UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

DESIGN AND IMPLEMENTATION OF A RISC-V SUPERSCALAR OUT-OF-ORDER SMT CORE FRONT-END ENGINE USING THE HLIB FRAMEWORK

JAVIER SALAMERO SANZ

Thesis supervisor
MIQUEL MORETÓ PLANAS (Department of Computer Architecture)

Thesis co-supervisor
JONNATAN MENDOZA ESCOBAR

Degree
Master's Degree in Innovation and Research in Informatics (High Performance Computing)

Master's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

01/07/2024

Acknowledgements

I would like to express my gratitude to my Master's thesis advisors, Miquel Moretó and Jonnatan Mendoza, for the opportunity to work on this project at the Barcelona Supercomputing Center (BSC) within the CORE group of the Computer Sciences Department. Their guidance and expertise have been instrumental in achieving the goals of this work and in shaping my development as a computer architect.

I am also thankful to the BSC for its financial support during my Master's studies and to my colleagues for their human quality and for fostering an enriching work environment that has greatly enhanced my knowledge, skills, and motivation. I look forward to many more exciting projects in the future.

A heartfelt thank you to my family and friends for their unwavering support and encouragement; your presence plays a pivotal role in my life. To my girlfriend, Pilar, your understanding and love have been a constant source of strength and comfort.

Abstract

Modern multithreaded processors employing aggressive superscalar Out-of-Order techniques dominate both personal computer and server environments. However, the development and evolution of these advanced technologies are controlled by a few major companies, shaping the market's trajectory. In recent years, there has been a significant push from academia towards the liberalization and openness of hardware technologies, mirroring the shift that occurred in the software industry with the advent of Linux.

In this context, RISC-V has emerged as a revolutionary open and free instruction set architecture (ISA), designed to learn from its predecessors and transform the hardware industry. This master's thesis aims to contribute to this innovative landscape by designing, implementing, and verifying a front-end engine micro-architecture for a simultaneous multithreading superscalar Out-of-Order RISC-V core.

Additionally, this project leverages a wide array of open-source tools for hardware development, emphasizing the use of HLib—a sustainable hardware library framework. This project not only utilizes HLib but also contributes to its growth and development.

Keywords: Micro-architecture, Out-of-Order execution, Superscalar processors, Multithreading, RISC-V, Frontend, Hardware design, Verification, SystemVerilog, HLib, CoCoTB.

Contents

Chapter 1

Introduction

The relentless pursuit of performance in computer architecture has been carried by advances in integrated circuit technology. Logic has progressively become faster while occupying less space, resulting in significant performance improvements solely through technological progress [1–3]. This constant improvement, famously known as Moore's Law, describes the observation of transistors of an integrated circuit doubling every two years. However, this exponential growth has slowed in recent decades due to underlying physical limitations in materials [2].

Despite this slowdown, the evolution of computer architecture has continued to be remarkable, especially in microprocessor design. Since the advent of mass production in the 1970s, which led to the rise of the personal computer (PC) platform, key innovations such as instruction sets, pipelining techniques, and fast cache memories became standard by the 1980s. The 1990s saw the emergence of powerful superscalar out-of-order processors with deep pipelines and aggressive branch predictors, further augmenting performance. However, instruction-level parallelism (ILP) started displaying its inherent limitations, contributing to the popularization of another form of parallelism, thread-level parallelism (TLP), in form of multi-core and multithreaded architectures [3, 4].

The TLP paradigm aims to enhance resource utilization and throughput by adopting multi-threading techniques [3, 5]. Multithreaded processors address the performance loss caused by pipeline stalls due to memory hierarchy misses, I/O events, and functional unit latencies, ultimately surpassing the performance ceiling achievable with instruction-level parallelism. In this work, we review traditional multithreading techniques described in the literature, including chip multiprocessors, coarse-grain multithreading, fine-grain multithreading, and simultaneous multithreading. Computer architectures that exploit ILP and TLP have become the cornerstones of modern desktop and server computing environments, where throughput, resource utilization, and energy efficiency metrics dictate the global market.

Furthermore, in the context of the European Union, many initiatives have been promoted in recent years, particularly after the COVID-19 crisis, targeting technological sovereignty in critical fields. One of these key technologies is chip design and manufacturing, which includes access to critical materials and components, fostering a new generation of know-how experts, and developing proprietary and open-source IPs and other hardware stack elements to strengthen the European technological industry and boost economic growth [6, 7].

Under these circumstances, this project aims to design, implement, and verify a complete front-end engine

of a modern superscalar out-of-order (OoO) core with simultaneous multithread (SMT) support, tailored for throughput-centric server workloads. It will contribute to the popularization of the emerging and open RISC-V ISA [8] along with many other tools in the open-source hardware development stack, including the Verilator SystemVerilog simulator [9], and the CoCotb co-simulation Python testbench framework [10], to provide a fully tested implementation following established coding standards. Additionally, this thesis is part of larger projects currently in active development at the Barcelona Supercomputing Center (BSC). It contributes to the modular HLib (Hardware Library) project [11], which provides fully developed and verified general-purpose modules along with a series of methodologies, infrastructure, and guidelines to speed up the hardware development process while promoting maintainability, re-usability, and scalability standards. Moreover, it is part of the latest iteration of the Lagarto series of processors at BSC, Lagarto Ox, a processor tailored for a high-performance 4-width superscalar out-of-order implementation for RISC-V, and to which this frontend will be an essential part. The rest of the processor elements, such as the backend, memory hierarchy, predictors, and many others, are being actively developed in parallel to this work. Both of these projects, to which this thesis adheres, interoperate and complement each other to boost BSC's hardware development productivity. As an additional benefit of being part of these projects at BSC, we will have access to the Genus Synthesis Solution tool from Cadence and the 7nm technology node from TSMC, which we will use to simulate synthesis results that we will analyze at the end of this document.

Finally, this work goes through many phases of the hardware design engineering cycle for the frontend design, starting with the initial exploration of current RISC-V implementations and computer architecture background, continuing with the design and specification phase, the implementation at RTL (register-transfer level) in SystemVerilog, the verification process at the modular level, and the synthesis analysis in terms of area and frequency. Due to the nature of this work being a section of a processor, we won't provide a performance analysis, as it is dependent on the complete processor design, beyond the scope of this master thesis. Nevertheless, we hope this work contributes to the competitive high-performance core that Lagarto Ox aims to be.

1.1 Motivation

RISC-V is one of the newest open-source ISAs, having begun its development in 2010. Since then, it has gained a lot of traction from academia and industry. RISC-V ISA aims to learn from the lessons of its predecessors and pave the way for an open-source hardware community [8, 12].

Despite the exceptional breakthroughs of many open-source implementations, from modest microcontrollers to complex superscalar out-of-order multi-cores, very few projects focus on superscalar processor design that implement in-core multithreading to take advantage of the parallelism of server workloads while maximising the hardware usage. This project aims to fill the gap by taking steps towards a high-performance multithreaded superscalar RISC-V processor open-source implementation.

Nevertheless, designing and implementing a processor with these characteristics is a highly time-consuming task that requires extensive expertise, often spanning multiple years and necessitating collaborative team efforts. To overcome this challenge and ensure the feasibility of completing our project, we made several strategic decisions.

Firstly, we opted to narrow the project scope to focus solely on delivering the frontend of the processors. In this context, we made several assumptions related to how the frontend interfaces with external elements, such as the backend and the memory hierarchy. The frontend elements of this project will span from the fetch stage, where instructions are obtained in a coherent and organized manner, to the instruction renaming and ROB (Reorder Buffer) allocation. This marks the transition between the frontend, where order is enforced, and the backend, where out-of-order techniques begin executing instructions as operands and resources become available. The ROB holds a special position as it connects both the frontend and backend, but we chose to include it as an additional component of the frontend, considering it completes this aspect of the work well.

Additionally, we focused on the frontend because most of the structures and redesign efforts required to support simultaneous multithreading for a out-of-order and superscalar microarchitecture reside in the frontend elements of the processor. The backend bases its dynamic scheduling on the operand availability with no regard of the thread origin as long as they remain isolated from each other. In other words, the backend can be mostly agnostic and dynamically shared with the same hardware, as instructions are executed independently of the thread they belong to, requiring minor changes to maintain thread isolation during execution [5].

Secondly, we chose to leverage the resources provided by the HLib hardware library [11]. By adhering to its established methodology, we anticipate significant reductions in development time, enabling us to achieve our objectives more efficiently.

Finally, a primary motivation for this thesis is to deepen my knowledge in computer architecture by achieving a better understanding of superscalar, out-of-order, and multithread techniques, along with their trade-offs, challenges, complexities, and background.

1.2 Objectives

The main goal of the project is to design, implement, test and evaluate a RISC-V multithreaded superscalar out-of-order front-end engine. Additionally, secondary objectives focus on scalability, maintainability, and re-usability of the implementation. The project objectives are outlined as follows:

- Research and study of multithreading techniques, processors, and current open-source implementations in RISC-V.
- Architectural definition of a parametric and modular design of a multithreaded out-of-order front-end engine, including:
  - Fetch Stage.
  - Decode Stage.
  - Rename Stage.
  - Reorder Buffer.
  - Inter-Stage Connection.
- Implement the design using System Verilog RTL language.
- Integrate the design into the HLib parametric library framework.
- Develop a testing environment in CoCoTB to verify the behavioral correctness of the derived front-end modules.
- Ensure clean and well-documented implementation adhering to HLib standards.
- Evaluate the implementation's frequency, area, and code quality using industry-standard tools.
- Discuss the trade-offs inherent in the implementation and assess the flexibility of the design.
- Analyze challenges in other processor elements to support complete multithread execution as future work.

1.3 Thesis organization

This document organizes the work conducted for this thesis in several chapters, structured as follows:

- Chapter 1 introduces this work, its motivations and contributions.
- Chapter 2 gives the terminology, context, and theoretical principles of out-of-order superscalar and multithreaded techniques.
- Chapter 3 explores current open RISC-V core implementations that inspired and motivated this work.
- Chapter 4 describes the the front-end engine design in a comprehensible and structured manner.
- Chapter 5 highlights the configurable characteristics of the front-end's implementation.
- Chapter 6 discloses the verification methodology, as well as some the software models developed.
- Chapter 7 provides synthesis results of critical elements in the micro-architecture and its commentary.
- Chapter 8 collects the final conclusions and the future ramifications and goals of this work.

Chapter 2

Background

In this chapter we present the theoretical background that supports this thesis from the architectural perspective, as well as other projects that synergizes with this work. We overview the limits of instruction level parallelism, the superscalar and out-of-order techniques, and the purpose of multithreading schemes. Additionally, we justify the use of RISC-V open ISA and the HLib hardware library, and the benefits they provide to this work.

2.1 Instruction Level Parallelism

Instruction-Level Parallelism (ILP) in a program measures the number of independent instruction flows that can be executed simultaneously or in parallel while yielding the same results as a sequential execution. More precisely, ILP refers to the average number of instructions executed per step in a parallel manner [4]. It is constrained by dependencies between instructions and is inherent to the program flow, agnostic to the the micro-architecture. There are three classifications of dependencies:

- **Data dependencies:** Also known as true dependencies, these occur naturally when there is a direct data flow between instructions. Two instructions are data-dependent if one instruction produces a value used by the other (a producer-consumer relationship) or if there is a chain of such dependencies connecting them.
- **Name dependencies:** These arise when two instructions share the same register or memory location during execution but do not have a data dependency, they are also known as false dependencies.
- **Control dependencies:** These occur when the correct program order and whether an instruction executes or not depends on the outcome of a conditional branch instruction.

Data and name dependencies can lead to data hazards, which can materialize when the execution order of instructions with such dependencies is not respected, and these instructions are close enough so their overlapping during execution can lead to incorrect program outcomes [4]. Data hazards are classified as follows:

- **Read-After-Write (RAW):** Materializes when a dependent instruction tries to read data before the producer writes it.
- **Write-After-Read (WAR):** Happens when an instruction writes to a register before an older instruction reads it, resulting in an incorrect read.
- **Write-After-Write (WAW):** Occurs when a younger instruction writes to a register before an older one, causing the older, incorrect value to be written last.

To ensure correct program execution, the processor must respect program order or maintain the correct final outcome by avoiding these hazards [4,13]. This requirement increases processor design complexity, necessitating structures and techniques to exploit ILP and avoid hazards, such as parallel execution pipelines, register renaming, branch speculation, instruction wake-up, reorder buffers, and speculation recovery mechanisms.

2.2 Superscalar Processors

In the search for performance improvements, scalar processors, which execute one instruction at a time, started reaching their limitations for several reasons. Bubble-inducing stalls, pipelines designed to accommodate the more complex instructions, and increasing pipeline depth to obtain higher frequencies were no longer cost-effective solutions, partly due to increased branch misprediction penalties. To overcome these limitations, superscalar processors emerged as natural successors [3,4].

The defining feature of superscalar processors is their ability to execute multiple instructions within a single clock cycle, adding a new dimension to the processor pipeline: pipeline width, which determines how many instructions can ideally be executed per cycle [3].

While superscalar capabilities promise higher performance, the intrinsic nature of in-order pipelines quickly limited effective parallelism. Despite wider instruction buses in each stage, program dependencies significantly hindered performance gains from ILP. Consequently, superscalar processors began adopting out-of-order execution techniques, allowing instructions to execute as soon as their dependencies are resolved, rather than strictly in program order, thereby better exploiting ILP [3].

2.3 Out-of-Order Execution

Out-of-Order (OoO) or dynamic scheduling execution technique exploits ILP by executing independent instruction flows in parallel. This technique decouples the execution of an instruction from its mandatory in-order retirement. While instructions can begin execution out of order, they must retire in order to maintain a recoverable and consistent state, allowing ready instructions to overtake others without halting the pipeline. This enables younger instructions to start execution as soon as their data dependencies are resolved, even if older instructions in the program order are still waiting for their source operands. This approach improves single-thread performance by overlapping waits caused by data dependencies with the execution of other instructions, thereby increasing the number of instructions executed per cycle [3,4,13].

The Figure 2.1 illustrates different execution flows of the same program, demonstrating varying degrees of ILP exploitation through out-of-order execution:

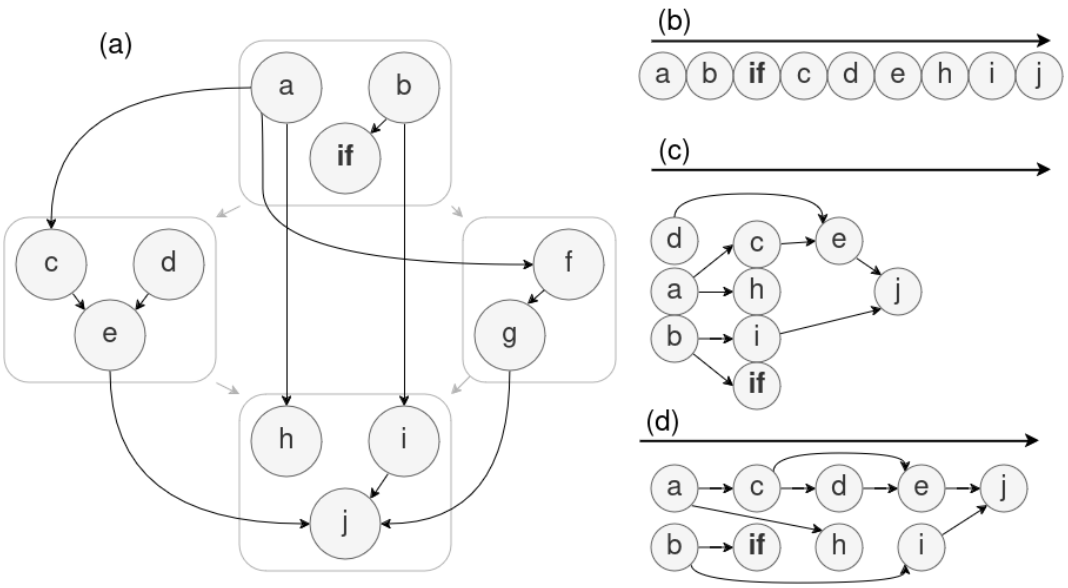


Figure 2.1: Program execution flows [13].

The Figure 2.1a shows the program's dependence graph. Each letter represents an instruction, with arrows indicating data dependencies. Name dependencies are not shown as techniques like instruction renaming, which we will see in future sections, and memory disambiguation, address them. Instructions within squares represent basic blocks, which must be executed in their entirety. Basic blocks are connected by light grey arrows representing control flow. In this example, the first basic block contains a conditional branch instruction (if), which can lead to either the right or left basic blocks, with execution converging afterwards. In this case, we will assume the branch takes the left path and each instruction takes a single cycle to execute.

Figure 2.1b represents instruction execution in a scalar processor, adhering to program order. ILP extraction is minimal as each instruction waits for the previous one to finish, ensuring all dependencies are resolved. Executing one instruction per step results in the program taking nine steps to complete.

If we analyze data dependencies without considering basic block membership, we can identify independent instructions that can execute in parallel, regardless of program order. Out-of-order techniques is based on this premise, allowing OoO processors to execute instructions as soon as their data dependencies are resolved, maximizing ILP and advancing useful work. Control dependencies (like those from the "if" instruction) are managed through branch prediction. We assume ideal branch speculation in this example.

Figure 2.1c illustrates all the ILP that can be ideally exploited each step, with data dependencies as the only limiting factor. By executing many instructions in parallel without affecting the program outcome, the program can ideally be reduced to four steps, corresponding to the longest data dependency flow: a→c→e→j.

Figure 2.1d also employs out-of-order execution, but considers another crucial aspect in dynamic scheduling processors, the issue width, which in this example is 2. The issue width determines the maximum ILP that can be exploited per cycle and significantly impacts the implementation's area, frequency, and complexity. Although ILP is inherent to the program's data dependencies, the issue width limits full exploitation, extending the program execution to five steps.

However, Out-of-Order execution is not exempt of drawbacks. First, OoO processor designs are sensibly

more complex than their in-order counterparts, severely impacting on the development time and hardware costs of their implementations. Secondly, as Amdahl's law stated, given a fixed workload, the speedup achieved with parallelism is limited by the fraction of the workload that can be parallelized [3, 14, 15]. In this context, ILP depends entirely on the program's workload, and designing processors to exploit a fixed amount of ILP, bounded by pipeline bandwidth, can be costly and inefficient in terms of hardware and energy resources [3, 5, 16].

Despite these challenges, since their introduction, superscalar out-of-order processors have become the baseline for high-performing modern computer systems due to their enhancements in single-thread performance, leading to overall system throughput improvements. In the next section, we explore how multithreading aims to address some of the shortcomings of OoO superscalar machines.

2.4 Multithreading

Hardware multithreading enables a microprocessor to execute two or more threads, instruction streams or hardware contexts without software intervention. This technique virtualizes the hardware resources in a processor and presents them to the software as multiple, independent processors, providing a multi-core architecture with a fraction of the area and implementation cost [3–5].

The key idea is to utilize idle processor resources during long-latency stalls in the pipeline to execute instructions of other contexts. These stalls, caused by events such as memory hierarchy misses and I/O accesses, can last several orders of magnitude longer than the processor's cycle time. By executing instructions from other hardware contexts during these idle periods, multithreading increases energy efficiency and total throughput. The unused cycles due to these stalls are referred to as vertical waste [3, 5], illustrated in Figure 2.2.

Executing instructions from multiple independent threads introduces thread-level parallelism (TLP), a form of parallelism completely orthogonal to ILP. By exploiting both ILP and TLP, processors can more efficiently utilize expensive resources like memory caches, issue queues, and execution units, thereby increasing total throughput, particularly beneficial in server environments with high workload demands [5].

As we commented previously in superscalar Out-of-Order execution, some workloads cannot provide sufficient ILP to fully utilize the pipeline width in a single-threaded execution, leading to gaps in the issue bandwidth. Those gaps are named horizontal waste, which are another source of inefficiency that some multithreading techniques aim to reduce [3, 5], also represented in Figure 2.2.

In following sections, we will overview the classical multithreading techniques described in literature which target vertical waste such as Chip Multiprocessors (CMPs), Coarse-Grain Multithreading (CGMT) and Fine-Grain Multithreading (FGMT), and Simultaneous Multithreading (SMT), which also targets horizontal waste.

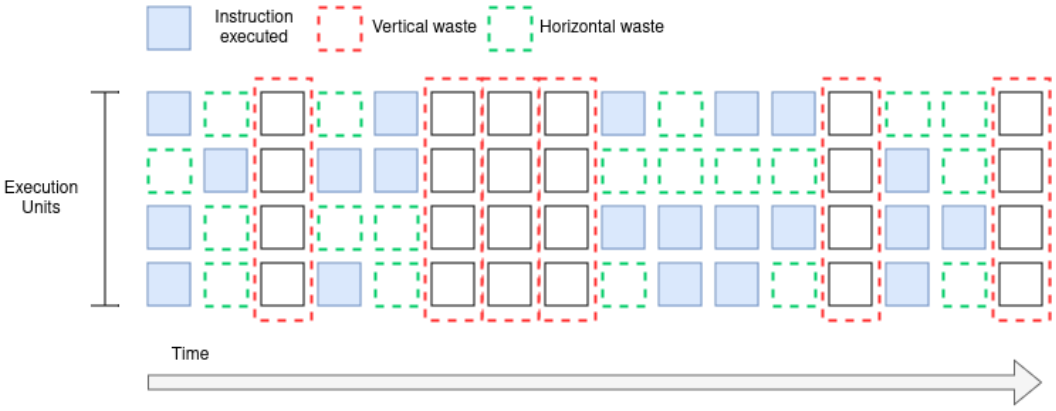


Figure 2.2: Vertical and horizontal waste in a OoO superscalar processor execution.

2.4.1 Multithreading Models

Multithreading can be approached in various ways, with differing strategies for sharing hardware resources among virtual processors. These range from CMPs, which focus on the memory communication across multiple single-threaded processors implementing coherent cache protocols, rather than improve the capacities of the pipeline, to more area-efficient in-core multithreading techniques like CGMT, FGMT, and SMT that implement computer architecture solutions to a apply finer control across the pipeline to allow a concurrent thread execution [3, 5].

Each multithreading model follows a distinct resource partitioning strategies, influencing workload fairness, throughput, and single-thread performance, which can be classified as static partitioning or dynamic partitioning [3]. In static partitioning, resources are allocated independently of workload behavior. In CMPs, resources are spatially distributed, while in FGMT, they are temporally distributed cycle by cycle. Conversely, in dynamic partitioning, resources are allocated based on workload behavior. For example, in CGMT, resources are shared per cycle in event-based decisions, while in SMT, resources are dynamically partitioned per functional unit [3]. Figure 2.3 illustrates the partitioning strategies across the mulithtreading models, while Table 2.1 summarizes which specific resources are typically shared and the context switch mechanisms, in which we will deepen in following sections.

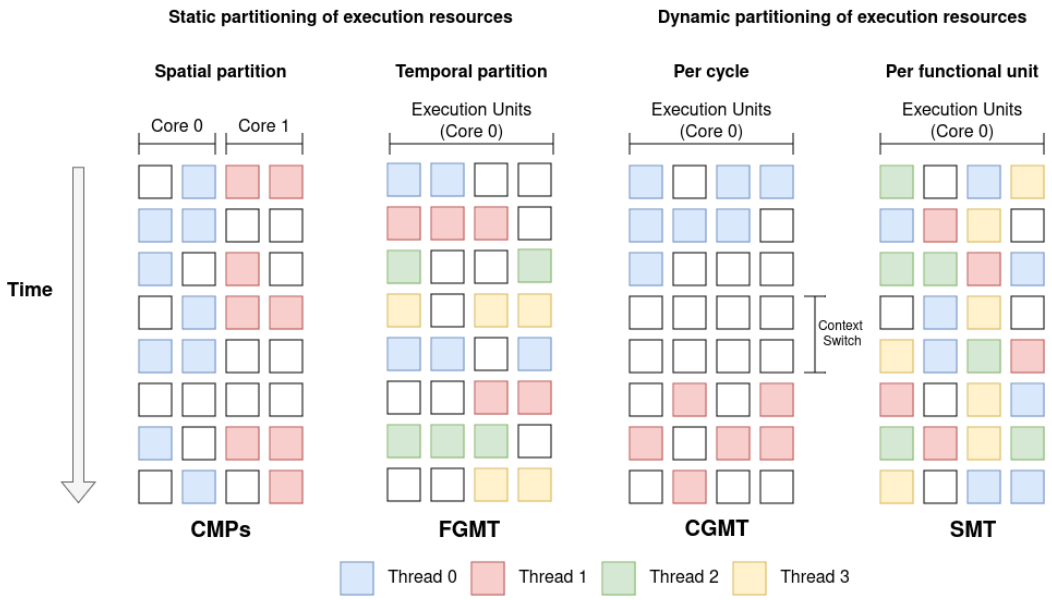


Figure 2.3: Multithreading models execution flow and resource partitioning strategies [3].

MT Approach	Resoruces Shared Between Threads	Context Switch Mechanism
None	Everything	Explicit OS context switch
CMP	Secondary cache, system interconnect	All contexts currently active; no switching
Coarse-grained	Everything but I-fetch buffers, register file and control logic/state	Switch on pipeline stall
Fine-grained	Everything but register file and control logic/state	Switch every cycle
SMT	Everything but I-fetch buffers, return address stack, physical register file, reorder buffer, store queue, control logic/state, etc.	All context concurrently active; no switching

Table 2.1: Multithreading models resource sharing and context switch summary [3].

2.4.2 Chip Multiprocessors (CMPs)

As logic became cheaper and smaller due to technology advancements, it has been possible to increase core complexity as well as the integration of many components into the chip [1–3]. As a result of this, it became feasible integrating multiple cores within the same chip, originating CMPs, which consists in many cores executing different contexts in parallel, thus, exploiting TLP. With this approach, some additional advantages advantages arise: reduction of communication and synchronisation latency between on-chip cores, as well as dynamic sharing of SoC components such as memory caches, coherence controllers, I/O devices, I/O bus interfaces, etc.

Nevertheless, this approach is not exempted of debate and there are some interesting discussion in the matter. For instance, one could argue that, given a fixed area constraint of a die, it is reasonable to think that if instead of a more complex core with larger structures, better predictors, etc., we could fit multiple and more

simple cores exploiting more thread-level parallelism [3, 15]. One the contrary, this could negatively affect workloads were single-thread performance is desired or there is low amount of TLP. Additionally, die area increases are not a trivial solution, given it increases manufacturing complexity [3].

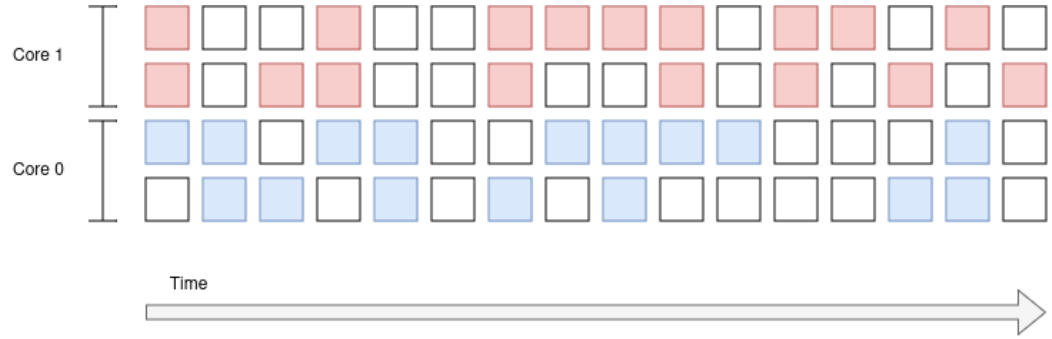


Figure 2.4: Chip multiprocessor execution scheme.

2.4.3 Coarse-grain Multithreading (CGMT)

In CGMT, multiple threads share core processor resources, but only one thread accesses the pipeline at a time. A main aspect in the CGMT models, is the context switch policy, defining how often and which events trigger a context switch, and affects the single-thread performance and thread-execution fairness of the model. For example, a common context switch cause is a long latency event, such as a miss in the cache subsystem. Still, if a thread in execution has few cache misses, it will achieve great single-thread performance while starving other threads for a long time. For this purpose, CGMT architectures apply operative systems schedule policies like time-slice preemption or priority schemes [3, 5].

In CGMT, after a context switch event occurs, the previous hardware context is saved and switched, the pipeline is flushed, and another thread restores its context and resumes its execution. CGTM presents a cost-effective solution by using the long-latency stalls to execute other threads, but requires some cycles for pipeline flushes and context restoration.

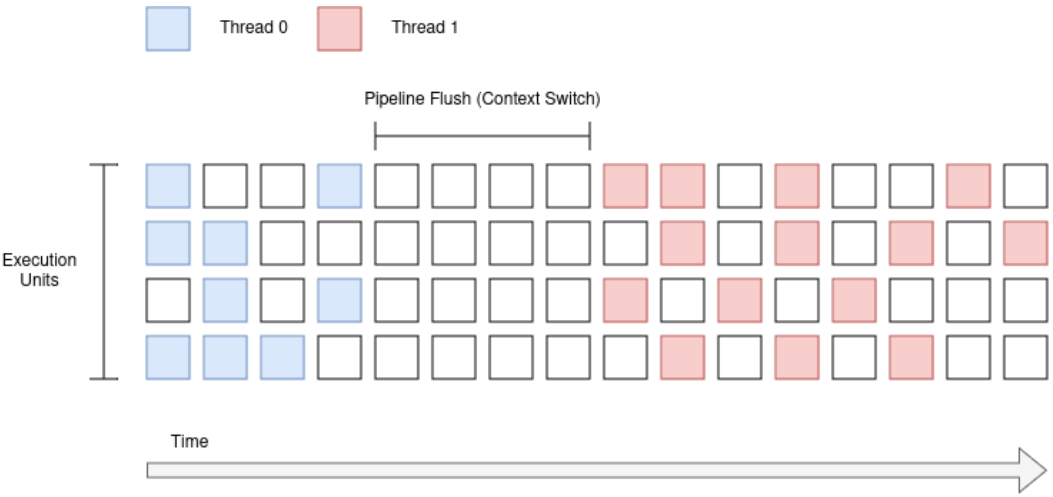


Figure 2.5: Coarse-grain multithreading execution scheme.

2.4.4 Fine-grain Multithreading (FGMT)

FGMT involves constant switching of hardware contexts, typically every cycle, to avoid pipeline stalls. Multiple threads coexist in the pipeline, but only one thread's instructions fill a pipeline stage per cycle. This approach aims to achieve high throughput by eliminating context switch stalls [3, 5].

It is worth noting that, for scalar processors with a high number of threads, FGMT can considerably simplify the pipeline architecture. It is possible to completely hide the latency of instructions and remove the aggressive execution elements such as branch predictors or bypass networks, as when it is the turn of a thread to progress again after the execution of many other treads, enough cycles have passed to solve the data and control dependencies [3].

The main drawback of FGMT is reduction in single-thread performance in favour of general throughput. Since each thread is preempted after every cycle, it takes significantly longer to finish instructions, and the higher the number of threads, the longer it takes to receive access to the pipeline stage to execute the next instruction of the thread [5].

Figure 2.6: Fine-grain multithreading execution scheme.

2.4.5 Simultaneous Multithreading (SMT)

In SMT, no hardware context has exclusive access to any pipeline stage, allowing interleaved instructions from multiple threads to coexist through the pipeline stages simultaneously. Superscalar OoO processors greatly benefit from SMT since they can exploit TLP and target horizontal waste by filling the gaps in those stages where the ILP limits the bandwidth utilization of a stage, optimizing its usage and increasing throughput [3, 5, 16].

In some stages of the core pipeline, it is challenging to implement an SMT approach, and it is generally hybridised with other classical approaches. For instance, in the fetch stage, when a design targets a single instruction cache port, only one thread address request instructions to the memory hierarchy in a given cycle, preventing true SMT. In this case, it is a reasonable workaround to replicate the fetch and its resources for each thread or use another multithreading model for sharing this stage, like FGMT or CGMT [3, 5].

Additionally, SMT needs to consider hardware context management policy to guarantee execution fairness

and avoid thread starvation for the same reasons as CGMT, although with some differences. For example, instead of preempting a thread after a long latency event or after consuming a time slice, it needs to decide which hardware contexts and how many instructions for each of them are allowed to use every pipeline stage at every cycle, compromising TLP and single-thread performance.

Finally, due to the addressing of vertical and horizontal waste inefficiencies, SMT can further improve the throughput and resource utilization of the core compared to other classical models of multithreading [5, 16]. However, it presents the cost overhead of designing the structures that can be shared by multiple threads simultaneously, statically or dynamically, such as physical register files, renaming schemes, bypass networks and wake-up logic, to name few.

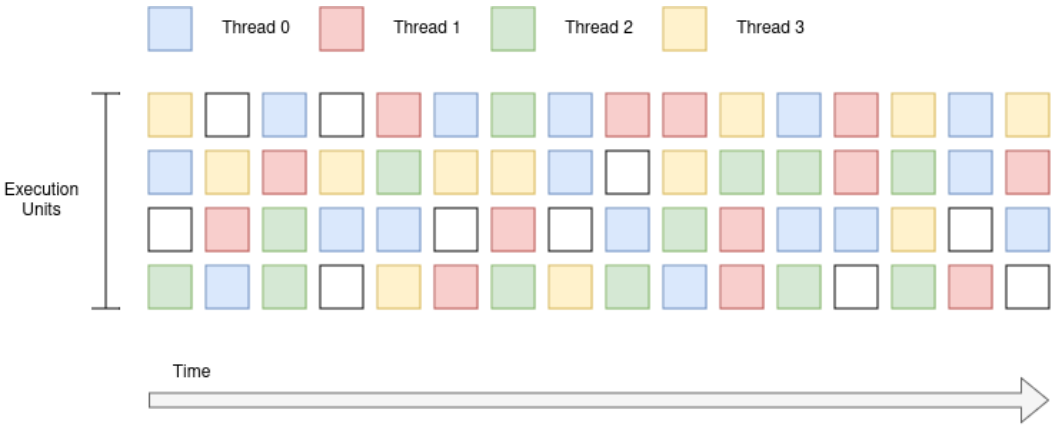


Figure 2.7: Simultaneous multithreading execution scheme.

2.5 RISC-V

In computer architecture, one of the most essential aspects is defining the interaction between hardware and software, which is encapsulated in the Instruction Set Architecture (ISA). An ISA is an abstract computer model that enables users to communicate with hardware by specifying a set of instructions, addressing modes, data types, memory structures, and how the processor operates with them [12].

This separation between hardware implementation and the software stack facilitates and encourages experimentation on one side of the interface while maintaining compatibility with the other, significantly reducing costs and development time. The most popular ISAs for high-performance micro-architecture processors, such as x86 and ARM versions, are proprietary, highly complex, and disallow custom extensions, making them costly options and innovation bottlenecks. Conversely, existing open and free ISAs like SPARC and OpenRISC have architectural issues stemming from their era of definition, such as limited encoding space for compressed instructions, condition codes for branches that complicate instruction renaming, and branch delay slots.

In this context, the increasingly popular RISC-V ISA, developed by Professor Krste Asanović along with graduate students Yunsup Lee and Andrew Waterman at the University of California, Berkeley, began in 2010 [8]. RISC-V is defined as a base integer ISA, similar to early RISC processors but without branch delay slots and with encoding space for optional variable-length instructions, known as compressed instructions.

A base is a minimal set of instructions that provides the software stack (compilers, assemblers, linkers, and operating systems) a reasonable target. This base integer ISA, which must be present in any RISC-V implementation, can be expanded with optional extensions such as atomic, floating-point, compressed, and vector instructions [8, 12].

Its initial simplicity, requiring only an integer base ISA as the minimum requirement, along with its modularity, openness, and the recent surge in RISC-V's popularity, makes it the perfect option for this project, which uses the ratified version of the specifications from December 13, 2019 [8].

2.6 HLib

In hardware development projects and coding projects in general, re-usability, maintainability, and testing of code significantly impact development efforts. In software projects, libraries are a standard tool that programmers use to improve code organization, reliability, maintainability, and re-usability of structures, procedures, and definitions. In contrast, these well-known lessons are not always applied in RTL development. Engineers often unknowingly re-implement standard structures in their designs, further increasing the verification overhead. Sharing these efforts among projects can be essential to increasing productivity among developers.

HLIB is a System Verilog hardware library being developed at the BSC that aims to address these issues by defining a collection of conventions and methodologies. The core objective of the library is to provide a series of parametric, maintainable, synthesizable, and documented general-purpose modules upon which more complex structures can be built in a modular fashion, significantly speeding up development. It offers RTL modules, documentation, code guidelines, project structure, and infrastructure [11].

This library is a collaborative effort from which the project will benefit while contributing to its enrichment. The implementation will use these verified generic modules and methodologies to build the design, and in return, it will contribute a variety of new modules—some simple and others more complex—that adhere to the library's quality standards.

2.6.1 Methodology

One of the critical components that HLib offers is a methodology for module development. All designs in this project will follow these rules to guarantee code quality and robustness. Figure 2.8 illustrates the engineering cycle of a module in Hlib.

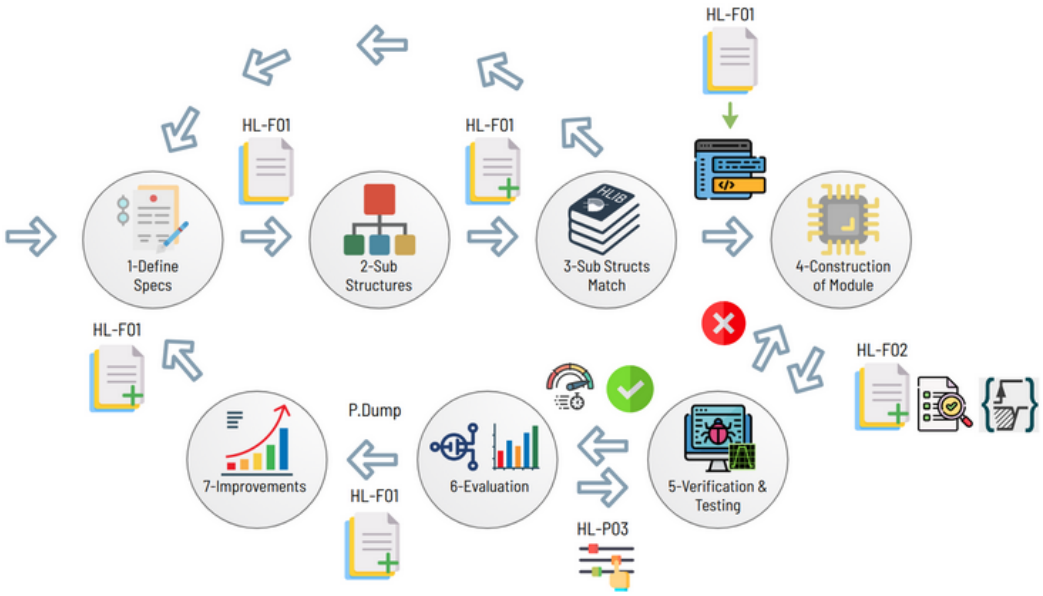


Figure 2.8: HLib engineering cycle.

1. **Definition of the module's specifications:** Define the general goal of the module, its interface with inputs and outputs, and the configuration parameters.
2. **Functionality and structure:** Analyze if some parts of the behavior can be encapsulated in a sub-module and determine if these new sub-modules cover specific functionalities or can be generalized as general-purpose modules. New general-purpose modules will follow its own engineering cycle.
3. **Sub-structures match:** Search the library for any existing general-purpose sub-modules; this ensures re-usability and shortens development time.
4. **Implementation:** Implement the RTL of the module and its new sub-modules with extensive in-code comments explaining their behavior. Any developer should be able to understand the module by its general definition, code, and comments.
5. **Verification:** Verify the module's behavior. The developer should implement a testbench in CoCoTB, a Python framework for RTL verification. The test should be extensive and thorough enough to validate all of the module's features, and corrections may be needed based on test results.
6. **Evaluation:** Ensure all modules in the library are synthesizable and analyze them with synthesis tools and performance modeling.
7. **Improvements:** Iterate the loop and modify the module based on lessons learned in other phases to improve performance, area, timing, or code quality.

Chapter 3

Related Work

In this chapter, we explore some of the current and open-source RISC-V implementations that have similarities to this project. This exploration provides valuable insights into various design methodologies, performance capabilities, and architectural innovations. This context sets a benchmark for the goals and expectations of the project, guiding the development of a RISC-V SMT Superscalar Out-of-Order Front-End Engine.

3.1 BOOM (Berkeley Out-of-Order Machine)

The Berkeley Out-of-Order Machine [17, 18] is an RV64GC configurable and open-source superscalar Out-of-Order core developed by the University of California, Berkeley. It is part of the Berkeley Architecture Research group and written in Chisel, a hardware description language based on Scala. BOOM is parameterizable, synthesizable, and currently represents the state of the art in academia for open-source high-performance RISC-V implementations.

There are three iterations of BOOM: BOOMv1, BOOMv2, and BOOMv3 (SonicBOOM), Figure 3.1 highlights their data-path stages.

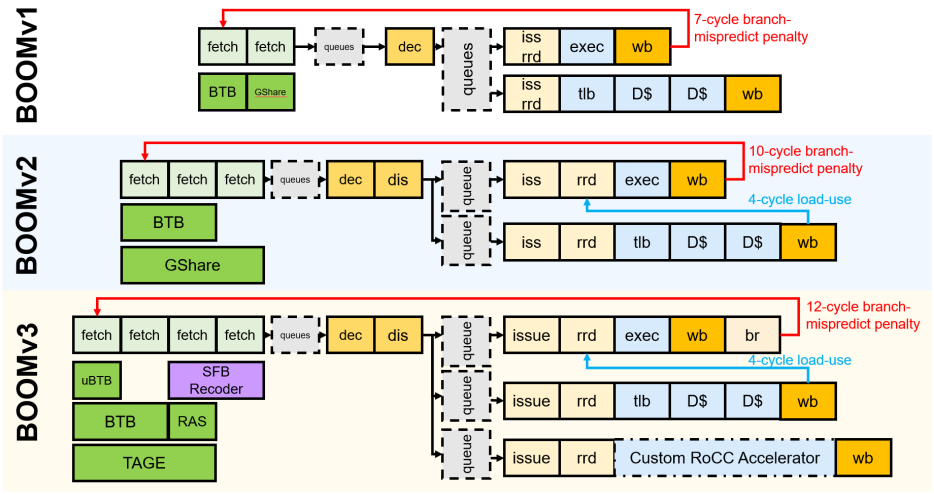


Figure 3.1: BOOM architectural evolution [17].

- **BOOMv1:** Designed as an educational tool resembling the MIPS R10000 processor [19]. It features a unified issue queue, register file, and a short pipeline, but suffers from critical path issues, making it physically infeasible. Other structures like caches, MMU and branch predictors came from another open-source project, the Rocket in-order core [20].

- **BOOMv2:** Focuses on addressing physical implementation issues by increasing pipeline stages in the front end, solving many critical paths, and improving scalability of the backend by splitting the issue queue into integer, memory, and floating-point operations.

- **SonicBOOM (BOOMv3):** Published in 2020, this third and currently most modern iteration is the fastest RISC-V open implementation, achieving performance similar to the AWS Graviton core with a 4-way superscalar configuration [17]. It addresses previous design problems and fully implements modules borrowed from the Rocket project. Key features include a 4-wide superscalar front end, 32KiB 8-way associative L1 instruction cache, and a comprehensive branch predictor system. Figure 3.2 illustrates its micro-architecture.

  In the frontend, the main architectural characteristics are a 4-wide superscalar configuration, the support for compress instructions, the 32KiB 8-way associative L1 instruction cache, and a composed branch predictor system. This composition includes a local history branch predictor BTBs, a higher accuracy and higher latency global predictor TAGE, and a Return Address Stack for *call* and *ret* instructions.

  The backend has an 8-wide issue bandwidth formed by three separate queues connected to the functional units, a 128-entry reorder buffer and a partitioned register file. In the memory execution pipeline stands out the 32KiB 8-way associative L1 data cache that supports two loads or one store per cycle.

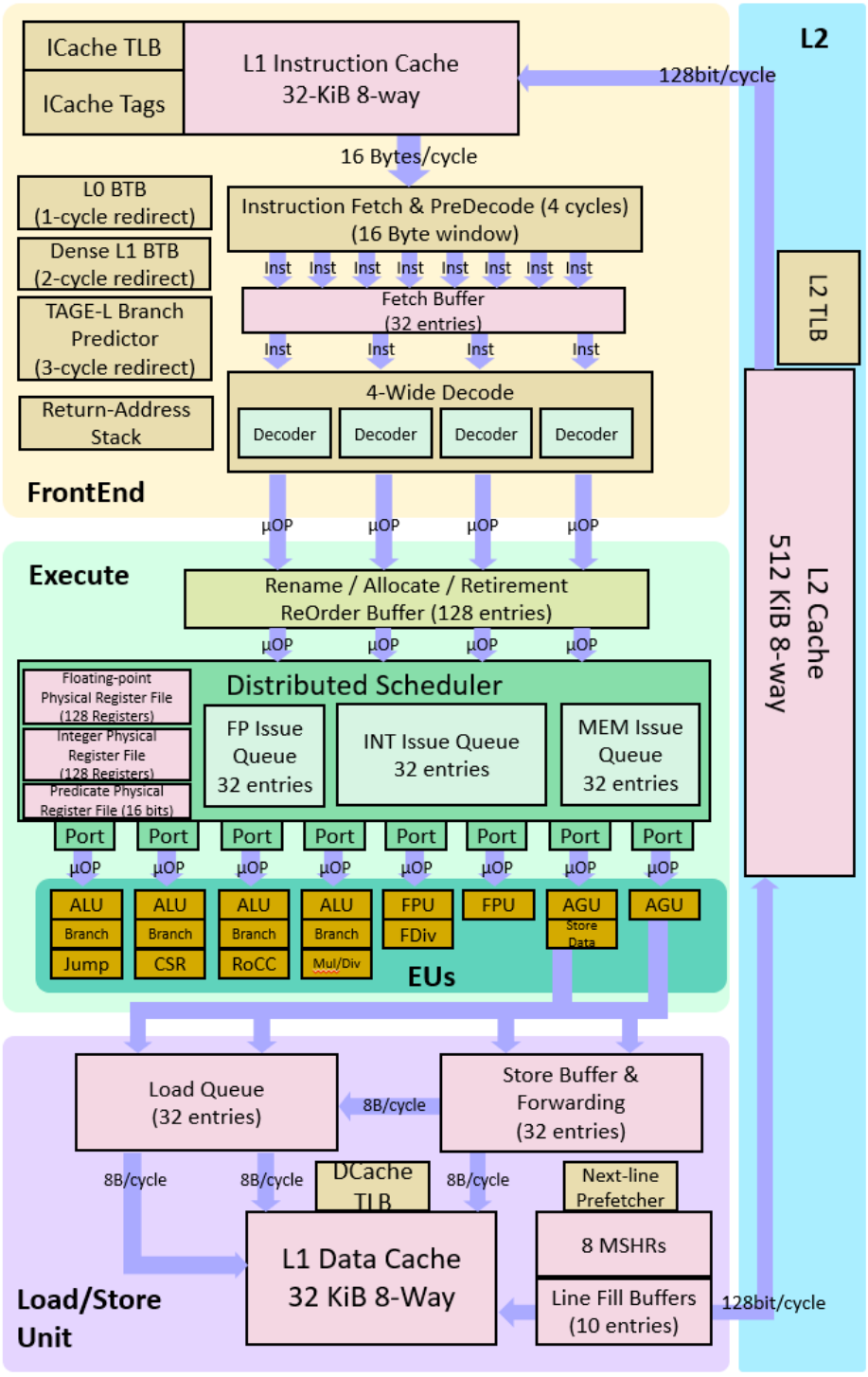


Figure 3.2: SonicBOOM micro-architectural diagram [17].

3.2 RiscyOO

RiscyOO [21,22] is an RV64G Out-of-Order superscalar cache-coherent multiprocessor core developed by the Massachusetts Institute of Technology (MIT) using Bluespec System Verilog HDL. It is capable of booting multicore Linux, reaching up to 40MHz on an FPGA and up to 1.1GHz in ASIC at 32nm technology synthesis.

Its architecture relies on the innovative Composable Modular Design (CMD) framework, characterized by:

- **Module Interface Methods:** These methods provide immediate access and execute atomic updates to internal state elements.
- **Guarded Interface Methods:** Each method is guarded to ensure readiness before application.
- **Interconnection via Atomic Rules:** Modules are interconnected using atomic rules that invoke interface methods across different modules. These rules either successfully update the state of all involved modules or remain unchanged.

RiscyOO offers a performance advantage over BOOMv2, achieving an IPC (Instructions Per Cycle) of 0.48 in the SPEC06 benchmark, compared to BOOMv2's IPC of 0.42 [17], and approaches that of the 3-wide Cortex-A57 ARM core [21]. The architectural structure is depicted in the Figure 3.3.

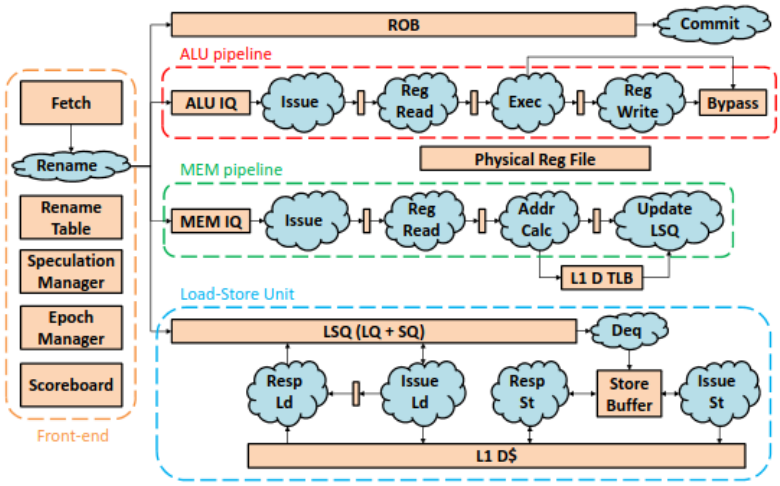


Figure 3.3: RiscyOO micro-architectural diagram [21].

3.3 TOOOBA

TOOOBA [23] is an open-source out-of-order core developed by Bluespec, Inc. It's a slight modification of the RISCYOO core, enhanced with support for compressed instructions and improvements for verification and debugging. This likely gives TOOOBA a light performance boost over RiscyOO, however, we could not find any performance data that confirms it.

Both RISCYOO and TOOOBA utilize a tournament branch predictor [17], which was first introduced in the Alpha 21264 processor [24]. This predictor includes choice predictor that learns whether best to use local or global branch history predictor decision, optimizing for better accuracy across different types of branch behavior.

In contrast, BOOMv2 employs a gshare branch predictor [17], which relies solely on global branch history. The gshare predictor combines global history with the branch program counter to index into a two-bit pattern history table. While the gshare predictor can be effective, it doesn't adapt as flexibly as the tournament predictor, which can choose between local and global history predictors.

3.4 Klessydra

Klessydra [25, 26] is a family of processing cores designed at the Digital System Laboratory, University of Sapienza, Rome, focused on IoT computing. It includes various designs aimed at fault tolerance, SIMD execution, systolic arrays, and more. Notable designs include:

- **Klessydra S0:** A single-cycle, in-order core with minimal gate count.
- **Klessydra S1:** An in-order pipelined core.
- **Klessydra OoO:** An Out-of-Order processor (beta version).
- **Hydra:** A dual-core lock-step processor.
- **Klessydra T0:** A configurable interleaved multithreading (IMT) processor.
- **Klessydra Morph:** A morphic processor that can switch from IMT to an in-order single thread based on workload.

The configurable Klessydra T0 features a 3-stage in-order pipeline implementing the 32-bit RV32IME instruction set in machine (M) privilege level. It uses the interleaved multithreading (IMT) technique, another name for FGMT, that avoids instruction dependency stalls by switching thread context every cycle, which increases throughput but requires a minimum number of active threads. To prevent bubbles from context switches, the design replicates the program counter, register files, and control status registers (CSRs) for each thread. The micro-architecture is illustrated in the Figure 3.4.

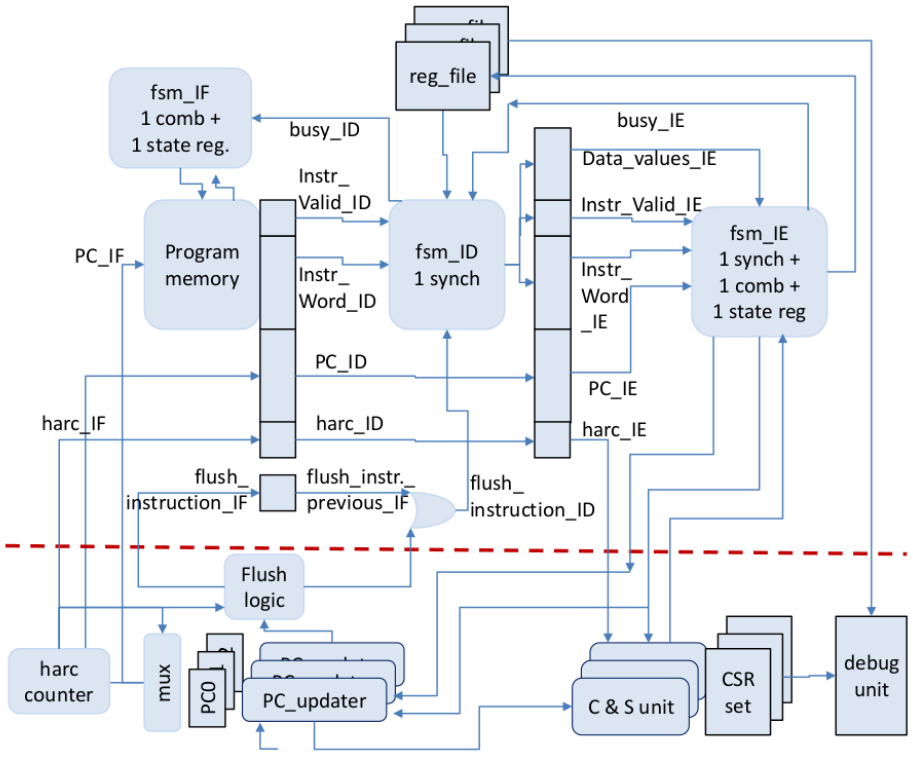


Figure 3.4: Klessydra micro-architectural diagram (Klessydra-T023) [25].

Configuration options for Klessydra-T0BS include modifiable parameters:

- **B (Thread Pool Baseline):** The minimum number of active threads required to execute stall-free workloads.
- **S (Thread Pool Size)**: Maximum number of active threads.

This configuration flexibility has spawned several publicly accessible implementations, including T022, T023, T024, T033, and T034. Internal designs like T012, T013, and T014 are reserved for evaluation.

The performance of these configurations is summarized in a table, showing the average throughput in MIPS (millions of instructions per second) and cycle time in nanoseconds for FPGA implementations on Xilinx Series 7 devices.

Core module	Pipeline stages	Cycle time [ns]	Average throughput (MIPS)			
			1 thread	2 threads	3 threads	4 threads
S0	2	12.0	71.84	-	-	-
T012	2	12.7	67.88	78.74	-	-
T022	3	8.9	48.43	96.86	-	-
T013	2	13.9	62.02	71.94	71.94	-
T023	3	9.7	44.44	88.87	103.09	-
T033	4	7.3	30.58	59.05	118.09	-
T014	2	15.9	54.22	62.89	62.89	62.89
T024	3	9.4	45.85	91.71	106.38	106.38
T034	4	7.4	30.16	58.25	116.50	135.14

Table 3.1: Klessydra performance results ("-" = not applicable) [25].

Chapter 4

Design

This chapter presents the design of a front-end engine micro-architecture for an Out-of-Order superscalar RISC-V processor, which will be expanded to support multithreading in subsequent sections. The front-end is tailored to utilize the RV64GC extensions of the ISA, where "G" represents the IMAFDZicsr_Zifencei combination of instruction-set extensions. The architecture is flexible and can be easily expanded to support additional instruction set extensions with minimal design changes. The standard RISC-V instructions use a 32-bit encoding, with the compressed (C) extension introducing 16-bit instructions [8]. The specific extensions used are detailed in the table 4.1.

Extension	**Description**
RV64I	Base Integer Instruction Set for a 64-bit architecture
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point
Zicsr	Control and Status Register (CSR) Instructions
Zifencei	Instruction-Fetch Fence
C	Standard Extension for Compressed Instructions

Table 4.1: Supported ISA by the design.

The Front-End engine is divided into three stages, resulting in a total pipeline depth of five cycles:

- **Fetch Stage:** The fetch stage spans three cycles and is responsible for requesting instructions from the memory hierarchy, delivering them to subsequent stages, and calculating the following address. It supports variable-width instructions: 32-bit for regular instructions and 16-bit for compressed instructions, which increases control complexity. The fetch stage delivers blocks of 16-bit instructions to the decode stage. In other words, a single block for a compress instruction or two blocks for a regular one.
- **Decode Stage:** The decode stage interprets these 16-bit blocks to form complete instructions, translating them into internal representations for further processing. This stage ensures that the instructions are properly aligned and ready for execution in the next stages.

- **Rename Stage:** The rename stage resolves dependencies and renames registers to avoid data hazards and allow Out-of-order execution. It also maps instructions to separate out-of-order issue queues in the backend engine, ensuring efficient and correct scheduling.

Additionally, these stages are interconnected through queues with a configurable number of entries. If the number of entries matches de bandwidth of the pipeline, these queues act as regular inter-stage registers. However, if the number of entries is higher, this queues allow decoupling stages from each other and continue processing instructions regardless of stalls in other stages of the pipeline.

The design is highly flexible and configurable, allowing adjustments to superscalar bandwidth, the size of structures such as queues, physical register file, and the ROB, supported extensions, and instruction mapping to different queues post-renaming. Tough, the explanation and diagrams are based on a 4-width superscalar configuration for simplicity, with fetch bandwidth measured in 16-bit blocks. For a 4-width superscalar, the fetch bandwidth is eight 16-bit blocks, which means it is possible to fetch up to 8 compressed instructions ideally. Many sub-modules used in the design are already present in the HLib library, including FIFO queues, selectors, priority encoders, and the ROB. For these, a high-level functional description will be provided.

4.1 Project Scope and Assumptions

This work forms part of a larger project at the BSC, Lagarto Ox, and in this thesis we focus exclusively on the frontend of the core. Given the complexity and number of elements in a processor, we make several assumptions based on state of these other elements outside the scope available to us. These assumptions correspond to elements that interface with the Front-End Engine:

- **Memory Hierarchy:** We will utilize a single-port, blocking instruction cache with a 1-cycle latency on a hit. This instruction cache receives virtual addresses and handles their translation transparently to the frontend.
- **Branch Prediction:** This speculative technique is excluded from the project initial design due to its complexity. Branch prediction will be considered for future exploration after this thesis.
- **Back-End Engine:** The frontend will interface with the backend through a composition of dynamic scheduling issue queues, each corresponding to a different execution path, improving scalability. The four execution paths will be:
  - Integer.
  - Floating-Point.
  - Memory.
  - CSRs.

Figure 4.1 illustrates the complete Front-End Engine diagram, highlighting in lilac the elements outside the scope of this thesis:

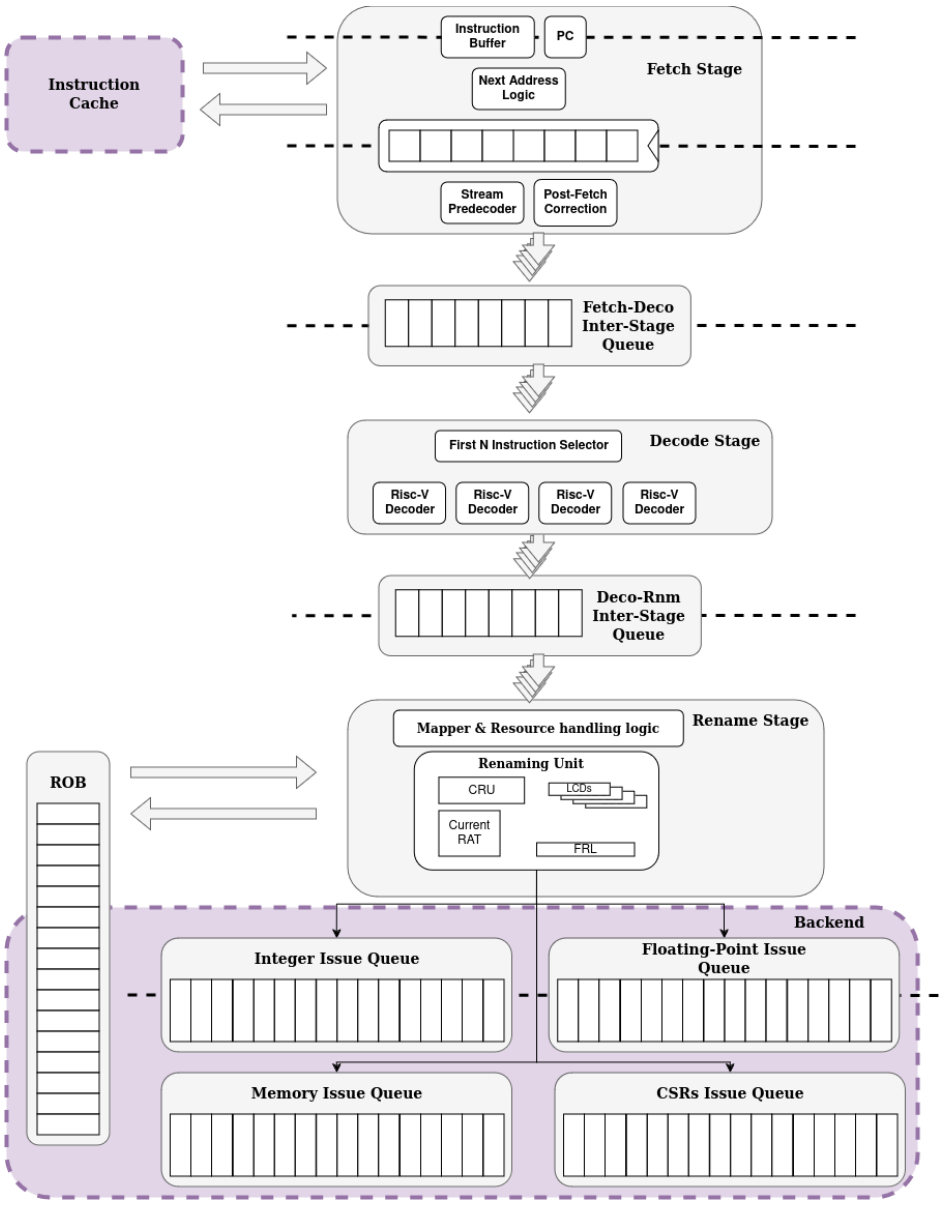


Figure 4.1: Fronted diagram.

4.2 Fetch Stage

The first action of any processor is to fetch the program instructions. This stage requests these instructions to the memory hierarchy, handles changes in the instruction flow, and predicts conditional branches. Our design is tailored for an RV64GC architecture, where support for compressed instructions significantly alters the design due to the their variable length, either 16-bit (compressed) or 32-bit (regular) long, and potential misalignment in memory. The fetch stage is responsible for obtaining 16-bit blocks that correspond to either an entire instruction or half of a regular one, and then allocating them in the fetch-decode inter-stage queue. It is acceptable to fetch only half of an instruction if, for instance, the remaining half is located in a different

cache line. However, decode stage will only progress with complete instructions.

As a reminder, we are considering a 1-cycle delay for an instruction cache hit. Branch predictors are beyond the scope of this project but can be easily integrated into this design in the future. Additionally, we decided to divide the fetch stage into three cycles due to timing constraints caused by the sequential nature of many elements, which we will see in a moment. Figure 4.2 provides a high-level diagram of the fetch stage.

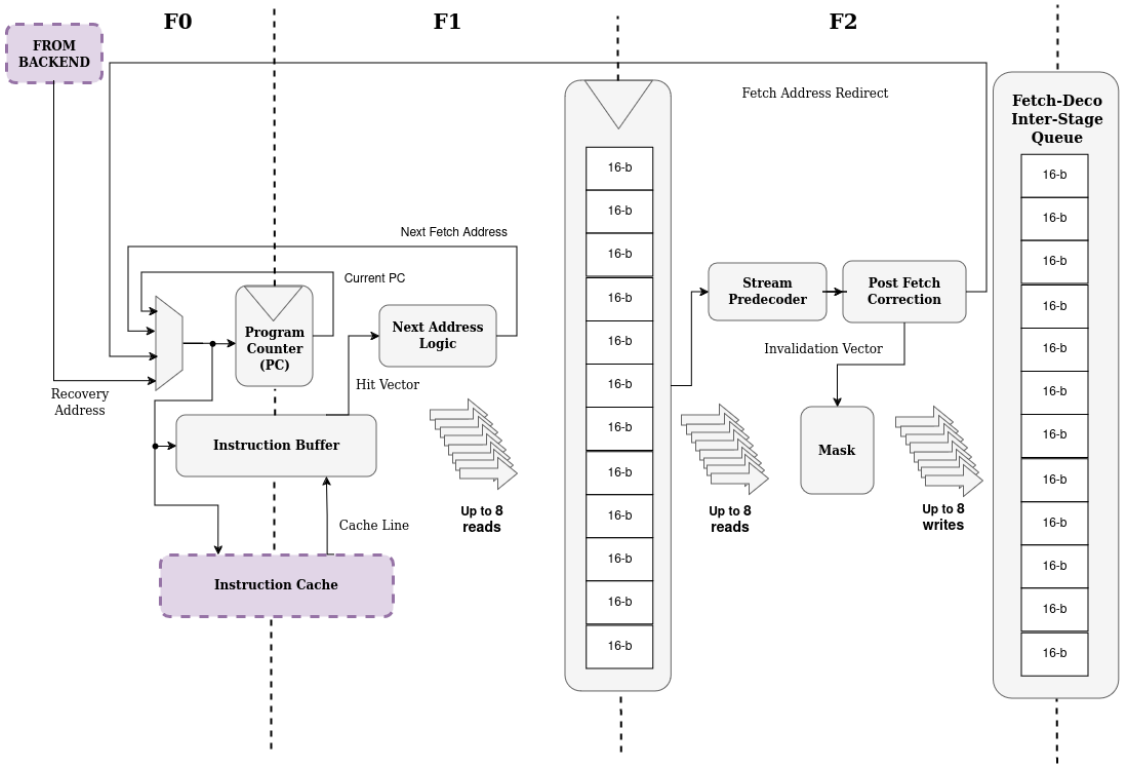


Figure 4.2: Fetch Stage diagram.

The first cycle of fetch is spent selecting the fetch address from different sources and priorities and storing it in the program counter (PC) register, then requesting the address to the memory hierarchy (Instruction Cache) and the Instruction Buffer (a small buffer storing a couple of cache lines; it will be explained in more detail in Section 4.2.1) simultaneously. The address source with highest priority comes from recovery through the speculative execution of instructions (outside the frontend), meaning there was an exception, an interruption, a branch misprediction or any other source of recovery that need to restart the execution. If there is no recovery, the next event with the highest priority is the Fetch Address Redirect, meaning an unconditional jump has been identified (in the third cycle) in the Stream Predecoder, and it is executed immediately. Lastly, if there is not a fetch stall event, in which the PC keeps its value, the address calculated in Next Fetch Address module is selected. This module provides the next address needed after the previous cycle, and it updates its value to the PC register.

In the second cycle of the fetch stage, requests to the Instruction Cache that result in a cache hit are automatically written into the Instruction Buffer in a round-robin fashion. Sequentially, the Next Address Logic module calculates the subsequent fetch addresses. Since we are fetching 16-bit blocks, the PC register is incremented by two bytes for each obtained block.

The third cycle involves the Stream Predecoder and Post Fetch Correction modules. These components identify compress instructions or execute unconditional jump instructions by adding their immediate value to the PC. In RISC-V, these instructions are known as Jump and Link (JAL). Early identification of JAL instructions shortens pipeline bubbles, as it allows control flow changes without waiting for JAL execution. After a mask invalidation, the remaining valid blocks are allocated in the fetch inter-stage queue for the decode stage to consume them.

4.2.1 Instruction Buffer

The Instruction Buffer is designed to deliver instructions from different cache lines, facilitating fetch bandwidth when requested blocks span two cache lines—something a single-port instruction cache cannot achieve. The buffer can hold at least two cache lines, with addressing at a half-word (16-bit) granularity. It features fully associative internal storage, can be read asynchronously, and supports read bypass.

The Instruction Buffer's write port, which follows a round-robin write policy, is used by the Instruction Cache responses. Each time the instruction cache responds, the cache line is added to the buffer if it is not already present. The read bypass mechanism is particularly useful when none of the blocks are in the buffer and must be read from the cache response. This allows every cache response to bypass the Instruction Buffer, thus avoiding a extra 1-cycle bubble.

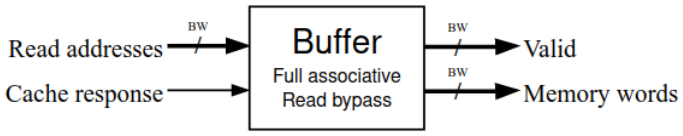


Figure 4.3: Instruction Buffer diagram.

4.2.2 Next Address Logic

The Next Address Logic is a very simple module that counts the number of 16-bit blocks obtained through the fetch request via the instruction cache or instruction buffer and calculates the next program counter (PC) based on this count. For example, if the fetch PC is 0x24 and it retrieves four 16-bit instruction blocks, the next PC will be 0x2C since each blocks is two-byte long.

4.2.3 Stream Predecoder

The Stream Predecoder fulfills two roles: it identifies which blocks correspond to compress instructions and which are regular half instruction blocks. In RISC-V, compress instructions are encoded with the first two bits being 00, 01, or 10, while regular instructions always start with 11. Additionally, it identifies if the fetched blocks contain *JAL* instructions, including their compressed versions. A regular *JAL* can be identified with the first instruction block (the first 16 bits), but, since it requires the second half to compute the target address, it will not be resolved in this third cycle of fetch and will be executed later in the pipeline.

This delay causes a bubble when the *JAL* is eventually executed in the backend. The predecoder earns its name by identifying certain instructions before they reach the decode stage.

4.2.4 Post Fetch Correction

The Post Fetch Correction computes all target addresses for complete JALs within the fetched blocks. Simultaneously, it selects the first JAL and invalidates all subsequent blocks following the jump. Finally, it sends the target address of the first JAL to the fetch address selection in the first cycle.

4.3 Decode Stage

The decode stage interprets 16-bit blocks allocated by the fetch stage to form complete instructions. Encoded instructions are translated into internal representations in the micro-architecture for further processing.

The decode stage, shown in Figure 4.4, is responsible for two main tasks:

- **Instruction Extraction:** After receiving 16-bit blocks of instructions from the fetch stage, the decode stage extracts as many instructions as possible to fill the decode bandwidth. Compressed instructions are encoded in one block only, while regular instructions require combining two blocks.
- **Instruction decodification:** The decode stage identifies the instructions, generates the necessary control signals, and reads the encoded operands and immediates. This process prepares the instructions information for execution in the next stages of the pipeline.

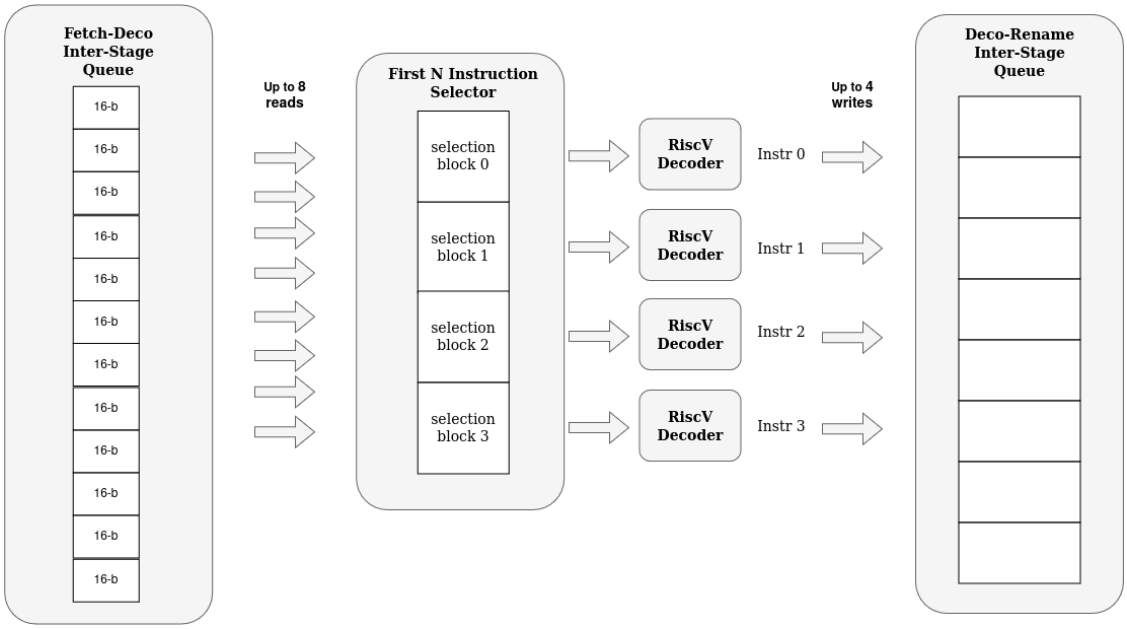


Figure 4.4: Decode Stage diagram.

4.3.1 First N Instruction Selector

This module is responsible for reading the first N instructions available in the Fetch-Decode Inter-Stage Queue, where N is the minimum number between the stage's maximum bandwidth and the number of free slots in the next inter-stage queue. This module sequentially reads 16-bit blocks, determining where each instruction starts and ends. Finally, the module delivers the instructions and the read vector to the Fetch-Decode queue.

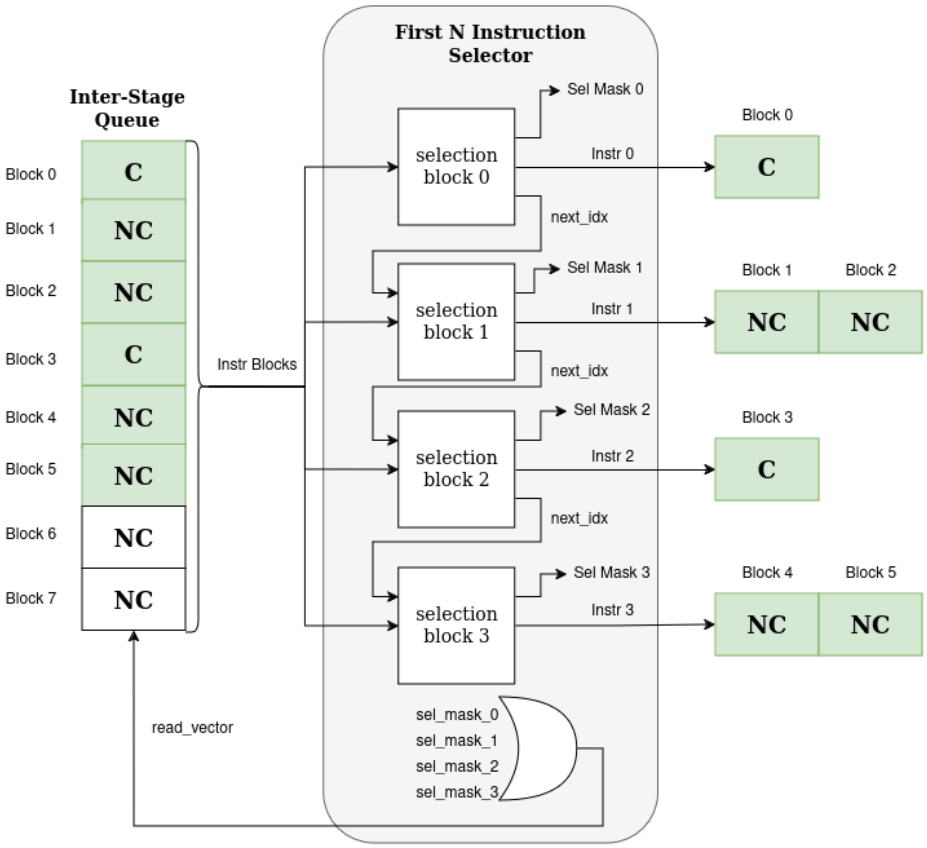


Figure 4.5: First N Instruction Selector behavior example.

The Figure 4.5 exemplifies the behaviour of the instruction selector. In this example, there are five instructions available, but only can advance to the next stage due to bandwidth limiation.

1. The first instruction is compress (block 0).
2. The second instruction is non-compress (block 1, block 2).
3. The third instruction is compress (block 3).
4. The fourth instruction is non-compress (block 4, block 5).
5. The fifth instruction is non-compress (block 6, block 7).

The first selection block reads the first instruction block and calculates and propagates the next index signal, where the following selection block should start searching for the next instruction. The following selection block repeats this process until all the bandwidth is filled, if possible. Finally, a read vector is sent to the inter-stage queue to clear the read blocks. In this example, blocks from 0 to 5 will be read as they correspond to the first four instructions.

4.3.2 Decoder

The decoder's critical tasks are instruction identification, generating the control signals used throughout the pipeline, and extracting the register and immediate operands encoded in the instruction. Since RISC-V is a modular ISA, parameterizing which extensions the decoder supports will be a central aspect of the decoder design. This module can support any valid subset of the ISA formed from the supported extensions, currently being RV64IMAFDQCZicsr_Zifencei. Additionally, this decoder provides a human-readable and flexible control table for the instruction signals, which is essential for its long-term maintainability. However since this characteristic is outside the design perspective, its explained in the implementation details shown in the Annex A.

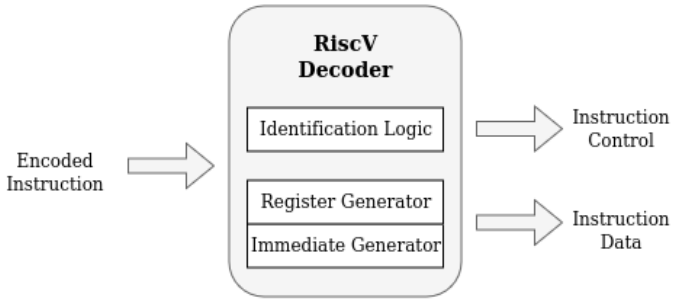


Figure 4.6: RISC-V Decoder high-level diagram.

From an architectural perspective, the decoder design produces two key outputs, as reflected in Figure 4.6:

- **Instruction Identification and Control Signal Generation:** The identification logic accurately detects which instruction is being decoded and generates the corresponding control signals.
- **Instruction Data Extraction:** This process refers to the extraction of encoded registers and immediate values. It requires minimal identification logic (primarily for instruction formats and compressed instructions) since these elements are typically encoded in fixed positions. Therefore, data extraction can be done in parallel with instruction identification to avoid sequentializing the logic.

4.4 Rename Stage

Out-of-Order execution technique needs to produce the same program outcome as it executed the instructions following program order and avoid data hazards. However, this technique introduces new data hazards, specifically Write-After-Read (WAR) and Write-After-Write (WAW), which are produced by name dependen-

cies and can corrupt the program's logical consistency. These hazards are caused by the reuse of architectural registers or memory locations during program execution and are known as false dependencies [4].

The Register Renaming technique eliminates register false dependencies (memory name dependencies are handled in the backend with memory disambiguation strategies, and are out of the scope of this thesis) by mapping or translating instruction registers into containers transparent to the ISA [3, 4, 27]. This allows multiple in-flight values per register, preventing value overwrites. The precursor to this technique, Tomasulo's algorithm, introduced reservation stations in the IBM 360/91 processor's floating-point execution path in 1967, setting the standard for future renaming schemes [4, 28].

This design uses explicit renaming, a technique derived from Tomasulo's algorithm and employed in processors like the MIPS R10000 [19] and Alpha 21264 [24, 27]. Instead of using reservation stations, it utilizes a larger register file called the Physical Register File (PRF), which is larger than the Architectural Register File (ARF) specified in the ISA. Speculative results are written directly into the PRF, and instructions are renamed from architectural to physical registers, with mappings tracked by a Register Alias Table (RAT). This approach has several advantages over Tomasulo's algorithm:

- Data is fetched from a single register file.
- It facilitates speculative execution and precise exceptions.
- The rest of the pipeline can use standard bypass networks.
- A precise state can be recovered by reverting instruction renaming.

In our case, the rename stage, illustrated by Figure 4.7, handles two main tasks:

- **Resource Handling:** This involves mapping instructions to their respective queues outside the front end, categorizing them into integer, floating-point, memory, and control-status registers (CSRs) operations. It also checks if the instructions have enough resources to allocate, such as free slots in the reorder buffer, queue slots, and physical registers if they have destinations.
- **Instruction Renaming:** The renaming unit renames instructions and manages the recovery of the precise state of rename control.

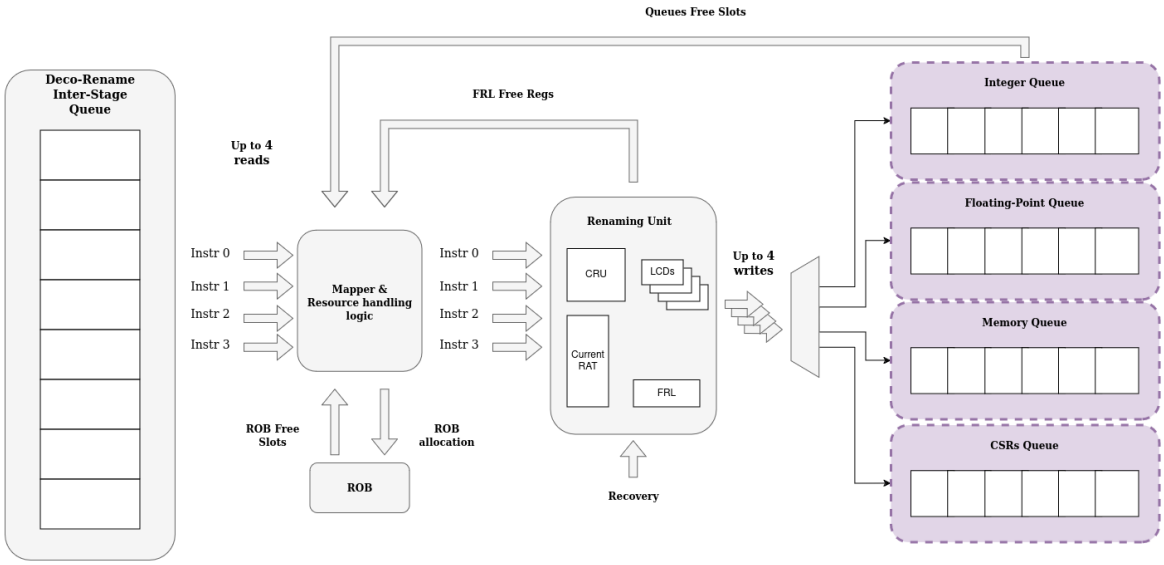


Figure 4.7: Rename Stage diagram.

4.4.1 Resource handling and mapping

After the renaming stage, instructions are placed in issue queues where dynamic scheduling occurs, initiating the out-of-order execution. In our design we use multiple smaller queues, one per execution path, instead of a single larger queue connected to all paths, to reduce the timing and area implications of the wake-up and selection logic for the queues and improve the general scalability.

In our implementation, mapping to these queues is handled in the rename stage logic. Each instruction is sent to its corresponding queue according to the decoded control signals, provided there is a free slot available. Initially, instructions are classified into four expected execution paths, but this classification is configurable and can be adjusted. Our four execution paths are:

- Integer operations.
- Floating-point operations.
- Memory operations.
- Control Status Register operations.

At this stage, instructions are also allocated in the Reorder Buffer (ROB). Since this is the last stage where program order is enforced, instructions are placed in the ROB to ensure in-order committing and are assigned a ROB ID, which identifies the relative age of the instruction throughout the execution.

Moreover, we must check if instructions can be renamed. Instructions without destination registers can always be renamed, but those that produce a value require a free physical register. Additionally, an instruction cannot proceed to the next stage if it lacks the necessary resources or if any older instruction cannot advance, as program order must be maintained at this stage. Therefore, the resource handling logic verifies the following conditions are met before altering the state of any structure in the rename stage:

1. The corresponding next-stage queue has a free slot.
2. There is a free slot in the ROB.
3. If the instruction has a destination register, a free physical register must me available.
4. All older instructions in the rename stage can advance.

If any of these conditions are not met, the instruction cannot advance in the pipeline, nor can it be sent to the renaming unit or allocated in the ROB.

4.4.2 Renaming Unit

This module offers a flexible, high-performance, and comprehensive design for renaming instructions and managing the state of its structures. It translates instructions using ISA architectural registers, also known as logical registers, into physical registers distributed across one or more register files and eliminates WAR and WAW data hazards of an out-of-order execution. The design's flexibility allows it to handle multiple register files for integer and floating-point renaming, the superscalar width, and the size of the physical register file. For simplicity, the following explanation focuses on a single register file and a 4-width superscalar bandwidth.

Additionally, the module supports state recovery, a crucial mechanism in out-of-order designs. It restores the processor to a precise and consistent state at any given point, enabling the reversal of speculative execution effects and precise exception processing. This mechanism is triggered by a recovery request, identified by the instruction's ROB ID, reverting the renaming state as if any younger instruction was never processed. The Figure 4.8 shows the renaming scheme without the recovery features to facilitate the explanation, and it is expanded in next sections.

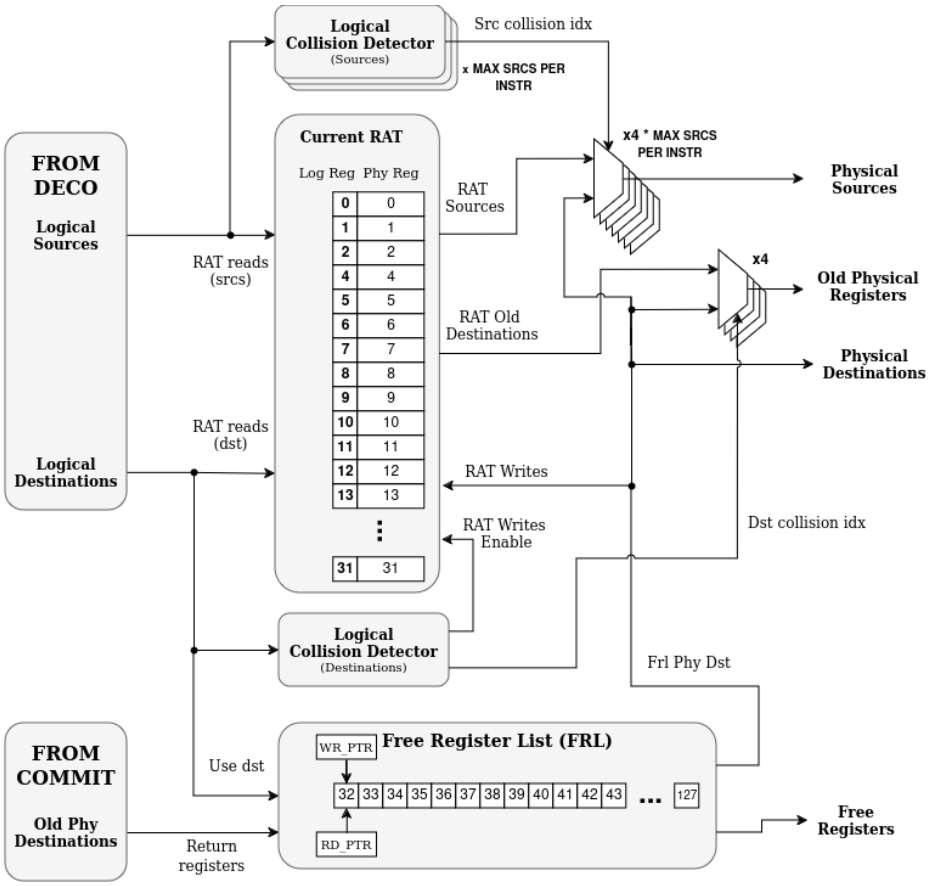


Figure 4.8: Simplified Renaming Unit diagram.

From the decode stage, the relevant instruction information for renaming includes the logical source and destination registers, if present (for example, some instructions have only one source or do not have a destination register). Logical source registers are translated into physical registers by reading the current Register Alias Table (RAT), using the logical register as a read index. The RAT stores the current logical-physical mappings with as many entries as there are ISA registers. Initially, the RAT maps logical registers to physical registers with the same identifier.

Simultaneously, logical destination registers are renamed to physical registers by requesting an unused container from the Free Register List (FRL). The FRL is a list of elements that operates as a FIFO structure and is already present in HLib. In our case, it is initialized with physical registers not present in the RAT after a reset. Therefore, it assigns a free register from the physical register file to store the value produced by the instruction and advances the read pointer of its FIFO control.

New physical destinations issued by the FRL are written in the current RAT, indexed by the logical destination register, updating the RAT state for new instructions in the next cycle. The old destination register, read using the logical destination register as a read index, will be freed when the instruction commits, but more of it in a moment.

In addition, renaming multiple instructions per cycle adds complexity due to potential WAR and WAW hazards within the instruction window, produced by events listed below:

- The source of an instruction is the destination of a younger instruction being renamed in parallel, producing a WAR hazard.
- The destination register is the same as the destination of a younger instruction being renamed in parallel, producing a WAW hazard.

There are two **Logical Collision Detector (LCD)** modules to solve this problem. An LCD is a sub-module that generates a 2D-matrix of comparisons and outputs if there are matching elements (collisions) and, in the case of more than one match, it provides the the index of the lowest and highest colliding ports (to identify the collision with the oldest and youngest instructions).

We are going to use this module for two purposes:

- **LCD for Sources:** Indicates if a source register depends on any destination register of an older instruction, preventing the use of outdated RAT values, and identifies the physical destination register to be used as a source, prioritizing the youngest among them.
- **LCD for Destinations:** On the one hand, it indicates if a destination register is the same as a older instruction, preventing the use of outdated RAT values for the old destination register read. It prioritizes the youngest instruction among the older to be used as true old destination register. One the other hand, it checks for younger instructions with the same logical destination register to invalidate writes of different values in the same RAT entry, since it is not physically possible, and only the youngest value should prevail.

The LCD modules control the multiplexers that select between the registers read in the RAT and those issued by the FRL as new destinations. Physical sources and old physical destinations come from reading the RAT by default. However, when there are WAR and WAW hazards, the module bypasses register mappings occurring during the same cycle to use the most updated information.

The next step is defining the timing and method for freeing physical registers, crucial for avoiding false dependencies. A physical register is freed when it can be ensured that the value in a physical register is no longer needed. This occurs when a younger instruction that writes into the same logical register commits, since the physical register previously mapped to the same logical register won't be needed as all older instructions before it already committed. In other words, when an instruction commits and has a destination register, the old physical register mapping (known as old destination register) replaced in the current RAT is returned to the free list, making it available for future instructions.

Finally, this module delivers fully renamed instructions to the next stage and provides the current availability of the physical register file.

Recovery

Figure 4.9 shows the renaming unit recovery diagram where logic unrelated to context saving and recovery has been hidden, and elements added in the recovery mechanism are highlighted in blue.

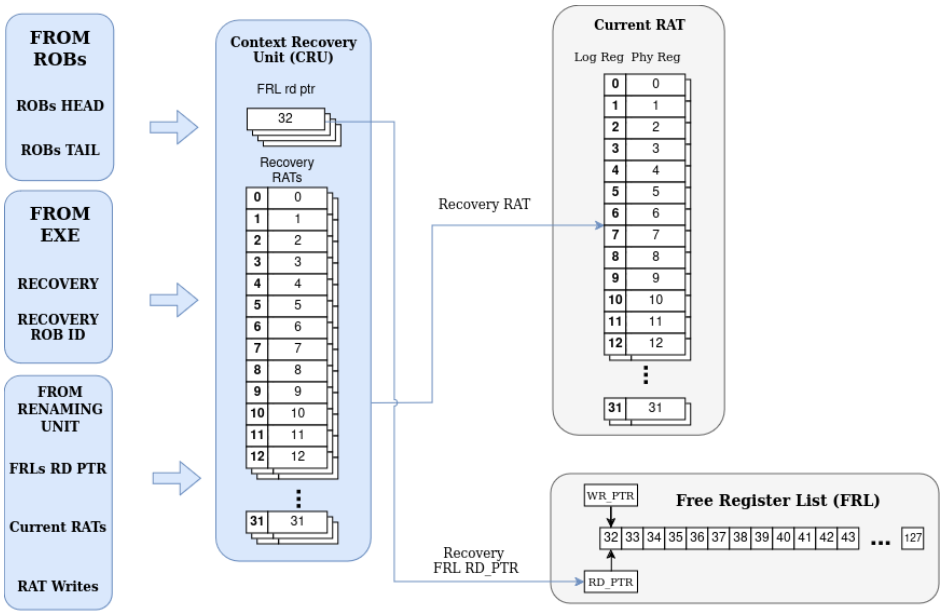


Figure 4.9: Renaming Unit recovery diagram.

The recovery mechanism is vital for reverting speculative execution effects and maintaining a precise and consistent processor state. Triggered by precise exception handling, interruption processing, or speculation errors (e.g., branch mispredictions, memory order violations), it restores the renaming unit's state to any point within in-flight instructions, saving as many states as the ROB's capacity.

The main component responsible for state saving and restoration is the Context Recovery Unit (CRU). Unfortunately, we cannot disclose the internal operative of this module in this thesis, since it will be subject of future works and research at the BSC. However, behaviorally, it saves a snapshot of the renaming unit's state for each renamed instruction. This snapshot includes the RAT table and the FRL read pointer (the write pointer is associated with the old destination register liberation), identified by the instruction's ROB ID. Upon a recovery event, the CRU provides the RAT values and the FRL read pointer, overwriting the current values in these structures and effectively undoing changes made by any younger instructions.

We would like to clarify the recovery handling design is efficient, saving partial changes rather than copying the entire RAT and FRL read pointer for each instruction since otherwise it would be very area-costly and very inefficient. Additionally, this process spans multiple cycles, varying in duration depending on the ROB ID of the recovery, but this latency is masked by the inherent penalty of recovery events, which involves flushing the front-end pipeline and changing the fetch PC. Consequently, when new instructions enter the rename stage post-recovery, the state restoration process has already finished, ensuring no additional stalls in the pipeline. Therefore, the complete diagram of the renaming unit remains as follows in Figure 4.10:

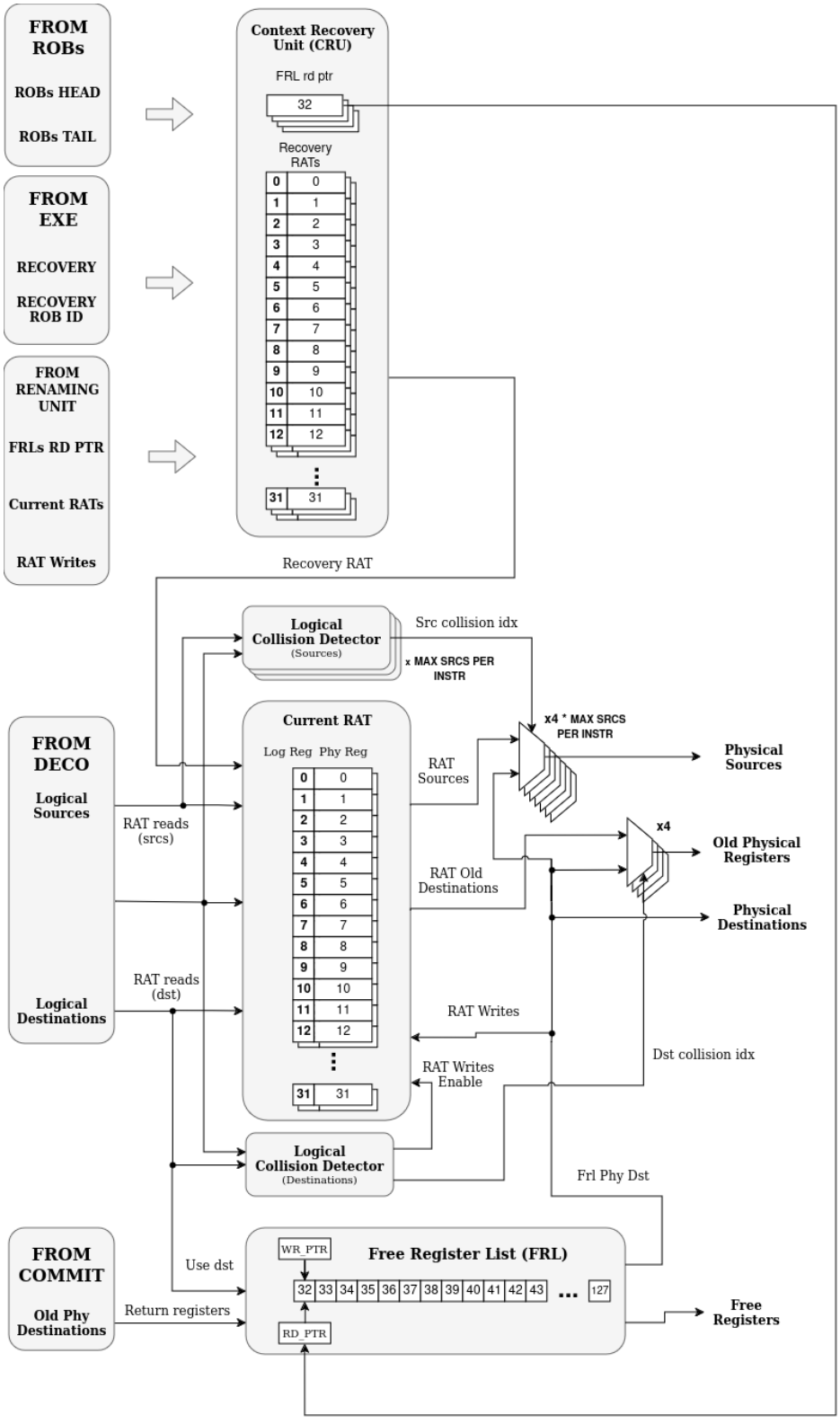


Figure 4.10: Renaming Unit diagram.

4.5 Inter-Stage Connection

In pipelined processors, stages can be connected through either registers or queues, and in this design, we can use both, since we will utilize configurable inter-stage queues. These modules behaves as FIFO lists and can be instanced with different number of elements.

For example, we have the option to configure them with as many elements as the pipelines' width, being functionally equivalent to using inter-stage registers that behave as simple value-storing flip-flops of classical architectures. On the contrary, if we choose queues with more capacity than the pipeline bandwidth, we can decouple stage stalls from each other. This way, if a downstream stage, such as the rename stage, stalls (e.g., because the ROB is full), the fetch and decode stages can continue processing instructions until their respective queues are full and overlap some stalled cycles with useful work.

Additionally, this inter-stage queues are capable of reading and writing as many elements as the stage's bandwidth. Each entry of the queue allocates an instruction and its metadata, minus the queue between fetch and decode, where each entry stores blocks of 16 bits and metadata that can correspond to a compressed instruction or half a regular instruction, and it is the work of the decode stage to identify them.

This flexibility in the design allow us to adopt a conventional architecture, with pipeline stages connected through simple registers, or experiment with a more aggressive design with inter-stage queues of higher capacity and enabling stage decoupling that may improve performance under certain circumstances. In this design, we will take the decoupling queues approach.

4.6 Reorder Buffer

The reorder buffer (ROB) is a crucial component in an out-of-order processor, as it maintains the program order of in-flight instructions while enabling Out-of-Order execution and precise exception handling [3, 4]. Functioning as a FIFO circular queue, the ROB allocates and deallocates (commits) instructions in order. The head pointer of the ROB points to the oldest in-flight instruction, while the tail pointer indicates the youngest.

Once instructions are allocated in the ROB, they can be completed at any time. An instruction can only be deallocated or committed if it has finished its execution and is the oldest instruction in the ROB. If multiple instructions can be committed per cycle, an instruction can only commit if is completed and all the older instructions will be committed in the same cycle. Each instruction allocated in the ROB is assigned a tag, known as the ROB ID, which indicates its relative age compared to the oldest in-flight instruction. This tag remains unique and is not reused until the instruction commits or is discarded. The tag serves as a unique identifier throughout the pipeline. The ROB also supports the recovery of any allocated instruction, allowing it to restore the state by flushing its entry and any younger instructions in it.

In addition to tracking the relative age of instructions, the ROB stores essential information to control instruction flow. For instance, it keeps the PC of the instruction for purposes such as computing the target address of PC-relative branches and jumps. The PC also facilitates restarting execution from any in-flight instruction by switching the fetch address. Other stored data include the old physical destination register used in renaming (freed upon commit) and branch prediction-related information. With explicit renaming,

speculative results are written directly to the register file, eliminating the need to store them in the ROB. Although there are no fixed rules about what information must be included in the ROB, and it varies for each platform, it is advisable to avoid oversizing the ROB with unnecessary metadata that could be pipelined or obtained by other means.

Fortunately, HLib provides a parametric and flexible ROB design and implementation that can be adapted to meet our requirements.

4.7 Frontend Modifications for Multithreading Support

This section describes the modifications applied to the frontend design to enable multithreading support. Our goal is to maximize the reuse of the core's computational resources across multiple threads, dynamically sharing elements such as the backend and caches among threads. However, while some frontend structures can be shared, many others must be partitioned or replicated to ensure thread isolation. This isolation is essential because, from a software perspective, each thread should function as a complete and independent logic CPU. Consequently, the execution of one thread must not affect the correctness of others.

As discussed in previous chapters, various multithreading models come with different trade-offs. Our target is a Simultaneous Multi-Threading (SMT) front-end, which aligns with our goal of improving throughput for server workloads with parallel tasks, and enhancing the implementation efficiency by addressing vertical and horizontal waste. We aim to support two threads in our multithreading design, but as other elements of the design, it remains configurable. Although this design can be scaled using a multi-core approach, our focus in this thesis is on in-core multithreading.

Most frontend stages, such as decode and rename, can be naturally extend to support SMT, as they do not rely directly on external structures outside the data-path. However, the fetch stage case is different because it interfaces with a blocking and single-port instruction cache, preventing parallel instruction requests from different threads. We proposed two alternatives to handle this restriction:

- **Replication of the Fetch Stage:** Each thread could have its own fetch stage and instruction cache, feeding a multithread inter-stage queue in parallel. However, due to area constraints and the current state of the project, this option was discarded, as instruction caches are expensive components of the micro-architecture [3].
- **FGMT Fetch Stage:** Since front-end stages can be decoupled, we could design the fetch stage using FGMT, allowing address requests from different threads in alternating cycles [3]. We chose this approach for several reasons:
  - It supports multithreading with a single instruction cache.
  - It can be combined with the first proposal of fetch stage replication to reduce the number of instruction caches. For example, with four threads, two FGMT fetch stages could share two instruction caches, each cache handling two threads.
  - Since stages are decoupled from stalls, instructions can be fetched in FGMT and consumed in SMT. Moreover, if the fetch stage bandwidth exceeds the decode stage's it can produce more instructions than a decode stage can consume, effectively using SMT when instructions from multiple threads accumulate.

Additionally, this design allows for enabling and disabling threads dynamically by stalling or resuming their instruction fetches. Disabling a thread can enhance the single-thread performance of the remaining active threads since the FGMT fetch stage selects threads from an active pool. This can be controlled using various heuristics, such as Reorder Buffer (ROB) occupation per thread or progress imbalance among threads. Although, we leave this management open to to be exploited in the future.

In following subsections we will outline specific modifications made to each stage and structures to support multithreading. Nevertheless, the following Figure 4.11, condensates all the changes made to the frontend, highlighting in lilac external elements and in light blue multithreading modifications.

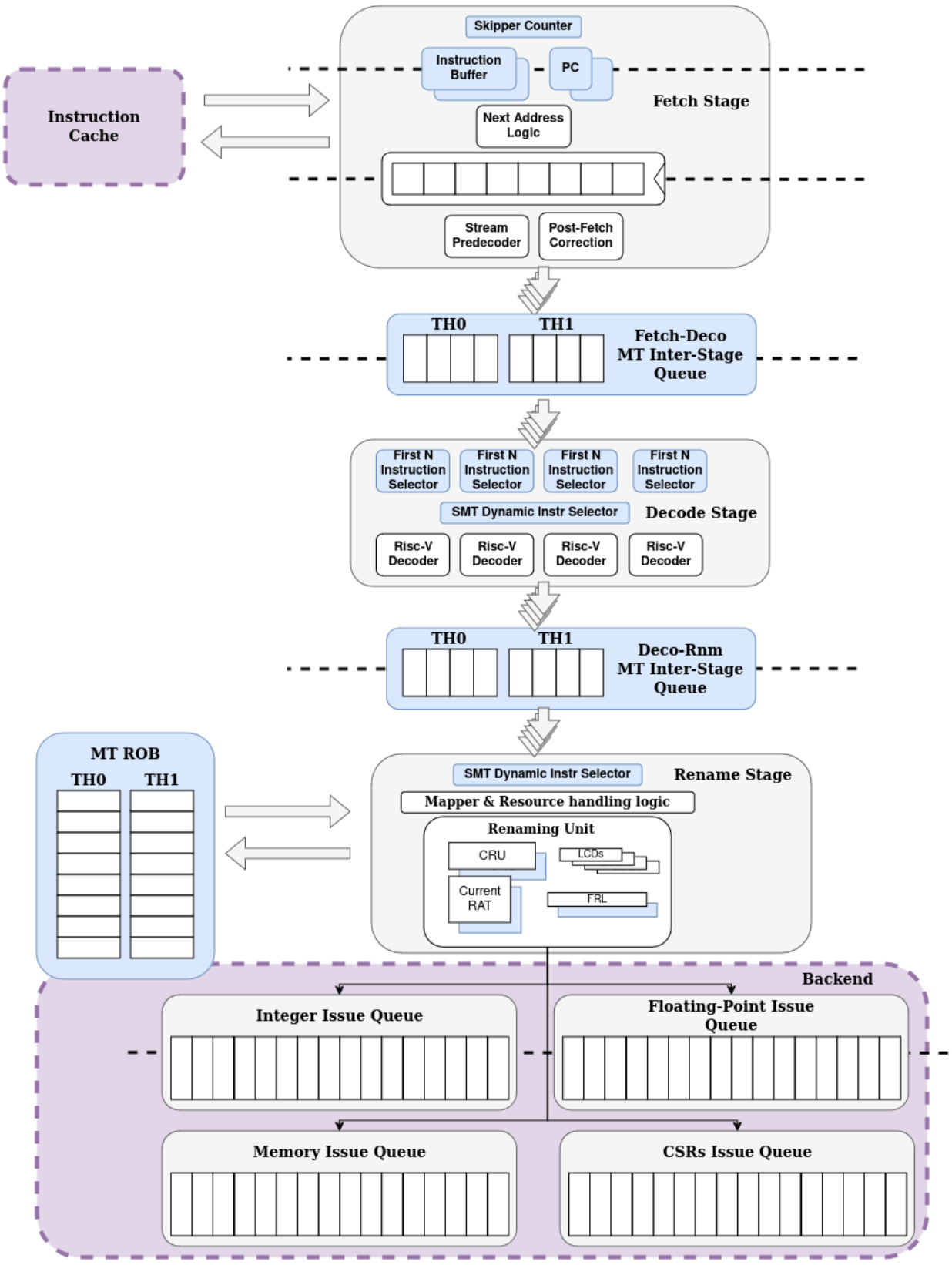


Figure 4.11: Multithreaded Frontend diagram.

4.7.1 Multithreaded Fetch Stage

The fetch stage occupies a unique position in the pipeline, affected by the presence of a single-port, blocking instruction cache, which prevents a true simultaneous multithreading (SMT) design like in the rest of the pipeline. Instead, we implemented a fine-grained multithreading (FGMT) approach for this stage, where each cycle a request is sent for a different thread from the active thread pool.

To isolate multiple threads from each others, certain structures need to be replicated, specifically the program counter (PC) register and the instruction buffer, as they must remain untainted from the execution of other threads. Additionally, a new sub-module called the Skipper Counter is introduced. This module receives a pool of active threads and selects a different one each cycle in a round-robin manner to request its address in the first cycle of the stage. The Skipper Counter is a general-purpose module available in the HLib library.

Additional considerations related to jumps (*JALs*) and stalls must be addressed. For *JALs*, detected by the Stream Predecoder and handled by the Post Fetch Correction module, a flush is typically produced in the first two cycles of a single-thread fetch due to a change in the instruction flow. However, in multithreaded execution, instructions from other threads should not be affected by these flow changes. If there are instructions from other threads during these cycles, they will proceed without flushing, and only the PC register of the affected thread will be modified. Regarding stalls, instructions from stalling threads must not block the stage for other threads. In such cases, the instructions from stalling threads are flushed, and their PC addresses are restored.

Finally, fetched instructions are then stored in a multithreaded inter-stage queue, labeled by their thread identifier. The rest of the design remains unchanged as it does not need to distinguish which thread's instruction flow is being fetched. Figure 4.12 summarizes the changes applied to the design.

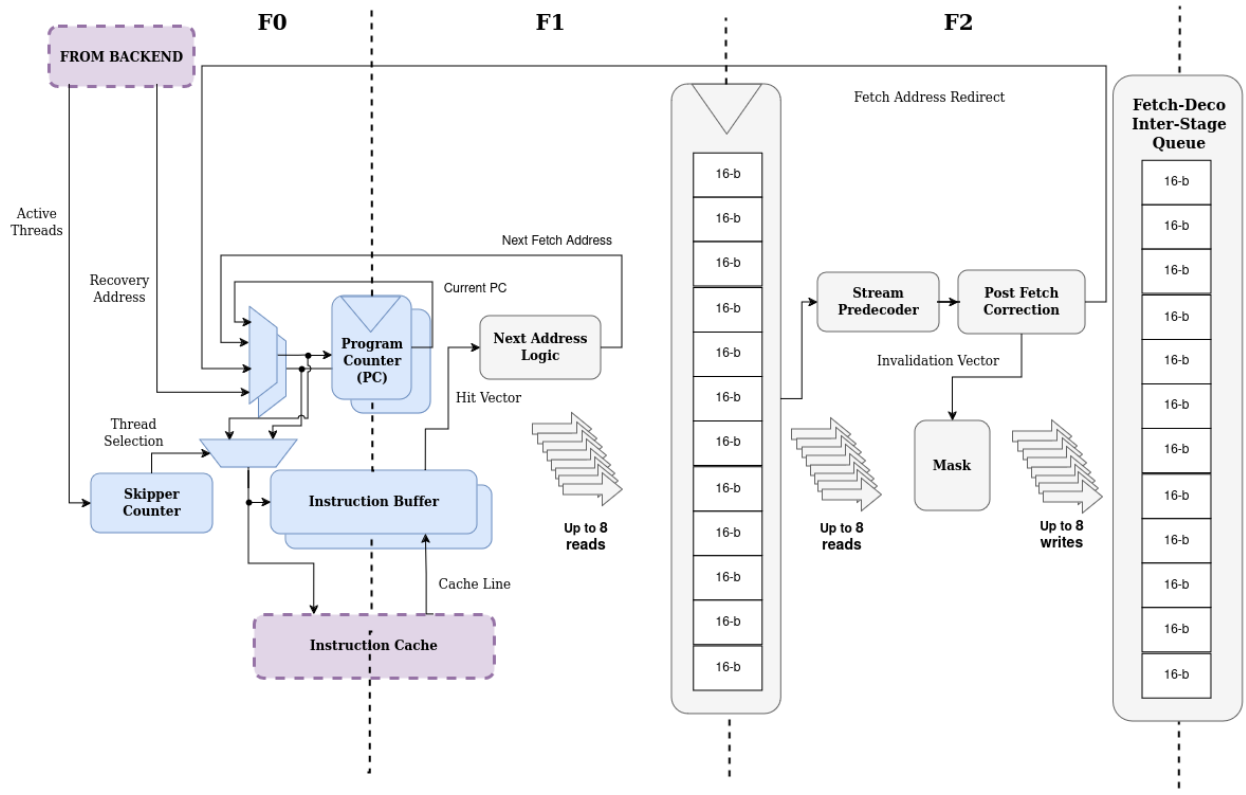


Figure 4.12: Multithreaded Fetch Stage diagram.

4.7.2 Multithreaded Decode Stage

The decode stage requires a few modifications to include support for SMT execution, though the decoder itself remains unchanged. The first modification involves replicating the First N Instr Selector for each thread. This allows the system to determine how many instructions can be decoded for each thread, thereby optimizing the SMT selection process. This is necessary because we cannot know in advance how many instructions are available for each thread without first analyzing the 16-bit blocks and reconstructing instructions, which would otherwise make the selection sub-optimal.

Once the number of available instructions per thread is known, the second modification involves introducing a new module called the SMT Dynamic Instruction Selector, which is described in detail in Section 4.7.6. But, as a simplified explanation, the role of this instruction selector is to determine which instructions from which threads are allowed to proceed to the next stage. This selection is based on a dynamic and rotating priority system, ensuring efficient and fair instruction processing in an SMT execution. Figure 4.13 illustrates the effective changes in decode stage.

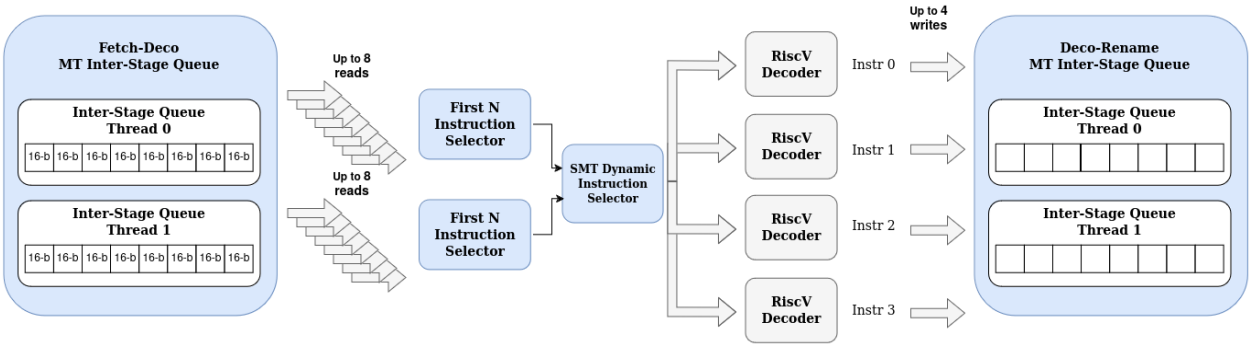


Figure 4.13: Multithreaded Decode Stage diagram.

4.7.3 Multithreaded Rename Stage

The rename stage is adapated to SMT, and is crucial in determining how backend resources are shared. In the renaming unit, each thread requires its own dedicated Register Alias Table (RAT) since the mapping of architectural registers must remain unaffected by other threads' executions. Initially, the plan was to dynamically share the Physical Register File (PRF), the Context Recovery Unit (CRU), the Reorder Buffer (ROB), and other backend resources. Nevertheless, dynamically sharing the state-saving capacity of the CRU between threads presented a challenge of significant complexity. Addressing this issue would far exceed the scope of this project, necessitating a complete redesign of the module and its underlying concept. The recovery of the renaming state in a multithreading processor introduces additional layers of complexity. In this context, the state must be identified not only by an instruction's ROB ID but also by the thread to which it belongs. Changes in the state should affect only the thread causing them, and the state recovery of one thread should not interfere with others. Furthermore, the renaming scheme for multithreaded execution must permit one thread to continue renaming instructions while others threads are recovering their state. Otherwise, a stall in a single thread would block the entire stage, contrary to the principles of multithreading. Additionally, it may be necessary to recover the states of multiple threads simultaneously to maximize stall overlapping and avoid throughput losses.

Given the complexity and effort required to design such an architecture, we decided to replicate a CRU for each thread and modules affected by the recovery, which involves the physical register file (statically partitioned) and the FRL (replicated). This conservative approach meets all the state allocation, isolation, and recovery throughput requirements at the cost of area overhead. Additionally, because the CRU mechanisms are parallel and isolated from each other, they do not increase the critical path of the module, as will be demonstrated in the synthesis analysis chapter. Figure 4.14 highlights in blue the replicated structures and newly connected signals in the renaming unit.

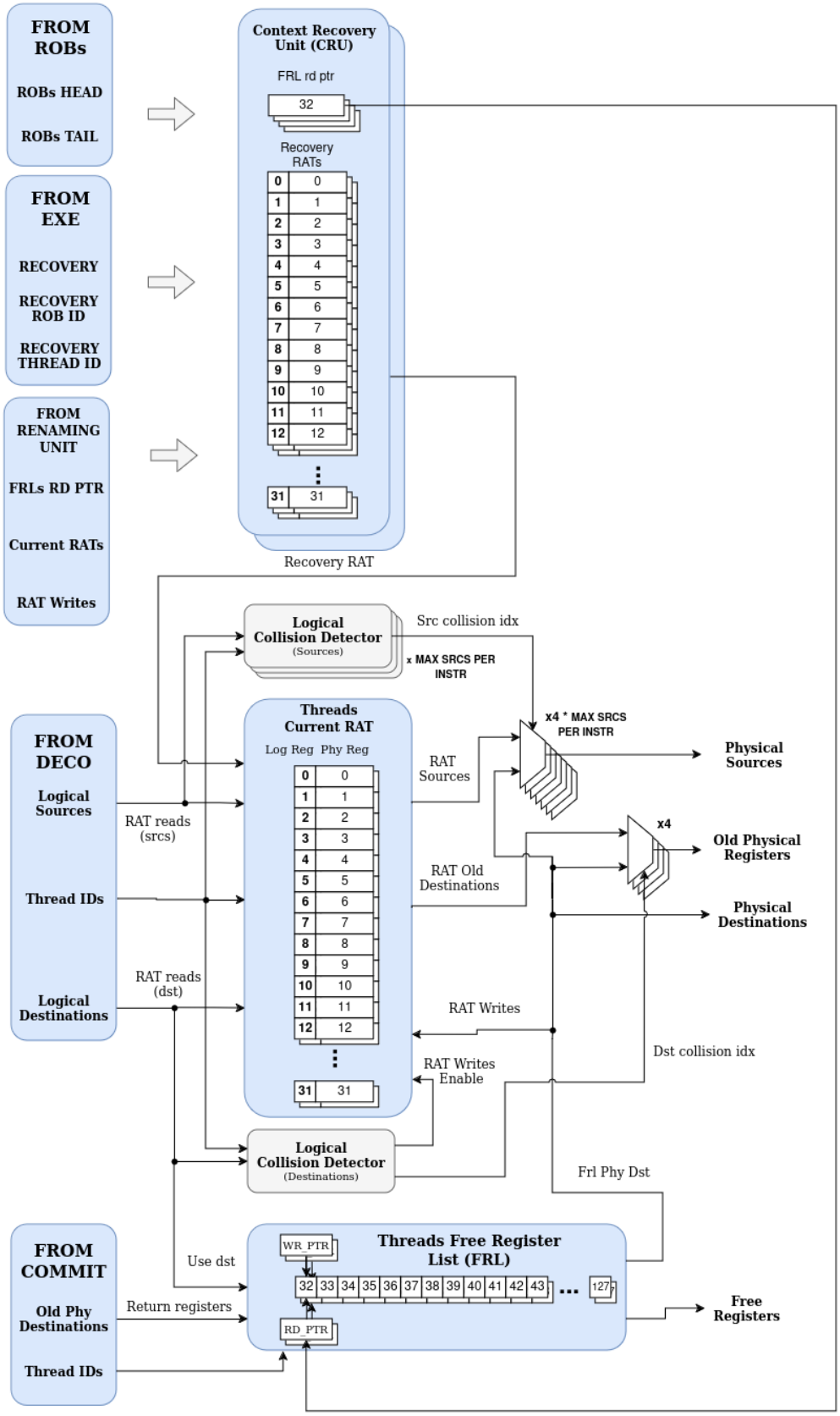


Figure 4.14: Multithreaded Renaming Unit diagram.

The rest of the rename stage remains unchanged, as shown in Figure 4.15, except for the inclusion of the SMT Dynamic Instruction Selector, detailed in the following Section 4.7.6. This selector, positioned after the multithread inter-stage queue, determines which instructions and which threads have access to the renaming stage in a round-robin rotating manner.

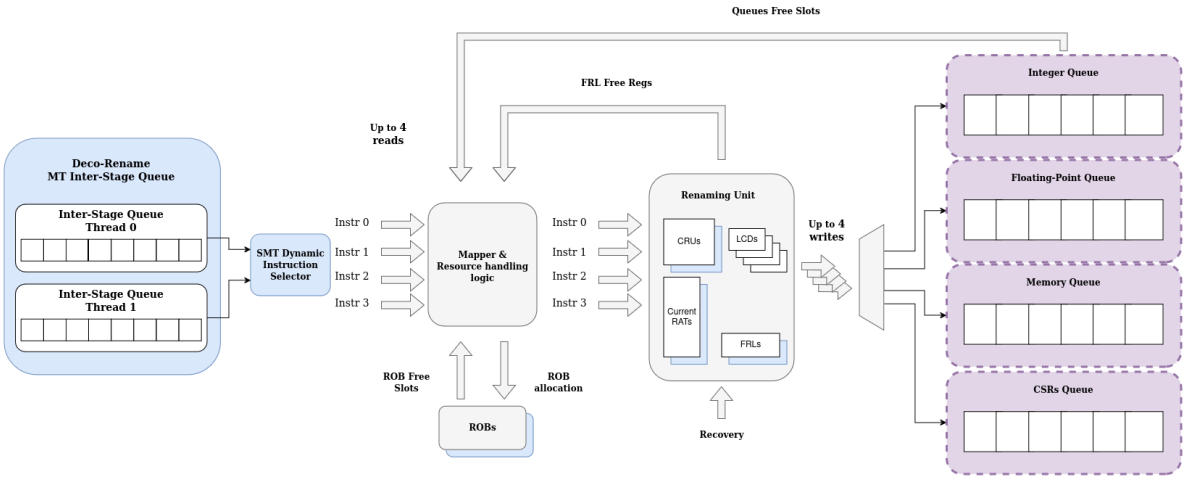


Figure 4.15: Multithreaded Rename Stage diagram.

4.7.4 Multithreaded Inter-stage Connection

The multithreaded inter-stage queues that interconnect the pipeline are based on the inster-stage queues presented earlier in the Section 4.5, and adding some modification required for multithreading support. The entries of the queues are statically partitioned among the threads, meaning that for each thread, the allocation capacity is the number of total entries of the queue divided by the number of threads. Therefore, the total size of these queues has been increased to not hurt the single-thread performance if some of the threads are inactive. Additionally, these queues include logic to reorganize reads and writes for each thread's queue, ensuring that they remain contiguous and in order, adhering to FIFO requirements. This is crucial since reads and writes from different threads can arrive mixed in a SMT execution.

We considered dynamically sharing the inter-stage queue entries among threads. However, it would imply a complete redesign of the module and, given the time constraints of the project we decided to adopt the static approach.

4.7.5 Multithreaded Reorder Buffer

As with other modules, a dynamically shared reorder buffer was considered. However, the major complexity in the redesign to support thread isolation for allocation, state recovery and communication with other modules discouraged us to pursue a dynamic sharing design.

Instead, we opted to statically partition the ROB among the threads, creating smaller reorder buffers for each thread and keep its regular behavior. Since instructions can arrive interleaved with those from other threads, but in order within each thread, we need to direct them to their corresponding thread's ROB and ensure

they are aligned contiguously in their allocation. This alignment is necessary because the single-thread ROB only supports contiguous writes on allocation, as instructions must arrive in program order. Figure 4.16 reflects these changes in ROB design.

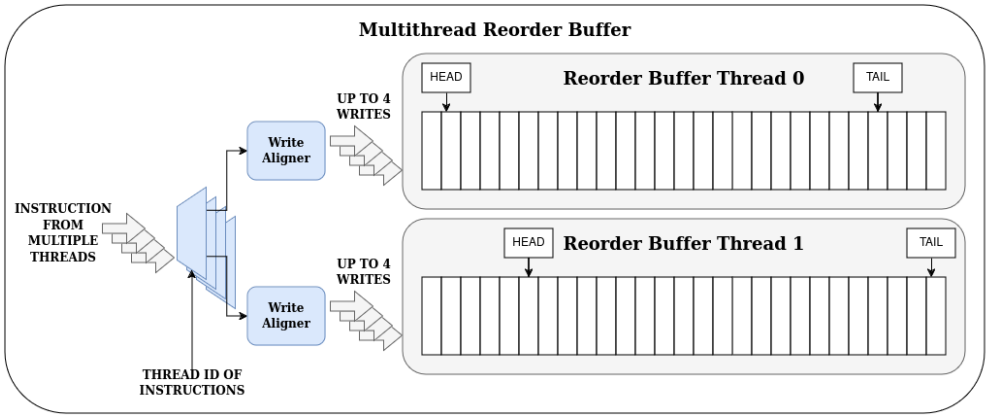


Figure 4.16: Multithreaded Reorder Buffer diagram.

4.7.6 SMT Dynamic Instruction Selector

In any SMT architecture, a crucial aspect is the decision process of instruction selection across all threads. In our design, we guarantee a local throughput optimality, which means we maximize bandwidth usage of a stage, if there are enough instructions across all contexts. Nevertheless, local throughput optimality does not consider how many threads are used, their priorities, etc. Therefore, there are other factors that should be considered for every SMT architecture.

Since all threads compete for the same hardware resources, we must provide a mechanism to avoid thread starvation, by adopting fairness policies in the thread selection. Our design provides a dynamic thread selection priority, that affects the thread order on which instructions are selected, and can be controlled by many heuristics. For instance, it can be controlled by the ocupation status of the ROBs, prioritizing threads with less instructions in-flight, the number of total instructions executed per thread, detecting a long latency event in one of the threads, etc. Since these heuristics can depend on the complete core architecture, we leave this possibility open as an easily modifiable element. However, we currently implement it through a round-robin schedule, which changes the threads priorities each cycle.

Moreover, and despite guaranteeing local throughput optimality there is in an inherent trade-off between single-thread performance (prioritizing one thread's instructions and then selecting from others if bandwidth is available) and Thread-Level Parallelism (TLP) in the selection window (promoting instructions from as many different threads as possible).

The SMT Dynamic Instruction Selector can be configured to prioritize one of both properties while maximizing local throughput, due to the inherent compromise. On the one hand, it can prioritize single-thread performance by selecting instructions from other threads only if a single thread does not fully utilize the bandwidth. On the other hand, by prioritizing TLP the single-thread performance is reduced, but total throughput is benefited as a result, allowing all threads to compete for the bandwidth until it is full or no instructions remain for each cycle. Additionally, intermediate configurations are possible, offering a

middle-ground that can be useful in some scenarios.

To enable this flexible configuration of this antagonistic properties of the design, we introduce the concept of skewness, which is a parameter ranging from 1 to the stage's bandwidth. Skewness quantifies how many instructions from one thread are prioritized before moving to the next thread in a round-robin fashion. For instance, a skewness of 1 prioritizes TLP, while a skewness equal to the stage's bandwidth prioritizes single-thread performance. Our design adopts a middle-ground approach with a skewness of 2 for a bandwidth of 4, adjusting timing constraints, although any configuration can be easily adopted. This approach ensures flexibility in balancing single-thread performance and a fair degree of TLP, improving general throughput.

Since the module is highly different depending on the skewness, there is no fixed design and thus, we won't provide a high level diagram. Instead, we provide a couple of example that varies in thread priority and skewness, illustrated in Figures 4.17, 4.17 and 4.19.

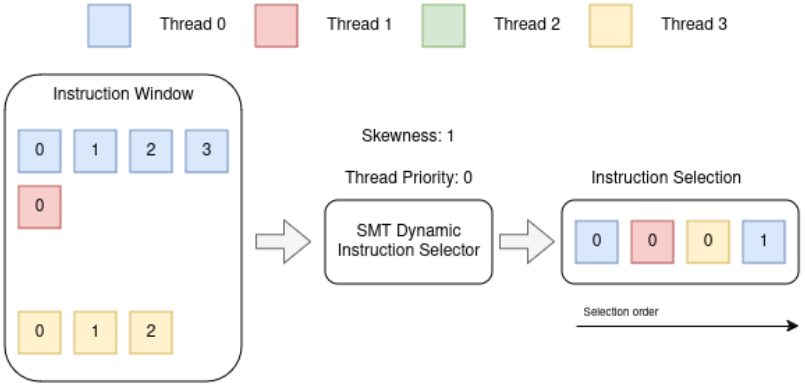


Figure 4.17: SMT Dynamic Instruction Selector scenario 1.

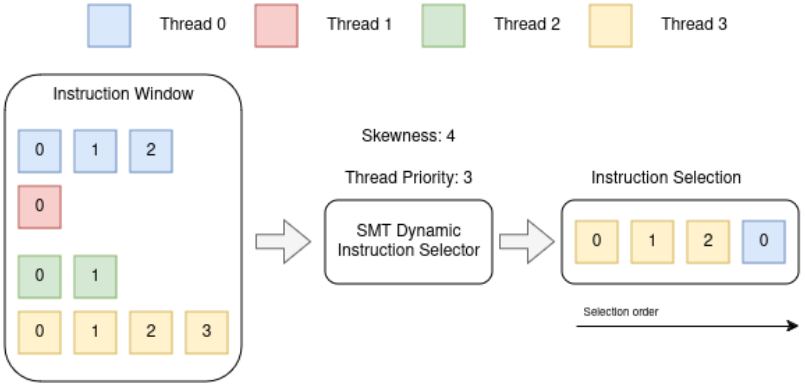


Figure 4.18: SMT Dynamic Instruction Selector scenario 2.

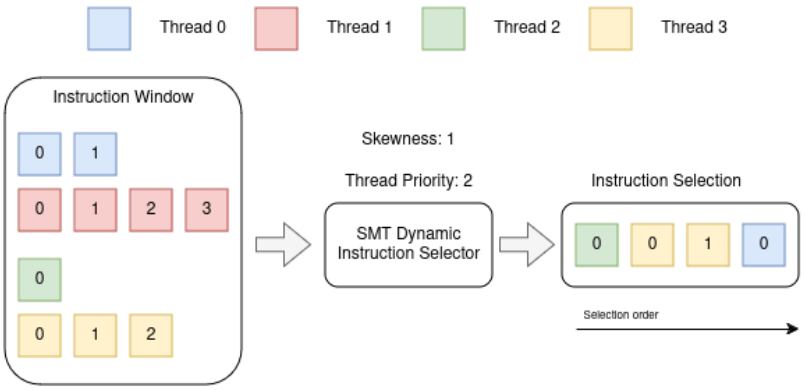


Figure 4.19: SMT Dynamic Instruction Selector scenario 3.

Figure 4.17 shows a SMT Dynamic Instruction Selector configured with a skewness of 1, prioritizing TLP and in this given moment the thread with highest priority is 0. In this case, the selection starts with the first instruction of the thread 0, since it has a skewness of 1, it goes to the next thread, to continue the selection and picks the instruction 0 of thread 1, and so on. Since it cannot fill the bandwidth in the first iteration of the search, as thread 2 did not have any electable instruction, it does a second iteration and selects instruction 1 of thread 0 to fill the bandwidth.

Figure 4.18 on the contrary prioritizes single-thread performance with a skewness of 4. In this case, the thread 3 has highest priority, so the selection begins with it. The bandwidth is filled with as many instructions from said thread as possible, in this case instructions 0, 1 and 2. Since a single thread is not enough to fill the bandwidth, it completes the selection with the next thread in the priority, picking instruction 0 from thread 0.

Finally, Figure 4.19 with a skewness of 2 has starts with the single instruction of the thread 2. Then selects the 2 instructions of the thread 3, the next in the priority, and finally, fills the bandwidth with the first instruction from thread 0.

Chapter 5

Implementation

The next stage of this project involves implementing the design at Register Transfer Level (RTL). We have chosen SystemVerilog [29] as our Hardware Description Language (HDL) due to its widespread use, extensive documentation, and compatibility with HLib, whose coding guidelines and conventions we will adopt [11]. Additionally, we will utilize Verilator, a fast, open-source RTL simulator that supports SystemVerilog and performs code-quality checks [9]. Simulation of the RTL implementations will be crucial for verifying our code, as will be discussed in the following Chapter 6.

In addition, the HLib methodology emphasizes flexibility and re-usability, given its nature as a hardware library. Therefore, one of our primary goals in implementing our design is to ensure extensive parameterization, making it as general-purpose as possible. This approach allows the design to be adaptable for use across a wide range of cores, enhancing its versatility and utility in various applications.

Due to the extensive list of modules developed in this project, detailing all their interfaces and parameter options would be overly lengthy and potentially tedious. Instead, we'll provide an overview of the most relevant and useful parameters for configuring the front-end engine. This approach ensures we highlight key configuration elements without unnecessary verbosity, making the document more concise and reader-friendly.

Disclaimer: Although numerous configurations are possible and functionally correct, they carry various implications for the physical implementation, particularly in terms of area and frequency. Not all elements in the design scale equally well. Nevertheless, scalability has been one of the primary objectives of our implementation, and we have endeavored to minimize overhead wherever possible. However, in many cases, the inherent algorithmic behavior presents certain limitations. In the Chapter 7, we synthetize and analyze in detail the scalability of many modules in several configurations.

5.1 Superscalar Width

The frontend supports the superscalar parametrization, allowing it to be part of a 2-width, 3-width, 4-width, or 8-width superscalar core. This enables the design to target from more modest 2-width superscalar configurations with a lower performance and less area footprint, to more ambitious 8-width high-performance

and expensive implementation, depending of the requirements. Additionally, this flexibility it provides can make it compatible with many backend designs.

5.2 Thread Contexts

This implementation supports any desired number of thread contexts, from single-thread execution to an aggressive multithreading front-end with four or more threads. However, as explained in the design section, this requires replicating some structures, impacting the implementation costs, as we will further analyze.

5.3 RISC-V Extension

The decoder module can be configured to support specific sets of extensions. The current configurations supports RV64GC extensions of the ISA. However, any valid subset of extensions can be used, such as RV64IM for 64-bit integer operations, multiplications, and divisions. It also supports other ISA bases, such as RV32 and RV128. The decoder configuration options include:

One ISA base from Table 5.1:

Base	**Description**
RV32I	Base Integer Instruction Set for a 32-bit architecture
RV64I	Base Integer Instruction Set for a 64-bit architecture
RV128I	Base Integer Instruction Set for a 128-bit architecture

Table 5.1: Available RISC-V base ISAs for our decoder implementation.

Additionally, the configuration can include any valid combination of extensions from Table 5.2:

Extension	**Description**
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point
Q	Standard Extension for Quad-Precision Floating-Point
Zicsr	Control and Status Register (CSR) Instructions
Zifencei	Instruction-Fetch Fence
C	Standard Extension for Compressed Instructions

Table 5.2: Available RISC-V extensions for our decoder implementation.

Note: Not all combination of extensions are valid, since some extensions imply the existence of others. For example, the double-precision floating-point extension (D) requires the single-precision floating-point extension (F).

Finally, the decoder is a special case since it is not fully implemented in SystemVerilog and it is complemented

with other tools. A more detailed explanation can be found in Annex A.

5.4 Structural Allocation Capacity

All allocation structures can be configured with different capacity sizes. Examples of this are the number of entries in the inter-stage queues, the reorder buffers, the cache lines allocated in the instruction buffers, registers in the free-register lists, etc. This flexibility will allow us in the future to experiment and optimize how many area resources are dedicated to critical components of the processor and how they affect to its performance, since a well-tailored micro-architecture with high utilization rate is a key aspect of a processor design success.

5.5 Renaming micro-architecture

The renaming scheme follows an explicit rename mechanism requiring the logical register file defined by the ISA (determining the number of entries in the RAT) and the physical register file capacity (determining the size of the FRL) where values are stored. This implementation is agnostic of RISC-V, meaning the module can be used for other architectures by defining the desired number of architectural registers, the maximum number of sources per instruction, the number of physical registers, etc.

Importantly, the design can handle multiple register files (each with its isolated RAT and FRL), allowing separate physical register files for different purposes. This reduces the number of read and write ports compared to a unified register file approach. For example, we can separate integer and floating-point registers or we could add a separated vector register file for vectorial execution, or we could even define a custom extension that uses a new set of logical registers with a different entries. These multiple register files are completely independent in configuration and can vary in size (e.g., integer registers can outnumber floating-point registers, could be a size not power of two, etc). This flexibility can adapt the renaming scheme to many designs and architectures.

5.6 Thread Selection

This design provides a lot of flexibility in how the threads are handled across the pipeline. From dynamically enabling or disabling the progress of specific threads, to benefit the single-thread performance of remaining active threads, or thread priorities that influence the instruction selection across threads and that be altered per cycle basis. To the static configuration of the thread skewness parameter, which defines single-thread aggressiveness in SMT thread selection to fill the bandwidth of the front-end engine, determining how many instructions from a single thread are selected before searching for available instructions from the next thread in priority. This configuration space gives the design flexibility to be adapted many different workload environments, where the main target could be single-thread performance or to maximize Thread-Level Parallelism (TLP) exploitation, and favour some threads' execution over others.

Chapter 6

Verification

This chapter outlines the verification process for the design, a crucial aspect of the hardware design cycle that often takes as much time, if not more, than the design and implementation of the RTL itself. To achieve this, we employ three open-source tools, detailed in the following sections.

6.1 Verilator

Verilator is an open-source tool that reads Verilog and SystemVerilog hardware description code, performs linting, and compiles it into multithreaded C++ or System-C models for bi-state cycle-accurate design simulation [9]. It is widely used in academia, open-source communities, and commercial semiconductor development for creating co-simulation environments. For this project, we will use Verilator version 5.014, released in August 2023.

6.2 CoCoTB

The COroutine based COsimulation TestBench (CoCoTB) is an open-source tool used to build verification environments for RTL designs using Python [10]. It works with many simulators, including Verilator, and connects stimulus to the models for simulation. CoCoTB provides easy-to-use synchronization primitives such as clock awaits and timers to facilitate testbench generation. Figure 6.1 shows a simple example of a testbench for an adder module, where inputs are generated, the model is simulated with synchronization primitives, and the output value is checked.

```python
@cocotb.test()
async def adder_randomised_test(dut):
    """Test for adding 2 random numbers multiple times"""

    for i in range(10):

        A = random.randint(0, 15)
        B = random.randint(0, 15)
        RES = A + B

        # Assigns value to the input ports of the DUT
        # Desing Under Testing (DUT)
        dut.A.value = A
        dut.B.value = B

        # Awaits for the RTL model synchronization
        await Timer(2, units="ns")

        # Check the results
        assert dut.X.value == RES
```

Figure 6.1: CoCoTB simple testbench code of an adder module.

6.3 GTKWave

When Verilator simulates the model with the inputs from the testbench, it generates a recorded data file (with a .vcd extension) containing the values of each external and internal signal at any synchronization event. This file is useful for a detailed examination of the design behaviour, since it records individual signals. We will use GTKWave, an open-source wave viewer [30], to analyze these signals when the testbench detects a bug. The following image is an example of VCD analysis using GTKWave.

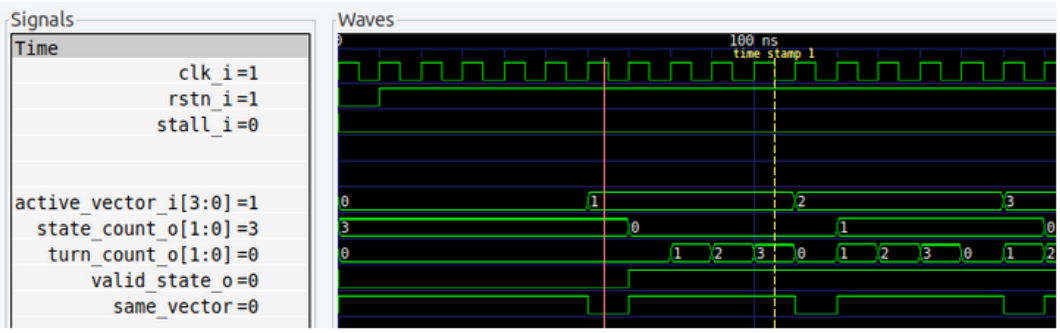


Figure 6.2: GTKWave viewer example.

6.4 Verification Engineering Cycle

Figure 6.3 presents the verification engineering cycle of this thesis closely following HLib methodology [11].

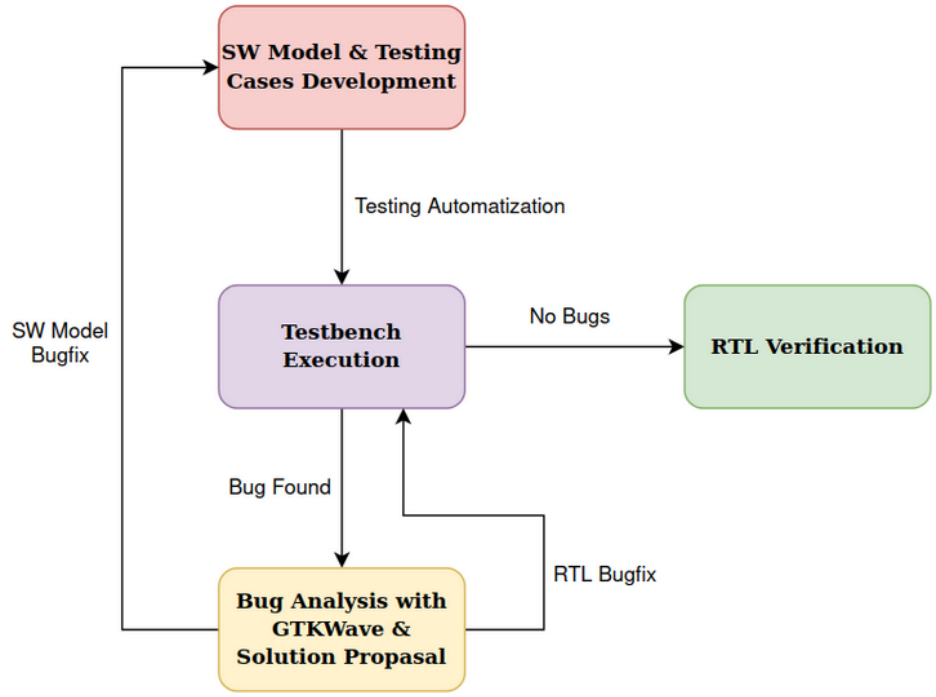


Figure 6.3: Verification engineering cycle.

- **SW Model & Testing Cases Development:** The first step is defining and implementing software models and test cases in Python. These models are software abstractions of the module, replicating the RTL behavior cycle by cycle or producing final results at the end of the testbench. The testbench synchronizes the software model with the RTL, automating output value checks. The goal is to cover as many cases and functionalities of the RTL module as possible to ensure correct behavior under various circumstances.
- **Testbench Execution:** The testbench is executed using the CoCoTB environment and the Verilator RTL simulator. Test logs and output checks identify any mismatches between the software model and the RTL. If any bugs are detected, further analysis is required; otherwise, the RTL implementation is considered verified.
- **Bug Analysis with GTKWave & Solution Proposal:** If a bug is detected, the next step is to analyze its source and determine which side of the simulation has the discrepancy. For in-depth analysis, GTKWave is an excellent tool for visualizing individual signals within the RTL implementation. This helps determine whether the bug originates in the RTL or the software model, leading to a proposed bug fix.
- **RTL Verification:** This stage is achieved when no discrepancies remain between the software model and the RTL implementation, and the covered cases and functionalities ensure the module behaves as expected in tested situations. Although complete coverage of all cases cannot be guaranteed, a thorough model and testbench can provide a high degree of confidence with reasonable effort.

Finally, each module developed in this work is verified with its own testbench. In the following sections, we will overview the most significant models and tests, representing the core of the verification effort.

6.5 Fetch Stage Testbench

In this section, we describe the verification efforts undertaken for the fetch stage, aiming to ensure the correct instruction flow of retrieved instructions. The approach involves executing a small program with an unconditional loop for each thread, with the testbench verifying that the loops execute correctly several times. The objectives of this test are:

- **Instruction Fetching:** Ensuring instructions are fetched and reach the end of the stage.
- **Sequential Flow:** Verifying the correct progression of the instruction sequence.
- **Predecoding and Correction:** Checking that the Stream Predecoder and Post-Fetch Correction identify and execute JAL instructions.
- **Program Counter (PC) Update:** Confirming the PC is correctly updated.

The used program is the following:

```
.section .text
.global _start

_start:
th_0_branch_target:
    add     t1, a1, t0
    add     t1, a1, t0
    add     t1, a1, t0
    add     t1, a1, t0
    add     t2, a1, t0
    add     t2, a1, t0
    add     t2, a1, t0
    add     t2, a1, t0
    add     t1, a1, t0
    add     t1, a1, t0
    add     t1, a1, t0
    add     t1, a1, t0
    add     t2, a1, t0
    add     t2, a1, t0
    add     t2, a1, t0
    j th_0_branch_target
th_1_branch_target:
    add     t1, a1, t0
    add     t1, a1, t0
    add     t1, a1, t0
    add     t1, a1, t0
    add     t2, a1, t0
    add     t2, a1, t0
    add     t2, a1, t0
    add     t2, a1, t0
    add     t1, a1, t0
    add     t1, a1, t0
    add     t1, a1, t0
    add     t1, a1, t0
    add     t2, a1, t0
    add     t2, a1, t0
```

```
37      add     t2, a1, t0
38      j th_1_branch_target
```

Listing 6.1: Fetch test assembler program.

It is a simple program with different loops for each thread, with identical content except for the jump instructions. This variation tests the handling of instruction flow changes and the management of different addresses for each thread.

In addition, an instruction cache model was developed for this testbench, which connects to the RTL implementation. The following class diagram illustrates this setup:

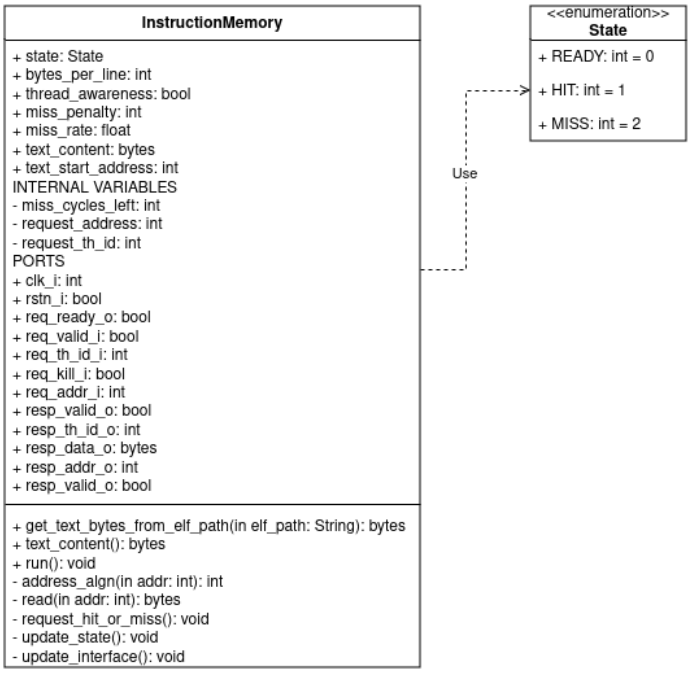


Figure 6.4: Instruction Cache class flow.

The **InstructionMemory** Python model simulates a blocking single-port instruction cache. It includes several public attributes used as input and output signal ports for the RTL module. The testbench assigns values to the input ports (noted with _i suffixes) and reads values from the output ports (noted with _o suffixes). These ports are processed using the *run()* method, which synchronizes the model by executing private methods like *updatestate()* for state updates and *read()* for read operations. An auxiliary enumeration class, **State**, represents the cache's current status: ready to receive requests, hit, or miss.

Figure 6.5 illustrates the testbench operative:

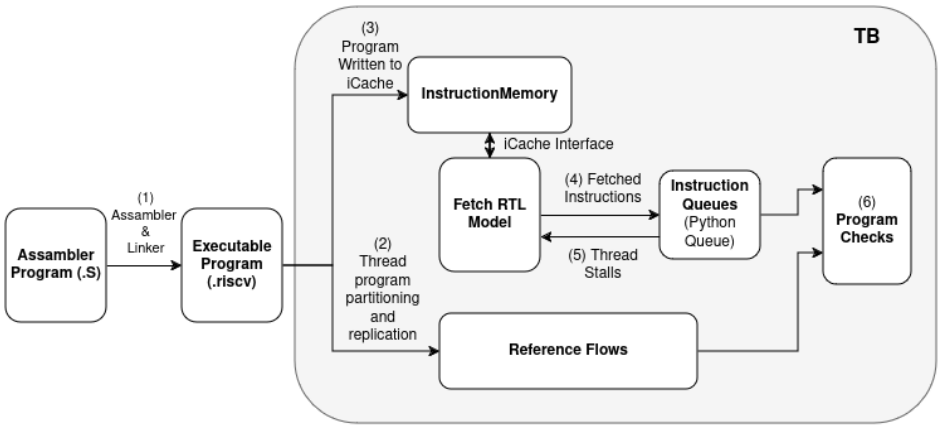


Figure 6.5: Fetch testbench diagram.

1. First, the assembler program (.S) is converted into an executable binary file (.riscv) using the assembler and linker programs from the RISC-V compiler tool-chain.
2. The executable binary is read and partitioned by loops for each thread. Each loop is replicated by N, the configured number of iterations, producing the expected set of instructions for each thread. This set of instructions serves as the reference flow to verify the fetched program.
3. The executable is loaded into an instruction cache model connected to the RTL simulation model. The RTL model requests instructions from this cache, loaded with the program, and begins fetching for each thread in a FGMT manner.
4. Instructions fetched from the RTL output ports are stored in queues, one per thread, in the Python testbench.
5. When a thread queue has accumulated the expected number of instructions per loop iteration, the thread is stalled.
6. Once all threads are stalled, the testbench checks each thread's queue against the reference flow. If the contents match, the fetch is considered correct.

Figures 6.6a and 6.6b of the testbench logs illustrate FGMT execution with a configuration of two threads and a bandwidth of four 16-bit blocks (up to two instructions per cycle), showing the effects of thread stalls (inactive thread).

Cycle	Thread 0	Thread 1
0	X	X
	X	X
1	X	X
	X	X
2	X	X
	X	X
3	X	add t1, a1, t0
	X	add t1, a1, t0
4	add t1, a1, t0	X
	add t1, a1, t0	X
5	X	add t1, a1, t0
	X	add t1, a1, t0
6	add t1, a1, t0	X
	add t1, a1, t0	X
7	X	add t2, a1, t0
	X	add t2, a1, t0
8	add t2, a1, t0	X
	add t2, a1, t0	X
9	X	add t2, a1, t0
	X	add t2, a1, t0
10	add t2, a1, t0	X
	add t2, a1, t0	X
11	X	add t1, a1, t0
	X	add t1, a1, t0
12	add t1, a1, t0	X
	add t1, a1, t0	X
13	X	add t1, a1, t0
	X	add t1, a1, t0

(a) FGMT Fetch execution with both threads active.

66	add t2, a1, t0 j	Inactive thread Inactive thread
67	Inactive thread Inactive thread	X X
68	Inactive thread Inactive thread	X X
69	Inactive thread Inactive thread	add t1, a1, t0 add t1, a1, t0
70	Inactive thread Inactive thread	add t1, a1, t0 add t1, a1, t0
71	Inactive thread Inactive thread	add t2, a1, t0 add t2, a1, t0
72	Inactive thread Inactive thread	add t2, a1, t0 add t2, a1, t0
73	Inactive thread Inactive thread	add t1, a1, t0 add t1, a1, t0
74	Inactive thread Inactive thread	add t1, a1, t0 add t1, a1, t0
75	Inactive thread Inactive thread	add t2, a1, t0 add t2, a1, t0
76	Inactive thread Inactive thread	add t2, a1, t0 j
77	Inactive thread Inactive thread	X X
78	Inactive thread Inactive thread	add t1, a1, t0 add t1, a1, t0
79	Inactive thread Inactive thread	add t1, a1, t0 add t1, a1, t0
80	Inactive thread Inactive thread	add t2, a1, t0 add t2, a1, t0

(b) FGMT Fetch execution with thread 0 inactive.

Figure 6.6: FGMT Fetch testbench execution logs in a 4-block bandwidth configuration (up to 2 regular instructions or 4 compressed).

6.6 Decoder Testbench

First, it is recommended to refer to Annex A for a detailed description of the decoder implementation, which will aid in understanding the verification methodology. However, this is not mandatory.

The decoder verification strategy focuses on ensuring that instructions are correctly identified, since the control signals for each instruction are platform-dependent and lack a direct reference for comparison. In our decoder implementation, each instruction is associated with a unique control constant structure. These structures are automatically generated using a script. One of the fields in this control constant, used for debugging purposes, is the "instruction" field. This field is present only in simulation and does not impact the hardware. It uses a unique enumeration value to identify the decoded instruction.

The following segments 6.2 and 6.3 of code illustrate these control constants for *BGE* and *ADDI*, these

instructions receive the field instruction with the enumeration values *BGE_DBG* (*DBG* stands for debug) and *ADDI_DBG*, respectively, so, in other words, if the decoder outputs the control constant of an instruction with the "instruction" field with the value of *BGE_DBG*, it has identified the instruction as *BGE*.

```
parameter instr_ctrl_t BGE_CTRL = '{
    `ifdef SIMULATION
    instruction: BGE_DBG,
    `endif
    valid: 'b1,
    use_src: 'b11,
    rf_src: {{RF_BITS{1'b0}}, INTEGER_RF, INTEGER_RF},
    queue_id: INTEGER_QUEUE,
    use_imm: 'b1,
    imm_ext_typ: IMM_19_0,
    use_pc: 'b1,
    func_unit: BRANCH_UNIT,
    instr_op: BGE_OP,
    default: '0
};
```

Listing 6.2: BGE example.

```
parameter instr_ctrl_t ADDI_CTRL = '{
    `ifdef SIMULATION
    instruction: ADDI_DBG,
    `endif
    valid: 'b1,
    use_src: 'b1,
    rf_src: {{RF_BITS{1'b0}}, {RF_BITS{1'b0}}, INTEGER_RF},
    use_dst: 'b1,
    rf_dst: INTEGER_RF,
    queue_id: INTEGER_QUEUE,
    use_imm: 'b1,
    imm_ext_typ: IMM_19_0,
    func_unit: ALU_UNIT,
    instr_op: ADD_OP,
    width: _DOUBLE_WORD_,
    default: '0
};
```

Listing 6.3: ADDI example.

We use this debug value to confirm the correct identification of instructions. By writing assembler code for a specific instruction and generating its machine code with an assembler, we can send it to the decoder and check if the output control signal's instruction field matches the corresponding enumeration value. This approach forms the basis of the decoder verification.

The testbenches consist of several assembler files, each corresponding to a different RISC-V extension. Since the decoder can be parameterized for different sets of extensions, only the supported extensions will be tested. Unsupported extensions would cause the testbench to throw an error due to unrecognized instructions. These assembler files contain all instructions for their respective extensions, written in assembly language (operand values are irrelevant as we are only verifying instruction identification). The code 6.4 illustrates the assembler test program for the RV32A extension, which deals with atomic operations.

```
.global _start
```

```
2
3  _start:
4
5     ############################
6     # RV32A Standard Extension #
7     ############################
8
9     lr.w        x0, (x0)
10    sc.w        x0, x0, (x0)
11    amoswap.w   x0, x0, (x0)
12    amoadd.w    x0, x0, (x0)
13    amoxor.w    x0, x0, (x0)
14    amoand.w    x0, x0, (x0)
15    amoor.w     x0, x0, (x0)
16    amomin.w    x0, x0, (x0)
17    amomax.w    x0, x0, (x0)
18    amominu.w   x0, x0, (x0)
19    amomaxu.w   x0, x0, (x0)
```

Listing 6.4: RV32IA.S Test assembler program.

These assembler files (.S extension) are converted into disassembled executable and linkable format (.ELF extension) using the objdump utility provided by the RISC-V compiler tool-chain. The .ELF file lists each instruction in the program along with three fields: address, binary encoding, and assembly instruction. For each instruction, we parse the binary encoding, send it to the decoder RTL model, and identify the instruction by its assembly name. The ELF program 6.5 is the result of applying this converstion to the assambler code 6.4.

```
1
2  .build/rv32IA.elf:     file format elf32-littleriscv
3
4
5  Disassembly of section .text:
6
7  00000000 <_start>:
8     0:   1000202f            lr.w    zero,(zero)
9     4:   1800202f            sc.w    zero,zero,(zero)
10    8:   0800202f            amoswap.w   zero,zero,(zero)
11    c:   0000202f            amoadd.w    zero,zero,(zero)
12   10:   2000202f            amoxor.w    zero,zero,(zero)
13   14:   6000202f            amoand.w    zero,zero,(zero)
14   18:   4000202f            amoor.w zero,zero,(zero)
15   1c:   8000202f            amomin.w    zero,zero,(zero)
16   20:   a000202f            amomax.w    zero,zero,(zero)
17   24:   c000202f            amominu.w   zero,zero,(zero)
18   28:   e000202f            amomaxu.w   zero,zero,(zero)
```

Listing 6.5: RV32IA.ELF Test ELF program example.

Therefore, the test proceeds as Figure 6.7 shows:

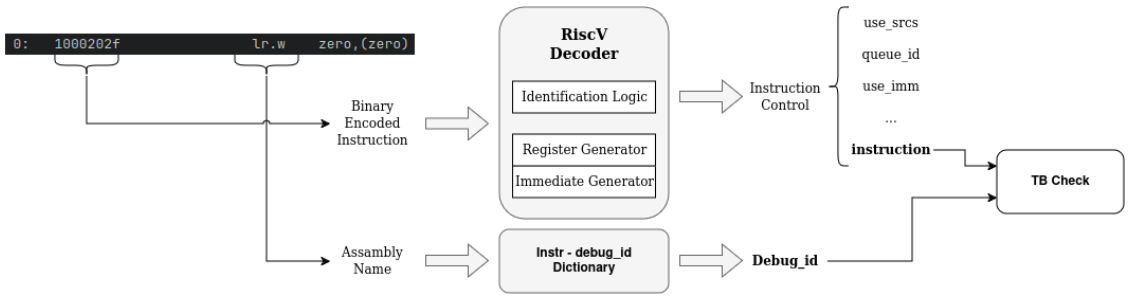


Figure 6.7: Decoder testbench flow.

1. For each supported extension, the test reads the .ELF file.
2. For each instruction in the .ELF, its binary encoding is sent to the RTL decoder, and the instruction is identified by its assembly name.
3. Using the assembly name, we reference a predefined dictionary that associates the instruction with its enumeration value, obtaining what we call the *debug_id* of the instruction. If the *debug_id* matches the "instruction" field value from the output control signals, the instruction is correctly identified.

6.6.1 Compressed Instruction Verification

There is a specific consideration for compressed instructions. Compressed instructions do not have their own constant control signals, as they are a compressed form of regular instructions. For instance, "*C.BEQZ rs1', offset*" is translated to "*BEQZ rs1', x0, offset*" with some restrictions (e.g., offset range and rs2 fixed to x0). Thus, the decoder identifies this as a *BEQZ* instruction. To reflect this behavior in the testbench, when a compressed instruction is identified by its assembly name, we use a different dictionary for compressed instructions, where the *debug_id* corresponds to the regular instruction version. After this translation, the rest of the testbench remains the same: the binary encoding is sent to the decoder, and the instruction control signal value is checked against the *debug_id*.

6.7 Renaming Unit Testbench

The renaming modelling is the most complex one in this work since it cycle-accurately replicates the behaviour of the renaming unit, and not only requires modelling the renaming scheme, it also needs modelling other elements, such as the pseudo-random generation of instructions and the Reorder Buffer structure. The verification strategy involves sending identical input stimuli to both the RTL implementation and the software renaming model each cycle, then comparing their outputs to detect any mismatches. The testbenching is structured as follows:

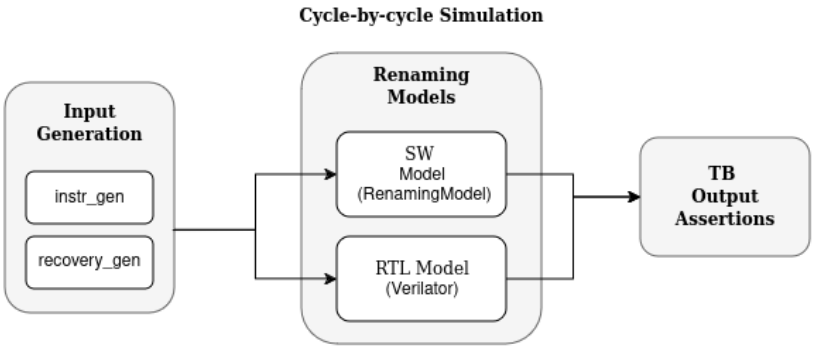


Figure 6.8: Renaming Unit testbench flow.

6.7.1 Input Generation

The first step is generating input stimuli to feed the models. There are two types of input events:

- **instr_gen():** This method generates pseudo-random logical instructions for renaming. These instructions vary in the number of sources, could or could not have destination, use different register files, be from different threads, etc.
- **recovery_gen():** This method generates recovery events for in-flight instructions, with varying frequencies and simultaneous recoveries across multiple threads.

6.7.2 Renaming Unit Modelling

The input generation feeds both the RTL model and the software model with the same input each cycle.

- **RTL Model:** This is the Verilator cycle-accurate simulation model generated from the RTL implementation.
- **SW Model:** This software model abstracts the behavior of the renaming unit, implementing the same functionalities as the RTL design but with the advantages of a high-level programming language. For example, the instruction renaming algorithm uses a for loop instead of checking register collisions when renaming multiple instructions in parallel. The state of a thread is saved for each instruction, making state recovery straightforward. The Figure 6.9 class diagram illustrates the classes used in this software modeling:
  - **RenameInstr:** This static class generates the input stimulus to the model: *instr_gen()* produces a list of logical instruction represented by objects of the class '**LogInstr**', *recovery_gen()* produces the recovery input with in-flight instructions for each thread.
  - **LogInstr:** Represents a logical instruction with attributes for thread identification, sources, and destination registers.
  - **PhyInstr:** Similar to '**LogInstr**', but represents a physical (renamed) instruction and includes the old destination register field.

- **RenamingState:** An auxiliary class used by '**RenamingModel**' to save the renaming state of a thread for each new instruction. It maintains the RATs and FRLs of the thread, associated with an instruction by its ROB ID.
- **ReorderBuffer:** Provides all functionalities of an RTL ROB. '**RenamingModel**' instances this class for each thread to simulate instruction allocation in the ROB (storing '**PhyInstr**' objects), instruction commit, tracking in-flight instructions for recoveries, and freeing space when instructions commit.
- **RenamingModel:** This class manages instruction renaming and state for different threads. It receives '**LogInstr**' objects and outputs '**PhyInstr**' objects each cycle, and it is also capable of recovering the state of multiple threads simultaneously, accurately replicating the behavior of the RTL Renaming Unit. It communicates through public attributes that function as module ports, processed with methods like *recover_state()* (checks the recovery attributes and restores the state if necessary) and *rename_instr()* (reads the '**LogInstr**' objects coming through the *log_instr_i* port attribute, renames them if possible, and writes the output attribute *phy_instr_o* with '**PhyInstr**' objects).

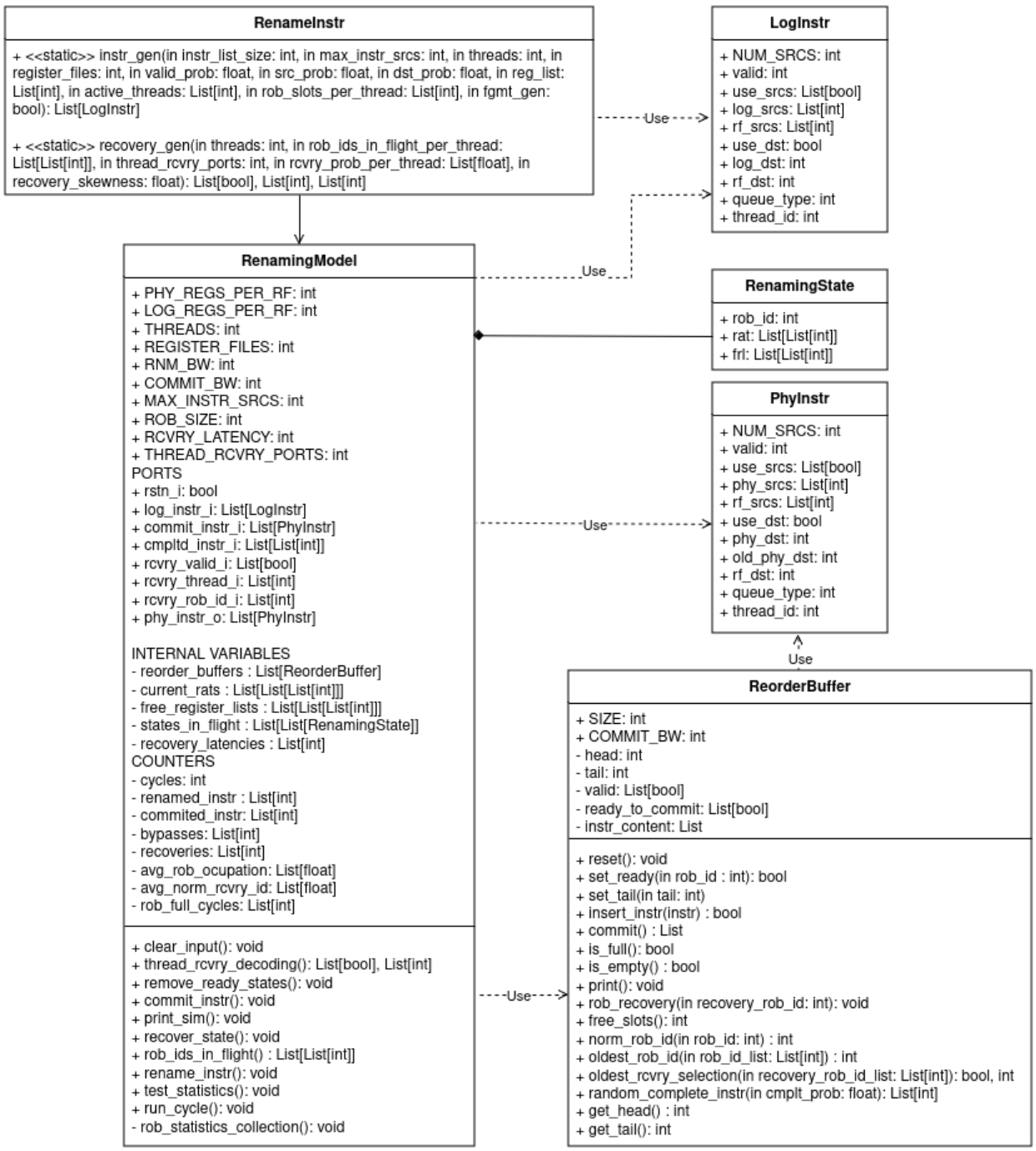


Figure 6.9: Python Renaming Unit model class diagram.

6.7.3 Output Validation

In the final step of the Renaming Unit verification, the outputs of each model are validated by comparing the models with each other. If any mismatch is detected, the test throws an exception, ending the test and logging the events causing the inconsistency and recording the signals of the RTL in VCD file. Additionally, we verified the module in three different configurations, to asses the correct flexibility of the verification framework and the parametric implementation, not only in the number of thread context, but also other elements of the architecture. Tables 6.1, 6.2 and 6.3, display the instance parameter for each configuration.

Single-thread configuration		
Parameter	Value	Description
PHY_REGS_PER_RF	128	Number of physical registers per register file
LOG_REGS_PER_RF	32	Number of logical registers per register file
THREADS	1	Number of threads
THREAD_RCVRY_PORTS	1	Configures the number of ports for selective thread recovery
REGISTER_FILES	2	Number of separated register files
FE_BW	4	Number of instructions to rename, the width of the front-end
COMMIT_BW	4	The maximum number of instructions committed per cycle
MAX_INSTR_SRCS	2	The maximum number of sources per instruction
ROB_SIZE	128	Size of the ROB

Table 6.1: Renaming Unit single-thread configuration parameters.

Dual-thread configuration		
Parameter	Value	Description
PHY_REGS_PER_RF	64	Number of physical registers per register file
LOG_REGS_PER_RF	32	Number of logical registers per register file
THREADS	2	Number of threads
THREAD_RCVRY_PORTS	1	Configures the number of ports for selective thread recovery
REGISTER_FILES	1	Number of separated register files
FE_BW	4	Number of instructions to rename, the width of the front-end
COMMIT_BW	4	The maximum number of instructions committed per cycle
MAX_INSTR_SRCS	2	The maximum number of sources per instruction
ROB_SIZE	64	Size of the ROB

Table 6.2: Renaming Unit dual-thread configuration parameters.

Finally, the verification testbench also provides software parameters that allows us to modify the input generation and profile different scenarios and workloads to produce a variety of situations in the architecture that may happen during a program execution. For instance, it is possible to increase the recovery frequency to stress the recovery mechanism, reduce the number of architectural registers used by the instructions to increase the number of name dependencies and force collisions, reduce the commit rate to stress the FRL resource management, etc. These parameters are randomized for each testbench execution with the possible values shown in Table 6.4.

Quad-thread configuration		
Parameter	Value	Description
PHY_REGS_PER_RF	64	Number of physical registers per register file
LOG_REGS_PER_RF	32	Number of logical registers per register file
THREADS	4	Number of threads
THREAD_RCVRY_PORTS	3	Configures the number of ports for selective thread recovery
REGISTER_FILES	1	Number of separated register files
FE_BW	6	Number of instructions to rename, the width of the front-end
COMMIT_BW	4	The maximum number of instructions committed per cycle
MAX_INSTR_SRCS	3	The maximum number of sources per instruction
ROB_SIZE	32	Size of the ROB

Table 6.3: Renaming Unit quad-thread configuration parameters.

Testbench software parameters		
Parameter	Value	Description
CYCLES	10000	Cycles of testbench simulation
DEBUG	0	Enables debug logs
VALID_RATIO	[0.85, 0.95]	Chance of each generated instruction of being valid
REG_AVAILABILITY_RATIO	[0.1, 0.3]	Percentage of the ARF used in the instruction generation
SRC_USAGE_RATIO	[0.7, 0.9]	Chance of a generated instruction of using each source register
DST_USAGE_RATIO	[0.5, 0.7]	Chance of a generated instruction of using destination
COMMIT_RATIO	[0.1, 0.4]	Chance of an in-flight instruction to be compleated
RECOVERY_RATIO	[0.00, 0.05]	Chance of a recovery per thread each cycle

Table 6.4: Renaming Unit testbench software parameters.

6.8 SMT Frontend Testbench

This section overviews the SMT Frontend testbench, which integrates all the modules presented in this work. The frontend fetches instructions, decodes them, renames them, allocates them in the ROB, and maps them to their corresponding issue queues. Instructions are fetched using a FGMT scheme, but once allocated in the multithreaded inter-stage queue, they are processed in a SMT manner for the rest of the pipeline.

The Frontend verification follows a similar approach to the fetch stage testbench. Instead of sending encoded instructions to output ports, it produces renamed instructions with their ROB IDs, ready for dynamic scheduling. In this test, we execute a simple program loaded into the **InstructionMemory** model, resembling a real-life scenario, displayed in the 6.6 code.

```
1 void vector_add(int *a, int *b, int *res, int const N) {
2     for (int i=0; i<N; i++){
3         res[i] = a[i] + b[i];
4     }
5 }
6
7 int vector_min(int *vec, int const N){
8     int min = vec[0];
```

```
9       for (int i=1; i<N; i++){
10          if (vec[i] < min) {
11              min = vec[i];
12          }
13      }
14      return min;
15  }
```

Listing 6.6: Vector operation functions used to verify the SMT Frontend.

The C program used for this demonstration contains two simple procedures. The *vector_add()* function accumulates the elements of two arrays of size N, while *vector_min()* finds the minimum value in an array of size N. This test runs with two threads, each executing a different function. Since the code contains conditional branches in the loops, and we lack branch predictors, we modify the assembly code by replacing these branches with unconditional jumps for simplicity.

The primary goal of this testbench is to execute these function loops using SMT technique (with potentially different loop iteration counts for each function). Since instruction fetching is done using FGMT, and we aim to test SMT capabilities in the rest of the pipeline, we will stall the subsequent stages for a few cycles. This action fills the multithreaded inter-stage queue between fetch and decode with instructions. When execution resumes, the instructions are processed using SMT.

Additionally, we will test the frontend's ability to handle recoveries and restart program execution from any point. This is done by randomly generating a recovery for any in-flight instruction after its renaming and ROB allocation. When the test observes that a thread has processed the expected number of instructions, it sends a stall signal to that thread, allowing the other thread to continue executing until it reaches the expected progress.

For the final demonstration of the project, we use the Tkinter Python package, which utilizes the Tk GUI window subsystem, to visualize the program's progress. The following Figure 6.10 captures an instant of the testbench execution frame shown in a Tk window, in this case, the frame corresponding to the thread 0.

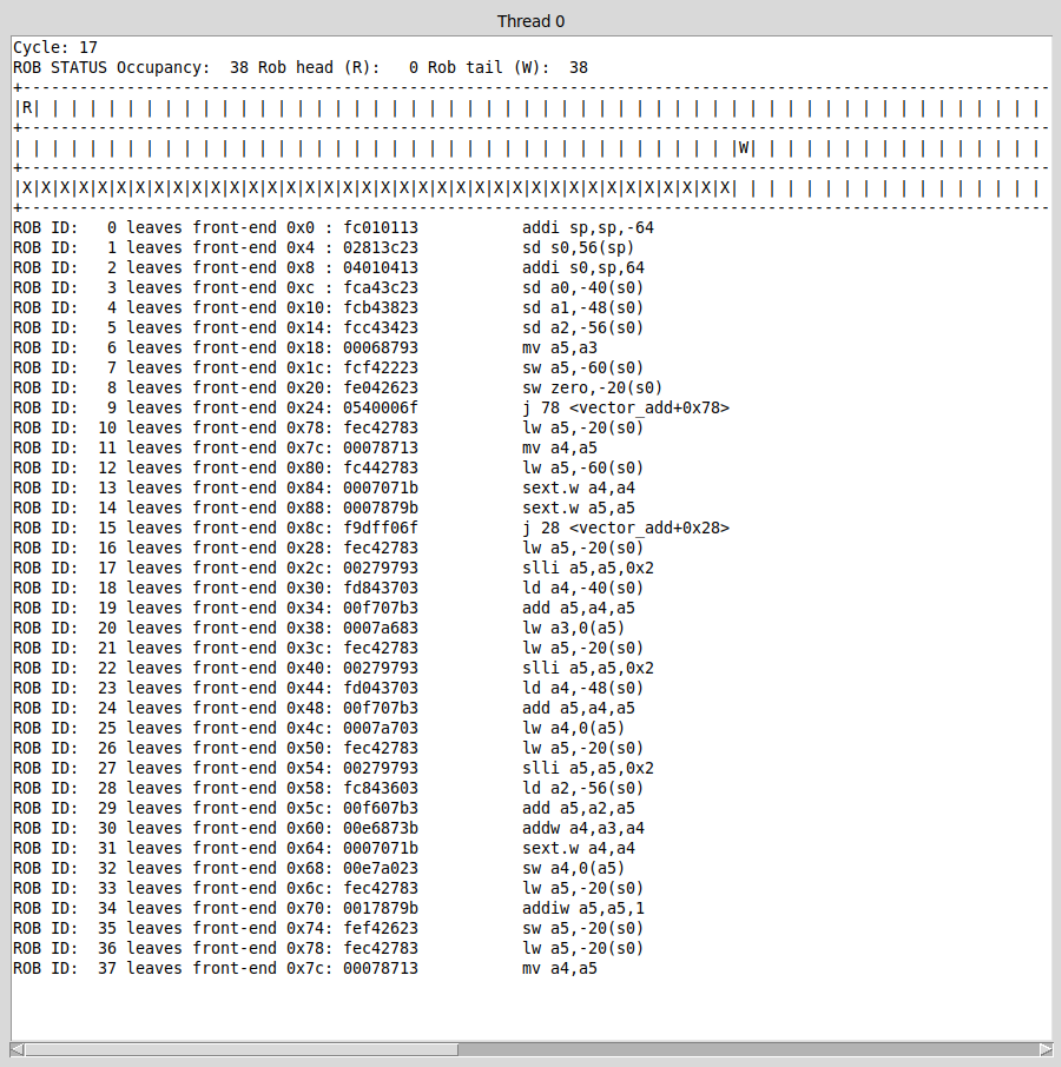


Figure 6.10: Frontend testbench execution frame of thread 0.

In this frame, which updates every cycle and shows the thread's progress, we can see displayed some information about the execution. Firstly, it shows the cycle count and the ROB status in real-time, including the head and tail ROB pointers. More importantly, the visualization displays the instructions coming out of the frontend, identified by their ROB IDs, PC addresses, binary encodings, and assembly representations. Any instruction exiting the frontend can trigger a recovery, restarting the execution from that point.

To verify the correctness of execution after threads stop fetching instructions, we can read the instruction stream and count the occurrences of the loop's jump instruction. If this count matches the number of loop iterations plus one (accounting for the initial appearance during initialization), we can confirm that the execution is accurate.

Chapter 7

Results

This final stage of our work presents the synthesis results and their analysis, focusing on some of the most relevant elements in this design as well as the complete frontend itself. In this chapter, we utilized the Genus Synthesis Solution, an RTL synthesis tool from Cadence, which is the sole commercial tool used in this project. Additionally, the results were simulated using a 7nm technology from TSMC. Traditionally, the "7nm" designation refers to the smallest length of a transistor's gate. However, in modern usage, it does not directly translate to the physical dimensions of the transistor and is more of a marketing term indicating the technology version.

The synthesis simulates the place and route of the physical implementation, and the results are measured in two terms:

- **Area:** This quantifies the number of cells and the network area required to implement the module, measured in square micrometers (μm^2).
- **Frequency:** The maximum feasible frequency of operation is determined by the critical path, which is the maximum time delay of combinational data paths between two registers. The critical path is measured in picoseconds (ps), and the frequency, which is the inverse of the period (the critical path in this case), is given in gigahertz (GHz).

The synthesis targets a frequency of 2 GHz, corresponding to a critical path of 500 ps. This is significant because the synthesis process optimizes the logic to achieve this frequency and then optimizes the area. If the design fails to meet this constraint, it attempts to reduce the critical paths by using larger cells, thus increasing the area until it achieves the target or determines it is physically unfeasible. For instance, if a design reaches a critical path of 600 ps with a target of 500 ps, the area results will be considerably larger than if it had an initial target adjusted to the achievable frequency.

Furthermore, since this is a configurable implementation, the analysis explores the scalability of the design in two orthogonal dimensions:

- **Superscalar Width:** This examines how well the implementation scales with the width of the pipeline in a single-thread configuration. It will be analyzed in three configurations: 2-width, 4-width, and 8-width.

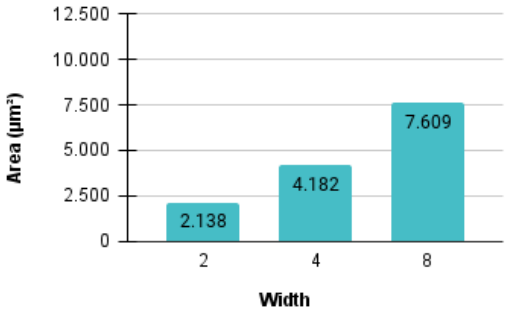
- **Multithreading Scalability:** This examines the scalability of increasing the number of threads, using a 4-width superscalar configuration as a base. It is analyzed in a single-thread, dual-thread and quad-thread configurations.

This comprehensive analysis will provide insights into both the efficiency and the scalability of the design under various configurations.

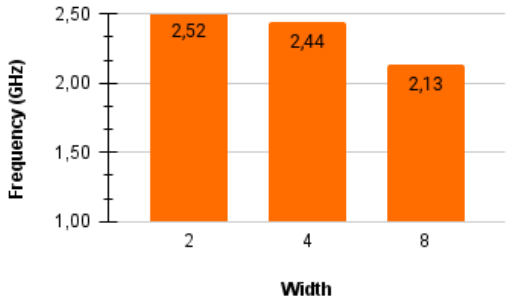
7.1 Fetch Synthesis

In this analysis, the bandwidth of the fetch stage is presented as a 32-bit instruction bandwidth, formed by two 16-bit blocks. In other words, this means a 4-width configuration effectively provides eight 16-bit blocks or compressed instructions.

The fetch stage demonstrates good scalability in terms of superscalar width, achieving over 2 GHz with a single-thread 8-width configuration. However, some modules, such as the Post-Fetch correction, experience performance impacts due to sequential logic processing tied to bandwidth. The area scales linearly with the bandwidth and their main contributors are the instruction buffer, the muxing and control, and inter-stage registers.



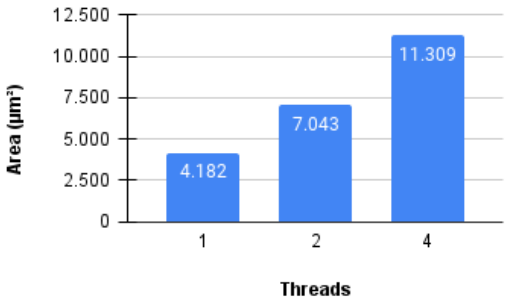
(a) Area results for superscalar configurations.



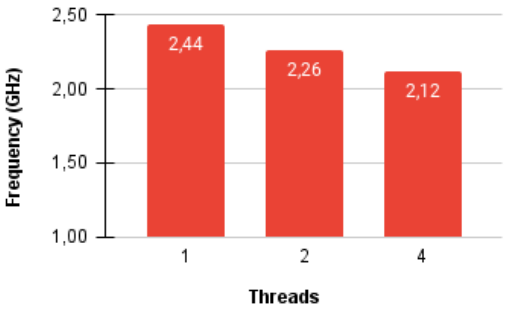
(b) Frequency results for superscalar configurations.

Figure 7.1: Scalability graphs in different superscalar and single-thread configurations.

The design scales similarly when increasing the number of supported threads, due to the replication of the Instruction Buffer module for each thread, as confirmed by the following graphs.



(a) Area results for multithread configurations.



(b) Frequency results for multithread configurations.

Figure 7.2: Scalability graphs in different multithread and 4-width superscalar configurations.

As this detailed area diagram indicates, the majority of the area is consumed by the Instruction Buffer, which is replicated for each thread. The next largest area category is "Muxing, Control Signals and Net Area", since the synthesis tool does not provide a more breakdown of RTL elements which are not a sub-module . The Post-Fetch Correction module, which accounts for 10.4% of the area, plays a significant role in enabling instruction flow changes upon detecting the first *JAL* instruction. The remaining modules collectively consume less than 20% of the total area.

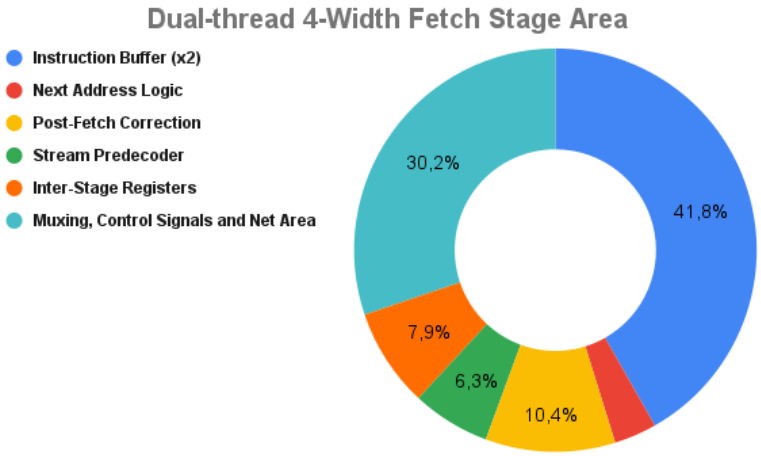
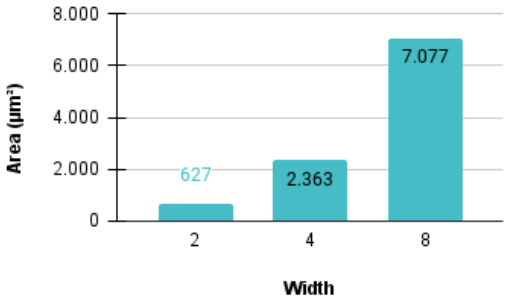


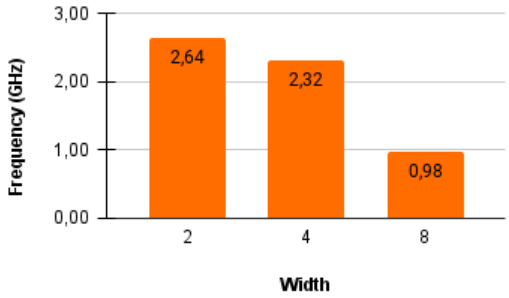
Figure 7.3: Area breakdown graph of Fetch Stage.

7.2 Decode Synthesis

The superscalar growth of the Decode Stage reveals scalability issues as the area significantly increases and the frequency decreases respectively with an 8-width configuration. This is an effect of the sequential logic in the First N Instruction Selector, required for the compressed instruction support. It re-aligns blocks of 16-bit blocks to form complete instructions. This combinational logic represents 60.1% (621ps) of the total Decode Stage critical path (1023 ps) in its 8-width configuration.



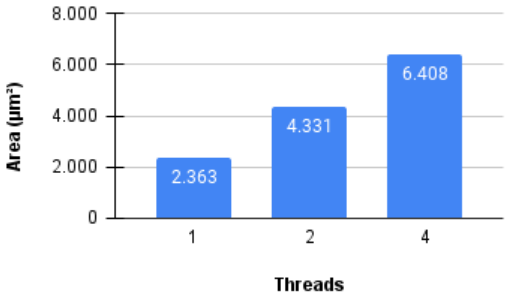
(a) Area results for superscalar configurations.



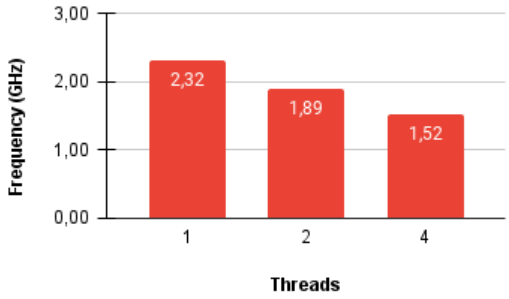
(b) Frequency results for superscalar configurations.

Figure 7.4: Scalability graphs in different superscalar and single-thread configurations.

The thread scalability demonstrates a more reasonable area growth as the number of threads increases, where most of the overhead comes from the replication the signals muxing, control signals, and network area, and then the replication of First N Instr Selectors for each thread, that guarantees an optimal selection. However, there is still a frequency penalty, due to the SMT Dynamic Instruction Selector, which sequentializes the selection of instructions among all threads after the First N Instruction Selector.



(a) Area results for multithread configurations.



(b) Frequency results for multithread configurations.

Figure 7.5: Scalability graphs in different multithread and 4-width superscalar configurations.

In the detailed area diagram, we can observe that the decoders represent a significant portion of the area due to being replicated for each instruction decoded in parallel. However, the most notable part is the "Muxing, Control Signals and Net Area" section, which comprises 57% of the area. Therefore, much of the decoder area growth with an increasing number of threads and bandwidth is attributed to the control mechanisms for instruction selection, block reading, and signal multiplexing.

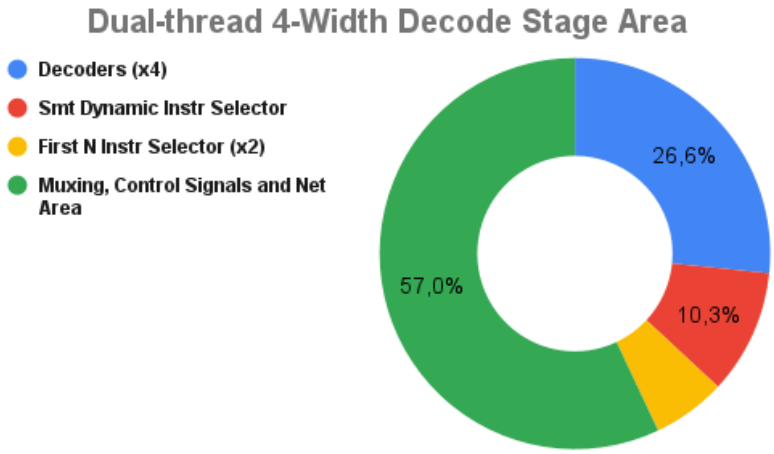


Figure 7.6: Area breakdown graph of Decode Stage.

7.3 Rename Synthesis

Since most of the logic in the Rename Stage is concentrated within the Renaming Unit module, as indicated by the Figure 7.7, we will focus our analysis on this module rather than the entire stage.

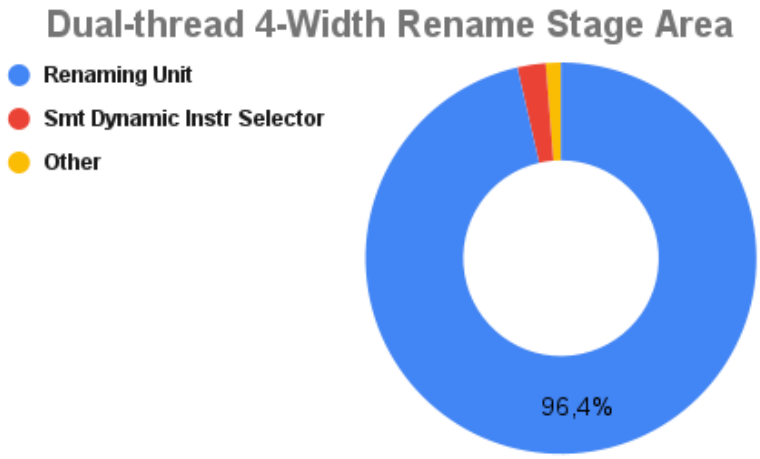
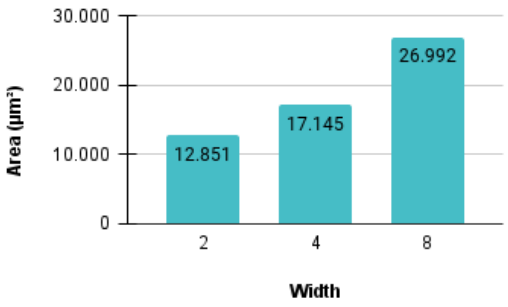
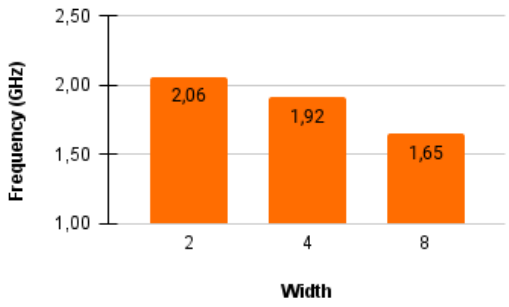


Figure 7.7: Area breakdown graph of Rename Stage.

The bandwidth of rename stage impacts in the renaming unit by the number of read and write ports of the Register Alias Tables (RATs) and Free Register Lists (FRLs), as well as the number of comparisons in the Logical Collision Detector (LCDs), among others. This is reflected in both the area footprint and the achievable frequency. However, it scales reasonably well, exhibiting a growth of local linear complexity.



(a) Area results for superscalar configurations.

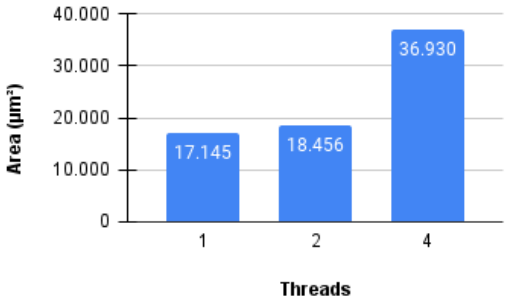


(b) Frequency results for superscalar configurations.

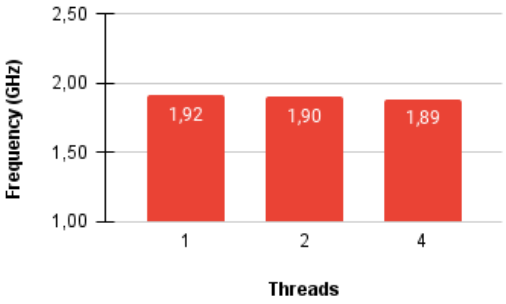
Figure 7.8: Scalability graphs in different superscalar and single-thread configurations.

The multithreading scalability illustrates expected results. On one hand, since the multithreading support is orthogonal and involves replicating parallel structures, the frequency remains effectively the same. On the other hand, the area footprint is very similar for single-thread and dual-thread configurations but increases significantly in a quad-thread configuration. The reasoning behind this increase is dependent on a hidden parameter: the total number of Reorder Buffer (ROB) entries. In both single-thread and dual-thread configurations, the number of entries is constant at 128 (2*64 in the dual-thread configuration) because the

ROB is statically partitioned. However, for the quad-thread configuration, we increased the total number of entries to 256 (4*64). This affects the storage capacity of the Context Recovery Unit and accounts for the observed area increase.



(a) Area results for multithread configurations.



(b) Frequency results for multithread configurations.

Figure 7.9: Scalability graphs in different multithread and 4-width superscalar configurations.

The detailed area diagram confirms this analysis, with the majority of the area (53%) corresponding to the Context Recovery Unit (CRU) modules. With 128 total ROB entries, this proportion is significant, and we can assume that with 256 entries, this percentage is even higher. Additionally, the Free Register Lists (FRLs) and "RATs, Muxing, Control Signals and Net Area" sections represent the rest of the area footprint.

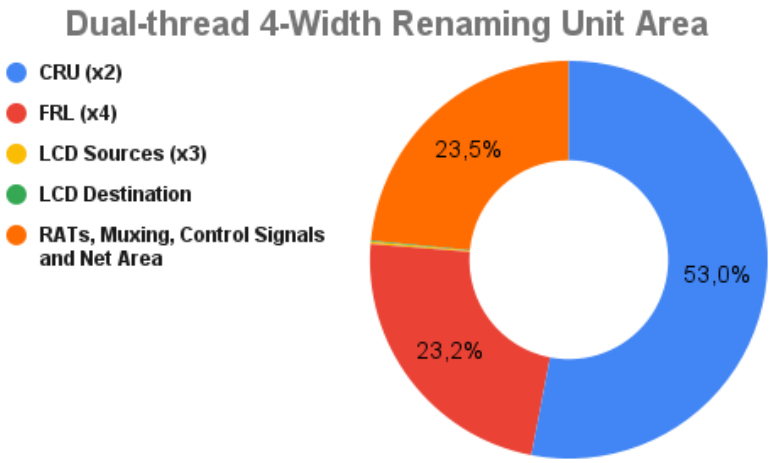
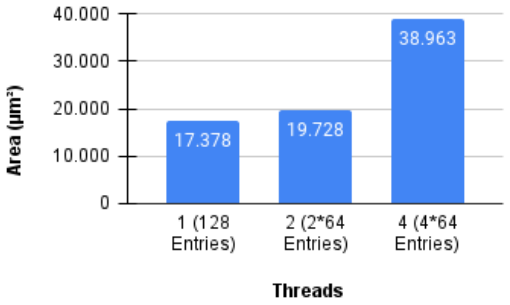


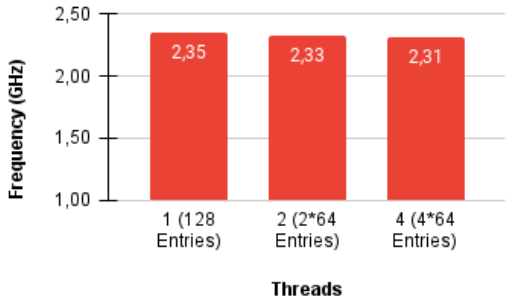
Figure 7.10: Area breakdown graph of Renaming Unit.

7.4 Reorder Buffer Synthesis

The ROB has been exclusively analyzed from the perspective of thread scalability, as the results across different bandwidths were found to be very similar. Here, we observe that multithreading support has minimal impact on the frequency of operation, as this module statically partitions the ROB among threads. The area overhead of multithreading is modest when maintaining the same number of entries, as evidenced by both the single-thread and dual-thread configurations. However, the key determining factor in the area footprint is the total number of entries in the ROB, as demonstrated by the quad-thread configuration with a total of 256 entries.



(a) Area results for multithread configurations.



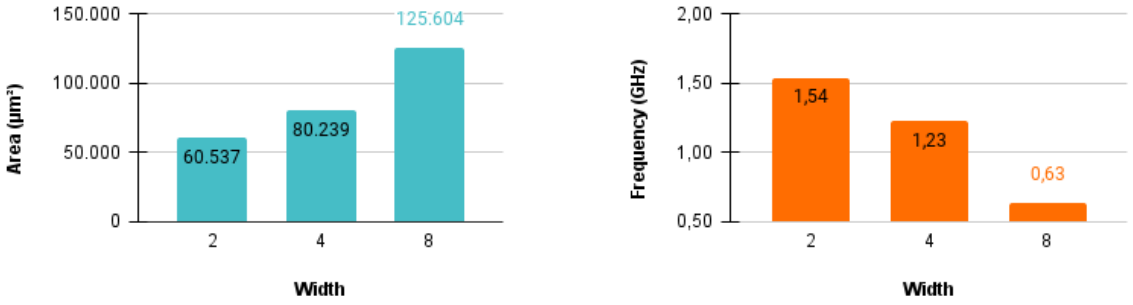
(b) Frequency results for multithread configurations.

Figure 7.11: Scalability graphs in different multithread and 4-width superscalar configurations.

7.5 Front-End Engine Synthesis

Next, we present the final results for the complete design, focusing on the front-end engine. Similar to previous analyses, we began with bandwidth scalability. We observ an important area increase in the single-thread configurations. The 4-width area growth compared to the 2-width configuration is reasonable. However, there is a significant area increase in the in the 8-width configuration motivated by the duplication of many read and write ports in all structures, pointer controls, muxes, etc.

The critical path primarily resides in the decode stage as we analyzed previously, attributed to sequential logic in the First N Instruction Selector, required for compressed instruction support, which accounts for 636 ps of the critical path (40.1%). Additionally the read pointers control of the Fetch-Deco Inter-Stage Queue module contributes with 441 ps (27.8%) of the total critical path, 1586 ps, for the 8-width configuration.

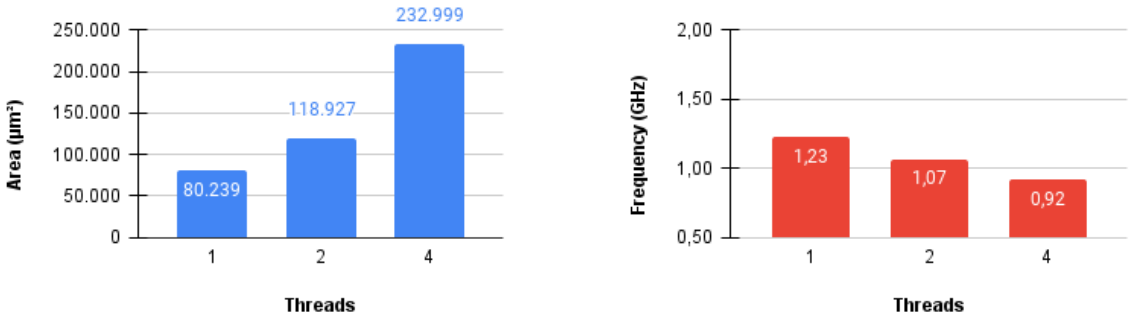


(a) Area results for superscalar configurations.

(b) Frequency results for superscalar configurations.

Figure 7.12: Scalability graphs in different superscalar and single-thread configurations.

In thread scalability analysis, while the same critical path remains, its impact is mitigated due to the utilization of a smaller 4-width superscalar configuration. Additionally, it introduces the sequential logic of the SMT Dynamic Instruction Selector, which represents 270 ps (24.9%) to the critical path for the quad-thread configuration. From an area perspective, the increase is rationalized by the replication of certain structures like Instruction Buffers, Inter-Stage Queue entries, and ROB entries increase (in the case of Quad-thread configuration it doubles from 128 to 256 total entries), alongside network area, among others.



(a) Area results for multithread configurations.

(b) Frequency results for multithread configurations.

Figure 7.13: Scalability graphs in different multithread and 4-width superscalar configurations.

The detailed area diagram provides insight into the area distribution of the Front-End Engine. Predominantly, the area is allocated to instruction allocation components (ROB + Fetch-Deco Inter-Stage + Deco-Rename Inter-Stage), comprising 52.9% of the total area. Since the ROB and the Inter-Stage Queues are statically partitioned, the diagram illustrates the total number of entries. In the case of Fetch-Deco Inter-Stage Queue, the number of entries represents blocks of 16-bit meaning it can store 32 regular instructions at most, or 64 compress instructions.

The second most area-consuming element, the Rename Stage, consumes a significant portion of the area, which is reasonable considering that more than half of it is dedicated to the Context Recovery Unit and its associated information.

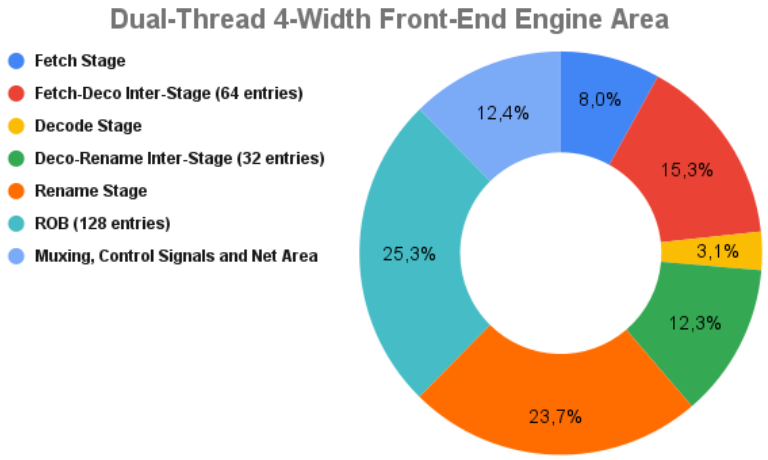


Figure 7.14: Area breakdown graph of the frontend.

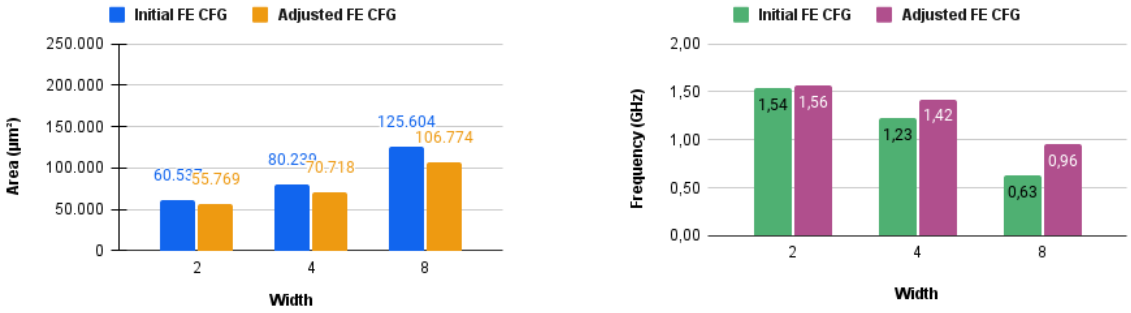
7.5.1 Refined Front-End Engine Configuration Synthesis

After analyzing the area footprint and critical path of the Front-End Engine, we concluded that certain elements were oversized and significantly contributed to the low frequency achieved. These elements were the FGMT Fetch Stage and the Fetch-Deco Inter-Stage Queue.

First, The FGMT Fetch Stage was capable of delivering up to double the superscalar bandwidth of compressed instructions per cycle. For instance, in the 4-width configuration, the fetch stage could deliver up to 8 compressed instructions per cycle. We decided to reduce this parameter by half since we cannot process such a high bandwidth, aligning the fetch stage's bandwidth with the rest of the pipeline. Supporting compressed instructions had a substantial impact on the design and its physical characteristics. Thus, it is logical to match the fetch bandwidth to the superscalar width of the core, allowing us to maintain the same superscalar capability (up to width-instructions per cycle) while reducing the implementation footprint.

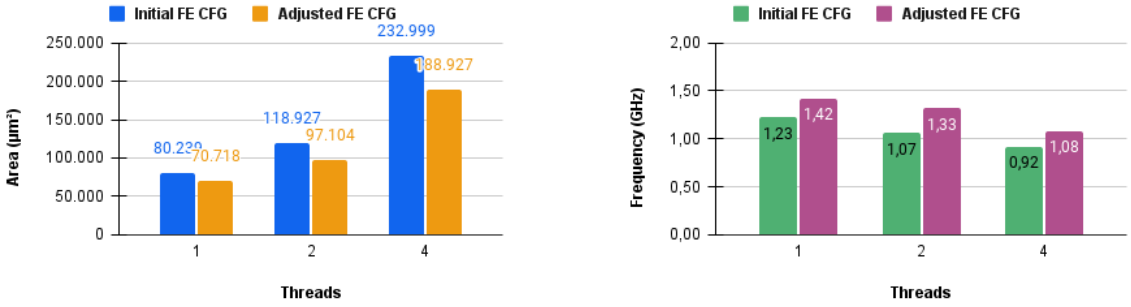
The second oversized element is the Fetch-Deco Inter-Stage Queue, which had 64 entries. This queue significantly impacted the area and, more critically, the delay of the critical path. Beyond the sheer number of entries, the read pointers tracking directly affected the critical path and, with the reduction in fetch bandwidth, it also makes sense to reduce the capacity of the inter-stage queue. We opted to decrease it to 32 total entries, providing storage for up to 32 compressed instructions, statically partitioned among threads.

This adjustment aligns with the reduced fetch bandwidth and helps mitigate the critical path delay and area footprint issues. For comparison purposes, we will refer to the previous oversized comparison as "Initial FE CFG" which stands for initial frontend configuration and to the adjusted configuration as "Adjusted FE CFG".



(a) Area comparison for superscalar configurations. **(b)** Frequency comparison for superscalar configurations.

Figure 7.15: Width-scalability comparison graphs between frontend parametrizations in single-thread configuration.



(a) Area comparison for multithread configurations. **(b)** Frequency comparison for multithread configurations.

Figure 7.16: Thread-scalability comparison graphs between frontend parametrizations in 4-width superscalar configuration.

The area and frequency result graphs illustrates a significant improvement in both metrics due to adjustments in the configuration of the Front-End Engine. However, there are still scalability issues because the root cause—sequentialization of the logic in the design and bad scalability in some modules—has not been addressed and should be improved in future work. Finally, we provide the detailed area distribution graph 7.17 with the new configuration, showing that the fetch stage and the inter-stage queue have reduced their area occupation.

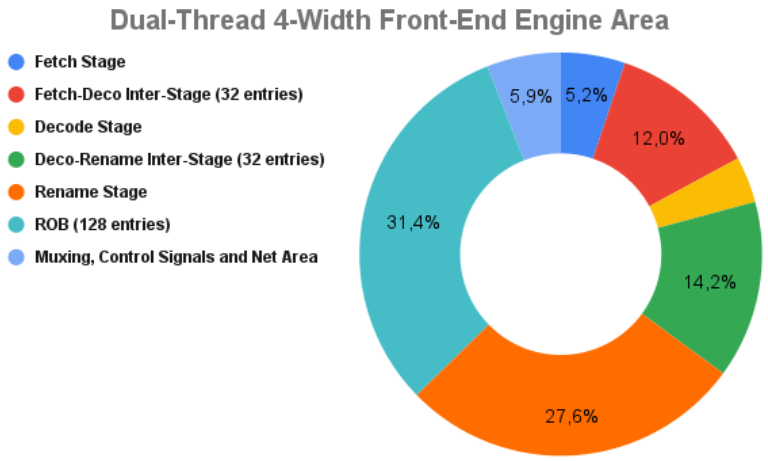


Figure 7.17: Area breakdown graph of the frontend with refined parametrization.

7.6 Preliminary Core Synthesis

Finally, we will provide preliminary area results of what will conform Lagarto Ox core at the BSC to give some context to the numbers obtained in our implementation. Lagarto Ox, and therefore the frontend, backend and other modules target a single-thread 4-width superscalar configuration. The core will be composed by the frontend we presented, the out-of-order execution backend, the memory caches, a composite branch predictor, the control status registers (CSRs), and the memory management unit (MMU). In this analysis, we provide preliminary synthesis results of these modules separately, since the integration needs to be completed, meaning it does not considered some additional control logic, muxes and other synchronization elements, but allows us to get a general approximation of the final area occupation.

As discussed in previous chapters, the idea of multithreading is to reuse some of the most costly elements of the core to improve their efficiency with minimal hardware overhead. Some elements, such as the memory cache and the backend, are mostly thread-agnostic, requiring minimal changes to support multithreading, mainly to maintain isolation between threads. Conversely, elements such as branch predictors or the MMU would require further research, but conventional approaches such as static partitioning can also be applied with low hardware overhead, however, the CSRs would need to be replicated for each thread.

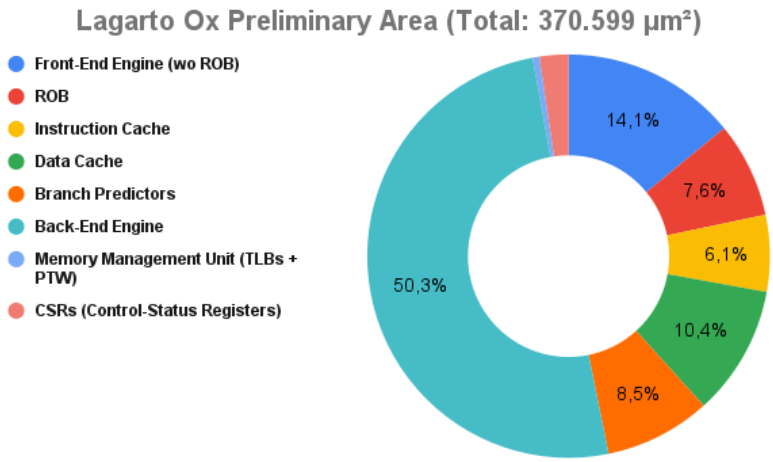


Figure 7.18: Preliminary area breakdown graph of Lagarto Ox core (4-width and single-thread).

The results of Figure 7.18 show that the elements that will require minimal changes represent more than half of the total area. This includes the backend engine, which encompasses the register files and accounts for 50.3% of the total area, and the memory caches, which combined represent 18%. Meanwhile, the frontend engine, excluding the reorder buffer (ROB), constitutes a smaller portion of the core at 14.1%.

This is significant, since the multithreading scheme adds most of the overhead area in the frontend elements of the core. If we use the same target configuration for the core but changing the frontend to a dual-thread parametrization (inlcuding the ROB) to have a multithreading core estimation, we would observe an area increase of 21% (from 80,239µm^2 to 97,104µm^2) of the frontend. This area overhead would represent less than 5% increase of the total area of the core, justifying the inclusion of multithreading with its expected throughput benefits. However, it is worth mentioning that we are not accounting for the overhead of integrating the modules, the complete multithread support and its control in the rest of the core, but it is

reasonable to expect and overhead considerably less than a 21% total area increase.

In conclusion, the inclusion of multithreading introduces a moderate area overhead in the frontend, while the overall core area would increase slightly, demonstrating the efficiency and feasibility of adding multithreading capabilities to the Lagarto Ox core.

Chapter 8

Conclusions

This projects covers all phases of development of a simultaneous-multithreading superscalar out-of-order front-end engine architecture for RISC-V ISA, from initial exploration of current open implementations and conception of our design, to modular verification and synthesis analysys.

This thesis contributes to many orthogonal fronts, to the Lagarto Ox and HLib projects at the BSC, to the popularization of many open-source tools and projects such as RISC-V, Verilator and CoCoTB, and perhaps more importantly, to expand and cement our knowledge in many areas of computer architecture such as supersclar, out-of-order and multithreading techniques, implementation skills in SystemVerilog, Python modelling and verification methodologies, and physical design analysis skills. Furthermore, we hope this project will eventually become part of a publicly available superscalar core, similar to BOOM, RISCYOO, Klessydra, and other contributions to the open-source RISC-V hardware ecosystem.

The following list of achievements summarizes the achievements made:

- We have obtained a better awareness of the current open RISC-V implementations in OoO and multithreaded processors.
- We have gained an in-depth understanding of classical multithreading techniques while cementing our knowledge of OoO superscalar processors and their associated challenges.
- We have designed a flexible RISC-V simultaneous-multithreading superscalar out-of-order front-end engine tailored for server workloads.
- An implementation in System Verilog using a modular and parametric methodology.
- Our work has integrated with and expanded the HLib framework.
- We have developed numerous testing models in the CoCoTB environment for each of our implementations to verify behavioral correctness.
- The implementation has followed HLib's coding and documentation quality standards.
- We have ensured that our design is synthesizable.

- We measured and discussed the synthesis results of our design across various configurations using the Genus Synthesis Solution tool in TSMC 7nm technology.
- We did an analysis of the potential area impact that multithreading could have on Lagarto Ox.
- We have a clear objective list of potential improvements for the continuation of this work.

We consider this achievements reflect the state of this work and are well aligned with the objectives established when we started project, and we think the knowledge and experience obtained are enough justifications for the time invested in this thesis. Moreover, the synthesis results demonstrate the cost-effectiveness of multithreading, with a reduced impact in area and frequency in relation to the total area of a processor, with the benefits of a more efficient usage of expensive elements of a core micro-architecture and its potential throughput increase.

8.1 Future Work

Despite the significant achievements in this work, there are several avenues for future exploration. In this section, we discuss some of these potential directions:

- **Core Integration:** A logical next step is connecting the front-end engine to a back-end execution engine. From a multithread perspective, few changes would be needed for a superscalar out-of-order (OoO) pipeline, as instructions are executed independently once they are ready, thanks to dynamic scheduling. However, bypass networks and memory disambiguation will need to be thread-aware to ensure isolated thread execution.
- **Branch predictors:** One of the main performance bottlenecks is the pipeline's induced bubbles due to the absence of branch predictors. Investigating branch prediction techniques that support multithreading should be a primary focus of future research.
- **Memory Management Unit (MMU):** The MMU, responsible for memory translation and protection, must support multithreading. Each thread should have its own access rights, memory mappings, and translation-lookaside buffers (TLBs). Enhancing the MMU to handle these requirements is essential for efficient multithreading support.
- **Physical Optimizations:** Various optimizations are needed to reduce critical path timing and area footprint, as observed in the synthesis analysis chapter. These optimizations involve refining the architecture, improving data paths, and enhancing component interactions to achieve better performance and efficiency.
- **Instruction Cache Improvements:** Supporting a thread-aware instruction cache is crucial for the front-end engine's integration. Additionally, single-port blocking instruction cache presents several limitations that directly affect the fetch stage design, not allowing the implementation of simultaneous multithreading (SMT) techniques. In the future, we should explore incorporating a multi-ported non-blocking instruction cache and the necessary changes to the fetch stage to take advantage of these more aggressive memory configuration. This enhancement would likely improve overall performance and efficiency by enabling overlapping cache misses.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, Sep. 2006.

[2] R. Schaller, "Moore's law: past, present and future," *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, June 1997.

[3] J. Shen and M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, ser. Electrical and Computer Engineering. McGraw-Hill Companies,Incorporated, 2005. [Online]. Available: https://books.google.es/books?id=Nibfj2aXwLYC

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[5] M. Nemirovsky and D. M. Tullsen, "Multithreading architecture," in *Multithreading Architecture*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:13560223

[6] E. Commission, C. Directorate-General for Communications Networks, Technology, K. Blind, S. Pätsch, S. Muto, M. Böhm, T. Schubert, P. Grzegorzewska, and A. Katz, *The impact of open source software and hardware on technological independence, competitiveness and innovation in the EU economy – Final study report*. Publications Office, 2021.

[7] "Key enabling technologies for europe's technological sovereignty." [Online]. Available: https://www.europarl.europa.eu/thinktank/en/document/EPRS_STU(2021)697184

[8] "Risc-v: The open standard risc instruction set architecture." [Online]. Available: https://riscv.org/

[9] "Verilator: The fastest verilog/systemverilog simulator." [Online]. Available: https://www.veripool.org/verilator/

[10] "Cocotb: Phyton verification framework." [Online]. Available: https://www.cocotb.org/

[11] O. U. J. Mendoza, A. Cristal, "Hlib: A system verilog hardware library conventions and methodology proposal." [Online]. Available: https://docs.google.com/document/d/1E_Ll50IYNwFwNSj4WJpL7S6m3a0_DmXAUPanCIv-WuM/edit?usp=sharing

[12] A. Waterman, "Design of the risc-v instruction set architecture," 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:63861396

[13] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Heterogeneous von neumann/dataflow microprocessors," *Communications of the ACM*, vol. 62, pp. 83–91, 5 2019.

[14] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, 1967, pp. 483–485. [Online]. Available: https://doi.org/10.1145/1465482.1465560

[15] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, July 2008.

[16] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 392–403.

[17] J. Zhao and A. Gonzalez, "Sonic boom: The 3rd generation berkeley out-of-order machine," 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:221212196

[18] "Boom: The berkeley out-of-order risc-v processor · github." [Online]. Available: https://github.com/riscv-boom

[19] K. Yeager, "The mips r10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–41, April 1996.

[20] "Risc-v rocket core repository." [Online]. Available: https://github.com/chipsalliance/rocket-chip

[21] S. Zhang, A. Wright, T. Bourgeat, and A. Arvind, "Composable building blocks to open up processor design," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 68–81.

[22] "Riscyoo: Risc-v out-of-order processors." [Online]. Available: https://github.com/csail-csg/riscy-OOO

[23] "Github - bluespec/toooba: Risc-v core; superscalar, out-of-order, multi-core capable; based on riscy-ooo from mit." [Online]. Available: https://github.com/bluespec/Toooba

[24] R. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[25] A. Cheikh, G. Cerutti, A. Mastrandrea, F. Menichelli, and M. Olivieri, "The microarchitecture of a multi-threaded risc-v compliant processing core family for iot end-nodes," 12 2017.

[26] "Klessydra: A family of processing cores and accelerators developed at the digital systems lab at sapienza university of rome, italy." [Online]. Available: https://github.com/klessydra

[27] D. Sima, "The design space of register renaming techniques," *IEEE Micro*, vol. 20, no. 5, pp. 70–83, Sep. 2000.

[28] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, Jan 1967.

[29] "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, Feb 2018.

[30] "Gtkwave: Fully featured gtk+ based wave viewer for unix, win32, and mac osx which reads lxt, lxt2, vzt, fst, and ghw files as well as standard verilog vcd/evcd files and allows their viewing." [Online]. Available: https://gtkwave.sourceforge.net/

Appendix A

RISC-V Decoder Implementation

The implementation of the decoder module in this thesis is unique because it is not entirely based on the SystemVerilog HDL language like the other modules. There are several reasons for this approach.

First, decoder implementations are generally difficult to maintain due to the numerous control signals they generate, which vary from one instruction to another. The decoder translates instructions into internal representations dependent on the micro-architecture and their encoded data (such as source and destination registers, immediate values, rounding modes, etc.). Typically, a control change in the micro-architecture implies many modifications in the decoder code. To address this, we will define constants for each instruction that specify its control signals. Rather than manually defining each constant, we will automatically generate them from a human-readable set of control tables in CSV format, with one table per RISC-V extension. The only exception to this rule are compressed instructions, which don't need their control in a table since the are compressed versions of regular instruction and receive their control instead.

Table A.1 represents a segment of the control signals for the "A" extension, which handles atomic operations for the 32-bit base, RV32I. This table includes only the control signals relevant to this extension; for example, control signals related to the use and type of the immediate are ignored and assume a default value of 0. The same applies to control signals with empty fields, such as the *unsigned_op* field in the example, which identifies if an operation is unsigned or not.

instruction	use_src	rf_src	use_dst	rf_dst	instr_op	unsigned_op
LR_W	1	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	AMO_LR_OP	
SC_W	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	AMO_SC_OP	
AMOSWAP_W	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	AMO_SWAP_OP	
AMOADD_W	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	AMO_ADD_OP	
AMOXOR_W	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	AMO_XOR_OP	
AMOAND_W	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	AMO_AND_OP	
AMOOR_W	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	AMO_OR_OP	
AMOMIN_W	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	AMO_MIN_OP	
AMOMAX_W	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	AMO_MAX_OP	
AMOMINU_W	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	AMO_MIN_OP	1
AMOMAXU_W	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	AMO_MAX_OP	1

Table A.1: Control signals table fragment of the RV32IA extension (not all control signals are included due to limited page width).

These tables are processed by a Python script that generates SystemVerilog control constants assigned to the identified instructions. These constants are stored in a SystemVerilog *instructions_pkg.sv* file. For instance, the first row corresponding to the LR.W instruction is converted to the following constant:

```
parameter instr_ctrl_t LR_W_CTRL = '{
    `ifdef SIMULATION
    instruction: LR_W_DBG,
    `endif
    valid: 'b1,
    use_src: 'b1,
    rf_src: {{RF_BITS{1'b0}}, INTEGER_RF, INTEGER_RF},
    use_dst: 'b1,
    rf_dst: INTEGER_RF,
    queue_id: MEMORY_QUEUE,
    func_unit: MEM_UNIT,
    instr_op: AMO_LR_OP,
    width: _WORD_,
    default: '0
};
```

Listing A.1: LR.W instruction constant control signals example.

Another reason for this approach is that SystemVerilog does not offer the best support for enabling and disabling extensions at will. In other implementations, we used parameters to configure the modules, but they cannot be used in certain structures within the decoder implementation. Another option was to use macros, but they often complicate scope tracking in large projects, such as a complete processor, and unnecessarily clutter the code.

To address these challenges, we will implement the decoder in a skeleton file, *riscv_decoder.skel*. This file contains the decoder RTL implementation in SystemVerilog, along with a series of tags. These tags, such as "$//--- < extension\_abbreviation > \_(on|off)$", mark the boundaries of code segments for specific extensions (e.g., "*//—C_on*" for compressed extension support). The code between the "*on*" and "*off*" tags is included or excluded based on whether the extension is supported.

In the following example, the code between "*//—M_on*" and "*//—M_off*" in the *riscv_decoder.skel* file is removed if the M extension is not supported.

```
OP_ALU: begin
    case ({instr_content_i.rtype.func3, instr_content_i.rtype.func7})
        {F3_0, F7_0}:   instr_ctrl = ADD_CTRL;
        {F3_0, F7_32}:  instr_ctrl = SUB_CTRL;
        {F3_1, F7_0}:   instr_ctrl = SLL_CTRL;
        {F3_2, F7_0}:   instr_ctrl = SLT_CTRL;
        {F3_3, F7_0}:   instr_ctrl = SLTU_CTRL;
        {F3_4, F7_0}:   instr_ctrl = XOR_CTRL;
        {F3_5, F7_0}:   instr_ctrl = SRL_CTRL;
        {F3_5, F7_32}:  instr_ctrl = SRA_CTRL;
        {F3_6, F7_0}:   instr_ctrl = OR_CTRL;
        {F3_7, F7_0}:   instr_ctrl = AND_CTRL;
        //---M_on
        {F3_0, F7_1}:   instr_ctrl = MUL_CTRL;
        {F3_1, F7_1}:   instr_ctrl = MULH_CTRL;
        {F3_2, F7_1}:   instr_ctrl = MULHSU_CTRL;
        {F3_3, F7_1}:   instr_ctrl = MULHU_CTRL;
        {F3_4, F7_1}:   instr_ctrl = DIV_CTRL;
        {F3_5, F7_1}:   instr_ctrl = DIVU_CTRL;
        {F3_6, F7_1}:   instr_ctrl = REM_CTRL;
        {F3_7, F7_1}:   instr_ctrl = REMU_CTRL;
        //---M_off
        default:        instr_ctrl = ILLEGAL_CTRL;
    endcase
end
```

Listing A.2: Code snippet of *riscv_decoder.skel* used in ALU instructions identification.

We then parse this *riscv_decoder.skel* file with a Python script called "extension parser". This script allows us to specify which extensions we want to support, reads the skeleton file, and generates a clean SystemVerilog RTL implementation containing only the necessary code for the specified extensions.

The workflow for the decoder implementation generation presented in Figure A.1 proceeds as follows:

1. We define control constants in extension tables in CSV format, with each row corresponding to the control signals of an instruction.
2. A Python script, referred to as the "instruction control generator" in Figure A.1, reads the tables of the specified ISA extensions and generates the *instruction_pkg.sv* package file with SystemVerilog constants that we will import, similar as code A.1.
3. The decoder implemented in SystemVerilog, contains tags to identify code sections for specific extensions, creating the *riscv_decoder.skel* file.
4. Another Python script, called the "extension parser," reads the *riscv_decoder.skel* file and generates the final RTL code, including only the segments necessary for the specified extensions.
5. The final file, the "RISC-V Decoder", includes the *instruction_pkg.sv* package of instruction control constants and contains clean, simple and maintainable code that exclusively supports the specified extensions.

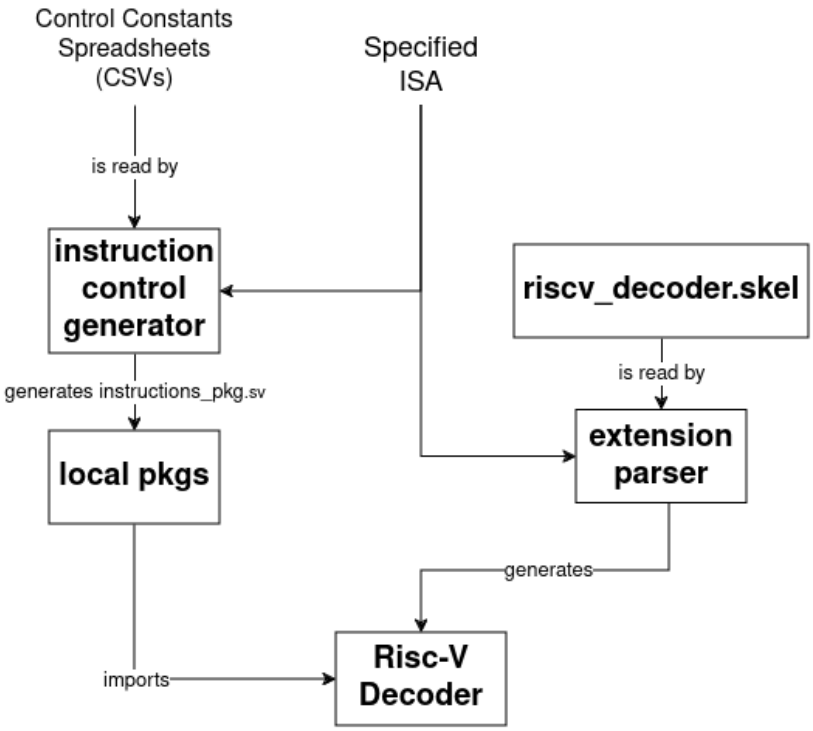


Figure A.1: Decoder implementation workflow diagram.