RISC-V Compliance Test Format Specification

Table of Contents

1. Introduction

1.1. About

This document contains the RISC-V *compliance test pool* structure and *compliance test* format specification which shall be used as a reference document for those who write or are going to write tests for the RISC-V compliance test pool and for those who are going to use the *compliance test pool* in their own compliance test framework.

- It includes, as example, source code listing and detailed description of one *compliance test*

Framework specification which includes description of how the *compliance test suites* are built and used for the appropriate RISC-V configurations is given in the complementary Framework Specification document. This document is made freely available under a [app_cc_by_4.0].

1.2. Intended audience

This document is intended for design and verification engineers who wish to develop new compliance tests and also for those who wish to write or adapt their own test framework.

1.3. Future work

This is a draft document; it partially documents what exists, and partially documents the longer-term goal. As such, this document still under review and its content will change. Its primary aim is to get a long-term stable version of the spec and to give test authors sufficient lead time to prepare test authoring tools and strategies.

1.4. Feedback and how to contribute

Comments on this document should be made through the RISC-V Compliance Task Group mailing list. Proposed changes may be submitted as git pull requests.

You are encouraged to contribute to this repository by submitting pull requests and by commenting on pull requests submitted by other people as described in the `README.md` file in the top level directory.

Don’t forget to add your own name to the list of contributors in the document.

1.4.1. AsciiDoc

This is a structured text format used by this document. Simple usage should be fairly self evident.

- Comprehensive information on the format is on the AsciiDoc website.
- Comprehensive information on the tooling on the AsciiDoctor website.
- You may find this cheat sheet helpful.

1.4.2. Installing tools

To generate the documentation as HTML you need *asciidoctor* and to generate as PDF you need *asciidoctor-pdf*.

- These are the installation instructions for asciidoctor.
- These are the installation instructions for asciidoctor-pdf.

To spell check you need *aspell* installed.

1.4.3. Building the documentation

To build HTML:

```
make html
```

To build PDF:

```
make pdf
```

To build both:

```
make
```

To check the spelling (excludes any listing or code phrases):

```
make spell
```

Any custom words for spell checking should be added to `custom.wordlist`.

1.5. Contributors

This document has been created by the following people (in alphabetical order of surname).

> Allen Baum, Jeremy Bennett, Radek Hajek, Premysl Vaclavik

1.6. Document history

Revision	*Date*	*Author*	*Modification*
1.2.5 Draft	22 Jan 2020	Allen Baum	* removed references to test pool reference doc, mentioned that the framework will generate it * clarified that macros defined in a test may be used in a test * minor clarifications, consistency changes, added page breaks

Revision	*Date*	*Author*	*Modification*
1.2.4 Draft	08 Jan 2020	Allen Baum	* typos fixed * added RVTEST_BASEUPD macro * added explanations for each macro * clarified restrictions on #ifdefs * added comment that test cases with identical conditions should be combined into a single case * documented that test case first parameter should match the #ifdef parameter that precedes it
1.2.3 Draft	02 Dec 2019	Allen Baum	* modified macro names to conformn to riscof naming convention of model specific vs. pre-defined * add more complete list of macros, their uses, parameters, and whether they are required or optional * minor structural changes (moving sentences, renumbering) and typo fixes * clarified impact of debug macros * clarified how SIGUPD and BASEUPD must be used, fixed parameter description
1.2.2 Draft	21 Nov 2019	Allen Baum	* remove section about test taxonomy, binary tests, emulated ops * clarify/fix boundary between test target and framework responsibilities (split test target into test target and test shell) * remove To Be discussed items that have been discussed * remove default case condition; if conditions are unchaged, part of same case * minor grammatical changes related to the above

Revisio n	*Date*	*Author*	*Modification*
1.2.1 Draft	19 Nov 2019	Allen Baum	* spec/TestFormatSpec.adoc: changed the format of the signature to fixed 32b data size only extracted from COMPLIANCE_DATA_BEGIN/END range. * made test suite subdirectories upper case, with sub-extensions camel case * updated example to match most recent riscof implement macros * fix format so Appendix is now in TOC * moved note about multiple test cases in a test closer to definition * fixed cut/paste error in example of test pool * more gramatical fixes, clarifications added * added To Be Discussed items regarding emulated instruction and binary tests * added graphic of test suite/test_pool/test/test_case hierarchy
1.2.1 Draft	12 Oct 2019	Allen Baum	minor grammar, wording, syntax corrections, added detail and clarification from suggestions by Paul Donahue

Revision	*Date*	*Author*	*Modification*
1.2 Draft	12 Sep 2019	Allen Baum	minor grammar, wording, syntax corrections, added detail and clarification Added detail regarding the 2 approaches for test selection: central database, or embedded conditions embedded in macros Added detail of proposed standard macros RVTEST_SIGBASE, RVTEST_SIGUPD, RVTEST_CASE More explanation of spec status in initial *future work* paragraph (i.e. goal, not yet accomplished) Removed many "to Be Discussed items and made them official Removed options, made POR for test selection and standard macros RVTEST_SIGBASE, RVTEST_SIGUPD, RVTEST_CASE Removed prohibition on absolute addresses Clarified which test suites a test should be in where they are dependent on multiple extensions Clarified use of includes and macros (and documented exsiting deviations) Clarified use of YAML files Added detail to description and uses of common compliance test pool reference document
1.1 Draft	15 Feb 2019	Radek Hajek	Appendix A: example assertions update
1.0 Draft	10 Dec 2018	Radek Hajek, Premysl Vaclavik	First version of the document under this file name. Document may contain some segments of the README.adoc from the compatibility reasons.

2. Foreword

The compliance test pool shall become a complete set of compliance tests which will allow developers to build a compliance test suite for any legal RISC-V configuration. The compliance tests will be very likely written by various authors and therefore it is very important to define the compliance test pool structure and compliance test form, which will be obligatory for all tests. Unification of tests will guarantee optimal compliance test pool management and also better quality and readability of the tests. Last but not least, it will simplify the process of adding new tests into the existing compliance test pool and the formal revision process.

3. Vocabulary

3.1. The compliance test

The compliance test is a nonfunctional testing technique which is done to validate whether the system developed meets the prescribed standard or not. In this particular case the golden reference is the RISC-V ISA standard.

For purpose of this document we understand that the compliance test is a single test which represents the minimum test code that can be compiled and run. It is written in assembler code and its product is a *test signature*. A compliance test may consist of several *test cases*.

3.2. The RISC-V compliance test pool

The RISC-V compliance test pool consists of all approved *compliance tests* that can be assembled by the test framework, forming the *compliance test suite*. The RISC-V compliance test pool must be test target independent (so, should correctly run on any compliant target). Note that this nonfunctional testing is not a substitute for verification or device test.

3.3. The RISC-V compliance test suite

The RISC-V compliance test suite is a group of tests selected from the *compliance test pool* to test compliance for the specific RISC-V configuration. Test results are obtained in the form of a *test suite signature*. Selection of tests is performed based on the target’s asserted configuration, and the spec, Execution Environment or platform requirements. Compliant processor or processor models shall exhibit the same test suite signature as the golden reference test suite signature for the specific configuration being tested.

3.4. The test case

A *test case* is part of the compliance test that tests just one feature of the specification.

```
Note: a single test can contain multiple test cases, each of which can have its own
test inclusion condition (as defined by the cond_str parameter of the RVTEST_CASE
macro.
```

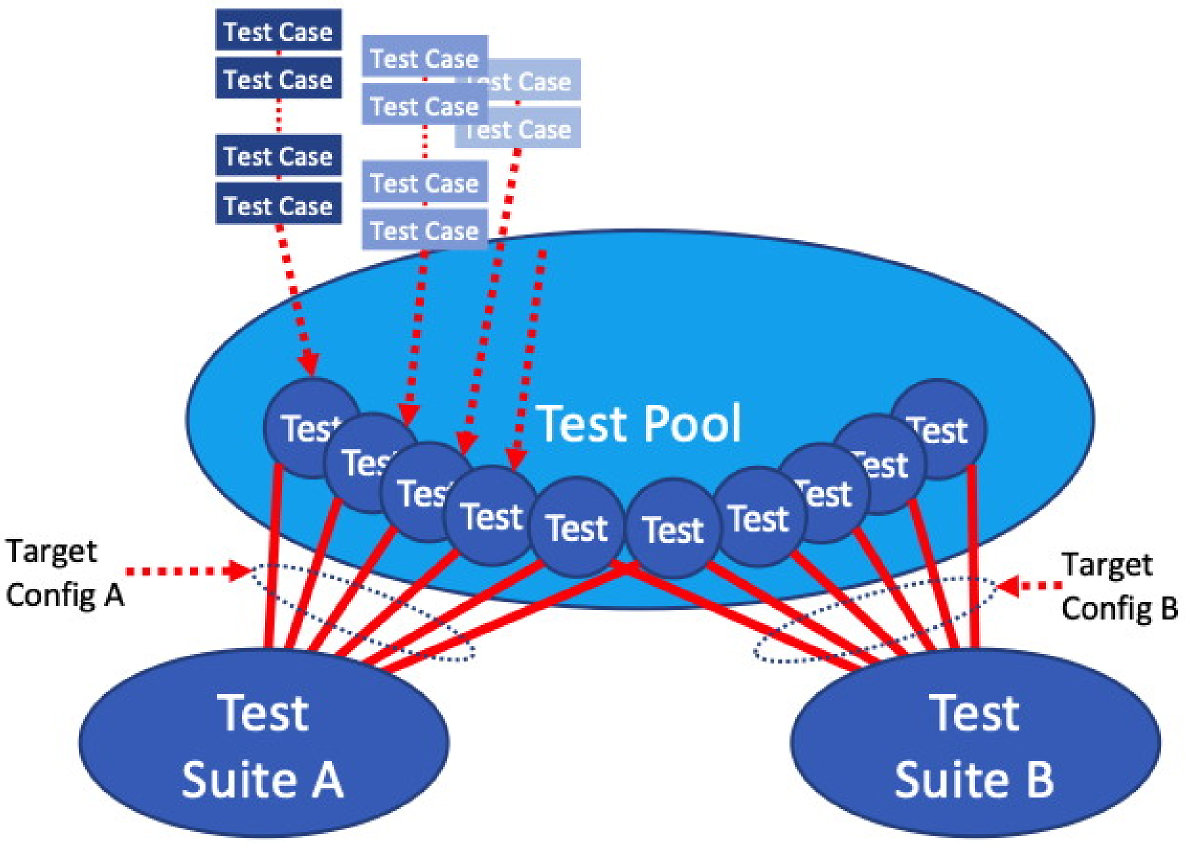

Figure 1. Test Suite, Test_Pool, Test, Test_Case relationship

3.5. The test case signature

The *test case signature* is represented by single or multiple values. Values are written to memory at the address starting at the address specified by the RVMODEL_DATA_BEGIN and ending at RVMODEL_DATA_END. Signatures can be generated most easily using the RVTEST_SIGUPD macro.

3.6. The test signature

The *test signature* is a characteristic value which is generated by the compliance test run. The *test signature* may consist of several *test case signatures*, prefixed with a separate line containing the name of the test and a unique value indicating its version (e.g. git checkin hash). The test target is responsible for extracting values from memory and properly formatting them, using metadata provided to it by the framework using the RVMODEL_DATA_BEGIN and RVMODEL_DATA_END macros. Test case signature values are written one per line, starting with the most-significant byte on the left-hand side with the format <hex_value> where the length of value will be 32 bits (so 8 characters), regardless of the actual value length computed by the test.

3.7. The test suite signature

The *test suite signature* is defined as a set of *test signatures* valid for given *compliance test suite*. It represents the test signature of the particular RISC-V configuration selected for the compliance test.

3.8. The target shell

The *target shell* is the software and hardware environment around the *test target* that enables it to communicate with the framework, including assembling and linking tests, loading tests into memory, executing tests, and extracting the signature. The input to the *target shell* is a .S *compliance test* file, and the output is a *test signature*.

3.9. The test target

The *test target* can be either a RISC-V Instruction Set Simulator (ISS), a RISC-V emulator, a RISC-V RTL model running on an HDL simulator, a RISC-V FPGA implementation or a physical chip. Each of the target types offers specific features and represents specific interface challenges. It is a role of the *target shell* to handle different targets while using the same *compliance test pool* as a test source.

3.10. The RISC-V processor (device) configuration

The RISC-V ISA specification allows many optional instructions, registers, and other features. Production directed targets typically have a fixed subset of available options. A simulator, on the other hand, may implement all known options which may be constrained to mimic the behavior of the RISC-V processor with the particular configuration. It is a role of the Compliance Test Framework to build and use the *compliance test suite* suitable for the selected RISC-V configuration.

3.11. The compliance test framework

The *compliance test framework* selects and configures the *compliance test suite* from the *compliance test pool* for the selected *test target* based on both the specific architectural choices made by an implementation and those required by the Execution Environment It causes the *target shell* to build, execute, and report a signature. The *compliance test framework* then compares reported signatures, inserts test part names and version numbers and summarizes differences (or lack of them) into a RISC-V compliance report. The primary role of the well-defined *compliance test pool* structure is to provide the tests in a form suitable for the Compliance Test Framework selection engine.

4. Compliance test pool

4.1. Test pool structure

The structure of *compliance tests* in the *compliance test pool* shall be based on defined RISC-V extensions and privileged mode selection. This will provide a good overview of which parts of the ISA specification are already covered in the *compliance test suite*, and which tests are suitable for certain configurations. The compliance test pool has this structure:

```
compliance-tests-suite (root)
|-- <architecture>_<mode>/<feature(s)>, where
<architecture> is [ RV32I | RV64I | RV32E ]
<mode> is [ M | MU | MS | MSU ], where
   M   Machine      mode tests - tests execute in M-mode only
   MU  Machine/User mode tests - tests execute in both M- & U-modes (S-mode may exist)
   MS  Machine/Supv mode tests - tests execute in both M- & S-modes (not U-mode)
   MSU All          mode tests - tests execute in all of M-, S-, & U-Modes
<feature(s)> are the lettered extension [A | B | C | M ...] or subextension [Zifencei
| Zam | ...] when the tests involve extensions, or more general names when tests cut
across extension definitionss (e.g. Priv, Interrupt, Vm). The feature string consists
of an initial capital letter, followed by any further letters in lower case.
```

Note that this structure is for organizational purposes, not functional purposes, although full test names will take advantage of it.

Tests that will be executed in different modes, even if the results are identical, should be replicated in each mode directory, e.g. RV32I_M/, RV32I_MS/, and RV32I_MU/. These tests are typically those involving trapping behavior, e.g load, store, and privilged ops.

4.2. Test naming

The naming convention of a single test:

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

- *test objective* – an aspect that the test is focused on. A test objective may be an instruction for ISA tests (ADD, SUB, …), or a characteristic covering multiple instructions, e.g. exception event (misaligned fetch, misalign load/store) and others.
- *test number* – number of the test. It is expected that multiple tests may be specified for one test objective. We recommend to break down complex tests into a set of small tests. A simple rule of thumb is one simple test objective = one simple test. The code becomes more readable and the test of the objective can be improved just by adding *test cases*. The typical example are instruction tests for the F extension.
- A test name shall not include an ISA category as part of its name (i.e. the directory, subdirectory names).
  Experience has shown that including ISA category in the test name leads to very long test

names. Instead, we have introduced the test pool structure where the full name is composed of the test path in the test pool structure and the simple test name.
Since full names can be reconstructed easily it is not necessary to include the path in test names.

4.3. The test structure of a compliance test

All tests shall use a signature approach. Each test shall be written in the same style, with defined mandatory items. There are both pre-defined and model-specific macros which shall be used in every test to guarantee their portability. In addition, there are both pre-defined and model specific macros that are not required, but may be used in tests.

Required, Pre-defined Macros
RVTEST_ISA(isa_str) // defines the Test Virtual Machine (TVM, the ISA being tested)
- Empty macro to specify the isa required for compilation of the test.
- This is mandated to be present at the start of the test.
RVTEST_CODE_BEGIN // start of code (test) section
- Macro to indicate test code start add and where test startup routine is inserted.
- No part of the code section should precede this macro
RVTEST_CODE_END // end of code (test) section
= Macro to indicate test code end.
- No part of the code section should follow after this macro.
RVTEST_CASE(CaseName, CondStr) // execute this case only if condition in cond_str are met
- CaseName is arbitrary string
- CondStr is evaluated to determine if the test-case is enabled and sets name variable
- CondStr can also define compile time macros required for the test-case to be enabled.
- the test-case must be delimited with an #ifdef CaseName/#endif pair
- the format of CondStr can be found in https://riscof.readthedocs.io/en/latest/cond_spec.html#cond-spec

Required, Model-defined Macros
RVMODEL_DATA_BEGIN // start of output data (signature) section
RVMODEL_DATA_END // end of output data (signature) section
RVMODEL_DATA_SECTION // model defined data area
- contains static input data and intermediate scratch area for the test (e.g. stack)
RVMODEL_HALT // defines model halt mechanism, which starts signature saving

Optional, Pre-defined Macros
RVTEST_SIGBASE(BaseReg,Val) // defines the base register used to update signature values
- Register BaseReg is loaded with value Val
- hidden_offset is initialized to zero
RVTEST_SIGUPD(BaseReg, SigReg [,Value]) // stores sig value, with optional value assertion
- Register Val is stored in mem(reg_Base+hidden_offset)
- hidden_offset is post incremented so repeated uses store signature values sequentially
RVTEST_BASEUPD(BaseReg[oldBase[,newOff]]) // [moves &] updates BaseReg past stored signature
- Register BaseReg is loaded with the oldReg+newOff+hidden_offset
- BaseReg is used if oldBase isn’t specified; 0 is used if newOff isn’t specified
- hidden_offset is re-initialized to 0 afterwards

Optional, Model-defined Macros
RVMODEL_BOOT // contains boot code for model; may include emulation code or trap stub
RVMODEL_IO_INIT // initializes IO for debug output
- this must be invoked if any of the other RV_MODEL_IO_* macros are used
RVMODEL_IO_CHECK // checks IO for debug output
- <needs description of how this is used >
RVMODEL_IO_ASSERT_GPR_EQ(ScrReg, Reg, Value) // debug assertion that GPR should have value
- outputs a debug message if Reg!=Value
- ScrReg is a scratch register used by the output routine; its final value cannot be guaranteed
RVMODEL_IO_WRITE_STR(ScrReg, String) // output debug string, using a scratch register
- outputs the message String - ScrReg is a scratch register used by the output routine; its final value cannot be guaranteed

The test structure of a compliance test shall have the following sections in the order as follows:

1. Header + license (including a specification link, a brief test description and RVTEST_ISA macro))
2. Includes of header files (see Common Header Files section)
3. Test Virtual Machine (TVM) specification
4. Test code between “RVTEST_CODE_BEGIN” and “RVTEST_CODE_END”
5. Input data section, marked with "RVMODEL_DATA_SECTION"
6. Output data section between “RVMODEL_DATA_BEGIN” and “RVMODEL_DATA_END”.

```
Note that there is no a requirement that the code or scratch data sections must be
contiguous in memory, or that they be located before or after data or code sections
(configured by embedded directives recognized by the linker)
```

4.3.1. Common test format rules

There are the following common rules that shall be applied to each *compliance test*:

1. Always use “//” as commentary. “#” should be used only for includes and defines.
2. A test shall be divided into logical blocks (*test cases*) according to the test goals. Test cases are enclosed in an `#ifdef <CaseName>, #endif` pair and begin with the RVTEST_CASE(CaseName,CondStr) macro that specifies the test case name, and a string that defines the conditions under which that *Test case* can be selected for assembly and execution. Those conditions will be collected and used to generate the database which in turn is used to select tests for inclusion in the test suite for this target.
3. Tests should use the RVTEST_SIGBASE(BaseReg,Val) macro to define the GPR used as a pointer to the output signature area, and its initial value. It can be used multiple times within a test to reassign the output area or change the base register. This value will be used by the invocations of the RVTEST_SIGUPD macro.
4. Tests should use the RVTEST_SIGUPD(BaseReg, SigReg, ScratchReg, Value) macro to store signature values using (only) the base register defined in the most recently encountered RVTEST_SIGBASE(BaseReg,Val) macro. Repeated uses will automatically have an increasing

offset that is managed by the macro.

   a. Uses of RVTEST_SIGUPD shall always be preceded sometime in the test case by RVTEST_SIGBASE.
   b. The SIGUPD macro may optionally invoke a test assertion macro (e.g. RVMODEL_IO_ASSERT_GPR_EQ) with an assertion value for debugging, determined by the presence of ScratchReg and Value parameters.
   c. Tests that use SIGUPD inside a loop or in any section of code that will be repeated (e.g. traps) must use the BASEUPD macro between each loop iteration or repeated code to ensure static values of the base and offset don't overwrite older values.

5. When macros are needed for debug purposes, only macros from compliance_model.h shall be used. Note that using this feature shall not affect the signature results of the test run.
6. Test shall not include other tests (e.g. #include "../add.S") to prevent non-complete tests, compilation issues, and problems with code maintenance.
7. Tests and test cases shall be skipped if not required for a specific model test configuration based on test conditions defined in the RVTEST_CASE macro. Tests that are selected may be further configured using variables (e.g. XLEN) which are passed into the tests and used to compile them. In either case, those conditions and variables are derived from the YAML specification of the device and execution environment that are passed into the framework. The flow is to run a compliance test suite built by the *Compliance Test Framework* from the *compliance test pool* to determine which tests and test cases to run.
8. Tests shall not depend on tool specific features. For example, tests shall avoid usage of internal GCC macros (e..g. __*risc_xlen*), specific syntax (char 'a' instead of 'a) or simulator features (e.g. tohost) etc.
9. Each test shall be ended by the (target specific) "RVMODEL_HALT" macro. Depending on branches in the test, there may be more than one instance of this in a test.
10. Macros defined outside of a test shall only be defined in specific predefined header files (see *Common Header Files* below), and once they are in use, they may be modified only if the function of all affected tests remains unchanged. It is acceptable that macros use may lead to operand repetition (register X is used every time).
    - The aim of this restriction is to have test code more readable and to avoid side effects which may occur when different contributors will include new *compliance tests* or updates of existing ones in the *compliance test pool*. This measure results from the negative experience, where the *compliance test suite* could be used just for one target while the compliance test code changes were necessary to have it also running for other targets.

4.3.2. Common Header Files

Each test shall include only the following header files:

1. *compliance_model.h* – defines target-specific macros, both required and optional: (e.g. RVMODEL_xxx)
2. *compliance_test.h* – defines pre-defined test macros both required and optional: (e.g. RVTEST_xxx)

Adding new header files is forbidden. It may lead to macro redefinition and compilation issues. Macros maybe defined and used inside a test, as they will not be defined outside that specific test. Assertions will generate code that reports assertion failures (and optionally successes?) only if enabled by the framework. In addition, the framework may collect the assertion values and save them as a signature output file if enabled by the framework.

```
Note that there are other legacy header files (aw_test_macros.h, riscv_test.h,
encoding.h, ..) already included and used in existing tests that.
These header files shall not be modified for testing purposes. New tests should must
either move them into compliance_test.h or not use them.
```

4.3.3. Framework Requirements

The framework will import files that describe

- the implemented, target-specific configuration parameters in YAML format
- the required, platform-specific configuration parameters in YAML format

The framework will generate intermediate files, including a Test Database YAML file that selects tests from the test pool to generate a test suite for the target.

The framework will also invoke the *target shell* as appropriate to cause tests to be built, loaded, executed, and results reported.

The YAML files define both the values of those conditions and values that can be used by the framework to configure tests (e.g. format of WARL CSR fields). Tests should not have #if, #ifdef, etc. for conditional assembly except those that surround RVMODEL_CASE macros Instead, each of those should be a separate *test case* whose conditions are defined in the common reference document entry for that test and test case number.

Appendix A: Example - ISA test *ADD-01.S*

a) Header and license

```
// RISC-V Compliance Test ADD-01
//
// Copyright (c) 2017, Codasip Ltd.
// Copyright (c) 2018, Imperas Software Ltd. Additions
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//    * Redistributions of source code must retain the above copyright
//      notice, this list of conditions and the following disclaimer.
//    * Redistributions in binary form must reproduce the above copyright
//      notice, this list of conditions and the following disclaimer in the
//      documentation and/or other materials provided with the distribution.
//    * Neither the name of the Codasip Ltd., Imperas Software Ltd. nor the
//      names of its contributors may be used to endorse or promote products
//      derived from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
// ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
// WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
// IN NO EVENT SHALL Codasip Ltd., Imperas Software Ltd. BE LIABLE FOR ANY DIRECT,
// INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
// NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;LOSS OF USE, DATA, OR
// PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
// WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
// POSSIBILITY OF SUCH DAMAGE.
//
// Specification: RV32I Base Integer Instruction Set, Version 2.0
// Description: Testing instruction ADD.
```

b) Includes of header files

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

c) TVM selection

```
// Test Virtual Machine (TVM) used by program.
RVTEST_ISA("RV32M")   //This is a standard macro
```

d) Test code

ISA test is divided into several test cases marked as “A“,“B“,“C“, etc. These test cases distinguish various logical tests. The test uses macros from compliance_io.h for debug purposes.

```
// Test code region.
RVTEST_CODE_BEGIN

   RVMODEL_IO_INIT
   RVMODEL_IO_ASSERT_GPR_EQ(x31, x0, 0x00000000)
   RVMODEL_IO_WRITE_STR(x31, "# Test Begin\n")
```

d.A) Test code - test case A

Test case “A“ focuses on checking corner case values of the ADD instruction. In particular, 0, 1, -1, 0x7FFFFFFF, 0x80000000 with 0, 1, -1, MIN, MAX values.

```
//
-------------------------------------------------------------------------------------------
#ifdef TEST_CASE_A
//  update case variable, describes test, defines framework test requirements
RVTEST_CASE(TEST_CASE_A ,"check ISA:=regex(.*I.*); def TEST_CASE_A=True")
RVMODEL_IO_WRITE_STR(x31, "// Test case A1 - general test of value 0 with 0, 1, -1,
MIN, MAX register values\n");

// Addresses for test data and results
la x1, test_A1_data
RVTEST_SIGBASE(x2, test_A1_res)      //this sets x2 as sig_base and initializes it and
sig_offset

// 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, and assert expected values
RVTEST_SIGUPD(x2, x3, 0x00000000) -- stores x3 at sig_base+sig_offset, updates
sig_offset
RVTEST_SIGUPD(x2, x4, 00000000) -- stores x4 at sig_base+sig_offset, updates
sig_offset
RVTEST_SIGUPD(x2, x5, 0x00000000)
RVTEST_SIGUPD(x2, x6, 0xFFFFFFFF)
RVTEST_SIGUPD(x2, x7, 0xFFFFFFFF)
RVTEST_SIGUPD(x2, x8, 0x80000000)


RVMODEL_IO_WRITE_STR(x31, "// Test case A1 - Complete\n");

// ---------------------------------------------------------------------------
RVMODEL_IO_WRITE_STR(x31, "// Test case A2 - general test of value 1 with 0, 1, -1,
MIN, MAX register values\n");

<similar code to A1>

// ---------------------------------------------------------------------------
RVMODEL_IO_WRITE_STR(x31, "// Test case A3 - general test of value -1 with 0, 1, -1,
MIN, MAX register values\n");

<similar code to A1>

// ---------------------------------------------------------------------------
RVMODEL_IO_WRITE_STR(x31, "// Test case A4 - general test of value 0x7FFFFFFF with 0,
1, -1, MIN, MAX register values\n");

<similar code to A1>

// ---------------------------------------------------------------------------
RVMODEL_IO_WRITE_STR(x31, "// Test case A5 - general test of value 0x80000000 with 0,
1, -1, MIN, MAX register values\n");

<similar code to A1>

#endif
```

d.B) Test code - test case B

Test case “B“ focuses on forwarding between instruction. It means that a result of an instruction is immediately passed to another instruction.

```
// ---------------------------------------------------------------------------
#ifdef TEST_CASE_B
//  update case variable, describes test, defines framework test requirements
RVTEST_CASE(TEST_CASE_B ,"check ISA:=regex(.*I.*); def TEST_CASE_B=True")
RVMODEL_IO_WRITE_STR(x31, "// Test case B - testing forwarding between
instructions\n");

// Addresses for test data and results
la x25, test_B_data
RVTEST_SIGBASE(x26, test_B_res)      //this sets x26 as sig_base and initializes it
and sig_offset

// 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, and assert expected values
RVTEST_SIGUPD(x26, x27, 0x00000001) //store x27 at sig_base+sig_offset, update
sig_offset, assert expected value
RVTEST_SIGUPD(x26, x29, 0x0000ABCE)
RVTEST_SIGUPD(x26, x30, 0x0000ABCF)
RVTEST_SIGUPD(x26, x31, 0x0000ABD0)
RVTEST_SIGUPD(x26, x1,  0x0000ABD1)
RVTEST_SIGUPD(x26, x2,  0x0000ABD2)
RVTEST_SIGUPD(x26, x3,  0x0000ABD3)

RVMODEL_IO_WRITE_STR(x31, "// Test case B - Complete\n");

#endif
```

d.C) Test code - test case C

Test case “C“ focuses on writing to x0. This register is hardwired to the 0 value, so in any RISC-V implementation, it must not be overwritten.

```
// -------------------------------------------------------------------
#ifdef TEST_CASE_C
//  update case variable, describes test, defines framework test requirements
RVTEST_CASE(TEST_CASE_B ,"check ISA:=regex(.*I.*); def TEST_CASE_C=True")
RVMODEL_IO_WRITE_STR(x31, "// Test case C - testing writing to x0\n");

// Addresses for test data and results
la x1, test_C_data
RVTEST_SIGBASE(x2, test_C_res)      //this sets x2 as sig_base and initializes it and
sig_offset

// Load testdata
lw x28, 0(x1)

// Register initialization
li x27, 0xF7FF8818

// Test
add x0, x28, x27

// store results using x2 as a base
RVTEST_SIGUPD(x2, x0, 0x00000000)

RVMODEL_IO_WRITE_STR(x31, "// Test case C - Complete\n");
#endif
```

d.D) Test code - test case D

Test case “D“ focuses on forwarding through x0. This register is hardwired to the 0 value, so a temporary non-zero result must not be passed to another instruction.

```
// ---------------------------------------------------------------------------
#ifdef TEST_CASE_D
//  update case variable, describes test, defines framework test requirements
RVTEST_CASE(TEST_CASE_D ,"check ISA:=regex(.*I.*); def TEST_CASE_D=True")
RVMODEL_IO_WRITE_STR(x31, "// Test case D - testing forwarding throught x0\n");

// Addresses for test data and results
la x1, test_D_data
RVTEST_SIGBASE(x2, test_D_res)      //this sets x2 as sig_base and initializes it and
sig_offset

// Load testdata
lw x28, 0(x1)

// Register initialization
li x27, 0xF7FF8818

// Test
add x0, x28, x27
add x5,  x0,  x0

// store results
RVTEST_SIGUPD(x2, x0, 0x00000000)
RVTEST_SIGUPD(x2, x5, 0x00000000)

RVMODEL_IO_WRITE_STR(x31, "// Test case D - Complete\n");
```

d.E) Test code - test case E

Test case “E“ focuses on ADD with x0. The ADD instruction performs the MOVE operation in that case.

```
// ---------------------------------------------------------------------------
#ifdef TEST_CASE_E
//  update case variable, describes test, defines framework test requirements
RVTEST_CASE(TEST_CASE_E ,"check ISA:=regex(.*I.*); def TEST_CASE_E=True")
RVMODEL_IO_WRITE_STR(x31, "// Test case E - testing moving (add with x0)\n");

// Addresses for test data and results
la x1, test_E_data
RVTEST_SIGBASE(x2, test_E_res)      //this sets x2 as sig_base and initializes it and
sig_offset

// 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, assert expected value
RVTEST_SIGUPD(x2, x4,  0x36925814)
RVTEST_SIGUPD(x2, x26, 0x36925814)
RVTEST_SIGUPD(x2, x27, 0x36925814)

RVMODEL_IO_WRITE_STR(x31, "// Test case E - Complete\n");

#endif
```

Note that because all the test conditions in the above example are identical, they should have been combined into a single test case.

d.F) Test code - section Test End

Every test environment should implement at least one instance of the HALT macro. IMore than one would be implemented in cases where the test has branches or traps that cause it to end in different locations. When the macro is called, operation of DUT is stopped and a comparison to the reference results can be performed.

```
RVMODEL_IO_WRITE_STR(x31, "// Test End\n")
// ------------------------------------------------------------------------------
// HALT
RVMODEL_HALT
RVTEST_CODE_END
```

e) Test code - section Input Data

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
```

f) Test code - section Output Data

Addresses used for storing results.

```
// Output data section.
RVMODEL_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
RVMODEL_DATA_END
```