RISC-V RVFI-DII 验证协议介绍

RISC-V 常见验证协议，框架介绍

Mingzhu Yan

2025-07-16

PLCT Lab

Outline

1. 背景

1.1 SAIL MODEL 需要一个更好的 trace/log 输出格式

Towards a defined log/trace output from the Sail model #545

Open

rsnikhil (Rishiyur S. Nikhil) opened on Sep 9, 2024 — Collaborator

This first comment just describes the issue(s). Follow-up comments discuss possible solutions.

Currently the Sail model writes out a log file in ASCII format. Issues:

- Logs can become very large (a binary format would be much smaller)
- Downstream tools (verification, trace-driven simulation, ...) have to parse ASCII text to recreate the log data
- We haven't been precise about defining a log format, and we update it as needed in an ad hoc fashion. As a result downstream tools break when we make such changes.

https://github.com/riscv/sail-riscv/issues/545

1.2 SAIL 目前基于纯文本的日志格式

- SAIL 在运行时使用纯文本打印 trace/log 信息到 stdout
- 日志主要包括三种类型
  - 寄存器读写
  - 内存读写
  - 正在运行的指令

```
 8  CSR mstatus (0x300) <- 0x0000000A00000000
 9  CSR misa (0x301) <- 0x800000000034112F
10  CSR mcause (0x342) <- 0x0000000000000000
11  mem[X,0x0000000000001000] -> 0x0297
12  mem[X,0x0000000000001002] -> 0x0000
13  [0] [M]: 0x0000000000001000 (0x00000297) auipc x5, 0x0
14  x5 <- 0x0000000000001000
15
16  mem[X,0x0000000000001004] -> 0x8593
17  mem[X,0x0000000000001006] -> 0x0202
18  [1] [M]: 0x0000000000001004 (0x02028593) addi x11, x5, 0x20
19  x11 <- 0x0000000000001020
20
21  clint mtime 0x0000000000000001 (mip.MTI <- 0b1, mip.STI <- 0b1)
22  mem[X,0x0000000000001008] -> 0x2573
23  mem[X,0x000000000000100A] -> 0xF140
24  [2] [M]: 0x0000000000001008 (0xF1402573) csrrs x10, mhartid, x0
25  CSR mhartid (0xF14) -> 0x0000000000000000
26  x10 <- 0x0000000000000000
27
28  mem[X,0x000000000000100C] -> 0xB283
29  mem[X,0x000000000000100E] -> 0x0182
30  [3] [M]: 0x000000000000100C (0x0182B283) ld x5, 0x18(x5)
31  mem[R,0x0000000000001018] -> 0x0000000080000040
32  x5 <- 0x0000000080000040
```

1.3 SAIL Trace 实现方法

SAIL MODEL 的实现预留了回调接口供用户实现自定义操作, 目前主要用于打印 TRACE

```
function mem_write_callback(_) = ()
function mem_read_callback(_) = ()
function mem_exception_callback(_) = ()

function pc_write_callback(_) = ()
function xreg_full_write_callback(_) = ()
function csr_full_write_callback(_) = ()
function csr_full_read_callback(_) = ()

function trap_callback(_) = ()
```

```
unit mem_write_callback(const char *type, sbits paddr, uint64_t width,
                        lbits value)
{
  if (config_print_mem_access) {
    fprintf(trace_log, "mem[%s,0x%0*" PRIX64 "] <- 0x", type,
            static_cast<int>((zphysaddrbits_len + 3) / 4), paddr.bits);
    print_lbits_hex(value, width);
  }
  if (config_enable_rvfi) {
    zrvfi_write(paddr, width, value);
  }
  return UNIT;
}
```

```
match (del_priv) {
  Machine => {
    mcause[IsInterrupt] = bool_to_bits(intr);
    mcause[Cause]       = zero_extend(c);

    mstatus[MPIE] = mstatus[MIE];
    mstatus[MIE]  = 0b0;
    mstatus[MPP]  = privLevel_to_bits(cur_privilege);
    mtval         = tval(info);
    mepc          = pc;

    cur_privilege = del_priv;

    handle_trap_extension(del_priv, pc, ext);

    track_trap(del_priv);      Implement callbacks for

    prepare_trap_vector(del_priv, mcause)
  },
  Supervisor => {
```

1.4 SAIL 缺点

1. 基于文本的日志可能会变得非常大（而二进制格式体积上会小得多）
2. 下游工具(verification, trace-driven simulation)必须编写专门的 parser 处理 SAIL 日志

```
def extractVirtualMemory(self, line):
    mem_r_pattern = re.compile(r'mem\[R,([0-9xABCDEF]+)\] -> 0x([0-9xABCDEF]+)')
    mem_x_pattern = re.compile(r'mem\[X,([0-9xABCDEF]+)\] -> 0x([0-9xABCDEF]+)')
    mem_depa_pattern = re.compile(r'mem\[([0-9xABCDEF]+)\]')
    instr_trap_pattern = self.instr_pattern_c_sail_trap.search(line)
```

3. 日志格式不够稳定, 目前会按需随时更新, 容易出现 break change, 导致下游工具出现 bug (ACT 多次由于 SAIL 日志格式更新出现 bug)

SAIL 需要一个更好的 trace/log 格式 用于形式化验证

2. RISC-V 现有 Trace 格式浅析

2.1 六种 Trace 格式

1. E-Trace (“Efficient Trace”) From RVI; ratified

   https://github.com/riscv-non-isa/riscv-trace-spec

2. N-Trace (“Nexus Trace”) From RVI; ratified

   https://github.com/riscv/tg-nexus-trace

3. RVFI (“RISC-V Formal Verification Interface”) From SymbioticEDA

   https://github.com/SymbioticEDA/riscv-formal

4. RVVI (“RISC-V Verification Interface”) From Imperas originally (now part of Synopsys), but now public.

   https://github.com/riscv-verification/RVVI

5. “Trace Protocol” From Bluespec, Inc. (but ready to contribute this spec to RVI as an open spec)

   https://github.com/user-attachments/files/16931294/2020-03-10_trace-protocol.pdf

6. RVFI-DII (“Risc-V Formal Interface - Direct Instruction Injection”) From CTSRD-CHERI

   https://github.com/CTSRD-CHERI/TestRIG

1. E-Trace (“Efficient Trace”) From RVI; ratified

   https://github.com/riscv-non-isa/riscv-trace-spec

   - 重点在于实现高效/高压缩比的控制流追踪
   - 依赖于解码器能够访问原始的 ELF 文件 (用于映射到源代码, 或者分析未追踪的指令序列)
   - 不适合自修改代码/JIT, 因为没有 ELF 文件
   - 不捕获架构状态更新(GPRs, FPRs, CSRs) E-Trace 有一个捕获内存更新的部分

2. N-Trace (“Nexus Trace”) From RVI; ratified

   https://github.com/riscv/tg-nexus-trace

   - IEEE-5001(Nexus) 标准是针对嵌入式处理器开发的一种调试和跟踪标准，它基于 JTAG 协议，并扩展了 JTAG 的功能，以支持对嵌入式系统的调试

   类似于 E-Trace，但是主要用于调试

2.4 RVFI / RVFI-DII / RVVI

3. RVFI (“RISC-V Formal Verification Interface”) From SymbioticEDA
   - 以每周期指令退休（instruction retirement per cycle）为粒度，输出处理器在每个周期所退休指令的完整状态信息。
6. RVFI-DII (“Risc-V Formal Interface - Direct Instruction Injection”) From CTSRD-CHERI
   - 基于 RVFI, 扩展了一个 instruction trace format, 规范化了 execution trace 的数据包格式
4. RVVI (“RISC-V Verification Interface”) Used by OpenHWGroup.
   - RVFI 的超集
   - 记录的数据太多, 例如包含了所有的 GPR, FPR, CSR

5. “Trace Protocol” From Bluespec, Inc. (but ready to contribute this spec to RVI as an open spec)
   - 二进制格式
   - 字段是可选的，因此可以支持从最小跟踪到详细跟踪
   - 可以捕获所有标准架构状态更新（尚未包含向量）
   - 可以捕获陷阱/终端
   - 可以捕获额外的中间状态
   - 已经应用在 Bluespec 内部多个硬件设计和模拟器实现中

3. RVFI-DII 协议介绍及 Sail 实现分析

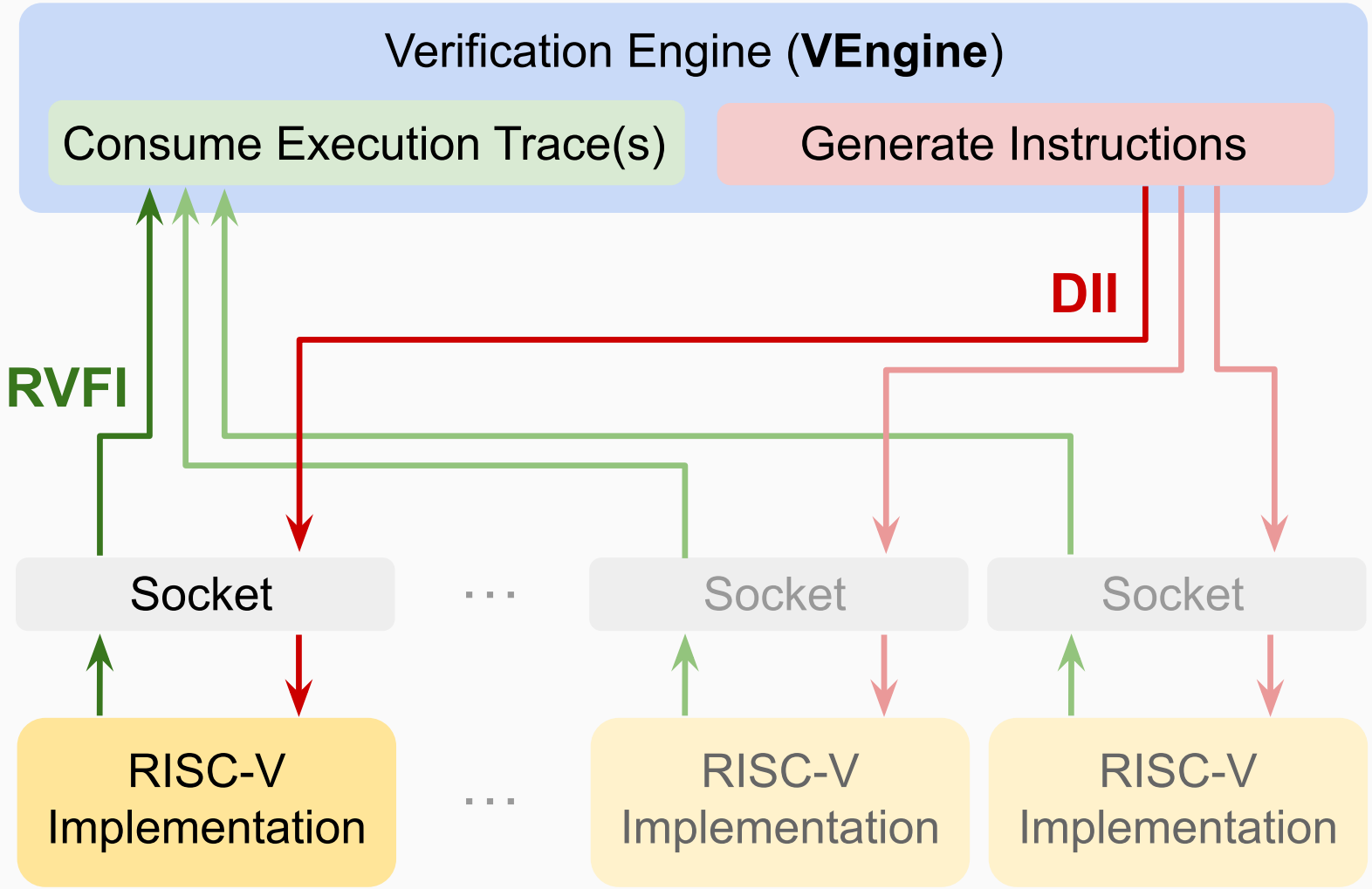
3.1 RVFI-DII 数据包格式

https://github.com/CTSRD-CHERI/TestRIG/blob/master/RVFI-DII.md

RVFI-DII 由两个数据包结构组成，旨在通过套接字发送。

- 从 vengine 发送到实现的指令跟踪格式 **instruction trace**
- 从实现返回给 vengine 的执行跟踪格式 **execution trace**
  - execution trace 有 V1, V2 两个版本

3.2 TestRIG 随机指令生成的 RISC-V 处理器测试框架



1. 被测设备需要通过 socket 和 VEngine 进行通信
2. fetch 取值从 VEngine 发送的数据包解析获得

3.3 Instruction Packet

```
struct RVFI_DII_Instruction_Packet {
    Bit8  padding;    // [7]
    Bit8  rvfi_cmd;   // [6] This token is a trace command.  For example, reset device under test.
    Bit16 rvfi_time;  // [5 - 4] Time to inject token.  The difference between this and the previous
                      // instruction time gives a delay before injecting this instruction.
                      // This can be ignored for models but gives repeatability for implementations
                      // while shortening counterexamples.
    Bit32 rvfi_insn;  // [0 - 3] Instruction word: 32-bit instruction or command. The lower 16-bits
                      // may decode to a 16-bit compressed instruction.
}
```

The `rvfi_cmd` field currently has two values defined:

```
0 = EndOfTrace  // Reset the implemenation, including registers, memory, and PC (to 0x80000000)
1 = Instruction // Execute the instruction in rvfi_insn
```

3.4 Execution Packet V1

```
struct RVFI_DII_Execution_Packet {
   Bit8  rvfi_intr;      // [87] Trap handler:           Set for first instruction in trap handler.
   Bit8  rvfi_halt;      // [86] Halt indicator:         Marks the last instruction retired
                         //                                   before halting execution.
   Bit8  rvfi_trap;      // [85] Trap indicator:         Invalid decode, misaligned access or
                         //                                   jump command to misaligned address.
   Bit8  rvfi_rd_addr;   // [84]      Write register address:  MUST be 0 if not used.
   Bit8  rvfi_rs2_addr;  // [83]                               otherwise set as decoded.
   Bit8  rvfi_rs1_addr;  // [82]      Read register addresses: Can be arbitrary when not used,
   Bit8  rvfi_mem_wmask; // [81]      Write mask:              Indicates valid bytes written. 0 if unused.
   Bit8  rvfi_mem_rmask; // [80]      Read mask:               Indicates valid bytes read. 0 if unused.
   Bit64 rvfi_mem_wdata; // [72 - 79] Write data:              Data written to memory by this command.
   Bit64 rvfi_mem_rdata; // [64 - 71] Read data:               Data read from mem_addr (i.e. before write)
   Bit64 rvfi_mem_addr;  // [56 - 63] Memory access addr:      Points to byte address (aligned if define
                         //                                        is set). *Should* be straightforward.
                         //                                        0 if unused.
   Bit64 rvfi_rd_wdata;  // [48 - 55] Write register value:    MUST be 0 if rd_ is 0.
   Bit64 rvfi_rs2_data;  // [40 - 47]                          above. Must be 0 if register ID is 0.
   Bit64 rvfi_rs1_data;  // [32 - 39] Read register values:    Values as read from registers named
   Bit64 rvfi_insn;      // [24 - 31] Instruction word:        32-bit command value.
   Bit64 rvfi_pc_wdata;  // [16 - 23] PC after instr:          Following PC - either PC + 4 or jump/trap target.
   Bit64 rvfi_pc_rdata;  // [08 - 15] PC before instr:         PC for current instruction
   Bit64 rvfi_order;     // [00 - 07] Instruction number:      INSTRET value after completion.
}
```

3.5 Execution Packet V2

```
bitfield RVFI_DII_Execution_PacketV2 : bits(512) = {
  magic : 63 .. 0,          // must be set to 'trace-v2'
  trace_size : 127 .. 64,   // total size of the trace packet + extensions
  basic_data : 319 .. 128, // RVFI_DII_Execution_Packet_InstMetaData
  pc_data : 447 .. 320,     // RVFI_DII_Execution_Packet_PC
  // available_fields : 511 .. 448,
  integer_data_available: 448, // Followed by RVFI_DII_Execution_Packet_Ex
  memory_access_data_available: 449, // Followed by R VFI_DII_Execution_Pa
  floating_point_data_available: 450, // TODO: Followed by RVFI_DII_Executi
  csr_read_write_data_available: 451, // TODO: Followed by RVFI_DII_Execut
  cheri_data_available: 452, // TODO: Followed by RVFI_DII_Execution_Packe
  cheri_scr_read_write_data_available: 453, // TODO: Followed by RVFI_DII_
  trap_data_available: 454, // TODO: Followed by RVFI_DII_Execution_Packet
  unused_data_available_fields : 511 .. 455, // To be used for additional F
}
```

1. Fetch from RVFI Instruction Packet

```
function fetch() → FetchResult = {
  if get_config_rvfi()
  then return rvfi_fetch();
  match ext_fetch_check_pc(PC, PC) {
```

2. Update packet if necessary

```
function rvfi_fetch() → FetchResult = {
  rvfi_set_pc_data_rdata(zero_extend(get_arch_pc()));
  rvfi_set_inst_data_order(minstret);
  rvfi_set_inst_data_mode(zero_extend(privLevel_to_bits(cur_privilege)));
  rvfi_set_inst_data_ixl(zero_extend(misa[MXL]));
```

3. Return Execution Packet

```
auto pkg = rvfi_get_exec_packet_v2();
send_packet_raw(&pkg, config_print);
if (rvfi_int_data_present) {
  send_packet_raw(&rvfi_int_data, config_print);
}
if (rvfi_mem_data_present) {
  send_packet_raw(&rvfi_mem_data, config_print);
}
```

4. RISC-V 部分验证方案总结

4.1 测试方法

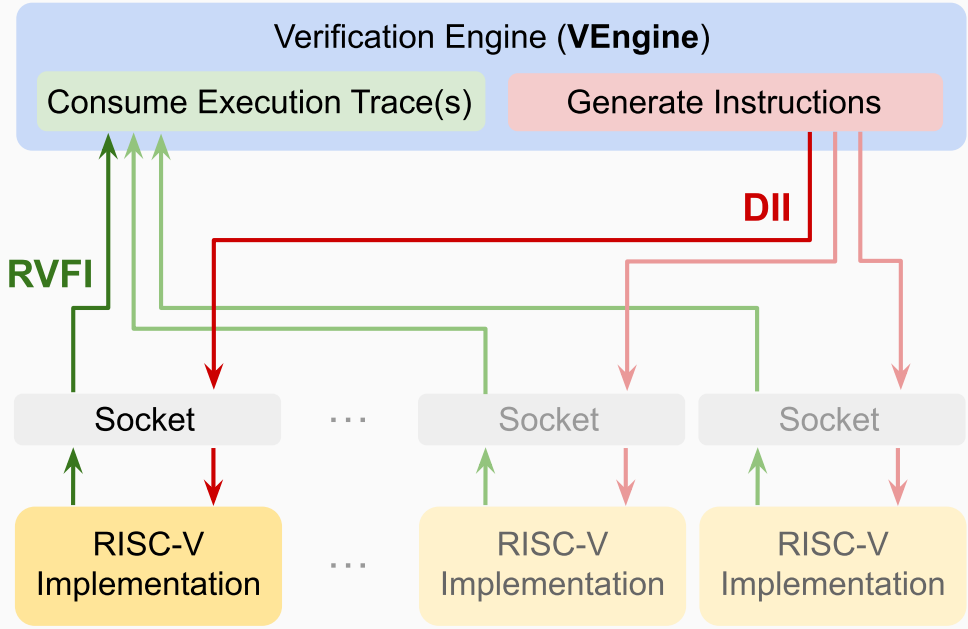
1. 差分测试
2. 自测

通过测试执行器，控制待测设备和标准设备单步执行指令，并对比执行后的状态是否保持一致

- RVFI / RVVI / RVFI-DII / Bluespec
- Xiangshan difftest

对实现 (DUT) 的要求

- 以某种方式支持单步运行
- 提供适当的接口获取信息



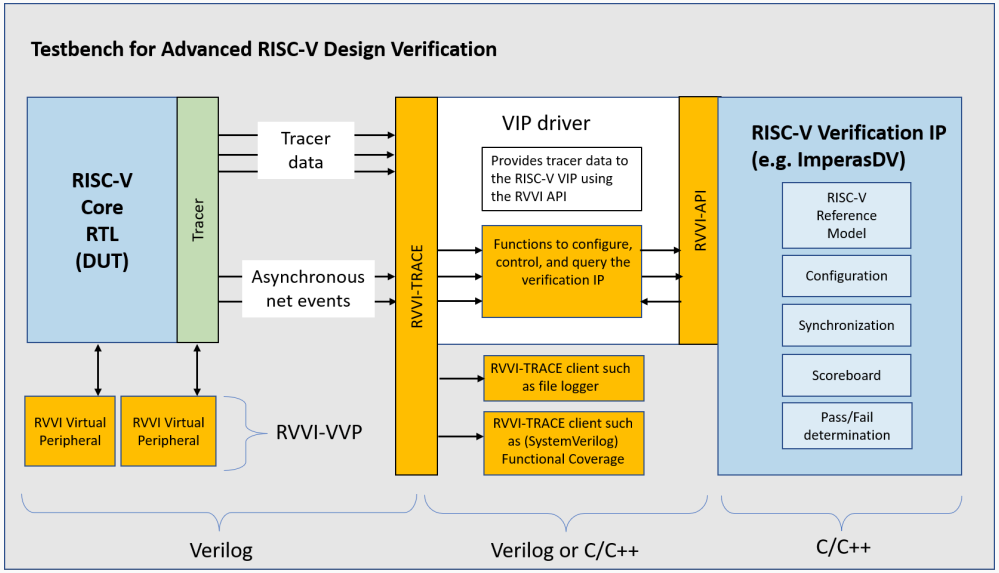
RVFI-DII

通过测试执行器，控制待测设备和标准设备单步执行指令，并对比执行后的状态是否保持一致

- RVFI / RVVI / RVFI-DII / Bluespec
- Xiangshan difftest

对实现 (DUT) 的要求

- 以某种方式支持单步运行
- 提供适当的接口获取信息



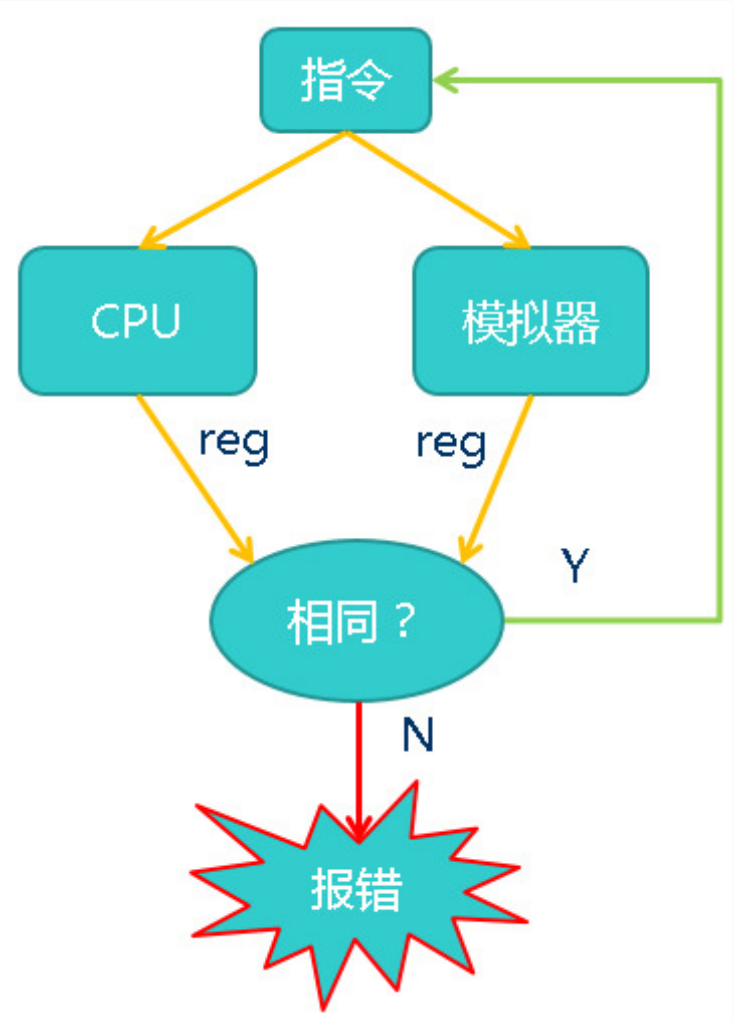
RVVI

通过测试执行器，控制待测设备和标准设备单步执行指令，并对比执行后的状态是否保持一致

- RVFI / RVVI / RVFI-DII / Bluespec
- Xiangshan difftest

对实现 (DUT) 的要求

- 以某种方式支持单步运行
- 提供适当的接口获取信息



4.2 trace-based

通过测试执行器，控制待测设备和标准设备单步执行指令，并对比执行后的状态是否保持一致

- RVFI / RVVI / RVFI-DII / Bluespec
- Xiangshan difftest

对实现 (DUT) 的要求

- 以某种方式支持单步运行
- 提供适当的接口获取信息

4.3 result-based

待测设备自行运行 ELF 文件

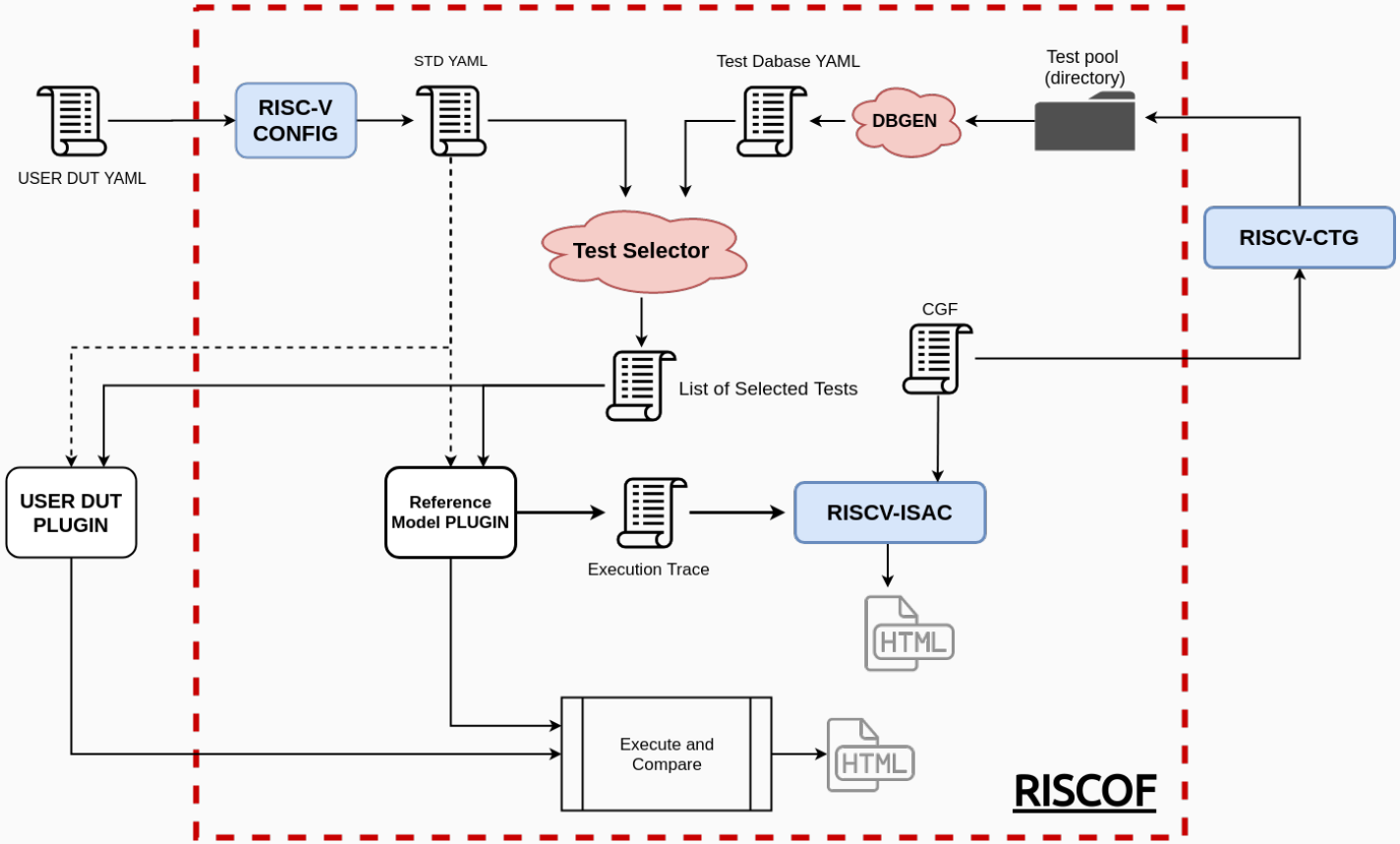
- riscv-tests/riscv-vector-tests

```
#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.
```

4.4 mixed

既可以差分测试，也可以自测

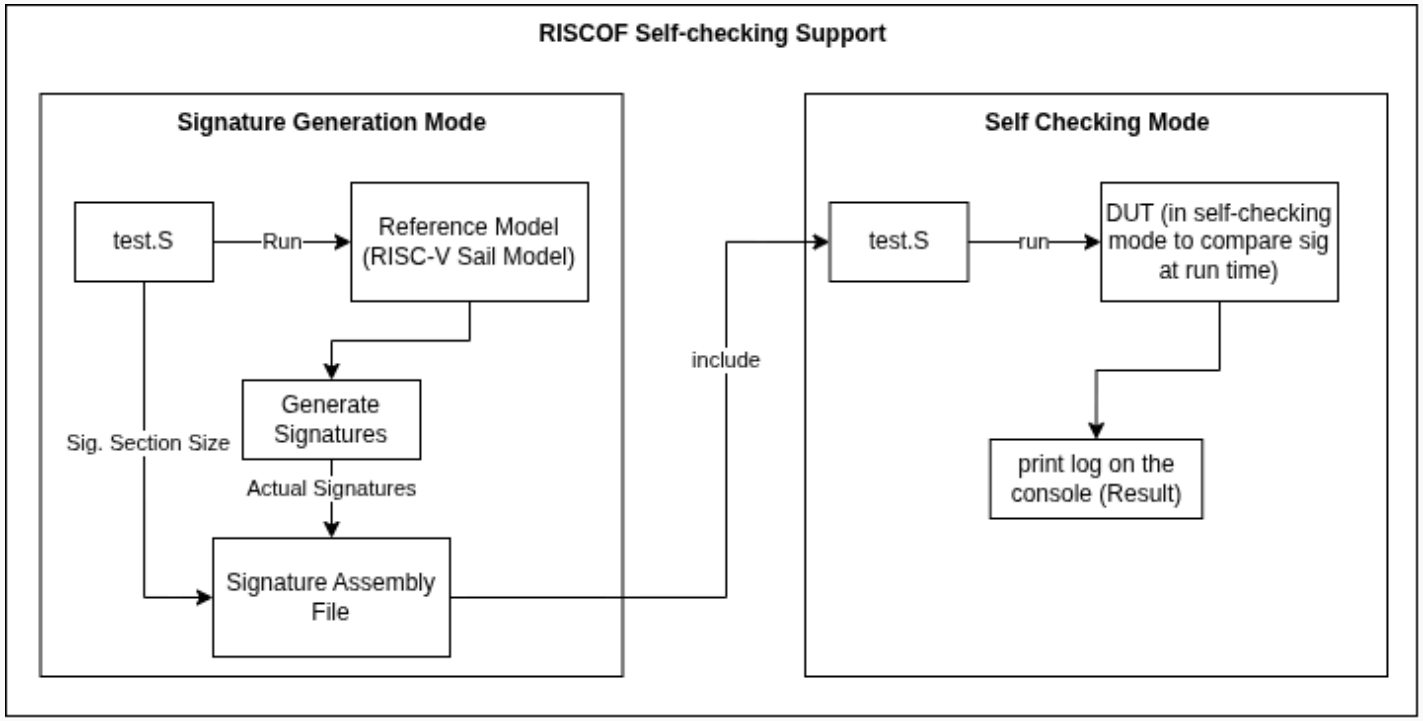
- riscv-arch-tests
- cvw-arch-verif



4.4 mixed

既可以差分测试，也可以自测

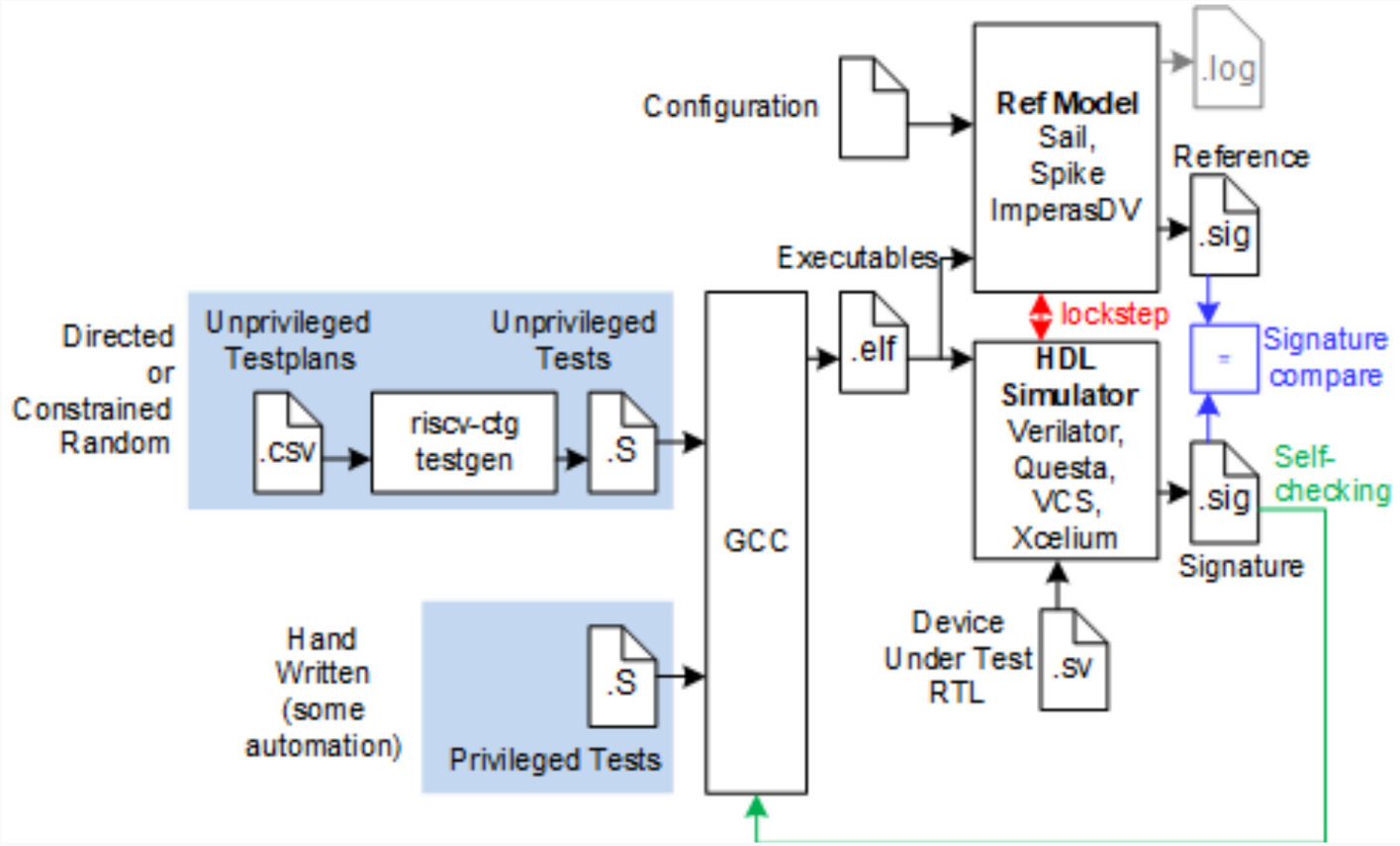
- riscv-arch-tests
- cvw-arch-verif



4.4 mixed

既可以差分测试，也可以自测

- riscv-arch-tests
- cvw-arch-verif



5. 总结

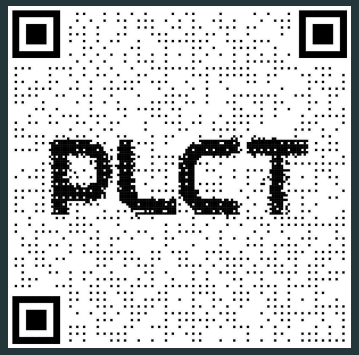
5.1 总结

1. 用户对验证中产生的 trace/log 的要求
   - 体积小，压缩率高
   - 捕获足够多的状态，包括陷阱中断，隐式修改等
   - 需要有稳定的接口，避免工具需要频繁变更
2. 各个 RISC-V 开源实现 / 厂商私有实现都需要进行处理器形式化验证，验证方法没有统一标准，协议多样
   - E-Trace/N-Trace
   - RVFI/RVVI/RVFI-DII/Bluespec

5.1 总结

3. 常见的验证方法包括两类，差分测试或者自测
   - 前者依赖于被侧设备和标准设备需要提供 访问处理器状态 和控制处理器执行 的相关接口
   - 自测测试集一般难以满足复杂场景的测试要求
   - riscv-tests/riscv-vector-tests
   - riscv-arch-tests/cvw-arch-verif
4. Sail 作为 RVI 推动的 Golden Model，目前可以通过日志输出用于 ACT 测试。同时也支持使用 RVFI-DII 协议进行差分测试
5. Sail 目前没有实现单步执行，提供友好的接口，无法方便的集成到广泛的测试框架中

THANKS!



欢迎加入 PLCT BJ88(SAIL) P121(ACT)
J160(Pydrofoil) 岗位实习