# Extending the RISC-V Verification Interface for Debug Module Co-Simulation

Michael Chan, Ravi Shethwala, Richa Singhal, Advanced Micro Devices, Inc.
michael.chan@amd.com, ravi.shethwala@amd.com, richa.singhal@amd.com,
Lee Moore, Aimee Sutton, Synopsys Inc.
moore@synopsys.com, aimees@synopsys.com

***Abstract* – This paper describes proposed changes to the RISC-V Verification Interface in order to support co-simulation of RISC-V processor RTL and a connected Debug Module. The proposed changes enable the use of a processor reference model and associated verification IP to provide verification of Debug Mode operations, including the execution of abstract commands by the Debug Module.**

## I. Introduction

As RISC-V has shifted responsibility for processor verification to a wider community of developers, the tools, techniques, and best practices surrounding it are constantly evolving. One of these best practices is the RISC-V Verification Interface [1], or RVVI. An open standard available under Apache license on GitHub, RVVI enables reuse and efficiency in processor RTL verification by formalizing the interfaces that all RISC-V CPU testbenches should contain.

RVVI has developed and matured over time through use in a number of projects, including commercial and open-source RISC-V cores [2]. Until recently the focus of RVVI was the verification of a single RISC-V core. However, there are several scenarios where it makes sense to extend the processor testbench to include other components. One of these scenarios is the verification of a Debug Module. A Debug Module is a separate component external to the processor. Its purpose is to provide visibility into the state of a microprocessor in order to enable bring-up and debug of low-level software and hardware. Rather than create a behavioural model of the processor in order to verify the Debug Module, it makes sense to use the processor RTL to respond to Debug Mode events, so long as there is a reference model ensuring that the processor's response is correct.

AMD is developing a configurable RISC-V core for use in FPGAs, the MicroBlaze™ V. As part of their verification strategy, they have chosen to use an RVVI-compliant processor verification IP (VIP) from Synopsys, called ImperasDV. This processor verification solution connects to the design under test (DUT) using the RVVI-TRACE interface. It provides an instruction-accurate reference model that implements the RVVI-API, and verification components to continuously compare the state of the DUT with the internal state of the reference model.

One of the motivators for AMD's use of processor verification IP is to enable constrained-random verification of Debug Mode, and to verify the DUT's response to Debug Module abstract commands [3]. Initial simulations led to the realization that while ImperasDV was well-suited to the task, the RVVI-TRACE interface did not provide sufficient information to achieve these verification objectives. It soon became clear that modifications to RVVI were needed to enable verification of features defined in the RISC-V Debug specification.

This paper will present an overview of the RVVI and explain how it enables reuse and efficiency in processor verification. The challenges of Debug Module co-simulation will be explained in detail, followed by an explanation of the proposed changes to RVVI and how they address these challenges. Lastly, areas for future development will be identified. While ImperasDV is used as an example of RISC-V processor verification IP, the challenges described here are present in any RISC-V processor verification testbench that intends to verify Debug Mode operations with a connected Debug Module.

## II. RISC-V Verification Interface (RVVI)

RVVI was initially developed with two main focuses. The first is to help processor developers understand the needs of simulation-based verification. With this understanding, they can develop a module known as a tracer, whose task it is to extract information from the processor under test (DUT) and supply it to the testbench. The tracer requirements have been formalized as a SystemVerilog interface called RVVI-TRACE. The benefits of a standardized tracer interface are many. Tracers are often developed by RTL design engineers because extracting the

information needed requires an understanding of the processor's microarchitecture. The availability of a tracer interface specification makes it clear what information the designer needs to provide in order to facilitate processor verification. Because the RVVI standard is based on the experience of others, DV engineers can feel confident that they have all the information they need for thorough RISC-V processor verification. Starting from a specification rather than starting from scratch gives both parties a head start on their work. Additionally, the existence of a standard tracer interface enables the development and use of RISC-V processor verification IP. RVVI-TRACE data can also be used as the input to formal verification tools.

The second focus of RVVI is to specify the requirements for a testbench component or verification IP that comprehensively verifies a RISC-V processor during functional simulation. This component must include an instruction-accurate reference model of the processor. The functions that it must provide are formalized in the RVVI-API. This API can be used by anyone developing RISC-V processor verification IP either as a product or for internal use. The RVVI-API gives VIP developers a head start on their work by making the requirements known ahead of time. For VIP users, choosing a verification solution that implements the RVVI-API instills confidence because it is based on a standard that has been vet by others. Fig. 1 below illustrates the architecture of a RISC-V processor testbench that is based on RVVI.

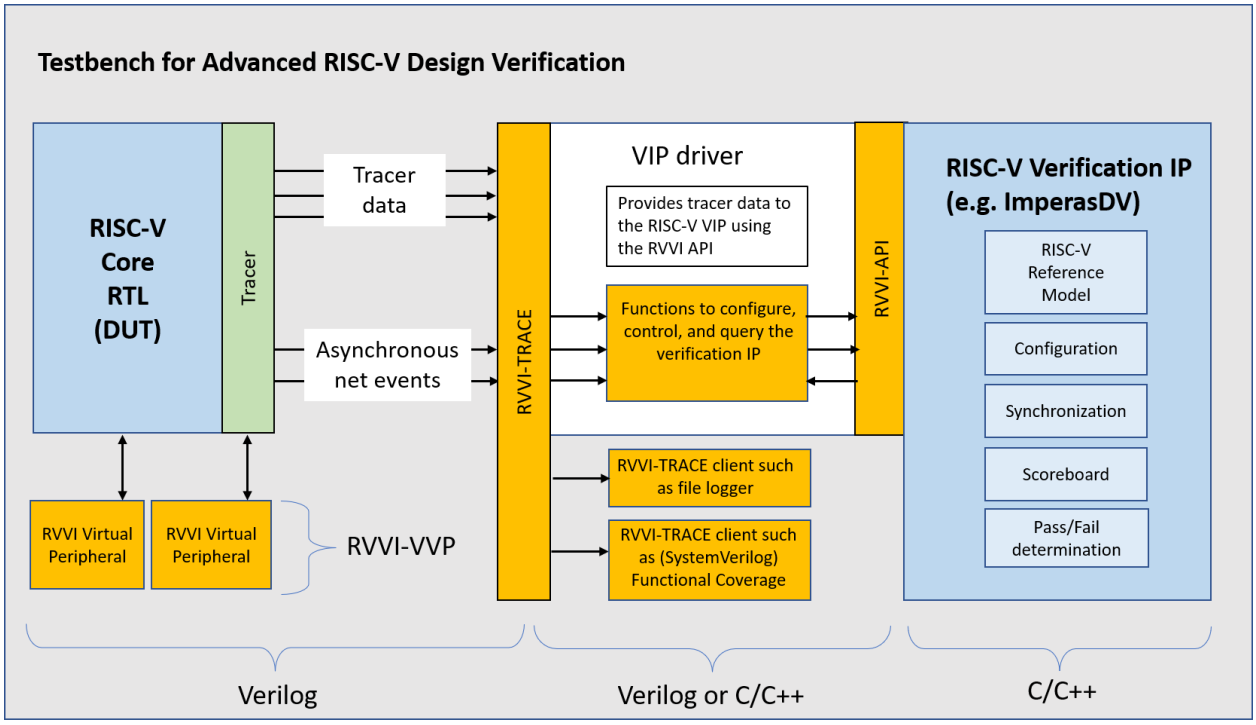


**Fig. 1: RISC-V Processor Testbench using RVVI**

## III. Debug Module Co-Simulation

As previously mentioned, the purpose of the Debug Module is to provide control and visibility into the state of a RISC-V processor core to allow for debug of low-level software and hardware. The architecture for a RISC-V Debug Module has been specified [4] by RISC-V International, however like many facets of RISC-V it permits flexibility in the implementation. Fig. 2 below presents the debug system overview from [4].

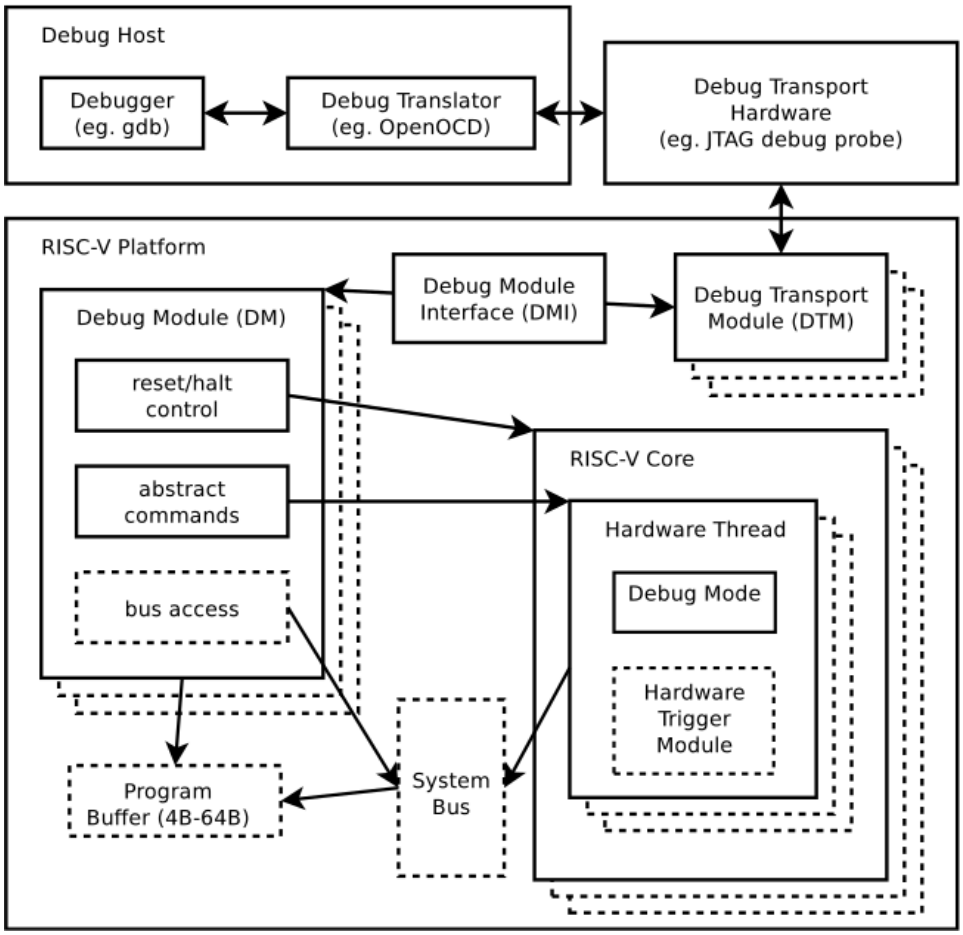


**Fig. 2: Debug System Overview**

AMD has chosen to implement the following features of the Debug specification for the MicroBlaze™ V:

### A. *Reset/halt control*

- Enables the core to be halted (program execution paused) so that debug operations can be carried out
- There is also the concept of reset halt request, whereby a core immediately enters a halted state once the reset signal is removed

### B. *Abstract Commands*

- These provide the Debug Module with access to the processor's general-purpose registers (GPRs). Both read and write access is available.

### C. *Program buffer*

- Enables the debugger to execute an arbitrary program on the halted hart.

A block diagram of AMD's processor verification testbench is shown in Fig. 3 below. Interaction between the Debug Module and the processor under test occurs in isolation from the RISC-V processor verification IP, shown on the right. In order to use ImperasDV to verify Debug Mode activity, this interaction must be conveyed to the VIP. The point of communication which must be modified is the RVVI-TRACE interface as this is the sole communication path between the processor under test and the VIP.

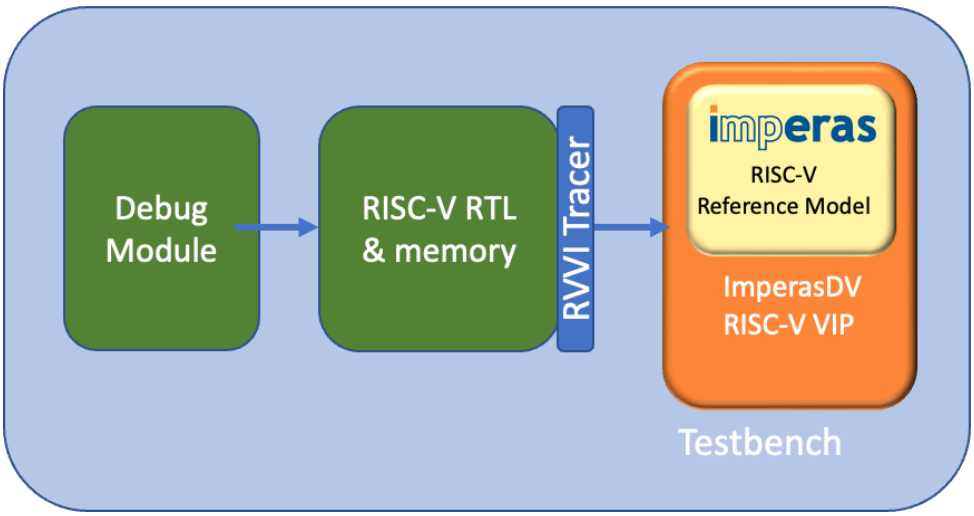


**Fig. 3: Debug Module Co-Simulation testbench**

## IV. Recognizing Debug Mode entry and exit

The RISC-V Debug specification defines several mechanisms by which a processor can enter debug mode. A common mechanism is via the execution of an EBREAK instruction. The EBREAK instruction causes a synchronous exception to occur, and control to be transferred to the debugging environment. Fig. 4 below illustrates a subset of the RVVI-TRACE data that is sent to the verification environment when an EBREAK instruction occurs. Using this data, as well as the value of the Debug CSR (DCSR) register, the processor verification IP can infer that the core has entered debug mode, and the next instruction executed will be in Debug Mode. However, in order to verify the correct execution of instructions in Debug Mode the location of the debug program must be available to the verification environment. In some implementations the start address of the debug program is provided using a bus that is external to the core.

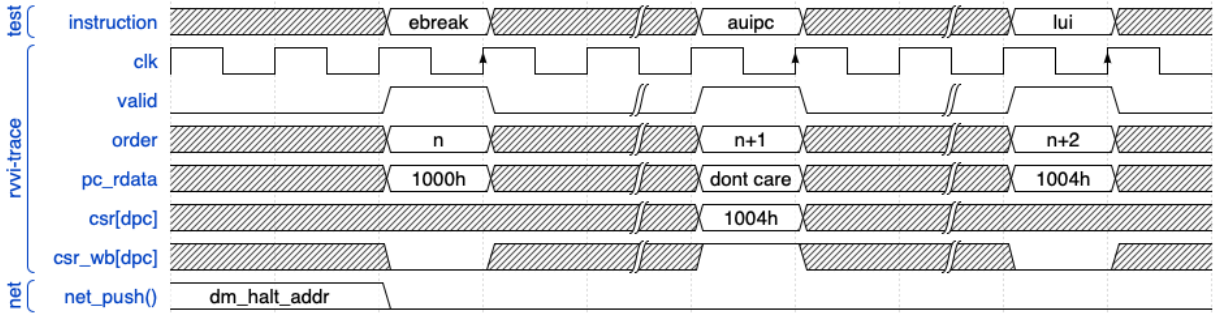


**Fig. 4: Debug Mode entry via EBREAK instruction**

In AMD's implementation, the haltreq (halt request) signal is one method used to enter debug mode. Haltreq is an external input to the core and its assertion/de-assertion happens asynchronously to the program execution. It may be asserted at any point between instructions or during the execution of an instruction, which can span multiple clock cycles. Changes to asynchronous nets are signaled to the verification IP using the RVVI-TRACE net_push() function. The net_push() function adds information about the changed signal (signal name, new value) into a queue. The queue is flushed by the reference model at the time of an instruction retirement. At this point the set of net changes must be resolved with the processor state changes provided on the other RVVI-TRACE interface signals and used to make a decision about the RTL's correctness.

In AMD's implementation there is an additional challenge: instructions executed in debug mode are not fetched from a predetermined address in memory, but are injected directly into the processor's pipeline. Using only the information provided in the RVVI-TRACE interface it was impossible for the verification environment to determine when an instruction had been executed in Debug Mode.

The proposed solution to this problem is to simply add a new signal called RVVI.debug_mode to the RVVI-TRACE interface. When the method of injecting instructions into the pipeline is used, the tracer module must assert this signal each time an instruction is retired in Debug Mode. This is done synchronously with the assertion of the signal RVVI.valid (indicting a valid RVVI transaction) and is independent of the haltreq signal, which is still provided asynchronously using net_push().

Fig. 5 below illustrates the use of the RVVI.debug_mode signal. Using this method, debug mode entry is indicated by a valid RVVI transaction where RVVI.debug_mode is 1. Exit from debug mode is indicated by a valid RVVI transaction where RVVI.debug_mode is 0. With the addition of this signal to RVVI-TRACE, AMD has been able to use constrained-random testing to verify instruction execution in Debug Mode. In these tests the timing of entry into and exit from debug mode are randomized and occur multiple times during the execution of a program. The testbench now has adequate information to verify the processor's behaviour under these conditions.

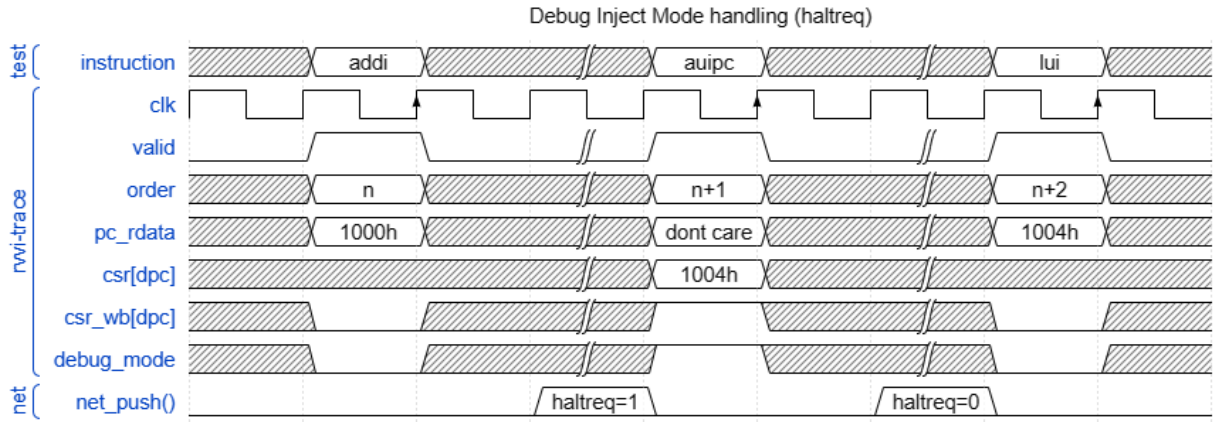


**Fig. 5: Debug Mode entry via haltreq (inject mode)**

## V. Verifying Debug Module Abstract Commands

The RISC-V Debug specification defines a set of abstract commands [4], whose purpose is to provide the debugger with access to the internal state of the core being debugged, and the ability to execute arbitrary instructions on the halted core using the program buffer. While most abstract commands are optional, the access register command is mandatory although the implementor can choose to which set of registers to provide access. In AMD's implementation, the access register command gives the debug module the ability to read from and write to the processor's GPRs (General Purpose Registers). It also provides the ability to execute the program buffer.

Abstract commands and the program buffer are implemented using a series of registers which are part of the Debug Module. The debug interface used to carry out this functionality is a series of signals connecting the Debug Module and the RISC-V processor under test. As shown in Fig. 2, this debug interface is completely separate to the RVVI-TRACE interface connecting the RISC-V processor and its testbench.

As part of Debug Module and debug mode verification, AMD would like to use the testbench's processor reference model to validate the processor's response to abstract commands. Specifically, to confirm that the values of the GPRs being read via abstract commands are correct, that values written to the GPRs via abstract commands will take effect, and that the instructions inserted via the program buffer are executed correctly. The verification environment is well-suited for this task, however initially, it was unable to do so because the only interface between the processor under test and the testbench is RVVI-TRACE, and RVVI-TRACE had no mechanism for conveying debug module activity.

To address this problem a new Debug Module interface is proposed as an addition to the RVVI-TRACE standard. The RVVI Debug Module interface is a SystemVerilog interface that is instantiated inside the RVVI-TRACE

interface. Implementation of the Debug Module interface is an optional part of the standard. The source code for the RVVI Debug Module interface is shown below in Fig. 6.

```
interface dm
#(
    parameter int ILEN   = 32,  // Instruction length in bits
    parameter int XLEN   = 32,  // GPR length in bits
    parameter int FLEN   = 32,  // FPR length in bits
    parameter int VLEN   = 256, // Vector register size in bits
    parameter int NHART  = 1,   // Number of harts reported
    parameter int RETIRE = 1    // Number of instructions that can retire during valid event
);
    //
    // RISC-V DM signals
    //
    wire                      clk;                              // Interface clock
    wire                      rd;                               // read
    wire                      wr;                               // write
    wire [31:0]               address;
    wire [31:0]               data;

    bit  [(XLEN-1):0]         store      [127:0];               // Storage for DM registers

endinterface
```

**Fig. 6: RVVI Debug Module interface**

To enable verification of Debug Module operations, the tracer must be updated to inform the testbench of read and write operations to Debug Module registers. It does this using the rd, wr, address and data signals in the RVVI.dm interface. The processor verification IP must maintain a model of the Debug Module registers in the RVVI.dm.store array. The testbench can use this register model to verify the contents of the actual Debug Module registers as they are updated in the RTL. The following example demonstrates the verification of a single access register command whereby the Debug Module is able to read the value of one of the processor's GPRs.

1. An abstract command to read a GPR is configured by writing to the Debug Module *command* register.
2. The tracer informs the processor VIP of the write to *command* using the RVVI.dm interface.
3. The processor VIP updates the shadow copy of the *command* register in RVVI.dm.store.
4. The processor VIP parses the abstract command and obtains the value of the specified GPR from the reference model using the RVVI-API function rvviRefGprGet [5].
5. The processor VIP updates the data register in RVVI.dm.store with the GPR value.
6. The testbench can use the GPR value in RVVI.dm.store to verify the value of the corresponding register in the Debug Module RTL.

## VI. Future Work

The proposed changes to RVVI have not yet been included in the public standard. Additional testing of the proposed changes with other RISC-V processor designs is needed to ensure that the changes are adequate and universally applicable. In particular, AMD implemented a subset of the Debug Module abstract commands (access register command) and the requirements imposed by the other abstract commands (quick access, access memory) [3] must also be considered.

## VI. Results and Conclusion

With the proposed changes to RVVI-TRACE, AMD is able to run testcases where Debug Mode operations are generated using constrained-random stimulus and can effectively verify the processor's response to Debug Mode abstract commands. The addition of the RVVI.debug_mode signal and the Debug Module interface represent an incremental improvement to the RVVI-TRACE standard. This enables RISC-V processor verification IP to provide more value during Debug Mode verification and operate in a testbench that also includes the Debug Module RTL. The RVVI standard continues to evolve as the discipline of RISC-V processor verification matures. The open nature of RVVI makes these best practices available to the wider community of RISC-V processor developers.

## References


[1] RISC-V Verification Interface https://github.com/riscv-verification/RVVI
[2] OpenHW Group on GitHub https://github.com/openhwgroup
[3] RISC-V Debug Specification, section 3.8.1
[4] RISC-V Debug Specification https://github.com/riscv/riscv-debug-spec/blob/master/riscv-debug-stable.pdf
[5] RVVI API https://www.riscv-verification.org/docs/rvvi/doxygen/rvvi-api_8h.html