Detecting Specification Mismatches using Machine Learning-Based Analysis of CPU Manuals

by

Rachel Guzman

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2024 by the
Graduate Supervisory Committee:

Xusheng Xiao, Chair
Adil Ahmad
Samira Ghayekhloo

ARIZONA STATE UNIVERSITY

May 2024

ABSTRACT

Having properly implemented instructions is key to computer architecture and the security of a computer. Without properly implemented instructions, there is a risk of security vulnerabilities such as privilege escalation. Current methods of checking specification mismatches are the various versions of the manual approach and the use of symbolic execution. These current methods can be time-consuming or have issues with scalability and efficiency.

In this thesis, an approach is proposed to improve the current methods by employing the aid of machine-learning, specifically large-language models (LLMs), testing on RISC-V architecture. RISC-V architecture is proposed to test this method due to its simplistic nature and smaller instruction set compared to other architectures like x86. In this approach, Chat-GPT is proposed as the LLM of choice due to its rising popularity as well as its capability and power. The approach combines manual aspects and the aid of Chat-GPT to fully test how well Chat-GPT is at generating expressions and test cases to detect specification mismatches. The Chat-GPT generated test cases are evaluated on a RISC-V framework to see if the Chat-GPT generated test cases can be used in the future to detect specification mismatches as well as being used in more complicated architectures.

DEDICATION

This is dedicated to my family. Thank you for all of your support.

ACKNOWLEDGMENTS

Special thanks to my committee chair and advisor, Professor Xusheng Xiao, and committee member Professor Adil Ahmad for their invaluable guidance and encouragement throughout this thesis. I would also like to thank PhD students Liangyi Huang and Naven Subramanian Rajkumar for their assistance. Without their help, this would not have been possible.

I would also like to thank committee member Samira Ghayekhloo for her support.

Acknowledgement is due for the financial support provided by the National Science Foundation's CyberCorps: Scholarship for Service. Their generous contribution allowed me to pursue my higher education goals and has truly been significant.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

Computer architecture is the basis of any computer system. The architecture dictates how a computer operates [1]. Having a correctly implemented architecture and all corresponding instructions are key to having a computer run smoothly as well as protect the computer from any possible security attacks. There are many different architectures available, but this thesis is focused on the RISC-V architecture due to its simplicity. No matter the architecture type, however, identifying properties that, if violated, would leave a processor vulnerable to attack is a challenge and is often left up to security validation teams to determine if a design is vulnerable to attack [2].

With Chat-GPT [3] [4] on the rise, Large Language Models (LLM) have come to the forefront of innovation and aid without many normal users realizing they are interacting with a LLM. In this thesis, use of the assistance of the LLM Chat-GPT is proposed to help automate the verification process of CPU instructions to make it easier. With one of its key features being in-context learning, that being it is trained to provide a response based on context and a prompt [5], using LLMs was the strongest choice for automation.

The current methods for checking the CPU/OS are via a manual approach or symbolic execution. The manual approach, in the context of this thesis, includes automated test generation with non-trivial manual work for specific cases [6] as well as manual sorting of exploitable or not exploitable [2] , to name a few examples. Symbolic execution occurs when one supplies symbols rather than arbitrary values to

a program that can be then checked against path conditions when executed to check validity [7] [8].

This work is motivated by the need to address possible security vulnerabilities that can occur at the OS level. The objective is to use the assistance of LLMs, such as Chat-GPT, to take manuals as input, derive formal representations of pre- and post-conditions from the manuals, and automatically generate test cases for CPU instructions. Doing so will make it easier to check if the CPU is actually working properly and does not have any security issues. Previous works have seen a significant amount of manual work in specifying odd test cases [6], which is what this thesis will be aiming to improve with the uses of Chat-GPT.

In this thesis, we propose to leverage Chat-GPT4 to generate test cases to detect specification mismatches of RISC-V architecture instructions, which can simplify and improve the use of manual work to test for specification mismatches by feeding and creating formal expressions of instructions. Chat-GPT will be utilized to see how well it can generate general test cases for instruction sets to be able to test the full extent of these instructions to thus be able to detect specification mismatches in the future. By utilizing Chat-GPT, this will allow for quicker creation of test cases that will check for improper implementation. To test Chat-GPT's accuracy, the use of RISC-V architecture is employed due to the simplistic nature of the instructions in this architecture. This allows for a more simplistic check of Chat-GPT's accuracy before scaling and employing Chat-GPT on larger, more complex architectures.

The contributions of this thesis are as follows:

1. First, this thesis delves into RISC-V architecture and the background of LLMs.
2. It performs a review of current other approaches (manual and symbolic execution) as well as a review of the privilege escalation vulnerability.
3. Presents an approach combining the creation of formal expressions

and test cases with the automation of Chat-GPT to properly address instruction specifications.
4. Employs the use of a RISC-V64 framework that runs on QEMU [9] to test Chat-GPT's accuracy in designing test cases to detect specification mismatches.
5. Finally, analyze Chat-GPT's accuracy and determine if it can improve what has been done to check for specification mismatches.

An overview of the article is as follows: Chapter 2 discusses instruction set architecture and RISC-V, insights into LLMs, as well as a past security vulnerability that has occurred due to improper instruction checking. Chapter 3 discusses the current approaches to how people are verifying CPU instructions today. In Chapter 4, the details of the proposed approach are discussed. Following this, Chapter 5 evaluates the proposed approach and results. Chapter 6 will have a discussion about the results and findings. Finally, Chapter 7 provides a summary as well as potential future work.

Chapter 2

BACKGROUND

2.1 RISC-V

Processor design and design concerns are closely linked to the instruction set implemented [1]. Instruction Set Architecture (ISA), aka computer architecture, is an abstract model of a computer - the connector between a computer's hardware and software [10]. RISC-V is an ISA.

Proposed in the 1980s as an alternative to current architectures, RISC is a computer architecture philosophy that focuses on simplicity [11]. The basis of RISC is its simplicity. As such, RISC supports few and simple instructions; however, these instructions it supports are those that are frequently used [12]. In addition, RISC only supports a few and simple instruction formats and addressing modes as well as having the expectation of fixed instruction lengths [12]. Due to its simplicity, this makes RISC much easier to understand and grasp compared to some other computer architectures.

RISC-V, the instruction set architecture used for this thesis, is the fifth generation of RISC, developed in 2010 [11]. RISC-V was developed as a modular design with a base instruction set but allowing for extensions for additional functionalities [13]. Since it is a modular design, the optional extensions allow RISC-V to be customized based on a computer system's requirements, allowing for both small and large-scale applications [13]. The development of RISC-V was made to be flexible so that any new, additional instructions are optional, but available for use, for all future RISC-V

implementations [13]. For simplicity and relevance, only RISC-V instruction format and privilege level will be discussed more in-depth, though it should be noted that RISC-V has a specified ISA, specified registers, a control and status register (CSR), as well as supporting exceptions and interrupts.

Due to its simplistic nature, RISC-V has only six instruction formats based on the handling of immediate operands, although two of these formats are optional and used based on system needs. As mentioned previously, RISC-V has fixed instruction lengths; as such, all the instructions have a fixed length of 32 bits and need to be positioned at a memory address divisible by four [13]. The six instruction formats, with variants labeled, are as follows:

- R-type, for register-register instructions
- I-type, for register-immediate and load instructions.
- S-type, for store instructions.
- U-type, for instructions with a large upper immediate.
- B-type (variant), for conditional branch instructions.
- J-type (variant), for unconditional jump instructions. [13]

Even though it is simple, like other ISAs, there must be a way to allow the computer to switch between execution privilege modes. Privilege levels protect certain software stacks from executing at a lower privilege level. RISC-V supports three privilege modes: Machine-Mode (M-mode), Supervisor-Mode (S-Mode), and User-Mode (U-Mode). M-mode is inherently trusted and has access to all low-level access to the computer system, S-Mode is for OS usage, and U-Mode is for standard software applications [13].

2.2 Large Language Models

Language models are a form of machine learning as it is a major approach to advancing machines' language intelligence [14]. To take a step back, language models are one of the biggest examples of modern Natural Language Processing (NLP). NLP is the computer's ability to process and respond to human language that mirrors human ability by using computational linguistics, statistics, machine learning, and deep-learning models to do so [15]. Early models of NLP were hand-coded and rule-based and lacked nuance understanding of language whereas modern NLP systems use deep-learning models and techniques to "learn" as they process information to better understand nuance [15]. As mentioned previously, language models are an example of modern NLP, as they use artificial intelligence (AI) and statistics to predict sentences [15]. As an offset of NLPs, language models seek to generate the likelihood of word sequences [14].

Large Language Models (LLM) are the latest in the development of language models themselves, the fourth generation to be exact. LLMs came about in 2017 in an effort to scale up the existing Pre-trained Language Models (PLM) [16] [14]. The idea behind PLMs is to, as the name suggests, pre-train a large model using a vast collection of text [17]. Many PLMs then follow-up with fine-tuning or few-shot learning to be applied to various contexts or tasks [18]. The scaling up of PLMs into LLMs meant that LLMs had different behaviors and emergent abilities [19] in solving complex tasks [14]. As the fourth generation in language model development, LLMs built upon the preceding language models (statistical, neural, and pre-trained) and as such is considered general purpose task-solvers [14].

LLMs are based on transformer architecture and are pre-trained on massive

amounts of data to be able to accomplish a multitude of tasks including answering questions and assisting in translation and summarizing [20]. Transformer are able to handle sequential data efficiently which allows for the handling of parallelization and capturing long-range dependencies in text [5]. Transformer is a neural network architecture with a multitude of parameters that has self-attention mechanisms that allow the model to understand the relationships between inputs [21]. In addition to having Transformer foundation, LLMs train in two stages. The first stage is self-supervised learning, learning from the data itself without any human annotations [21]. The second stage is fine-tuning in which they are trained on small, annotated data sets to increase the knowledge gleaned from the self-supervised learning stage [21]. By using these two stages to be trained/learn, LLMs produce high-accuracy responses.

The major approach for humans to access LLMs is through a prompting interface [14]. As such, one of the biggest applications of LLMs is Chat-GPT, which is the chosen LLM used for this thesis. Chat-GPT is trained to follow an instruction via a human prompt and generate a detailed response [22]. Chat-GPT models are trained on a vast number of text data to be able to generate human-like responses to language prompts with high accuracy [20]. Chat-GPT was trained on a diverse 570 GB of internet texts, ranging from books to websites, over a vast range of topics and then subsequently trained through reinforcement learning from human feedback [21].

Chapter 3

MOTIVATION

The goal of this thesis is to demonstrate a way to make it easier to create test cases and thus test computer systems for proper instruction implementation to prevent security vulnerabilities, specifically mitigating the amount of human/manual labor required. As such, this chapter discusses current test generation approaches as well as an example of a security vulnerability that can arise from improper instruction implementation and checking.

3.1 Manual Approach

The most common method in the industry in validating security involves a mostly manual approach: designers and testers examine the specifications and designs themselves to identify and understand the necessary and desired security features of the processor [2]. There is no one specific definition of the manual approach like there is for symbolic execution (see Section 3.2). However, there is a commonality - the generation of test cases and the manual implementation of the testing and manual validation of such tests. There have been many approaches to the manual approach and here will be a discussion of a few of them.

One such approach is called the Cardinal Pill Testing [23] - a malware analysis tester. Their aim for their test cases was to generate a set of test cases for each instruction that explores all possible code paths, including defined and undefined behavior, and potential exceptions. To generate these test cases, the authors programmed a

template to automatically generate most test cases. In their code, they have a section for general-purpose initialization, which generates most of the test cases as registers and memory are initialized with pre-defined values, as well as a section for specialized initialization for specific modifications needed for specific test cases. From there, since random test generation cannot guarantee all paths and branches are followed, the test generation was followed up by manually analyzing instruction execution flows defined in Intel manuals and classifying all possible input parameter values into ranges that lead to distinct execution flows. The IA-32 Intel CPU architecture contains 906 instruction codes, which are grouped into five categories: arithmetic, data movement, logic, flow control, and miscellaneous to more easily examine. The instructions are then tested by initializing registers and memory with specific values and evaluating the outcomes.

Another approach is called Examiner [24], which combines a test case generator with a differential testing engine that detects inconsistent instruction streams by comparing the execution results of emulators and real devices. They generate representative test cases for instructions in a 32-bit instruction set from ASL (ARM architecture specification language) code. This is done by parsing the encoding schema of the instructions to get the encoding symbols, infer the type for symbols, generate an initialized mutation set, and develop a symbolic execution engine. Even though a symbolic execution engine is utilized for the Examiner, this method is, in the context of this thesis, considered a manual approach as the decoding and execution ASL code has limited constraints, resulting in limited paths. The differential testing engine then receives the generated instruction streams and detects inconsistent ones. To conduct the differential testing, instructions added at the beginning and end of the instruction stream are inserted, along with signal handlers, setting the initial state to zero, execute

an instruction stream, and dump the CPU state either after the execution or in the signal handler to be able to compare the execution result. Using Capstone [25] for memory, specifically to get the target memory address the instruction is written to, this is manually checked for its effectiveness, which it was. Finally, they manually compared the result from the emulator and a real device to find inconsistent instruction streams.

3.2 Symbolic Execution

Introduced in the 70s, symbolic execution is a program analysis technique to see whether a piece of software can break certain properties [26]. Symbolic execution provides symbols representing arbitrary values rather than normal inputs, such as numbers, to a program [7]. As such, the output values determined by the program are described based on the input symbolic values [27]. A program is executed symbolically over a class of inputs, which is equivalent to running a large number of normal test cases [7]. The class of inputs, and thus how many normal test cases it is essentially equivalent to is dependent on the control flow of the program's inputs [7]. The use of symbolic execution ranges from automated test input generation to proving program partial correctness [27]. The execution itself is carried out by a symbolic execution engine that keeps track of two things for each path explored: a boolean formula that keeps track of the conditions of the branches along a path, updating with each branch execution and a symbolic memory store that links variables to symbolic expressions or values, updating with each assignment [26]. A model checker is then used to check whether there are any rule violations along each path and if a path can actually happen with concrete values in place of the symbolic inputs [26].

There are a few types of symbolic execution techniques such as classic, dynamic, forward, selective, and backward. Classic symbolic execution techniques analyze a program without running it, that being it is essentially a static analysis technique [28]. On the other hand, dynamic symbolic execution techniques collect symbolic constraints at run time during concrete executions [28]. Forward symbolic execution is when a symbolic execution engine analyzes multiple paths at the same time starting from one main entry point [26]. Selective symbolic execution combines concrete and symbolic execution in an effort to apply the thought that one may only want to explore certain aspects of the software stack and not care about others [26]. Backward symbolic execution is when the exploration begins at the target point and moves to the entry point in order to find a test input that can trigger a specific line of code, which can be very helpful for debugging or regression testing [26].

No matter the technique, the execution paths of the program can create an execution tree [7]. A node in the tree is associated with a number labeled statement that is executed and a connected directed arc joining any associated nodes representing transitions between statements [7]. Since each path in the tree created is independent of another, there have been efforts at parallelizing symbolic execution and splitting the exploration process to multiple people [28].

In theory, exhaustive symbolic execution is complete and sound, that being able to generate all possible control flow paths a program can take using specific inputs on concrete execution, but this is infeasible in practice [26]. However, symbolic execution has been successfully used in regards to software testing, vulnerability analysis, malware analysis, and exploit generation [29]. When symbolic execution is run, the system follows branches based on symbolic values, maintaining path conditions, that being a set of constraints, that the path must follow to be valid [8].

If a path ends or encounters an error a test case can be generated by finding specific values that satisfy the path condition [8], thus providing a way to patch the error.

3.3 Security Vulnerabilities

There is no approach for verification that is perfect. There have been a various number of security breaches due to improper instruction implementation and verification. Here, one such instance will be discussed: the Intel SYSRET privilege escalation. Although these vulnerabilities occurred on Intel x86 architecture, which is more complicated than RISC-V, the same principle still applies in that the instruction was not implemented and verified correctly, leaving the system vulnerable.

Privilege management is key to the security of computer architecture. Making sure privilege levels are maintained and implemented correctly, especially the ability to execute code in RAM, can prevent attacks such as buffer overflows and malicious code execution [30]. While RISC-V supports three privilege modes (M, S, U) [13], in x86, the CPU has four privilege rings starting at 0, with that being the most privileged and going to 3, with that being the least privileged [31]. Ring 0 runs kernel code, whereas ring 3 is the user space, running applications; rings 1 and 2 are not really used in practice [31]. Instructions that can access machine registers such as those containing security-critical instructions are fully available in kernel mode (ring 0) whereas trying to execute those same instructions running in user mode (ring 3) would be denied and an exception would be raised [32]. As there is such a difference in what is run between rings 0 and 3, it is important that switching between the modes is done properly and access to one ring from another is isolated, otherwise, there is the risk of

security vulnerabilities due to the security-critical instructions available to manipulate in kernel mode.

The Intel SYSRET allows for privilege escalation to occur between kernel (ring 0) and user (ring 3) mode when an operating system is written to AMD specifications running on Intel hardware [33]. This is due to the difference in implementations between AMD and Intel [33] and affects a number of operating systems including Xen, NetBSD, FreeBSD, some versions of Microsoft Windows (including Windows 7) [34]. On the surface, both AMD and Intel's specifications of SYSRET are the same, however, there is a subtle difference. They both load the instruction pointer register RIP from the RCX register then change the code segment selector to the corresponding mode [34]. Since RIP is used as a virtual address, it must be canonical but RCX can be any 64-bit number so a general protection fault (#GP) will be thrown if the address is non-canonical in RCX [34], which is where the problem arises. Both Intel and AMD specifications contain pseudo-code for what the instruction is precisely supposed to do [34]. In AMD, there is no explicit mention of the check for a canonical address in RIP and RIP in AMD is not assigned until after privilege level escalation and thus will throw #GP in guest mode [34]. On the other hand, Intel explicitly checks for a canonical address and happens before the privilege level is changed and thus will throw #GP in privilege mode [34]. Using Xen [35] switch from hypervisor to guest mode as an example, the problem arises in the provided specification pseudo-code.

> If a #GP is delivered in guest mode, the processor will load the hypervisor's RSP, the stack pointer, from a special hypervisor-designated entry point. But if the #GP is delivered while in hypervisor mode, the processor will use the current RSP, so that it can effectively "nest" the exception [34].

However, the hypervisor is what is in charge of changing RSP, not the SYSRET instruction itself [34]. As such, if the hypervisor has improperly updated RSP due to

when the #GP was thrown by the guest giving the hypervisor a non-canonical RIP, the guest can control the hypervisor, leading to possible security issues [34].

Chapter 4

APPROACH AND RESULTS

This chapter discusses the overall steps of the approach as well as the results as the results of Chat-GPT go hand-in-hand with the steps of the approach.

While Chat-GPT is a very capable LLM, it still may produce seemingly credible but incorrect responses [21]. As such, the project began with getting a baseline for functions and formal expressions to express RISC-V instructions to be able to check the accuracy of Chat-GPT. Chat-GPT has had a few versions since its birth, with GPT-4 being the most current and most powerful. As such, it was chosen to use GPT-4 for the generation of both formal expressions and test cases. It should be noted that, at the current time, to be able to access and use GPT-4, a paid or sponsored subscription is required. Twenty RISC-V instructions were chosen at random, using this GitHub [36] as the manual, to be the baseline and teaching instructions to check Chat-GPT's accuracy before having it generate the remaining thirty-two instructions, using the few-shot prompting approach. The overall steps of the approach are as follows:

1. Identify the pre- and post-conditions of the chosen instructions and check against Chat-GPT's response of what the pre- and post-conditions are to see how aligned they are.
2. Manually create formal expressions for the chosen instructions and check to see if Chat-GPT can correctly correlate the created formal expressions with the correct instruction.

3. Based on the created formal expressions, ask Chat-GPT to generate formal expressions for the rest of the instructions.
4. Using the formal expressions, ask Chat-GPT to generate test cases for the instructions.
5. Inject generated instruction test cases into RISC-V framework, using QEMU to do so, to check the correctness and ability of the test cases. This is to be explored in Chapter 5: Evaluation.

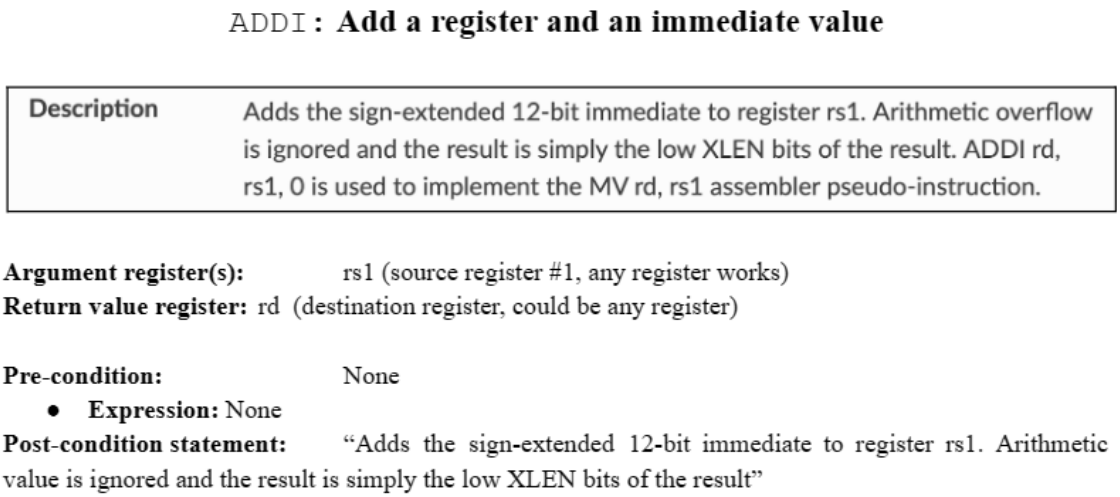
Step 0.5: Choosing Instructions

The unofficial first step of the approach was choosing the instructions. Even though Chat-GPT as a LLM and a machine-learning model is very capable, I will still be training it based on the formal expressions and functions I create. As such, manually choosing instructions and manually working with them is key. To choose the instructions, I used msyksphinz's RISC-V GitHub manual [36]. From there, twenty instructions were chosen at random while also attempting to choose at least one instruction from each possible sub-type of instructions. The chosen instructions to use as a baseline are as follows: ADDI, SLTI, LUI, AUIPC, ADD, SUB, SRET, MRET, ECALL, SLL, LB, SB, JAL, BEQ, BLTU, SRA, OR, SLTU, SRAI, and XORI.

Step 1: Pre- and Post-Conditions

To more deeply understand the instructions and what will need to be addressed in the formal expressions, I first extracted the pre- and post-conditions of the twenty instructions. For each instruction, I first kept track of the description of the instruction. From the instruction descriptions, I extracted argument register(s), return value

register, pre-condition statement, and post-condition statement. Figure 1 more clearly shows the format that I kept to keep track of them all.



`ADDI`: **Add a register and an immediate value**

Description	Adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI rd, rs1, 0 is used to implement the MV rd, rs1 assembler pseudo-instruction.

Argument register(s): rs1 (source register #1, any register works)
Return value register: rd (destination register, could be any register)

Pre-condition: None
- **Expression:** None

Post-condition statement: "Adds the sign-extended 12-bit immediate to register rs1. Arithmetic value is ignored and the result is simply the low XLEN bits of the result"

Figure 1. Pre- and Post-condition Tracking

Step 2: Formal Expressions - Manual Generation

In using formal expressions, it is the hope that it encompasses all options that need to be considered (i.e. overflow) and Chat-GPT can recognize this and thus create a test case that can trigger possible bugs. Utilizing the extracted pre- and post-conditions, I created formal expressions for each instruction. Table 1 shows the functions I created to make the formal expressions while Table 2 shows the formal expressions themselves.

ADDI	$rd =$ LOW_BITS$(rs1 + imm)$
SLTI	$rd = 1$ IF MIN(rs1, SIGN_EXT(imm)) ELSE $rd = 0$
LUI	$rd =$ LOW_BITS(imm)
AUIPC	$rd = pc$+LOW_BITS(imm)
ADD	$rd =$LOW_BITS$(rs1 + rs2)$

SUB	rd =LOW_BITS($rs1 - rs2$)
SRET	SIE = SPIE SPIE = READ(sstatus) JMP(sepc)
MRET	MIE = MPIE MPIE = READ(mstatus) JMP(mepc)
ECALL	READ(exceptionCall) IF exceptionCall == S spec = pc JMP(stvec) mstatus = S ELIF exceptionCall == M mpec = pc JMP(mtvec) sstatus = M
SLL	rd =LEFT_SHIFT(rs1, LOW_5(rs2))
LB	rd =LOAD_BYTES(rs1 + LOW_BITS(rs1))
SB	STORE_BYTES(LOW_BITS(rs2))
JAL	pc+ =SIGN_EXT(offset) rd = JMP($pc + 4$)
BEQ	IF rs1 == rs2 THEN BRANCH
BLTU	IF UNSIGNED_MIN(rs1, rs2) THEN BRANCH
SRA	rd =RIGHT_SHIFT(rs1, LOW_5(rs2))

OR	rd =OR_OP(rs1, rs2)
SLTU	$rd = 1$ IF MIN(UNSIGNED(rs1), UNSIGNED(rs2)) ELSE $rd = 0$
SRAI	rd =RIGHT_SHIFT(rs1, LOW_5(imm))
XORI	rd =XOR_OP(rs1, SIGN_EXT(imm))

Table 2. This Table Is a List of All the Formal Expressions I Created

After creating the formal expressions, I checked the created formal expressions against Chat-GPT to see if it could correctly identify the correct instructions. Table 3 shows the results of this check.

Instruction	GPT Correctly Determine?
ADDI	YES
SLTI	YES
LUI	YES (although also catalogs it as pseudo-instruction LI (load immediate) for small values)
AUIPC	YES
ADD	YES
SUB	YES
SRET	YES
MRET	YES
ECALL	YES
SLL	YES
LB	YES
SB	YES

JAL	YES
BEQ	YES
BLTU	YES
SRA	YES
OR	YES
SLTU	YES
SRAI	YES
XORI	YES

Table 3. This Table Shows How Many of My Formal Expressions Chat-gpt Was Able to Determine.

With the promising validation of Chat-GPT being able to correctly determine all of my created formal expressions, it was time to move on to the next step – get Chat-GPT to generate the rest of the RISC-V instructions.

Step 3: Formal Expressions - Chat-GPT Generation

Moving on to the next step, I fed Chat-GPT the following prompt:

> Given these defined functions: ∗inserted functions here (see Table 1 for full list – not inserted here for simplicity)∗, and knowledge of RISC-V architecture, and these created formal expressions as a guide: ∗insert created formal expressions here (see Table 2 and the tables in Appendix A for full list of formal instructions – not inserted here for simplicity)∗, generate a simple formal expression for the instruction ∗insert instruction want generated∗, generating new functions as necessary.

This prompt generated the following functions in Table 4 and the formal expressions in Table 5. It should be noted for all tables that have Chat-GPT generated responses, the

LOW_BITS	low XLEN bits
MIN	compare provided registers and determine if the first register is less than the second register
MAX	compare provided registers and determine if the first register is greater than the second register
SIGN_EXT	sign extended
JMP	jump to address
READ	read register value
LEFT_SHIFT	perform logical left shift
LOW_5	lowest 5 bits held in a register
BRANCH	take the branch
UNSIGNED_MIN	compare unsigned version of the provided registers and determine if the first register is less than the second register
RIGHT_SHIFT	perform arithmetic right shift
STORE_BYTES	Store 8-bit value
LOAD_BYTES	load 8-bit value
OR_OP	perform bitwise OR
UNSIGNED	unsigned
XOR_OP	perform bitwise XOR

Table 1. This Table Is a Glossary of the Functions I Created

contents of the table have been simplified to only include the pertinent information of the actual functions and formal expressions rather than the entire generated response.

COMPARE_LT	Returns 1 if x < y, else 0
COMPARE_LT_UNSIGNED (x, y)	Returns 1 if x (unsigned) is less than y (unsigned), else 0
AND_OP(x, y)	Returns the result of a bitwise AND between x and y
MEM_BARRIER(pred, succ)	Enforces that all memory operations specified in the predecessor set ‘pred‘ are completed before any memory operation in the successor set ‘succ‘ begins

INSTRUCTION_SYNC()	is a placeholder for the operation that ensures any recent writes to instruction memory are completed and visible to the instruction fetch unit. This is a conceptual representation, as the actual implementation details would depend on the internal architecture and behavior of the processor
READ_CSR(imm)	Returns the value in the CSR specified by ‘imm‘.
WRITE_CSR(imm, value)	Writes ‘value‘ to the CSR specified by ‘imm‘
READ_CSR(csr)	Reads the value from the CSR specified by ‘csr‘.
SET_BITS(x, y)	Returns a value with all the bits set in ‘y‘ also set in ‘x‘
CLEAR_BITS(x, y)	Returns a value where all the bits set in ‘y‘ are cleared in ‘x‘
WRITE_CSR(csr, val)	Writes the value ‘val‘ to the specified CSR
SET_BITS_IMM(x, imm)	Returns a value with the bits set in ‘imm‘ also set in ‘x‘
CLEAR_BITS_IMM(x, imm)	Returns a value with the bits cleared in ‘x‘ that are set in ‘imm‘
TRIGGER_DEBUG()	Triggers a transition to the debug environment
RESTORE_USER_ CONTEXT()	Restores the processor’s context/state from user mode exception handling
WAIT_FOR_INTERRUPT()	Instructs the processor to enter a state where it can wait for an interrupt efficiently

ENSURE_STORE_ COMPLETION	This function ensures that all previous store operations are completed before proceeding.
INVALIDATE_TLB	This function represents the invalidation of the Translation Lookaside Buffer (TLB), ensuring that subsequent memory accesses use the latest page table entries.
LOAD_HALFWORD(address)	Loads a 16-bit value from the specified address, sign-extends it to XLEN bits, and returns the result
LOAD_WORD(address)	Loads a 32-bit value from the specified address and returns the result
LOAD_BYTE_ UNSIGNED(address)	Loads an 8-bit value from the specified address, zero-extends it to XLEN bits, and returns the result
LOAD_HALFWORD_ UNSIGNED(address)	Loads a 16-bit value from the specified address, zero-extends it to XLEN bits, and returns the result
STORE_HALFWORD (address, value)	Stores the 16-bit value at the specified memory address
STORE_WORD (address, value)	Stores the 32-bit value at the specified memory address.
& ~1	operation clears the least significant bit for alignment
BRANCH_IF_NOT_ EQUAL(value1,value2,target)	Jumps to ‘target‘ if ‘value1‘ is not equal to ‘value2‘

Table 4. This Table Shows the Functions That Chat-gpt Created to Create the Formal Expressions in Table 5.

SLTIU	rd = COMPARE_LT_UNSIGNED(rs1, imm)
ORI	rd = OR_OP(rs1, SIGN_EXT(imm))
ANDI	rd = AND_OP(rs1, SIGN_EXT(imm))
SLLI	rd = LEFT_SHIFT(rs1, imm)
SRLI	rd = RIGHT_SHIFT(UNSIGNED(rs1), LOW_5(imm))
SLT	rd = COMPARE_LT(rs1, rs2)
SRL	rd = RIGHT_SHIFT(UNSIGNED(rs1), LOW_5(rs2))
AND	rd = AND_OP(rs1, rs2)
FENCE	MEM_BARRIER(pred, succ)
FENCE.I	INSTRUCTION_SYNC()
CSRRW	rd = READ_CSR(imm); WRITE_CSR(imm, rs1)
CSRRS	rd = READ_CSR(csr); IF $rs1! = 0$ THEN CSR = SET_BITS(READ_CSR(csr), rs1)
CSRRC	rd = READ_CSR(csr); IF $rs1! = 0$ THEN CSR = CLEAR_BITS(READ_CSR(csr), rs1)
CSRRWI	rd = READ_CSR(csr); WRITE_CSR(csr, LOW_BITS(imm))
CSRRSI	rd = READ_CSR(csr); IF $imm! = 0$ THEN CSR = SET_BITS_IMM(READ_CSR(csr), imm)
CSRRCI	rd = READ_CSR(csr); IF $imm! = 0$ THEN CSR = CLEAR_BITS_IMM(READ_CSR(csr), imm)
EBREAK	TRIGGER_DEBUG()
URET	RESTORE_USER_CONTEXT()
WFI	WAIT_FOR_INTERRUPT()

SFENCE.VMA	ENSURE_STORE_COMPLETION() AND INVALIDATE_TLB()
LH	rd = LOAD_HALFWORD(LOW_BITS($rs1 + imm$))
LW	rd = LOAD_WORD(LOW_BITS($rs1 + imm$))
LBU	rd = LOAD_BYTE_UNSIGNED(LOW_BITS($rs1 + imm$))
LHU	rd = LOAD_HALFWORD_UNSIGNED(LOW_BITS($rs1 + imm$))
SH	STORE_HALFWORD(LOW_BITS($rs1 + imm$), LOW_BITS(rs2))
SW	STORE_WORD(LOW_BITS($rs1 + imm$), LOW_BITS(rs2))
JALR	$rd = pc + 4$; JMP(LOW_BITS($rs1 + imm$) & ~1)
BNE	IF $rs1 != rs2$ THEN BRANCH(pc+SIGN_EXT(offset))
BLT	IF MIN(rs1, rs2) == rs1 AND $rs1 != rs2$ THEN BRANCH(pc+ SIGN_EXT(offset))
BGE	IF MIN(rs1, rs2) != rs1 OR $rs1 == rs2$ THEN BRANCH(pc+ SIGN_EXT(offset))
BLTU	IF UNSIGNED_MIN(rs1, rs2) == rs1 AND $rs1 != rs2$ THEN BRANCH(pc+ SIGN_EXT(offset))
BGEU	IF UNSIGNED_MIN(rs1, rs2) != rs1 OR $rs1 == rs2$ THEN BRANCH(pc+ SIGN_EXT(offset))

Table 5. This Table Shows the Chat-gpt Generated Formal Expressions for the Remaining 32 Risc-v Instructions

While Chat-GPT was able to generate a formal expression for all the remaining RISC-V instructions, some remain ambiguous. The instructions/formal expressions that remain ambiguous are those that rely on processor implementation, specifically noting such things for the FENCE, FENCE.I, and WFI instructions. This makes it difficult to generate general test cases as needed in the next step.

Step 4: Test Case Generation

To generate test cases, I fed Chat-GPT the following prompt:

> Given these defined functions: inserted functions here (see Table 1 and Table 4 for the full list – not inserted here to save space), knowledge of RISC-V architecture, and this formal expression insert formal expression for instruction want to generate here, generate an assembly test case.

This prompt came after one other iteration, as I originally asked Chat-GPT to just "generate a test case." However, due to the nature of some of the formal expressions I created, that being they include programming language such as if/else statements, Chat-GPT generated pseudo-code or example cases in other programming languages other than assembly. Since I would be testing in assembly, I then changed the prompt for Chat-GPT to specify I wanted an "assembly test case" to be generated. It should also be noted that for cases such as SRET, MRET, URET, and ECALL, I tweaked the prompt to generate an assembly test case in whichever mode is required (i.e. S-Mode, M-Mode, etc). With ECALL, I prompted Chat-GPT to generate a test case for each corresponding mode. Some of the resulting test cases are seen in Table 6. To see the rest of the test cases, see the tables in Appendix A.

ADDI	li a0, 10 addi a1, a0, 5

SLTI	li a0, 10 li a1, SIGN_EXT_IMM #Replace SIGN_EXT_IMM with actual immediate value slti a2, a0, SIGN_EXT_IMM
LUI	addi a1, zero, 1234
SRET	csrrs t0, sstatus, 0 #Read sstatus into t0 andi t1, t0, 0x20 #Extract the SPIE bit (5th bit) from t0 and store it in t1 slli t1, t1, 1 #Shift left t1 by 1 to align SPIE with SIE (assuming SIE is the 6th bit in sstatus) or t0, t0, t1 #Set the SIE bit in t0 csrrw zero, sstatus, t0 #Write back the modified value to sstatus csrr a0, sepc #Read sepc into a0 jalr zero, a0, 0 #Jump to the address in a0 (effectively the value in sepc)
SLL	li a2, 0x01 li a3, 4 sll a1, a2, a3

LB	#Mask the lower 12 bits of rs1 andi t0, rs1, 0xfff #t0 = lower 12 bits of rs1 #Add the masked value to rs1 add t1, rs1, t0 #$t1 = rs1 + t0$ #Load a byte from the address in t1 to rd lb rd, 0(t1)
SB	#Isolate the lowest 12 bits of x10 and use it as an address andi x11, x10, 0xFFF #x11 = lower 12 bits of x10 #Store the lowest byte of x10 at the address in x11 sb x10, 0(x11)
JAL	jal rd, label #Jump to 'label' and store pc+4 in 'rd' nop #This instruction will be skipped due to the jump label: #The target of the jump; execution continues here after the jump #Your subsequent instructions...

BEQ	li a1, 10 li a2, 10 beq a1, a2, equal_label #If a1 == a2, branch to equal_label j next_instruction #Jump to next_instruction if $a1 != a2$ equal_label: #The target label for the branch; this is where execution continues if $a1 == a2$ next_instruction: # Execution continues here if $a1 != a2$
SLTU	li a1, 1 li a2, 2 sltu a3, a1, a2 #$a3 = 1$ if a1 (unsigned) is less than a2 (unsigned), else a3 = 0
SRAI	li a1, 0x10 #Perform right shift arithmetic of a1 by an immediate value, say 3 #The immediate value is used directly, and only its LOW_5 bits are considered (which is the case by default) srai a2, a1, 3
XORI	li a1, 0x0F #Perform bitwise XOR between a1 and a sign-extended immediate value, say 0xFF xori a2, a1, 0xFF

ANDI	li t0, 0x0F0F #Perform ANDI with immediate value 0xFF, expected result is 0x0F andi t1, t0, 0xFF
SRLI	li t0, 0xFF srli t1, t0, 3
FENCE	#Assuming we've performed some memory operations beforehand #Let's assume we want to order writes before the barrier and reads after the barrier fence w, r #Perform operations after the barrier
CSRRW	#Assume 'imm' is the immediate value representing the CSR address. For example, let's use 0x300 for mstatus #In practice, 'imm' should be an encoded CSR address, not an immediate value directly used in instructions csrrs t0, mstatus, x0 #Read mstatus into t0, x0 is zero register #Assume rs1 contains the value to be written to the CSR, let's use t1 for rs1 li t1, 0x5 #Load some value into t1 csrrw x0, mstatus, t1 #Write the value from t1 to mstatus, don't care about the old value (hence x0)

EBREAK	#Trigger Debug ebreak #Verification #Verification of the debug trigger typically requires an external debugger #or a simulation environment capable of handling ‘ebreak‘
WFI	#Wait For Interrupt wfi #Post-Interrupt #Execute some instructions to indicate the processor has resumed execution nop #No operation, just a placeholder nop #Another no operation as a placeholder
LH	#Initialize rs1 with a base address, here we use 0x1000 as an example li t0, 0x1000 #t0 will be our rs1 #Assume there is a meaningful value at address 0x1000 + 0x002 = 0x1002 #For instance, let's assume we have initialized memory at 0x1002 with some value beforehand #Perform the load halfword operation with an immediate offset lh t1, 2(t0) #t1 will be our rd, load halfword from address in t0 plus offset 2

SH	#Initialize rs1 with a base address, here we use 0x1000 as an example li t0, 0x1000 #t0 will be our rs1 #Initialize rs2 with a value to store, here we use 0x1234 as an example li t1, 0x1234 #t1 will be our rs2 #Perform the store halfword operation with an immediate offset #We'll use an offset of 2, storing the value into address 0x1000 + 0x002 = 0x1002 sh t1, 2(t0) #Store the lower 16 bits of t1 into the address computed by t0 + 2

BLT	#Assume rs1 is in register t0 and rs2 is in register t1 #Assume a label 'branch_target' for the branch destination #Check if rs1 < rs2 (MIN(rs1, rs2) == rs1) blt t0, t1, check_inequality #If t0 < t1, go to check_inequality j next_instruction #Jump to next_instruction if not less check_inequality: #Check if rs1 ! = rs2 bne t0, t1, branch_target #If t0 ! = t1, branch to branch_target next_instruction: #Instructions that will execute if the branch is not taken nop #Placeholder for other instructions branch_target: #Target instruction if branch is taken nop #Placeholder for target instructions

Table 6. This Table Shows Some of the Chat-gpt Generated Test Cases.

Though not seen here, it should be noted that Chat-GPT still has issues with its own generated formal expressions, however, in the full Chat-GPT response, this is noted and addressed when creating the test cases. One such example is the CSRRW instruction where it was noted that "CSR addresses are usually encoded as part of the instruction and not passed as immediate values. The assembly syntax typically involves the CSR's symbolic name (like 'mstatus') rather than an immediate value.

This test case is a simplified representation to illustrate the concept." Another example is the SRET test case. With the SRET and MRET test cases, Chat-GPT generated more complicated test cases based on what the formal expression expressed, but then noted "This assembly snippet illustrates how the 'SRET' operation might be manually implemented in S-mode, although in practice, you would use the 'sret' instruction directly to achieve this behavior."

It should also be noted that while Chat-GPT has come up with seemingly viable test cases (to be fully discussed in Chapter 5: Evaluation), some test cases still require their own implementation (such as JALR) or assume that values have been pre-loaded (such as ADD and SUB).

In a quick glance, the test cases generated by Chat-GPT are valid test cases, though how well they do in testing will be explored in the next chapter, Evaluation. It should be noted before moving on to evaluating the test cases that I have asked Chat-GPT to generate an assembly case, asking very generally to see how well Chat-GPT is able to encompass edge cases without that specific prompting.

Chapter 5

EVALUATION

This chapter takes a look at the efficacy and accuracy of the Chat-GPT generated test cases. It examines how well-crafted the Chat-GPT test cases were as well as the test cases' ability to tackle the irregular and edge cases that may arise with instruction implementation. The testing of the generated test cases occurred by injecting the test cases on the default RISC-V framework available on QEMU [9] and observe if the test cases work as they should to be later tested to see if these test cases can be used to detect specification mismatches.

To test the generated test cases, I injected the cases on the default RISC-V framework on QEMU, downloading from the RISC-V branch in the instruction-testing Git-Hub created by Adil Ahmad and Naven Allen [37]. This specific framework allows the running of the test cases in the three RISC-V modes: User mode, Machine mode, and Supervisor mode, which will allow for the full testing of the Chat-GPT generated cases. As such, I am testing to see if the Chat-GPT generated cases behave as they should, using the assumption that the framework has no bugs. When generating the test cases in Step 4 of the Approach (see Chapter 4: Approach and Results section for more details), Chat-GPT had generated more than what was presented in Table 6 and the tables in Appendix A, including what should be in certain registers to validate the test cases, which is helpful in checking the validity of the test cases on the RISC-V framework. It should also be noted that while Chat-GPT generated viable test cases, there are quite a few that require manual editing to actually be able to run the test cases and are not just plug-and-play. One such example is SLTI where it has "li a1,

SIGN_EXT_IMM #Replace SIGN_EXT_IMM with actual immediate value", which requires manual editing to be able to run the code. Another example is the ADD and SUB test cases, as it assumes that a2 and a3 were already loaded with values. As such, for some cases, I manually changed some test cases to test them, though this should be mitigated in the future with better prompt engineering. However, besides some of these simple substitutions, I tested the Chat-GPT cases as they were generated to see how well they would work with little manual help For simplicity's sake, Figure 2 shows what the output of the run test cases looks like.

It is interesting to take note that with certain formal expressions that were generated by Chat-GPT, the corresponding Chat-GPT generated test cases resulted in the use of a different instruction, defeating the purpose of checking that instruction. Of the instructions that can be classified as "simple," that being arithmetic, logical, sets, shifts, memory manipulation (i.e. load and store), pc, jumps and branches, the instructions that had different instructions tested in their test case are LUI, SLTIU, ORI, and JALR. As such, these generated test cases were designated as fails since the test case was testing a different instruction.

In addition, with the load and store byte instructions (LB, SB), where the formal expression was created by me, Chat-GPT makes tries to use 0xFFF to get the lower 12 bits of a register. Even though 0xFFF is 12 bits, because the ANDI instruction is sign extended, it only takes in values from -0x7FF to 0x7FF, making 0xFFF an illegal operand and causing the test case to error out. In changing the 0xFFF to 0x7FF in the code, as well as changing the rs1 and rd registers to valid registers for the LB instruction test case specifically, both the LB and SB test cases ran and behaved as expected.

In all the "simple" instruction test cases, besides the aforementioned LUI, SLTIU,

ORI, and JALR, 100% of the test cases behaved as they should, with some modifications as previously mentioned. In regards to the "simple" instruction test cases as a whole, the Chat-GPT generated cases were 89.47% accurate.

While the "simple" instruction test cases were able to be evaluated and tested, the same could not be said of the more complex instructions, such as SRET, MRET, URET, ECALL, fence instructions, CSR instructions, and others of similar nature. For SRET and MRET, while the majority of the test case works how SRET and MRET would work manually rather than just using the instruction, Chat-GPT makes the assumption that sepc and mepc are pre-loaded with addresses and tries to jump to instruction address 0, which causes it to error out and not finish the test cases. With the URET instruction, Chat-GPT incorrectly handles the LW instruction, causing the program to error. The LW instruction is an I-Type instruction, so the value inside the parenthesis as the second argument must be a register. However, Chat-GPT provides an address (0x1000) in the parenthesis rather than a register. The LW instruction should look like lw x5, 0(t0) whereas Chat-GPT provided lw x5, 0(0x1000) in the test case. This small difference caused the URET test case to error. In changing this, the URET test case was able to be run, although this test case does not use the actual registers where the memory context is saved that the user wants to return to. With ECALL, Chat-GPT does not provide a fleshed-out enough test case. In its test case, it creates two labels, stvec_handler and mtvec_handler, both with just a "ret" and the comment "Handler code for S-mode/M-Mode would go here," which causes the code to stop running as these labels are referenced in the code to "Load the address" into a register, but since no address is loaded or coded, the code errors. With the fence instructions, EBREAK, and WFI, they are very vague and only have the instruction itself as the test case with no way to verify its correctness. Chat-GPT

does make a note, however, that with fence instructions, verification "depends on the specific hardware and execution context", which indicates it is capable of generating a correct test case for these instructions but requires specifics about the hardware; it is unable to generate a test in the scope of a simple assembly test case. With the CSR instructions, interestingly, the test case for CSRRW works exactly as expected; however, the same could not be said for the other similar instructions. CSRRS ends up error-ing out, while the other instructions, CSRRC, CSRRWI, CSRRSI, and CSRRCI had issues with how the test cases were generated. With CSRRC, Chat-GPT put the psuedo-instructions of "CSR_ADDRESS," "VALUE," and "csr" to be changed with actual CSR addresses and values relevant to the context of the system. With the other CSR instructions (CSRRWI, CSRRSI, and CSRRCI), Chat-GPT generated test cases utilizing the wrong versions of the instructions, CSRRW, CSRRS, and CSRRC respectively, which nullifies the ability to test if these instructions are tested correctly.

Since many of the test cases did not work due to requiring specifics about the hardware or needing a register to be initialized, I tried to tweak my Chat-GPT prompt to "generate an assembly test case for the RISC-V framework on QEMU v6.0.0." However, the response Chat-GPT gave did not differ in the actual test case provided, the only difference was that it provided information on how to install QEMU on your device. As such, I was still unable to test the generated test cases for the aforementioned problem instructions.

With the instructions I was able to test, the Chat-GPT generated test cases were successful and behaved as they should with no issues. However, in further examination of the generated test cases, most were very simple. This is actually a downfall of the working generated cases as they do not necessarily encompass all possible edge cases which can mean that the system is not fully tested and secure and detected

zero specification mismatches. Due to many of the generated test cases not working or having generated the wrong instruction to test, they are unable to be used to detect specification mismatches. Of all the instructions, only 67.3% of the generated test cases were viable. However, due to their simplistic nature, they do not catch specification mismatches and only take care of "normal" use cases. In addition, many of the run test cases had to be manually changed to accommodate certain values to be able to run the test case Chat-GPT generated; only a few of the Chat-GPT generated cases were able to run as is, which indicates that at the present time, Chat-GPT does not help decrease the amount of manual work that needs to occur for detecting specification mismatches.

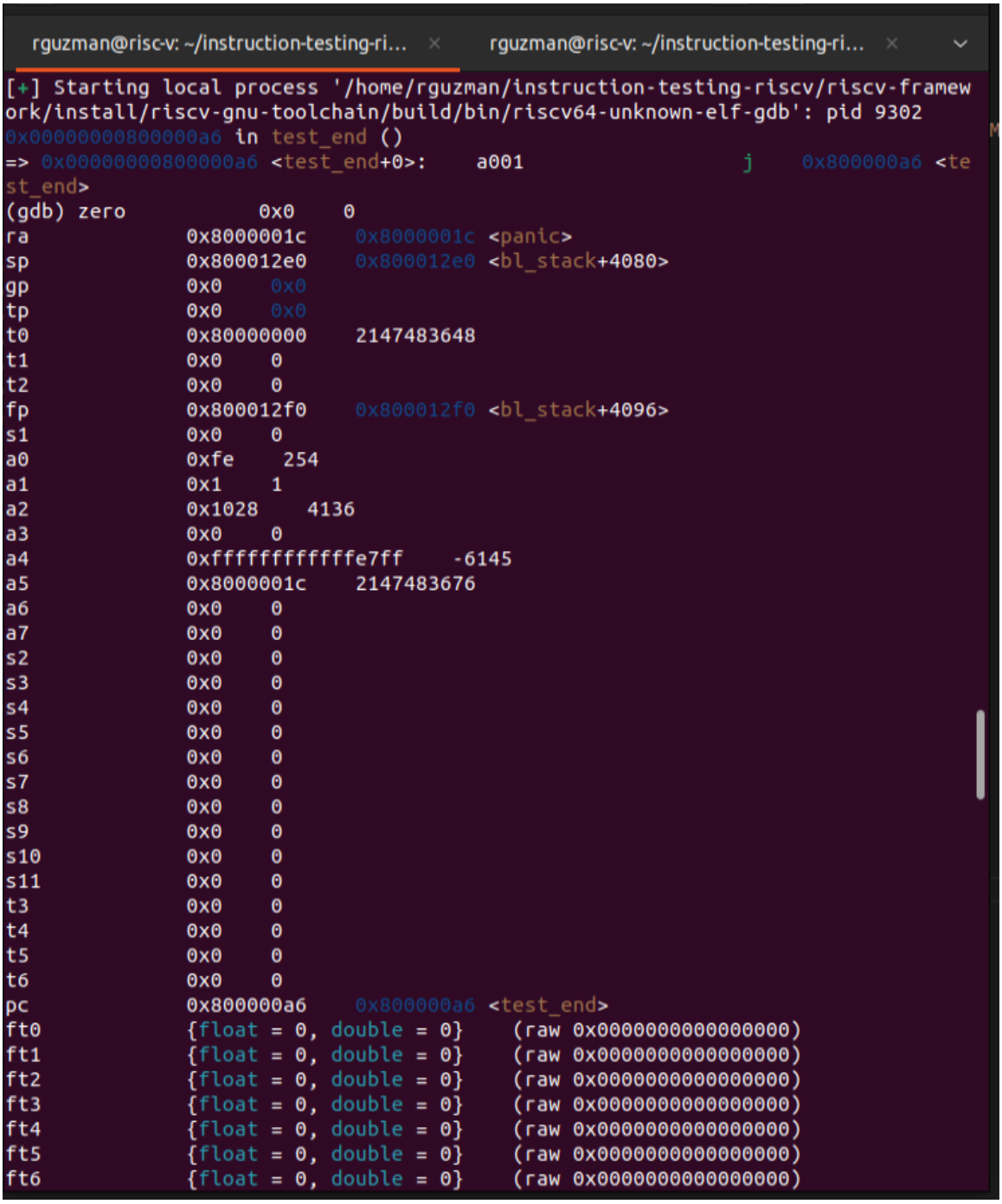


Figure 2. Sample output of running test cases on the RISC-V framework.

Chapter 6

DISCUSSION

As found in the Evaluation, there is a limit to current LLMs. By themselves, the LLM of Chat-GPT was unable to successfully generate usable test cases for all instructions in an instruction set. Many of the test cases generated required human integration to be usable. Of the fifty-two RISC-V instructions 40.38% were usable as generated. After human assistance, an additional 26.92% were usable test cases. Human assistance included changing the registers in the test cases, changing an immediate value, and adding a check to labels for jump or branch instructions to check their correctness. For the changing of registers, the LB instruction specifically had a test case generated using rs1 and rd as the registers, which are not real registers in the CPU. As such, these registers were changed to a1 and a0 respectively to run the test case. For changing an immediate value, SLTI specifically had SIGN_EXT_IMM in its test case, which needed to be changed to an actual immediate value. In addition, for SB and LB, 0xFFF was used as the immediate value to get the lower 12 bits of a register, but the ANDI instruction used to do so is signed so 0xFFF is out of range. As such, this 0xFFF immediate value was changed to 0x7FF. In addition to these changes, 19.23% of all the test cases require the prompt for the generation to include specifics about the hardware and the context of how the instruction is to be used and was unable to be tested in the context of this thesis. The instructions include ECALL, EBREAK, WFI, and fence instructions, as Chat-GPT just provided the instruction itself, with no set-up or verification provided to test if the instruction was behaving correctly. In the context of this thesis, test cases were considered able to test

if they were able to be run as-is or very small changes. In that regard, 13.46% of the instructions were thrown out as the wrong instruction was generated in the test case. The instructions that were thrown out were LUI, SLTIU, ORI, CSRRWI, CSRRSI, CSRRCI, and JALR. For SLTIU, ORI, CSRRWI, CSRRSI, CSRRCI, and JALR, test cases were generated using their non-immediate, or register, instruction counterparts, that being SLTU, OR, CSRRW, CSRRS, CSRRC and JAL, respectively. For LUI, Chat-GPT generated a test case using ADDI, despite being able to correctly identify the LUI formal expression when given it. However, Chat-GPT showed great promise as 67.31% of all generated test cases were usable and detected specification matches.

The few-shot prompting approach provided some promising results, but perhaps using chain-of-thought prompting would provide more test cases that are usable right off the bat. Chain-of-thought prompting includes providing intermediate steps in the prompting to give the LLM more reasoning capabilities. In addition, perhaps finding an even smaller or simpler instruction set to use over RISC-V would provide even more of a targeted, broken down task for the LLM to follow, generating more test cases that are accurate.

Chapter 7

CONCLUSION AND FUTURE WORK

Correctly implemented instructions are key to a fully functioning computer as well as key to the security of a computer. Being able to identify specification mismatches is key in catching possible security vulnerabilities that leave a computer at risk. The use of Chat-GPT and other machine-learning/LLMs has significantly increased in recent years. As such, an approach was proposed to see how well Chat-GPT, a large language model (LLM), would be at assisting in finding the instruction specification mismatches. This approach began by manually identifying the pre- and post-conditions of twenty semi-randomly chosen RISC-V instructions before taking those conditions and creating formal expressions for the instructions. Using the created formal expressions, Chat-GPT was asked to generate the remaining thirty-two RISC-V instructions. Once all the instructions had formal expressions, these were utilized in the prompt to Chat-GPT to generate test cases. The prompt given to Chat-GPT was a generalized prompt of "generate an assembly test case" to see how well it could do at generating a test case with edge cases without that specific prompting. It was found that the test cases that worked as they generated with no modifications beyond some substitutions, were simple but effective, but did not encompass many edge cases. Due to the nature of some of the Chat-GPT generated formal expressions, test cases were generated without the use of the instruction to be tested, which contradicts the purpose of testing that instruction. In addition, due to some assumptions that Chat-GPT makes, such as re-loaded addresses, some test cases error out, unable to even simply run. As such, there are possibilities that Chat-GPT can be used to detect specification mismatches,

but there is much work to be done by doing very specific prompt engineering for each specific system to be tested.

There is still much work that can be done with this approach and project, as this was just to get a baseline and see the capability of Chat-GPT to be used to make the finding of specification mismatches easier. For one, there can be more specific prompt engineering done to gain the test cases for all possible edge cases to make sure a system is fully tested, as I just asked Chat-GPT generally to see its baseline. For another, even though this approach mitigated the manual aspect of the manual approach, there are many manual aspects of the current proposed approach, but with the accuracy of the single test cases generated by Chat-GPT, this can be leveraged in the future to create a more automated process. One such way is to leverage Chat-GPT's ability to be interacted with via API keys in Python. Using the API keys instead of the UI can be leveraged in future works to clean up the prompt provided as well as streamline the extraction of the formal expressions and test cases generated in Chat-GPT's response, rather than doing so by hand. It can also be used to feed Chat-GPT multiple instructions at once to generate a large dump of formal expressions/test cases rather than doing them one by one.

REFERENCES

[1] M. Flynn, *Computer architecture*, Dec. 2007. DOI: 10.1002/9780470050118.ecse071.

[2] C. Deutschbein and C. Sturton, "Evaluating security specification mining for a cisc architecture," 2020. DOI: 10.1109/HOST45689.2020.9300291.

[3] OpenAI, "Chatgpt: Applications, opportunities, and threats," *arXiv preprint arXiv:2304.09103*, 2023.

[4] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[5] Y. Chang, X. Wang, J. Wang, *et al.*, "A survey on evaluation of large language models," *ACM Transactions on Intelligent Systems and Technology*, 2024, ISSN: 2157-6912. DOI: https://doi.org/10.1145/3641289.

[6] Q. Yan and S. McCamant, "Fast pokeemu: Scaling generated instruction tests using aggregation and state chaining," in *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '18, ACM, Mar. 2018. DOI: 10.1145/3186411.3186417.

[7] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, pp. 385–394, 1976. DOI: https://doi.org/10.1145/360248.360252.

[8] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, 2008, pp. 209–224.

[9] F. Bellard, "Qemu, a fast and portable dynamictranslator," *Proceedings of the USENIX Conferenceon Annual Technical Conference*, 2005.

[10] I. Baili, "A quick introduction to instruction set architecture and extensibility," 2021. [Online]. Available: https://www.embedded.com/a-quick-introduction-to-instruction-set-architecture-and-extensibility/.

[11] E. Corpeño, "An introduction to risc-v - understanding risc's open isa," 2022. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/introductions-to-risc-v-instruction-set-understanding-this-open-instruction-set-architecture/.

[12] S. O. Aletan, "An overview of risc architecture," 1992. DOI: https://doi.org/10.1145/143559.143570.

[13] B. W. Mezger, D. A. Santos, L. Dilillo, C. A. Zeferino, and D. R. Melo, "A survey of the risc-v architecture software support," *IEEE Access*, vol. 10, pp. 51 394–51 411, 2022, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3174125.

[14] W. X. Zhao, K. Zhou, J. Li, *et al.*, "A survey of large language models," 2023. DOI: https://doi.org/10.48550/arXiv.2303.18223.

[15] T. Ramanathan, "Natural language processing," *Encyclopedia Britannica*, 2024. [Online]. Available: https://www.britannica.com/technology/natural-language-processing-computer-science.

[16] M. McDonough, "Large language model," *Encyclopedia Britannica*, 2024. [Online]. Available: https://www.britannica.com/topic/large-language-model.

[17] S. Edunov, A. Baevski, and M. Auli, "Pre-trained language model representations for language generation," 2019. DOI: https://doi.org/10.48550/arXiv.1903.09722.

[18] H. Wang, J. Li, H. Wu, E. Hovy, and Y. Sun, "Pre-trained language models and their applications," *Engineering*, vol. 25, pp. 51–65, 2023. DOI: https://doi.org/10.1016/j.eng.2022.04.024.

[19] J. Wei, Y. Tay, R. Bommasani, *et al.*, "Emergent abilities of large language models," 2022. DOI: https://doi.org/10.48550/arXiv.2206.07682.

[20] E. Kasneci, K. Sessler, S. Küchemann, *et al.*, "Chatgpt for good? on opportunities and challenges of large language models for education," *Learning and Individual Differences*, vol. 103, 2023. DOI: https://doi.org/10.1016/j.lindif.2023.102274.

[21] Y. Shen, L. Heacock, J. Elias, *et al.*, "Chatgpt and other large language models are double-edged swords," *Radiology*, vol. 307, 2023, ISSN: 1527-1315. DOI: https://doi.org/10.1148/radiol.230163.

[22] OpenAI, "Introducing chatgpt," 2022. [Online]. Available: https://openai.com/blog/chatgpt#OpenAI.

[23] H. Shi, A. Alwabel, and J. Mirkovic, "Cardinal pill testing of system virtual machines," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 271–285.

[24] M. Jiang, T. Xu, Y. Zhou, *et al.*, "Examiner: Automatically locating inconsistent instructions between real devices and cpu emulators for arm," in *Proceedings of*

the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022, pp. 846–858.

[25] *Captsone*. [Online]. Available: https://www.capstone-engine.org/.

[26] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, pp. 1–39, 2018. DOI: https://doi.org/10.1145/3182657.

[27] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer*, vol. 11, pp. 339–353, 2009. DOI: https://doi.org/10.1007/s10009-009-0118-1.

[28] C. S. Păsăreanu, R. Kersten, K. Luckow, and Q.-S. Phan, "Symbolic execution and recent applications to worst-case execution, load testing, and security analysis," in *Advances in Computers*. Elsevier, 2019, pp. 289–314. DOI: https://doi.org/10.1016/bs.adcom.2018.10.004.

[29] N. Hasabnis and R. Sekar, "Extracting instruction semantics via symbolic execution of code generators," FSE'16, 2016. DOI: https://doi.org/10.1145/2950290.2950335.

[30] T. Lu, "A survey on risc-v security: Hardware and architecture," 2021. DOI: https://doi.org/10.48550/arXiv.2107.04175.

[31] L. Duflot, "Cpu bugs, cpu backdoors and consequences on security," *Journal in Computer Virology*, vol. 5, pp. 91–104, 2008. DOI: https://doi.org/10.1007/s11416-008-0109-x.

[32] M. Roitzsch, T. Miemietz, C. Von Elm, and N. Asmussen, "Software-defined cpu modes," 2023. DOI: https://doi.org/10.1145/3593856.3595894.

[33] Administrator, "Intel sysret," 2017. [Online]. Available: https://pentestlab.blog/2017/06/14/intel-sysret/.

[34] G. Dunlap, "The intel sysret privilege escalation," 2012. [Online]. Available: https://xenproject.org/2012/06/13/the-intel-sysret-privilege-escalation/.

[35] P. Barham, B. Dragovic, K. Fraser, *et al.*, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 164–177, 2003. DOI: https://doi.org/10.1145/1165389.945462.

[36] *Riscv-isa-pages*. [Online]. Available: https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html#addi.

[37] *Risc-v framework*. [Online]. Available: https://github.com/adilahmad17/instruction-testing/tree/riscv.

APPENDIX A

TABLES

The following tables are the remaining test cases generated by Chat-GPT broken down by type of instruction: arithmetic, logical, sets, shifts, memory, pc, jumps, branches, privilege, csrr types, and other.

ADD	add a1, a2, a3
SUB	sub a1, a2, a3

Table A1. This Table Shows the Remaining Chat-gpt Generated Test Cases in the Arithmetic Category.

OR	li a1, 0x01 li a2, 0x02 or a3, a1, a2
ORI	li t0, 0x0F #Assuming we want to sign-extend and load -1 (0xFFFFFFFF in 32-bit) into t1 li t1, -1 #Load the sign-extended immediate into t1 or t2, t0, t1
AND	li t0, 0x0F li t1, 0xF0 and t2, t0, t1

Table A2. This Table Shows the Remaining Chat-gpt Generated Test Cases in the Logical Category.

SLTIU	li t0, 0xFFFFFFFF #Load a large unsigned value into t0 li t1, 10 #Load the immediate value into t1 for comparison sltu t2, t0, t1 #t2 (rd) will be 1 if t0 (rs1) is less than t1 (imm), else 0 #For demonstration, branch to a label if comparison is false beq t2, zero, no_less_than #If t2 is 0 (comparison false), branch to no_less_than li t3, 1 #If we don't branch, set t3 to 1 to indicate success no_less_than: #Additional code can go here..
SLT	li t0, 5 li t1, 10 slt t2, t0, t1

Table A3. This Table Shows the Remaining Chat-gpt Generated Test Cases in the Sets Category.

SRA	li a1, -16 li a2, 2 sra a3, a1, a2
SLLI	li t0, 0x01 slli t1, t0, 4 #Perform left shift on t0 by 4 bits
SRLI	li t0, 0xFF srli t1, t0, 3
SRL	li t0, 0xFF li t1, 0x1A #Load a test value into t1 (rs2). The shift amount will use the lowest 5 bits, which is 0x1A & 0x1F = 0x1A. srl t2, t0, t1

Table A4. This Table Shows the Remaining Chat-gpt Generated Test Cases in the Shifts Category.

LW	#Initialize rs1 with a base address, here we use 0x1000 as an example li t0, 0x1000 #t0 will be our rs1 #Assume there is a meaningful value at address 0x1000 + 0x004 = 0x1004 #For instance, let's assume we have initialized memory at 0x1004 with some value beforehand #Perform the load word operation with an immediate offset lw t1, 4(t0) #t1 will be our rd, load word from address in t0 plus offset 4
LBU	#Initialize rs1 with a base address, here we use 0x1000 as an example li t0, 0x1000 #t0 will be our rs1 #Assume there is a meaningful value at address 0x1000 + 0x003 = 0x1003 #For instance, let's assume we have initialized memory at 0x1003 with some value beforehand #Perform the load byte unsigned operation with an immediate offset lbu t1, 3(t0) #t1 will be our rd, load unsigned byte from address in t0 plus offset 3

LHU	#Initialize rs1 with a base address, here we use 0x1000 as an example li t0, 0x1000 #t0 will be our rs1 #Assume there is a meaningful value at address 0x1000 + 0x002 = 0x1002 #For instance, let's assume we have initialized memory at 0x1002 with some value beforehand #Perform the load halfword unsigned operation with an immediate offset lhu t1, 2(t0) #t1 will be our rd, load unsigned halfword from address in t0 plus offset 2
SW	#Initialize rs1 (t0) with a base address, here we use 0x1000 as an example li t0, 0x1000 #t0 will be our rs1 #Initialize rs2 (t1) with a value to store, here we use 0xABCD1234 as an example li t1, 0xABCD1234 #t1 will be our rs2 #Perform the store word operation with an immediate offset #Let's use an offset of 4, storing the value into address 0x1000 + 0x004 = 0x1004 sw t1, 4(t0) #Store the word in t1 into the address computed by t0 + 4

Table A5. This Table Shows the Remaining Chat-gpt Generated Test Cases in the Memory Category.

AUIPC	addipc a1, 0 addi a1, a1, 20

Table A6. This Table Shows the Remaining Chat-gpt Generated Test Cases in the Pc Category.

JALR	#Initialize rs1 (t0) with a base address, for this example let's use 0x1000 li t0, 0x1000 #t0 will be our rs1 #we ensure that the target address is aligned by clearing the least significant bit #We'll use the JAL instruction to perform the jump and store the return address #For simplicity, we'll assume the label "target" is 8 bytes away from the current PC #The actual address will be computed during assembly jal t1, target #t1 will be our rd, storing PC + 4 #The following label represents the target instruction target: #Additional instructions can follow here

Table A7. This Table Shows the Remaining Chat-gpt Generated Test Cases in the Jumps Category.

BLTU	li a1, 2 li a2, 10 bltu a1, a2, less_than_label #If a1 (unsigned) is less than a2 (unsigned), branch to less_than_label j next_instruction #Jump to next_instruction if $a1 >= a2$ less_than_label: #The target label for the branch; this is where execution continues if a1 (unsigned) is less than a2 (unsigned) next_instruction: #Execution continues here if a1 is not unsigned less than a2

BNE	li t0, 5 #Load immediate value 5 into t0 (rs1) li t1, 10 #Load immediate value 10 into t1 (rs2) #Assume a label 'branch_target' represents the target instruction after the branch #The offset would be calculated based on the label's position relative to this instruction in the actual code bne t0, t1, branch_target #If t0 ! = t1, branch to 'branch_target' #Additional instructions here that would be skipped if the branch is taken nop #No operation (placeholder instruction) branch_target: #The target instruction that will be executed if t0 is not equal to t1 nop #No operation (placeholder instruction)
BGE	#Assume rs1 is in register t0, rs2 is in register t1, and the branch target label is branch_target #Check if rs1 >= rs2 (rs1 is not the minimum) bge t0, t1, branch_target #Branch if t0 >= t1 #Check if rs1 == rs2 beq t0, t1, branch_target #Branch if t0 == t1 #The next instruction if no branch occurs next_instruction: #Placeholder for subsequent instructions branch_target: #Instructions to execute if branching occurs
BLTU	#Assume rs1 is in register t0, rs2 is in register t1, and the branch target label is branch_target #Check if rs1 is unsigned less than rs2 bltu t0, t1, check_equal j next_instruction check_equal: #Check if rs1 is not equal to rs2 bne t0, t1, branch_target #The next instruction if no branch occurs next_instruction: #Placeholder for subsequent instructions branch_target: #Instructions to execute if branching occurs

BGEU	#Assume rs1 is in register t0, rs2 is in register t1, and the branch target label is branch_target #Check if rs1 is unsigned greater than or equal to rs2 (opposite of UNSIGNED_MIN) bgeu t0, t1, branch_target #If the previous check fails, check if rs1 is equal to rs2 beq t0, t1, branch_target #The next instruction if no branch occurs next_instruction: #Placeholder for subsequent instructions branch_target: #Instructions to execute if branching occurs

Table A8. This Table Shows the Remaining Chat-gpt Generated Test Cases in the Branches Category.

MRET	#Read mstatus into t0 csrr t0, mstatus #Extract MPIE (bit 7) and place it in MIE (bit 3) #Shift right by 4 then left by 4 to clear other bits and position MPIE at MIE's position srli t1, t0, 4 #Shift right logical immediate slli t1, t1, 4 #Shift left logical immediate #Write back to mstatus, updating only the MIE bit csrs mstatus, t1 #Atomic Read/Set bits in CSR, only affecting MIE bit #Jump to the address in mepc csrr t2, mepc #Read mepc into t2 jalr zero, t2, 0 #Jump to the address in t2

URET	#Assume memory starting at address 0x1000 contains saved register values
	#For simplicity, we'll just restore x5 and x6 as an example
	#Restore Context
	#Load the saved values into x5 and x6
	lw x5, 0(0x1000) #Load the value saved at 0x1000 into x5
	lw x6, 4(0x1000) #Load the value saved at 0x1004 into x6
	#Following instructions would continue normal execution using restored context

ECALL	#Assume x10 holds the artificial 'exceptionCall' value: 0 for S-mode, 1 for M-mode #Assume x5 is a general-purpose register, x6 is used for the pc, stvec is at label stvec_handler, mtvec is at label mtvec_handler #Let's say we've loaded this value into x10 beforehand #Check if we're simulating an S-mode exception li x1, 0 #Load 0 into x1 to represent S-mode beq x10, x1, s_mode #Branch to s_mode label if x10 == 0 #Check if we're simulating an M-mode exception li x2, 1 #Load 1 into x2 to represent M-mode beq x10, x2, m_mode #Branch to m_mode label if x10 == 1 #S-mode exception handling simulation s_mode: csrr x6, sepc #Read sepc into x6 la x5, stvec_handler #Load the address of the S-mode handler into x5 jalr x0, x5, 0 #Jump to the S-mode exception handler #Update mstatus, simulated (not how it's actually done) li x3, 0 #Assume 0 represents the desired mstatus value for S-mode csrw mstatus, x3 #Write the value to mstatus (in reality, hardware does this) j end #Jump to the end of the test case #M-mode exception handling simulation m_mode: csrr x6, mepc #Read mepc into x6 la x5, mtvec_handler #Load the address of the M-mode handler into x5 jalr x0, x5, 0 #Jump to the M-mode exception handler #Update mstatus, simulated (not how it's actually done) li x4, 1 #Assume 1 represents the desired mstatus value for M-mode csrw mstatus, x4 #Write the value to mstatus (in reality, hardware does this) #End of the test case end: #Handlers (would normally contain actual exception handling logic) stvec_handler: #Handler code for S-mode would go here ret mtvec_handler: #Handler code for M-mode would go here ret

Table A9. This Table Shows the Remaining Chat-gpt Generated Test Cases in the Privilege Category.

CSRRS	csrr t0, csr # Read the CSR into t0 (rd) #Assuming t1 is rs1 for this example li t1, 0x5 #Load a test value into t1 (rs1) #Conditional CSR Update bnez t1, update_csr #Branch to update_csr if t1 (rs1) is not zero j end #Jump to end if t1 is zero update_csr: csrrs zero, csr, t1 #Atomically read and set bits in CSR using t1 (rs1), and don't store the read value end: #The CSR value could be read again and compared to expected results csrr t2, csr #Read the updated CSR value into t2 for verification
CSRRC	#Setup (Assuming rs1 and csr are already populated with values) #li t0, CSR_ADDRESS #Pseudo instruction to load CSR address into t0 #li rs1, VALUE #Load some value into rs1 #Read CSR csrr t1, csr #Read CSR into t1 (rd) bnez rs1, clear_bits #If rs1 is not zero, go to clear_bits j end #Otherwise, jump to end clear_bits: not t2, rs1 #t2 = ~rs1 and t1, t1, t2 #Clear the bits in t1 where rs1 has bits set csrw csr, t1 #Write the result back to the CSR end: #Continue with the rest of the program

CSRRWI	#Read CSR csrr t0, 0xF14 #Read CSR value into t0 #Modify Value (simulated here by loading an immediate into t1) li t1, 0x123 #Load immediate (the value to be written into the CSR) andi t1, t1, 0xFF #Ensure it's only the low bits (LOW_BITS(imm)) #Write CSR csrw 0xF14, t1 #Write the modified value back to the CSR
CSRRSI	#Read CSR csrr t0, 0xF14 #Read CSR 0xF14 into t0 #Check Immediate li t1, 0x5 #Load immediate value 0x5 into t1 beqz t1, end #If the immediate is zero, skip the update #If imm is non-zero, set the corresponding bits in the CSR csrrs zero, 0xF14, t1 #Atomically read and set bits in CSR 0xF14 end: #Subsequent instructions...
CSRRCI	csrr t0, 0xF14 #Read CSR 0xF14 into t0 #Check Immediate li t1, 0xA #Load immediate value 0xA into t1 (example immediate) beqz t1, end #If the immediate is zero, skip the update #If imm is non-zero, clear the corresponding bits in the CSR csrrc zero, 0xF14, t1 #Atomically read and clear bits in CSR 0xF14 specified by t1 end: #Subsequent instructions...

Table A10. This Table Shows the Remaining Chat-gpt Generated Test Cases in the Csr Category.

FENCE.I	#Let's assume some operations are performed here that modify instructions in memory #Synchronization fence.i #Execute subsequent instructions that would rely on the modifications being visible

SFENCE.VMA	#Assume address to check is in t0 li t0, SOME_ADDRESS #Load some address to check la t1, handle_fault #Load the address of the fault handler #Attempt to read from the address and see if it's successful csrw mtvec, t1 #Set the trap handler lw t2, 0(t0) #Attempt to load from the address li t2, 1 #If execution continues here, memory access was successful j end #Jump to the end handle_fault: li t2, 0 #If a fault occurs, set t2 to 0 end: #t2 now contains the result of the check (1 for valid, 0 for invalid) mv rd, t2 #Move the result to rd

Table A11. This Table Shows the Remaining Chat-gpt Generated Test Cases in the Other Category.