2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

An Extension to RISC-V Test Generator: A Quick Exception Check

Ranjan Kumar Barik[1]
ranjan-kumar.barik@thalesgroup.com
Sai Krishna Pidugu[2]
sai-krishna.pidugu@thalesgroup.com
Manju Bhargavi[3]
manju-bhargavi.kandadi@thalesgroup.com
Subhra Kanti Das[4]
subhra-kanti.das@thalesgroup.com
[1, 2, 3, 4] Thales India Private Limited, Bengaluru, Karnataka, 560025

***Abstract*- Any verification movement may be sub-divided into two testing i.e. positive approach & negative approach. For example RISC-V compliance testing approach ensure compatibility for a complete RISC-V ecosystem. Compliance test suite are positive scenario with respect to RISCV-ISA specification and mostly it's targeted for correct implementation of Instructions. This paper provide a quick exception testing for any RISC-V architecture with negative approach. Here, we have provided an extension version of RISCV-DV random generator aiming RISC-V exception check. This work offers an enhanced test-generator (ETG) with extensive user exception configurations which allows user to generate exception test scenarios. The ETG is a UVM based environment comes with additional features without affecting existing test generator flow. The intended exception testing for this user extension is demonstrated to be accomplished with less instruction when compared to traditional analysis, and it is tested across open RISC-V cores.**

I. INTRODUCTION

When it comes to processor core, the instruction set architecture i.e. ISA has the crucial role for the overall eco-system. RISC-V provide an open & royalty free ISA leading towards processor innovation & other eco-system collaboration. Furthermore RISC-V offers high flexibility in terms of custom and features implementation. Thus it is rapidly gaining popularity in both academics & industry substituting heart of the embedded system. However, the flexibility in design and features, leads towards its own verification challenges. In recent time, we have encountered several RISC-V specific solutions towards its verification. First approach is to provide hand-written or generated test-suites for the RISC-V compliance testing. The compliance test-suite is basically sufficient for the basic sanity check including certain corner cases targeting RISC-V instruction extensions. This is good resource to start and validate design in-terms of specification however, this has no role to play for functional verification. Nowadays, the verification of program execution against a golden reference model is becoming a common practice to verify processors core.

An efficient cross-level verification approach is proposed for processor in [1]. The authors have proposed RTL with respect to RISCV-ISA generating instruction stream on-the-fly during simulation. Here the solution is provided as a testbench that populates instruction to both core & Instruction set simulator (ISS) for execution & detection of the any inaccuracies. The torture test generator is the framework consists of various sub-projects that build one upon another [2]. This framework is based on monitoring all the registers state out of memory during a test program execution. The test generation consists of both generation with test sequences and hand written test scenario followed by comparison with spike simulator [3]. Similarly the force RISC-V test generator automates the generation of test cases, reducing the manual effort in [4]. This test generator claim to achieve better test coverage and corner cases towards various instruction combinations. However, the force RISC-V test generator requires some knowledge base like symbolic execution techniques & constraint solving concepts for effective utilization of its capabilities. Further, the configuration and fine-tuning of the tool for specific processor verification may require expertise and understanding of the underlying hardware design principles. Considering the trend in the verification of core using ISS, there are some research activities on verification of instruction set simulator [5]. Here, authors have adapted simulation based approaches with adequate number of test set for the validation. The authors have

claimed to have improved test case generation with suggested coverage-guided fuzzing (CGF) and a custom mutation procedure approach.

Formal verification method is also one of the active development alongside of test generation approach towards verification of RISC-V core. The RISCV-Formal is one of the open-source framework providing formal test benches using FOSS SymbiYosys verification flow [6]. Similarly, OneSpin 360 is one of the notable commercial RISC-V verification app [7]. The open-source tool supports limited RISC-V privileged specification further demanding implementation of RISC-V formal interface (RVFI) for the processor core. Although, the usage of formal method ensures the correctness, it comes with complexity and potential scalability issues. Thus this technique is preferred to be complemented by simulation based verification approach.

RISCV-DV is one of the well-known test generator developed and maintained by Google [8]. This use advantages of system-verilog and Universal Verification Methodology (UVM) for the test generation environment. This method emphasis on result comparison between RTL simulation & ISS. The RISCV-DV is utilized towards verification of popular RISC-V cores like cva6, ibex etc [9] [10]. However, RISC-DV has high overhead factor as it is intended for generic RISC-V cores and various ISS. The verification process is highly time consuming as individual test program need to be complied, loaded and need to execute at both RTL & ISS simulator. Further, the generated log files from both the simulators need to be converted to appropriate format (i.e .CSV) for final test result comparison.

A mutation based study is presented for RISCV-DV evaluating its error finding competences in [11]. Along with evaluation, authors have also recommended additional performance metrics for the framework and believed that constraint random verification (CRV) to be ideal for further directional research. In recent development [12], authors have offered verification framework by adding a direct instruction injection (DII) interface to RISCV-formal interface which is used to provide test harness. Here, authors have discussed about the RISCV-DV to be the most advanced sequence generator for RISC-V cores. However, authors has also mentioned several drawbacks of this stating the automation in generic test generation leads to overlong and complex test sequence.

Even though, RISCV-DV supports exception testing, this also follows above described time consuming generation & verification flow. There is a handshaking method for passing execution response to the verification environment however this part is not much explored for generated test-cases due to log to log comparison approach. In this paper, we have considered exception testing is the basic test requirement for any RISC-V core before any further functional validation. This paper presents a quick exception testing approach avoiding usage of ISS. Here we have offered an extension version of RISCV-DV in terms of handshaking between test-bench and generated test program. The rest of the paper is organized as follows: section 2 states about overview for RISC-V exception. Section 3 describes the proposed quick exception check. section 4 provides case study & experimental evaluation. The paper concludes in Section 5.

II. Overview for RISC-V Exception

As per RISC-V specification, a synchronous exception is triggered when any violations of any ISA extension is raised. This also reflects towards any integrity of any inappropriate memory-access. The synchronous exception is also related to design error when any illegal instruction or illegal accessed is identified by the core. The control and status registers (CSR) play an important role displaying status of the core during the course of the execution. In addition, some of the CSRs provides essential information related to accessible level, address space and current implementation. Thus, to capture information related to exception (IRE), this is essential to monitor some of the CSRs. There are several important CSRs and instruction for the exception related execution.

- *mcause*: it denotes machine cause register, it stores the cause of the most recent exception, and is updated when an exception happens.
- *mepc*: It denotes machine exception program counter. As per the RISC-V specification, when trap is taken into Machine mode, mepc is written with virtual address of the instruction which is interrupted or encountered.
- *mtval*: It denotes machine trap value register, when an exception is encountered, this register holds exception-specific information to assist software in handling the trap. In the case of errors in the load-store unit mtval holds the address of the transaction causing the error. If this transaction is misaligned, the mtval holds the address of the missing transaction part. In the case of illegal instruction exceptions, it holds the actual faulting instruction. For all other exceptions, MTVAL register is 0.

- *mtvec*: mtvec stores the trap handler address and access configuration.
- *mret*: Machine Mode Trap Handler Return is used to return from a trap handler that is executing in the Machine Mode. The mret instruction tells the processor to pass control back to the address in the mepc register.
  Note that: There must be a dedicated sret and sepc CSR if core supports supervisor privilege mode.
- *ecall*: This instruction put request to the supportive environment. Depending upon the mode of operation, this generate particular type of exception code (EC) and halt other operation.

TABLE I
MODIFICATION OF VARIOUS CSRS WRT OCCURRING EXCEPTION

Description	Exception type	Change in mcause	Change in mepc	Change in mtval value
Instruction address misaligned	00	0 -> 0	E-PC value*	F-Virtual Address #
Instruction access fault	01	0 -> 1	E-PC value	Virtual address of the portion of the instruction that caused the fault
Illegal instruction	02	0 -> 2	E-PC value	Actual Faulting Instruction (i.e Instruction causing Exception)
Breakpoint	03	0 -> 3	E-PC value	F-Virtual Address
Load address misaligned	04	0 -> 4	E-PC value	F-Virtual Address
Load access fault	05	0 -> 5	E-PC value	F-Virtual Address
Store/AMO address misaligned	06	0 -> 6	E-PC value	F-Virtual Address
Store/AMO access fault	07	0 -> 7	E-PC value	F-Virtual Address
Environment call from U-mode	08	0 -> 8	E-PC value	0
Environment call from S-mode	09	0 -> 9	E-PC value	0
Environment call from M-mode	11	0 -> 11	E-PC value	0
Instruction page fault	12	0 -> 12	E-PC value	Virtual address of the portion of the instruction that caused the fault
Load page fault	13	0 -> 13	E-PC value	Virtual address of the portion of the access that caused the fault.
Store/AMO page fault	15	0 -> 15	E-PC value	Virtual address of the portion of the access that caused the fault.

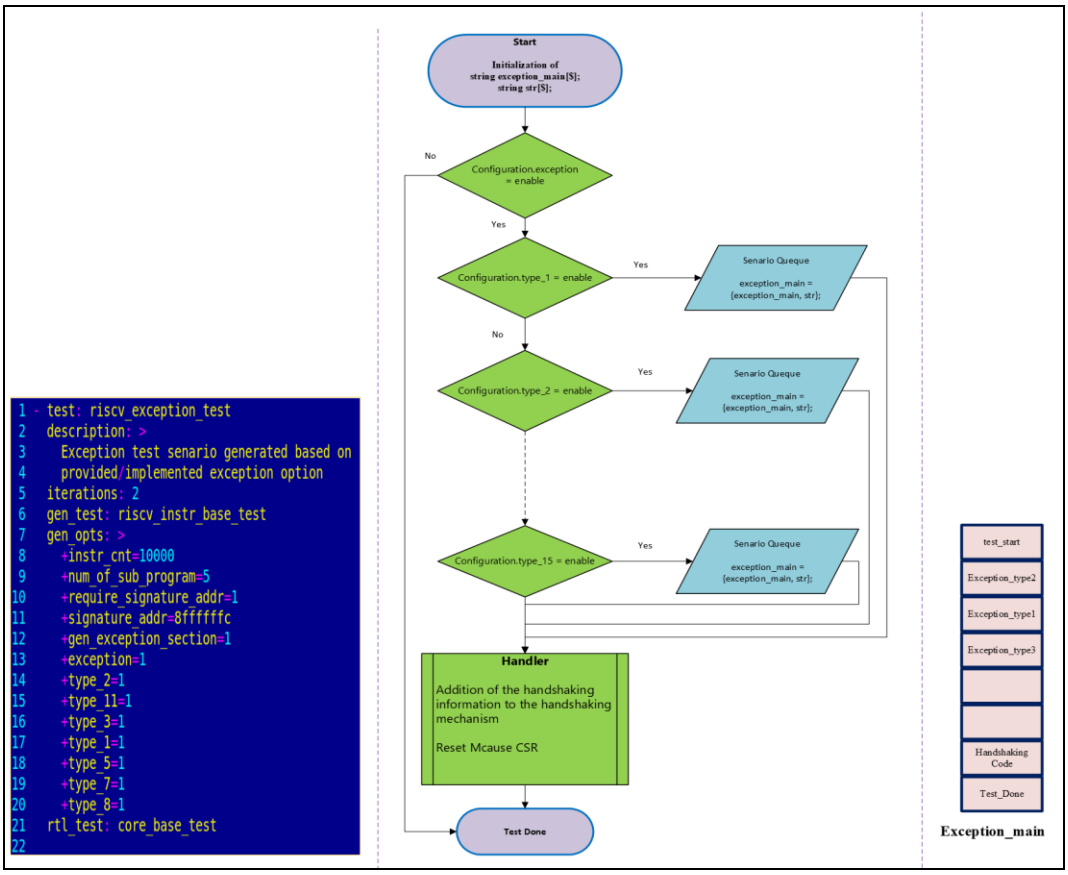
** E-PC value =Exception PC value*
 # F-Virtual Address= Faulting Virtual Address

Note that: The hardware platform specifies which exceptions must set mtval informatively. Some may unconditionally set it to zero and some may exhibit either behaviour on the primary event that caused the exception. If the hardware platform specifies that no exceptions set mtval to a nonzero value, then mtval is read-only zero.

III. Proposed Quick Exception Check

In this paper, we have extended RISCV-DV test generator keeping specific to the exception scenarios. The test generator is based on System Verilog and UVM which generate assembly test program based on the specified ISA Extension. We have provided essential environmental configuration for the generation of desired assembly test program. Considering RISC-V privilege specification, we have accumulated possible scenarios targeting to various exception types. RISCV-DV generator supports handshaking mechanism consisting of code or set of instructions which has a fixed signature address or virtual address. Thus the execution of the generated test program, the verification environment start monitoring the signature address for any write operation. This signature address acts as memory mapped address where we captures the DUT operation and evaluate at testbench end.

A RISCV-DV based CPU verification environment utilizes the handshaking mechanism providing a basic set of tasks to monitor this signature_addr for any writes. The write operation indicates that the processor core is executing a section of handshake assembly code and transmitting information through signature address for evaluation in verification environment. Note that this mechanism is not constrained to only UVM environment, this also can be easily implemented with normal system verilog environment.



(a) (b) (c)

Figure 1. Exception generation flow (a) Sample exception yaml description (b) Test generation flow for RISC-V exception scenario (c) Exception queue containing exception string configured in the test yaml

Figure 1 shows exception generation flow. Figure 1 (a) is the sample exception yaml description for generation of different set of exception type. Here, user has to specify different set of exception type supposes to be generated for the RISC-V core. Figure 1 (b) represents test generation flow for RISCV-V exception scenario. Here we have utilized system verilog override concept to provide option for various exceptions generation configuration without affecting the existing generation classes. The RISC-V privilege specification is considered for standard exception scenario for each exception type. Figure 1 (c) shows a sample exception queue containing exception string configured as per user test yaml. Note that, the generation of the exception is dependent to the test yaml allowing user to generate single test program intended for a single exception or multiple exception scenario.

In this work the signature_addr acts as a memory-mapped address that is used as monitoring exception related information at testbench. As demonstrated the value of signature_addr is 0x8ffffffc, but this can be any address depending on the CPU memory map. However, this value may varies as per the implemented RISC-V core. Figure 2 shows handshaking mechanism for the introduced quick exception check**.** The proposed quick exception check has an updated handshaking mechanism between UVM testbench and the test program. For this we have overridden the existing RISCV-DV asm generation code. The mechanism is mostly targeted for capturing the actual exception code (EC) and the information related to exception (IRE) in the verification environment. The actual exception code is captured in MCAUSE CSR in test program, the same value is taken and stored in the UVM test environment with a signature address.

IRE is specific to the type of expected exception code. For exception code i.e. illegal instruction the IRE has only actual faulting Instruction. Although, the mcause & mepc are guaranteed to hold supported exception codes and valid virtual addresses respectively, the mtval is implementation specific. As per RISCV specification mtval provides exception specific information i.e. the instruction data in case of an illegal instruction exception, however, it is also legal behavior to simply set MTVAL to zero in case the feature is not supported. Thus we have used MEPC CSR to capture the actual instruction value responsible for the exception. The MEPC contain the address of the actual instruction, thus we have added a small section of code for store operation in handshaking mechanism to capture and store in test environment.

For environmental call, exception code and its validation is not possible to evaluate only with instruction. Here we have modified the test generation flow in such that based on the type of the exception, the relevant exception related information also communicated to verification environment. The proposed handshaking ensures to provide relevant instruction along with current mstatus value for accurate evaluation in verification environment. Similarly, the exception code associated to all access fault, the test generation is liable for providing handshaking section of code along with physical memory protection (pmp) configuration option. For the simplicity, we have only considered a single pmp region for our test generation. The same signature address has been used for the capturing both actual exception code as well as relevant information. The proposed handshaking mechanism between generated test program and testbench is illustrated in figure 2. Note that the current development is restricted for the exception related to page fault scenario and supervisor privilege mode.

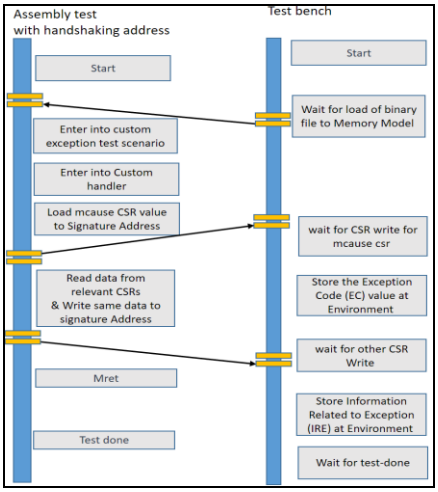


Figure 2. Handshaking mechanism for quick exception check

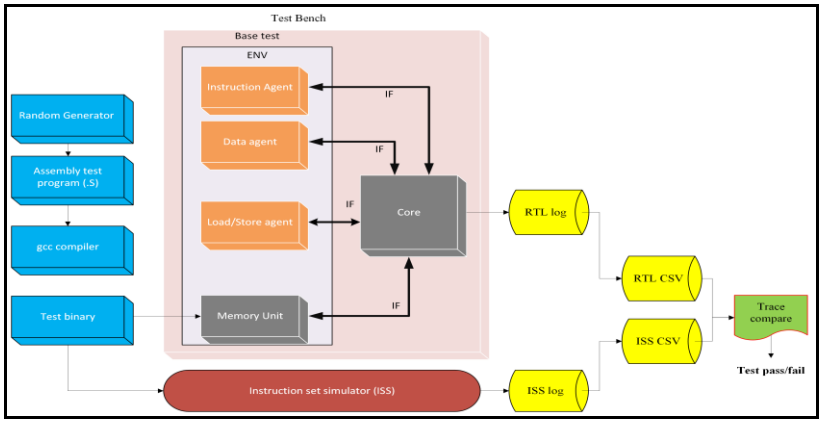


Figure 3. Conventional Core verification flow

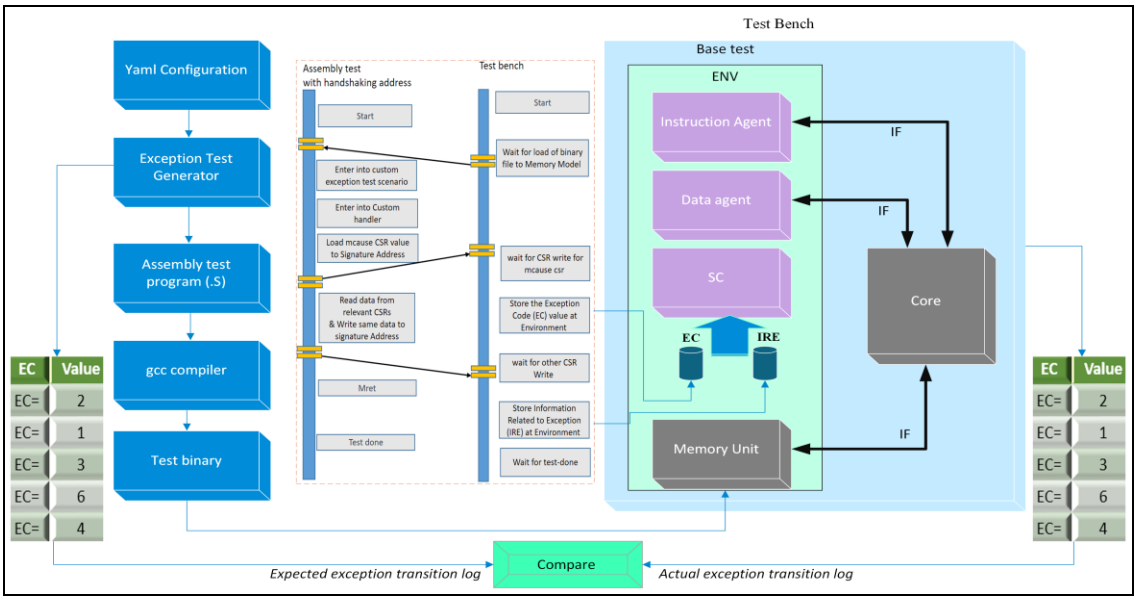


Figure 4. Proposed framework for Quick exception check

Figure 3 shows the conventional core verification approach. The generated or custom test program is compiled for executable binary. The both core and ISS are simulated with the same binary file for generation of RTL log & ISS log respectively. Then these logs are converted to better file extension i.e. .csv for evaluation of test pass/fail outcome. The main challenge of this flow is that regardless of any generated instructions, the ISS and RTL core must perform as identical architectural and configuration state (i.e. register updates).

Figure 4 shows proposed framework for quick exception check. This is relatable to the described conventional core verification flow presented in figure 3. However, considering the fact of dedicated purpose of exception check, we have omitted the usage of instruction set simulator in verification flow.

This is achieved by the following contributions:

a) We have added exception related scenario for the extension of UVM based RISCV-DV test generator. The additional configuration parameters are added to the UVM class to accompaniment these scenario for the test program.

b) Considering the verification of the injected exception, the handshaking mechanism is tuned so that the appropriate exception related information pass on to the testbench.

c) The received actual exception related information is captured at the verification environment and compared with the excepted result.

d) We have introduced exception transition log during generation of exception program and core execution. This ensures correctness in core behavior for multiple exception scenarios.

IV. Case Study & Simulation Result

RISCV-DV tool provides choice to generate a wide range of test program for a RISC-V processor verification, however, the ETG provides an additional layer to core verification specifically for exception testing. The open-source ibex verification environment is integrated with the proposed ETG without upsetting existing features of RISCV-DV test generator. In this case study, we have enabled supported exception configuration for ibex & cva6 for the generation of the assembly program followed by execution of these tests in respective RISC-V cores. In terms of performance parameters we have captured core status with "mcycle" and "minstret" CSRs. Here mcycle CSR provides a count of the number of clock cycles for execution, on the other hand minstret CSR that simply counts number of executed instructions. For evaluation, we have also executed the existing exception scenario in current ibex verification environment obtaining the same performance parameters. Table 2 presents performance parameter for ibex with RISCV-DV for default illegal & ebreak test, similarly table 3 provides performance parameter for ibex & cva6. Note that this observation is based on all implemented ibex exception.

TABLE 2
PERFORMANCE PARAMETER FOR IBEX WITH RISCV-DV

Ibex core with RISC-DV	riscv_illegal_instr_test *(gen_opts:> illegal_instr_ratio=5 along with all default Config)*	riscv_ebreak_test *(gen_opts:> instr_cnt=6000 along with all default Config)*
Mcycle	6337	1583
Minstret	245	495

TABLE 3
PERFORMANCE PARAMETER FOR RISCV-CORES WITH ETG

RISC-V Core Performance parameter	Ibex	CVA6
Mcycle	3375	420
Minstret	207	438

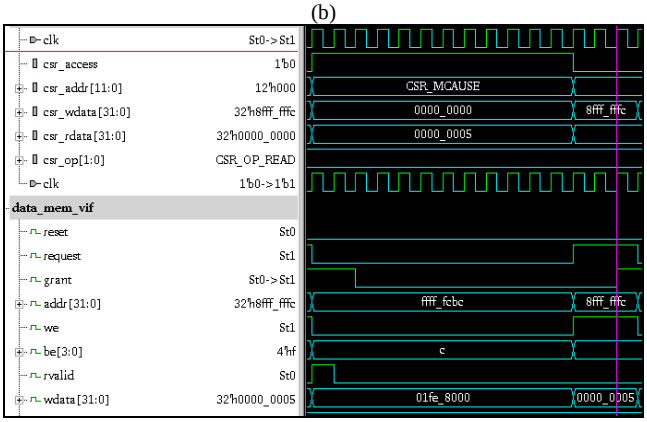
As described in figure 4, the input yaml description along with implemented exception option is provided to ETG. ETG provides a test program along with expected exception transition (EET) log reporting exception generation sequence as per the test generator. Similarly, while RTL simulation, core interacts with the testbench through handshaking to transfer data from the signature address. Testbench captures exception data and it generates actual exception transition (AET) log. The comparison of both EET and AET logs validate of the correct order of exception generation and execution at RISC-V core end. Further, it ensures the exceptions outlined in the user yaml file have been sufficiently addressed. This analysis enables user to confirm that the expected exceptions have been encountered and appropriately managed during the execution of the RISC-V core. The previously discussed handshaking mechanism in figure 4 ensure accurate collection of EC & IRE in the scoreboard component. Then, the comparison is made in between the captured data & the predicated data as per the implemented exception.

```
custom_exception_code:
                        csrr x28, 0x300        // Read mstatus CSR into register x28
                        csrr x29, 0x342        // Read mcause CSR  into register x29
                        csrr x30, 0x341        // Read mepc CSR into register x30
                        la x31, mtvec_handler   // Load the address of the mtvec_handler label into register x31
                        csrw 0x305, x31         // Write the value of register x31 to the mtvec CSR
                        li x6 , 10              // load the value to x6
                        csrw 0xf11, x6          // Write the value x6 to the mvendorid CSR
```

(a)

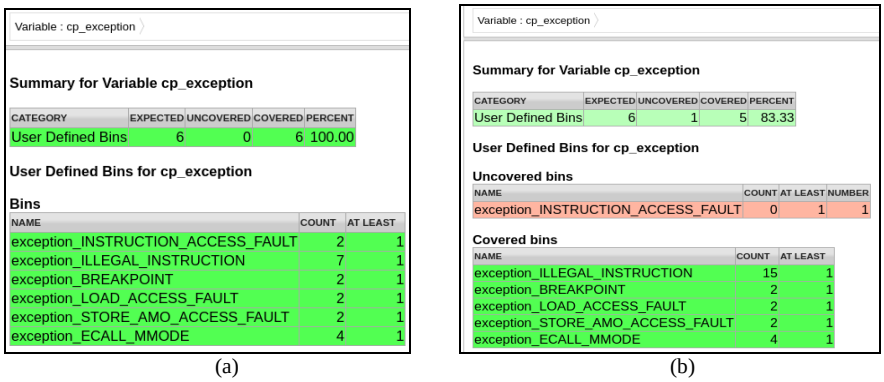
```
li x22, 0x8ffffffc // Load the signature address into a GPR
li x7, 0x342 // Load the address of MCAUSE into the second GPR
slli x7, x7, 8 // Left-shift the CSR address by 8 bits
addi x7, x7, 0x3 // Load the WRITE_CSR signature_type value into the bottom 8 bits of the data word. At this point, the data word is 0x00034203
sw x7, 0(x22) // Store this formatted word to memory at the signature address
csrr x7, 0x342 // Read the actual CSR value into the second GPR
sw x7, 0(x22) // Write the value held by the CSR into memory at the signature address
```

(b)



(c)

Figure 5: (a) Sample generated exception test for illegal write to read only CSR i.e. mvendorid (b) Generated handshaking assembly code that writes the value of the mcause CSR to signature addres (c) Waveform illustrating CSR read for mcause followed by same data write at signature address (i.e. exception type 05 is stored at signature address h'8fff_fffc)



Variable : cp_exception

Summary for Variable cp_exception

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	6	0	6	100.00

User Defined Bins for cp_exception

Bins

NAME	COUNT	AT LEAST
exception_INSTRUCTION_ACCESS_FAULT	2	1
exception_ILLEGAL_INSTRUCTION	7	1
exception_BREAKPOINT	2	1
exception_LOAD_ACCESS_FAULT	2	1
exception_STORE_AMO_ACCESS_FAULT	2	1
exception_ECALL_MMODE	4	1

(a)

Variable : cp_exception

Summary for Variable cp_exception

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	6	1	5	83.33

User Defined Bins for cp_exception

Uncovered bins

NAME	COUNT	AT LEAST	NUMBER
exception_INSTRUCTION_ACCESS_FAULT	0	1	1

Covered bins

NAME	COUNT	AT LEAST
exception_ILLEGAL_INSTRUCTION	15	1
exception_BREAKPOINT	2	1
exception_LOAD_ACCESS_FAULT	2	1
exception_STORE_AMO_ACCESS_FAULT	2	1
exception_ECALL_MMODE	4	1

(b)

Figure 6: Exception related coverage report for ibex core (a) test generation for all the implemented exception type (b) test generation excluding instruction access fault

Figure 6 shows coverage report for the ibex core. This report is pointing to only exception related cover group i.e. *cp_exception*. Note that the measure of cover bins for this group is dependent on the implementation of the RISC-V core. Figure 6 (a) shows the coverage details for all the implemented exception type. In case the test generation yaml excludes some of the supported exception type, this is not considered for the coverage as shown in figure 6 (b). For cva6, although we have not generated any coverage report, we have tested the generated ETG test as a custom test, further the EET & AET logs for the core ensure expected analysis of the supportive exception coverage.

V. Conclusion & Future work

In this work we have introduced an ETG which allows user to generate scenarios for triggering exceptions for a RISC-V core. To create such multiple scenarios, we have added functionality for the implemented exception type and provided handshaking mechanism with respect to the exception type. This approach is adapted in UVM environment, which brings familiarity for design verification engineers enhancing verification cycle, coverage measurement & reusability. This approach updated the communication between test program and simulation environment via an enhanced handshaking procedure. Therefore, this effort do not demand any instructor set simulator (ISS) and both the excepted & actual exception code is captured and compared in the test-bench. We believe since RISC-V is still an ISA under development, it is likely that more open-source frameworks where collaborative efforts can help build a strong verification framework. This work may also be improved for exception related to page fault and supervisor privilege mode. In terms of future work, this work can also be updated with a predictor model for the generated exception scenario for better handing in verification environment.

References

[1] Herdt, Vladimir, et al. "Efficient cross-level testing for processor verification: A RISC-V case-study." *2020 Forum for Specification and Design Languages (FDL)*. IEEE, 2020.

[2] "RISC-V torture test generator," https://github.com/ucb-bar/ riscv-torture.

[3] "Spike RISC-V ISA Simulator,"https://github.com/riscv-software-src/riscv-isa-sim

[4] "FORCE-RISCV," https://github.com/openhwgroup/force-riscv

[5] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in DATE, 2019, pp. 360– 365.

[6] "RISC-V formal verification framework," https://github.com/SymbioticEDA/riscv-formal.

[7] "OneSpin 360 DV RISC-V Verification App," https://www.onespin.com/ solutions/risc-v.

[8] "RISCV-DV," https://github.com/google/riscv-dv.

[9] "Ibex," https://github.com/lowRISC/ibex.

[10] "cva6," https://github.com/openhwgroup/cva6

[11] S. Ahmadi-Pour, V. Herdt, and R. Drechsler, "Constrained random verification for RISC-V: overview, evaluation and discussion". In *MBMV 2021; 24th Workshop* (pp. 1-8). VDE. March, 2021

[12] Joannou, Alexandre, et al. "Randomized Testing of RISC-V CPUs Using Direct Instruction Injection." *IEEE Design & Test* 41.1 (2023): 40-49.