Rolling the Dice with Random Instructions is the Safe Bet on RISC-V Verification

Simon Davidmann and Lee Moore
Imperas Software Ltd.
Imperas Buildings, North Weston
Thame, Oxfordshire OX9 2HA United Kingdom

Richard Ho and Tao Liu
Google LLC.
1600 Amphitheater Parkway
Mountain View, CA 94043 USA

Doug Letcher and Aimee Sutton
Metrics Technologies Inc.
100 Gloucester Street
Ottawa, Ontario K2P 0A4 Canada

***Abstract*- The traditional SoC verification approach has until now been based on the fundamental assumption of known good processor IP from the mainstream semiconductor IP providers. With Open ISAs such as RISC-V, developers can exploit a greater degree of implementation flexibility but must also assume a greater role in the verification task.**

To complement established techniques, this paper illustrates the approach of using an open source random instruction generator for RISC-V with a cloud-based environment for capacity flexibility to compare implementation RTL against a reference simulation model.

This latest framework covers the needs of specialist core designers and all SoC adopters.

I. Introduction

The verification of an open Instruction Set Architecture (ISA) is a new industry challenge brought about by the growing adoption of the RISC-V ISA [1]. In contrast to other ISAs with a single source for processor IP, the RISC-V community has more than ten (fast approaching one hundred) processor IP providers, with both open source and proprietary implementations and both commercial and free business models, plus there are SoC design teams that are developing their own RISC-V processor implementations. As an open ISA, RISC-V allows developers freedom of implementation choices, optional features, extension and the opportunity to add optimized extensions or custom instructions.

A processor verification approach needs to cover the range of requirements from basic ISA compliance tests through to functional coverage and microarchitectural details, and the orthogonal range from the ISA specification to the addition of custom instructions and other features. While a single method may be unrealistic, many of the traditional SoC verification methods and tools can be employed and used successfully. Verification methodology for an open ISA, specifically RISC-V, is the focus of this paper.

Starting with an instruction stream generator, initially developed by Google, and now available on GitHub, the starting point for this verification process is a random instruction sequence. This can be compiled and used to drive the cloud-based environment from Metrics to simulate the RTL. In addition, the same instruction sequence can be used on the reference Imperas RISC-V simulator, riscvOVPsim. The design flow includes a comparison of the two log files to identify areas of discrepancy that should be investigated further.

While the concept of an open ISA is not new, this is the first time it has been considered and adopted by broad market segments, as both large established suppliers and new startups are embracing the freedoms and flexibility offered by an open ISA. While specialist design teams that are building a custom RISC-V core will be attracted to the verification methodology discussed in this paper, the intention is to make the usability suitable for a wide range of SoC developers. Given the range of ISA options, both technical aspects and business models, every SoC development project will need to consider as mandatory verification of any core considered to be integrated into a complex SoC design.

II. Compliance Versus Design Verification

With RISC-V, as an open ISA specification, any implementation will need to be tested against the latest RISC-V compliance suite.

The objective of the compliance process is to ensure that implementations are correctly following the specifications, with the expectation that compliant devices will exhibit sufficient compatibility to leverage the emerging ecosystem for tools and software. Put more simply, compliance is confirming that the designers have understood the specifications. Since the ISA specification does not include details of microarchitecture, differences in device performance and application focus are expected and of course permitted.

Since the compliance tests use expected functionality as the basis of the test suite, this incurs an overlap with some aspects of Design Verification (DV). However, the compliance suite is not exhaustive for all functionality and is focused purely with the structural specification aspects of the ISA, i.e. compliance is a subset of DV.

A. Compliance

The RISC-V Compliance Suite is developed within the RISC-V Foundation Working Group on Compliance ("Compliance WG"), and the latest test suites are available from the RISC-V compliance GitHub repository [2].

While the Compliance Suite contains tests for all the specification options and configurations, it is possible to test a target implementation with just the subset of tests that are targeted at that particular configuration. However, there is discussion within the Compliance WG that all cases of all tests should be performed, as it is more reliable to conclude a test failure was due to an intended unsupported feature than have the risk of omitting a required test. In addition, this may be a useful way to inspect an initial processor IP delivery and confirm the features are present as expected.

While compliance does not exhaustively test all functional operations, it does confirm the correct operands and registers over a sample set of functional exercises.

B. Verification

The verification flow discussed in this paper has as its goal finding the hard corner cases related to the processor implementation that may not be easily uncovered with traditional verification methods. This is not intended to be a replacement for well-established verification methods such as the Universal Verification Methodology (UVM) [3], but should be seen as a useful and complementary extension.

Fig. 1 shows the basic verification flow. Starting with an instruction stream generator, initially developed by Google, and now available as open source, the starting point for this verification process is a random instruction sequence. This can be compiled and used to drive the cloud-based environment from Metrics to simulate the RTL. In addition, the same instruction sequence can be used on the reference Imperas RISC-V simulator, riscvOVPsim, which is available via the RISC-V compliance GitHub repository. The design flow includes a comparison of the two log files to identify areas of discrepancy that should be investigated further by the RTL development team.

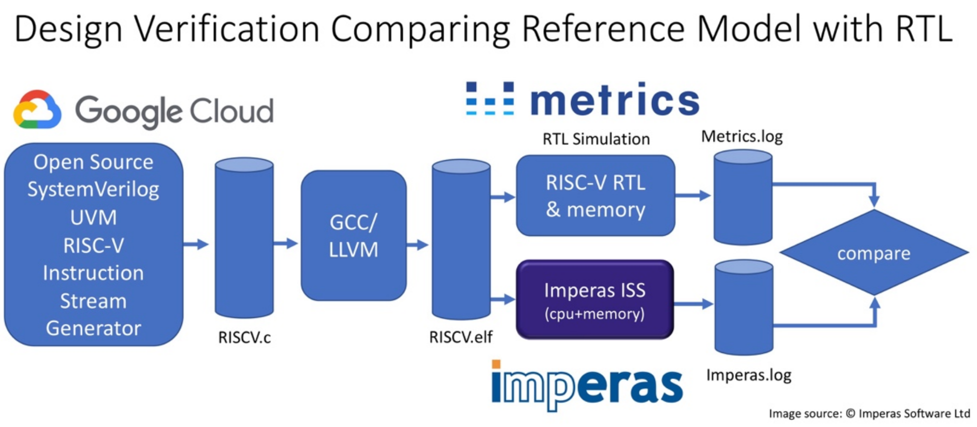

Design Verification Comparing Reference Model with RTL
Google Cloud
metrics
Open Source SystemVerilog UVM RISC-V Instruction Stream Generator
RISCV.c
GCC/ LLVM
RISCV.elf
RTL Simulation
RISC-V RTL & memory
Metrics.log
Imperas ISS (cpu+memory)
Imperas.log
compare
imperas
Image source: © Imperas Software Ltd

Fig. 1. Design verification flow comparing RTL with a reference model.

III. Components of the Verification Flow

A. *Instruction Stream Generator*

The Instruction Stream Generator (ISG) was originally developed by Google and is now available on GitHub [4]. The motivation behind this development effort was the need for verification of RISC-V processor IP (in RTL form for implementation), not only for current projects but also for reuse on future projects. The previous tools available for RISC-V processor verification had significant holes: the complete RISC-V specification was not supported, exceptions were not supported, complex branch structures were not supported and no coverage metrics were available.

The goal for ISG was to develop a tool with the following features: randomness, architecture awareness, high performance, extendibility and functional coverage.

Random instruction generation needs to occur on three levels. First, instruction level randomization, so that all possible operands and immediate values of instructions are covered. Second, sequence level randomization, so all possible instruction orders and dependencies can be tested. Third, platform level randomization, for example to test privilege modes, page table organization and program calls.

Architecture awareness means being to understand the specific architecture subsets and to configure tests for those subsets. More than that simple definition of architecture awareness, ISG needs to understand and be able to generate tests for branch prediction, MMU implementations, the pipeline including exception handling, and other key architectural features such as privilege mode.

The high-performance requirement simply means that ISG should be able to generate a large set of instructions in a short amount of time.

Extendibility is required so that additional architectural features, for example the RISC-V vector instructions, can be added to ISG. This is being addressed by making ISG an open source project so that anyone can adopt and contribute to the tool.

Functional coverage is a critical component of any constrained-random DV environment. Functional coverage enables the measurement of what has and has not been tested through random stimulus. Coverage data is a key metric in the determination of verification completeness.

B. *Cloud Based Verification*

The flow outlined in this paper is based on the assumption, based on semiconductor industry best known methods, that pre-tape out simulation is essential to ensure the quality and first pass success of a SoC. As the open ISA approach of RISC-V is changing the way SoC architects source the processor IP, it is also impacting the workloads of SoC verification teams. In the past, an SoC would be assembled from known good IP and the verification would be focused around the key interface points from the processor IP to the rest of the SoC design. In this way verification plans and test capacity are concerned with the SoC blocks and functions that are outside the processor IP. With RISC-V processors being implemented from open source RTL, or built from scratch by the design team, or adding custom extensions to a commercial IP core the verification of the RISC-V processor becomes a new and additional requirement.

The need for additional simulation capacity for processor verification is not itself a reason to move to cloud-based simulation. However, with the growing complexity of SoCs, there has been a need for more simulation capacity just for the SoC verification. This capacity requirement is also not a constant requirement, but occurs for short periods of days/weeks/months, in contrast to the annual license business model of the major RTL simulation vendors. The additional processor verification requirements only increase this need for peak capacity. Moving to cloud-based simulation addresses the need for a business model more closely aligned with the project schedule, while at the same time reducing the need for costly compute farms of even thousands of x86 boxes to try to meet that peak capacity requirement.

The Metrics Cloud Platform used in this flow provides a high performance SystemVerilog simulator in the cloud, and includes a suite of web-based verification management and debug tools that facilitate the analysis of simulation data. These include a coverage browser, waveform viewer, log viewer, and triage/failure analysis. The Metrics Cloud Simulator supports the complete SystemVerilog standard (IEEE 1800-2017) [7] as well as UVM and other base class libraries.

C. *Reference Simulator*

The reference RISC-V simulator highlighted in this paper is riscvOVPsim. The riscvOVPsim simulator implements the full and complete functionality of the RISC-V Foundation's public Unprivileged (formally known as

User) and Privilege ratified specifications. The simulator is command line configurable to enable/disable all current optional and processor specific options in the RISC-V specification. The simulator is developed, licensed and maintained by Imperas Software Ltd., and is fully compliant to the Open Virtual Platforms (OVP) [5] open standard APIs. To support the work of the RISC-V community, riscvOVPsim is available on GitHub [2] for both academic and commercial users.

As a member of the RISC-V Foundation community of software and hardware innovators collaboratively driving RISC-V adoption, Imperas originally developed the riscvOVPsim simulator to assist RISC-V adopters to become compliant to the RISC-V specifications. The latest RISC-V compliance test suite and framework (which includes riscvOVPsim) can be downloaded from the RISC-V compliance GitHub repository.

riscvOVPsim includes an industrial quality model and simulator of RISC-V processors for use for compliance and test development. It has been developed for personal, academic, or commercial use, and the model is provided as open source under the Apache 2.0 license. The simulator is provided under the Open Virtual Platforms (OVP) Fixed Platform Kits license that enables download and usage. riscvOVPsim and the OVP RISC-V processor models used by riscvOVPsim are actively maintained and enhanced. Most recently, support for the vector and bit manipulation instructions were added to the OVP RISC-V processor models.

Runtime configurable settings for all RISC-V specification options makes it very easy to compare run time results with any RTL implementations.

D. *Combined Flow*

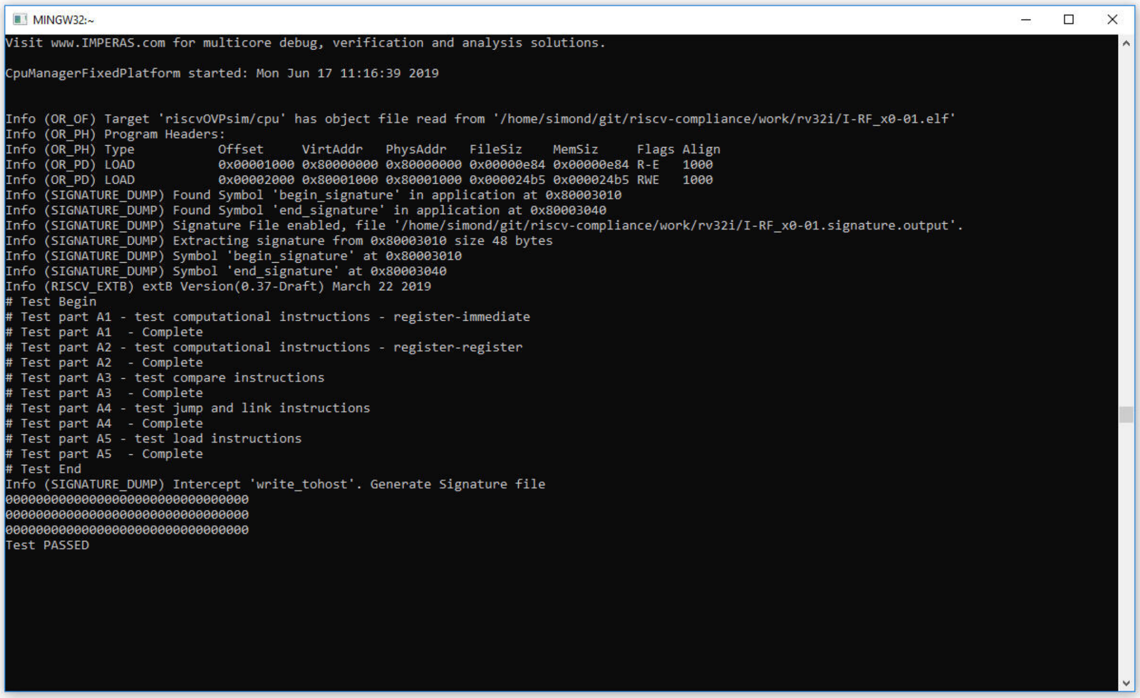
With these three components, here is the verification flow:

1. ssh to remote shell in Metrics Cloud Platform
2. Clone tests from a Git repository, e.g. compliance suite. In Fig. 2, running the RV32I compliance suite is shown.
3. Run as if in local shell

Steps 4-8 are shown in Figs. 3 and 4.

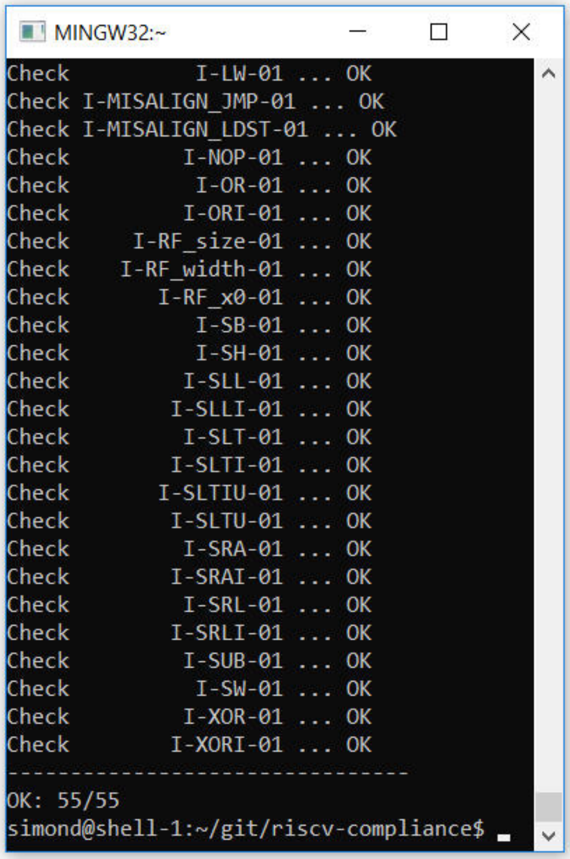
4. Run ISG to generate tests
5. Compile to .elf file for execution on the RTL (on the Metrics Cloud Platform SystemVerilog simulator) and on the reference model (riscvOVPsim).
6. Run riscvOVPsim to generate reference results
7. Run RTL simulation and generate RTL results
8. Compare run logs, report pass/fail and show mismatches

Detailed failure reporting shows the mismatching instructions, as in Fig. 4. Full traces and additional information are kept for review. Also, a VCD file can be dumped for full waveform analysis. Functional coverage can be analyzed using the Metrics Cloud Platform, as shown in Fig. 5.



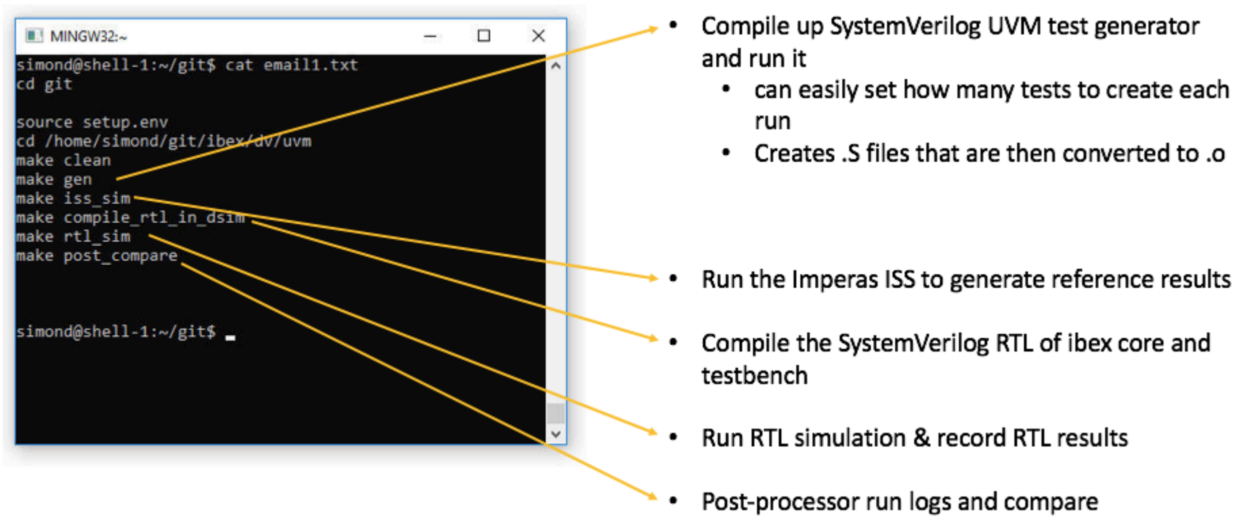
```
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

CpuManagerFixedPlatform started: Mon Jun 17 11:16:39 2019


Info (OR_OF) Target 'riscvOVPsim/cpu' has object file read from '/home/simond/git/riscv-compliance/work/rv32i/I-RF_x0-01.elf'
Info (OR_PH) Program Headers:
Info (OR_PH) Type           Offset     VirtAddr   PhysAddr   FileSiz    MemSiz     Flags Align
Info (OR_PD) LOAD           0x00001000 0x80000000 0x80000000 0x00000e84 0x00000e84 R-E   1000
Info (OR_PD) LOAD           0x00002000 0x80001000 0x80001000 0x000024b5 0x000024b5 RWE   1000
Info (SIGNATURE_DUMP) Found Symbol 'begin_signature' in application at 0x80003010
Info (SIGNATURE_DUMP) Found Symbol 'end_signature' in application at 0x80003040
Info (SIGNATURE_DUMP) Signature File enabled, file '/home/simond/git/riscv-compliance/work/rv32i/I-RF_x0-01.signature.output'.
Info (SIGNATURE_DUMP) Extracting signature from 0x80003010 size 48 bytes
Info (SIGNATURE_DUMP) Symbol 'begin_signature' at 0x80003010
Info (SIGNATURE_DUMP) Symbol 'end_signature' at 0x80003040
Info (RISCV_EXTB) extB Version(0.37-Draft) March 22 2019
# Test Begin
# Test part A1 - test computational instructions - register-immediate
# Test part A1  - Complete
# Test part A2 - test computational instructions - register-register
# Test part A2  - Complete
# Test part A3 - test compare instructions
# Test part A3  - Complete
# Test part A4 - test jump and link instructions
# Test part A4  - Complete
# Test part A5 - test load instructions
# Test part A5  - Complete
# Test End
Info (SIGNATURE_DUMP) Intercept 'write_tohost'. Generate Signature file
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
Test PASSED
```

Fig. 2a. Shell in cloud for running verification flow.



```
Check          I-LW-01 ... OK
Check I-MISALIGN_JMP-01 ... OK
Check I-MISALIGN_LDST-01 ... OK
Check         I-NOP-01 ... OK
Check          I-OR-01 ... OK
Check         I-ORI-01 ... OK
Check     I-RF_size-01 ... OK
Check    I-RF_width-01 ... OK
Check       I-RF_x0-01 ... OK
Check          I-SB-01 ... OK
Check          I-SH-01 ... OK
Check         I-SLL-01 ... OK
Check        I-SLLI-01 ... OK
Check         I-SLT-01 ... OK
Check        I-SLTI-01 ... OK
Check       I-SLTIU-01 ... OK
Check        I-SLTU-01 ... OK
Check         I-SRA-01 ... OK
Check        I-SRAI-01 ... OK
Check         I-SRL-01 ... OK
Check        I-SRLI-01 ... OK
Check         I-SUB-01 ... OK
Check          I-SW-01 ... OK
Check         I-XOR-01 ... OK
Check        I-XORI-01 ... OK
--------------------------------
OK: 55/55
simond@shell-1:~/git/riscv-compliance$
```

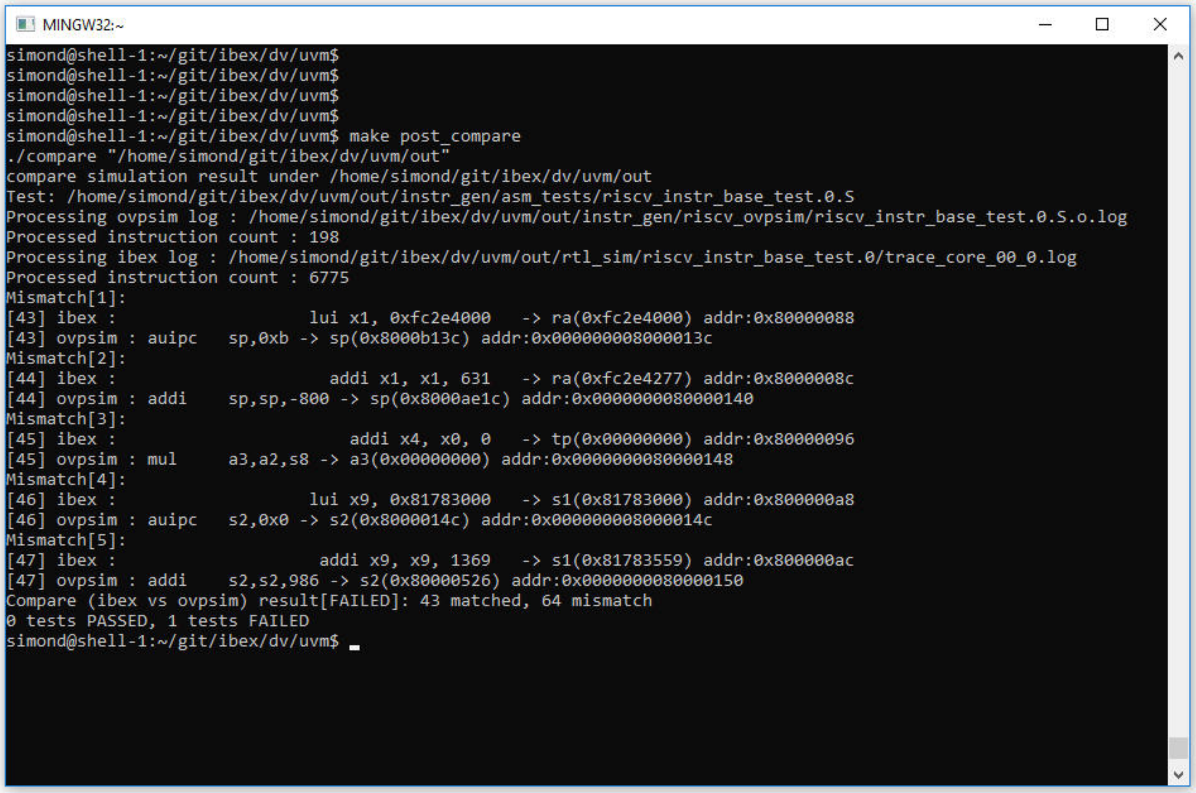
Fig. 2b. Example results for running RV32I test suite.



```
simond@shell-1:~/git$ cat email1.txt
cd git

source setup.env
cd /home/simond/git/ibex/dv/uvm
make clean
make gen
make iss_sim
make compile_rtl_in_dsim
make rtl_sim
make post_compare


simond@shell-1:~/git$ _
```

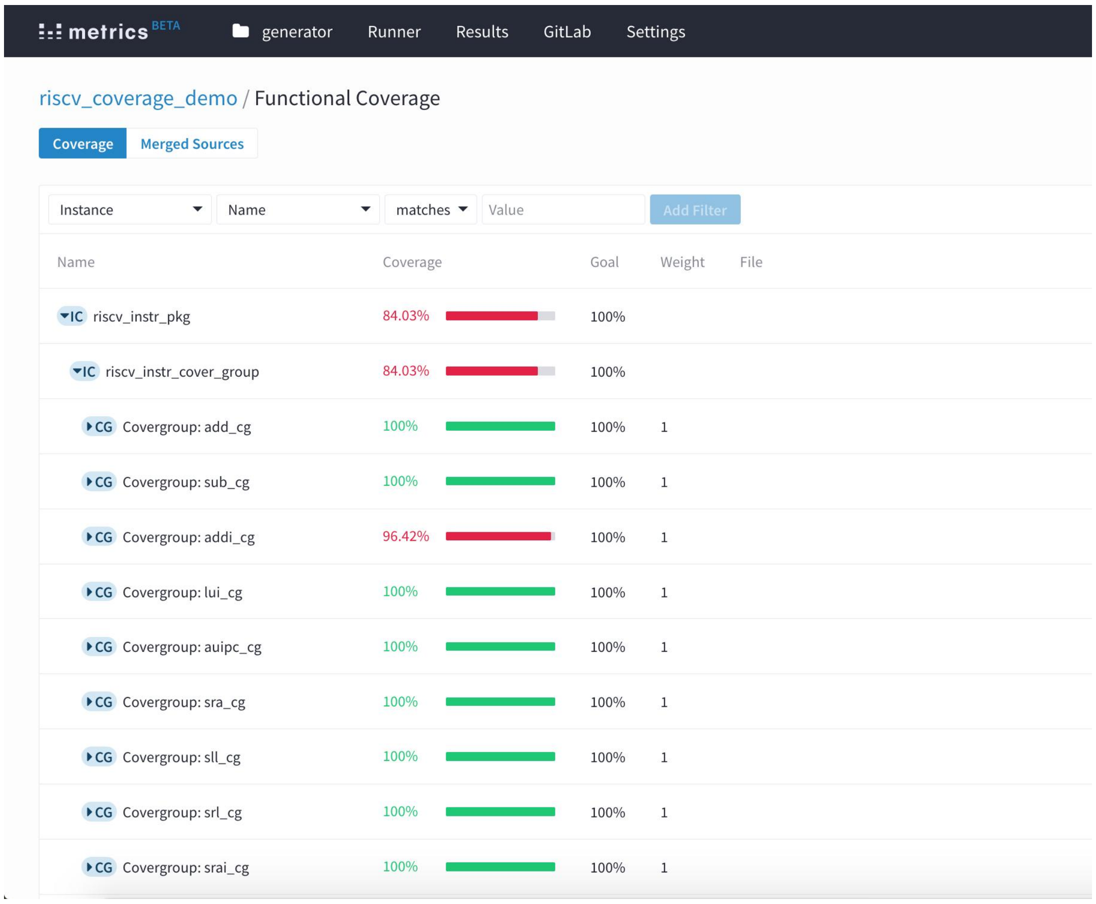
MINGW32:~
Compile up SystemVerilog UVM test generator and run it
can easily set how many tests to create each run
Creates .S files that are then converted to .o
Run the Imperas ISS to generate reference results
Compile the SystemVerilog RTL of ibex core and testbench
Run RTL simulation & record RTL results
Post-processor run logs and compare

Fig. 3. The verification flow is executed via scripts in the shell.



```
simond@shell-1:~/git/ibex/dv/uvm$
simond@shell-1:~/git/ibex/dv/uvm$
simond@shell-1:~/git/ibex/dv/uvm$
simond@shell-1:~/git/ibex/dv/uvm$
simond@shell-1:~/git/ibex/dv/uvm$ make post_compare
./compare "/home/simond/git/ibex/dv/uvm/out"
compare simulation result under /home/simond/git/ibex/dv/uvm/out
Test: /home/simond/git/ibex/dv/uvm/out/instr_gen/asm_tests/riscv_instr_base_test.0.S
Processing ovpsim log : /home/simond/git/ibex/dv/uvm/out/instr_gen/riscv_ovpsim/riscv_instr_base_test.0.S.o.log
Processed instruction count : 198
Processing ibex log : /home/simond/git/ibex/dv/uvm/out/rtl_sim/riscv_instr_base_test.0/trace_core_00_0.log
Processed instruction count : 6775
Mismatch[1]:
[43] ibex :                   lui x1, 0xfc2e4000   -> ra(0xfc2e4000) addr:0x80000088
[43] ovpsim : auipc   sp,0xb -> sp(0x8000b13c) addr:0x000000008000013c
Mismatch[2]:
[44] ibex :                     addi x1, x1, 631   -> ra(0xfc2e4277) addr:0x8000008c
[44] ovpsim : addi    sp,sp,-800 -> sp(0x8000ae1c) addr:0x0000000080000140
Mismatch[3]:
[45] ibex :                       addi x4, x0, 0   -> tp(0x00000000) addr:0x80000096
[45] ovpsim : mul     a3,a2,s8 -> a3(0x00000000) addr:0x0000000080000148
Mismatch[4]:
[46] ibex :                   lui x9, 0x81783000   -> s1(0x81783000) addr:0x800000a8
[46] ovpsim : auipc   s2,0x0 -> s2(0x8000014c) addr:0x000000008000014c
Mismatch[5]:
[47] ibex :                    addi x9, x9, 1369   -> s1(0x81783559) addr:0x800000ac
[47] ovpsim : addi    s2,s2,986 -> s2(0x80000526) addr:0x0000000080000150
Compare (ibex vs ovpsim) result[FAILED]: 43 matched, 64 mismatch
0 tests PASSED, 1 tests FAILED
simond@shell-1:~/git/ibex/dv/uvm$ _
```

Fig. 4. Pass/fail results are output from the comparison of results.



riscv_coverage_demo / Functional Coverage

Name	Coverage	Goal	Weight	File
IC riscv_instr_pkg	84.03%	100%		
IC riscv_instr_cover_group	84.03%	100%		
CG Covergroup: add_cg	100%	100%	1	
CG Covergroup: sub_cg	100%	100%	1	
CG Covergroup: addi_cg	96.42%	100%	1	
CG Covergroup: lui_cg	100%	100%	1	
CG Covergroup: auipc_cg	100%	100%	1	
CG Covergroup: sra_cg	100%	100%	1	
CG Covergroup: sll_cg	100%	100%	1	
CG Covergroup: srl_cg	100%	100%	1	
CG Covergroup: srai_cg	100%	100%	1	

Fig. 5. Functional coverage reporting from the Metrics Cloud Platform.

IV. Results

A. Results

The first core implementation that was tested using the flow outlined in this paper is the popular Ibex core that was originally developed by ETH Zurich under the name “Zero-riscy” [6], and more recently adopted by LowRISC as Ibex [7]. Ibex implements the RISC-V RV32IMC instructions, which is the 32-bit RISC-V with integer (I), multiplier/divider (M) and compressed (C) instructions. Fig. 6 shows a block diagram of the Ibex.

Fig. 7 shows the categories of bugs found using the verification flow discussed in this paper, while Fig. 8 shows examples of the specific bugs.

B. Future Work

The flow outlined in this paper can be extended to cover the future extensions and additions of the RISC-V ISA. In 2020, the RISC-V Foundation is expected to complete the ratification process of both the Bit Manipulation and Vector extensions. Work to support these extensions in the flow outlined in this paper is already underway.

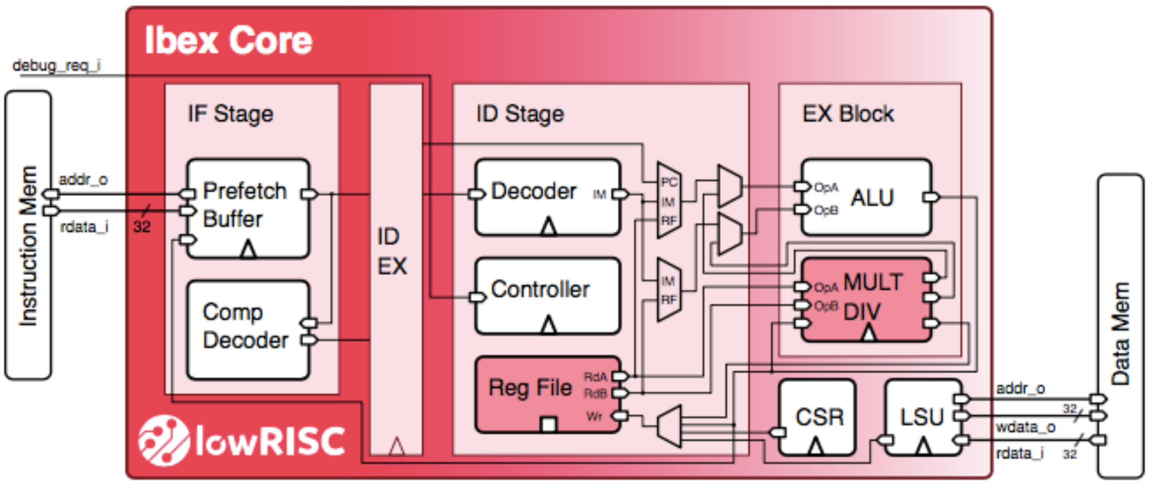


Fig. 6. Block diagram of the Ibex core.

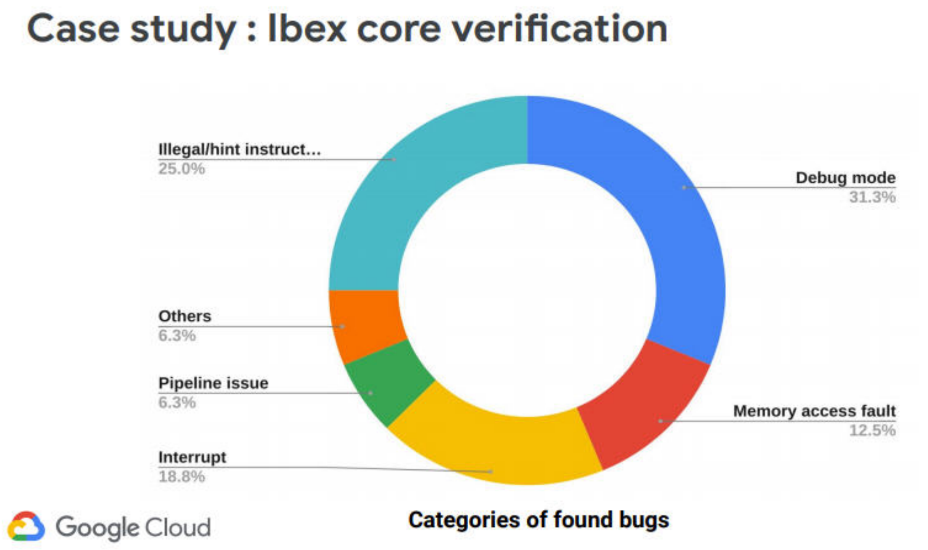


Fig. 7. Categories of bugs found in the Ibex core using the verification flow described in this paper.

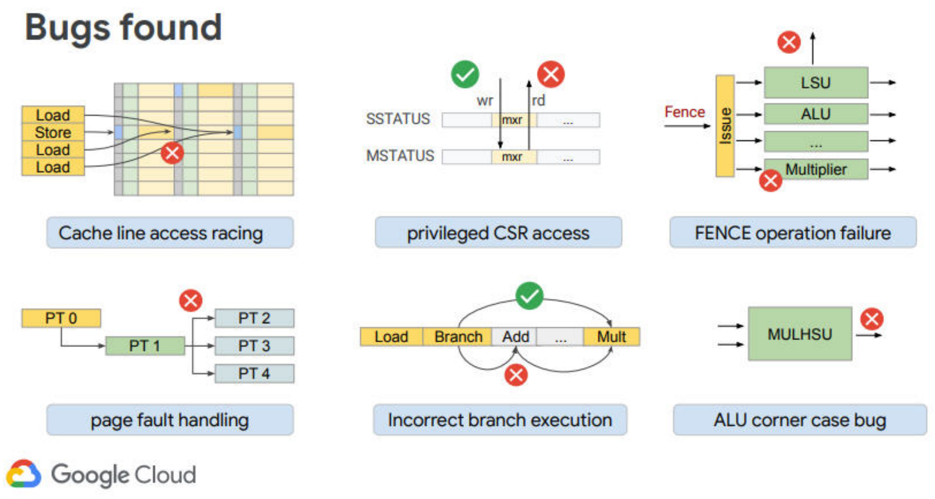


Fig. 7. Examples of the specific bugs found.

V. Conclusions

The flow outlined in this paper is intended to find the hard corner cases that may not be easily discovered with traditional verification methods. This is not intended to be a replacement for well-established verification methods but should be seen as a useful extension as SoC developers address the verification requirements associated with the open ISA of RISC-V processor cores.

The initial work was based around a popular open source RISC-V core, and the analysis results and coverage summary are a key highlight for this paper. In addition to the Random Instruction approach to verification several of the authors are also involved with the RISC-V Foundation working group on Compliance, which is a key starting point for the overall verification approach and work flow.

Finally, the results of this work highlight the improvements to coverage, flexibility and usability following the close collaboration between Imperas, Google and Metrics on this project. For additional comments on this topic please contact the authors.

Acknowledgments

The authors would like to thank all the voluntary members of the RISC-V Foundation Compliance Working Group for their efforts.

The authors would like to thank all the voluntary members of the Chips Alliance Verification Working Group for their efforts.

The authors would like to thank all the voluntary members of the OpenHW Group Verification Working Group for their efforts.

The authors wish to acknowledge the help and support from the Ibex development team at LowRISC.

References

[1] https://riscv.org/specifications/
[2] https://github.com/riscv/riscv-compliance
[3] https://accellera.org/downloads/standards/uvm
[4] https://github.com/google/riscv-dv
[5] Open Virtual Platforms (OVP) API specifications are publicly available at http://www.ovpworld.org
[6] Schiavone, Pasquale Davide et al. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications." 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS 2017)
[7] https://github.com/lowRISC/ibex