Coreographer

Resource sharing and Distribution in Heterogeneous Multi Core Architectures

CS2323 - Computer Architecture

E. Mihir Divyansh
Roll Number: EE23BTECH11017
ee23btech11017@iith.ac.in

October 2, 2025

Contents

List of Figures

List of Tables

Listings

Abstract

This report documents the design and implementation of a heterogeneous RISC-V based multi-core architecture that demonstrates register file sharing, and instruction distribution techniques inspired by GPU streaming multiprocessors. I try to show how context switching and heterogeneous compute capabilitiescan show performance improvements.

Keywords: RISC-V, multi-core, latency hiding, workload distribution

Chapter 1

Introduction

1.1 Overview

At the most fundamental level, any processor is designed to execute a sequence of instructions. This typically involves three things: fetching and decoding instructions, performing computation, and interacting with memory. While these capabilities are sufficient for general-purpose workloads, some applications often demand higher performance, particularly when dealing with data-parallel tasks.

One approach to improving performance is parallelism. By deploying multiple processing cores, systems can execute independent or partially independent streams of instructions concurrently. Traditional multi-core processors achieve this by replicating cores, each with its own private resources. This improves throughput but comes at the cost of increased hardware area and resource duplication. Consider, for example, a quad-core processor where each core has its own register file. If the workload consists of four independent instructions, all cores can be kept busy, and throughput improves. However, if due to a dependence, only 1 instruction can be executed, three of the cores remain idle while still having their unused compute and memory resources. In such a scenario, hardware resources are underutilized. This motivates the study and design of architectures that allow multiple cores to share resources and adapt to varying levels of parallelism in the workload.

This project explores an alternative approach: accelerating a subset of parallel workloads by combining multiple RISC-V cores with a shared register file. Inspired by GPU streaming multiprocessors, the design explores mechanisms for instruction distribution, register file sharing, and centralized scheduling. The goal is to demonstrate that resource sharing and context-switching techniques can improve utilization and efficiency without incurring the full hardware overhead of a homogeneous multi-core design.

1.2 Scope of the Project

The project involves the following tasks:

1. Port an existing single-core RISC-V processor implementation to FPGA hardware and verify functionality.

2. Modify the baseline design to support resource sharing (shared register file) and instruction distribution across multiple cores.
3. Design and implement a scheduler/dispatcher to manage instruction allocation and context switching.
4. Integrate a configurable number (4-8) of lightweight RISC-V cores with the shared register file and scheduler at the system level.
5. Identify representative workloads that can benefit from this architecture.
6. Demonstrate the working design on FPGA hardware.
7. *(Optional)* Implement and execute benchmarks on both baseline and proposed designs.
8. *(Optional)* Compare performance improvements against baseline and homogeneous multi-core systems.

Input Specification

The input to the system will be assembly programs that are pre-structured to align with the proposed architecture. These programs will be prepared to reduce scheduler complexity, allowing focus on architectural evaluation rather than handling uncommon corner cases during scheduler design.

1.2.1 Architecture Overview

The proposed system consists of:

- **Heterogeneous worker cores:** 4-8 configurable RISC-V cores with different ISA extensions (e.g., RV32I, RV32IM, RV32IMF). All cores maintain the same bitwidth to avoid bus complexity.
- **Shared register file:** A large, banked register file (512-1024 32-bit registers) partitioned into contexts. Each context provides 32 registers, matching the standard RISC-V ISA. The shared design enables context migration between cores without data copying.
- **Instruction cache:** Shared or per-core instruction caches. Data caches are omitted in the initial design to reduce complexity.
- **Centralized scheduler:** A dedicated hardware scheduler (see Section 1.2.2) that manages micro-task dispatch and context allocation.

1.2.2 Scheduler Specifications

The scheduler is implemented as a **dedicated hardware core**, distinct from the worker RISC-V cores. It executes a customized, lightweight version of the RISC-V ISA optimized for scheduling operations and context management, functioning as a pseudo-hardware OS scheduler.

Context Management Model

A **context** in this architecture consists of:

- A 32-register slice allocated from the shared register file
- Program counter (PC) value maintained at the scheduler
- Status flags (ready, stalled, incapable)

When a context must migrate due to long-latency operations (e.g., memory access), the scheduler can reallocate the context's register slice to any idle, capable core without copying register contents—the physical registers remain in the shared file.

Core Responsibilities

The scheduler manages:

- **Micro-task dispatch:** Maintain a queue of ready micro-tasks and allocate them based on core capabilities and availability.
- **Context management:** Maintain per-context metadata including register file slice mapping, PC values, and status (ready, incapable, stalled on memory, stalled on compute) via a Core Capability and Availability Table.
- **Workload distribution:** Match micro-tasks to cores based on required ISA extensions and current availability.
- **Register allocation and reclamation:** Allocate physical register-file slices to active contexts and reclaim on context retirement. Track occupancy and commits.
- **Context migration:** Enable context switching between cores when long-latency events occur and the original core becomes unavailable but another capable core is idle.
- **Observability and counters:** Expose counters for dispatched micro-tasks, context switches, context migrations, register-file utilization, core idle time, and stalls.

Register File Design Considerations

The shared register file presents several design challenges:

- **Banking:** Register file will likely be banked to reduce port contention and area overhead. Specific banking strategy to be determined based on access patterns and referenced literature [**?**, **?**].
- **Timing:** May use BRAMs with negative-edge latching to achieve single-cycle apparent latency if running at slightly reduced frequency.
- **Sizing:** 512-1024 registers total, supporting 2-4 contexts per core for 4-8 cores respectively.

1.2.3 Design Constraints and Simplifications

To maintain project feasibility within time constraints, the following simplifications are made:

- **No data caches:** The system omits L1 data caches to reduce design complexity. Only instruction caches are included.
- **Micro-task level scheduling:** Rather than instruction-level scheduling (impractical in hardware), the system schedules pre-analyzed micro-tasks with explicit dependencies.
- **Same bitwidth ISA:** All cores use the same register bitwidth (32 or 64) to avoid bus width mismatches.
- **Pre-annotated programs:** Input programs are prepared with capability requirements and dependency hints to simplify scheduler logic.

1.2.4 Success Criteria

Functional Requirements

- The multi-core system functions correctly on FPGA hardware.
- Scheduler successfully distributes micro-tasks across heterogeneous cores based on capability matching.
- Context migration works correctly—contexts can be reassigned to different capable cores without data corruption.
- System maintains or improves performance compared to single-core baseline for suitable parallel workloads.
- Register file sharing operates without deadlocks, bank conflicts cause acceptable (not catastrophic) performance degradation.
- Observability counters accurately report system behavior for analysis.

Chapter 2

Background and Previous Work

Before I embark on designing the blocks for the project, I need to study and understand a few concepts. The following sections document my reading process.

2.1 Streaming Multiprocessors in GPUs

Streaming Multiprocessors are the general purpose repeating blocks found in NVIDIA GPUs They have the necessary units to perform massive parallel computations of different types. They can perform

- Float32, Float64 operations
- Integer operations
- Texture operations (DSP ops, Fast Random Memory Access)
- Tensor Operations (Matrices)
- Other Special Functions (sin, cos, exp)

2.2 Existing Architectures

Chapter 3

System Specifications

3.1 Functional Requirements

Instruction Set Architecture

- Instruction format: [specify format]
- Instruction types: [list types]
- Register file: [specify size and organization]
- Memory addressing: [describe addressing modes]

3.2 Performance Requirements

Performance Targets

- Target clock frequency: X.X GHz
- Maximum CPI: X.X
- Pipeline depth: X stages
- Cache miss penalty: ¡ XX ns

Chapter 4

Design and Implementation

4.1 System Architecture

Chapter 5

Experimental Results

5.1 Performance Analysis

5.2 Analysis and Discussion

Chapter 6

Conclusions and Future Work

6.1 Project Summary

6.2 Future Enhancements

Appendix A

Source Code Listings

Appendix B

Additional Test Results

Appendix C

Design Documents

Bibliography

[1] Patterson, D. A., & Hennessy, J. L. (2017). *Computer organization and design: the hardware/software interface*.

[2] GPU Glossary by Modal. https://modal.com/gpu-glossary.