README.md

riscv-tests

About

This repository hosts unit tests for RISC-V processors.

Building from repository

We assume that the RISCV environment variable is set to the RISC-V tools install path, and that the riscv-gnu-toolchain package is installed.

```
$ git clone https://github.com/riscv/riscv-tests
$ cd riscv-tests
$ git submodule update --init --recursive
$ autoconf
$ ./configure --prefix=$RISCV/target
$ make
$ make install
```

The rest of this document describes the format of test programs for the RISC-V architecture.

Test Virtual Machines

To allow maximum reuse of a given test, each test program is constrained to only use features of a given *test virtual machine* or TVM. A TVM hides differences between alternative implementations by defining:

- The set of registers and instructions that can be used.
- Which portions of memory can be accessed.
- The way the test program starts and ends execution.
- The way that test data is input.
- The way that test results are output.

The following table shows the TVMs currently defined for RISC-V. All of these TVMs only support a single hardware thread.

TVM Name	Description
`rv32ui`	RV32 user-level, integer only
`rv32si`	RV32 supervisor-level, integer only
`rv64ui`	RV64 user-level, integer only
`rv64uf`	RV64 user-level, integer and floating-point
`rv64uv`	RV64 user-level, integer, floating-point, and vector
`rv64si`	RV64 supervisor-level, integer only
`rv64sv`	RV64 supervisor-level, integer and vector

A test program for RISC-V is written within a single assembly language file, which is passed through the C preprocessor, and all regular assembly directives can be used. An example test program is shown below. Each test program should first include the `riscv_test.h` header file, which defines the macros used by the TVM. The header file will have different contents depending on the target environment for which the test will be built. One of the goals of the various TVMs is to allow the same test program to be compiled and run on very different target environments yet still produce the same results. The following table shows the target environment currently defined.

Target Environment Name	Description
`p`	virtual memory is disabled, only core 0 boots up
`pm`	virtual memory is disabled, all cores boot up
`pt`	virtual memory is disabled, timer interrupt fires every 100 cycles
`v`	virtual memory is enabled

Each test program must next specify for which TVM it is designed by including the appropriate TVM macro, `RVTEST_RV64U` in this example. This specification can change the way in which subsequent macros are interpreted, and supports a static check of the TVM functionality used by the program.

The test program will begin execution at the first instruction after `RVTEST_CODE_BEGIN`, and continue until execution reaches an `RVTEST_PASS` macro or the `RVTEST_CODE_END` macro, which is implicitly a success. A test can explicitly fail by invoking the `RVTEST_FAIL` macro.

The example program contains self-checking code to test the result of the add. However, self-checks rely on correct functioning of the processor instructions used to implement the self check (e.g., the branch) and so cannot be the only testing strategy.

All tests should also contain a test data section, delimited by `RVTEST_DATA_BEGIN` and `RVTEST_DATA_END`. There is no alignment guarantee for the start of the test data section, so regular assembler alignment instructions should be used to ensure desired alignment of data values. This region of memory will be captured at the end of the test to act as a signature from the test. The signature can be compared with that from a run on the golden model.

Any given test environment for running tests should also include a timeout facility, which will class a test as failing if it does not successfully complete a test within a reasonable time bound.

```
#include "riscv_test.h"

RVTEST_RV64U        # Define TVM used by program.

# Test code region.
RVTEST_CODE_BEGIN   # Start of test code.
        lw      x2, testdata
        addi    x2, 1         # Should be 42 into $2.
        sw      x2, result    # Store result into memory overwriting 1s.
        li      x3, 42        # Desired result.
        bne     x2, x3, fail  # Fail out if doesn't match.
        RVTEST_PASS           # Signal success.
fail:
        RVTEST_FAIL
RVTEST_CODE_END     # End of test code.

# Input data section.
# This section is optional, and this data is NOT saved in the output.
.data
        .align 3
testdata:
        .dword 41

# Output data section.
RVTEST_DATA_BEGIN   # Start of test output data region.
        .align 3
result:
```

```
        .dword -1
RVTEST_DATA_END     # End of test output data region.
```

User-Level TVMs

Test programs for the `rv32u*` and `rv64u*` TVMs can contain all instructions from the respective base user-level ISA (RV32 or RV64), except for those with the SYSTEM major opcode (syscall, break, rdcycle, rdtime, rdinstret). All user registers (pc, x0-x31, f0-f31, fsr) can be accessed.

The `rv32ui` and `rv64ui` TVMs are integer-only subsets of `rv32u` and `rv64u` respectively. These subsets can not use any floating-point instructions (major opcodes: LOAD-FP, STORE-FP, MADD, MSUB, NMSUB, NMADD, OP-FP), and hence cannot access the floating-point register state (f0-f31 and fsr). The integer-only TVMs are useful for initial processor bringup and to test simpler implementations that lack a hardware FPU.

Note that any `rv32ui` test program is also valid for the `rv32u` TVM, and similarly `rv64ui` is a strict subset of `rv64u`. To allow a given test to run on the widest possible set of implementations, it is desirable to write any given test to run on the smallest or least capable TVM possible. For example, any simple tests of integer functionality should be written for the `rv64ui` TVM, as the same test can then be run on RV64 implementations with or without a hardware FPU. As another example, all tests for these base user-level TVMs will also be valid for more advanced processors with instruction-set extensions.

At the start of execution, the values of all registers are undefined. All branch and jump destinations must be to labels within the test code region of the assembler source file. The code and data sections will be relocated differently for the various implementations of the test environment, and so test program results shall not depend on absolute addresses of instructions or data memory. The test build environment should support randomization of the section relocation to provide better coverage and to ensure test signatures do not contain absolute addresses.

Supervisor-Level TVMs

The supervisor-level TVMs allow testing of supervisor-level state and instructions. As with the user-level TVMs, we provide integer-only supervisor-level TVMs indicated with a trailing `i`.

History and Acknowledgements

This style of test virtual machine originated with the T0 (Torrent-0) vector microprocessor project at UC Berkeley and ICSI, begun in 1992. The main developers of this test strategy were Krste Asanovic and David Johnson. A precursor to `torture` was `rantor` developed by Phil Kohn at ICSI.

A variant of this testing approach was also used for the Scale vector-thread processor at MIT, begun in 2000. Ronny Krashinsky and Christopher Batten were the principal architects of the Scale chip. Jeffrey Cohen and Mark Hampton developed a version of torture capable of generating vector-thread code.

Releases

No releases published

Packages

No packages published

Contributors 62

+ 51 contributors

Languages

C 48.8% Assembly 33.2% Python 14.2% Perl 2.0% Makefile 1.4% Scala 0.4%