Jump start your RISCV project with OpenHW

Mike Thompson
OpenHW Group
mike@openhwgroup.org

Jingliang (Leo) Wang
Futurewei Technologies, Inc.
jingliangwang@futurewei.com

Steve Richmond
Silicon Labs
Steve.Richmond@silabs.com

Lee Moore
Imperas Software Ltd.
moore@imperas.com

David McConnell
EM Microelectronic-US
David.McConnell@emmicro-us.com

Greg Tumbush
EM Microelectronic-US
Greg.Tumbush@emmicro-us.com

***Abstract*- The OpenHW group is a member driven global organization with the shared goal of developing RISC-V compliant open source IP which meet commercial standards for delivery and quality. This paper will address the verification methodology adopted by the OpenHW Verification Task Group to assure commercial standards for quality.**

I. Introduction

The OpenHW group is a not-for-profit, global organization of members with the shared goal of developing RISC-V compliant open source IP which meet commercial standards for delivery and quality. The initial processor designs originated within the PULP platform at the University of Bologna and ETH Zurich under the guidance of Professor Luca Benini. These early RISC-V implementations became well known within the RISC-V community and soon attracted interest from a wide range of SoC early adopters. The OpenHW Group has rebranded specific PULP cores under the name "CORE-V". The verification environment for these cores is collectively known as "core-v-verif". CORE-V specifications, design and verification code are open-source artifacts, licensed under Solderpad 2.0, a permissive license extension of Apache 2.0 [1][2].

The delivery of trusted and reliable semiconductor IP depends on many design aspects including software ecosystem support, core hardware features, complete documentation and repeatable verification. This paper will address the verification aspects and the flows adopted by the OpenHW Verification Task Group based on leading industry best practices with commercial tools.

This paper outlines the details of the OpenHW verification methodology using commercial tools with open scripts and experiences of collaboration across a distributed team. Code snippets of the testbench, step-and-compare, coverage, scripting, etc will be provided. Instructions on downloading and running the suite of tests to obtain 100% functional coverage on a RISCV core will be shown. A reader will obtain the ability to use the OpenHW verification environment as a baseline to design and verify a custom RISCV core with their own value-add features.

II. Processor IP Verification planning

In general terms the verification of complex SoC's is a well understood problem. Modern SoC hardware design practices help teams undertake the most complex of designs and achieve high success rates with first-time-right prototypes. One key assumption for all design/verification flows is the initial quality of 3rd-party IP such as processor cores. Leading commercial IP suppliers have established a remarkably high standard of quality that the industry has come to rely on. In order to achieve wide-spread adoption in industry, open-source IP must work to achieve, and demonstrate, the same high standards. The design freedoms of open-source IP are obvious, but the unknown risk is the quality of the incoming IP and consequences of adoption for the SoC design/verification process.

To enable SoC design teams to adopt CORE-V core(s) and establish confidence in both the initial quality and suitability as a framework for further enhancement, a complete end-to-end verification environment has been published as open-source code and documentation. In addition, to ensure smooth adoption within current SoC designs flows the environment is based on industry best practices and leading commercial tools. To support adopters looking to further optimize the designs, the complete verification environment, including testbenches, individual verification components, coverage models and testcases are published as open source assets under the same license as the cores themselves [1].

IP verification has 5 key elements: 1) a complete Design Verification (DV) plan that details what is to be verified, 2) a verification environment (testbench) to achieve the goals of the DV plan, 3) a reference model to compare against, 4) a method of generating either automatic or manual stimulus (test cases) to exercise the device under test, and 5) coverage data to demonstrate that the goals of the DV plan have been achieved. Each of these are discussed in the following sections.

Elements 2..5 will be discussed in the next section of this paper. The remainder of this section focuses on the DV plan. We provide significant detail on this topic as it is an often overlooked activity. One of the reasons for this is a lack of usable examples. Even though there is at least one text book on the topic [12], DV planning is not well covered in academia and commercial organizations tend to treat their own DV plans as proprietary. By contrast, the DV plans for the CV32E40P are available as open source artifacts on the core-v-docs GitHub repository. As the specifications and source code are also open source, these DV plans can be cross-referenced against the specification and implementation, making them a valuable teaching aid.

A. The Design Verification Plan

A key activity of any verification effort is to capture a Design Verification Plan. The DV plan is a reflection on the quality of the final design as an error or oversight in the DV plan may result in untested or improperly tested features in the final design. A complete, accurate DV plan represents a significant investment of engineering time and many teams lack the engineering and time resources to capture their own DV plans. OpenHW Group DV plans are open-source artifacts and are subject to on-going reviews within the OpenHW Verification Task Group and by the wider open source community. Access to these DV plans can result in significant cost savings for teams involved in projects developing their own RISC-V cores.

1) Purpose of the DV Plan:

The primary purpose of a DV plan is to establish the goals of the verification effort and establish the quantitative metrics used to measure progress. It does this by identifying what features need to be verified; the stimulus required to exercise the features; the success or failure criteria of the feature and the coverage metrics to determine that the feature has, in fact, been verified. DV plans also allow engineers to reason about the required capabilities of the verification environment at early stages of the project. For example, the CV32E40P DV plan specifies that all RV32I instructions be generated and their results checked. This implies that the verification environment needs the capability to generate all RV32I instructions and to predict how CSRs, GPRs and memory will be changed as a result of executing these instructions.

In the initial stages at least, the DV Plan should focus on the *what*, and not the *how* of verification. At this stage, the team is primarily interested in creating a laundry list of features that need to be verified, and is not (yet) concerned with *how* to verify them.

2) A Trivial Example: the RV32I ADDI Instruction:

Let's assume your task is to verify the RV32I ADDI instruction. Checking the arithmetic result (rd = rs1 + imm), is necessary, but insufficient. We also need to verify that:

- Overflow is detected and flagged correctly;
- Underflow is detected and flagged correctly;
- No instruction execution side-effects (e.g. unexpected GPR changes, unexpected condition codes);
- Program counter updates appropriately.

It's also important that the instruction is fully exercised, so we may need to cover the following cases:

- Use x0..x31 as rs1
- Use x0..x31 as rd (Note: the result of this operation will always be 0x0 when rd is x0)
- Set/Clear all bits of immediate
- Set/Clear all bits of rs1
- Set/Clear all bits of rd

Note the simplifying assumptions made here. With one 32-bit and one 12-bit operand there are 2^44 unique sums that can be calculated. Including the cross-products of source and destination register yields 2^54 unique instruction calls. The RV32I ISA specifies 40 instructions so this gives us O(10^17) instruction executions simply to fully

verify the most basic instructions in an RV32I core. Obviously, this is impractical and one of the things that makes Verification an art is determining the minimal amount of coverage to have confidence that a feature is sufficiently tested. Whether the above is seen as over-kill or under-kill depends on an organizations understanding of the micro-architecture or risk aversion.

3) Format of a DV Plan:

CORE-V projects use spreadsheets to capture DV plans. Spreadsheets are a non-ideal format for tracking and revision control, but they have proven to be the best format for writing and reviewing this type of data. OpenHW's DV Plan template is simple enough that either Microsoft Office Excel or LibreOffice Calc can be used to write and/or review. What follows is a brief overview of our template, which is available as an open source artifact on the core-v-docs GitHub repository. The template enforces a set of common headers which attempts to provide an easy-to-use format to capture and review verification intent.

Requirement Location

This is a pointer to the source Requirements document of the Features in question. It can be a standards document, such as the RISC-V ISA, or a micro-architecture specification. Every item in a DV Plan must be attributed to one or more of these sources.

Feature

The high-level feature you are trying to verify. For example, RV32I Register-Immediate Instructions. In some cases, it may be natural to use the section header name of the reference document.

Sub-Feature

This is an optional, but often used column. Using our previous examples, ADDI is a sub-feature of RV32I Register-Immediate Instructions. Additional columns can be added in those situations where the specification itself has deep hierarchy.

Feature Description

A summary of what the feature does. It should be a summary, not a verbatim copy and paste from the Requirements Document.

Verification Goals

A summary of what stimulus and/or configuration needs to be generated/checked/covered to ensure sufficient testing of the Feature. Recall the example of the ***addi*** instruction. The verification goals of that feature are:

- Overflow is detected and flagged correctly
- Underflow is detected and flagged correctly
- No instruction execution side-effects (e.g. unexpected GPR changes, unexpected condition codes)
- Program counter updates appropriately

Pass/Fail Criteria

Here we attempt to answer the question, "how will the testbench know the test passed?". There are several methods that are typically used in OpenHW Groups projects, and it is common to use more than one for a given item in a Verification Plan.

- **Self Checking:** A self-checking test encodes the correct result directly into the testcase and compares what the DUT does against this "known good" outcome. This strategy is used extensively by the RISC-V Foundation Compliance tests.

- **Signature Check**: This is a more sophisticated form of a self checking test. The results of the test are used to calculate a checksum (signature) and this is compared against a "known good" signature. This strategy is also used by the RISC-V Foundation Compliance tests.

- **Check against Reference Model**: Here, the testcase does not "know" the correct outcome of the test, it merely provides stimulus. The pass/fail criteria is determined by a reference model, and the verification environment must compare the actual results from the DUT and the expected results from the reference model. When practical, this is the preferred approach because it simplifies testcase maintenance.

- **Assertion Check**: Failure is detected by an assertion, typically coded in SVA.

- **Other**: If one of the above Pass/Fail Criteria does not fit your needs, specify it here.

- **N/A**: Select this for those (rare) features in the specification that do not have side effects that are observable in a functional simulation of an RTL model.

Test Type

The source of stimulus is an important consideration as it helps to specify the capabilities of specific components in the verification environment such as generators, reference models and checkers. In the early stages of a project it can be difficult to have specific details about testcases, so the OpenHW DV plan template defines a set of broad categories. These have proven to be sufficiently general for the early stages of the project:

- **RISC-V Compliance**: a self-checking ISA compliance testcase from the RISC-V Foundation.

- **OpenHW Compliance**: OpenHW Compliance is compliance testing of the custom instructions supported by CORE-V cores.

- **Directed Self-Checking**: a directed (non-random) self-checking testcase from the OpenHW Group that is not specifically targeting ISA compliance.

- **Directed Non-Self-Checking**: a directed (non-random) non-self-checking testcase from the OpenHW Group that is not specifically targeting ISA compliance. Note that these tests assume that the pass/fail criteria will be "Check against Reference Model".

- **Constrained-Random**: a constrained-random testcase. Typically, the stimulus for these will come from the Google random instruction stream generator or the OpenHW Group's FORCE-RISCV generator. While a constrained-random test may include a signature, it is considered a best practice to use a reference model to check the pass/fail of such tests due to the difficulty of creating a comprehensive signature for random stimulus.

- **ENV capability, not specific test**: Often, a specific feature is not covered by a specific test or check. For example, an assertion checking for bus protocol errors could reasonably expect to cause a failure with any type of test.

- **Other**: If one of the above Test Types does not fit your needs, specify it here.

Coverage Method

Coverage should be used to determine when a feature has been verified (covered). There are several choices here:

- **Functional Coverage**: the testbench supports SystemVerilog covergroups that measure stimulus/configuration/response conditions to show that the Feature was tested. For OpenHW projects this is the preferred method of coverage.

- **Assertion Coverage**: an alternate form of functional coverage, implemented as SVA cover properties.

- **Code Coverage**: the Feature is deemed to be tested when the specific block of code or conditions in the RTL have been exercised.

- **Testcase**: if the testcase was run, the Feature was tested.

Link to Coverage

This field, used to link the Feature to coverage data generated in regression, is the most problematic for open-source projects since automating links from a DV plan to functional coverage code in the verification environment is

supported by commercial tools using proprietary technology. Thus, the CV32E40P DV plan captures links to the core-v-verif environment as SystemVerilog hierarchical paths. These links are manually generated and there is no tooling available to automate linking coverage results directly back to the DV plan.

III. Verification Environment

The verification environment for CORE-V processor cores is known as "core-v-verif", named for the GitHub repository that hosts it [1]. A major goal of core-v-verif is to provide a single, unified verification environment that supports all OpenHW CORE-V cores, and indeed, could in theory be used to support any RISC-V core. The remainder of this discussion will focus on those aspects of core-v-verif that are specific to the CV32E40P. A future paper will discuss how core-v-verif supports different cores with a single environment.

What follows is an overview of the core-v-verif structure as it pertains to the CV32E40P core. A more detailed overview, including some historical background can be found in the OpenHW Group CORE-V Verification Strategy, available on ReadTheDocs [9].

In this chapter, the specifics of verifying a CORE-V processor, namely the CV32E40P, are provided. The intent is to provide sufficient detail for the reader to use the OpenHW verification environment as a baseline to design and verify a custom RISCV core with their own value-add features.

A. The CV32E40P

CV32E40P is a RV32IMC RISC-V compliant 4-stage in-order machine-mode core. Execution-based debug, RISC-V compliant machine-mode interrupts (MEI, MTI and MSI) and an additional 16-bits of custom interrupts are supported. Instructions are fetched from a read-only memory interface, which supports an AMBA AXI-like protocol called the Open Bus Interface or OBI [14]. Load and store instructions access memory via a second read-write memory interface which follows the same OBI protocol. The top-level interfaces of the CV32E40P are:

Clock and Reset:	The core has one clock domain and one reset domain.
Configuration:	External inputs that define the start-up boot address, etc.
Instruction Memory Interface:	OBI used to fetch instructions
Data Memory Interface:	OBI used to load and store data
Interrupts:	CLINT compliant interface with 16-bits of custom interrupt inputs
Debug:	Used by an external Debug Module to request the core to enter debug mode.
Special Control and Status:	I/O to signal core state.
Auxiliary Processor Unit:	Custom interface to an off-core auxiliary processor, such as a floating point unit.

Note that for the CV32E40P, the Auxiliary Processor Unit interface is implemented, but not fully verified. The APU will be fully supported in a future release of the core.

A complete User Manual is available as an open-source artifact and is viewed on-line using ReadTheDocs [7] (search for "CV32E40P User Manual").

B. The CORE Testbench

The architecture of core-v-verif is strongly influenced by the structure of previous generations of testbenches for RISC-V cores. It is typical for RISC-V testbenches to be comprised of a test-program generator that produces a program to be executed by the core RTL model, a software toolchain that compiles/assembles the test-program into machine code and a simple Verilog or SystemVerilog testbench to support simulation. Often, these are separate and independent components that are run sequentially to realize a RISC-V simulation. Although this structure is somewhat awkward to use and difficult to scale, the core-v-verif environment preserves it as much as is practical for two reasons:

1. There is a comprehensive set of pre-existing test-programs that can be re-used. Chief among this is the RISC-V Compliance test-suite. It is highly desirable to be able to re-use such open-source artifacts.

2. OpenHW wishes to support members of the open source community who may not have access to commercial simulators.

The SystemVerilog components of the core-v-verif environment that implement the above structures are collectively known as the "core" testbench. The core testbench is light-weight, fast and can run using Verilator, an open-source SystemVerilog simulator. The core testbench runs a set of hand-written assembler and C test-programs (including a CV32E40P-specific version of the classical "hello-world") and can also run the RISC-V compliance test-suite. The structure of the "core" testbench is shown below in **Figure 1**.

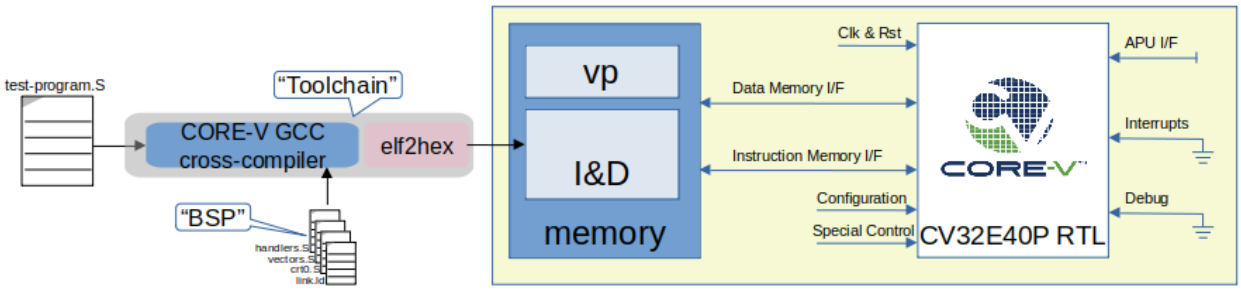

Figure 1: Using the Core Testbench with a Test-program, BSP and Toolchain

The core testbench is simple, yet provides all the infrastructure required to run a program on the simulated core. It instantiates the core RTL model, clock and reset generator, drives configuration inputs as necessary and also instantiates a memory model that connects to both the instruction and data interfaces. All of these are implemented as SystemVerilog modules, using a sub-set of the language that can be run using Verilator. The memory model supports the OBI interfaces for instruction and data memory and implements a minimal set of "virtual peripherals" that a test-program can use to dump signatures (as required by the Compliance test-suite), print to stdout and signal simulation termination and/or pass/fail status to the testbench. A complete description of the virtual peripheral capabilities is provided here in the verification strategy document [9].

Test-programs are translated into machine code by the "Toolchain", a cross-compiler to translate RISC-V assembly or C into native RISC-V machine code [11.] Typically, this is an ELF format file. In order to be usable by a Verilog or SystemVerilog simulation, the ELF must be translated into a HEX format. Fortunately, this is readily supported by most toolchains. A detailed discussion of the toolchain is beyond the scope of this paper, and the core-v-verif repository has additional information in its TOOLCHAIN "readme" files.

A "board support package" that matches the memory map of the testbench is read-in by the Toolchain. Additional information is available in the BSP readme. Thus, the core testbench implements a very low-level "bare-metal" programming environment for test-programs that are executed by the core.

C. UVM Verification Environment

It was recognized early that the core testbench would not be able to support the goal of verifying the CV32E40P to match the expectations of commercial standards for delivery and quality. It was therefore decided to deploy a UVM verification environment for the project. Part of the motivation for this came from the availability of ***riscv-dv***, the Google pseudo-random instruction stream generator which is written in SystemVerilog as a collection of classes that extend several UVM base classes. Another motivation is that SV/UVM is the most popular verification methodology in use today by commercial silicon and IP vendors [10] and using a methodology familiar to industrial users will ease CORE-V adoption by commercial organizations looking to integrate RISC-V into their products.

The OpenHW CORE-V verification environment is implemented in SystemVerilog following established industry approaches for both design implementations (RTL) and test benches. Again, the OpenHW approach is to publish the verification environment as an open-source reference, allowing CORE-V adopters to re-create our results for themselves, or as a platform for further development. The key features of this verification environment are its ability to accept either manually written or automatically generated stimulus; on-the-fly, per-instruction, validation of the core's state against an instruction accurate reference model; asynchronous interrupts and debug requests and a complete functional coverage model. The structure of the core-v-verif UVM environment is illustrated below in Figure 2.

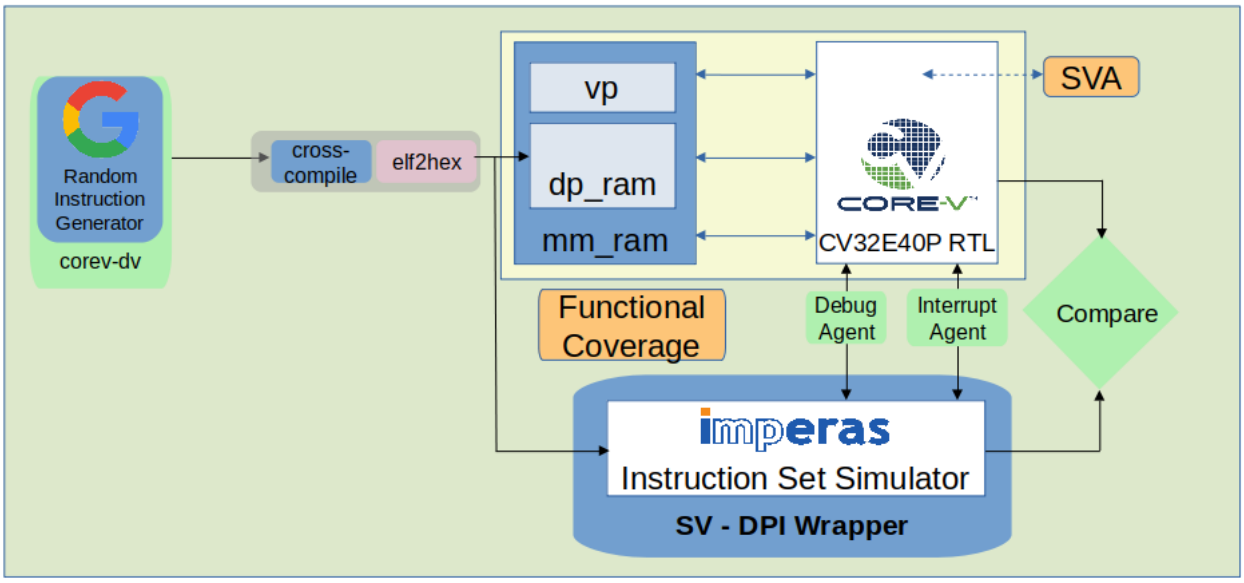

Figure 2: OpenHW CORE-V Verification Environment

As with the core testbench, the test-program generator can be a human who manually writes an assembly or C test-program. More typically, it will be ***corev-dv***, which is directly extended from the Google riscv-dv [13]. Our use of riscv-dv is discussed in detail later in this paper.

The toolchain used by the UVM environment is the same toolchain used by the core testbench. A goal of the UVM environment is to be able to run any test-program developed for the core testbench. Note that the inverse may not be true – a test-program developed for the UVM environment may not run as expected on the core testbench.

The core testbench used by the UVM environment is conceptually identical to the core testbench discussed above. It supports additional features that are either not required to achieve the goals of the core testbench or are not yet supported by Verilator. An example of the latter is a capability to add random cycle delays to the OBI bus protocol to stress the core's instruction fetch and data read/write features.

The UVM environment instantiates the core testbench as a component. Additional components are as follows:

- **Test-program Generation.** Another key feature of any verification effort is stimulus. Within the RISC-V community some test-programs are already available, most notably a set of reference testcases from the RISC-V International Compliance working group [3]. OpenHW uses these plus a small set of manually written test-programs to target specific features in the DV plan. Most stimulus for the CV32E40P verification effort used "riscv-dv" an open source instruction stream generator (ISG) from Google.

- **Reference Model**. Use of a reference model (RM) is viewed as a critical feature of the CORE-V environment. While the environment itself is reference model agnostic, based on experience within the RISC-V International Compliance working group, the Imperas OVPsim instruction set simulator was selected as the reference model for the OpenHW CV32E40P. The RM is tightly coupled to the core allowing for checks of the PC as well as all CSRs and GPRs at the retirement of each instruction. Thus, the UVM environment has the ability to flag errors on-the-fly at the moment they occur. Use of an RM also allows for test-programs to be non-self checking, which is a significant benefit when using pseudo-random program generation.

- **Step-and-Compare**. This is SystemVerilog module that maintains synchronization between the RM and RTL, enabling the checks of the PC, CSRs and GPRs. This module is explained in detail in this paper.

- **Functional Coverage Models**. Functional coverage is a primary mechanism for measuring the completeness of the verification effort against the goals of the DV plan. Coverage is used to close the loop from verification intent, as captured in the DV plan to what is achieved in regression. Adopters of the CORE-V verification environment can use their own EDA tools to run the environment and generate their own coverage reports.

- **UVM Agents for Debug and Interrupts**. Debug requests and external interrupts can be driven by either a virtual peripheral in the core testbench memory model, or UVM agents dedicated to these functions. The primary benefit of using the UVM Agents is that the debug requests and interrupts are defined by UVM sequences that are wholly asynchronous to the program running on the core's RTL model. This reduced the effort required to close coverage and also realized many corner case conditions in simulation that would have otherwise been difficult to hit. Driving debug requests and interrupts via the virtual peripherals allows for test-program control of these events which eased the ability to hit specific cases in a deterministic manner. An important item to highlight is that the debug requests and interrupts generated by either the Agents or the virtual peripherals were directly connected to both the core RTL model and the Reference Model.

D. CORV-VERIF repository organization

The following assumes the reader has at least some familiarity with git, GitHub and Make. The task of compiling and running simulations in core-v-verif is implemented using make. The Makefiles may optionally invoke support scripts written in Python or bash.

Core-v-verif resides at https://github.com/openhwgroup/core-v-verif. Cloning the repository into a directory pointed to by a shell environment variable CVV_HOME yields the follow directory tree:

$CVV_HOME

├── bin	Scripts for regressions, etc.
├── ci	User-level continuous integration script
├── core-v-cores	Location of the RTL
├── cv32	CV32E40P-specific verification files and directories
├── cva6	CVA6 -specific verification files and directories
├── docs	GitHub Pages source for coverage reporting
├── lib	Common verification components used by all CORE-V projects
├── vendor_lib	Libraries provided by third-parties
└── README.md	Context-specific user documentation

The README is a key resource for users. As much as possible we have attempted to place context-specific documentation at each level of the repository in a README. GitHub automatically renders this file at the bottom of each page, and in theory, everything required for a new user to be productive with core-v-verif should be easy to find.[1]

Note that core-v-verif does not contain the RTL source files for the device-under-test, nor are these cloned as a git sub-module. The Makefiles used to compile the environment will automatically clone the appropriate version of the RTL from its repository and place it in the **core-v-cores** directory. Thus, when core-v-verif is first cloned, the core-v-cores directory contains only a README file. Running a simulation of the CV32E40P will cause *make* to test for existence of the RTL and if not found will clone https://github.com/openhwgroup/cv32e40p to **$CVV_HOME/core-v-cores/cv32e40p**.

Similarly, the source files for the Google riscv-dv ISG and the RISC-V compliance test-suite are not stored in the core-v-verif repository. These will be cloned by the Makefiles as needed. For example, running a compliance

1 If this is *not* the case, please do open an issue.

regression will automatically clone a specific hash of https://github.com/riscv/riscv-compliance to **$CVV_HOME/vendor_lib/riscv** (unless it has already been cloned by a previous command).

As with the RTL, these repositories are ***not*** git sub-modules: using the –recursive command-line argument to clone core-v-verif will not clone the RTL, riscv-dv or riscv compliance repositories.

Except for libraries (both local and vendor supplied), all of the tests and components used to implement both the core testbench and UVM environment are in the cv32 directory:

$CVV_HOME/cv32

├── bsp	Board support package for core
├── env	UVM environment
├── README.md	Context-specific user documentation
├── regress	Regression configuration files (yaml)
├── sim	Simulation directories
├── tb	Testbench files for both core and UVM
└── tests	test-programs

The tests directory is the location of all test-programs and UVM testcases. In both the core testbench and UVM environment, a test-program is machine code that runs on the simulated core. The UVM environment also has the notion of a testcase, which is a UVM class that instantiates and configures the UVM environment. Further details about this are provided in the Verification Strategy [9]. Note that the core testbench does not support the notion of a testcase as all "stimulus" in the core testbench is provided directly in the form of test-programs running on the core.

Generated test-programs are placed in the **$CVV_HOME/cv32/tests/corev-dv** directory.

E. Running Tests

The user runs a simulation on the core testbench from the **$CVV_HOME/cv32/sim/core** directory and on the UVM environment from the **$CVV_HOME/cv32/sim/uvmt_cv32** directory. Extensive user documentation detailing the required tooling and command-lines for running simulations is provided in the associated README file.

IV. IMPLEMENTATION DETAILS

In this section we discuss some of the key implementation details of core-v-verif, specifically as it pertains to CV32E40P.

A. Using a Reference Model for Step and Compare

The integrated reference model (RM) is used in a step-and-compare mode in which the RM and register transfer level (RTL) execution are always in sync. Step and compare is invaluable for debug because the RM and RTL are executing the same instruction in the same compare cycle. Step and compare is implemented in *cv32/tb/uvmt_cv32/uvmt_cv32_step_compare.sv* in [1]. No modifications to the RTL is required for step and compare but the reference model must support instruction stepping. The testbench must be capable of individually causing the RM and RTL to execute an instruction to retirement, the specifics of which are explained in the following section.

1) Step:

Stepping is implemented as a 4-state machine as depicted in Figure 3. The simulation begins with Step RTL=1 which causes the RTL to be clocked. Once the RTL retires an instruction, indicated by an RTL retire event from the tracer, the RM is commanded to step until a RM retire event. Once both the RTL and RM have retired an instruction the results can be collected and compared. After comparison, the RTL is clocked again until retirement and the cycle repeats. The RTL throttles the RM. The tracer is found in module cv32e40p_tracer which is implemented in bhv/cv32e40p_tracer.sv in [2].

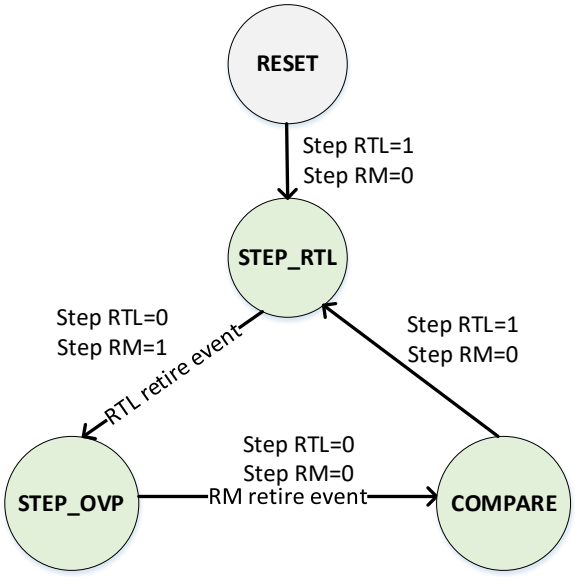

Figure 3: Step and Compare state machine

As an example, consider the objdump snip in Figure 4.

```
12c: csrw mepc,a2
130: jal ra,961c
```

Figure 4: Snip of objdump

This code is executed by the RM and RTL and compared as shown in Figure 5 where the instruction *at PC=0x12C* is converted to *csrrw x0, x12, 0x341* by the tracer. The ABI name of register *x12* is *a2* and the MEPC register is at address 0x341. The instruction at PC=0x130 is converted to *jal x1, 38124* by the tracer. The ABI name of register *x1* is *ra* and the offset to address 0x961c from 0x130 is decimal 38124. The RTL retires the jump instruction at the time marked as 1 and the RM retires the jump instruction at the time marked as 2. In 0 simulation time, the compare is completed so the COMPARE state is not shown but is indicated by the Compare event.

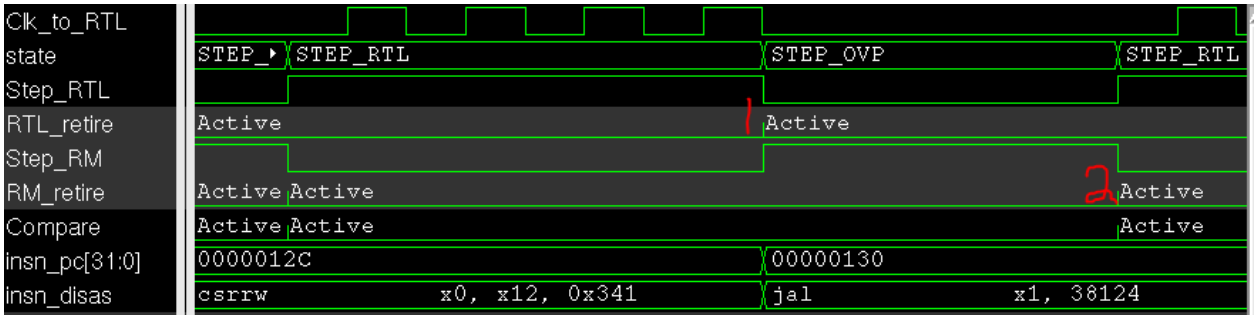

Figure 5: Step and compare for jump instruction

2) Compare:

In the COMPARE state the *compare()* function is called which compares the PC, General Purpose Registers (GPRs) and Control and Status Registers (CSRs).

GPR Comparison

For GPR comparison the actual RTL GPR register, *<gpr>_q* is compared against the RM GPR value. However, for some instructions when the RTL retire event is triggered *<gpr>_q* may not yet have updated. For this reason, the tracer maintains queue *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 one, which implies that only 0 or 1 GPR registers change as a result of a retired instruction.

Figure 6 demonstrates that for a *lw x8, 24(x2)* instruction the GPR value is updated one clock cycle after the RTL retire signal. The load to *x8* is retired but RTL value *RTL_GPR[8]* has not updated to *0x364C* 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 <gpr>*_q* can be used.

If the size of queue *insn_regs_write* is one, the GPR at the specified address is compared to that predicted by the RM. The remaining 31 registers are then compared to the RM. 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 zero, all 32 registers are compared, *<gpr>_q* is used for the observed value.

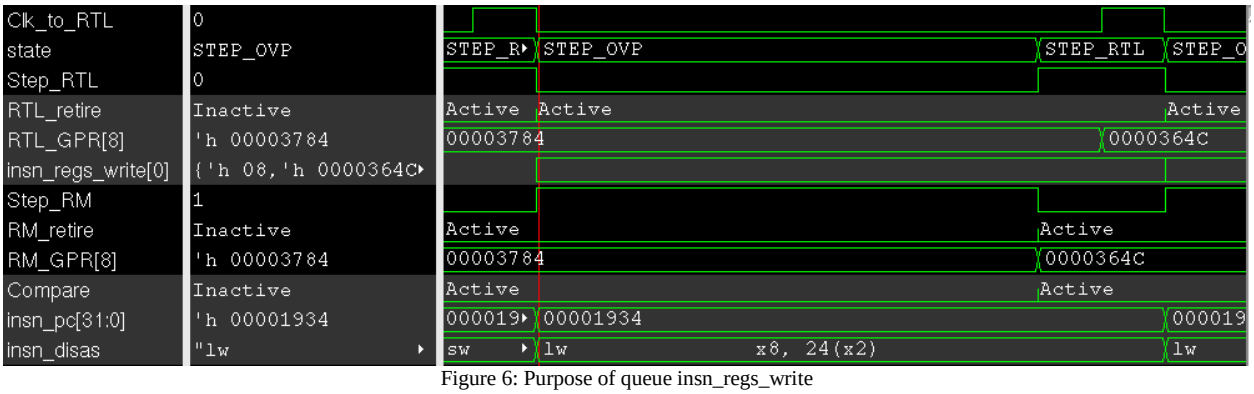

Figure 6: Purpose of queue insn_regs_write

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 at 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.

B. Stimulus Generation

1) Google:

IP verification typically involves a large planning and development effort to scope, develop, test, and measure random stimulus generation using the available constrained-random features of UVM and SystemVerilog. For processors this concept extends into the idea of an *Instruction Set Generator* (ISG), which can generate random instructions that exercise all intended features of an ISA for the processor. The general state space for ISGs, even in a relatively simple ISA such as RISCV32IMC is large. The ISG must randomize effectively to cover the instruction set, generate illegal and corner conditions effectively while maintaining a coherent state for such concepts as interrupt handlers, stacks, CSRs and other features that cannot be randomized blindly,
To efficiently utilize the available engineering resources of OpenHW the decision was made to utilize the *riscv-dv* ISG provided by Google as an open-source project [4]. The riscv-dv project provides a SystemVerilog package which contains data structures for all RV32IMAFDC instructions, infrastructure for generating handlers and other special sequences, a suite of general random sequences to exercise a RISC-V processor, and extensions to create constrained-random "directed" sequences to extend the capabilities of the ISG. Note that riscv-dv also provides co-simulation and coverage capabilities, but those features were not utilized by OpenHW in the verification of the CV32E40P.

To properly utilize ***riscv-dv*** in a core verification environment, configuration and customization is necessary to properly generate the correct instructions for the supported RISCV extensions in the CV32E40P. The core-v-verif environment includes the riscv-dv code-base as an immutable package within its build scripts. The scripts will clone a known-working fixed Git hash of riscv-dv in a fixed location and compile that package into the larger verification

environment. Any customization or configuration of riscv-dv extends from provided hooks utilizing object-oriented programming practices (e.g. virtual methods, the UVM factory) or provided configuration hooks from riscv-dv. The core-v-verif customization layer is referred to as *corev-dv* and is the subject of the rest of this section.

The *corev-dv* configuration tunes the *riscv-dv* generator for the following list. These configurations are included in the *riscv_core_setting.sv* file as a supported hook for configuring *riscv-dv* (and *corev-dv).*

- RV32IMC
- Machine-mode only supported
- No PMP
- Debug mode supported
- Unaligned load-store supported
- No vectors
- Illegal instructions, breakpoint, ecall exceptions supported

Even with the above configuration, further customization is required to *extend* the functionality of *riscv-dv*. To support this, *riscv_instr_gen_config.sv* was extended using object-oriented inheritance to provide configuration hooks (i.e. plusargs) for the user to configure *corev-dv* features as well as adding random constraints to randomize these knobs in a legal fashion.

The CV32E40P introduces additional machine-mode CSRs not directly specified in the RISCV Privileged Specification. This functionality includes the addition of 16 additional interrupt sources added to the Machine Interrupt-Enable Register (MIE) and Machine Interrupt-Pending Register (MIP). By simply inheriting from the *riscv_privil_reg* class and reimplementing the *init_reg* method, the necessary fields for the additional external interrupt sources were added to the CSR definitions and included in the random CSR configuration already supported by *riscv-dv.*

In a typical *riscv-dv* implementation interrupt verification is performed by generating a handshake in interrupt handlers that will communicate to the testbench that a handler is entered. The core-v-verif does not use this handshake to verify interrupt handler entry and interrupt state as mentioned above. This gives the ISG more flexibility to generate arbitrary interrupt handler code. For interrupt verification *corev-dv* supported DIRECT or VECTORED mode of interrupt handlers per the RISCV Privileged Specification as a configuration randomization option. If VECTORED interrupts are selected, *corev-dv* further extends interrupt verification by randomly selecting some handlers to be implemented with a single *mret* instruction. As the CV32E40P supports a hardware acknowledge mechanism via the *irq_ack_o* signal and the interrupt UVM agent also supports this, interrupt functionality can be more effectively stressed by creating a very minimal interrupt handler. Additionally nested interrupt functionality was extended and better supported in *corev-dv* using CSR stack-save mechanisms as defined in the CV32E40P User Manual.

The debug ROM generation was also extended in *corev-dv.* The CV32E40P supports an external debug request signal that must be honored to place the core into debug mode at any point in execution. Therefore the code generated by the ISG must be robust with respect to its stack state to ensure that the core can be interrupted via debug request at any point in time. The debug ROM generation was extended in *corev-dv* to support an additional stack and stack register (designated *dp*) as a debug stack to be used by the debug ROM routine. Using the debug stack provided the stability to ensure that EBREAKs, Single-stepping, and external requests could occur randomly at any point in time to re-enter the debug ROM. The debug ROM was also extended to include random *wfi* instructions to ensure that these are converted to NOPs randomly at any point in time during debug mode. Note that the debug DCSR.EBREAK and single-step modes for debug usage in *riscv-dv* were preserved in the new debug ROM generator.

Some directed sequences were added via *corev-dv* to increase testing effectiveness and overall coverage. Some of these additions were directly initiated due to coverage analysis and some to fulfill verification plans (e.g. the interrupt verification plan). Two directed sequences were added for interrupt verification: *corev_interrupt_csr_instr_stream* and *corev_interrupt_wfi_csr_instr_stream*. This instruction stream injects random writes to the MIE register and MSTATUS.MIE during random instruction operation. In conjunction with the random interrupt assertion/deassertion supported by the UVM agent this stream ensured that these CSRs were

correctly implemented in the CV32E40P's interrupt controller. Cover properties developed on each interrupt source ensured not only that all interrupt enable/asserted states were covered but that interrupt CSR enable changed during each possible interrupt signal state. The *wfi* instruction stream inherits from the *corev_interrupt_csr_instr_stream* with the only change being to ensure that MIE is never set to a zero vector. If this were to occur then the next WFI instruction would cause the core to remain in sleep indefinitely as no interrupts could wake up the core. Note however that MSTATUS.MIE has no effect on whether an enabled interrupt will wake up the core-it only affects where an interrupt routine is entered after the core awakens.

Other directed streams were added to address functional coverage holes as the CV32E40P coverage was analyzed. An ECALL sequence *corev_ecall_instr_stream* was added to include random ECALLs during execution. In typical *riscv-dv* implementations the ECALL instruction is used to end the test. However, *corev-dv* uses a simple jump to a *test_done* routine to complete a test. Therefore, the ECALL instruction may be called at any time in a *corev-dv* instruction stream. Two additional instruction streams were added to increase specific instruction *transition* holes (i.e. sequence of instructions). Instruction stream *corev_jal_wfi_instr* extended the *riscv_jal_instr* stream to inject WFI instructions within JAL instructions. Instruction stream *corev_jalr_wfi_instr* is a new sequence created to address coverage holes with JR and JALR and WFI instructions.

2) Directed:

Some aspects of the CV32E40P instruction set were not modelled in the Instruction Set Simulator used during testing. This mostly revolved around the XPULP extension, which in and of itself is unique from the unprivileged RISCV ISA [6] and utilizes areas of the instruction space that are available and not reserved for future use in the specification. In order to test the XPULP extension, directed tests were required as our golden model had to be disabled while testing the extension. This resulted in around 15,000 lines of assembly being written in order to test the 330ish instructions in the extension. Thankfully, much of the framework of the firmware to test the instructions can be reused with only minor modification. The framework consists of several steps to test each instruction. First, the starting values are loaded into registers for use by the instruction. Then, the instruction in question is executed. Next, the correct output value is loaded into a separate register, and the resultant value is compared with the expected value. If the values match, the firmware continues testing. Otherwise, it increments a register being used as an error counter before continuing testing. This continues throughout the firmware until the end of the firmware, where the error count register is check, and if the value is zero, the firmware writes to a virtual peripheral denoting that the test has passed. If the error count is not zero, the firmware will write to the same peripheral, denoting that the test has failed. Each XPULP instruction is tested 6 times in this manner. A short example of this code can be seen in Figure 7.

```
1  test71:
2      li x17, 0x9c24b7e7
3      pv.maxu.sci.b x19, x17, 63
4      li x20, 0x9c3fb7e7
5      beq x20, x19, test72
6      c.addi x15, 0x1
7  test72:
8      li x17, 0x809e5f46
9      pv.maxu.sci.b x19, x17, 12
10     li x20, 0x809e5f46
11     beq x20, x19, exit_check
12     c.addi x15, 0x1
13 exit_check:
14     lw x18, test_results /* report result */
15     beq x15, x0, exit
16     li x18, 1
17 exit:
18     li x17, 0x20000000
19     sw x18,0(x17)
20     wfi
```

Figure 7: Sample assembly code of directed XPULP test

In the above example, a virtual peripheral at memory location 0x20000000 is written to based on the result of the test, writing a value of 1 in the case of a failure, and a value of test_results (a global defined earlier in the test) if the test has passed.

This method of testing is rather inefficient and prone to have errors, but this was deemed the best option for providing at least limited coverage of the XPULP instructions without using the ISS. With this limited coverage of the XPULP instructions, it is noted in the CV32E40P User Manual that while the XPULP extension is in the core and can be enabled, it is not completely validated [7]. ISS support and complete validation of the XPULP extension are planned for the future, but no timeline has been approved for that yet.

Additionally, directed testing was utilized to validate several smaller aspects of the exceptions specification in the RISCV privileged ISA specification [8]. These include testing the results of loading values into x0, trying to use reserved instructions slliw, srliw, and sraiw, and usage of instructions ebreak, c.ebreak, and ecall. This test utilized a more complex exception handler than other tests and incremented a counter by different amounts depending on which type of exception occurred. This value was then compared to a hardcoded value at the end of the test to ensure that the correct amount of each exception type was executed during the test. An example of this can be seen in Figure 8.

```
    csrrc x31, mtvec, x0
    beq x31, x30, continue_check
    lui a3, 0x1
    add x26, x26, a3
continue_check:
    li t6, 0xf
    csrrc t5, mcause, x0
    and t6, t6, t5
    li t4, 2
    bne t6, t4, _check_3
    addi x26, x26, 0x1
    csrrc s0, mepc, x0
    c.addi s0, 4
    csrrw x0, mepc, s0
    j _end_trap_Generic_Handler_ASM
_check_3:
    li t4, 3
    bne t6, t4, _check_11
    addi x26, x26, 0x10
    csrrc s0, mepc, x0
    c.addi s0, 2
    csrrw x0, mepc, s0
    j _end_trap_Generic_Handler_ASM
_check_11:
    li t4, 11
    bne t6, t4, _end_trap_Generic_Handler_ASM
    addi x26, x26, 0x100
    csrrc s0, mepc, x0
    c.addi s0, 4
    csrrw x0, mepc, s0
```

Figure 8: Custom exception handler for directed testing

For the CV32E40P the CSR mtvec is used to list the value of the exception or interrupt that just occurred. By reading it and checking the bottom four bits (exception code) the type of exception that is occurring can be detected. Exception code 2 coincides with illegal instruction exceptions and increments the counter in register x26 by 0x1. Exception code 3, which is for ebreak and c.ebreak instructions increments the counter by 0x10, and finally ecall instructions (exception code 11) increment the counter by 0x100.

Another aspect of the system that was tested in a unique way is the decoder, which was tested by running large quantities of illegal instructions to ensure that the core did not incorrectly handle or detect any instruction codes that are considered invalid by the compiler. To accomplish this, a script was created that generates a large number of

randomized, 32-bit hex values, and then compiles them to see which ones are valid instructions. Upon parsing the compiled objdump file of the randomized instructions, all legal instructions are deleted, leaving a test of exclusively illegal instructions. An example of the code for the script can be seen in Figure 9.

```
1  #!/usr/bin/perl
2
3  open OUT_FILE, '>', "illegal_instr_temp.S" or die "Can't open illegal_instr_temp.S";
4  my @set = ('0' .. '9', 'a' .. 'f');
5  for ($i = 0; $i<$ARGV[0]; $i++){
6      $str = join '' => map $set[rand @set], 1 .. 8;
7      $str = (".word(0x" . $str . ")");
8      print OUT_FILE ("$str\n");
9  }
10 close OUT_FILE;
11
12 #compile .S file into .o file
13 #compile .o file into .objdump file
14
15 open my $fh, '<', "illegal_instr_temp.objdump" or die "Can't open illegal_instr_temp.S";
16 while(<$fh>){
17     chomp;
18     @fields = split(/\s+/);
19     if($fields[3]=~/^0x[0-9a-f]{4,8}$/i){
20         $str = (".word(0x" . $fields[2] . ")");
21         $illegal_instr{$str} = 1;
22     }
23 }
24 close $fh;
25
26 open my $fh, '<', "illegal_instr_temp.S" or die "Can't open illegal_instr_temp.S for parsing";
27 open OUT_FILE, '>', "illegal_instr_test.S" or die "Can't open illegal_instr_test.S";
28 #print globals and startup instructions
29 while(<$fh>){
30     chomp;
31     $comp_string = $_;
32     if(exists($illegal_instr{$comp_string})){
33         print OUT_FILE "$comp_string\n";
34     }
35 }
36 close $fh;
37 close OUT_FILE;
38
39 #compile .S file into .o file
40 #compile .o file into .objdump file
```

Figure 9: Illegal instruction generation script

In the example, first a file of 32 bit random values is generated with the length of the file being defined by the user (lines 3-10). Then the file is compiled into an objdump file (lines 12-13) and the objdump is parsed to find which 32 bit values are illegal instructions and store the data in a hash for later use (lines 15-24). This step uses the fact that the objdump file in question uses the hex value instead of the mnemonic if there is no mnemonic associated with the given hex value (since it's an illegal instruction). Then, the script takes the data from the hash of illegal instructions and writes a complete test in assembly that is ready for use in testing (lines 26-37). It should also be noted that when using this script, $ARGV[0] defines the size of the initial list, and only 9-9.5% of all values generated are actually illegal instructions.

3) Random:

The core-v-verif supports a robust and comprehensive random test flow that enables complete functional coverage of the CV32E40P. The random test environment truly decouples stimulus generation from checking methodology to ensure that any stimulus can be verified using assertion checking developed from verification specifications and the step and compare ISS methodology described above.

The stimulus generation for random tests consists of a combination of the corev-dv instruction set generator as described above. A user may define a random test configuration in a YAML specification file that consists of a series of plusargs that are read by the corev-dv tool. The YAML file includes all of those plusargs supported by the

original Google riscv-dv generator as well as plusargs introduced to control and configure the corev-dv added sequences. The corev-dv generator is compiled and executed first in the process of a test execution to generate a randomly generated assembly test program. The following is an example of a corev-dv YAML configuration file.

```
name: corev_rand_interrupt_debug
uvm_test: corev_instr_base_test
description: >
    RISCV-DV generated random interrupt tests with exceptions
plusargs: >
    +instr_cnt=5000
    +num_of_sub_program=15
    +directed_instr_0=riscv_load_store_rand_instr_stream,4
    +directed_instr_1=riscv_loop_instr,4
    +directed_instr_2=riscv_hazard_instr_stream,4
    +directed_instr_3=riscv_load_store_hazard_instr_stream,4
    +directed_instr_4=corev_interrupt_csr_wfi_instr_stream,5
    +no_fence=0
    +enable_interrupt=1
    +enable_fast_interrupt_handler=1
    +randomize_csr=1
    +boot_mode=m
    +no_csr_instr=1
    +no_ebreak=0
    +gen_debug_section=1
    +set_dcsr_ebreak=1
    +illegal_instr_ratio=0
    +no_wfi=0
```

Figure 10: Random corev-dv generation control YAML

After generating the corev-dv assembly file, the standard directed test flow is invoked to compile the assembly file using the configured COREV toolchain. The random assembly program is then loaded and executed on the CV32E40P. The core-v-verif testbench has additional features to perturb the core during execution of the test program. All of the following features are enabled (or disabled if on by default) via plusargs defined in the test YAML specification file. These include:

- Randomly stalling the external OBI bus interfaces (on by default)
- Randomly toggling the *fetch_enable_i* input
- Changing bootstrap pin inputs (e.g. default mtvec boot address)
- Debug request on reset
- Random external debug request assert/deassert
- Random external interrupt request assert/deassert

C. Coverage Results

The functional and code coverage results for the CV32E40P have been published to the web. At the time of this writing, the URL for these data is https://mikeopenhwgroup.github.io/core-v-docs/. It is anticipated that this will move to a more permanent location and there will be a link to it from the core-v-verif home README page.

IV. Future Work

The OpenHW Group is actively integrating the CVA6 core into the core-v-verif environment. There are also plans for the next generation of CV32E4 and a possible CV32E2 core. In order to ensure the core-v-verif project can scale to support multiple, simultaneous CORE-V projects the team is considering a number of future enhancements:

- A common Tracer/Interposer interface, similar in concept to the proposed RISC-V Formal Interface (RVFI). This interface would simplify the effort required to integrate the core with the step-and-compare module.

- Integration of the test-program generator. The Google riscv-dv generator is written in SV/UVM, but it is not a UVM component in the strictest sense. An update to either riscv-dv or the corev-dv extensions to encapsulate the generator into an extension of the uvm_component base class would allow the UVM

environment to have direct control of the generator at run-time, allowing for a single-step simulation and run-time control of generation constraints.

- Toolchain by-pass. Direct generation of machine code would eliminate the environment's dependency on toolchains.

- Score-boarding of retired instructions. The step-and-compare method requires the use of a cycle accurate Reference Model. Complex cores with out-of-order execution pipelines will therefore each require their own core-specific RM, each of which requires significant development effort. Replacing step-and-compare with transaction scoreboards may be a practical solution.

- Alternative generators. FORCE-RISCV is an open-source instruction stream generator developed by Futurewei with a proven industrial track record. This generator will almost certainly be integrated into core-v-verif in the coming months.

- Alternative reference models. There are multiple instruction set simulators available, both open and closed source. It is desirable to support an environment that allows users of core-v-verif to select an RM that meets the specific needs of their project.

GET INVOLVED!

The OpenHW Group is a member-driven organization dedicated to the development of open-source RISC-V IP and related artifacts. The real work of writing specifications, RTL design and verification is performed by individual and corporate members coming together for a common purpose. If this is something that interests you or your organization, contact us at openhwgroup.org.

ACKNOWLEDGMENT

There are many more contributors to core-v-verif than the authors of this paper. The authors thank you for your invaluable contributions to OpenHW.

REFERENCES

[1] CORE-V SV/UVM verification environment. https:/github.com/openhwgroup/core-v-verif.
[2] CV32E40P RTL. https://github.com/openhwgroup/cv32e40p
[3] RISC-V International Compliance test suite. https://github.com/riscv/riscv-compliance
[4] SV/UVM based instruction generator for RISC-V processor verification. https://github.com/google/riscv-dv
[5] FORCE-RISCV. https://github.com/openhwgroup/force-riscv
[6] RISCV Unprivileged ISA. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf
[7] CV32E40P User Manual. https://core-v-docs-verif-strat.readthedocs.io/projects/cv32e40p_um/en/latest/index.html
[8] RISCV Privileged ISA. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf
[9] OpenHW Group CORE-V Verification Strategy. https://core-v-docs-verif-strat.readthedocs.io/en/latest/
[10] Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study.
[11] Embecosm RISC-V and CORE-V Toolchain downloads are available at https://www.embecosm.com/resources/tool-chain-downloads/
[12] Verification Plans: the five-day verification strategy for modern hardware verification languages. James, Peet, Springer.
[13] riscv-dv, a SV/UVM based open-source instruction generator for RISC-V processor verification. https://github.com/google/riscv-dv
[14] https://github.com/openhwgroup/core-v-docs/blob/master/cores/cv32e40p/OBI-v1.0.pdf