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 Revision History

Revision	Date	Author	Organization	Comment
V0.1	2020-01-08	Mike Thompson	OpenHW Group	First published draft
V0.2	2020-01-09	Mike Thompson	OpenHW Group	Minor updates

1.2 License

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

Unless required by applicable law or agreed to in writing, products distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and limitations under the License.

1.3 Verification Planning

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 is that the CV32E40P testplan 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, someone 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.

1.4 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.

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

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.

1.5 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 GitHub fork of the RI5CY project, and CV64A is a fork of Ariane. 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 2.2.

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

2 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, the next sub-section provides a (very) quick summary. Sub-section 2.2 review the status of RI5CY and Ariane testbenches in sufficient detail to provide the necessary context for sub-section 2.3, which details how the RI5CY and Ariane simulation environments will be migrated to CV32E and CV64A simulation environments.

2.1 Executive Summary

In the case of the CV32E, we have an existing verification environment developed for RI5CY. This environment 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 new testcases will be developed for the existing RI5CY environment 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 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.

2.2 RI5CY and Ariane Verification Environment Overviews

2.2.1 RI5CY

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

2.2.1.1 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[2] 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.

2.2.1.2 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[3]. 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

2 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 Derived from the PULP platform SDK.

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`[4] 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 its destination register.

4 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.

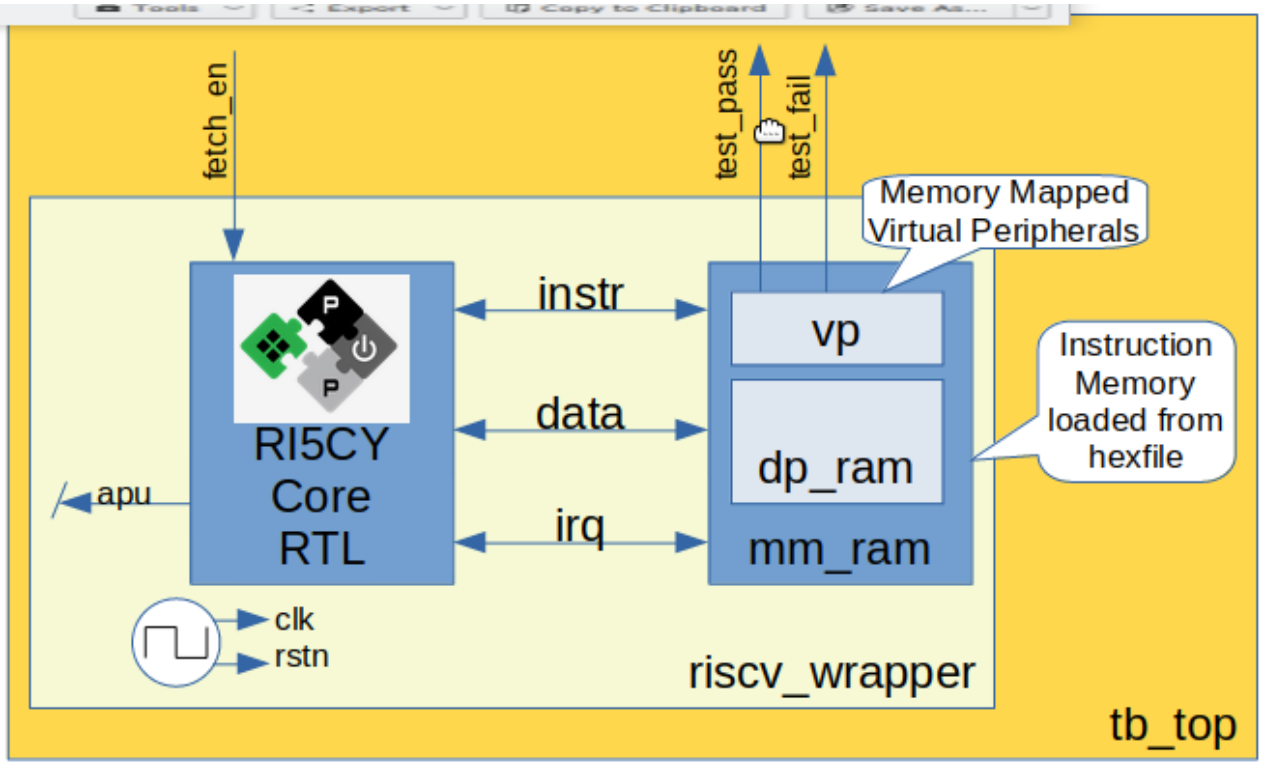

Illustration 1: RI5CY Testbench

The testcases are broadly divided into two categories, riscv_tests and riscv_compliance_tests. These are located in the **tb/core/ riscv_tests** and **tb/core/ riscv_compliance_tests** respectively.

2.2.1.2.1 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 **tb/core/ riscv_tests/rv64ui/addi.S**, a program that 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.

2.2.1.2.2 RISC-V Compliance Tests

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

2.2.1.2.3 Firmware Tests

There are a small set of C programs in the **tb/core/ 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.

2.2.1.3 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 2.2.3 in this document).
b) UVM environment class supporting the complete UVM run-flow and messaging service (logger).
c) Constrained-random stimulus of instructions 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 effort has already been significantly de-risked by Metrics and Imperas (as will be discussed in a future revision of this document - ToDo).

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[5]. The overall effort will be reduced via testcase automation and the probability of finding corner-case bugs will be greatly enhanced.

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

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.

2.2.2 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.

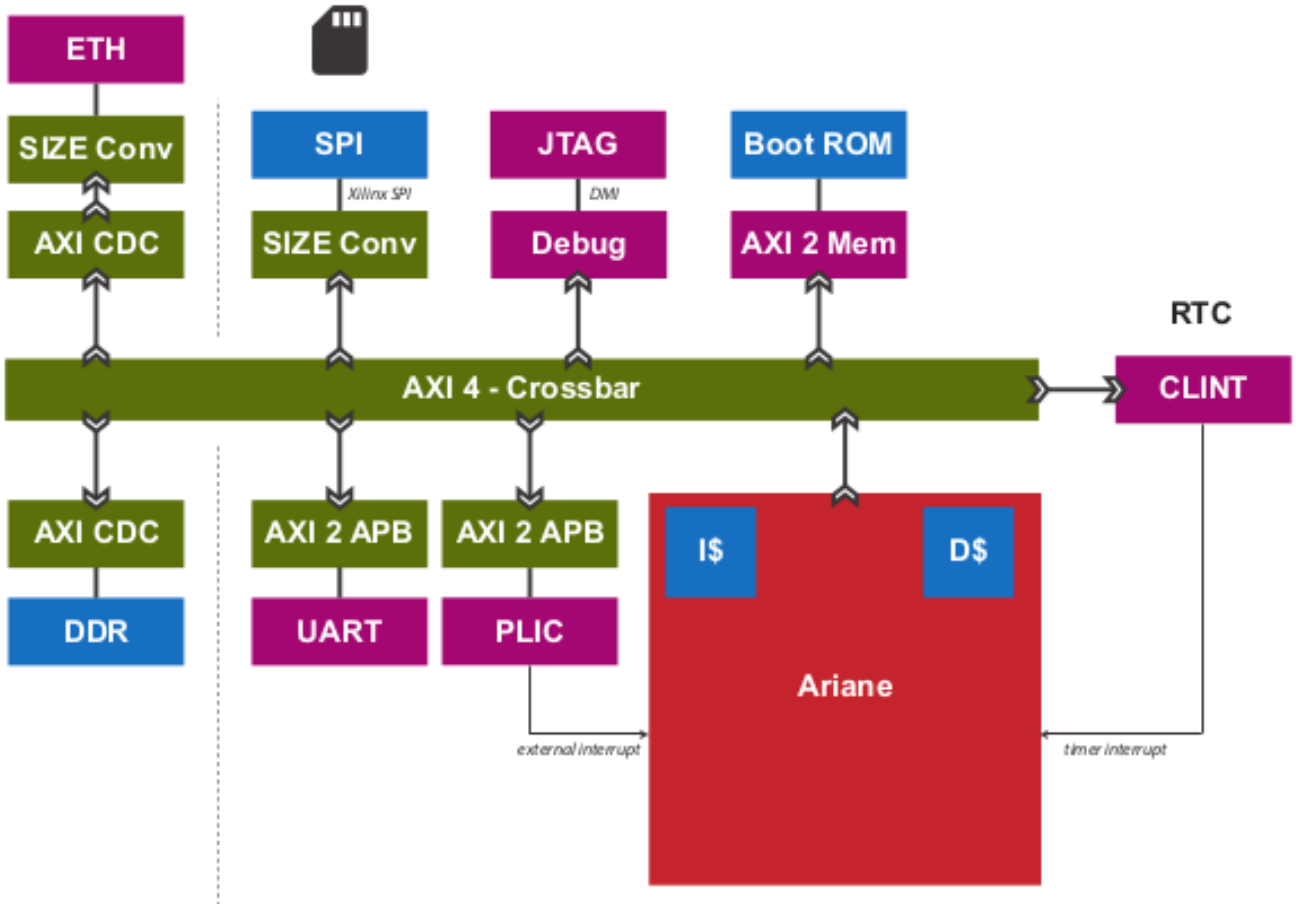

Illustration 2: Ariane Testbench

2.2.2.1 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.

2.2.2.2 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.

2.2.3 IBEX

From a verification perspective, the Ibex core is the most mature of the three cores discussed in this document. 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.

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.

2.2.3.1 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.

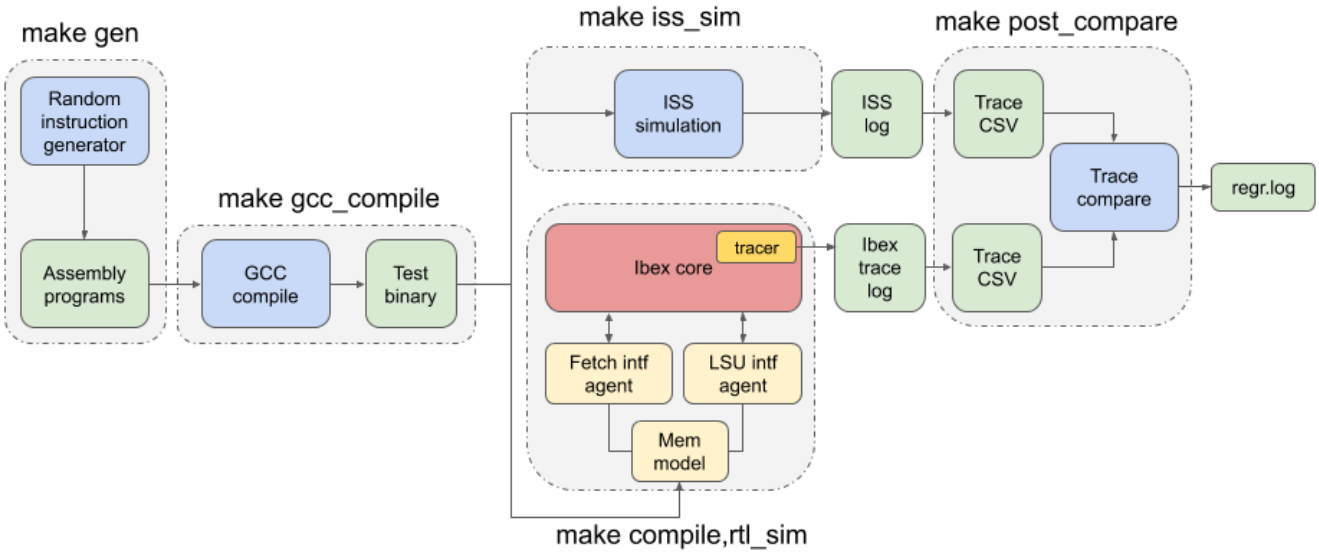

Illustration 3: Ibex Verification Environment

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

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

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.

2.3 Incremental Development Strategy

ToDo

3 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.

3.1 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).

3.2 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.

3.2.1 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.

3.2.2 Development of Sail Models for CORE-V Cores

At the time of this writing[7], 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.

3.3 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 1: 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 use of the Sail model. See Section 3.2 for a discussion of this topic.
4	Compute Infrastructure	OpenHW Group	OpenHW will create one or more VMs on the IBM Cloud

7 First week of January, 2020.

			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 3.3.4.
8	Formal Verification of Cores	OpenHW Group and OneSpin Solutions	See the sub-section 3.4.

3.3.1 Specifications

See rows #1 and #2 in Table 1, 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.

3.3.2 Compute and Tool Resources

This is rows #4 and #5 in Table 1, 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.

3.3.3 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

3.3.4 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[8].

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

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.

3.4 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.

4 CORE-V FPGA Prototyping

ToDo. This may be captured in a separate document.