpin will jointly develop Formal Testplans for both the CV32E and CV64A. The high-level goals of the FTBs will be two-fold:

1. Prove that the core designs conform to the RISC-V+Pulp-extended ISA. Specifically, every instruction must:
    - decode properly
    - perform the correct function
    - complete as specified (location of results, condition flag settings, etc.)

In particular, the above must be true in the presence or absence of exceptions, interrupts or debug commands.

2. Prove the logical correctness of the implementation with respect to the micro-architecture (note that not all of these features are support by every CORE-V core):
    - Interface logic
    - Pipeline hazards
    - Exception handling
    - Interrupt handling
    - Debug support
    - Out of order execution
    - Speculative execution
    - Memory management

7.3.4 Formal Testbenches

Conceptually, a formal testbench is a collection of assumptions, assertions and cover statements. The assumptions provide the necessary scaffolding logic in order to support the operation of the formal engines. Examples of these include the identification of clocks, and resets, constraints on clock and reset cycle timing and input wire-protocol constraints. Most assertions in the formal testbench exist to prove one or more items in the Testplan. Covers exist to prove that a specific function has, in fact, been tested. The formal testbench coding is considered complete when all assumptions, assertions and covers are coded.

OneSpin will initiate development of Formal testbenches (FTB) for CV32E and CV64A as soon as possible. These FTBs will be open-source, ideally implemented in SystemVerilog, and may be based on OneSpin's RISC-V Verification App[16].

It is not expected that OneSpin will deliver a complete formal testbench. Rather, OneSpin will deliver a formal testbench that has two specific attributes:

1. Assertions to prove that the core implementation (RTL model) conforms to the RISC-V+Pulp-extended ISA. The ISA used for this will be the Sail model (see Section X).
2. Sufficient assumptions, assertions and covers such that ACs from other OpenHW member companies are able to read the Testplan and add the required assumptions, assertions and covers to move the project towards completion.

7.4 Formal Verification Workflow

ToDo: add a figure here to illustrate the workflow

The workflow for CORE-V formal verification will be similar to that used by simulation verification. The three key elements of the workflow are:

- A **GitHub** centralized repository.
- **Distributing** the work across multiple teams in multiple organizations;
- **Continuous Integration.** Once the compute environment on the IBM Cloud is established and OneSpin tools deployed, OneSpin will assist OpenHW to generate script-ware to support automated checks whenever a new branch or update is pushed to the central repository. Such check can pinpoint relatively simple errors without running a lot of verification. OpenHW would then maintain these scripts. In addition, there will be scripts for more comprehensive/full regression runs that OpenHW should maintain after initial delivery (if the file list for compilation changes due to RTL re-organization, for example, this needs adaption in the respective compile scripts).

The most significant difference between the simulation and formal verification workflows is that all formal verification will use tools provided by OneSpin Solutions. OneSpin engineers will run either on OneSpin's own compute infrastructure or on the Virtual Machines provided by IBM and managed by OpenHW. ACs from other member companies will run on the IBM Cloud and use OneSpin tools.

16 OneSpin White paper: Assuring the Integrity of RISC-V Cores and SoCs. OneSpin Solutions, 2019.

8 CORE-V FPGA Prototyping

ToDo. This may be captured in a separate document.

9 Revision History

Revision	Date	Author	Org.	Comment
V0.1	2020-01-08	Mike Thompson	OpenHW Group	First published draft.
V0.2	2020-01-09	Mike Thompson	OpenHW Group	Minor updates.
V0.3	2020-01-14	Mike Thompson	OpenHW Group	Move all Verification Planning to Section 1. Started Section 2.3, “CV32E Sim Verif Env”.
V0.4	2020-02-07	Mike Thompson	OpenHW Group	Moved Revision History to end of document. Add Section 1.4 “A Note About EDA Tools”. Significant restructuring of Section 3 & 4. Updated Illustration 1.
V0.5	2020-03-23	Mike Thompson	OpenHW Group	Add new Section 6: Simulation Tests in the UVM Environments
V0.6	2020-03-26	Mike Thompson	OpenHW Group	Section 1.2: Added $PROJ_ROOT. Section 3.2.2: Fixed confusion between paths used in the RI5CY and core-v-verif repositories.