Computer Architecture Final Project

Oklahoma State University

Psuedo Random Cache Replacement

Author
Garrett Page

Instructor
Rose Thompson

May 9, 2024

Contents

1 Introduction

1.1 RISC-V

At its core, RISC-V stands for Reduced Instruction Set Computing, Version 5. It's an open standard, offering a breath of fresh air in an industry often dominated by proprietary designs. RISC-V is built on a few fundamental principles that make it both powerful and adaptable.

First and foremost, RISC-V embraces simplicity. Its instruction set is lean and efficient, comprising a small number of basic operations that form the building blocks of computing. This simplicity not only makes RISC-V easy to understand but also allows for streamlined hardware implementation.

Another key feature of RISC-V is its modular structure. The architecture defines a base instruction set, which forms the foundation for all RISC-V implementations. This base set includes essential operations like arithmetic, logic, and memory access. Beyond the base set, RISC-V also supports optional extensions, allowing designers to tailor the architecture to specific application domains. This structure allowed us to create relatively simple methods to read different types of instructions from only a few instruction layouts, making the design and development of the simulator much easier than other instruction sets may have been.

1.2 Final Project Caches

Building upon previous knowledge and practical experience with the RISC-V RV32 architecture, this project introduces an in-depth exploration of cache mechanisms using the OpenHW Group's CV-Wally processor. CV-Wally, a versatile 5-stage pipelined processor configurable to support various RISC-V extensions and features, serves as the focal point for understanding cache dynamics.

The primary objective of this project is twofold: firstly, to expand the capabilities of CV-Wally's cache by incorporating an additional replacement policy, specifically a pseudo-random policy, alongside the existing pLRU policy. Secondly, to comprehend the intricacies of cache operations through simulation and verification, ensuring accurate data retrieval and comparison with established cache management strategies.

Furthermore, this project ventures into practical experimentation by validating the effectiveness of implemented replacement policies on hardware using the Arty A7 FPGA development board. Through rigorous testing and analysis, the project aims to highlight the performance disparities between different cache replacement policies and their implications on real-world applications.

Additionally, the exploration extends to examining the impact of different writing policies, focusing on the write-through policy and its influence on cache coherence and performance when compared to existing strategies employed by CV-Wally.

2 Baseline Design

Before delving into the intricacies of cache replacement policies and their implications on processor performance, it's important to establish a solid foundation for the project's execution. The baseline design phase encompasses a series of crucial steps aimed at setting up the development environment, configuring the CV-Wally processor, and preparing for both simulation and hardware experimentation.

2.1 Setup

2.1.1 GitHub Setup

The initial step involves ensuring seamless collaboration and version control through GitHub. We initiate this process by adding their SSH key to GitHub, enabling secure access to repositories. This facilitates forking and cloning the CV-Wally repository, laying the groundwork for subsequent development tasks.

2.1.2 Repository Forking and Cloning

Upon forking the CV-Wally repository on GitHub, we clone their forked repository to their local environment using Git. This step ensures direct access to the source code and facilitates modifications and enhancements as required for the project's objectives.

2.1.3 Environment Setup

A critical aspect of the setup involves configuring the development environment for CV-Wally. Executing the 'setup.csh' script establishes essential environment variables necessary for regression testing and other development activities.

2.1.4 Regression Tests Preparation

The project requires access to compiled regression tests, which are precompiled and provided within the course environment. I copied these tests to the CV-Wally repository, enabling thorough testing and validation of cache replacement policies and other enhancements.

2.1.5 Simulation Setup

Simulations play a vital role in assessing the effectiveness of cache replacement policies. By navigating to the simulation directory and executing the 'make deriv' command, we generate derivative configurations necessary for simulating cache behavior and performance.

2.1.6 Running Regression Tests

We execute regression tests to ensure the integrity and functionality of the CV-Wally processor. This step involves running multiple tests in parallel and verifying the outcomes to identify any potential issues or discrepancies.

2.1.7 Benchmarking Setup

With regression testing completed, we transition to benchmarking core functionalities such as CoreMark. This involves compiling and simulating CoreMark benchmarks to gauge processor performance metrics, including cycles per instruction and cache hit rates.

2.1.8 FPGA Setup

Transitioning from simulation to hardware experimentation, we set up the FPGA environment for CV-Wally. This involves cloning the repository on Windows using Cygwin, configuring FPGA synthesis parameters, and building the FPGA bitstream.

2.1.9 Hardware Programming

Once the FPGA bitstream is generated, we program the Arty A7 100 board using Vivado, ensuring seamless deployment of the FPGA configuration.

2.2 Random Replacement Policy

A random cache replacement policy provides several advantages in managing cache memory within computing systems. Firstly, it offers simplicity in implementation compared to more intricate replacement strategies such as Least Recently Used (LRU) or Least Frequently Used (LFU). Unlike LRU or LFU, which necessitate the tracking of access history for each cache line, a random policy merely involves selecting a cache line randomly for eviction. This simplicity translates to lower hardware and software overhead, reducing computational costs.

Moreover, a random replacement policy ensures fairness in cache management. By treating all cache lines equally, regardless of their access history, it prevents any specific cache line from consistently receiving preferential treatment. This fairness can enhance overall system performance by preventing certain access patterns from monopolizing cache resources.

Another benefit lies in the predictability of the policy's behavior. While random replacement does not consider access patterns or temporal locality, its eviction decisions are consistent over time. This predictability simplifies performance analysis and debugging processes, as cache behavior becomes less reliant on specific workload characteristics.

Furthermore, random replacement decouples cache eviction decisions from access patterns. Unlike deterministic policies like LRU or LFU, which may bias eviction decisions based on past behavior, random replacement makes decisions independent of historical accesses. This trait can be advantageous in scenarios where access patterns are unpredictable or subject to frequent change.

Lastly, random replacement offers resistance to exploitation. Unlike some deterministic policies that may be vulnerable to manipulation by adversaries seeking to exploit cache behavior for malicious purposes, random replacement cannot be easily manipulated in the same way. This enhances system security and resilience, particularly in environments where cache behavior may be targeted for attacks.

2.2.1 Pseudo Random Number Generation

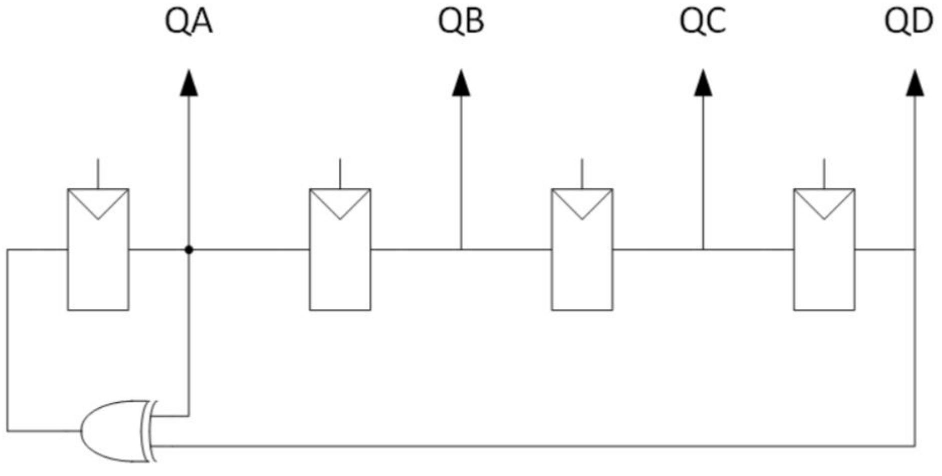
A Linear Feedback Shift Register (LFSR) is a type of shift register used in digital circuits for generating pseudo-random sequences. It operates on the principle of feedback, where the output bit of the shift register is determined by a linear combination of its previous state bits. The LFSR consists of a shift register of length N with feedback from select taps within the register. As the name suggests, it shifts its contents by one bit position at each clock cycle, and the feedback mechanism introduces a new bit into the register based on a predefined feedback polynomial.

At each clock cycle, the contents of the register are shifted one position to the left, and the rightmost bit, also known as the output bit, is calculated based on the feedback polynomial. The feedback polynomial determines which bits in the register are XORed together to produce the output bit. These bits, called tap points or taps, are selected based on specific mathematical properties to ensure maximal sequence length and good randomness characteristics. The output bit is then fed back into the register, becoming the new leftmost bit, while the other bits shift to the left.

The sequence generated by an LFSR depends entirely on its initial state and the feedback polynomial. Despite its deterministic nature, LFSRs can produce sequences that appear random, making them useful for applications such as encryption, scrambling, and pseudo-random number generation. The length of the sequence generated by an LFSR is determined by the number of bits in the register and the feedback polynomial. For a maximum-length sequence, the LFSR must be configured with a primitive polynomial, which ensures that the generated sequence cycles through all possible 2Ñ̂1 states before repeating.

LFSRs find wide application in various fields, including telecommunications, cryptography, simulation, and testing. Their simplicity, efficiency, and ability to produce pseudo-random sequences with good statistical properties make them a popular choice for generating random-like patterns in digital systems.

Figure 2.1: 4-bit LFSR



3 Design Details

The LFSR module is a SystemVerilog circuit module that generates a pseudo-random sequence of bits based on a linear feedback mechanism. The implementation provided is parameterized to support LFSRs with different widths, supporting 2, 4, 8, 16, 32, 64, or 128 bits. The following sections define the more detailed implementation details of the pseudo random replacement policy.

3.1 LFSR Module

State Register The module maintains an internal state register state of width WIDTH, which holds the current state of the LFSR.

Feedback Polynomial The core of the LFSR is the feedback polynomial, which determines the next state of the register. Each bit in the state register serves as an input to an XOR gate, with the feedback polynomial specifying which bits are XORed together to produce the feedback. The feedback calculation is critical, as it directly influences the randomness and period of the generated sequence. For simplicity, the implementation uses primitive polynomials known for producing maximal-length sequences, ensuring good randomness properties.

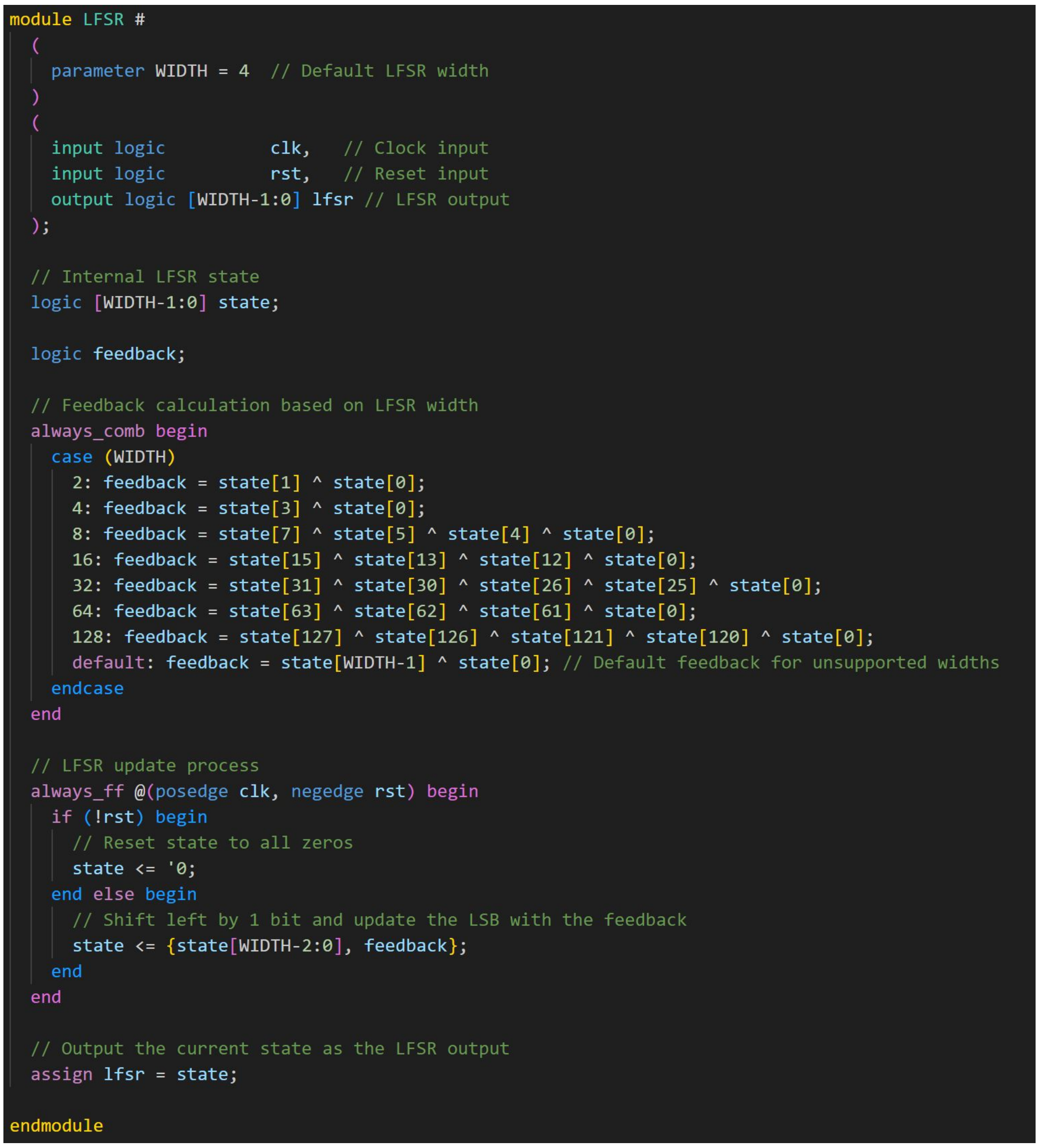
Feedback Calculation The feedback calculation is designed to be parameterized based on the selected LFSR width. Depending on the width, different tap positions within the state register are XORed to generate the feedback bit. This approach allows for flexibility in selecting the feedback polynomial and ensures compatibility with various LFSR configurations.

Clock and Reset Handling The module operates on a clocked synchronous process, triggered by the rising edge of the clock signal. It also includes an asynchronous reset signal (rst) to initialize the state register to a known state when asserted.

Shift and Update During each clock cycle, the LFSR shifts its state register left by one bit, effectively generating a new pseudo-random bit. Simultaneously, the feedback bit calculated from the current state is inserted into the least significant bit (LSB) position of the register, updating the state for the next cycle.

Output Generation The output of the LFSR is provided through the lfsr output port, which directly exposes the current state register. This output can be used externally for various applications, such as generating random numbers, test pattern generation, or cryptographic operations.

Figure 3.1: LFSR module implementation



```
module LFSR #
  (
    parameter WIDTH = 4  // Default LFSR width
  )
  (
    input logic          clk,   // Clock input
    input logic          rst,   // Reset input
    output logic [WIDTH-1:0] lfsr // LFSR output
  );

  // Internal LFSR state
  logic [WIDTH-1:0] state;

  logic feedback;

  // Feedback calculation based on LFSR width
  always_comb begin
    case (WIDTH)
      2: feedback = state[1] ^ state[0];
      4: feedback = state[3] ^ state[0];
      8: feedback = state[7] ^ state[5] ^ state[4] ^ state[0];
      16: feedback = state[15] ^ state[13] ^ state[12] ^ state[0];
      32: feedback = state[31] ^ state[30] ^ state[26] ^ state[25] ^ state[0];
      64: feedback = state[63] ^ state[62] ^ state[61] ^ state[0];
      128: feedback = state[127] ^ state[126] ^ state[121] ^ state[120] ^ state[0];
      default: feedback = state[WIDTH-1] ^ state[0]; // Default feedback for unsupported widths
    endcase
  end

  // LFSR update process
  always_ff @(posedge clk, negedge rst) begin
    if (!rst) begin
      // Reset state to all zeros
      state <= '0;
    end else begin
      // Shift left by 1 bit and update the LSB with the feedback
      state <= {state[WIDTH-2:0], feedback};
    end
  end

  // Output the current state as the LFSR output
  assign lfsr = state;

endmodule
```

3.2 CacheLRU Changes

To transform the existing cache replacement policy from LRU to a pseudo-random replacement policy using an LFSR, several changes were made to the cacheLRU module.

The most fundamental change involved introducing a parameterized LFSR module that generates a pseudo-random index. This module is responsible for producing a random selection among the cache ways when a cache replacement is necessary. The LFSR module is parameterized to accommodate various widths, allowing flexibility in selecting the number of bits used for random index generation.

Instead of relying on LRU information to determine the victim way for cache replacement, the module now utilizes the pseudo-random index generated by the LFSR. The LFSR output represents a randomly selected cache way, effectively implementing a pseudo-random replacement policy. The random index is computed based on the LFSR output, bypassing the need for maintaining LRU state information.

The logic responsible for computing the victim way is modified to incorporate the pseudo-random index generated by the LFSR module. When a cache miss occurs, the LFSR-generated index is used to select the victim way among the cache sets, ensuring a pseudo-random replacement policy. This change seamlessly integrates the LFSR-based random replacement mechanism into the existing cacheLRU module architecture.

By implementing these changes, the cache replacement policy is transformed from LRU to pseudo-random, leveraging the properties of the LFSR to introduce randomness into the cache replacement process. This modification enhances the cache's ability to distribute cache misses more evenly across cache ways, potentially improving overall cache performance in scenarios with non-uniform memory access patterns.

4 Testing Strategy

In this project, a meticulous testing strategy was pivotal to ensure the reliability, functionality, and performance of the cache replacement policies integrated into the CV-Wally processor. The testing approach underscores an iterative design process, where refinements are made incrementally based on feedback from simulation and FPGA-based hardware validation.

Initially, cache replacement policies are subjected to rigorous testing within the simulation environment. The simulation ensures that the code will run and the bitstream will generate properly later. It also ensures that there are no unforeseen issues with our code that may only appear when parameters are changed, like using a different number of ways.

Upon satisfactory simulation results, the code undergoes synthesis to generate hardware descriptions suitable for FPGA implementation. The synthesized hardware descriptions are then deployed onto FPGA platforms such as the Arty A7 100 board. FPGA testing involves programming the board with the synthesized configurations, including various cache replacement policies.

Feedback from both simulation-based validation and FPGA testing informs iterative refinements to the cache architecture and replacement policies. Test results guide iterative improvements aimed at enhancing cache efficiency and overall system performance. The testing strategy follows a continuous improvement cycle, where each iteration builds upon the insights gained from previous tests and refinements. Iterative development could foster continuous enhancement of cache performance and efficiency throughout the project lifecycle.

By adhering to a comprehensive testing strategy and iterative design process encompassing simulation-based validation and FPGA testing, we can systematically evaluate, refine, and optimize cache replacement policies within the CV-Wally processor, ultimately enhancing system performance and efficiency and ensuring our code works reliably.

5 Evaluation

5.1 Test Results

The CV-Wally code runs ons the Arty A7 FPGA and the cache policy succeeds at randomizing replacement as designed.

5.2 Performance Analysis

LRU replacement policy is widely used due to its simplicity and effectiveness in capturing temporal locality, a common characteristic of memory access patterns in many applications. By evicting the least recently used cache lines upon a cache miss, LRU aims to retain frequently accessed data in the cache, reducing cache misses and improving overall performance. However, LRU may exhibit suboptimal behavior in scenarios where access patterns deviate from strict temporal locality, such as irregular or cyclic memory access patterns. In such cases, LRU may evict cache lines prematurely, leading to increased cache misses and decreased performance.

In contrast, pseudo-random replacement policies, such as those implemented using LFSR's, introduce an element of randomness into the cache replacement process. By selecting cache lines for eviction based on pseudo-random indices generated by the LFSR, these policies aim to mitigate biases inherent in deterministic replacement policies like LRU. Pseudo-random replacement policies can be particularly effective in scenarios with irregular or unpredictable memory access patterns, where traditional replacement policies may struggle to capture access behavior accurately. However, pseudo-random replacement policies may introduce additional overhead due to the need for LFSR computation and maintenance, potentially impacting overall system performance.

One of the key advantages of LRU over pseudo-random replacement policies lies in its ability to exploit temporal locality more effectively. LRU inherently prioritizes recently accessed data for retention in the cache, making it well-suited for applications with strong temporal locality characteristics. In contrast, pseudo-random replacement policies do not explicitly consider access history and may evict recently accessed data if randomly selected by the LFSR. Consequently, pseudo-random replacement policies may exhibit poorer performance than LRU in scenarios where temporal locality plays a significant role in access patterns.

However, pseudo-random replacement policies excel in scenarios where access patterns lack strong temporal locality or exhibit cyclic behavior. In such cases, LRU may struggle to adapt to changing access patterns, leading to suboptimal cache utilization and performance degradation. Pseudo-random replacement policies, by introducing randomness into the replacement process, can distribute cache evictions more evenly across cache lines, reducing the likelihood of hot spots and improving overall cache performance in these scenarios. While LRU is effective in capturing temporal locality and may offer superior performance in many scenarios, pseudo-random replacement policies provide a valuable alternative for applications with irregular or unpredictable access patterns, helping to mitigate biases inherent in deterministic replacement policies and improve overall cache performance.

5.3 Summary

The project focuses on enhancing the cache architecture of the CV-Wally processor through the integration and evaluation of alternative cache replacement policies. Beginning with the setup of development environments and repositories, we meticulously navigate through iterative design and testing phases to refine the cache design.

6 References

[1] GitHub Repository

https://github.com/openhwgroup/cvw/