Politecnico di Torino

Politecnico di Torino

Electronics for Embedded Systems

A.A. 2025/2026

Graduation Session July 2026

Study and Development of a Simultaneous Multithreading RISC-V Architecture

Supervisors:

Corrado De Sio

Luca Sterpone

Giorgio Cora

Candidate:

Alex Turiani

Abstract

In recent years, RISC-V has emerged as a highly attractive Instruction Set Architecture (ISA) thanks to its open standard nature, portability, and modularity. Unlike proprietary ISA, RISC-V enables extensive customization through both standard and custom extensions, making it particularly suitable for specialized domains such as artificial intelligence, aerospace, and defense. Its transparent specification also supports greater hardware inspectability and design trust, which are essential aspects in security- and reliability-critical environments.

This thesis presents the design and implementation of a custom RISC-V processor architecture supporting Out-of-Order (OoO) execution and Simultaneous Multithreading (SMT).

A central contribution of this work is the integration of a Reorder Buffer (ROB) into the processor pipeline. The ROB represents a key structure for enabling OoO execution, as it allows instructions to be issued and executed as soon as their operands become available, independently of their original program order. By decoupling instruction execution from architectural state update, the ROB ensures in-order commit, thereby preserving precise exceptions and supporting correct recovery from branch mispredictions. This mechanism reduces pipeline stalls caused by long-latency operations and increases the Instruction-Level Parallelism (ILP).

Building on this OoO infrastructure, the processor is further extended to support SMT, which allows a single core to execute multiple hardware threads. By sharing execution resources among different threads, the processor can optimize functional unit utilization, especially when one thread is stalled due to data dependencies or memory latency. The proposed architecture includes parallelized Fetch and Decode stages and adopts two independent ROBs to manage the state of two hardware threads. This solution aims to increase overall throughput while avoiding the area and power overhead associated with duplicating the entire processor core.

The architecture was implemented in VHDL and validated through synthesis on FPGA platforms. Experimental results show that a configuration with an 8-entry ROB achieves a stable clock frequency of 31 MHz. Although this result was obtained without exhaustive low-level architectural optimizations and without register renaming, it demonstrates the feasibility of the proposed design. The current implementation successfully manages Read-After-Write (RAW) hazards through the ROB, ensuring correct data dependency handling across both hardware threads.

The proposed architecture provides a solid baseline for future developments, including the introduction of dedicated ISA extensions for thread lifecycle management, the implementation of register renaming to further reduce data hazards and improve timing performance, and the exploration of reliability-oriented optimizations for mission-critical applications.

Table of Contents

List of Tables

List of Figures

List of Acronyms

AI Artificial Intelligence

ALU Arithmetic Logic Unit

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

BRAM Block Random-Access Memory

CGMT Coarse-Grained Multithreading

CISC Complex Instruction Set Computer

CLB Configurable Logic Block

CPI Cycles Per Instruction

CPLD Complex Programmable Logic Device

CSR Control and Status Register

DSP Digital Signal Processing

DUT Design Under Test

EHE Efficient Hint-Based Event

ELF Executable and Linkable Format

EX Execute (pipeline stage)

FGMT Fine-Grained Multithreading

FIFO First-In, First-Out (buffer)

FPGA Field-Programmable Gate Array

FSM Finite-State Machine

GHDL GHDL (open-source VHDL simulator and compiler)

GPIO General-Purpose Input/Output

HDL Hardware Description Language

HW Hardware

ID Instruction Decode (pipeline stage)

IF Instruction Fetch (pipeline stage)

ILP Instruction-Level Parallelism

IMT Interleaved Multithreading

IOB Input/Output Block

IOE Input/Output Element

IoT Internet of Things

IP Intellectual Property (pre-designed hardware core)

IPC Instructions Per Cycle

IQ Issue Queue

ISA Instruction Set Architecture

ISR Interrupt Service Routine

LE Logic Element

LED Light-Emitting Diode

LUT Look-Up Table

MEM Memory Access (pipeline stage)

MMCM Mixed-Mode Clock Manager

NRE Non-Recurring Engineering

OoO Out-of-Order

PAL Programmable Array Logic

PC Program Counter

PLA Programmable Logic Array

PLD Programmable Logic Device

PLL Phase-Locked Loop

POSIX Portable Operating System Interface

PS7 Zynq Processing System (Zynq-7000 hard processor subsystem)

pthreads POSIX Threads

RAT Register Alias Table

RAW Read-After-Write

RISC Reduced Instruction Set Computer

ROB Reorder Buffer

ROM Read-Only Memory

RS Reservation Station

RT-OS Real-Time Operating System

RTL Register-Transfer Level

RV32I RISC-V 32-bit Integer (base instruction set)

SMT Simultaneous Multithreading

SoC System-on-Chip

SW Software

TCL Tool Command Language

TLP Thread-Level Parallelism

UART Universal Asynchronous Receiver-Transmitter

VCD Value Change Dump (waveform file format)

VHDL VHSIC Hardware Description Language

WAR Write-After-Read

WAW Write-After-Write

WB Writeback (pipeline stage)

Zicsr Zicsr (RISC-V Control and Status Register instruction-set extension)

Chapter 1

Introduction

Over the past few decades, the design of computer architectures has consistently evolved toward paradigms aimed at maximizing hardware efficiency and processing performance. A primary driver of this evolution is the exploitation of instruction-level parallelism, which allows multiple instructions to be processed simultaneously. Within this domain, the Reduced Instruction Set Computer (RISC) philosophy has provided a solid foundation by establishing simplified instruction sets that streamline hardware design. Recently, the open-source RISC-V Instruction Set Architecture (ISA) has emerged as a major standard in many fields, from ground to space applications [1] [2].

Furthermore, the integration of custom processor microarchitectures on reconfigurable hardware has been greatly accelerated by the use of soft-core processors implemented on Field-Programmable Gate Arrays (FPGAs). FPGAs are semiconductor devices consisting of a matrix of configurable logic blocks and programmable interconnects that can be reconfigured by the designer after manufacturing to implement any digital circuit. Unlike hard-core processors, which are physically etched into the silicon die, soft cores are synthesized from Hardware Description Languages (HDLs) directly onto this programmable logic fabric. This approach provides unprecedented design flexibility, allowing engineers to customize execution units, cache structures, and instruction extensions to match specific workload demands. The open-source and modular nature of the RISC-V ISA makes it an ideal architecture for soft-core implementations, enabling rapid prototyping, architectural design exploration, and physical verification at near-silicon speeds without the prohibitive costs and time-to-market constraints of custom Application-Specific Integrated Circuit (ASIC) fabrication.

1.1 The RISC-V Instruction Set Architecture

An Instruction Set Architecture (ISA) serves as the abstract boundary and fundamental contract between hardware and software [3]. It defines the set of instructions, registers, memory models, and addressing modes that a programmer or compiler can target, while leaving the physical implementation details (the microarchitecture) open to the hardware designer. This separation of interface from implementation is crucial for software portability, enabling compilers and operating systems to run unmodified across different microarchitectural designs.

Since the inception of the first microprocessors in the late 1960s and their subsequent commercialization in 1969, computer design has undergone substantial improvements. Early digital systems heavily relied on Complex Instruction Set Computer (CISC) architectures. CISC machines were characterized by a rich and variable-length instruction set, where individual instructions could execute multiple sequential operations, including direct memory-to-memory manipulations. This organization was primarily motivated by the need to minimize the memory footprint of programs, which was a critical requirement during the early decades of computing due to the extremely high cost of memory [3].

In contemporary computing, memory density and cost are no longer limiting factors, whereas processor execution speed and power efficiency have become the primary design constraints. Consequently, architectural research and industrial implementations have shifted toward the Reduced Instruction Set Computer (RISC) paradigm. RISC architectures focus on simple, orthogonal instructions that are executed in a few clock cycles. By simplifying the instruction set, RISC processors require fewer hardware resources and exhibit a less complex control logic. This simplicity not only increases the execution speed of the microprocessor but also significantly reduces the hardware design, simulation, and verification cycle [3].

Historically, the proprietary nature of CISC and early RISC architectures alike meant that instruction sets were tightly controlled by a small number of vendors, and microarchitectural innovation was largely confined within closed corporate ecosystems. The fifth generation of RISC research, developed in 2010 at the University of California, Berkeley, marked a departure from this model with the introduction of the RISC-V Instruction Set Architecture [4]. Unlike traditional proprietary architectures (such as x86 and ARM), RISC-V is structured as a completely free and open-standard ISA, placing the specification itself in the public domain rather than behind licensing agreements. Modularity is the other core principle of its design, separating a mandatory base integer instruction set from optional, standard extensions (covering, among others, multiplication, atomic operations, and floating-point arithmetic). This combination of openness and modularity allows the research and industrial community to inspect, modify, and extend processor architectures without restriction, which is precisely what

makes RISC-V an attractive vehicle for academic microarchitectural exploration: a designer can implement only the subset of the architecture required by a given study, and freely augment the core with custom, non-standard functional units.

For the purposes of this work, it is sufficient to note that the processor implements the base 32-bit integer instruction set (RV32I) together with the Zicsr extension for Control and Status Register access [5], which provides the minimal but complete foundation on which the Out-of-Order and multithreading mechanisms are built. The detailed specification of this instruction subset, including its register organization, instruction formats, and CSR mechanism, is deferred to Chapter 3, where it is discussed alongside the remaining architectural background required to introduce the processor design.

1.1.1 Five-Stage Pipeline Architecture

Pipelining is a fundamental implementation technique that enables multiple instructions to overlap in execution [3]. Instead of waiting for a single instruction to complete all its processing phases before starting the next one, the processor is partitioned into a sequence of processing units called pipe stages. Each stage performs a specific sub-task of the instruction execution. The instructions advance synchronously from one stage to the next at each clock edge. This temporal parallelism increases the overall instruction throughput, which is defined as the number of instructions completed per unit of time.

In a non-pipelined processor, each instruction must run to completion before the next begins, resulting in a high Cycles Per Instruction (CPI) value. Pipelining aims to reduce the CPI closer to the ideal limit of 1.0 by overlapping the execution of instructions. Simultaneously, dividing the processor logic into smaller, balanced pipe stages reduces the propagation delay between registers, allowing the Clock Cycle Time (and thus the clock frequency) to be improved. Although pipelining increases throughput, it does not reduce the execution latency of an individual instruction; in fact, the overhead of pipeline control and register propagation can slightly increase individual instruction latency.

To prevent instructions in different stages from interfering with each other, pipeline registers or latches are placed between adjacent stages. These registers temporarily store the data and control signals generated in one stage and present them as inputs to the next stage at the start of the next clock cycle. The cycle time of a pipelined processor is determined by the propagation delay of the slowest stage, plus the setup and hold time overhead of the pipeline registers. Therefore, achieving optimal performance requires balancing the processing load across all stages.

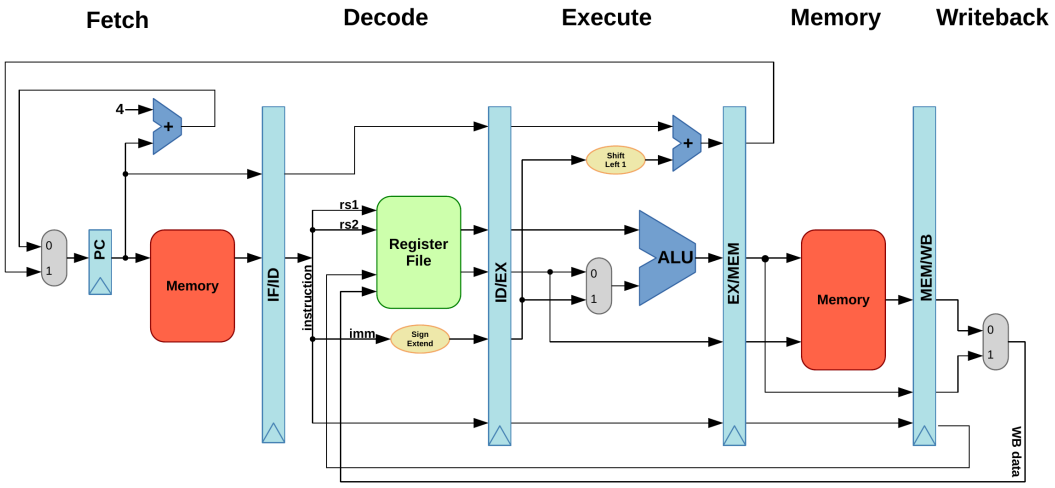

Figure 1.1: 5-Stage Pipeline Datapath

The pipeline technique is applied to RISC processors by splitting the architecture into the five distinct stages shown in Figure 1.1 [3]:

1. **Instruction Fetch (IF):** During this first stage, the processor reads the instruction from instruction memory using the address contained in the Program Counter (PC). Simultaneously, the PC is incremented by 4 bytes to point to the next sequential instruction, preparing the system for the next cycle.

2. **Instruction Decode (ID):** In the second stage, the instruction word fetched in the previous stage is parsed. The decoder identifies the opcode and extracts the register specifiers to access the register file. The register file yields the operand values stored in the requested registers. At the same time, any immediate field present in the instruction format is extracted and sign-extended to a 32-bit value.

3. **Execute (EX):** The third stage is the Execute stage, where the actual computation occurs. The Arithmetic Logic Unit (ALU) performs the operation specified by the instruction, such as addition, subtraction, shift, or logical operations, using the operands fetched during the decode stage. For memory access instructions, the ALU calculates the effective memory address by adding the base register value and the sign-extended immediate offset. For branch instructions, the ALU compares the operands to evaluate the branch condition, while additional hardware computes the target branch address.

4. **Memory Access (MEM):** During the fourth stage, the processor interacts with the data memory. For load instructions, the processor reads data from

the memory address computed in the execute stage. For store instructions, the processor writes the source register value into the memory. Instructions that do not perform memory operations simply pass their execution results from the ALU directly to the next stage without performing any active memory operations.

5. **Writeback (WB):** The final stage is the Writeback stage. During this phase, the processor updates the register file by writing the final result of the instruction, which can originate either from the ALU or from the data memory, into the destination register specified by the instruction. This step commits the execution of the instruction and makes its result available to subsequent instructions.

1.2 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) are configurable semiconductor integrated circuits that can be programmed and reconfigured by the user after manufacturing [6]. To understand the significance of FPGAs, it is essential to trace the evolution of Programmable Logic Devices (PLDs) and contrast them with Application-Specific Integrated Circuits (ASICs).

Historically, digital system designers utilized various classes of custom hardware [7]:

- **Full-Custom ASICs:** These are custom-manufactured silicon chips tailored for a single dedicated application. ASICs offer optimal performance, high logical density, and low dynamic power consumption. However, they suffer from extremely high Non-Recurring Engineering (NRE) costs (due to the expensive production of photolithographic silicon masks) and a long time-to-market. A design error in an ASIC requires a complete redesign and remanufacturing, incurring severe financial and scheduling penalties.

- **ROM-Based PLDs:** Early programmable logic used memory arrays to implement combinational functions, where address pins acted as inputs and data pins acted as outputs. The memory content represented the truth table. However, their size grows exponentially ($2^m \times n$, where m is the number of inputs and n is the number of outputs), making them impractical for more than a few inputs.

- **Programmable Logic Arrays (PLAs):** To reduce size, PLAs introduced programmable AND and OR planes, allowing only the required product terms (implicants) to be implemented. While flexible, PLAs are slow due to the high parasitic resistance and capacitance of the two programmable planes.

- **Programmable Array Logic (PALs):** PALs improved speed by using a programmable AND plane and a fixed OR plane. Limiting the programmability of the OR plane reduced parasitic capacitances, making PALs faster than PLAs, although less flexible.
- **Complex Programmable Logic Devices (CPLDs):** CPLDs grouped multiple macrocells (consisting of a small PLA/PAL structure and a flip-flop) into macroblocks connected by a programmable routing matrix. CPLDs are permanent, instant-on, and highly predictable in terms of propagation delays, making them ideal for glue logic. However, they contain a very limited number of storage elements (flip-flops), making it impossible to implement complex processors or data paths.

FPGAs address these limitations by offering a highly scalable, register-rich architecture. They provide off-the-shelf hardware programmability, reducing the NRE costs to zero for the end-user, shortening the time-to-market, and permitting in-system updates or bug fixes. These advantages come at the cost of higher per-unit prices, larger silicon area, higher dynamic power consumption, and lower maximum operating frequencies compared to equivalent ASIC implementations.

1.2.1 FPGA Internal Architecture

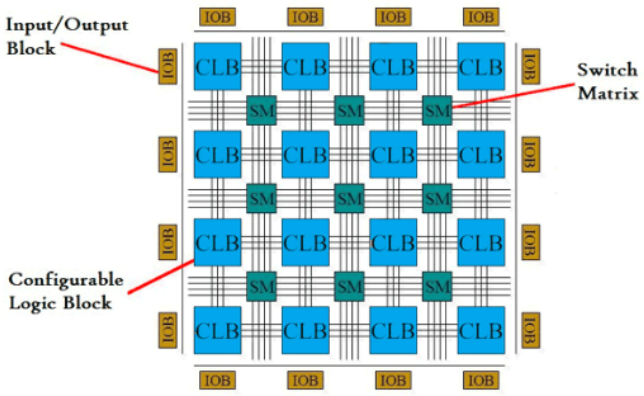

Figure 1.2: FPGA Structure

The internal structure of an FPGA is characterized by a matrix array of configurable elements and programmable routing resources, as shown in Figure 1.2 [6]. The primary building blocks of an FPGA include:

- **Configurable Logic Blocks (CLBs) or Logic Elements (LEs):** These represent the basic units of logic implementation, organized in rows and

columns. Each CLB/LE contains one or more Look-Up Tables (LUTs), storage elements (flip-flops), and multiplexers. The LUTs, implemented using SRAM cells, act as small memories that store truth tables to implement arbitrary combinational functions of n inputs. To prevent an exponential size explosion in memory cells (2^n), FPGAs typically restrict LUT inputs to a range between 3 and 6. The associated flip-flops provide sequential logic capabilities, and the multiplexers select between combinational (direct LUT output) or registered/sequential outputs, as illustrated in Figure 1.3.

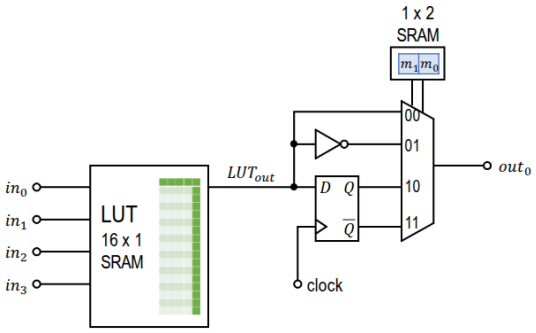

Figure 1.3: Logic Element (LE)

- **Programmable Interconnect Fabric:** This is a network of segmented wire channels and switch matrices. The switch matrices consist of pass transistors or transmission gates controlled by SRAM configuration cells, enabling dynamic routing of signals between different CLBs/LEs.
- **Input/Output Blocks (IOBs or IOEs):** These blocks interface the internal logic fabric with the external pins of the device. They support various configurable properties, such as bidirectional signals, pull-up/pull-down resistors, variable drive strengths and different voltage standards.
- **Embedded Hard Blocks (Hard IP Cores):** To overcome the performance and area limitations of the programmable fabric, modern FPGAs embed dedicated, non-programmable circuits directly in silicon. These hard blocks include Block RAMs (such as Altera's M9K memory blocks) to provide high-speed temporary storage, hardware multipliers and digital signal processing (DSP) blocks (such as the DSP48E2 core in Xilinx Virtex-7) for complex arithmetic operations, Phase-Locked Loops (PLLs) for clock generation and management, and high-speed multi-gigabit transceivers.

Ultimately, FPGAs serve as the primary platform for academic research and

rapid computer architecture prototyping. Rather than relying solely on software-based simulator models, which are slow and do not capture hardware-level timing and physical constraints accurately, implementing custom processor cores (such as the RISC-V implementation detailed in this work) on an FPGA fabric allows for physical, real-time testing. This enables the execution of benchmark software suites at near-hardware speeds and facilitates precise hardware-level measurements, providing a critical validation step prior to any potential ASIC fabrication.

1.3 Main Contribution

The work presented in this thesis addresses the design, implementation, and evaluation of a custom RISC-V processor microarchitecture that extends the open-source Potato soft-core with Out-of-Order (OoO) execution and Simultaneous Multithreading (SMT) capabilities. The baseline Potato core follows a classical five-stage in-order pipeline, which, while simple and resource-efficient, is unable to exploit the Instruction-Level Parallelism (ILP) available in a program whenever an instruction is delayed by a long-latency operation or an unresolved data dependency: every subsequent instruction must stall behind it, regardless of its own readiness to execute. The central contribution of this thesis is the design of a Reorder Buffer (ROB), a hardware structure inserted between the Decode and Execute stages of the pipeline, which decouples the point at which an instruction executes from the point at which it commits its result to the architectural state. This decoupling allows instructions to execute out of program order as soon as their operands become available, while a strict in-order commit policy is enforced at retirement, thereby preserving precise exceptions and enabling correct recovery from branch mispredictions. Read-After-Write (RAW) hazards are resolved directly through the ROB, without resorting to a dedicated register renaming mechanism, which keeps the additional hardware footprint contained.

Building on this OoO infrastructure, the architecture is further extended to support SMT, allowing a single physical core to manage two independent hardware threads concurrently. This extension required parallelizing the Fetch and Decode stages so that both threads can be serviced every cycle, and instantiating two independent ROBs, one per hardware context, arbitrated to share the remaining execution and memory resources. This design increases the utilization of the functional units, particularly when one thread is stalled due to a data hazard or a memory access, at a fraction of the area cost that duplicating the entire core would require. Both the OoO-only and the SMT variants of the processor were implemented in VHDL; the Reorder Buffer was first synthesized on its own across several table sizes, and the complete core was further evaluated in its in-order, OoO, and SMT configurations, with the resulting timing, area, and power figures

discussed in detail in Sections 5.2.1 and 5.2.2.

The remainder of this thesis is organized as follows. Chapter 2 reviews the state of the art in hardware multithreading and Out-of-Order execution, surveying existing RISC-V-based implementations and positioning the present work with respect to them. Chapter 3 introduces the theoretical and microarchitectural background necessary to follow the remainder of the thesis, covering the RISC-V RV32I and Zicsr instruction subset, classical pipeline hazards, the baseline Potato processor, and the conceptual foundations of Out-of-Order execution and hardware multithreading. Chapter 4 describes the design and implementation of the Reorder Buffer and details the modifications applied to the Potato pipeline to obtain, first, the OoO-only architecture and, subsequently, the full SMT architecture. Chapter 5 presents the experimental evaluation, first verifying the functional correctness of the Out-of-Order core through simulation against the RV32I ISA compliance suite and characterizing the resulting cycles-per-instruction cost relative to the In-Order baseline, and then reporting synthesis results across different Reorder Buffer sizes and comparing the complete in-order, OoO, and SMT cores against one another at a fixed table size, discussing the resulting trade-offs between maximum clock frequency, hardware area, and power consumption. Finally, Chapter 6 summarizes the main findings of this work and outlines directions for future development, including register renaming and reliability-oriented extensions of the proposed architecture.

Chapter 2

State of the Art

The demand for high-performance and resource-efficient processing units has led to the adoption of advanced microarchitectural techniques in modern microprocessors. While instruction-level parallelism (ILP) has been historically exploited via superscalar and out-of-order execution, it is increasingly limited by program dependencies and the memory wall. Hardware multithreading has emerged as a key paradigm to improve pipeline utilization by exploiting thread-level parallelism (TLP). With the rise of the open-source RISC-V Instruction Set Architecture (ISA), researchers and hardware designers have gained an extensible platform to implement and evaluate various multithreading microarchitectures.

In Coarse-Grained Multithreading (CGMT), the processor executes a single thread until it encounters a long-latency event, such as an external memory access or a cache miss. When a stall is detected, the hardware switches context to another thread, which minimizes the hardware cost by sharing most of the pipeline stages and registers but incurs a context-switch overhead of several clock cycles to flush the pipeline and load the state of the new thread. Because the switch is triggered only by infrequent, easily detectable events, CGMT keeps the required hardware modifications close to those of a single-threaded pipeline, at the cost of leaving most of the achievable thread-level parallelism unexploited between switches.

Conversely, Fine-Grained Multithreading (FGMT) and Interleaved Multithreading (IMT) models are widely used in embedded and real-time systems due to their deterministic timing behavior and low hardware complexity. For low-power Internet-of-Things (IoT) end-nodes, Cheikh et al. [8] proposed an open-source processing core family compliant with the RISC-V ISA on the software side and the PULPino platform on the hardware side. The design implements an interleaved multithreading pipeline with four hardware contexts (logical cores). By interleaving instructions from the four threads cycle-by-cycle, the microarchitecture tolerates memory access latency without requiring complex data dependency detection or bypass multiplexers, achieving high energy efficiency suitable for edge IoT devices.

However, this simplicity is obtained by not implementing any hardware interlock for data hazards at all: correctness is instead guaranteed either by keeping the number of active threads above a fixed *Thread Pool Baseline*, filling the pipeline with void, NOP-like threads whenever fewer real threads are available, or by relying on the compiler to statically schedule instructions so that no thread reads a register before its update has completed. Hazard avoidance is therefore pushed onto a minimum thread-count constraint or onto software scheduling, and the fixed round-robin interleaving cannot favor a thread whose operands are ready over one that is not. The Reorder Buffer (ROB) at the core of this thesis resolves RAW hazards directly in hardware, on a per-thread basis, allowing instructions to issue out of program order as soon as their operands are available, without requiring a minimum active-thread count or hazard-aware instruction scheduling.

Escobar [9] adopted the same fine-grained, interleaved approach in a multithreaded extension of the Lagarto processor, integrating it into the preDRAC System-on-Chip (SoC) to leverage its multicore capabilities. Because the arithmetic unit and the memory interface are shared among threads without any decoupling mechanism, a single long-latency instruction issued by one thread, such as a division or a memory access, locks the entire core and stalls every hardware thread simultaneously rather than only the issuing one. The author's own evaluation identifies this full-core lock as the dominant source of throughput loss and reports the corresponding analysis of cache-related effects as inconclusive, leaving the introduction of a dedicated memory stage to decouple thread progress as future work. This is precisely the limitation the design in this thesis is built to avoid: because instruction lifetimes and hazards are tracked independently for each thread in its own ROB entries, a long-latency operation in one thread never blocks the fetch, issue, or commit of the other thread, decoupling their progress at the microarchitectural level rather than relying on best-effort software mitigation.

In reconfigurable logic (FPGA) environments, Lang [10] introduced *FGMT-RiscV*, a fine-grained multithreaded soft-core processor. The microarchitecture organizes thread management using a closed-ring structure where tokens circulate during normal execution, with each token representing one of the independent threads and containing the thread's Program Counter (PC) and its thread identifier. A key contribution of Lang's work is its zero-latency interrupt handling mechanism. When a thread executes a `wfi` (Wait For Interrupt) instruction, its token is routed to a wait stage in the interrupt controller. Upon receiving an interrupt, the token is reinjected into the processing ring immediately, avoiding the latency associated with traditional software interrupt service routines (ISRs) and context switches. Being an early-stage proof of concept, however, the work reports only FPGA area occupancy (approximately one fifth of a small Artix-7 device) and provides no throughput, IPC, or maximum clock frequency data, nor any discussion of how hazards between overlapping instructions of the same thread are resolved

beyond the token-passing discipline itself. This thesis complements this gap with a full synthesis-based characterization of the proposed ROB across several sizes, reporting maximum operating frequency together with LUT, register, and slice utilization, thereby quantifying an area/performance trade-off that FGMT-RiscV's proof-of-concept evaluation does not address.

While traditional Interleaved Multithreading (IMT) microarchitectures utilize static round-robin scheduling, this approach can lead to pipeline starvation if some threads are idle or stalled. To address this limitation, Eni et al. [11] proposed the Efficient Hint-Based Event (EHE) issue scheduling algorithm. The EHE scheduler utilizes early instruction decoding in the RISC-V pipeline to predict whether an instruction will cause a memory block or ALU stall. By dynamically scheduling instructions based on these hints rather than a static round-robin rotation, the EHE algorithm improves pipeline utilization and yields an Instruction Per Cycle (IPC) increase of up to 26% for a four-threaded configuration under the standard MiBench benchmark suite. Nevertheless, EHE remains a single-issue interleaved pipeline in which only one thread's instruction can be dispatched per cycle: while the hint-based scheduler steers around predictable load/store and branch stalls, a division instruction still blocks the shared execution unit for up to 32 cycles, during which no instruction from any thread is issued, reproducing on a smaller scale the very coarse-grained-like penalty that fine-grained scheduling otherwise seeks to avoid. The SMT design proposed in this thesis resolves this structurally rather than heuristically: independent ROBs together with a hardware arbiter for the shared execution stage allow one thread to keep dispatching and committing instructions while another thread's long-latency operation is still in flight, instead of relying on instruction-type hints to work around a fundamentally single-issue pipeline.

Finally, Simultaneous Multithreading (SMT) represents the state of the art in high-performance RISC-V processors, especially when combined with Out-of-Order (OoO) execution. By separating instruction dispatch from the strict program order, OoO pipelines execute instructions as soon as their operands are ready, while SMT ensures that execution units do not remain idle when a single thread lacks instruction-level parallelism. An implementation of a RISC-V SMT core for edge Artificial Intelligence (AI) acceleration was proposed by Tanaka et al. [12]. The processor, integrated into the "Chichibu" heterogeneous multicore SoC, supports two hardware threads running on an out-of-order pipeline with speculative execution. To reduce the hardware cost on FPGAs, the designers shared the arithmetic units and memory access interfaces between the two threads, duplicating only the thread-specific control logic, register files, and renaming tables. This sharing mechanism reduced the hardware resource utilization by more than 50% compared to a dual-core configuration, while achieving a 66% increase in the cumulative IPC. Achieving out-of-order execution in this design, however, requires a full register-renaming

mechanism, comprising a per-thread map table, a merged physical register file, and a free list, which the authors explicitly identify as an area cost that grows with both the number of architectural threads and the issue width; the same conclusion states that the number of threads and the issue width are hard-coded rather than parameterized, and that the core, being restricted to the base integer ISA, cannot execute the floating-point operations common in the AI workloads it targets. This thesis pursues the same OoO/SMT goal from an explicitly opposite design choice: RAW hazards are resolved directly by the ROB entries rather than through register renaming, eliminating the physical register file, map tables, and free list altogether, and the design is evaluated with a parameterized ROB across multiple sizes so that the resulting area/performance trade-off is made explicit rather than fixed by construction.

In the domain of real-time systems, Nojiri and Yamasaki [13] designed a multi-threaded RISC-V processor featuring eight logical cores. Unlike traditional SMT cores that execute threads with equal priority, this design implements a hardware priority manager to emulate the task scheduler of a Real-Time Operating System (RT-OS). High-priority threads are granted preferential access to the execution stage, guaranteeing deterministic execution times for critical tasks. Furthermore, to support more active software threads than the eight physical hardware contexts, the design includes a hardware-managed *context cache* that performs context switches in just four clock cycles, increasing the total system throughput (IPC) by up to 5.5 times compared to single-threaded execution [13]. This flexibility, however, is obtained through considerable microarchitectural complexity: the processor combines speculative out-of-order execution with five reservation stations covering vector and floating-point units and a fully custom, 19-instruction thread-control ISA extension supporting 256 priority levels and a 32-entry context cache, yet its hardware cost is never quantified, since the evaluation is carried out purely through RTL simulation without any FPGA synthesis, area, or maximum-frequency result. The evaluation is further limited by a toolchain issue the authors acknowledge directly: compiler optimizations could not be applied together with the inline-assembly thread-control instructions, so every reported IPC figure is measured on unoptimized binaries, which the authors note artificially depresses the baseline. This thesis instead targets FPGA resource efficiency directly, reporting measured synthesis results, namely maximum frequency, LUTs, registers, and slices, for the proposed ROB-based design, and keeps both the thread model and the ISA extension deliberately minimal (two threads, no custom priority mechanism) so that the resulting area/performance trade-off stays quantifiable and comparable across configurations.

For multicore RISC-V configurations, resource sharing at the bus and memory hierarchy level introduces significant bottleneck challenges. Kieu-Do-Nguyen et al. [14] proposed a hardware-software co-design framework to optimize thread

scheduling and bus arbitration. By implementing lightweight hardware monitors that feed performance metrics to a software scheduler, the co-design framework reduces memory coherence overhead and bus contention, outperforming traditional software-only scheduling algorithms. This approach, however, obtains thread-level parallelism through core replication rather than intra-core resource sharing: the proposed system instantiates four complete RISC-V cores, each with a private L1 cache, in addition to a shared L2 cache and per-core bus arbitrators, so that the area and static power of the design scale with the number of cores, and bus/coherence contention on the shared memory hierarchy is only partially mitigated by the proposed hybrid cache. This thesis pursues efficient multithreaded execution at a fraction of this hardware cost by obtaining thread-level parallelism within a single physical core: the ALU and the memory interface are shared between two hardware threads managed by dual ROBs, avoiding both core replication and the coherence infrastructure it requires.

Alongside hardware innovations, supporting software infrastructure is critical for multithreaded architectures. In standard operating systems (like Linux), thread scheduling and synchronization are managed by the kernel. However, in deeply embedded, real-time, or bare-metal environments, the operating system is either absent or lightweight, requiring direct hardware-software integration. Arisoy [15] addressed this challenge by developing a bare-metal implementation of the POSIX Threads (pthreads) standard library for RISC-V architectures. The library is built on top of a modified version of the *musl* C standard library, which is optimized for low-power embedded platforms like PULPissimo. Arisoy's work provides the necessary runtime environment to initialize multiple hardware contexts, handle inter-thread synchronization (mutexes and barriers), and manage thread creation without a full operating system kernel. By mapping software threads directly to RISC-V hardware contexts, the library achieves low-overhead synchronization and enables multithreaded C applications to execute efficiently on bare-metal SMT/IMT soft-cores. This solution, however, operates entirely above the hardware boundary: on the single-threaded cores it targets, concurrency is necessarily emulated by time-multiplexing software threads onto the same hardware context, so independent threads are never simultaneously present in the pipeline, and every thread switch, mutex, or barrier operation incurs a full software context save/restore handled by the runtime rather than by dedicated hardware. The SMT architecture developed in this thesis addresses this limitation at its source, keeping multiple hardware threads genuinely and simultaneously resident in the pipeline, with hazard resolution and in-order commit guaranteed by the ROB, removing the software scheduling and context-switch overhead that a bare-metal threading library such as Arisoy's must otherwise pay on every thread transition.

Table 2.1 summarizes the multithreading model and execution type of these state-of-the-art RISC-V designs, together with the main limitation identified in the

discussion above for each of them and the architectural choice through which this thesis compensates for it.

Reference	**Model / Exec.**	**Key Limitation**	**Compensation in This Thesis**
Cheikh et al. [8]	IMT / In-Order	No HW interlocks; needs min. thread count or SW scheduling	ROB resolves RAW hazards per thread, no min. threads
Escobar [9]	FGMT / In-Order	Shared units lock entire core on one long op	Independent ROBs decouple thread stalls
Lang [10]	FGMT / In-Order	No IPC/throughput/f_{max} data (PoC only)	Full synthesis results across ROB sizes
Eni et al. [11]	IMT / In-Order	Single-issue; division blocks all threads (32 cyc.)	Dual ROB + arbiter overlap execution during stalls
Tanaka et al. [12]	SMT / OoO	Renaming overhead; threads/issue width fixed	No renaming; ROB size parameterized
Nojiri & Yamasaki [13]	SMT / OoO	No synthesis data; unopt. IPC (toolchain bug)	Measured f_{max}/area; minimal 2-thread design
Kieu-Do-Nguyen et al. [14]	Multicore / In-Order	TLP via 4× core replication; bus contention	TLP within one core, no replication
Arisoy [15]	SW library	SW-only concurrency; time-multiplexed on 1 context	True simultaneous HW threads, ROB-managed

Table 2.1: State-of-the-art multithreaded RISC-V architectures, their main limitation, and how this thesis compensates for it.

The design developed in this thesis positions itself within the domain of Out-of-Order (OoO) SMT architectures that are straightforward to implement, achieved through a centralized component that manages each thread's execution state, hazard resolution, and the sharing of execution stage resources. Specifically, this work explores a microarchitectural design that keeps area overhead below that of the full-replication or full-renaming alternatives surveyed above, addressing a notable gap in the current state of the art. The proposed dual-thread Out-of-Order SMT core, implemented on an FPGA using a shared arithmetic logic unit (ALU) and dual Reorder Buffers (ROBs), serves as a resource-efficient solution for FPGA-based embedded acceleration. By combining out-of-order execution—using independent ROBs to resolve read-after-write (RAW) hazards per thread—with a

shared execution stage governed by a hardware arbiter, this work avoids both the full core replication and the register-renaming infrastructure that the state-of-the-art designs surveyed above rely on, at the cost of a measurable but comparatively lean area increase over the in-order baseline, quantified in Chapter 5; sharing the Arithmetic Logic Unit and memory interface between threads keeps the incremental cost of the second hardware context below that of a full core duplication, positioning this design between low-complexity in-order interleaved multithreading cores and resource-heavy multi-core processors.

Chapter 3

Background

This chapter provides the necessary theoretical and microarchitectural foundation required to understand the design, implementation, and evaluation of the multi-threaded Out-of-Order (OoO) processor core proposed in this thesis. It begins with an analysis of the RISC-V Instruction Set Architecture (ISA), focusing on the baseline integer specification and the Control and Status Register (CSR) extension. Subsequently, the fundamental pipeline hazards and their mitigation strategies are discussed. The baseline hardware architecture, the Potato processor, is then described, outlining its in-order organization and System-on-Chip (SoC) peripherals. Finally, the chapter introduces the core conceptual paradigms of Out-of-Order execution and hardware multithreading, including a detailed treatment of the Reorder Buffer (ROB), setting the context for the simultaneous multithreading modifications detailed in Chapter 4.

3.1 Instruction Set Architecture

The Instruction Set Architecture (ISA) serves as the abstract boundary between the hardware execution engine and the software instruction stream, defining the register organization, memory addressing modes, and instruction encoding. This separation of interface from implementation is fundamental to software portability: compilers and operating systems can target a given ISA regardless of the underlying microarchitecture [3].

For this work, the RISC-V architecture was chosen due to its open-source nature, modularity, and growing adoption in both academic research and industry [5]. Specifically, the processor implements the 32-bit base integer instruction set (RV32I) together with the Zicsr extension. This configuration provides a resource-efficient baseline sufficient for embedded systems without the hardware area overhead of floating-point or vector extensions. The performance advantages and architectural

complexity of Simultaneous Multithreading (SMT) and Out-of-Order execution can be thoroughly evaluated on this instruction subset, providing a solid foundation for lightweight soft-core processors targeting Field-Programmable Gate Arrays (FPGAs).

3.1.1 RV32I and the Zicsr Extension

Under the RV32I specification, all instructions are fixed at 32 bits in length and must be aligned to 4-byte boundaries in memory [5]. The base integer instructions are classified into several functional categories, each mapping directly to distinct hardware blocks within the execution datapath:

- **Register-Register Operations (R-Type)**: These instructions perform arithmetic, logical, or comparison operations between two source registers, writing the result back to a destination register. A representative example is `ADD x1, x2, x3`, which adds the contents of `x2` and `x3` and stores the result in `x1`. R-Type instructions require the register file to provide two simultaneous read ports and one write port, with the operation performed by the Arithmetic Logic Unit (ALU).
- **Register-Immediate Operations (I-Type)**: These instructions combine a source register value with a 12-bit sign-extended immediate operand encoded within the instruction word. For instance, `ADDI x3, x4, 5` adds the value of `x4` to the sign-extended constant 5 and stores the result in `x3`. An immediate generation unit in the decode stage extends the 12-bit constant to 32 bits before routing it to the ALU.
- **Upper Immediate Operations (U-Type)**: These instructions load a 20-bit immediate value into the upper 20 bits of a destination register, zeroing the lower 12 bits. The two U-Type instructions serve distinct purposes:
  - `LUI` (Load Upper Immediate) places the immediate directly into the destination register, enabling the construction of arbitrary 32-bit constants when paired with a subsequent I-Type instruction that supplies the lower 12 bits.
  - `AUIPC` (Add Upper Immediate to PC) adds the shifted immediate to the current Program Counter and writes the sum to the destination register, providing a mechanism for PC-relative address computation over a wide address range.

  Because U-Type instructions carry no source register specifiers, the entire instruction word beyond the opcode and destination fields is available for the immediate, yielding a compact encoding for large constant generation.

- **Control Transfer Instructions (B-Type and J-Type)**: These instructions alter the sequential execution flow by modifying the Program Counter (PC). They are divided into:
  - *Unconditional Jumps* (`JAL` and `JALR`): Force the PC to a target address while saving the return address (PC + 4) into a destination register. `JAL` uses the J-Type encoding (PC-relative offset); `JALR` uses the I-Type encoding (register-offset).
  - *Conditional Branches* (`BEQ`, `BNE`, `BLT`, `BGE`, etc.): Compare two registers and branch to a target address if the comparison evaluates to true; otherwise execution continues sequentially. These B-Type instructions require comparison logic in the execute stage.
- **Load and Store Instructions (I-Type and S-Type)**: These instructions transfer data between the register file and data memory.
  - *Load instructions* (I-Type, e.g., `LW`, `LH`, `LB`): Compute a memory address by adding a sign-extended 12-bit immediate to a base register, read the value from memory, and write it to a destination register.
  - *Store instructions* (S-Type, e.g., `SW`, `SH`, `SB`): Compute the address in the same fashion but write the contents of a source register to the specified memory location. Because stores read two registers (base address and data), the S-Type format splits the immediate field across two separate bit ranges, leaving both register specifiers accessible at fixed positions.
- **System Instructions**: This category includes instructions that manage the execution environment, trigger system calls (`ECALL`), and generate software breakpoints (`EBREAK`).

The Zicsr extension introduces instructions to interact with the Control and Status Registers (CSRs). These registers hold critical information regarding the processor state, execution privilege level, performance counters, and exception configuration. The CSR access instructions (e.g., `CSRRW`, `CSRRS`, `CSRRC`, and their immediate variants) perform atomic read-modify-write operations, allowing the software to query and configure system state without race conditions. In particular, cycle and instruction-retired counters accessible via CSRs provide a hardware mechanism for accurate performance evaluation. In a multithreaded core, CSR access logic must be carefully partitioned to handle thread-specific configurations and to ensure that hardware threads cannot read or write unauthorized system registers, preserving isolation and correctness.

3.2 Pipeline Hazards and Stalls

While pipelining significantly improves instruction throughput by overlapping the execution of multiple instructions, it introduces microarchitectural conflicts known as hazards. Hazards are conditions that prevent the next instruction in the instruction stream from executing in its designated clock cycle, thereby degrading processor performance and risking incorrect execution [3]. Hazards are classified into three primary categories:

1. **Structural Hazards**: These occur when two or more instructions in different pipeline stages attempt to access the same physical hardware resource simultaneously. A classic example is a single-ported memory shared for both instruction fetching and data access. When an instruction in the memory stage performs a load or store while the fetch stage requests a new instruction, a resource conflict arises. To eliminate structural hazards, architectures must either duplicate resources (e.g., implementing separate instruction and data memories, as in a Harvard organization) or stall one of the conflicting pipeline stages.

2. **Data Hazards**: These arise when an instruction depends on the result of a preceding instruction that has not yet completed execution. Because the pipeline overlaps execution stages, a subsequent instruction may attempt to read a source operand from the register file before an earlier instruction has written the updated value. This is known as a Read-After-Write (RAW) hazard, or true dependency: instruction B must read a register that instruction A, appearing earlier in program order, has not yet written. A RAW hazard reflects an actual data dependency between the two instructions, rather than a mere conflict over the reuse of a limited set of architectural register names, and cannot be removed by renaming or by reallocating storage: instruction B must, in one way or another, obtain the value produced by instruction A before it can execute correctly. For this reason, RAW hazards are the most common and performance-critical data hazard in pipelined processors. Detecting and resolving them, whether through stalling, forwarding, or a dedicated buffering structure that tracks every in-flight instruction's dependencies, is central to the correct operation of any pipeline that allows instructions to overlap or to execute out of their original program order.

3. **Control Hazards**: Also known as branch hazards, these are caused by instructions that modify the PC (branches and jumps). When a branch instruction is fetched, the processor cannot determine the next instruction to fetch until the branch condition is evaluated in the execute stage. This uncertainty creates a window during which either the wrong instructions are fetched or the pipeline must be stalled.

To resolve these hazards and guarantee correct program execution, processor microarchitectures rely on three primary mitigation strategies. The most straightforward approach is the insertion of pipeline stalls, commonly referred to as bubbles or no-operation (NOP) cycles. When a hazard is detected, the control logic temporarily suspends the advancement of instructions while allowing those already in later stages to proceed. Although conceptually simple, stalling increases the average Cycles Per Instruction (CPI), as idle cycles are introduced during which no productive work is performed.

CPI is a standard measure of processor performance, expressing the average number of clock cycles required to retire a single instruction over the course of a program's execution [3]. It is computed as the ratio between the total number of clock cycles consumed by the program and the total number of instructions it retires:

$$CPI = \frac{\text{Total Clock Cycles}}{\text{Instruction Count}}$$

A CPI of exactly 1 corresponds to the ideal case of a single-issue pipeline retiring one instruction on every clock cycle, with no stalls of any kind; any hazard-induced bubble, whether structural, data, or control in nature, adds otherwise unproductive cycles to the numerator without contributing additional retired instructions, and therefore raises CPI above this baseline. CPI is a useful metric because it isolates the architectural efficiency of a pipeline, that is, how effectively it keeps its stages occupied with useful work, from the raw clock frequency and from the length of the program under test, both of which vary independently of the pipeline's hazard-handling behavior. Combined with the clock cycle time and the instruction count of a given program, CPI directly determines total execution time,

$$Execution\ Time = Instruction\ Count \times CPI \times Clock\ Cycle\ Time,$$

which makes it the natural metric through which the cost of the stalls introduced by hazards, and the benefit of the mitigation techniques discussed below, can be quantified and compared across different pipeline designs [3].

To mitigate performance losses arising from RAW dependencies, hardware designs implement data forwarding (or bypassing). Rather than waiting for an instruction to complete the writeback stage, forwarding paths route computed values directly from the outputs of the execute or memory stage registers back to the inputs of the ALU. This bypasses the register file writeback latency and allows a dependent instruction to execute without delay, provided its required operand has already been computed.

Control hazards arising from branches and jumps are typically addressed through branch prediction and speculative execution. The processor employs prediction algorithms to estimate the outcome of a conditional branch and speculatively fetches instructions along the predicted path. If the prediction is correct, execution

proceeds without penalty; if incorrect, the speculatively fetched instructions are flushed from the pipeline and execution resumes from the correct target address, incurring a misprediction penalty proportional to the number of pipeline stages between fetch and branch resolution.

3.3 The Potato Processor

The baseline architecture selected for this thesis is the Potato processor [16], an open-source 32-bit RISC-V soft-core designed by Kristian Skordal. Written in modular VHDL, the Potato processor targets FPGAs and provides a clean, well-documented implementation. It was selected as the baseline for this work on account of its compliance with standard bus interfaces and its inclusion of a complete System-on-Chip peripheral subsystem, which allows rapid prototyping, hardware synthesis, and software validation without the need to develop supporting infrastructure from scratch.

The original Potato processor implements a classic 5-stage in-order execution pipeline, consisting of Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Writeback (WB). A description of these stages and their general operations is provided in Section 1.1.1.

The core supports the complete RV32I base integer specification (version 2.0) and machine-mode privileged operations as defined in the RISC-V Privileged Architecture (version 1.10) [16]. It interfaces with external memory and peripherals using the Wishbone Bus B4 protocol [16]. To handle external events, the processor provides up to eight maskable interrupt lines connected to an on-chip interrupt controller [16].

During this thesis, the Potato processor SoC was successfully ported and implemented on the AMD Pynq-Z2 development board, which features a Xilinx Zynq-7000 SoC FPGA, demonstrating the portability of the design beyond the originally targeted Arty board. The system-on-chip configuration integrates the processor core with several Wishbone-compatible peripherals [16]:

- **Timer**: A 32-bit timer with compare-match capability, generating periodic interrupts for real-time task scheduling.
- **GPIO**: A general-purpose input/output controller for interfacing with board switches, buttons, and LEDs.
- **UART**: A universal asynchronous receiver-transmitter with hardware FIFOs and a configurable baud rate, used for serial communication and console output during debugging and software validation.
- **Memory Controller**: An interface to Block RAM (BRAM) on the FPGA fabric, providing combined instruction and data storage.

Figure 3.1 shows the resulting Potato processor core together with its Wishbone-connected peripherals.

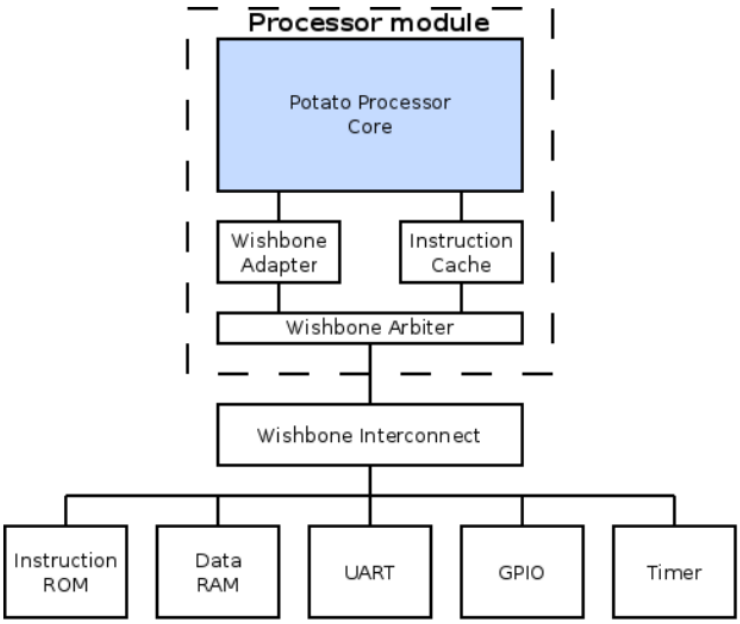

Figure 3.1: Potato Processor with Wishbone interfaces and Peripheral Modules.

3.4 Out-of-Order Concept

Classic in-order execution engines suffer from a fundamental limitation: when an instruction stalls (e.g., due to a long-latency memory access or a data dependency), all subsequent instructions are blocked regardless of whether they are independent of the stalling instruction. This phenomenon, known as head-of-line blocking, severely restricts the exploitable Instruction-Level Parallelism (ILP) and leaves functional units idle [3].

Out-of-Order (OoO) execution addresses this bottleneck by decoupling the sequential fetching of instructions from their actual execution. In an OoO processor, an instruction is dispatched to an execution unit as soon as its source operands are available and a suitable resource is free, irrespective of its position in the original program order. This allows the processor to bypass stalled instructions and execute independent work, maximizing functional unit utilization. Despite the out-of-order execution, the results must appear to the programmer and to exception-handling logic as if instructions executed strictly in program order.

To satisfy this requirement, modern OoO processors employ a hybrid microarchitectural model that preserves the illusion of sequential execution, illustrated in Figure 3.2. The model consists of three conceptual parts: an in-order front-end, where instructions are fetched sequentially and decoded; an out-of-order execution engine, where instructions issue and execute as soon as their operands are ready; and an in-order back-end, where results are committed to the permanent architectural state in the original program order.

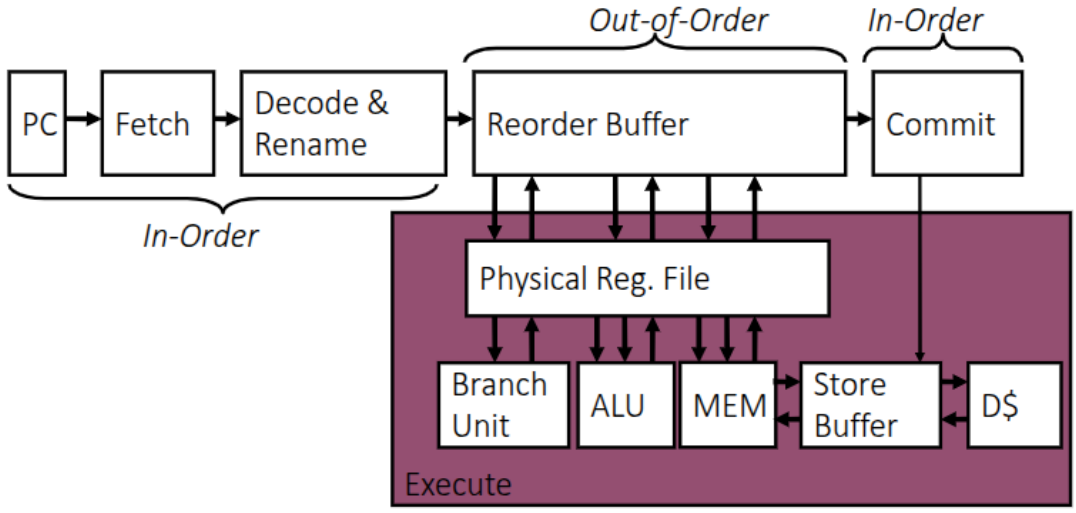

Figure 3.2: Out-of-Order Architecture of a Generic Processor.

The execution pipeline of a generic Out-of-Order processor is organized into the following microarchitectural stages [3]:

- **Fetch & Decode (In-Order)**: Instructions are fetched from instruction memory and decoded sequentially. The decode stage identifies the instruction type, extracts register specifiers, and generates immediate values.
- **Register Renaming (In-Order)**: To eliminate false dependencies caused by the reuse of a limited set of architectural register names, these names are mapped to entries in a physical register file or to Reorder Buffer (ROB) slots. This mapping is maintained in a Register Alias Table (RAT). Whenever a new instruction writes to a logical register, it is assigned a fresh physical destination, allowing earlier instructions that read the same logical register to proceed concurrently without conflict.
- **Dispatch and Allocation (In-Order)**: The renamed instruction is assigned a slot in a Reservation Station (RS) or Issue Queue (IQ) and allocated an

entry at the tail of the Reorder Buffer. If the queues are full, the front-end stalls until space becomes available.

- **Issue and Execute (Out-of-Order)**: Instructions in the Reservation Station monitor their source operands via the result broadcast network. When all operands are available and a suitable execution unit (e.g., ALU or load/store unit) is free, the instruction is issued and executed. This is the stage where true out-of-order execution occurs.
- **Writeback and Result Broadcast**: Upon completing execution, the result is broadcast on a common data bus so that waiting instructions in the Reservation Station can immediately capture it. The result is simultaneously written to the physical register file and recorded in the allocated ROB entry, which is then marked as complete.
- **Commit and Retirement (In-Order)**: Instructions are retired strictly from the head of the ROB in program order. When the instruction at the head is marked complete, its result is permanently committed to the architectural register file (or, for store instructions, to data memory). If the retiring instruction is associated with an exception or a branch misprediction, all subsequent ROB entries are squashed and execution resumes from the correct address.

3.4.1 The Reorder Buffer

The Reorder Buffer (ROB) is the central microarchitectural structure that enables out-of-order execution while preserving in-order commitment and precise exception semantics. The concept was formalized by Smith and Pleszkun [17] and is extensively discussed in Hennessy and Patterson [3]. Structurally, the ROB is a circular buffer managed by a tail pointer (allocation) and a head pointer (commitment), maintaining a record of all in-flight instructions in the exact order they were fetched from the instruction stream.

Each ROB entry stores the information required to commit the instruction's effect on the architectural state: the destination register identifier, the computed result value, an instruction status field (indicating whether the entry is pending execution, executing, or complete), and exception status information. When an instruction is decoded, a new entry is allocated at the tail of the ROB. The instruction then enters the out-of-order execution engine and, upon completing execution, writes its result into its assigned ROB entry and marks it as complete. Crucially, the architectural register file is not updated at this point; the result is held exclusively within the ROB until retirement.

The commit logic continuously monitors the instruction at the head of the ROB. When the head entry is marked complete and carries no exception, its result is permanently written to the architectural register file, the ROB entry is released, and the head pointer advances. This strictly in-order retirement policy guarantees that the architectural state always represents a coherent, exception-precise execution prefix of the program. If the instruction at the head of the ROB is associated with an exception or a branch misprediction, all entries following it in the ROB are squashed, the architectural register file state—which reflects only committed instructions—is used to restore the processor, and execution resumes from the correct address.

A critical operational function of the ROB is result forwarding, which directly resolves Read-After-Write (RAW) data hazards without requiring the result to pass through the register file first. When a newly decoded instruction reads a source register, the hardware checks whether any in-flight ROB entry holds the most recent write to that register. If a matching complete entry is found, its value is forwarded directly to the waiting instruction. If the matching entry is still executing, the waiting instruction must stall until the result becomes available. This mechanism allows dependent instructions to proceed as soon as their data is ready, regardless of whether it has been committed to the register file.

In a full-featured Out-of-Order processor, the ROB operates in conjunction with a register renaming stage to eliminate false dependencies caused by the reuse of a limited set of architectural register names. Renaming maps each new write to a logical register onto a freshly allocated physical register (or ROB entry), tracked by a Register Alias Table (RAT). This ensures that a later instruction writing to the same logical register does not interfere with an earlier instruction that is still reading it, and that the final committed value of a logical register is always the one produced by the architecturally last write to it. Upon commitment, the physical register holding the result of the retired instruction is written to the architectural register file, and the physical register previously aliased to that logical name is released back to the free list. At the same time, the ROB entry is deallocated and the head pointer advances. Together, the ROB and the renaming logic form the backbone of precise, high-performance Out-of-Order execution in modern superscalar processors [3].

The architecture developed in this thesis, presented in Chapter 4, deliberately departs from this full-featured model in two respects. First, it omits the register renaming stage entirely: RAW hazards are instead resolved directly within the ROB itself, at the cost of the all-pairs dependency comparison discussed in Section 4.1.1. Second, it does not implement the general result-forwarding mechanism described above: rather than making a completed-but-uncommitted entry's value immediately available to a waiting instruction, a dependent instruction remains blocked until the producing instruction has reached the head of the table and been committed

to the architectural Register File, as detailed in Section 4.1.1. This is a more conservative hazard-resolution policy than the textbook model just described, and its performance cost is quantified directly in Chapter 5, where it is identified as the dominant contributor to the cycles-per-instruction increase measured for the Out-of-Order core. Reintroducing register renaming and/or a completed-entry forwarding path on top of this design, so as to relieve both the all-pairs comparison and this stall, is discussed as a direction for future work in Chapter 6.

3.5 Hardware Multithreading

While Out-of-Order execution is effective at extracting Instruction-Level Parallelism (ILP) from a single instruction stream, single-thread performance is increasingly bounded by true data dependencies and the growing latency gap between processor and memory. Even the most aggressive OoO engine cannot execute an instruction before all of its dependencies are resolved. To overcome this limitation, modern architectures exploit Thread-Level Parallelism (TLP) via hardware multithreading [3].

Hardware multithreading allows a single physical processor core to execute multiple independent instruction streams (threads) concurrently. Each thread is represented by its own architectural state—a Program Counter (PC), a private register file, and a set of control and status registers. The execution resources of the core (ALUs, memory ports, caches) are shared among the active threads. The primary goal is to fill execution slots that would otherwise remain idle due to stalls or dependencies in a single thread, thereby reducing both vertical waste (idle cycles caused by long-latency events) and horizontal waste (unused issue slots in a superscalar machine).

There are three primary paradigms of hardware multithreading, which differ in the granularity and policy of resource sharing [3]:

1. **Fine-Grained Multithreading (FGMT)**: The processor switches between active threads on a cycle-by-cycle basis, typically following a round-robin policy. If a thread is stalled, the scheduler skips it and selects the next ready thread. This cycle-by-cycle interleaving hides short-latency stalls and simplifies hazard detection, since instructions from different threads occupying adjacent pipeline stages have no data dependencies between them. The main drawback is a reduction in single-thread performance: a given thread can issue at most one instruction every N cycles, where N is the number of active hardware threads.

2. **Coarse-Grained Multithreading (CGMT)**: A single thread executes continuously until it encounters a long-latency stall event, such as a last-level

cache miss. The processor then drains the pipeline and switches to another ready thread. CGMT preserves good single-thread performance under normal execution conditions because the active thread occupies the pipeline without interruption. However, it cannot hide short-latency stalls, and each context switch incurs a pipeline drain penalty of several clock cycles.

3. **Simultaneous Multithreading (SMT)**: is the most advanced paradigm. It extends a superscalar or Out-of-Order processor to issue and execute instructions from multiple threads within the same clock cycle, dynamically sharing execution resources (reservation stations, physical register files, load/store queues) across threads. Any thread whose instructions have ready operands may contribute to filling available issue slots simultaneously. SMT maximizes both ILP and TLP, substantially reducing vertical and horizontal waste. The primary disadvantage is the increased complexity of the renaming, scheduling, and resource arbitration logic, together with the potential for inter-thread resource contention.

Figure 3.3 illustrates how execution slots are utilized across these paradigms for a representative superscalar core.

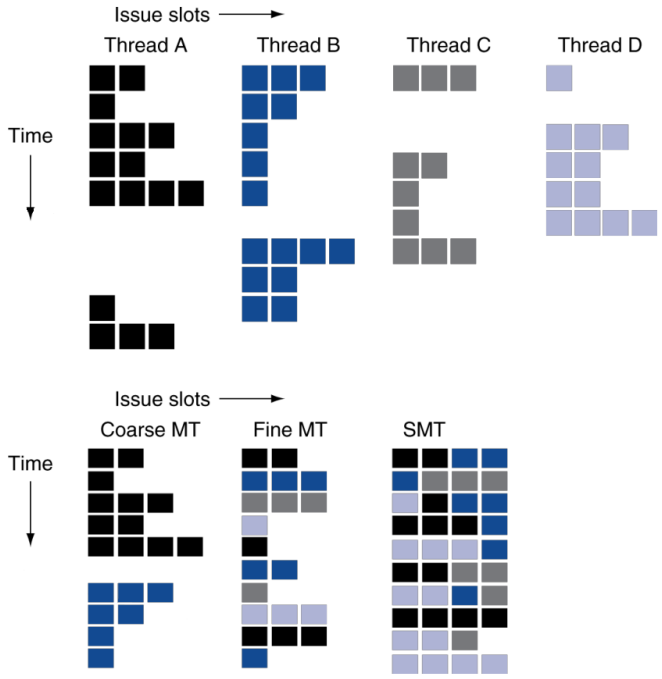

Figure 3.3: Comparison of execution slot utilization among multithreading paradigms: single-thread execution, Fine-Grained Multithreading (FGMT), Coarse-Grained Multithreading (CGMT), and Simultaneous Multithreading (SMT).

The theoretical and microarchitectural concepts introduced in this chapter form the direct foundation for the design decisions described in Chapter 4. The RISC-V ISA subset, pipeline hazard taxonomy, Reorder Buffer model, and multithreading paradigms discussed here are all concretely realized in the implementation of the Out-of-Order and SMT extensions to the Potato processor core.

Chapter 4

Architectures Development

The development of a multithreaded architecture required a systematic and incremental approach, given the complexity involved in achieving correct operation. Reaching this goal first required understanding which kind of component was actually needed, before moving to an initial Out-of-Order implementation and, finally, to the intended Simultaneous Multithreading (SMT) target architecture.

A design decision was made to develop a single module capable of managing the required complexity while retaining a degree of portability. Indeed, the Reorder Buffer (ROB) developed in this work is able to support out-of-order execution and, with the appropriate surrounding modifications, SMT as well.

Considerable room for improvement remains, since this constitutes an initial body of work; however, it has also shown interesting potential applications in the context of reliability, a topic discussed as part of future work in Section 6.1.

This chapter therefore addresses the implementation and the features of the Reorder Buffer. Subsequently, its use within an Out-of-Order architecture is presented, followed by a description of the modifications required to the Potato architecture in order to realize the SMT architecture.

4.1 Reorder Buffer Design

The module was developed in VHDL, in order to maintain coherence with the hardware description language used throughout the rest of the architecture. Figure 4.1 shows the ports exposed by the component: inputs are placed on the left-hand side, while outputs are placed on the right-hand side.

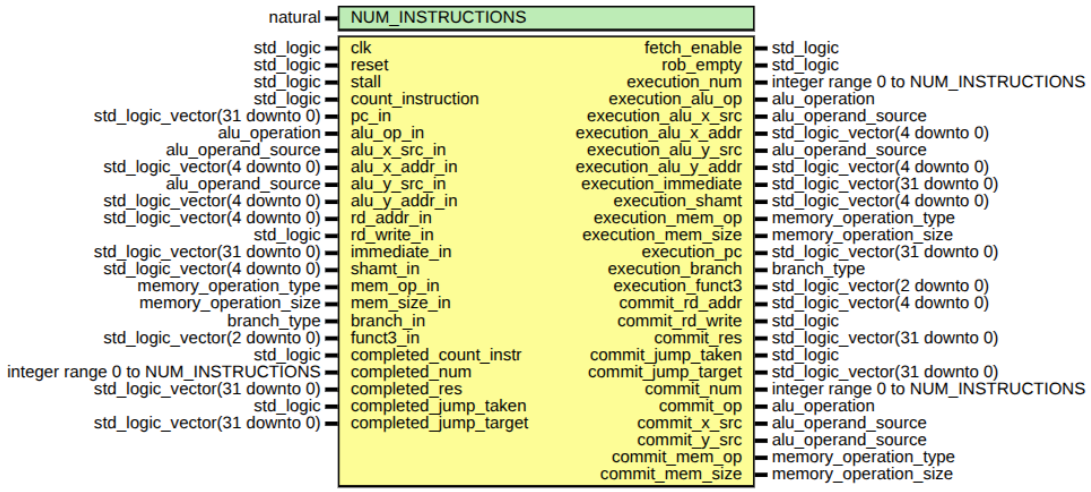

Figure 4.1: Reorder Buffer Module.

Their purpose and the motivation behind their presence are explained in the remainder of this section. The inputs are described first, followed by the outputs.

Before examining each port individually, it is useful to summarize the lifecycle that an instruction follows once it reaches the ROB. Each decoded instruction is allocated a new entry at the tail of the table, and its source operands are checked for dependencies against every older, still in-flight instruction. Once both operands are free of hazards, the instruction becomes eligible for dispatch to the Execute stage; upon completion, its result is written back into its own entry rather than directly into the Register File. Only when an instruction reaches the head of the table, in strict program order, is its result actually committed to the architectural state. This separation between out-of-order completion and in-order commitment is what allows the ROB to reorder execution internally while still exposing to the rest of the core a processor that appears, from the outside, to execute instructions strictly in sequence.

The component was designed to be parametric, in order to allow greater implementation flexibility. `NUM_INSTRUCTIONS` is the generic that defines the number of instructions that can be held within the ROB table at any given time. This allows the module to be resized and optimized according to the specific intended use.

A set of generic control signals ensures correct coordination with the rest of the processor, namely `clk`, `reset`, and `stall`.

The signals required to execute an instruction arrive from the Decode stage, together with `count_instruction`, which acts as an enable, informing the ROB that the incoming data is valid. This allows for faster sampling, since no equality check against the data already stored in the table needs to be performed. This signal was already present in the original Potato architecture, where it drove — and continues to drive — the enable of the CSR unit's counter used for performance monitoring. The signals required to execute the instruction are the following:

- `pc_in`: Defines the Program Counter associated with the current instruction, required for J-Type and U-Type instructions.
- `alu_op_in`: Defines the operation to be executed and, consequently, the Arithmetic Logic Unit (ALU) resource it targets.
- `alu_x_src_in & alu_y_src_in`: Define the resource targeted by a specific operand. For example, `alu_y_src_in=ALU_SRC_PC` indicates that the second operand must be taken from the field dedicated to the Program Counter (PC).
- `alu_x_addr_in & alu_y_addr_in`: Define the Register File address from which the operand values are to be read.
- `rd_addr_in & rd_write_in`: The former defines the Register File address to which the instruction result is to be written; the latter acts as the write enable for the Register File, since not every instruction produces a result to be written back.
- `immediate_in & shamt_in`: Define, respectively, the immediate and shamt values used by instructions such as those of I-Type.
- `mem_op_in & mem_size_in`: Define the type and size of the memory operation to be performed. For example, `LW` is described as a load operation of word size.
- `branch_in & funct3_in`: Define the type of control-flow operation, distinguishing and categorizing B-Type and J-Type instructions.

From the Writeback stage arrive the signals required to describe the result produced by the executed instruction. Here too, the signal `completed_count_instr` acts as an enable, indicating the cycle in which the corresponding values must be sampled. The remaining signals are the following:

- `completed_num`: Identifies the ROB row to which the incoming result refers.

- `completed_res`: Carries the result produced by the executed instruction.
- `completed_jump_taken`: Indicates whether the result corresponds to a taken jump.
- `completed_jump_target`: Defines the Program Counter value in the case of a taken jump.

Regarding the outputs, two signals are used to guarantee correct coordination between the ROB and the rest of the core. `fetch_enable` is asserted whenever the number of free rows in the table falls below a fixed margin, and is used to stall the Fetch stage, preventing the ROB from overflowing. `rob_empty` is directed to the Decode stage to indicate that the ROB currently holds no in-flight instruction.

The output ports prefixed with `execution_*` are directed towards the Execute stage. They carry the information required to correctly execute the instruction and share the same names as the corresponding inputs coming from the Decode stage, so they are not described again. The exception is the signal `execution_num`, which indicates the ROB row to which the instruction being executed corresponds. This allows the result to be written back to the correct entry upon completion without having to search for its position across the entire table.

The ports dedicated to Commit are connected to the Register File and to the Program Counter register in the Fetch stage, so that results can be written and the program flow altered as required. The Register File is connected to the signals `commit_rd_addr` and `commit_rd_write`, which define the address at which the value carried by `commit_res` must be stored. The Program Counter is managed through the signal `commit_jump_taken`, which indicates that its value must be updated with the one carried by `commit_jump_target`. The remaining signals, `commit_num`, `commit_op`, `commit_x_src`, `commit_y_src`, `commit_mem_op`, and `commit_mem_size`, are used exclusively to simplify diagnosis in the event of malfunctions. In the instantiation of the ROB, these signals are left *open*, informing the synthesizer that they have no physical connection and can therefore be safely eliminated.

4.1.1 Internal Structure

The ROB is structured as a table, in which each row corresponds to an instruction and each column represents one of the fields required to describe and manage that instruction. The fields needed to describe the instruction itself were determined by the pre-existing architecture of the Potato processor. It would be possible to modify them in order to improve performance, but this has been left as a subject for future improvements, further discussed in Chapter 6.

As explained above, the fields dedicated to describing the instruction are the following: `alu_op`, `alu_x_src`, `alu_x_addr`, `alu_y_src`, `alu_y_addr`, `rd`, `rd_write`, `imm`, `shamt`, `res`, `mem_op`, `mem_size`, `branch`, `funct3`, `jump_taken`, and `pc`.

The fields dedicated instead to managing the instruction's lifecycle are the following:

- `uses`: Defines whether the row is currently occupied by an instruction. A high value indicates occupation; otherwise it is zero.
- `p1 & p2`: Define whether the operands present true data dependencies with other instructions. `p1` refers to operand x (rs1), while `p2` refers to operand y (rs2).
- `exec`: Defines whether the instruction is currently in execution.
- `comm`: Defines whether the instruction is eligible for commit.

The natural progression of these flags consists of the assertion of `uses` at the moment an instruction is inserted into the ROB. Its operands are then compared against the other instructions present in the table, and `p1` and `p2` are asserted whenever no dependency is detected. When searching for the operation to be dispatched for execution, these two fields are inspected: if both are asserted, `exec` is asserted and the instruction is issued for execution. Once the operation completes, its result is stored and the `comm` flag is asserted. Once the operation is committed in program order, all fields are reset, making the row available again.

The last row of the ROB is hardwired to a fixed `NOP` value. This choice was adopted to prevent an incorrect instruction from being executed in the event that no instruction is currently eligible for dispatch.

Pointer	**Range**	**Role**
`pntr_start`	0 to `NUM_INSTRUCTIONS`-1	Tail: insertion point for the next decoded instruction.
`pntr_end`	0 to `NUM_INSTRUCTIONS`-1	Head: oldest in-flight instruction, the only commit candidate.
`pntr_nextend`	0 to `NUM_INSTRUCTIONS`-1	Next value of `pntr_end`, advanced as soon as the head instruction is eligible for commit.
`pntr_exec`	0 to `NUM_INSTRUCTIONS`	Slot currently dispatched to the Execute stage.
`pntr_nextexec`	0 to `NUM_INSTRUCTIONS`	Next slot selected for dispatch.
`pntr_exec_prev`	0 to `NUM_INSTRUCTIONS`	Previous value of `pntr_exec`, used to detect that a new instruction has been dispatched.

Table 4.1: Pointers used to manage the Reorder Buffer.

Table 4.1 summarizes the five pointers used to manage the table, whose individual roles are detailed in the remainder of this section. The ROB is structured as a circular buffer, thanks to the pointers `pntr_start` and `pntr_end`. The former acts as the tail, marking the insertion point for a newly decoded instruction, while the latter acts as the head, identifying the oldest in-flight instruction and therefore the candidate for commit. Both pointers range from zero to `NUM_INSTRUCTIONS` $- 1$, wrapping back to zero once the maximum value is reached. Each time a new instruction is loaded into the position addressed by `pntr_start`, the pointer is incremented circularly. Analogously, once the instruction addressed by `pntr_end` is committed, that pointer is incremented as well.

Within the body of the circular buffer, a third pointer, `pntr_exec`, moves from one position to another to identify the instruction currently selected for

execution. This pointer is driven by `pntr_nextexec`, which identifies the next instruction eligible for dispatch. Unlike the other two pointers, `pntr_nextexec` (and, consequently, `pntr_exec`) ranges up to the value `NUM_INSTRUCTIONS`, since it must be able to point to the fixed NOP sentinel row whenever no instruction is currently eligible for execution. In support of this mechanism, the signal `pntr_exec_prev` is used to guarantee that the same instruction is not dispatched for execution more than once, a situation that could otherwise arise whenever the same instruction remains the only one eligible for issue over consecutive cycles.

In the event of a flush or a reset, all pointers are reset to their initial value, so that the ROB can resume correct operation from a known state.

The ROB does not expose an input signal for flush, as it is capable of managing this process autonomously. Every time an instruction is committed, the `flush` signal is assigned the value held in `jump_taken[pntr_end]`. In this way, in the case of a taken jump, the ROB performs a flush of the entire table, since the validity of the subsequent instructions it contains can no longer be guaranteed. Conversely, when `jump_taken[pntr_end] = '0'`, the flush is not asserted, allowing the ROB to continue its normal operation without triggering any exception.

The ROB does not implement a branch predictor. When the Fetch stage encounters a branch instruction, it continues fetching from the next sequential address; no prediction of the outcome or target is made. Instructions fetched after the branch are decoded and inserted into the ROB as normal, forming a speculative tail: they may be dispatched to the execution unit and even complete out of order before the branch itself is resolved, but they cannot commit, because commit is strictly in-order and the branch sits ahead of them at a lower `pntr_end` value.

This strategy is correct by construction. Because no speculative instruction can modify the architectural register file before the branch commits, no state rollback is required in the case of a misprediction: a full ROB flush is sufficient.

4.1.2 Instruction Fetching

The process `input_instrs_managing` is responsible for storing incoming instructions. Whenever the signal `count_instruction` is asserted, the instruction is valid and is stored in the position indicated by `pntr_start`; otherwise, it is discarded and the pointer is not incremented.

This process is also responsible for resetting the entire row whenever the corresponding instruction is committed, or whenever a reset or a flush is asserted.

The process `p1p2_flags_managing` scans the entire table on every clock cycle in order to set `p1` and `p2` according to the presence of Read-After-Write (RAW) hazards. In this way, on every cycle the ROB maintains an up-to-date status for every instruction, guaranteeing correct execution.

The instruction located at `pntr_end` is always eligible for execution and cannot

be affected by a hazard. The scan therefore starts from the following position and, by means of a nested loop, iterates the check against all preceding instructions. The check is carried out in parallel for `p1` and `p2`, and their values are lowered whenever a hazard is detected.

A hazard check is also carried out for Load and Store operations. Whenever a memory operation shares the same base register and immediate value as a preceding memory operation, both flags are cleared, resulting in a conservative approach that guarantees correct program execution.

4.1.3 Issue and Commit

On every clock cycle, the ROB circularly scans the table starting from `pntr_end` in search of an instruction that is occupied (`uses='1'`), not yet in execution (`exec='0'`), and free of hazards (`p1='1'` and `p2='1'`). The index found is stored in `pntr_nextexec`, so that it is dispatched for execution in the following cycle. If no instruction satisfies these conditions, `pntr_nextexec` takes the value `NUM_INSTRUCTIONS`, causing a `NOP` to be executed instead.

An additional check is performed to guarantee that the instruction about to be dispatched differs from the one dispatched in the previous cycle, expressed as `pntr_exec` $\neq$ `pntr_exec_prev`. This ensures that the same operation is never issued for execution more than once.

In the case of memory operations, an additional condition must be satisfied. This originates from the fact that the ROB has no knowledge of the memory address targeted by a given operation, which could lead to a RAW hazard whenever two instructions reference the same memory location. Enforcing this condition allows out-of-order execution to proceed without compromising correctness.

The ROB detects every pair of back-to-back memory operations (Load-Load, Store-Store, Load-Store, and Store-Load): whenever `pntr_exec` and `pntr_nextexec` both point to memory operations, `multiple_load_op` is asserted for one cycle, freezing the update of both pointers. This introduces a controlled bubble that grants the memory bus the time required to complete the first operation before the second one is issued. This mechanism was introduced to ensure correct integration with the Potato core, since it is the Execute stage that is responsible for computing the memory address referenced by the instruction; by inserting a bubble, this value is preserved, guaranteeing correct access to the data cache.

On the rising clock edge where `completed_count_instr='1'` and `completed_num` $\neq$ `NUM_INSTRUCTIONS` (i.e., the completing instruction is not the NOP sentinel), four processes fire in parallel:

- `input_completed_instr_managing`: writes `completed_res` into `res[completed_num]`.

- `pc_values_managing`: overwrites `pc[completed_num]` with `completed_jump_target`. The `pc` field therefore serves a dual purpose: it holds the instruction's fetch address from insertion time, but is repurposed to carry the branch/jump target once execution completes. At commit time, this field is driven out as `commit_jump_target`.
- `jump_taken_flag_managing`: records `completed_jump_taken` into `jump_taken[completed_num]`.
- `comm_flag_managing`: sets `comm[completed_num] <= '1'`, marking the slot as ready to commit.

Setting `comm` does not immediately commit the instruction. The ROB enforces in-order commit: the slot at `pntr_end` — the oldest in-flight instruction — is the only eligible commit candidate. If `completed_num` $\neq$ `pntr_end`, the result simply waits in place until older instructions retire and `pntr_end` advances to that slot.

This decoupling between completion and commit is the core mechanism enabling out-of-order execution: instructions may complete in any order determined by data availability, yet the architectural state (register file, PC) is only updated in strict program order.

On every clock cycle, the instruction at the head, addressed by `pntr_end`, is evaluated for commit. If the `comm` field is asserted, the instruction is ready to be committed. Given the sequential nature of this logic, a naive implementation would commit instructions only once every two clock cycles, since the `comm` field is asserted only on the clock edge following the instruction's completion. To improve throughput, dedicated logic was integrated to also check whether the instruction has just arrived from the Writeback stage.

Consequently, an instruction may be committed not only when `comm='1'`, but also whenever `completed_num=pntr_end`. This avoids the introduction of a bubble, allowing an instruction to be committed on every clock cycle and enabling the pointer `pntr_nextend` — and, consequently, `pntr_end` — to advance without delay.

Given the presence of multiple processes managing the various fields of the ROB, an internal signal named `committing` was introduced to flag the occurrence of a commit at a given point in time. This allows the correct timing to be guaranteed when resetting fields that are no longer needed, while avoiding the overwriting of instructions that are still in use.

4.2 Out-of-Order Potato Architecture

The second step towards the desired architecture was to obtain an Out-of-Order processor. This was achieved by introducing a sixth pipeline stage, the ROB stage, positioned between Decode and Execute. This addition enables the management of out-of-order execution and the eventual commit of results to the register file.

Data forwarding was removed from the datapath in order to guarantee the correct computation of results. Since hazard management is now handled internally by the ROB through the `p1` and `p2` flags described in Section 4.1.1, it was no longer necessary to integrate that technique within the core. Forwarding was, however, retained for instructions belonging to the Zicsr extension. Since these instructions require strictly in-order execution, as discussed in Section 4.2.1, the technique was preserved for this specific path in order to improve the throughput of the core.

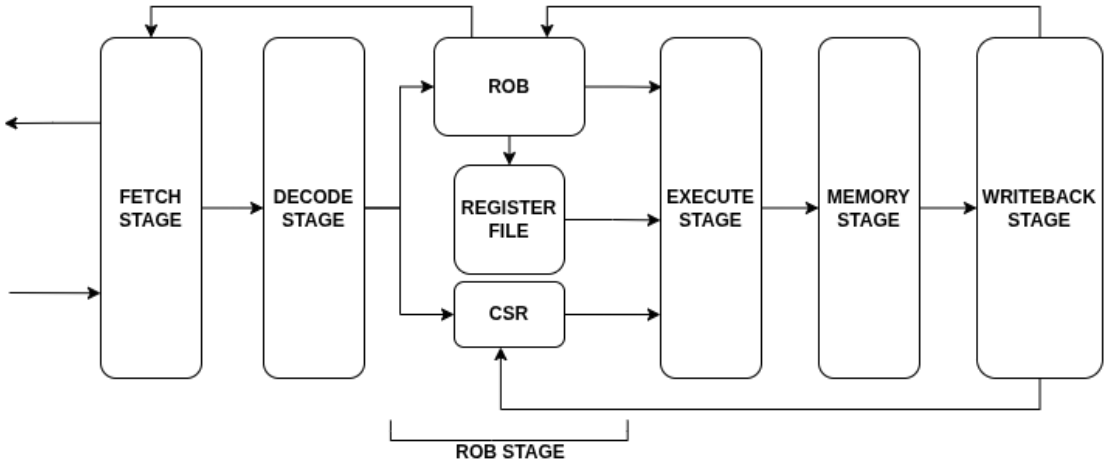

Figure 4.2: Out-of-Order Potato Architecture.

Realizing this architecture required several modifications to the original core, in order to obtain the separation between an in-order front-end, an out-of-order execution engine, and an in-order back-end, following the generic out-of-order model introduced in Section 3.4. The Reorder Buffer acts precisely as the bridge between these three domains.

The Decode stage is connected directly to the ROB, so that instructions are forwarded to it as soon as they are decoded. Once hazard checks have been performed, an instruction is selected, its operands are requested from the Register File, and the whole is then dispatched to the Execute stage for processing. Upon reaching the Writeback stage, the instruction and its result are sent back to the ROB and stored within it. Under normal circumstances the connection would instead be made directly with the Register File, but since the Register File constitutes the

in-order back-end, this is no longer possible: it is the ROB that, at the appropriate time, requests the write of a register into the Register File.

The same mechanism applies to J-Type and B-Type instructions. In this case, once the instruction has been executed following the flow just described, at commit time the ROB does not write to the Register File; instead, it updates the value of the Program Counter held in the Fetch stage.

4.2.1 CSR Management

As described previously, CSR instructions are responsible, among other things, for managing execution privileges. This means that they determine whether a given process, or part of it, is allowed to interact with specific regions of memory. From a security standpoint, this is a critical aspect, since mishandling it could lead to backdoors and fraudulent execution paths. It is therefore essential that these instructions be managed as safely as possible.

An out-of-order execution model would compromise this guarantee, since it could allow an instruction to access a data memory address before the corresponding privilege level has actually been updated. For this reason, a strictly in-order treatment of Control and Status Register (CSR) instructions was required.

As a consequence, a clear separation was introduced between the ROB and the execution of these instructions. Whenever a CSR instruction is handled, the ROB stage is effectively bypassed, and the architecture temporarily reverts to a five-stage pipeline: from Decode, execution proceeds directly to Execute, allowing the instruction to be processed without incurring the additional latency of the ROB.

To guarantee in-order execution, the Decode stage stalls the Fetch stage and waits until all previously decoded instructions have been executed and committed. This behavior is implemented as a small finite-state machine within the Decode stage, with three states: `normal_execution`, `rob_empty`, and `csr_execution`. A dedicated combinational block continuously inspects the incoming instruction to determine whether it belongs to the Zicsr extension. When a CSR instruction is detected and the ROB has not yet drained its in-flight instructions, the FSM moves to the `rob_empty` state, asserts `stall_csr` — which in turn stalls the Fetch stage — and inserts a bubble instead of forwarding the instruction. The signal `rob_empty`, exposed by the ROB and asserted only when its `uses` flags are entirely cleared, is monitored on every cycle while in this state. Once the ROB signals that it has been completely drained, the FSM transitions to the `csr_execution` state: the CSR instruction is issued directly to the Execute stage, marked through the dedicated signal `count_instruction_csr`, and consequently bypasses and is ignored by the ROB entirely. As shown in Figure 4.2, the Writeback stage is connected directly to the CSR unit, so that the corresponding registers can be updated without passing

through the Reorder Buffer.

Since consecutive CSR instructions cause the core to temporarily fall back to behaving like the original in-order Potato processor, it was decided to retain the forwarding paths for this specific case. This allows successive CSR instructions to execute without unnecessary stalls, resulting in faster execution.

4.2.2 Store and Load Management

The ROB is able to manage hazards arising from memory operations, as explained in Section 4.1.3; however, the original core architecture presents a crucial limitation. Once the address of a memory operation has been computed, it is held in the Execute stage and transmitted to the data memory. At this point, the memory may not yet be ready, requiring a stall. This causes the whole core to stall until the requested data becomes available, after which normal execution resumes. Whenever multiple consecutive load or store instructions are present, the architecture is structured so that the address of the second operation is kept in the Execute stage and sent directly from there to the data memory once the first operation has completed, making a stall strictly necessary in this case as well.

To guarantee correct execution, it was decided that the ROB itself should be capable of recognizing the presence of multiple consecutive load and store operations, so that it can stall accordingly. Had this not been implemented, an incorrect memory address could have been accessed: the intrinsically multi-stage nature of the pipeline does not allow the ROB to hold back the Execute stage during a stall condition originating further downstream, since doing so would require the Execute stage to begin computing a new operation before the first one has actually completed, resulting in the wrong address being issued. This is precisely the role fulfilled by the `multiple_load_op` mechanism described in Section 4.1.3, which detects back-to-back memory operations and freezes pointer advancement for one cycle until the first access completes.

A different approach that could be adopted instead is discussed in Chapter 6, dedicated to future work.

4.3 Simultaneous Multithreading Potato Architecture

The ROB developed in this thesis was designed to be as versatile as possible with respect to its employment in different architectures. Its internal structure and its ability to resolve hazards autonomously allow it to be integrated even into very simple cores, adding significant capability while requiring minimal external support. The previous section showed how a single ROB is sufficient to enable out-of-order execution; this section shows how employing more than one of them makes it possible to implement Simultaneous Multithreading.

Since this work constitutes an initial exploratory study, it was decided to realize a two-thread architecture, consisting of a main process and a second thread. By definition, threads possess their own private data space, which must not be shared with other processes. This requirement was translated into hardware by duplicating the Register File, so that no register storage is shared between the two threads.

The resulting architecture is shown in Figure 4.3.

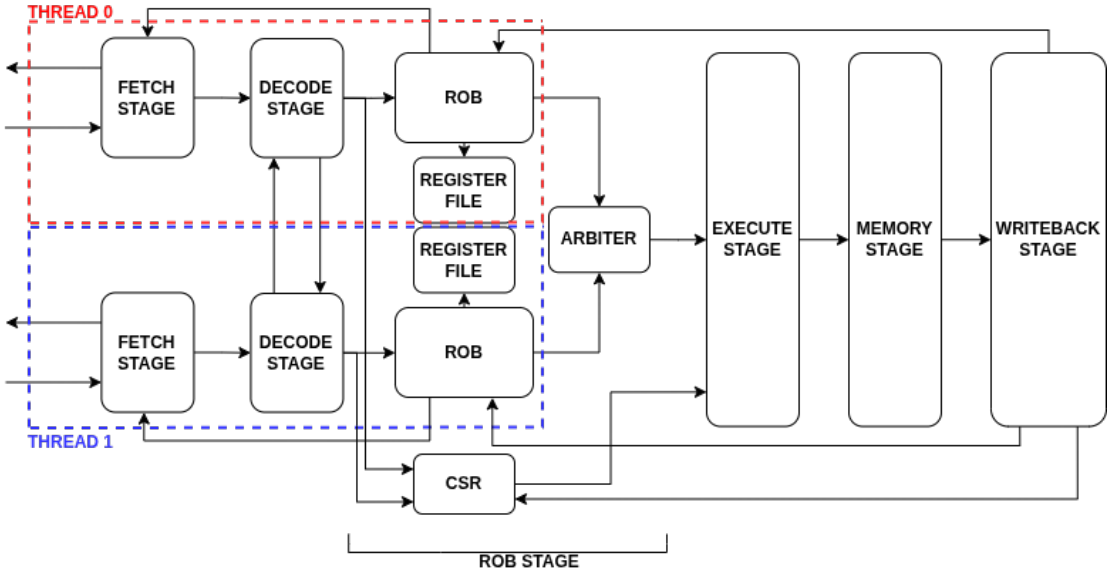

Figure 4.3: Simultaneous Multithreading Potato Architecture.

In an early stage of development, a version of the ROB capable of managing both threads within the same component was implemented. This approach was abandoned for two reasons: first, it would have restricted the module's reusability, since it could not have been employed in architectures featuring an odd number of threads; second, from a synthesis standpoint, a single larger component with

additional functionality would have resulted in a less favorable optimization outcome, increasing the overall area of the module.

The modifications made to the Potato processor mainly concern three stages: Fetch, Decode, and ROB. These are therefore discussed in detail in the following subsections, while the Execute, Memory, and Writeback stages did not undergo substantial changes, aside from the propagation of a second `count_instruction` signal, required so that the two ROBs can identify which thread the completed instruction returning from the Writeback stage belongs to.

4.3.1 Dual Fetch and Decode Frontend

As shown in the architecture of Figure 4.3, the Fetch and Decode stages were duplicated for simplicity of implementation. This choice implies the use of an instruction cache capable of exposing two independent read ports. A more general architecture, able to handle the case of a single-read-port instruction cache, is discussed in Chapter 6, dedicated to future work.

In this architecture, a modification was made to the Decode stage to handle the execution of CSR instructions. As discussed in Section 4.2.1, these operations must be executed in-order, and this guarantee must now additionally hold across threads, so that the two Decode units do not interfere with one another. The signal `ext_count_instruction_csr` was therefore added, cross-connecting the two Decode units: each one asserts it towards the other whenever it is executing a CSR instruction. Concretely, the finite-state machine of each Decode unit, once its own ROB has drained (`rob_table_empty = '1'`), also checks the value of `ext_count_instruction_csr` received from its counterpart before transitioning to the `csr_execution` state: if the other thread is currently executing a CSR instruction of its own, the requesting Decode unit remains in the waiting state until that execution completes.

This requirement follows from the fact that the CSR unit is unique and shared among all hardware threads of the core, since its role is to manage the state of the entire core rather than thread-local state. For this reason, both Decode units are connected to the same CSR unit.

4.3.2 Arbiter Design

The architecture developed in this thesis was designed to manage the sharing of execution resources without resorting to reservation stations, so as to significantly reduce the area overhead caused by the FIFOs of which they are typically composed. To achieve this, a separation was introduced that allows each unit to be realized independently, following architectures found in the literature that favor flexibility of employment. The shared Arithmetic Logic Unit (ALU) within the Execute stage

was structured to be able to service, in parallel, two instructions belonging to two of the following functional categories:

- **Arithmetic**: performs the arithmetic operations `ADD` and `SUB`, required, for instance, for address computation.
- **Logical**: performs the logical operations `AND`, `OR`, and `XOR`.
- **Shift**: performs the logical and arithmetic shift operations, such as `SLL`, `SRA`, and `SRL`.
- **Compare**: performs the comparison operations `SLT` and `SLTU`, required by B-Type instructions.

CSR instructions are not serviced by this shared ALU: they are executed on a dedicated CSR unit and, as described in Section 4.2.1, are guaranteed strictly in-order and conflict-free access to the Execute stage through a dedicated priority path, independently of the arbitration mechanism described below.

Since neither ROB is aware of what the other is about to execute, it was necessary to design a component capable of avoiding resource conflicts between the two threads. This role is fulfilled by a dedicated arbiter, designed according to the logic shown in Figure 4.4:

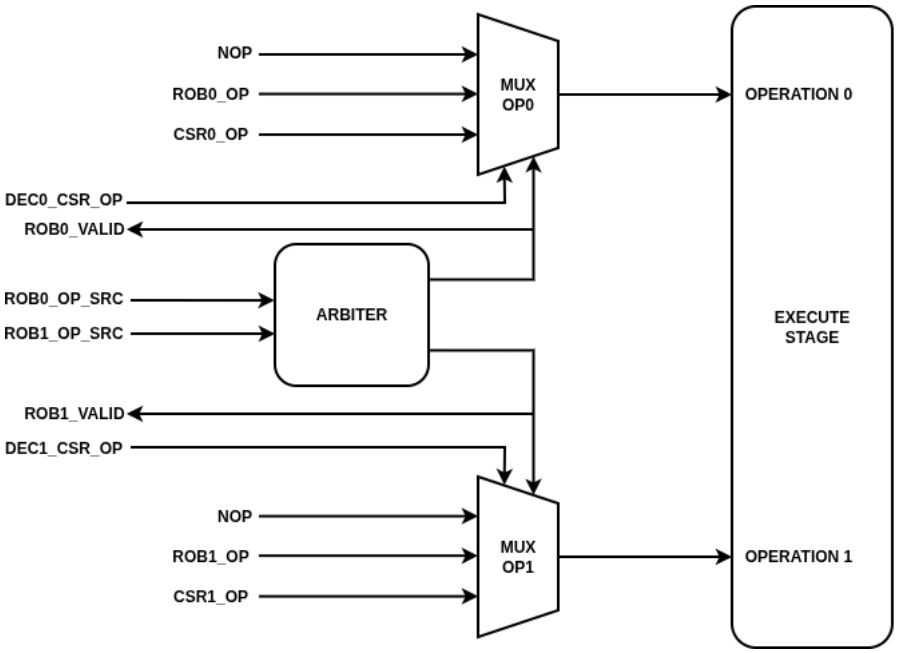

Figure 4.4: Circuit for managing resource hazards.

Its operation is illustrated below through two representative cases: the resolution of a CSR instruction issued by one thread, and the resolution of a conflict between two ordinary instructions.

In the case of a CSR instruction issued by thread 0, the Decode stage of that thread stalls and waits until its own ROB reports empty. Once this condition is satisfied, it verifies that Decode unit 1 is not itself executing a CSR instruction. The instruction is then issued, setting the selector `DEC0_CSR_OP` of `MUX OP0` so that the CSR instruction of thread 0 is executed. This grants priority to the CSR instruction, which is executed strictly in-order, entirely independently of the arbiter. Since the ROB of thread 0 has no new instruction to propose while this occurs, the arbiter observes a NOP coming from ROB0 and therefore grants uncontested execution priority to ROB1, allowing thread 1 to operate normally.

In the case where no CSR operation is present in either thread, the two Reorder Buffers will each attempt to issue an operation. The arbiter receives `ROB0_OP_SRC` and `ROB1_OP_SRC` — corresponding to the ALU operation each ROB proposes for execution — from which it internally derives the requested functional-unit category among the four listed above. The arbiter is then able to set the selectors of the multiplexers `MUX OP0` and `MUX OP1` according to whether a conflict exists. If no conflict is detected, both selectors are set so that the operations `ROB0_OP` and `ROB1_OP` are both forwarded to the Execute stage. If a conflict is detected, one of the two multiplexers instead outputs a `NOP`.

The signals `ROB0_VALID` and `ROB1_VALID` notify the corresponding ROB that the instruction it proposed for execution was not accepted, and that it must therefore propose it again on the following clock cycle. To prevent either thread from being permanently stalled by repeated conflicts, the two threads are guaranteed equal priority: on the first conflict, the instruction belonging to the ROB of thread 0 is executed; when a second conflict occurs, priority is instead granted to thread 1. This priority indicator toggles every time an actual resource conflict is detected between the two threads, guaranteeing a fair, round-robin resolution over time.

Chapter 5

Verification and Synthesis

Once the Out-of-Order and Simultaneous Multithreading extensions described in Chapter 4 had been designed and implemented, it remained to establish whether they behave correctly and what they cost in terms of achievable performance and occupied resources on FPGA. This chapter addresses both questions in turn.

The first part of the chapter, Section 5.1, describes how the developed architecture is simulated: the two testbenches used to exercise the core in isolation and as part of the complete System on Chip, the GHDL-based flow used for interactive, single-test debugging during development, and the independent Vivado-based flow used to run the complete regression suite of ISA compliance and local hazard tests against both testbenches. The outcome of this verification effort, together with the verification gaps that remain, is discussed in Section 5.1.4.

The second part, Section 5.2, turns to the physical implementation of the design on the Xilinx Zynq-7000 SoC FPGA mounted on the Pynq-Z2 board. The Reorder Buffer is first evaluated on its own, in Section 5.2.1, across a sweep of table sizes ranging from 8 to 16 entries, in order to characterize how its achievable frequency and occupied area scale with `NUM_INSTRUCTIONS`. The complete Potato core is then evaluated in Section 5.2.2 at a single, fixed Reorder Buffer size, comparing the in-order baseline against the Out-of-Order and Simultaneous Multithreading variants, in order to quantify the area and power cost that each architectural extension introduces with respect to a conventional in-order pipeline.

5.1 Testbenches

A testbench is a piece of hardware description code that does not correspond to any physical component of the final design; rather, it is written for simulation purposes only, in order to provide a controlled environment in which a Design Under Test (DUT) can be instantiated, driven with stimuli, and observed. Since the

testbench is never synthesized, it can make use of constructs that would otherwise be unsuitable for hardware implementation, such as file I/O, which is used here to load memory contents at the beginning of a simulation and, more generally, to reproduce the behavior of a real memory subsystem around the DUT.

Two testbenches are used to verify the developed architecture: `tb_processor.vhd` and `tb_soc.vhd`. The former instantiates the processor core alone, together with two idealized memories that model instruction and data storage as single-cycle, always-ready components. The latter instead instantiates the complete System on Chip (SoC), including the processor core and its memory-mapped peripherals. The use of `tb_soc.vhd` is essential, since it also accounts for the possible stalls introduced by memory and peripheral accesses, allowing for a more complete and realistic evaluation of the system's behavior than the one obtained from the core alone.

5.1.1 Simulation Flow

Simulation played a central role throughout the development of this thesis, as it is the primary means through which the propagation of data within the architecture, and the way it is handled by the various pipeline stages, can be observed and debugged. In order to keep this process fast and repeatable, a dedicated scripting infrastructure was developed around an open-source simulation toolchain, avoiding the need for a proprietary license during the iterative debugging phase of development.

The tool of choice for this purpose is GHDL, an open-source VHDL simulator supporting the VHDL-2008 standard used throughout the project. GHDL performs the syntactic analysis and elaboration of the VHDL sources describing both the design units and the testbenches, reporting any error or warning that could compromise the correctness of a subsequent simulation run. Within the `sim/` directory, a dedicated Makefile automates the sequence of Bash commands required to analyze, elaborate, and simulate either the processor core alone or the complete SoC, exposing simple targets that can be invoked to run a single test at a time.

For a testbench to run, it requires two binary files representing the initial contents of the instruction and data memories. The instruction memory file is generated starting from an assembly source file containing the routine used to exercise the component under analysis; together with a dedicated linker script, this allows the RISC-V GNU toolchain to produce an executable image that is subsequently converted into the plain-text hexadecimal format read by the testbench through VHDL `textio` constructs. The data memory file is instead produced directly, by writing the 32-bit words that are expected to be present in memory at the beginning of the simulation, whenever a test requires pre-initialized data.

Once a simulation completes, GHDL is able to generate a `.vcd` file containing the complete set of signal values recorded during the run. This waveform is then

inspected using GTKWave which, through a dedicated TCL script, automatically opens with a predefined layout showing the signals most relevant to the analysis and diagnosis of the simulated program. This setup proved particularly valuable during the debugging of one failing test at a time, as it avoids the need to manually search for and add the relevant signals to the viewer on every run.

5.1.2 Simulation Tests

Any single test — whether drawn from the ISA compliance suite, from the local regression tests, or written ad hoc purely to observe a specific behavior during development — can be simulated in isolation through this flow by setting the `TEST` variable exposed by the `sim/` Makefile to the desired assembly source and invoking the `run_test` (or `run_soc_test`) target, which then triggers the GTKWave inspection step described above.

Figure 5.1 shows a representative trace obtained through this same flow, illustrating how the Reorder Buffer interacts with the Decode stage while fetching a short, hand-written instruction sequence, not part of the automated regression suite. To keep the resulting waveform compact and readable, the Reorder Buffer was reduced to `NUM_INSTRUCTIONS=4` entries for this example, which exercises the following sequence:

```
SRLI x2, x2, 12
 ADD x3, x1, x2
 ADDI x4, x0, 1
 SUB x5, x3, x1
 AND x6, x3, x4
   LI x14, -72
SRA x15, x11, x4
```

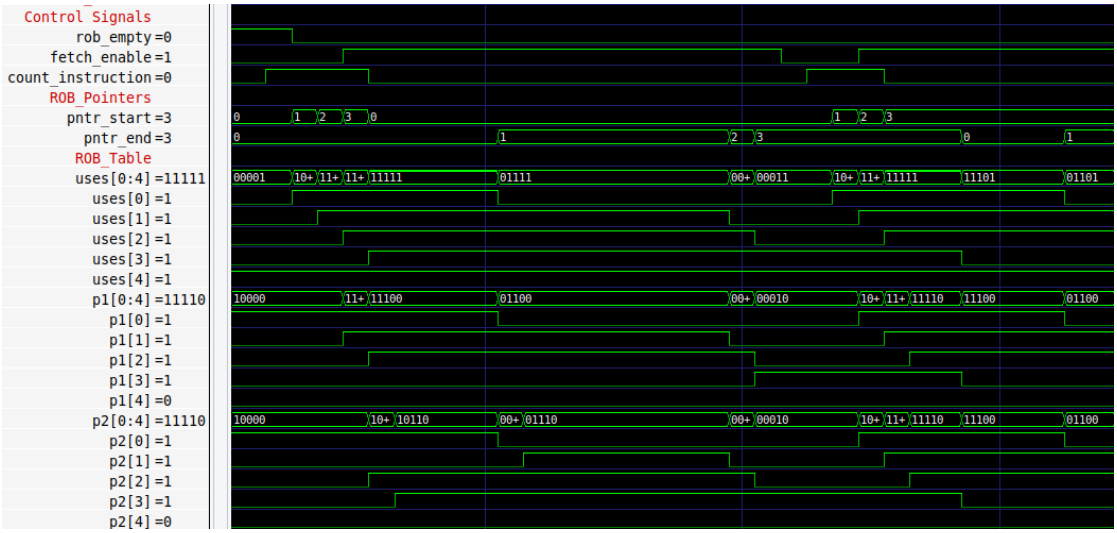

Figure 5.1: Instruction Fetching Example.

At the beginning of the trace the table is empty, `rob_empty` is asserted, and `fetch_enable` is low, signaling the Decode stage that it may keep supplying new instructions. As `count_instruction` pulses for each instruction accepted from Decode, `pntr_start` advances circularly through the four available slots described in Section 4.1.1, wrapping back to 0 once the table has been filled. Because only four entries are available in this reduced example, the table approaches capacity after comparatively few instructions, and `fetch_enable` is raised to stall Fetch and Decode until enough slots are freed again by commit; `pntr_end` advances in step with these commits, each retiring instruction clearing its `uses` bit and making its slot available for a subsequent insertion.

A Read-After-Write hazard is visible on the second instruction, which reads `x2` immediately after the first instruction has written it: the Reorder Buffer detects the dependency through the `p1p2_flags_managing` process and clears `p2` for that slot, preventing the instruction from being selected for dispatch until the hazard clears. As expected, `p1` and `p2` at the slot addressed by `pntr_end` remain 1 throughout, since the oldest in-flight instruction can never be blocked by a hazard against an even older one.

This interactive, single-test flow was the primary tool used throughout the development of the architecture described in Chapter 4. It is complementary to, rather than a substitute for, the automated regression suite discussed next: the latter establishes whether the architecture is correct across its entire targeted instruction set, while the workflow illustrated here is what makes it possible to understand, at the signal level, why a specific test passes or fails.

Figure 5.2 shows a second representative trace illustrating how the Decode stage

and the Reorder Buffer cooperate to guarantee the in-order execution of Control and Status Register (CSR) instructions, as described in Section 4.2.1. The sequence exercised is:

```
ADD x3, x1, x2
ADDI x4, x0, 1
SUB x5, x3, x1
LI x10, 100
CSRW mscratch, x10
CSRRWI x11, mscratch, 2
CSRRW x12, mscratch, x11
SRLI x2, x2, 12
ADD x3, x1, x2
```

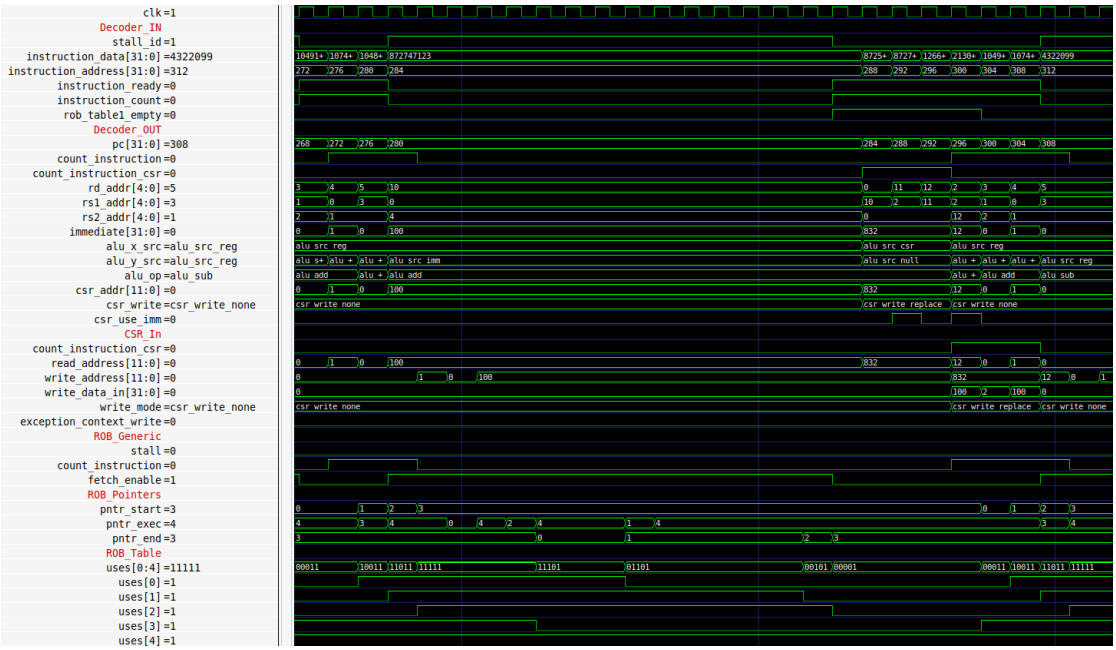

Figure 5.2: CSR Instructions Management Example.

For the first three instructions, which are ordinary register-register and register-immediate operations, Decode behaves exactly as in the previous example, accepting and inserting each of them into the Reorder Buffer as soon as it is fetched. The table, already holding instructions retired prior to the start of this trace, reaches its full four-slot capacity once `ADDI x10, x0, 100` — the expansion of the `LI` pseudo-instruction, fetched from address 280 — is inserted. At this point `fetch_enable` is raised, exactly as described for the ordinary buffer-full case above: the Fetch stage

freezes at address 284, the address of the following `CSRW` instruction, and Decode stops accepting new instructions.

What distinguishes this trace from the earlier, purely arithmetic example is how long the resulting stall lasts. Rather than clearing as soon as a couple of slots are freed again, it persists for roughly 150 ns — about fifteen clock cycles — while the Reorder Buffer commits its five in-flight instructions one after another, and only clears once `rob_table1_empty` asserts, together with the Reorder Buffer's own `rob_empty`, indicating that every older instruction has been committed and the table is completely empty. This is the behavior described in Section 4.2.1: because a CSR instruction bypasses the Reorder Buffer and is issued directly to the Execute stage, Decode must first drain every older, still in-flight instruction before it is allowed to proceed, so that the CSR instruction can never be observed to execute out of order with respect to any instruction preceding it in the program. This is a stricter condition than the ordinary buffer-full stall illustrated earlier, which only requires a couple of slots to free up again, and the difference in stall length between the two traces is a direct, observable consequence of that stricter requirement.

Once the table reports empty, Decode admits the three CSR instructions in immediate succession, one per clock cycle and with no further stalling between them: `CSRW mscratch, x10` at address 284, whose decoded `csr_addr` field reads 832 (`0x340`, the architectural address of `mscratch`) and whose destination register is `x0`, consistent with its expansion into `CSRRW x0, mscratch, x10`; `CSRRWI x11, mscratch, 2` at address 288, whose five-bit source register field is reused by the decoder to carry the literal immediate value 2, as dictated by this instruction's encoding; and `CSRRW x12, mscratch, x11` at address 292. Throughout these three cycles, `count_instruction_csr` remains asserted in Decode while its ordinary `count_instruction` signal, and the Reorder Buffer's own `count_instruction` input, remain low, directly confirming that none of the three CSR instructions is ever inserted into the table: they are issued, executed, and committed entirely outside the Reorder Buffer's bookkeeping, through the dedicated CSR path described in Section 4.2.1.

Normal operation resumes as soon as Decode reaches the first non-CSR instruction following the sequence, `SRLI x2, x2, 12` at address 296: `count_instruction` reasserts on both Decode and the Reorder Buffer, and `rob_table1_empty` deasserts one cycle later as the table accepts this new entry, marking the return to ordinary out-of-order dispatch for the instructions that follow.

Figure 5.3 shows a third representative trace, illustrating how the Reorder Buffer and the Execute stage cooperate to handle multiple, consecutive load and store instructions, the mechanism described in Section 4.2.2. Rather than exercising a single pair of adjacent memory instructions, the sequence is organized as eight short, independent groups, each isolated from the next by an `ADDI x5, x0, N` marker instruction (omitted from Table 5.1 for brevity) that both breaks adjacency

in the Reorder Buffer and identifies the group in the waveform through the value of its immediate. `x3` and `x4` are first loaded with two distinct addresses, 4096 and 4224 respectively, and `x1` and `x2` with two arbitrary store values, so that every subsequent load or store can be written compactly:

```
LI x3, 4096   LI x4, 4224   LI x1, 111   LI x2, 222
```

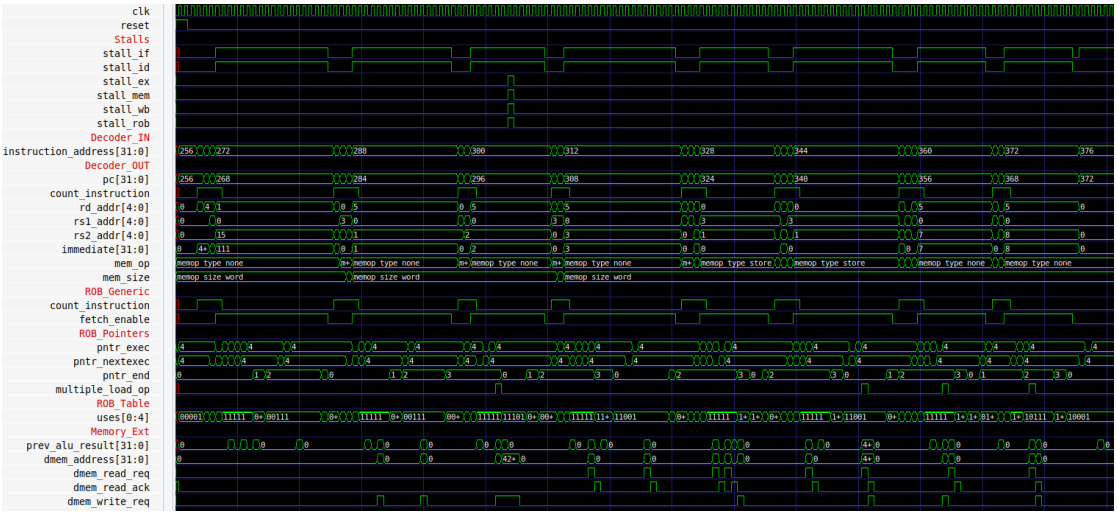

Figure 5.3: Load and Store Instructions Management Example.

For each group, the trace was inspected for three things: the number of clock cycles separating the two instructions `dmem_read_req`/`dmem_write_req` pulses at the Execute/Memory interface, whether `multiple_load_op` was asserted, and whether any core-wide stall (`stall_ex`, `stall_mem`, `stall_wb`, `stall_rob`, all four of which always pulse together) occurred. Table 5.1 reports both the instruction pair exercised by each group and the corresponding result.

Grp	Instr1	Instr2	Addr	Gap	`mult_load_op`	Stall
1	`SW x1,0(x3)`	`SH x2,0(x3)`	same	7 cyc.	no	no
2	`SW x1,0(x3)`	`SW x2,0(x4)`	diff	1 cyc.	yes	1 cyc.
3	`LW x11,0(x3)`	`LH x12,0(x3)`	same	9 cyc.	no	no
4	`LW x11,0(x3)`	`LW x12,0(x4)`	diff	2 cyc.	no	no
5	`SW x1,0(x3)`	`LW x11,0(x3)`	same	11 cyc.	no	no
6	`LW x11,0(x3)`	`SW x1,0(x3)`	same	1 cyc.	yes	no
7	`SW x1,0(x3)`	`LW x11,0(x4)`	diff	1 cyc.	yes	no
8	`LW x11,0(x3)`	`SW x1,0(x4)`	diff	1 cyc.	yes	no

Table 5.1: Behavior of the Reorder Buffer and Execute stage across all combinations of two consecutive load/store instructions, with matching or differing target addresses.

The pattern in Table 5.1 is sharper than either the target address or the combination of instruction types: `multiple_load_op` asserts in exactly the four groups where the Reorder Buffer actually attempts to dispatch the two memory instructions on immediately consecutive clock cycles (Groups 2, 6, 7, and 8, each with a one-cycle dispatch gap), and never asserts otherwise — including Group 4, whose two-cycle gap is close but not adjacent, and Group 1, whose pair targets the same address with the same combination of store types that triggers the mechanism in Group 2, yet is separated by seven cycles and never engages it. Neither same-address aliasing nor a specific pairing of instruction types is, by itself, sufficient or necessary to trigger `multiple_load_op` in this trace; what the four triggering groups have in common is purely that the Reorder Buffer is about to place two memory operations back to back at the Execute stage, regardless of whether they target the same address or different ones, or whether they are two stores, two loads, or one of each.

This matches precisely the condition described in Section 4.1.3: `multiple_load_op` is asserted whenever `pntr_exec` and `pntr_nextexec` both point to memory operations, that is, exactly when the Reorder Buffer is about to dispatch two memory instructions on consecutive cycles — the same condition observed here as a one-cycle dispatch gap — and Section 4.1.3 explicitly states that the mechanism covers all four combinations, Load-Load, Store-Store, Load-Store, and Store-Load, consistent with Groups 2, 6, 7, and 8 covering exactly these four combinations between them. The address-independence observed in the table is likewise not a coincidence but a direct consequence of a limitation stated explicitly in Section 4.2.2 and Section 4.1.3: at the point where this check is performed, the Reorder Buffer has no knowledge of the memory address targeted by either instruction, since

address computation is the responsibility of the Execute stage and has not yet taken place. Lacking this information, the mechanism cannot distinguish a pair that would actually alias, such as the same-address stores of Group 1 had they been adjacent, from one that provably would not, such as the different-address pair of Group 2; it must instead treat every back-to-back memory pair conservatively, which is exactly what Groups 2, 6, 7, and 8 show it doing regardless of address. How often two memory operations actually end up adjacent enough to trigger this conservative check for a given program depends on the surrounding instruction mix and on the Reorder Buffer's occupancy at the time, which is why otherwise similar pairs, such as Groups 1 and 2, can end up on opposite sides of the same table.

The four triggering groups do not, however, all cost the same. Group 2, the only pair of two consecutive *stores*, is also the only one that costs a core-wide stall cycle, and correspondingly the only one whose combined `dmem_write_req` window extends to four cycles rather than resolving in the minimum two; Groups 6, 7, and 8, each pairing one load with one store, resolve in exactly two clock cycles — one `dmem_read_req` pulse immediately followed by one `dmem_write_req` pulse, or vice versa — with no stall anywhere in the pipeline. This asymmetry is consistent with the read side of the data memory interface completing combinationally within the same cycle it is requested, as already observed for the isolated loads in Section 5.1.2: a read can be immediately followed by a write, or a write immediately followed by a read, without forcing the core to wait, whereas two writes contending for the same single-ported memory interface on consecutive cycles cannot both be accepted without one of them stalling. The `multiple_load_op` mechanism is accordingly better understood as guaranteeing correct *sequencing* of back-to-back memory operations at the Execute stage in general, while the cost of that sequencing — whether it requires an actual stall cycle or is absorbed for free — depends specifically on whether the pair being serialized are both stores.

5.1.3 Verification Tests

While the GHDL-based flow described above is the preferred tool for interactively debugging a single test, the complete regression suite is executed through a second, independent flow based on the Xilinx Vivado toolchain. A top-level Makefile drives the `xelab` and `xsim` tools to compile and elaborate the testbenches together with the entire set of tests, and to run them in sequence, scraping the pass/fail outcome of each test from the simulator's console report. Two targets are provided for this purpose: one exercises every test against `tb_processor.vhd`, verifying the core in isolation, while the other exercises the same set of tests against `tb_soc.vhd`, additionally validating the interaction between the core and the memory-mapped peripherals. A `-vcd` variant of each target is also available, which additionally dumps a full-signal waveform for every test executed, at the cost of significantly

longer runtimes; this variant is useful when a regression failure needs to be inspected offline, without having to reproduce it first through the GHDL flow.

Two categories of tests are used to exercise the architecture. The first is a suite of Instruction Set Architecture (ISA) compliance tests, one for each instruction defined by the RV32I base integer instruction set: arithmetic and logical operations, both register-register and register-immediate; every conditional branch; the two jump instructions; loads and stores of every supported width; the upper-immediate instructions; and the shift and comparison operations. These tests, adapted from the upstream `riscv-tests` project, are self-checking: each one exercises a specific instruction with a range of operand values chosen to expose common implementation mistakes, comparing the obtained result against the expected one and signaling a failure as soon as a mismatch is detected. Taken together, they exercise every opcode that the architecture must correctly fetch, dispatch, execute, and commit.

The second category consists of a small set of local, hand-written tests targeting corner cases that are specific to the developed architecture rather than to the RISC-V ISA in general. The relevant example is `csr_hazard.S`, which issues sequences of back-to-back `csrw` and `csrr` instructions on the `mscratch` register, with an increasing number of writes performed before the final read. Since Control and Status Register (CSR) instructions do not operate on the general-purpose register file, they are not covered by the ordinary data-hazard resolution exercised by the ISA compliance suite; this test instead specifically verifies that a read correctly observes the value produced by the most recent preceding write to the same register, regardless of how many writes precede it.

Both categories of tests share the same pass/fail signaling mechanism, which is specific to this architecture rather than being inherited from the upstream `riscv-tests` convention. The original `riscv-tests` suite signals completion through an `ecall` trap and a `tohost`/`fromhost` memory word; this mechanism is still present as unused scaffolding in the adapted `riscv_test.h` header, but it is not the one actually used to determine the outcome of a test. Instead, a dedicated CSR, `POTATO_TEST_CSR`, located at address `0xbf0`, is used to encode both a test identifier and one of four states: idle, running, failed, or passed. The macros `POTATO_TEST_START`, `POTATO_TEST_PASS`, and `POTATO_TEST_FAIL`, defined in `potato-test.h`, write this encoding to the CSR at the beginning of a test and upon reaching a passing or failing condition, respectively. On the hardware side, the CSR unit decodes writes to this address into a dedicated `test_register` signal, which is exposed through the `test_context` record, threaded from the CSR unit up to the top level of the core and, from there, to the testbench. The testbenches poll the `state` field of this record and report either `"Success!"` or `"Failure in test N!"` once it reaches the `TEST_PASSED` or `TEST_FAILED` value.

Because this signaling mechanism relies on an ordinary CSR write, retired through the pipeline like any other instruction, a passing test implicitly validates

more than the specific instruction it targets: it also confirms that CSR writes are correctly committed in program order and that their architectural effect is correctly exposed, rather than only checking that arithmetic and branch results are computed correctly.

Every test follows the same build pipeline before being simulated. The assembly source is first compiled with the RISC-V GNU toolchain, targeting the `rv32i_zicsr` architecture, and linked against a dedicated linker script that places the code segment at address `0x100`, the data segment at address `0x1000`, and the stack pointer at address `0x2000`, matching the default reset and memory configuration assumed by both testbenches. The resulting ELF executable is then disassembled with `objdump` by a dedicated script, which extracts the instruction and data segments into the two separate hexadecimal files consumed by the testbenches at the beginning of each simulation.

5.1.4 Results

The regression suite of 37 tests described in Section 5.1.3 — the 35 RV32I ISA compliance tests, the `simple` bring-up program, and the local `csr_hazard` regression — passes with zero failures against the Out-of-Order core, each test reporting `"Success!"` as described above. Beyond confirming correctness, every run also reports the number of elapsed clock cycles and the number of retired instructions, which makes it possible to compare, test by test, the cycles-per-instruction (CPI) figure achieved by the Out-of-Order core against the original In-Order baseline it extends. Tables 5.2, 5.3, 5.4, and 5.5 report this comparison, grouping the 37 tests respectively into Arithmetic and Logic (ALU) instructions, Branch and Jump instructions, Memory (load and store) instructions, and a final, Miscellaneous and System category collecting the `simple`, `auipc`, `lui`, and `csr_hazard` tests, none of which fits cleanly into the first three groups.

	In-Order			OoO		
Test	**C.C.**	**N.Instr.**	**CPI**	**C.C.**	**N.Instr.**	**CPI**
add	1941	486	3.99	2638	556	4.74
addi	1116	263	4.24	1435	291	4.93
and	1942	506	3.84	3032	574	5.28
andi	904	219	4.13	1292	246	5.25
or	1972	509	3.87	3093	573	5.40
ori	930	226	4.12	1359	256	5.31
sll	2064	514	4.02	2718	589	4.61
slt	1971	480	4.11	2563	553	4.63
slti	1112	258	4.31	1431	287	4.99
sltiu	1112	258	4.31	1431	287	4.99
sltu	1971	480	4.11	2563	553	4.63
sra	2233	533	4.19	2823	610	4.63
srai	1191	277	4.30	1544	311	4.96
srl	2132	527	4.05	2838	600	4.73
sub	1903	478	3.98	2600	548	4.74
xor	1959	508	3.86	3065	573	5.35
xori	947	228	4.15	1387	253	5.48

Table 5.2: CPI comparison between In-Order and Out-of-Order architectures for Arithmetic and Logic (ALU) instructions.

Signed and unsigned variants of the same comparison, `slt`/`sltu` and `slti`/`sltiu`, retire an identical number of cycles and instructions in both architectures, as expected since the two only differ in the sign interpretation of the comparison rather than in control flow.

	In-Order			OoO		
Test	**C.C.**	**N.Instr.**	**CPI**	**C.C.**	**N.Instr.**	**CPI**
beq	1272	312	4.08	1945	374	5.20
bge	1392	330	4.22	2160	407	5.31
bgeu	1472	355	4.15	2362	433	5.45
blt	1272	312	4.08	1953	374	5.22
bltu	1352	337	4.01	2176	402	5.41
bne	1273	312	4.08	2016	377	5.35
jal	384	76	5.05	508	78	6.51
jalr	587	129	4.55	891	160	5.57

Table 5.3: CPI comparison between In-Order and Out-of-Order architectures for Branch and Jump instructions.

The `beq` and `blt` tests likewise retire an identical number of cycles and instructions in the In-Order core, and an identical number of instructions in the Out-of-Order core as well, reflecting the fact that both exercise symmetric taken/not-taken branch patterns of the same length.

	In-Order			OoO		
Test	**C.C.**	**N.Instr.**	**CPI**	**C.C.**	**N.Instr.**	**CPI**
lb	1134	476	2.38	1724	335	5.15
lbu	1134	476	2.38	1724	335	5.15
lh	1175	447	2.63	1789	328	5.45
lhu	1200	498	2.41	1805	332	5.44
lw	1236	420	2.94	1877	328	5.72
sb	1851	887	2.09	2976	597	4.98
sh	2045	923	2.22	3427	660	5.19
sw	2051	928	2.21	3550	688	5.16

Table 5.4: CPI comparison between In-Order and Out-of-Order architectures for Memory (Load and Store) instructions.

Unlike the two categories above, the Memory tests depart sharply from the roughly 25% CPI increase observed so far — a discrepancy examined in detail below.

	In-Order			OoO		
Test	**C.C.**	**N.Instr.**	**CPI**	**C.C.**	**N.Instr.**	**CPI**
simple	276	62	4.45	386	63	6.13
auipc	375	80	4.69	498	83	6.00
lui	407	86	4.73	528	87	6.07
csr_hazard	328	71	4.62	532	91	5.85

Table 5.5: CPI comparison between In-Order and Out-of-Order architectures for Miscellaneous and System instructions.

Despite being the only test in this group to exercise the CSR bypass path of Section 4.2.1 rather than ordinary Reorder Buffer dispatch, `csr_hazard` shows a CPI increase broadly in line with the rest of the group.

Across the Arithmetic/Logic, Branch/Jump, and Miscellaneous/System categories, CPI grows from a fairly narrow In-Order range of 3.84–4.31, 4.01–5.05, and 4.45–4.73 respectively, to a wider Out-of-Order range of 4.61–5.48, 5.20–6.51, and 5.85–6.13; averaged over these 29 tests, CPI increases by roughly 25% overall, though the increase varies more from test to test than a fixed per-test overhead would produce — from about 10% for `sra` to about 40% for `or` — and is somewhat smaller on average for Arithmetic/Logic instructions (around 22%) than for Branch/Jump and Miscellaneous/System ones (around 29% and 30% respectively). This pattern is still informative when read against the specific way the Out-of-Order core departs from the In-Order baseline, discussed in Section 4.2: the Reorder Buffer is introduced as a sixth pipeline stage between Decode and Execute, and general operand forwarding is removed from the datapath entirely, retained only for the Zicsr path of Section 4.2.1. Were the extra ROB stage the dominant cause of the CPI increase, its cost would be expected to behave as a fixed, one-time pipeline fill/drain overhead per test run, amortized over the number of instructions executed, and therefore to produce a *larger* relative CPI penalty on short tests than on long ones; instead, the additional cycles introduced by the Out-of-Order core track the instruction count of each test reasonably closely — for the Arithmetic/Logic category, for instance, the additional cycles per retired instruction range between 1.11 and 2.20 regardless of whether the test retires 219 or 533 instructions — which is closer to the signature of a recurring, per-instruction cost than to a one-time one. This is consistent with the removal of general forwarding being a major contributor:

without it, a RAW-dependent instruction can no longer receive its operand as soon as it is computed, and must instead wait in the Reorder Buffer until the producing instruction has been committed and written back to the Register File, as described in Section 4.1.1, incurring a comparable stall on essentially every dependent instruction rather than only once per test. The additional pipeline stage contributed by the Reorder Buffer is accordingly better understood as a structural precondition for Out-of-Order execution — and a comparatively minor, amortized cost in itself — than as the sole source of the CPI figures reported above, though the test-by-test variation noted here suggests that instruction mix also plays a role that a purely per-instruction model does not fully capture.

The Memory category in Table 5.4 does not fit this otherwise consistent picture and deserves separate comment. In-Order CPI for these tests, 2.09–2.94, is markedly lower than for any other category, while Out-of-Order CPI, 4.98–5.72, remains broadly in line with the rest of the suite; the resulting CPI increase, averaging 121% across the eight memory tests, is far larger than the roughly 25% observed elsewhere. At the same time, the number of instructions reported for the In-Order run of these tests is consistently higher — from 28% for `lw` up to 50% for `lhu` — than for the Out-of-Order run of the identical test source, the opposite of the smaller, more moderate increase seen in the other three categories, which suggests that the two architectures are not counting retired instructions on an equal basis specifically for multi-cycle memory accesses. Rather than treat the Memory category's CPI increase as evidence of a substantially larger Out-of-Order penalty for loads and stores, it is more appropriate to treat it as an open measurement inconsistency: identifying and correcting the source of the instruction-count mismatch between the two testbenches for memory operations was not carried out as part of this work, and the Arithmetic/Logic, Branch/Jump, and Miscellaneous/System figures, which agree more closely with one another, should be regarded as the more reliable characterization of the structural CPI cost introduced by the Out-of-Order extension.

Taken together with the pass results reported above, these observations confirm that the Out-of-Order core correctly implements the targeted subset of the RV32I instruction set together with the Zicsr extension, and characterize the CPI cost that Out-of-Order execution adds over the In-Order baseline. Some verification gaps remain, however, and are worth highlighting rather than treated as fully closed.

The regression suite described above is entirely composed of directed, hand-written tests: it verifies that each instruction behaves correctly in isolation, or in the specific hazard configuration targeted by a local test, but it does not amount to a systematic exploration of the space of possible instruction interleavings and hazard combinations that the Reorder Buffer may encounter over long-running programs. A constrained-random or coverage-driven verification approach, capable of automatically generating and checking large numbers of instruction sequences, was

not implemented in this work and would provide a stronger correctness guarantee, particularly regarding hazards that only manifest under specific timing conditions.

The scope of the ISA compliance suite is also narrower than the full set of tests distributed upstream: the `riscv-tests` sources include a test for the `fence.i` instruction, `fence_i.S`, which is present in the `riscv-tests/` directory but deliberately left out of the set of tests actually compiled and run. This is a consequence of scope rather than an oversight, since `fence.i` belongs to the Zifencei extension, which is not part of the RV32I plus Zicsr subset targeted by this work; its exclusion is nonetheless worth recording explicitly, both to document precisely which upstream tests were left out and why, and because instruction-fetch/memory coherence hazards of the kind `fence.i` is designed to expose are a natural candidate for future extension of the verified instruction set, a point returned to in Chapter 6.

Additionally, dedicated testbenches exist for several individual peripherals of the SoC, such as the GPIO, UART, timer, memory, and interconnect blocks; these are, however, meant to be run manually during the development of the corresponding peripheral and are not wired into either automated regression target, so they do not contribute to the pass/fail figures reported above.

Finally, it is worth estimating the results that the Simultaneous Multithreading variant would be expected to achieve, extending the single-thread CPI comparison of Tables 5.2–5.5 to the dual-thread case. The design of the SMT arbiter described in Section 4.3.2 provides the basis for this estimate: the shared Arithmetic Logic Unit can service one instruction from each thread in the same cycle whenever the two proposed operations fall into different functional categories — arithmetic, logical, shift, or compare — and stalls only one of the two threads, for a single retry cycle, when both simultaneously request the same category, with a round-robin priority indicator guaranteeing that neither thread can be repeatedly starved by the other.

A single thread running alone on the SMT core, with the second thread idle, is expected to reproduce the single-threaded Out-of-Order CPI figures of Tables 5.2–5.5 essentially unchanged, since it never contends for the shared Execute stage. With both threads simultaneously active and issuing continuously, a first-order estimate of the additional per-thread cost can be obtained from the arbiter's four mutually exclusive ALU categories: treating the two threads' instruction streams as uncorrelated and each category as roughly equally likely — a simplification, since address computation makes the arithmetic category somewhat more frequent in practice — a same-category collision would be expected on the order of one cycle in four when both threads are ALU-bound, each costing the losing thread a single retry cycle under the fairness guarantee described above. This places the expected per-thread CPI increase over the single-threaded Out-of-Order baseline at roughly 20–25% in this fully-contended worst case, comparable in magnitude to the roughly 22% increase already measured for Arithmetic/Logic instructions

when moving from the In-Order baseline to the single-threaded Out-of-Order core, and far short of a naive doubling. Aggregated across both hardware threads, the combined instruction throughput of the SMT core would accordingly be expected to approach, though remain somewhat below, twice that of a single Out-of-Order thread — the improvement Simultaneous Multithreading is designed to deliver, at the area and power cost already quantified in Section 5.2.2. This estimate is derived analytically from the arbiter's documented behavior rather than from direct measurement, and should be read as a first-order prediction to be confirmed by future dual-thread benchmarking, rather than as a substitute for it.

5.2 Synthesis and Implementation

Once functional correctness had been verified through simulation, the architecture was synthesized and implemented in order to evaluate its physical characteristics on FPGA. The target device is the Xilinx Zynq-7000 SoC FPGA mounted on the AMD Pynq-Z2 development board, using the Xilinx Vivado toolchain for synthesis, placement, and routing.

Rather than synthesizing the Reorder Buffer as a standalone, disconnected module, the results were obtained by integrating it into a complete Vivado block design. The `rob_wrapper.vhd` entity, already introduced in Section 4.1.1 as the component that exposes the Reorder Buffer's ports using only plain `std_logic` and `std_logic_vector` types, was packaged as a custom IP and instantiated alongside the Zynq Processing System (PS7) block and a set of AXI GPIO IP cores. The wrapper's inputs and outputs were connected to the parallel I/O channels exposed by the AXI GPIO cores, which in turn communicate with the Processing System through the AXI interconnect automatically generated by the block design. This arrangement allows the Processing System to drive stimulus into the Reorder Buffer and to read its outputs back through ordinary memory-mapped GPIO register accesses. Beyond making the module individually testable on hardware, this also prevents the synthesis tool from optimizing away logic connected to unused ports, which would otherwise misrepresent the resource utilization and achievable frequency of the component evaluated in isolation. The Block Design used is shown in Figure 5.4:

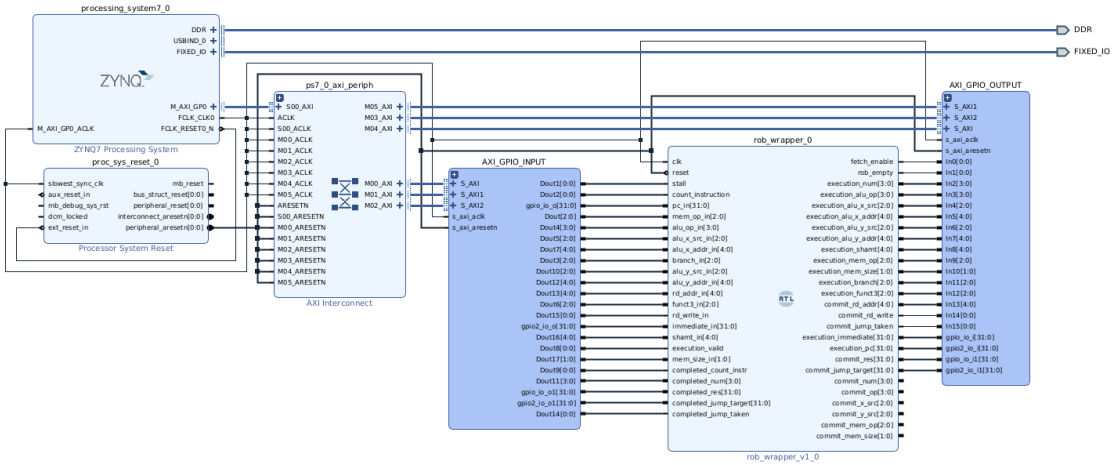

Figure 5.4: Reorder Buffer Block Diagram.

Two figures of merit were collected for each configuration under test. The maximum achievable clock frequency, f_{max}, was obtained from the post-implementation timing summary report, taking the worst negative slack across the design into account. Resource utilization was instead collected from the post-optimization (`opt_design`) report, and includes the number of occupied Slice LUTs, Slice Registers, F7 and F8 multiplexers, and the resulting number of occupied Slices, which accounts for how efficiently the individual LUTs and registers are packed together by the tool.

Since the size of the Reorder Buffer is expected to directly affect both the achievable frequency and the occupied area, a sweep across several table sizes was performed. This was achieved by manually changing the `NUM_INSTRUCTIONS` generic exposed by `rob_wrapper.vhd`, and repeating the synthesis and implementation flow from scratch for each value of interest, rather than relying on a single parameterized run. Five configurations were evaluated in this way, with the Reorder Buffer sized to hold 8, 10, 12, 14, and 16 instructions respectively.

5.2.1 ROB Results

Table 5.6 reports the collected results for the five evaluated Reorder Buffer sizes.

N. Entries	f_{max}[MHz]	LUTs	Registers	F7 Mux	F8 Mux	Slices
8	31	2561	1442	74	0	895
10	27	4092	1836	328	0	1356
12	21	5002	2128	20	0	1775
14	18	7983	2434	463	1	2568
16	14	7811	2638	381	130	2669

Table 5.6: Reorder Buffer resource utilization and maximum clock frequency, post-optimization, as a function of the number of entries.

The maximum clock frequency decreases monotonically as the number of entries grows, from 31 MHz with an 8-entry table down to 14 MHz with a 16-entry table, more than halving over the range considered. Resource utilization follows the opposite trend: the number of Slice Registers grows close to linearly with the number of entries, as expected, since every additional row of the table requires its own set of state-holding fields described in Section 4.1.1. The growth in Slice LUTs and in occupied Slices is comparatively steeper and less regular, which is consistent with the combinational hazard-detection logic scaling faster than linearly with table size, rather than with purely additive storage costs.

The process `p1p2_flags_managing`, described in Section 4.1.2, is the critical path responsible for this degradation. It recomputes the `p1` and `p2` hazard flags for every instruction on every clock cycle by comparing it against every other, older instruction currently held in the table. This all-pairs comparison scales combinationally with the square of `NUM_INSTRUCTIONS`: doubling the number of entries roughly quadruples the number of comparisons that this logic must resolve within a single clock period. The synthesis figures obtained are consistent with this behavior, although a more direct confirmation would require inspecting the specific timing path reported as critical by Vivado for the largest configurations, which was not carried out as part of this work.

The utilization figures are not perfectly monotonic across every column: for instance, the 16-entry configuration reports fewer Slice LUTs than the 14-entry one, despite occupying more Slice Registers and more Slices overall, and it is the only configuration in which F8 multiplexers are used in any significant number. This is not unexpected, since the Vivado synthesis and technology-mapping heuristics can pack logic differently from one run to the next depending on the specific netlist being optimized, and the presence of F7/F8 multiplexer chains in particular

reflects a choice made by the tool between implementing wide multiplexing logic in dedicated multiplexer resources or in look-up tables, rather than a property that scales predictably with `NUM_INSTRUCTIONS`. The overall trend of increasing area and decreasing frequency with table size nonetheless remains clear across the five configurations evaluated.

Table 5.7 reports the on-chip power breakdown obtained from Vivado's power analysis for each of the five configurations, with every design evaluated at its own maximum frequency as reported in Table 5.6.

N. Entries	Total	Static	Dynamic
8	1.397W	0.134W	1.262W
10	1.396W	0.134W	1.262W
12	1.396W	0.134W	1.262W
14	1.399W	0.134W	1.265W
16	1.394W	0.134W	1.260W

Table 5.7: On-chip power, post-implementation, as a function of the number of entries.

Unlike the frequency and area figures discussed above, total power consumption is essentially constant across the whole sweep, varying by less than half a percent between the lowest and the highest value recorded. Static power in particular is fixed at 0.134 W regardless of `NUM_INSTRUCTIONS`, which is expected, since static power is primarily a property of the target device and of the amount of silicon area powered on, rather than of the specific logic function implemented. Dynamic power follows the same essentially flat trend, remaining within a narrow 1.260–1.265 W band despite the more than threefold growth in Slice Registers observed between the 8-entry and the 16-entry configurations.

The reason becomes clear once the dynamic component is broken down further, as reported in Table 5.8.

N. Entries	Clocks	Signals	Logic	PS7
8	0.005W	0.001W	0.001W	1.256W
10	0.004W	0.001W	0.001W	1.256W
12	0.004W	0.001W	0.001W	1.256W
14	0.004W	0.003W	0.002W	1.256W
16	0.003W	0.001W	<0.001W	1.256W

Table 5.8: On-chip dynamic power, broken down by contributor.

The Zynq Processing System (PS7) block alone accounts for essentially the entire dynamic power budget in every configuration, consistently around 1.256 W, that is, more than 99% of the total dynamic power regardless of Reorder Buffer size. The remaining Clocks, Signals, and Logic contributions, which together power the user logic instantiated in the block design described above, never exceed a few milliwatts combined, and do not grow monotonically with `NUM_INSTRUCTIONS`: the 16-entry configuration, despite being the largest, reports a smaller combined contribution than the 8-entry one. It should be noted that these three categories are not attributable to the Reorder Buffer alone, since, as discussed in Section 5.2, the block design used to make the module independently testable on hardware also instantiates the AXI GPIO cores and the Processing System reset logic required to interface it with the PS7, and their contribution is folded into the same figures.

Taken together, these results indicate that the Reorder Buffer's own power draw is negligible in absolute terms when compared to the rest of the system evaluated, to the point where it does not noticeably affect the total power figure even at the largest table size considered. This is a consequence of the specific evaluation setup, in which a small custom IP is attached to a PS7 block whose own power consumption dominates the design almost entirely, rather than a general property of how power scales with `NUM_INSTRUCTIONS`; a dedicated, PS7-independent power measurement would be required to isolate and characterize the Reorder Buffer's own scaling behavior. Within the scope of this evaluation, however, the results are encouraging from a low-power design perspective, since the combinational hazard-resolution logic responsible for the frequency degradation discussed above does not translate into a comparably significant increase in power consumption.

5.2.2 Potato Processor Results

The previous section isolated the Reorder Buffer from the rest of the design in order to study how its own area and frequency scale with `NUM_INSTRUCTIONS`. This

section instead evaluates the complete Potato processor, integrated with its memory-mapped peripherals as in `tb_soc.vhd`, across the three architectural variants developed in this thesis: the original in-order pipeline, the single-Reorder-Buffer Out-of-Order (OoO) extension of Chapter 4, and the Simultaneous Multithreading (SMT) extension of Section 4.3, which instantiates two independent Reorder Buffers behind a shared Arithmetic Logic Unit and memory interface. For this comparison, every Reorder Buffer instance is fixed at eight entries, the smallest size considered in the sweep of Section 5.2.1; the goal here is not to repeat that sweep at the whole-core level, but to isolate, at a single fixed Reorder Buffer size, the additional area and power cost that each architectural extension introduces with respect to the conventional in-order baseline.

Unlike the Reorder Buffer sweep of Section 5.2.1, where each table size was implemented at its own maximum achievable frequency, the three architectures compared in this section were all synthesized and implemented under the same timing constraint of 25 MHz. This value corresponds to the f_{max} achieved by the slowest of the three designs, the SMT variant, whose two Reorder Buffers, shared arbiter, and dual Fetch/Decode front-end make it the most timing-critical of the configurations evaluated; the in-order and OoO designs, each individually capable of closing timing at a higher frequency, were deliberately constrained to this same, more conservative clock instead. Fixing a single common operating frequency across all three variants, rather than comparing each at its own best achievable rate, is what allows the power figures reported later in this section to be attributed to the architectural differences between the designs rather than to differences in clock rate between them.

For each configuration, a device (floorplan) view of the implemented design was generated in Vivado, with each figure below highlighting the placed cells belonging to different parts of the hierarchy: the non-Reorder-Buffer processor logic is shown in green, the Reorder Buffer (`ROB0` in the SMT case) in orange/red, the second Reorder Buffer of the SMT variant (`ROB1`) in yellow, and the memory-mapped peripherals (UART, GPIO, timer, and interconnect) in cyan; the narrow, evenly spaced columns visible at fixed horizontal positions in every figure correspond to the device's own I/O and memory-resource columns rather than to any highlighted logic. In every utilization table that follows, the `Potato Processor` row reports the resource usage of the complete design, and therefore already includes the contribution of the Reorder Buffer or Reorder Buffers instantiated within it; the separate `ROB`/`ROB0`/`ROB1` rows are not to be added to this total, but are given alongside it purely to break down how much of the complete design's resource usage is attributable to the Reorder Buffer component specifically.

Table 5.9 and Figure 5.5 report the baseline in-order configuration, which contains no Reorder Buffer and therefore no corresponding breakdown row.

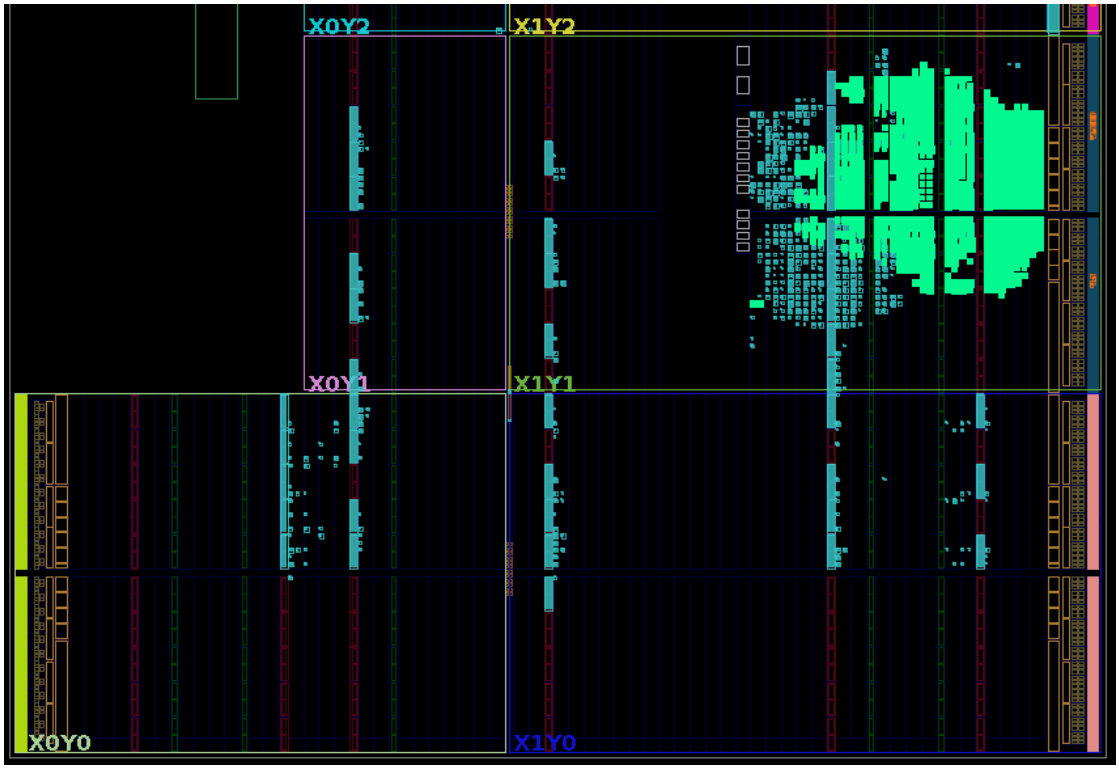

Figure 5.5: Potato Processor In-Order Area.

components	Slice LUTs	Slice Registers	F7 Muxes	Slice	LUT as Logic	LUT as Memory
Potato Processor	1827	1194	1	628	1783	44

Table 5.9: Potato Processor In-Order utilization.

Table 5.10 and Figure 5.6 report the same design extended with a single, eight-entry Reorder Buffer.

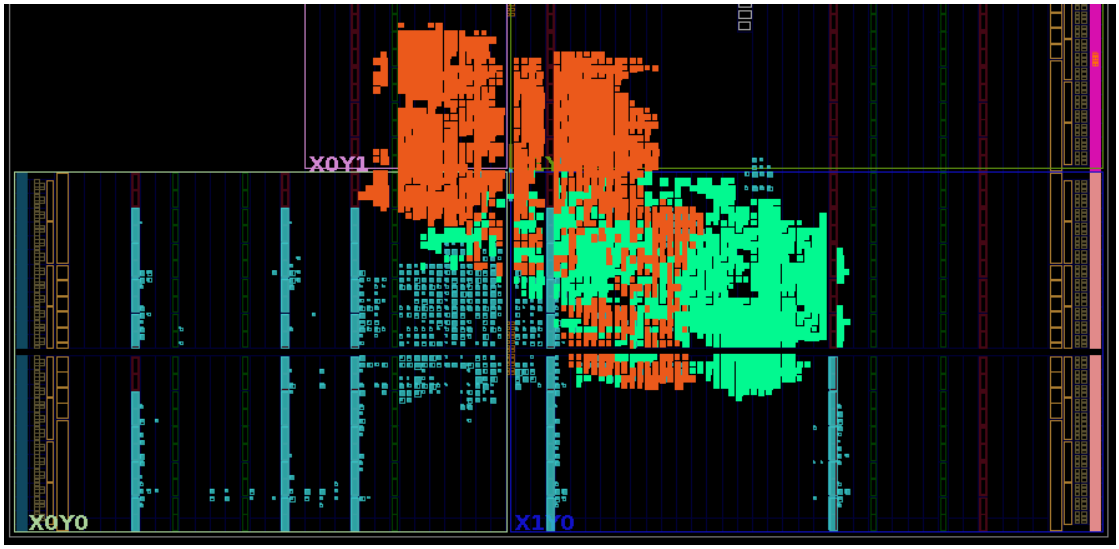

Figure 5.6: Potato Processor Out-of-Order Area.

components	Slice LUTs	Slice Registers	F7 Muxes	Slice	LUT as Logic	LUT as Memory
Potato Processor	4301	2821	79	1676	4257	44
ROB	2260	1396	79	1005	2260	0

Table 5.10: Potato Processor OoO utilization.

Adding the Reorder Buffer more than doubles every resource figure with respect to the in-order baseline: Slice LUTs grow from 1827 to 4301 (a 2.4× increase), Slice Registers from 1194 to 2821 (2.4×), and occupied Slices from 628 to 1676 (2.7×). The Reorder Buffer itself, at 2260 LUTs, 1396 registers, and 1005 Slices, accounts for roughly half of this total, leaving approximately 2041 LUTs and 1425 registers attributable to the rest of the processor. This remainder is itself larger than the entire in-order baseline (1827 LUTs, 1194 registers), which indicates that supporting Out-of-Order execution has a cost beyond the Reorder Buffer module in isolation: additional multiplexing and control logic is required elsewhere in the pipeline, for instance to route operands and results between the register file, the execution unit, and the Reorder Buffer's dispatch and completion ports described in Section 4.1.1. It is also worth noting that the 2260-LUT figure measured here for the eight-entry Reorder Buffer is somewhat lower than the 2561-LUT figure obtained for the same configuration in Table 5.6; the two values are not directly comparable, for at least two reasons. First, they were obtained from different synthesis boundaries, the standalone module wrapped for AXI GPIO-driven hardware testing in one case, and the module as instantiated and optimized together with the rest of the processor in

the other, and such boundary effects on technology mapping were already observed to affect the utilization figures in Section 5.2.1. Second, the two figures were obtained under different timing constraints, 31 MHz for the standalone eight-entry configuration against the 25 MHz constraint used throughout this section, and a looser timing target can itself lead Vivado to make different technology-mapping and packing choices even for logically identical netlists.

Table 5.11 and Figure 5.7 report the SMT variant, in which two independent, eight-entry Reorder Buffers are instantiated behind the shared arbiter of Section 4.3.2.

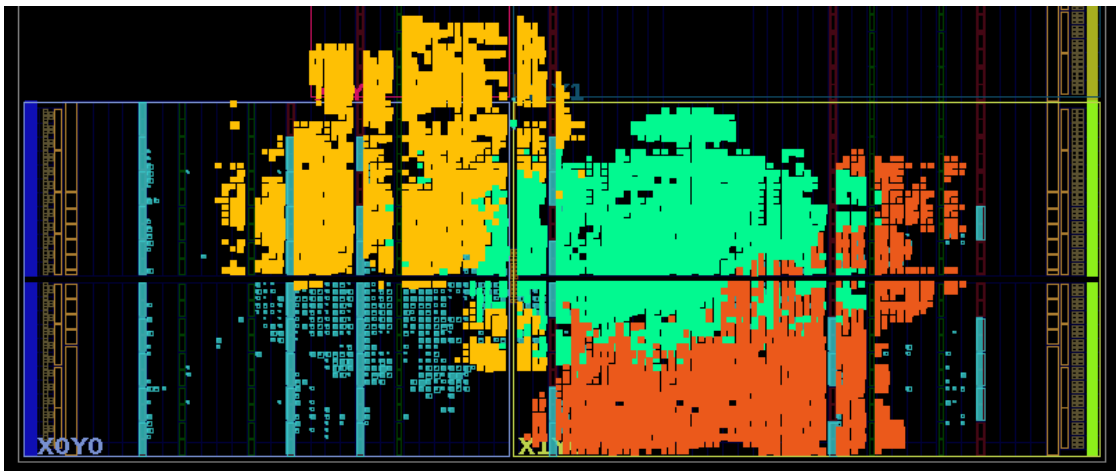

Figure 5.7: Potato Processor Simultaneous Multithreading Area.

components	Slice LUTs	Slice Registers	F7 Muxes	Slice	LUT as Logic	LUT as Memory
Potato Processor	8007	4816	158	3045	7919	88
ROB0	2334	1396	76	1015	2334	0
ROB1	2537	1461	82	1007	2537	0

Table 5.11: Potato Processor SMT utilization.

The two Reorder Buffer instances together account for 4871 LUTs and 2857 registers, leaving approximately 3136 LUTs and 1959 registers for the rest of the design. Compared to the equivalent non-Reorder-Buffer remainder of the OoO configuration (2041 LUTs, 1425 registers), this is a further increase of about 54% in LUTs and 37% in registers, consistent with the SMT variant requiring more than a second Reorder Buffer alone: the arbiter mediating shared access to the Arithmetic Logic Unit and memory interface, and the parallelized dual Fetch and Decode front-end of Section 4.3.1, both add logic outside the Reorder Buffers themselves.

At the same time, the complete design does not simply double with respect to the OoO configuration: 8007 LUTs against 4301 is a 1.86× increase, not 2×, which is consistent with the Arithmetic Logic Unit and memory unit being shared between the two hardware threads, as described in Section 4.3, rather than duplicated alongside the Reorder Buffers. The two Reorder Buffer instances also differ slightly from one another, despite being structurally identical (2334 against 2537 LUTs, 1396 against 1461 registers); as with the non-monotonic figures already observed in Table 5.6, this is attributed to Vivado's placement and technology-mapping heuristics rather than to any functional asymmetry between `ROB0` and `ROB1`.

Table 5.12 reports the on-chip power breakdown for the three architectures, each evaluated under the common 25 MHz constraint described above, so that the reported differences reflect the architecture under evaluation rather than a difference in clock rate.

Architecture	**Total**	**Static**	**Dynamic**
In-Order	0.236W	0.110W	0.126W
OoO	0.239W	0.110W	0.129W
SMT	0.249W	0.110W	0.139W

Table 5.12: Potato Processor On-Chip Power.

As in the standalone Reorder Buffer analysis, static power is essentially unaffected by the architecture under evaluation, remaining fixed at 0.110 W across all three designs, consistent with static power being primarily a property of the device and the fraction of it that is powered on rather than of the specific logic implemented. Dynamic power grows monotonically across the three configurations, from 0.126 W to 0.129 W (a modest 2.4% increase for the OoO variant) and further to 0.139 W (a 10.3% increase over the in-order baseline, or 7.8% over the OoO variant for the SMT extension). This growth is markedly more moderate than the corresponding growth in area, which in terms of Slice LUTs is roughly 2.4× for OoO and a further 1.86× for SMT: once all three designs are compared under the same 25 MHz clock, the additional switching activity introduced by the Reorder Buffer's combinational hazard-resolution logic and, for SMT, by the arbiter and dual front-end, turns out to have only a small effect on dynamic power relative to its substantial cost in LUTs and registers. This gap between the growth in area and the growth in power would have been considerably harder to isolate had each architecture instead been evaluated at its own, different maximum frequency, since a large part of any observed power difference would then have been attributable to the clock rate rather than to the architecture itself.

Table 5.13 breaks the dynamic power figure down further by contributor.

Architecture	Clocks	Signals	Logic	BRAM	MMCM	I/O
In-Order	0.003W	0.001W	0.001W	0.004W	0.115W	0.002W
OoO	0.004W	0.003W	0.003W	0.002W	0.115W	0.001W
SMT	0.007W	0.007W	0.006W	0.002W	0.115W	0.001W

Table 5.13: Potato Processor On-Chip Dynamic Power Details.

Unlike the standalone Reorder Buffer evaluation of Section 5.2.1, where the board's hard Processing System dominated the dynamic power budget, here the design is a self-clocked System on Chip with no such block, and it is instead the Mixed-Mode Clock Manager (MMCM) generating the core's clock that accounts for the large majority of dynamic power in every configuration, at 0.115 W identically for the in-order, OoO, and SMT variants alike. Since MMCM dynamic power is primarily determined by the output clock frequency it is configured to generate rather than by the size of the logic it drives, this identical figure is a direct consequence of the common 25 MHz constraint under which all three designs were synthesized and implemented, and offers a useful sanity check on that methodology: had each architecture instead been closed at its own, different maximum frequency, the MMCM contribution would have varied across the three designs and confounded the comparison of the remaining, logic-dependent contributions. The remaining Clocks, Signals, and Logic contributions, which power the user logic of the processor itself, grow monotonically across the three architectures (from 0.003 W, 0.001 W, and 0.001 W for the in-order baseline to 0.007 W, 0.007 W, and 0.006 W for the SMT variant), consistent with the growing amount of combinational hazard-resolution and arbitration logic switching at the same, shared clock rate. Block RAM and I/O power remain small in absolute terms across all three configurations, marginally higher for the in-order baseline than for the OoO and SMT variants, as expected since neither the memory sizing nor the external interfacing changes substantially between the evaluated architectures.

Taken together, the results of this section indicate that, at a fixed eight-entry Reorder Buffer size and under the common 25 MHz constraint imposed to keep the comparison meaningful, enabling Out-of-Order execution increases the Slice LUT count of the Potato core by roughly 2.4× with respect to its in-order baseline while increasing its dynamic power consumption by only about 2%, and that extending it further to Simultaneous Multithreading adds a further 1.86× in Slice LUTs but only a further 7.8% in dynamic power, in part because the Arithmetic Logic Unit and memory interface continue to be shared between the two hardware threads rather than duplicated. Power scales considerably more gently than area throughout, which is an encouraging property from the point of view of low- and moderate-power

FPGA-based applications, even though the area cost of these extensions, and in particular of the Reorder Buffer's hazard-detection logic, remains substantial. This gentle scaling of power relative to area is itself a direct consequence of holding the clock frequency fixed across all three designs: had each been evaluated at its own best achievable frequency instead, differences in clock rate between the architectures would have been folded into the power figures, obscuring how much of the additional power draw is actually attributable to the extra logic introduced by Out-of-Order execution and Simultaneous Multithreading themselves.

Chapter 6

Conclusions

This thesis addressed the extension of the open-source Potato RISC-V soft core with Out-of-Order execution and Simultaneous Multithreading, with the aim of exploring how a small, in-order educational core can be pushed towards a more advanced microarchitecture while retaining a compact and reusable design. The starting point was the standard five-stage Potato pipeline, implementing the RV32I base integer instruction set together with the Zicsr extension, as introduced in Chapter 3.

The central contribution of this work is the Reorder Buffer developed in Section 4.1, a self-contained module able to resolve Read-After-Write hazards autonomously, without relying on register renaming, and to guarantee in-order commit of results in the presence of Out-of-Order execution. Its integration into the Potato pipeline, described in Section 4.2, required removing the original data-forwarding paths, while preserving a dedicated in-order treatment of Control and Status Register instructions and of back-to-back memory accesses, both of which fall outside the general hazard-resolution mechanism provided by the ROB. Building on the observation that the module was designed to be largely architecture-independent, Section 4.3 showed how instantiating two independent Reorder Buffers, one per hardware thread, together with a duplicated Fetch and Decode front-end and an arbitrated shared Execute stage, is sufficient to obtain a two-thread Simultaneous Multithreading architecture without resorting to reservation stations.

The functional correctness of the Out-of-Order variant was verified in Chapter 5 through two independent simulation flows, GHDL and Xilinx Vivado, exercising both an ISA compliance suite covering every RV32I instruction and a set of local, hand-written tests targeting hazard conditions specific to this architecture, such as back-to-back CSR accesses. Every test passed against both the core-only and the complete SoC testbenches, as reported in Section 5.1.4, confirming that the ROB correctly preserves the architectural behavior of the processor while enabling Out-of-Order execution.

Beyond pass/fail correctness, Section 5.1.4 also characterized the cycles-per-instruction (CPI) cost that the Out-of-Order core adds over the In-Order baseline it extends. Across the Arithmetic/Logic, Branch/Jump, and Miscellaneous/System instruction categories, CPI increased by roughly 25% on average, ranging from about 10% to 40% test by test and somewhat lower for Arithmetic/Logic instructions than for the other two categories, a cost attributed primarily to the removal of general operand forwarding from the datapath rather than to the additional pipeline stage contributed by the Reorder Buffer, since the extra cycles observed track the number of retired instructions rather than behaving as a one-time, per-test overhead. The Memory category departed sharply from this pattern, showing a much larger, 121% average increase; this was traced to an instruction-count mismatch between the In-Order and Out-of-Order testbenches for multi-cycle memory accesses rather than to a genuine architectural penalty, and was accordingly treated as an open measurement inconsistency rather than as a reliable characterization of Out-of-Order memory performance. Extending this comparison to the dual-thread case, Section 5.1.4 further estimated, from the design of the Simultaneous Multithreading arbiter of Section 4.3.2, a per-thread CPI increase of roughly 20–25% under full Execute-stage contention between both threads, comparable to the roughly 22% increase already measured for Arithmetic/Logic instructions moving from the In-Order baseline to the single-threaded Out-of-Order core.

The physical characteristics of the Reorder Buffer were instead evaluated in Section 5.2 by synthesizing and implementing it, together with a supporting AXI infrastructure, on the Xilinx Zynq-7000 SoC FPGA mounted on the Pynq-Z2 board, across five table sizes ranging from 8 to 16 entries. The results collected in Table 5.6 showed a clear trade-off between table size and achievable performance: the maximum clock frequency more than halves, from 31 MHz to 14 MHz, as the number of entries grows, while area occupation grows accordingly. This degradation was attributed to the `p1p2_flags_managing` process, whose all-pairs hazard comparison scales with the square of the number of entries, making it the critical path of the design for every configuration evaluated. Complementing this standalone sweep, Section 5.2.2 compared the complete in-order, Out-of-Order, and Simultaneous Multithreading Potato cores against one another at a single, fixed eight-entry Reorder Buffer size, showing that enabling Out-of-Order execution more than doubles the area of the core (a 2.4× increase in Slice LUTs) while increasing its dynamic power consumption by only about 2%, and that extending it further to Simultaneous Multithreading adds a comparatively smaller further increment in area (1.86×) but a somewhat larger relative increment in dynamic power, 7.8% over the OoO variant and 10.3% over the in-order baseline, in part because the Arithmetic Logic Unit and memory interface remain shared between the two hardware threads rather than being duplicated alongside the two Reorder Buffers.

Taken together, these results confirm that a Reorder Buffer of the kind developed in this thesis is a viable and reasonably self-contained way of introducing Out-of-Order execution, and by extension Simultaneous Multithreading, into a simple RISC-V soft core such as Potato, without requiring a register renaming stage or reservation stations. At the same time, they highlight that the scalability of this specific implementation is fundamentally limited by the combinational cost of its hazard-detection logic, which grows faster than the storage it is meant to manage. Power consumption, by contrast, scales considerably more gently than either frequency or area throughout this evaluation, both for the Reorder Buffer sweep of Table 5.6 and for the whole-core comparison of Section 5.2.2: enabling Out-of-Order execution and, further, Simultaneous Multithreading increases dynamic power by a comparatively modest margin next to the corresponding growth in area, a favorable property for FPGA-based low- and moderate-power applications that partially offsets the substantial area cost of these extensions. The following section discusses a number of directions along which the work presented in this thesis could be extended or refined.

6.1 Future Works

The Simultaneous Multithreading architecture presented in Section 4.3 was synthesized and evaluated in Section 5.2.2 only at a single, fixed Reorder Buffer size of eight entries per thread, alongside the in-order and Out-of-Order baselines, rather than across the full range of table sizes considered for the standalone Reorder Buffer in Table 5.6. Applying a synthesis sweep analogous to that one, repeated for the complete Simultaneous Multithreading core rather than for the Reorder Buffer in isolation, would establish whether the same frequency and area trends still hold once two ROBs and the shared Arithmetic Logic Unit arbiter of Section 4.3.2 are integrated together and scaled up, and whether the cross-thread CSR ordering mechanism introduced in Section 4.3.1 continues to behave correctly under concurrent thread activity as both Reorder Buffers grow.

The dual Fetch and Decode front-end described in Section 4.3.1 assumes an instruction cache exposing two independent read ports, which was the simplest option available for an initial exploratory implementation but is also the more costly one in terms of memory resources. A more general front-end, capable of sharing a single-read-port instruction cache between the two threads, could be realized by arbitrating access to the cache and introducing a small FIFO buffer per thread to temporarily hold fetched instructions whenever a thread is denied access to the cache in a given cycle, so that neither thread is forced to stall for the entire duration of the arbitration.

As anticipated in Section 4.1.1, Read-After-Write hazards are currently resolved

entirely within the Reorder Buffer, through the `p1` and `p2` flags compared against every other in-flight instruction, without any form of register renaming. Introducing a dedicated renaming stage between Decode and the ROB, mapping architectural registers onto a larger set of physical ones, would remove the need for this all-pairs comparison for the specific case of Read-After-Write hazards, likely easing the pressure on the `p1p2_flags_managing` process identified in Section 5.2.1 as the critical path of the design, at the cost of the additional area and complexity required by the renaming logic itself and its associated free-list management.

The fields composing each row of the Reorder Buffer, discussed in Section 4.1.1, were sized directly according to the control signals already defined by the pre-existing Potato datapath, rather than being redesigned specifically for this purpose. Since Table 5.6 shows that the number of occupied Slice Registers grows close to linearly with the number of entries, a dedicated review of these fields, aimed at identifying and eliminating redundant or oversized encodings, could reduce the per-entry storage cost of the table and, consequently, improve the area figures reported for the larger configurations without affecting the number of entries available to the scheduler.

A related limitation concerns the handling of consecutive memory operations, discussed in Section 4.2.2: the current implementation requires the ROB to recognize back-to-back load and store instructions through the `multiple_load_op` mechanism and to freeze pointer advancement for one cycle until the first access completes, since the underlying Execute stage cannot otherwise hold a second memory address while waiting on the first. A dedicated load/store queue, decoupling address generation from the point at which a memory operation is actually issued to the data memory, would allow consecutive memory accesses to be buffered rather than stalling the whole pipeline, at the cost of an additional structure to be integrated with the existing Reorder Buffer.

The processor developed in this thesis targets only the RV32I base integer instruction set together with the Zicsr extension, which was sufficient to support the Out-of-Order and Simultaneous Multithreading mechanisms studied here. Extending the supported ISA, for instance with the M extension for integer multiplication and division, would introduce execution units with latencies different from those of the current arithmetic and logical operations, and would therefore constitute a more demanding test of the ROB's ability to manage instructions that complete out of program order over a wider range of execution times.

A second, SMT-specific direction for ISA extension concerns thread management. In the current design, both hardware threads are implicitly active from reset, and the arbiter described in Section 4.3 allocates fetch and execution bandwidth between them without any means for software to influence this allocation at run time. Introducing dedicated instructions for thread creation and termination would give software explicit control over the lifecycle of a hardware thread, allowing a thread

context to be spawned or torn down on demand rather than being permanently resident. Beyond lifecycle management, such instructions would also make it possible to optimize the utilization of the architecture, for example by letting software signal when a thread is idle so that the arbiter can temporarily reassign its share of shared resources to the remaining active thread, an optimization that the present implementation, with its fixed two-thread arbitration scheme, does not support.

The current architecture does not provide any mechanism for handling asynchronous interrupts, being limited to the synchronous, in-order treatment of CSR instructions described in Section 4.2.1. Integrating interrupt support would require the ability to flush the Reorder Buffer of any speculative or in-flight state and to redirect Fetch to the appropriate handler address at an arbitrary point in the instruction stream, a capability that the current design, centered on in-order commit of an uninterrupted instruction flow, does not yet provide.

Finally, as already noted in Section 5.1.4, the regression suite used throughout this work is entirely composed of directed, hand-written tests, and does not amount to a systematic exploration of the space of instruction interleavings that the Reorder Buffer may encounter over long-running programs. Complementing it with an extended verification campaign, for instance by running representative bare-metal software over a significantly longer simulated runtime, or by adopting a constrained-random instruction generation approach, would help expose timing-dependent hazards that short, directed tests are unlikely to trigger, and would provide a stronger confidence guarantee before the architecture is considered for any use beyond the scope of this thesis.

A further direction, not related to performance, concerns the reliability of the design. Each Reorder Buffer resolves hazards and selects the instruction to dispatch as a purely deterministic function of the sequence of instructions and completions it receives, through the `p1`/`p2` comparison and pointer logic described in Section 4.1.1: no part of this process depends on anything external to the table itself. As a consequence, two Reorder Buffers fed with the same instruction stream from the same initial state are guaranteed to reach the same sequence of committed results, cycle for cycle. This property means that the pair of ROBs already instantiated for the Simultaneous Multithreading architecture of Section 4.3 could, without any modification to the module itself, be repurposed to execute two redundant copies of the same thread rather than two independent ones. A dedicated comparator, continuously checking both the `completed_res` and `completed_jump_target` signals produced as each instruction finishes execution and the `commit_res`, `commit_rd_addr`, and `commit_jump_target` outputs produced as each instruction commits, of both ROBs against one another at every cycle, would then be able to flag a mismatch as evidence of a transient fault, such as a Single Event Upset caused by radiation, without requiring an entirely separate

redundant core to be instantiated for this purpose. Such a scheme would come at the cost of forfeiting the throughput advantage that motivates using two ROBs for genuine SMT in the first place, since both hardware contexts would then execute the same program instead of two independent ones; the throughput-oriented and the reliability-oriented modes of operation would need to be selectable rather than pursued at the same time. Rather than committing permanently to one mode or the other, the core could instead switch into a dedicated test mode during idle periods, that is, whenever the second hardware thread has no independent work to dispatch, temporarily mirroring the active thread onto the otherwise unused ROB and comparing their commit outputs for the duration of the idle interval. Such a scheme would confine the throughput cost of redundant execution to cycles that would otherwise be wasted, allowing periodic correctness checks to be performed opportunistically instead of continuously, at the cost of only providing fault coverage intermittently rather than on every cycle of execution. Quantifying this trade-off, and the additional comparator and mode-switching logic it requires, would extend the dual-ROB architecture developed in this thesis towards the security- and reliability-critical application domains for which the inspectability of an open, custom RISC-V core is particularly valuable.

Bibliography

[1] G. Cora, C. De Sio, S. Azimi, and L. Sterpone. «Selective hardening of RISCV soft-processors for space applications». In: *Microelectronics Reliability* 167 (2025), p. 115667. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2025.115667 (cit. on p. 1).

[2] Giorgio Cora, Eleonora Vacca, Corrado De Sio, Sarah Azimi, and Luca Sterpone. «RePAIR: Reconfigurable Platform for AI Resilience Within RISC-V Ecosystem». In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Cham: Springer Nature Switzerland, 2025, pp. 71–87. ISBN: 978-3-031-87995-1 (cit. on p. 1).

[3] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed. Waltham, MA: Morgan Kaufmann, 2017 (cit. on pp. 2–4, 17, 20, 21, 23–27).

[4] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case for RISC-V*. Tech. rep. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, Aug. 2014 (cit. on p. 2).

[5] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture*. Tech. rep. Document Version 20260120, Official Release. RISC-V International, 2026 (cit. on pp. 3, 17, 18).

[6] Wikipedia contributors. *Field-programmable gate array*. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Field-programmable_gate_array (visited on 06/22/2026) (cit. on pp. 5, 6).

[7] Claudio Passerone. *Programmable Logic Devices*. Lecture slides, Electronics for Embedded Systems. A.A. 2022/23. 2022 (cit. on p. 5).

[8] Abdallah Cheikh, Gianmarco Cerutti, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri. «The Microarchitecture of a Multi-threaded RISC-V Compliant Processing Core Family for IoT End-Nodes». In: *Applications in Electronics Pervading Industry, Environment and Society*. Cham: Springer International Publishing, 2019, pp. 83–90. ISBN: 978-3-319-93082-4. DOI: 10.1007/978-3-319-93082-4_12 (cit. on pp. 10, 15).

[9] Jonnatan Mendoza Escobar. «A Multithreading RISC-V Implementation for Lagarto Architecture». Master's Thesis. Universitat Politècnica de Catalunya, 2020 (cit. on pp. 11, 15).

[10] Bernhard Lang. *FGMT-RiscV: A fine grained multi threading processor for FPGA systems*. Poster session at RISC-V Summit Europe 2025. Extended abstract presented at RISC-V Summit Europe 2025, Sub. #19, P2.3.01-Tue. 2025 (cit. on pp. 11, 15).

[11] Yossi Eni, Shlomo Greenberg, and Yehuda Ben-Shimol. «Efficient Hint-Based Event (EHE) Issue Scheduling for Hardware Multithreaded RISC-V Pipeline». In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 69.2 (2022), pp. 735–745. DOI: 10.1109/TCSI.2021.3129377 (cit. on pp. 12, 15).

[12] Hidetaro Tanaka, Tomoaki Tanaka, Ryosuke Higashi, Tsutomu Sekibe, Shuichi Takada, and Hironori Nakajo. «Implementation of a RISC-V SMT Core in an AI Processor». In: *Proceedings of the 11th International Symposium on Information and Communication Technology (SoICT 2022)*. Association for Computing Machinery, 2022, pp. 15–22. DOI: 10.1145/3568562.3568634 (cit. on pp. 12, 15).

[13] Yuta Nojiri and Nobuyuki Yamasaki. «A Design of Multithreaded RISC-V Processor for Real-Time System». In: *Proceedings of the Eleventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE, 2023, pp. 31–37. DOI: 10.1109/CANDARW60564.2023.00014 (cit. on pp. 13, 15).

[14] Binh Kieu-Do-Nguyen, Khai-Duy Nguyen, Nguyen The Binh, Khai-Minh Ma, Tri-Duc Ta, Duc-Hung Le, Cong-Kha Pham, and Trong-Thuc Hoang. «Hardware Software Co-Design for Multi-Threaded Computation on RISC-V-Based Multicore System». In: *IEEE Access* 12 (2024), pp. 177312–177326. DOI: 10.1109/ACCESS.2024.3505940 (cit. on pp. 13, 15).

[15] Mert Dogan Arisoy. «Multithreaded support Embedded Application on RISC-V». Master's Thesis. Politecnico di Torino, 2021 (cit. on pp. 14, 15).

[16] Kristian Skordal. *The Potato Processor*. 2014. URL: https://github.com/skordal/potato (visited on 01/01/2024) (cit. on p. 22).

[17] James E. Smith and Andrew R. Pleszkun. «Implementation of Precise Interrupts in Pipelined Processors». In: *Proceedings of the 12th Annual International Symposium on Computer Architecture (ISCA)*. 1985, pp. 36–44 (cit. on p. 25).