Contents:

1 Introduction 3
   1.1 License ......................................................... 3
   1.2 CORE-V Projects ................................................. 3
   1.3 Definition of Terms ............................................. 5
   1.4 Conventions Used in this Document ............................ 6
   1.5 CORE-V Genealogy .............................................. 6
   1.6 A Note About EDA Tools ........................................ 6

2 CORE-V-VERIF Quick Start Guide 7
   2.1 Where is the RTL? .............................................. 8
   2.2 UVM ............................................................. 8
   2.3 Doing More in CORE-V-VERIF ................................. 9

3 Verification Planning and Requirements 13

4 CORE-V Verification Environment 15
   4.1 Environment Structure ......................................... 16
   4.2 Repository Structure .......................................... 17

5 CV32E40P Simulation Testbench and Environment 19
   5.1 Core Testbench ............................................... 19
   5.2 The CV32E40* UVM Verification Environment .................. 20
   5.3 File Structure and Organization .............................. 37

6 CVA6 Simulation Testbench and Environment 41

7 Test Programs 43
   7.1 Hardware Environment ......................................... 43
   7.2 Virtual Peripherals .............................................. 44
   7.3 Board Support Package ........................................ 45
   7.4 Debug Mode Support ............................................ 48
   7.5 Interrupt Support .............................................. 48

8 UVM Testcases in the CORE-V-VERIF Environments 49
   8.1 Test-Programs in the CORE-V-VERIF UVM Environment ......... 49
   8.2 UVM Test ....................................................... 50
   8.3 Run-flow in a CORE-V Test .................................... 50
8.4 CORE-V Testcase Writer’s Guide .................................................. 53

9 COREV-DV ............................................................................. 55
  9.1 Using COREV-DV .......................................................... 55
  9.2 Extending COREV-DV ................................................... 55

10 PULP-Platform Simulation Verification ........................................... 57
  10.1 Executive Summary ......................................................... 57
  10.2 RI5CY ................................................................. 58
  10.3 Ariane ................................................................. 61
  10.4 IBEX ................................................................. 63
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

Copyright 2020, 2021 OpenHW Group.

The document is licensed under the Solderpad Hardware License, Version 2.0 (the “License”); you may not use this document except in compliance with the License. You may obtain a copy of the License at:

https://solderpad.org/licenses/SHL-2.0/

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

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

### 1.2 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.
• CV32E40X RTL source
• CV32E40X user manual
• CV32E40S RTL source
• CV32E40S user manual
• CVA6 RTL source
• CVA6 user manual

The OpenHW Group also maintains the following repositories for stand-alone verification components:

• FORCE-RISCV Instruction stream generator denoted by Futurewei.
## 1.3 Definition of Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSP</td>
<td>Board Support Package. 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 environment used by a test-program.</td>
</tr>
<tr>
<td>Committer</td>
<td>A Contributor who has privileges to approve and merge pull-requests into OpenHW Group GitHub repositories.</td>
</tr>
<tr>
<td>Contributor</td>
<td>An employee of a Member Company that has been assigned to work on an OpenHW Group project.</td>
</tr>
<tr>
<td>CORE-V</td>
<td>A family of RISC-V cores developed by the OpenHW Group.</td>
</tr>
<tr>
<td>ELF</td>
<td>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.</td>
</tr>
<tr>
<td>Instruction Set Simulator (ISS)</td>
<td>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.</td>
</tr>
<tr>
<td>Member Company (MemberCo)</td>
<td>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.</td>
</tr>
<tr>
<td>Toolchain</td>
<td>A set of software tools used to compile C and/or RISC-V assembler code into an executable format.</td>
</tr>
<tr>
<td>Testbench</td>
<td>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.</td>
</tr>
<tr>
<td>Testcase</td>
<td>In the context of the CORE-V UVM verification environment, a a testcase is distinct from a testprogram. A testcase is extended from the uvm_test class and is used to control the the UVM environment at run-time. In core-v-verif a testcase may be aware of the test-program.</td>
</tr>
<tr>
<td>Test-Program</td>
<td>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). In core-v-verif a test-program is not aware of the UVM testcase.</td>
</tr>
<tr>
<td>TPE</td>
<td>Test-program Environment. A less widely used term for BSP.</td>
</tr>
<tr>
<td>Verification Environnement</td>
<td>An object constructed from a SystemVerilog class that is an extension of uvm_environment. In common usage “verification environment” can also mean the environment object plus all of its members.</td>
</tr>
</tbody>
</table>

```bash
$CORE_V_VERIF
```

Path of a cloned working directory of this GitHub repository. An example to illustrate:

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

Here $CORE_V_VERIF is /wrk/rick/openhw/core-v-verif. Note that this is not a variable the user is required to set. Its use in this document is merely used as a reference point for an absolute path to your working directory.

```bash
$COREV
```

Shell and Make variable identifying a specific CORE-V core. The most often used example in this document is CV32E40P.
1.4 Conventions Used in this Document

**Bold** type is used for emphasis.

Filenames and filepaths are in italics: ./cv32e40p/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 CV32E40P and CVA6. Currently, two variants of the CV32E40P are defined: the CV32E40X and CV32E40S. The OpenHW Group’s work builds on several RISC-V open-source projects, particularly the RI5CY and Ariane projects from PULP-Platform. CV32E40P is a derivation of the RI5CY project, and CVA6 is derived from Ariane. In addition, the verification environment for CORE-V leverages previous work done by lowRISC and others for the Ibex project, which is a fork of the PULP-Platform’s zero-riscy core.

This is germane to this discussion because the architecture and implement of the verification environments for both CV32E40P and CVA6 are strongly influenced by the development history of these cores. This is discussed in more detailed in *PULP-Platform Simulation Verification*.

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 CORE-V 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.
Many people who come to core-v-verif for the first time are anxious to ‘get something running’ and this section is written to satisfy that itch.

Note: in several places in this chapter a reference is made to $CORE_V_VERIF. This is used as short hand for the absolute path to your local working directory. You will not need to set this shell environment variable yourself.

A good place to start is the CV32E40P core testbench since the CV32E40P is mature, and the core testbench is very simple and runs with open-source tools. You will need:

1. A Linux machine (core-v-verif has been successfully run under Ubuntu, Debian and CentOS).

2. Python3 and a set of plug-ins. The current plug-in list is kept in $CORE_V_VERIF/bin/requirements.txt. The easiest way to get these requirements installed on your machine is:

   $ cd $CORE_V_VERIF/bin
   $ pip install -r requirements.txt

3. A GCC cross-compiler (aka “the Toolchain”). Even if you already have a toolchain, please do follow that link and read TOOLCHAIN.md for recommended ENV variables to point to it.

4. Verilator.

Once the above is in place type the following:

   $ git clone https://github.com/openhwgroup/core-v-verif.git
   $ cd core-v-verif/cv32e40p/sim/core
   $ make

The above will compile the RTL and the core testbench using Verilator and run the ‘hello-world’ test-program. The simulation run should produce the following:

HELLO WORLD!!!
This is the OpenHW Group CV32E40P CORE-V processor core.
CV32E40P is a RISC-V ISA compliant core with the following attributes:
   mvendorid = 0x602
   marchid = 0x4

(continues on next page)
mimpid = 0x0
misa = 0x40001104
XLEN is 32-bits
Supported Instructions Extensions: MIC

The README in the cv32e40p/sim/core directory will explain how to run other tests and use other simulators.

2.1 Where is the RTL?

As core-v-verif is a single repository for the verification of several different CORE-V cores, you will not find the RTL here. Each CORE-V core is managed in its own repository, and is automatically cloned into $CORE_V_VERIF/cores/<core_name> by the Makefiles, as needed. If you have successfully run the hello-world test above, then you may have noticed that the RTL was cloned to $CORE_V_VERIF/cores/cv32e40p/rtl (have a look!).

2.2 UVM

Up to this point, the discussion has been about the “core” testbench, which is intended as demonstration vehicle only. The primary verification environments implemented in core-v-verif are all based on the UVM. The UVM environment for CV32E40P is completely separate from the core testbench and uses a different set of Makefiles.

In order to use the UVM environments you will need items 1, 2 and 3 from the list above, plus a SystemVerilog simulator capable of supporting UVM and the Imperas OVPsim Instruction Set Simulator (ISS). You should also review the README in $CORE_V_VERIF/mk/uvmt for a description of the shell ENV vars used by the UVM environment. With these in place you can do the following:

```bash
$ git clone https://github.com/openhwgroup/core-v-verif.git
$ cd core-v-verif/cv32e40p/sim/uvmt
$ make test TEST=hello-world CV_SIMULATOR=<xrun|vcs|vsim|riveria|dsim>
```

The above will compile the RTL and the UVM testbench using the selected simulator and run the ‘hello-world’ test-program. Note that this is the same test-program as was used in the core testbench example above. The simulation run will produce similar results, with lots of additional UVM messaging.

If you do not have access to the Imperas ISS you can disable it at run-time:

```bash
$ make test TEST=hello-world CV_SIMULATOR=<xrun|vcs|vsim|riveria|dsim> USE_ISS=NO
```

2.2.1 Why UVM?

Core-v-verif was specifically created to bring industrial practises to bear for CORE-V verification. The UVM is by far the most popular methodology used in dynamic (simulation) based verification today.

2.2.2 Do I need an ISS?

The short answer is an emphatic yes. Core-v-verif uses an Instruction Set Simulator (ISS) as a reference model of the CORE-V core (the Device Under Test). A key feature of the core-v-verif UVM environments is that the state of the DUT is compared to the state of the reference model after each instruction is retired. Without a comparison to a reference model, the pass/fail status of a given simulation is mostly vacuous.
There are two popular options for a RISC-V ISS, Spike <https://github.com/riscv-software-src/riscv-isa-sim> and Imperas OVPsim <https://www.ovpworld.org/technology_ovpsim> At the time of this writing (2021-12-07) core-v-verif uses a commercial version of Imperas OVPsim for the CV32E4 cores. A contribution to integrate another reference model into core-v-verif would be welcome.

2.3 Doing More in CORE-V-VERIF

As far as is practical, core-v-verif maintains “in place” documentation. That is, most directories will have a README.md that provides information relevant to that directory and/or its sub-directories. GitHub renders these to HTML automatically, and so this document will point you to a lot of this type of content.

At the top-level of core-v-verif, there is a subdirectory for each supported core. All of the verification code specific to that core will be in this directory. Look at the README in cv32e40p to get a sense of the directory structure of the cv32e40p-specific testbenches. The structure of the other cores will be similar, but need not be identical.

The cv32e40p sub-tree supports a simple “core” testbench and a complete UVM verification environment. Partial instructions to run the core testbench are provided above; see the README at $CORE_V_VERIF/cv32e40p/sim/core for full details. To run the CV32E40P UVM environment, go to $CORE_V_VERIF/cv32e40p/sim/uvmt and read the README.

This chapter uses the CV32E40P as its example, but there are equivalent READMEs in directories for the other supported cores.

2.3.1 Supported Simulators

It is a goal of core-v-verif to support all known SystemVerilog 1800-2017 compliant simulators. The Makefiles for the UVM environments have a variable CV_SIMULATOR which is used to select the simulator used to compile and run a testcase. So you can run hello-world with Cadence Xcelium like this:

```bash
$ make test TEST=hello-world CV_SIMULATOR=xrun
```

To run the same test with Metrics Dsim:

```bash
$ make test TEST=hello-world CV_SIMULATOR=dsim
```

The variable is used to select one of a set of simulator-specific Makefiles that are located at $CORE_V_VERIF/mk/uvmt.

Note that core-v-verif tries to support all simulators and this requires support from OpenHW Group members. From time to time a Makefile for a specific simulator will not see a lot of use and will inevitably suffer from bit-rot. If you notice an issue with a simulator-specific Makefile, please do raise an issue.

2.3.2 Verifying other Cores

At the time of this writing (2021-12-17), core-v-verif supports verification of multiple CORE-V cores:

- **CV32E40P**: UVM environment is stable and v1.0.0 is complete. Work on v2.0.0 has started. A simple “core” testbench which can be run with open-source tools is available.
- **CV32E40X**: UVM environment is stable, and verification is on-going.
- **CV32E40S**: UVM environment is stable, and verification is on-going.
- **CVA6**: UVM environment is in the early stages of development.
2.3.3 CV32E40P Directory Tree (simplified)

Below $CORE_V_VERIF$ you will find a directory named $cv32e40p$. This directory contains all of the CV32E40P-specific sources to compile and run simulations on the CV32E40P CORE-V core. The tree below is a somewhat simplified expansion of the directory highlighting the names, locations and purposes of key directories and files. Other cores, e.g. CV32E40X will implement a similar directory tree.

```
cv32e40p
  |--- bsp // Board-support Package
  |     |--- docs
  |     |     |--- VerifPlans
  |     |--- env // UVM environment
  |     |     |--- corev-dv
  |     |     |     |--- uvme
  |     |     |     |     |--- cov
  |     |     |     |     |--- vseq
  |     |     |--- regress // Regression configurations
  |     |--- sim
  |     |     |--- README.md
  |     |     |--- Common.mk
  |     |     |--- core // Place to run simulations of the
  |     |     |     |--- "core" testbench
  |     |     |     |     |--- TOOLCHAIN.md
  |     |     |     |     |--- tools
  |     |     |     |     |     |--- uvmt // Place to run simulations of the
  |     |     |     |     |     |     |--- Makefile
  |     |     |     |     |     |     |--- README.md
  |     |     |     |--- "uvm" environment
  |     |     |     |     |--- tb
  |     |     |     |     |     |--- README.md
  |     |     |     |     |     |--- core // the "core" testbench
  |     |     |     |     |     |     |--- dp_ram.sv
  |     |     |     |     |     |     |--- mm_ram.sv
  |     |     |     |     |     |     |--- tb_top.sv
  |     |     |     |     |     |     |--- tb_top_verilator.cpp
  |     |     |     |     |     |     |--- tb_top_verilator.sv
  |     |     |     |     |     |--- uvmt // the UVM environment
  |     |     |     |     |     |     |--- uvmt_cv32e40p_constants.sv
  |     |     |     |     |     |     |--- uvmt_cv32e40p_tb.sv
  |     |     |     |     |     |     |--- ...
  |     |     |     |     |     |     |--- uvmt_cv32e40p_pkg.sv
  |     |     |     |--- tests // test-programs and UVM testcases.
  |     |     |     |     |--- cfg
  |     |     |     |     |     |--- default.yaml
  |     |     |     |     |     |--- no_pulp.yaml
  |     |     |     |     |     |--- num_mhpmcounter_29.yaml
  |     |     |     |     |     |--- ovpsim_no_pulp.ic
  |     |     |     |     |     |--- pulp.yaml
  |     |     |     |     |--- programs
  |     |     |     |     |     |--- corev-dv // configurations for randomly
  |     |     |     |     |     |     |--- generated test-programs
  |     |     |     |     |     |     |     |--- corev_rand_arithmetic_base_test
  |     |     |     |     |     |     |     |     |--- corev-dv.yaml
  |     |     |     |     |     |     |     |     |--- test.yaml
  |     |     |     |     |     |     |     |     |--- ...
  |     |     |     |     |     |     |--- corev_rand_jump_stress_test
  |     |     |     |     |     |     |     |--- corev-dv.yaml
```

(continues on next page)
test.yaml
  custom // "custom" (manually written) test-
  programs
    hello-world
      hello-world.c
      test.yaml
    ...
    debug_test
      debug_test.c
      test.yaml
  uvmt // UVM testcase(s) and virtual_
  sequences
  vendor_lib // Libraries from third-parties
    README.md
    google
    imperas
    riscv
    verilab

2.3. Doing More in CORE-V-VERIF
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. Refer to Verification Planning 101 for a tutorial on how verification planning of CORE-V IP is done.

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. In this case, the testbench needs to have these capabilities and its the purpose of the Verification Strategy document to explain how that is done.
A 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 capable of supporting any-and-all CORE-V cores. Rather, core-v-verif supports the verification of multiple cores by enabling the rapid creation of core-specific verification environments. There is no attempt to define a one-size-fits-all environment as these inevitably 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 hierarchy with the device-under-test (the core) at the bottom and testcases at the top. In between are various components with increasing degrees 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 easy 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. hardware loops in CV32E40P).

The core-v-verif project exploits this idea to maximize re-use across multiple cores by striving to keep as much of the environment as possible independent 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 above 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 instantiates 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: - instantiates 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 exception 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.

4.2 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 description 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 $CORE_V_VERIF/mk/uvmt/README.md.

- **core-v-cores**: the Makefiles in the <core>/sim directory will clone the RTL for <core> into core-v-cores/<core>/rtl. This structure allows for the simultaneous verification of multiple cores from the same core-v-verif repository.

- **<core>**: this directory contains the <core> specific environment, testbench, tests and simulation directories.

- **ci**: This directory supports common and core-specific scripts and configuration files to 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.

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

4.2. Repository Structure
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 directory – 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 RISCY 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 1.

Fig. 1: Illustration 1: 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 2. Phase 2 introduces the Google Random Instruction Generator and the Imperas ISS as a stand-alone components. The most significant capabilities of the phase 2 environment are:

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

![Illustration 2: Phase 2 Verification Environment for CV32E](image)

As shown in Illustration 2, 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, 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](https://github.com/openhwgroup/core-v-verif/tree/master/cv32/tests/core) to see what this does. Note that the User Manual for the Verification Environment, which explains how to write and run testcases, will be maintained there, not in the core-v-docs project which is home for this document.
- **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 R15CY 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 as 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 2* shows the ISS as an entity external to the environment. Phase 3 adds significant capabilities 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 comparison 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.
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>step_compare_if.ovp_cpu_retire</td>
<td>event</td>
<td>RM has retired an instruction, triggers ev_ovp event</td>
</tr>
<tr>
<td>step_compare_if.riscv_retire</td>
<td>event</td>
<td>RTL has retired an instruction, triggers ev_rtl event</td>
</tr>
<tr>
<td>step_ovp</td>
<td>bit</td>
<td>If 1, step RM until ovp.cpu.Retire event</td>
</tr>
<tr>
<td>ret_ovp</td>
<td>bit</td>
<td>RM has retired an instruction, wait for compare event. Set to 1 on ovp.cpu.Retire event</td>
</tr>
<tr>
<td>ret_rtl</td>
<td>bit</td>
<td>RTL has retired an instruction, wait for compare event. Set to 1 on riscv_tracer_i.retire event</td>
</tr>
<tr>
<td>ev_ovp</td>
<td>event</td>
<td>RM has retired an instruction</td>
</tr>
<tr>
<td>ev_rtl</td>
<td>event</td>
<td>RTL has retired an instruction</td>
</tr>
<tr>
<td>ev_compare</td>
<td>event</td>
<td>RTL and RM have both retired an instruction. Do compare.</td>
</tr>
</tbody>
</table>

Referring to Illustration 3:

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 3: Step and Compare Sequencing](image)

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 4 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.

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 Step-and-Compare 2.0**

The second generation of step-and-compare builds upon and fixes many issues with the previous iteration of step-and-compare while maintaining the same verification effectiveness.

The following improvements are realized in Step-and-Compare 2.0:

- Formalize interface definition for the processor under test
- Formalize interface definition for the ISS/Reference Model
- Implement the Step-and-Compare architecture in UVM using standard methodologies such as UVCs (Universal Verification Components) when applicable.
- Encapsulate data collection and movement throughout the testbench as transactions
- Provide standard logging mechanisms for Step-And-Compare to facilitate easy debug and triage
**RVFI**

The monitoring of processor activity was enabled by the *tracer* in the cv32e40p. The tracer was a SystemVerilog bound to the cv32e40p RTL that monitored all processor activities such as GPR, PC and CSR state. However the tracer interface was difficult to maintain and implemented an unspecified interface, requiring customized frequent changes in the Step-and-Compare implementation.

For future Step-and-Compare implementations, the RISC-V Formal Interface (RVFI) will be adapted to implement the processor monitor for the solution. In general the RVFI will follow its standard but may be adapted for extra requirements introduced by its usage in Step-and-Compare. All attempts will be made to ensure backwards-compatibility to the initial RVFI specification.

‘The existing RVFI specification can be found here: <https://github.com/SymbioticEDA/riscv-formal/blob/master/docs/rvfi.md>’. Any extensions to the RVFI will be described in this Verification Strategy document.

The RVFI consists of 2 major components. First, the processor itself must implement an RVFI SystemVerilog module that can be bound (using SystemVerilog bind) or another unobtrusive integration into the processor itself. The RVFI module presents a wired interface of the following signals. Note that the entire signal set may be replicated if the processor support retiring multiple instructions on the same clock cycle.

**Note:** The table below indicates RVFI signals which are not currently explicitly used in the RVFI/RVVI. However note that all RVFI signals are monitored and collected into sequence items for usage in logging and analysis port subscribers.
### Signal Description

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rvfi_valid</td>
<td>Signals that the rest of the bus is valid</td>
</tr>
<tr>
<td>rvfi_order</td>
<td>Monotonically increasing value which represents instruction ordering. Can be used with multiple retirement instruction interfaces to re-order instructions (if needed)</td>
</tr>
<tr>
<td>rvfi_insn</td>
<td>The instruction word that was retired. This should represent the ISA instruction directly. For example if this is a C instruction it should be the 16-bit instruction word and not the uncompressed ILEN-bit word.</td>
</tr>
<tr>
<td>rvfi_intr</td>
<td>Indicates that the current instruction is the first of a trap handler. This could map an interrupt, exception, or debug handler</td>
</tr>
<tr>
<td>rvfi_trap</td>
<td>Indicates that an instruction will cause an exception such as a misaligned read or write (if not supported), a jump to a misaligned address (if not supported) or an illegal instruction.</td>
</tr>
<tr>
<td>rvfi_halt</td>
<td>Indicates that an instruction is last before instruction execution halt. an interrupt, exception, or debug handler. Not currently used in RVFI/RVVI checkers</td>
</tr>
<tr>
<td>rvfi_dbg</td>
<td>Indicates that an instruction is the first instruction of a debug handler.</td>
</tr>
<tr>
<td>rvfi_mode</td>
<td>Indicates the processor mode for this instruction (0-U, 1-S, 2-Reserved, 3-M)</td>
</tr>
<tr>
<td>rvfi_ixl</td>
<td>Indicate the current XL setting of the privilege mode. Not currently used in RVFI/RVVI checkers</td>
</tr>
<tr>
<td>rvfi_pc_rd</td>
<td>The PC for the currently executed instruction</td>
</tr>
<tr>
<td>rvfi_pc_we</td>
<td>The expected PC for the next executed instruction, not taking into account interrupts, exceptions, or debug entry</td>
</tr>
<tr>
<td>rvfi_rs1_ad</td>
<td>The first source register address for the instruction. Set to zero if rs1 is not valid for the instruction opcode. Not currently used in RVFI/RVVI checkers</td>
</tr>
<tr>
<td>rvfi_rs1_rd</td>
<td>The first source register operand register read data Not currently used in RVFI/RVVI checkers</td>
</tr>
<tr>
<td>rvfi_rs2_ad</td>
<td>The second source register address for the instruction. Set to zero if rs1 is not valid for the instruction opcode. Not currently used in RVFI/RVVI checkers</td>
</tr>
<tr>
<td>rvfi_rs2_rd</td>
<td>The second source register operand register read data Not currently used in RVFI/RVVI checkers</td>
</tr>
<tr>
<td>rvfi_rd1_ad</td>
<td>The destination register address for the instruction. Set to zero if rd1 is not valid for the instruction opcode.</td>
</tr>
<tr>
<td>rvfi_rd1_wd</td>
<td>The destination register operand register write data</td>
</tr>
<tr>
<td>rvfi_mem_ad</td>
<td>The memory address accessed for the instruction Not currently used in RVFI/RVVI checkers</td>
</tr>
<tr>
<td>rvfi_mem_rd</td>
<td>The memory read data for this instruction. Valid bits indicated by rvfi_mem_rmask Not currently used in RVFI/RVVI checkers</td>
</tr>
<tr>
<td>rvfi_mem_wd</td>
<td>The memory write data for this instruction. Valid bits indicated by rvfi_mem_wmask Not currently used in RVFI/RVVI checkers</td>
</tr>
</tbody>
</table>

### CSR Interfaces

Each CSR implemented for a processor will have a CSR bus. Each CSR bus will consist of the following signals. Note that the CSR bus is valid based on rvfi_valid. In other words it is valid with the rest of the RVFI interface. The <csr> in the following table should be replaced with the CSR name (e.g. rvfi_csr_mstatus_mask)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rvfi_csr_rmask</td>
<td>Signals bits of the CSR that are valid at the beginning of instruction execution</td>
</tr>
<tr>
<td>rvfi_csr_wmask</td>
<td>Signals bits of the CSR updated during the execution of this instruction</td>
</tr>
<tr>
<td>rvfi_csr_rdata</td>
<td>Initial state of the CSR before the instruction is executed. Only bits enabled by rvfi_csr_&lt;csr&gt;<em>wmask are valid. For any bits not enabled, the value of the CSR for this instruction should use rvfi_csr</em>&lt;csr&gt;_rdata</td>
</tr>
</tbody>
</table>
RVFI Agent

The RVFI Agent is a passive agent that continuously monitors the connected RVFI interfaces, publishes full RVFI transactions on its analysis port, and logs the RVFI interface.

Each RVFI agent instance is parameterizable for ILEN and XLEN depending on the core being tested. Note that because of this, all object creation, uvm_config_db access and other general UVM calls must use proper parameterization to avoid difficult-to-debug compiler and elaborator errors.

The configuration object for the RVFI agent requires 2 settings. (This does not include standard log and monitor disables.)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nret</td>
<td>rand int unsigned</td>
<td>Constrain to match the number of parallel instruction retirements possible for the processor tested. The Agent will expect nret number of virtual interfaces to be configured for this instance</td>
</tr>
<tr>
<td>csrs</td>
<td>string[$]</td>
<td>Add all supported csrs to this queue by name. The agent will expect nret number of RVFI CSR virtual interfaces named for each configured CSR in the queue.</td>
</tr>
</tbody>
</table>

RVFI Monitor

The RVFI monitor is a simple UVM component that checks a configured RVFI interface each clock cycle for rvfi_valid.

In an agent instance there are nret RVFI monitors instantiated, each checking one RVFI virtual interface and one set of RVFI CSR virtual interfaces.

When the rvfi_valid is detected, the monitor simply samples all RVFI signals and packs them into an RVFI sequence item object. Each RVFI CSR virtual interface will also be sampled into a sequence item object which is attached to the instruction sequence item via a queue.

To facilitate asynchronous event simulation with step-and-compare, the RVFI monitor does include some logic to further determine whether the instruction represents an external debug entry request or an external interrupt. This information is encoded into the sequence item.

Once the sequence item is complete, it is broadcast to the rest of the testbench via its analysis. For debug via logs, a transaction log monitor is implemented which logs all RVFI transactions in single line to the following file: uvm_test_top.env.rvfi_agent.trn.log

The following is a snippet from the log transaction file:

<table>
<thead>
<tr>
<th>0.000 ns: RVFI Order PC Instr M rsl rsl_data rs2 rs2_data rd rd_data mem_op mem_addr mem_data</th>
</tr>
</thead>
<tbody>
<tr>
<td>138.000 ns: RVFI 1 00000080 0000d197 M x 1 00000000 x 0 00000000 x 3 0000d80 N/A</td>
</tr>
<tr>
<td>149.000 ns: RVFI 2 00000084 53018193 M x 3 0000d080 x16 00000000 x 3 0000d5b0 N/A</td>
</tr>
</tbody>
</table>

RVVI

The RISC-V Verification Interface provides a consistent interface to monitor and control a RISC-V Reference Model. The reference model often incorporates a instruction set simulator. The RVVI provides an implementation of a monitor to collect and report state of the reference model for checking. It provides an interface to the RVFI to be instructed of instruction retirements and other asynchronous events to properly control the reference model.

Note that the RVVI itself does not attempt to verify processor functionality. That testbench functionality should be handled by the processor UVM environment. The RVVI simply provides control and monitoring for a Reference Model in a CORE-V testbench.
**RVVI Control Interface**

The RVVI control interface is used to control the reference model. The reference model is typically a software implementation of the processor under test that focuses on ISA-level functionality. The step-and-compare methodology (and therefore the RVVI) assume that the reference supports a resolution of a single instruction. In other words, the reference model can execute a single instruction and stop and wait for further instruction.

The following represents the RVVI interface:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stepi task</td>
<td>Called to instruct the reference model to step a single instruction</td>
<td></td>
</tr>
<tr>
<td>notify event</td>
<td>Signals that a command was issued on the control interface <em>Not currently used in RVFI/RVVI checkers</em></td>
<td></td>
</tr>
<tr>
<td>cmd enum</td>
<td>Indicates the state of the reference model (STEP, IDLE, etc.). Can be useful (in waves) to indicate state of the reference model <em>Not currently used in RVFI/RVVI checkers</em></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The control interface (or a similar new interface) will be updated to incorporate more signaling required for the reference model. These signals include asynchronous event detection. For example a reference model normally has no indication that an external interrupt is causing the processor-under-test to vector to an interrupt handler without this signaling. The current solution reuses vendor-specific interrupt signaling but this will be formalized into RVVI in a soon-to-be-released update.

**RVVI State Interface**

The RVVI state interface is used to monitor the execution of the reference model. The following signals comprise the interface:
<table>
<thead>
<tr>
<th>Signal</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>notify</td>
<td>event</td>
<td>Indicates that the reference model has completed an operation. After trigging on this event, all other fields of the state interface are valid</td>
</tr>
<tr>
<td>valid</td>
<td>bit</td>
<td>Indicates that the notify event triggered on a valid instruction retirement</td>
</tr>
<tr>
<td>intr</td>
<td>bit</td>
<td>Indicates that the notify event triggered on an instruction retirement that entered an interrupt handler</td>
</tr>
<tr>
<td>trap</td>
<td>bit</td>
<td>Indicates that the notify event triggered on an instruction retirement that entered an exception handler</td>
</tr>
<tr>
<td>halt</td>
<td>bit</td>
<td>Indicates that the notify event triggered on an instruction retirement that entered a debug handler</td>
</tr>
<tr>
<td>insn</td>
<td>bit[ILEN-1:0]</td>
<td>For valid instructions, the instruction word just retired. This should always be the ISA instruction word as read on the instruction interface.</td>
</tr>
<tr>
<td>order</td>
<td>bit[XLEN-1:0]</td>
<td>An integer that increments for each retired instruction</td>
</tr>
<tr>
<td>isize</td>
<td>bit[2:0]</td>
<td>The number of bytes in the insn word. For example, this should be set to 2 for C-extension (compressed) instructions</td>
</tr>
<tr>
<td>ixl</td>
<td>bit[1:0]</td>
<td>Current XLEN for the retired instruction</td>
</tr>
<tr>
<td>mode</td>
<td>bit[1:0]</td>
<td>Privilege mode for the retired instruction</td>
</tr>
<tr>
<td>pc</td>
<td>bit[XLEN-1:0]</td>
<td>PC for the currently retired instruction</td>
</tr>
<tr>
<td>pc-next</td>
<td>bit[XLEN-1:0]</td>
<td>Expected PC for the next instruction, not considering interrupts, exceptions, nor debug</td>
</tr>
<tr>
<td>x</td>
<td>bit[XLEN-1:0][32]</td>
<td>The values for all General Purpose Registers after instruction retirement</td>
</tr>
<tr>
<td>csr</td>
<td>bit[XLEN-1:0][string]</td>
<td>Values for all CSRs after instruction retirement</td>
</tr>
<tr>
<td>decode</td>
<td>string</td>
<td>Assembly for the retired instruction for debug Not currently used in RVFI/RVVI checkers</td>
</tr>
</tbody>
</table>

**RVVI Agent**

The RVVI functionality is implemented in a single RVVI agent which controls and samples a single reference model instance in a testbench. The RVVI agent is typically an active agent, but should be configurable to be purely passive (for execution without the reference model as required, even though this mode should be avoided in most tests.)

**Note:** The RVVI Agent is intended to be targeted to multiple reference model implementations. Those implementations may well require additional considerations (e.g. sequences, configuration). It is expected that the testbench implementer will derive from the RVVI agent as necessary to implement reference-model-specific functionality. The description in this document will focus on common functionality provided in the RVVI agent in core-v-verif.

The RVVI agent does not generally require specific configuration beyond typical virtual interface configuration and enabling logging. There should be virtual interface configuration of the control and state interfaces from the SystemVerilog testbench. As mentioned above the RVVI agent is typically active, which implies that the control interface is active (via the RVVI driver) to actively control the reference model connected through the control virtual interface. This can be switched to passive to disable the driver.
RVVI State Monitor

The RVVI agent implements a monitor which continuously observes the state interface to report any activity from the reference model. The state monitor simply waits for the *notify* event from the state interface to trigger, then all signals from the interface are sampled, added to a rvvi state sequence item and broadcast to an analysis port.

There is also a state log monitor that logs all RVVI state activity to:

```
uvm_test_top.env.rvvi_agent.trn.log
```

The following is a snippet from the log transaction file:

<table>
<thead>
<tr>
<th>Time (ns)</th>
<th>RVVI</th>
<th>Order</th>
<th>PC</th>
<th>Instr</th>
<th>M</th>
<th>rd</th>
<th>rd_data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
<td>RVVI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>147.000</td>
<td>RVVI</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>159.000</td>
<td>RVVI</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>171.000</td>
<td>RVVI</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

When visually debugging the RVVI state interface it is important to recall the 0-time nature of the *notify* signal with respect to the rest of the bus. The signal values *after* *notify* should be read to understand the field values at the time of the *notify* event. In the example below the *valid*, *order*, *pc*, and *pcnext* values are updated coincident to the cursor at the *notify* event and will be recorded as 1, 1, 0x80 and 0x84 respectively.

![Fig. 5: Illustration 5: RVVI State Instruction Example](image)

RVVI Driver

The RVVI contains an active driver component which actually controls the reference model via the control virtual interface that was configured to the RVVI agent instance. The actual driver implementation will be discussed below.

The important features that the RVVI driver must accomplish are to:

- Notify the reference model when normal program interruptions occur. These are usually external interrupts or external debug requests, but the implementation could be extended to any external event.
- Step the reference model at the right time (i.e. after the RVFI completes an instruction)
The RVVI driver is implemented as a reactive slave. This differs from an active master where the user’s test would typically create and send sequence items to the driver. In this case, the RVVI driver is “driven” with sequence items that are created when the RVFI Monitor broadcasts a sequence item to its analysis port. The sequencer for RVVI implements an *analysis_export* that can receive RVFI instruction sequence items.

Thus when using the RVVI control driver the implementer must connect the RVFI monitor analysis port to the sequencer’s *analysis_export*. Typically this higher-level connection would be handled in the processor’s UVM environment as below:

```verbatim
function void uvme_cv32e40x_env_c::connect_rvfi_rvvi();
    foreach (rvfi_agent.instr_mon_ap[i])
        rvfi_agent.instr_mon_ap[i].connect(rvvi_agent.sequencer.rvfi_instr_export);
endfunction : connect_rvfi_rvvi
```

Within the sequencer, the analysis export simply pushes the RVFI sequence item to a queue in the sequencer:

```verbatim
function void uvma_rvvi_sqr_c::write_rvfi_instr(uvma_rvfi_instr_seq_item_c #(ILEN, 
    --> XLEN) rvfi_instr);
    rvfi_instr_q.push_back(rvfi_instr);
endfunction : write_rvfi_instr
```

The RVVI agent itself will start a sequence on the sequencer at time 0 of the simulation. This sequence will run in perpetuity until the end of the simulation. The sequence will continuously poll the `rvfi_instr_q` and issue rvvi control sequence items to the driver (to step the reference model for instance) as RVFI sequence items are received.

In the actual driver it may be necessary to *step* the reference model multiple times per transaction received if an external asynchronous event is signaled in the RVFI sequence item. See the Examples section for examples of this.

### Core Scoreboard

With the RVFI and RVVI implemented, one can now create a scoreboard component that can verify processor operation as individual instructions are executed. The following diagram shows the full testbench environment for a RVFI/RVVI scoreboard.

The Core Scoreboard is implemented and considered part of the core environment (e.g. `cv32e40x_env_c`). The scoreboard simply compares the sequence items received from the RVFI Monitor with the state received from the RVVI Monitor. In most embedded cores the instructions from each agent should arrive in order, however the scoreboard can work with multi-retirement cores (i.e. \( nret > 1 \)) by using the unique *order* field in both RVFI and RVVI transactions to determine proper ordering.

The scoreboard is configured based on the following fields in the environment’s configuration object (e.g. `cv32e40x_cfg_c`).

<table>
<thead>
<tr>
<th>Configuration class variable</th>
<th>Default value</th>
<th>Simulation make control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scoringabling_enabled</td>
<td>1</td>
<td>USE_ISS=&lt;0</td>
<td>1&gt;</td>
</tr>
<tr>
<td>scoringabling_disable_csr_chck_all</td>
<td></td>
<td>Plusarg: +disable_csr_chck_all</td>
<td>Disables checking of all CSR values in scoreboard</td>
</tr>
<tr>
<td>scoringabling_disable_csr_chck_mip+me</td>
<td>Indexable hash. Indexed by CSR name</td>
<td>Plusarg: +disable_csr_chck=mip+me</td>
<td>Disables checking of specific CSR values in scoreboard</td>
</tr>
</tbody>
</table>

### 5.2. The CV32E40* UVM Verification Environment
Fig. 6: Illustration 6: Step and Compare 2.0 Scoreboard with RVFI and RVVI Agents
An individual test may configure the scoreboard within its test.yaml file. Each of the above could be configured via the following:

```yaml
tests/programs/custom/hello-world/test.yaml

name: hello-world
uvm_test: uvmt_${CV_CORE_LC}_firmware_test_c
description: >
  Simple hello-world sanity test
# This would disable the ISS (i.e. set scoreboard_enabled = 0)
is: 0
# This would disable all CSR checking
disable_csr.chk_all: 1
# This would disable CSR checks for mip, mepc, mcause
disable_csr.chk: >
  mip
  mepc
  mcause
```

For each instruction the following is checked:

- PC of the retired instruction
- GPR state including any updated GPR from the retired instruction
- order field should be monotonically increasing (i.e. no skipped nor repeated order fields on either interface)
- order field should match from RVFI to RVVI
- CSR state as result of the instruction execution (i.e. CSR state at instruction retirement) should match

### Examples

The following sections will give some examples to illustrate how instructions are checked in the scoreboard. Note that for all of these examples the RVVI wraps an Imperas RISCVOVPSim instance as the Reference Model. Other reference model implementations may slightly differ (when and if added in the future).

### Instruction Check

The following waveform shows an instruction being successfully compared in the hello-world test.

The first event which start the instruction check cycle is the RVFI signalling that an instruction is complete by asserting rvfi_valid (see the cursor location in waveform). The RVFI Monitor will record all signals from the RVFI instruction, register, memory, and CSR interfaces into an RVFI sequence item as stated above.

When the RVVI Sequencer receives this transaction, the perpetual sequence that steps the Reference Model will send an appropriate sequence item to the RVVI Driver. Note that this manifests on the waveforms by gating clocks to the DUT (i.e. RVFI). This is a consequence of how the Imperas RISCVOVPSim works and may not apply to other reference models. The RVVI Control Interface will step the Reference Model for one instruction.

On the RVVI state interface, the notify event asynchronously signals that the Reference Model has retired an instruction. The RVVI State Monitor will sample all relevant signals from the RVVI State Interface into a sequence item for broadcast on its analysis port. Since the Reference Model and the state interface signal asynchronously as soon as the Reference Model completes, one should read waveform values to the right of the event. In the simulation these values are updated before the notify event such that the RVVI state monitor will pick up those new values. This is slightly non-intuitive versus reading signals at a clock edge (where you would typically read to the left of the positive edge for synchronous logic).
Fig. 7: Illustration 7: Step and Compare 2.0 Scoreboard Instruction Check Example
The Scoreboard upon receipt of the RVVI sequence item will then initiate a check (at the same simulation time as the notify event) versus the first RVFI sequence item in its queue. All fields will be checked, any errors reported and the sequence items then discarded.

**CSR Check**

**Note:** CSR checks occur simultaneously with RVFI and RVVI sequence items in an instruction check as described above. However they are presented in this separate section for clarification such that the specific CSR waveforms are highlighted.

The following waveform shows a CSR (mstatus) being check for an instruction in which it is being updated by the controller (from the interrupt_test test).

![Fig. 8: Illustration 8: Step and Compare 2.0 Scoreboard CSR Check Example](image)

This example is a mret instruction retirement, which will update the mstatus register interrupt enables. The RVFI initially signals rvfi_valid and as part of the instruction monitoring, the CSR masks and data for mstatus are recorded. Note that since rvfi_csr_mstatus_wmask is set to 32’hffff_fffff, this designates that all bits of mstatus are updated by the instruction and thus rvfi_csr_mstatus_wdata will contain the value of the CSR at instruction retirement, which is

5.2. The CV32E40* UVM Verification Environment
32'h0000_1888). Note that the previous instruction has a write mask of 0, and thus the CSR value of `mstatus` for the previous instruction on RVFI is 32'h0000_1080.

The RVVI state interface only shows the Reference Model’s expected value of `mstatus` at instruction retirement. This value is contained in the associative array `csr` under the csr name (`csr["mstatus"]`). This value is also 32'h0000_1888 so the CSR will check properly at the notify event.

### Interrupt Handler Check

On its own, the Reference Model cannot determine when interrupts to normal program flow occur, such as external interrupts or external debug requests. The RVFI monitors and reports those events, and thus the RVVI can be used to inform the Reference Model to interrupt normal program flow, maintaining processor state lock-step with the DUT.

The following example waveform shows entry into an interrupt handler due to external interrupt request assertion.

**Fig. 9: Illustration 9: Step and Compare 2.0 Scoreboard Interrupt Check Example**

The example shows the very first instruction of an instruction handler. In this case the RVFI signals a valid instruction retirement with the `rvfi_intr` signal set. From the RVFI specification `rvfi_intr` signals that this instruction is the first instruction of any trap, which could be an exception (which the Reference Model can generally model without assistance) or an interrupt handler due to external signaling which requires assistance. To determine this, the RVVI control driver for the OVPSim will use the `mcause` value reported by the RVFI instruction. In this case the RVFI is reporting a value of 0x8000_0003. Since bit[31] is set, the trap is therefore an external interrupt.

Now that the RVVI OVPSim driver knows that an external interrupt of 0x3 is signaled from RVFI, it will assert `deferint` (active low) and `irq_i` on the RVVI IO interface. Then it will step for an “instruction”. The Reference model then is signaled that an interrupt is to be entered. Note that due to the way OVPSim works, an extra step is required to
actually step into the interrupt handler to retire the first instruction of the handler. Thus the RVFI and RVVI are re-synchronized. The RVVI monitor uses the *rvvi_valid signal in conjunction with the rvvi.csr[*mcause*] value to determine that the notify event should be discarded for the deferint step.

**Note:** If an exception (e.g. illegal instruction) handler were entered and mcause[31] is deasserted, then that instruction should be checked against an RVFI instruction. (This may be better clarified via the *valid* signal on the RVVI state interface in the future).

### 5.2.7 Beyond Phase 3 Environment

At the time of this writing (2020-04-21) there is a proposal to develop a CV32E40P Subsystem, comprised 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 [https://github.com/openhwgroup/core-v-docs/blob/master/cores/cv32e40p/OBI-v1.0.pdf]’.

Illustration 10 shows a simple (?) change to the uvmt_cv32_tb that allows the testbench (and thereby the UVM environment) 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.\(^\text{11}\)

In Illustration 10 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 “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 output 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.

### 5.3 File Structure and Organization

**ToDo**

\(^{11}\) 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. 10: Illustration 10: Moving Memory Model to the Testbench
Fig. 11: Illustration 11: Subsystem Wrapper (compiled for Core-level verification)
5.3.1 Naming Convention

5.3.2 Directory and File Structure

5.3.3 Compiling the Environment
 CHAPTER 6

CVA6 Simulation Testbench and Environment

TODO.
CHAPTER 7

Test Programs

The purpose of this chapter is to define the “programming environment” of a test program in the CORE-V-VERIF verification environment. The current version of this document is specific to the CV32E40 family of CORE-V cores. Further versions will be sufficiently generic to encompass all CORE-V cores.

A “test program” is a set of RISC-V machine instructions that are loaded into the testbench memory and executed by the core RTL model. Test-program source code is typically written in C or RISC-V assembler and is translated into machine code by the toolchain. Test programs can be either human or machine generated. In either case it needs to be compatible with the hardware environment supported by the core and its testbench. Programming for CORE-V-VERIF is the ultimate bare-metal coding experience as the environment supports the bare minimum of functionality required to verify the core.

7.1 Hardware Environment

For the purposes of this discussion, the “hardware environment” is a set of hardware resources that are visible to a program. In CORE-V-VERIF this is essentially the Core testbench comprised of the RTL model of the core itself, plus a memory model.

The core testbench for the CV32E40 cores has an instruction and data memory plus a set of memory mapped virtual peripherals. The address range for I&D memory is 0x0..0x10_0000 (1Mbyte). The virtual peripherals start at address 0x1000_0000.

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 (and obviously this address must exist).
- The address dm_halt_addr_i must exist in the memory map, it should not be overwritten by the test program and the code there must produce a predictable result.
• The program must have knowledge about the addressing and operation of the virtual peripherals (using the peripherals is optional).

### 7.2 Virtual Peripherals

The memory module in the Core testbench implements a set of virtual peripherals by responding to read or write cycles at specific addresses on the data bus. These virtual peripherals provide the features listed in Table 1.

Three virtual peripheral functions are in place to support RISC-V Compliance test-programs. These are a virtual printer that allows a test program to write to stdout, a set of status flags used to communicate end-of-sim and pass/fail status and a signature writer.

The debug control virtual peripheral can be 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.

The use of the interrupt timer control and instruction memory stall controller are not well understood and it is possible that none of the testcases 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.
<table>
<thead>
<tr>
<th>Virtual Peripheral</th>
<th>VP Address (data_addr_i)</th>
<th>Action on Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Range Check</td>
<td>&gt;= 2**20</td>
<td>Terminate simulation (unless address is one of the virtual peripherals, below).</td>
</tr>
<tr>
<td>Virtual Printer</td>
<td>32'h1000_0000</td>
<td>$write(&quot;%c&quot;, wdata[7:0]);</td>
</tr>
<tr>
<td>Interrupt Timer Control</td>
<td>32'h1500_0000</td>
<td>timer_irq_mask &lt;= wdata; timer_count &lt;= wdata; This starts a timer that counts down each clk cycle. When timer hits 0, an interrupt (irq_o) is asserted.</td>
</tr>
<tr>
<td>Debug Control</td>
<td>32'h1500_0000</td>
<td>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 range</td>
</tr>
<tr>
<td>Random Number Generator</td>
<td>32'h1500_0000</td>
<td>Reads return a random 32-bit value with generated by the simulator’s random number generator. Writes have no effect.</td>
</tr>
<tr>
<td>Cycle Counter</td>
<td>32'h1500_0000</td>
<td>Reads return the value of a free running clock cycle counter. Writes resets the cycle counter to 0.</td>
</tr>
<tr>
<td>Instruction Memory Interface Stall Control</td>
<td>32'h1600_XXXX</td>
<td>Program a table that introduces “random” stalls on IMEM I/F.</td>
</tr>
<tr>
<td>Virtual Peripheral Status Flags</td>
<td>32'h2000_0000</td>
<td>Assert test_passed if wdata==’d123456789 Assert test_failed if wdata==’d1 Note: asserted for one clk cycle only.</td>
</tr>
<tr>
<td></td>
<td>32'h2000_0000</td>
<td>Assert exit_valid; exit_value &lt;= wdata; Note: asserted for one clk cycle only.</td>
</tr>
<tr>
<td>Signature Writer</td>
<td>32'h2000_0000</td>
<td>signature_start_address &lt;= wdata; signature_end_address &lt;= wdata; 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=&lt;sig_file&gt; plusarg. Note: this will also assert exit_valid with exit_value &lt;= 0.</td>
</tr>
</tbody>
</table>

Table 1: List of Virtual Peripherals

### 7.3 Board Support Package

The role of the Board Support Package (BSP) is to align the test-programs with the hardware. There are a number of files that define the BSP. These are discussed below.

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 BSP (crt0.S and link.ld). The toolchain uses the BSP to align resources used by the test program with resources supported by the hardware. Note that 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)

### 7.3.1 Elements of the BSP

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.

#### 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](http://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.

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:

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

MEMORY
{

```
(continues on next page)
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 ./cv32e40p/bsp/crt0.S. crt0.S performs the bare minimum required to run a C program. Note that support for command-line arguments is deliberately not supported.
Additional Information

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

7.4 Debug Mode Support

The BSP allocates a region of memory for debug instructions. It's important to note that this debug memory is not a separate memory in the testbench; it is merely a specific region of the memory. The debug memory is loaded with a hex image defined with the plusarg +debugger=<filename.hex>.

It is a requirement that the hardware configuration of the debug memory and the start of the debug region as defined in the BSP be aligned. Using the CV32E40P as an example, asserting the debug_req_i will cause it to enter debug mode and start executing code located at dm_halt_addr_i. The debug origin in the BSP must match the dm_haltaddr_i configuration input to the core RTL.

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 program that will use the debug control virtual peripheral and provide a specific debugger code image.

7.5 Interrupt Support

TBD
CHAPTER 8

UVM Testcases in the CORE-V-VERIF Environments

The overall structure of the CORE-V-VERIF UVM environments should be familiar to anyone with UVM experience. This section discusses the CORE-V-VERIF 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.

Before reading this chapter, it is recommended that you read the Test Programs chapter.

8.1 Test-Programs in the CORE-V-VERIF UVM Environment

The UVM environment can support test-programs regardless of how they are created, so long as they are compatible with the BSP. However, the UVM 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?

Many of the test programs inherited from the RISCY 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.

CORE-V-VERIF incorporates a random instruction stream generator to generate many test programs. It is expected that most of 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” 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. 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
as described in *Virtual Peripherals*.

### 8.2 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 exec-
   uted 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.

#### 8.2.1 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 incom-
plete 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.

### 8.3 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 develop-
ers. 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
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.

The UVM Testcases chapter of this document discusses how the configure_phase() and run_phase() manage the interaction between the UVM environment and the test program. This interaction 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 \texttt{dp\_ram} using the \texttt{$readmemh()} DPI call. The next sub-section explains how to select which test program to run from the command-line. During the configuration phase the test signals the TB to load the memory. The TB assumes the test file already exists and will terminate the simulation if it does not.

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

At this point the run flow will simply wait for the test program to flag that it is done via the status flags virtual peripheral. 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.

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 testbench memory. 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

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

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

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

At this point the run flow will simply wait for the test program to flag that it is done via the status flags virtual peripheral. 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.

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

## 8.4 CORE-V Testcase Writer’s Guide

TODO

### 8.4.1 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 `$CORE_V_VERIF/cv32e40p/tests/programs`.

### 8.4.2 Writing a UVM Test to run a Test Program

The CV32E40P base test, `uvmt_cv32e40p_base_test_c`, has been written to support all five of the test program types discussed above.

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 `$CORE_V_VERIF/cv32e40p/tests/uvmt_cv32e40p/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.

### 8.4.3 Running the testcase

Testcases are intended to be launched from `$CORE_V_VERIF/cv32e40p/sim/uvmt`. 
COREV-DV is the name given to a library of extensions to the Google riscv-dv instruction stream generator. Riscv-dv is implemented as a collection SystemVerilog classes extending from uvm_object, so extending riscv-dv classes to modify its behavior is straightforward. For those cases where a specific instruction stream is required, it is implemented as an extension of an existing riscv-dv class. Thus, riscv-dv is not modified allowing core-v-verif to easily take advantage of updates to the Google project.

Note that for the most part, core-v-verif uses riscv-dv “as is” and each core that is verified in core-v-verif is free to use whatever version of riscv-dv that suits the core’s needs:

- riscv-dv “as is”.
- corev-dv (extensions at $CORE_V_VERIF/lib/corev-dv
- core-specific extensions at $CORE_V_VERIF/$COREV_CORE)/env/corev-dv

A non-core-specific version of corev-dv resides at $CORE_V_VERIF/lib/corev-dv. Here you will find a set of extensions to riscv-dv that are intended to be common across all (or at least most) CORE-V cores.

9.1 Using COREV-DV

When a Make target requires it, a specific hash of riscv-dv is cloned to $CORE_V_VERIF/$COREV_CORE/vendor_lib/riscv-dv. The compile Makefiles will compile in the required extensions to generate the core-specific version of corev-dv needed for a test. The README at $CORE_V_VERIF/mk/uvmt provides instructions for building and running a corev-dv generated test-program.

9.2 Extending COREV-DV

It is to be expected that each unique core will require multiple extensions to riscv-dv. These should be placed in $CORE_V_VERIF/$COREV_CORE/env/corev-dv.

For example $CORE_V_VERIF/lib/corev-dv/corev_asm_program_gen.sv implements an override of riscv_gen_program_header::gen_program_header() to enforce the use of common symbols required by the
board support package. Since each core’s verification environment is likely to use a core-specific BSP, it is expected that each core will need to implement core-specific extensions to corev_asm_program_gen.sv.

Below is simplified view of the core-v-verif directory tree showing the locations of riscv-dv and the base corev-dv extensions plus an example of core-specific extensions for the CV32E40P.
OpenHW’s starting point was the RI5CY (CV32E40P) and Ariane (CVA6) cores from PULP-Platform. The structure of the testbenches for these projects had a direct influence on the architecture of CORE-V-VERIF, so it’s help to review these. It is also informative to consider the Ibex project, another open-source RISC-V project derived from the ‘zero-riscy’ PULP-Platform core. The reader should keep in mind that this chapter was written in the early days of core-v-verif and will now be out of date in many places. Nevertheless, it is a useful historical document that captures the initial conditions of the project.

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 **CVA6 Simulation Testbench and Environment** chapters, which detail how the RI5CY and Ariane simulation environments will be migrated to CV32E and CVA6 simulation environments.

### 10.1 Executive Summary

In the case of the CV32E40P, 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 CV32E4 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 environments, and will have the same ability to run the RISC-V Compliance test-suite.
10.2 RI5CY

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

10.2.1 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 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.

10.2.2 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\(^4\). 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:

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

This macro expands to:

```plaintext
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.\(^7\)

**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.

---

\(^4\) 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.

\(^7\) These tests have been deprecated and are no longer part of the core-v-verif repository.
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`, a program that exercises the `add immediate` instruction. The testcase exercises the `addi` instruction with a set of 24 calls to `TEST_*` macros as shown above.

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

**RISC-V Compliance Tests**

There are 56 assembly language tests in the `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.

**10.2.3 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 RISCY 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 RISCY 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 RISCY 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 instructions using a UVM sequence-item generator. An example is the Google RISC-V instruction generator.

d) Prediction of execution results using a reference model built into the environment, not the individual testcases. Imperas has an open-source ISS that could be used for this component.

e) Scoreboarding to compare results from both the reference model and the RTL.

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

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

10.3 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.

10.3.1 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
10.3.2 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.

10.4 IBEX

Note: the following was written in April of 2020 and is by now somewhat out of date. It is included here for its historical value for understanding the thinking behind the development of CORE-V-VERIF.

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.

10.4.1 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 CV32E9. Immediately following, an effort will be undertaken to integrate

---

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