The State of CORE-V Simulation Verification

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

Introduction

The OpenHW Group is launching a pre-silicon verification effort of two select cores from the family of CORE-V cores, the goal of which is 'production ready' RTL. This effort will rely primarily on simulation verification and may also use formal verification and FPGA prototyping. In this document we focus exclusively on simulation.

The simulation methodology used will be the UVM, implemented using SystemVerilog. As much as possible, testcases will use constrained-random stimulus rather than hand-coded, directed testcases. The verification environment will also be capable of running the RISC-V compliance test-suite[1]. The completeness of the verification effort will be measured by a combination of code and functional coverage.

The two cores to be verified are known as CV32 and CV64. CV32 is derived from the RI5CY core (pronounced "risky") and CV64 is based on the Ariane core. Both of these cores were developed by the PULP Platform team. A third core of interest to this discussion is Ibex, developed by lowRISC. Ibex is a derivative of the zero-riscy core also developed by PULP Platform.

Given that the CV32 and CV64 are derived from the RI5CY and Ariane cores respectively, it is natural to assume that the verification environments for CV32 and CV64 should also be directly derived from RI5CY and Ariane. This document demonstrates that this is not the best path to our goal.

With that in mind, the purpose of this document is threefold:

1. Introduce the structure and status of the verification environments for each of RI5CY, Ibex and Ariane cores.
2. Comment on the 'completeness' of the verification that these environments currently have, or with some additional effort, could have.
3. Propose a plan for developing CV32 and CV64 verification environments using the above as a starting point.

1 Supported by the RISC-V Foundation Compliance Task Group.

Currently, this is a stand-alone document. A later version will be merged into the CORE-V Verification Strategy, at which time this document will be deprecated. The Verification Strategy will detail, among other things, what ‘production ready’ RTL means and how we will achieve it.

Executive Summary

In the case of the CV32, we have an existing verification environment developed for RI5CY. This environment is useful, but insufficient to achieve the goal of ‘production ready’ RTL. A two-pronged approach is recommended 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 CV32 core. 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 CV64, 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 CV64. This environment will re-use many of the components developed for the CV32 verification environment, and will have the same ability to run the RISC-V Compliance test-suite.

License

Core Repositories

Unless otherwise noted the verification environments discussed in this document come from the follow 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

Definition of Terms

Verification Environment: a fancy term for all of the code, scripts and bailer-twine 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.

RI5CY

The verification environment for RI5CY is shown in Figure 1 on page 4. 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 popular of these is a virtual printer that provides something akin to a "hardware printf" capability. When the core writes ASCII data to a specific memory location the wrapper writes it 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.

Testcases

Testcases are written as RISC-V assembler which is compiled/linked using a light SDK developed to support these test[3]. 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. 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);
```

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.

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.

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 and x29 as its immediate targets.

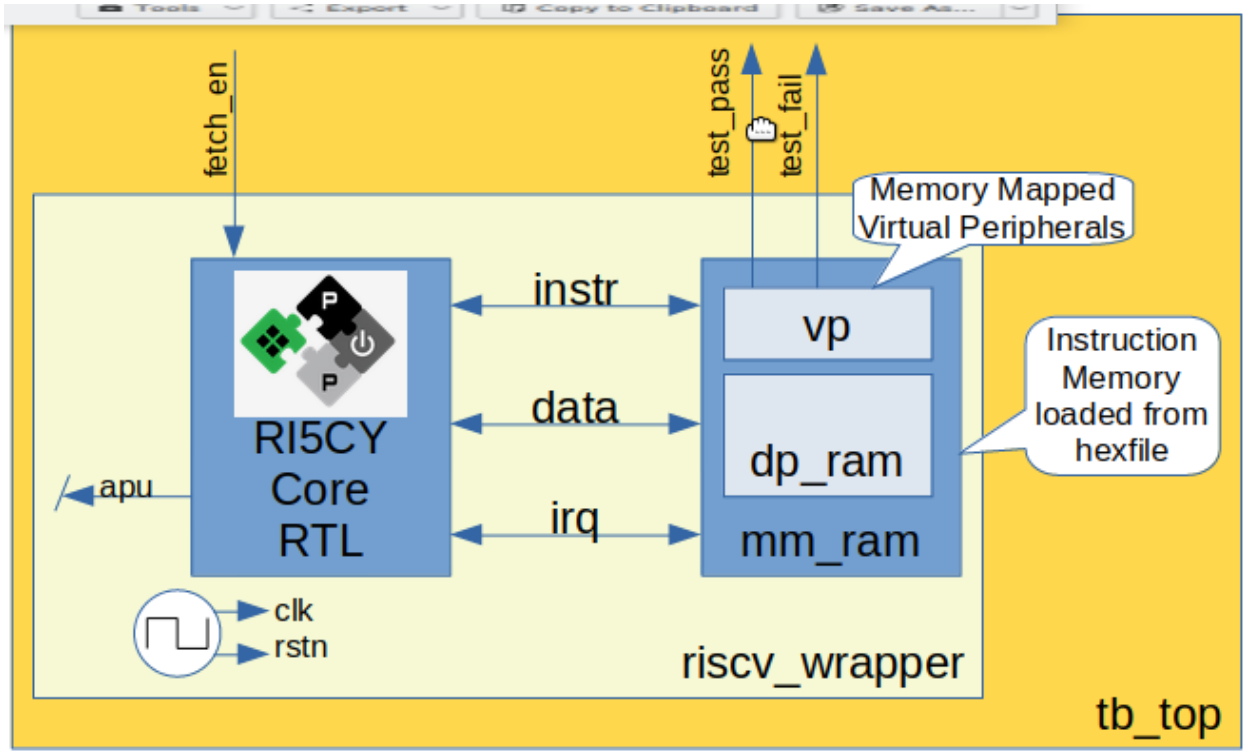

Figure 1: RI5CY Testbench

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.

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.

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.

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.

Comments and Recommendations for CV32 Verification

The RI5CY verification environment has several attractive attributes:

1. It exists and it runs. Never underestimate the value of a working environment 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 can run on open-source tools. This is not a requirement for the OpenHW Group or its member companies, but it is 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. The vast majority of the testing focuses on only the core itself with little attention paid to "core complex" 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 – what happens in the failure case? (e.g. programming error)
viii. There is no functional coverage model, and code coverage data has not been collected.

So, much work remains to be done, and the effort to scale the existing RI5CY verification environment and testcases to 'production ready' CV32 RTL is not warranted given the shortcomings of the approach taken. It is therefore recommended to replace this verification environment with a UVM complaint environment with the following attributes:

a) Structure modelled after the verification environment used for the low-RISC Ibex core (see Section "IBEX" 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 Developer, 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).

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.

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 CV32 (RI5CY) and CV64 (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 CV32 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.

Ariane

The verification environment for Ariane is shown in Figure 2 on page 8. It is coded entirely in SystemVerilog, using more modern syntax than the RI5CY environment.

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.

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

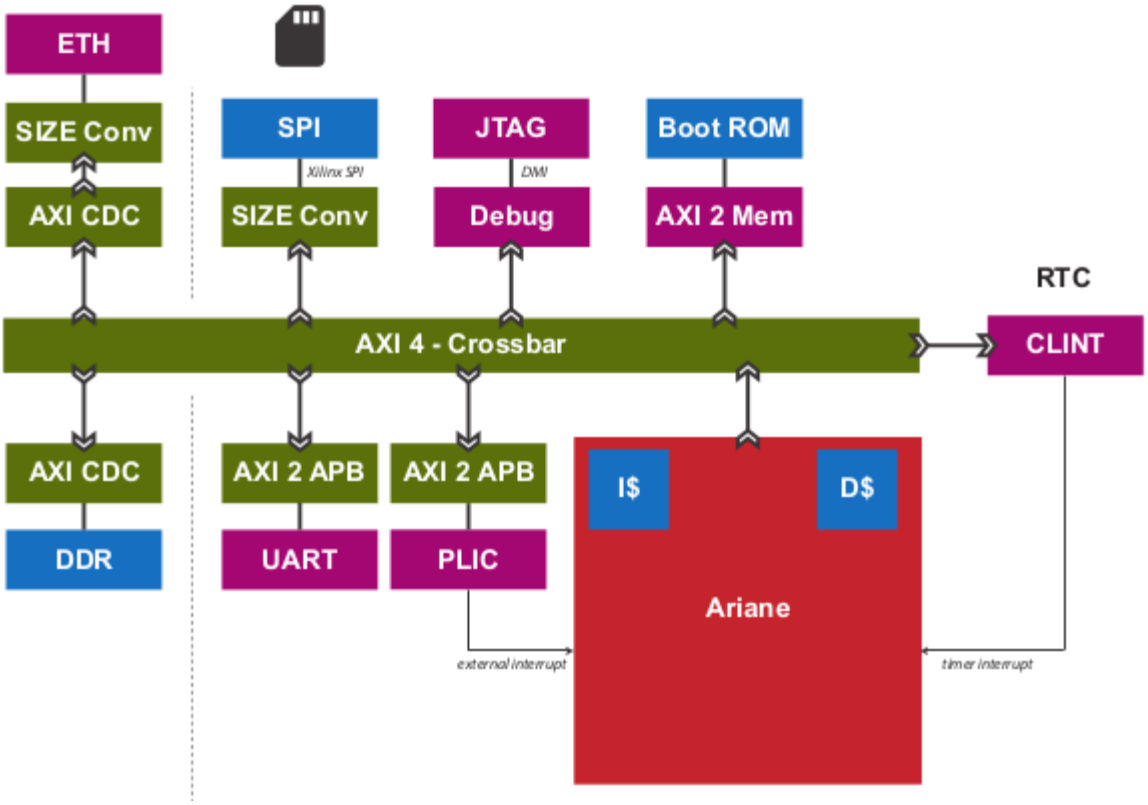

Figure 2: Structure of Ariane Testbench *(source: PULP Platform)*

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 CV64 lacks a good starting point for its verification efforts.

Comments and Recommendations for CV64 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 CV64. This would be a core-based verification environment as is envisioned for CV32 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 CV32 can be easily extended for CV64, thereby allowing a single environment to support both, or completely independent environments for CV32 and CV64 will be required.

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 CV32 and CV64.

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.

Ibex Impact on CV32 and CV64 Verification

The Ibex verification environment, shown in Figure 3, below, 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 Figure) is applied to measure whether or not the verification goals have been achieved.

As shown in the Figure, what we actually have here 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 elf format. The program is produced according to a set of input constraints.
2. A compiler that translates the elf into 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.

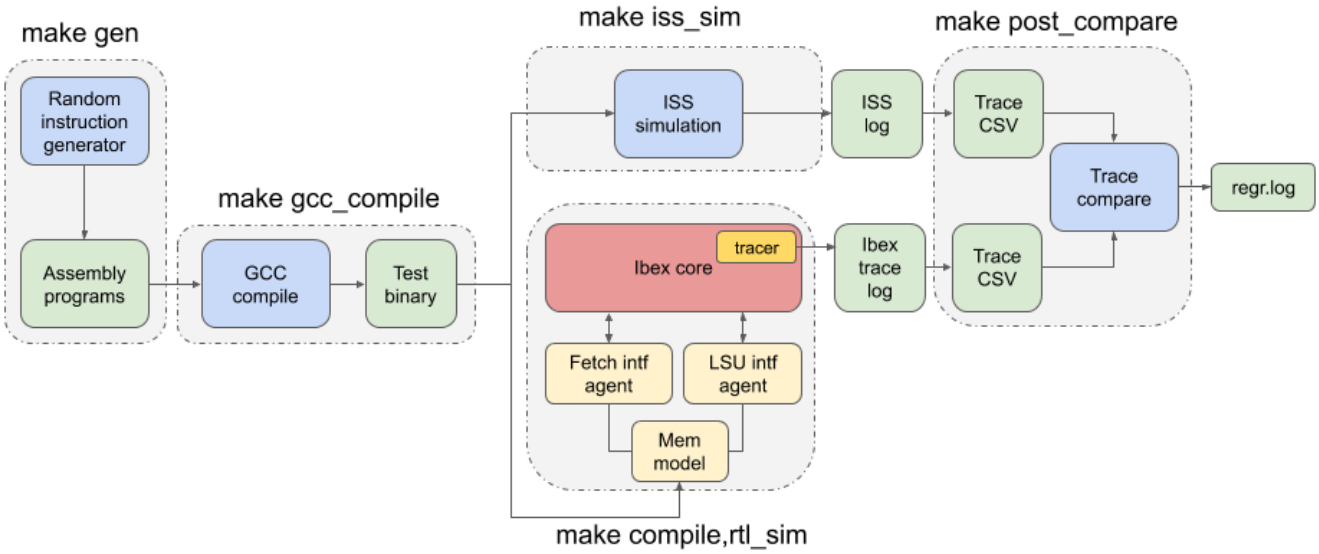

Figure 3: IBEX Verification Environment *(source: low-RISC)*

This is an excellent starting point for the CV32 verification environment and our first step should be to clone the Ibex environment and get it running against the CV32[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 CV32 and CV64 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:

- 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.

Conclusion

See the Executive Summary near the beginning of this document.