RISC-V Processor Verification: Case Study

Adi Maymon, Shay Harari
Nvidia Networking
Mevo Sivan Street 1
Kiryat Gat
Israel

Lee Moore, Larry Lapides
Imperas Software Ltd.
Imperas Buildings, North Weston
Thame, Oxfordshire
United Kingdom

***Abstract*- The open RISC-V instruction set architecture is gaining traction with both semiconductor vendors and systems companies. A key question is how to verify the RISC-V processor implementation, especially when developing the RTL and/or adding custom instructions? This paper reports on the techniques used and lessons learned for the verification of a RV64IMACBNSU RISC-V processor by an experienced SoC design team, including the development of the reference model and the SystemVerilog and C encapsulation of the reference model, the step-and-compare flow used included co-debug, and the Specman-based verification environment.**

I. Introduction

The RISC-V instruction set architecture (ISA) has been gaining momentum in the semiconductor community in part because of the design freedoms of the open specification. This openness enables adopters of RISC-V to add custom instructions to develop domain-specific processors, however, the openness also requires significant verification effort. The verification effort is obvious for the custom instructions; however, dedicated verification effort is also needed for the processor RTL implementation that has been built per the RISC-V specification.

Since processor IP verification has been done primarily by the processor IP vendors for the last couple of decades, some of the techniques have been "lost" to the general SoC design community. Several companies and organizations in the RISC-V community have been focused on assisting with design verification (DV) of RISC-V, including reference models, simulation methodology, test generation, compliance and functional coverage. The goal of this paper is to report on the techniques used and lessons learned for the verification of a RISC-V processor by an experienced SoC design team.

Within the RISC-V International organization, the body charged with evolving and maintaining the RISC-V ISA specification, "verification" efforts have focused on compliance testing. The RISC-V International Compliance Working Group (CWG) has been in place for a few years, and has developed a test framework and various test suites that make use of a RISC-V reference model, typically an instruction set simulator (ISS). The framework and test suites are available at no charge from the CWG GitHub repository [1]. To date, the ISS most often used as a reference model has been the Imperas riscvOVPsim ISS.

Compliance testing, however, is not the same as verification. Compliance testing confirms that the implementation adheres to the specification at the architectural level. Compliance testing does not verify the functionality of the implementation at the micro-architectural level.

For RTL verification, in the RISC-V community we have the same long-standing discussion as for SoC DV: static versus dynamic verification techniques, or formal versus simulation-based flows. Of course, both flows have their place for RISC-V processor verification, as for other processor architectures and for SoCs. However, in this paper we are focused only on simulation-based DV flows, and will discuss the development of the reference model and the SystemVerilog and C encapsulation of the reference model, the step-and-compare flows used included co-debug, and the Specman environment.

II. Verification Flows

A. RISC-V Processor Description

The processor being developed is a RISC-V RV64IMACBNSU, meaning it is a 64-bit RISC-V core with integer, multiply, atomic, compressed and bit manipulation instructions (IMACB), plus machine, supervisor and user privilege modes. The processor includes 4 harts (4 hardware cores) and also supports the RISC-V Debug mode. There are a few custom instructions added to the processor.

B. Design Verification Overview

The DV environment consists of the Cadence Xcelium RTL simulator and Specman verification tools, Imperas Open Virtual Platforms (OVP) RISC-V reference models and Imperas simulators, and various test generation tools including the Google open source riscv-dv instruction stream generator (ISG) [2]. The processor RTL is run in two different step-and-compare flows with the reference models. Step-and-compare is needed because trace-based DV flows are not as efficient in the use of simulation resources [3], and are much more difficult, if not impossible, to use for verification of certain features having to do with asynchronous events (such as interrupts) and privilege modes. In this case, a simple step-and-compare flow was used with the free Imperas riscvOVPsim simulator/model and the riscv-dv ISG for verification of the pieces of the processor that conform to the RISC-V user mode specification. A more complex step-and-compare DV flow was used with a custom OVP RISC-V model and the Specman tool for the verification of custom instructions, asynchronous events and privilege modes.

III. RISC-V Reference Models

The reference models are built using the OVP APIs, and are based on the RISC-V envelope model available from the OVP website [4]. This RISC-V envelope model contains over 160 parameters to cover a variety of implementation-specific configuration choices. These include supporting all versions of all the specifications, the various RISC-V instruction subsets and other, more complex functionality. Examples of the configuration parameters available in the envelope model are shown in Table 1.

Custom features, including custom instructions and registers, are added to the model using an external (extension) library. An example of the decode table for the custom instructions is shown here, with the custom instruction in this example being from the ChaCha20 encryption algorithm.

```
//
// Create the RISCV decode table
//
static vmidDecodeTableP createDecodeTable(void) {

        vmidDecodeTableP table = vmidNewDecodeTable(RISCV_INSTR_BITS, RISCV_EIT_LAST);

        // handle custom instruction
        DECODE_ENTRY(0, CHACHA20QR1, "|0000000.....000.....0001011|");
        DECODE_ENTRY(0, CHACHA20QR2, "|0000000.....001.....0001011|");
        DECODE_ENTRY(0, CHACHA20QR3, "|0000000.....010.....0001011|");
        DECODE_ENTRY(0, CHACHA20QR4, "|0000000.....011.....0001011|");

        return table;
}
```

TABLE 1. OVP RISC-V ENVELOPE MODEL CONFIGURATION PARAMETERS

Parameter Category	Examples
Specification version	Privilege spec version 1.10, 1.11, 1.12
	Debug configuration spec version 0.13.4, 0.14
	Bit manipulation spec version 0.90, 0.91, 0.92, 0.93
	Vector spec version 0.71, 0.8, 0.9, 1.0
Simple parameters	MISA subsets 32/64 I, M, A, C, F, D, B, V, H, K, …
	Misaligned Code/Data access behavior
	CSR field/behavior configuration
	CLINT configuration
	CLIC configuration
	DEBUG configuration

While the base OVP model is open source, the extension library is under the control of the developers since the complete source tree is available under the Apache 2.0 open source license. The model is written in C, using the OVP

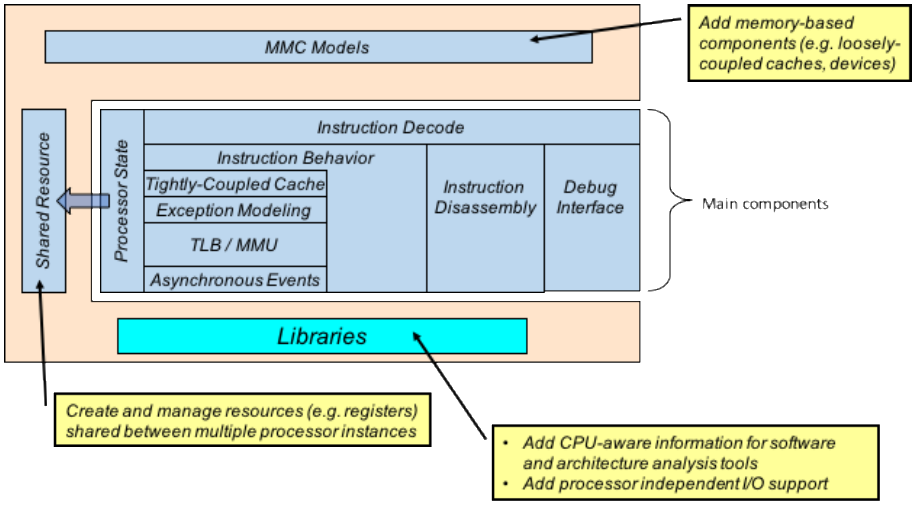

Figure 1. Block diagram of an Open Virtual Platforms (OVP) Fast Processor Model.

APIs, with those APIs supported by the Imperas simulators. A block diagram of a generic OVP processor model is shown in Fig. 1.

The RISC-V envelope model is developed, as are all OVP processor models, using a test-driven development (TDD) methodology. Essentially this means that as a new feature is added to the processor model a corresponding test is added to the model test environment. This test environment is automated, such that prior to a developer checking in new code for a model the entire set of tests is executed. This continuous integration (CI) methodology helps to ensure high model quality and increase developer productivity.

In addition, because riscvOVPsim has been the primary ISS used for RISC-V compliance testing, it has been tested against a wide range of implementations. This community-wide adoption of riscvOVPsim, together with the TDD development methodology and CI test methodology, has made the OVP RISC-V envelope processor model the highest quality RISC-V ISS in the industry.

riscvOVPsim uses only the RISC-V envelope model plus memory coupled to a basic simulation engine. There is no ability for the user to add custom features, however, riscvOVPsim does accommodate implementation-specific configuration options within the RISC-V specification through the use of parameters which define the configuration under test.

riscvOVPsim does have some limitations. While riscvOVPsim can generate a trace log and connect to GDB for debug, it does not have the appropriate APIs to enable encapsulation in a SystemVerilog environment. In addition to not supporting custom instructions, riscvOVPsim does not support multi-core ("multi-hart" in the RISC-V lexicon) processors.

The more complex step-and-compare flow uses the Imperas M*DEV simulator product, and a RISC-V processor model specific to the design under test (DUT). As discussed above, the RISC-V processor model used for this flow included both the base envelope model (configured with parameters to match the processor being developed) plus an extension library. Developers from both companies contributed to the development and maintenance of the extension library which supports the custom features.

IV. RISCVOVPSIM SIMPLE STEP-AND-COMPARE FLOW

The riscvOVPsim based "simple" step-and-compare DV flow is shown in Fig. 2. The goal of this flow is the verification of processor features that are fully compliant with the RISC-V specification and do not require an asynchronous event to test the functionality. These features, such as the I-integer instructions, are routine and relatively simple compared with features such as exception handling and Debug mode. The goal of this simpler flow is to reduce the relative effort and resources involved for verification of these simpler processor features.

This simple step-and-compare flow starts with running the test stimuli through the riscvOVPsim reference model and simulator and generating a trace log. Independent test stimuli can be generated for each hart, making use of the different address regions for code and data for each hart. This simulation runs quite fast, as the performance of riscvOVPsim is quite fast, typically greater than 250 million instructions per second (MIPS). The trace log is then incorporated into a Specman scoreboard, and the same test stimuli executed in the Xcelium/Specman environment

with the RTL implementation of the processor as the device under test (DUT). Within the Specman environment, the RTL state is compared to the trace log in the scoreboard after each instruction is executed.

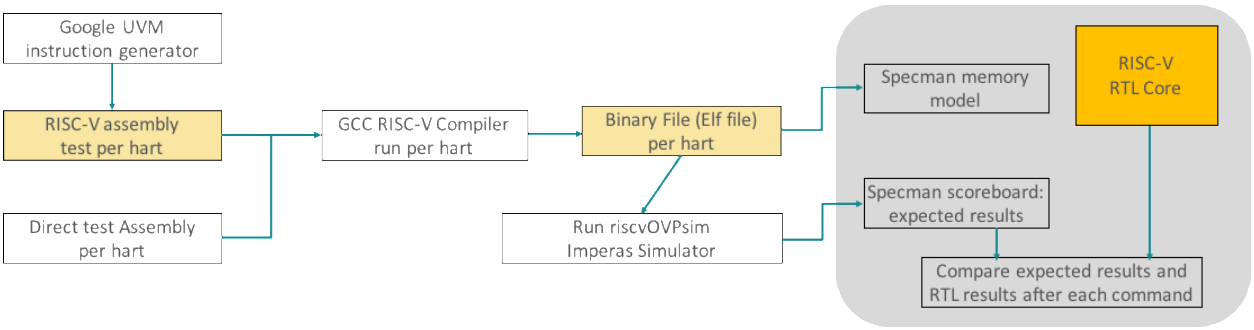

Figure 2. Simple step-and-compare flow with pre-generation of expected test results.

This flow has the advantages of enabling the simulation to stop after a compare failure, thus saving simulation resources. Also, as there is no need to encapsulate the reference model and synchronize that model with the RTL DUT, the RTL simulation can be run without any environment-related slowdowns.

V. Complex Step-and-Compare Flow with M*DEV Simulator

The complex step-and-compare flow is shown in Fig. 3. This is used to check interrupts, Debug mode, hardware triggers, performance counters and all custom features including custom instructions, misaligned access, 64-bit address space and PMA configuration.

The Specman encapsulation of the OVP processor model shown in Fig. 4 is similar to the SystemVerilog encapsulation discussed in Thompson et al [5]. While the SystemVerilog encapsulation consists of two pieces, the SystemVerilog wrapper and the Direct Programming Interface (DPI), the Specman encapsulation just uses a C API in Specman to interface to the C OVP APIs of the processor model. This encapsulation provides both the ports necessary for the reference model to communicate directly with the Specman verification environment and the introspection into the state of the reference model. This enables not only a fully lockstepped operation of the processor RTL and the processor model, but also operation mode where the processor model receives information from the RTL portion of the simulation to inform it of the direction that the RTL simulation has taken. The two simulators are run in sync based on instruction retirement.

This co-simulation of the RTL and reference model has another benefit: co-debug. Breakpoints can be set in either simulator, and the complete simulation halted based on the breakpoint in one or the other of the simulators. This is especially important for debug of asynchronous events and privilege modes. See Fig. 5.

The Specman environment is setup in a conventional manner, although there are some differences due to the goal of processor verification. The Specman environment consists of monitors, stubs, scoreboards and verifiers. This

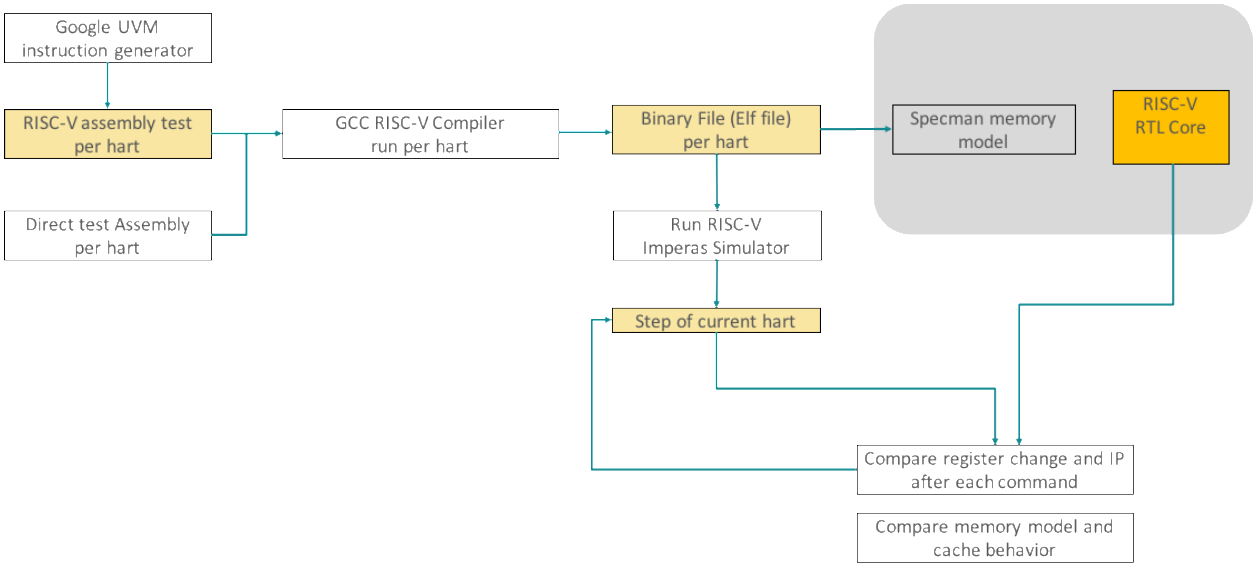

Figure 3. Step-and-compare verification flow for verification of asynchronous events and complex features.

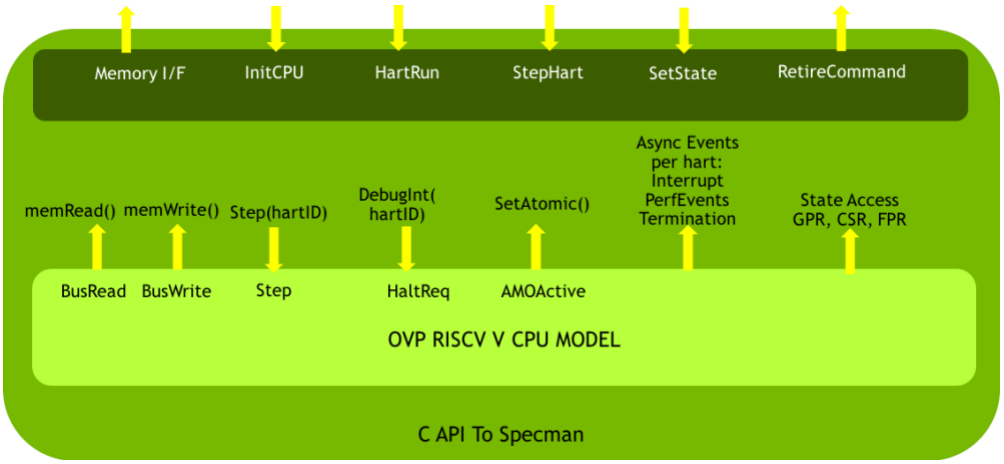

Figure 4. Encapsulation of the OVP RISC-V golden reference model using the Specman C API.

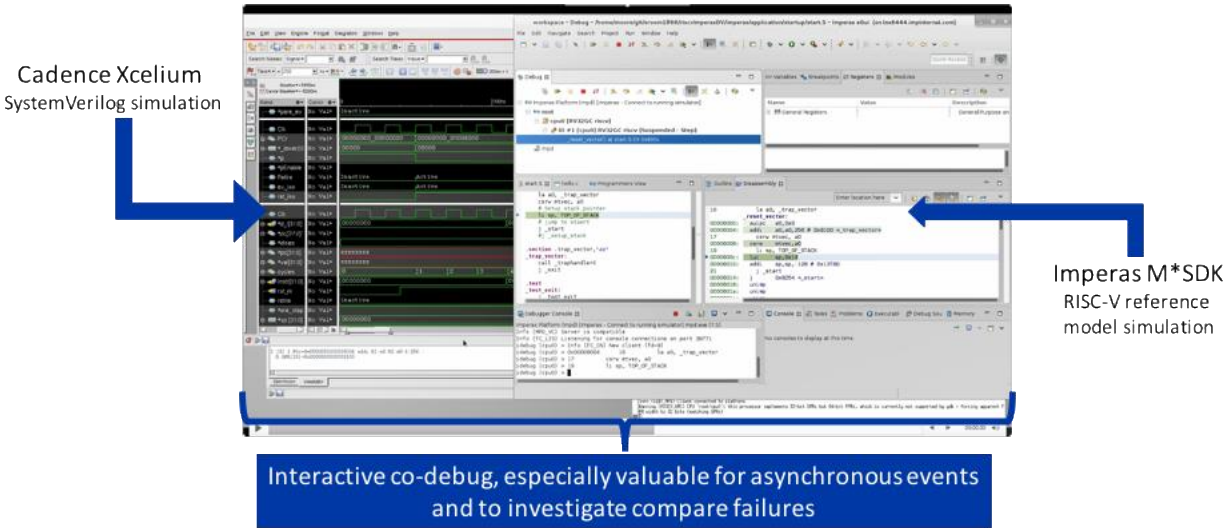

Figure 5. Co-debug for RISC-V processor DV.

environment randomly selects the flow, the processor and ISG configuration, and when and if asynchronous events will be injected. According to it, the ISG generates the binary file that will be loaded to both RTL and simulator.

The Specman environment compares the command, general registers, CSRs and memory after instruction retire indication from both design and simulator.

VI. Results

As expected, the simple step-and-compare flow was relatively easy to set up and run. The effort in this flow went to modifying the Google ISG to add support for the bit manipulation instructions, creating the directed tests for the custom instructions, automating the processor of moving the reference model simulation results into the Specman scoreboard and creating the instruction functional coverage points. A sample of the changes made to the riscv-dv code to support the Bit Manipulation instructions is shown below.

```
function bit [6:0] get_opcode();
        case (instr_name) inside
                ANDN, ORN, XNOR, GORC, SLO, SRO, ROL, ROR, SBCLR, SBSET, SBINV, SBEXT,
                 GREV: get_opcode = 7'b0110011;
                SLOI, SROI, RORI, SBCLRI, SBSETI, SBINVI, SBEXTI, GORCI, GREVI, CMIX, CMOV,
```

```
                    FSL: get_opcode = 7'b0010011;
                    …
                    default: get_opcode = super.get_opcode();
            endcase
    endfunction

### This is a simple piece about generation

            class riscv_b_instr extends riscv_instr;

            rand riscv_reg_t rs3;
            bit has_rs3 = 1'b0;

            `uvm_object_utils(riscv_b_instr)

    function new(string name = "");
            super.new(name);
    endfunction

    virtual function void set_rand_mode();
            super.set_rand_mode();
            has_rs3 = 1'b0;
            case (format) inside
                    R_FORMAT: begin
                            if (instr_name inside {CLZW, CTZW, PCNTW, SEXT_B, SEXT_H, CLZ, CTZ,
                                PCNT, BMATFLIP, CRC32_B, CRC32_H, CRC32_W, CRC32C_B,
                                CRC32C_H, CRC32C_W, CRC32_D, CRC32C_D}) begin
                              has_rs2 = 1'b0;
                            end
                    end
                    R4_FORMAT: begin
                            has_imm = 1'b0;
                            has_rs3 = 1'b1;
                    end
                    I_FORMAT: begin
                            has_rs2 = 1'b0;
                            if (instr_name inside {FSRI, FSRIW}) begin
                                has_rs3 = 1'b1;
                            end
                    end
            endcase

    endfunction
```

The resulting instruction functional coverage results are shown in Fig. 6.

Since the scarce resource in the verification flows was the M*DEV simulator, the ability to use riscvOVPsim for the simple flow enabled use of the critical resource for the complex flow.

The complex step-and-compare flow was challenging to set up. Key issues encountered included

- Creation of tests with the appropriate asynchronous events to cause the device under test (DUT) to react as desired
- Development of the communication between the RTL DUT and the reference model
- Debugging of test results (as failures could be due to issues with the specific test, the RTL, the reference model or the communication between RTL and reference model)

Average Grade	Covered Grade	Goal	Weight	Uncovered Bins	Excluded Bins	Total Bins	Item	Name	Comment
84.58%	84.58% (203/240)	n/a	1	37	0	240	CoverPoint	CoverCommand.CommandsToCoverage	

Figure 6a. ISA coverage report for IMACB plus custom commands and non-supported spec (e.g. F as an illegal command).

Average Grade	Covered Grade	Goal	Weight	Uncovered Bins	Excluded Bins	Total Bins	Item	Name	Comment
100.00%	100.00% (5/5)	n/a	1	0	0	5	CoverPoint	CoverJumpBranchCommandsSeq.JBSequence	

Figure 6b. ISA coverage report for branch jump sequences: loops, backward, forward branches.

Average Grade	Covered Grade	Goal	Weight	Uncovered Bins	Excluded Bins	Total Bins	Item	Name	Comment
100.00%	100.00% (5/5)	n/a	1	0	0	5	CoverPoint	CoverMultiCycleSeq.MCSequence	

Figure 6c. ISA coverage report for multicycle sequential insertion.

Functional coverage results for this flow are shown in Fig. 7.

Average Grade	Covered Grade	Goal	Weight	Uncovered Bins	Excluded Bins	Total Bins	Item	Name	Comment
62.50%	62.50% (10/16)	n/a	1	6	0	16	CoverPoint	CoverMemReq.RequestAddress	
100.00%	100.00% (3/3)	n/a	1	0	0	3	CoverPoint	CoverMemReq.RequestOpcode	
100.00%	100.00% (16/16)	n/a	1	0	0	16	CoverPoint	CoverMemReq.RequestThreadId	
100.00%	100.00% (3/3)	n/a	1	0	0	3	CoverPoint	CoverMemReq.RequestAgent	
71.43%	71.43% (5/7)	n/a	1	2	0	7	CoverPoint	CoverMemReq.RequestSize	
100.00%	100.00% (128/128)	n/a	1	0	0	128	CoverPoint	CoverMemReq.RequestData	
100.00%	100.00% (16/16)	n/a	1	0	0	16	CoverPoint	CoverMemReq.RequestWriteByteEnable	
100.00%	100.00% (48/48)	n/a	1	0	0	48	Cross	CoverMemReq.cross__RequestThreadId__RequestAgent	

Figure 7. Functional coverage reports for the memory interface including the complete 64-bit address range, different data sizes for read, write byte enable and atomic commands towards L2.

VII. Conclusions

A key milestone on the road to RISC-V adoption, a road being taken increasingly by design teams for embedded and other SoCs, is the verification of the RISC-V processor implementation. While compliance to the RISC-V specification is being addressed by RISC-V International, the organization responsible for the RISC-V architecture, design verification is implementation specific and therefore left to the developers. Thus the need for a DV methodology that can be used for both fully compliant designs and for designs that take advantage of the openness and flexibility of the RISC-V architecture.

This paper has shown the development and implementation of a verification methodology using step-and-compare RTL simulation together with an instruction accurate reference model. Two flows were built, one for verifying the basic RISC-V instructions and one for verifying the more complex functionality. These flows complement each other, and make efficient use of the simulation resources to achieve high functional coverage for verification.

Further work in this area will focus on adding automation to the complex step-and-compare flow for building the wrapper for the reference model and for building the communications between the RTL DUT and the reference model, so that future projects with different RISC-V cores can have verification flows up and running more quickly.

Acknowledgment

The authors would like to thank their colleagues who have helped with support of the Google Instruction Stream Generator and with the Specman verification environment, and those colleagues who have helped with reference model development.

References

[1] RISC-V International GitHub repository: https://github.com/riscv/riscv-compliance
[2] riscv-dv open source Instruction Stream Generator: https://github.com/google/riscv-dv
[3] S. Davidmann et al, “Rolling the dice with random instructions is the safe bet on RISC-V verification,” DVCon 2020 Proceedings.
[4] www.OVPworld.org/riscv.
[5] M. Thompson et al, “Jump start your RISC-V project with OpenHW,” DVCon 2021 Proceedings.