Capture The Bug 2023: RISC-V Verification Challenges

Gabriel Villanova Novaes Magalhães[1]

I. Challenge 1 Level 1 (C1L1): Logical

For this challenge, two bugs were encountered, referred to as **1) Undefined Register** and **2) Incorrect Operand**. Both issues are presented below:

A. Bug 1: Undefined Register

1) Cause: The instruction, which is shown in Listing 1 on line 15852 in "test.S," contains a bug. RISC-V provides a standard set of register names, such as `x0`, `x1`, ..., `x31`, which are 32 general-purpose registers. These registers are named: zero, ra, sp, gp, tp, t0, t1, ..., s0, s1, ..., a0, a1, etc., and are valid registers. Since `z4` does not exist, the compiler returns an error.

Listing 1: Invalid register instruction.

```
and s7, ra, z4 # line 15852 in test.S
```

2) Solution: In the test context, any of the valid registers can be used. Randomly, `z4` was replaced with the valid register `s4`, as presented in Listing 2. Note that, in a real application, the appropriate valid register should be chosen.

Listing 2: Fix Listing 1 instruction.

```
and s7, ra, s4 # line 15852 in test.S
```

B. Bug 2: Incorrect Operand

1) Cause: The instruction in "test.S" line 25581 contains an incorrect operand. The "addi" RISC-V instruction format is "addi rd, rs1, const," where:

1) `rd` is the valid destination register;
2) `rs1` is the first operand register;
3) `const` is a value of 12 bits.

This instruction is shown in Listing 3. Analyzing the instruction, it's possible to verify that the last operand `s0` is not a constant. As a result, a compiler error occurs.

Listing 3: Incorrect operand in "addi" instruction.

```
andi s5,t1,s0 # line 25581 in test.S
```

2) Solution: To fix this bug, two solutions are possible:

1) Change `s0` to a 12-bit constant.
2) Change the "addi" instruction to "add."

To maintain the pattern of "test.S," which is testing only the "add" instruction, the code was updated with the instruction with solution number 2), as presented in Listing 4. Note that the operands stayed the same, and now it is a valid instruction.

Listing 4: Fix Listing 3 instruction.

```
add s5, t1, s0 # line 25581 in test.S
```

By implementing these fixes, both bugs were resolved.

II. Challenge 1 Level 2 (C1L2): Loop

The Challenge 1 Level 2 bug is described below, along with its solution.

A. Introduction

The "test.S" file takes an input constant vector formatted using the "rule": the first word will be the value of the first add instruction, the second word will be the value of the second add operand, and the third word is the expected result. The three first lines are presented in Listing 5 as an example.

Listing 5: Test vector.

```
test_cases:
    .word 0x20 # input 1
    .word 0x20 # input 2
    .word 0x40 # sum = 0x20+0x20
    ...
    ...
```

The "test.S" was designed as follows:

1) Load the first three word values into registers;
2) Perform the "add" operation;
3) Check the performed value against the expected value.

Therefore, the number of tests will be the number of constant vectors, which shall be always a multiple of 3x, divided by 3. In this case, there are 9 memory allocations, resulting in 3 test cases. The number of tests is loaded into the t5 register by the instruction presented in Listing 6.

Listing 6: Instruction to save the num. of tests into t5.

```
li t5, 3
```

B. Cause

The main routine presented in Listing 7 is explained as follows. The first set of instructions will load the two inputs and the expected result into t1, t2, and t3, respectively. After that, the register t4 receives the sum of t1 and t2. Next, the pointer t0, which points to the allocated test cases, is incremented by 12 to get the next test case. Finally, the "beq" instruction checks if the expected value saved in t3 is equal to the calculated value in t4. If they are equal, the loop restarts to get the next test case. Otherwise, a jump to the "fail" routine is executed.

Listing 7: Main routine of C1L2 which has a loop.

```
loop:
lw t1, (t0)
lw t2, 4(t0)
lw t3, 8(t0)
add t4, t1, t2
addi t0, t0, 12

# check if the sum is correct
beq t3, t4, loop
j fail
```

The loop occurs due to this code does not have a stop condition. After the 3 tests, which do not have a fail condition, the pointer t0 continues to fetch, but it should be finalized by a "pass routine". Additionally, it was observed that after simulation cycles the Spike simulator returns the error 669. It occurs due to the undefined memory read of t0, i.e., it is not an infinite loop, but it is undesirable behavior.

```
*** FAILED *** (tohost = 669)
```

C. Solution

To fix this bug, it's necessary to implement a stop condition in the loop. One way to achieve this is by decrementing the t5 register, which holds the number of test cases, in each cycle. Additionally, it's needed to add a condition to check if t5 is zero. If t5 is zero, then the code should jump to the "test_end" - an existing function in the assembly code. Otherwise, the loop continues until the simulation fails (tohost = 669).

The solution proposed is presented in Listing 8. This code performs the desired number of test cases and checks if the test passes or fails.

Listing 8: Fix Listing 7.

```
loop:
beqz t5, test_end # new instruction
lw t1, (t0)
lw t2, 4(t0)
lw t3, 8(t0)
add t4, t1, t2
addi t0, t0, 12
addi t5, t5, -1 # new instruction
beq t3, t4, loop # check correct sum
j fail
```

D. Exercise Validation

To validate the exercise, it's possible to modify the expected value in the last word of test_cases. For example, updating the 0xcaff to 0xcafe. The output should be incorrect, and the Spike should return the error code (2) as presented as follows. Using the original test_cases vector nothing is returned by Spike indicating that the test was well succeeded.

```
*** FAILED *** (tohost = 2)
```

III. Challenge 1 Level 3 (C1L3): Infinite Loop Caused by Illegal Instruction Exception

This challenge, which consists of an infinite loop caused by an exception, is described below as well as its solution.

A. Cause

In the given code (Listing 16), an exception is intentionally created by executing an illegal instruction (code = 2). The trap is treated in the `mtvec_handler` function; it's presented in Listing 10. The issue arises from only checking the fail condition (`bne t0, t1, fail`) but never checking the pass condition. The effect of this is that the instruction `mret` will return the program counter (PC) to the last instruction, which is illegal, resulting in an infinite loop.

Listing 9: Snippet to cause an illegal instruction exception.

```
illegal_instruction:
  .word 0
  j fail
```

Listing 10: Handle to illegal instruction exception.

```
mtvec_handler:
  li t1, CAUSE_ILLEGAL_INSTRUCTION
  csrr t0, mcause
  bne t0, t1, fail
  csrr t0, mepc

  mret
```

B. Solution

To fix the bug and avoid the infinite loop, it is necessary to add the instruction `beq t0, t1, pass` before or after the `bne t0, t1, fail` instruction. This will ensure that if the pass condition is met, the program proceeds to the `pass` label, thus preventing the infinite loop. The corrected code is presented in Listing 11

Listing 11: Fix Listing 16.

```
  li t1, CAUSE_ILLEGAL_INSTRUCTION
  csrr t0, mcause
  bne t0, t1, fail
  beq t0, t1, pass # new inst. added
  csrr t0, mepc
  mret
```

With this modification, the code will properly check both the fail and pass conditions, preventing the infinite loop and resolving the bug.

IV. Challenge 2 Level 1 (C2L1): Unrecognized Opcode

This Section presents the tool Automated Assembly Program Generator as well as the bug description and solution proposed.

A. Automated Assembly Program Generator (AAPG)

The `aapg` is a versatile tool used to automatically generate assembly programs for RISC-V architectures based on specified configurations. It simplifies the process of creating complex assembly programs, especially when testing and evaluating RISC-V processors or architecture extensions. By defining various parameters and characteristics in a configuration file, users can customize the generated assembly code to suit their specific use cases.

The `aapg` supports a wide range of RISC-V extensions, allowing users to include or exclude specific instructions and features as needed. It also provides options to control the level of complexity in the generated code, such as the number of instructions, branching patterns, and data dependencies.

With the `aapg` tool, users can quickly create diverse assembly programs for testing processor functionalities, evaluating performance, and verifying hardware implementations.

B. Bug

In the context of using the `aapg` tool, a bug has been encountered, resulting in "unrecognized opcode" errors during the assembly process.

The provided config file "rv32i.yaml" is used to configure the desired extension and other characteristics of the design. This configuration file instructs the `aapg` to generate assembly code that adheres to the RV32I base integer instruction set, which includes the standard integer arithmetic, logical, and control flow instructions.

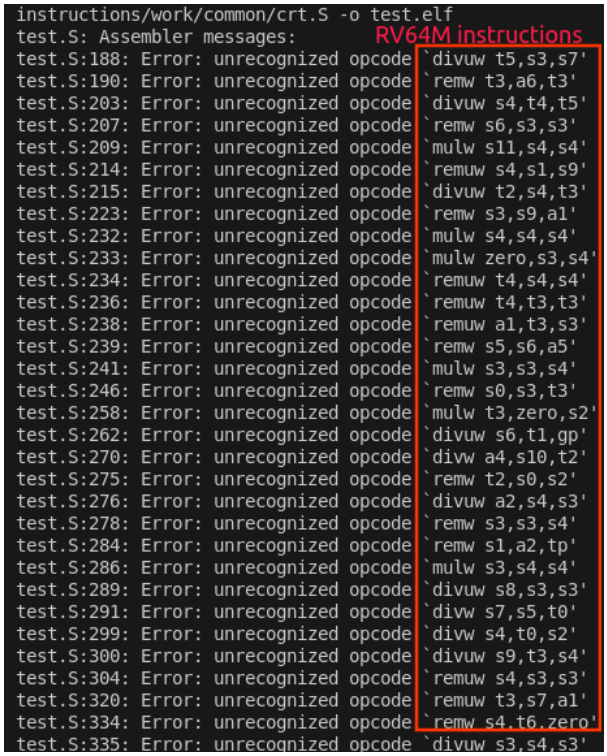
However, in the current context, the bug has occurred due to the inadvertent addition of RV64M instructions. The RV64M extension includes instructions that are not part of the RV32I base instruction set. As a result, when the `aapg` generates assembly code with these additional RV64M instructions, the assembler encounters errors of the type "Error: unrecognized opcode." This happens because the assembler cannot recognize and process the non-supported RV64M instructions while attempting to assemble the code intended for RV32I.

C. Solution

To resolve this bug, it is necessary to update (turn off) the generation of RV64M instructions. To do this, change the code according with Figure 1.

Fig. 1: Updated code for AAPG generation in C2L1 bug.

By making this modification, the `aapg` will no longer generate RV64M instructions, ensuring that the assembler doesn't encounter any "unrecognized opcode" errors as is shown in Figure .




```
instructions/work/common/crt.S -o test.elf
test.S: Assembler messages:
test.S:188: Error: unrecognized opcode `divuw t5,s3,s7'
test.S:190: Error: unrecognized opcode `remw t3,a6,t3'
test.S:203: Error: unrecognized opcode `divuw s4,t4,t5'
test.S:207: Error: unrecognized opcode `remw s6,s3,s3'
test.S:209: Error: unrecognized opcode `mulw s11,s4,s4'
test.S:214: Error: unrecognized opcode `remuw s4,s1,s9'
test.S:215: Error: unrecognized opcode `divuw t2,s4,t3'
test.S:223: Error: unrecognized opcode `remw s3,s9,a1'
test.S:232: Error: unrecognized opcode `mulw s4,s4,s4'
test.S:233: Error: unrecognized opcode `mulw zero,s3,s4'
test.S:234: Error: unrecognized opcode `remuw t4,s4,s4'
test.S:236: Error: unrecognized opcode `remuw t4,t3,t3'
test.S:238: Error: unrecognized opcode `remuw a1,t3,s3'
test.S:239: Error: unrecognized opcode `remw s5,s6,a5'
test.S:241: Error: unrecognized opcode `mulw s3,s3,s4'
test.S:246: Error: unrecognized opcode `remw s0,s3,t3'
test.S:258: Error: unrecognized opcode `mulw t3,zero,s2'
test.S:262: Error: unrecognized opcode `divuw s6,t1,gp'
test.S:270: Error: unrecognized opcode `divw a4,s10,t2'
test.S:275: Error: unrecognized opcode `remw t2,s0,s2'
test.S:276: Error: unrecognized opcode `divuw a2,s4,s3'
test.S:278: Error: unrecognized opcode `remw s3,s3,s4'
test.S:284: Error: unrecognized opcode `remw s1,a2,tp'
test.S:286: Error: unrecognized opcode `mulw s3,s4,s4'
test.S:289: Error: unrecognized opcode `divuw s8,s3,s3'
test.S:291: Error: unrecognized opcode `divw s7,s5,t0'
test.S:299: Error: unrecognized opcode `divw s4,t0,s2'
test.S:300: Error: unrecognized opcode `divuw s9,t3,s4'
test.S:304: Error: unrecognized opcode `remuw s4,s3,s3'
test.S:320: Error: unrecognized opcode `remuw t3,s7,a1'
test.S:334: Error: unrecognized opcode `remw s4,t6,zero'
test.S:335: Error: unrecognized opcode `divuw s3,s4,s3'
```

Fig. 2: Unrecognized instruction error due to RV64M generation in AAPG.

V. Challenge 2 Level 2 (C2L2): AAPG Generation Test

In this Section, Challenge 2 Level 2 (C2L2) is presented. First, a description of the challenge is explained, second, the implementation to solve is presented, and finally, a discussion about the AAP tool is given.

A. Description

The challenge is to create an AAPG (Automated Assembly Program Generator) config file that generates a test with 10 illegal exceptions, each with the correct handler code. This challenge aims to assess the ability to configure the AAPG tool to produce specific test cases with the desired number of illegal exceptions and their corresponding handlers.

B. Solution

To complete this challenge, the following steps were followed:

1) The config file used in "challenge_level2/challenge_instructions" was copied to this directory. The existing config file likely specified configurations for the RV32I base integer instruction set and other relevant parameters.
2) The line in "rv32i.yaml" was updated from:

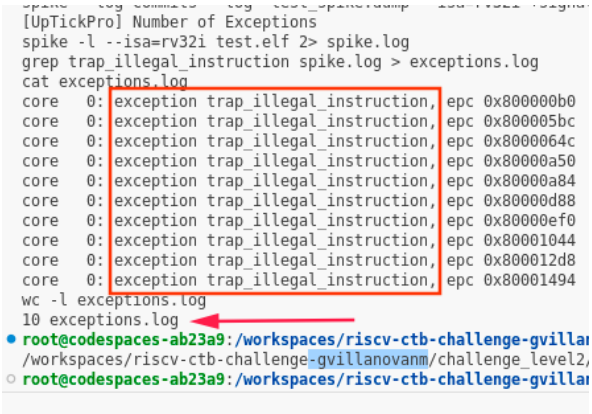
   `ecause02: 0`

   to:

   `ecause02: 9`

   This modification sets the value of `ecause02` to 9, effectively instructing the AAPG to generate 10 illegal exceptions during the test.
3) The AAPG tool was run to generate the assembly code based on the updated config file. The generated assembly code will contain the specified test cases with the desired illegal exceptions.

The reason for using "9" instead of "10" was due to the output log test, as shown in Figure 3. Using "9" resulted in 10 exceptions, in accordance with the challenge specification.



```
[UpTickPro] Number of Exceptions
spike -l --isa=rv32i test.elf 2> spike.log
grep trap_illegal_instruction spike.log > exceptions.log
cat exceptions.log
core   0: exception trap_illegal_instruction, epc 0x800000b0
core   0: exception trap_illegal_instruction, epc 0x800005bc
core   0: exception trap_illegal_instruction, epc 0x8000064c
core   0: exception trap_illegal_instruction, epc 0x80000a50
core   0: exception trap_illegal_instruction, epc 0x80000a84
core   0: exception trap_illegal_instruction, epc 0x80000d88
core   0: exception trap_illegal_instruction, epc 0x80000ef0
core   0: exception trap_illegal_instruction, epc 0x80001044
core   0: exception trap_illegal_instruction, epc 0x800012d8
core   0: exception trap_illegal_instruction, epc 0x80001494
wc -l exceptions.log
10 exceptions.log
root@codespaces-ab23a9:/workspaces/riscv-ctb-challenge-gvillar
/workspaces/riscv-ctb-challenge-gvillanovanm/challenge_level2/
root@codespaces-ab23a9:/workspaces/riscv-ctb-challenge-gvillar
```

Fig. 3: Output of the C2L2 simulation.

C. How Is Exception Generated in AAPG?

Configuring the `ecause02` fields and running AAPG will generate random assembly routines with exceptions. These routines are then incorporated into the `test.S` code, as shown in the figure below.

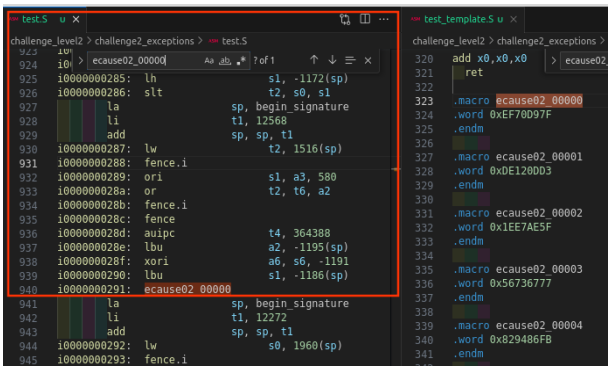


Fig. 4: Code Generation Process

Each snippet will consist of a `.word` instruction, encompassing anything that does not meet the criteria for a legal instruction in the context of the system. This will cause an exception to be invoked.

This process resembles the challenge exercise found in `challenge_level1/challenge3_illegal`, where scenarios involving illegal instructions were encountered.

D. AAPG Future Work

When executing the config file presented in the last section, the output was not always the same. Sometimes, after execution, the simulation never stopped, and other times, the simulation stopped and returned the log presented in the last section.

Attempts were made to create some modifications in "rv32i.yaml," but the same behavior persisted. It is suspected that there may be a bug in AAPG that causes this inconsistent behavior. Further investigation is needed to resolve this issue.

The inconsistent behavior observed during the AAPG execution raises concerns about the tool's reliability. It is crucial to investigate and address this issue to ensure consistent and accurate generation of assembly programs for testing and verification purposes.

VI. Challenge 3 Level 1 (C3L1): Capture The Bug - Given Design (`riscv_buggy`)

This challenge involves using the AAPG to create an infrastructure that exposes bugs in the given design. In this section, the methodology or strategy used will be presented, along with its implementation, the results obtained, and finally the conclusion.

A. C3L1: Methodology

For verifying the Design Under Test (DUT), we have chosen the Functional Verification (FV) methodology. This choice is justified by the existing infrastructure, including the AAPG generation and the comparison method based on the "diff" command between the DUT and the Reference/Golden Model (refmod or spike). Other strategies, such as UVM or Formal Verification, could also be used, but they would require more software and RTL (Register Transfer Level) access.

The fundamental steps of the Functional Verification flow are as follows:

1) Study Design;
2) Define a Verification Plan (VP);
3) Implement the test;
4) Measure, refine and validate (Results);

The process must be repeated if the VP's goals was not reached, otherwise the verification is done. This process is applied to the given design and described in detail in the next Subsections.

B. C3L1: STEP1 - Study Design

The riscv_buggy is a black-box RISC-V processor that supports RV32I (RISC-V 32-bit base integer instructions) and CSR (Control and Status Register) instructions (information gotten from Slack).

Considering this level of knowledge about the system it's appropriate to implement a verification environment to test every instruction specified (RV32I and CSR instructions). The instruction set details are presented in the Annex section [1]. In a real scenario much more should be considered.

The next step is to define a Verification Plan.

C. C3L1: STEP2 - Verification Plan (VP)

In general, the Verification Plan (VP) is a document that defines the coverage specification, the tests that need to be implemented, and the test architecture.

Based on the information about the DUT, Reference Model (spike), and the AAPG tool, which assists in test creation, the coverage can be based on the expected instructions, and the tests can be designed to leverage the AAPG possibilities. In addition, the traditional verification architecture can be adapted with the available tools. These topics are presented further in the following subsections.

1) Coverage Specification: The verification environment must measure the stimulated instructions and check if all instructions - RV32I and CSR - were covered. It's a stop criterion.

2) Test Specification: In this context, the test specification was divided into 1) sequence generation and 2) test infrastructure (stimulus, comparison, and traceability).

a) Sequences generation: The sequences should be created using AAPG tool. The AAPG allows configuring the group of instructions which is better to understand the bugs. It can be done by configuring the fields in the YAML file. These fields are presented in Listing 12.

Listing 12: AAPG Configuration.

```
total_instruction: <config>
rel_sys.csr: <config>
rel_rv32i.ctrl: <config>
rel_rv32i.compute: <config>
rel_rv32i.data: <config>
rel_rv32i.fence: <config>
```

The RV32I instructions can be considered as compute, data, fence, and ctrl. The csr is the Control and Status Register. The field ¡config¿ is a value or a weight that the AAPG uses to create the sequences randomly.

An important note is that the "total_of_instructions" field in the first regressions should be kept equal to one or two. This is made to avoid false positives. For example, if a buggy instruction was executed, and its result was saved in register x7, any operation involving x7 could potentially be a false bug. Therefore, by keeping the "total_of_instructions" small and increasing the number of tests, we improve the chances of capturing a real bug. Additionally, each test uses a new seed, which further increases the likelihood of finding a real bug. The planned tests are presented in the next paragraphs.

b) Test Architecture: The test architecture is based on the traditional verification architecture, as shown in Figure 5. It includes the following steps:

1) Stimulus generation represented as ".bin"
2) Driving the DUT and the Golden Model.
3) Comparing the outputs.
4) Extracting the results (scoreboard).

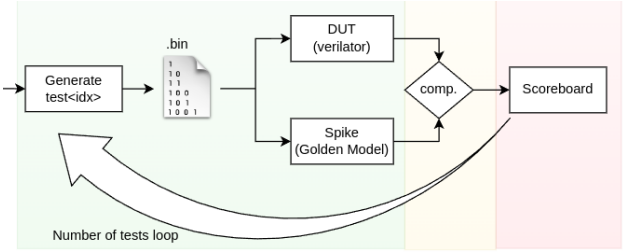


Fig. 5: Traditional verification architecture adapted to the RISC-V verification environment.

D. C3L1: STEP1 - Test Implementation

The sequences planned, and the details about the verification environment implementation are presented in this Section.

1) Sequences: The sequences were planned to separate the group of analysis due to the facilitate associate in analysis and debugging.

a) TEST_ONLY_DATA: The configuration file is presented in Listing 13. It will generate only data instructions (lw, sw, li, sh, lh, etc).

Listing 13: TEST_ONLY_DATA.

```
rel_sys.csr: 0
rel_rv32i.ctrl: 0
rel_rv32i.compute: 0
rel_rv32i.data: 1
rel_rv32i.fence: 0
```

b) TEST_ONLY_COMPUTE: The configuration file is presented in Listing 14. It will generate only compute instructions (or, xor, add, addi, sub, etc).

Listing 14: TEST_ONLY_COMPUTE.

```
rel_sys.csr: 0
rel_rv32i.ctrl: 0
rel_rv32i.compute: 1
rel_rv32i.data: 0
rel_rv32i.fence: 0
```

c) TEST_CSR_DATA_FENCE: The configuration file is presented in Listing 15. It will generate data, fence, and csr instructions. In this case, it's not possible to isolate CSR, then data and fence were set to allow the generation.s

In addition, the total instruction needs to be increased to fit the distribution specification.

Listing 15: TEST_CSR_DATA_FENCE.

```
total_instructions: 3 # needed
rel_sys.csr: 1 # focus
rel_rv32i.ctrl: 0
rel_rv32i.compute: 0
rel_rv32i.data: .5
rel_rv32i.fence: .5
```

d) TEST_CTRL_DATA: The configuration file is presented in Listing 16. It will generate Ctrl (control) and data instructions. Also, it's not possible to isolate Ctrl, then data was included in the generation.

In addition, the total instruction needs to be increased to fit the distribution specification.

Listing 16: TEST_CTRL_DATA.

```
total_instructions: 500
rel_sys.csr: 0
rel_rv32i.ctrl: 0.2 # focus
rel_rv32i.compute: 0
rel_rv32i.data: 2
rel_rv32i.fence: 0
```

2) Test Implementation: The test infrastructure was implemented in Python (file: `run_tests.py`) in three steps, as shown in Figure 6.

The routines are based on these three steps. Step 1 calls the Makefile (# make) and performs the following actions: AAPG generation, cross-compilation of the generated test.S using the RISC-V Toolchain, stimulation (spike and verilator dut), and comparison of the models' outputs using the "diff" command. Step 2 reads the generated "diff" file, verifies if it is empty or not, and increments the "match" variable with a PASS message if it is empty, or increments the "mismatch" variable and presents information about the bug (see Figure 7). Finally, Step 3 prints the scoreboard as shown in Figure 8.

Before executing this script, the test configuration (TEST_ONLY_DATA, TEST_ONLY_COMPUTE, etc.) and the number of tests (set by the variable "num_of_tests" inside `run_tests.py`) must be defined. After that, the script can be run (# python3 `run_tests.py`). The main infrastructure is based on the last challenges learned.

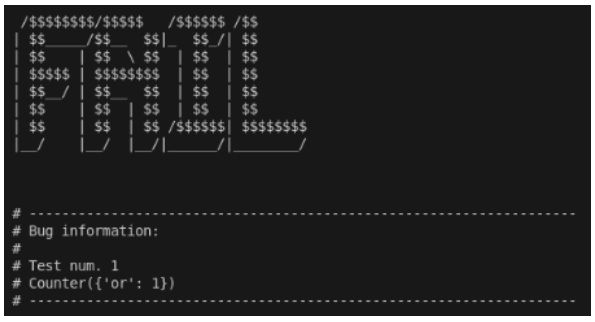


Fig. 7: Bug information presentation.

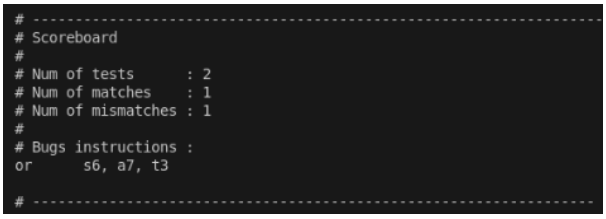


Fig. 8: Scoreboard presentation.

Another script (`run_analysis_reg.py`) was implemented for generating histograms and conducting coverage analysis. After executing regressions (see Figure 9), this script reads the generated artifacts, calculates the number of executed instructions and their percentage, plots the histogram, and calculates the percentage of instructions exercised compared to all expected instructions (Annex). In summary, it implements the coverage analysis by reading the dump files, applying disassembly, and measuring the instructions from that.

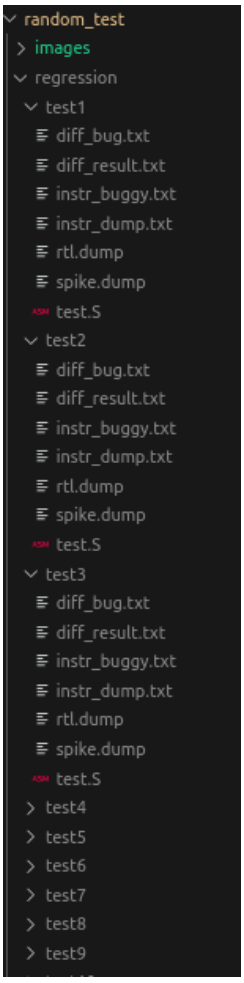


Fig. 9: Sample of regressions execution.

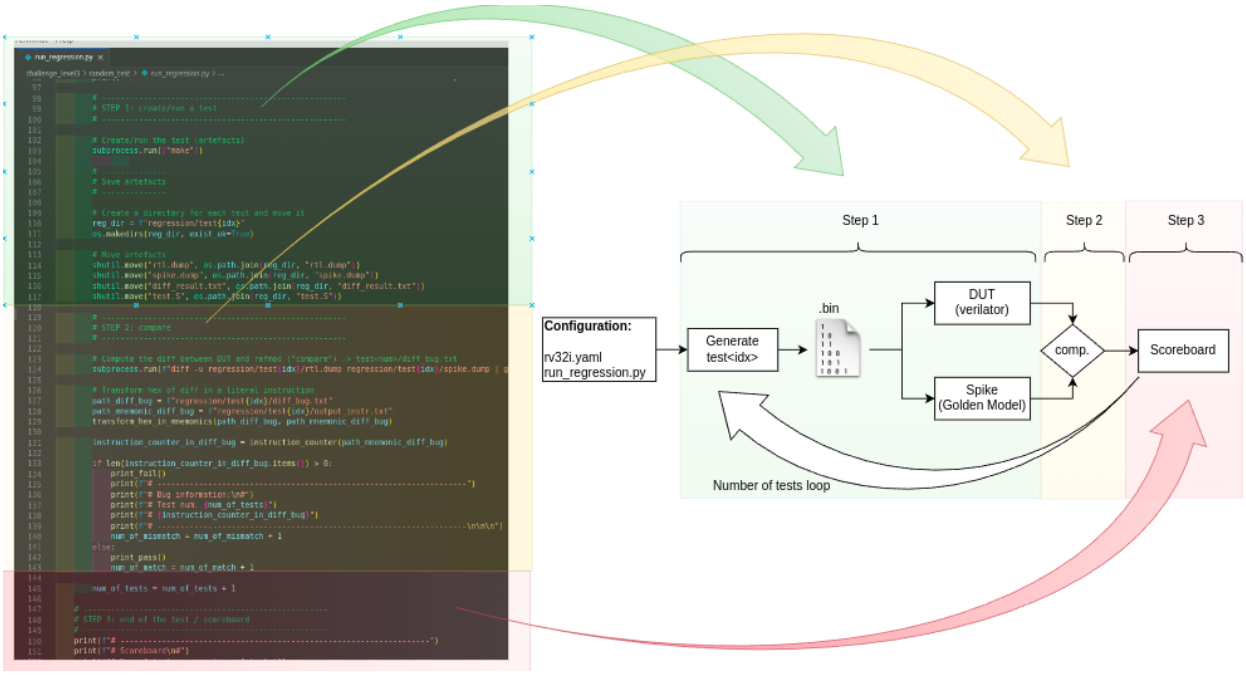


Fig. 6: Steps for the test implementation.

The results obtained from applying the defined Methodology will be presented in this section, along with the bugs found.

E. C3L1 Results: TEST_ONLY_DATA - Regression 1

1) Configuration:

- num_of_tests = 10
- total_instructions = 2

2) Result: PASS.

```
 /$$$$$$$   /$$$$$$   /$$$$$$  /$$$$$$
| $$__  $$ /$$__  $$ /$$__  $$/$$__  $$
| $$  \ $$| $$  \ $$| $$  \__/ $$  \__/
| $$$$$$$/| $$$$$$$$|  $$$$$$|  $$$$$$
| $$____/ | $$__  $$ \____  $$\____  $$
| $$      | $$  | $$ /$$  \ $$/$$  \ $$
| $$      | $$  | $$|  $$$$$$/  $$$$$$/
|__/      |__/  |__/ \______/ \______/



# ------------------------------------------------------------------
# Scoreboard
#
# Num of tests      : 10
# Num of matches    : 10
# Num of mismatches : 0
#
# Bugs instructions :
(none)
# ------------------------------------------------------------------
```

Fig. 10: Regression 1 results for TEST_ONLY_DATA.

3) Histogram of instructions stimulated: Even configuring only data, the initialization of the system uses other instructions, then this is captured in the analysis.

F. C3L1 Results: TEST_ONLY_DATA - Regression 2

1) Configuration:

- num_of_tests = 100
- total_instructions = 20

2) Result: PASS.

```
 /$$$$$$$   /$$$$$$   /$$$$$$  /$$$$$$
| $$__  $$ /$$__  $$ /$$__  $$/$$__  $$
| $$  \ $$| $$  \ $$| $$  \__/ $$  \__/
| $$$$$$$/| $$$$$$$$|  $$$$$$|  $$$$$$
| $$____/ | $$__  $$ \____  $$\____  $$
| $$      | $$  | $$ /$$  \ $$/$$  \ $$
| $$      | $$  | $$|  $$$$$$/  $$$$$$/
|__/      |__/  |__/ \______/ \______/



# ------------------------------------------------------------------
# Scoreboard
#
# Num of tests      : 100
# Num of matches    : 100
# Num of mismatches : 0
#
# Bugs instructions :
(none)
# ------------------------------------------------------------------
```

Fig. 11: Regression 2 results for TEST_ONLY_DATA.

3) Histogram of instructions stimulated: The instructions kept the same but the frequency increased compared to regression 1 (see Figure 12 and 14.

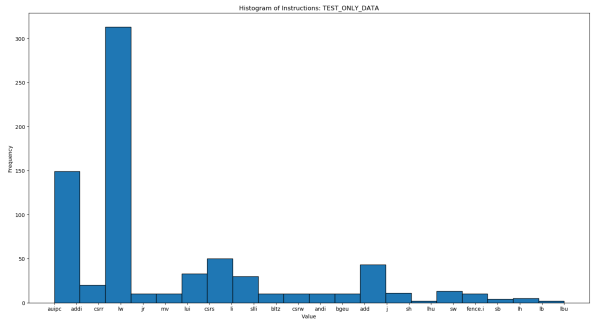


Fig. 12: Histogram of instructions stimulated for TEST_ONLY_DATA (regression 1).

4) Discussion and Coverage Status: Two regressions were executed. The first one with 10 tests and 2 instructions, but no bugs were encountered. The second regression, with 100 tests and 20 instructions, still did not find any bugs. It suggests that the bug might not be in the RV32I.data set (see the histogram).

The coverage was calculated for this case and, as expected, it did not reach 100% since it is a subset of all instructions.

Instruction Percentages:

Instruction	Percentage
add	7.31%
sub	0.00%
xor	0.00%
or	0.00%
and	0.00%
sll	0.00%
srl	0.00%
sra	0.00%
slt	0.00%
sltu	0.00%
addi	13.71%
xori	0.00%
ori	0.00%
andi	0.89%
slli	2.68%
srli	0.00%
srai	0.00%
slti	0.00%
sltiu	0.00%
lb	2.45%
lh	2.20%
lw	29.90%
lbu	2.41%
lhu	2.38%
sb	2.13%
sh	2.06%
sw	2.96%
beq	0.00%
bne	0.00%
blt	0.00%
bge	0.00%
bltu	0.00%
bltz	0.89%
bgeu	0.89%
j	0.89%
jal	0.00%
jalr	0.00%
lui	6.42%
auipc	9.99%
csrr	1.79%
csrs	0.89%
csrrw	0.00%
csrrs	0.00%
csrrc	0.00%
csrrwi	0.00%
csrrsi	0.00%
csrrci	0.00%

Average Instruction Percentage: 1.98%

Fig. 13: Coverage for TEST_ONLY_DATA.

```
Instructions Tested: 24/47
Percentage Instructions Tested: 51.06%
```

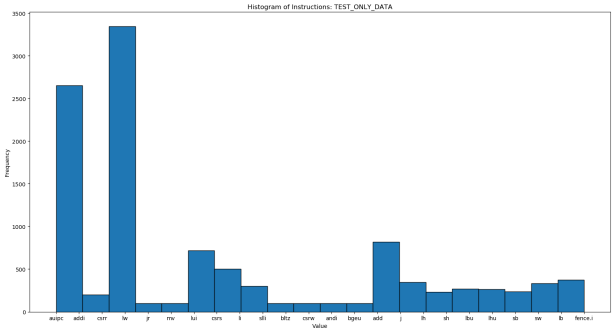


Fig. 14: Histogram of instructions stimulated for TEST_ONLY_DATA (regression 2).

G. C3L1 Results: TEST_ONLY_COMPUTE-Regression1

1) Configuration:

- num_of_tests = 10
- total_instructions = 2

2) Result: The scoreboard captured bugs in the instructions compute OR and ORI. An deep investigation is presented after regressions.

```
# ------------------------------------------------------------------
# Scoreboard
#
# Num of tests      : 10
# Num of matches    : 7
# Num of mismatches : 3
#
# Bugs instructions :
ori     t1, a3, -2034
or      t3, t1, s6
ori     s3, gp, 2007

# ------------------------------------------------------------------
```

Fig. 15: Regression 1 results for TEST_ONLY_COMPUTE.

3) Histogram of instructions stimulated: as expected the compute instructions are presented.

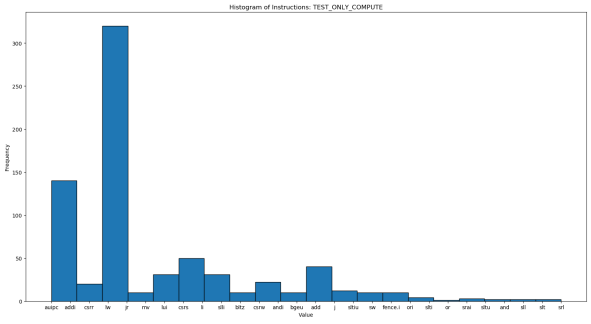


Fig. 16: Histogram of instructions stimulated for TEST_ONLY_COMPUTE.

H. C3L1 Results: TEST_ONLY_COMPUTE-Regression2

1) Configuration:

- num_of_tests = 100
- total_instructions = 2

2) Result: Scoreboard.

```
# ----------------------------------------------------------------
# Scoreboard
#
# Num of tests      : 100
# Num of matches    : 77
# Num of mismatches : 23
#
# Bugs instructions :
or      gp, s4, t5
ori     t1, a3, -2034
ori     s1, s8, 363
or      a6, s10, s0
or      a5, a5, s9
ori     t2, s5, 2039
or      t3, t1, s6
ori     t6, a4, 407
or      a3, t4, s4
ori     tp, a4, -2026
ori     s10, s1, -592
ori     s3, gp, 2007
ori     t5, t5, 1035
or      a6, t3, s1
or      t1, t5, t4
ori     a6, a1, 77
or      s1, a4, t1
ori     a1, s10, 888
or      a4, s3, s4
ori     s9, s11, -1413
ori     a1, s8, -966
or      s4, s9, s6
or      s10, s11, a6
or      t0, a7, a7

# ----------------------------------------------------------------
```

Fig. 17: Regression 2 results for TEST_ONLY_COMPUTE.

3) Histogram of instructions stimulated: as expected the compute instructions are presented.

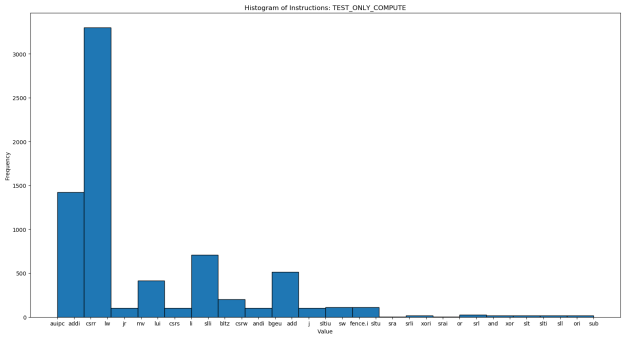


Fig. 18: Histogram of instructions stimulated for TEST_ONLY_COMPUTE.

4) Discussion and Coverage Status: The test captured a bug in the OR and ORI instruction. To understand the cause of the bug, let's consider the scenario presented below.

By analyzing the literal operation (Figure 20), it's possible to understand the real cause of the bug. When A and B are equal to 1, the output bit expected is inverted. The OR instruction also has a similar bug, and the same analysis can be done to verify the cause in detail.

ORI BUG

CASE: **ori** s11, t2, 1276 | t2 = x7 = 0xcfcd9d25 | imm = 0x4fc

	CF	CD	9D	**25**	A:	0	0	1	0	0	1	0	1
X7 = 0xcfcd9d25	11001111	11001101	10011101	**00100101**	B:	1	1	1	1	1	1	0	0
	00	00	04	**FC**	R expected	1	1	1	1	1	1	0	1
IM = 0x4fc	00000000	00000000	00000100	**11111100**	R get:	1	1	0	1	1	0	0	1
	CF	CD	**9D**	25	A:	1	0	0	1	1	1	0	1
X7 = 0xcfcd9d25	11001111	11001101	**10011101**	00100101	B:	0	0	0	0	0	1	0	0
	00	00	**04**	FC	R expected	1	0	0	1	1	1	0	1
IM = 0x4fc	00000000	00000000	**00000100**	11111100	R get:	1	0	0	1	1	0	0	1
EXPECTED:			**RESULT:**		**Bug:**								
9d	fd		99	d9	**When A and B are equal to 1, the output bit expected is inverted.**								
10011101	11111101		10011001	11011001									

Fig. 20: Analysis of the ORI bug.

Important note: Increasing the number of instructions in this test will return many bugs. However, without accurate analysis, it's hard to identify if it is a real bug or generated due to the wrong value assigned to registers after the OR or ORI operation creating a false positive (or "fake" bug). Therefore, it's safer and easier, in terms of bug capture, to keep the number of instructions equal to one or two and increase the number of tests.

5) Coverage: The coverage was calculated in regression 2, and the result is shown below. The increase in coverage is due to the set of instructions configured. The compute set (add, xor, xori, etc.) is greater than the compute data set. As a result, the coverage reached 70.21%.

```
Instructions Tested: 33/47
Percentage Instructions Tested: 70.21%
```

I. C3L1 Results: TEST_CSR_DATA_FENCE - Regr. 1

1) Configuration:

- num_of_tests = 30
- total_instructions = 3

2) Result: No bugs were encountered in this case.

J. C3L1 Results: TEST_CSR_DATA_FENCE - Regr. 2

1) Configuration:

- num_of_tests = 30
- total_instructions = 10

2) Result: The instruction **csrrci** appears in the Spike dump but not in the RTL dump due to this inconsistency the test environment returns a bug as presented in Figure 21). However, apparently the bug is related to the tool and not to the design. As a result, it will be not considered a bug in riscv_buggy.

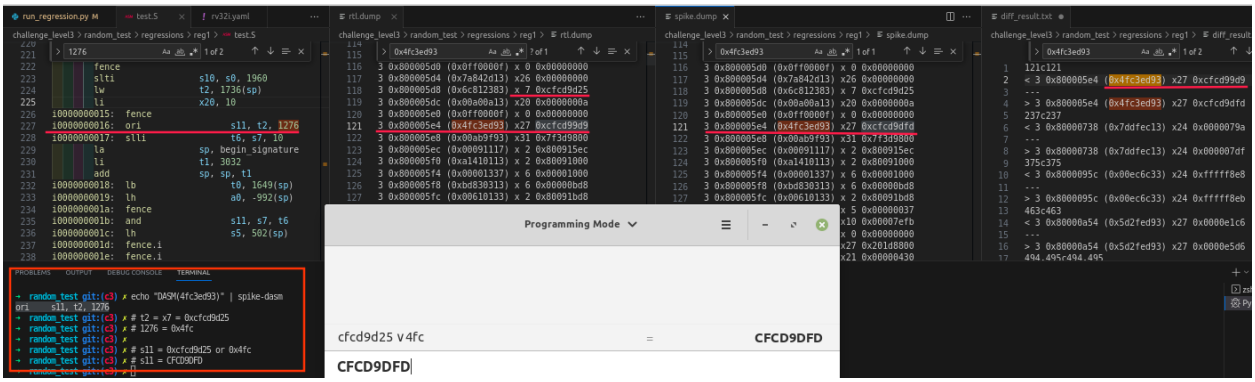


Fig. 19: Analysis of the ORI bug.

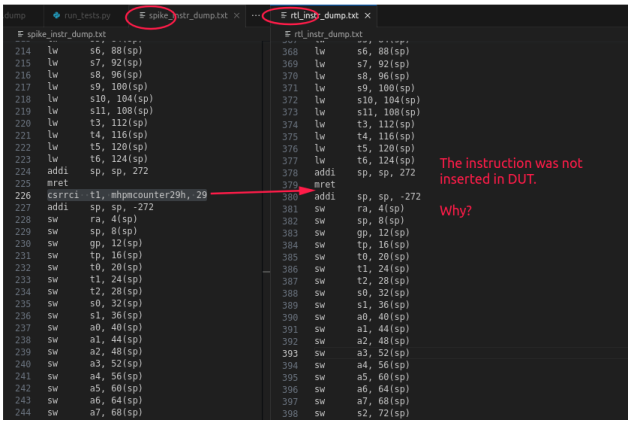


Fig. 21: Regression 2 results for TEST_CSR_DATA_FENCE.

K. C3L1 Results: TEST_CTRL_DATA

The method was executed under this configuration, and it demonstrated exceptional stability with no encountered bugs, even during exhaustive stimulation.

L. C3L1: Conclusion riscv_buggy

Based on the tested extensions and strategy, the result of riscv_buggy verification is summarized below. In other words, the tests encountered bugs in the OR and ORI instructions and a inconsistency in CSR.

```
# Extension          # RESULT/STATUS
rel_sys.csr          # INCONSISTENT
rel_rv32i.ctrl:      # NO BUGS
rel_rv32i.compute:   # BUGS: OR/ORI instr.
rel_rv32i.data:      # NO BUGS
rel_rv32i.fence:     # NO BUGS
```

Considering all the regression results, the specified coverage reaches 100%. However, it is important to note that due to the encountered bugs, merging the results might not provide a meaningful representation of the overall coverage in this case. The identified bugs should be carefully addressed and resolved before concluding the final coverage assessment.

In a real development flow, these results should be discussed with the designer responsible to address the problems. After that, the same tests should pass, and finally, a refinement in the VP also should be made to stimulate in different ways the design and achieve coverage.

VII. Challenge 3 Level 2 (C3L2): RISCV-DV Test Coverage Enhancement

A. Introduction

The objective of this activity was to improve test coverage in the RISCV-DV tool for the rv32i ISA. Despite various configurations being tested and evaluated, achieving 100% coverage was challenging due to time constraints, lack of documentation, and limited knowledge of the tool.

B. Test Generation and Interaction

RISCV-DV was used for test generation with the following command:

```
run --target rv32i --test riscv_arithm
etic_basic_test --testlist testlist.yaml
--simulator pyflow
```

To control the number of tests, the `-i` (interaction) parameter was utilized. For instance:

```
run --target rv32i --test riscv_arithm
etic_basic_test --testlist testlist.yaml
--simulator pyflow -v -i 50
```

This command generated 50 tests as observed in Figure 22.

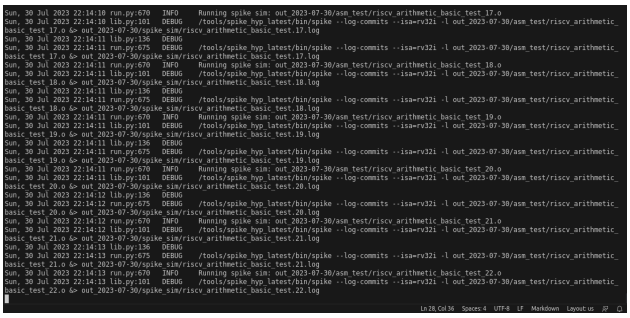


Fig. 22: Test Generation with Interaction

C. Test Analysis and Coverage

The generated tests can be analyzed as shown in Figure 24.

Coverage information can be obtained using the `cov` tool with the command below. The resulting output is depicted in Figure 23.

```
cov --dir out_*/spike_sim --enable_visua
lization  --simulator pyflow
```

To analyze the coverage metrics, it is necessary to read the `CoverageReport.txt` in the `cov_*/` folder, as demonstrated in Figure 25.

D. Conclusion and Next Steps

While various configurations were tested and evaluated, achieving 100% coverage was not realized due to the limited availability of documentation, time constraints, and unfamiliarity with the tool.

For future improvements, it is essential to explore the tool's documentation and seek assistance from the community to address the coverage issues efficiently. Additionally, further analysis of uncovered areas in the ISA and targeted test generation could be performed to enhance coverage results.

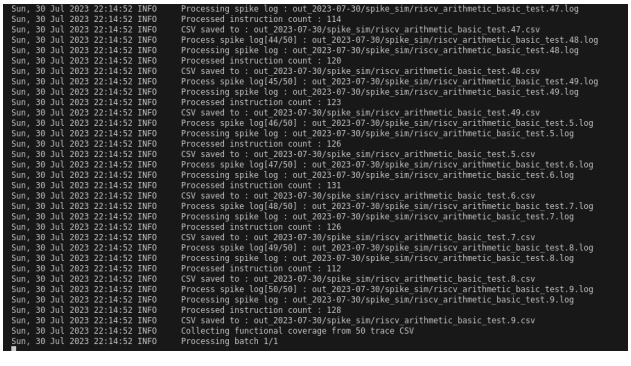


Fig. 23: Coverage Report

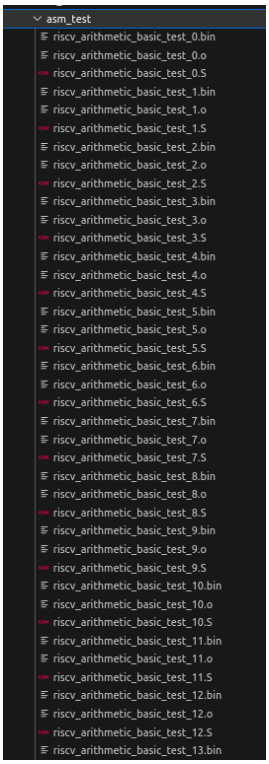


Fig. 24: Test Analysis

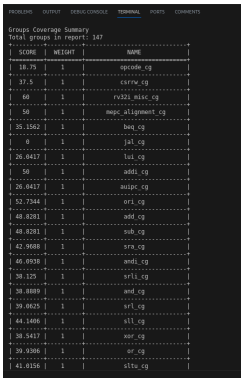


Fig. 25: Coverage Metrics

REFERENCES

[1] Andrew Waterman, Krste Asanović. (2019, December 13). *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. SiFive Inc., CS Division, EECS Department, University of California, Berkeley. Retrieved from `https://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf`

VIII. ANNEX

A. ISC-V RV32I Base Integer Instructions Card (to be reviewed)

Instruction	Type	Opcode	funct3	funct7	Format	Description
ADD	R-Type	0110011	0x0	0x00	rd = rs1 + rs2	Add
SUB	R-Type	0110011	0x0	0x20	rd = rs1 - rs2	Subtract
XOR	R-Type	0110011	0x4	0x00	rd = rs1 ĥs2	XOR (Bitwise Exclusive OR)
OR	R-Type	0110011	0x6	0x00	rd = rs1 || rs2	OR (Bitwise OR)
AND	R-Type	0110011	0x7	0x00	rd = rs1 & rs2	AND (Bitwise AND)
SLL	R-Type	0110011	0x1	0x00	rd = rs1 ¡¡ rs2	Shift Left Logical
SRL	R-Type	0110011	0x5	0x00	rd = rs1 ¿¿ rs2	Shift Right Logical
SRA	R-Type	0110011	0x5	0x20	rd = rs1 ¿¿ rs2 (msb-extended)	Shift Right Arithmetic
SLT	R-Type	0110011	0x2	0x00	rd = (rs1 ¡ rs2) ? 1 : 0	Set Less Than
SLTU	R-Type	0110011	0x3	0x00	rd = (rs1 ¡ rs2) ? 1 : 0 (zero-ext.)	Set Less Than (Unsigned)
ADDI	I-Type	0010011	0x0	-	rd = rs1 + imm	Add Immediate
XORI	I-Type	0010011	0x4	-	rd = rs1 îmm	XOR Immediate
ORI	I-Type	0010011	0x6	-	rd = rs1 || imm	OR Immediate
ANDI	I-Type	0010011	0x7	-	rd = rs1 & imm	AND Immediate
SLLI	I-Type	0010011	0x1	0x00	rd = rs1 ¡¡ imm[0:4]	Shift Left Logical Immediate
SRLI	I-Type	0010011	0x5	0x00	rd = rs1 ¿¿ imm[0:4]	Shift Right Logical Immediate
SRAI	I-Type	0010011	0x5	0x20	rd = rs1 ¿¿ imm[0:4] (msb-extended)	Shift Right Arithmetic Immediate
SLTI	I-Type	0010011	0x2	-	rd = (rs1 ¡ imm) ? 1 : 0	Set Less Than Immediate
SLTIU	I-Type	0010011	0x3	-	rd = (rs1 ¡ imm) ? 1 : 0 (zero-ext.)	Set Less Than Immediate (Unsigned)
LB	I-Type	0000011	0x0	-	rd = M[rs1 + imm][0:7]	Load Byte
LH	I-Type	0000011	0x1	-	rd = M[rs1 + imm][0:15]	Load Half
LW	I-Type	0000011	0x2	-	rd = M[rs1 + imm][0:31]	Load Word
LBU	I-Type	0000011	0x4	-	rd = M[rs1 + imm][0:7] (zero-ext.)	Load Byte (Unsigned)
LHU	I-Type	0000011	0x5	-	rd = M[rs1 + imm][0:15] (zero-ext.)	Load Half (Unsigned)
SB	S-Type	0100011	0x0	-	M[rs1 + imm][0:7] = rs2[0:7]	Store Byte
SH	S-Type	0100011	0x1	-	M[rs1 + imm][0:15] = rs2[0:15]	Store Half
SW	S-Type	0100011	0x2	-	M[rs1 + imm][0:31] = rs2[0:31]	Store Word
BEQ	B-Type	1100011	0x0	-	if (rs1 == rs2) PC += imm	Branch if Equal
BNE	B-Type	1100011	0x1	-	if (rs1 != rs2) PC += imm	Branch if Not Equal
BLT	B-Type	1100011	0x4	-	if (rs1 ¡ rs2) PC += imm	Branch if Less Than
BGE	B-Type	1100011	0x5	-	if (rs1 ¿= rs2) PC += imm	Branch if Greater Than or Equal
BLTU	B-Type	1100011	0x6	-	if (rs1 ¡ rs2) PC += imm (unsigned)	Branch if Less Than
BGEU	B-Type	1100011	0x7	-	if (rs1 ¿= rs2) PC += imm (unsigned)	Branch if Greater Than or Equal
JAL	J-Type	1101111	-	-	rd = PC + 4; PC += imm	Jump and Link
JALR	I-Type	1100111	0x0	-	rd = PC + 4; PC = rs1 + imm	Jump and Link Register . . . .

TABLE I: RISC-V RV32I Base Integer Instructions Card.