embeddedworld2023
Exhibition&Conference
...it's a smarter world

Advanced RISC-V verification methodology projects

Jon Taylor, Aimee Sutton
Imperas Software
Oxford, UK
jont@imperas.com; aimees@imperas.com

Mike Thompson
OpenHWGroup
mike@openhwgroup.org

I. ABSTRACT

The open standard of RISC-V offers developers new freedoms to explore new design flexibilities and enable innovations with optimized processors. As a design moves from concept to implementation new resources are appearing to help with standards for testbenches, verification IP reuse and coverage analysis. RISC-V offers every SoC team the possibility to design an optimized processor, but this also implies the SoC design verification teams will need to address the challenge and complexity of processor verification.
This paper outlines open standards and methodologies that assist in both the efficiency and support for the growing community of RISC-V adopters.
Key aspects include:

- Test Bench integration standards to support SystemVerilog flows based on traditional SoC techniques extended for RISC-V processor design verification.
- Coverage methodologies that support the complexities of process design with asynchronous events including interrupts and debug operations, plus hardware configurations including Out-Of-Order pipelines, vector extensions and custom instructions.

Based on examples from several popular open-source cores this paper will provide guidelines that can help both open-source and commercial projects address the RISC-V functional verification challenge.

I. INTRODUCTION

The popularity of the RISC-V instruction set architecture (ISA) for microprocessors is growing, and with it is the need for, and interest in, processor-specific design verification techniques. Until recently, CPU IP was generally provided from a small set of companies. As RISC-V has grown in popularity, many more engineers are getting involved in processor design. Even prior to this, the cost of SoC verification was half the total cost of the design while a recent report[1] shows that even design engineers spend half their time on verification. Introducing custom or customized CPUs introduces additional verification challenges. Industry standard techniques such as constrained-random generation and functional coverage can help, but it is inefficient for every designer to create their own methodology and integration of verification IP components.

OpenHW group was created to develop open source IP, designed and verified to industry standards and ready for industrial deployment, collectively known as the CORE-V family [2]. This includes verification of their cores, and the organization includes a Verification Task group with a focus on ensuring that the CORE-V family of open-source RISC-V processor cores is verified to industry standards, so that anyone who adopts these cores can feel confident that they are ready for silicon. The verification environment used to verify the CORE-V cores is known as CORE-V-VERIF [3] and it too is an open-source artifact, available on GitHub.

The first and second generations of OpenHW Group's CORE-V-VERIF utilized a C Reference Model (RM) of the processor embedded in a UVM testbench. The reference model and the Device Under Test (DUT) were run in lock-step, each executing the same program, while the testbench compared the internal state of the two at the retirement of every instruction. This became known as the step-and-compare methodology. Used together with a random instruction stream generator and UVM agents providing peripheral stimulus (such as interrupts), it was an effective method of achieving verification closure for the CV32E40P RISC-V processor RTL.

Although successful, the time and human effort required to develop the first and second generation CORE-V-VERIF environments and achieve verification closure was high. The OpenHW Group was developing ambitious plans to achieve the same level of verification for at least five additional RISC-V cores, and it was clear that a more efficient methodology was required.

The experience with these verification projects led to the observation that some components should be common to all RISC-V processor testbenches, and common components should conform to standard interfaces. This led to the development of RVVI: the RISC-V Verification Interface. In

turn, the availability of standard interfaces made it possible to develop verification IP that implements these common components. These innovations have contributed to the third generation of the CORE-V-VERIF environment which is in use today with multiple RISC-V core developments.

Building on these standards is also a need for functional coverage of the design. This coverage is a way of measuring how much of the design has been comprehensively tested (as opposed to compliance testing which is far from exhaustive) . Imperas has developed functional coverage for much of the RISC-V ISA, including many standard extensions. With RISC-V supporting customisation, again a flow is needed to allow designs with custom instructions to add their own functional coverage points.

Functional coverage for the ISA itself is not enough to fully validate a design, since there are architectural behaviours beyond just instruction execution. Events such as interrupts, aborts or debug requests can interrupt the program flow asynchronously and this often exposes bugs. Thorough verification requires a process for testing these.

The following sections will provide greater detail into the evolution of the CORE-V-VERIF environment and RISC-V verification methodology. The first and second generation CORE-V-VERIF environments will be described, along with the highlights and shortcomings of each. Next the RISC-V Verification Interface (RVVI) will be described in detail, with an explanation of how it starts to address the shortcomings mentioned previously. The evolution continues with the introduction of verification IP to promote reuse, improve checking, and gain further efficiency. This lead to the present day third generation CORE-V-VERIF environment which will be shown to demonstrate the successful application of these techniques and provide context for future work. We then discussion the importance of functional coverage and the riscvISACOV project developing reusable functional coverage for RISC-V.

II. THE FIRST GENERATION CORE-V-VERIF ENVIRONMENT

The OpenHW Group develops open-source design and verification IP centered on the CORE-V family of RISC-V cores and related IP. The verification environments for the CORE-V cores are maintained as open-source artifacts in the CORE-V-VERIF GitHub repository[3]. The first generation CORE-V-VERIF environment is illustrated in Figure 1 below.

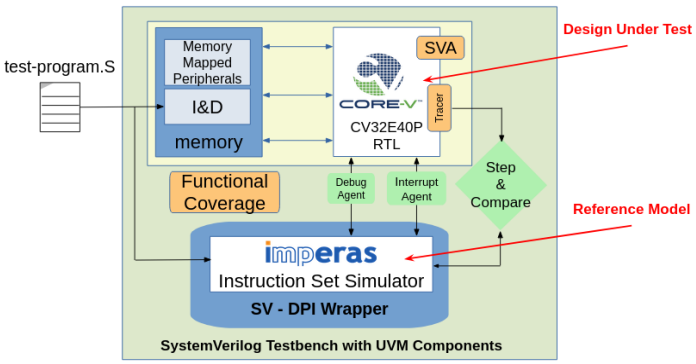

Figure 1: First Generation CORE-V-VERIF environment

A. *Reference Models in Verification*

As illustrated in Figure 1, a common strategy in simulation-based verification is to integrate the DUT and a reference model (RM) in a testbench and compare the state of the DUT and RM at specific times during the simulation. In processor core verification the state variables of interest will typically be the program counter (PC), general purpose registers (GPRs) plus the Control and Status Registers (CSRs). The most natural time to compare the state of the reference model and DUT is when an instruction has been *retired*. That is, execution of an instruction has completed and the values of the PC, GPRs and CSRs have been updated to reflect the effects of the instruction.

Conceptually, the theory of operation of such a testbench is as follows:

- Both the DUT and RM execute the same program. Note that in the CORE-V-VERIF testbench, these programs can be manually written by a human or generated by a random instruction stream generator (not shown in Figure 1).
- Non-programmatic interrupts to normal program flow such as external interrupts and debug requests are sent to both the DUT and RM. (Note: these “non-programmatic interrupts to normal program flow are termed “asynchronous events” in the remainder of this paper.)
- As each instruction retires, the state variables of the DUT and RM are compared. Mismatches are reported as errors. This method is known as step-and-compare.

In a typical processor core, the state of the PC, GPRs and CSRs are not accessible externally. Additionally, due to the implementation of the processor’s internal pipeline, the values of these state variables may be updated at differing clock cycles as instructions are executed. Exposing the state of the DUT to

the testbench is the job of a module called a **Tracer**[1]. An important consideration for a Tracer is ensuring that all aspects of the core's state are presented in a way that allows the consumer of Tracer outputs to attribute the state to a specific instruction.

In the first generation testbench, the Tracer used was an ad-hoc behavioral model bound to the RTL exposing all processor states such as GPR, PC and CSR values. This Tracer lacked a documented interface and required frequent changes in both the Tracer itself and the Step-and-Compare testbench logic.

The design and implementation of the reference model is a key consideration for this testbench architecture. An Instruction Set Simulator (ISS) is often used as an RM. Often developed and/or used by software development teams, an ISS can be custom-built in-house, a commercial product, or available as an open-source artifact. The CORE-V-VERIF environment used the Imperas RISC-V reference model[5,6] as the reference model for the embedded class cores. This was the strategy employed by the first generation CORE-V-VERIF environment. However, as we will see, it came with significant challenges, and the remainder of this paper will discuss these and introduce strategies for addressing them.

The most obvious problem encountered when using an ISS as a reference model is caused by the different abstraction levels of the ISS and DUT. The majority of DUTs are simulated at the Register Transfer Level, which by definition is a clock-cycle accurate description of hardware logic in which time is modeled as clock events. This is in contrast to a transaction-level model in which time is modeled as transaction cycles. In our case, an ISS of a processor core models time in terms of instructions. While an RTL model of a core may take multiple clock cycles to execute an instruction, depending on instruction fetch bus timing, the specific instruction being executed or the state of the internal pipeline; an ISS will usually execute all instructions in a single instruction cycle because the cycle timing of the physical interfaces to memory and the operation of the core's pipeline are abstracted away.

The abstraction inherent in an ISS is a powerful modeling technique, and it is highly effective in producing predictions of the core's state when executing a series of instructions which are not subject to asynchronous events such as interrupts. Nevertheless, a method must be found to maintain synchronization between the ISS, which operates in terms of instruction cycles, and the DUT, which operates in terms of clock cycles. The method used by CORE-V-VERIF to maintain this synchronization is called "step-and-compare" and will be discussed below.

A second problem when using an ISS as a reference model is the timing of "side effects". An instruction is said to have side effects if it updates one or more state variables which are not explicitly part of the instruction. For example, the CSR minstret is updated each time an instruction is retired. In most cases, side effects are easy to predict. In other cases, particularly in the event of an asynchronous interrupt, the timing of side effects in the ISS may differ from the RTL. This needs to be correctly managed to ensure that the DUT and RM don't diverge and give a false positive for a bug.

A. Step-and-Compare Architecture

[7] documents the original step-and-compare method used in the first generation of CORE-V-VERIF testbenches. An ISS is used as a reference model and is kept in sync with the RTL by running the RTL clock until an instruction is retired, at which point the RTL clock is stopped ("gapped") and the reference model is allowed to run until it retires an instruction. The testbench then compares the predicted processor state (from the reference model) to the actual processor state (from the RTL). This process repeats until the last instruction is retired and compared.

This technique works very well in the absence of external asynchronous events such as debug requests, interrupts, memory access delays and memory bus errors. In the first generation of CORE-V-VERIF, the ISS and RTL were connected to the same interrupt and debug request inputs. Due to the clocking system enforced by the step-and-compare logic, the ISS would always "see" these external events as being synchronous to the start of an instruction. Due to the internal micro-architecture of the RTL, these external events may or may not be "seen" as occurring before or after the start of the same instruction. This would inevitably result in differences in the predicted and actual processor states and a false-negative comparison would result.

To make matters worse, each asynchronous event needed to be handled as a unique event with core-specific code in the testbench to properly handle the behavior and side effects of these asynchronous events. This made the testbench "buggy" (prone to false-negatives) and difficult to maintain.

In response to this, a second generation of step-and-compare was developed to build upon and fix many issues with the first generation while maintaining the same verification effectiveness. For the purposes of this discussion the most purposeful improvements were to formalize the Tracer interface definition.

[1] Unrelated to the Efficient Trace for RISC-V specification (https://github.com/riscv-non-isa/riscv-trace-spec/releases)

III. THE SECOND GENERATION CORE-V-VERIF ENVIRONMENT

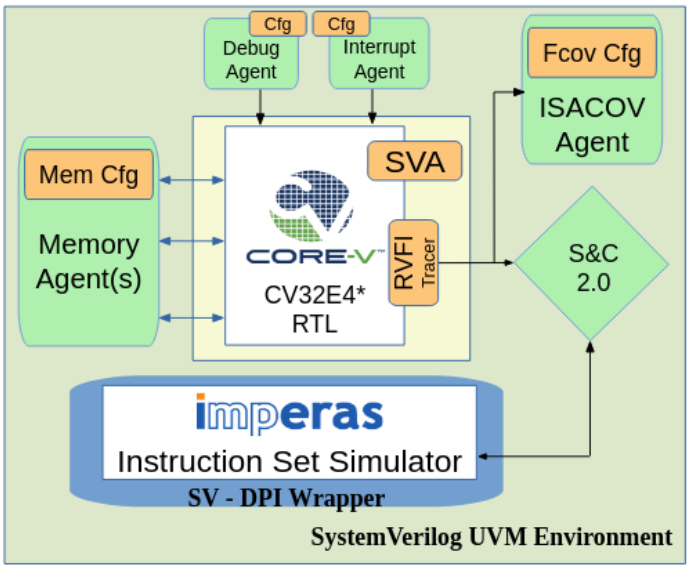

Figure 2: Second Generation CORE-V-VERIF UVM Environment

Figure 2 above, illustrates the changes implemented to create the second generation UVM environment:

- The ad-hoc Tracer used in the first generation is replaced by a Tracer conforming to an interface extended from the RVFI [8] specification. A new, simplified, second generation "step-and-compare 2.0" was developed.
- Asynchronous events such as interrupts and debug requests are connected to the core RTL (device under test), but not the ISS.

Each point from the above list is significant and will be discussed further below.

B. CORE-V Tracer using the RVFI + extensions

In the first generation testbench, monitoring of processor activity was enabled by a specific-purpose Tracer. The Tracer was bound to the RTL exposing all processor states such as GPR, PC and CSR values. This Tracer was difficult to maintain and implemented with an ad-hoc interface, requiring customized frequent changes in the Step-and-Compare implementation.

To address the issues with the ad-hoc tracer, the RISC-V Formal Interface (RVFI) specification was adapted to define the requirements for the processor Tracer. As much as was practical the RVFI was followed verbatim, with updates for extra requirements introduced by its usage in Step-and-Compare. The details of the specific CORE-V RVFI implementation extensions used is captured in the User Manual [9] of the CV32E40X, an RV32 processor core targeting embedded applications.

The CORE-V Tracer probes the internal design of the core to perform the following functions:

- Unambiguously identify instruction retirement.
- Unambiguous reporting of both synchronous traps and asynchronous exceptions, interrupts and debug requests.
- Expose the state of the core (PC, GPRs, CSRs).

Depending on the complexity of the core's pipeline, the logic required to implement the CORE-V Tracer ranges from trivial to complex. To simplify the CORE-V Tracer, it is typically implemented as a behavioral module that is bound (using SystemVerilog *bind*) into the RTL model. This simplifies the implementation because the coding is not constrained by synthesis requirements and can be maintained as one or more source files outside of the RTL sources.

A significant benefit of having a tracer interface specification is that it provides a well-defined blueprint of the requirements of a complete Tracer. This is more impactful than it at first appears. It can be difficult to fully specify the requirements of a Tracer apriori, without first having the experience of using a Tracer that does not fully fit your needs. It is often the case that such requirements can only be fully understood after several cycles of trial-and-error. This experience is what eventually led to the new open-standard RISC-V Verification Interface (RVVI) specification [10] which will be discussed in more detail later

C. Handling Asynchronous Events with CORE-V Second Generation Tracer

In the first generation of CORE-V-VERIF, the reference model "saw" asynchronous events such as interrupts and debug requests simultaneously with the RTL model. Due to differences in timing, this could mean that the RTL and reference model would "take" the interrupt on different instructions. This placed a burden on the step-and-compare logic in the testbench to ensure that the reference model would take the interrupt on the same instruction as the RTL.

In the second generation of CORE-V-VERIF, the reference model is not connected to asynchronous events. Thus, on its own, the reference model cannot determine when interrupts to normal program flow occur, such as external interrupts or external debug requests. The Tracer monitors and reports those events, and thus the "Step-and-Compare 2.0" logic can be used to *inform* the reference model to interrupt normal program flow, maintaining processor state lock-step with the DUT. This is possible because the tracer interface is explicitly defined to indicate when this information is presented to the reference model and all of the information required by the reference model is provided at the time the RTL retires an instruction.

For an example, consider what happens when a debug request is asserted. The CORE-V tracer interface defines two signals ***rvfi_dbg*** and ***rvfi_dbg_mode*** that are valid when an instruction is retired. Together these two signals indicate whether the core executed the retired instruction in debug mode and for the first instruction after entering debug, ***rvfi_dbg*** contains the debug cause. In addition to providing this information, the CORE-V Tracer specifies clear rules for how debug entry is recognized:

Debug entry is seen by the tracer interface as happening between instructions. This means that neither the last instruction before debug entry nor the first instruction of the debug handler will signal any direct side-effects. The first instruction of the handler will however show the resulting state caused by these side-effects (e.g. the CSR rmask/rdata signals will show the updated values, pc_rdata will be at the debug handler address, etc.).

The CORE-V tracer interface has similarly comprehensive and rigorous definitions for how an interrupt is signaled and how side effects are modeled. This information greatly simplified the step-and-compare logic required to keep the RTL and ISS state in sync.

However, this methodology left a serious verification hole: the reference model was not able to provide independent verification of the DUT's response to asynchronous events. For example, in a case where multiple interrupts are enabled and pending, it was unable to verify that the correct one (by priority) was taken. It simply mirrored the actions of the DUT. Checking this type of behaviour was left to other testbench components. Given the difficulty of validating responses to randomly generated asynchronous events, this checking was often incomplete. This serious deficiency was addressed in the third generation CORE-V-VERIF environment.

a) RVVI: The RISC-V Verification Interface

The work of the OpenHW Group Verfication Task Group (VTG) on the first and second generation CORE-V-VERIF environments led to the observation that certain components should be common to all RISC-V processor verification environments, and common components should be accessible through standard interfaces. These interfaces have now been formalized in RVVI [10], an open and evolving standard for functional verification of RISC-V processors. Two major components of RVVI will be discussed below: RVVI-TRACE and RVVI-API.

B. RVVI-TRACE

A common component that benefits from a standard interface is the previously-discussed tracer. Every RISC-V processor under test needs a tracer module in order to extract internal state information required for effective verification. While the implementation of the tracer is specific to each processor's microarchitecture, the requirements for the information that a tracer should provide are common, and are defined by the needs of the testbench. These requirements led to the creation of the RVVI-TRACE specification. RVVI-TRACE is specified as a SystemVerilog interface that connects the processor under test with the testbench.

When determining what the standard interface for tracers should look like it was natural to study formal interfaces being used in processor verification and see if they were appropriate for dynamic (simulation based) verification. It was clear that RVFI did not meet the needs of functional verification, as it was designed for formal verification, and heavily extended and modified for CORE-V-VERIF. The CORE-V extensions to it were insufficient for a standard interface as they were specific to the requirements of the CV32E4* cores and did not anticipate the needs of the full set of RISC-V ISA variations. For example, RVFI did not have a method of signalling more than one register change (side effect) per instruction retirement, something which does happen in the RISC-V Zc extension. Today the RVVI-TRACE interface addresses this and other issues with RVFI, and can be seen as a natural evolution and extension into dynamic verification of RVFI.

Another key feature of RVVI-TRACE is the mechanism for handling asynchronous inputs to the processor. RVVI-TRACE contains a SystemVerilog queue that is used to store multiple net changes that occur during the interval between instruction retirements, as well as when those changes occurred. This information is now available to the reference model or checker for validation of the processor's response to these events.

For the CORE-V-VERIF environment, the benefits of adopting RVVI-TRACE further extend the benefits realized by the development of the initial CORE-V tracer interface. There is still the existing benefit of knowing upfront what are the requirements for an effective tracer and being able to clearly communicate these to the design team. There is the additional benefit of having an interface that supports the full suite of RISC-V ISA subsets so there is no need to modify it between different processor projects. This, in turn, permits reuse of any component that is a client of the RVVI-TRACE interface.

C. RVVI-API

Earlier we explained that comprehensive RISC-V processor verification requires a behavioural reference model of the processor. The reference model provides an independent representation of the processor's internal state. It is subject to the same configuration and initial conditions and executes the same program as the DUT. The model should have the ability to run in lock-step with the DUT so that the two states can be continuously compared and bugs can be identified at the time they occur. This makes it faster to identify the failure and avoids unnecessary compute time running tests beyond a failure point (which happens in a log-compare based approach) In addition to the reference model, a processor verification environment needs a component to perform the comparisons between the model and the DUT, to keep track of any mismatches in state.

This set of requirements led to the development of RVVI-API: a set of functions that must be implemented in the processor verification IP and supporting testbench components in order to comprehensively check the behaviour of the RISC-V processor under test.

The diagram in Figure 3 below illustrates a canonical RISC-V processor verification environment using RVVI and RVVI-compliant processor verification IP.

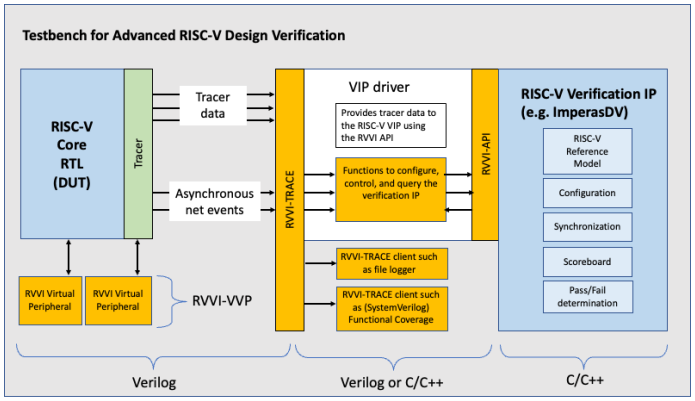

Figure 3: Testbench for Advanced RISC-V processor verification using RVVI

The introduction of the RVVI-API into CORE-V-VERIF has addressed some of the shortcomings and issues encountered with the previous step-and-compare environments. One of these areas is the configuration of the RISC-V verification IP and processor reference model. RVVI-API specifies functions (Figure 4) to configure specific memory regions, registers, or register fields as volatile.

Volatile control and status registers (CSRs) and memory regions are those that change asynchronously to the program execution. They often require a cycle-accurate representation of the processor to model accurately. An example of a volatile CSR is a counter that increments every clock cycle. An example of a volatile memory location is the address space used by a memory-mapped peripheral. T

In the first and second generation CORE-V-VERIF environments the verification IP and processor reference model had no notion of volatility. It was up to the step-and-compare logic to maintain a list of register comparisons to discard and to populate the correct register values in the reference model. This was necessary to ensure that the program running on the model and the processor core would exhibit the same behaviour.

To address this problem, RVVI-API specifies functions to mark a register, register field, or memory region as volatile. It is now the responsibility of the reference model to keep track of volatile addresses and to ensure that the contents of these regions stay consistent with the DUT. RVVI-API also specifies functions to inform the reference model (and/or the VIP that encapsulates it) about processor read/write activity. When a volatile region is accessed the data from that location can be propagated to the reference model's memory. This ensures that the test program will run as expected.

```
import "DPI-C" function int
rvviRefCsrSetVolatile(
    input int hartId,
    input int csrIndex);

import "DPI-C" function int
rvviRefMemorySetVolatile(
    input longint addressLow,
    input longint addressHigh);
```

Figure 4: RVVI-API volatile functions

THE THIRD GENERATION CORE-V-VERIF ENVIRONMENT

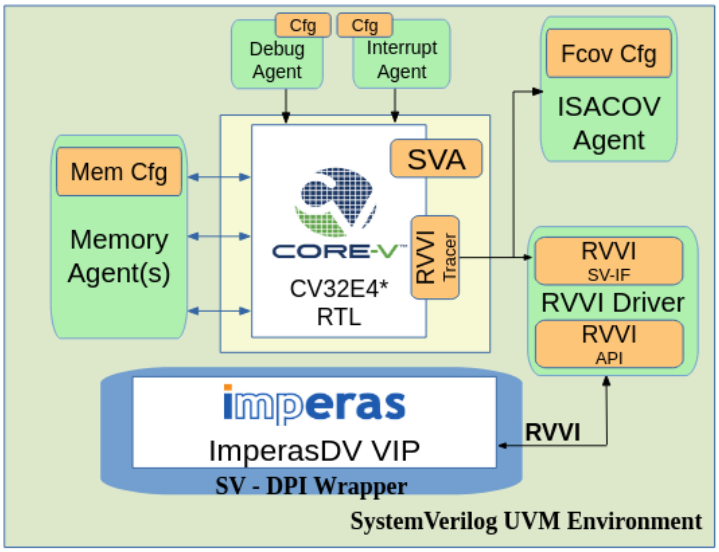

Figure 5: The third generation CORE-V-VERIF environment

Figure 5, above, illustrates the changes implemented to create the third and current generation UVM environment:

- The CORE-V tracer has been replaced with an RVVI-compliant tracer
- The Imperas Reference model has been replaced with ImperasDV verification IP (VIP) that incorporates the reference model
- The step and compare logic has been replaced by RVVI

The benefits of adopting an RVVI-compliant tracer have been discussed in previous sections. At the time of writing, the migration from CORE-V's use of RVFI with extensions to RVVI is an ongoing activity.

The most impactful change has been the replacement of the reference model and the step-and-compare logic with a piece of verification IP. The ImperasDV VIP encapsulates a reference model of the target processor, and implements the functions specified by RVVI-API. It performs the internal state comparisons between the reference model and the DUT using information from the RVVI-TRACE interface, and keeps track of those results in an internal scoreboard. This eliminates the need for the error-prone and complex step-and-compare logic being hand coded in the testbench. Since asynchronous events are now communicated to the VIP using RVVI-TRACE the processor's response to these can now be independently validated.

The following section contains an explanation and example of how an architectural reference model can be used to provide validation of a processor's handling of asynchronous inputs such as interrupts and halt requests.

a) Handling Asynchronous Events with ImperasDV

As previously discussed, one of the most challenging tasks in processor verification is maintaining a consistent view of program execution between an architectural and micro-architectural representation. Using the predictions and validations from an architectural model in order to verify an RTL implementation is highly desirable, but it is a challenge to provide useful data and useful predictive behavior.

Let's consider a very simple example: how to determine the correct point during program execution to apply an external asynchronous event such as an interrupt. When applying an interrupt to an architectural representation it can be taken immediately (if enabled) upon receipt, causing the processor to take the exception and begin execution at whatever is defined as the interrupt handling address.

In a micro-architectural implementation, it is not so simple. For example, the interrupt input logic may contain oversampling to ensure that the interrupt logic is observed to be active for N clock cycles. Once the logic has decided that an interrupt is active, it then has to be merged into the instruction pipeline at an appropriate time. It may be decided that it is wasteful to discard a complex instruction which executes for 32 cycles if it has already been executing for 28 of those. It's better to take the interrupt latency penalty of 4 cycles, rather than discard and lose 28 cycles. This is one of many micro-architectural performance decisions that must be considered in order to ensure the best throughput versus responsiveness.

Using ImperasDV to analyse legal scenarios recently revealed a bug in the OpenHW Group's CV32E40X processor core. It involved the following sequence of events:

A set of randomly-generated external interrupt signals have been propagating into the local interrupts of the processor core. These interrupts are masked by two levels of logic, firstly there is the MIE (Machine interrupt Enable) CSR, and the global interrupt enable field of the MSTATUS register (MSTATUS.MIE). An interrupt is detected by evaluating the following expression:

```
IRQ = MSTATUS.MIE && ((MIE & MIP) != 0x0);
```

Figure 6 Expression for detecting a valid interrupt

The expression states that for an IRQ to be pending and enabled we must have the equivalent positional bits True in both MIE and MIP, and the global interrupt enable MSTATUS.MIE must also be True.

The upper 16 MIP bits (local Interrupt 0-15) are a direct representation of the interrupt pins on the core, bearing in mind that there is clocked logic to sample these pins.

When the mret instruction is executed, the expression in Figure 6 evaluates to true. However, since the interrupt pins have been toggling during the interval since the previous instruction was retired it is unknown which interrupt should currently be active. The verification environment must handle these events and ensure that the DUT's response is legal.

In this instance, the DUT actually did not service any interrupt, it executed the ebreak instruction and entered debug mode instead. Since this did not match any of the legal scenarios an error was flagged. This bug is now captured in the OpenHW Group's GitHub issue tracker [11].

IV. FUNCTIONAL COVERAGE

The methodology discussed so far checks functional behaviour, but it does not provide any evidence about how much of the design's possible behaviour has been tested. A verification plan should have a comprehensive list of all the behaviours that need to be tested, and functional coverage provides a way to measure that.

Developing functional coverage requires a comprehensive list of behaviours to be covered. For RISC-V, this is further complicated by optional extensions and customisations that further change legal behaviour of a design. The functional coverage required for a RISC-V processor verification project can be divided into two categories. The first is coverage of the RISC-V ISA. This involves covering the instructions and their operands as specified in the ISA for the extensions being used. Considering the RV64 ISA and some of the more common

extensions, if we just look at the number of instructions to be covered:

- Integer: 56
- Maths: 13
- Compressed: 30
- FP-Single: 30
- FP-Double: 32
- Vector: 356
- Bitmanip: 47
- Krypto-scalar: 85
- P-DSP: 318

For RV64 that is 967 instructions. Each instruction requires coverpoints and covergroups, taking a few 10s of lines of code. So that means perhaps 10000 to 40000 lines of code to be written and tested. It's important to note that this coverage code is not specific to any processor implementation, it is defined by the ISA specification and should be resuable. While it is possible to manually create coverage points, if these are closely coupled to the RTL design, they are unlikely to be portable between projects and the manual nature of creating them means they will have their own need to be debugged and validated.

The second category of functional coverage code addresses custom core features such as privilege ISA items, interrupts, debug block, pipeline, custom extensions and CSRs. These covergroups and coverpoints are specific to the processor's implementation and have a lower potential for reuse.

OpenHW group has created an Advanced RISC-V Verification Methodology (ARVM) working group [12] to develop and review an approach for functional coverage for RISC-V. As part of this group, Imperas proposes an approach where the coverage for the ISA can be automatically generated from a machine-readable description of the RISC-V ISA. The coverage data is sampled from the RVVI-TRACE interface, making it seamless to use in any project that has an RVVI-compliant tracer (see Figure 7). This approach is design agnostic and can be used with the previously described lock-step-compare simulation to ensure operation is correct at the same time as coverage data is captured. Imperas has made functional coverage for the RV32I base ISA available via the RISC-V ISACOV project on GitHub[13], with additional coverage for other extensions being available under a commercial license.

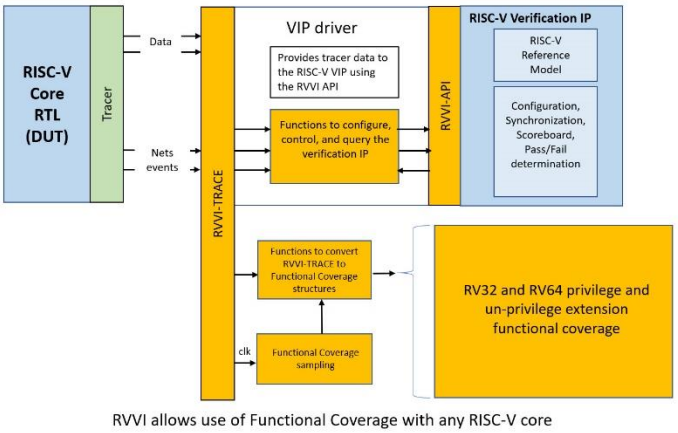

Figure [7] shows how data sampled by RVVI-TRACE can be used to measure coverage.

The Imperas approach avoids both the manual effort and high potential for error involved in developing a large amount of functional coverage code. This approach allows for better reuse and is much more scalable. Using riscvISACOV it is easy to select the appropriate coverage points to match the design configuration (ie RV32 vs RV64, which ISA extensions are chosen etc).

V. FUTURE WORK

The CORE-V-VERIF verification environment continues to evolve both in response to new requirements from new cores and to on-going learnings from current and previous verification efforts. The OpenHW repository of cores continues to grow, with a recent contribution being an applications core from Harvey Mudd College. The methodology described above is being applied to the core and it has already found a number of bugs.

For functional coverage, the plan is to extend this to include CSRs and data hazards. Testing the Havey Mudd core will also allow for development of coverage for complex areas such as Memory Management Unit (MMU) and other features found in applications cores.

VI. SUMMARY

The OpenHW Group's Verification task group has been a pioneer in the field of RISC-V processor verification. Through the CORE-V-VERIF environment we have employed different approaches and evaluated the merits and shortcomings of each. With each generation the CORE-V-VERIF environment has improved to become more robust, more reusable, and ultimately better at finding RTL bugs. The current generation of CORE-V-VERIF uses RISC-V processor verification IP enabled by the RVVI to realize a comprehensive verification methodology that encompasses asynchronous peripheral events that occur

randomly during program execution. This is the state of the art methodology at present, however the verification task group members are highly motivated to continue to innovate and advance the practice of RISC-V processor verification.

VII. ACKNOWLEDGMENTS

The authors would like to recognize the participation and contribution to this work of several of the OpenHW Group collaborators: Simon Davidmann and Aidan Dodds of Imperas Software, Greg Tumbush of EM Microelectronics, Steve Richmond, formerly of Silicon Labs, and Dolphin Design.

VIII. REFERENCES

[1] https://semiwiki.com/eda/324443-the-state-of-ic-and-asic-functional-verification/

[2] https://github.com/openhwgroup/core-v-cores

[3] https://github.com/openhwgroup/core-v-verif

[4] https://docs.openhwgroup.org/projects/core-v-verif/en/latest/quick_start.html

[5] https://en.wikipedia.org/wiki/OVPsim

[6] https://www.imperas.com/

[7] *Jump Start your RISCV project with OpenHW*. DVCon US, March 1-4, 2021 (Virtual) - https://www.imperas.com/articles/dvcon-2021-paper-jump-start-your-riscv-project-openhw

[8] https://github.com/SymbioticEDA/riscv-formal/blob/master/docs/rvfi.md

[9] https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/rvfi.html

[10] *RISC-V Verification Interface* - https://github.com/riscv-verification/RVVI

[11] *https://github.com/openhwgroup/cv32e40x/issues/665*

[12] OpenHW group Advanced RISCV Verification methodology https://github.com/openhwgroup/programs/tree/master/TGs/verification-task-group/projects

[13] RISC-V ISAcov https://github.com/riscv-verification/riscvISACOV