# RV32I Compliance Tests

Package Documentation

## RISC-V ISA 2.2 RV32I Compliance Tests 1.0.A (Draft)

## COMPATIBILITY:

The RISC-V Instruction Set Manual, version: 2.0
Spike simulator, version: HEAD
Test Virtual Machine, version: HEAD
GCC, version: 7.1.1 20170509
GDB, version: 8.0.50.20170808

Authors: *Radek Hajek, Milan Nostersky, Marcela Zachariasova,*

Release Date: December 24, 2017

| *Revision* | *Modification* | *Author* | *Date* |
|---|---|---|---|
| 1.0 | First version of the document. | hajek@codasip.com<br>nostersky@codasip.com<br>zachariasova@codasip.com | 24/12/2017 |

# Table of Contents


**1. PREFACE** ..... 3

1.1 *About* ..... 3

1.2 *Intended Audience* ..... 3

*1.3 References* ..... 3

*1.4 Feedback* ..... 3

**2. INTRODUCTION** ..... 3

*2.1 Content of the Package* ..... 3

2.2 *Purpose of Compliance Tests* ..... 4

**3. COMPLIANCE TESTS** ..... 4

*3.1 Binary Coding Tests and Coverage* ..... 4

*3.2 ISA Tests and Coverage* ..... 6

**4. REFERENCES** ..... 20

*4.1 Available Reference Signatures* ..... 20

4.2 *Possibility of Future References* ..... 20

**5. TEST ENVIRONMENT** ..... 20

*5.1 Test Virtual Machines* ..... 20

*5.2 User Test Environment* ..... 20

**6. APPENDIX A** ..... 21

**7. APPENDIX B** ..... 22

# 1. PREFACE

## 1.1 About

This document describes RV32I compliance tests and their reference signatures, explains how to run tests in Test Virtual Machine (TVM) environment available publicly at the RISC-V Foundation GIT repository and contains recommendations how to run tests in other user test environments.

## 1.2 Intended Audience

This document is intended for design and verification engineers who wish to check if their implementation (simulation models, HDL models, etc.) of RISC-V processor is compliant to the RV32I specification.

## 1.3 References

[1] RISC-V Foundation, May 2017. The RISC-V Instruction Set Manual Volume I: User-Level ISA, document Version 2.2, url: https://riscv.org/specifications/.

## 1.4 Feedback

If you have comments on this document then please send an email to *zachariasova@codasip.com*. Give:

- The document title,
- The chapter number, page numbers and version to which your comments apply,
- A concise explanation of your comments.

# 2. INTRODUCTION

## 2.1 Content of the Package

The package contains compliance tests, reference signatures and documentation in the following hierarchy:

```
compliance-tests
|-- rv32i
  |-- ISA              // tests dedicated to instructions behavior
    |-- references     // reference results for ISA tests
    |-- src            // assembler tests
    |-- Makefile
    `-- Makefrag       // list of tests

  `-- binary coding    // tests dedicated to binary coding
    |-- references     // reference results for BC tests

    |-- src            // assembler tests
    |-- Makefile
    `-- Makefrag       // list of tests
|-- riscv-test-env     // TVM available at Foundation gitlab
  |-- p
  `-- LICENCE
|-- documentation      // this document
```

```
  `-- RV32I_Compliance_Tests.docx
`-- run_test.sh       // main running script
```

## 2.2 Purpose of Compliance Tests

The goal of compliance tests is to check whether the processor under development meets the open RISC-V standards or not. It is considered as a non-functional testing meaning that it doesn't substitute verification. In other words this can be interpreted as a request to check all important aspects of the specification but without focusing on details, for example, on all possible values of instruction operands or all combinations of possible registers.

The result that compliance tests give to the user is an assurance that he/she interpreted the specification correctly and the design under test (DUT) can be declared as RISC-V compliant. Version 1.0 of RV32I compliance tests has following limitations:

- Most of the instruction aliases are not covered and have to be added to the tests (avoid some mismatches caused by linker optimizations).
- FENCE instruction is partially covered only in the binary coding test (fence alias) and has to be extended. Moreover, ISA test should be added later, but Section 2.7 of [1] is currently under revision).
- FENCE.I instruction is covered, but changes may be required (Section 2.7 of [1] is currently under revision).
- Tests should work with all implementations covering RV32I instructions. It may happen that when instructions are combined with other instruction extensions, there will be some dependencies involved. These are right now not covered in the tests as this will require a configuration layer above the tests. This will be solved in future test environment.

# 3. COMPLIANCE TESTS

Two categories of tests are available in the package under BSD license protection:

1. *Binary coding tests* - checking whether assembler tool recognizes all instructions. Prerequisites for this test are an assembler and a linker tool (GCC or a proprietary assembler/linker).
2. ISA tests – checking whether instructions are implemented according to the standard.

Compliance tests should cover the specification as much as possible, therefore, links to the User-level RISC-V ISA specification are essential for tracking the **specification coverage**. For now there is no automated way of measuring specification coverage, therefore, mapping is done manually. It is expected that this will change in future versions of compliance tests when executable formal model able to process coverage is available.

## 3.1 Binary Coding Tests and Coverage

RV32I standard defines 6 formats of instructions: base instruction formats (R-type, I-type, S-type, U-type) and immediate variants (B-type, J-type). These are defined in Section 2.2 and 2.3 of [1], see also Fig. 1 and Fig. 2. Binary coding is defined in Chapter 20 of [1]: RISC-V Assembly Programmer's Handbook.

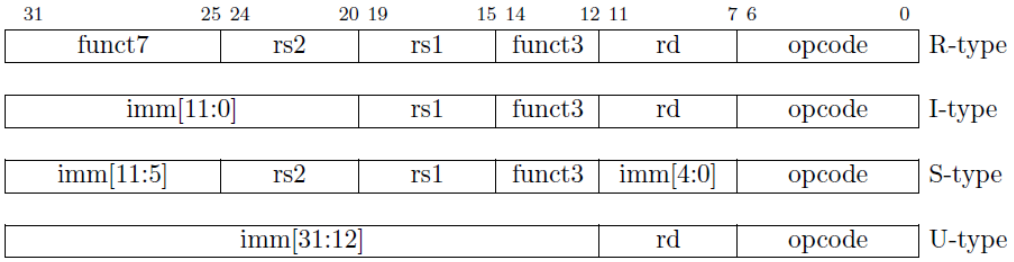



*Figure 1: RISC-V base instruction formats.*

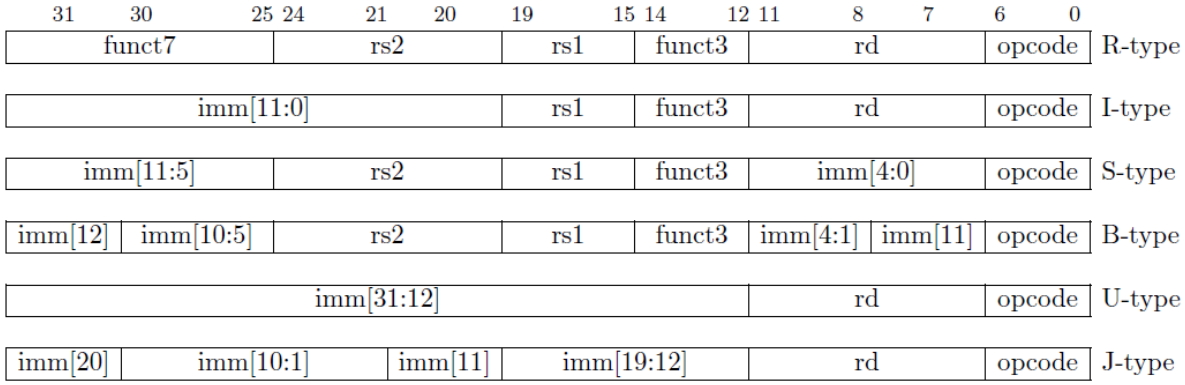



*Figure 2: RISC-V base instruction formats showing immediate variants.*

In the compliance tests package, there are separate binary coding compliance tests available for every instruction. Moreover, in every test, instruction is present in more variants, because it is possible to combine different immediate values, values stored in registers, or values stored in the memory. In order to avoid all possible combinations of all possible values of immediates and operand resources, suitable representatives should be chosen (this should be enough for compliance purposes, we do not target verification goals). Following rules are applied for the selection of representatives:

- For immediate values, an algorithm called „walking 1“ and „walking 0“ can be used together with all zeros and all ones values. Example:

  ```
  …
  0010000000000000
  0100000000000000
  1000000000000000
  1111111111111111
  1111111111111110
  1111111111111101
  …
  ```

  But of course, for some instructions it is beneficial to focus on some specific corner case values.

- For register operands, an approach with always different registers in one instruction can be applied, while always different means a *periodical exchange of consequent registers*. Example:

  ```
  …
  ADD x16, x17, x18
  ```

```
ADD x17, x18, x19
ADD x18, x19, x20
ADD x19, x20, x21
…
```

One example of complete binary coding test with explanation of its sections is available in Appendix A of this document.

It is recommended to consider tests of this category as pre-requisite tests for ISA Tests. When binary codes are not matching it doesn't make sense to continue with ISA tests.

## 3.2 ISA Tests and Coverage

RISC-V ISA defines several instructions categories. This document and tests delivered in the corresponding package consider only the RV32I instructions described in Chapter 2 of [1]. In the compliance tests package, there are separate tests available for every instruction. Aliases are not considered in version 1.0 of RV32I compliance tests. The naming convention of a single test is:

<*ISA category*>-<*test objective*>-<*test number*>.S

*ISA category* – it is expected that for all instruction categories there will be dedicated compliance tests available. Such instruction categories are: "I" or "E" representing integer instructions, "M" representing multiplication and division extension, "F" representing floating point extension, "D" representing double-precision floating point extension, "Q" representing quad-precision floating point extension, "C" representing compressed instructions extension, or "A" representing atomic instructions extension. In this package, only "I" prefix is applied.

*test objective* – the aspect on which the test is focusing on, it can be instruction, exceptions, etc.

*test number* – number of the test, it is expected that there can be more tests specified per one instruction in the future.

The list of tests with links to specific parts of the specification follows.

- *compliance_test.h* – the header file for all tests. It contains macros for defining halt of the test, the start and the end of the code section, the start and the end of the data section.

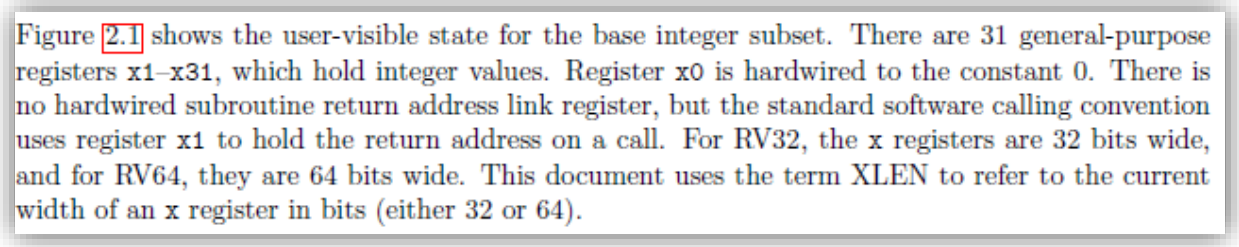


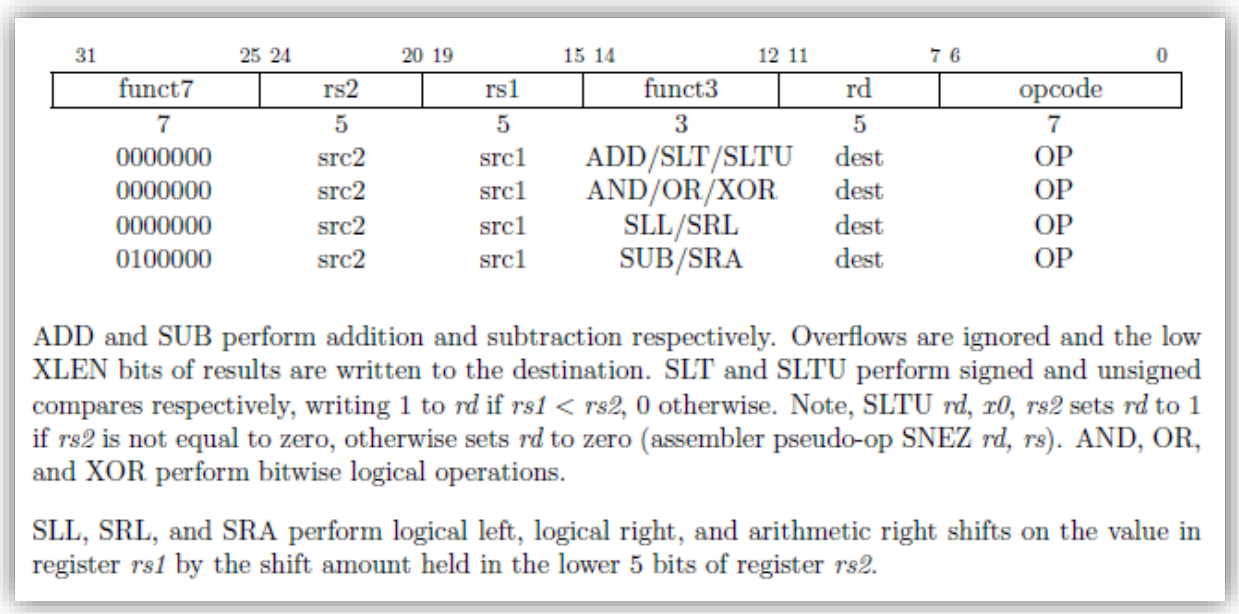
Figure 2.1 shows the user-visible state for the base integer subset. There are 31 general-purpose registers `x1`–`x31`, which hold integer values. Register `x0` is hardwired to the constant 0. There is no hardwired subroutine return address link register, but the standard software calling convention uses register `x1` to hold the return address on a call. For RV32, the `x` registers are 32 bits wide, and for RV64, they are 64 bits wide. This document uses the term XLEN to refer to the current width of an `x` register in bits (either 32 or 64).

*Figure 3: Part of the specification [1] describing registers.*

- *I-RF_size-01.S* – test checking the size of the register file. It must be possible to approach all 32 registers.

- *I-RF_width-01.S* – test checking the width of the register file. It must be possible to approach all 32 bits of all 32 registers.

- *I-RF_x0-01.S* – test checking that register x0 is hardwired to 0. All instructions that can rewrite a value in a register are tested for that purpose: LUI, ADDI, ORI, ANDI, XORI, SLLI, SRAI, SRLI, AUIPC, ADD, OR, AND, XOR, SLL, SRA, SRL, SUB, SLT, SLTU, SLTI, SLTIU, JAL, JALR, LW, LH, LB, LBU.



| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| 0000000 | src2 | src1 | ADD/SLT/SLTU | dest | OP |
| 0000000 | src2 | src1 | AND/OR/XOR | dest | OP |
| 0000000 | src2 | src1 | SLL/SRL | dest | OP |
| 0100000 | src2 | src1 | SUB/SRA | dest | OP |

ADD and SUB perform addition and subtraction respectively. Overflows are ignored and the low XLEN bits of results are written to the destination. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, *x0*, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudo-op SNEZ *rd*, *rs*). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

*Figure 4: Part of the specification [1] for instructions: ADD, SLT, SLTU, AND, OR, XOR, SLL, SRL, SUB, SRA.*

- *I-ADD-01.S* – test checking ADD (*addition*) instruction according to Section 2.4 of [1]. This instruction belongs to register-register operations, which means that destination and source operands are registers. The ISA test focuses on checking corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX, on forwarding between instructions, on writing to x0 register by ADD instruction (shouldn't change its hardwired 0 value), on forwarding through x0 register, on move operation represented by ADD instruction (x0 register is used as one of the source operands).

- *I-SLT-01.S* – tests checking SLT (*set on less than*) instruction according to Section 2.4 of [1]. This instruction belongs to register-register operations, which means that destination and source operands are registers. The ISA test focuses on checking comparing corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX, on comparing values with x0 register, on writing to x0 register by SLT instruction (shouldn't change its hardwired 0 value), on forwarding between instructions.

- *I-SLTU-01.S* – unsigned version of SLT instruction, test is focusing on the same aspects.

- *I-AND-01.S* - tests checking AND (*logical and*) instruction according to Section 2.4 of [1]. This instruction belongs to register-register operations, which means that destination and source operands are registers. The ISA test focuses on checking

logical operations over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX, on forwarding between instructions, on writing to x0 register by AND instruction (shouldn't change its hardwired 0 value), on forwarding through x0 register and moving (AND with the register containing the value -1).

- *I-OR-01.S* - tests checking OR (*logical or*) instruction according to Section 2.4 of [1]. This instruction belongs to register-register operations, which means that destination and source operands are registers. The ISA test focuses on checking logical operations over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX, on forwarding between instructions, on writing to x0 register by OR instruction (shouldn't change its hardwired 0 value), on forwarding through x0 register and moving (OR with x0).

- *I-XOR-01.S* - tests checking XOR (*logical xor*) instruction according to Section 2.4 of [1]. This instruction belongs to register-register operations, which means that destination and source operands are registers. The ISA test focuses on checking logical operations over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX, on forwarding between instructions, on writing to x0 register by XOR instruction (shouldn't change its hardwired 0 value), on forwarding through x0 register and moving (XOR with x0).

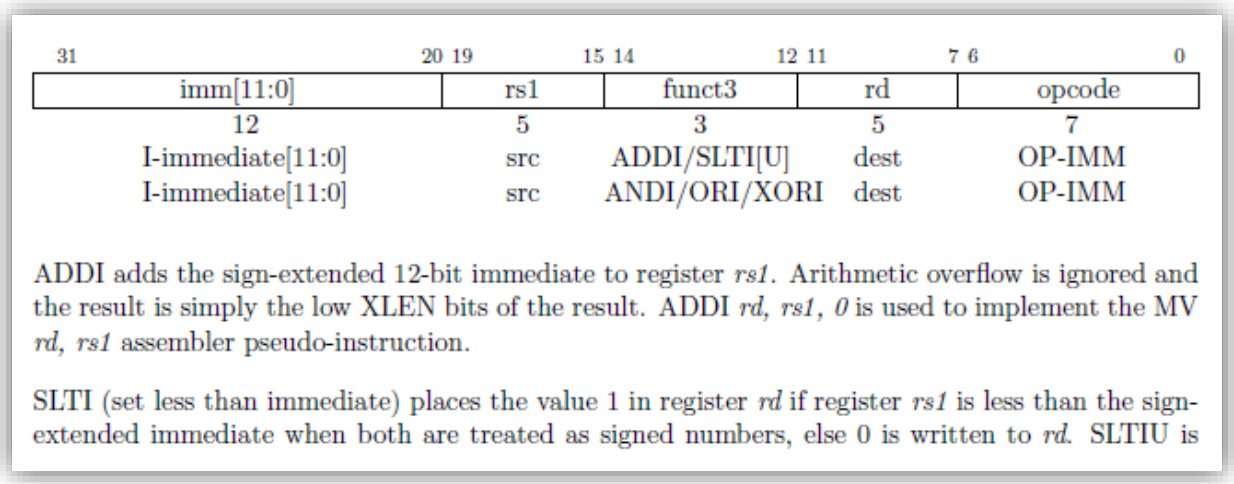
- *I-SLL-01.S* - tests checking SLL (*shift left logical*) instruction according to Section 2.4 of [1]. This instruction belongs to register-register operations, which means that destination and source operands are registers. The ISA test focuses on checking shifts over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX (on 5 bits), on forwarding between instructions, on writing to x0 register by SLL instruction (shouldn't change its hardwired 0 value), on forwarding through x0 register and shifting by value greater than 31 – only low 5 bits of the register value should be used.

- *I-SRL-01.S* - tests checking SRL (*shift right logical*) instruction according to Section 2.4 of [1]. This instruction belongs to register-register operations, which means that destination and source operands are registers. The ISA test focuses on checking shifts over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX (on 5 bits), on forwarding between instructions, on writing to x0 register by SRL instruction (shouldn't change its hardwired 0 value), on forwarding through x0 register and shifting by value greater than 31 – only low 5 bits of the register value should be used.

- *I-SUB-01.S* - test checking SUB (*subtraction*) instruction according to Section 2.4 of [1]. This instruction belongs to register-register operations, which means that destination and source operands are registers. The ISA test focuses on checking corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX, on forwarding between instructions, on writing to x0 register by SUB instruction (shouldn't change its hardwired 0 value), on forwarding through x0 register, on move operation and negation (SUB with x0).

- *I-SRA-01.S* - tests checking SRA (*shift right arithmetic*) instruction according to Section 2.4 of [1]. This instruction belongs to register-register operations, which means that destination and source operands are registers. The ISA test focuses on checking shifts over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX

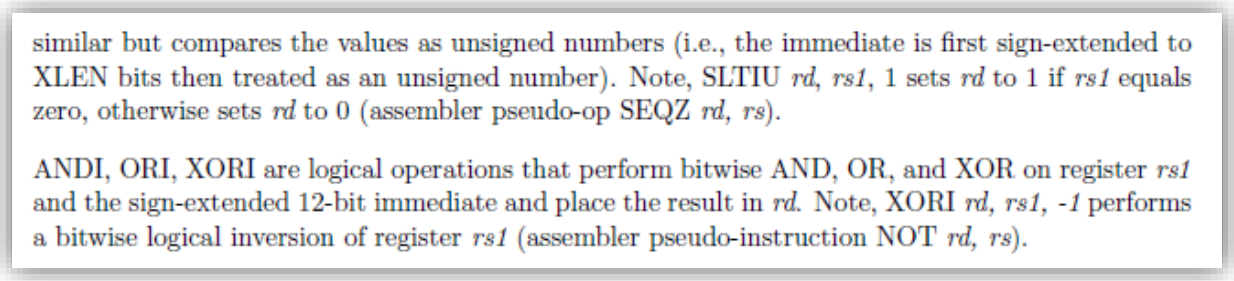
(on 5 bits), on forwarding between instructions, on writing to x0 register by SRA instruction (shouldn't change its hardwired 0 value), on forwarding through x0 register and shifting by value greater than 31 – only 5 low bits of the register value should be used.



| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| I-immediate[11:0] | src | ADDI/SLTI[U] | dest | OP-IMM |
| I-immediate[11:0] | src | ANDI/ORI/XORI | dest | OP-IMM |

ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI *rd, rs1, 0* is used to implement the MV *rd, rs1* assembler pseudo-instruction.

SLTI (set less than immediate) places the value 1 in register *rd* if register *rs1* is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to *rd*. SLTIU is



similar but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to XLEN bits then treated as an unsigned number). Note, SLTIU *rd, rs1*, 1 sets *rd* to 1 if *rs1* equals zero, otherwise sets *rd* to 0 (assembler pseudo-op SEQZ *rd, rs*).

ANDI, ORI, XORI are logical operations that perform bitwise AND, OR, and XOR on register *rs1* and the sign-extended 12-bit immediate and place the result in *rd*. Note, XORI *rd, rs1, -1* performs a bitwise logical inversion of register *rs1* (assembler pseudo-instruction NOT *rd, rs*).

*Figure 5: Part of the specification [1] for instructions: ADDI, SLTI, SLTIU, ANDI, ORI, XORI.*

- *I-ADDI-01.S* – test checking ADDI (*addition with immediate*) instruction according to Section 2.4 of [1]. This instruction belongs to register-immediate operations, which means that destination and optionally one source operand are registers and second source operand is immediate. The ISA test focuses on checking corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX immediates, on forwarding between instructions, on writing to x0 register by ADDI instruction (shouldn't change its hardwired 0 value), on forwarding through x0 register, on move operation represented by ADDI instruction with the 0 immediate value.

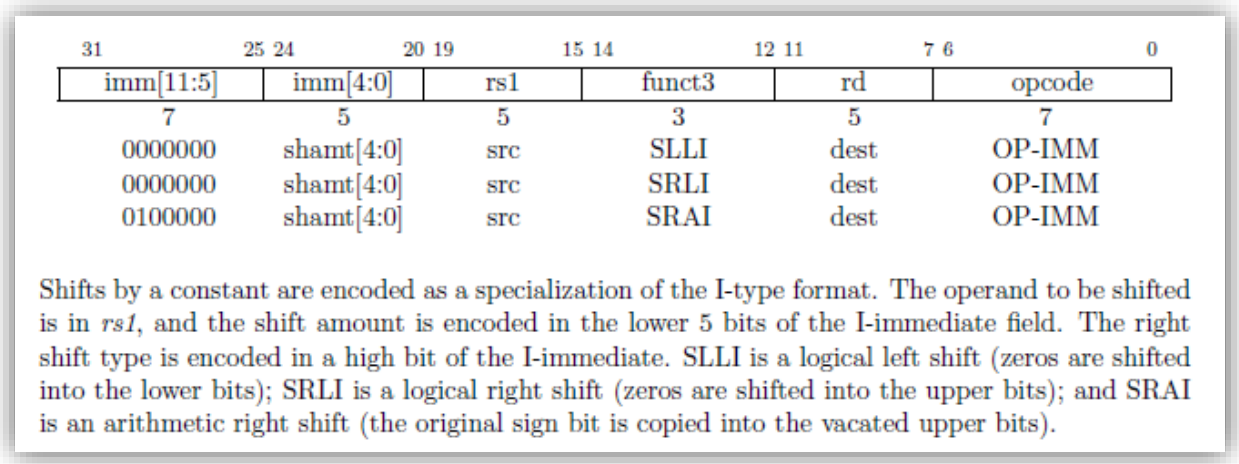
- *I-SLTI-01.S* - tests checking SLTI (*set less than immediate*) instruction according to Section 2.4 of [1]. This instruction belongs to register-immediate operations, which means that destination and optionally one source operand are registers and second source operand is immediate. The ISA test focuses on checking comparing corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX immediates, on comparing values with x0 register, on writing to x0 register by SLTI instruction (shouldn't change its hardwired 0 value), on forwarding between instructions.

- *I-SLTIU-01.S* - unsigned version of SLTI instruction, test is focusing on the same aspects.

- *I-ANDI-01.S* - tests checking ANDI (*logical and with immediate*) instruction according to Section 2.4 of [1]. This instruction belongs to register-immediate operations, which means that destination and optionally one source operand are registers and second source operand is immediate. The ISA test focuses on checking logical operations over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX immediates, on forwarding between instructions, on writing to x0 register by ANDI instruction (shouldn't change its hardwired 0 value), on forwarding through x0 register and moving (ANDI with the -1 immediate value).

- *I-ORI-01.S* - tests checking ORI (*logical or with immediate*) instruction according to Section 2.4 of [1]. This instruction belongs to register-immediate operations, which means that destination and optionally one source operand are registers and second source operand is immediate. The ISA test focuses on checking logical operations over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX immediates, on forwarding between instructions, on writing to x0 register by ORI instruction (shouldn't change its hardwired 0 value), on forwarding through x0 register and moving (ORI with the 0 immediate value).

- *I-XORI-01.S* - tests checking XORI (*logical xor with immediate*) instruction according to Section 2.4 of [1]. This instruction belongs to register-immediate operations, which means that destination and optionally one source operand are registers and second source operand is immediate. The ISA test focuses on checking logical operations over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX immediates, on forwarding between instructions, on writing to x0 register by XORI instruction (shouldn't change its hardwired 0 value), on forwarding through x0 register and moving (XORI with the 0 immediate value).



| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| imm[11:5] | imm[4:0] | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| 0000000 | shamt[4:0] | src | SLLI | dest | OP-IMM |
| 0000000 | shamt[4:0] | src | SRLI | dest | OP-IMM |
| 0100000 | shamt[4:0] | src | SRAI | dest | OP-IMM |

Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 5 bits of the I-immediate field. The right shift type is encoded in a high bit of the I-immediate. SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

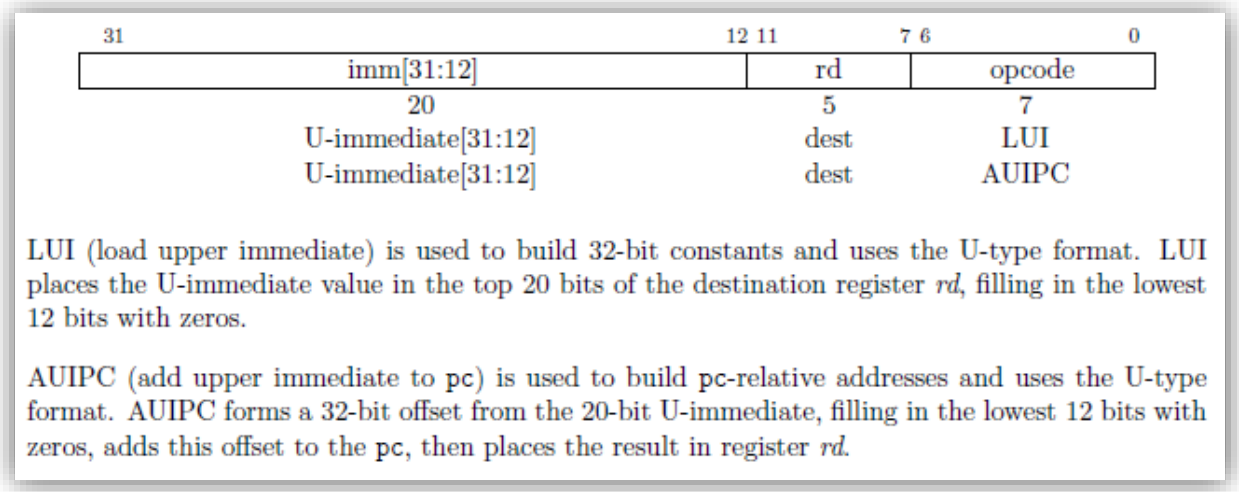
*Figure 6: Part of the specification [1] for instructions: SLLI, SRLI, SRAI.*

- *I-SLLI-01.S* - tests checking SLLI (*shift left logical by immediate*) instruction according to Section 2.4 of [1]. This instruction belongs to register-immediate operations, which means that destination and optionally one source operand are registers and second

source operand is immediate. The ISA test focuses on checking shifts over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX (5 bits immediates), on forwarding between instructions, on writing to x0 register by SLLI instruction (shouldn't change its hardwired 0 value), and on forwarding through x0.

- *I-SRAI-01.S* - tests checking SRAI (*shift right arithmetic by immediate*) instruction according to Section 2.4 of [1]. This instruction belongs to register-immediate operations, which means that destination and optionally one source operand are registers and second source operand is immediate. The ISA test focuses on checking shifts over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX (on 5 bits immediate), on forwarding between instructions, on writing to x0 register by SRAI instruction (shouldn't change its hardwired 0 value), and on forwarding through x0.

- *I-SRLI-01.S* - tests checking SRLI (*shift right logical by immediate*) instruction according to Section 2.4 of [1]. This instruction belongs to register-immediate operations, which means that destination and optionally one source operand are registers and second source operand is immediate. The ISA test focuses on checking shifts over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX (on 5 bits immediate), on forwarding between instructions, on writing to x0 register by SRL instruction (shouldn't change its hardwired 0 value), and on forwarding through x0 register.



| 31 — 12 | 11 — 7 | 6 — 0 |
|---|---|---|
| imm[31:12] | rd | opcode |
| 20 | 5 | 7 |
| U-immediate[31:12] | dest | LUI |
| U-immediate[31:12] | dest | AUIPC |

LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros.

AUIPC (add upper immediate to pc) is used to build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register *rd*.

*Figure 7: Part of the specification [1] for LUI and AUIPC instructions.*

- *I-LUI-01.S* - test checking LUI (*load upper immediate*) instruction according to Section 2.4 of [1]. This instruction belongs to register-immediate operations, which means that destination and optionally one source operand are registers and second source operand is immediate. The ISA test focuses on loading 0, 1, -1, MIN, MAX immediates to destination register, on using LUI to overwrite low bits and on loading immediate using LUI and ADDI, or LI alias.

- *I-AUIPC-01.S* - test checking AUIPC (*add upper immediate to pc*) instruction according to Section 2.4 of [1]. This instruction belongs to register-immediate operations, which

means that destination and optionally one source operand are registers and second source operand is immediate. The ISA test focuses on adding 0, 1, -1, MIN, MAX immediates to the program counter and then storing the value to the destination register, on using AUIPC to overwrite low bits and on loading immediate using AUIPC and ADDI, or LA alias.

**NOP Instruction**

| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| 0 | 0 | ADDI | 0 | OP-IMM |

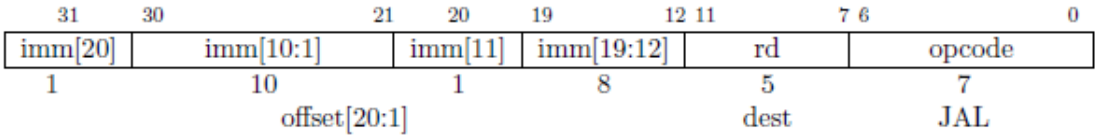
The NOP instruction does not change any user-visible state, except for advancing the pc. NOP is encoded as ADDI *x0, x0, 0*.

*Figure 8: Part of the specification for NOP instruction.*

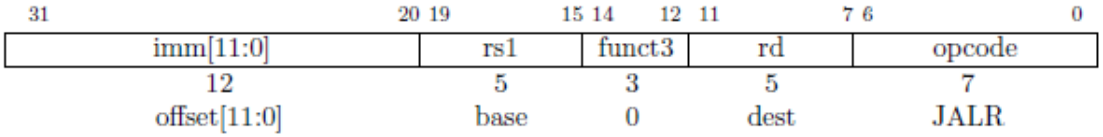
- *I-NOP-01.S* - test checking NOP (*no operation*) instruction according to Section 2.4 of [1]. The ISA test focuses on checking whether it doesn't influence values in registers (only program counter should be incremented).

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the pc to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following the jump (pc+4) into register *rd*. The standard software calling convention uses x1 as the return address register and x5 as an alternate link register.

Plain unconditional jumps (assembler pseudo-op J) are encoded as a JAL with *rd*=x0.




The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the 12-bit signed I-immediate to the register *rs1*, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register *rd*. Register x0 can be used as the destination if the result is not required.




The JAL and JALR instructions will generate a misaligned instruction fetch exception if the target address is not aligned to a four-byte boundary.

*Figure 9: Part of the specification describing JAL andf JALR instructions.*

- *I-JAL-01.S* - test checking JAL (*jump and link*) instruction according to Section 2.5 of [1]. This instruction belongs to unconditional jumps. The ISA test focuses on jumps forward, jumps backward and on linking (correct return address is saved).

- *I-JALR-01.S* - test checking JALR (*jump and link register*) instruction according to Section 2.5 of [1]. This instruction belongs to unconditional jumps. The ISA test focuses on jumps forward, jumps backward, on linking (saving correct return address), on clearing low bit of the address – low bit of the address is not causing exception, and on jumps using immediate offset.

- *I-DELAY_SLOTS-01.S* – test checking delay slots of conditional or unconditional jumps.

- I-MISALIGN_JMP-01.S – test checking exception caused by misaligned conditional or unconditional jumps.

All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2, and is added to the current pc to give the target address. The conditional branch range is ±4 KiB.

| 31 | 30 25 | 24 20 | 19 15 | 14 12 | 11 8 | 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode |
| 1 | 6 | 5 | 5 | 3 | 4 | 1 | 7 |
| offset[12,10:5] | | src2 | src1 | BEQ/BNE | offset[11,4:1] | | BRANCH |
| offset[12,10:5] | | src2 | src1 | BLT[U] | offset[11,4:1] | | BRANCH |
| offset[12,10:5] | | src2 | src1 | BGE[U] | offset[11,4:1] | | BRANCH |

Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively. BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

*Figure 10: Part of the specification describing BEQ, BNE, BLT, BLTU, BGE, BGEU.*

- *I-BEQ- 01.S* - test checking BEQ (*branch if equal*) instruction according to Section 2.5 of [1]. This instruction belongs to conditional branches. The ISA test focuses on checking conditions over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX, on testing compare with x0, and on jumps forward and backward.

- *I-BGE-01.S* - test checking BGE (*branch if greater or equal*) instruction according to Section 2.5 of [1]. This instruction belongs to conditional branches. The ISA test focuses on checking conditions over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX, on testing compare with x0, and on jumps forward and backward.

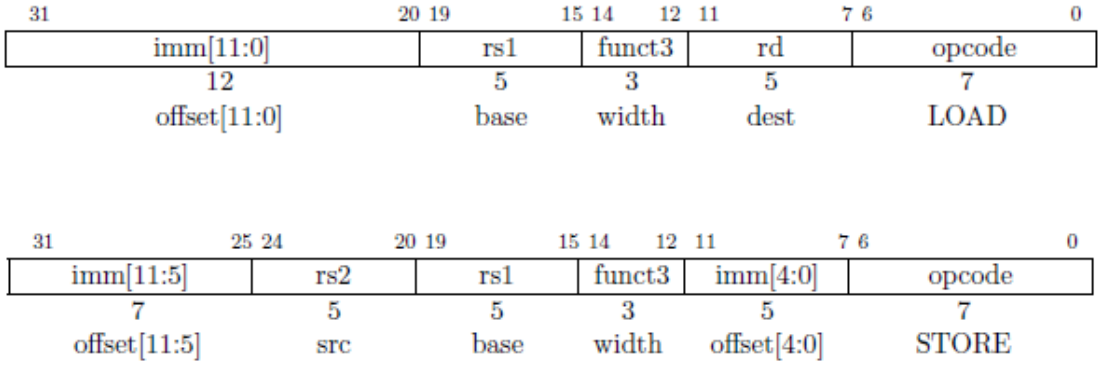
- *I-BGEU-01.S* - unsigned version of BGE instruction, test is focusing on the same aspects.

- *I-BLT-01.S* - test checking BLT (*branch if less*) instruction according to Section 2.5 of [1]. This instruction belongs to conditional branches. The ISA test focuses on checking conditions over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX, on testing compare with x0, and on jumps forward and backward.

- *I-BLTU-01.S* - unsigned version of BGE instruction, test is focusing on the same aspects.

- I-BNE-01.S - test checking BNE (*branch if not equal*) instruction according to Section 2.5 of [1]. This instruction belongs to conditional branches. The ISA test focuses on checking conditions over corner case values: 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX, on testing compare with x0, and on jumps forward and backward.

RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. RV32I provides a 32-bit user address space that is byte-addressed and little-endian. The execution environment will define what portions of the address space are legal to access. Loads with a destination of `x0` must still raise any exceptions and action any other side effects even though the load value is discarded.




Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective byte address is obtained by adding register *rs1* to the sign-extended 12-bit offset. Loads copy a value from memory to register *rd*. Stores copy the value in register *rs2* to memory.

The LW instruction loads a 32-bit value from memory into *rd*. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in `rd`. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in *rd*. LB and LBU are defined analogously for 8-bit values. The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register *rs2* to memory.

For best performance, the effective address for all loads and stores should be naturally aligned for each data type (i.e., on a four-byte boundary for 32-bit accesses, and a two-byte boundary for 16-bit accesses). The base ISA supports misaligned accesses, but these might run extremely slowly depending on the implementation. Furthermore, naturally aligned loads and stores are guaranteed to execute atomically, whereas misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity.

*Figure 11: Part of specification describing LOAD and STORE instructions and endianity.*

- I-LB-01.S - test checking LB (*load byte*) instruction according to Section 2.6 of [1]. This instruction belongs to load instructions. The ISA test focuses on checking <base address + 0> load, <base address -1> load, <base address + 1> load, <base address - 2048> load, <base address + 2047> load, <base address [from -4 to +7]> load, load to x0, forwarding, and consequent load and store with the same base and destination addresses.

- I-LBU-01.S - unsigned version of LB instruction, test is focusing on the same aspects.

- I-LH-01.S - test checking LH (*load word and sign-extend high bits*) instruction according to Section 2.6 of [1]. This instruction belongs to load instructions. The ISA test focuses on checking <base address + 0> load, <base address -1> load, <base address + 1> load, <base address - 2048> load, <base address + 2047> load, <base address [-4, -2, 0, +2, +4, +6]> load, load to x0, forwarding, and consequent load and store with the same base and destination addresses.

- I-LHU-01.S - unsigned version of LH instruction, test is focusing on the same aspects.

- I-LW-01.S - test checking LW (*load word*) instruction according to Section 2.6 of [1]. This instruction belongs to load instructions. The ISA test focuses on checking <base address + 0> load, <base address -1> load, <base address + 1> load, <base address - 2048> load, <base address + 2047> load, <base address [-4, 0, 4]> load, load to x0, forwarding, and consequent load and store with the same base and destination addresses.

- I-SB-01.S - test checking SB (*store byte*) instruction according to Section 2.6 of [1]. This instruction belongs to store instructions. The ISA test focuses on checking <base address + 0> store, <base address -1> store, <base address + 1> store, <base address - 2048> store, <base address + 2047> store, <base address [from -4 to +7]> store, store from x0, forwarding to address an data register, WAR hazard with address and data register, RAW hazard in memory, and WAW hazard in memory.

- I-SH-01.S - test checking SH (*store word and sign-extend high bits*) instruction according to Section 2.6 of [1]. This instruction belongs to store instructions. The ISA test focuses on checking <base address + 0> store, <base address -1> store, <base address + 1> store, <base address - 2048> store, <base address + 2047> store, <base address [-4, -2, 0, +2, +4, +6]> store, store from x0, forwarding to address an data register, WAR hazard with address and data register, RAW hazard in memory, and WAW hazard in memory.

- I-SW-01.S - test checking SW (*store word*) instruction according to Section 2.6 of [1]. This instruction belongs to store instructions. The ISA test focuses on checking <base address + 0> store, <base address -1> store, <base address + 1> store, <base address - 2048> store, <base address + 2047> store, <base address [-4, 0, 4]> store, store from x0, forwarding to address an data register, WAR hazard with address and data register, RAW hazard in memory, and WAW hazard in memory.

- I-MISALIGN-LDST-01.S – test checking misaligned load/store exception.

- I-ENDIANESS-01.S – test checking correct endianity.

We define the full set of CSR instructions here, although in the standard user-level base ISA, only a handful of read-only counter CSRs are accessible.

| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|
| csr | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| source/dest | source | CSRRW | dest | SYSTEM |
| source/dest | source | CSRRS | dest | SYSTEM |
| source/dest | source | CSRRC | dest | SYSTEM |
| source/dest | uimm[4:0] | CSRRWI | dest | SYSTEM |
| source/dest | uimm[4:0] | CSRRSI | dest | SYSTEM |
| source/dest | uimm[4:0] | CSRRCI | dest | SYSTEM |

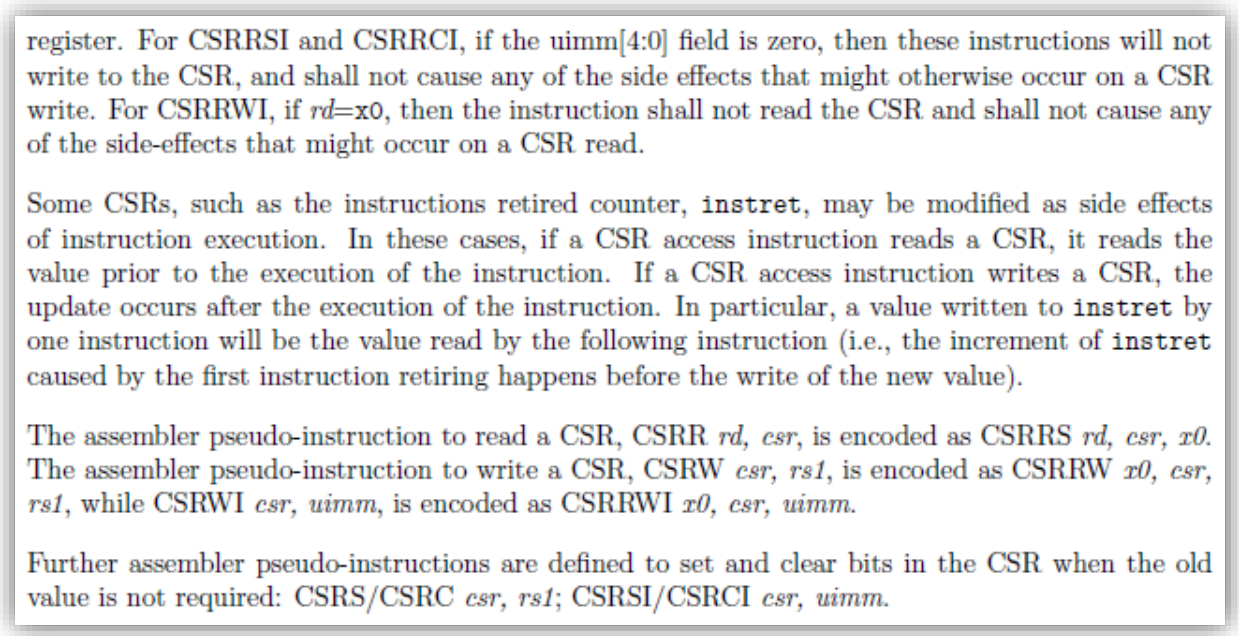
The CSRRW (Atomic Read/Write CSR) instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register *rd*. The initial value in *rs1* is written to the CSR. If *rd*=`x0`, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

The CSRRS (Atomic Read and Set Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register *rd*. The initial value in integer register *rs1* is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written).

The CSRRC (Atomic Read and Clear Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register *rd*. The initial value in integer register *rs1* is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected.

For both CSRRS and CSRRC, if *rs1*=`x0`, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs. Note that if *rs1* specifies a register holding a zero value other than `x0`, the instruction will still attempt to write the unmodified value back to the CSR and will cause any attendant side effects.

The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the *rs1* field instead of a value from an integer



register. For CSRRSI and CSRRCI, if the uimm[4:0] field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write. For CSRRWI, if *rd*=x0, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

Some CSRs, such as the instructions retired counter, `instret`, may be modified as side effects of instruction execution. In these cases, if a CSR access instruction reads a CSR, it reads the value prior to the execution of the instruction. If a CSR access instruction writes a CSR, the update occurs after the execution of the instruction. In particular, a value written to `instret` by one instruction will be the value read by the following instruction (i.e., the increment of `instret` caused by the first instruction retiring happens before the write of the new value).

The assembler pseudo-instruction to read a CSR, CSRR *rd, csr*, is encoded as CSRRS *rd, csr, x0*. The assembler pseudo-instruction to write a CSR, CSRW *csr, rs1*, is encoded as CSRRW *x0, csr, rs1*, while CSRWI *csr, uimm*, is encoded as CSRRWI *x0, csr, uimm*.

Further assembler pseudo-instructions are defined to set and clear bits in the CSR when the old value is not required: CSRS/CSRC *csr, rs1*; CSRSI/CSRCI *csr, uimm*.

*Figure 12: Part of the specification [1] describing CSR instructions.*

- I-CSRRC-01.S - test checking CSRRC (*atomic read and clear bits in CSR*) instruction according to Section 2.8 of [1]. This instruction belongs to control and status register instructions. The ISA test focuses on checking base operation of CSRRC, on forwarding between instructions, on writing and reading to/from x0, on forwarding through x0, and on testing CSRRC with the same destination and source registers. This test is coupled with CSRRW instruction.

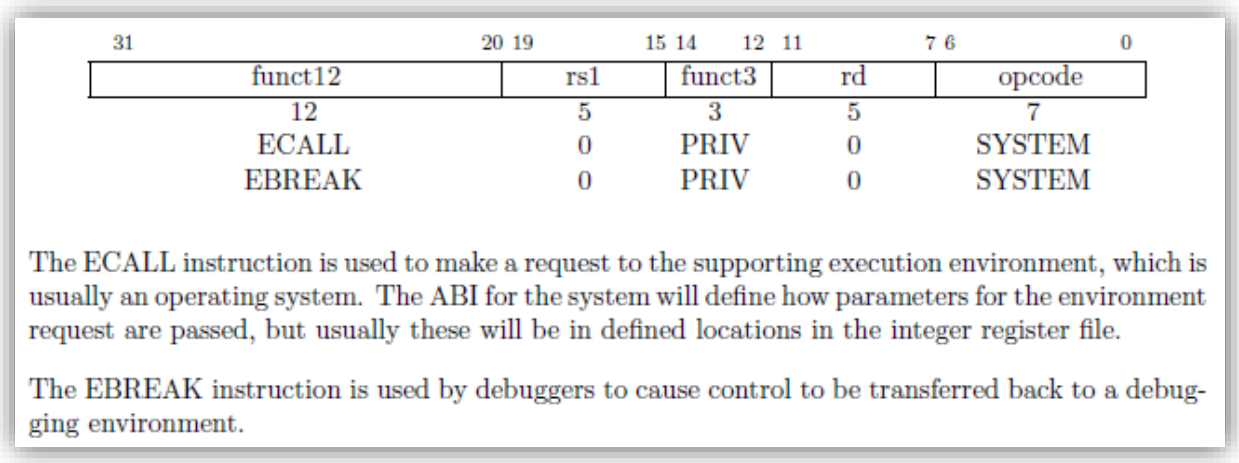
- I-CSRRCI-01.S – variant of CSRRC with immediate. Test is focusing on the same aspects.

- I-CSRRS-01.S - test checking CSRRS (*atomic read and set bits in CSR*) instruction according to Section 2.8 of [1]. This instruction belongs to control and status register instructions. The ISA test focuses on checking base operation of CSRRS, on forwarding between instructions, on writing and reading to/from x0, on forwarding through x0, and on testing CSRRS with the same destination and source registers. This test is coupled with CSRRW instruction.

- I-CSRRSI-01.S - variant of CSRRS with immediate. Test is focusing on the same aspects.

- I-CSRRW-01.S - test checking CSRRW (*atomic read/write CSR*) instruction according to Section 2.8 of [1]. This instruction belongs to control and status register instructions. The ISA test focuses on checking base operation of CSRRW, on forwarding between instructions, on writing to x0, on forwarding through x0, and on testing CSRRW with the same destination and source registers.

- I-CSRRWI-01.S - variant of CSRRW with immediate. Test is focusing on the same aspects.



| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|
| funct12 | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| ECALL | 0 | PRIV | 0 | SYSTEM |
| EBREAK | 0 | PRIV | 0 | SYSTEM |

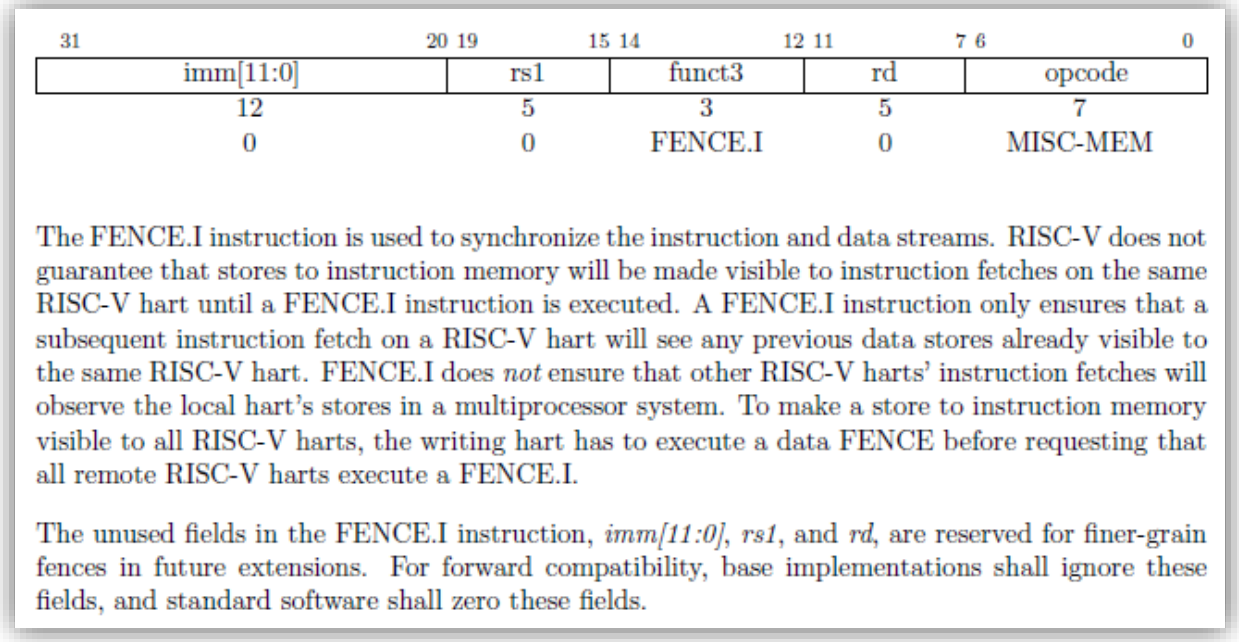
The ECALL instruction is used to make a request to the supporting execution environment, which is usually an operating system. The ABI for the system will define how parameters for the environment request are passed, but usually these will be in defined locations in the integer register file.

The EBREAK instruction is used by debuggers to cause control to be transferred back to a debugging environment.

*Figure 13: Part of the specification [1] describing ECALL and EBREAK instructions.*

- I-EBREAK-01.S - test checking EBREAK (*environment break*) instruction according to Section 2.9 of [1]. This instruction belongs to environment calls and breakpoints instructions. The ISA test focuses on checking base operation of EBREAK.

- I-ECALL-01.S - test checking ECALL (*environment call*) instruction according to Section 2.9 of [1]. This instruction belongs to environment calls and breakpoints instructions. The ISA test focuses on checking base operation of ECALL.



| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| 0 | 0 | FENCE.I | 0 | MISC-MEM |

The FENCE.I instruction is used to synchronize the instruction and data streams. RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on the same RISC-V hart until a FENCE.I instruction is executed. A FENCE.I instruction only ensures that a subsequent instruction fetch on a RISC-V hart will see any previous data stores already visible to the same RISC-V hart. FENCE.I does *not* ensure that other RISC-V harts' instruction fetches will observe the local hart's stores in a multiprocessor system. To make a store to instruction memory visible to all RISC-V harts, the writing hart has to execute a data FENCE before requesting that all remote RISC-V harts execute a FENCE.I.

The unused fields in the FENCE.I instruction, *imm[11:0]*, *rs1*, and *rd*, are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard software shall zero these fields.

- I-FENCE.I-01.S - test checking FENCE.I instruction according to Section 2.7 of [1]. This instruction belongs to environment calls and breakpoints instructions. The ISA test focuses on checking base operation of FENCE.I.

One example of complete ISA test with explanation of its sections is available in Appendix B of this document.

# 4. REFERENCES

## 4.1 Available Reference Signatures

After compilation of every binary test in `binary_coding/src` directory, the resulting binary code is compared to the reference one available in `binary_coding/references` and inconsistences are immediately reported.

After compilation of every ISA test in `ISA/src` directory, results of the test are compared to the reference results available in `ISA/references` and inconsistences are immediately reported.

Reference signatures (reference results for every compliance test) are defined as images from the registers at the end of processing every compliance test in the Spike simulator. So they originate from the Spike simulator. Moreover, they were **manually inspected** for correctness based on the test intent. References are available in the form of text files.

## 4.2 Possibility of Future References

In the future, it would be feasible to combine more reference models of different natures. One viable option seems to be utilization of executable formal reference model which is preparing the Formal Task Group of RISC-V Foundation.

# 5. TEST ENVIRONMENT

For running compliance tests of Version 1.0.A, Test Virtual Machine (TVM) “p” available at https://github.com/riscv/riscv-tests is utilized. In addition to the basic functionality of TVM, the script for running compliance tests performs comparison of the *Design Under Test* (DUT) to the manually reviewed reference signatures. When all tests pass it means that DUT is compliant to RV32I ISA.

Provided package requires running RISC-V tools available at https://github.com/riscv/riscv-tools. RISC-V SW tools are used for test compilation and Spike is considered as DUT. Tests can be run by command

*./run_tests.sh* <path_to_RISCV_tools/bin>

It is expected that the user will compliance-check his/her own implementation of the RISC-V processor and will substitute Spike or SW tools while running tests. The steps how to do this are defined in Section 5.2.

## 5.1 Test Virtual Machines

For a detailed description of TVM please see description provided at https://github.com/riscv/riscv-tests.

## 5.2 User Test Environment

In this section, a short tutorial how to replace the default DUT (the Spike simulator) for the user DUT in TVM is provided.

If you do not want to use TVM at all, it is recommended to just take the tests and references and incorporate them into your testing environment. The only requirement needed in this case is that there must be an option to dump the results from the DUT in the test environment so as the comparison to references is possible.

The following steps demonstrate an example in which Spike DUT was replaced by Codasip ISA simulator. In a similar way, any RISC-V ISA simulator can be connected or any RTL simulation model of the RISC-V processor can be connected.

- Redefining macros in *ISA/src/compliance_test.h* and *binary_coding/src/compliance_test.h*.

  For example, to support Codasip ISA simulator as DUT, it was necessary to redefine `RV_COMPLIANCE_HALT` macro, `RV_COMPLIANCE_DATA_BEGIN` macro and `RV_COMPLIANCE_DATA_END` macro in *ISA/compliance_test.h* in the following way:

```
#define RV_COMPLIANCE_HALT
    add x31, x0, 1
    sw x31, codasip_syscall, t0
```

  ➔ It means that on the address defined by *codasip_syscall*, the 1 value is stored and this is interpreted as HALT for Codasip ISA simulator.

```
#define RV_COMPLIANCE_DATA_BEGIN
    .align 4;
    .global codasip_signature_start;
    codasip_signature_start:

#define RV_COMPLIANCE_DATA_END
    .align 4;
    .global codasip_signature_end;
    codasip_signature_end:
```

  ➔ Codasip ISA simulator dumps data from the addresses bounded by labels `codasip_signature_start` and `codasip_signature_end` to stdout (dumped data represent results of the tests).

- Modifying Makefiles in *ISA/Makefile* and *binary_coding/Makefile*. It is important to change tools that are evaluated and parameters that are passed to the tools.

  For example, to support Codasip ISA simulator as DUT, it was necessary to change `RISCV_SIM` from spike to `codix_berkelium-ia-isimulator -r` and parameters for running the simulator from `+signature=$(work_dir)/$<_signature.output` to `-info 5` plus handle redirection to a file by `1>$(work_dir)/$<_signature.output`.

# 6. APPENDIX A

A detailed description of one binary code test.

➔ Header file including riscv-test.h from TVM, but in case you are not using TVM, header files of YOUR test environment should be included.

```
#include "compliance_test.h"
```

➔ Code region – selected instruction is checked for binary compatibility in many its variants. The variants are selected based on the rules defined in Section 3.1 of this document. So for example, instruction ADD is Register-Register instruction, therefore, all

source and destionation operands are registers. It means that the algorithm for periodical exchange of consequent registers is applied.

```
# Test code region.
RV_COMPLIANCE_CODE_BEGIN

    # -----------------------------------------------------------------------
    # ADD
    add     x0, x1, x2
    add     x1, x2, x3
    add     x2, x3, x4
    add     x3, x4, x5
    add     x4, x5, x6
    add     x5, x6, x7
    add     x6, x7, x8
    add     x7, x8, x9
    add     x8, x9, x10
    add     x9, x10, x11
    add     x10, x11, x12
    add     x11, x12, x13
    add     x12, x13, x14
    add     x13, x14, x15
    add     x14, x15, x16
    add     x15, x16, x17
    add     x16, x17, x18
    add     x17, x18, x19
    add     x18, x19, x20
    add     x19, x20, x21
    add     x20, x21, x22
    add     x21, x22, x23
    add     x22, x23, x24
    add     x23, x24, x25
    add     x24, x25, x26
    add     x25, x26, x27
    add     x26, x27, x28
    add     x27, x28, x29
    add     x28, x29, x30
    add     x29, x30, x31
    add     x30, x31, x0
    add     x31, x0, x1

# CODE_END
```

# 7. APPENDIX B

A detailed description of one ISA test.

➔ Header file including riscv-test.h from TVM, but in case you are not using TVM, header files of YOUR test environment should be included.

```
#include "compliance_test.h"
```

➔ TVM selection.

```
# Test Virtual Machine (TVM) used by program.
RV_COMPLIANCE_RV32M
```

➔ Code region – ISA test is divided into several parts marked as "A","B","C", etc. These parts differenciate different logical tests.

```
# Test code region.
RV_COMPLIANCE_CODE_BEGIN
```

➔ "A" parts of this test focus on checking corner case values of the ADD instruction. In particular, 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX values.

```
    # -----------------------------------------------------------------------
    # Test part A1 - general test of value 0 with 0, 1, -1, MIN, MAX
register values

    # Addresses for test data and results
    la      x1, test_A1_data
    la      x2, test_A1_res

    # Load testdata
    lw      x3, 0(x1)

    # Register initialization
    li      x4, 0
    li      x5, 1
    li      x6, -1
    li      x7, 0x7FFFFFFF
    li      x8, 0x80000000

    # Test
    add     x4, x3, x4
    add     x5, x3, x5
    add     x6, x3, x6
    add     x7, x3, x7
    add     x8, x3, x8

    # Store results
    sw      x3, 0(x2)
    sw      x4, 4(x2)
    sw      x5, 8(x2)
    sw      x6, 12(x2)
    sw      x7, 16(x2)
    sw      x8, 20(x2)

    # -----------------------------------------------------------------------
    # Test part A2 - general test of value 1 with 0, 1, -1, MIN, MAX
register values

    <similar code to A1>

    # -----------------------------------------------------------------------
    # Test part A3 - general test of value -1 with 0, 1, -1, MIN, MAX
register values

    <similar code to A1>

    # -----------------------------------------------------------------------
    # Test part A4 - general test of value 0x7FFFFFFF with 0, 1, -1, MIN,
MAX register values

    <similar code to A1>
```

```
    # ------------------------------------------------------------------------
    # Test part A5 - general test of value 0x80000000 with 0, 1, -1, MIN,
MAX register values

    <similar code to A1>
```

➔ "B" part of this test focuses on forwarding between instruction. It means that result of one instruction is immediately passed to another instruction

```
    # ------------------------------------------------------------------------
    # Test part B - testing forwarding between instructions

    # Addresses for test data and results
    la      x25, test_B_data
    la      x26, test_B_res

    # Load testdata
    lw      x28, 0(x25)

    # Register initialization
    li      x27, 0x1

    # Test
    add     x29, x28, x27
    add     x30, x29, x27
    add     x31, x30, x27
    add     x1, x31, x27
    add     x2, x1, x27
    add     x3, x2, x27

    # store results
    sw      x27, 0(x26)
    sw      x28, 4(x26)
    sw      x29, 8(x26)
    sw      x30, 12(x26)
    sw      x31, 16(x26)
    sw      x1, 20(x26)
    sw      x2, 24(x26)
    sw      x3, 28(x26)
```

➔ "C" part of this test focuses on writing to x0. This register is hardwired to 0 value so it cannot happen in any RISC-V implementation that it is overwritten.

```
    # ------------------------------------------------------------------------
    # Test part C - testing writing to x0

    # Addresses for test data and results
    la      x1, test_C_data
    la      x2, test_C_res

    # Load testdata
    lw      x28, 0(x1)

    # Register initialization
    li      x27, 0xF7FF8818

    # Test
    add     x0, x28, x27
```

```
# store results
sw      x0, 0(x2)
```

➔ "D" part of this test focuses on forwarding through x0. This register is hardwired to 0 value, so temporary nonzero result cannot be passed to another instruction.

```
# ----------------------------------------------------------------------
# Test part D - testing forwarding throught x0

# Addresses for test data and results
la      x1, test_D_data
la      x2, test_D_res

# Load testdata
lw      x28, 0(x1)

# Register initialization
li      x27, 0xF7FF8818

# Test
add     x0, x28, x27
add     x5, x0, x0

# store results
sw      x0, 0(x2)
sw      x5, 4(x2)
```

➔ "E" part of this test focuses on ADD with x0. The ADD instruction performs MOVE operation in that case.

```
# ----------------------------------------------------------------------
# Test part E - testing moving (add with x0)

# Addresses for test data and results
la      x1, test_E_data
la      x2, test_E_res

# Load testdata
lw      x3, 0(x1)

# Test
add     x4, x3, x0
add     x5, x4, x0
add     x6, x0, x5
add     x14, x6, x0
add     x15, x14, x0
add     x16, x15, x0
add     x25, x0, x16
add     x26, x0, x25
add     x27, x26, x0

# Store results
sw      x4, 0(x2)
sw      x26, 4(x2)
sw      x27, 8(x2)
```

➔ Every test environment should implement HALT macro. When this macro is called, operation of DUT is stopped and comparison to the reference results can be performed.

```
    # -----------------------------------------------------------------------
    # HALT
    RV_COMPLIANCE_HALT

RV_COMPLIANCE_CODE_END
```

➔ Addresses used for storing input data.

```
# Input data section.
    .data

test_A1_data:
    .word 0
test_A2_data:
    .word 1
test_A3_data:
    .word -1
test_A4_data:
    .word 0x7FFFFFFF
test_A5_data:
    .word 0x80000000
test_B_data:
    .word 0x0000ABCD
test_C_data:
    .word 0x12345678
test_D_data:
    .word 0xFEDCBA98
test_E_data:
    .word 0x36925814
```

➔ Addresses used for storing results.

```
# Output data section.
RV_COMPLIANCE_DATA_BEGIN

test_A1_res:
    .fill 6, 4, -1
test_A2_res:
    .fill 6, 4, -1
test_A3_res:
    .fill 6, 4, -1
test_A4_res:
    .fill 6, 4, -1
test_A5_res:
    .fill 6, 4, -1
test_B_res:
    .fill 8, 4, -1
test_C_res:
    .fill 1, 4, -1
test_D_res:
    .fill 2, 4, -1
test_E_res:
    .fill 3, 4, -1

RV_COMPLIANCE_DATA_END
```