SHAKTI

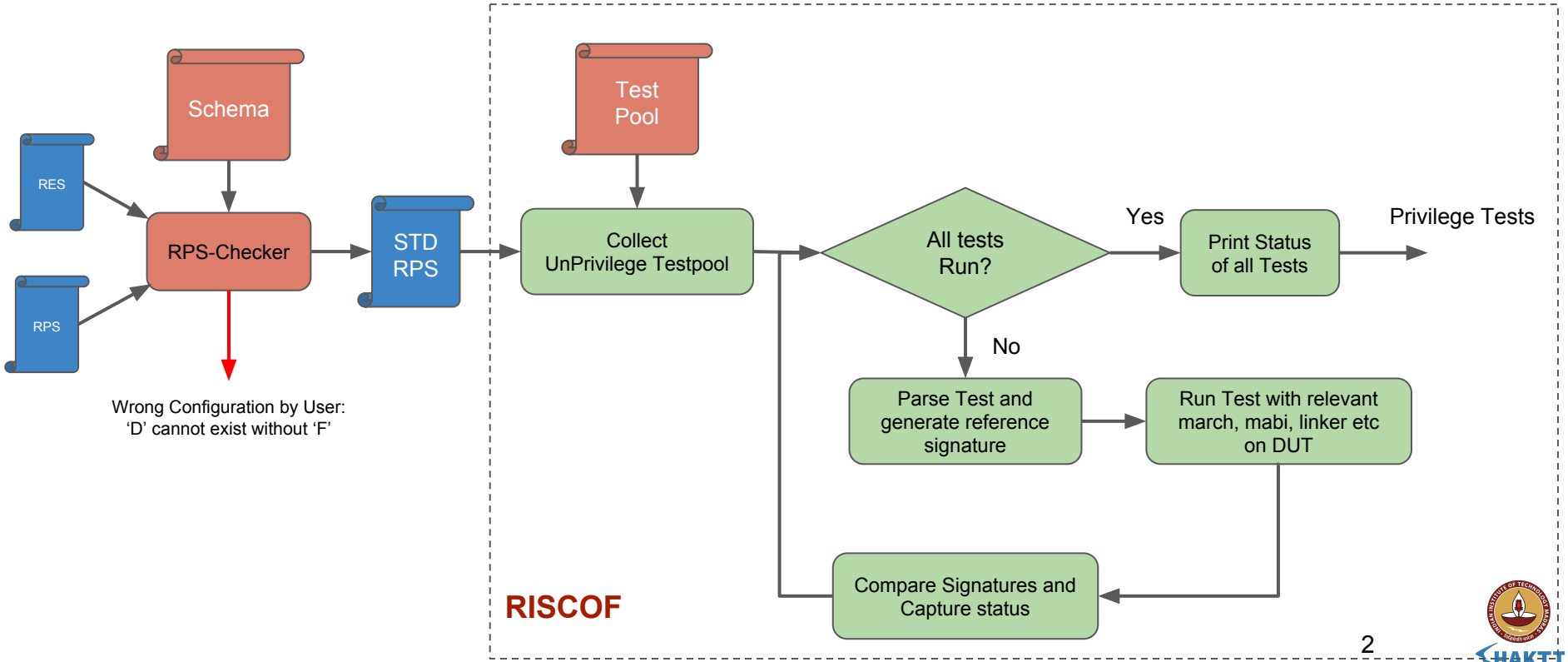
YAML Based Compliance Framework

Proposal by

SHAKTI Group | CSE Dept | RISE Lab | IIT Madras

Overall Flow

RES: RISC-V Env Spec
RPS: RISC-V Platform Spec
RISCOF: RISC-V COmpliance Framework



RPS - RISC-V Platform Spec

1. YAML syntax inspired by **RIFFL from Nikhil**
2. Field types are different
3. Schema check is automated using cerberus library for Python3
4. Added **WARL** fields proposed by **Allen** as a template
5. Stronger type and value checks than RIFFL - Personal opinion
6. Default values for all fields specified

RES: RISCV Env Spec

1. Includes target specific environment variables like
    a. GCC
    b. Linker Script
    c. Signature File
    d. Executable
    e. Pre-sim commands
    f. Post-sim commands
    g. Platform specific headers - for boot up code
2. No checks on this - to be used as provided by the user

RIPS-Checker

1. YAML syntax checker for user input
2. Generates an error for wrong configurations
3. Generates a Standard RPS YAML file with all fields defined - default values for those not defined by the User

Test Pool - Format

- Single Directory containing all the tests
  - No hierarchy of directories required
- The segregation is done as a list in the python framework itself
- Example:

```
Test_pool = [

      # Test-Name , MARCH , Conditions to enable test
      ['M-MUL-01','rv32im', 'M in ISA'],
      ['M-REM-01','rv32im', 'M in ISA'],
      ['M-REMU-01','rv32im', 'M in ISA'],
      ['I-ADD-01' ,'rv32i', 'I in ISA'],
      ['I-ADDI-01' ,'rv32i', 'I in ISA'],
      ['I-AND-01' ,'rv32i', 'I in ISA'],
      ['I-ANDI-01' ,'rv32i', 'I in ISA'],  ..............  ]
```

- Conditions fields can be multiple and is used by RISCOF to pull the test for the DUT

Reference Signature Generation

- Requires the TEST to have the following format:
  a. **#ifdef TEST_PART_1**: This is prefixed before each test part within an assembly file which could be optional. For a assembly file where all parts are to be run a single #ifdef is required
  b. **RVTEST_PART_RUN(1, "MSTATUS_MPP>distinct>modes:unchanged")**: This is an empty macro with 2 arguments:
     i. First Argument: part number. Should be equal to the previous RVTEST_PART_START argument
     ii. Second Argument: String defining the condition for which this test-part should be enabled. Hierarchical dictionary elements are separated using **'>'**. Value of the field is the test after **':'**. RISCOF parses the second argument, checks if the value against the STD-RPS. If validated, then RISCOF will define the TEST_PART_1 during compile else the undefined.
     iii. The signature is not updated for parts which are not enabled.
     iv. This macro can be dropped if the test-part needs to be run by default on any implementation spec.
  c. **RVTEST_IO_ASSERT_GPR_EQ**: has to be specified for each signature update. RISCOF parses this macro to create a reference signature of the test
  d. **#endif**: To indicate end of a test-part

Reference Signature Generation

- Requires the TEST to have the following format:
  g. **Requires signatures to not have holes**. If there are holes insert dummy RVTEST_IO_ASSERT_GPR_EQ to indicate that.

```
#include "test macros.h"

# Test Virtual Machine (TVM) used by program.
RV COMPLIANCE RV32M

# Test code region
RV COMPLIANCE CODE BEGIN

  #ifdef TEST PART 1
    RVTEST PART START(1)
    # -------------------------------------------------------------------------
    RVTEST IO WRITE STR(x31, "# Test part A1 - general test of value 0 with 0, 1, -

    # Addresses for test data and results
    la      x1, test A1 data
    la      x2, test A1 res

    # Load testdata
    lw      x3, 0(x1)

    # Test
    andi    x4, x3, 1

    # Store results
    sw      x3, 0(x2)
    sw      x4, 4(x2)

    //
    // Assert
    //
    RVTEST IO CHECK()
    RVTEST IO ASSERT GPR EQ(x2, x3, 0x00000000)
    RVTEST IO ASSERT GPR EQ(x2, x4, 0x00000000)

    RVTEST PART END(1)
  #endif
 # -------------------------------------------------------------------------
    # HALT
    RV COMPLIANCE HALT
```

```
    //-------------------------------------------------------------------------------------------
    // This test will be run on DUTs whose MPP WARL encoding follows the distinct:unchanged behavior
    #ifdef TEST PART 2
      RVTEST PART START(2)
      RVTEST PART RUN(2, "MSTATUS MPP>distinct>modes:unchanged")

      la x2, test part2 res

      // write legal
      li t2, MSTATUS MPP
      not t2, t2
      csrr t5, mstatus
      and t5, t5, t2
      li t3, LEGAL << MSTATUS MPP INDEX
      or t5, t5, t3
      csrw mstatus, t5
      csrr t2, mstatus
      // write illegal
      li t2, MSTATUS MPP
      not t2, t2
      csrr t5, mstatus
      and t5, t5, t2
      li t3, ILLEGAL << MSTATUS MPP INDEX
      or t5, t5, t3
      csrw mstatus, t5

      // check for legal
      csrr t2, mstatus
      srli t2, t2, MSTATUS MPP INDEX
      andi t2, t2, 3

      sw t2, 0(x2)

      RVTEST IO ASSERT GPR EQ(x31, t2, LEGAL)

      RVTEST PART END(2)
    #endif
```


SHAKTI

Current Status:

- RV32I tests from compliance have been ported to this framework
  - Minimal changes required for porting
- RV32IM tests from compliance have been ported to this framework
  - Had to convert the 'TEST_CASE' macro to the format followed in rv32i tests
  - Minimal changes required for porting
  - Also changes the test names to follow the current rv32i standard - prefix with M and post-fix with test-number
- WARL tests:
  - Simple test for MPP to demonstrate how such tests can be ported to RISCOF

TODO:

- Other 32-bit tests to be ported
- USER environment inputs need to be more robust to handle different pre-sim and post-sim runs.
  - Pre-sim might include configuring a simulator based on the STD-YAML file. - riscvOVPsim?
- Need to port other simulators to check stability

WARL Resolvers:

2 types of resolvers:

- Exhaustive Resolvers:
  - Find all legal values of the field based on user inputs
  - Find all illegal values = ^(legal values) within the range of the field
  - Run the test based for all possible combinations of legal and illegal values
  - Suitable for small sized WARL fields : 2-5 bits wide.
  - Eg: MPP, MISA fields, etc
- Non-Exhaustive Resolvers:
  - Capture legal values from the user input
  - Create a small subset of illegal values
  - Eg: MTVEC: distinct(0x10000, 0x20000, unchanged)
    - Here legal_val = [0x1000, 0x2000]
    - Illegal_val = [0x3000] - randomly chosen value which is not legal and within the max value of the bit-field.
    - Run 2 scenarios: legal=0x1000 and illegal=0x3000 ; legal=0x2000 and illegal=0x3000
  - Suitable for large bit-width fields which are represented as ranges or bitmasks
  - Eg: MTVEC-BASE