CACHE PERFORMANCE AND DYNAMIC MEMORY PROTECTION IN EMBEDDED SYSTEMS

By Andrew Kendall

A Thesis

Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Science

in Electrical Engineering

Northern Arizona University

May 2023

Approved:

Paul Flikkema, Ph.D., Chair

Michael Gowanlock, Ph.D.

James Palmer, Ph.D

ABSTRACT

CACHE PERFORMANCE AND DYNAMIC MEMORY PROTECTION IN EMBEDDED SYSTEMS

ANDREW KENDALL

In embedded system design, there is an ever-increasing need to improve security to meet the demands imposed by cyber-attack advancement. Key to the security of embedded systems are ways to exchange information between trusted zones and external memories vulnerable to attacks. System designers face the issue of increased program execution times due to increasingly complex security hardware. The research in this thesis is part of a larger effort to design a secure processor that develops specialized cryptographic and data randomization techniques in hardware. This thesis integrates caching to increase the security and performance of the secure processor. The design approach of the instruction and data caches allows us to test the effectiveness of cache size versus performance. The caches also interface with a shielding module that implements a dynamic computational diversity architecture. The shielding module and the caches implement continuous and periodic memory shuffling to protect against learning-based attacks. This thesis performs tests for execution time, perceived randomness of data, and hardware utilization. These results assess and optimize system security, computation, and hardware resource use.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

1.1 Secure Computing

In System-on-Chip (SOC) design, there is an ever-increasing need to improve security to meet the demands imposed by new and more advanced cyber-attacks. Key to the security of embedded systems are ways to exchange information between trusted zones and external memories vulnerable to attacks. Fast and lightweight embedded systems seek to offer the best performance with low hardware costs. The main challenge of providing a computing system with additional security is the cost introduced by the complexity of the hardware. In addition to more complex, expensive, and specifically designed security components, program execution time is increased when there is a memory request and new information needs to be secured. One solution to this issue is increasing the efficiency of memory accesses.

A typical improvement of the memory hierarchy is the addition of instruction and data caching. Caches are smaller and faster memories than main program and data memories. When a large portion of the information the processor needs is already located within the cache, it reduces the time it needs to stall due to the slower access time of the main memory. Cache memories are used as high-speed buffers between processors and main system memories, allowing memory access time to compete with the increase in processing speeds.

Embedded systems implement cache memories to decrease program execution times but can also be used for increasing security performance. Some caches are designed to work directly with cryptographic techniques within the embedded system

design industry. The cache system developed in this thesis implements a secondary function that hinders the ability of an attacker to analyze patterns.

1.2 Motivation

The primary aim of this research is to assess the ratio of security improvements and the cost associated with the extra hardware required. In addition to this analysis, this research designs a new memory hierarchy with the addition of continuous and periodic instruction and data caching. This new cache design aims to find optimizations considering security, hardware area, and execution time. This research is part of a larger project implementing a processor integrating multiple cryptographic techniques for embedded systems. This thesis covers the design and performance testing of instruction and data caches and how cache size affects the security and performance of embedded systems. These caches work directly with a previously implemented shielding module that provides the main information security functionality. Parameters considered in system testing include execution times of benchmark programs and hardware cost in terms of area for each module in the system.

In addition to these parameters, the perceived randomness is quantified using a statistical randomness test suite. Performance testing aims to find a cache size optimization point where execution time, area, and randomness are the most efficient. This thesis also includes research on commonly used cache architectures and the design process that resulted in the final system.

1.3 DCD Computing System

This research utilizes the Dynamic Computational Diversity (DCD) Secure Computing System [9]. This system implements a secure way to access data and instructions from system memories that are not trusted. This secure computing system

combines multiple cryptographic hardware primitives to achieve this goal. This includes continuous memory shuffling, a lightweight block cipher, a physical unclonable function (PUF), and other methods that add to the perceived randomness. The main goal of the hardware modules is to ensure that information in the data and program memories is unreadable to an attacker monitoring the memories or memory busses. These modules also include detecting unauthorized access or tampering through verification methods. This secure architecture design intends to implement the desired security behavior while maintaining the best possible program execution time.

This system is designed with a hardware description language and is synthesized and implemented on a field-programmable gate array (FPGA). This allows the design to be updated and tested for different system versions and allows internal parameters to be changed. The main advantage of using these hardware design tools is to analyze system behavior and performance before finalizing and designing an application-specific integrated circuit (ASIC). The list below states the steps for the computing system's implementation:

- Step 1: Design Verilog HDL files.
- Step 2: Initialize memory files.
- Step 3: Compile all design and memory files in Vivado.
- Step 4: Synthesize design.
- Step 5: Implement design.
- Step 6: Generate the FPGA configuration file.
- Step 7: Upload the configuration file to the FPGA board.

The entire system is designed in the Verilog hardware description language and is divided into modules that make up the system hierarchy. The top-level module of the DCD system contains four main modules that each provide a specific function necessary to the overall behavior of the system architecture:

- RISC-V Processor: 5-stage pipelined 32-bit processor.
- System Controller: Module that controls data flow, instructions, and all system control signals.
- Shielding Module: System security module that implements cryptographic techniques.
- Main Memory: External off-chip main memory.

1.3.1 RISC-V Processor

The processor module is the trusted zone of the system, which implements a 32-bit RISC-V instruction set architecture (ISA) that uses a base integer instruction set and an extension for multiplication and division instructions [25]. The RISC-V platform is an open-source ISA that allows for customization and innovation and is simple to integrate with application-specific systems [25]. This ISA is commonly used for FPGA and embedded applications [25]. This processor design can also be pipelined.

1.3.2 System Controller

The DCD secure computing system control module is designed as a state machine that interfaces with main memory, the processor, and the shielding module. The state machine has four main states: Initialize, Idle, Fetch, and Write. Each state has sub-states that send specific control signals at different times. This is because the

shielding process requires multiple functions executed at specific times. The shielding process takes multiple clock cycles, and the processor must be stalled for the proper amount of time.

1.3.3 Shielding Module

The shielding module implements the functions that secure data and instructions in memory. The DCD computing system uses periodic fetch and write operations that de-shield information to be stored in the cache and re-shield information to be stored back into main memory. The shielding process consists of multiple security and encryption techniques. These functions include a lightweight block cipher, pseudo-random number generation, controlled noise injection, and periodic memory shuffling of blocks. This module uses these techniques to apply the DCD architecture.

1.3.4 Main Memory

This memory module is the untrusted zone of this system that a potential attacker could access. For this reason, only protected instructions and data are communicated between main memory and the SOC. Individual blocks fetched and written between the two never appear the same on subsequent cycles. This is due to the nature of the shielding module and the sequence of unique variants it produces for each block.

1.4 Objectives

The objectives for this thesis are:

- Design and implement instruction and data caches in the hardware description language Verilog.
- Create different system versions, implementing caches of 2, 4, 8, and 16 cache

lines.

- Design and implement a replacement policy for the data cache.
- Integrate the hardware modules into the DCD secure computing system.
- Test system for functionality with a series of Verilog test benches.
- Synthesize the system design and program onto the FPGA.
- Use performance benchmark programs to analyze program execution time.
- Use the DieHarder test suite to test the system's perceived randomness.
- Record and create visuals for the results obtained from the system tests.
- Determine optimization points for system performance based on the cache design variables.
- Conclude this thesis with a definitive statement about the optimizations and how they apply to the system's application and functionality.

1.5 Research Methods

The first step was to research and design an instruction and data cache for the system. This was done in the Verilog hardware description programming language. The initial designs were simulated using Vivado, a software tool for simulating hardware. Designs were compiled and tested in functional simulations for behavior verification. Once the design was finalized, multiple system versions were made for each cache size. These designs were synthesized and implemented onto an FPGA and run with benchmark programs to test performance for each version. The parameters we focused on were cost, security, execution times, cache miss rates, and miss penalties.

These results were then compiled and analyzed with MATLAB for visual and statistical interpretation. The next step was to analyze the test results to determine if optimizations could be made based on cache size. The final goal of this thesis is to understand the trade-off of cost versus performance.

Chapter 2

LITERATURE REVIEW

2.1 Caching Fundamentals

Cache memory is a type of memory that is located between the main memory and the processor. It stores data or instructions that the processor may frequently need, reducing the time it takes to receive data [14, 20]. Processor speeds have become much faster than memory speeds, and the gap continues to increase [26]. Caching combats this problem by introducing smaller and faster memory as a buffer between the two [6, 26]. When memory blocks need to be reused and stored in the cache, they are immediately available. The job of cache memory is to store data that is highly likely to be used again. Blocks of memory are addressed within the cache in different ways depending on the type of cache architecture [10, 14]. Blocks of memory are the sets of data or instructions that are stored in the cache from the main memory [10]. These memory blocks are transmitted to the cache and then sent to the processor to be used.

The block size depends on the main memory and system design. The cache size and the size of a block change how memory is placed and addressed. Each block is placed within a cache line, a section of the cache that stores a block of information individually. The number of cache lines is equal to the number of blocks that the cache can store; as the size of a block increases, the number of cache lines decreases [10, 20, 22]. There are some trade-offs to increasing or decreasing the size of a block. Increasing the block size typically decreases the cache's miss rate but also increases the miss penalty [10]. The next section covers how each type of cache architecture

places blocks within cache lines.

2.2 Cache Architectures

2.2.1 Direct-Mapped

Direct-mapped is a type of cache architecture that maps each block in memory to a single location in the cache [10]. Since main memory is larger than cache memory, many blocks can be mapped to the same location. The tag of a block is a set of address bits that distinguish memory blocks from one another [12, 14]. Each cache line stores a tag that is unique to the block that is placed. When the processor requests a block, the tag of the requested block and the tag of the stored blocks are compared. One tag comparison is needed to determine if a block is present in the cache because the requested block is mapped to a single line [10, 12, 14]. This reduces the complexity needed to check the address tag compared to other cache placement architectures [10]. Part of the memory address is used as a block index to determine the cache line for comparison. This identifies the specific set of blocks mapped to a line [10]. The remaining bits determine the exact location of data within a block. These offset bits address a byte or word within a block [10].

2.2.2 Set Associative

Set associative is a cache placement architecture that allows blocks to be stored in multiple locations in the cache [12]. Associativity is a value that defines the number of cache lines each block is mapped. This architecture's associativity is a predetermined value ranging from one to the number of cache lines [12]. Set associativity equal to one is another definition for direct-mapped, and set associativity equal to the number of cache lines is another definition for fully associative, which will be covered in the

next section. This architecture is concerned with associativity with values between one and the number of cache lines. To check whether a block is stored in the cache, the search logic checks several cache lines equal to the value of associativity [12] [14]. As this value increases, more comparisons are needed to check for a cache hit. The tag address bits correspond to the block's location within a set, and the index bits represent the set. The tag size increases compared to direct mapping because the number of blocks that can be stored in a line also increases. The index size decreases as the associativity rises because the number of sets decreases. This reduces the redundancy in the address where a block can be located, and more bits are utilized for the tag field [12].

2.2.3 Fully Associative

Fully associative cache architectures implement a placement policy where every memory block can be stored in any of the cache lines [10, 12, 14]. This architecture represents a set-associative model where the associativity equals the number of cache lines. More tag bits must be used to address a single block because the cache needs to know specifically where the block was in memory [12]. Index bits are no longer needed to determine the set of cache lines. The tag bits now represent the exact index of a block from the main memory. Fully associative caches reduce the miss rate compared to set associative and direct-mapped architectures because there is a higher probability that a requested block can be in any of the lines [10]. This creates a higher complexity because the cache needs to check every single cache line to determine the location. The performance and cost of each of these cache architectures must be considered when determining the best one to use for a specific application.

2.3 Replacement Policies

Replacement policies are algorithms used to determine which blocks are evicted once the cache is full and a new block needs to be stored [14, 20]. Some common replacement policies include first-in-first-out (FIFO), least recently used (LRU), and random replacement. FIFO uses queuing logic to evict data stored in the cache the longest. The order in which the blocks have been placed in the cache is the order in which they will be evicted. Random replacement selects any block in the cache and evicts it to make room for a new block. This policy does not need to store information about how long the block has been stored in the cache. The replacement policy chosen for the cache design in this thesis is LRU. LRU was chosen for this design because it is more likely to keep blocks that are more frequently used [14, 20]. The addition of the shielding module increases memory access time. It is beneficial that a block does not have to move between memory and the cache more times than it has to.

2.3.1 Least Recently Used

Least Recently Used (LRU) is a cache replacement algorithm that predicts which blocks in the cache are least likely to be used again and replaces them with new blocks [10, 14, 20, 22]. This prediction is determined by keeping track of the order in which each block has been accessed. The block highest in the order is the most recently used, and the lowest is the least recently used [14, 22]. LRU is a highly efficient algorithm because it is easy to implement at low associativity and generally achieves low cache miss rates [22]. However, as higher associativity is needed for the cache, the LRU algorithm becomes more complex [22]. Since this algorithm has increased complexity at higher associativity, efficient hardware implementations are needed to make it a viable replacement policy. There is also a need for testing

the efficiency and reliability of the LRU for different cache sizes to determine an optimization point [1, 8, 13, 18].

The LRU algorithm keeps a history for each line and the order in which each has been accessed. When there is a miss, and the cache needs to fetch a new block, the LRU algorithm determines the line with the oldest block, and the cache will place the new block in that line [14, 22]. The LRU algorithm then adjusts and calculates a new LRU line by determining each block's age and making the newly placed block the most recently used [14, 22]. This is updated every time the processor requests a new block. When there is a cache hit, the line that has been hit will become the most recently used, and then the age of each block calculated by the LRU will update [14]. This computation can be done in multiple ways depending on the type of implementation of the LRU.

2.4 Security Review

2.4.1 Embedded Systems

Embedded systems apply application-based solutions and can be complex. The constraints applied to these systems depend on their interaction with the physical world [11]. System constraints include processing speeds, hardware footprint, power requirements, available memory, and application [11]. Embedded systems involve complex interactions on both hardware and software levels [11]. Both components are designed to work together and compensate for functional or performance inconsistencies in hardware or software implementations. This design process results in unique components you may not see in standard computer systems [11]. For example, a system may favor computation best applied in hardware. In this case, specialized hardware can be designed to perform this functionality. Embedded systems can be

designed for a seemingly infinite amount of applications [11].

The DCD system architecture targets secure processor design for embedded system applications. It executes programs while securely fetching and writing all data and instructions between the processor and external memory. Unique and sophisticated security and computational hardware were designed to meet the functionality requirements and provide the best performance possible. This leads us to review cybersecurity risks and techniques designed for embedded systems and SOCs.

2.4.2 Security Risks

A security risk is the potential for data loss or leakage in the system resulting from a cyber-attack. The main parameters, the security triangle, used to define risk are confidentiality, integrity, and availability (CIA) [2, 23]. Attackers seek to exploit the vulnerability of these three parameters to gain access to data, corrupt data, or disable system functionality. To measure a security risk, the parameters of CIA and all potential attacks that an attacker could use are considered [2, 23]. Cyber-attack methods are typically separated into two main categories: physical and software-based methods [2, 23]. Software methods attack embedded systems that are connected to a network. These attacks target vulnerabilities and errors within operating systems and program images. The physical category includes a wide range of attacks; some examples include fault attacks, side-channel attacks, eavesdropping attacks, and data injection attacks [2, 23].

A side-channel attack is when an attacker monitors the behavior and output of system operations [2, 23]. Monitoring these components may reveal patterns or vulnerabilities that lead to the leakage of information [2, 23]. An attacker will analyze data and physical parameters such as timing, power usage, frequency of communication, and patterns that may occur in encrypted data [2, 23]. Potential side-channel

attacking methods are acoustic, brute force, cache-based, electromagnetic analysis, error message, frequency-based, glitch, power analysis, timing analysis, safe error, scan-based, and visible light [2, 23]. This thesis considers these attacks when evaluating security performance. Security risk is quantified by the potential for attacks and the CIA parameters associated with each [2, 23].

2.4.3 Security Techniques

Two main techniques used to increase system security are cryptography and randomization. One cryptography technique commonly used in embedded systems is ciphers [4, 17]. Ciphers are algorithms that perform encryption and decryption [4, 17]. They can be implemented in hardware or software depending on cost constraints and needed complexity mentioned earlier [4, 17]. The cipher utilized in this thesis is a lightweight block cipher called Prince [4]. A block cipher is a deterministic encryption algorithm operating on fixed-length bit-streams, otherwise known as blocks [4, 17]. They can be implemented cheaply and are commonly used for encrypting large amounts of data [4, 17]. Another cryptography technique is called hashing. A hash function is commonly used to authenticate data based on an equation or a series of mathematical formulas [17, 21]. This one-way deterministic function always generates a unique value corresponding to a given input [17, 21]. Additionally, a hash function should make it near-impossible to transform the hash value back into the input string [17, 21].

The other primary technique for improving security is randomization. One randomization technique covered in this research is random number generation (RNG). RNG can be implemented by an algorithm or hardware device that outputs numbers that cannot be reasonably predicted [24]. RNG output numbers can be utilized for cipher keys, memory addresses, and noise injection [24]. Some hardware devices that

implement RNG described in Section 3.2, are the DCD System's physical unclonable function (PUF) and linear-feed back shift registers (LFSR). Other forms of randomization include noise injection, logical functions, block ordering, and block tweaking. Noise injection transforms the data with a randomly generated value [2, 23]. Logical randomization can include a variety of transformations of two or more inputs, such as XOR, AND, OR, NAND, etc. Block ordering is a randomization technique that mitigates how much information can be collected from accessed memory locations [2, 23]. Block tweaking is a technique that transforms a block on multiple iterations of a program run so that it does not repeat a variation of itself at any point [2, 23]. These techniques are showcased in the DCD computing system architecture in Chapter 3. The system implements randomization and cryptography techniques to implement stealth and secrecy.

Chapter 3

DCD SYSTEM DESIGN

3.1 Module Hierarchy

The DCD Computing System is broken down into multiple modules and sub-modules. The DCD Top-Level module contains many modules within it. The design hierarchy is described as follows:

- Top-Level Module: DCD System.
- Module: All system components are placed directly in the Top-Level Module.
- Sub-Module: All components placed inside a Module.

Figure 3.1 shows an example diagram for the design hierarchy described in this thesis for reader clarity.

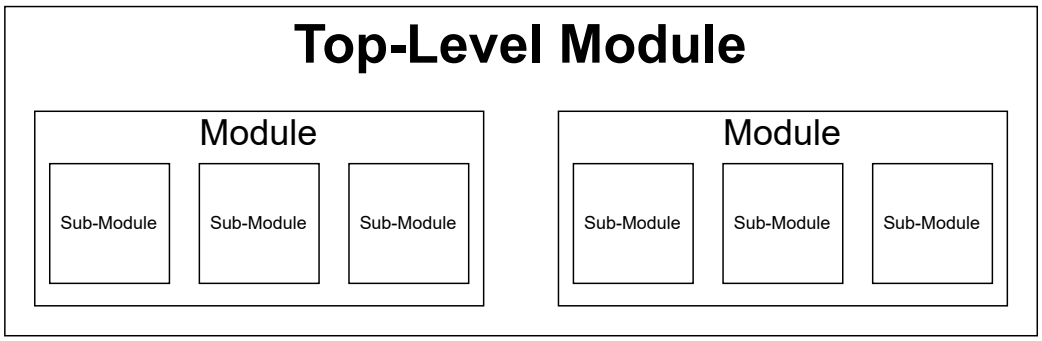


Figure 3.1: Design Hierarchy Example Diagram

The DCD Secure Computing System follows this layout. Figure 3.2 shows the DCD Top-Level module and the modules inside. This block diagram does not show the sub-modules contained within each module.

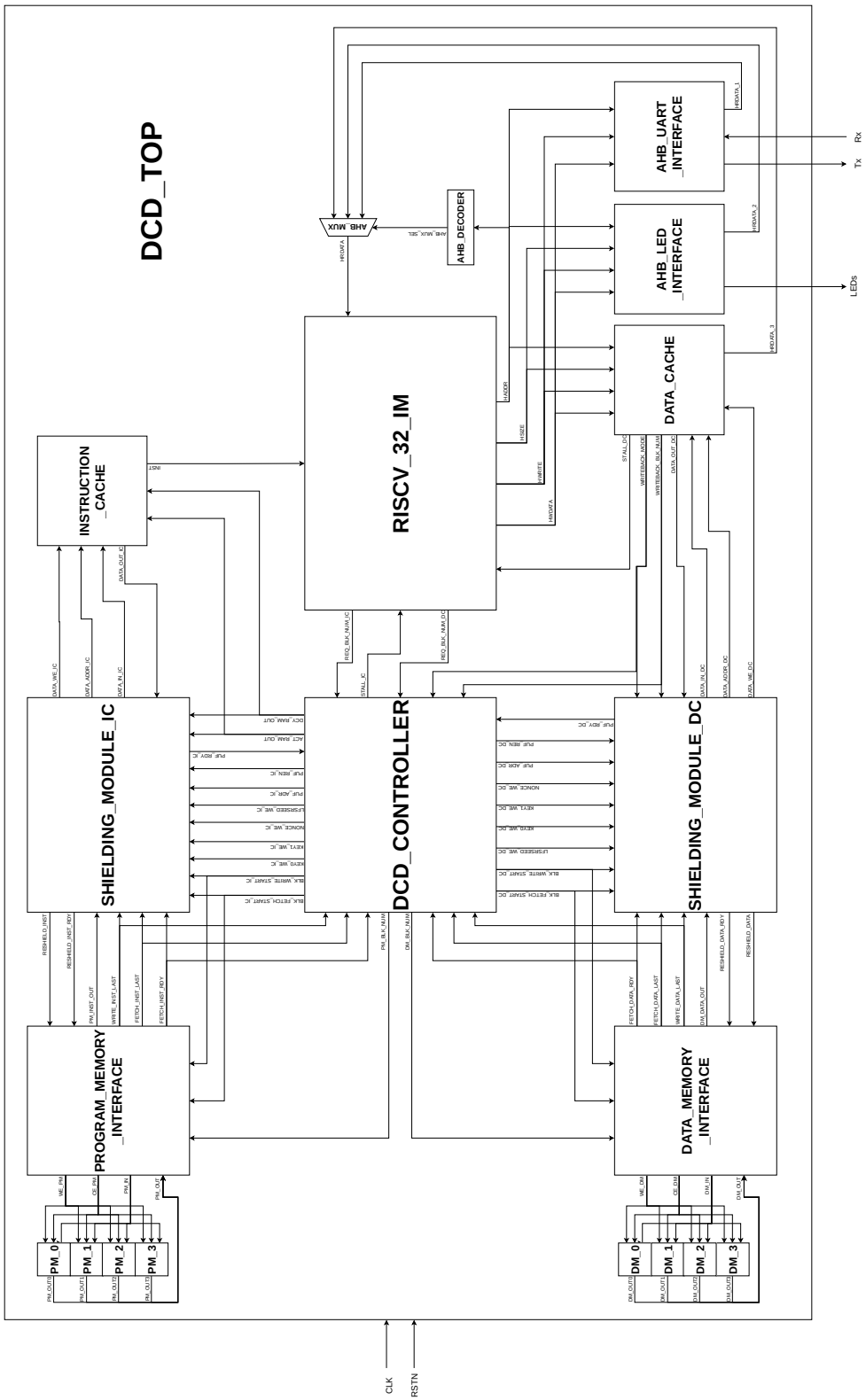


Figure 3.2: DCD Top-Level Block Diagram

3.2 Shielding Module

The DCD shielding module implements an XOR-Encrypt-XOR (XEX) secrecy design. The XOR elements of this design are pre-whitening and post-whitening keys, also known as tweaks. This is a minimal cipher design scheme where the encryption element is publicly known [7, 15]. This design follows the Even and Mansour method for minimal cipher designs, and the DCD shielding module expands on it [7]. This design seeks to stay efficient by keeping the same key and increasing variability by manipulating the plain-text before and after the encryption [15]. The cost of changing the tweak for plain-text manipulation is much smaller than changing the key for every encryption cycle [7, 15]. The ideal case would be implementing the encryption as a one-time pad (OTP).

In a computing system, instructions are often used repeatedly throughout program execution. Implementing this design as an OTP would require an impractically large cipher [7]. Tweaking offers a near-OTP design, which represents a new variant of a block in memory on each iteration of encryption. The DCD Secure Computing System strengthens this minimal cipher tweaking architecture. The list below shows a breakdown of the different elements that the DCD system implements to secure data:

- Key and Tweak Generation: The key and tweak inputs are generated using pseudo-random number generation hardware.
- XEX Design: The XEX design is similar to the proposed model with the addition of controlled noise and a much stronger permutation element.
- Decoy Shuffling: This element improves security by periodically re-encrypting random blocks in memory to minimize frequency analysis of when and where

blocks are being used in program execution.

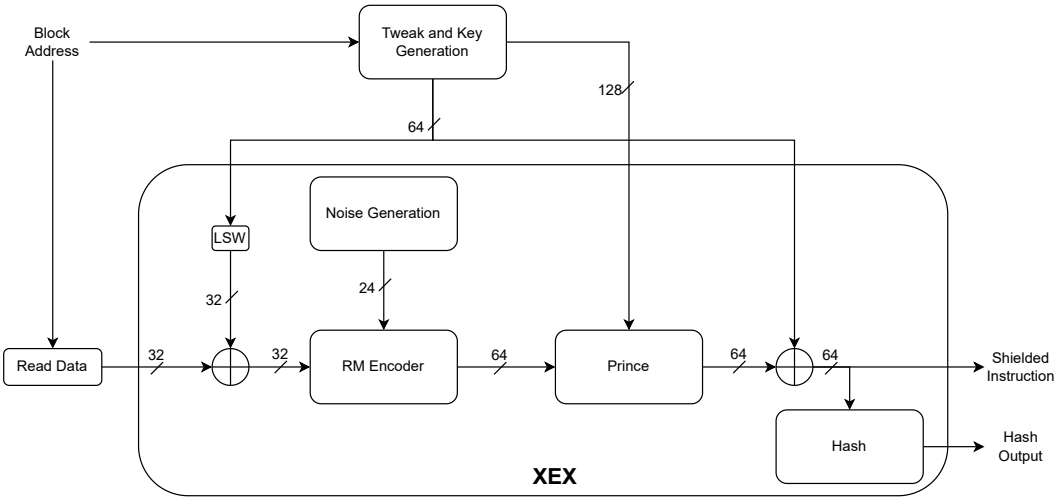


Figure 3.3: Shielding Top Level Block Diagram

3.2.1 Key and Tweak Generation

The shielding module utilizes pseudo-random numbers for encryption keys and tweaks for the XEX shielding element. Figure 3.3 visualizes this functionality as an element named Tweak and Key Generation. This represents the components described in this section. The DCD computing system is designed to interface with a PUF for pseudo-random number generation. The system version in this research does not integrate a PUF device. PUF integration is future work for the DCD project. Instead, the shielding module emulates this with a read-only memory (ROM) that is initialized with a set of randomly generated values in an external memory file at the time of programming. The ROM used to emulate a PUF outputs a 64-bit value based on a specific 8-bit memory address. Figure 3.4 shows a simplified block diagram of the ROM.

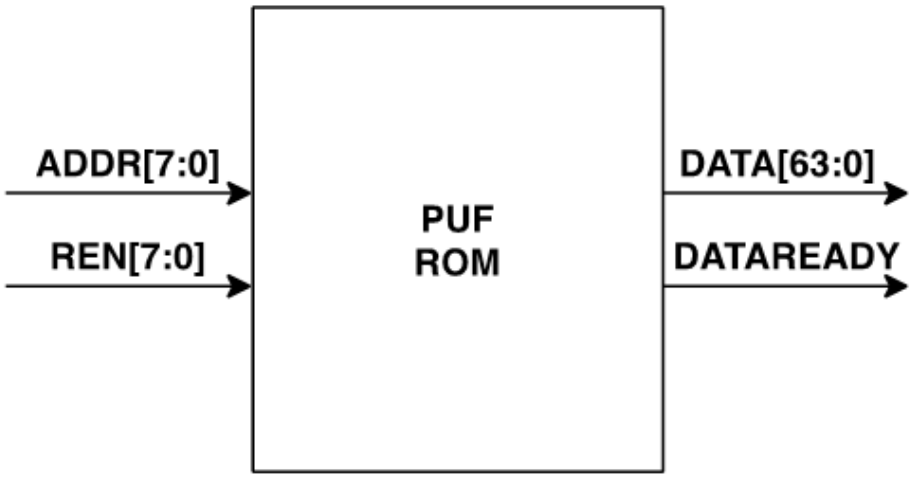


Figure 3.4: PUF Emulation ROM

The ROM is accessed twice to generate the key used in the encryption element. These two 64-bit accesses are initialized only at system start-up and reset. As stated in Section 3.2, this keeps the 128-bit key fixed for efficiency. For many block ciphers, changing the encryption key is costly regarding access time and resource use for key generation and setup operations [15]. Generating a new tweak value is much cheaper comparatively. This increases efficiency and benefits the system by introducing encryption-independent variability [15]. Every time a new block is encrypted, the ROM is accessed once for the 64-bit tweak value. This results in a new variation of the block for each iteration of block encryption in program execution.

The ROM access is used as the tweak for the first word in a block and is then transformed using a Galois Field (GF) Double at the end of a cycle. Data representation as a vector in a GF allows logical operations to scramble data with low hardware and execution time costs [3]. The logical operation, in this case, is simply a bit-wise XOR of the tweak and the field polynomial: $x^2 + x + 1$. This results in a new tweak value for each word within a memory block. Not only is the tweak changing for each block, but the tweak is also changing for each word in a feedback loop, as shown in

Figure 3.5. This limits the number of times the shielding module needs to access the ROM while continuously changing the tweak for every word.

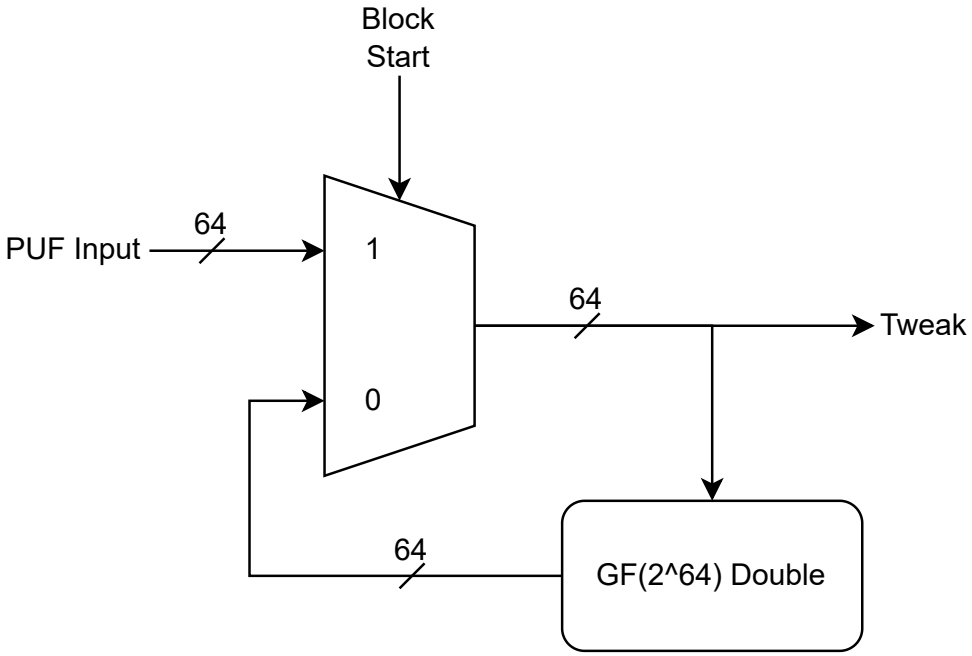


Figure 3.5: Tweak Galois Field Feedback Loop

3.2.2 XEX Design

The XEX design follows the proposed minimal cipher design and strengthens it to include extra layers of security. Figure 3.3 shows this in the block diagram labeled XEX. The XEX element of the shielding module is broken down into four stages:

- Stage 1: First Tweak.
- Stage 2: Channel Coding and Controlled Noise Injection.
- Stage 3: Prince Encryption.
- Stage 4: Second Tweak.

The first stage of the XEX implementation is the first tweak that takes the input plain text and performs an XOR operation with the least significant word of the 64-bit tweak value. This is the first data manipulation method similar to the minimal XEX design.

The DCD system strengthens this design in the second stage by adding controlled noise between the first tweak and the permutation element. The noise is controlled using Reed-Muller (RM) Channel Coding, $RM(8, 4, 4)$. The RM encoder breaks down the first stage tweak output into eight 4-bit sections and encodes them individually with their respective 8-bit RM error code. There are eight different instances of RM encoder sub-modules. This sub-module uses a series of XOR operations to encode each 4-bit section of the input word, as shown in Figure 3.6.

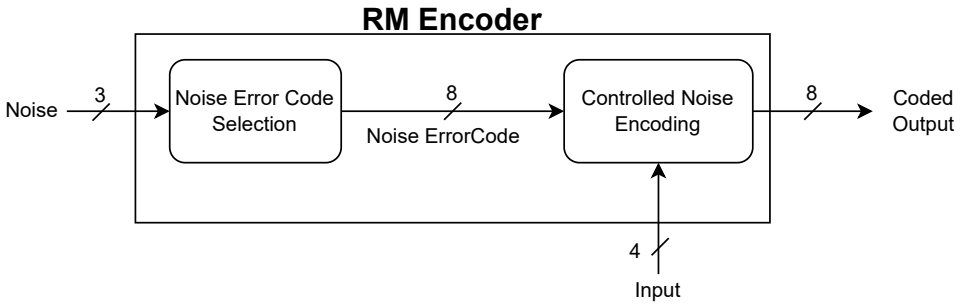


Figure 3.6: Reed-Muller Encoder Block Diagram

The noise is generated using a pseudo-random number from a bank of LFSRs. LFSRs are shift registers for pseudo-random number generation and pseudo-noise sequences through a linear function of previous register states. The LFSR bank implemented in the shielding module comprises 24 binary sequence generators, ranging from 29 to 32 flip-flops.

The 24-bit LFSR output is broken down into eight 3-bit sections, where each is used to select the noise error code used in the RM Channel Coding. This turns the noise generated by the LFSRs into the controlled 8-bit noise error code. Resulting

in the addition of a one-bit error per 8-bit output word. Table 3.1 shows the transformation from a 3-bit section of the LFSR output into the 8-bit RM noise error code.

LFSR Noise	RM Noise Error Code
3'b000	8'b10000000
3'b001	8'b01000000
3'b010	8'b00100000
3'b011	8'b00010000
3'b100	8'b00001000
3'b101	8'b00000100
3'b110	8'b00000010
3'b111	8'b00000001

Table 3.1: LFSR Noise to Reed-Muller Code Selection

The third stage of the XEX is the permutation element which DCD strengthens by using a lightweight block cipher called Prince [4]. The algorithm module is designed to have little impact on the system's hardware requirements while providing a high level of security. Prince is the permutation element of the XEX process. The key used for this module is fixed and generated by two accesses from the PUF as described in Section 3.2.1. This key is extended from 128 to 192 bits for additional whitening stages within Prince. The key breakdown is listed below:

- 128-bit key from PUF extended to 192-bits for:
  - 64-bit pre-whitening key.
  - 64-bit post-whitening key.
  - 64-bit key for Prince 12-round core.
- Key is fixed and extracted from PUF at reset or power cycle.

The four main stages of Prince are:

- Key Expansion: The key is expanded to 192 bits.
- Pre-Whitening: 64-bit input is XORed with a 64-bit sub-key.
- 12-Round Core: The main cipher encrypts the previous stage's output with a 64-bit sub-key.
- Post-Whitening: Output from the previous stage is XORed with a 64-bit sub-key.

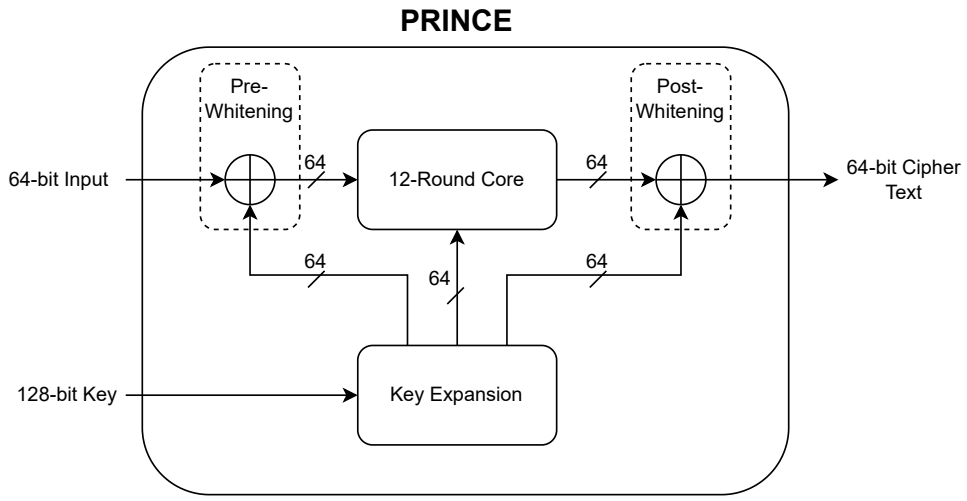


Figure 3.7: Prince Block Diagram

Between the pre-whitening and post-whitening stages, Prince implements a 12-round core permutation. A single round of encryption includes a key addition, an S-box layer, a matrix multiplication layer, and an addition of a round constant.

- Key Addition: This layer XORs the 64-bit input with a 64-bit sub-key of the extended original key.
- S-Box Layer: This layer performs byte substitutions using a predetermined 4-bit S-box.

- Matrix Multiplication Layer: This layer multiplies the 64-bit input with a 64 x 64 matrix.
- Round Constant: Each round has a unique 64-bit constant XORed with the 64-bit input to each layer.

Figure 3.8 shows the process of the 12-round Prince core and the breakdown of a single round.

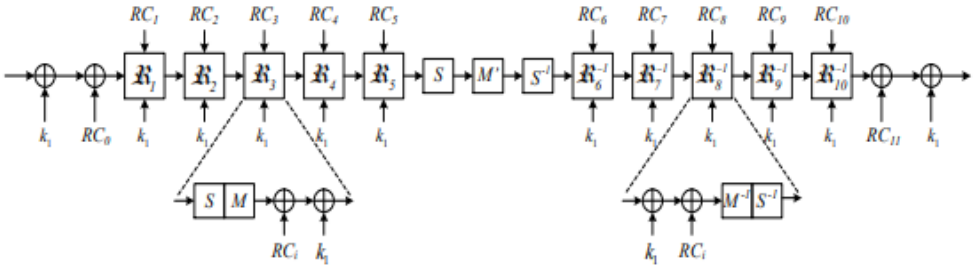


Figure 3.8: Prince Encryption Round Diagram [4]

Once the Prince encryption stage is complete, the final stage is the second tweak. This stage performs an XOR operation on the 64-bit cipher text and the 64-bit block tweak.

3.2.3 Hashing

Hashing is used to implement a layer of verification to ensure that an attacker cannot modify the program image or inject malicious code. The DCD shielding module implements the hash algorithm using GF multiplication. The operation is a multiplication between a base hash input and the output from the last tweaking stage. This method utilizes the same polynomial in the GF Double defined in Section 3.2.1. The multiplication uses 64 partial sums implemented with XOR operations. Currently, the hash table and comparison logic are not implemented yet. This is future work for the DCD computing system.

3.2.4 *Decoy Shuffling*

Another security element used in this system is decoy shuffling. This process is a continuous block fetch and write cycle that uses one cache line to shuffle randomly selected blocks from memory. This process runs continuously during both processor execution and stalls. This decoy shuffling protects against possible patterns that may occur when a program is running that accesses the same block multiple times. It also adds more variability of shielded instructions in the external main memory due to the repeated block tweaks every time a block is shielded. Figure 3.9 shows the flow that results in periodic and continuous shuffling independent of processor execution.

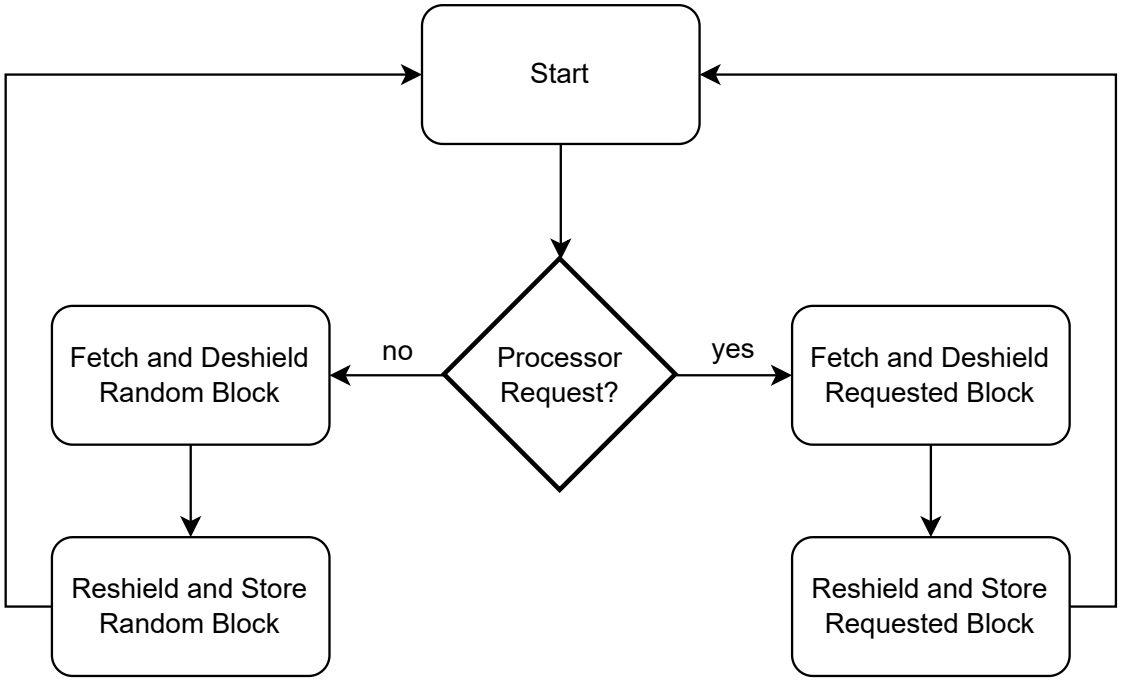


Figure 3.9: Decoy Shuffling Flowchart

3.3 RISC-V Processor

This processor is an implementation of a 32-bit RISC-V instruction set architecture (ISA) that uses a base integer instruction set and an extension for multiplication

and division instructions. The RISC-V processor is implemented in a 5-stage pipeline.

The five stages of the RISC-V pipeline consists of:

- Instruction Fetch: Reads the instruction cache and sends an instruction to the decoder.
- Instruction Decode: The instruction is decoded and sends control signals and inputs to the execution stage.
- Execution: All computation, such as arithmetic and bit shifting, is done in this stage. The output containing the computation result is forwarded to the next stage.
- Memory Access: If memory needs to be accessed, it is done in this stage and single-cycle instruction results are forwarded to the next stage.
- Write-back: The final stage where single-cycle and dual-cycle instruction results are stored in the register file.

Multi-cycle instructions such as multiply and divide are exceptions to the five-stage pipeline, and the processor stalls until the computation is complete. Figure 3.10 is a diagram of the implemented RISC-V pipeline.

During the instruction fetch stage, if there is a cache hit, an instruction is fetched from the instruction cache and sent to the processor for use. Each instruction is executed in the pipeline until the processor requests an instruction from a new block. When the processor requests an instruction not stored in the cache, a new block containing the instruction is fetched, de-shielded, and stored in the cache. When this happens, the processor stalls, and no new computations or functions occur.

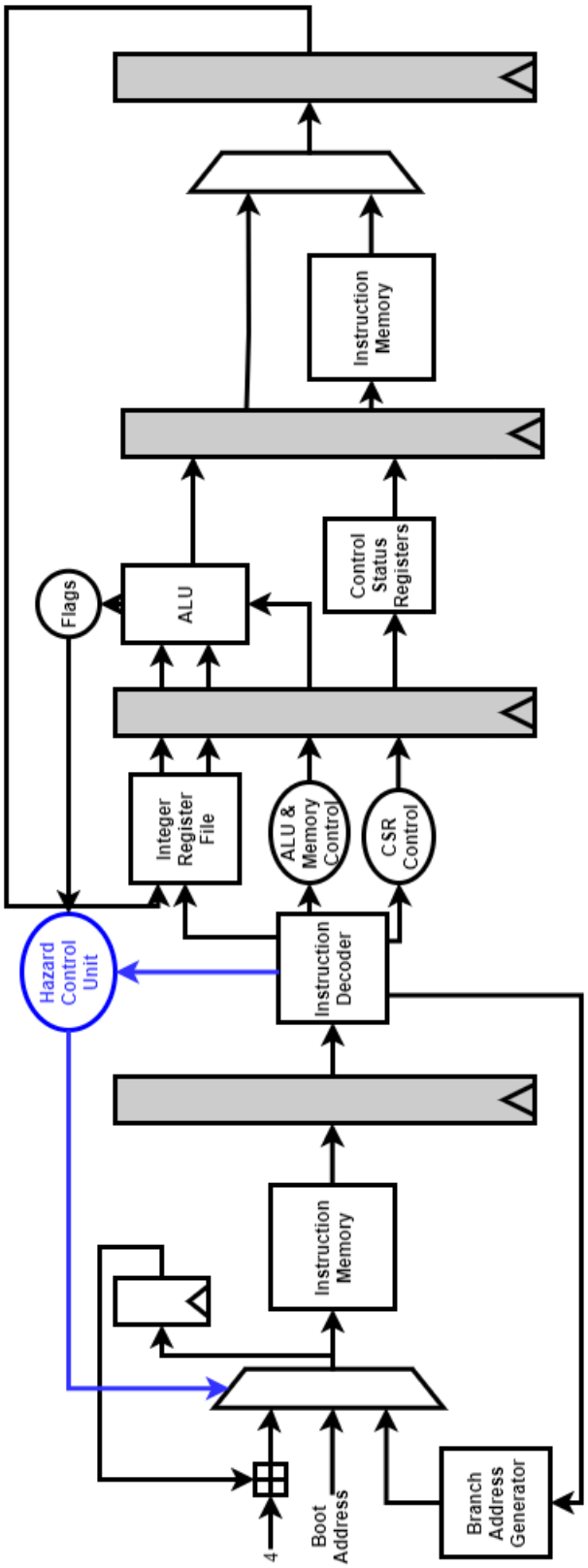


Figure 3.10: RISCV Processor 5-Stage Pipeline Block Diagram

Chapter 4

HARDWARE DESIGN

4.1 Design Goals

The main goal of this thesis is to implement fully functional data and instruction caches and test the performance of the DCD secure computing system. The goals for the design include adding a data cache, variable-size functionality for the cache modules, and a replacement policy. The different versions of the system include a 2, 4, 8, and 16-line cache design. Each cache line consists of 128 32-bit words. The goal of the replacement policy is to replace blocks that were unlikely to be used again and keep those that were. The policy chosen for the design was LRU, described in Section 2.3.1.

To meet the design criteria for the cache implementation and testing, modifications to the existing computing system had to be made. Each new module was integrated and tested thoroughly to ensure it did not interfere with processor execution or the shielding and block shuffling process.

4.2 Instruction Cache

4.2.1 Designing the Memory Modules

The first part of the cache design process was to implement a memory device in Verilog that is large enough to store one block of instructions. The goal was to reuse this module for each cache line. Each cache line needed to store 128 32-bit instructions. Each cache line also needed to be accessed by the main memory and the processor. The cache memory module also needed a second port for reads and

writes for the continuous block shuffling during processor execution. The final design after these constraints is a 512-byte Dual Port RAM, Figure 4.1, for each of the cache lines.

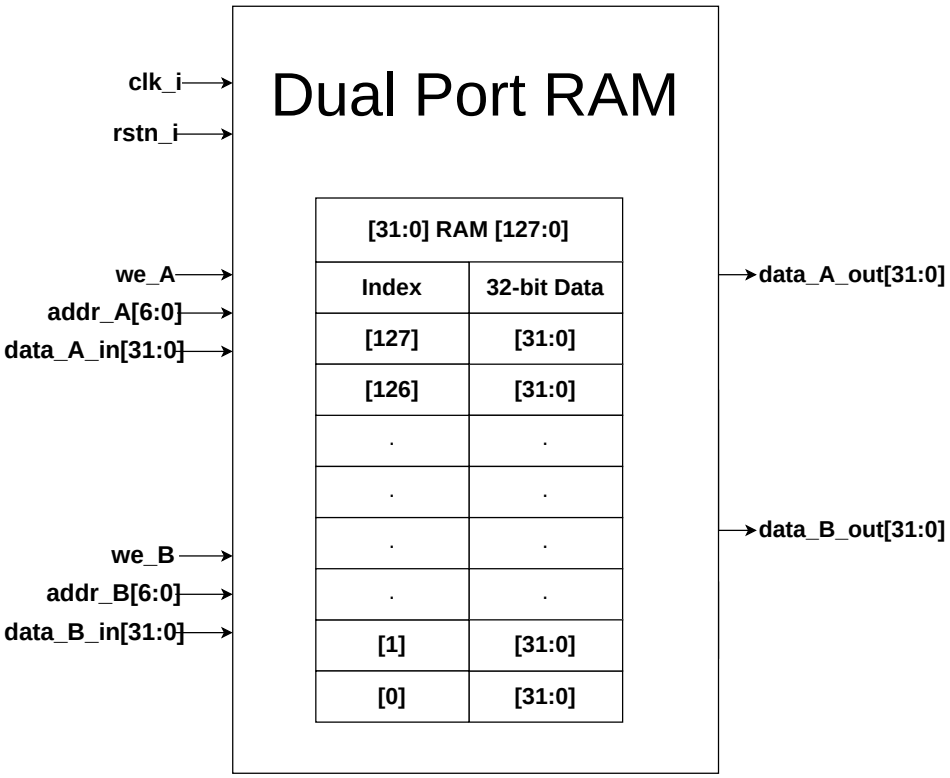


Figure 4.1: Dual Port RAM Block Diagram

The next step was to design another memory module that would store the tag bits for each block in the cache. This memory is variable based on the number of cache lines. The size of the tag is determined by the number of blocks that can be stored in the main memory. The tag size needed to address all 512 blocks in memory is 9 bits, Figure 4.2.

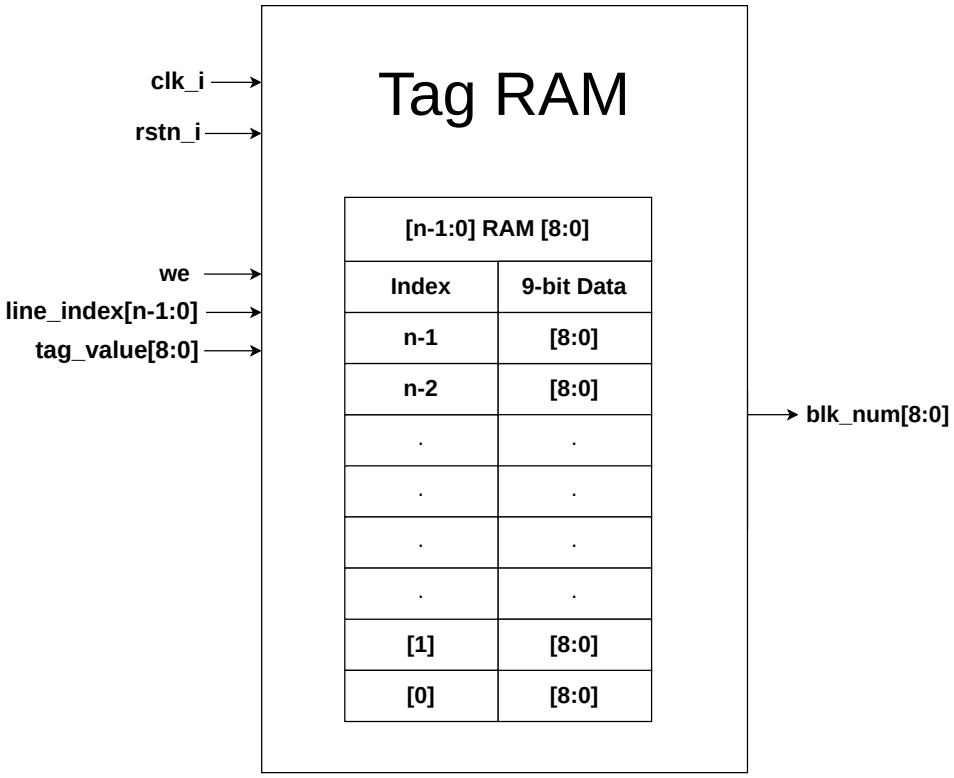


Figure 4.2: Tag RAM Block Diagram

4.2.2 Least Recently Used Module

One of the most important parts of the fetch state in the control logic is to consider when all of the cache lines are full. The first part of this logic is to decide which block to evict from the cache for the newly requested block. The LRU module uses a counter method where each cache line has a count assigned to it from zero to $n-1$, where n is equal to the number of cache lines. The least recently used line contains the value zero and is used by the cache control logic that the corresponding line is next to be replaced.

The counter implementation contains a register for each cache line. It also contains comparison logic to determine which registers need to be updated to fit the new order depending on which line has been accessed. This logic compares the order value of

each line to the value of the line that has been accessed. The circuit's operation depends on whether there is a cache miss or hit. When there is a cache hit, the LRU compares each value with the accessed line's value. If the value of a given line is larger than the accessed line's value, that value decrements by one. If the value is less than the accessed line's value, that value stays the same. The accessed line's value is then set to the greatest value, making it the most recently used line, and the line with the value of zero is the least recently used line. When there is a cache miss, the LRU module determines the least recent line that has been used and stores the new block in that line. Next, the LRU updates, which decrements every other line by one. This process makes the accessed line the highest value, the most recently used line, and the new line with the value zero is now the least recently used. Figure 4.3 shows the block diagram that this design is based on.

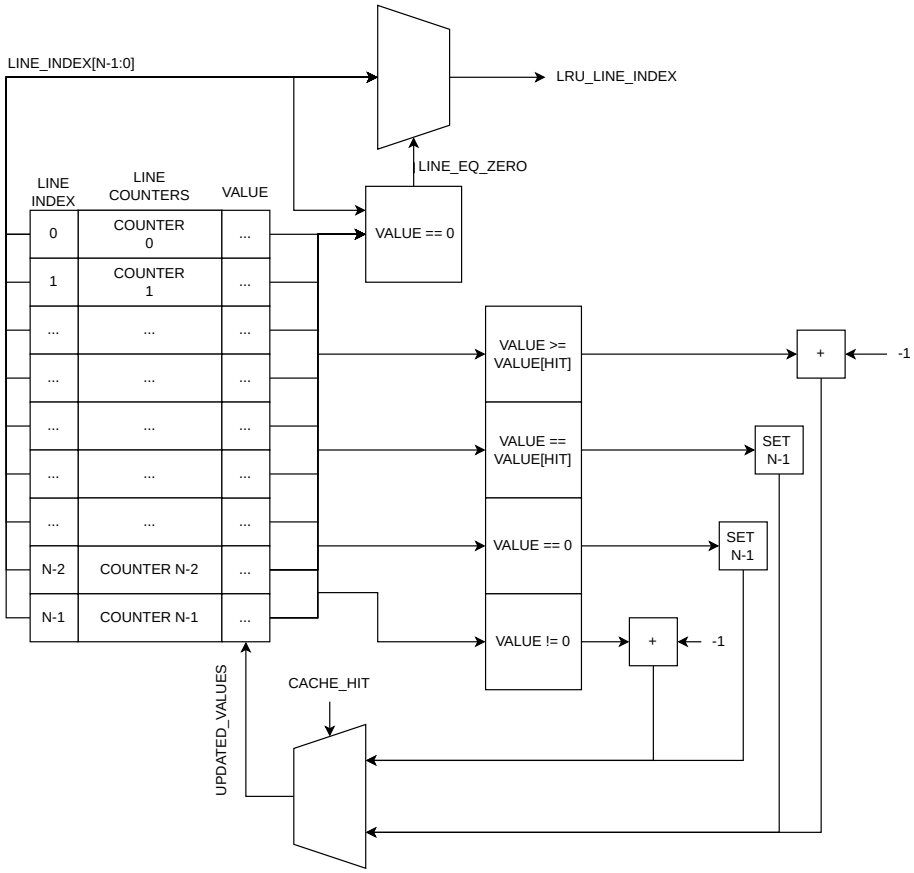


Figure 4.3: LRU Block Diagram

4.2.3 Designing the Control Logic

After the memory and LRU modules were designed, the next step was to design control logic for the instruction cache module. The control logic is a state machine, Figure 4.4, that sends busy and ready signals to the processor, telling the processor when it needs to stall. When a ready signal is sent to the processor, this informs it that the requested instruction is available. The cache control logic sends read and write signals to each cache line based on the line being accessed. The control module waits in an Idle state until the processor makes a request to the cache. Once the

processor makes a request, the logic decides if the requested block is in the cache module during the Compare Tag state. This also means the logic checks whether the tag address bits are stored in the tag memory module. When there is a cache hit, the requested instruction is read from the matching cache line, and the state machine returns to Idle. When there is a cache miss, the cache sends a busy signal to stall the processor execution and move into the next state. In addition to updating the state and control signals to the processor, this state also includes the update to the LRU module.

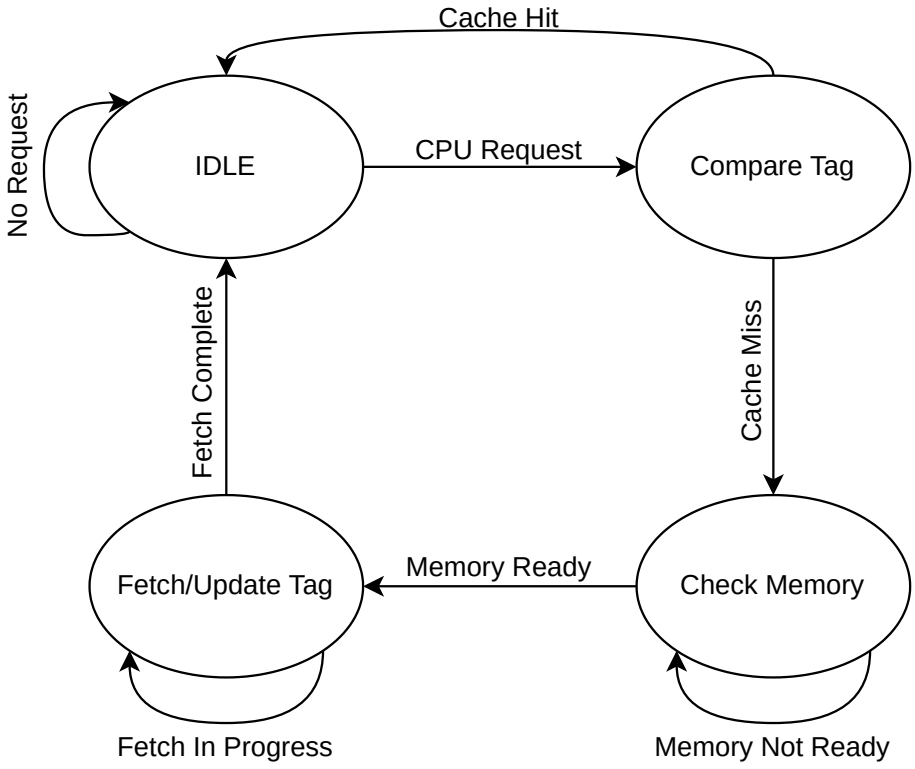


Figure 4.4: Instruction Cache State Diagram

Next, the cache state machine enters the Check Memory state, where the cache communicates with the main memory. This state waits for the main memory module to become available to read from. The final state of the cache control logic is the fetch

state. Here, the LRU line is sent a write-enable signal and fetches the requested block of instructions. Once the final instruction is fetched, the tag RAM is updated, and a ready signal is sent to the processor. This allows the processor to fetch the requested instruction and continue executing the program until a new request is made. The control logic then returns to the Idle state, awaiting the next CPU request. Figure 4.5 shows the state machine block diagram for the instruction cache.

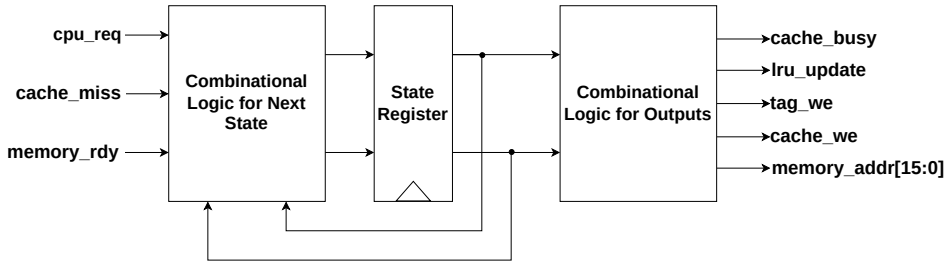


Figure 4.5: Instruction Cache State Machine

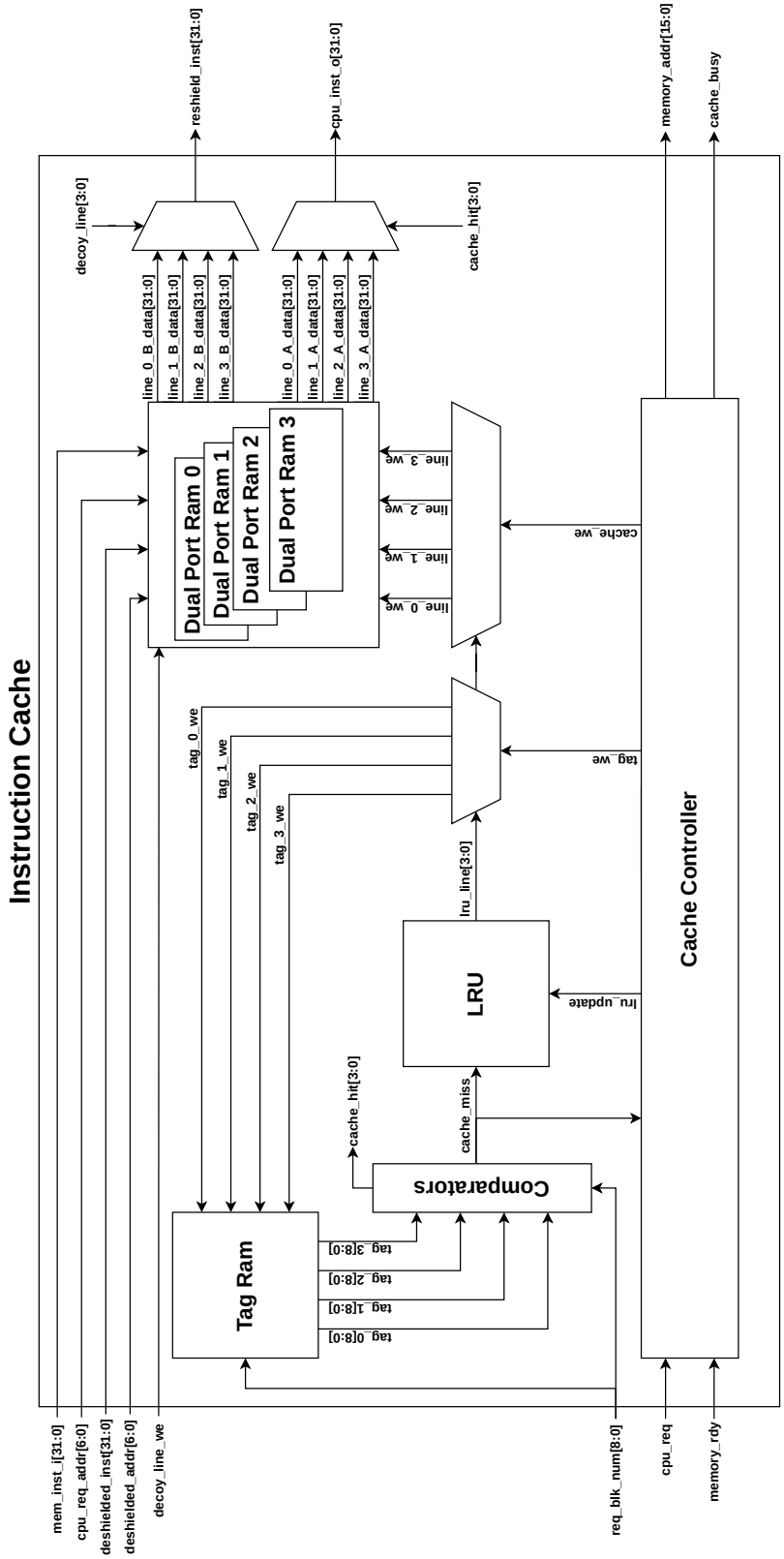


Figure 4.6: Instruction Cache Module

4.3 Data Cache

4.3.1 Reused Modules

The data cache reuses the memory modules and the LRU from the instruction cache design. The main difference in the data cache is the addition of memory write-back functionality. This resulted in modifying the cache control module and adding additional multiplexers. More modules needed to be modified and added to the complete computing system. These changes are covered in Section 4.4 on cache integration.

4.3.2 Control Logic Modifications

With adding the write-back functionality, the cache control state diagram was modified, Figure 4.7. The solution was an additional state that could be accessed if the replacement cache line was dirty. A dirty line in the cache is a line that has been modified from the original data currently stored in memory. At the end of the Check Memory state, the control logic checks if the cache line is dirty and waits until the memory is ready to move on. Once the memory-ready signal is received, the controller enters one of two states. The first state is the fetch state, where the block is fetched and the tag is updated. This state takes place when the replacement line is not dirty. The second state is the Writeback state, where the block in the replacement line is written to the main memory. This state is entered if the line is dirty; at the end, the dirty flag for that line is set back to zero. After the write-back is complete, the controller then enters the fetch state. At this point, the data cache controller returns to the same operations in the instruction cache controller.

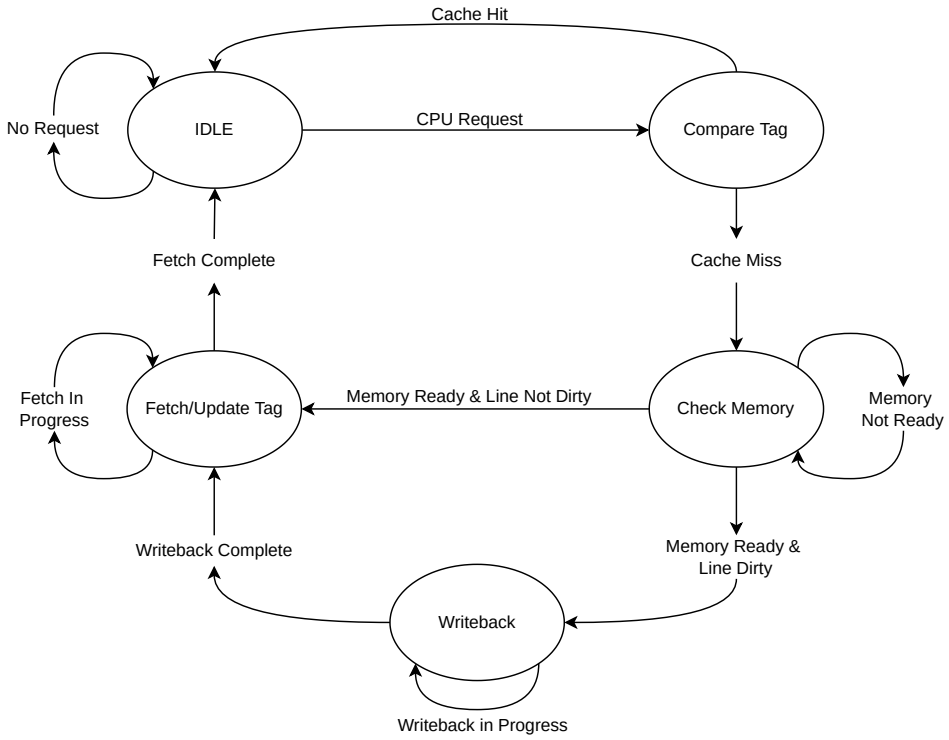


Figure 4.7: Data Cache State Diagram

The data cache controller has five different states:

- IDLE: 000
- Compare Tag: 001
- Check Memory: 010
- Fetch/Update Tag: 011
- Write-back: 100

Additional inputs and outputs are also added to the controller. The new input added is the dirty line signal which communicates to the controller that the replacement line is dirty. Two new outputs are added: a memory write-enable signal for the

write-back process, and an update signal that resets the replacement line's respective dirty flag back to zero. Figure 4.5 shows the state machine block diagram for the data cache.

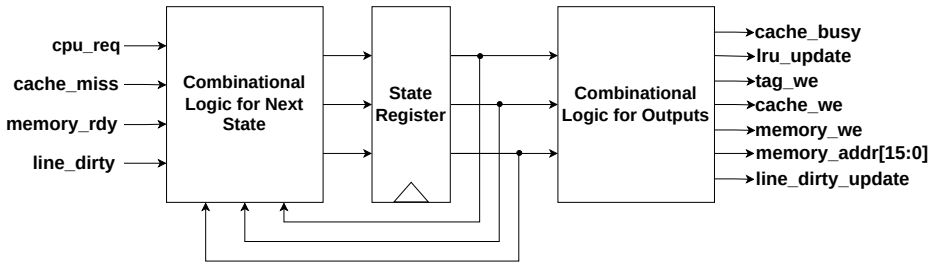


Figure 4.8: Data Cache State Machine

4.3.3 Additional Hardware

The data cache adds hardware to provide the write-back functionality. A register is implemented for each cache line for the new dirty cache line value. These registers update whenever there is a write from the processor, or the new control update signal is active. Additionally, a new multiplexer was added to select which register to update. The selection input to the multiplexer is the current LRU line index.

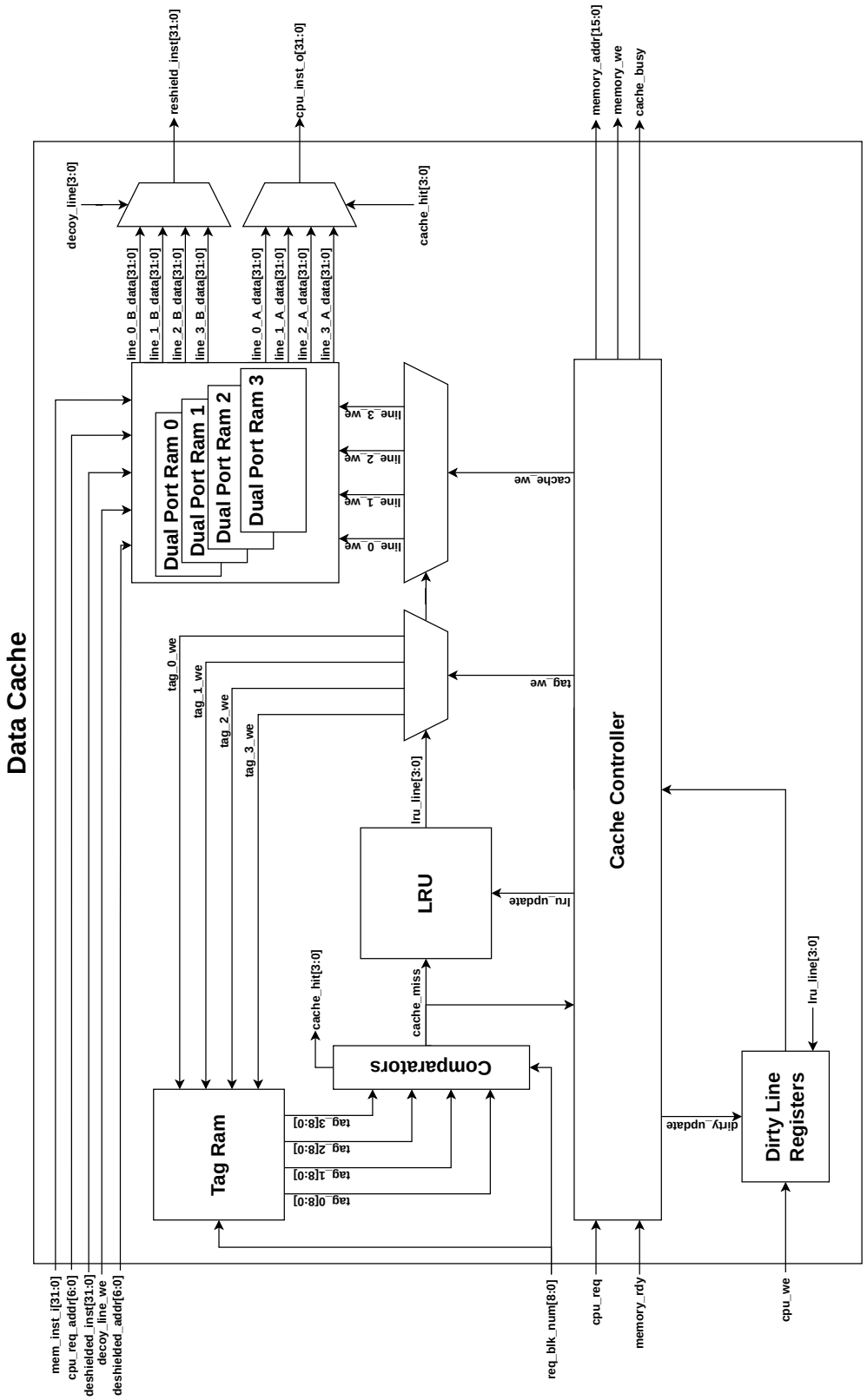


Figure 4.9: Data Cache Module

4.4 Testing and Integration

The next step of the hardware design process was fully integrating the caches into the DCD Secure Computing System. The easiest way to start this process was to begin the integration with a simpler system version that did not contain the shielding modules. The random block fetch and write cycles would make integration into the final system more complex. This led to a much easier time testing whether the finalized cache design would correctly connect to each part independently. The newly created system with only the cache, main memory, and processor also provides an advantage where the effect of the shielding module on performance can be quantified. This part is shown later in Chapter 5.

4.4.1 System Version: No Shielding

The integration for the system version with no shielding followed the steps listed below:

- Instruction Cache and Memory Functional Testing.
- Instruction Cache, Memory, and Processor Functional Testing.
- Modifications to Memory for Write-back.
- Modifications to Processor.
- Data Cache and Memory Functional Testing.
- Data Cache, Memory, and Processor Functional Testing.

Step one of integration was to connect and test the path between the cache and main memory. The transmitted instructions were checked during the fetch stage of operation to ensure that the data was being fetched correctly from memory. A test

bench was created to simulate processor requests and check for instruction errors when the cache state diagram begins to fetch. Since each block contains 128 instructions, there are 128 instruction checks during a single block fetch. Every time the cache would increment the requested address, the test bench would generate the correct instruction and compare it with the instruction that was fetched. The correct instruction was generated from an external memory file loaded with a test program. A counter that tracked errors would increase if the two values did not match.

Once the instruction cache and memory were working with no errors, the next step was to add the processor and test functionality. The previous test bench was altered, and the simulated processor requests were no longer needed. The test bench would run the same test program, and the results in memory were analyzed. During program execution, the test bench keeps track of the processor's stall and execution states to ensure they happen during the appropriate times. At the end of program execution, the results stored in memory are compared with the expected values.

Before the data cache was added to the functionality test, additional changes were made to the main memory module and the processor. In the initial system design, the data memory module was only connected to the processor and did not shield the data. The un-shielded data in the data memory created a vulnerability for the system. For this reason, the data cache and another shielding module were implemented. An additional interface was added to control the 128 fetches and writes during the data cache controller Fetch and Writeback states. This part was simple to implement due to the module already in place for the instruction cache. The only change that needed to be made was adding another state corresponding to the write-back stage. The new state was a copy of the Write state for the re-shielded data that would be shuffled in and out of the shielding module. The new design contains two different write states, one for shuffled data and one for write-back data. For the case of this system version

with no shielding, the shielding write state is unused.

The first change made to the processor was a connection to the data cache module instead of the data memory. The second change was to change the stall control signal to trigger when there was a data cache miss in addition to the instruction cache miss already in place. After these changes were made, all new modules were added to the test bench and tested to ensure that the introduction of the data cache did not interfere with processor or shielding functionality. The additions to the test bench included data cache checks for every read and write, processor stall and execution state checks, and error checking when a write-back would occur.

4.4.2 System Version: Shielding

The next step in the hardware design process was to integrate the shielding modules. This system version requires an additional shielding module for the data cache. This design was chosen to avoid cases where both caches need to use the module simultaneously. With the way the instruction and data caches were designed, the shielding modules could easily be placed between the main memory and the caches. The shielding module can communicate with each main memory interface in the same way as the original system design. The only changes that needed to be made were a few additional connections to the caches. Instead of the memory-ready signal that triggers the controller to move on from the Check Memory state, a new ready signal was implemented for the shielding process. This is due to the shielding module separating the connection between memory and the cache. This signal triggers whenever the first shielded instruction/data value is ready for each block fetch. The memory address signal connects the same; the only change is that it is sent to the shielding module and then forwarded to the main memory with a few cycle delays.

The shielding controller also needed the ability to allocate one of the cache lines

as a decoy line for the block shuffling functionality. This change reduced the number of available cache lines for normal fetches and writes by one. New states for fetches and writes were also implemented in each control module for this additional type of access. These states would be functionally the same; however, the same line and tag would always be updated, and it would not affect the LRU line index.

In the case of the data cache shielding module, much like the additional state to the data memory interface, a similar state was added to the shielding control module. This additional state determines whether a write was due to the block shuffling process or a write-back request from the cache controller. A write from the decoy line of the data cache to the main memory does not affect the data cache's dirty line flag signals.

Once all of these changes were made to the system, the tests from the previous system version test bench were run. The only change made to this system's test bench was the memory file used to test for instruction and data comparison. These files were pre-shielded by an external shielding module before the test bench was run.

4.5 FPGA Implementation

All system designs are created in Verilog and synthesized, implemented, and programmed onto an FPGA using Vivado. The FPGA utilized for this project is the Nexys A7 100-T Development Board. The software and hardware design steps for the DCD system are:

- Step 1: Develop a C file for the program image.
- Step 2: Compile the program with the RISC-V GCC compiler.
- Step 3: The compiler creates a .elf executable file.
- Step 4: Use elf2hex to turn the executable into hex instructions as the program image.

- Step 5: Use the secure boot emulator to pre-shield the program image before the FPGA is programmed.
- Step 6: The secure boot emulator outputs memory files for the shielded program image and the emulated PUF ROM.
- Step 7: Synthesize design in Vivado.
- Step 8: Implement the design in Vivado.
- Step 9: Vivado generates an FPGA bit-stream configuration file.
- Step 10: Program FPGA with the configuration file.

Figure 4.10 shows a diagram of the software-to-hardware design flow. After the FPGA is programmed with the configuration file, the secure computing system executes the program image until it is powered off. Once the FPGA is powered off, it must be reprogrammed with the configuration file to run again.

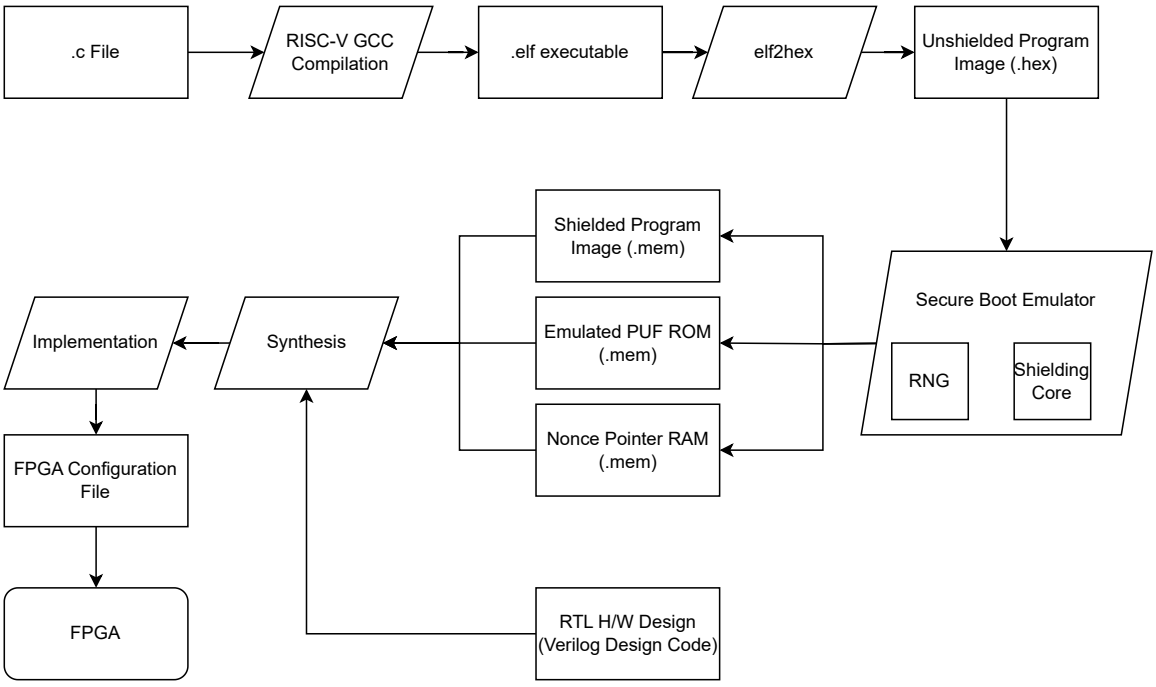


Figure 4.10: DCD Software to Hardware Design Flow Diagram

Chapter 5

PERFORMANCE

5.1 Tests

The results collected are generated from tests that record parameters necessary to evaluate the hardware cost, security, and computation performance. From these results, we make conclusions about security versus performance, cache size versus performance, cache size versus security, and optimizing system parameters. The set of tests performed are:

- Benchmark Program Tests:
  - Tested Parameters: Execution time, miss rate, write-back rate.
  - Modules Tested: All 24 combinations of system versions.
  - Total Number of Tests: 24 system versions x 5 benchmarks = 120 tests.
- Hardware Cost Analysis:
  - Tested Parameters: Area (K-Gate Equivalence) and FPGA Resource Usage.
  - Modules Tested: All components in the DCD top-level design.
- DieHarder Test Suite:
  - Tested Parameter: Perceived Randomness.
  - Modules Tested: The Four-Stages of Shielding plus a control program file.
  - Total Number of Tests: 5 module combinations x 26 DieHarder tests = 130 tests.

5.2 Computation Performance

Computation performance was measured by analyzing program execution times. A standard set of benchmark programs was developed to ensure the system was tested with a wide range of functionality. The goal was to create programs that implemented multiple computation attributes and memory access patterns. The system versions tested in this section are the 24 possible combinations that vary based on the following parameters:

- Shielding or Non-Shielding.
- Number of Instruction Cache Lines: 2, 4, 8.
- Number of Data Cache Lines: 2, 4, 8, 16.

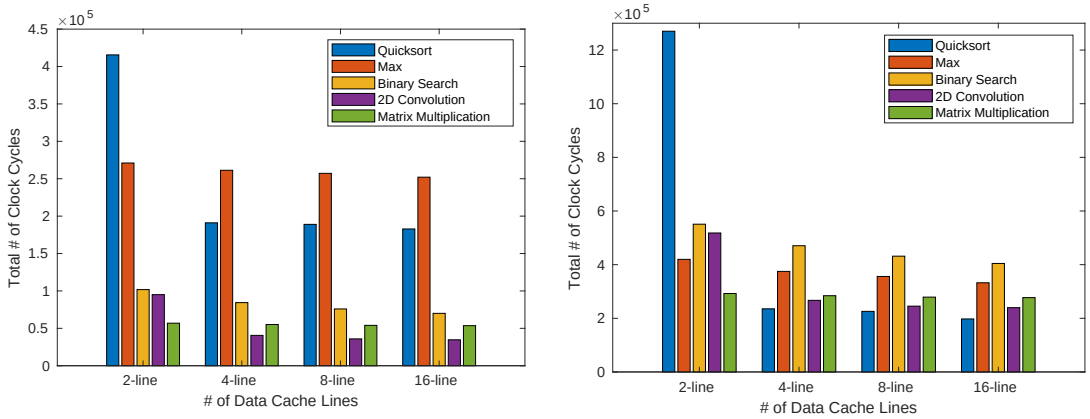
Five benchmark programs were run on each system version. The C code for the benchmark programs can be in found in Appendix A. This data set includes a total of 120 individual tests. Each test result is a measure of execution time in clock cycles. Other parameters collected and analyzed from these tests are the number of cache misses and the number of misses that resulted in a memory write-back. The size of each block is 128 words. Table 5.1 displays each benchmark program's number of instructions, memory blocks, and values.

Benchmark	# of Instructions	# of Blocks	Data Size
Quicksort	584	5	128 Values
Binary Search	564	5	10000 Values
Matrix Multiplication	1351	10	26x26 Matrices
Max	546	5	10000 Values
2D Convolution	1292	10	26x26 Matrices

Table 5.1: Benchmark Program Parameters

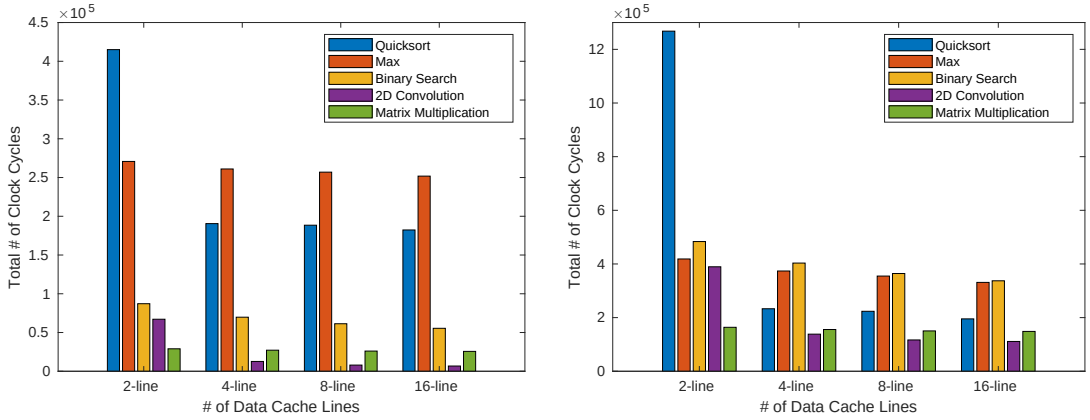
5.2.1 *Execution Time*

After each benchmark program run, the number of clock cycles was recorded. This data was compiled and visualized in MATLAB. Figures 5.1, 5.2, and 5.3 show the bar graphs generated.



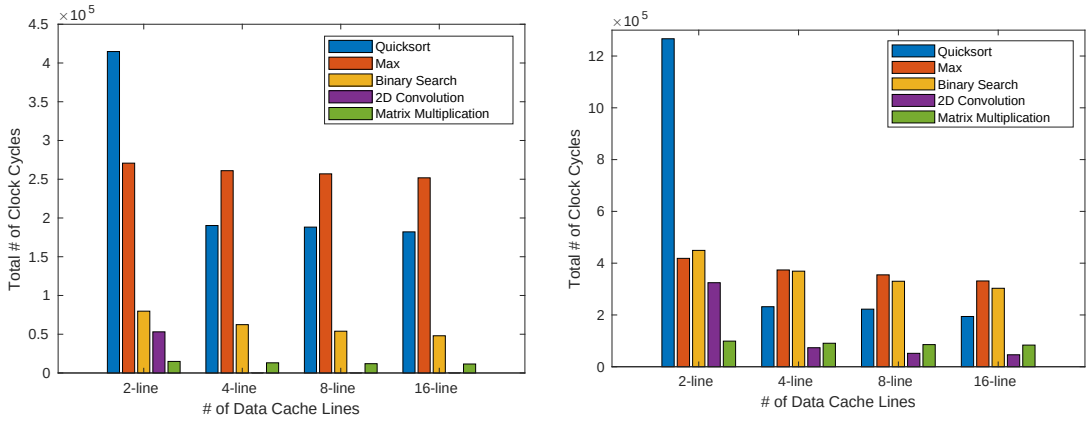
(a) Without Shielding (b) With Shielding

Figure 5.1: Benchmark Execution Times for 2 Instruction Cache Lines



(a) Without Shielding (b) With Shielding

Figure 5.2: Benchmark Execution Times for 4 Instruction Cache Lines



(a) Without Shielding

(b) With Shielding

Figure 5.3: Benchmark Execution Times for 8 Instruction Cache Lines

5.2.2 Cache Rates

The next step in analyzing computation performance was to collect information about cache miss rates and penalties. These values were collected from the FPGA during each benchmark program test. This was done by mapping wires from the Verilog design to output pins on the FPGA. The parameters collected include:

- Number of Cache Misses.
- Number of Cache Write-backs.
- Number of Memory Accesses.
- Shield/De-Shield Time.
- Fetch/Write Time.

The first parameter is the number of cache misses. This data was collected from running the benchmark programs on 12 system versions. The versions varied based on the number of instruction cache lines (IC): 2, 4, 8, and the number of data cache lines (DC): 2, 4, 8, 16. There is no difference in the number of cache misses between the

shielding and non-shielding versions of the system. The cache miss rate is independent of shielding. Table 5.2 shows the number of cache misses for each combination of instruction and data cache sizes.

Number of Cache Misses												
	System Version											
	2 IC				4 IC				8 IC			
Benchmark	2 DC	4 DC	8 DC	16 DC	2 DC	4 DC	8 DC	16 DC	2 DC	4 DC	8 DC	16 DC
Quicksort	651	31	25	18	649	29	23	16	648	28	17	15
Binary Search	139	131	129	126	82	74	72	69	53	45	43	40
Matrix Mult	112342	66758	64568	58726	82987	37403	35213	29371	68304	22720	20530	14688
Max	119	100	90	68	118	99	89	67	118	99	89	67
2D Conv	54758	38374	37844	37839	35841	19547	18927	18922	26382	9998	9468	9463

Table 5.2: Number of Cache Misses for Each Benchmark Program

The next parameter is the number of cache write-backs. Cache write-backs occur when a block in the cache needs to be replaced, but the line storing the block is dirty. The block in the cache line must be updated in memory before the new block can be stored. This parameter varies based on the number of data cache lines only. Table 5.3 shows the number of write-backs for each data cache line size.

Number of Cache Write-Backs				
	System Version			
Benchmark	2 DC	4 DC	8 DC	16 DC
Quicksort	274	17	15	0
Binary Search	11	9	9	9
Matrix Mult	18553	2287	1274	11
Max	42	23	17	17
2D Conv	6284	221	10	5

Table 5.3: Number of Data Cache Writebacks for Each Benchmark Program

The final parameter for this stage is the number of memory accesses. Memory accesses occur when the processor requests data or instructions or writes data to one of the cache lines. This parameter is useful for calculating the cache miss and write-back rates. Table 5.4 displays the number of memory accesses for each benchmark

program.

Number of Memory Accesses	
Benchmark	Accesses
Quicksort	44592
Binary Search	686
Matrix Mult	239447
Max	60009
2D Conv	90311

Table 5.4: Number of Memory Accesses for Each Benchmark Program

The next stage was to calculate the miss rate and penalty. Cache miss rate is a function of the number of cache misses. This analysis also defines a cache write-back rate. Separating the number of write-backs from cache misses helps build a more accurate model. This is due to the large block size and the significant time it takes to shield and de-shield. A cache miss that includes a memory write-back doubles the miss penalty. The miss and write-back rates are calculated as percentages. These rates are defined as:

- Miss Rate % $= \frac{\text{\# of Misses}}{\text{\# of Accesses}} * 100$
- Write-Back Rate % $= \frac{\text{\# of Write-Backs}}{\text{\# of Accesses}} * 100$

The data from Tables 5.2 and 5.3 were used for this calculation. The miss and write-back rates for each benchmark test are shown in Tables 5.5 and 5.6.

Cache Miss Rate (%)												
	System Version											
	2 IC				4 IC				8 IC			
Benchmark	2 DC	4 DC	8 DC	16 DC	2 DC	4 DC	8 DC	16 DC	2 DC	4 DC	8 DC	16 DC
Quicksort	1.46	0.07	0.06	0.04	1.46	0.07	0.05	0.04	1.45	0.06	0.04	0.03
Binary Search	20.3	19.1	18.8	18.4	12.0	10.8	10.5	10.1	7.73	6.56	6.27	5.83
Matrix Mult	46.9	27.9	27.0	24.5	34.7	15.6	14.7	12.3	28.5	9.49	8.57	6.13
Max	0.20	0.17	0.15	0.11	0.20	0.17	0.15	0.11	0.20	0.17	0.15	0.11
2D Conv	60.6	42.5	41.9	41.9	39.7	21.6	21.0	20.9	29.2	11.1	10.5	10.4
Average	25.9	17.9	17.6	17.0	17.6	9.66	9.27	8.69	13.4	5.50	5.10	4.52

Table 5.5: Cache Miss Rate for Each Benchmark Program

Cache Write-Back Rate (%)				
	System Version			
Benchmark	2 DC	4 DC	8 DC	16 DC
Quicksort	0.61	0.04	0.03	0.00
Binary Search	1.60	0.01	0.01	0.01
Matrix Mult	7.75	0.96	0.53	0.01
Max	0.07	0.04	0.03	0.03
2D Conv	6.96	0.24	0.01	0.01
Average	3.40	0.26	0.12	0.01

Table 5.6: Cache Write-Back Rate for Each Benchmark Program

To show the effect of cache size on miss rate, averages for each of the five benchmarks were calculated for each system version. As the number of cache lines increases, the miss rate decreases. Figures 5.4 and 5.5 display that the miss rate decreases as the number of cache lines increases. More analysis of this data is covered in Chapter 6.

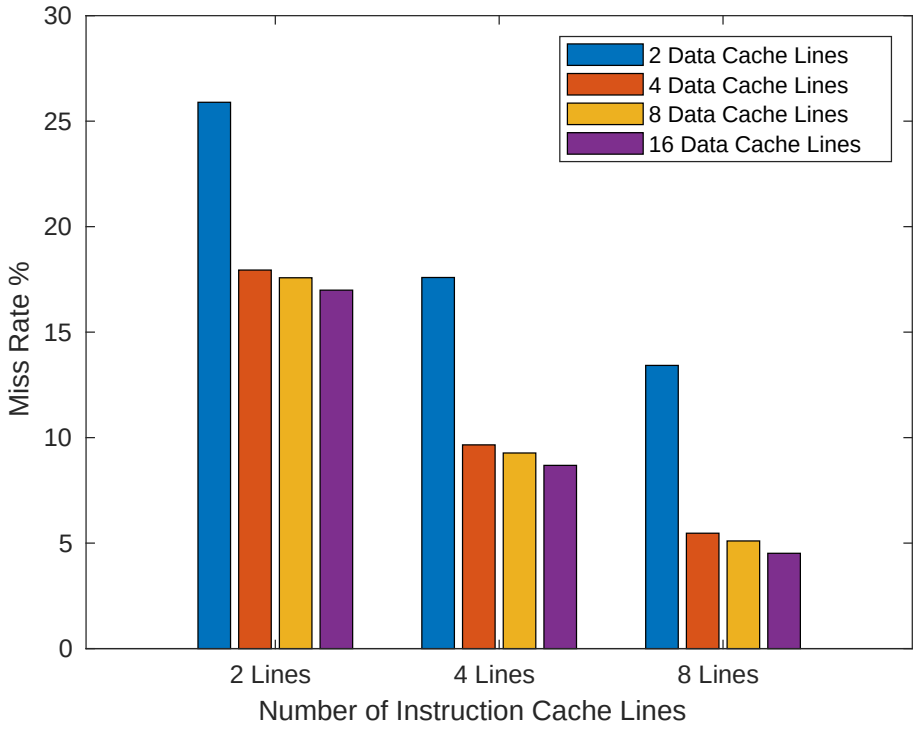


Figure 5.4: Average Miss Rates of all Benchmark Programs

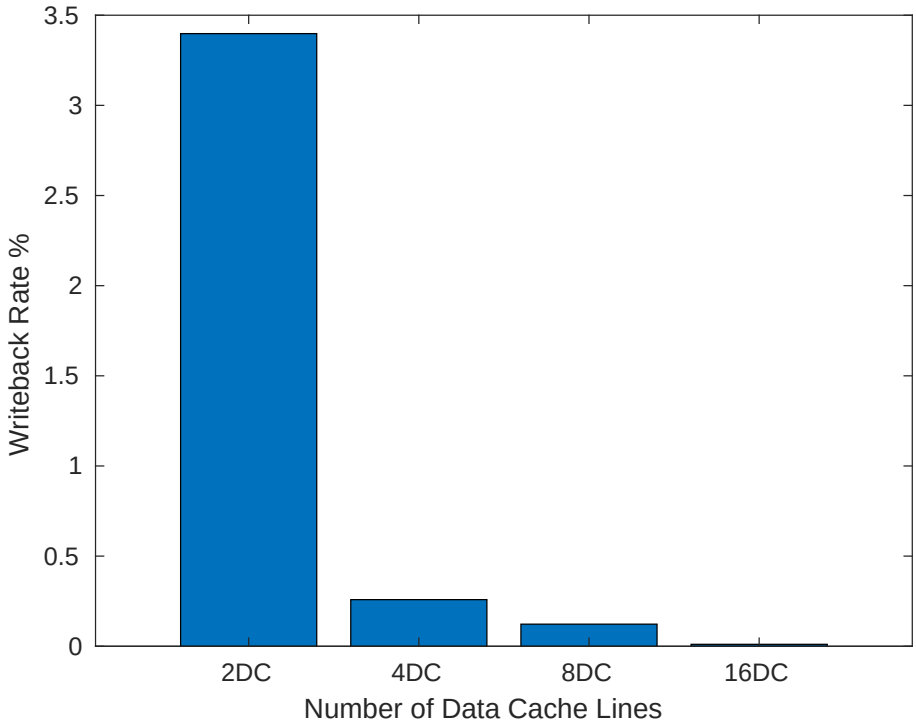


Figure 5.5: Average Write-Back Rates of all Benchmark Programs

5.3 Hardware Cost

This section quantifies hardware cost by analyzing kilo-gate equivalence (KGE), an estimate of the on-chip area for the system design. The KGE test uses the open-source Verilog synthesis tool, Yosys [27]. For the FPGA implementation of the system, the resource usage parameters include the number of look-up tables (LUT), flip-flops (FF), block RAM (BRAM), and I/O.

To calculate the overall KGE of the system, the system was divided into multiple modules. The breakdown of the modules includes the RISC-V processor, both shielding modules, the system controller, memory control logic, UART, and peripherals. This test analyzes the difference between the shielding and non-shielding versions of the system. Tables 5.7 and 5.8 display the KGE for each module.

Module	KGE
RISC-V Processor	19.7
Shielding Modules	43.2
System Controller	0.8
Memory Control Logic	1.2
UART	10.3
Peripherals	1
Total	76.2

Table 5.7: KGE: Shielding

Module	KGE
RISC-V Processor	19.7
System Controller	0.8
Memory Control Logic	1.2
UART	10.3
Peripherals	1
Total	33

Table 5.8: KGE: No Shielding

The first step in analyzing the FPGA resource use was synthesizing and implementing each system version in Vivado. The top-level design was run for the shielding (Sh) and non-shielding (NoSh) versions for this analysis, Figure 5.9. Separately, each instruction and data cache module was tested, Figures 5.10 and 5.11. The Vivado project summary reports the FPGA's resource utilization.

Number of Resources Utilized		
	System Version	
Resource	NoSh	Sh
LUT	1606	2886
FF	1661	1817
BRAM	13	17
IO	20	20

Table 5.9: FPGA Resource Utilization with Respect to Shielding

Number of Resources Utilized			
	# of IC Lines		
Resource	2 IC	4 IC	8 IC
LUT	35	40	80
FF	16	28	51
BRAM	3	5	9

Table 5.10: FPGA Resource Utilization for the Number of Instruction Cache Lines

Number of Resources Utilized				
	# of DC Lines			
Resource	2 DC	4 DC	8 DC	16 DC
LUT	36	41	81	138
FF	18	32	59	116
BRAM	3	5	9	17

Table 5.11: FPGA Resource Utilization for the Number of Data Cache Lines

5.4 Security Performance

For evaluating the level of security provided by the different cryptography techniques used by the system, a test of perceived randomness was applied. Individual parts of the shielding process were evaluated to test the secure computing system. Individually testing each stage of the shielding system allowed us to determine which modules provided the greatest amount of randomness. The randomness level of each part of the system was analyzed along with other results from the computation and hardware cost testing. This allowed us to determine the cost versus security performance. From these collective results, conclusions can be drawn about the viability of the shielding module in comparison to the associated performance decrease.

Statistical test suites, such as the National Institute of Standards and Technology (NIST) test suite, are common in randomness testing [19]. However, the common issue with the NIST test suite, and others like it, is that it does not accurately meet the needs of a cryptographic system. A cryptographic system is more deterministic than a random number generator, and these tests could reject a secure system. Due to this, other test suites that do not completely rely on raw statistical randomness were considered.

5.4.1 Dieharder Test Suite

The test suite used to determine the security measure of the shielding module is the DieHarder test suite [5]. This test suite expands the DieHard randomness test suite [16]. DieHarder is intended to test random-number generator output but also includes the ability to test files of random numbers. Testing raw data files is not the intended method but can be used to create a baseline. The optimal way of using the test suite would be to replicate the shielding module stages in C programs. These programs should produce results identical to the hardware implementation and be able to interface with DieHarder directly.

In this thesis, we use the unintended way of collecting and storing raw output data in files. This data was collected from each stage of the shielding process. The results from these tests should not be considered the final measure of the system's security. Creating C programs and performing an optimal system evaluation is future work for the project.

The data used in these tests was collected by running example program instructions through each stage of the shielding module. One problem with running the DieHarder test suite on files of random numbers is collecting enough data to perform the tests accurately. As this way is unintended, no minimum data size is specified. In the DieHarder documentation, it appears that at least 32 million bits are needed for all tests to run. To meet this requirement and give the test suite more data to analyze, we generate 24MByte files for each stage. Below is a list of the shielding stages that were tested:

- Control: Plain-Text.
- Stage 1: First Tweak.

- Stage 2: Channel Coding and Controlled Noise Injection.
- Stage 3: Prince Encryption.
- Stage 4: Second Tweak.

In addition to each stage of the shielding module, a control test was added. This test analyzes the un-shielded plain-text. When the DieHarder suite completes testing, it displays a p-value that determines whether the test failed or succeeded based on a predetermined range. The default DieHarder p-value ranges are:

- Failure: [0:p-value:0.1] or [0.9:p-value:1]
- Weak: [0.1:p-value:0.25] or [0.75:p-value:0.9]
- Pass: [0.25:p-value:0.75]

Table 5.12 displays the results from all 26 DieHarder tests at each stage of the shielding module.

Test Name	Plain Text	Stage 1	Stage 2	Stage 3	Stage 4
Birthday Spacings	Fail	Pass	Fail	Pass	Pass
Overlapping Permutations	Fail	Fail	Fail	Fail	Fail
Ranks of Matrices(32x32)	Fail	Weak	Fail	Pass	Weak
Ranks of Matrices(6x8)	Fail	Fail	Weak	Fail	Fail
Bitstream	Fail	Fail	Fail	Fail	Fail
OPSO	Fail	Fail	Fail	Fail	Fail
OOSO	Fail	Fail	Fail	Weak	Fail
DNA	Fail	Fail	Fail	Pass	Pass
Count the 1s (Str)	Fail	Fail	Fail	Pass	Pass
Count the 1s (Byte)	Fail	Fail	Fail	Weak	Pass
Parking Lot	Fail	Fail	Fail	Fail	Fail
2D Circle	Fail	Fail	Fail	Pass	Pass
2D Sphere	Fail	Fail	Fail	Pass	Pass
Squeeze	Fail	Fail	Fail	Fail	Fail
Overlapping Sums	Fail	Pass	Pass	Pass	Weak
Runs 1	Fail	Fail	Fail	Pass	Pass
Runs 2	Fail	Fail	Fail	Pass	Pass
Craps 1	Fail	Fail	Weak	Pass	Pass
Craps 2	Fail	Fail	Fail	Weak	Weak
STS Monobit	Fail	Fail	Fail	Fail	Fail
STS Runs	Fail	Fail	Fail	Fail	Fail
STS Serial	Fail	Fail	Fail	Fail	Weak
RGB Bit Distribution	Fail	Fail	Fail	Weak	Pass
RGB Minimum Distance	Fail	Fail	Fail	Weak	Pass
RGB Permutations	Fail	Fail	Fail	Fail	Pass
RGB Lagged Sum	Fail	Fail	Fail	Fail	Fail
RGB KS	Fail	Fail	Fail	Fail	Weak
Total Tests Passed	0	2	1	10	13
Total Tests Weak	0	1	3	5	4
Total Tests Failed	26	23	22	11	9

Table 5.12: Dieharder Results for the Shielding Module Stages

As each stage of the shielding module is added, the number of tests passed increases, with stage two as an exception. More analysis is needed for the perceived randomness of the shielding module. This will be done in future work when each stage is replicated as a C program.

Chapter 6

OPTIMIZATION RESULTS

6.1 Performance

The goal of performance testing was to find an optimization point where hardware cost, computation, and security meet. From the results collected in Chapter 5, comparisons are made between the tests. The optimization analysis covers the following:

- Miss Rate vs. Data Cache Size.
- Miss Rate vs. Instruction Cache Size.
- Average Memory Access Time.
- Hardware Cost vs. Cache Performance.
- Hardware Cost vs. Shielding Presence.

6.1.1 Miss Rates

To optimize the cache size parameters, the average miss rate for the number of instruction and data cache lines was analyzed. This helps determine the effectiveness of increasing each cache size. Figure 6.1 shows the average miss rate of the benchmark programs versus the number of data cache lines. The data was separated into a set for each number of instruction cache lines.

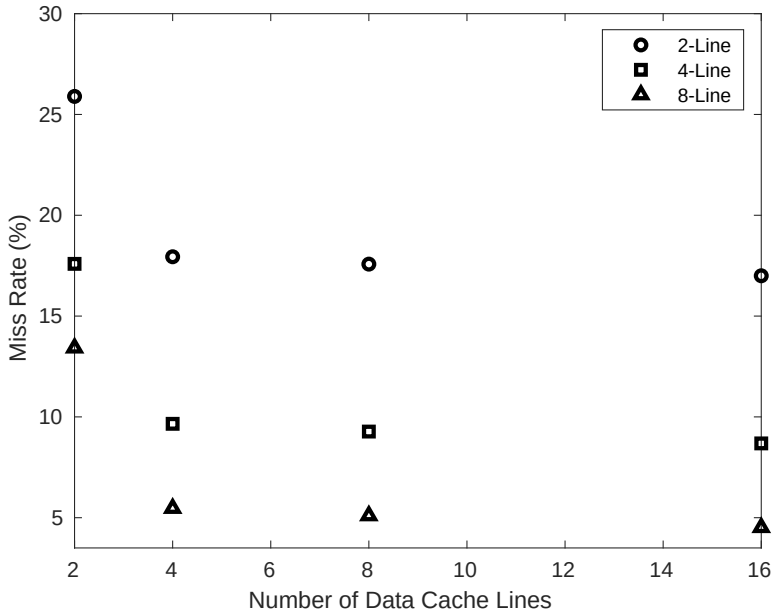


Figure 6.1: Average Miss Rate as a Function of the Number of Data Cache Lines

This plot analyzes how the number of data cache lines affects the miss rate. From this data, we can conclude that miss rate improvement degrades as the number of data cache lines approaches sixteen lines. The improvement between eight and sixteen lines is minimal.

Figure 6.2 shows the average miss rate versus the number of instruction cache lines. Each data set is separated into four groups for each data cache size.

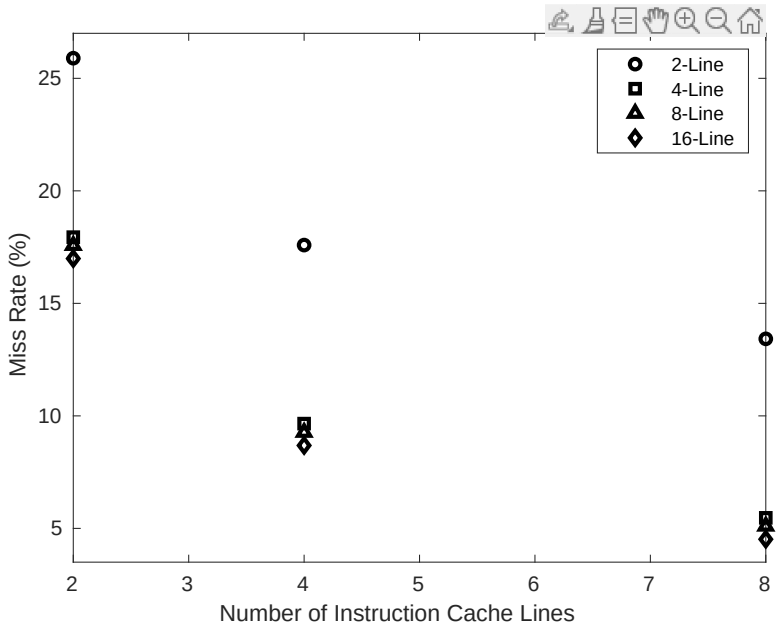


Figure 6.2: Average Miss Rate as a Function of the Number of Instruction Cache Lines

From this data, we can conclude that as the number of instruction cache lines approaches eight lines, miss rate improvement degrades. The average write-back rate and the number of data cache lines were also analyzed. Figure 6.3 shows the write-back rate versus the number of data cache lines.

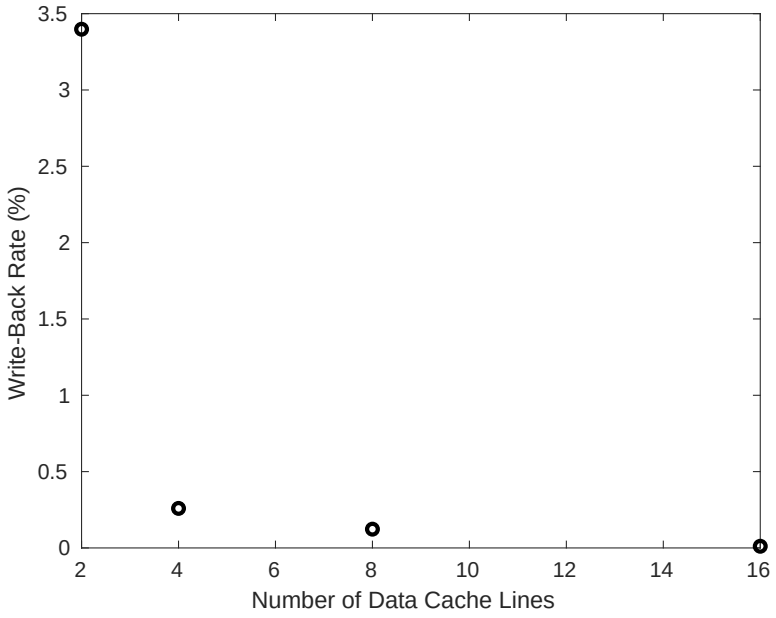


Figure 6.3: Average Write-Back Rate as a Function of the Number of Data Cache Lines

This plot analyzes how the number of data cache lines affects the write-back rate. The write-back rate follows the same pattern as the miss rate, where improvement degrades as the number of data cache lines approaches sixteen.

6.1.2 Average Memory Access Time

Average memory access time (AMAT) is a metric used to measure computer memory performance [10]. AMAT is determined by hit time, miss rate, and miss penalty. The formula is: $AMAT = HitTime + MissPenalty * MissRate$

The parameters used for the instruction cache calculation are:

- Instruction Cache:
  - Hit Time = 1 Clock Cycle
  - Miss Penalty:
    - Without Shielding: 256 Clock Cycles

    - With Shielding: 1180 Clock Cycles
  - Miss Rate:
    - 2-Line: 19.6%
    - 4-Line: 11.3%
    - 8-Line: 7.13%

Table 6.1 shows the average memory access time in the number of clock cycles for each instruction cache size. The results for this calculation were rounded up because the system does not operate on less than one cycle.

# Lines	Sh/NoSh	AMAT
2 Lines	NoSh	52
4 Lines	NoSh	30
8 Lines	NoSh	20
2 Lines	Sh	233
4 Lines	Sh	135
8 Lines	Sh	86

Table 6.1: Average Memory Access Time for the Instruction Cache

For the data cache, we alter the AMAT formula to include the write-back rate. It is important to distinguish between the data cache miss and write-back rates. Cache write-backs double the time that the processor stalls for a miss. The formula is: $AMAT = HitTime + MissPenalty * MissRate + MissPenalty * WriteBackRate$

The parameters used for the data cache calculation are:

- Hit Time = 1 Clock Cycle
- Miss Penalty:
  - Without Shielding: 256 Clock Cycles
  - With Shielding: 1180 Clock Cycles

- Miss Rate:
  - 2-Line: 19.0%
  - 4-Line: 11.0%
  - 8-Line: 10.7%
  - 16-Line: 10.1%
- Write-Back Rate:
  - 2-Line: 3.40%
  - 4-Line: 0.26%
  - 8-Line: 0.12%
  - 16-Line: 0.01%

Table 6.2 shows the average memory access time in the number of clock cycles for each data cache size.

# Lines	Sh/NoSh	AMAT
2 Lines	NoSh	51
4 Lines	NoSh	30
8 Lines	NoSh	29
16 Lines	NoSh	27
2 Lines	Sh	272
4 Lines	Sh	135
8 Lines	Sh	129
16 Lines	Sh	122

Table 6.2: Average Memory Access Time for the Data Cache

6.1.3 Hardware Cost Analysis of Shielding

We compare the resource usage of the shielding and non-shielding versions of the system. The addition of the shielding modules increases the KGE:

- Total KGE without Shielding: 33
- Total KGE with Shielding: 76.2
- Increase: 131%

The DCD computing system includes two instances of the shielding module. For a system that seeks to introduce additional shielding modules, a designer could expect an approximate 21.6 KGE increase for each additional module.

For FPGA implementations of the design, the shielding functionality increases the total LUTs, FFs, and BRAM. The resulting increases are:

- LUTs
  - Without Shielding: 1609
  - With Shielding: 2886
  - Increase: 79.4%
- FFs
  - Without Shielding: 1661
  - With Shielding: 1817
  - Increase: 9.39%
- BRAM
  - Without Shielding: 13
  - With Shielding: 17
  - Increase: 30.8%

6.1.4 Hardware Cost Analysis of Cache Size

For FPGA implementations of the design, the number of data cache lines increases the total LUTs, FFs, and BRAM. The resulting increases are:

- Instruction Cache
  - LUTs
    - Increase From 2 to 4 Lines: 14.3%
    - Increase From 4 to 8 Lines: 100%
  - FFs
    - Increase From 2 to 4 Lines: 75.0%
    - Increase From 4 to 8 Lines: 82.1%
  - BRAM
    - Increase From 2 to 4 Lines: 66.7%
    - Increase From 4 to 8 Lines: 80.0%
- Data Cache
  - LUTs
    - Increase From 2 to 4 Lines: 13.9%
    - Increase From 4 to 8 Lines: 97.6%
    - Increase From 8 to 16 Lines: 70.4%
  - FFs
    - Increase From 2 to 4 Lines: 77.8%
    - Increase From 4 to 8 Lines: 84.4%
    - Increase From 8 to 16 Lines: 96.6%

- BRAM
    - Increase From 2 to 4 Lines: 66.7%
    - Increase From 4 to 8 Lines: 80.0%
    - Increase From 8 to 16 Lines: 88.9%

# Lines	Miss Rate	LUTs	FFs	BRAM
2 to 4 Lines	-41.9%	+13.9%	+77.8%	+66.7%
4 to 8 Lines	-3.4%	+97.6%	+84.4%	+80.0%
8 to 16 Lines	-5.51%	+70.4%	+96.6%	+88.9%

Table 6.3: Data Cache Miss Rate and Resource Utilization Percent Change

# Lines	Miss Rate	LUTs	FFs	BRAM
2 to 4 Lines	-42.4%	+14.3%	+75.0%	+66.7%
4 to 8 Lines	-36.9%	+100%	+82.1%	+80.0%

Table 6.4: Instruction Cache Miss Rate and Resource Utilization Percent Change

6.2 Optimal Design Decision

Optimal system parameters vary based on desired execution time, security, hardware area, FPGA resource usage, and available memory. From this analysis, we optimize the instruction and data cache of the system. The first optimization we make is for the instruction cache size. This analysis shows a significant performance increase up to the max size of eight cache lines. Although the performance increase degrades as the number of cache lines approaches this size, there is still a substantial increase from four cache lines. The increase in performance is large enough to warrant testing of larger sizes for the instruction cache, the analysis of AMAT demonstrates this. This is evident when analyzing the access time of the shielding version of the system. The main goal of implementing the cache is to develop a way to combat the increase in execution time due to the addition of complex security hardware. We

also analyze the relationship between miss rates and hardware cost. The increase in FPGA resource utilization for eight cache lines is a good trade-off for the decrease in the miss rate. Table 6.4 shows the hardware cost and miss rate performance changes between the two, four, and eight-line caches. The miss rate decrease from four to eight lines is approximately 37 percent. While an instruction cache with eight lines is the current system optimization, future work will implement larger cache sizes to determine where the performance change is insignificant.

The second optimization we make is the number of data cache lines. Initial analysis shows the same deceleration curve as the instruction cache. This indicates that increasing the number of data cache lines above eight lines does not have much effect on the miss rate. Through further AMAT and hardware cost analysis, the data cache miss rate does not improve enough to warrant a size over four lines. The average miss rate decreases by 3.38%. Between the AMAT of four-line and eight-line is a difference of six clock cycles. From Table 6.3, the increase in hardware resources is more significant in comparison. It is up to the user or application specifications to determine if this slight change in access time is worth it. Other than increasing the number of lines, a different approach is needed to improve the performance of the data cache further.

Based on the current cache line size options, the miss rate performance change, shielding memory access time, and hardware cost were considered for a baseline-optimized system. Optimization is driven by the application. This optimization is subject to change for different desired parameters in a given application. The optimization is an eight-line instruction cache and a four-line data cache based on the benchmarks we employed. The main goal of this optimization is to combat the time it takes to shield data and instructions while still considering hardware costs.

Chapter 7

CONCLUSION

7.1 Summary

This thesis integrates secure processor design and caching to improve security and program execution time. The literature review and analysis of the DCD Secure Computing System creates a better understanding of the need for security improvements in embedded systems. As a part of the larger DCD project, the cache design implemented in this thesis focuses on the issue of increased program execution time due to secure, complex, and expensive hardware. The instruction and data cache designs are variable in size, allowing for analysis and optimization of the system. Multiple parameters were tested to show the relationship between security, computation performance, and hardware cost. Execution time was tested using a standardized set of benchmark programs. The execution time results allowed for estimating cache miss rate, miss penalty, and average memory access time. KGE is estimated using the Yosys synthesizing tool, and FPGA resource utilization is analyzed with Vivado. The combination of these tests resulted in optimizations for cache size.

7.2 Applications

Part of the research and development for computer hardware is accounting for versatility. The DCD computing system is modular. A breakdown of each module and sub-module can be provided. This allows the cache modules to be integrated without a complete system redesign. The caches are designed to be variable in size, modify or implement new control states, change the replacement policy, or be removed

from the system. The shielding module is also designed so that each sub-module can be modified or removed. The encryption method could easily be swapped out for a stronger or less hardware-intensive method. This applies to most shielding sub-modules as long as new additions do not interfere with timing or data size. If new changes or additions do not meet these requirements, more modifications must be made. This type of design process was used to easily test and change parts in the system with a minimal amount of trouble.

There could be conflicts or more work needed if entire architectures are changed in the system. The cache design in this thesis implements a Harvard architecture that uses separate memories and caches for instructions and data. Most modules need rearrangement and modification to restructure the system as a unified instruction and data memory architecture. There could be some possible benefits to a unified memory. The system's current design includes two separate instruction and data caches, each requiring a shielding module. A unified architecture would mean that only one instance of the shielding module is needed. This could be beneficial regarding hardware resource use because the shielding module is the largest part of the system. However, there are some implications to this. Data and instructions cannot be accessed in parallel. Simultaneous access to both will delay one of them and slow down the processor.

Another issue is the time it takes for the system to shield and de-shield blocks of data and instructions. The average memory access time is expected to increase without two shielding modules working simultaneously to secure data. This could slow access times, but there is some potential for increased security. For the block shuffling functionality of the shielding module, there could be some benefit to shuffling random blocks in a unified memory. The current Harvard architecture distinguishes between blocks of data shuffled in the data memory and blocks of instructions shuffled

in the program memory. An attacker could potentially exploit the information that a particular block is a set of data or instructions. Processor op-codes and instructions are used frequently in program execution and may reveal patterns in the shielded output. Shuffling data and instructions together would not reveal this information to an attacker.

7.3 Future Work

More work still needs to be completed for randomness and cache size testing. As stated in Chapter 5, the DieHarder test suite was not used in a way that yields definitive results. Future work for this includes creating C programs that replicate the shielding functionality to output directly to DieHarder. Currently, the DieHarder test suite has to rewind the data file many times to complete the tests. The C program versions of the shielding process are necessary for an accurate test. It would be difficult to output a file size large enough from the system to give DieHarder the needed data. The tests performed in this thesis are not the final measure of security for the system.

Work still needs to be done to optimize the system. Potential additions to the cache design are increasing the number of instruction cache lines, testing the cache for variable block size, and adding multiple levels. Once the DieHarder tests are completed, further analysis can be performed for the perceived randomness of the shielding module, hardware cost, and execution time.

Appendix A

Binary Search Benchmark:

```
#include "uart.h"
#include <stdlib.h>
#include "intToStr.h"



// Function Prototypes
void binarySearch( int keyValue, int *counter, int array[], int upper,
                                                                int lower, char* UART );




int main()
   {



   // variable initialization
   int array[ DATA_SIZE ];

   char* LED = ( char* )0x00010000;
   char* UART = ( char* )0x00010004;


   /* change variables for use in vivado */
   unsigned int* searchCounter = ( int* )0x00000500; // 320 in memory
   int loopIndex, iteration, offsetVal, searchVal;
   *searchCounter = 0;

   srand( SEED );

   offsetVal = rand() % 14 + 1;
   searchVal = rand() % DATA_SIZE * offsetVal;

   for( loopIndex = 0; loopIndex < DATA_SIZE; loopIndex++ )
      {

      // set the value of loop index * 12 in the array
      array[ loopIndex ] = loopIndex * offsetVal;

      }

   for( iteration = 0; iteration < ITERATIONS; iteration++ )
      {

      asm volatile( "li tp, 1" );
      binarySearch( searchVal, searchCounter, array, DATA_SIZE, 0, UART );
```

```
49
50
51       }
52
53    asm volatile( "li tp, 2" );
54
55
56
57
58    return 0;
59    }
60
61
62 // Function Implementation
63 void binarySearch( int keyValue, int *counter, int array[],
      int upper,
64                                                          int
      lower, char* UART )
65    {
66
67
68    // function/variable initialization
69    *counter = *counter + 1;
70    int middle = ( upper + lower ) / 2;
71    char intAsStr[ 33 ];
72
73    // check if the value at the middle is less than the key
74    if( array[ middle ] < keyValue )
75       {
76
77       binarySearch( keyValue, counter, array, upper, middle,
      UART );
78
79       }
80    // otherwise, check if the value at the middle is greater
      than the key
81    else if( array[ middle ] > keyValue )
82       {
83
84       binarySearch( keyValue, counter, array, middle, lower,
      UART );
85
86       }
87
88    else
89       {
90
91       sendStringToUart( "The search key was found after ",
      UART );
92       getStr( *counter, intAsStr );
93       sendStringToUart( intAsStr, UART );
94       sendStringToUart( "searches. The value was ", UART );
95       getStr( keyValue, intAsStr );
96       sendStringToUart( ".\n\r", UART );
97
98       }
```

```
 99
100
101
102
103     return;
104     }
```

Matrix Multiplication Benchmark:

```
 1  #include <stdlib.h>
 2 #include "uart.h"
 3 #include "intToStr.h"
 4
 5 void multiplyMatricies( const int factorOne[][ MATRIX_COLS],
    const int
 6           factorTwo[][ MATRIX_COLS ], int productMatrix[][
    MATRIX_COLS ] );
 7 int findDotProduct( const int row, const int col, const int
 8         factorOne[][ MATRIX_COLS ], const int factorTwo[][
    MATRIX_COLS ] );
 9 void printMatrix( int matrix[MATRIX_ROWS * MATRIX_COLS * 3],
    char* address );
10 int getRand();
11 void moveMatricies( const int matrixOne[][ MATRIX_COLS ],
12                                          const int
    matrixTwo[][ MATRIX_COLS ],
13                          const int productMatrix[][
    MATRIX_COLS ],
14                int printMatrix[ MATRIX_ROWS * MATRIX_COLS * 3
    ] );
15
16 // main program
17 int main()
18    {
19
20    // declare variables
21    int rowIndex, colIndex, count;
22    char* LED = ( char* )0x00010000;
23    char* UART = ( char* )0x00010004;
24
25    int matrixOne[ MATRIX_ROWS ][ MATRIX_COLS ];
26    int matrixTwo[ MATRIX_ROWS ][ MATRIX_COLS ];
27    int productMatrix[ MATRIX_ROWS ][ MATRIX_COLS ];
28    int printableMatrix[ MATRIX_ROWS * MATRIX_COLS * 3 ];
29
30    *LED = 1;
31    srand( SEED );
32
33    for( count = 0; count < ITERATIONS; count++ )
34       {
35
36
37       // fill the matricies
38       for( rowIndex = 0; rowIndex < MATRIX_ROWS; rowIndex++ )
39          {
40          for( colIndex = 0; colIndex < MATRIX_COLS;
    colIndex++ )
41             {
42             matrixOne[ rowIndex ][ colIndex ] = getRand();
43             }
44          }
45
46       for( rowIndex = 0; rowIndex < MATRIX_ROWS; rowIndex++ )
47          {
```

```
48          for( colIndex = 0; colIndex < MATRIX_COLS;
    colIndex++ )
49              {
50              matrixTwo[ rowIndex ][ colIndex ] = getRand();
51      }
52          }
53
54       *LED = 3;
55
56       asm volatile( "li tp, 1");
57
58       // multiply the matricies together
59
60       multiplyMatricies( matrixOne, matrixTwo, productMatrix
    );
61
62       *LED = 4;
63
64
65
66    }
67
68    asm volatile( "li tp, 2");
69
70
71
72    sendStringToUart( "Done\n\r", UART );
73    moveMatricies( matrixOne, matrixTwo, productMatrix,
    printableMatrix );
74    printMatrix( printableMatrix, UART );
75    sendStringToUart( "\n\r", UART );
76
77
78
79
80
81
82
83
84
85    *LED = 8;
86
87    while( 1 );
88
89
90    // end the program
91    return 0;
92
93    }
94
95
96 void printMatrix( int matrix[ MATRIX_ROWS * MATRIX_COLS * 3
    ], char* address )
97    {
98
99    char intAsStr[ 33 ];
```

```
100    int index;
101    int length = 27;
102
103
104    for( index = 0; index < length; index++ )
105       {
106
107       /*
108       if( index % ( MATRIX_ROWS * MATRIX_COLS ) == 0 )
109          {
110
111   if( index == 0 )
112      {
113            sendStringToUart( "Factor One:", address );
114      }
115   else if( index == MATRIX_ROWS * MATRIX_COLS )
116      {
117            sendStringToUart( "Factor Two:", address );
118      }
119   else
120      {
121            sendStringToUart( "Product:", address );
122            }
123   }
124       if( index % MATRIX_COLS == 0 )
125          {
126          sendStringToUart( "\n\r", address );
127
128   }
129       */
130       getStr( matrix[ index ], intAsStr );
131       sendStringToUart( intAsStr, address );
132       sendStringToUart( ", ", address );
133
134
135       }
136
137    }
138
139 void multiplyMatricies( const int factorOne[][ MATRIX_COLS ],
140   const int factorTwo[][ MATRIX_COLS ], int productMatrix[][
      MATRIX_COLS] )
141    {
142    // initialize function/variables
143    int rowIndex, colIndex;
144
145    // loop across the entire product array
146    for( rowIndex = 0; rowIndex < MATRIX_ROWS; rowIndex++ )
147       {
148       for( colIndex = 0; colIndex < MATRIX_COLS; colIndex++ )
149          {
150
151          // set the row and col of the product matrix to the
      sum of the dot
152
              // product
```

```
153           productMatrix[ rowIndex ][ colIndex ] =
      findDotProduct( rowIndex,
154                                                   colIndex,
      factorOne, factorTwo );
155
156           }
157        }
158     }
159
160
161
162
163  int findDotProduct( const int row, const int col, const int
164           factorOne[][ MATRIX_COLS ], const int factorTwo[][
      MATRIX_COLS ] )
165     {
166     // intialize function/variables
167     int dotProduct = 0, index;
168
169     // loop from starting index to ending index
170     for( index = 0; index < MATRIX_COLS; index++ )
171        {
172        // add the product of corresponding places in the
      factors
173        dotProduct +=
174           ( factorOne[ row ][ index ] * factorTwo[ index ][
      col ] );
175        }
176
177
178     // return the dot product
179     return dotProduct;
180     }
181
182  int getRand()
183     {
184
185     int num = 0;
186
187     while( num == 0 )
188        {
189
190        num = rand() % 4864;
191
192        }
193
194     return num;
195
196     }
197
198  void moveMatricies( const int matrixOne[][ MATRIX_COLS ],
199                                                const int
      matrixTwo[][ MATRIX_COLS ],
200                         const int productMatrix[][
      MATRIX_COLS ],
201                 int printMatrix[ MATRIX_ROWS * MATRIX_COLS * 3
```

```
    ] )
202 {
203
204 int rowIndex, colIndex, printIndex = 0;
205
206 for( rowIndex = 0; rowIndex < MATRIX_ROWS; rowIndex++ )
207    {
208    for( colIndex = 0; colIndex < MATRIX_COLS; colIndex++ )
209       {
210       printMatrix[ printIndex ] = matrixOne[ rowIndex ][
    colIndex ];
211 printIndex++;
212       }
213    }
214
215 for( rowIndex = 0; rowIndex < MATRIX_ROWS; rowIndex++ )
216    {
217    for( colIndex = 0; colIndex < MATRIX_COLS; colIndex++ )
218       {
219       printMatrix[ printIndex ] = matrixTwo[ rowIndex ][
    colIndex ];
220 printIndex++;
221       }
222    }
223
224
225 for( rowIndex = 0; rowIndex < MATRIX_ROWS; rowIndex++ )
226    {
227    for( colIndex = 0; colIndex < MATRIX_COLS; colIndex++ )
228       {
229       printMatrix[ printIndex ] = productMatrix[ rowIndex
    ][ colIndex ];
230 printIndex++;
231       }
232    }
233
234 }
```

Max Benchmark:

```
 1
 2 #include <stdlib.h>
 3 #include "uart.h"
 4 #include "intToStr.h"
 5
 6 // constants
 7
 8
 9 // function declaration
10 void findLargestNumber( const int* arr, int* store );
11
12
13
14 // main function
15 void main()
16    {
17
18    // declare variables
19    int arr[ DATA_SIZE ];
20    int store[ 2 ];
21    int loopIndex;
22    int maxValIndex;
23    int maxVal;
24    char* LED = ( char* )0x00010000;
25    char* UART = ( char* )0x00010004;
26    char intAsStr[ 33 ];
27
28    srand( SEED );
29
30    *LED = 1;
31
32    maxVal = rand();
33    maxValIndex = rand() % DATA_SIZE;
34
35    getStr( maxVal, intAsStr );
36    sendStringToUart( "The max value should be ", UART );
37    sendStringToUart( intAsStr, UART );
38    sendStringToUart( "\n\r", UART );
39
40    *LED = 2;
41
42
43    // add the nodes
44    for( loopIndex = 0; loopIndex < DATA_SIZE; loopIndex++ )
45       {
46
47       if( loopIndex != maxValIndex )
48          {
49
50  arr[ loopIndex ] = rand() % maxVal - 1;
51
52  }
53       else
54          {
55
```

```
56          arr[ loopIndex ] = maxVal;
57
58   }
59
60       }
61
62    *LED = 3;
63
64    asm( "li tp, 1" );
65
66
67    // find the largest value
68    findLargestNumber( arr, store );
69
70
71
72    // set value to stop timer
73    asm( "li tp, 2" );
74
75    *LED = 4;
76
77
78
79    getStr( store[ 0 ], intAsStr );
80    sendStringToUart( "The max value was found to be ", UART );
81    sendStringToUart( intAsStr, UART );
82    sendStringToUart( "\n\r", UART );
83
84    *LED = 8;
85
86
87    while( 1 );
88
89
90    }
91
92
93 // functions
94 void findLargestNumber( const int* arr, int* store )
95    {
96    // initialize function/variables
97    store[ 0 ] = arr[ 0 ];
98    int loopIndex;
99
100   for( loopIndex = 0; loopIndex < DATA_SIZE; loopIndex++ )
101      {
102
103      if( arr[ loopIndex ] > store[ 0 ] )
104         {
105
106   store[ 0 ] = arr[ loopIndex ];
107   store[ 1 ] = loopIndex;
108
109   }
110      }
111    }
```

Two-Dimensional Convolution Benchmark:

```
1     #include <stdbool.h>
2  #include <stdlib.h>
3  #include "intToStr.h"
4  #include "uart.h"
5
6  // Macros - used only for array initialization
7  #define KERNEL_INIT 3
8
9  // Global Constants
10 const int KERNEL_ROWS = 3;
11 const int KERNEL_COLS = 3;
12
13
14 // Prototypes
15 void convolution( const int matrix[][ MATRIX_COLS ], const int
16                          kernel[][ KERNEL_COLS ], int
      result[][ MATRIX_COLS ] );
17 bool hasNeighbors( const int matrix[][ MATRIX_COLS ], int
      row, int col );
18 int findValue( const int matrix[][ MATRIX_COLS ], const int
19                                    kernel[][ KERNEL_COLS ],
      int row, int col );
20 void printMatrix( int matrix[ MATRIX_ROWS * MATRIX_COLS * 3
      ], char* address );
21
22
23 // Main Program
24 int main()
25    {
26
27    int startCycles = 0, endCycles = 0;
28    char* LED = ( char* )0x00010000;
29    char* UART = ( char* )0x00010004;
30    int rowIndex, colIndex, iteration = 0;
31
32    srand( SEED );
33
34    *LED = 1;
35
36
37
38    // Declare Variables
39    int matrix[ MATRIX_ROWS ][ MATRIX_COLS ];
40
41    int result_matrix[ MATRIX_ROWS ][ MATRIX_COLS ];
42
43    int kernel[][ KERNEL_INIT ] = {
44                                    { 0, 0, 0 },
45      { 0, 1, 0,},
46      { 0, 0, 0 },
47      }; // identity kernel
48
49
50    for( iteration = 0; iteration < ITERATIONS; iteration++ )
51       {
```

```
52       for( rowIndex = 0; rowIndex < MATRIX_ROWS; rowIndex++ )
53          {
54          for( colIndex = 0; colIndex < MATRIX_COLS;
     colIndex++ )
55             {
56
57       matrix[ rowIndex ][ colIndex ] = rand() % 4684;
58
59       }
60           }
61
62        asm volatile( "li tp, 1");
63
64        // Process the two dimension convolution
65        convolution( matrix, kernel, result_matrix );
66
67
68        }
69
70    *LED = 8;
71
72    asm volatile( "li tp, 2" );
73
74
75    // printMatrix( result_matrix, UART );
76
77
78    }
79
80
81 // function implementation
82 void convolution( const int matrix[][ MATRIX_COLS ], const int
83                            kernel[][ KERNEL_COLS ], int
     result[][ MATRIX_COLS ] )
84    {
85    // function/variable initialization
86    int rowIndex, colIndex;
87
88
89    // loop through the matrix
90    for( rowIndex = 0; rowIndex < MATRIX_ROWS; rowIndex++ )
91       {
92       for( colIndex = 0; colIndex < MATRIX_COLS; colIndex++ )
93          {
94
95   // check if the "pixel" has neighbors
96          if( hasNeighbors( matrix, rowIndex, colIndex ) )
97             {
98
99             // run the convolution
100            result[ rowIndex ][ colIndex ] =
101                         findValue( matrix, kernel, rowIndex,
     colIndex );
102
103            }
104   // otherwise, assume the "pixel" is on the edge
```

```
105           else
106              {
107
108       // copy the value straight to the result
109       result[ rowIndex ][ colIndex ] = matrix[ rowIndex ][
      colIndex ];
110
111      }
112           }
113        }
114     }
115
116
117  bool hasNeighbors( const int matrix[][ MATRIX_COLS ], int
      row, int col )
118     {
119
120
121     // check if the four places around the given "pixel" are
      in bounds
122     return ( KERNEL_ROWS <= MATRIX_ROWS && row + KERNEL_ROWS /
      2 < MATRIX_ROWS ) &&
123             ( KERNEL_COLS <= MATRIX_COLS && col + KERNEL_COLS /
      2 < MATRIX_COLS );
124
125     }
126
127
128  int findValue( const int matrix[][ MATRIX_COLS ], const int
129                                       kernel[][ KERNEL_COLS ],
      int row, int col )
130     {
131     // function/variable initialization
132     int rowLowerLimit = row - KERNEL_ROWS / 2;
133     int rowUpperLimit = row + KERNEL_ROWS / 2;
134
135     int colLowerLimit = col - KERNEL_COLS / 2;
136     int colUpperLimit = col + KERNEL_COLS / 2;
137
138     int dotProduct = 0, rowIndex, colIndex;
139
140     // loop from starting index to ending index
141     for( rowIndex = rowLowerLimit; rowIndex <= rowUpperLimit;
      rowIndex++ )
142        {
143        for( colIndex = colLowerLimit; colIndex <=
      colUpperLimit; colIndex++ )
144           {
145           // add the product of corresponding places in the
      factors
146           dotProduct += matrix[ rowIndex][ colIndex ] *
147           kernel[ rowIndex - rowLowerLimit ][ colIndex -
      colLowerLimit ];
148           }
149        }
150
```

```
151
152
153    // return that dot product
154    return dotProduct;
155
156
157
158
159    }
160
161
162 void printMatrix( int matrix[MATRIX_ROWS * MATRIX_COLS * 3],
       char* address )
163    {
164
165    char intAsStr[33];
166    int index;
167    int length = 27;
168
169
170    for( index = 0; index < length; index++ )
171       {
172
173       /*
174       if( index % ( MATRIX_ROWS * MATRIX_COLS ) == 0 )
175          {
176
177   if( index == 0 )
178      {
179             sendStringToUart( "Factor One:", address );
180      }
181   else if( index == MATRIX_ROWS * MATRIX_COLS )
182      {
183             sendStringToUart( "Factor Two:", address );
184      }
185   else
186      {
187             sendStringToUart( "Product:", address );
188             }
189   }
190       if( index % MATRIX_COLS == 0 )
191          {
192          sendStringToUart( "\n\r", address );
193   }
194       */
195       getStr( matrix[ index ], intAsStr );
196       sendStringToUart( intAsStr, address );
197       sendStringToUart( ", ", address );
198
199
200       }
201
202    }
```

Quicksort Benchmark:

```
1     //#include <stdio.h>
2  #include <stdlib.h>
3  #include "uart.h"
4  #include "intToStr.h"
5
6
7  // Prototypes
8  void runQuickSort( int array[], int size );
9  void runQuickSortHelper( int array[], int lowIndex, int
      highIndex );
10 int runPartition( int array[], int lowIndex, int highIndex );
11
12
13
14 // Main Program
15 int main()
16    {
17
18
19    // intitialize variables
20    int array[ DATA_SIZE ];
21    int loopIndex, arrIndex;
22    char* LED = ( char* )0x00010000;
23    char* UART = ( char* )0x00010004;
24    char intAsStr[ 33 ];
25    srand( SEED );
26    int index, iteration;
27
28    for( iteration = 0; iteration < ITERATIONS; iteration++ )
29       {
30       for( index = 0; index < DATA_SIZE; index++)
31          {
32          array[ index ] = rand();
33          }
34
35       asm volatile( "li tp, 1" );
36
37
38       // sort the array
39       runQuickSort( array, DATA_SIZE );
40
41
42
43       *LED = 4;
44
45
46
47
48       }
49
50
51    asm volatile( "li tp, 2" );
52    sendStringToUart( "Done\n\rArray:\n\r", UART );
53
54
```

```
55    for( index = 0; index < DATA_SIZE; index++ )
56       {
57       getStr( array[ index ], intAsStr );
58       sendStringToUart( intAsStr, UART );
59       sendStringToUart( ", ", UART );
60       }
61
62
63    *LED = 8;
64
65    while( 1 );
66
67    return 0;
68    }
69
70
71
72 // Function Implementation
73
74 void runQuickSort( int array[], int size )
75    {
76
77    // run the helper function
78    runQuickSortHelper( array, 0, size - 1 );
79
80    }
81
82 void runQuickSortHelper( int array[], int lowIndex, int
      highIndex )
83    {
84    // initialize function/variables
85    int pivotIndex;
86
87    // check if low index is less than high index
88    if( lowIndex < highIndex )
89       {
90       // call partition to get the pivot value
91       pivotIndex = runPartition( array, lowIndex, highIndex );
92
93       // run the quicksort on the left and right sides of the
      list
94       runQuickSortHelper( array, lowIndex, pivotIndex - 1 );
95       runQuickSortHelper( array, pivotIndex + 1, highIndex );
96
97       }
98    }
99
100 int runPartition( int array[], int lowIndex, int highIndex )
101    {
102
103    // intialize function/variables
104    int workingIndex, pivotIndex = lowIndex, temp;
105    int pivot = array[ lowIndex ]; // set the pivot as the
      first element
106
107    // loop across the array
```

```
108     for( workingIndex = lowIndex; workingIndex <= highIndex;
        workingIndex++ )
109        {
110        // check if the element at working index is less than
        the pivot
111        if( array[ workingIndex ] < pivot )
112           {
113
114           // increment the pivot index
115           pivotIndex++;
116
117           // swap the working index and the pivot index
118           temp = array[ workingIndex ];
119           array[ workingIndex ] = array[ pivotIndex ];
120           array[ pivotIndex ] = temp;
121
122           }
123
124
125        }
126
127     // swap the pivot with the pivot index
128     temp = array[ pivotIndex ];
129     array[ pivotIndex ] = pivot;
130     array[ lowIndex ] = temp;
131
132     // return the pivot index
133     return pivotIndex;
134
135     }
```

REFERENCES

[1] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *Journal of the ACM (JACM)*, 18(1):80–93, 1971.

[2] J. A. Ambrose, R. G. Ragel, D. Jayasinghe, T. Li, and S. Parameswaran. Side channel attacks in embedded systems: A tale of hostilities and deterrence. In *Sixteenth International Symposium on Quality Electronic Design*, pages 452–459. IEEE, 2015.

[3] C. J. Benvenuto. Galois field in cryptography. *University of Washington*, 1(1):1–11, 2012.

[4] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, et al. Prince–a low-latency block cipher for pervasive computing applications. In *Advances in Cryptology–ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings 18*, pages 208–225. Springer, 2012.

[5] R. G. Brown. Dieharder: A gnu public licensed random number tester. *Draft paper included as file manual/dieharder. tex in the dieharder sources. Last version dated*, 20, 2006.

[6] C. Carvalho. The gap between processor and memory speeds. In *Proc. of IEEE International Conference on Control and Automation*, 2002.

[7] S. Even and Y. Mansour. A construction of a cipher from a single pseudorandom permutation. *Journal of cryptology*, 10:151–161, 1997.

[8] R. Fagin and T. G. Price. Efficient calculation of expected miss ratios in the independent reference model. *SIAM Journal on Computing*, 7(3):288–297, 1978.

[9] P. G. Flikkema, J. Palmer, T. Yalcin, and B. Cambou. Dynamic computational diversity with multi-radix logic and memory. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2020.

[10] J. L. Hennessy and D. A. Patterson. Computer organization and design risc-v edition: The hardware software interface, 2017.

[11] T. A. Henzinger and J. Sifakis. The embedded systems design challenge. In *FM 2006: Formal Methods: 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006. Proceedings 14*, pages 1–15. Springer, 2006.

[12] M. D. Hill. A case for direct-mapped caches. *Computer*, 21(12):25–40, 1988.

[13] W. King. Analysis of paging algorithms. In *Proc. IFIP 1971 Congress, Ljubljana*, pages 485–490. North-Holland, 1972.

[14] S. Kumar and P. Singh. An overview of modern cache memory and performance analysis of replacement policies. In *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, pages 210–214. IEEE, 2016.

[15] M. Liskov, R. L. Rivest, and D. Wagner. Tweakable block ciphers. In *Advances in Cryptology—CRYPTO 2002: 22nd Annual International Cryptology Conference Santa Barbara, California, USA, August 18–22, 2002 Proceedings 22*, pages 31–46. Springer, 2002.

[16] G. Marsaglia. The marsaglia random number cdrom including the diehard battery of tests of randomness. *http://www. stat. fsu. edu/pub/diehard/*, 2008.

[17] G. Nişancı, P. G. Flikkema, and T. Yalçın. Symmetric cryptography on risc-v: Performance evaluation of standardized algorithms. *Cryptography*, 6(3):41, 2022.

[18] G. S. Rao. Performance analysis of cache memories. *Journal of the ACM (JACM)*, 25(3):378–395, 1978.

[19] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, Booz-allen and hamilton inc mclean va, 2001.

[20] A. J. Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[21] R. Sobti and G. Geetha. Cryptographic hash functions: a review. *International Journal of Computer Science Issues (IJCSI)*, 9(2):461, 2012.

[22] T. Sudarshan, R. A. Mir, and S. Vijayalakshmi. Highly efficient lru implementations for high associativity cache memory. In *Proceedings of 12th IEEE International Conference on Advanced Computing and Communications*, pages 24–35. Citeseer, 2004.

[23] E. B. Talaki, O. Savry, M. Bouvier Des Noes, and D. Hely. A memory hierarchy protected against side-channel attacks. *Cryptography*, 6(2):19, 2022.

[24] Y. Wang, W.-k. Yu, S. Wu, G. Malysa, G. E. Suh, and E. C. Kan. Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints. In *2012 IEEE Symposium on Security and Privacy*, pages 33–47. IEEE, 2012.

[25] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2, 2014.

[26] M. V. Wilkes. The memory gap and the future of high performance memories. *ACM SIGARCH Computer Architecture News*, 29(1):2–7, 2001.

[27] C. Wolf, J. Glaser, and J. Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, page 97, 2013.