RISC-V Architectural Functional Verification

10xEngineers Final Report

Harvey Mudd College Engineering Clinic

Project Team

Hamza Jamal (Sr, Team Lead)

Ahlyssa Santillana (Sr)

Corey Hickson (Jr)

Marina Bellido (Jr-S)

Jordan Carlin (Jr-S)

Roman De Santos (Jr-S)

Vikram Krishna (Jr-F)

Project Advisor: Prof. David Harris

Project Liaisons: Huda Sajjad (F), Umer Shahid, Fatima Saleem

May 2025

ABSTRACT

This report outlines the development of an open source architectural functional verification suite for the RISC-V RVA22S64 profile as part of the 10xEngineers Global Clinic Project 2024-25. As a collaboration of Harvey Mudd College (HMC), 10xEngineers, Habib University (HU), UET Lahore (UET), and the OpenHW Group, this project aimed to achieve full architectural functional coverage of the CORE-V Wally rv64gc and rv32gc configurations to advance them to Technology Readiness Level 5. An additional goal was to produce reusable verification IP that could be adopted across various RISC-V processors, contributing to the evolving RISC-V ecosystem. To accomplish this, verification plans, coverpoints, and directed assembly tests were created to validate both privileged and unprivileged features. The project emphasized collaborative engineering practices across international teams and gave students hands-on experience in digital design and functional verification.

TABLE OF CONTENTS

1. Introduction

Since 2020, the Clay-Wolkin Fellowship at Harvey Mudd College has collaborated with the University of Nevada at Las Vegas, Oklahoma State University, and the non-profit Open Hardware Foundation to develop CORE-V Wally, a configurable and open source computer processor [1]. In processor development, design verification is critical to a project's success. The developers of Wally, with their commitment to an open source project, seek to employ free and open source software to the greatest degree possible. However, as discussed in Section 1.2, existing open source verification tools are insufficient. For this reason, this Clinic team and 10xEngineers are collaborating to produce this set of verification tools. While the initial deployment of the tools is happening on Wally, the team will decouple the two so that any design team with a compatible verification interface can use them as well, as discussed in Section 3.

This introduction covers the historical context and technical terms needed to understand the responsibilities associated with this project (1.1-1.5), gives the project statement (1.6), and a list of deliverables for this project (1.7). Section 2 enumerates some potential impacts of our project on the computer chip industry. Sections 3 and 4 describe the workflow associated with deliverables the team has been working on for this past school year: writing tests and covergroups for privileged and unprivileged instructions. Section 5 describes the steps taken to obtain 100% code coverage. Section 6 details the changes required in our toolset to ensure open source access for all users. Section 7 discusses the bugs discovered in CORE-V Wally as a result of this project and how those bugs were resolved. Sections 8 and 9 detail the project management practices throughout the year and

future work that could be taken up to improve both this verification suite and CORE-V Wally. Finally, Section 10 contains a list of the references used throughout this report.

1.1. A Primer on Computer Architectures

The architecture of a digital device refers to the specific methods and structures the device uses to process and convey digital information. Instruction set architectures (ISAs) describe the architecture of a processor in terms of the instructions that the processor should be capable of executing. One type of ISA, a reduced instruction set computer (RISC), defines a small set of common instructions rather than the many specialized instructions typically found in a complex instruction set computer (CISC) [2]. As such, a key feature of RISC architectures–as opposed to CISC ISAs such as x86–is that they execute one simple action per instruction. While this simplicity necessitates more instructions to accomplish the same task when compared to CISC architectures, the simplicity of the datapath allows a higher clock rate and lower power consumption, making up for the increase in program length.

Two major computer processor ISAs currently dominate industry use: ARM, a proprietary architecture developed by Arm Holdings; and x86, which is Intel's proprietary architecture. However, as they are both proprietary, obtaining or paying for licenses can be troublesome. Intel has only ever granted an x86 license to Advanced Micro Devices (AMD), leaving Arm Holdings a near monopoly over the rest of the market. ARM, being a RISC ISA, is attractive for its combination of high-performance chip designs at low power consumption. ARM also offers different configurations of their architecture for different use cases, like the Cortex A, M, and R series for Application, Microcontroller, and Real-Time, respectively. However, ARM IP licenses can be expensive and restrictive. Licensees can either purchase

the rights to pre-designed cores or a license to design their own ARM core, but access to the latter is so heavily restricted that only a handful of corporations worldwide have obtained this special license [3].

Alternatives to ARM and x86 (e.g., MIPS, POWER 9) have existed for decades, but none has ever come close to being truly competitive with it. However, in 2010, a team led by David Patterson in his Parallel Computing Lab at UC Berkeley developed a fifth-generation RISC architecture, RISC-V [5]. Unlike ARM and x86, RISC-V is free and open source. Nevertheless, it matches the performance and power consumption of its proprietary competitors, making it a good choice of architecture for applications ranging from embedded systems to high-performance computing. Over the past several years, RISC-V has gained popularity among industry giants and startups alike, such as Nvidia, Qualcomm, and Esperanto, with its popularity only slated to grow in the coming years.

1.2. Overview of the Digital Design Process

When bringing any System-on-Chip (SoC) to production, the chip is designed, verified, sent for tapeout, and then validated before finally being deployed in the desired application. Of these steps, verification is almost always the most time-intensive. Since chip fabrication, or “tapeout,” is very expensive–Apple spent over $1 billion on the M3 tapeout [5]–and time-consuming–typically requiring 3-6 months–a design validation (DV) team runs simulations of the chip to find bugs before tapeout. As long as a processor supports a given ISA, whether that be RISC-V, ARM, or MIPS, a standardized suite of tests could, in theory, be used to verify its adherence to the ISA’s specifications. Proprietary test suites currently exist, such as those offered by Breker Verification Systems and Synopsys, as do several open

source options. The most prominent of the open source options is riscv-arch-test, which has the backing of RISC-V International (RVI), the governing body of RISC-V, and many independent contributors. However, it has a high barrier to entry for contribution, meaning new tests are developed slowly, and many core features are not yet covered. Additionally, the coverage files are written in a non-standard YAML format virtually unused in industry, making it difficult to tell what is tested. As insufficient design verification is plaguing the development of RISC-V cores across the board [6], a robust open source test suite would improve design verification for RISC-V cores to reduce the barrier to entry in this already rapidly advancing industry.

1.3. Technology Readiness Metrics for Validation

The US National Aeronautics and Space Administration (NASA) defines a Technology Readiness Level (TRL) metric to grade devices, which the OpenHW Foundation has adapted to describe processors. CORE-V Wally, created by Prof. David Harris's VLSI Design Lab in collaboration with Oklahoma State University and the University of Nevada Las Vegas, currently satisfies TRL-4, which requires Power-Performance-Area (PPA) measurements from synthesis and/or a demonstration of preliminary application code [7]. The OpenHW Foundation requires a demonstration of TRL-5 for a core to be included in the Eclipse Foundation Working Group's IP listings. To be certified as TRL-5, a core must have design verification (DV) plans for all features of the design under test, 100% functional coverage of all items in the DV plan, 100% code coverage, all tests passing in regression, and no open issues.

1.4. 10xEngineers

10xEngineers is a business-to-business firm specializing in RISC-V design and verification services and Image Signal Processing (ISP). Their products and services include compilers and toolchains for RISC-V development, machine-learning models for custom hardware, and ISP algorithms. 10xEngineers' Cloud V is a RISC-V architecture testing platform that provides an on-demand continuous integration environment [8].

Although the tools we are developing are free and open source, 10xEngineers will benefit from investing in them by selling DV services built around them to RISC-V IP developers. By funding and contributing to this project, they will lead the industry in expertise on these tools. With RISC-V continuing to grow in popularity, and therefore the need for verification growing as well [9], this project will support RISC-V IP developers and set up 10xEngineers to become an important player in the commercial DV space.

1.5. RISC-V and Verification Terminology

The RISC-V specification divides instructions into classes called extensions [10]. The nomenclature generally follows the pattern of RV, the register width in bits (either 32 or 64, as of Dec. 2024), then a set of letters that denote the supported extensions. The base instruction set may be either I, for a 32-register system, or E for a 16-register system. Beyond the base instruction set, the M extension supports multiply, divide, and remainder instructions. The F, D, and Q extensions support single-, double-, and quad-precision floating-point arithmetic. The A extension introduces atomic (uninterruptible) instructions used for synchronization between multiple RISC-V cores operating in parallel. Most non–embedded application processors support the M, F, D, A, Zicsr, and Zifencei extensions,

e.g., RV64IMAFD_Zicsr_Zifencei. The letter G denotes a shorthand for the collection of all of the aforementioned extensions. If the base compressed instruction extension (C) is supported on top of the G extensions, then C is added to the list: RV64GC. RVA22S64, a profile for a full-featured application class processor, requires support for all extensions in RV64GC, plus some additional extensions: B (bit manipulation), Zicntr (hardware counters), Zicbom (cache management), and Sv39 (virtual memory). Wally conforms to this profile when all features are enabled, so we are designing our verification tools around RVA22S64.

There are two main levels at which a processor can be verified: architectural and microarchitectural. Microarchitecture refers to the specific sub-circuits that implement an architecture. For example, the datapath–the series of circuits that access, process, and store data in a processor–may be pipelined into an arbitrary number of stages. Additionally, designs may be superscalar, with replicated arithmetic units to execute multiple instructions at a time, or support out-of-order execution, where the order of instructions is changed to avoid data and hardware dependencies that limit simultaneous execution. Given that many different processor designs implement the necessary instructions, verification at the microarchitectural level requires testing of each specific path that data could take through the processor, which is specific to the device under test.

Architectural verification exists at a higher level of abstraction. The architectural level contains specifications for how any processor that adheres to the given architecture should behave under certain conditions. For example, any RISC-V processor that implements the RV32I extension should be able to correctly add two 32-bit numbers. An architectural verification system testing this feature must only determine that whenever an add instruction

is executed, it produces the correct sum and stores the result in the correct place without impacting other parts of the architectural state.

The verification tools designed by the 10xEngineers clinic team are intended for use with any RISC-V core following the RVA22S64 profile, regardless of implementation. Since architectural verification can be done without knowledge of the implementation-specific microarchitecture of a processor, we are developing our tools for architectural verification. As such, there are certain microarchitectural features that are intentionally excluded from verification, which represent the limitations of the project. Among these are caches, branch prediction, stalls, flushes, and forwarding outside of consecutive instruction hazards.

There also exist a few different verification paradigms. For example, formal testing uses mathematical analysis to prove that the system works as intended. This is feasible for relatively simple circuits, but is much more difficult for a device as complex as a CPU. Firms are working on formal verification at the processor level, but this is in relatively early development. On the other hand, functional verification runs specific tests on a device to give a reasonable expectation that all necessary functions give correct outputs. Key components of functional verification are coverpoints and tests. Coverpoints are conditions that must be sampled from the device under test to be confident that the entire functionality has been tested. Tests are intended to cause the coverpoint conditions to occur in the device under test, and can be either random or directed. Directed tests are written to intentionally cover the coverpoint, while random tests are generated with the hope that all coverpoints will be covered given enough tests. The approach of our tools is to generate directed tests with random values for anything not relevant to the coverpoint that is to be covered. For example,

if a test is generated to cover a certain value of source register 1, then source register 2 will be a random value. Section 3.3 describes the specifics of test generation.

For trivial digital circuits, it is possible for tests to exercise every possible combination of inputs and verify that the output is correct in every case. However, this becomes increasingly unrealistic as the device under test gains complexity. For an application processor like Wally, this is completely unattainable, requiring upwards of 1.8×10^{19} unique tests to exhaustively verify just one 32-bit R-type instruction, even ignoring the effect of CSRs on the system state.

Instead of testing every possible input, we choose a set of test cases that we think cover the most important and interesting situations. Deciding which cases to include is a human judgment made by the contributors and will be reviewed by others who did not write the tests. Sections 3.1 and 3.2 describe these decisions. If the system responds correctly to these cases according to the ISA standards, we consider the feature to be "covered." Overall, we test using a thorough, diverse subset of tests, which is intended to cover enough of the possibilities to catch all bugs.

1.6. Project Statement

The 2024-2025 HMC 10xEngineers Clinic team will develop a suite of RISC-V architectural functional verification test plans, coverpoints, and tests. This includes maintaining a complete list of features to be tested, creating covergroups that adhere to these test plans, and writing the tests to exercise all coverpoints in the defined covergroups. This project will develop open source coverage for all architectural features.

1.7. Deliverables

The list of deliverables for this project are as follows:

- Design verification test plans for each extension in the RVA22S64 profile and associated 32-bit versions
- SystemVerilog covergroups and coverpoints to meet all coverage goals enumerated in the test plans
- Directed assembly language tests to exercise all of the coverpoints, accompanied by any scripts used to generate them
- Application of the tests to the CORE-V Wally core running in lock-step with ImperasDV, a proprietary RISC-V reference model, to demonstrate 100% functional coverage, a major component of TRL-5
- This Final Report and accompanying Final Presentation and Poster detailing the work completed during the school year
- Extending test generation for code coverage, a type of verification that ensures that every line of the Hardware Description Language (HDL, e.g., SystemVerilog) code that describes the design is fully exercised
- Running other test suites against our coverpoints, such as those from Breker Verification Systems, which has recently partnered with the CORE-V Wally team
- Switching the lockstep reference model from ImperasDV, a proprietary software, to SAIL, which is the open source and official RISC-V reference, but is currently just starting work on lockstep operation

- Signature-based tests, which produce a signature to be checked against a reference without the need for a lock-step reference model–a vital feature for post-silicon design verification.

Because the RVA22S64 profile requires coverage of many instructions and extensions, responsibility has been split among the three student teams collaborating on this project. Figure 1.3.1 below provides a breakdown of the extensions that must be covered, their designated “owners,” and an expected completion date for each extension.

Test plan	**Description**	**Owner**
I	**Integer**	Shared
M	**Multiplication & Division**	Habib
Zba	**Bit Manipulation**	Habib
Zbb		
Zbc		
Zbs		
Zknd	**Scalar Cryptography**	Habib
Zkne		
Zknh		
Zbkb		
Zbkc		
Zbkx		
Zaamo	**Atomic Memory Operations**	Habib
Zalrsc		
Zifencei	**Instruction Fetch Fence**	HMC
Zicond	**Integer Conditional Operations**	Habib

F	**Floating-Point**	HMC
D		
Zfh		
Zfa		
Zicsr	**Control and Status Registers**	HMC
ZicsrM		
ZicsrS		
ZicsrU		
ZicsrF		
ExceptionsM	**Exceptions**	HMC
ExceptionsS		
ExceptionsU		
ExceptionsF		
ExceptionsZc		
ExceptionsZicboU		
ExceptionsZicboS		
ExceptionsZalrsc		
ExceptionsZaamo		
ExceptionsVM		
EndianM	**Endianness**	HMC
EndianS		
EndianU		
InterruptsM	**Interrupts**	HMC
InterruptS		
InterruptsU		
InterruptsSstc		

SsstrictM	**Ssstrict**	HMC
SsstrictS		
Zicntr	**Counters**	HMC
Zihpm		
Zca	**Compressed Instructions**	UET
Zcb		
Zcd		
Zcf		
PMP	**Physical Memory Protection**	UET
VM_PMP		
Svbare	**Virtual Memory**	UET
VM		
Svade		
Svadu		
Svnapot		
Svpbmt		
Svinval		

Figure 1.3.1: Features Covered by Verification Suite

2. Impact

RISC-V, in addition to being a royalty-free open source ISA, has also been demonstrated to be viable in commercial applications [11]. Open source technologies make it significantly easier to adhere to a standard across an industry, accelerating innovation and providing economic returns to its adopters as a byproduct [12]. The presence of an industry standard also makes it easier for small corporations and startups to grasp a foothold among the dominant competitors since they need not devote additional time and money to convincing customers to buy into a new standard.

Beyond the benefits of open source in general, RISC-V offers benefits of its own as an ISA. The first is that since it is a reduced instruction set computer (RISC), like ARM, it has the same benefits of low power consumption and high throughput. Under the typical ARM license, developers are prohibited from changing the ISA. These licenses are also usually expensive, as ARM sometimes charges a percentage of the selling price of the chip in the final product. However, the opposite is true for RISC-V, as there are no architecture licensing restrictions and zero royalties. A testament to the accessibility of open source lies in the RISC-V-based core Veyron V2, developed by Ventana Microsystems, which was reported at the 2024 RISC-V Summit to outperform AMD's EPYC Bergamo 9754 CPU [13]. As such, corporations that seek to begin development with RISC-V could save a great deal of time and money even while delivering a better final product, but this comes at the cost of losing access to ARM's robust ecosystem of tools and verification IP that offer fast adoption capability. The RISC-V community is working to develop a similar ecosystem for its architecture,

though, in its current state, it still falls short in some aspects, of which the most hindering is the short supply of good, cost-effective design verification (DV) tools and services [14].

When developing a processor in any architecture, only a small portion of the total labor goes toward the design of the chip; rather, the average ASIC team spends upwards of 50% of their total project time on verification. In specific industries, such as processor design, verification can take more than 80% of the total project time [15]. Additionally, even dedicated design engineers still spend almost half their time on verification [15]. As such, tools and services that speed up the verification process are likely to have a significant impact on the overall development cycle of a chip. Some proprietary verification tools are already on the market, such as those of Breker Verification Systems. 10xEngineers, our sponsor, is one of the companies that offers similar products. However, there are no free and complete verification tools currently available for use. In addition, some verification methodologies run simulations against a "gold-standard" reference model–again, no free reference model for RISC-V exists yet that has all of the features this would require. Therefore, any team designing a RISC-V-based system-on-chip would save a great deal of time and money by having access to a free, standard-adhering test suite, such as the one we delivered in May. Our test suite is set to become the world's official free test suite, replacing much of the RVI-maintained `riscv-arch-test`, which serves to jumpstart its adoption in industry. 10xEngineers, by sponsoring this project, will get ahead of their competition in offering new verification services that make use of our testing methodology. Since our tool is open source, there is no economic barrier to using it, but users could pay an additional cost to receive tailored services from 10xEngineers.

Once our tool is adopted, we expect there to be changes in how both design and verification are done in the industry. Since the need for test-writing would be addressed by our work, jobs that used to involve writing tests for standard features in the RVA22 specification might change to include developing and maintaining a more automated system. Additionally, we are developing our tool under the OpenHW Group's contributor license, meaning the tool will also be royalty-free and open source. With a wider availability of fast and cheap design verification, we expect to see an increase in the pace of design as well. The total amount of time and money available in this space would not decrease, so resources previously allocated to DV would now be reallocated to additional design time or higher-quality verification. Providers of proprietary verification technologies and services would need to either carefully distinguish their value from that of the open source tools or find some way to build on them as 10xEngineers will. Just like how ARM was forced to respond to the emergence of RISC-V, we expect a similar disruption, though on a smaller scale, with the release of our test suite.

Through working on this project, the team has gained extensive knowledge of the RISC-V ISA specification and current verification methodologies. Each member has also gained an in-depth understanding of the design of our tool. The value of all of these is compounded by the rising popularity of RISC-V in industry and the projected growth over the next few years, at a minimum. All members have also gained experience in working with large, cross-functional teams and in using industry-standard tools like version control through Git.

3. Functional Coverage

The goal of the project is to use directed tests to achieve full architectural functional coverage of Wally while comparing the architectural state of the processor in lockstep against a reference model. Lockstep simulation is a technique of simulating a design alongside a reference model, giving the same stimuli to both to confirm that the device under test matches the reference at every step. Using this technique, these directed tests will either catch bugs within the Wally RTL (Register Transfer Level code) or confirm that Wally correctly implements the RISC-V ISA standard. However, this is dependent on confidence that the stimuli exercise every scenario that is deemed “interesting”. Sections 3.4 and 3.6 describe decision-making for what is tested and how tests are evaluated for success and coverage. Additionally, the tools use the RISC-V Verification Interface (RVVI) to sample architectural state during a simulation. RVVI is an open standard of interfaces between RISC-V cores, testbenches, and reference models, which is widely used in RISC-V verification. Therefore, only processors that support RVVI can use our test suite.

The workflow begins with human-readable test plans that describe covergroups and accompanying directed test suites. Covergroups are groups of coverpoints, which are sets of interesting architectural states that must be observed during simulation for full functional coverage. Coverpoints specify which signals within the device under test are relevant, as well as which values (or bins) are required for coverage. Directed tests are assembly language programs created to exercise all bins in each coverpoint. For features of the unprivileged RISC-V specification, Python scripts create the covergroups and assembly tests from CSV test plans, while for features of the privileged specification, members of the team hand-write

them from plain English Excel spreadsheet test plans. The tests are simulated on the SystemVerilog model using Questa while collecting functional coverage. They are run in lockstep against ImperasDV over an RVVI interface. Therefore, both Questa and ImperasDV licenses are required. Figure 3.1 demonstrates this tool flow from CSVs to coverage report.

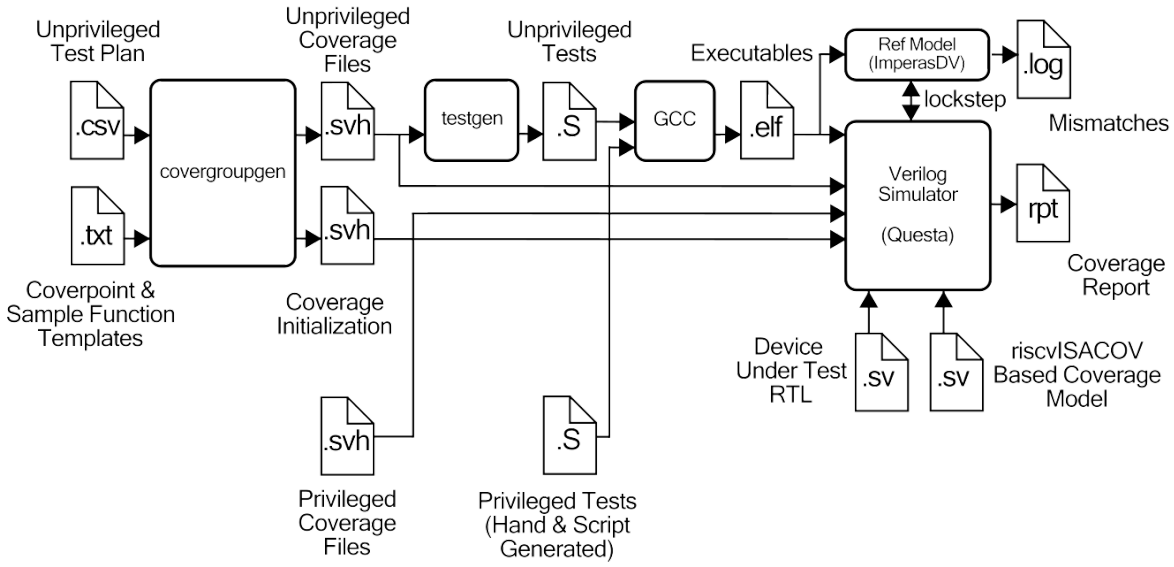

Figure 3.1: Tool Flow Block Diagram

We anticipate that Sail may eventually have full lockstep capabilities as well, and that Verilator may eventually handle functional coverage, eliminating the need for licensed software. These points are elaborated in section 6.

3.1. Test Plans

The RISC-V ISA is inherently configurable with a choice of 32 or 64-bit bus width (XLEN), a 16- or 32-register base ISA, and many optional extensions. As such, the unprivileged test plans for the project are organized by XLEN and extension. Each of these

test plans contains many instructions, each of which has different requirements for full coverage. To manage this, we define many coverpoints to cover different aspects of the architectural state of the core. These coverpoints are adapted from `riscvISACOV`, a collection of RV32I coverpoints released by Imperas as open source.

For example, the instruction `add` takes two register arguments and writes to one register destination. To be confident that this instruction is executed correctly, regardless of the arguments, we must see that each of the 32 integer registers is used as the destination register for an `add` instruction. As such, we can define a coverpoint called `cp_rd,` which requires that each register be used as the destination register for a given instruction at least once. So, if registers `x1` and `x2` are used as the destination in some `add` function call but not `x3`, then the bins for `x1` and `x2` will be marked as covered, while the bin for `x3` will not be.

There are many possible coverpoints, and different instructions will need different combinations of them. To organize this, each extension-XLEN pair has a comma-separated value (CSV) test plan file which contains a column listing each instruction that the extension requires, as well as columns for each of the different coverpoints and a column for the type (format) of the instruction. For the example of `add` and `cp_rd`, there should be an `x` in the test plan in the row corresponding to `add` and the column labeled `cp_rd`.

Figure 3.1.1 demonstrates the general format of the unprivileged test plans. The leftmost column contains the instruction name, followed by the instruction format, and then by the various coverpoints.

Instruction	Type	cp_asm_count	cp_rd	cp_rs1	cp_rs2	cp_rd_corners	cp_rs1_corners	**cr_rs1_rs2_corners**
lw	L	x	x	nx0		x		
sw	S	x		nx0	x			,
addi	I	x	x	x		x	x	
add	R	x	x	x	x	x	x	**x**
sub	R	x	x	x	x	x	x	**x**
beq	B	x		x	x		x	**x**
jalr	JR	x	x	nx0				
jal	J	x	x					
lui	U	x	x			lui		

Figure 3.1.1: Abbreviated RV32I Test Plan

The CSV format of these plans maintains some level of human readability while making it possible to parse them with Python scripts to automate the rest of the project, as described in Sections 3.2 and 3.3. Some rows have entries that are some string other than simply `x`, which indicates a variant on that coverpoint. For example, the `nx0` in the `cp_rs1` column for `jalr` indicates to use the `cp_rs1_nx0` coverpoint rather than `cp_rs1`. `cp_rs1_nx0` checks that an instruction uses all but `x0` as the source register, because a jump to `x0` is an illegal destination that will throw an exception. Additionally, some coverpoints begin with "`cr`" instead of "`cp`," which means that it is a cross product of two or more existing coverpoints (i.e. every bin in one coverpoint is checked in combination with bins from all others).

3.2. Covergroup Generation

Once the unprivileged test plans are carefully populated and reviewed, a Python script named `covergroupgen.py` parses the CSV test plans to create the SystemVerilog

covergroups for each instruction. For each instruction, the script checks which coverpoints are needed, finds the plain text file with the template for that coverpoint, and writes that template into the SystemVerilog file that contains the covergroups for the entire extension.

Code Example 3.2.1 shows an example of a covergroup generated using `covergroupgen.py`. The majority of the coverpoints has been omitted for brevity, but there are examples of different types of coverpoints that are used throughout the project.

```
covergroup I_add_cg with function sample(ins_t ins);
    option.per_instance = 0;
                            ...
   cp_rd : coverpoint ins.get_gpr_reg(ins.current.rd)  iff (ins.trap == 0 )  {
        //RD register assignment
    }
                            ...
    cp_rs1_corners : coverpoint unsigned'(ins.current.rs1_val)  iff (ins.trap == 0 )  {
        wildcard bins zero  =    {0};
        wildcard bins one  =     {32'b00000000000000000000000000000001};
        wildcard bins two  =     {32'b00000000000000000000000000000010};
        wildcard bins min  =     {32'b10000000000000000000000000000000};
        wildcard bins minp1  =   {32'b10000000000000000000000000000001};
        wildcard bins max  =     {32'b01111111111111111111111111111111};
        wildcard bins maxm1  =   {32'b01111111111111111111111111111110};
        wildcard bins ones  =    {32'b11111111111111111111111111111111};
        wildcard bins onesm1  =  {32'b11111111111111111111111111111110};
        wildcard bins walkodd =  {32'b10101010101010101010101010101010};
        wildcard bins walkeven = {32'b01010101010101010101010101010101};
        wildcard bins random   = {32'b01011011101111001000100001110111};
     }
                            ...
   cr_rs1_rs2_corners : cross cp_rs1_corners,cp_rs2_corners  iff (ins.trap == 0 )  {
        //Cross coverage of RS1 corners and RS2 corners
    }
endgroup
```

Code Example 3.2.1: `add` Instruction Covergroup

Some coverpoints have auto-generated bins, such as `cp_rd`. For these, the bins are all possible values of the relevant signal (e.g., 32 bins, 00000 to 11111 for 5-bit register arguments). For register values, it is not reasonable to check all 2^{32} or 2^{64} possible values. Instead, we explicitly define the corner-case value bins for `cp_rs1_corners` and similar coverpoints, as seen in Code Example 3.2.1. Corner value coverpoints check that the arguments or results from an operation fulfill a list of values that are considered interesting. These include the most positive and most negative values, zero, one, negative one, walking ones and zeros, among others. These values are of interest since they are likely to cause interesting behavior when used in operations. For example, adding anything positive to the most positive value a register can hold will cause overflow.

Additionally, we have cross-products of certain groups of coverpoints, ensuring that every combination of bins across said coverpoints is exercised. For example, `cr_rs1_rs2_corners` creates a cross-product of the `cp_rs1_corners` bins and `cp_rs2_corners` bins.

The script also reads the type of the instruction and uses a plain text file template for the instruction type to create a sample function at the end of the covergroups file that samples each instruction's covergroup when that instruction is retired by the DUT (device under test).

```
function void i_sample(int hart, int issue, ins_t ins);
    case (traceDataQ[hart][issue][0].inst_name)
        "add"     : begin
            I_add_cg.sample(ins);
        end
        "addi"     : begin
            I_addi_cg.sample(ins);
        end
```

```
        "and"     : begin
            I_and_cg.sample(ins);
        end
              ...
    endcase
endfunction
```

Code Example 3.2.2: RV32I Sample Function With Abbreviated Instruction Case Statement

In the backend of the coverage model, each extension's sample function is called whenever the DUT executes an instruction, which conditionally samples a single instruction's covergroup and collects coverage for that instance of that instruction. This ensures that each covergroup can only be covered by its corresponding instruction, thus eliminating the need to sample every covergroup or to check for the instruction type within the covergroups.

3.3. Test Generation

Just as `covergroupgen.py` uses the test plan CSV files to create plans, another script called `testgen.py` generates RISC-V assembly programs to hit all of the coverpoints in the covergroup files. This script parses through the SystemVerilog covergroups, retrieves each instruction and all of the coverpoints required for the instruction, and uses that information to write lines of RISC-V assembly code that exercise each bin of each coverpoint. Code Example 3.3.1 shows some cases in one of these test files.

```
# Testcase cp_rd (Test destination rd = x1)
li x15, 0×2df19c49bf9d2b53 # initialize rs1
li x3, 0×e21482b7cfc50d6b # initialize rs2
add x1, x15, x3 # perform operation
RVTEST_SIGUPD(x5, x1)

# Testcase cp_rd (Test destination rd = x2)
```

```
li x24, 0×214606929e470dd7 # initialize rs1
li x23, 0×2391fecd330ad053 # initialize rs2
add x2, x24, x23 # perform operation
RVTEST_SIGUPD(x5, x2)

...

# Testcase cp_rd (Test destination rd = x31)
li x19, 0×ea92d50933697752 # initialize rs1
li x22, 0×1d3e035ce1739ac5 # initialize rs2
add x31, x19, x22 # perform operation
RVTEST_SIGUPD(x1, x31)
```

Code Example 3.3.1: `add` Instruction Assembly Test Program Snippet

For any given coverpoint, there is a part or parts of the instruction that are specified. All other parts of the instruction are randomized. For example, when executing an `add` instruction with `x0` as the destination register, the other register arguments, as well as the values they contain, are random legal values. Similarly, for corner value coverpoints, the value of the relevant source register is specified, while the number of that register, as well as the other source(s) and destination, and the value(s) in the other source(s) are random legal values. The `RVTEST_SIGUPD(x5, x2)` signature update macro is borrowed from the `riscv-arch-test` repository and provides a concise way to update signature memory with the results of the test case.

3.4. Simulation

Once `testgen.py` has created the assembly files, these programs are run on Wally in a Questa simulation. They are run in lockstep with a reference model called ImperasDV to check the architectural state of CORE-V Wally against what the reference model predicts

after every instruction in the test program. This allows us to ensure that Wally correctly executes every test program instruction, or find where Wally does not match the specification. With this and the fact that the tests are created to exercise every relevant edge case for each feature, we can reasonably assume that the Wally implementation of a given instruction is correct. Although this gives a reasonable idea of how robust the implementation is, there are some drawbacks. The definition of edge cases is subjective, and as a result, the edge cases we have (such as a choice of a random value) can be different from edge cases in other test suites. Further, there is a possibility–however unlikely–for both the implementation and ImperasDV to give the same wrong result, which would go undetected in testing.

3.5. Tool Flow

To create the covergroups and tests, as well as run the simulation for functional coverage, we have a handful of terminal commands. Start in the `cvw-arch-verif` directory and use:

```
$ make covergroupgen
```

This runs the covergroup generation script, writing the unprivileged covergroups to `cvw-arch-verif/fcov/unpriv`. Each instruction has its covergroup contained in the file that matches the name of its test plan. These files should be inspected to ensure that the covergroups contain the expected coverpoints for the instructions they cover.

```
$ make testgen
```

This runs the test program generation script. An assembly file is written for each instruction in the `cvw-arch-verif/tests/rv{32,64}/<EXTENSION>` directory. These

files should also be inspected after generation to check that tests are being made for all coverpoints required by the instruction. This is made simpler by the comments preceding each test case to note which coverpoint they correspond to.

```
$ make <un>priv
```

This Makefile target assembles either the unprivileged or privileged assembly language test programs.

The preceding three steps are typically done together, so the team has a single Makefile command that executes all three.

```
$ make --jobs
```

This instruction does the equivalent of the first three instructions in this section and is the preferred way of producing auto-generated covergroups and compiling tests. The `--jobs` tag runs the make jobs in parallel and greatly reduces runtime. After the files have been made, a simulation can follow.

```
$ regression-wally --fcov
```

This simulates Wally in Questa and runs a regression of all of the assembled test programs in lockstep with ImperasDV, checking for both mismatches and coverpoint coverage. Finally, it merges the simulation results and creates report files, described in Section 3.6.

On occasion, it is necessary to remove all result and intermediate files, which can be done using:

```
$ make clean
```

3.6. Results

The goal of testing is to be able to see not only if an instruction is implemented incorrectly, but also to examine the results and ensure the tests cover everything we expect to hit. If there is a bug in Wally that causes a mismatch with the reference model, then the simulation will print an error message, issuing a warning of the discrepancy. In the more common case of confirming that everything is running as expected, we wish to see how much coverage we have for each of the instructions in a given extension.

The simulation software writes the results to a Unified Coverage Database (UCDB) file, which is not human-readable. The Python script `coverreport.py` merges the UCDB files from all of the extensions for each XLEN and then uses the `vcover` shell command to turn these merged files into human-readable reports. These report files outline the percent coverage of the instruction, as well as a breakdown of which coverpoint bins are not being exercised, if any.

Code Example 3.6.1 illustrates the human-readable coverage reports created by `coverreport.py` in the `results` directory. Each bin is enumerated for each coverpoint alongside the number of times it is hit. In this example, every bin of every coverpoint for the `add_cg` covergroup is covered.

```
TYPE /RISCV_coverage_pkg/RISCV_coverage__1/I_add_cg     100.00%       100          -    Covered
    covered/total bins:                                    617       617          -
    missing/total bins:                                      0       617          -
    % Hit:                                             100.00%       100          -
    Coverpoint cp_asm_count                            100.00%       100          -    Covered
        covered/total bins:                                  1         1          -
        missing/total bins:                                  0         1          -
        % Hit:                                         100.00%       100          -
        bin count[1]                                       567         1          -    Covered
    Coverpoint cp_rd                                   100.00%       100          -    Covered
        covered/total bins:                                 32        32          -
        missing/total bins:                                  0        32          -
        % Hit:                                         100.00%       100          -
        bin auto[x0]                                         5         1          -    Covered
        bin auto[x1]                                        24         1          -    Covered
        bin auto[x2]                                        15         1          -    Covered
        bin auto[x3]                                        18         1          -    Covered
        bin auto[x4]                                        19         1          -    Covered
        bin auto[x5]                                        23         1          -    Covered
        bin auto[x6]                                        13         1          -    Covered
        bin auto[x7]                                        15         1          -    Covered
        bin auto[x8]                                        16         1          -    Covered
        bin auto[x9]                                        17         1          -    Covered
        bin auto[x10]                                       21         1          -    Covered
        bin auto[x11]                                       15         1          -    Covered
        bin auto[x12]                                       20         1          -    Covered
        bin auto[x13]                                       22         1          -    Covered
        bin auto[x14]                                       22         1          -    Covered
        bin auto[x15]                                       16         1          -    Covered
        bin auto[x16]                                       13         1          -    Covered
        bin auto[x17]                                       24         1          -    Covered
        bin auto[x18]                                       19         1          -    Covered
        bin auto[x19]                                       16         1          -    Covered
        bin auto[x20]                                       21         1          -    Covered
        bin auto[x21]                                       18         1          -    Covered
        bin auto[x22]                                       16         1          -    Covered
        bin auto[x23]                                       20         1          -    Covered
        bin auto[x24]                                       15         1          -    Covered
        bin auto[x25]                                       16         1          -    Covered
        bin auto[x26]                                       16         1          -    Covered
        bin auto[x27]                                       19         1          -    Covered
        bin auto[x28]                                       20         1          -    Covered
        bin auto[x29]                                       18         1          -    Covered
        bin auto[x30]                                       17         1          -    Covered
        bin auto[x31]                                       18         1          -    Covered
                                        ...
```

Code Example 3.6.1: `add` Instruction Coverage in the RV32GC Functional Coverage Report

When an instruction is found to have less than full coverage, we can inspect the missing cases in the report file to see which cases are not being hit. Usually, the issue is with the test generation, so the next step is to inspect the tests to ensure that the test case is being created correctly by `testgen.py`. If, after *careful* consideration, it is found that the

coverpoint is impossible to hit or otherwise unnecessary for the instruction, it may be removed from the test plan for that instruction. This should only be done if deemed necessary, as the percent coverage figure is meaningless if relevant coverpoints are not being checked.

This leads us to a potential issue with dependencies in the project. Since the goal is 100% functional coverage, and that coverage is dependent on the test plans that we are modifying, it is possible to falsely achieve full coverage by unintentionally tailoring the test plans and coverpoint templates to overlook or omit difficult coverpoints. Improvements to one test plan often must propagate to many related test plans. The teams have found that keeping test plan modification in the hands of a single member (dubbed the “test plan Czar”) is the best way to avoid this issue by ensuring that any modifications to the test plans are carried out with significant consideration.

This issue is much more prominent for the privileged specification, which requires hand-made tests as opposed to generated ones. This is because one minor misunderstanding can be propagated through the coverpoints and tests and cause false coverage. To combat this, the HMC team (responsible for the privileged instructions) has decided to separate the covergroup and test development between different members of their team.

4. Privileged Functional Coverage

The RISC-V privileged specification is much more nuanced compared to the unprivileged specification. Therefore, as previously stated, the teams and advisors find it unreasonable to automate the covergroups and tests as is done for the unprivileged portions of the ISA. This section provides an overview of the changes to the workflow that are necessary for developing and running tests for the privileged specification.

4.1. Test Plans

Since there is no need to be machine-readable, privileged test plans are plain English Excel spreadsheets to make them as clear as possible to the members of the team who write the tests and covergroups.

Figure 4.1.1 exemplifies the format of the privileged test plans with coverpoints organized into covergroups, each with an English description of what is required for coverage, what cases test programs need to exercise, and fields to track coverage progress.

coverpoint	covergroup	Sub Feature	Coverpoint Description	Test Hints	Pass/Fail Criteria (checker)	Functional Coverage	Feature covered
cp_instr_adr_misaligned_branch	exceptionsm	Instruction Address Misaligned	Branch taken to an address that is an odd multiple of 2 PC[1] = 0 and imm[1]=1	Only throws an exception if C is not supported.	Check against RM	To be Done	Uncovered
cp_instr_adr_misaligned_branch_nottaken	exceptionsm	Instruction Address Misaligned	Each type of branch not taken to an address that is an odd multiple of 2	No exception	Check against RM	To be Done	Uncovered
cp_instr_adr_misaligned_jal	exceptionsm	Instruction Address Misaligned	jal to an address that is an odd multiple of 2 PC[1] = 0 and imm[1]=1	Instruction at multiple of 4. Offset is an odd multiple of 2.	Check against RM	To be Done	Uncovered
cp_instr_adr_misaligned_jalr	exceptionsm	Instruction Address Misaligned	jalr to an address that is an odd multiple of 2 and odd multiple of 2 + 1. Every combination of rs1_val[1:0] and offset[1:0]	Only throws an exception if C is not supported.	Check against RM	To be Done	Uncovered
cp_instr_access_fault	exceptionsm	Instruction Access Fault	Instruction access fault is raised				
cp_illegal_instruction	exceptionsm	Illegal Instruction	Executing instruction 0x00000000 and 0xFFFFFFFF throws an illegal instruction exception.				

Figure 4.1.1: ExceptionsM (Machine-mode Exceptions) Test Plan

4.2. Privileged Covergroups

As mentioned, the privileged covergroups are written by hand. Many interesting cases relating to the privilege specification are dependent on the architectural state beyond the instructions themselves, such as CSRs. Given how nuanced and unique each coverpoint is, it is not feasible to automate the process of writing them. Additionally, the organization of privileged covergroups is not tied directly to RISC-V extensions, but rather to more general features of the privilege ISA, such as CSRs, exceptions, and interrupts.

To construct coverpoints that account for intricate combinations of architectural state, such as various CSR values, privilege mode, and instruction being executed, we construct cross products of many simpler coverpoints, each of which describes a specific part of the architectural state. For example, in the machine mode exceptions covergroup, we want to check that instruction address misaligned faults, instruction access faults, illegal instructions, breakpoints, load/store misalignments and faults, and environment calls are triggered and properly handled in machine mode. To do this, we have one coverpoint for the cause of each of these exceptions. Then, we have a cross product with each of these coverpoints and one that specifies machine mode. Therefore, if we write tests that cover all of these bins, we will have covered the cause of every type of exception in machine mode. With this, alongside matching the reference model, we ensure that the DUT handles exceptions correctly in machine mode.

```
covergroup ExceptionsM_exceptions_cg with function sample(ins_t ins);
   option.per_instance = 0;

    `include "coverage/RISCV_coverage_standard_coverpoints.svh"

 // BUILDING BLOCKS TO THE MAIN COVERPOINTS
                         ...
```

```
ecall: coverpoint ins.current.insn {
       bins ecall  = {32'h00000073};
   }
illegalops: coverpoint ins.current.insn {
               bins zeros = {'0};
               bins ones  = {'1};
   }
                        ...
//MAIN COVERPOINTS
cp_ecall_m:             cross ecall, priv_mode_m;
cp_illegal_instruction: cross illegalops, priv_mode_m;
                        ...
endgroup
```

Code Example 4.2.1: Abbreviated ExceptionsM covergroup

Code Example 4.2.1 demonstrates these coverpoints. Under the “building blocks for main coverpoints” comment, there are two coverpoints that each check for the instruction being all zeros and all ones, as well as for the current instruction being an `ecall`. Additionally, the `` `include "coverage/RISCV_coverage_standard_coverpoints.svh" `` statement gives us the `priv_mode_m` coverpoint, which checks for operation in machine privilege mode. At the bottom of the file, under the “main coverpoint” comment, we see the main coverpoint, created as a cross product of the helper and standard coverpoints.

4.3. Privileged Tests

The privileged tests are a set of assembly instructions designed to hit all of the bins in all of the covergroups described in section 4.2. Tests are created independently and in parallel with the privileged covergroups by different people to prevent shared biases or replication of potential errors.

To get started writing new tests, create a new file in the tests/priv directory called `YourNewFeature.S`. Then, from here, the following starter code can be pasted in:

```
///////////////////////////////////////
// YourNewFeature.S
// Written: <Author> <Date>
// Purpose: <Purpose>
```

```
// SPDX-License-Identifier: Apache-2.0 WITH SHL-2.1
///////////////////////////////////////////

// General notes:
// Use csrrw/csrrs/csrrc t6, csr, rs1    when modifying a CSR to also check the old value.
// included from $WALLY/tests/coverage
#include "WALLY-init-lib.h"

main:


<New Tests Here>
```

Code Example 4.3.1: Hand-written Test Starter Code

Individual tests are added to this file, with a function name followed by the relevant assembly instructions to try and hit that covergroup. It's also important to store and restore the state for each test. For example, for the Zicsr tests, at the start of every test, the current `mstatus` CSR is stored in a temporary register and is recovered at the end of the test.

Once tests are written, they can be run using the steps detailed in Section 3.5, the only difference being that `testgen.py` is not used to generate the test executables. For an example, we can look at the coverpoint `cp_illegal_instruction` for `ExceptionsM` tests:

```
///////////////////////////////////////////
// ExceptionsM.S
//
// Written: Roman De Santos rdesantos@hmc.edu 6 February 2025
//
// Purpose: Functional coverage test for Exceptions in M mode
//
// SPDX-License-Identifier: Apache-2.0 WITH SHL-2.1
///////////////////////////////////////////

#include "WALLY-init-lib.h"

main:
   # set mstatus.FS to 01 to enable fp
   li t0,0x4000
   csrs mstatus, t0
                        ...
   ////////////////////////////////
   //cp_illegal_instruction
   ////////////////////////////////
   //ExceptionsInstr.S tests all other illegal instructions exhaustively
   // Attempt to execute illegal instructions
   .word 0x00000000
   .word 0xFFFFFFFF
                        ...
```

```
# Restore the original mstatus.
   csrw    mstatus, t0

finished:
   j done
```

Code Example 4.3.2: Abbreviated ExceptionsM Test

The `cp_illegal_instruction` coverpoint is met whenever an illegal instruction exception occurs in machine mode, as specified by the RISC-V privileged architecture. In this test, invalid opcodes (`.word 0×00000000` and `.word 0×FFFFFFFF`) are manually injected to force such exceptions. When the processor attempts to execute these invalid instructions, it should raise an exception and enter the machine mode trap handler. When we run the simulator, the corresponding bin for this coverpoint will be hit when the illegal instruction exception occurs, verifying that the core properly detects and responds to invalid instructions.

5. Code Coverage

In addition to 100% functional coverage, 100% code coverage is required to meet the Open Hardware Foundation's criteria for technology readiness level 5. Code coverage involves measuring how much of the SystemVerilog source code is exercised by tests. This consists of 4 coverage types [16]:

1. Line: each line in the code has been executed at least once;
2. Condition: each boolean condition has evaluated to both true and false;
3. FSM: every state and every transition of each finite state machine has been taken;
4. Branch: all possible paths in the control structures have been traversed.

For the past three years, the Wally team has worked to improve the code coverage of the Wally RTL in the RV64GC configuration, the one they intend to raise to TRL 5. While the majority of the SystemVerilog code is covered, the remaining coverage holes are subtle, and some are impossible to cover. As the team completed writing the privileged covergroup files in March, one member switched to improving code coverage.

5.1. Code Coverage Methodology

Similar to functional coverage, collecting code coverage involves running a comprehensive test suite on the device under test in simulation and configuring the simulation software (Questa in our case) to collect code coverage. Initially, we used the `riscv-arch-test` suite, but elected to transition to using our `cvw-arch-verif` test suite as we believe it is more extensive. It also provides more consistency between functional and code coverage. If there are code coverage holes after running the functional coverage suite, it either reveals a hole in functional coverage or a microarchitectural edge case. For the latter,

we write additional tests in the `$WALLY/tests/coverage/` directory. However, this is a less desirable solution compared to more robust functional coverage tests when the hole relates to architectural features.

As mentioned above, the focus of code coverage is on the RV64GC configuration of Wally. Since all configurations of Wally share the same RTL source code, many lines pertain to features that are not supported by this configuration, such as quad precision floating point. As such, it is sometimes necessary to exclude coverage for entire lines or specific possible scenarios in a line. In addition, some parts of the RTL source code are intentionally redundant for future proofing or readability, and are therefore impossible to cover and are excluded from coverage. Coverage exclusions are described in section 5.4.

5.2. Tool Flow

Similar to functional coverage, we run a suite of tests on the DUT with Questa configured to collect code coverage. This is done by running:

```
$ regression-wally --ccov
```

This runs all of our functional coverage tests along with custom tests in `$WALLY/tests/coverage` that the team writes specifically to close code coverage holes These custom tests are described in section 5.3. Additionally, code coverage regression merges the UCDB output files produced by Questa and produces reports in the `sim/quesa/cov` directory. The most helpful report is `rv64gc_uncovered_hierarchical.rpt`, which shows what parts of the code are uncovered as well as their scope in the RTL hierarchy. Additionally, there are separate reports for each major unit of the

core (e.g. `rv64gc_uncovered_ifu.rpt`). These show exactly which lines are uncovered, along with which cases are missing for coverage and hints on how to cover them.

It is often more convenient to run a single .elf file rather than a full regression when attempting to close a code coverage hole. This can be done by running a simulation with

```
$ wsim rv64gc <path to elf> --ccov
```

This produces a new UCDB output file, which is not accounted for in the reports. To merge the results and produce the report files, run

```
$ cd $WALLY/sim
$ make QuestaCodeCoverage
```

This keeps all of the existing results from the most recent regression, and also adds the results from the single `wsim` command. Note that for a coverage exclusion to take effect, a fresh regression is required.

5.3. Custom Tests

Since we collect code coverage using the functional coverage tests, code coverage holes commonly unveil functional coverage holes as well. In these cases, the functional coverage coverpoints and tests should be updated to close both holes.

When code coverage holes relate to microarchitectural state, they do not reveal any interesting missing functional coverage. When this is the case, we write a custom test in `$WALLY/tests/coverage` to target the hole. Depending on the complexity of the uncovered RTL, these vary significantly in length and difficulty. Simple combinational logic can often be covered in just a few lines, while certain FSM coverage requires extensive setup to get to the correct state with the specific conditions required to cover the hole.

Code Example 5.3.1 demonstrates a custom test that the team wrote for microarchitectural data cache signals. There is a line of SystemVerilog that handles flushing the writeback stage while setting the dirty bit for each cache way. For one of the four cache ways, the tests suite didn't cause both `FlushStage` and `SetDirtyWay` to be high simultaneously.

```
#include "WALLY-init-lib.h"
main:
        // way 0
        li t0, 0x80100770
        sd zero, 0(t0)
        sd zero, 1(t0)
        lr.d t1, (t0)
        addi t0, t0, 1
        sc.d t2, t1, (t0)

        // way 1
        li t0, 0x80101770
        sd zero, 0(t0)
        sd zero, 1(t0)
        lr.d t1, (t0)
        addi t0, t0, 1
        sc.d t2, t1, (t0)

        // way 2
        li t0, 0x80102770
        sd zero, 0(t0)
        sd zero, 1(t0)
        lr.d t1, (t0)
        addi t0, t0, 1
        sc.d t2, t1, (t0)

        // way 3
        li t0, 0x80103770
        sd zero, 0(t0)
        sd zero, 1(t0)
        lr.d t1, (t0)
        addi t0, t0, 1
        sc.d t2, t1, (t0)

        j done
```

Code Example 5.3.1: Custom test for uncovered data cache logic

This code uses a misaligned `sc.d` instruction to cause both of these signals to go high in each of the four cache ways. Being a store causes `SetDirtyWay` to be high, and being an

illegal misaligned instruction causes an illegal instruction exception that makes the `FlushStage` signal go high.

5.4. Coverage Exclusions

As mentioned, several components of the RTL cannot be covered. The major source of this is optional features that are not supported by the rv64gc configuration of Wally, such as quad precision floating point, or dirty bits in the instruction cache. Other times, the code contains redundant checks that are maintained for readability or an extra level of robustness.

To exclude certain conditions from code coverage, we add `coverage exclude` lines to `$WALLY/sim/questa/coverage-exclusions-rv64gc.do`. Code Example 5.4.1 demonstrates a coverage exclude line. The `-scope` flag specifies the scope in the SystemVerilog hierarchy for the exclusion. This is useful when there are multiple instances of the same module or code in a generate loop. The `-linerange` flag is self-explanatory, but we use a script called `GetLineNum` to find the line number from a unique identifier in the file to avoid hard coding line numbers, which would break the exclusions whenever the RTL changed. The `-item` flag specifies which item within the line the exclusion applies to; in this case the `e 1` flag specifies expression number 1. Finally, the `-fecexprrow` flag specifies which expression row (i.e. truth table lines) is excluded, allowing us to exclude only impossible scenarios instead of all scenarios.

```
coverage exclude -scope /dut/core/priv/priv/trap -linerange [GetLineNum
${SRC}/privileged/trap.sv "assign ExceptionM"] -item e 1 -fecexprrow 2
```

Code Example 5.4.1: Example coverage exclude expression

There are different arguments for the `-item` flag depending on the type of coverage being excluded (e.g. branch, condition, expression, statement, FSM, or toggle coverage). The best way to know what to use is to see the type of coverage that is missing in the uncovered reports, and reference the QuestaSim User Manual for the specific syntax.

5.5. Code Coverage Status

When the team began work on code coverage this spring, there were 90 lines in the `rv64gc_uncovered_hierarchical.rpt` file. The team wrote exclusions and custom tests for the hazard, decomp, floating point division and square root, privilege, and cache modules, achieving full coverage of each. At the time of this writing, the uncovered report is around 30 lines, demonstrating that the team has closed two thirds of the coverage holes from the start of the spring term. The vast majority of the remaining coverage holes are finite state machines. FSM coverage is the most difficult to hit, but also the most difficult to prove uncoverable.

6. Open Source Tool Flow

While the coverpoints and tests developed for this project have always been open source, much of the underlying infrastructure relied on various proprietary tools for coverage collection, result checking, and simulation. With one of the central commitments of this project being to produce open source tests and coverage that can be run on any core by anyone, moving to a flow based on open source tools was a major focus for the latter part of the project. This section will walk through the proprietary tools that this project was built on and (where possible) the work that was done to migrate to open source alternatives.

6.1. Proprietary Tools

The initial version of the test suite developed by the clinic team relied upon three main pieces of proprietary software: ImperasDV, Questa, and proprietary extensions to the open source `riscvISACOV`. The following section provides an overview of these three tools and their use in `cvw-arch-verif`.

6.1.1. ImperasDV and RVVI

ImperasDV is a lockstep reference model developed by Imperas (now a part of Synopsys). It simulates RISC-V instructions and compares the contents of all architectural state (PC, registers, and memory) with a DUT after each instruction. Examples of the output from ImperasDV are provided in Section 7.

The reference model and DUT communicate over RVVI (RISC-V Verification Interface), a series of connections that exposes all of the DUT's CSRs and registers, the current PC, instruction, operating mode, and other signals that collectively give a complete

view of what the DUT is doing. RVVI is an open source interface developed by Synopsys and is also the primary interface with ImperasDV. While the standard for the interface itself is open source, Synopsys's implementation of it is not, and relies on components provided by ImperasDV.

6.1.1.1. RVVI Implementation

RVVI needs to be manually configured for each DUT. The interface needs to be instantiated, and each of the relevant internal signals from the DUT needs to be connected. Code Example 6.1.1.1 shows the instantiation of the interface and an example of connecting the `FFLAGS` CSR to it. The RVVI connection needs the state of the processor in its writeback stage to compare what is happening as each instruction retires, so some signals need to be pipelined along to ensure they are available. Code Example 6.1.1.2 shows the current instruction being pipelined through Wally's stages and eventually connected to the RVVI `insn` port.

```
rvviTrace #(.XLEN(P.XLEN), .FLEN(P.FLEN)) rvvi(); // Instantiate RVVI
// CSR connection macro
`define CONNECT_CSR(name, addr, val) \
  logic [P.XLEN-1:0] prev_csr_``name; \
  always_ff @(posedge clk) \
    prev_csr_``name <= rvvi.csr[0][0][addr]; \
  assign rvvi.csr_wb[0][0][addr] = (rvvi.csr[0][0][addr] != prev_csr_``name); \
  assign rvvi.csr[0][0][addr] = valid ? val : prev_csr_``name;
`CONNECT_CSR(FFLAGS, 12'h001, testbench.dut.core.priv.priv.csr.csru.csru.FFLAGS_REGW); // Connect FFLAGS
```

Code Example 6.1.1.1: RVVI instantiation and CSR connection

```
// Connect the current instruction bits to RVVI through several pipeline registers
flopenrc #(32)    InstrRawEReg (clk, reset, FlushE, ~StallE, InstrRawD, InstrRawE);
flopenrc #(32)    InstrRawMReg (clk, reset, FlushM, ~StallM, InstrRawE, InstrRawM);
flopenrc #(32)    InstrRawWReg (clk, reset, FlushW & ~TrapM, ~StallW, InstrRawM, InstrRawW);
assign rvvi.insn[0][0]     = InstrRawW;
```

Code Example 6.1.1.2: RVVI instruction connection

6.1.2. RISCVISACOV

The coverpoints that serve as the cornerstone of this functional verification project are built on top of `riscvISACOV`. `riscvISACOV` is a set of SystemVerilog classes, functions, and other structures developed by Synopsys as an open source framework for RISC-V architectural functional verification [17], but in practice, it relies on proprietary components from ImperasDV and their proprietary implementation of RVVI. Synopsys also only provides the RV32I coverpoints as part of the open source suite. While many of the coverpoints themselves are not present in the open source version of `riscvISACOV`, the framework still serves as a strong starting point for the infrastructure to develop new coverpoints and covergroups.

A forked version (changes from upstream described in Section 6.2.1) of `riscvISACOV` is stored in the `fcov` directory of `cvw-arch-verif`. The highest level component of this suite is `RISCV_coverage.svh` (Code Example 6.1.2.1). This creates a SystemVerilog class that receives an RVVI instance as input and calls several functions that save all of the data from an instruction and sample the covergroups to check which coverpoints have been covered.

```
class coverage #(
  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
) extends RISCV_coverage #(ILEN, XLEN, FLEN, VLEN, NHART, RETIRE);

  function new(virtual rvviTrace #(ILEN, XLEN, FLEN, VLEN, NHART, RETIRE) rvvi);
    super.new(rvvi);
  endfunction

  function void sample(bit trap, int hart, int issue, string disass);
```

```
    save_rvvi_data(trap, hart, issue, disass);
    sample_extensions(hart, issue);
  endfunction
endclass
```

Code Example 6.1.2.1: `RISCV_coverage.svh`

All of the other files in the `fcov/coverage` directory define support functions for measuring coverage and writing coverpoints. `RISCV_coverage_base.svh` contains the `RISCV_coverage` class that the `coverage` class above inherits from. It is the main class that defines the highest level functions including the initialization of the coverage infrastructure and the top level sampling function. `RISCV_coverage_common.svh` has enums and functions for different types of registers and other instruction data so that they can be called in the coverpoints and specific registers can be associated with certain instructions. `RISCV_coverage_csr.svh`, `RISCV_coverage_exceptions.svh`, and `RISCV_coverage_hazards.svh` define helper functions for use in coverpoints to avoid duplicated logic and to allow for more concise and easy to understand coverpoints. Finally, `RISCV_coverage_rvvi.svh`, `RISCV_instruction_base.svh`, and `RISCV_trace_data.svh` handle storing information about each instruction that is executing, the state of the DUT (passed in via RVVI) during each of those instructions, and decoding the information so that it can be used by the coverpoints. Several shortcomings in this framework (as well as the reliance on external, proprietary tools) drove the team to develop a modified version of `riscvISACOV` (appropriately named `cvw-arch-verif`) with extended functionality and no reliance on Synopsys software. The modified version is presented in Section 6.2.1.

6.1.3. Questa

The final proprietary tool that this project relied on is Questa. Questa is a commercial SystemVerilog logic simulator from Siemens. It is used to simulate the DUT, collect coverage for the coverpoints, and interface with the ImperasDV reference model. While open source simulators exist (the open source Verilator is capable of simulating Wally and is used for many of its tests), none of them are capable of collecting functional coverage from coverpoints at this time. This is something that Verilator plans to implement eventually [18], but there is no estimated timeframe for this work yet, and as of May 2025 has not been started.

6.2. Open Source Alternatives

All of the proprietary tools from Section 6.1 are heavily interwoven with the approach to both coverage and testing that have been described thus far. Shifting to a more open source flow required careful consideration of each of these components. While it was possible to move to fully open source coverpoints and reference models with significant changes to the overall flow, there was no viable way to switch to an open source simulator due to the lack of functional coverage support. The team envisions this being possible once Verilator supports these constructs, but for now, that is out of the scope of this project.

6.2.1. Custom Open Source Functional Coverage Backend

One of the most critical components of this project is the functional coverage metrics that are measured using the coverpoints (explained in Sections 3.2 and 4.2). As explained in Section 6.1.2, these coverpoints are built on top of a customized version of the open source `riscvISACOV` functional coverage framework that provides functions, interfaces, classes,

and other constructs necessary to craft the coverpoints themselves. While the existing open source version of this toolset was a sufficient starting point, it did not meet the long term goals of the project due to missing components necessary for a fully open source approach and differences in the coverpoint writing approach.

Most of the existing aspects of `riscvISACOV` described in Section 6.1.2 were copied into the new `cvw-arch-verif` framework with minimal modifications. To use this framework without connecting it to ImperasDV, the team needed to develop a new way of providing the input data. In addition to the previously discussed RVVI interface, the framework needs the disassembled instruction as an input. This instruction is parsed to populate each of the registers and immediate values for use by the coverpoints. When `riscvISACOV` is connected to ImperasDV, this data is already provided, but when running standalone, a new way of creating this data is needed. The team developed a SystemVerilog native disassembler that outputs a string with the appropriate instruction when given a machine code instruction. The open source `riscv-opcodes` [19] project was used to generate the appropriate bit matching patterns. Code Example 6.2.1.1 shows a snippet of these bit patterns from riscv-opcodes, and Code Example 6.2.1.2 shows their use to generate the disassembly. Similar new functions were also added for decoding CSRs as the original version also relied on ImperasDV for that functionality.

```
localparam [31:0] ADD              = 32'b0000000??????????000?????0110011;
localparam [31:0] ADD_UW           = 32'b0000100??????????000?????0111011;
localparam [31:0] ADDI             = 32'b???????????????????000?????0010011;
localparam [31:0] ADDIW            = 32'b???????????????????000?????0011011;
localparam [31:0] ADDW             = 32'b0000000??????????000?????0111011;
```

Code Example 6.2.1.1: `RISCV_imported_decode_pkg.svh` snippet

```
string decoded;

// Mask top bits if compressed instruction
automatic bit compressedInstruction = instrRaw[1:0] =/= 2'b11;
automatic bit [31:0] instr = compressedInstruction ? {16'b0, instrRaw[15:0]} : instrRaw;

// Registers
automatic bit [4:0] rs1Bits  = instr[19:15];
automatic bit [4:0] rs2Bits  = instr[24:20];
automatic bit [4:0] rdBits   = instr[11:7];

// Register names
automatic string rs1  = get_gpr_name(rs1Bits);
automatic string rs2  = get_gpr_name(rs2Bits);
automatic string rd   = get_gpr_name(rdBits);

casez (instr)
  ADD:     $sformat(decoded, "add %s, %s, %s", rd, rs1, rs2);
  SUB:     $sformat(decoded, "sub %s, %s, %s", rd, rs1, rs2);
  AND:     $sformat(decoded, "and %s, %s, %s", rd, rs1, rs2);
  OR:      $sformat(decoded, "or %s, %s, %s", rd, rs1, rs2);
   ...
endcase
if (compressedInstruction)
  return $sformatf("%04h %s", instr[15:0], decoded);
else
  return $sformatf("%08h %s", instr[31:0], decoded);
```

Code Example 6.2.1.2: `disassemble.svh` snippet

Beyond the changes necessary to support an open source flow, the forked version of `riscvISACOV` was also enhanced with several additional features. One of the most significant was an overhaul of the sampling methodology. The original `riscvISACOV` relied on a separate (identical) typedef of the `RISCV_instruction` class for each covergroup. Unprivileged covergroups only sample values when the relevant instruction is run (the `add_cg` covergroup will only sample when an `add` instruction is executed), so the existing method is acceptable. However, privileged covergroups don't have particular relevant instructions, so they need to sample every instruction. Since all privileged covergroups are checked at the same time, all of them would attempt to sample the same instruction. Previous instructions are kept in a queue, and each time an instruction is sampled the new

instruction is pushed onto the queue and the oldest instruction falls off. That means repeated sampling of the same instruction would push all of the previous instructions off of the queue. These previous instructions are needed by several privileged coverpoints to check transitions from one state to another, rendering the queue unusable. To solve this, the team implemented a new sampling approach. A single instance of the instruction class was created for each instruction, irrespective of which covergroups were sampling it. Then, a sample function was called that populated the class with all of the relevant data about the instruction (registers, immediate values, etc.), and that pre-populated class was passed to each of the covergroups. In addition to solving the issue with the lost data, this new approach had the advantage of a noticeable speedup since sampling was happening significantly less often.

A variety of other small changes were made to the framework to make it more configurable. Standard coverpoints were also added in a new `RISCV_coverage_standard_coverpoints.svh` file so that the same coverpoint didn't need to be recreated repeatedly (like checking the privilege mode). All of this work enabled a functional coverage backend that matched the needs of the team's coverpoints while being fully open source.

6.2.1.1. Extended RVVI

Modifications to RVVI were needed in addition to all of the previously described changes to the functional coverage backend. While RVVI already includes most of the architectural state that would be useful for coverpoints, the team found it to be insufficient for creating virtual memory coverpoints. The virtual memory coverpoints that the team created relied on a combination the virtual and physical page addresses and properties of the page table, none of which was included in RVVI. To work around this, the team developed a

new Extended RVVI that added virtual and physical addresses, the page table entry, number, and type, and the kind of access that was occurring. These signals were all added for both instructions and data memory accesses. These additional signals provided additional insight into what was happening in the DUT, and therefore the team was able to write more detailed coverpoints. For example, with the new signals it was easy to create coverpoints that check the page type or whether a page table entry is global (see Code Example 6.2.1.1.1). Similar new signals were added for interrupts for the same reason. All three machine interrupts and supervisor external interrupts were added.

```
PageType_i: coverpoint ins.current.page_type_i {
    bins mega = {2'b01};
    bins kilo = {2'd0};
}
...
global_PTE_d: coverpoint ins.current.pte_d[7:0] {
    wildcard bins leaflvl_u = {8'b??111111};
    wildcard bins leaflvl_s = {8'b??101111};
}
```

Code Example 6.2.1.1.1: Virtual Memory Coverpoint

6.2.2. Open Source Reference Model & Signature Based Tests Overview

Beyond the coverage, the other major component of this project is the tests themselves. As explained in Section 3, the tests do not inherently check for correct results. The approach to check correctness described previously relies on the proprietary ImperasDV reference model running in lockstep, which poses another barrier to adoption for individuals, institutions, or companies that cannot acquire a license to that tool. RVI backs the open source RISC-V Sail model as the official golden reference model of the RISC-V ISA, so it is a natural choice to use as a replacement reference model. Unfortunately, Sail does not currently support lockstep simulation. While this is planned, it is a significant undertaking

and was not ready in time to be used for this project. Furthermore, while lockstep is easy to design tests for, the testbench to run those tests becomes far more complex, and it poses significant challenges to run those tests on actual silicon [20]. To address these constraints, the team adapted the tests to add an option for signature-based checking instead.

Signature-based checking is already used by the previously described official `riscv-arch-test` suite of tests from RVI, and signature generation is supported by the most popular open source reference models, Sail and Spike. Signature-based checking works by storing calculated values in a specified region of memory and then comparing this region of memory to the corresponding region from the reference model once the test is complete. This process is described in more detail in Section 6.3.

Signature-based checking using the Sail reference model removes the need for the proprietary ImperasDV simulator and the complex testbench setup. It also poses a new issue for coverage collection: the complicated testbench interface was what sent all of the relevant signals from the core to the coverage backend. While this seems like a problem, the team was able to turn this problem into a further optimization for the overall flow. As explained in Section 1, the measurement of architectural functional coverage is a function of the tests, not the specific DUT. For any processor with the same configuration, the same set of tests will achieve the same degree of coverage. This means there is no reason to check the coverage each time the tests are run on an actual processor. Instead, the tests' coverage can be checked by running them directly on the reference model. Once they have reached 100% coverage on the reference model, the tests will also have 100% coverage on any core with a matching configuration.

While this seems like a great simplification, none of the reference models are written in SystemVerilog, so they cannot directly interface with the coverpoints. To get around this, the team developed a new approach that required further changes to the coverage infrastructure: a new top-level SystemVerilog testbench, several Python scripts, and changes to the Sail reference model itself. These changes allow the reference model's output to be piped into the SystemVerilog simulator to generate coverage metrics.

The alternative interface to the functional coverage begins with the new testbench. This testbench forgoes an actual DUT and instead directly populates the RVVI with data from a trace file (described below) containing instruction by instruction information from a reference model. It begins by loading the trace file (provided at runtime by a plusarg) and instantiating the `rvviTrace` interface and `cvw_arch_verif` module, as seen in Code Example 6.2.2.1. Then, on every clock cycle it scans a line from the trace file and uses that data to populate the relevant signals (Code Example 6.2.2.2).

```
// Load pre-formatted trace file from traceFile plusarg
initial begin
  if (!$value$plusargs("traceFile=%s", traceFile)) begin
    $display("Error: trace file not provided");
    $finish;
  end
  traceFileHandler = $fopen(traceFile, "r");
  if (traceFileHandler == 0) begin
    $display("Error: Could not open trace file");
    $finish;
  end
end

// Load coverage model and connect to RVVI trace interface
rvviTrace #(.XLEN(XLEN), .FLEN(FLEN), .VLEN(VLEN)) rvvi();
cvw_arch_verif cvw_arch_verif(rvvi);
```

Code Example 6.2.2.1: `testbench.sv` setup

```
  // Parse line and set signals
  if (line ≠ "" & line ≠ "\n") begin // Skip empty lines
    splitLine(line, words);          // Split line into queue of individual words
```

```
    while (words.size > 0) begin
      key = words.pop_front();
      val = words.pop_front();
      case(key)
        // Standard signals
        "INSN":           num = $sscanf(val, "%h", insn);
        "TRAP":           num = $sscanf(val, "%b", trap);
         ...
        // Registers
        "X": begin
          num = $sscanf(val, "%d", regNum);
          val = words.pop_front();
          num = $sscanf(val, "%h", regVal);
          x_wdata[regNum] = regVal;
          x_wb |= (1 << regNum);
        end
         ...
        default: begin
          $display("Unknown key: %s", key);
          $finish();
        end
      endcase
    end
    order++;
    valid = 1;
  end else begin
    $display("Skipping empty line");
  end
end

// Connect testbench signals to RVVI trace interface
// Basic signals
assign rvvi.clk = clk;
assign rvvi.valid[0][0] = valid;
assign rvvi.order[0][0] = order;
assign rvvi.insn[0][0] = insn;
...
assign rvvi.x_wb[0][0] = x_wb;
assign rvvi.x_wdata[0][0] = x_wdata;
```

Code Example 6.2.2.2: `testbench.sv` trace parsing

SystemVerilog is not a powerful language for string parsing, so the testbench expects the trace file to be preformatted to match a very specific layout. Each line corresponds to one instruction. Each of those lines is a set of space-separated key value pairs. Code Example 6.2.2.3 shows an example of one line of this trace from the `add` test. It lists the PC, the instruction encoding, the current privilege mode (3 for machine), and which registers were written (`x25` written with `0x00000000EAA512B3` in this case).

```
PC 80000190 INSN 01800CB3 MODE 3 X 25 00000000EAA512B3
```

Code Example 6.2.2.3: `testbench.sv` trace parsing

The previously described format is convenient for parsing with SystemVerilog, but it is not the native format for the output of Sail. To work around this, two different approaches were taken: a conversion Python script and changes to the Sail model itself. Starting with the Python conversion script, `sail-parse.py` takes a Sail log as an input and outputs a file with the same data converted to the trace format. It is run as `bin/sail-parse.py <input_log> <output_trace>`. While this script is currently only compatible with Sail, similar scripts could be created for Spike, QEMU, or other reference models if desired. The other approach was to generate the trace directly from Sail. To work towards this approach, the team successfully merged a new method of logging using callbacks [20] so that any desired output format can be described directly in C++. The eventual goal is to have one of the choices for the output to be this trace format directly. While that isn't yet possible, several changes were made as part of the switch to callbacks that enabled the previously described Python flow. The most significant of these was outputting the CSR address in addition to the name for CSR writes. This was necessary for a direct conversion to the trace format described above.

Once this trace file is ready, it needs to be passed to the testbench and Questa needs to be launched to collect functional coverage. A new tcl script (`cvw-arch-verif.do`) was created for this. It takes the trace directory, test name, path to `cvw-arch-verif/fcov`, and path to a `coverage.svh` file as four positional arguments and then runs the simulation.

Finally, `trace-coverreport.py` takes a directory of UCDB files and generates a report similar to that described in Section 3.6.

Going from an assembly file to functional coverage measurements using Sail requires a lot of commands that all need to be strung together in the proper order with many arguments. To make this process easier, these commands were integrated into RISCOF, the existing tool for running the `riscv-arch-test` suite of tests. RISCOF takes a directory of tests, compiles them, runs them on Sail (and Spike), converts the log to the required trace, runs Questa to generate UCDBs, and finally generates a coverage report based on those UCDBs. The commands for running all of this using RISCOF are described in Section 6.3.3.

6.3. RISCV-ARCH-TEST INTEGRATION

One of the major concerns with this project was its longevity. There must be a team to continue maintaining and developing the suite past the end of the academic year. Since `riscv-arch-test` is the official RVI compliant test suite, the team determined it would be far more effective to contribute enhancements to that project rather than maintaining a parallel test suite. To this end, the team decided to contribute the new tests to `riscv-arch-test`, replacing most of their existing unprivileged tests (except the floating point tests) and adding tests for many previously untested features, primarily privileged. This required significant changes to the tests the team developed. One of the biggest changes was integrating macros into the tests from `cvw-arch-verif` to align them with the `riscv-arch-test` macro-driven structure and signature-based verification process.

The diagram in Figure 6.3.1 illustrates how the final integrated system would support both lockstep execution with a reference model and signature-based checking. Tests are compiled and run on both the DUT and a reference model, such as Sail, producing signature files that are then compared.

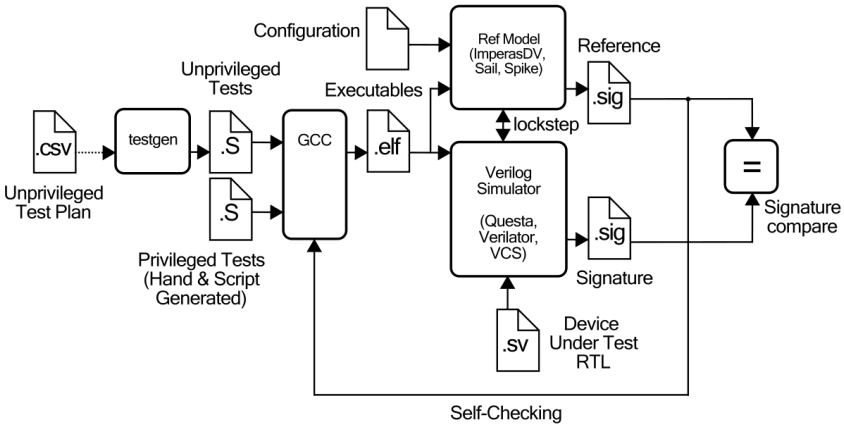

Figure 6.3.1: Signature-Based Tool Flow

6.3.1. RISCV-ARCH-TEST MACROS

The integration of `cvw-arch-verif` into `riscv-arch-test` required aligning with its macro-driven structure and signature-based verification process. To do this, we adopted macros from the existing `riscv-arch-test` framework to ensure compatibility and simplify future maintenance.

Macros in `riscv-arch-test` are reusable code blocks written in assembly that abstract away repetitive low-level operations such as initializing registers, executing instructions, capturing results, and updating memory-based signatures. By using macros, tests reduce code repetition and ensure consistency across tests targeting various RISC-V

extensions. They also tend to require fewer instructions to perform the same actions as a helper function.

The `riscv-arch-test` macros are key in enabling signature-based checking. During test execution, specific macros write result values into designated memory locations. These values, known as test signatures, represent the observable outputs of the test. After execution, the collected signatures are compared against the known reference signature to determine if the device under test behaves correctly and has passed architectural verification. This verification process is supported by RISCOF, which compiles `riscv-arch-test` tests and simulates them using reference models such as Sail or Spike to generate and validate expected signatures [22].

All macros used in this project are based on those already defined within the `riscv-arch-test` framework, which supports a variety of test scenarios and architectural extensions. Their official definitions and usage guidelines are documented in the RISC-V Architecture Test Format Specification [23]. In the codebase, macros are defined in the `riscv-arch-test/riscv-test-suite/env/` directory, primarily in the files `arch_test.h, encoding.h, test_macros_vector.h`, and `test_macros.h`.

To integrate these macros into our workflow, we updated `testgen.py` within `cvw-arch-verif` to automatically insert them into the generated test files, as detailed in Section 6.3.3. This integration ensures that each test conforms to the expected `riscv-arch-test` structure, which relies on specific macros for setup, teardown, and signature tracking. For example, `RVTEST_CODE_BEGIN` and `RVTEST_CODE_END` appear in

all tests to clearly mark the beginning and end of the test code, ensuring consistent initialization and finalization across the framework. These macros also handle extracting values from memory and formatting them based on metadata defined by the specification [23]. In `cvw-arch-verif`, their functionality is implemented in `testgen_header.S` and `testgen_footer.S`.

Additionally, a macro is implemented after every test case. While the `riscv-arch-test` framework defines a variety of macros for this purpose, our implementation standardizes on using `RVTEST_SIGUPD` wherever applicable to record register values in the test signature. The macro has the following format:

`RVTEST_SIGUPD(BaseReg, SigReg [, Offset])`

- If `Offset` is provided, it sets an internal `hidden_offset` to that value
- The value in `SigReg` is stored at the memory address `BaseReg + hidden_offset`
- After the store, `hidden_offset` is automatically incremented so that subsequent calls store results at consecutive memory locations.

Code Example 6.3.1.1 shows how `RVTEST_SIGUPD` is used in practice. The first argument is the register which contains the signature pointer (x3), and the second is the register whose value is being recorded (x25 and x7).

```
RVTEST_SIGBASE(x3,signature_base)
# Testcase cp_rs1 (Test source rs1 = x0)
li x0, 0×b4e96718 # initialize rs1
li x24, 0×eaa512b3 # initialize rs2
add x25, x0, x24 # perform operation
RVTEST_SIGUPD(x3, x25)

# Testcase cp_rs1 (Test source rs1 = x1)
li x1, 0×81c4ef2a # initialize rs1
```

```
li x6, 0×917cfa69 # initialize rs2
add x7, x1, x6 # perform operation
RVTEST_SIGUPD(x3, x7)

…

signature_base:
   .fill 164*(XLEN/32),4,0×deadbeef
```

Code Example 6.3.1.1: Macro usage in the I-extension (add instruction)

In this example, the result of the `add` instruction is stored in a destination register (`x25` or `x7`) and then passed to the `RVTEST_SIGUPD` macro. This macro writes the result to signature memory using the pointer in `x3`, which is incremented by an internal offset after each call. The base of the signature memory (`signature_base`) is initialized at the start of the code using the `RVTEST_SIGBASE` macro in `testgen_header.S`, which defines where signature values will be written during test execution. This macro also initializes an internal counter (`hidden_offset`) to track where the next signature should go:

`RVTEST_SIGBASE(BaseReg,Val)`

- `BaseReg` is the register to hold the signature base address.
- `Val` is the memory address assigned as the signature base.
- An internal offset (`hidden_offset`) is initialized to zero.

The `signature_base` label points to the reserved region of memory used to store test results. The memory region's size is calculated based on how many times `RVTEST_SIGUPD` (or any equivalent macro) is expected to be called, ensuring enough space is allocated for all test outputs.

While most unprivileged instructions allow direct signature storage using `RVTEST_SIGUPD`, certain instruction types require special handling. For example, with branch and jump instructions, the goal is not to store the result of a computation, but to determine whether the branch was taken. In such cases, the signature must capture information related to the program counter (PC) to reflect control flow behavior accurately.

In Code Example 6.3.1.2, the `auipc` and `addi` instructions are used together to store the PC value after the `beq` (branch if equal) instruction. This sequence uses `auipc x18, 0` to load the upper bits of the current PC into register `x18`, followed by `addi x18, x18, 4` to complete the PC calculation of the instruction following `beq`. The resulting value is stored using `RVTEST_SIGUPD`, allowing the test infrastructure to assess whether the branch was taken by analyzing the recorded control flow.

```
# Testcase cp_rs1 (Test source rs1 = x0)
li x0, 0×2947ca01 # initialize rs1
li x26, 0×895e02a2 # initialize rs2
auipc x18, 0
beq x0, x26, 1f # perform operation
addi x18, x18, 4
1:
RVTEST_SIGUPD(x3, x18)
# same values in both registers
li x0, 0×895e02a2 # initialize rs1
li x26, 0×895e02a2 # initialize rs2
auipc x18, 0
beq x0, x26, 1f # perform operation
addi x18, x18, 4
1:
RVTEST_SIGUPD(x3, x18)
```

Code Example 6.3.1.2: Macro Usage in the I-extension (beq instruction)

For store-type instructions, the `CHK_OFFSET` macro is used instead of `RVTEST_SIGUPD`, as store operations involve manual address calculation and memory

updates. Since RVTEST_SIGUPD handles address and offset logic automatically, using it with store instructions would conflict with the manual logic required for this specific instruction behavior.

In Code Example 6.3.1.3, a memory offset is computed using addi, and an sb (store byte) instruction writes a value to that calculated address. After the store, CHK_OFFSET is called to update the signature offset manually. In this case, the sb instruction stores a byte at a manually determined offset from the signature base register. Because the address is explicitly computed in the test, manual control using CHK_OFFSET is necessary. This method ensures accurate modeling of low-level memory behavior and precise tracking of signature progression for store-type instructions.

```
# Testcase cp_rs1 (Test source rs1 = x1)
li x15, 0×d878dded # initialize rs2
mv x1, x3 # move sigreg value into rs1
addi x1, x1, 1238
sb x15, -1238(x1)
addi x1, x1, -1238
addi x1, x1, REGWIDTH
CHK_OFFSET(sigReg, XLEN/4, True)      # updating sigoffset
```

Code Example 6.3.1.3: Macro Usage in the I-extension (sb instruction)

Currently, macros have been implemented for the I (except jal), M, and Zicond extension tests within cvw-arch-verif. The goal is to extend the macro-based approach to all remaining extensions and the privileged test in the coming month. To enable this, the macro definition files (arch_test.h, encoding.h, test_macros_vector.h, and test_macros.h) have also been added to the cvw-arch-verif environment directory. To

prevent breaking compatibility for others working in the repository, all macros—except those used for test initialization and finalization—are replaced with empty versions.

6.3.2. riscv-arch-test Trap Handler

Tests for privileged extensions are often expected to trap, meaning that they cause an architectural state that the processor must respond to before continuing normal operations. A program, called a trap handler, responds to these traps. Initially, when creating tests, the team wrote their own trap handler to resolve interrupts and exceptions that appeared in the tests. However, as the overhead to implement more functionality in the trap handler increased, the team decided to adopt the trap handler created and maintained by `riscv-arch-test`. The `riscv-arch-test` trap handler has the additional benefit of already storing relevant aspects of the state to the trap signature region of the test whenever a trap occurs. While this saved significant time and effort, the adoption came with required changes to ensure compatibility between the tests and the trap handler. The trap handler also required some bug fixes. The following sections provide an overview of all of the required changes.

6.3.2.1. S-mode Interrupt Handler Macros

On a RISC-V processor, all S-mode interrupts may be raised either in Machine mode or Supervisor mode. However, depending on the mode in which the interrupt is handled, the trap handler will have different CSRs available for it to read from or write to. S-mode CSRs are accessible in either M- or S-mode, but M-mode CSRs are only accessible in M-mode. Therefore, interrupt handler logic must take different actions to reset interrupts based on the current privilege mode. However, at the time of integration, the `riscv-arch-test` trap handler was missing support for two separate interrupt handlers for the same interrupt type.

Adding this functionality required two changes. First, the previous interrupt handler selection logic only allowed one interrupt handler function for one interrupt type. Adding selection logic for two interrupt handlers for the same interrupt type was done in the fashion depicted in Code Example 6.3.2.1. The `riscv-arch-test` trap handler already contained macros to detect the current privilege mode, making the selection logic very simple and readable. The second change required was to add the two separate interrupt handler functions into the Sail and Spike model configuration files used to compile the tests. Only the declarations of the macros are necessary in the `riscv-arch-test` configuration files, as the specific sequence of instructions required to execute these functions may be implementation-specific. The handler routines used by this project are within the model-specific configuration files in `$WALLY/tests/riscof/sail_cSim/env` and `$WALLY/tests/riscof/spike/env`. The specific functions are provided in Code Example 6.3.2.2.

```
\__MODE__\()clr_Ssw_int:                // int 1 default to just return if not defined
                                        // S-mode software interrupts need to be reset
                                           differently when raised in M or S mode
        .ifc \__MODE__ , M              // Select the interrupt handler function based
                                           on current privilege mode
            RVMODEL_MCLR_SSW_INT
        .else
                .ifc \__MODE__ , S
                        RVMODEL_SCLR_SSW_INT
                .else
                        RVMODEL_CLR_SSW_INT
                .endif
        .endif

        j       resto_\__MODE__\()rtn
```

Code Example 6.3.2.1: Logic for Selecting Interrupt Handler Macro

```
#define RVMODEL_MCLR_SSW_INT \
csrrci t6, mip, 2;

#define RVMODEL_SCLR_SSW_INT \
```

```
csrrci t6, sip, 2;
```

Code Example 6.3.2.2: Mode-dependent Interrupt Handler Macros

6.3.2.2. Mode-Change Macros

The `riscv-arch-test` trap handler uses two macros to handle changes in privilege mode: `RVTEST_GOTO_MMODE` and `RVTEST_GOTO_LOWER_MODE {Smode/Umode}`. However, each of these macros had bugs in it at the time of adoption. Since the implementation of `ecall` in the `riscv-arch-test` trap handler uses the stack pointer (register `x2`) as its argument, the `RVTEST_GOTO_MMODE` macro would overwrite the current stack pointer whenever it was used. Since many Wally Interrupts* tests use the stack pointer, `RVTEST_GOTO_MMODE` had to be changed to temporarily store the stack pointer in t0 before it was used for `ecall`, then restore it afterward. While this solution is not ideal due to wasting some instruction cycles, it was selected because changing the implementation of `ecall` could have cascading effects into other tests that have already been implemented. The team decided that in the interest of time, it was not worthwhile to chase down all of the possible effects of such a change and instead opt for a simpler solution.

The other bug was found in the `RVTEST_GOTO_LOWER_MODE` macro. When a test is run in the `riscv-arch-test` framework, some setup code stores information relevant to the trap handler in a “save area” in program memory. The base address of this save area is stored in the CSR `mscratch`, and this address is typically loaded into the stack pointer by the trap handler whenever this information needs to be accessed. This load is necessary before any trap handler stack operations, since a program could overwrite the stack pointer at

any point in execution. However, the RVTEST_GOTO_LOWER_MODE macro did not contain any code to perform this address load. Therefore, it attempted to load and store to memory that was likely to be meaningless. Fortunately, the loading of mscratch into the stack pointer is implemented with a single instruction, so adding this into the macro's definition right before the loads/stores (and restoring the stack pointer afterward) was the only action required to fix this macro.

6.3.3. Running Tests Using riscv-arch-test

Now that tests are being ported over to riscv-arch-test, the procedure to run these tests will be different from the guide presented in section 3.5. Like before, run bin/testgen.py to generate the unprivileged tests. These tests can then be moved into our fork of riscv-arch-test, named cvw-riscv-arch-test using the command: make riscv-arch (for all unprivileged tests) or make riscv-arch-<extension_name> (for just one extension) from the cvw-arch-verif directory. This will generate the cvw-arch-verif tests with testgen.py and copy the assembly files over to the addins/cvw-riscv-arch-test directory.

Once the tests are in cvw-riscv-arch-test, use make cvw-riscv-arch-test --jobs from the $WALLY/tests/riscof directory to run all of the previously copied tests with RISCOF. This will compile them with the riscv-arch-test macros defined, run them on Spike and Sail, compare the signatures from the two reference models, and measure the coverage based on the Sail log. The coverage results are placed in addins/cvw-arch-verif/work, and the coverage reports are named rv64i and rv32i.

If all tests pass, run `regression-wally --fcov-act` to run the RISCOF compiled versions of the tests on Wally and collect coverage using the DUT method described previously in Section 3. This currently runs them in lockstep on Wally only, meaning it does not compare the signature with the one generated by Sail. The coverage report is generated in `$WALLY/addins/cvw-arch-verif/work` and can be used for identifying issues with the new Sail based coverage flow.

Note that the automated test-copy commands (`make riscv-arch-*`) are designed only for unprivileged tests. For privileged tests, files must be copied manually into the `cvw-riscv-arch-test` directory and should eventually be committed to that repository once they are ready to be used with RISCOF[23].

Additionally, if RISCOF fails (e.g., due to signature mismatches between Spike and Sail), compiled tests are not copied to the `$WALLY/tests/riscof/work` directory, and `regression-wally --fcov-act` will not find them. In such cases, fix the error, manually copy the tests, or run individual `wsim` runs. In the failure case, the compiled test files can still be found in `$WALLY/tests/riscof/riscof_work.`

7. Bugs Found Through Verification

The Harvey Mudd team has found and corrected several bugs in CORE-V Wally as a result of the functional verification tests. The summary of these bugs is found in table 7.1.

Description	Affected Instruction	Root Cause	Issue Number	Fixed in PR Number
fround result missing ones	`fround`, `froundnx`	Arithmetic right shift incorrect	1025	1044
fmvp result nonsensical	`fmvp`	Incorrect control signals	1055	1069
fround truncating for negative args	`fround` (with `rmm`)	Typo in case statement	1056	1058
fp flags floating	`froundnx`	Incorrect case statement	1066	1093
froundnx asseriting NX for sNaN input	`froundnx`	Missing logic	1095	1093
Misaligned lr.x and sc.x should throw an Access Fault	`lr.x` and `sc.x`	Incorrect exception thrown in logic	1348	1347
Incorrect Mepc update during an IAF	load instructions	TLB and LSU interaction	1350	W.I.P
Virtual memory SV32 reserved test mismatch	`jalr`	Dut executes instruction instead of raising fault	1198	1206
Decoder shift bug	logical shifts	Shifts larger than 31 should throw exception in decoder	1150/1147	1144

Description	Affected Instruction	Root Cause	Issue Number	Fixed in PR Number
RV32GC f-long exceptions	converts to/from long, moves to/from double	Should throw illegal instruction exception	1149	1144
Mtvec Mismatch	`wfi`	Interrupt timer offset issues	1111	1113
PMP Interval Bug	Any store into a pmp region	Needs byte access implementation	1354	W.I.P

Table 7.1: Summary of Bugs Caught by Functional Verification

We find bugs when simulating in lockstep with ImperasDV, which outputs a warning message in the terminal when the architectural state of the device under test does not match the reference. Code Example 7.1 demonstrates the terminal output for a reference model mismatch.

```
# Info (IDV) ──────────────────────────────────────────────────
# Info (IDV) Instruction executed prior to mismatch '0×80000020(rvtest_entry_point+20): b2b08153 fmvp.d.x
f2,x1,x11'
# Error (IDV) FPR register value mismatch (HartId:0, PC:0×80000020 rvtest_entry_point+20):
# Error (IDV) Mismatch 0> FPR f2
# Error (IDV)   . dut:0×41ad2e284c000000
# Error (IDV)   . ref:0×a563c1850e971426
# Error (IDV) testbench.idv_trace2api.state_compare @ 560: MISMATCH
```

Code Example 7.1: Terminal Output for Reference Model Mismatch

From this example, we see the program counter and mnemonic for the instruction executed just before the mismatch (`0×80000020`), the location in the architectural state where the mismatch occurred (floating-point register (FPR) `f2`), as well as the values in that location for the reference model and device under test. Once a mismatch is detected, the team must inspect the test program to confirm that the reference model produced the expected

value and Wally did not, as there have been cases of mismatches caused by ImperasDV being configured incorrectly.

The process of debugging generally involves starting a session with the remote desktop software X2GO and running

```
$ wsim rv32gc test.elf –gui
```

to start a simulation with the Questa graphical user interface (GUI). With the GUI, inspect the waveforms of the signals within Wally as it executes the instructions. To narrow down the specific mismatch, we scroll to the beginning, click on the PCM (program counter in the memory stage) signal, type in the program counter where the mismatch occurred in the search box, and click the forward arrow. It may be necessary to click the arrow multiple times as the PC can be repeated (e.g., if the instruction was fetched but flushed before being executed). Once the waveform shows where the instruction is retired, we can examine the other waveforms and walk through the intended behavior to find the exact bug and fix it. It is helpful to inspect the Verilog and the block diagrams to outline the expected values of all relevant signals and trace the source of the incorrect signals.

7.1. Floating-Point Round Instructions

Tests for `fround.*` and `froundnx.*` instructions revealed that Wally occasionally dropped a few ones from the output of a round operation when compared to the reference model. For example, when a source register is loaded with the value `0×c3f0787f` (decimal -480.941375732421875) before a `fround.s` instruction, the value was rounded by the reference to `0×c3f08000` (decimal -481.0) but incorrectly rounded by Wally to

`0×c3808000` (decimal -257.0), Code Example 7.1.1 shows this discrepancy. The team consulted the documentation for the rounding algorithm used in Wally. After stepping through the operation by hand, we found that the reference model was correct, and as such, this was not due to a configuration issue with ImperasDV, so the issue was with Wally's RTL. Upon inspection of the Questa waveforms produced by executing the tests, the team found that an arithmetic right shift was not producing the expected result, as a signal with a 1 in the most significant bit has the upper bits filled with zeros. Investigating the SystemVerilog specification revealed that `logic` signals are regarded as unsigned by default, and the function `$signed(<signal>)` is needed to copy the sign bit with an arithmetic right shift. After correcting this, Wally matched the reference model for this input.

```
flw     f1,0(x2)
fround.s f1,f1
        # Error (IDV) FPR register value mismatch
        # Error (IDV)   . dut:0×ffffffffc3808000
        # Error (IDV)   . ref:0×ffffffffc3f08000
```

Code Reference 7.1.1: `fround` Mismatch

7.2. Move Register Pair to Floating Point Register Instruction

The `fmvp.d.x` instruction moves the contents of two 32-bit integer registers to a 64-bit floating-point register. This instruction is missing from the `riscv-arch-test` and has gone largely untested. Our tests found mismatches for every instance of `fmvp.d.x`. The team traced the issue to internal control signals, which were inconsistent with integer-to-floating-point moves. After correcting this bug and an analogous bug affecting the untested `fmvp.q.x` instruction, the instruction is no longer mismatched. These two

instructions are missing from the `riscv-arch-test` repository, which has allowed this bug to persist to the present. Code Reference 7.2.1 shows the scenario that produced the mismatch.

```
# Testcase cp_rs1 (Test source rs1 = x0)
lui     x0,0×abb1
addi    x0,x0,-761
lui     x22,0×d77e9
addi    x22,x22,-1557
fmvp.d.x        f2,x0,x22
        # Error (IDV) Mismatch 0> FPR f2
        # Error (IDV)   . dut:0×41ad2e284c000000
        # Error (IDV)   . ref:0×a563c1850e971426
```

Code Reference 7.2.1: `fmvp` Mismatch

7.3. Floating Point Round with RMM Rounding Mode

After the team corrected the initial `fround.*` bug, we began testing explicit rounding modes for all instructions that can round their results. Naturally, `fround.*` required this coverpoint. In lockstep simulation, this instruction caused a mismatch with the explicit `rmm` founding mode. For negative values, Wally was truncating rather than rounding towards the more negative even number when necessary. Tracking this in the waveforms, the team found that a case statement in the rounding unit RTL has a reserved rounding mode encoding associated with `rmm`. With this simple change, Wally no longer mismatches for this instruction and rounding mode.

```
# Testcase cp_frm
        li x28, 0×c9e4835b # initialize rd to a random value that; helps covering rd_toggle
```

```
la x2, scratch
li x3, 0×9f19c872 # load x3 with value 0×9f19c872
sw x3, 0(x2) # store 0×9f19c872 in memory
flh f11, 0(x2) # load 0×9f19c872 from memory into f11
fround.h f28, f11, rmm # perform operation
       # Error (IDV) Mismatch 0> FPR f28
       # Error (IDV)   . dut:0×fffffffffffc800
       # Error (IDV)   . ref:0×fffffffffffc880
```

Code Reference 7.3.1: `fround rmm` Mismatch

7.4. Floating FFlags

The tests for the `froundnx` instruction revealed instances where floating point flags were unexpectedly present. While floating values may be permissible internally during intermediate computations, they should never manifest at the architectural level in states accessible to or modifiable by the user. Upon further investigation, the team traced the issue to the floating comparator unit, which was inadvertently generating a floating value. This behavior occurred because the comparator had not been updated to account for specific cases introduced by the `fround` instruction group. By explicitly enumerating these missing cases and ensuring that the comparators' inexact signal was consistently tied low for this instruction, the team successfully resolved the issue. Code Reference 7.4.1 shows the instruction that caused a mismatch. The specific values of the mismatch were not recorded in issue 1066 and it cannot be recreated easily since the SystemVerilog code was modified.

```
fcvtmod.w.d s6,fa2,rtz
```

Code Reference 7.4.1: `fflags` Mismatch

7.5. Incorrect NaN-Boxed Input Behavior

When conducting additional testing of the `froundnx` instruction, the team identified a discrepancy between its behavior and that of the reference model. Specifically, the inexact flag was being erroneously triggered in scenarios where it should not have been, particularly when the input was not properly NaN-Boxed. NaN-Boxing is the process of filling the upper, unused bits of a floating-point register with 1s when the value stored is a lower bit precision than the width of the register. For example, if a half-precision floating point value is written to a 32-bit register, bits 31 through 16 are set to 1. This practice ensures that the register will be a NaN if interpreted as a larger precision value than what was written by making the exponent bits all 1 and the mantissa non-zero. According to the RISC-V specification, the `froundnx` instruction is explicitly required not to raise the inexact flag if the input is classified as a NaN. To align with this specification and resolve the inconsistency, the Wally source code was revised to correctly incorporate this logic, ensuring adherence to the standard and predictable behavior under such conditions. The specific values of the mismatch were not recorded in issue 1095 and it cannot be recreated easily since the SystemVerilog code was modified.

```
froundnx.d  fs1,ft2
```

Code Reference 7.4.1: Nan-Boxing Mismatch

7.6. Sv32 Reserved Test Mismatch

While executing the SV32 Virtual Memory test suit on Wally all the tests passed except for the Reserved tests. This test attempts to execute memory access to page tables with reserved encodings. If the processor attempts to access a reserved memory encoding

when attempting to access memory with virtual memory enabled, a trap should be thrown. The memory operations `sw` and `lw` instructions correctly raised an exception when they attempted to access reserved memory locations. The DUT and REF mismatch occurs when there is a `jalr` to a reserved address in virtual memory. Before the bug fix, Wally only checked reserved addresses for the Data Translation Lookaside Buffer (Data TLB) and excluded checking instruction TLB. This means that the `jalr` instruction was executing since instructions were not checked if the jump target address was reserved. This led to a mismatch in CSR values between Wally and the reference model since Wally executed the `jalr` instruction while ImperasDV threw an exception as expected. After Wally's logic for reserved encoding was corrected, by adding logic to check the Instruction TLB, all reserved tests in SV32 passed.

```
add     x5,x15,x5
# Info   x5 003ffffc → 917ffffc
jalr    x1,0(x5)
# Info   x1 feedbead → 90000948
jalr    x1,0(x5)'
# Error (IDV) PC mismatch
# Error (IDV) Mismatch 0>
# Error (IDV)   . dut:0×917ffffc
# Error (IDV)   . ref:0×80001280 Mtrampoline+0
```

Code Reference 7.6.1: Sv32 Mismatch

7.7. Decoder Shift Bug

Wally was not throwing an exception for word shifts greater than 31 in RV32 while the reference model was. A word shift only operates on the lower 32 bits of a register. The

RISC-V specification states that any shift larger than 31 in RV32 for a word shift should throw an exception. After changing the exception logic to check for word shifts larger than 31, Wally and the reference model match. Code Reference 7.7.1 shows that the reference model threw an exception while Wally did not.

```
sraiw   x25,x28,0×2b: Illegal instruction - 128-bit XLEN is absent
        # Error (IDV) PC mismatch:
```

Code Reference 7.7.1: Decoder Shift Mismatch

7.8. RV32GC F-LONG EXCEPTIONS

While executing the floating point tests, Wally and the reference model had a mismatch on converts to/from long, moves to/from double, for example `fcvt.lu.w` and `fmv.x.d` respectively. A floating point conversion operation, `fcvt`, converts an integer in a register to its floating point equivalent and places it in a floating point register. The issue with these instructions is that longs and doubles are both larger than a word (32 bits). So, when these instructions are executed in the RV32 configuration these instructions are not supported; only word and half conversions/moves are supported. The floating point controller was updated to check if XLEN is 64 when attempting to execute converts to/from long and moves to/from double.

7.9. MTVEC MISMATCH

After changing the CLINT initialization, machine timer interrupts were not occurring. This was caused by of the way `wfi` was handled. After changing the CLINT initialization, the processor had to check if the current time + the offset was exceeded. Only then should it

stop waiting for interrupts. This was handled by conditionally updating the most significant word of the MTIMECMP register, signaling that the `wfi` is done waiting.

7.10. PMP Interval Bug

Physical Memory Protection (PMP) prevents any privilege mode from altering data in a memory region to prevent accidental or malicious memory manipulation. Before implementing a fix, Wally's PMP did not match if any byte of an access is within the PMP interval. To partially correct this, logic to check the top of range (TOR) and access size were added. This makes any byte access to a PMP region throw a trap. PMP match fixing is still in progress, and in the future, the team must add misaligned access matching for the PMP region.

8. Functional Coverage Results

As the stated purpose of the project was to achieve 100% functional coverage of the unprivileged and privileged RISC-V ISA specifications, the final metric of the success of this project is the number of extensions that reached 100% coverage. Table 8.1 provides a list of each extension and the coverage status as of May 2025.

Feature	Coverpoints	RV64 Test kLOC	Percent Coverage
Unprivileged			
I	468	81	100%
M	252	38	100%
A	244	21	100%
Zc{a,b,d,f}	233	14	100%
F, D, Zf{h/a}	1332	2284	100%
Zb{a,b,c,s}	672	80	100%
Zkn	292	13	100%
Zicond	28	4	100%
Zicbo*	In Progress	In Progress	
Privileged			
Zicsr	187	1.6	100%
Zicntr	39	1.9	100%
Exceptions	249	3	100%
Interrupts	187	4	80%
Endian	130	1.5	100%
PMP	In Progress	In Progress	
Virtual Mem	249	18	Sv32 & Sv39 at 100%

Table 8.1: Summary of Functional Coverage Completion

9. Project Management

Throughout the project, the team developed and maintained Project Management (PM) practices to help manage the division of work between individual members of the team and to ensure the entire project stays on track. The following sections detail the PM practices used for this project and a reflection on this past year's progress.

9.1. Overview of PM Practices Used

In the fall semester, weekly progress meetings were held among all liaisons and students to review task completion, discuss any roadblocks encountered, and plan the activities for the upcoming week. Assigned tasks, progress, and blockers were tracked through a shared spreadsheet among the three schools. In addition to the weekly progress meetings, the HMC team held one weekly advisor meeting to discuss team progress and get help with finishing touches on technical work. Two weekly dedicated group work sessions were also scheduled to ensure that team members receive an opportunity to start on technical work early with the help of their teammates. Starting in the Spring, the project liaisons held separate weekly progress meetings for each student team in place of the full-group meetings. When the meetings were split, the task tracker spreadsheet was also split into three sheets to track the deliverables of each student team independently.

9.2. Proposed vs. Actual Schedule Comparison

Based on their experience in the fall semester, the team created a chart for the spring semester to plan the relatively larger number of concurrent project threads. Figure 8.2.1 shows the Gantt chart that was created at the end of the fall semester. Compared to the

proposed schedule, completion of the privileged tests took much longer than expected, and some of the less critical deliverables were dropped from the project. The abandoned threads were assertion tests, torture tests for the memory system, IBM FP coverpoints, and functional coverage of the RISC-V Debug extension. Section 8.3 examines the reasons why the proposed schedule was not representative of how the work progressed.

	Corey Hickson (CH), Hamza Jamal (HJ), Ahlyssa Santillana (AS), Jordan Carlin (JC), Marina Bellido (MB), Roman De Santos (RDS)				
TASK NUMBER	**TASK TITLE**	**Owner**	**START DATE**	**PROJECTED DUE DATE**	**COMPLETION DATE**
0	**Introduction Material**				
0.1	Read Ch. 1-5 and 8 of RISC-V textbook	JC, MB, RDS	01/21/25	01/30/25	2/6/25
0.2	Login into chips server and make first PR on github README	JC, MB, RDS	01/21/25	01/23/25	1/23/25
1	**Covergroups**				
1.1	4 Interrupts	CH	01/21/25	02/06/25	3/5/25
1.2	3 Zicntr	CH	01/21/25	02/06/25	2/2/25
1	**Tests**				
1.1	Zicntr	AS	01/21/25	02/20/25	3/28/25
1.2	Exceptions	JRs	01/23/25	02/20/25	5/5/25
1.3	Interrupts	HJ	01/21/25	02/20/25	5/9/25
1.4	Endian	MB	01/23/25	02/20/25	2/22/25
2	**Extra Features**				
2.1	Assertions	CH	02/06/25	04/25/25	—
2.2	Breker	JC	02/20/25	04/25/25	—
2.3	Code Coverage	CH	02/20/25	04/25/25	5/9/25
2.4	Tool Docs	HJ	02/20/25	04/25/25	—
2.5	Sail Functional Coverage	JC	02/20/25	04/25/25	5/9/25
2.6	IBM FP	HJ, CH	02/20/25	04/25/25	—
2.7	Torture Test for Memory System	AS	02/06/25	04/25/25	—

2.8	fcov of RV debug		02/20/25	04/25/25	–
2.9	signature-based checking	HJ/MB	02/20/25	04/25/25	WIP
3	**Final Presentation/Report**				
3.1	Final Poster	ALL	03/31/25	04/25/25	4/25/25
3.2	Final Presentation	ALL	03/31/25	05/06/25	5/06/25
3.3	Final Report	ALL	03/31/25	05/07/25	5/09/25

Figure 8.2.1: Proposed/Actual Schedule for Spring Deliverables

9.3. Postmortem Assessment

The major issue with the pacing of the spring semester was that weekly deliverables were frequently missed for various reasons. The most common of these reasons was failing to start technical work early in the week and failing to ask questions in real-time when stuck. Another one of the issues with the pace of the spring semester was that the team underestimated the amount of ramp-up necessary for both new juniors and the continuing team members before they could begin writing privileged tests. While only a week was allotted for this process, it ended up taking between two and three weeks. In addition to the systematic issues, extenuating circumstances impacted some of the team members significantly. The potential impacts of the disturbances were not accounted for during planning, and this made it difficult to react to them in real-time.

10. Future Work

While this project achieved its core objectives of functional verification of the RISC-V RVA22S64 profile for CORE-V Wally, several enhancements were identified as valuable. However, they were not completed due to time constraints. Below is an outline of these opportunities for future work.

10.1. Signature-Based Tests and Handoff

As discussed in section 6.3.3, the team has decided to contribute their tests to `riscv-arch-test` to ensure that the tests are maintained beyond the end of this project. For this to occur, all tests must have support for signature-based checking, which is currently incomplete. The changes detailed in section 6.3.3 still need to be propagated to the entire test suite, so this will be completed by part of an HMC research team over the summer.

10.2. Self-Checking Tests

The `cvw-arch-verif` functional verification tests currently rely on lock-step simulation with ImperasDV. However, the RISC-V Certification Steering Committee requires tests to be self-checking, so to become the official open-source certification test suite, `cvw-arch-verif` must be adapted to include support for self-checking tests. Macros for these are currently being developed by Allen Baum (Esperanto) and Muhammad Hammad Bashir (10xEngineers).

10.3. IBM Floating Point Coverpoints

One of the dropped items from the current semester's deliverables was SystemVerilog coverpoints to describe IBM's floating point specification. While this is a large undertaking,

the floating point specification proposed by IBM is the most detailed specification of floating-point arithmetic coverage to date. `riscv-arch-test` currently contains an implementation that attempts to fully cover the IBM coverage spec, but there are glaring algorithmic mistakes that signal the need for more robust coverage measurements. Full coverage of the IBM floating point coverpoints would guide significant improvements in `riscv-arch-test`, but would require far more time to complete than would be feasible for this project.

10.4. Stress Tests

One enhancement that could benefit the verification suite would be the implementation of stress tests. Stress tests are designed to help evaluate the performance of a core under extreme or prolonged workload. Examples of these tests include memory tests which repeatedly read and write data to test memory performance. Another are branch tests that rigorously verify the correctness and robustness of branch predictions (beq, bne, blt, bge etc). These tests also help identify potential bugs and other edge cases that otherwise would not appear under normal operations. Tests targeting the memory subsystem and branch logic would be especially valuable.

11. References

[1] D. Harris, et al., *openhwgroup/cvw*. (May 09, 2025). Assembly. Accessed: May 14, 2025. [Online]. Available: https://github.com/openhwgroup/cvw

[2] Arm Ltd, “What is RISC?,” Arm | The Architecture for the Digital World. Accessed: Dec. 04, 2024. [Online]. Available: https://www.arm.com/glossary/risc

[3] Anderson, Tom. “Verification Challenges for RISC-V Adoption,” GSA - Global Semiconductor Alliance. Accessed: Dec. 03, 2024. [Online]. Available: https://www.gsaglobal.org/forums/verification-challenges-for-risc-v-adoption/

[4] Chen, Tony, and David A. Patterson. “History – RISC-V International.” *RISC-V International*, https://riscv.org/about/history/. Accessed 16 October 2024.

[5] “Apple M3 chip: Apple reportedly spent $1 billion on developing M3 chips.” Accessed: April 10, 2025. [Online]. Available: https://interestingengineering.com/innovation/apples-m3-chips-the-result-of-a-1-billion-project

[6] 10xEngineers. “RISC-V Design Services.” 10xEngineers, 2021, https://10xengineers.ai/risc-v-acceleration/. Accessed 16 October 2024.

[7] “Technology Readiness Levels - NASA.” Accessed: April 10, 2025. [Online]. Available: https://www.nasa.gov/directorates/somd/space-communications-navigation-program/technology-readiness-levels/

[8] “RISC-V Processors : The Comprehensive Guide (2024),” Stromasys. Accessed: Dec. 04, 2024. [Online]. Available: https://www.stromasys.com/resources/all-about-the-risc-v-processors/

[9] “The growth of RISC-V across industries - Electronic Products & TechnologyElectronic Products & Technology.” Accessed: Dec. 04, 2024. [Online]. Available: https://www.ept.ca/features/the-growth-of-risc-v-across-industries/

[10] “RISC-V Technical Specifications - Home - RISC-V Tech Hub.” Accessed: Dec. 04, 2024. [Online]. Available: https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications

[11] “Why open source? : google open source,” Google Open Source, https://opensource.google/documentation/reference/why/ (accessed Dec. 2, 2024).

[12] OpenHW group Core-V: Open Source RISC-V ..., https://riscv-europe.org/summit/2024/media/proceedings/plenary/Wed-09-30-Balaji-Baktha.pdf (accessed Dec. 3, 2024).

[13] K. McDermott, “Getting Started with RISC-V Verification – RISC-V International.” Accessed: Dec. 06, 2024. [Online]. Available: https://riscv.org/blog/2020/05/getting-started-with-risc-v-verification/

[14] H. Foster, “Part 8: The 2022 Wilson Research Group Functional Verification Study,” Verification Horizons, Accessed: Nov. 18, 2024 [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2022/12/12/part-8-the-2022-wilson-research-group-functional-verification-study/

[15] M. Aharoni, “Floating-Point Test-Suite for IEEE,” IBM Labs, version 1.02, Jan. 2008.

[16] M. Thompson, “CodeCoverageTRL-5.pptx,” Accessed: April 10, 2025. [Online]. Available: https://docs.google.com/presentation/d/1B_kt_JEcLfm1e7IEwnnPGHXxc1ww2tLA

[17] riscv-verification, *riscv-verification/riscvISACOV*. (May 09, 2025). SystemVerilog. Accessed: April 10, 2025. [Online]. Available: https://github.com/riscv-verification/riscvISACOV

[18] “Support covergroup · Issue #784 · verilator/verilator,” GitHub. Accessed: May 10, 2025. [Online]. Available: https://github.com/verilator/verilator/issues/784

[19] *riscv/riscv-opcodes*. (May 09, 2025). Python. RISC-V. Accessed: April 10, 2025. [Online]. Available: https://github.com/riscv/riscv-opcodes

[20] R. Thompson *et al.*, “Shared Recurrence Floating-Point Divide/Sqrt and Integer Divide/Remainder With Early Termination,” *IEEE Transactions on Computers*, vol. 74, no. 2, pp. 740–748, Feb. 2025, doi: 10.1109/TC.2024.3500380.

[21] “Implement callbacks for state-changing events by kseniadobrovolskaya · Pull Request #494 · riscv/sail-riscv,” GitHub. Accessed: April 10, 2025. [Online]. Available: https://github.com/riscv/sail-riscv/pull/494

[22] “riscv-arch-test/spec/TestFormatSpec.adoc at dev · riscv-non-isa/riscv-arch-test,” GitHub. Accessed: April 10, 2025. [Online]. Available: https://github.com/riscv-non-isa/riscv-arch-test/blob/dev/spec/TestFormatSpec.adoc

[23] “riscv-arch-test/spec/TestFormatSpec.adoc at dev · riscv-non-isa/riscv-arch-test,” GitHub. Accessed: April 10, 2025. [Online]. Available: https://github.com/riscv-non-isa/riscv-arch-test/blob/dev/spec/TestFormatSpec.adoc

[24] J. Carlin, *jordancarlin/riscv-arch-test*. (May 09, 2025). Assembly. Accessed: May 10, 2025. [Online]. Available: https://github.com/jordancarlin/riscv-arch-test