CORE-V-Docs Documentation

Mike Thompson

Nov 23, 2020

Contents:

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

CHAPTER 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 CVA6.

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 CVA6 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 CVA6.

1.1 License

1.2 CORE-V Projects

The core-v-verif project is being developed to verify all CORE-V cores. The cores themselves are in their own git repositories. Below are links to the RTL sources and documentation for CORE-V cores currently in development:

- CV32E40P RTL source
- CV32E40P user manual

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

- CVA6 RTL source

The OpenHW Group also maintains multiple repositories for stand-alone verification components. At the time of this writing two are up and running (more are planned):

- core-v-isg Instruction stream generator denotated by NVIDIA.
- FORCE-RISCV Instruction stream generator denotated by Futurewei.

1.3 Definition of Terms

Term	Defintion
CORE-V	A family of RISC-V cores developed by the OpenHW Group.
Member Company (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.
Active Contributor	An employee of a Member Company that has been assigned to work on an OpenHW Group project.
Instruction Set Simulator (ISS)	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. 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 CVA6, this includes the supported RISC-V ISA compliant instructions, plus a set of XPULP extended instructions.
Toolchain	See SDK.
Test-Program	A software program, written in C or RISC-V assembly, that executes on the simulated RTL model of a core. Test-Programs may be manually written or machine generated (e.g. riscv-dv).
TPE	Test-Program Envionment. A set of support files, such as a C runtime configuration (crt0.S), linker control script (link.ld), etc. that are used to define the software envrionment used by a test-program.
BSP	Board Support Package. A more widely used term for BSP.
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.
Testcase	In the context of the CORE-V UVM verification environment, a a testcase is distinct from a test-program. A testcase is extended from the *uvm_test* class and is used to control the simulation of the UVM environment. A test-program is a set of instructions loaded into the testbench memory and executed by the simulated core.
$PROJ_ROOT	Local path of a cloned copy of a GitHub repository. An example to illustrate: [prompt]$ 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.

1.4 Conventions Used in this Document

Bold type is used for emphasis.

Filenames and filepaths are in italics: *./cv32/README.md*.

1.5 CORE-V Genealogy

The first two projects within the OpenHW Group's CORE-V family of RISC-V cores are the CV32E4 and CVA6. Currently, two variants of the CV32E4 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. CV32E40(P) is a derived of the RI5CY project[2], and CVA6 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 CV32E40(P) and CVA6 are strongly influenced by the development history of these cores. This is discussed in more detailed in *PULP-Platform Simulation Verification*.

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

1.6 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 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.

The "core" testbench of the CV32E40P can be compiled/simulated using Verilator, an open-source software tool which translates a subset of the SystemVerilog language to a C++ or SystemC cycle-accurate behavioural model. Note that "core" testbench is not considered a production verification environment that is capable of fully verifying the CV32E40(P) cores. The purpose of the "core" testbench is to support software teams wishing to run test-programs in a simulation environment.

[2] Note that CV32E40P 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] CVA6 is not a fork of the Ariane. The GitHub repository https://github.com/pulp-platform/ariane was moved to https://github.com/openhwgroup/cva6.

CHAPTER 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

ToDo

2.3 XPULP Instruction Extensions

ToDo

2.4 Custom Circuitry

ToDo

2.5 Interrupts

ToDo

2.6 Debug

ToDo

2.7 RVI-Compliant Interface

ToDo

CHAPTER 3

PULP-Platform Simulation Verification

Before discussing the verification strategy of the CV32E and CVA6, we need to consider the starting point provided to OpenHW by the RI5CY (CV32E) and Ariane (CVA6) 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 Executive Summary below provides a (very) quick summary. Those wanting more background should read the *RI5CY* and *Ariane* sub-sections of this chapter which review the status of RI5CY and Ariane testbenches in sufficient detail to provide the necessary context for the *CV32E40P Simulation Testbench and Environment* and *CV6A Simulation Testbench and Environment* chapters, which detail how the RI5CY and Ariane simulation environments will be migrated to CV32E and CVA6 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 CVA6, 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 CVA6. 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.

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.

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. By cloning the core-v-verif repository, these original RI5CY tests can be found at *$PROJ_ROOT/cv32/tests/core/riscv_tests* and *$PROJ_ROOT/cv32/tests/core/riscv_compliance_tests*. Updated versions of these tests for CV32 are located at *$PROJ_ROOT/cv32/tests/core/cv32_riscv_tests* and *$PROJ_ROOT/cv32/tests/core/cv32_riscv_compliance_tests*.

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 exercises the *add immediate* instruction.

[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.

[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.

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

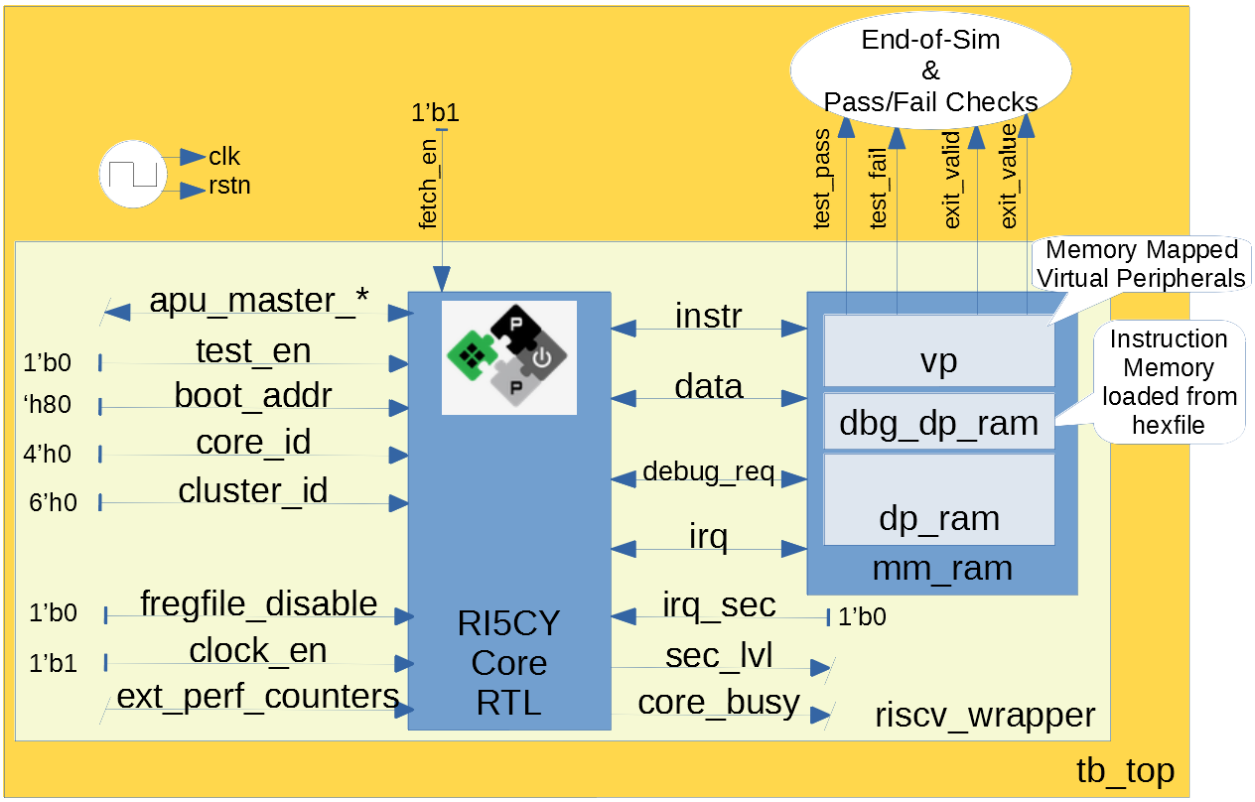

Fig. 1: Illustration 1: RI5CY Testbench

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** 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 *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 CVA6 (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.

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 CVA6 lacks a good starting point for its verification efforts.

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

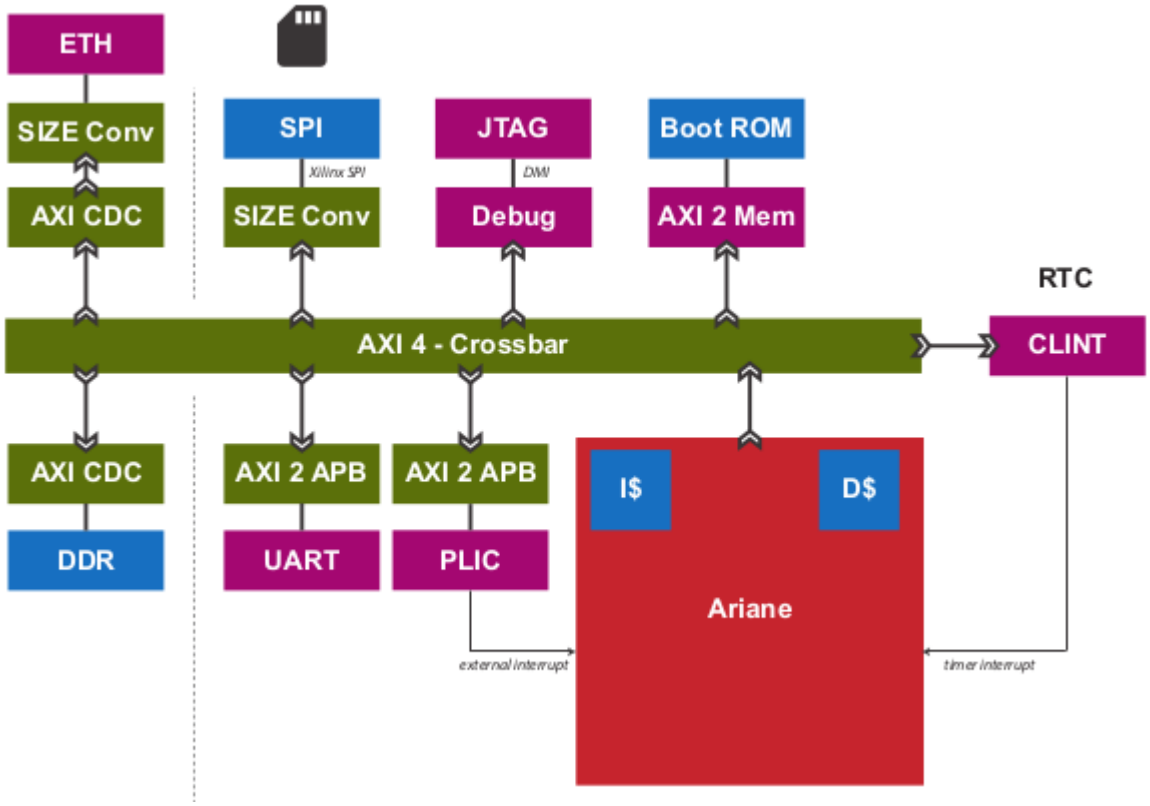

Fig. 2: Illustration 2: Ariane Testbench

3.3.2 Comments and Recommendations for CVA6 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 CVA6. 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 CVA6, thereby allowing a single environment to support both, or completely independent environments for CV32E and CVA6 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 CVA6.

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 CVA6 Verification

Illustration 3 is a schematic of the Ibex UVM verification environment. The flow of the Ibex environment is very close to what you'd expect to see in a UVM environment: constraints define the instructions in the generated program which is fed to both the device-under-test (Ibex core RTL model) and an ISS reference model. 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.

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

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

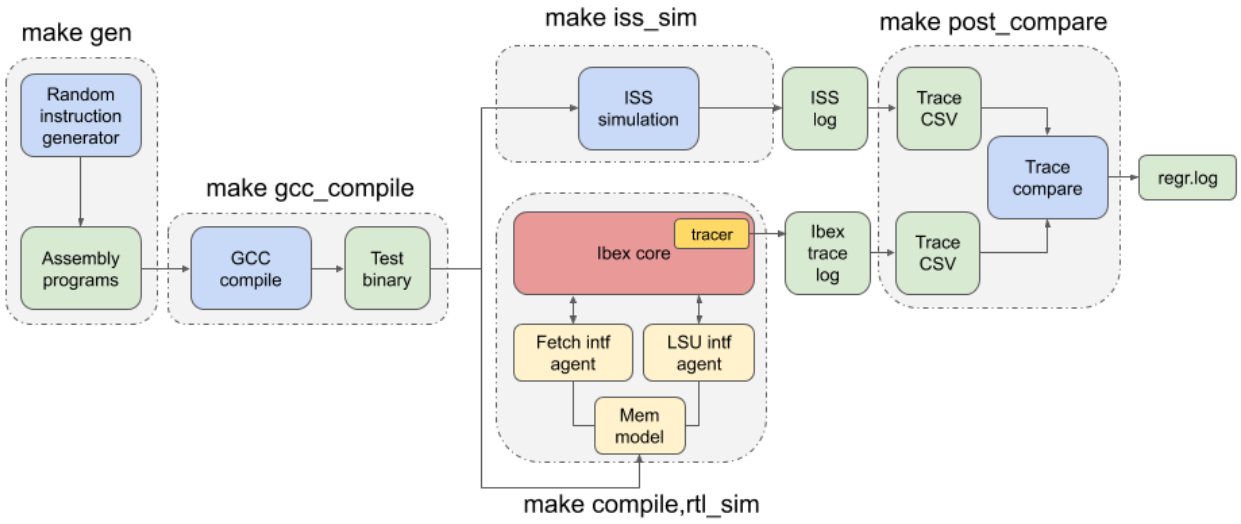

Fig. 3: Illustration 3: Ibex Verification Environment

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 CVA6 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.

CHAPTER 4

CORE-V Verification Environment

A previously stated goal of the core-v-verif project is to produce a unified verification environment for all CORE-V cores supported by the OpenHW. While several of the chapters of this document focus on the specifics of a single core, such as the CV32E40P, this chapter details how the components, structure and organization of core-v-verif are used to implement and deploy a complete simulation verification environment for any core in the CORE-V family.

Note: This chapter of the Verification Strategy is a work-in-progress. It contains several forward looking statements about verification components that are defined, but yet to be implemented.

The core-v-verif project is not a single verification environment that is capbable of supporting any-and-all CORE-V cores. Rather, core-v-verif supports the verification of multiple cores by enablng the rapid creation of core-specific verification environments. There is no attempt to define a one-size-fits-all environment as these inevidibly lead to either bloated code, needless complexity or both. Instead, the idea is to create a toolkit that allows for the rapid development of core-specific environments using a set of high-level reusable components and a standard UVM framework.

UVM environments are often described as a heirarchy with the device-under-test (the core) at the bottom and testcases at the top. In between are various components with increasing degress of abstraction as we go from the bottom levels (the register-transfer level) to the middle layers (transaction-level) to the top (tests). The lower layers of the environment see the least amount of re-use owing to the need to deal with core-specific issues. Components at this level are core-specific. At the transaction level there can be considerable amounts of re-use. For example, it is eary to imagine a single UVM Debug Agent serving the needs of any and all CORE-V cores. The test level sees a mix of re-usable tests (e.g. RV32IMAC compliance) and core-specific tests (e.g. hardward loops in CV32E40P).

The core-v-verif project expliots this idea to maximize re-use across multiple cores by striving to keep as much of the environment as possible independant of the core's implementation. Components such as the instruction generator, reference model, CSR checkers can be made almost entirely independent of a specific core because they can be based on the ISA alone. Other components such as the functional coverage model, debug and interrupt Agents and the test-program environment can be implemented as a mix of re-usable components and core-specific components.

Depending on the details of the top-level interfaces of individual cores, the lowest layers of the core-v-verif environment may not be re-usable at all.

4.1 Environment Structure

A CORE-V verification environment, built from the resources provided by core-v-verif can be conceptually divided into four levels: Testbench Layer, Translation Layer, Abstraction Layer and Test Layer. Each of these will be discussed in turn.

4.1.1 Testbench Layer

A CORE-V testbench layer is comprised of two SystemVerilog modules and a number of SystemVerilog interfaces. We will discuss the SystemVerilog interfaces first, as this will make it easier to understand the structure and purpose of the modules.

SystemVerilog Interfaces

On any given CORE-V core, the top-level ports of the core can be categorized as follows:

- Instruction and Data memory interface(s)
- Clocks and Resets
- Configuration
- Interrupts
- Trace
- Debug
- Special Status and Control

The Instruction and Data memory interface is listed first for a reason. This interface is generally the most core-specific. For example, CV32E supports I&D interfaces that are AHB-like while CVA6 supports AXI-like interfaces. These are significant difference and so the Testbench Layer deliberately hides this interface from the higher-level layers. This is done in the "DUT Wrapper" module, see below.

The remaining interface categories can be defined as generic collections of input or output signals whose operation can be defined by higher layers. A few examples should illustrate this point:

- Clocks and resets can be parameterized arrays of clock and reset signals. The upper layers of the environment will define the number of clocks and implement the appropriate frequency and phase relationships. Resets are managed in the same manner.
- The Interrupts interface can also be implemented as parameterized array of bits. The upper layers of the environment are responsible for asserting and deasserting these signals under direction of a UVM sequence and/or test.

Testbench Modules

The two modules of the Testbench Layer are the "DUT Wrapper" and the "Testbench". The purpose of the wrapper is to conceal as many core-specific physical attributes as possible. As hinted at abovem this is done by keeping control of the core's memory interface(s) and mapping all other ports to one of the non-memory interface types.

The wrapper instaniates a memory model that connects directly to the core's instruction and data interface(s). This memory model also supports a number of memory mapped virtual peripherals. The core's memory interface is not "seen" by any other part of the environment, so this interface (or these interfaces, as the case may be) can be completely different from other cores and the only part of the environment affected is the DUT wrapper, and its memory model.

The address map of the modeled memory and peripherals is implemented to ensure compatibility with the test-program environment (described later in this chapter).

The Testbench module is mostly boiler-plate code that does the following: - instaniates the wrapper - push handles of the SV interfaces to the UVM configuration database - invoke run_test() - Implement a final code-block to display test pass/fail

The expection is that the DUT Wrapper module will be core-specific and will need to be coded from scratch for each CORE-V core. The Testbench module is also expected to be core-specific, but can be easily created by copying and modifying a Testbench module from a previous generation. The SystemVerilog interfaces for Clocks and Resets, Configuration, Interrupts, Trace, Debug, plus Special Status and Control are generic enough to be fully re-used.

4.1.2 Abstraction Layer

Instruction Set Generators

Reference Models

Functional Coverage Models

CORE-V UVM Agents

Test Program Environment

4.1.3 Test Layer

TODO

4.2 Verification Plans

TODO

4.3 Repository Structure

The top-level of the core-v-verif repository is specifically organized to support multiple verification environments. The directory structure below shows a version of core-v-verif that supports multiple CORE-V cores. What follows is a brief discription of the purpose of each top-level directory. Refer to the README files at each of these locations for additional information. If you read nothing else, *please* read *$PROJ_ROOT/<core>/sim/README.md*.

- **core-v-cores**: the the Makefiles in the <core>/sim directory will clone the RTL for <core> into core-v-cores/<core>/rtl. This structure allows for the simulataneous verification of multiple cores from the same core-v-verif repostory.
- **<core>**: this directory contains the <core> specific environment, testbench, tests and simulation directories.
- **ci**: This directory supports common and core-specific scripts and configuration filesto support user-level regressions and the Metrics continuous integration flow.
- **lib**: This is where the bulk of the re-usable components and tests are maintained. This is where you will find the instruction generator, reference model, common functional coverage models, UVM Agents for clocks-and-resets, interrupts, status, etc.

```
$PROJ\_ROOT/
├── core-v-cores
│   ├── <core1>
│   ├── <core2>
│   └── ...
├── <core>
│   ├── env
│   ├── sim
│   ├── tb
│   └── tests
├── ci
│   ├── common
│   ├── <core1>
│   ├── <core1>
│   └── ...
└── lib
    ├── sim_libs
    ├── riscv_tests
    ├── uvm_tests
    ├── uvm_agents
    └── uvm_libs
```

4.4 Getting to There from Here

CHAPTER 5

CV32E40P Simulation Testbench and Environment

As stated in the *PULP-Platform Simulation Verification* chapter (in the *Executive Summary*), 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 Instruction Set Simulator (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 as detailed below.

5.1 Core Testbench

The "core" testbench, is essentially the RI5CY testbench (shown in Illustration 1 of *PULP-Platform Simulation Verification*) 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 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.

5.2 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.

5.2.1 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 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*.

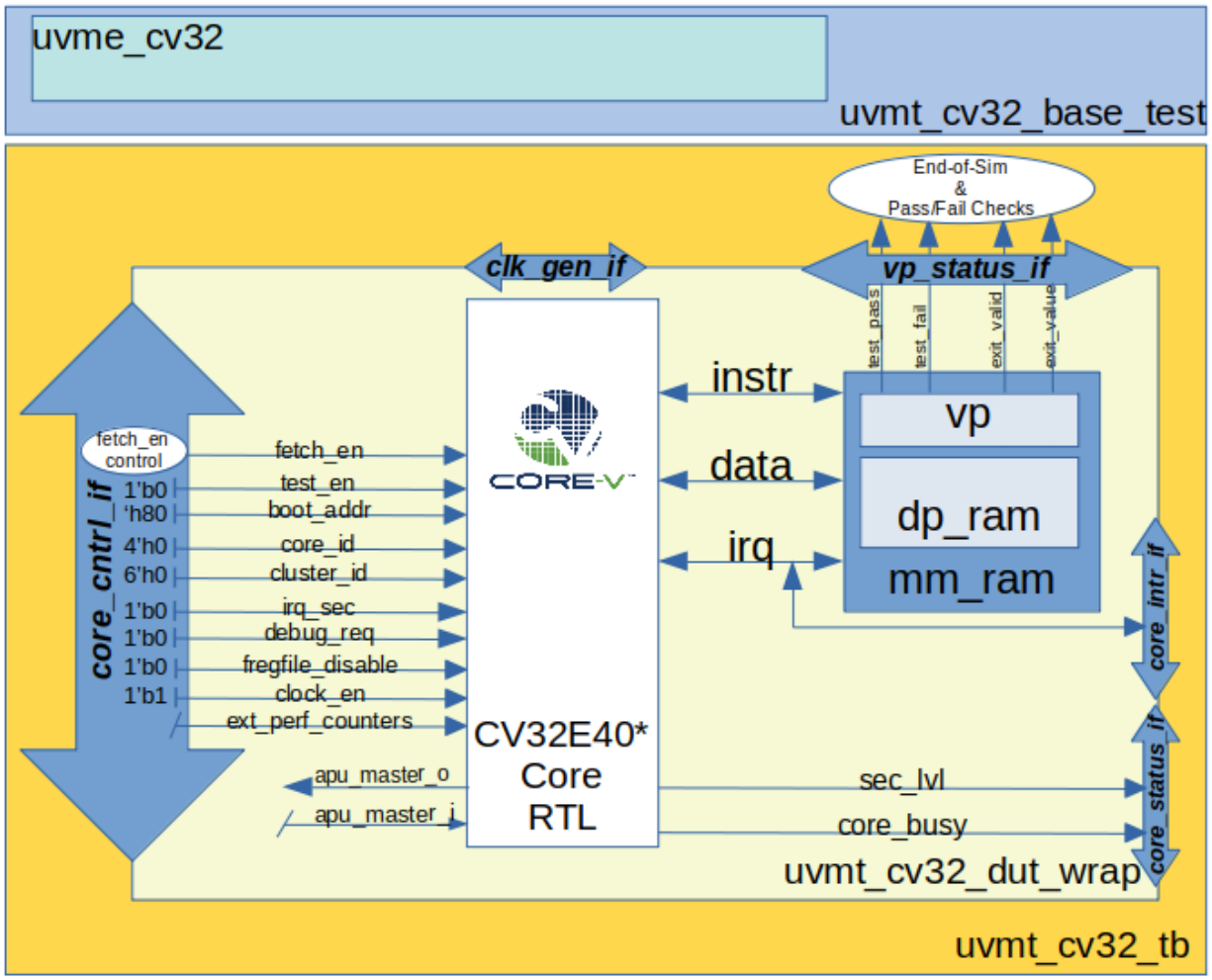

Fig. 1: Illustration 4: Phase 1 CV32E40P UVM Environment

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 , 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.

5.2.2 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.

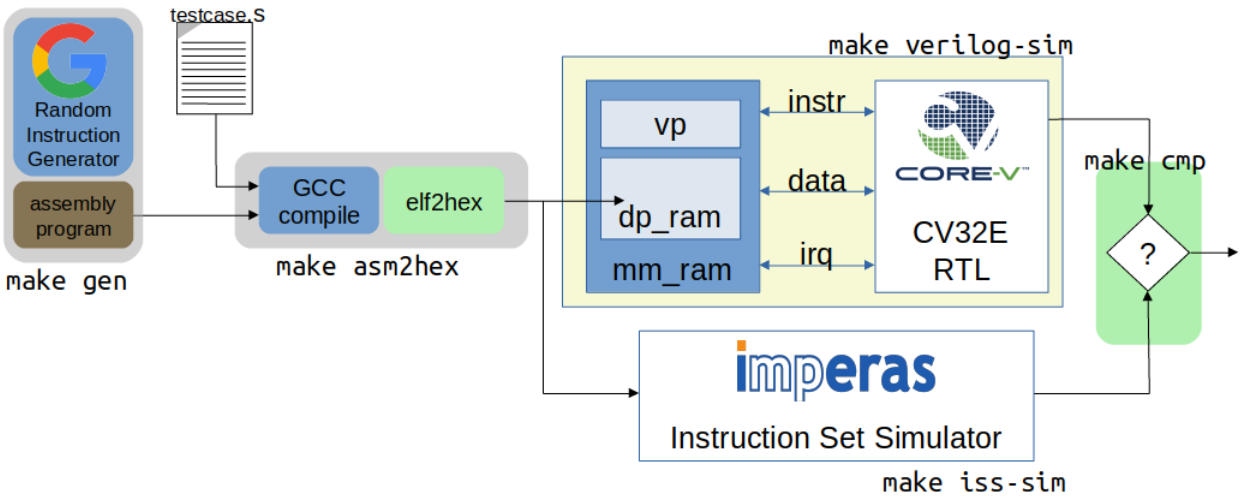

Fig. 2: Illustration 5: Phase 2 Verification Environment for CV32E

As shown in *Illustration 5*, 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*.

[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-docsproject which is home for this document.

- **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, 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.

5.2.3 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 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.

5.2.4 Phase 3 Environment Reference Model (ISS) Integration

Illustration 5 shows the ISS as an entity external to the environment. Phase 3 adds significant capabilies to the environment, notably the integration of the ISS as a fully integrated component in the UVM environment and a **Step-and-Compare** instruction scoreboard. After Phase 3 the Imperas ISS is used as the reference model to predict the status of the core's PC, GPRs and CSRs after each instruction is executed. (Note that after this point in the document the terms "RM" and "ISS" are often used interchangeably.)

Wrapping the RM in a DPI layer allows the RM to be integrated into the UVM environment and thus controllable via the UVM run-flow. The benefit of this is that testcases will have direct control over the operation of the RM and comparision between the predictions made by the RM and actual instruction execution by the Core are done in real time. This is a significant aid to debugging failures.

5.2.5 Step-and-Compare

The integrated RM is used in a step-and-compare mode in which the RM and RTL execution are in lock-step. Step and compare is invaluable for debug because the RM and RTL are executing the same instruction in a compare cycle.

The table below contains the main signals used in stepping and comparing the RTL and RM.

Name	Type	Meaning
step_compare_if.ovp_cpu_retire	event	RM has retired an instruction, triggers ev_ovp event
step_compare_if.riscv_retire	event	RTL has retired an instruction, triggers ev_rtl event
step_ovp	bit	If 1, step RM until ovp.cpu.Retire event
ret_ovp	bit	RM has retired an instruction, wait for compare event. Set to 1 on ovp.cpu.Retire event
ret_rtl	bit	RTL has retired an instruction, wait for compare event. Set to 1 on riscv_tracer_i.retire event
ev_ovp	event	RM has retired an instruction
ev_rtl	event	RTL has retired an instruction
ev_compare	event	RTL and RM have both retired an instruction. Do compare.

Referring to Illustration 6:

1. The simulation starts with step_rtl=1. The RTL throttles the RM.
2. Once the RTL retires an instruction, indicated by ev_rtl, the RM is commanded to Step and retire an instruction, indicated by ev_ovp.
3. The testbench compares the GPR, CSR, and PC after both the RTL and RM have retired an instruction.
4. Once the testbench performs the compare, step_rtl asserts, event ev_compare is triggered, and the process repeats.

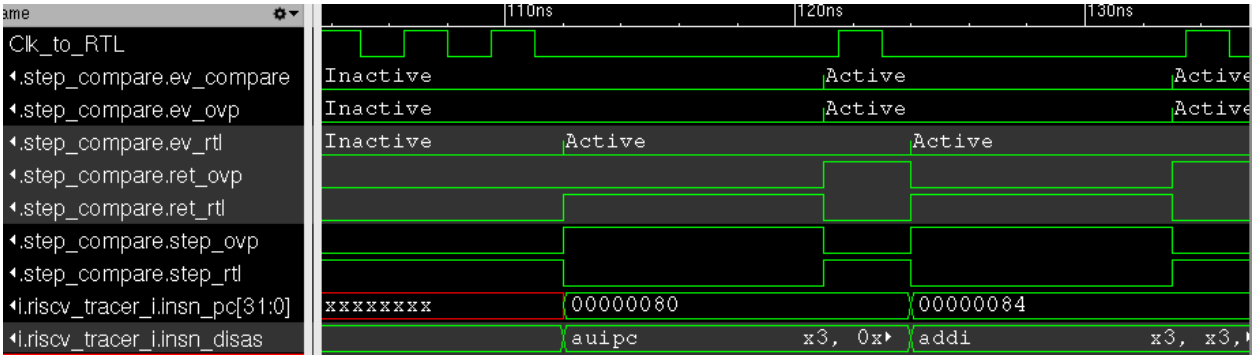

Fig. 3: Illustration 6: Step and Compare Sequencing

Step and compare is accomplished by the *uvmt_cv32_step_compare* module.

Compare

RTL module *riscv_tracer* flags that the RTL has retired an instruction by triggering the *retire* event. The PC, GPRs, and CSRs are compared when the *compare* function is called. The comparison count is printed at the end of the test. The test will call UVM_ERROR if the PC, GPR, or CSR is never compared, i.e. the comparison count is 0.

GPR Comparison

When the RTL retire event is triggered <*gpr*>*_q* may not yet have updated. For this reason RTL module *riscv_tracer* maintains queue *reg_t insn_regs_write* which contains the address and value of any GPR which will be updated. It is assumed and checked that this queue is never greater than 1 which implies that only 0 or 1 GPR registers change as a result of a retired instruction.

Illustration 7 demonstrates that for a *lw x11, -730(x11)* instruction the GPR value is updated one clock cycle after the RTL retire signal. The load to *x11* is retired but RTL value *riscy_GPR[11]* has not updated to *0x075BCD15* yet. However, the queue *insn_regs_write* has been updated and is used for the compare. It is assumed that all other RTL GPR registers are static for this instruction and can be compared directly.

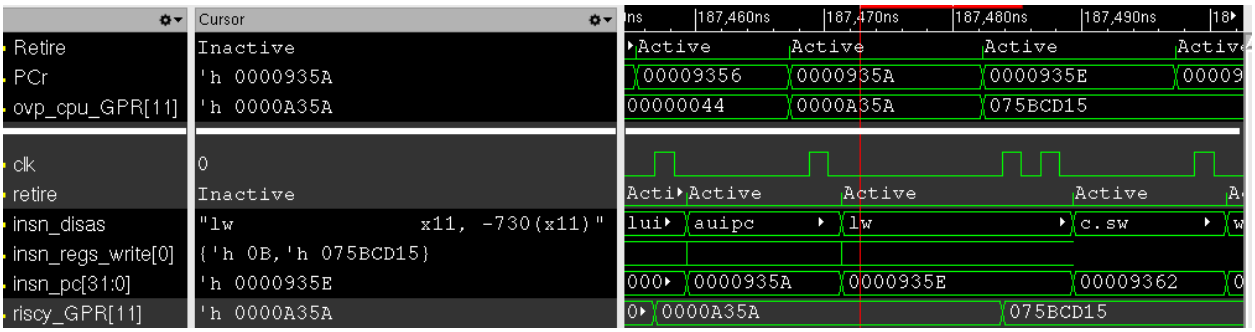

Fig. 4: Illustration 7: Purpose of queue insn_regs_write

If the size of queue *insn_regs_write* is 1 the GPR at the specified address is compared to that predicted by the RM. The remaining 31 registers are then compared. For these 31 registers, <*gpr*>*_q* will not update due to the current retired instruction so <*gpr*>*_q* is used instead of *insn_regs_write*.

If the size of queue *insn_regs_write* is 0 all 32 registers are compared, <*gpr*>*_q* is used for the observed value.

CSR Comparison

When the RTL retire event is triggered the RTL CSRs will have updated and can be probed directly. At each Step the RM will write the updated CSR registers to array *CSR* which is an array of 32-bits indexed by a string. The index is the name of the CSR, for example, *mstatus*. Array *CSR* is fully traversed every call of function *compare* and compared with the relevant RTL CSR. A CSR that is not to be compared can be ignored by setting bit *ignore=1*. An example is *time*, which the RM writes to array *CSR* but is not present in the RTL CSRs.

5.2.6 Beyond Phase 3 Environment

At the time of this writing (2020-04-21) there is a proposal to develop a CV32E40P Subsystem, comprized of the Core, a Debug Module and Debug Transport Module, plus a cross-bar which will allow for Debug Module and an external AHB master to access instruction and data memory. Details of this Subsystem can be found in the Architecture Specification for the Open Bus Interface.

Illustration 8 shows a simple (?) change to the **uvmt_cv32_tb** that allows the testbench (and thereby the UVM enviroment) to switch between a Core-level DUT and a Subsystem-level DUT.

Here, the memory model **mm_ram** has been moved from the dut_wrap module to the testbench module. The connection between the memory model and the dut_wrap is via new SystemVerilog interfaces, **itcm** and **dtcm**. These SystemVerilog interfaces support both the Core-level instruction and data interfaces as well as the OBI instruction and data interfaces. This is possible because the OBI standard is a super-set of the Core's interfaces. Any difference in operation between these interfaces is controlled at compile time[11].

In *Illustration 9* the **uvmt_cv32_dut_wrap** (or core wrapper) is replaced with **uvmt_cv32_ss_wrap** (subsystem wrapper). This subsystem wrapper has the same SystemVerilog interfaces as the core wrapper and instantiates the CV32E40* Subsystem directly. For Core-level testing, the the OBI XBAR and DM_TOP modules are replaced with

[11] The memory model is currently implemented as a SystemVerilog module. Replacing this with a SystemVerilog class based implementation would allow for run-time control of the SystemVerilog interface operation. This is a nice-to-have feature and is not, on its own, enough of a reason to re-code the memory model.

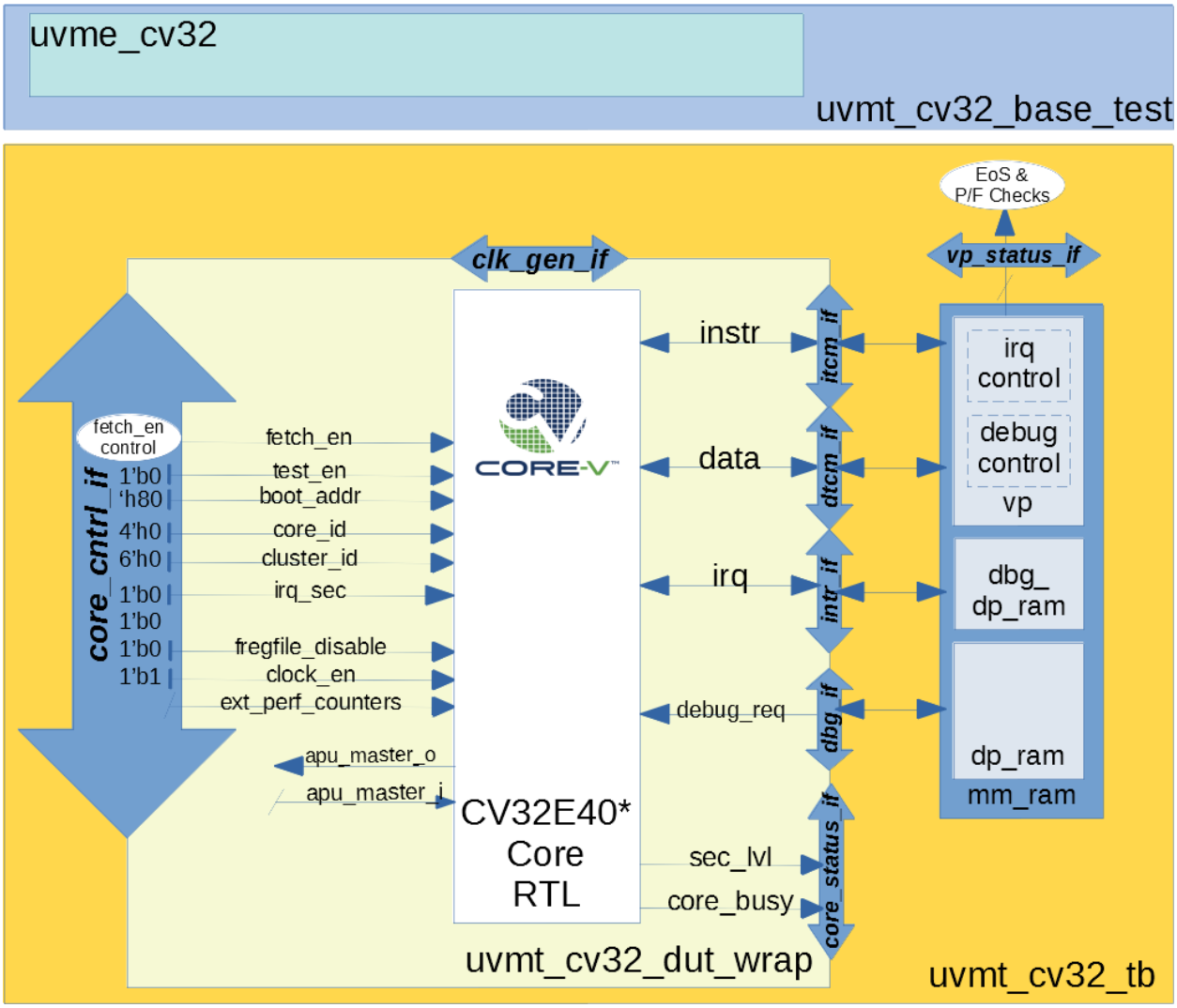

Fig. 5: Illustration 8: Moving Memory Model to the Testbench

"shell" modules at compile-time. The XBAR shell is a pass-through for the instruction and data buses to directly connect to itcm_if and dtcm_if respectively. Likewise, the DM is also replaced with a shell that drives Hi-Z on its debug_req output, thereby allowing debug_req to be driven directly from the dbg_if. The DM shell drives the ready output on the DMI low to ensure that the Debug Agent (in the UVM environment, not shown in the Illustration) does not inadvertently attempt debug access via the DMI. Instead, the Debug Agent is configured to drive debug_req directly.

Testing the Subsystem involves no compile-time changes to the UVM environment as it is able to use the same SystemVerilog interfaces. The run-time configuration is changed such that the Debug Agent drives Hi-Z on its debug_req ouput and all accesses to the DM use the DMI signalling. At compile-time the RTL for the OBI XBAR and DM modules are instantiated. The AHB master and slave interfaces of the Subsystem (not shown in the Illustration) connect to their own SystemVerilog interfaces which connect to AHB Agents in the UVM environment. If the wrapper has been compiled to instantiate just the Core, these AHB Agents are configured to be inactive.

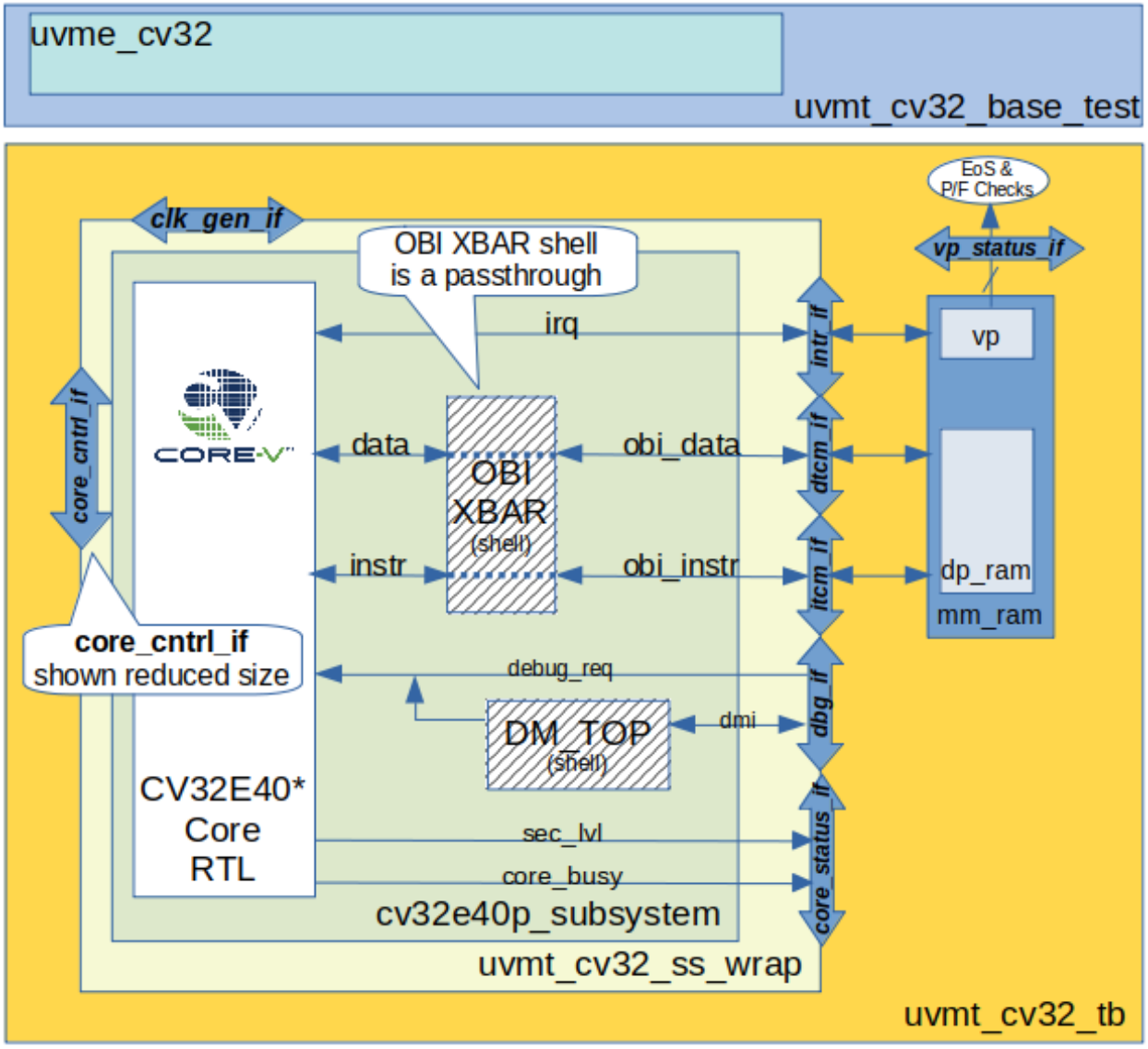

Fig. 6: Illustration 9: Subsystem Wrapper (compiled for Core-level verification)

5.3 File Structure and Organization

ToDo

5.3.1 Naming Convention

5.3.2 Directory and File Structure

5.3.3 Compiling the Environment

CHAPTER 6

CV6A Simulation Testbench and Environment

TODO.

CHAPTER 7

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 CVA6 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.

7.1 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[12].

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 *RI5CY Testcases* 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.

[12] 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.

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[13]" virtual peripheral for pass/fail information.
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[14]. 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.

7.1.1 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 4: Phase 1 CV32E40P UVM Environment*. In addition to supporting the instruction, data memory (*dp_ram*), and debug memory (*dbg_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 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 debug control virtual peripheral is used by a test program to control the debug_req signal going to the core. The assertion can be a pulse or a level change. The start delay and pulse duration is also controllable. Once the debug_req is seen by the core, it will enter debug mode and start executing code located at DM_HaltAddress, which is mapped to the debug memory (*dbg_dp_ram*).

The debug memory is loaded with a hex image defined with the plusarg +debugger=<filename.hex>

[13] See Section *Virtual Peripherals*.
[14] Generation of illegal or malformed instructions is also supported, and will be discussed in a later version of this document.

If the +debugger plusarg is not provided, then the debug memory will have a single default instruction, dret, that will result in the core returning back to main execution of the test program. The debug_test is an example of a test that will use the debug control virtual peripheral and provide a specific debugger code image.

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	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.
Debug Control	32'h1500_0008	Asserts the debug_req signal to the core. debug_req can be a pulse or a level change, with a programable start delay and pulse duration as determined by the wdata fields:
		wdata[31] = debug_req signal value
		wdata[30] = debug request mode: 0= level, 1= pulse
		wdata[29] = debug pulse duration is random
		wdata[28:16] = debug pulse duration or pulse random max range
		wdata[15] = start delay is random
		wdata[14:0] = start delay or start random max rangee
Random Number Generator	32'h1500_1000	Reads return a random 32-bit value with generated by the simulator's random number generator. Writes have no effect.
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

7.2 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.

7.2.1 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.

7.3 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*[15]. 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.

Section *Test Program* introduced the five types of core test programs supported by the CORE UVM environment and section *UVM Test* 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 8 shows how the CORE-V UVM base test supports a type 1 test program.

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).

[15] This is termed Execution Environment Interface or EEI by the RISC-V ISA.

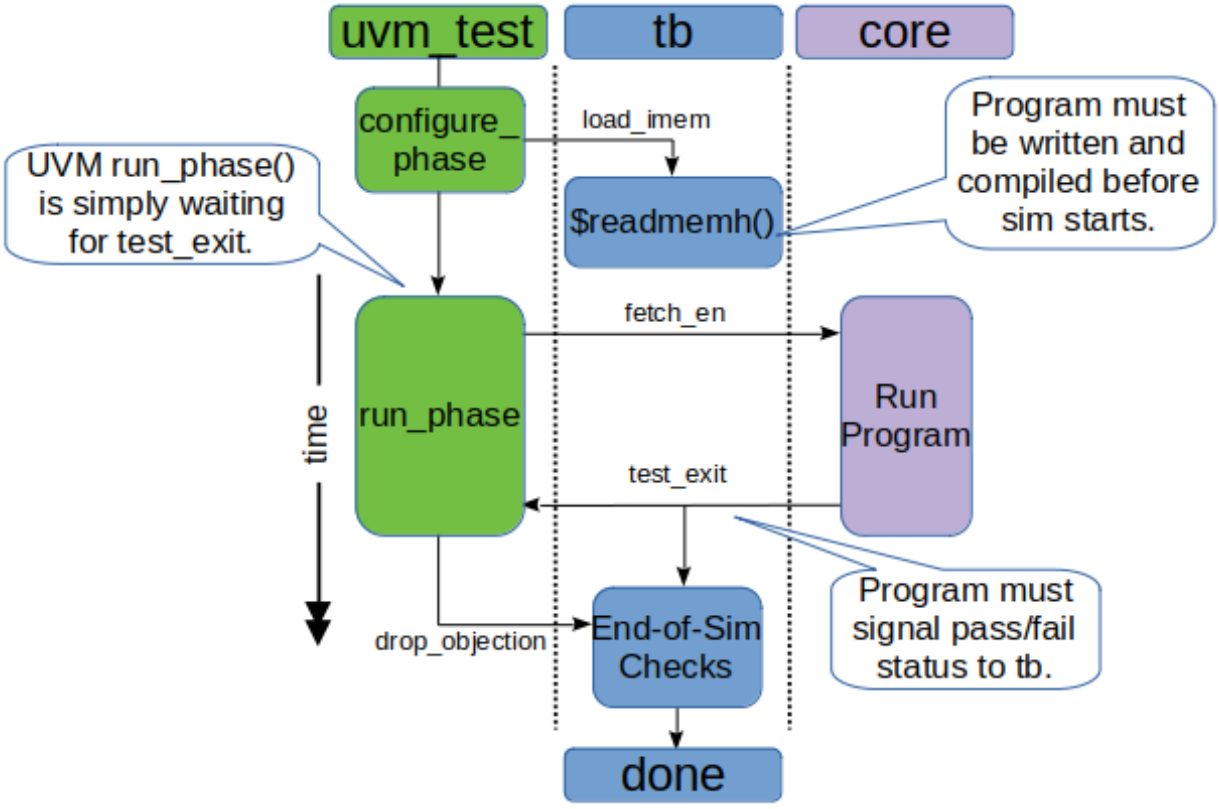

Fig. 1: Illustration 8: Preexisting, Self-checking Test Program (type 1) in a CORE-V UVM test

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. 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.

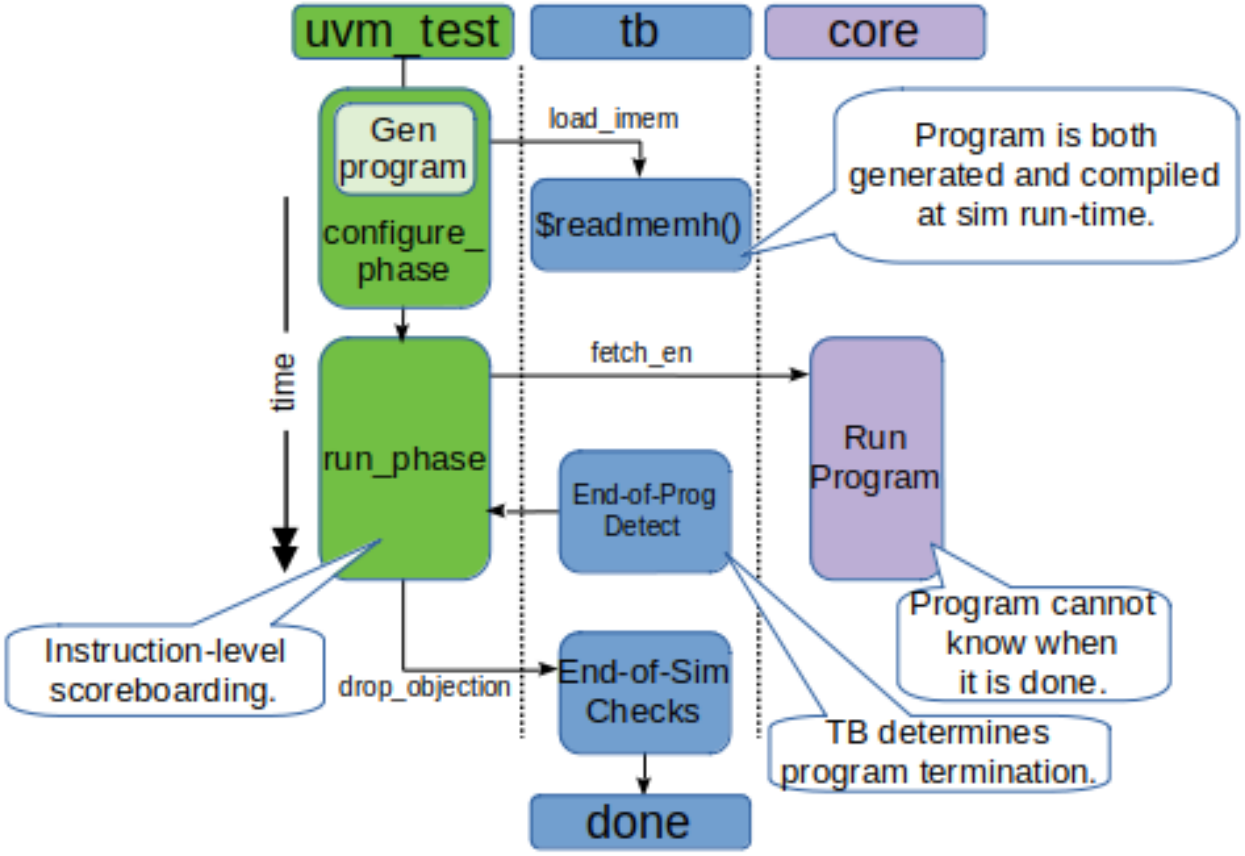

Fig. 2: Illustration 9: Generated, non-self-checking (type 4) Test Program in a CORE-V UVM test

The flow for a type 4 (generated, non-self checking) test program is only slightly different as shown in Illustration 9. 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 . Note that most UVM tests that run self-checking test programs will also use the ISS as part of its pass/fail determination.

7.4 CORE-V Testcase Writer's Guide

TODO

7.4.1 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
```

7.4.2 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**.

7.4.3 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 *Test Program*.

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.

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.

7.4.4 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.

CHAPTER 8

Test Program Environment (BSP)

The purpose of this chapter is to define the "programming environment" of a test program in the CORE-V verification environments. The current version of this document is specific to the CV32E40P. Further versions will be sufficiently generic to encompass all CORE-V cores.

Software teams will generally use the term **Board Support Package** or BSP to refer to what this document calls the **Test-Program Environment** or TPE. If you are familiar with BSPs then you may consider that term interchangable with TPE for the remainder of the discussion below.

Recall from *Simulation Tests in the UVM Environments* that a "test program" is set of RISC-V instructions that are loaded into the testbench memory and executed by the core RTL model. Test-program are typically written in C or RISC-V assembler and can be either human or machine generated. In either case it needs to be "aware" of the hardware environment supported by the core and its testbench.

Illustration 9 uses the Core testbench as an example to illustrate the relationship between the testbench (everything inside the yellow rectangle), the test program (testcase.S) and a test program environment (crt0.S and link.ld). The UVM verification environment will use the same test program environment as the Core testbench.

This linkage between the test-program and hardware needs to be flexible to support a variety of test-program sources:

- manually written assembler and C test-programs inherited from RI5CY
- test-programs from the RISC-V Foundation Compliance Test Suite
- manually written OpenHW test-programs
- machine generated test-programs from an instruction generator (e.g. riscv-dv)

Test-programs, the toolchain that translates them into machine code and the testbench all have to be compatible with one-another.

8.1 Hardware Environment

The testbench supports an instruction and data memory plus a set of memory mapped virtual peripherals. The address range for I&D memory is 0x0..0x40_0000 (4Mbyte) for both the core and UVM testbenches. The virtual peripherals start at address 0x1000_0000.

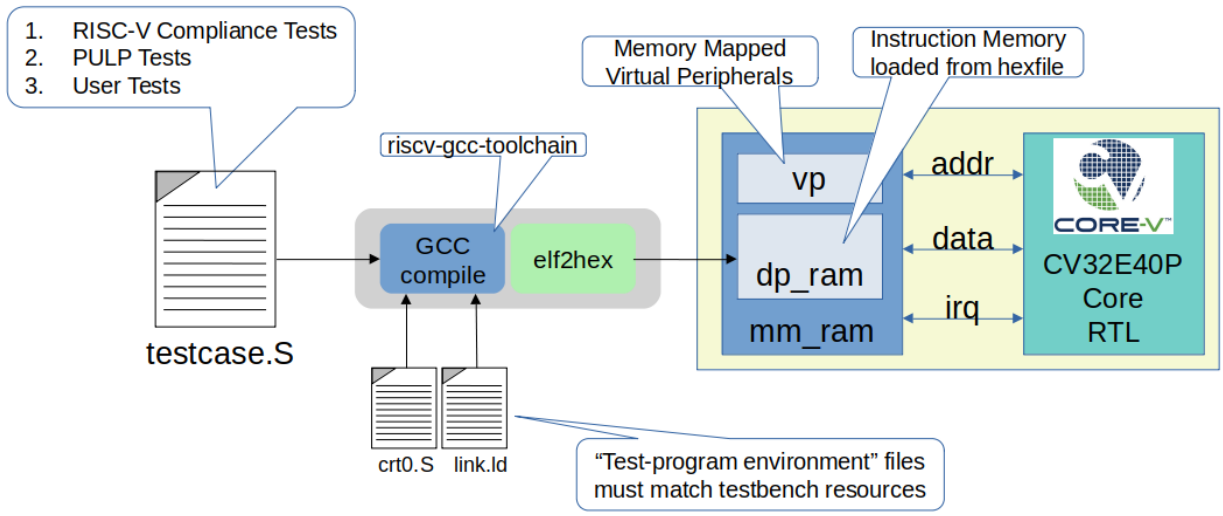

Fig. 1: Illustration 9: Test Program Environment for "Core" testbench

The addresses and sizes of the I&D memory and virtual peripheral must be compatible with the Configuration inputs of the core (see Core Integration in the CV32E40P User Manual. The core will start fetching instructions from the address provided on its **boot_addr_i** input. In addition, if debug_req_i is asserted, execution jumps to **dm_halt_addr_i**. This hardware setup constrains the test-program in important ways:

- The entire program, including data sections and exception tables must fit in a 4Mbyte space starting at address 0.
- The first instruction of the program must be at the address defined by **boot_addr_i**.
- The address **dm_halt_addr_i** must exist in the memory map, it should not be stomped on by the test-program and there should be something there to execute that will produce a predictable result.
- The program must "know" about the addressing and function of the virtual peripherals (using the peripherals is optional).

8.2 Test-Programs

Most of our test-programs are written/generated in RISC-V assembler. A set of global symbols are used to define control points to a linker that will generate the machine-code in a memory image. Examples of these are **.start**, **.vectors**, **.text**, **.data** and **.string**. Here we will define a minimal set of symbols for use in CORE-V test-programs. A sub-set of these will be mandatory (e.g. **.start**), while others may be optional.

8.3 Aligning the Test-Programs to the Hardware Environment

Beyond the hardware, there are a number of files that define the test program environment. These are discussed below.

8.3.1 Linker Control File

A file variously refered to as the linker command file, linker control file or linker script and typically given the filename **link.ld** is used to map the symbols used in the test-program to physical memory addresses. Some excellent background material on the topic is available at Sourceware.org.

Typically, linker scripts have two commands, **MEMORY** and **SECTIONS**. If MEMORY is not present then the linker assumes that there is sufficient contiguous memory to hold the program. We are constrained by a need to support the Compliance test-suite and the Google generator, so it is possible we need more than one linker control file, although a single script for all uses should be out goal.

Jeremy Bennett of Embecosm has provided a minimalist linker control file, and Paul Zavalney of Silicon Labs suggested additions to support the debugger. The two contributions have been merged into a single script by Mike Thompson:

```
OUTPUT_ARCH( "cv32e40p" )
ENTRY(_start)

MEMORY
{
   /* This matches the physical memory supported by the testbench    */
   mem (rwxai) : ORIGIN = 0x00000000, LENGTH = 0x00100000

   /* ORIGIN must match the DM_HALTADDRESS parameter in the core RTL */
   dbg (rwxai) : ORIGIN = 0x1A110800, LENGTH = 0x800
}

SECTIONS
{
   DBG :
   {
      .debugger (ORIGIN(dbg)):
      {
        KEEP(*(.debugger));
      }

      /* Debugger Stack*/
      .debugger_stack         : ALIGN(16)
      {
       PROVIDE(__debugger_stack_start = .);
       . = 0x80;                    /* Is this ORIGIN + 0x80? */
      }
   } >dbg

   MEM :
   {
      . = 0x00000000;
      .vectors : { *(.vectors) }
      . = 0x00000080;                   /* must equal value on boot_addr_i */
      _start = .;
      .text : { *(.start) }
      . = ALIGN (0x80)
      .legacy_irq : { *(.legacy_irq) }  /* is this still needed? */
      . = ALIGN(0x1000);
      .tohost : { *(.tohost) }
      . = ALIGN(0x1000);
      .page_table : { *(.page_table) }
      .data : { *(.data) }
      .user_stack : { *(.user_stack) }
      .kernel_data : { *(.kernel_data) }
      .kernel_stack : { *(.kernel_stack) }
      .bss : { *(.bss) }
      _end = .;
   } > mem
```

(continues on next page)

(continued from previous page)

```
}
```

A few open issues:

1. How does the linker control file need to change to support interrupts?

   At the time of this writting (2020-07-07), this is an area of active development in the CV32E40P projects. This document (or its associated README) will be updated at a later date.

2. Will this linker script fully support test-programs generated by the Google generator (riscv-dv)?

   This issue has been resolved by extending the riscv_asm_program_gen class in the corev-dv extensions.

8.3.2 C Runtime

While it is assumed that the vast majority of test programs written for CORE-V pre-silicon verification will be captured as assembly (*.S) programs, The environment provides support for minimalist C programs via a C runtime file in *./cv32/bsp/crt0.S*[16]. crt0.S performs the bare minimum required to run a C program. Note that **support for command-line arguments is deliberately not supported**.

[16] Additional information on the "Board Support Package" can be found in its associated README in the core-v-verif GitHub repository.

CHAPTER 9

CORE-V Formal Verification

Formal verification of the CV32E and CVA6 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 CVA6 formal verification.

9.1 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).

9.2 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.

9.2.1 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.

9.2.2 Development of Sail Models for CORE-V Cores

At the time of this writing[17], 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 CVA6. 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 CVA6 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.

9.3 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: CV32E40P Formal Verification Work Items

[17] First week of January, 2020.

#	Work Items	Provided By	Comment
1	CV32E40P User Manual	OpenHW Group	Viewable at readthedocs: https://core-v-docs-verif-strat.readthedocs.io/projects/cv32e40p_um/en/latest/
2	ISA Sail Models	OpenHW Group	Based on the RV64IMAC Sail model developed by the RISC-V Foundation. Status information as of 2020-04-20 at https://github.com/rems-project/sail-riscv/blob/master/doc/Status.md
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 *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 Testplan	OpenHW Group and OneSpin Solutions	ToDo: work with OneSpin to define template.
7	Delivery of Open Source assertion model to support Formal Testplan	OneSpin Solutions	
8	Formal Verification of Cores	OpenHW Group and OneSpin Solutions	See the sub-section *7.4*.

9.3.1 Specifications

See rows #1 and #2 in , 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 CVA6 cores, including the PULP ISA extensions.

9.3.2 Compute and Tool Resources

This is rows #4 and #5 in , above. Tool licenses in sufficient numbers to allow for "reasonable" regression turn-around time on CVA6 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.

9.3.3 Formal Testplans

OpenHW and OneSpin will jointly develop Formal Testplans for both the CV32E and CVA6. 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

9.3.4 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 CVA6 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[18].

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.

9.4 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;

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

- **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.

CHAPTER 10

CORE-V FPGA Prototyping

ToDo. This may be captured in a separate document.