NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

RISC-V PROCESSOR PERFORMANCE ANALYSIS OF SECURE DESIGN PRINCIPLES

by

Roy S. Shin

December 2023

Thesis Advisor:	Chad A. Bollmann
Second Reader:	Douglas J. Fouts

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2023	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE RISC-V PROCESSOR PERFORMANCE ANALYSIS OF SECURE DESIGN PRINCIPLES	**5. FUNDING NUMBERS**
6. AUTHOR(S) Roy S. Shin	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

This project explores processor microarchitecture features that impact security and performance by conceptualizing and describing a RISC-V processor design with security as the priority.

We begin by evaluating causes of several key classes of security vulnerabilities and then considering alternative architectures that address principal causes. We implemented portions of our design in SystemVerilog and demonstrated the functionality and performance of implemented features through simulation. Instantiation efforts are limited to microarchitecture design and writing register-transfer level (RTL) descriptions of the processor; formal verification, synthesis, and fabrication steps are specifically excluded.

Specifically, we implemented a single-core RISC-V processor with a modified Harvard architecture for improved isolation of memory resources between privilege levels. Our implementation also mitigates side-channel attacks by avoiding data-dependent timing and adding power obfuscating features. We found that these changes reduced IPC performance by 55%, due to the increased impact of memory latency while eliminating most security vulnerabilities due to cache timing, branch prediction, and power analysis.

14. SUBJECT TERMS RISC-V, side-channel, security, HDL, microprocessor	**15. NUMBER OF PAGES** 65
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

RISC-V PROCESSOR PERFORMANCE ANALYSIS OF SECURE DESIGN PRINCIPLES

Roy S. Shin
Captain, United States Marine Corps
BS, Carnegie Mellon University, 2016

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2023

Approved by: Chad A. Bollmann
Advisor

Douglas J. Fouts
Second Reader

Preetha Thulasiraman
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This project explores processor microarchitecture features that impact security and performance by conceptualizing and describing a RISC-V processor design with security as the priority.

We begin by evaluating causes of several key classes of security vulnerabilities and then considering alternative architectures that address principal causes. We implemented portions of our design in SystemVerilog and demonstrated the functionality and performance of implemented features through simulation. Instantiation efforts are limited to microarchitecture design and writing register-transfer level (RTL) descriptions of the processor; formal verification, synthesis, and fabrication steps are specifically excluded.

Specifically, we implemented a single-core RISC-V processor with a modified Harvard architecture for improved isolation of memory resources between privilege levels. Our implementation also mitigates side-channel attacks by avoiding data-dependent timing and adding power obfuscating features. We found that these changes reduced IPC performance by 55%, due to the increased impact of memory latency while eliminating most security vulnerabilities due to cache timing, branch prediction, and power analysis.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Research Questions 1

1.3 Scope 1

1.4 Thesis Organization 2

2 Background **3**

2.1 Overview 3

2.2 Major Security Vulnerabilities 3

2.3 Prior Work 10

2.4 Summary 15

3 Methodology **17**

3.1 Overview 17

3.2 Secure Design Principles 17

3.3 Instruction Set Architecture 19

3.4 Simulation and Testing Environment 19

3.5 Summary 21

4 Results **23**

4.1 Feature Omissions 23

4.2 Design Features 23

4.3 Simulation Results 31

5 Conclusion **33**

5.1 Assessment of Design and Goals 33

5.2 Future Work 33

Appendix A RTL Code Repository **37**

Appendix B Compliance Test Repositories **39**

Appendix C Benchmark Code Repositories **41**

List of References **43**

Initial Distribution List **47**

List of Figures

Figure 2.1 Spectre Attack Diagram 7

Figure 2.2 Meltdown Attack Diagram 9

Figure 2.3 Side-Channel Protection Classification 12

Figure 2.4 Spectre and Meltdown Classification 14

Figure 4.1 Processor Overview 25

Figure 4.2 Two-bit Counter FSM 27

Figure 4.3 Encryption Overview 30

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Table 2.1 RISC-V Privilege Levels 10

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AES Advanced Encryption Standard

ALU arithmetic logic unit

BTB branch target buffer

CHERI Capability Hardware Enhanced RISC Instructions

CSR control and status register

DPA differential power analysis

ECC error correction code

FIFO first in, first out

HDL hardware description language

HVL hardware verification language

IPC instructions per clock

ISA instruction set architecture

IRAM instruction random access memory

IROM instruction read only memory

JALR jump and link register

LFSR linear feedback shift register

NIST National Institute of Standards and Technology

ORAM open random access memory

PC program counter

PHT pattern history table

PMP physical memory protection

PRAM privileged random access memory

PRNG pseudorandom number generator

RAW read after write

RISC reduced instruction set computer

RTL register-transfer level

ROB reorder buffer

RSA Rivest–Shamir–Adleman

SGX Software Guard Extensions

SRAM static random access memory

TRNG true random number generator

UOP micro-operation

Acknowledgments

I wish to thank my advisor, Captain Chad Bollmann, for facilitating this research. It was a great opportunity, and I learned so much from this. I appreciated your candor and outlook on both life and academia during our work together.

Also, I want to give special thanks to G. Glenn Henry for his time and mentorship over the year. Much of this research would not have been possible without his knowledge, expertise, and guidance. Our many meetings and discussions have imparted much greater awareness and appreciation for the complexities of computer design that I would not have gotten through classes alone.

A special mention to Krystof C. Zmudzinksi for his assistance in the testing and program compilation portion of this research. His help was instrumental in overcoming my inexperience with program compilation and linker files for bare metal execution.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

1.1 Motivation

Processor design has historically prioritized performance improvement. Unfortunately, many features that improve performance have also led to numerous security vulnerabilities; vulnerabilities that are difficult or even impossible to mitigate after a design is produced and installed in a system. Many modern military systems are highly dependent on these processors, which renders our systems vulnerable to cyberattacks. As such, effective military cyber defense needs to start with the foundation of secure hardware design instead of relying on software patches and post-production mitigations.

1.2 Research Questions

This research aims to answer how a processor should be designed if security is the driving priority. To accomplish this, we must understand the mechanisms of significant processor features as well as the vulnerabilities that they cause. We must also consider proposed solutions and examine their effectiveness against their related vulnerabilities.

1.3 Scope

The scope of this research is limited to the microarchitecture design of a central processing unit. Post-design process steps of design verification, synthesis, and fabrication are outside the scope of this research due to the time and resource constraints at the Naval Postgraduate School. This research makes a few assumptions in the hardware timing of our design in order to simulate its functionality for compliance testing and benchmarking. These assumptions are detailed in Chapter 4. Future work would involve actual timing analysis and refinement based on a synthesized gate level design.

1.4 Thesis Organization

Chapter 2 details significant processor design features that are particularly vulnerable as well as proposed solutions. Chapter 3 describes the design principles that we used to design our processor as well as the design and testing methodology used. Chapter 4 details our design features in-depth as well as analysis of benchmark results. Chapter 5 is the concluding chapter. Appendix A contains the link to the GitHub repository of the author containing the full RTL description for the processor. Appendix B contains the links to GitHub repositories containing the compliance testing files. Appendix C contains the links to GitHub repositories containing the benchmark files.

CHAPTER 2:
Background

2.1 Overview

The purpose of this research was to explore processor design choices that are inherently simple, yet effective, against major vulnerabilities and malicious attacks. Generally, processor designs have prioritized performance and function, which have created numerous security vulnerabilities that have been discovered over time. We aim to prioritize security above all else, so we need to establish an understanding of the causes and mechanisms of security vulnerabilities in order to effectively mitigate them.

2.2 Major Security Vulnerabilities

We briefly review major processor security vulnerabilities that were considered in our design process. While this is not an exhaustive list of all hardware vulnerabilities, we believe that these issues are the most important vulnerabilities to consider at this stage of microarchitecture design.

2.2.1 Main Memory Organization

Protecting computer memory is a critical aspect of security. The organization and management of instructions and data by the processor in memory has a significant impact on the ability of the system to handle confidentiality and integrity. The processor needs to prevent unauthorized access to sensitive memory contents as well as maintain integrity of stored information. Two common descriptions for memory organization are the von Neumann architecture and the Harvard architecture.

A computer using the von Neumann architecture stores instructions and data in the same main memory with a shared memory address space; the contents of any given address can be interpreted and used as an instruction or data. This provides a lot of flexibility with how memory is allocated for program instructions versus data structures. Also, programs are capable of writing new instructions during run-time through data store operations.

In contrast, a computer using the Harvard memory architecture separates instructions and data in different memory spaces. This is inherently less flexible than a von Neumann architecture, but provides a natural partition between instructions and data. This can prevent programs from accidentally or intentionally accessing and modifying instructions through data operations.

A simple type of attack on systems using shared memory resources without adequate protection is a buffer overflow attack. This type of attack exploits how program instructions may be stored adjacent to data structures in memory. By attempting to access beyond the bounds allocated for a data structure, it is possible to read or write into memory addresses holding instructions if there are not adequate control mechanisms to prevent these actions. Furthermore, an attacker could hijack the processor by rewriting their own instructions into this space and forcing the processor to execute them. Von Neumann architectures are generally more vulnerable to these types of attack compared to Harvard architectures, due to the how instructions and data are stored in a shared memory resource.

2.2.2 Differential Power Analysis

A significant amount of information can be indirectly determined from hardware devices through side-channel attacks. One such attack is differential power analysis (DPA), which is accomplished by monitoring and deducing information by observing the power consumption or other physical emanations of a device as it operates. Different operations within the processor will consume varying amounts of power, such as writing to a register or performing arithmetic. By observing and correlating the subtle patterns in power consumption while the processor is executing a particular program, an attacker can learn information about the program or the data being processed. DPA has been proven to be effective in recovering the encryption key of cryptographic algorithms, as patterns created by the encryption operations are quite distinct and easily measurable [1].

2.2.3 Cache Memory

Cache memory improves memory access times where processor speeds can significantly outpace the latency of cheap, high-capacity devices used for main memory. By storing select portions of memory in a highly responsive cache memory device, a processor is able to quickly access the contents of frequently-used memory locations and adjacent addresses. A

cache miss occurs when the processor attempts to access a memory address that is not in cache, and the processor must wait for the relatively long latency of main memory to load the appropriate data into cache.

This time difference between a cache hit or a miss is easily measurable, and attackers can exploit this difference in timing as a side-channel. A attack called FLUSH+RELOAD has demonstrated how cache architectures used in many modern processors can be exploited as a side-channel [2]. According to the paper, the attack is set up by intentionally flushing a specific portion, or line, of memory from the cache in order to monitor the specific line. A victim program that shares a cache with the attacker might end up refilling this monitored line. The attacker can re-attempt to access the monitored cache line through a reload and measure how quickly the processor can return the data. Depending on the measured time, the attacker can determine whether the victim program accessed the monitored cache line or not. In [2], Yarom proves how this information could be used to extract secret data, such as encryption keys used in a Rivest–Shamir–Adleman (RSA) cryptographic algorithm.

Cache timing is also used frequently as a side-channel to enable other attacks, including those discussed in the next sections.

2.2.4 Branch Prediction

Branch instructions can be a significant performance hindrance for pipelined processors. Normally, the program counter (PC) increments sequentially to fetch instructions that are adjacent to one another in memory. A branch instruction can change the PC to a nonsequential address, which requires the pipeline to be flushed of all previously fetched instructions which have become invalid due to the direction of the branch. To prevent a pipeline flush, modern processors will often attempt to predict where the PC may branch to and preemptively fetch from the predicted instruction address. Most predictor designs will store information such as previous branch directions in a pattern history table (PHT) and previously calculated branch target addresses in a branch target buffer (BTB). When fetching an instruction from memory, the processor will index into the PHT and BTB in order to predict whether the fetched instruction is a branch instruction as well as the target address. Neither of these tables are typically large enough to hold unique values for each PC value; this results in aliasing where more than one instruction will index into the same PHT and

BTB values. This aliasing behavior is critical to how PHT and BTB states are manipulated because an attacker can access and alter PHT and BTB values used by a victim program without directly modifying the victim program instructions.

In [3], Evtyushkin et al. demonstrates how branch prediction can be exploited in a side-channel attack called BranchScope. According to the paper, the attack begins executing a program to prime the PHT into a known state. Once the PHT is in this known state, a victim program is executed on the same processor. Upon completion of the victim program, the attacker can determine changes to the PHT from the known state by probing each entry and measuring the response times. Despite not being able to directly observe the execution of the victim program, an attacker can potentially infer the behavior of the victim program as well as determine memory contents that influence branches. This attack has been proven to be effective against computations used for RSA [3].

2.2.5 Speculative Execution

Speculative execution is a design feature intended to address potential performance loss from branch instructions. While branch prediction anticipates which instructions are to follow a branch instruction, speculative execution issues and executes the instructions before the actual branch result has been fully resolved. While the actual results of transient instruction can be discarded, remnants of their execution can still be found in other parts of the processor.

In [4], Kocher et al. demonstrates how to exploit speculative execution in an attack known as Spectre. According to the paper, the Spectre attack begins by priming the branch predictor to deliberately cause a misprediction for a future branch instruction. The second stage of the attack will execute a program where a branch misprediction will allow speculative instructions to be executed but ultimately flushed from the pipeline once the branch result is resolved. These speculatively-executed instructions will attempt to load a memory address based on the contents of some secret data location. Normally, memory access controls would raise an exception against this illegal access and handle the exception once it reaches retirement. However, because the memory load is only speculative and will never actually retire due to the branch misprediction, the illegal memory access exception is ignored as it was never supposed to happen in the first place.

Meanwhile, the effects of the speculatively executed instructions remain in the system in the

form of a cache line update in anticipation of the memory load. An attacker can determine the memory address for the attempted load instruction by probing the cache state and measuring cache timings to determine a hit or miss. By determining the attempted address, the attacker can determine the value of the secret data based on how it related to the load address. This attack sequence is illustrated in Figure 2.1. Through repetition of this process, the attacker can potentially reveal the entire memory state of the processor [4].

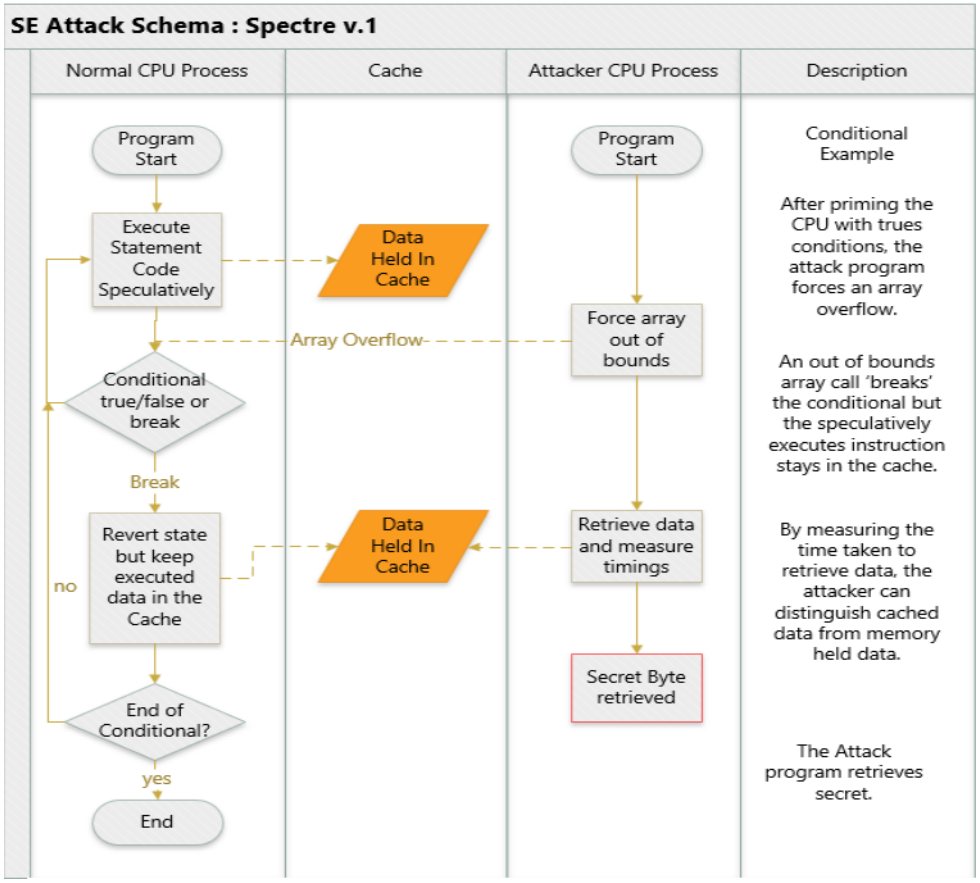

Figure 2.1. Spectre Attack Diagram. Source: [5].

2.2.6 Out-of-Order Execution

Out-of-order execution is another critical design feature meant to optimize the processor pipeline usage of available resources. For maximum efficiency in the pipeline, instructions may be issued and executed in a different order than they were fetched and decoded to fill the gaps created by stalls due to data dependencies between pipeline stages. These out-of-order instructions will ultimately retire through a construct known as the reorder buffer (ROB) that ensures that executed instructions are committed in the order that the program dictates.

Similar to speculative execution, out-of-order execution can be exploited to reveal secret data without proper privileges. In [6], Lipp et al. demonstrate how out-of-order execution can be exploited in another attack known as Meltdown. According to the paper, the Meltdown attack functions similarly to Spectre by creating conditions where the processor executes transient instructions that will leave behind information in a side-channel. However, Meltdown exploits out-of-order execution to create a race condition where transient instructions are executed but flushed out of the ROB. Figure 2.2 illustrates how a basic Meltdown attack functions. Meltdown has been proven to bypass memory protections and reveal the memory state of the processor of all privilege levels [6].

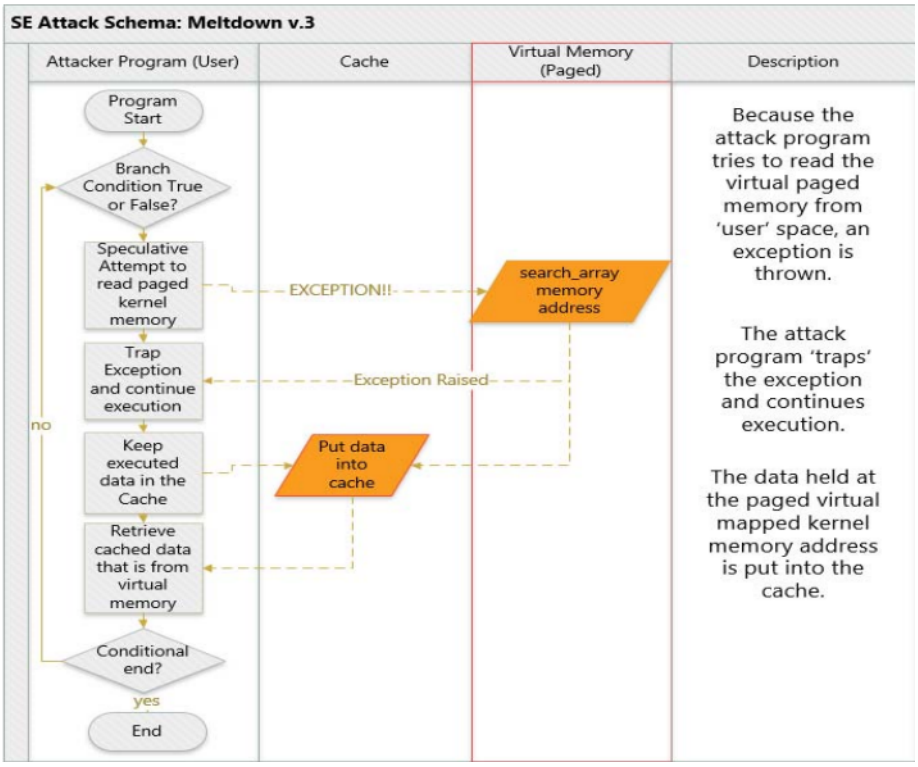

Figure 2.2. Meltdown Attack Diagram. Source: [5].

2.2.7 Quantum Computing and Cryptographic Algorithms

Recent advances in quantum computing have created a growing concern over the security of existing cryptographic algorithms. In 2016, a National Institute of Standards and Technology (NIST) report evaluated the vulnerability of widely used ciphers to large-scale quantum computing and found that a number of ciphers such as RSA would no longer be considered secure [7]. Many of these ciphers have been essential to data and communication confidentiality, and quantum-computing threatens to dissolve many trust and security protocols in cyberspace. Finding effective quantum-resistant algorithms and implementations will only become more important over time, as cryptography is a necessity for modern cybersecurity.

2.3 Prior Work

We must also review some mitigation strategies for the discussed hardware vulnerabilities that were considered in our design. While the effects of these mitigation measures are discussed in this chapter, their impact on our design choices is detailed in Chapter 4.

2.3.1 Memory Protection

Memory protection is critical for maintaining data integrity and confidentiality. Unauthorized programs should not be able to access or modify sensitive memory locations, especially for designs with shared memory resources. A common approach uses privilege levels and memory tagging to control access to shared memory resources.

Privilege levels are used to restrict access to special functions and resources based on the particular program being executed on the processor. Table 2.1 lists the various privilege levels for a RISC-V processor. The most trusted programs are executed in machine mode, which grants the greatest access to processor functions and resources. Programs executed in lower modes, such as user mode, have more restrictions based on designer specifications.

Table 2.1. RISC-V Privilege Levels. Source: [8].

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

A tagged memory architecture uses tag bits as metadata for memory addresses. In a simple memory protection implementation, these tags can correspond to a privilege level requirement for accessing particular memory regions. A program executing in user mode would not be allowed to access a memory region that is tied to a memory tag requiring supervisor

or machine mode privileges. A processor can dynamically change access requirements for memory regions by changing these tag bits.

Capability Hardware Enhanced RISC Instructions (CHERI) is a modern example of memory protection using memory tagging and metadata to control memory access [9]. With CHERI, all memory operations (such as data load/store or instruction fetch) must be authorized by a construct called a capability. A capability consists of a memory address (to be used as a pointer to a program instruction or data) and metadata bits that control the privileges and access permissions specific to that address. This scheme allows greater control over the permissions and memory boundaries granted to a given program.

Many modern processor designs use privilege levels and memory tagging to protect sensitive instructions and data that are stored in a shared memory resource. However, Meltdown demonstrated how standard memory protection implementations using privilege levels and memory tagging are not fully effective against speculative execution attacks [6]. Even CHERI has been found to be vulnerable to speculative execution attacks like Spectre [10].

Another approach to memory protection is Intel Software Guard Extensions (SGX), where protected portions of memory are dynamically encrypted to further protect the contents from unauthorized programs [11]. While SGX was initially thought to be resistant against speculative execution attacks such as Spectre and Meltdown, a new attack called Foreshadow was proven to be able to bypass the countermeasure [12]. Foreshadow takes advantage of the fact that encrypted secrets that have been recently accessed by the processor are stored in plaintext form in the cache.

2.3.2 Side-Channel Protection

According to [13], protections against side-channel attacks such as cache timing analysis or DPA can be categorized as either a hardware or software approach, depicted in Figure 2.3.

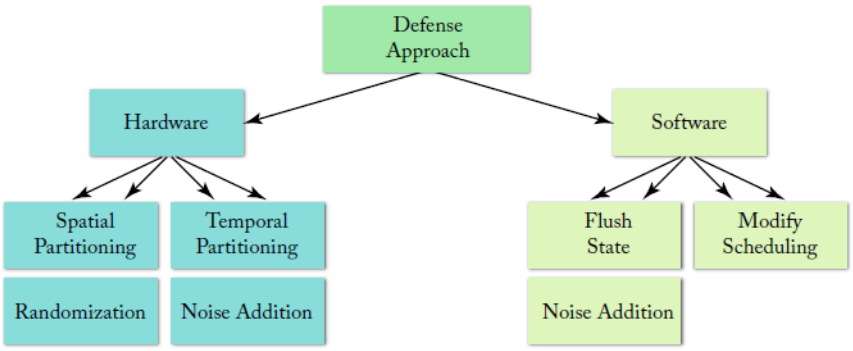

Figure 2.3. Defense Approaches to Side-Channel Protection. Source: [13].

Countermeasures for DPA and other physical emission analysis attacks primarily involve adding noise to make observations more difficult to analyze as well as reducing the amount of information that could be leaked by sensitive operations such as encryption [14]. Shielding a processor to physically prevent information leaks is beyond the scope of this research (and most practical implementations), so we primarily considered methods for introducing noise in the power consumption of the system.

Countermeasures for cache timing attacks primarily involve randomization and/or partitioning to prevent attackers from accessing information that could be stored in the cache state [13]. We considered a proposed cache design using partitioning as the security feature, known as Partition-Locked cache [15]. In [15], Wang and Lee proposed and evaluated the Partition-Locked cache design that uses an extra lock bit that can indicate whether a particular cache line is not to be evicted. This implementation is intended to prevent an attacker from manipulating the cache state when sensitive data is locked within the cache. However, further testing and evaluation have discovered that Partition-Locked cache can still be vulnerable to other types of side-channel attack methods [16].

We also considered a different design using randomization as a security measure, known as the Random-Fill Cache [17]. In [17], Liu and Lee proposed and evaluated the Random-Fill cache design that does not always fill the cache with the requested address on a cache miss; instead, the fetched cache line is randomly selected from a range of addresses based on

the originally requested address. This implementation is intended to make it difficult for an attacker to determine a particular address was accessed by the victim program. However, since the randomly fetched cache line is still related to the requested address, Random-Fill cache was also demonstrated to be vulnerable to certain side-channel attack methods [16].

2.3.3 Spectre and Meltdown Mitigation

Due to the widespread vulnerability of modern processors to Spectre and Meltdown, there has been significant amount of work to mitigate and patch against these attacks without sacrificing significant amounts of performance. Simultaneously, many Spectre and Meltdown variants have also been discovered, making it difficult for a single mitigation to fully protect against every variation [18]. A recent survey of major variants are shown in Figure 2.4.

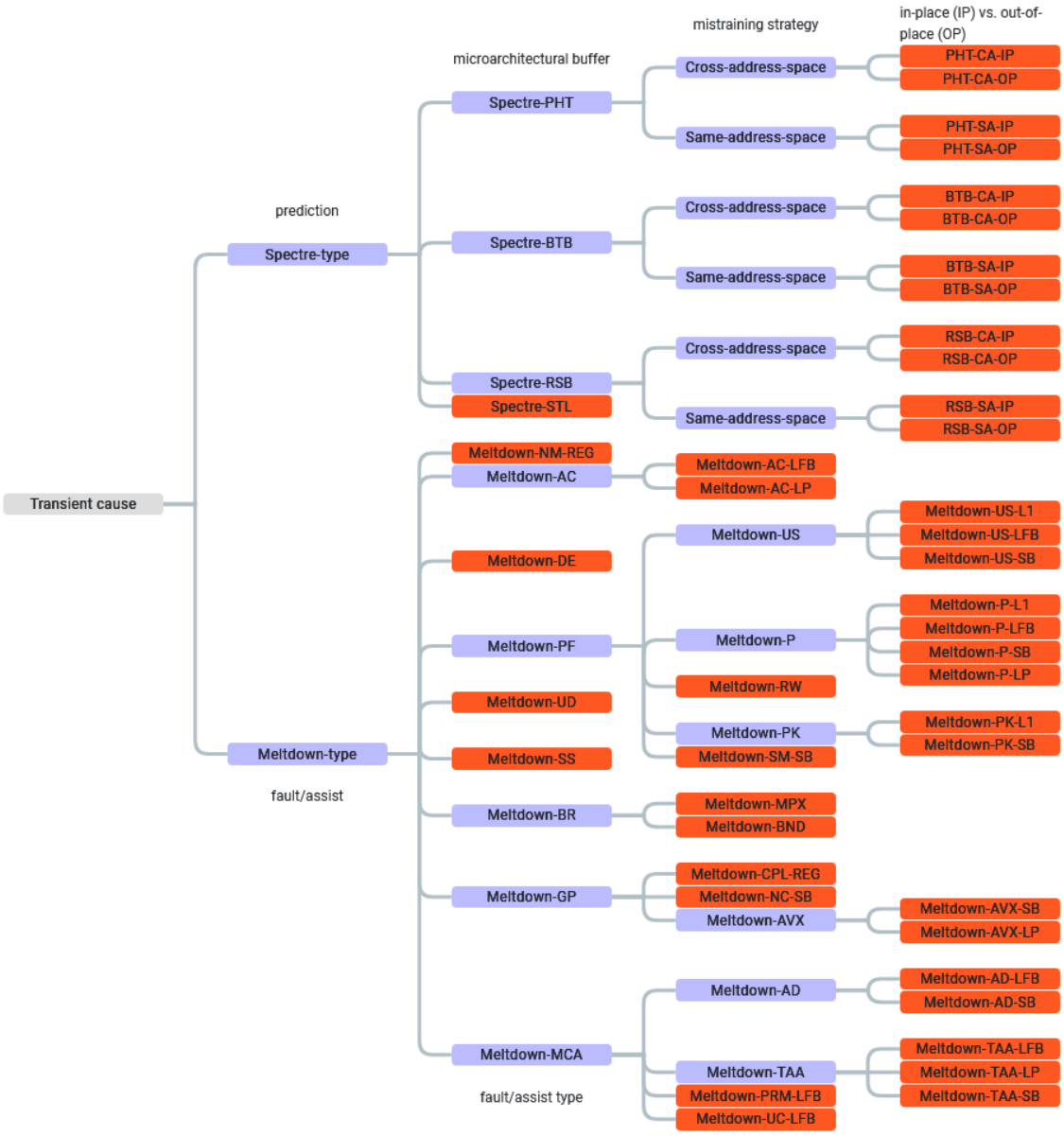

Figure 2.4. Spectre and Meltdown Classification Tree. Source: [19].

In [4], several suggested countermeasures to Spectre are weighed against the increased hardware requirements and performance impacts. The simplest countermeasure considered is to disable speculative execution, as the attack relies on the execution of transient instructions.

Given the performance impact of disabling speculative execution, this solution is generally avoided by most hardware designers. Canella et al. found that most Spectre defenses only address the effectiveness of a specific covert channel rather than addressing the root cause of the vulnerability [18]. These defenses are often found to be ineffective against different Spectre variants.

Similarly, in [6], the most effective countermeasure against Meltdown would be to disable out-of-order execution. This is also difficult for most modern hardware designs to accept given the performance loss. An interesting solution proposed by Lipp et al. was to implement hardware-level separation of memory spaces between privilege levels, where privilege requirements for accessing a given memory location could be quickly determined based on the address alone [6].

2.3.4 Post-quantum Cryptography

Many public key ciphers used for cybersecurity were assessed to be vulnerable against large-scale quantum computing, while some symmetric key ciphers such as Advanced Encryption Standard (AES) were determined to remain secure as long as sufficiently larger key sizes are used [7]. In anticipation for a post-quantum world, more robust ciphers have since been developed and proposed to become standards as quantum-resistant algorithms, such as Crystals-Kyber and Crystals-Dilithium [20]. Many of these quantum-resistant algorithms are too complex for the scope of this research, so our attention is primarily be focused on easier, yet robust ciphers such as AES.

2.4 Summary

This chapter discussed various processor design features, security vulnerabilities that exploit them, and prior work studying how future designs could mitigate or prevent these attacks. These works have informed our decision-making and selection of design features to maximize security. In the next chapter, we will discuss the design philosophies that will guide our decision-making and selection of design features, as well as the testing environment used for our research.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Methodology

3.1 Overview

The purpose of this research was to explore processor design choices that can effectively mitigate or eliminate major vulnerabilities and malicious attacks. In the previous chapter, we highlighted several major processor design features and the associated security vulnerabilities they created. In this chapter, we describe specific secure design principles that will guide our design as well as explain how our design was written, simulated, and tested.

3.2 Secure Design Principles

We used the following five principles to guide our design choices. We believe that these principles are foundational to our security philosophy because they address the root cause of hardware vulnerabilities.

3.2.1 Reduce Bugs through Simplicity

Hardware bugs pose a significant security risk, as unintended behaviors can be exploited to undermine the functionality and reliability of our system. Simple designs are easier to test and verify to eliminate bugs, therefore secure designs must be as simple as possible.

However, a simple design is not enough to guarantee security. In order to prove that our design functions as intended, formal verification is required. This is a lengthy process using formal mathematical methods to prove and guarantee the functionality of our design. Unfortunately, formal verification is beyond the scope of this research due to time and resource constraints. Nonetheless, our design must be as simple as possible in anticipation of future work involving formal verification.

3.2.2 Be Unpredictable to Observation

Predictable behavior is important to ensuring that results of the processor are consistent, but predictability can also allow side-channels to reliably extract information through anal-

ysis. We need to simultaneously ensure that our processor produces consistent results in execution while being unpredictable under observation. Out of the four major side-channel defense categories, we chose to adopt noise addition as our primary defense mechanism. By obfuscating the power consumption related to the processor execution with noise, it should be difficult to reliably correlate discernible features and infer information from the processor.

3.2.3 Avoid Hidden Secrets in Design

Achieving security through obscurity is rarely an effective strategy. Obscure secrets can be discovered by a determined attacker through hardware reverse engineering. While there are countermeasures to hardware reverse engineering, these mitigations are outside the scope of this project. Therefore, we must ensure that the security of our design cannot be compromised simply by an attacker understanding the system.

3.2.4 Isolate Resources Between Privileges

Many of the discussed attacks are able to bypass privilege mode protections and access sensitive memory regions because resources are not well isolated between privilege levels. Shared resources, such as general purpose registers and memory regions, are commonly used by programs running at different privilege modes, and memory protection features cannot always detect and prevent unauthorized access attempts before secrets are revealed. One previously discussed method for resource isolation was using a Harvard memory architecture to separate instruction memory from data memory. This is an important design feature that will be examined in detail in Chapter 4.

3.2.5 Fast Encryption Capability

Cryptography is a necessity for secure systems, because encryption is used for data authentication, integrity, and confidentiality. A secure processor will likely need to interact with external systems, so a fast encryption process is needed to keep up with external communication in order to validate the security of data entering and exiting the processor. At the same time, this encryption must also be unpredictable to observation as well as isolated from unauthorized access.

3.3 Instruction Set Architecture

In order to simulate and test the design of our processor, we need to select an instruction set architecture (ISA) that determines how the processor should function. We chose to use RISC-V, an open-source reduced instruction set computer (RISC) ISA that is gaining popularity with many hardware designers. Another benefit of RISC-V is its load-store architecture; only load and store instructions can access memory while other instructions can only access registers. This simplicity is advantageous for security because memory operations are easier to control. The basic functions are outlined in the unprivileged specification [21], while details regarding privilege management and exception handling are found in the privileged specification [8].

We implemented the base RV32I module, which includes arithmetic, logical, memory, and branch control instructions. The Ziscr extension was also implemented for control and status register (CSR) operations required for exception handling and feature control.

3.4 Simulation and Testing Environment

Our processor was written as a register-transfer level (RTL) design using a hardware description language (HDL) known as SystemVerilog. The Xilinx Vivado 2022.1 software suite was used to simulate and test the design.

3.4.1 SystemVerilog

The RTL design, found in Appendix A, outlines the structure and function of the processor components. SystemVerilog was originally an extension to another HDL known as Verilog, providing greater functionality for hardware design, modeling, and verification. The use of SystemVerilog as a hardware verification language (HVL) is especially useful for future work in verification of our design.

However, certain SystemVerilog features (such as delay statements and looping statements) cannot be synthesized into gate-level designs using software tools because they model behaviors that cannot translate directly to hardware logic. However, many of these nonsynthesizable features are useful for simulation, testing, and verification, and they are extensively used in our testbench design. In order to ensure that our design was synthesizable we restricted the scope of SystemVerilog functions and features used in our RTL design. While

a standard for synthesizable SystemVerilog does not exist, [22] was a useful reference for best practices for RTL design.

Our design was primarily written to model combinational logic using continuous assignment, logic variables, and simple operators (such as bitwise OR, AND, and XOR functions). Modeling behavioral logic using procedural assignment functions (such as always_ff or always_comb) is generally avoided because synthesis tools may need to infer the intent and functionality of the design; this can result in unexpected post-design elements that were not explicitly declared in the RTL design. The most common issue created from the use of procedural assignment features include unintended latches and flip-flops being added to the design during synthesis. We used a common library of pre-defined modules to model registers and multiplexers to model sequential logic in our processor components. The use of procedural assignment for these specific components are not an issue with synthesis tools because they are standard designs that do not require inference from synthesis tools.

The specific details of the processor design choices are discussed further in Chapter 4.

3.4.2 Program Compilation and Testing

We used the RISC-V GNU Compiler Toolchain [23] to compile programs into RISC-V instructions. A RISC-V simulator called Spike [24] to generate logs of expected outputs for executed programs. These logs were used with a testbench in Vivado in order to compare against the simulated execution of our processor design. As part of our testing process, we used compliance tests developed by the RISC-V Architectural Framework [25] to partially verify that instructions function as intended on our design. We can only claim partial verification because the compliance tests are not a replacement for formal verification.

All programs simulated through Spike and tested on our design were executed on bare metal, which means that program instructions are executed directly on the processor without the aid of an operating system or kernel. This allows the processor to be more efficient and secure, and the security of the processor is not be compromised by software vulnerabilities found in operating systems.

3.4.3 Linker Control Script

In order to execute programs in bare metal on our processor, we had to ensure that programs are compiled properly to execute on our processor. When a program written in C is compiled into an executable binary code format, the last major step involved is called linking. One of the important functions of the linking process is to organize the compiled and assembled machine code in memory. In order to ensure that program instructions and program data are organized properly for the memory architecture of our design, we wrote a linker control script that describes how assembled objects such as instructions or data should be organized in memory. The linker control script can be found in Appendix B.

3.5 Summary

In this chapter, we outlined the concepts and ideas that we aim to implement in our processor design. We also described how our design was realized in RTL for simulation and testing. In the following chapter, specific design features are discussed along with results collected from benchmark simulation.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Results

4.1 Feature Omissions

Certain features were omitted from our design due to the severity of their proven vulnerabilities as well as the complexity of imperfect mitigation measures. While these features provide significant performance benefits, their inclusion would seriously compromise the security of our design.

One of the most important features we chose to omit is cache memory. Many of the security vulnerability examples discussed in Chapter 2 rely on cache memory as a measurable and reliable side-channel to retrieve information from the processor. By not using a cache, we can completely negate the potential of cache-timing attacks. While we considered a few secure cache design options, we currently do not believe that a particular cache design has been fully tested and formally verified to be completely secure. In order to prevent the possibility of including a potentially flawed feature, we chose not to include a cache in our processor design. However, not using cache memory should have a significant impact on our processor performance, and this is evident in our benchmark results.

We also omitted speculative execution and out-of-order execution due to the significant vulnerability of executing transient instructions. In addition, they would add to the overall complexity of our design. We expected that omission of these features should also have an impact on our processor performance, but the effect of their omission in our benchmark results is not as clear.

4.2 Design Features

Here we detail the specific features implemented in our microarchitecture design. All RTL files are included in Appendix A.

4.2.1 Memory Organization

In the previous chapter, we discussed the vulnerability of the von Neumann memory architecture. We also considered a few memory protection features such as memory tags and privilege modes. Due to the various residual vulnerabilities presented by many of these solutions, we believe that we can achieve greater memory protection by permanently compartmentalizing different memory resources and outright preventing shared resources between privilege levels.

For these reasons, we chose to implement a modified Harvard memory architecture for our design. Instructions and data are stored in separately-defined memory regions and accessed through separate data pathways. This architecture prevents instructions from being corrupted by data structure attacks or self-modifying code. This design also has a subtle performance benefit because instructions and data can be simultaneously accessed by the processor without conflict.

Instruction memory consists of instruction read only memory (IROM) and instruction random access memory (IRAM). IROM is for static programs that are immutable to the processor while IRAM can have programs can have loaded onto it by IROM. Because IROM instructions cannot be changed, they are considered the most trustworthy programs. As the designers, we can control its contents and verify its security before production. Due to this high level of trust, IROM instructions have the most privileged access to processor features, such as memory locations and sensitive registers. The boot code is stored here as well as other special programs such as an exception handler or feature control functions. Branching into random addresses in IROM from IRAM is prohibited and raises an exception in order to prevent unauthorized access to privileged instructions. However, specific starting addresses for special functions can be branched into in order to allow the processor to request privileged functions to be executed. IROM also has the exclusive privilege of accessing and writing data into IRAM in order to load new programs onto the processor. Instructions executing from IRAM inherently have less privilege in order to prevent potentially malicious code from accessing sensitive data.

Data memory is composed of open random access memory (ORAM) and privileged random access memory (PRAM). ORAM is the main memory space that can be used by any program being executed on the processor. PRAM is a data memory region that is exclusive to IROM

instructions, providing a secure memory space for data that should not be accessed by IRAM instructions. This prevents sensitive data used by programs with elevated privileges from being leaked through attacks that can bypass memory protection features.

All memory components were simulated using a Vivado Block Memory Generator, simulating a three-clock-cycle latency for load operations. We simulated memory in this fashion in order to mimic how memory components are generally slower than the base clock speed of the processor. This latency could be further reduced in post-design synthesis by implementing and synthesizing memory components using high-speed hardware like static random access memory (SRAM) that is capable of keeping up with the processor clock speed.

4.2.2 Pipeline Organization

Our design is organized as a five stage, single pipeline processor, depicted in Figure 4.1. The five stages are fetch, decode, register file, execute, and writeback. An instruction is converted into a micro-operation (UOP) at the decode stage to control the later stages in the pipeline. A UOP is a set of signals that control the behavior and output of the register file, execute, and retirement stages.

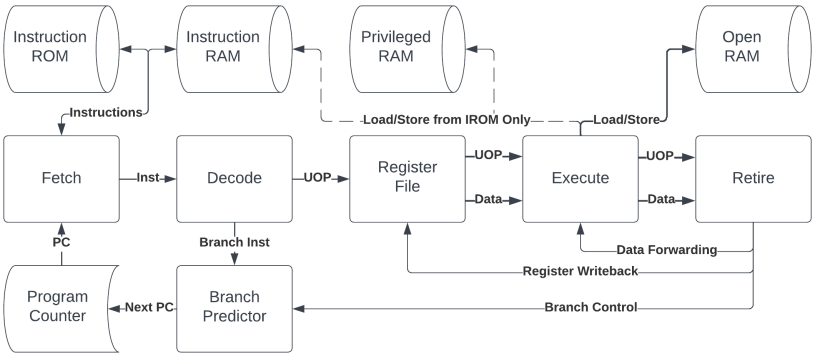

Figure 4.1. Block Diagram Overview of Processor Stages and Features

A 16-element first in, first out (FIFO) queue was also included at the end of the decode stage; the queue provides a buffer so that fetch and decode stages can continuously fetch

instructions from instruction memory and decode them while later stages of the pipeline are stalled from pipeline hazards. The execution stage handles arithmetic logic unit (ALU) operations and memory operations, while the retirement stage handles data writes to the register file and branch control.

Due to our single pipeline design without out-of-order execution, the most likely data hazard requiring the pipeline to stall is a read after write (RAW), where an instruction in the execute stage requires a register value that is actively being handled by the writeback stage. We eliminated the need to stall for this hazard through data forwarding, enabling the register data to be simultaneously written to the register file and input to the execute stage. Due to the latency of our simulated memory components, memory load operations require a three-clock-cycle stall before the data is retrieved from memory. During this latency period, the register read and execute stages stall while the fetch and decode stages continue to cycle and store new instructions into the FIFO queue.

4.2.3 Branch Prediction

While our processor does not feature speculative execution, we can still benefit from pre-emptively fetching and decoding instructions in anticipation of a predicted branch.

Our design is different from most others because our system does not attempt to predict on instructions at the fetch stage before the instruction is decoded. Instead, our branch predictor predicts at the decode stage for instructions that are decoded and verified as a conditional branch. Due to this difference, an attacker cannot force the processor to incorrectly predict a fetched instruction as a branch.

Our branch predictor uses a PHT containing 4096 two-bit saturating counters. A decoded branch instruction indexes to a particular PHT counter using its memory address. The state of the corresponding saturating counter determines the predicted direction of a decoded branch, depicted in Figure 4.2.

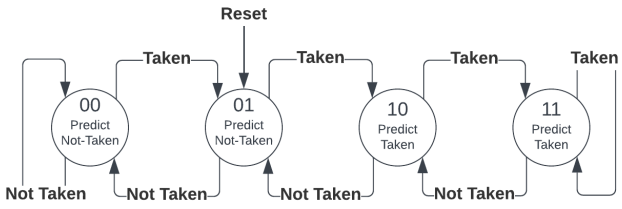

Figure 4.2. Finite State Machine Depiction of a Two-Bit Saturating Counter

When a conditional branch retires from the pipeline, its corresponding counter is incremented if the branch is taken or decremented when the branch is not taken.

Furthermore, our design is different than most because branch targets are calculated rather than predicted. This is possible due to how target address values are typically encoded in the branch instruction for RISC-V. An attacker cannot alter the predicted branch address without changing the instruction itself. Due to this ISA feature, we do not need to use a BTB.

However, RISC-V has one indirect branch instruction where the target address cannot be accurately calculated at the decode stage, which is the jump and link register (JALR) instruction. The target address for a JALR instruction depends on a register value, which may change by the time the branch instruction reaches the execute stage. Instead of using a BTB to predict the branch address, our design never predicts for JALR instructions. This choice prevents indirect branch poisoning, which is used in many Spectre attack variants [18].

As previously discussed, the small size of the PHT compared to the instruction address space can cause aliasing, where multiple different branch instructions map to the same PHT counter. This aliasing makes our design still vulnerable to side-channel attacks such as BranchScope [3]. For improved security in situations where attacks like BranchScope are of concern, our branch predictor can be disabled through a custom CSR. This CSR is a special register whose sole purpose is to control the branch predictor. When the processor is booted or reset, the branch predictor is automatically enabled. A CSR write instruction

from either IROM or IRAM may disable the branch predictor and force the decode stage to always assume that a branch is never taken. Re-enabling the branch predictor requires privileged access from IROM.

4.2.4 Data Independent Instruction Timing

To defend against side-channel vulnerabilities such as timing attacks, we need to design our processor to prevent leaking information about the data being executed on through patterns in its timing. To prevent this, we ensured that instruction timing does not vary based on the data being executed. All ALU instructions are assumed to take one-clock-cycle regardless of the input data. Similarly, load instructions have consistent latency regardless of the address being accessed because we are not using cache memory. This prevents attackers from deducing information about the data being executed on the processor through observation of subtle timing differences and patterns.

4.2.5 Power and Emission Obfuscation

Another significant side-channel we addressed is a processor leaking information about its execution state through power consumption and physical emissions. Unlike most modern processors, our design does not feature an idle state where power conservation measures such as varying clock speeds or power gating are used to consume less power.

Our design also issues dummy instructions during pipeline stalls or pipeline where the execution stage would otherwise be unused. Dummy instructions are issued whenever the FIFO queue is directed to stall; these instructions are always considered invalid in order to prevent any meaningful modifications to the processor state, but they consume power as if they were regular ALU operations.

Currently, dummy instructions are produced using a pseudorandom number generator (PRNG) based on a 32-bit linear feedback shift register (LFSR). We chose to use a LFSR because it was a simple and quick method of generating non-sequential instructions over a period of 4 294 967 295 clock cycles. However, this method is not truly random as the LFSR output depends directly on the previous value and the design of the feedback loop. An attacker with enough knowledge on the specific design of our LFSR can easily predict the

sequence of dummy instructions being issued. The LFSR is just a temporary measure until we can implement a true random number generator (TRNG) that uses a physical process.

Additionally, writing data to a register consumes a noticeable amount of power, so we created a solution to obfuscate when instructions write to registers. Within the RISC-V specification, register 0 is fixed to the value 0, and writing to register 0 does not change the register file state. While some implementations would simply omit the physical register and hardwire a fixed value in place of register 0, we chose to implement a physical register in its place to accept data writes to register 0. However, any data read from register 0 is hardwired to the value 0 and has no connection to the output of the physical register. Furthermore, instructions without a destination register, as well as dummy instructions, write to this dummy register in order to further obfuscate when the register file is modified.

4.2.6 Encryption Engine

Typically, many ISAs use a dedicated instruction set extension for encryption algorithms. The proposed cryptographic extension to the RISC-V ISA uses unique instructions to direct the processor to execute individual stages of encryption for AES [26]. We deviated from this approach because we believe that encryption through the pipeline can create recognizable patterns in the processor execution. This has also been proven in many side-channel attacks focused on detecting and analysing encryption in processors.

We believe that an encryption engine separate from the pipeline provides better security over executing encryption instructions on the processor. By separating from the pipeline, we can outright prevent any user-defined code from revealing intermediate states of the encryption process and allow the encryption process to operate independently from the rest of the processor. Furthermore, a dedicated encryption engine can encrypt faster as well as execute concurrently with the processor.

Given the concerns for the security of encryption ciphers in a post-quantum world, we chose 128-bit AES for the processor encryption engine. The structure of the proposed engine is depicted in Figure 4.3.

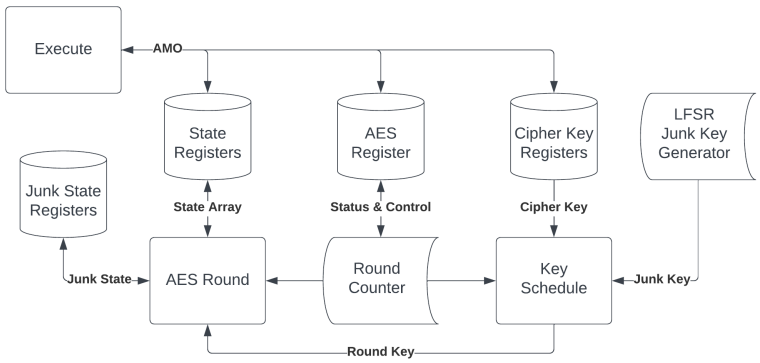

Figure 4.3. Block Diagram Overview of Encryption Engine

The engine is designed to be constantly encrypting with minimal intervention required by the processor. Similar to how the pipeline executes dummy instructions during pipeline stalls and flushes, the engine encrypts dummy data when it does not actively have useful data provided by the processor. This design will obfuscate the presence of meaningful encryption because the engine simply looks like it is always encrypting when observed. This behavior also contributes to the power and emissions obfuscation scheme of the processor by introducing more noise in power consumption and physical emanation. By relying solely on a separate encryption engine rather than utilizing the standard processor pipeline resources, we can ensure that the timing of encryption and decryption process is independent of stalling or data dependencies in the normal pipeline.

The processor communicates with the encryption engine through CSRs. The data block consists of four CSRs while the 128-bit encryption key is held by eight CSRs. Only IROM instructions have the ability to load the encryption key into the these CSRs in order to ensure that the remnants of the key are not leaked in the processor state upon resuming normal operation. Reading data from these key CSRs are never permitted. An encryption control CSR is used to direct the encryption engine to the loaded data. An encryption status CSR will indicate to the processor when the engine has completed the encryption/decryption process. The processor must poll this status CSR in order to know when the process is complete. When attempting to read from the encryption data CSRs before encryption/decryption is complete, the engine simply outputs a zero value in order to prevent the processor from

accessing the intermediate states of the block cipher.

Due to time constraints, we were not able to fully complete the RTL implementation for the encryption engine. The completion and testing of an encryption engine for our processor is deferred as future work.

4.3 Simulation Results

We will now discuss the results of our simulation and testing. The testbench simulates a clock signal for the processor and monitors for retiring instructions. Retiring instructions and their effect on the processor state (changes to registers, memory, or program counter) are compared to a log generated by the Spike simulation to ensure that the processor is executing instructions in the proper order and producing the correct results.

4.3.1 Compliance Testing

The RISC-V Compliance Tests are designed to test whether our design will meet the bare minimum standard of the specification. Each test executes multiple iterations of a specific instruction in order to test whether our design can execute according to the specification. For example, the ADD test will execute 587 different add instructions using various values across all available registers to ensure that the simple ADD instruction functions as expected. Our design successfully passed all tests included in the RV32I suite of tests, passing the minimal standard for RISC-V. However, these tests cannot be considered as substitutes for formal verification, therefore we cannot claim that our design has fully passed the verification process. All compliance test files can be found in Appendix B.

4.3.2 Dhrystone Benchmark

Dhrystone is a synthetic benchmark program originally written in 1984 by Reinhold P. Weicker [27], and it tests the integer performance of a processor. While Dhrystone is not a realistic benchmark for measuring real world performance, it is often used in the processor design industry as a standard. The benchmark score is normally represented in amount of time taken to complete an iteration of the benchmark, but we chose to focus on instructions per clock (IPC) as a benchmark metric as it is independent of the clock timing. The adapted source files for the benchmark program can be found in Appendix C.

The benchmark was run for ten iterations, or ten Dhrystones. The benchmark executed a total of 7563 instructions in 17 475 clock cycles, resulting in approximately 0.43 IPC. These results include the clock cycles used to execute the setup instructions executed prior to the start of a Dhyrstone iteration. In comparison, a similar RISC-V processor designed for performance, the SiFive E31 Standard Core, is capable of reaching 0.95 IPC [28].

Further analysis of the benchmark run shows that memory load operations account for a significant amount of the stalled cycles. Of the 2137 load instructions were executed in the benchmark, stalls represented 36 percent of the total runtime. This underscores that the IPC performance of the design is highly dependent on the memory load latency due to the lack of a memory cache.

Another significant contributor to the performance loss is branch misprediction. The benchmark executed a total of 1008 total branch instructions. 189 JALR instructions were not predicted because our design does not predict for indirect branch instructions. 134 branches (or 16%) of the 819 remaining branch instructions were mispredicted. Many of these mispredictions are difficult to avoid during the first few iterations of the benchmark because the predictor must guess without prior branch histories. Further iterations will improve the overall misprediction rate. Each mispredicted and unpredicted branch instruction incurs a ten-clock-cycle penalty before the next valid instruction retires in the pipeline. Branch misprediction accounted for 18% of the total clock cycles. If branch prediction is disabled, the benchmark runtime increases by 39%.

Throughout the benchmark, a total of 9912 dummy instructions were also executed. Dummy instructions accounted for over half of the total time of the benchmark. Without the dummy instructions, the pipeline would simply be stalled for more than half of the total runtime of the benchmark. However, our design makes it difficult for an attacker to determine when meaningful instructions are being executed because the processor is technically executing all the time. A consequence of this power obfuscation design is that the system could consume up to 130% more power. Comparing power consumption to other processors is difficult at this stage of design and is an item of future work. Key factors such as clock frequency and voltage specific to this processor are required to calculate how much power would be consumed.

CHAPTER 5:
Conclusion

5.1 Assessment of Design and Goals

Our research highlights how difficult secure processor design is without noticeable compromises to performance and power consumption. However, these compromises may be acceptable for computer systems that do not need to be the fastest or most power-efficient, but they should be more protected because they are likely targets of interest to adversaries. Many military embedded systems handling important functions such as encryption, navigation, or fire control systems would benefit from increased resilience to cyberattacks.

5.2 Future Work

5.2.1 Additional RISC-V Extensions and Features

RISC-V offers many extensions and features that we did not implement due to time constraints. Their additional functionality was not a priority for this project, but they are of interest for future work. Extensions for multiplication, division, floating point operations would improve the processor capabilities to execute more sophisticated programs while other features such as encryption, privilege modes and physical memory protection (PMP) features would further enhance the security of the processor design.

An important feature of our design was eliminating variable instruction timing depending on data. Each instruction consumes the same number of clock cycles regardless of the data that is being operated on in order to prevent side-channel attacks that could deduce data values based on their effect on instruction timing. Future implementations of extensions and their instructions, such as integer multiply or divide operations, must adhere to data independent instruction timing in order to align with our design principle.

5.2.2 True Random Number Generation

An important design principle for security that we previously discussed was being unpredictable to observation. However, we noted how the PRNG we used to issue dummy instructions is not truly unpredictable. Its predictability poses a vulnerability to our desired unpredictable noise generation. A TRNG using an unpredictable physical process would be a vital enhancement for the security of our design.

5.2.3 Encryption Implementation

As previously discussed, encryption capabilities are a requirement for secure hardware, as they are used to ensure data integrity, confidentiality, and authentication. Thus, completing the AES encryption engine for the processor is a priority for future work. Furthermore, quantum-resistant ciphers such as Crystals-Kyber should be of great interest for future work in anticipation of encryption algorithms in a post-quantum world.

5.2.4 Root of Trust

While our design is organized to control how data can be written into IRAM, we do not have a mechanism to ensure a valid program is being loaded and executed. Many of the discussed attacks are possible because attackers are able to execute their own code on the processor. We can heavily limit the ability of an attacker to manipulate the state of the processor by authenticating a loaded program. What we need to incorporate into our design is a set of built-in cryptographic keys, known as a root of trust, to authenticate programs and data before they are used by our system [13]. After the processor is produced, programs developed for the system can be written and submitted to an approval authority. Once a program is deemed secure and approved to be executed, a digital signature can be appended to prove to the processor that the program has been authorized for execution. Without the proper signature, the processor should reject the program and any data attempting to enter into memory.

5.2.5 Error Detection and Correction

A key factor that we did not specifically address in our design was error detection and correction. Like bugs, hardware errors pose a security risk due to the unintended side effects they might cause. While it is not possible to prevent all hardware errors entirely,

our design should prevent as many undetected errors as possible. At minimum, we need to protect data integrity and reliability with some form of error correction code (ECC) to detect errors.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
RTL Code Repository

The RTL code can be examined at the following repository.

RTL Repository: https://gitlab.nps.edu/roy.shin/NPS_RISC-V_Thesis/-/tree/main/RTL

POC: Author, shin@usna.edu, roy.shin@usmc.mil.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
Compliance Test Repositories

The source code for the compliance tests were adapted from the following repository.

Compliance Tests Source Code Repository: https://github.com/riscv-non-isa/riscv-arch-test/tree/main

POC: Neel Gala, CTO, InCore Semiconductors, neelgala@incoresemi.com.

The adapted code and simulation outputs can be examined at the following repository.

Test Binaries and Results Repository: https://gitlab.nps.edu/roy.shin/NPS_RISC-V_Thesis/-/tree/main/testing

POC: Author, shin@usna.edu, roy.shin@usmc.mil.

List of compliance tests passed:

add-01

addi-01

and-01

andi-01

auipc-01

beq-01

bge-01

jal-01

jalr-01

lui-01

or-01

ori-01

sll-01

slli-01

slt-01

slti-01

sltiu-01

sra-01

srai-01

srl-01

srli-01

sub-01

xor-01

xori-01

APPENDIX C:
Benchmark Code Repositories

The source code for the Dhrystone benchmark can be found at the following repository.

Benchmark Source Code Repository: https://github.com/riscv-software-src/riscv-tests/tree/master/benchmarks/dhrystone

POC: Steven Pemberton, CWI, Amsterdam, Steven.Pemberton@cwi.nl.

The adapted code used for the simulated benchmark can be examined at the following repository.

Adapted Source Code and Benchmark Binaries Repository: https://gitlab.nps.edu/roy.shin/NPS_RISC-V_Thesis/-/tree/main/benchmark

POC: Author, shin@usna.edu, roy.shin@usmc.mil.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] P. C. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, pp. 5–27, 2011 [Online]. Available: https://api.semanticscholar.org/CorpusID:12262854

[2] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, August 2014, pp. 719–732 [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[3] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '18). New York, NY, USA: Association for Computing Machinery, 2018, p. 693–707 [Online]. Available: https://doi.org/10.1145/3173162.3173204

[4] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019 [Online]. Available: https://spectreattack.com/spectre.pdf

[5] A. Johnson and R. Davies, "Speculative execution attack methodologies (SEAM): An overview and component modelling of spectre, meltdown and foreshadow attack methods," in *2019 7th International Symposium on Digital Forensics and Security (ISDFS)*, 2019, pp. 1–6 [Online]. Available: https://doi.org/10.1109/ISDFS.2019.8757547

[6] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018 [Online]. Available: https://meltdownattack.com/meltdown.pdf

[7] L. Chen *et al.*, "Report on post-quantum cryptography," 2016-04-28 2016 [Online]. Available: https://doi.org/https://doi.org/10.6028/NIST.IR.8105

[8] A. Waterman, K. Asanović, and J. Hauser, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203," Rep., December 2021 [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[9] R. N. M. Watson, S. W. Moore, P. Sewell, and P. G. Neumann, "An introduction to CHERI," Rep., 2019 [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf

[10] F. A. Fuchs, "Developing a test suite for transient-execution attacks on RISC-V and CHERI-RISC-V," Rep., 2021 [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/202106-carrv-transient-execution.pdf

[11] N. C. Will and C. A. Maziero, "Intel software guard extensions applications: A survey," *ACM computing surveys*, vol. 55, no. 14s, pp. 1–38, 2023.

[12] J. Van Bulck *et al.*, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018. See also technical report Foreshadow-NG [29].

[13] J. Szefer, "Principles of secure processor architecture design," *Synthesis Lectures on Computer Architecture*, vol. 13, pp. 1–173, 10 2018 [Online]. Available: https://doi.org/10.2200/S00864ED1V01Y201807CAC045

[14] O. Lo, W. J. Buchanan, and D. Carson, "Power analysis attacks on the AES-128 S-box using differential power analysis (DPA) and correlation power analysis (CPA)," *Journal of Cyber Security Technology*, vol. 1, no. 2, pp. 88–107, 2017 [Online]. Available: https://doi.org/10.1080/23742917.2016.1231523

[15] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture* (ISCA '07). New York, NY, USA: Association for Computing Machinery, 2007, p. 494–505 [Online]. Available: https://doi.org/10.1145/1250662.1250723

[16] S. Deng, N. Matyunin, W. Xiong, S. Katzenbeisser, and J. Szefer, "Evaluation of cache attacks on arm processors and secure caches," *IEEE Transactions on Computers*, vol. 71, no. 9, pp. 2248–2262, 2022 [Online]. Available: https://doi.org/10.1109/TC.2021.3126150

[17] F. Liu and R. B. Lee, "Random fill cache architecture," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 203–215 [Online]. Available: https://doi.org/10.1109/MICRO.2014.28

[18] C. Canella *et al.*, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 249–266 [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[19] C. Canella *et al.*, “A Systematic Evaluation of Transient Execution Attacks and Defenses,” in *USENIX Security Symposium*, 2019. extended classification tree at https://transient.fail/.

[20] G. Alagic *et al.*, “Status report on the third round of the nist post-quantum cryptography standardization process,” 2022-07-05 04:07:00 2022 [Online]. Available: https://doi.org/https://doi.org/10.6028/NIST.IR.8413

[21] A. Waterman and K. Asanović, “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2,” Rep., May 2017 [Online]. Available: https: //github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[22] S. Sutherland, *RTL Modeling with SystemVerilog for Simulation and Synthesis Using SystemVerilog for ASIC and FPGA Design*. Tualatin, OR, USA: Sutherland HDL, Inc, 2017.

[23] T. R. of the University of California, 2023. *GNU toolchain for RISC-V, including GCC*. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain/tree/master

[24] T. R. of the University of California, 2021. *Spike, a RISC-V ISA Simulator*. [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim

[25] T. R. of the University of California, 2023. *RISC-V-tests*. [Online]. Available: https: //github.com/riscv-software-src/riscv-tests/tree/master

[26] A. Zeh *et al.*, “RISC-V Cryptography Extensions Volume I,” February 2022 [Online]. Available: https://github.com/riscv/riscv-crypto/releases/tag/v1.0.1-scalar

[27] R. P. Weicker, March 1995. *Dhrystone*. [Online]. Available: https://github.com/riscv-software-src/riscv-tests/tree/master/benchmarks/dhrystone

[28] “Dhrystone performance tuning on the freedom platform,” *SiFive* [Online]. Available: https://www.sifive.com/blog/dhrystone-performance-tuning-on-the-freedom-platform

[29] O. Weisse *et al.*, “"Foreshadow-NG": Breaking the virtual memory abstraction with transient out-of-order execution,” Rep., 2018. See also USENIX Security paper *Foreshadow* [12].

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California