# Freeflow Core

## Project Report

Submitted in partial fulfillment of the requirements
for the completion of the course

**EE748: Advanced Topics in Computer Architecture**

by

**Team - 1**

**Neeraj Prabhu (200070049)**
**Anubhav Bhatla (200070008)**

Under the guidance of
**Prof. Virendra Singh**


**Department of Electrical Engineering**
**Indian Institute of Technology Bombay**
**October 2023**

# Contents

# List of Figures

# Chapter 1

# Introduction

Processor cores have evolved from simple in-order designs to complex superscalar out-of-order machines. There have been several efforts to expose instruction-level parallelism (ILP) so that all the execution units can be occupied and hence improve single-threaded application performance. This led to the emergence of multi and many-core processors, which introduced challenges such as increasing performance within a given power budget. Such advancements were also accompanied by an increase in memory hierarchy levels. Hence, the focus shifted towards hiding memory latency in the execution of programs. Since ILP-extracting techniques automatically expose memory hierarchy parallelism (MHP) as well, out-of-order cores are naturally successful in coping with this problem. However, OoO cores have a high area and power budget. This has hence created two diverse types of cores: in-order cores, with their high energy efficiency but low ILP extraction, and out-of-order cores, which give a high performance but consume a lot of power as well. The ideal many-core building block is, therefore, an energy-efficient core that can still maximize extraction of memory hierarchy parallelism, a combination that neither traditional in-order nor out-of-order processors provide. One such technique is the slicing of instructions, either on software or on hardware, in order to execute these slices out-of-order with respect to the general instruction flow.

## 1.1 sOoO Cores

Slice-out-of-order (sOoO) cores are a new class of cores which help minimize the energy cost of the MLP-exploiting techniques used by traditional OoO cores. They achieve this by adding just enough OoO functionality so as to extract MLP. Slices, or groups, of MLP-generating instructions, are constructed that contain the address-generating instructions (AGI) leading up to loads and/or stores. These slices are executed out-of-order with respect to the remaining instructions. Executing these slices out-of-order compared to the rest of the instructions helps these MLP-extracting instructions bypass the rest of the potentially stalled instructions. Even though these sOoO can only extract limited MLP from the program, they are able to bridge a large chunk of the performance gap between in-order and out-of-order processors. And, since they only require a few hardware additions on top of in-order cores, they are able to achieve such high performance at a fraction of the energy cost of OoO processors. Load-Slice Core [1] is a state-of-the-art implementation of sOoO which provisions different queues for compute and memory instructions and implements an iterative backward dependency analysis technique to identify AGIs in a hardware-efficient manner.

Even though sOoO cores help massively bridge the MLP gap between InO and OoO cores, they still suffer from frequent stalls for two reasons. First, slices can depend on

each other, which may limit the MLP-extraction possibilities. Consider the case where the head of the memory-access pipeline is stalled due to a dependency on a load in another slice. Now, even if a younger instruction in the memory-access pipeline, which belongs to an independent slice, is ready to execute, it will get blocked. The Freeway Core [2] tackles this problem by provisioning an additional pipeline which buffers these dependent slices and allows smooth execution of independent slices in the memory-access queue.

Second, an instruction stalled at the head of the compute pipeline due to its dependency on a long-latency load in the memory-access queue will lead to unnecessary stalls for younger, ready-to-execute instructions in the compute pipeline. This necessitates special treatment for such blocking instructions so that younger instructions can progress. Freeflow Core [3] is one such architecture which bypasses these blocking instructions to another queue and thus reduces unnecessary stalls in the compute pipeline.

## 1.2 Freeflow Core

Freeflow proposes a micro-architectural technique to detect and bypass blocking instructions in the compute pipeline to another queue, called the bypass queue. Freeflow core focuses on instructions which stall the compute pipeline because of dependency on long latency loads in the memory access pipeline. Such instructions block the execution of younger instructions in the compute pipeline, which may have their operands ready. These kinds of blocking instructions are moved to another queue (separate from the compute and memory access queue), which dispatches instructions only when ready and, at the same time, prevents blocking the younger instructions from execution. Freeflow core maintains a freeflow instruction slice table (FIST) to identify instructions dependent on unresolved memory instructions.

# Chapter 2

# Literature Survey

*The Decoupled Access/Execute architecture* [4] uses two separate pipelines: one for memory accesses to fetch operands and store results and another for execution of operands to generate the required results. Hence, it improves performance without a high design complexity. Additionally, it hides the memory communication delays by using another pipeline to run memory operations.

*The Load Slice core* [1] implements an iterative backward dependency analysis algorithm to identify address-generating instructions which eventually lead to memory accesses. All such address-generating instructions are moved to the bypass queue along with the memory-access instructions. Such an architecture gives a 53% improvement in performance over an in-order core, bridging more than half the gap between an in-order core and an OoO core. It is 43% more energy efficient as compared to an in-order core and 4.7x more energy efficient compared to an OoO core.

*Freeway core* [2] exploits MLP by identifying instructions which block the memory access pipeline because of some dependency and hence prevent the execution of younger instructions which may be ready. It tracks such dependent slices and moves them to a third queue so that the independent slices can be executed. This gives a performance benefit of 53% over an in-order core and is within 7% of the MLP limits of full OoO execution.

*Freeflow core* [3] identifies instructions which block the compute pipeline due to unresolved memory access instructions. Such instructions are moved to another queue depending on the value of a counter which stores the number of cycles the instruction blocks a particular queue. This type of architecture gives 89.4% better performance than an InO core and lags behind the OoO core by 14%. It is 100%, 18% and 46% more energy efficient than OoO, LSC and in-order cores, respectively.

# Chapter 3

# Overview of Freeflow Core

Freeflow core aims to exploit the inherent ILP in the compute pipeline of sOoO architectures. It does so by detecting *high-latency* instructions that are dependent on unresolved memory instructions in the memory-access queue (*B-Q*) and could potentially stall the compute queue (*A-Q*) once it reaches the head of the queue. Once such instructions are detected, they are guided to a different execution path, i.e., an additional queue called the *C-Q*.

## 3.1 Detecting high-latency instructions

Instructions which are dependent on long latency memory instructions in the B-Q can stall the A-Q for a long period of time. Freeflow detects such instructions using a *Set Counter* at the head of the A-Q. This is used to count the number of cycles an instruction remains stalled at the head of the A-Q. If this instruction remains stalled for more than six cycles, it is registered in an additional hardware structure called the Freeflow Instruction Slice Table (*FIST*). In the next iteration, these high-latency instructions get a hit in the FIST and are bypassed to the C-Q.

An additional counter called the *Reset Counter*, is used at the head of the C-Q to count the number of cycles an instruction remains stalled at the head of the C-Q. In the case where this counter is below the threshold of six cycles, the FIST entry for this instruction is invalidated, as it is no longer a high-latency instruction. Finally, all instructions are committed in order with the help of the scoreboard.

Figure 3.1 provides an overview of the Freeflow Core architecture and highlights its additions and modifications on top of the state-of-the-art Load-Slice Core.

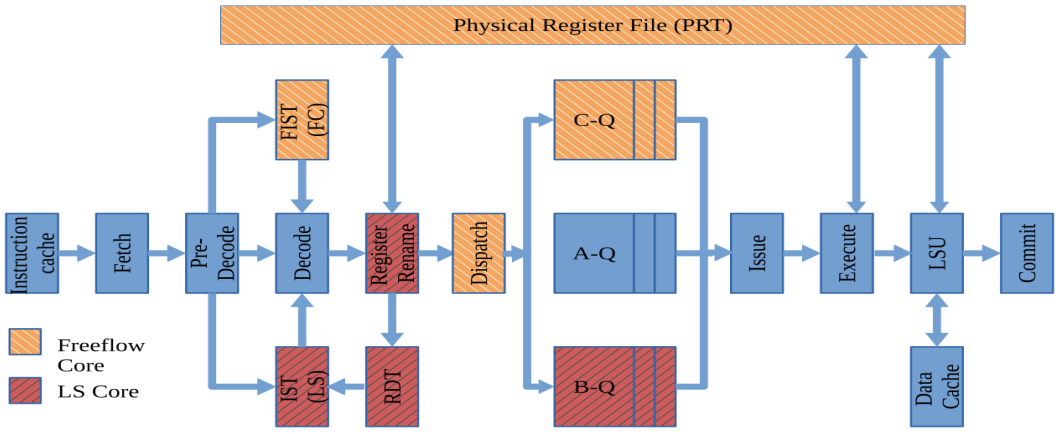


Figure 3.1: An overview of the Freeflow Core Micro-architecture

# Chapter 4

# Implementation

We are implementing the Load Slice core and the Freeflow core using Gem5 [5], a cycle-accurate micro-architecture simulator. We have used the in-order (gem5 MinorCPU) and the out-of-order (gem5 O3CPU) cores as baselines for comparison.

We are modifying the O3CPU model in gem5 to incorporate the A-Q, B-Q and C-Q along with the instruction slice table, the register dependency table and the freeflow instructions slice table.

## 4.1 Issuing in dispatch order

We used the O3CPU on gem5 and converted the dispatch to in-order. We decided to use the O3CPU instead of the in-order MinorCPU because we needed the property of register renaming in the Freeflow core. This was conveniently used form the O3CPU model rather than creating a new register renaming block in the minor CPU model.

The gem5 implementation of the dispatch stage consists of an instruction list and a priority queue consisting of the instructions executed. For the out-of-order execution, the instruction at the head of the priority queue is selected. However, we modified this to control the dispatch such that the ready queue waits till its head has the oldest instruction which has not been dispatched. The instruction list stores every instruction that is fetched by the CPU till it has been retired. Hence it fills up in program order. Hence, we checked the variable `isIssued()` to check the issue status of the instructions from the front of the list. When we reached the first instruction, which was not issued, we labelled it as the oldest instruction in the sequence. The head of the ready instructions priority queue was checked to be equal to the oldest instruction. The instruction was dispatched only if the equality was satisfied. Since the ready instructions struct was a priority queue, the head was guaranteed to be the oldest instruction after a few cycles, if not immediately.

Once we implemented this, we noticed the presence of squashing instructions, which were causing some issues, especially because they caused the instruction list entries to get deleted but not the entries in ready instructions. Hence, we tackled this problem by using the `isSquashed()` object of the instruction. In the cases when the instruction at the head of the priority queue was squashed, the entry was removed from the priority queue, and the next entry was checked.

## 4.2 Implementation of Multiple Queues

We decided to keep multiple instruction lists for this purpose, such that each incoming instruction was checked to be either a load, store or memory reference instruction and was passed to either of the lists accordingly. The oldest instruction from these two lists

was selected for issue, depending on whether they were ready or not. The ready queues were already partitioned based on the functional unit and execution pipeline to be used. Hence, we did not make any changes to parts concerning dispatch after the ready queue.

# References


[1] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, p. 272–284, 2015.

[2] R. Kumar, M. Alipour, and D. Black-Schaffer, "Freeway: Maximizing mlp for slice-out-of-order execution," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 558–569, 2019.

[3] R. K. Choudhary, N. Singh, H. Nair, R. Rawat, and V. Singh, "Freeflow core: Enhancing performance of in-order cores with energy efficiency," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pp. 702–705, 2019.

[4] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, p. 112–119, 1982.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," 2011.