RISC-V Architecture Test Format Specification

Table of Contents

1. Introduction

1.1. About

This document contains the RISC-V *architectural test pool* structure and *architectural 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 architectural test pool and for those who are going to use the *architectural test*

pool in their own architectural test framework.

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

Framework specification which includes description of how the *architectural 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 architectural 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. Contributors

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

- Allen Baum
- Jeremy Bennett
- Radek Hajek
- Premysl Vaclavik

1.5. Document history

Revision	*Date*	*Author*	*Modification*
1.2.6 Draft	24 September 2020	Neel Gala	Replaced Compliance with Architecture/Architectural. Minor beautification in macro definitions.
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

Revision	*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 architectural test pool shall become a complete set of architectural tests which will allow developers to build an architectural test suite for any legal RISC-V configuration. The architectural tests will be very likely written by various authors and therefore it is very important to define the architectural test pool structure and architectural test form, which will be obligatory for all tests. Unification of tests will guarantee optimal architectural 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 architectural test pool and the formal revision process.

3. Vocabulary

3.1. The architectural test

The architectural 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 architectural 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 architectural test may consist of several *test cases*.

3.2. The RISC-V architectural test pool

The RISC-V architectural test pool consists of all approved *architectural tests* that can be assembled by the test framework, forming the *architectural test suite*. The RISC-V architectural 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 architectural test suite

The RISC-V architectural test suite is a group of tests selected from the *architectural test pool* to test adherence 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 architectural 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.

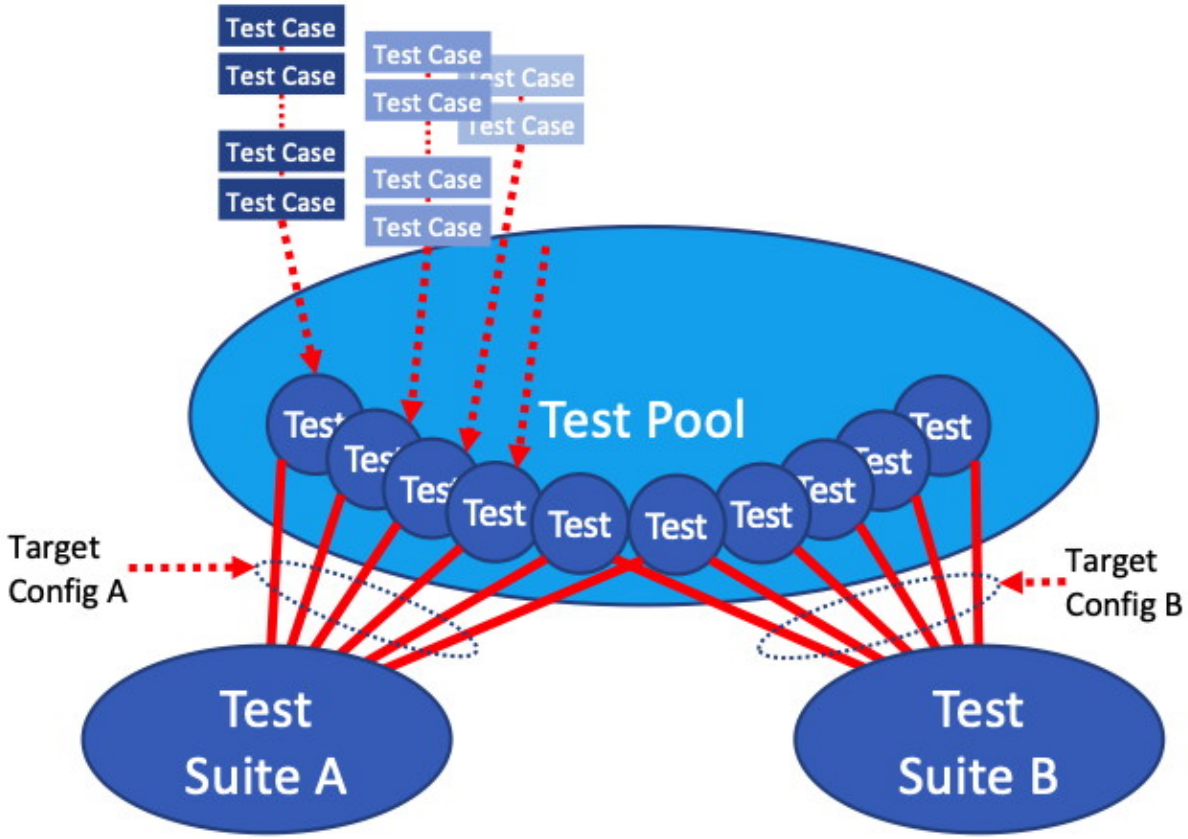

Figure 1. Test Suite, Test_Pool, Test, Test_Case relationship

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 architectural 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. The file should start with values stored at the lowest address of the signature (i.e. from RVMODEL_DATA_BEGIN to RVMODEL_DATA_END). Furthermore, the signature should always begin at a 16-byte (128-bit)

boundary and the size of the signature should be a multiple of 16-bytes (i.e. it should also end at a 16-byte boundary).

3.7. The test suite signature

The *test suite signature* is defined as a set of *test signatures* valid for given *architectural test suite*. It represents the test signature of the particular RISC-V configuration selected for the architectural 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 *architectural 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 *architectural 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 Architectural Test Framework to build and use the *architectural test suite* suitable for the selected RISC-V configuration.

3.11. The architectural test framework

The *architectural test framework* selects and configures the *architectural test suite* from the *architectural 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 *architectural test framework* then compares reported signatures, inserts test part names and version numbers and summarizes differences (or lack of them) into a RISC-V test report. The primary role of the well-defined *architectural test pool* structure is to provide the tests in a form suitable for the Architectural Test Framework selection engine.

4. Architectural test pool

4.1. Test pool structure

The structure of *architectural tests* in the *architectural 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 *architectural test suite*, and which tests are suitable for certain configurations. The architectural test pool has this structure:

```
architectural-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. Assembly macros and test labels

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 for either convenience or debugging purposes.

4.3.1. Required, Pre-defined Macros

These macros are be defined in the file **compilance_test.h** by the author of the test. A significant amount of the framework shall depend on the existence of these 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 test-code section should precede this macro
- this macro includes an initialization routine which pre-loads all the GPRs with unique values (not `0xdeadbeef`). Register t0 and t1 are initialized to point to the labels : `rvtest_code_begin` and `rvtest_code_end` respectively.
- the macros contains a label `rvtest_code_begin` after the above initilization routine to mark the begining of the actual test.

`RVTEST_CODE_END`

- end of code (test) section
- macro to indicate test code end.
- no part of the test-code section should follow after this macro.
- the macro enforces a 16-byte boundary alignment
- the macro also inlcudes the label `rvtest_code_end` which marks the end of the actual test.
- if trap handling is enabled, this macro contains the entire trap handler code required by the test.

`RVTEST_DATA_BEGIN`

- marks the begining of the test data section
- used to provided initialized data regions to be used by the test

- this region starts at a 16-byte boundary
- the start of this is macro can be addressed using the label: `rvtest_data_begin`
- when trap handling is enabled, this macro also includes the following labels :
    1. trapreg_sv: This region is used to save the temporary registers used in the trap-handler code
    2. tramptbl_sv: This region is used to save the contents of the test-target's initial code-section which is overwritten with the necessary trampoline table.
    3. mtvec_save: a double-word region to save the test-target specific mtvec register
    4. mscratch_save: a double-word region to save the test-target specific mscratch register

`RVTEST_DATA_END`

- this macros marks the end of the test input data section.
- the start of this macro can be addressed using the label: `rvtest_data_end`

`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

4.3.2. Required, Model-defined Macros

These macros are be defined by the owner of the test target in the file **model_test.h**. These macros are required to define the signature regions and also the logic required to halt/exit the test.

`RVMODEL_DATA_BEGIN`

- This macro marks the start of signature regions. The test-target should use this macro to create a label to indicate the begining of the signature region. For example : `.globl begin_signature; begin_signature`. This macro must also begin at a 16-byte boundary and must not include anything else.

`RVMODEL_DATA_END`

- This macros marks the end of the signature-region. The test-target must declare any labels required to indicate the end of the signature region. For example : `.globl end_signature; end_signature`. This label must be at a 16-byte boundary. The entire signature region must be included within the RVMODEL_DATA_BEGIN macro and the start of the RVMODEL_DATA_END macro. The RVMODEL_DATA_END macro can also contain other target specific data regions and initializations but only after the end of the signature.

`RVMODEL_HALT`

- This macros must define the test-target halt mechanism. This macro is called when the test is to be terminated either due to completion or dur to unsupported behavior. This macro could also include routines to dump the signature region to a file on the host system which can be used for comparison.

4.3.3. 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 [, Offset])`

- if Offset is present in the arguments, hidden_offset if set to Offset
- Sigreg is stored at hidden_offset[BaseReg]
- 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

`RVTEST_SIGUPD_F(BaseReg, SigReg, FlagReg [, Offset])`

- This macro is used for RV32F and RV64D (where XLEN==FLEN).
- if Offset is present in the arguments, hidden_offset if set to Offset+(XLEN*2)
- SigReg is stored at hidden_offset[BaseReg]
- FlagReg is stored at hidden_offset+XLEN[BaseReg]
- hidden_offset is post incremented so repeated uses store signature values sequentially

4.3.4. Optional, Model-defined Macros

`RVMODEL_BOOT`

- contains boot code for the test-target; may include emulation code or trap stub. If the test-target enforces alignment or value restrictions on the mtvec csr, it is required that this macro sets the value of mtvec to a region which is readable and writable by the machine mode. May include code to copy the data sections from boot device to ram. Or any other code that needs to be run prior to running the tests.

`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_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
- Can be used to help debug what tests have passed/failed

`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

`RVMODEL_SET_MSW_INT`

- This macro needs to include a routine to set the machine software interrupt.
- Currently the test forces an empty macro if a target does not declare this. Future tests may change this.

`RVMODEL_CLEAR_MSW_INT`

- This macro needs to include a routine to clear the machine software interrupt.
- Currently the test forces an empty macro if a target does not declare this. Future tests may change this.

`RVMODEL_CLEAR_MTIMER_INT`

- This macro needs to include a routine to clear the machine timer interrupt.
- Currently the test forces an empty macro if a target does not declare this. Future tests may change this.

`RVMODEL_CLEAR_MEXT_INT`

- This macro needs to include a routine to clear the machine external interrupt.
- Currently the test forces an empty macro if a target does not declare this. Future tests may change this.

4.4. Required labels

The test must define a `rvtest_entry_point` label to indicate the location to be used by the linker as the entry point in the test. Generally, this would be before the `RVMODEL_BOOT` macro and should belong to the `text.init` section.

4.5. The test structure of an architectural test

All tests shall use a signature approach. Each test shall be written in the same style, with defined mandatory items. The test structure of an architectural 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 "RVTEST_DATA_BEGIN" and "RVTEST_DATA_END".
6. Output data section between “RVMODEL_DATA_BEGIN” and “RVMODEL_DATA_END”.

Note

Note that there is no 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)

4.5.1. Common test format rules

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

1. Always use “//” as commentary. “#” should be used only for includes and defines.
2. As part of the initialization code, all GPRs are preloaded with unique predefined values (which is not `0xdeadbeef`). However, t0 is initialized with `rvtest_code_begin` and t1 is initialized with `rvtest_data_begin`.
3. The signature section of every test is pre-loaded with the word `0xdeadbeef`
4. The signature region should always begin at a 16-byte boundary
5. 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.
6. 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.
7. 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. 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.
8. When macros are needed for debug purposes, only macros from *model_test.h* shall be used. Note that using this feature shall not affect the signature results of the test run.

9. Test shall not include other tests (e.g. #include "../add.S") to prevent non-complete tests, compilation issues, and problems with code maintenance.
10. 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 an architectural test suite built by the *Architectural Test Framework* from the *architectural test pool* to determine which tests and test cases to run.
11. 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.
12. A test will end by either jumping to or implicitly reaching the `RVTEST_CODE_END` macro (i.e. rvtest_code_end label). The `RVTEST_CODE_END` macro is always followed by the `RVMODEL_HALT` macro.
13. 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 *architectural tests* or updates of existing ones in the *architectural test pool*. This measure results from the negative experience, where the *architectural test suite* could be used just for one target while the architectural test code changes were necessary to have it also running for other targets.
14. All contents of the signature region must always be initialized to `0xdeadbeef`.
15. The result of no operation should be stored in the signature even though not register has been altered.
16. Pseudo ops other than `li` and `la` which can map to multiple standard instruction sequences should not be used.
17. The actual test-section of the assembly must always start with the `RVTEST_CODE_BEGIN` which contains a routine to initialize the registers to specific values.

4.5.2. Common Header Files

Each test shall include only the following header files:

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

The inclusion of the *arch_test.h* should always occur after the *model_test.h* file.

Important points to be noted regarding header files :

1. Adding new header files is forbidden in the test. It may lead to macro redefinition and compilation issues.

2. Macros maybe defined and used inside a test, as they will not be defined and used outside that specific test.

4.5.3. 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*

1) Header to inlcude comments

```
#This assembly file tests the add instruction of the RISC-V I extension for the add
covergroup.
```

2) Includes of header files

```
#include "model_test.h"
#include "arch_test.h"
```

3) Set the TVM of the test

```
RVTEST_ISA("RV32I")
```

4) Test target specifc boot-code

```
RVMODEL_BOOT
```

5) Start of GPR intialization routine and test code

```
RVTEST_CODE_BEGIN
```

6) Define the RVTEST_CASE string and conditions

```
#ifdef TEST_CASE_1

// this test is meant for devices implementing rv32I extension and requires enabling
the compile
// macro TEST_CASE_1. This test will contribute to the "add" coverage label.

RVTEST_CASE(0,"//check ISA:=regex(.*32.*);check ISA:=regex(.*I.*);def
TEST_CASE_1=True;",add)
```

7) Initialize pointer to the signature region

```
RVTEST_SIGBASE( x16,signature_x16_1) // x16 will point to signature_x16_1 label in the
signature region
```

8) Define the test cases

```
TEST_RR_OP(add, x9, x4, x6, 0x80000005, 0x80000000, 0x00000005, x16, 0, x24)
TEST_RR_OP(add, x5, x5, x14, 0xffffffeff, 0x00000000, 0xffffffeff, x16, 4, x24)
...
...
```

9) Change signature base register

```
// this will change the signature base register to x3. x3 will not point to
signature_x3_0 in
// the signature region
RVTEST_SIGBASE( x3,signature_x3_0)

// continue with new test cases ..
TEST_RR_OP(add, x4, x24, x27, 0x55555955, 0x00000400, 0x55555555, x3, 0, x5)
...
...
```

10) End the test and halt the test-target

```
RVTEST_CODE_END
RVMODEL_HALT
```

11) Create test input data section

```
RVTEST_DATA_BEGIN
rvtest_data:
.word 0xbabecafe
RVTEST_DATA_END
```

12) Create pre-loaded signature region

```
RVMODEL_DATA_BEGIN
.align 4

signature_x16_0:
    .fill 0*(XLEN/32),4,0xdeadbeef


signature_x16_1:
    .fill 16*(XLEN/32),4,0xdeadbeef


signature_x3_0:
    .fill 86*(XLEN/32),4,0xdeadbeef

#ifdef rvtest_mtrap_routine

mtrap_sigptr:
    .fill 64*(XLEN/32),4,0xdeadbeef

#endif

#ifdef rvtest_gpr_save

gpr_save:
    .fill 32*(XLEN/32),4,0xdeadbeef

#endif

RVMODEL_DATA_END
```