# RISC-V Architectural Functional Verification

David Harris[1], Jordan Carlin[1], Corey Hickson[1], Larry Lapides[2], Lee Moore[2], Huda Sajjad[3], Umer Shahid[3], Aimee Sutton[2], Mike Thompson[4], Rose Thompson[5], Muhammad Zain[6]

[1]Harvey Mudd College [2]Synopsys [3]10x Engineers [4]OpenHW Foundation [5]Oklahoma State University [6]UET Lahore

## Abstract

*This work describes a comprehensive open functional verification suite for RVA22S64. The tests run on a Device Under Test (DUT) communicating with the ImperasDV reference model via the RISC-V Verification Interface (RVVI). The testbench collects functional coverage while the reference model checks that the DUT demonstrates correct behavior. Lockstep eliminates the burden of generating signatures and the risk of incomplete signatures. The suite contains manually-written privileged coverpoints for virtual memory, CSRs, traps, and PMP as well as automatically-generated coverpoints for all unprivileged instructions.*

## Introduction

RISC-V is an open architecture with low barriers to entry, encouraging a wealth of commercial and open implementations. However, verification is more work than design. The RISC-V ecosystem presently lacks a comprehensive open functional verification suite that can be easily reused across implementations. This work introduces such a suite for RVA22S64 and lower profiles, and for corresponding RV32 extensions.

We target *architectural functional verification*, testing that a RISC-V core implements the architecture specification in an implementation-independent fashion. This work does not attempt to address full design verification, which includes microarchitectural corner cases related to pipeline hazards, memory hierarchy, or asynchronous interrupt timing, nor does it exercise SoC features such as peripherals or shared memory consistency. We provide test plans, SystemVerilog covergroups, and assembly language tests. A key feature is that the tests are run in lockstep with a reference model configured to match the DUT, making the tests easy to write and check. The DUT and reference model communicate over an extended RISC-V Verification Interface (RVVI) [1], which also conveys the architectural state required to measure functional coverage.

## Architectural Functional Verification

The architectural functional verification suite [2] draws on the Synopsys open-source riscvISACOV [3] coverage definitions and sampling methodology, and on the RISC-V International riscv-arch-test ACT suite [4]. We support both RV32 and RV64 for all of the mandatory and many optional unprivileged and privileged extensions in the RVA22S64 profile. This includes most of the RVA23S64 features excluding vector and hypervisor.

## Unprivileged Tests

Fig. 1 shows the architectural functional verification flow. Unprivileged verification begins by authoring a testplan

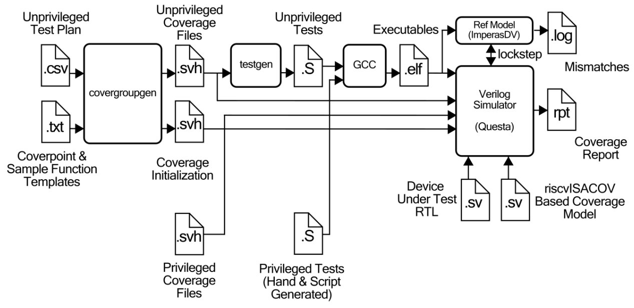
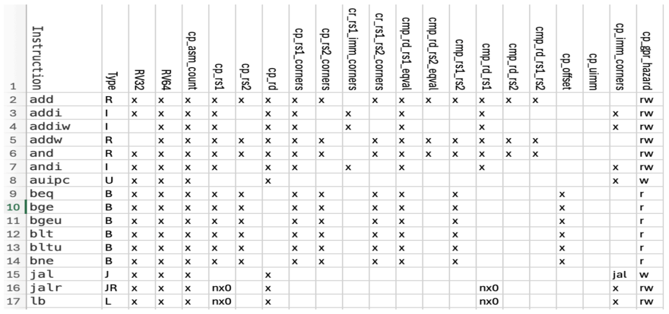


Fig. 1 Architectural functional verification flow



| 1 | Instruction | Type | RV32 | RV64 | cp_asm_count | cp_rs1 | cp_rs2 | cp_rd | cp_rs1_corners | cp_rs2_corners | cr_rs1_imm_corners | cr_rs1_rs2_corners | cmp_rd_rs1_eqval | cmp_rd_rs2_eqval | cmp_rs1_rs2 | cmp_rd_rs1 | cmp_rd_rs2 | cmp_rd_rs1_rs2 | cp_offset | cp_uimm | cp_imm_corners | cp_gpr_hazard |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | add | R | x | x | x | x | x | x | x | x |  | x | x | x | x | x | x | x |  |  |  | rw |
| 3 | addi | I | x | x | x | x |  | x | x |  | x |  | x |  |  | x |  |  |  |  | x | rw |
| 4 | addiw | I |  | x | x | x |  | x | x |  | x |  | x |  |  | x |  |  |  |  | x | rw |
| 5 | addw | R |  | x | x | x | x | x | x | x |  | x | x | x | x | x | x | x |  |  |  | rw |
| 6 | and | R | x | x | x | x | x | x | x | x |  | x | x | x | x | x | x | x |  |  |  | rw |
| 7 | andi | I | x | x | x | x |  | x | x |  | x |  | x |  |  | x |  |  |  |  | x | rw |
| 8 | auipc | U | x | x | x |  |  | x |  |  |  |  |  |  |  |  |  |  |  |  | x | w |
| 9 | beq | B | x | x | x | x | x |  | x | x |  | x | x |  | x |  |  |  | x |  |  | r |
| 10 | bge | B | x | x | x | x | x |  | x | x |  | x | x |  | x |  |  |  | x |  |  | r |
| 11 | bgeu | B | x | x | x | x | x |  | x | x |  | x | x |  | x |  |  |  | x |  |  | r |
| 12 | blt | B | x | x | x | x | x |  | x | x |  | x | x |  | x |  |  |  | x |  |  | r |
| 13 | bltu | B | x | x | x | x | x |  | x | x |  | x | x |  | x |  |  |  | x |  |  | r |
| 14 | bne | B | x | x | x | x | x |  | x | x |  | x | x |  | x |  |  |  | x |  |  | r |
| 15 | jal | J | x | x | x |  |  | x |  |  |  |  |  |  |  |  |  |  |  |  | jal | w |
| 16 | jalr | JR | x | x | x | nx0 |  | x |  |  |  |  |  |  |  | nx0 |  |  |  |  | x | rw |
| 17 | lb | L | x | x | x | nx0 |  | x |  |  |  |  |  |  |  | nx0 |  |  |  |  | x | rw |

Fig. 2 Test plan

spreadsheet; see Fig. 2 for the I extension. The spreadsheet has one row for each instruction, defining the type, whether the instruction applies to RV32 and/or RV64, and which coverpoints are applicable. Certain coverpoints have special variants, such as jalr cp_rs1 (Fig. 2, row 16), where nx0 means to exclude testing x0 because address 0 may not contain usable memory.

Next, run the covergroupgen script to parse the testplan CSV files and emit SystemVerilog functional coverage files for each extension. The files contain one covergroup for each instruction with the coverpoints indicated in the testplan, as shown in Fig. 3. For example, the add covergroup has coverpoints for each register being used as rs1, rs2, and rd, corner cases for rs1 and rs2, and the cr_rs1_rs2_corners cross-product of these corner cases. The coverage model relies on riscvISACOV classes and functions to populate a data object ("ins") with the current instruction name and architectural state (register values, etc). Architectural state is sampled from the RVVI connecting the DUT and the test bench. The covergroup uses `ifdef for coverpoints such as corners that differ for RV32 vs. RV64. It also has `ifdef for instructions that only exist for one XLEN.

```
covergroup I_add_cg with function sample(ins_i_t ins);
 option.per_instance = 0;
 cp_asm_count : coverpoint ins.ins_str == "add"  iff (ins.trap == 0 )  {
     bins count[]  = {1};
 }
 cp_rs1 : coverpoint ins.get_gpr_reg(ins.current.rs1)  iff (ins.trap == 0 )  { }
 cp_rs2 : coverpoint ins.get_gpr_reg(ins.current.rs2)  iff (ins.trap == 0 )  { }
 cp_rd : coverpoint ins.get_gpr_reg(ins.current.rd)  iff (ins.trap == 0 )    { }
 cp_rs1_corners : coverpoint (ins.current.rs1_val) iff (ins.trap == 0 )  {
     `ifdef XLEN32
       bins zero  =    {0};
       bins one  =    {32'b00000000000000000000000000000001};
       bins min  =    {32'b10000000000000000000000000000000};
    …
       bins walkeven = {32'b01010101010101010101010101010101};
    `else
       zero        = {0};
       bins one      = {64'b000000000000000000000000000…0000000000000001};
    …
```

Fig. 3 I.svh coverage file

The testgen script produces an assembly language test file for each instruction in each extension for RV32 and RV64. It creates directed random tests that systematically target each coverpoint while randomizing all aspects of the instructions not being covered. Fig. 4 shows an example of some of the cp_rs1_corners tests, with the directed values in bold. The unprivileged tests never trap.

```
# Testcase cp_rs1_corners (Test source rs1 value = 0x0)
li x17, 0x00000000 # initialize rs1
li x11, 0x535942e8 # initialize rs2
add x26, x17, x11 # perform operation

# Testcase cp_rs1_corners (Test source rs1 value = 0x1)
li x19, 0x00000001 # initialize rs1
li x3, 0x07bbf8de # initialize rs2
add x13, x19, x3 # perform operation

# Testcase cp_rs1_corners (Test source rs1 value = 0x80000000)
li x2, 0x80000000 # initialize rs1
li x24, 0x197ecbd3 # initialize rs2
add x6, x2, x24 # perform operation
…
```

Fig. 4 rv32/I/add.S test file

The tests run in lockstep with a reference model such as ImperasDV via RVVI. Therefore, there is no need for code to generate signatures or check itself, and no risk of failing to check all architectural state that changes, such as fflags.

## Privileged Tests

Privileged testing begins with a human-readable spreadsheet specifying the requirements. Most entries are manually translated into SystemVerilog coverpoints and then into assembly language tests, although some repetitive tests such as exercising all CSRs and all illegal instruction templates are automated.

The tests include a simple trap handler that resets interrupts, handles system calls to change privilege modes, and returns to the instruction after the trap. Again, lockstep simulation makes it easy to check the large amount of privileged state that might change, and avoid rotating pointers into normal and trap handler signature memories.

Virtual memory coverpoints depend on page table entries. We define an extended RVVI interface that adds addresses, page table entries, and page types to check this coverage.

## Results

Table 1 summarizes the number of coverpoints and assembly language test instructions produced by the generator scripts. Note that some coverpoints are cross-products with a large number of bins. The tests achieve 100% coverage of the unprivileged coverpoints. Privileged development is at about 50%. This coverage is independent of the device under test, so test coverage only needs to be checked at development time.

Table 1 Size of coverage files and lines of test code

| Feature | Coverpoints | RV64 Test kLOC |
|---|---|---|
| **Unprivileged** | | |
| I | 468 | 81 |
| M | 252 | 38 |
| A | 244 | 21 |
| Zc{a,b,d,f} | 233 | 14 |
| F, D, Zf{h/a} | 1332 | 2284 |
| Zb{a,b,c,s} | 672 | 80 |
| Zkn | 292 | 13 |
| Zicond | 28 | 4 |
| Zicbo* | in progress | in progress |
| **Privileged** | | |
| Zicsr | 187 | 1.6 |
| Zicntr | 39 | 1.9 |
| Exceptions | 249 | 3 |
| Interrupts | 187 | 4 |
| Endian | 130 | 1.5 |
| PMP | In Progress | N/A |
| Virtual Mem | 249 | 18 |

The tests run correctly in lockstep with ImperasDV on the OpenHW Foundation CORE-V Wally core [5, 6] for rv32gc and rv64gc (RVA22S64-compatible) configurations. ImperasDV requires configuration to match Wally behavior. Testing uncovered 9 bugs that were not detected by riscv-arch-test or other custom tests:

- fround bad shift in some situations
- fmvp untested and produces garbage
- Certain illegal instructions and CSRs did not trap

## References

[1] github.com/riscv-verification/RVVI
[2] github.com/openhwgroup/cvw-arch-verif
[3] github.com/riscv-verification/riscvISACOV
[4] github.com/riscv-non-isa/riscv-arch-test
[5] github.com/openhwgroup/cvw
[6] D. Harris, R. Thompson, J. Stine, and S. Harris, *RISC-V System-on-Chip Design*, Elsevier, 2025.