RISC-V Testing – status and current state of the art

Jon Taylor, Synopsys, jon.taylor@synopsys.com

I. Introduction

As the number of organizations developing RISC-V CPUs grows, it is useful to look at the ecosystem around testing RISC-V processors. Historically companies such as Intel and Arm have had to develop all the tools and techniques for CPU design verification (DV) themselves; verification methodology is usually seen as part of their secret sauce. Developing rigorous verification methodology for RISC-V will be critical to the wide success of the architecture. RISC-V verification tooling efforts today are fragmented across multiple projects and a mix of commercial and open source tools so it's hard to understand at a glance what can be leveraged from the ecosystem and what might have to be built. While there is value in having a well verified product, in another sense it is a hygiene factor; all successful CPUs will need high quality verification.

A thorough verification process starts from a verification plan, which describes every element of the design that needs testing. An essential complement to that plan is functional coverage to show that all the necessary elements of the design have been tested. There are numerous existing libraries of tests for RISC-V today, as well as several test generators. None of these offer a complete solution for CPU DV in isolation but it is worth reviewing what they provide as we can then consider the gaps. RISC-V offers a further complication – the ability to customize the ISA. If a designer chooses to take advantage of this flexibility, the verification methodology also needs to match.

This paper will discuss the current state of the RISC-V simulation-based testing ecosystem before identifying gaps and opportunities. There are also growing numbers of formal tools with ISA specific support for RISC-V. These include the SAIL formal model [1] as well as commercial tools from OneSpin[2] and Axiomise/Jasper[3]. Formal and simulation based verification often complement each other, but as we will see there is already enough to consider for one paper concerning simulation-based testing. Perhaps a future paper can examine the state of formal testing.

Simulation-based verification tooling requirements can be broken down into a few steps as shown in Figure 1. Each test is binary code running on the CPU. The test could originate from an existing directed test suite, from a random instruction stream generator or written by an engineer working on the core. Generally these will be written in assembly code to remove side effects from a compiler.

Part of the testing methodology is to decide how to validate the test results. This can vary from simple self-checking tests, to fully asynchronous continuous compare in conjunction with a golden reference model. The paper will consider these approaches and briefly look at what types of bugs each can find.

Finally, we need to know when to stop testing. As already mentioned, functional coverage is a critical part of this, since it can tell the tester how much of the architectural state space has been reached; this will also be cross referenced to the verification plan. Relatedly, benchmarks are often used to check the microarchitectural performance and ensure the design reaches its performance targets.

II. Test validation

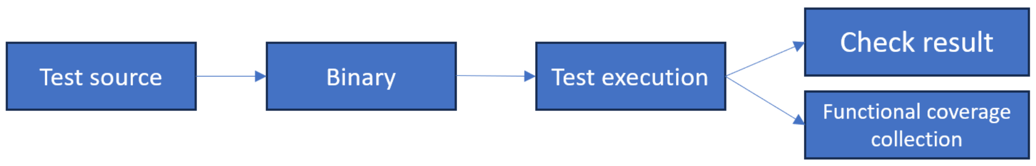


Figure 1 - typical testing flow

Before reviewing the test sources, it is useful to understand how tests can be validated as this will provide some useful context when discussing the test sources. All methods are able to support custom instructions.

There are four primary ways of validating whether a test has executed correctly. These are:

1. Signature compare
2. Trace compare
3. Synchronous lockstep compare
4. Asynchronous continuous compare

In each case, the behavior from the Device Under Test (DUT) is compared to a known-good reference. RISC-V has a number of reference sources, from open source models such as SAIL and Spike to commercial options from Synopsys. In cases 1 and 2 above, the output from the reference model can be generated offline. For 3 and 4, the test result is determined from live parallel simulations of the DUT and a golden reference model. In all cases a reference model is needed which matches the DUT – in terms of extensions, custom instructions and other architectural features. There are benefits and compromises to each approach which we will now consider:

Signature compare

For this validation method, a signature is generated. This is often as simple as dumping out a region of memory with an expected data pattern (signature) which has been generated by the test program running. The known good values are generated by the user running the same test on a reference model. While this benefits from simplicity and low overheads (eg no trace file needs to be stored), it is very limited in the types of bugs it can detect.

For example it is good for simple instructions such as ALU testing, but limited when it comes to testing features such as Control and Status Registers (CSRs), which often have implementation specific behaviors. Even for simple test cases though, it is hard to guarantee the result of the tested functionality is propagated into the signature file unless mutation testing (ie deliberately inserting bad functional behavior) is used to prove incorrect behavior would be detected.

Figure 2 shows diagrammatically how this method works

Many aspects of the RISC-V specification can be implementation defined; thus each implementation will need its own golden reference. Random testing with signature compare is challenging as it requires a mechanism to make sure the results of any intended operation are captured. This is much easier to manage with directed tests.

The approach of signature compare is the one used by the RISC-V Architectural Compatibility Test suite [4]

Trace compare

This is a logical extension of signature compare where the execution trace for a given test between DUT and a golden reference are compared, rather than just a signature. Figure 3 shows the similarity in approach.

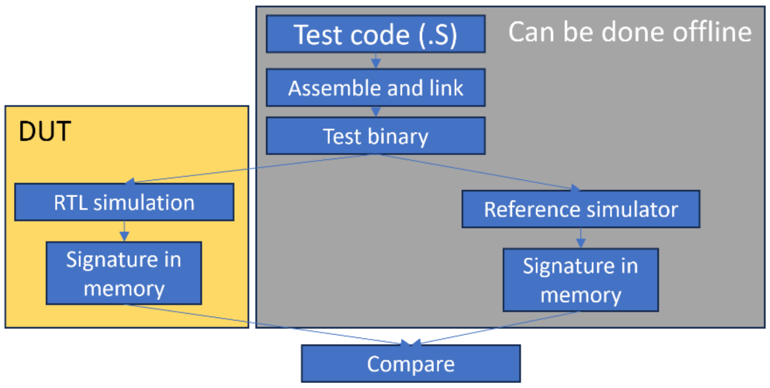


Figure 2 - signature compare flow

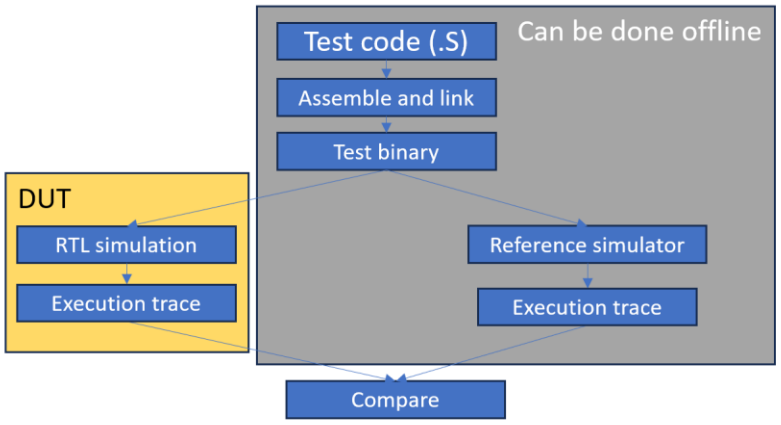


Figure 3 - Trace compare

This is more comprehensive than just signature checking as it can check every instruction, rather than just those whose effects end up in the signature region. However the trace formats between simulators will often differ, meaning post-processing is needed to perform comparisons, and it will be impractical to match traces across asynchronous events such as interrupts because the exact timing is likely to vary between DUT and simulator. Table 1 shows an example of trace files from riscvOVPsimPlus and Spike for the start of a program. It can already be seen that a comparison script may be non-trivial – note the different nomenclatures for register naming and different mnemonics for instructions (eg Spike uses c. for compressed while riscvOVPsimPlus does not).

This approach also becomes more challenging for multi-hart or Out-of-Order processors, and CSRs (particularly those with Write-Any-Real-Legal (WARL) behaviour) may prove hard to compare or have to be skipped. A further practical consideration isthat the storage space needed for the log files can become significant.

Cosimulation synchronous lock-step

Again this builds on the previous trace compare methodology by running the reference model and DUT simultaneously and comparing the state after each instruction retirement, shown in Figure 4. There are several benefits to this approach

Table 1, Trace output from different simulators for a simple code snippet

riscvOVPsimPlus	Spike
`Info 1: 'riscvOVPsim/cpu',`	`core   0: 0x0000000080000000 (0xf14022f3) csrr`
`0x0000000080000000(_start): Machine f14022f3`	`t0, mhartid`
`csrr    t0,mhartid`	`core   0: 3 0x0000000080000000 (0xf14022f3) x5`
`Info 2: 'riscvOVPsim/cpu',`	`0x0000000000000000`
`0x0000000080000004(_start+4): Machine 4301`	`core   0: 0x0000000080000004 (0x00004301) c.li`
`li      t1,0`	`t1, 0`
`Info 3: 'riscvOVPsim/cpu',`	`core   0: 3 0x0000000080000004 (0x4301) x6`
`0x0000000080000006(_start+6): Machine 00628263`	`0x0000000000000000`
`beq     t0,t1,8000000a`	`core   0: 0x0000000080000006 (0x00628263) beq`
`Info 4: 'riscvOVPsim/cpu',`	`t0, t1, pc + 4`
`0x000000008000000a(_start+a): Machine 00000417`	`core   0: 3 0x0000000080000006 (0x00628263)`
`auipc   s0,0x0`	`core   0: 0x000000008000000a (0x00000417) auipc`
`Info   s0 0000000000000000 -> 000000008000000a`	`s0, 0x0`
`Info 5: 'riscvOVPsim/cpu',`	`core   0: 3 0x000000008000000a (0x00000417) x8`
`0x000000008000000e(_start+e): Machine 00c40413`	`0x000000008000000a`
`addi    s0,s0,12`	`core   0: 0x000000008000000e (0x00c40413) addi`
`Info   s0 000000008000000a -> 0000000080000016`	`s0, s0, 12`
`Info 6: 'riscvOVPsim/cpu',`	`core   0: 3 0x000000008000000e (0x00c40413) x8`
`0x0000000080000012(_start+12): Machine 00040067`	`0x0000000080000016`
`jr      s0`	`core   0: 0x0000000080000012 (0x00040067) jr`
	`s0`
	`core   0: 3 0x0000000080000012 (0x00040067)`

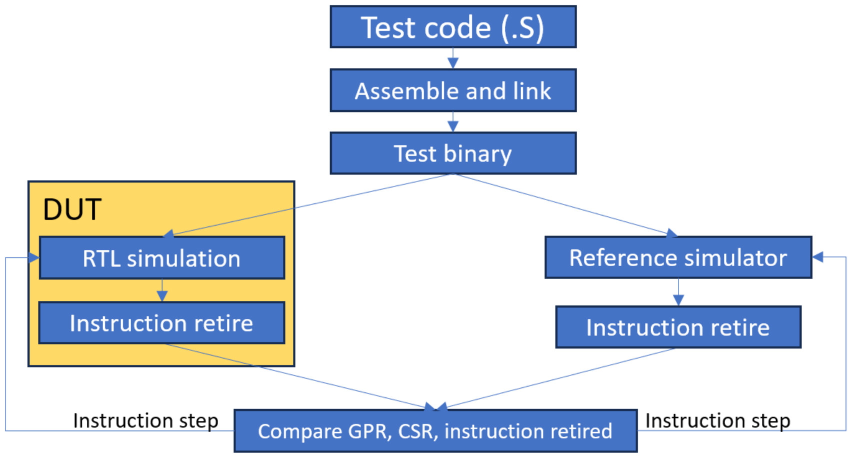


Figure 4 - Synchronous lock-step cosimulation

1. Tests stop at the point of failure, rather than running to completion before detecting a failure. This both reduces wasted runtime and can accelerate debug as it's easier to identify the point of failure
2. All CPU state can be compared on each instruction retirement, rather than just information in the trace file (ie all CSRs, GPRs etc). So any instruction side effects will be checked whether intentional or unintentional
3. There is no requirement to save log files for passing tests, reducing disk space requirements.

However there is a limitation to this approach that it cannot deal with asynchronous events such as interrupts or debug events. The synchronization is done on an instruction retirement, but the exact cycle timing of this will vary, meaning the reference design is unable to resynchronize to the instruction stream.

Imperas created the RISC-V Verification Interface (RVVI) open standard [5,6] in conjunction with OpenHW Group to create a common standard interface for the tracer signals required to implement a lock-step simulation. It is a superset of the earlier RISC-V Formal Interface (RVFI) [7] – ie a core implementing RVVI can also use the RVVI interface for formal verification as well as simulation-based verification.

Cosimulation asynchronous continuous compare

This final and most comprehensive method builds on the previous lock-step cosimulation approach and adds the ability to resynchronise the simulations around asynchronous events such as debug and interrupts. The only known solution today is from Synopsys and that includes pipeline synchronization technology which means it can track asynchronous events correctly as shown in Figure 5

While a complete output trace from such a run in ImperasDV is too verbose to include here, Figure 6 shows the

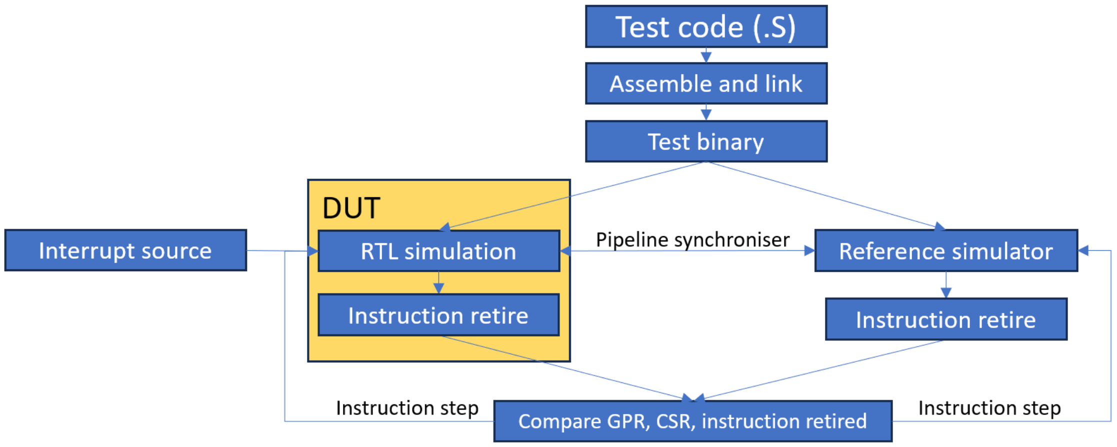


Figure 5 - Asynchronous cosimulation lockstep

```
Info 5734: 'refRoot/cpu', 0x0000000000001164(test1_impl+68): Machine 06030863 beqz    x6,11d4
Info 5735: 'refRoot/cpu', 0x0000000000001168(test1_impl+6c): Machine 06068663 beqz
x13,11d4
Info (IDV) Net change 'MSWInterrupt' = 0x0 added to queue
Info (OP_NCH) GlobalTime:0.000000 LocalTime:0.000000 Net:refRoot/MSWInterrupt 0 => 1
Info 5736: 'refRoot/cpu', 0x000000000000116c(test1_impl+70): Machine *** FETCH EXCEPTION ***
Info   mstatus 00001808 -> 00001880 [SD:0 XS:0(Off) FS:0(Off) MPP:3 VS:0(Off) MPIE:0->1
MIE:1->0]
Info   mepc 00000000 -> 0000116c
Info   mcause 00000000 -> 80000003 [Interrupt:1 Code:3(Machine software interrupt)]
Info   mip 00000000 -> 00000008 [MEIP:0 MTIP:0 MSIP:0->1]
Info 5737: 'refRoot/cpu', 0x000000000000000c(vector_table+c): Machine 5ca0006f j       5d6
Info 5738: 'refRoot/cpu', 0x00000000000005d6(m_software_irq_handler): Machine fc010113 addi
x2,x2,-64
```

Figure 6 - ImperasDV trace across interrupt

trace output across an interrupt. It can be see that the program counter (PC) changes from 0x116c to the interrupt vector at 0xc and then in turn takes the handler at 0x5d6. The reference model is kept synchronized to follow this behaviour and the simulation continues to completion.
At the end of a test a summary is presented (Figure 7). This provides information about how many comparisons (between DUT and reference model) are made during the test, including handling of asynchronous events. It can be seen that for a relatively short test (~115000 instructions) there are a very large number of comparisons (>500000)

Some CSRs cannot be meaningfully compared, for example the CPU cycle counter cannot be modelled in an instruction accurate simulator. However the reference model can be made to assume the DUT is correct and inherit the value from there, so for example code conditional on the value in that registers can still be executed in this environment. Correct functionality of these registers must be verified in a different way.

III. TEST SOURCES

While it is not for this paper to discuss the merits of random vs directed testing, both types of test source already exist for RISC-V. Examples of directed test suites include:

- Architecture compatibility tests from RISC-V international [8]
- Imperas directed test suite [9]
- Other test sources such as compiler torture suites [10]

Random instruction stream generators include:

- RISC-V DV [11]
- FORCE RISC-V [12]
- Valtrix STING [13]
- Breker Systems Trek-SoC [14]
- RISCV-CTG [15]

We will review each of the random instruction stream generators in turn below. All of the test generators include some method of checking their results, however it is also possible to use a more stringent method; for example the Architecture Compliance Tests use signature compare by default, but you could also use trace compare, or cosimulation. However the converse is not possible – i.e. an environment set up to use asynchronous lock-step compare could not use trace compare or signature compare (as no signature will be generated and tracing across asynchronous events isn’t possible).

```
# Info (IDV) ---------------------------------------------------
# Info (IDV) ImperasDV VERIFICATION REPORT
# Info (IDV)   Instruction retires   : 114,943
# Info (IDV)   Traps                 : 19
# Info (IDV)   Interrupt events      : 214
# Info (IDV)   Ending cycle count    : 220,956
# Info (IDV)                              Sets / Compares
# Info (IDV)     PC                  :  114,962 / 114,943
# Info (IDV)     Instruction         :  114,962 / 114,943
# Info (IDV)     GPR                 :   54,811 / 54,811
# Info (IDV)     CSR                 :  459,687 / 231,087
# Info (IDV)     FPR                 :        0 / 0 (disabled)
# Info (IDV)     VR                  :        0 / 0 (disabled)
# Info (IDV)
# Info (IDV)   Total compares        : 515,784
# Info (IDV)   Mismatches            : 0
# Info (IDV) ---------------------------------------------------
```

Figure 7 – ImperasDV summary output

With respect to custom instructions, generally available directed tests won't include these. The architecture compatibility tests only target instructions that are included in ratified standard extensions, not custom extensions; the same is true for the Imperas directed tests. However the random instruction stream generators have mechanisms to add custom instructions into the stream.

RISC-V DV

This project was originally developed by Google before being donated to ChipsAlliance where it is now hosted. It uses the permissive Apache license. It has a wide range of features supported, including the base ISA and standard extensions to RV32IMAFDC and RV64IMAFDC and a framework for adding custom instructions. The test framework uses trace compare to check results against a known good simulator, and includes support for spike, riscvOVPsimPlus, Whisper and Sail-riscv out of the box. It is also possible to compare the simulators to each other. As previously discussed, writing the comparator may be non-trivial and Table 1 shows this problem.

The tool is written in SystemVerilog/UVM and requires an EDA tool that supports this to run – at the time of writing these are only available in commercial EDA tools.

One of the limitations of trace compare used for RISC-V DV is that it does not check all instructions. There is a longstanding issue open against it [16] but it will report a test pass potentially giving a misleading sense of confidence. This is discussed further when we consider functional coverage and an example of the tool output shown in Figure 9. While trace compare is built into RISC-V DV, it can also be used as a standalone test generator and used with one of the cosimulation methods described elsewhere.

FORCE RISC-V

This project was originally developed by Futurewei under an Apache license before being donated to OpenHW where it is now hosted. It integrates the Handcar simulator (itself based on Spike) to test whether generated instructions are valid, and supports a wide range of features similar to RISC-V DV. The tool is written in Python. At the time of writing (October 2023), it does not support the compressed C extension, although it does have a framework for adding custom instructions.

Having a separate simulator (Handcar) creates a maintenance burden – it also has to be updated and maintained to follow changes in the RISC-V spec as well as changes in Spike. The documentation is also stale, having not been updated in over three years. There is activity on this project through OpenHW so hopefully that will be updated as work on the project continues.

Valtrix STING

This is a commercial tool developed by Valtrix. It offers a comprehensive solution for random test generation and has been widely used by commercial RISC-V IP vendors. Because it is a commercial tool, there is a limited amount of public information available, however at a high level it is similar to the other generators described here in that it

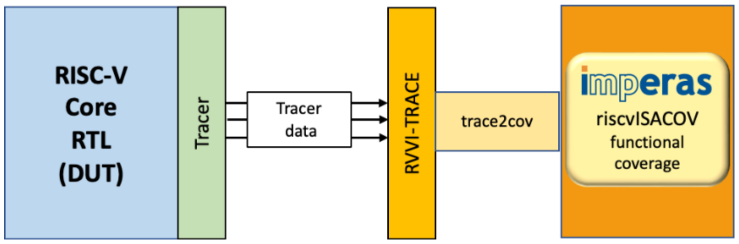


Figure 8 - functional coverage with ImperasDV

generates constrained random instruction streams which can be targeted at specific areas of the design to achieve good coverage. It supports multiple methods for checking results [17] including using a golden reference simulator model.

Breker Systems Trek-SoC

This is a commercial tool developed by Breker Systems [18]. While the primary goal of this tool is not random instruction stream generation, it can be used to target system features such as MMU and memory system with an understanding of how to find corner cases. Again, as a commercial tool there is limited public information available.

RISCV-CTG

This is an open source tool developed by InCore [15] and licensed under BSD 3-clause permissive license. It is used to generate the tests for the official RISC-V Architectural Test Suite. It relies on coverage definitions to generate the tests (see the next section for details on RISCV-ISAC which is used to drive this). It uses signature generation to create a pass/fail result. The generated tests use random solvers, and do not use a seed which means tests cannot be regenerated. Their argument is that since the tests are generated to meet coverage requirements, there is no requirement for reproducibility. While this may be true for functional coverage, it has limited utility as a general purpose test environment where reproducibility is key to debug any bugs found.

IV. Functional Coverage

A well-crafted verification plan should describe all the features to be tested, but proving it can be more difficult. Functional coverage ensures that the architectural design space is fully tested. Developing this for RISC-V may seem like a relatively simple task as the base ISA is only a few tens of instructions, but once extensions are added this quickly grows and that's without the complexity of custom instructions too.

Beyond the ISA state, there is also a need to capture functional coverage for other architectural behaviors such as CSRs, MMU, PMP, Debug etc. Some of these behaviors are also dependent on core configuration. For example, a second stage of MMU translation is only applicable when the core supports hypervisor mode.

Like the tests themselves, there are a number of sources of functional coverage.

Synopsys RISC-V ISACOV

This is System Verilog functional coverage for the ISA features of RISC-V. Functional coverage points are generated from the ISA description used by Synopsys for simulation. This makes the generation very efficient. The base ISA (RV32I) is available from github [19] under an Apache license. The remaining coverage files are available from Imperas under a proprietary license, however all of the coverpoints are published as part of the package documentation on github.[20] A snippet from the RV32I coverage documentation is show in Figure 10.

Coverage is available for all ratified RISC-V extensions, on a one-extension-per-file basis making it easy to pull in the appropriate coverage for any given configuration.

Synopsys also has a flow to generate functional coverage for custom instructions and CSRs, and is developing scenario coverage for other architectural behaviors such as MMU and Interrupts. The latter can be complex with a need to test not just the interrupt mechanism itself works, but the cross coverage of interrupts with other instructions, particularly those that are multi-cycle.

```
info: PASSED test: riscv_arithmetic_basic_test_0 instructions:11463 matches:9635
info: PASSED test: riscv_ebreak_debug_mode_test_1 instructions:29982
matches:11828
info: PASSED test: riscv_mmu_stress_test_0 instructions:17896 matches:8896 means
EXECUTED
```

Figure 9 - log output from RISCV-DV with comparison information

When used with ImperasDV and the RVVI trace interface, the coverage can be automatically included as part of the verification flow. This is shown in Figure 8

RISCV-DV coverage

Previously discussed as a test generator, RISCV-DV also includes some functional coverage. It does this as a function of the instructions executed by parsing the instruction trace log. It does not need an additional RTL tracer. As of the time of writing, it is still flagged as a work in progress. One major limitation is that the instructions analysed for coverage are a superset of those compared as part of the trace-compare verification process. Figure 9 shows the output from a small set of tests, showing that in some cases (riscv_ebreak_debug_mode_test_1), less than half of instructions are compared.

This means that some instructions may be marked as covered, even if they haven't been compared, which means the coverage data cannot be relied on as part of verification closure.

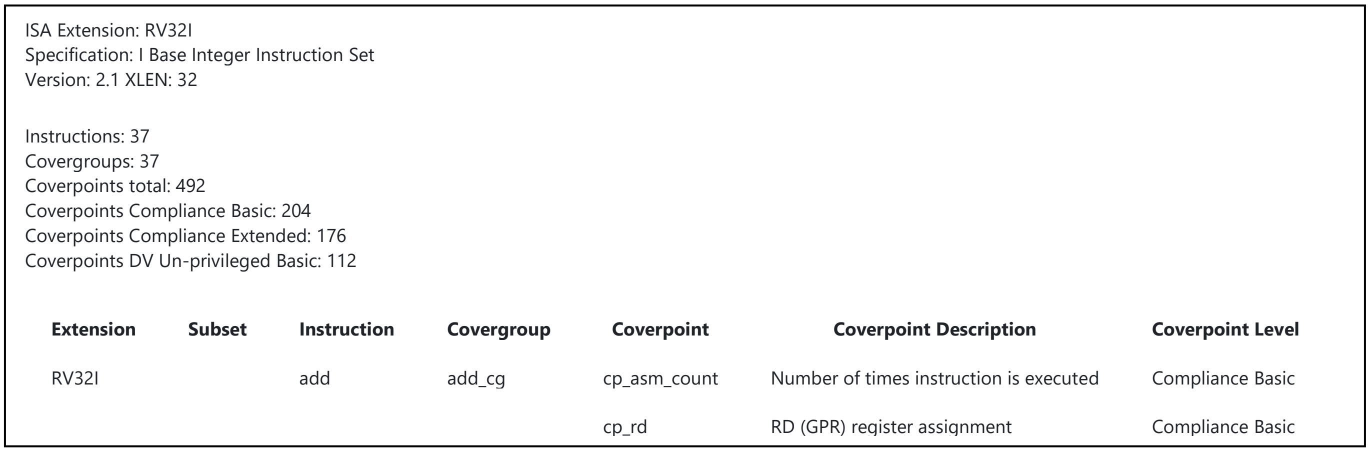
Incore RISCV-ISAC

This coverage tool is developed by InCore [21] and released under permissive BSD license. RISCV-ISAC is a tool which extracts coverage from a trace file and matches it to a set of coverage points. It is designed to be used to feed into RISCV-CTG for test generation, but this creates a circular dependency which is not desirable from a CPU DV perspective i.e. the generated tests will always hit 100% coverage. Because the coverage is manually created, it is hard to know whether that coverage is adequate to cover the design space. This might be acceptable for the use-case of compliance testing, but it does not scale well to a use-case of verification sign-off.

V. Microarchitectural Testing and Benchmarks

The tests described so far only consider the correct functional behaviour of the core, but not performance aspects. Anything more than the most basic embedded core will add features such as branch predictors, memory prefetchers, caches etc. to improve performance. However without benchmarks to measure that performance, it is impossible to know whether they are behaving correctly. For example if a branch predictor is disabled, or mis-designed such that every prediction is wrong, purely functional tests will still pass, so it is important to use benchmarks as part of testing to ensure the microarchitecture is behaving as desired. Benchmarks are designed to match the end application for types of core; those designed for embedded cores will give misleading results when run on large applications cores meanwhile some larger benchmarks may not even be able to run on small embedded cores since they depend on having an OS available.

Examples of CPU benchmarks scale from Dhrystone[22], Coremark[23], Embench[24] to Spec[25], Geekbench[26] and Antutu[27]. Analysing these in detail is beyond the scope of this paper, but below are some of the considerations when selecting benchmarks for performance validation.



ISA Extension: RV32I
Specification: I Base Integer Instruction Set
Version: 2.1 XLEN: 32

Instructions: 37
Covergroups: 37
Coverpoints total: 492
Coverpoints Compliance Basic: 204
Coverpoints Compliance Extended: 176
Coverpoints DV Un-privileged Basic: 112

Extension	Subset	Instruction	Covergroup	Coverpoint	Coverpoint Description	Coverpoint Level
RV32I		add	add_cg	cp_asm_count	Number of times instruction is executed	Compliance Basic
				cp_rd	RD (GPR) register assignment	Compliance Basic

Figure 10 - riscvISACOV functional coverage for RV32I

All cores will be trading off performance, power and area. For a given application there will be a sweet spot where the combination of design choices best matches the application needs. While there is a performance spectrum and overlap between high end embedded cores and low end applications cores, a simple way to split the naming is to look at the software running on the CPU. Embedded applications are bare metal or use an RTOS, while applications cores typically run a rich OS such as Android or Linux. The benchmarks often reflect this too, with embedded benchmarks usually being bare metal while applications benchmarks often needing an OS such as Linux. Embedded cores are typically more concerned about code size since they may be running from smaller embedded memories; many benchmarks don’t consider memory footprint, just performance.

Benchmark choice

Areas to consider when trying to find an appropriate benchmark:

- Is the real application data or compute bound?
- Does the application use floating point? Does the core have an FPU?
  - What level of precision is needed?
- Are there any special data types (eg for ML/AI) such as int8 or bfloat16?
- Is there any SIMD or vector code? Does the benchmark allow optimized libraries to be used? (eg will it allow intrinsics, autovectorization or custom maths libraries)
- How long will the benchmark take to run? (something designed for seconds or minutes of real silicon time may not be realistic to run on RTL simulation)

Ultimately, running real applications code and profiling it is the most comprehensive way to measure performance, but that may not always be practical, particularly if reliant on only RTL simulation where performance is relatively slow.

VI. Future work

So far we have looked at the existing state of the art for simulation based verification, but a few areas for improvement have been identified. Asynchronous events already stand out as being one of the trickier areas to verify, with only asynchronous lock-step cosimulation providing a good solution for testing, but there’s still no mechanism to automate coverage for these requirements. This includes the need for coverage both of the interrupt CSRs themselves, but also coverage for multicycle instructions that could potentially be interrupted at any time during their execution.

VII. Conclusions

RISC-V has created many opportunities for innovation, but within that are challenges around CPU verification that until now had been mostly solved in a proprietary manner by only one or two companies. The industry has quickly risen to the challenge, and a growing number of options are appearing in the RISC-V testing space from both the open source community and commercial vendors. However many of these projects are targeted at a single part of the overall problem and there is still no complete solution encompassing CPU DV from start to finish and combining both formal and simulation based approaches.

If we review those initial requirements – test generation, test running and validation, functional coverage collection then we can see there's a depth to those that may not be initially apparent and some methods may be missing key features or the ability to find certain classes of bug. Many aspects of RISC-V have a make vs buy consideration but the complexity of building a comprehensive verification solution should not be underestimated.

Of the projects and products considered, ImperasDV with support for asynchronous lock-step compare, coupled with complete ISA functional coverage is the most comprehensive.

As the RISC-V ecosystem matures, we can expect to see more complete solutions pulling together all the component elements needed for CPU DV.

---

[1] SAIL RISC-V formal description https://github.com/riscv/sail-riscv
[2] OneSpin RISC-V https://www.onespin.com/solutions/risc-v
[3] Axiomise/Jasper https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/axiomise
[4] RISC-V Architectural Compliance Tests documentation https://github.com/riscv-non-isa/riscv-arch-test/tree/main/doc#the-test-signature
[5] RISC-V Verification Interface: https://github.com/riscv-verification/RVVI
[6] DVCon 2023, The evolution of Processor Verification (https://www.imperas.com/sites/default/files/documents/DVCon2023_Imperas_RISC-V_Verification_Slides.pdf)
[7] RISC-V Formal Interface for Formal verification: https://github.com/SymbioticEDA/riscv-formal/blob/master/docs/rvfi.md
[8] RISC-V architectural test suite https://github.com/riscv-non-isa/riscv-arch-test
[9] RISC-V architectural test suite https://github.com/riscv-non-isa/riscv-arch-test
[10] RISC-V Berkeley compiler torture suite : https://github.com/ucb-bar/riscv-torture
[11] RISC-V DV https://github.com/chipsalliance/riscv-dv
[12] FORCE-RISCV https://github.com/openhwgroup/force-riscv
[13] Vatrix Sting https://www.valtrix.in/sting/
[14] Breker Systems TrecSoC https://brekersystems.com/
[15] RISC-V CTG https://github.com/riscv-software-src/riscv-ctg
[16] RISC-V DV doesn't check all instructions https://github.com/chipsalliance/riscv-dv/issues/919
[17] Valtrix solution PDF, downloaded 26th October 2023 https://www.valtrix.in/assets/pdf/STING_A_Complete_RISC-V_Functional_Verification_Solution.pdf
[18] Breker RISC-V solutions - https://brekersystems.com/products/risc-v/
[19] Synopsys RISC-V ISACOV https://github.com/riscv-verification/riscvISACOV
[20] Synopsys RISC-V ISACOV documentation https://github.com/riscv-verification/riscvISACOV/tree/20230420/documentation
[21] RISC-V ISAC from InCore https://github.com/riscv-software-src/riscv-isac
[22] Weicker, Reinhold (October 1984). "Dhrystone: A Synthetic Systems Programming Benchmark". *Communications of the ACM*. **27** (10): 1013–30. doi:10.1145/358274.358283
[23] Coremark - https://www.eembc.org/coremark/
[24] Embench - https://www.embench.org/
[25] SPEC - https://www.spec.org/benchmarks.html
[26] Geekbench - https://www.geekbench.com/
[27] Antutu - https://www.antutu.com/en/index.htm