ACA Project - Final Report

Scoreboarding and Register-Renaming in Risc-V processsor

Aryan (2022113007) and Abhiram (2022113011)

November 28, 2024

Contents

1 Problem Statement

The focus of this project is to implement **Scoreboarding** and **Register Renaming**.

1.1 Scoreboarding

Scoreboarding is a hardware-based dynamic scheduling technique that tracks the availability of functional units and registers during runtime. It ensures that instructions are executed only when their operands are ready, thus preventing pipeline stalls caused by RAW hazards.

1.2 Register Renaming

Register Renaming is a technique that addresses name dependencies between instructions by assigning multiple physical registers to the same architectural register. This eliminates WAW and WAR hazards, enabling smoother and more efficient out-of-order execution.

This project involves modifying an existing open-source RISC-V core (Flute) to integrate these techniques.

Through this project, we aim to demonstrate how Scoreboarding and Register Renaming can significantly enhance instruction throughput, reduce execution latency, and resolve data hazards in a RISC-V processor.

2 Requirements Analysis

The successful implementation of Scoreboarding and Register Renaming in a RISC-V processor requires a comprehensive understanding of the architecture, along with the necessary hardware and software tools. This section outlines the key requirements for the project.

2.1 Processor Core Selection

The project requires selecting a processor core that can be extended with advanced techniques. Options include modifying an existing open-source core such as BOOM, TOoBA, or RISC-V O, or designing a custom in-order core.

2.2 Software and Toolchain Requirements

- Bluespec Verilog Compiler: For developing and simulating the FLUTE processor.
- RISC-V GCC Toolchain: To compile and run RISC-V assembly and C programs for testing the processor's functionality.
- Version Control System: Git or equivalent for maintaining and collaborating on the project codebase.

2.3 Conceptual Requirements

The project requires in-depth knowledge of:

- Pipeline stages in RISC-V architecture.
- Dynamic scheduling techniques, including Scoreboarding.

- Register renaming logic and the distinction between physical and architectural registers.

2.4 Verification and Validation Needs

To ensure correctness and evaluate performance, the project will:

- We need Proper test cases to validate functionality and identify potential errors..
- Evaluate performance improvements using key metrics like instruction throughput etc..

3 Related Work

The implementation of advanced pipelining techniques, such as Scoreboarding and Register Renaming, has been a focus of research and development in processor design. Several projects and open-source cores have contributed significantly to this field, providing a foundation for further innovation.

FLUTE Flute is a simple RISC-V CPU designed with a 5-stage in-order pipeline, targeting low-end applications requiring MMUs and moderate performance. It includes pipeline stages for Fetch, Decode, Execute, Memory Access, and Writeback, with specialized handling for branch prediction, compressed instructions, and bypassing logic. Additionally, Flute incorporates near-memory subsystems with L1 caches and MMU support, making it suitable for embedded systems and similar use cases

To gain insights into the code's structure, we referred to [3] Flute GitHub repository.

TOOBA (Totally Out-of-Order and Beyond Architecture) is a RISC-V-based processor that demonstrates a highly modular and extensible design capable of out-of-order execution. It incorporates advanced scheduling and resource management mechanisms, making it a valuable reference for implementing Scoreboarding and Register Renaming. The TOoBA project highlights how dynamic scheduling can optimize resource utilization and reduce pipeline stalls[4] Tooba GitHub repository.

RISCY-OOO O (Out-of-Order) is another open-source RISC-V core designed to implement out-of-order execution. It uses a reorder buffer, physical registers, and a reservation station to manage dependencies and scheduling dynamically. The integration of these components ensures efficient hazard resolution and supports a high degree of instruction-level parallelism. RISCY-OOO GitHub repository.

These existing implementations provide a practical framework and demonstrate the feasibility of incorporating advanced techniques into RISC-V processors. Building on their principles, this project aims to design a custom implementation of Scoreboarding and Register Renaming, with a focus on improving performance metrics such as instruction throughput and execution latency. The knowledge gained from TOoBA and RISC-V O serves as an essential guide for addressing the challenges associated with these techniques.

4 Your Contribution

As the project was divided in three phases. Below is an overview of our primary contributions:

4.1 Phase 1: Exploration and Selection of CPU Core (20th October to 26th October)

In this phase, we explored several CPU core projects before settling on FLUTE (a RISC-V core based on Bluespec Verilog). We evaluated the following options, each with their own set of limitations:

To understand the majority of Bluespec syntax and code, we relied on [1]

- **BOOM (Berkeley Out-of-Order Machine)**:
  - The documentation is excellent, but it doesn't align well with our project's goals.
  - Features such as register renaming and reorder buffers are pre-implemented, which didn't fit our project's requirements.
- **Chipyard**:
  - While the project is well-maintained and extensive, it is overly complex for our needs.
  - The scale of Chipyard is too large for the simpler tasks we were attempting.
- **RISC-V Mini**:
  - This project had been archived for several years and lacked official support, making it an unreliable choice.
- **Rocket Chip**:
  - While Rocket Chip serves as the basis for many processors based on Chisel, its codebase has been archived and is no longer maintained, which made it unsuitable for our needs.

After exploring these options, we settled on **FLUTE** due to its relative simplicity, being based on **Bluespec Verilog**. Additionally, Bluespec is easier to use compared to languages like Chisel, which made it a good fit for our project.

- **Why FLUTE?**:
  - FLUTE is one of the most popular RISC-V cores based on Bluespec Verilog.
  - It is more recently maintained compared to alternatives like **Piccolo** and **Rocket**.

Next, we installed **Bluespec System Compiler (BSC)** and compiled FLUTE from source.

4.1.1 Compiling FLUTE

Step 1. Install Bluespec System Compiler (BSC)

To compile FLUTE, you need to first install the Bluespec System Compiler (BSC). You can either download the binary files or compile it from source. For instructions, visit the official BSC repository: https://github.com/B-Lang-org/bsc.

Once you have BSC installed, add the 'bsc' binary files to your system's PATH to make the compiler accessible from anywhere in your terminal.

Step 2. Clone the FLUTE Repository

Clone the FLUTE repository from GitHub:

Step 3. Run Tests

After compilation, you can run the tests to ensure everything works correctly:

```
# Clone the FLUTE repository from GitHub
git clone https://github.com/your-repo/flute.git

# Navigate to the builds directory and select a build configuration
cd flute/builds/any_build_of_your_choice_preferrably_rv64

# Compile the FLUTE code
make all

# Run the tests to verify the compilation
make test
```

4.2 Phase 2: Register Renaming and Scoreboarding Implementation

Implemented Register Renaming for Architectural Registers

We implemented register renaming to handle the mapping of architectural registers to physical registers. This involved exploring the interface for register renaming and understanding the methods in the scoreboarding interface. While we haven't fully explored all aspects, we now have a working understanding of the process.

4.2.1 Register Renaming Overview

Purpose:

- Manage mapping of architectural registers to physical registers.
- The rename stage identifies physical registers for sources and renames destination registers.
- The commit stage recycles unused physical registers and manages mis-speculation recovery.

Physical Register File:

- The physical register file contains values for each physical register and a presence bit for each physical register value. This is not visible to the user.
- Instructions at the renaming stage (decode stage) unset the presence bits of destination physical registers.
- Instructions that finish execution write data into the physical register file and set the presence bits simultaneously.

A significant portion of the code is adapted from [2]

```
// PhyRIndex is the index of a physical reg
// SupSize is the superscalar size
interface RFileWr;
  method Action wr(PhyRIndx rindx, Data data);
endinterface
interface RFileRd;
  method Maybe#(Data) rd1(PhyRIndx rindx);
  method Maybe#(Data) rd2(PhyRIndx rindx);
  method Maybe#(Data) rd3(PhyRIndx rindx);
endinterface
interface SbSetBusy;
  method Action set(Maybe#(PhyRIndx) dst);
endinterface
interface RFileSbCons#(numeric type wrNum, numeric type rdNum);
  interface Vector#(wrNum, RFileWr) write;
  interface Vector#(rdNum, RFileRd) read;
  interface Vector#(SupSize, SbSetBusy) setBusy;
endinterface
module mkRFileSbCons(RFileSbCons#(wrNum, rdNum));
  // module implementation
endmodule
```

Interface of physical register file and conservative scoreboard

Register Rename Table:

- The renaming table manages the mapping from architectural registers to physical registers.
- The rename stage queries this table to get the physical registers for the source registers of instructions and rename the destination registers.
- The commit stage calls this table to recycle unused physical registers. Upon misspeculation, the renaming table also needs to be recovered.

```
interface RTRename;
  method PhyRegs getRename(ArchRegs r);
  method Action claimRename(ArchRegs r, SpecBits sb);
  method Bool canRename;
endinterface
interface RTCommit;
  method Action commit;
  method Bool canCommit;
endinterface
interface RegRenamingTable;
  interface Vector#(SupSize, RTRename) rename;
  interface Vector#(SupSize, RTCommit) commit;
  interface SpeculationUpdate specUpdate;
endinterface
module mkRegRenamingTable(RegRenamingTable);
  // module implementation
endmodule
```

Interface of the renaming table

Conflict Matrix:

- A conflict matrix handles conflicts in methods being called simultaneously.
- The conflict matrix determines the order in which methods in the queues are executed if they appear in the same clock cycle.
- The priority of the `CLAIMRENAME` function is higher than the `GETRENAME`, and commit is allowed only after all claims are done.

4.2.2 Implementation Details: Issues Faced

Issues in FLUTE:

- In FLUTE, the issue stage is divided into the fetch and decode stages. During the decode stage, when the `FC_DECODE` function is called (a built-in function in the Bluespec library), we get the destination register directly. To resolve this, I had to add a variable to the decode structure, called `PHYREG`, and save the claim register there instead.
- In out-of-order processors, a commit stage is used instead of a write-back stage, but it consists of various checks which we haven't implemented yet.
- In the CPU stage 3 (write-back stage), the ALU output is stored in the architectural register, which is fine, but we must unset the present bit of the physical register to mark it as free. This call needs to be made manually.

Scoreboarding Implementation Plan:

- Once we figure out how to execute multiple instructions, we will need to map all the functional units.
- In phase 2 implementation, the decode stage assigns a unique ID to each process being executed. We can use this to handle more epoch conditions.

4.2.3 Challenges in Phase 2

- Making the processor out-of-order by editing the execute file in pipelining.
- Properly implementing the register file commit logic.

4.3 Phase 3: Implementing the Scoreboard

In Phase 3, the scoreboard mechanism was implemented to manage register read/write hazards effectively. The scoreboard is designed as an array for evevry 1-bit register. It ensures that instructions do not proceed prematurely, thereby maintaining data consistency in the pipeline.

Mechanism

The scoreboard operates as follows:

- When an instruction enters the **Register-Read** stage and writes to a register (e.g., `x7`), the corresponding bit in the scoreboard (e.g., bit 7) is set to `1`, marking the register as *busy*.
- Once the instruction reaches the **Retire** stage and updates the register, the scoreboard bit is reset to `0`, indicating the register is no longer busy.
- If a subsequent instruction tries to read the same register while it is marked as busy, it stalls in the **Register-Read** stage until the scoreboard clears the condition.

Handling Pipeline Bubbles

While an instruction is stalled in the **Register-Read** stage due to a busy register, the pipeline stage immediately following it (e.g., **Execute**) may become vacant. This condition is referred to as a *pipeline bubble*. Although bubbles reduce the throughput of the pipeline temporarily, the scoreboard ensures correct program execution.

4.4 Processor Out-of-Order Execution

Although we were unable to fully transition to out-of-order execution, we successfully demonstrated the implementation of proper scoreboarding,

5 Obstacles Faced

During the course of the project, we encountered several challenges, particularly in understanding and implementing the complexities of advanced pipelining techniques:

- **Initial Learning Curve:** Adapting to Bluespec Verilog syntax and compiler workflows required substantial effort, as we had limited prior experience with this hardware description language.
- **Integration with FLUTE:** Modifying FLUTE's existing architecture, especially adding new pipeline stages and components for register renaming, introduced unforeseen compatibility issues. For instance:

  - Resolving conflicts in the decode stage where destination registers had to be replaced with physical registers.
  - Manual calls were needed to update the presence bits in the physical register file, which complicated integration.

- **Pipeline Modifications:**
  - Although we explored the potential transition to an out-of-order execution model, we decided to retain the in-order pipeline structure due to the extensive changes required in the execution and commit stages.
  - Introducing new components like the scoreboard and ensuring they interacted correctly with other pipeline stages led to debugging challenges even within the in-order execution model.

- **Incomplete Git Repository Setup**

  The provided Git repository for FLUTE was not properly configured, leading to several issues during setup and compilation. One notable problem was the need to modify the `Makefile` by introducing the `-mcmodel=medium` flag to ensure successful compilation. Without this change, the code failed to build correctly, which required significant time and debugging to identify.

6 Proposed Approach

To address the challenges of implementing we adopted a structured and incremental approach.

6.1 Exploration and Architecture Selection

The initial step involved an in-depth evaluation of existing RISC-V cores to identify one that could serve as a robust foundation for our implementation. After thoroughly analyzing projects like BOOM, Rocket Chip, and RISC-V Mini, we selected FLUTE for its balance of simplicity and maintainability. Its modular design and use of Bluespec Verilog offered a powerful platform to implement our modifications.

6.2 Incremental Development of Core Features

Our implementation focused on integrating key features systematically to avoid disrupting the pipeline's functionality:

- **Scoreboarding:** We implemented a scoreboard to manage read/write hazards by tracking the availability of registers. The scoreboard was designed to stall dependent instructions until data dependencies were resolved, ensuring correct program execution while maintaining data consistency across the pipeline stages.

6.3 Incomplete Git Repository Setup

The repository's incomplete setup required manual intervention to ensure proper compilation. After extensive debugging, we identified the missing `-mcmodel=medium` flag as a critical issue in the `Makefile`. This flag, necessary for supporting larger memory models, was introduced to resolve the build errors. The solution was found on StackOverflow, which provided a clear explanation of the issue and guided us in implementing the fix effectively. This adjustment allowed the project to proceed smoothly, ensuring the compilation environment was stable.

7 Potential Challenges

- **Complexity of Out-of-Order Execution:** The transition to out-of-order execution might introduce new issues, such as reorder buffer management and instruction window scheduling, which could require additional debugging and refinement.
- **Scalability:** Expanding the architecture to support out of order execution may lead to significant increases in hardware resource usage and design complexity.
- **Debugging and Verification:** Debugging a hardware design, especially with speculative execution, is inherently more difficult than debugging software.
- **Limited Bluespec Documentation and Community Support:** Bluespec Verilog, while powerful, lacks extensive community resources and documentation compared to more widely used hardware description languages like Verilog or Chisel. This made it difficult to find external guidance or examples, prolonging the debugging and development process.

8 Conclusion

This project focused on implementing advanced CPU architectural features using FLUTE. Through hands-on development and tackling key challenges, we deepened our understanding of processor functionality, pipeline behavior, and the complexities of CPU design.

Crucial component was the integration of a scoreboard to manage read/write hazards. The scoreboard ensured data correctness by stalling dependent instructions when necessary, preventing hazards such as read-after-write dependencies. While this introduced occasional pipeline bubbles, it maintained the integrity of the pipeline and demonstrated a practical solution for hazard management.

The project highlighted the intricate trade-offs involved in CPU design, such as balancing correctness with performance. By building on the capabilities of the FLUTE core and utilizing Bluespec Verilog, we successfully implemented essential mechanisms that are foundational to advanced processor architectures. This work provides a strong basis for further exploration of out-of-order execution, pipeline optimization, and other advanced features in CPU design.

References

[1] Lennart Augustsson, Jacob Schwartz, and Rishiyur S. Nikhil. *BluespecTM Language Definition.* LATEX'd January 14, 2019, File revision 1.176 (as of August 21, 2003). Bluespec, Inc. 2003.

[2] C-Sail. *Composable Building Blocks to Open up Processor Design*. https://github.com/csail-csg/RiscyOO_design_doc.

[3] RS Nikhil. *Microarchitecture overview of RISC-V Processors*. https://github.com/bluespec/Flute/blob/master/Doc/Microarchitecture/Microarchitecture.pdf. Since 2005.

[4] Sizhuo Zhang et al. "Composable Building Blocks to Open up Processor Design". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 68–81. DOI: 10.1109/MICRO.2018.00015.