Politecnico di Torino

Master's Degree in Computer Engineering (Embedded Systems)

Academic year 2024/2025

Graduation Session December 2025

Implementation of intermittent-robust computing on RISC-V architecture

RISE: RISC-V Intermittent System Extensions for Batteryless IoT Devices

Supervisor:

Arman Roohi

Guido Masera

Candidate:

Samuele Pasquale

Abstract

The growing deployment of batteryless Internet of Things (IoT) devices powered by ambient energy harvesting highlights the need for architectures capable of sustaining correct and reliable execution under frequent and unpredictable power interruptions. This thesis addresses this challenge by proposing RISE (RISC-V Intermittent System Extensions), an architectural and ISA-level framework that enables intermittent computing on a pipelined RISC-V processor.

The proposed approach introduces four lightweight hardware modules: the Intermittent Computing Register Wrapper (ICRW), which encapsulates the processor state and tracks modifications through dirty-bit management; the Power Control Unit (PCU), which performs selective background backups of modified registers; the Restore Control Unit (RCU), which reloads saved state upon power resumption; and the Dispatcher, which transparently arbitrates memory bus usage between normal execution and backup transfers. In addition, the instruction set architecture is extended with the `.ICA` primitive, which allows programmers to define atomic code regions that guarantee correctness and consistency despite intermittent power supply.

RISE is designed to preserve compatibility with standard RV32I pipelines, requiring only minimal modifications to the decode stage, while maintaining full portability across different RISC-V cores. The framework avoids reliance on non-volatile elements within the processor itself, ensuring CMOS compatibility and scalability. Backup and restore operations are executed concurrently with regular computation, thereby minimizing performance and energy overhead.

The framework was implemented in Verilog HDL and evaluated through simulation and synthesis using the Xilinx Vivado™ 2024.1 toolchain. Experimental results demonstrate that the proposed architecture achieves efficient and reliable execution in intermittent environments. In benchmark workloads, a complete processor state backup requires on average 203 cycles (0.203 µs at 1 GHz), confirming that RISE provides a practical and effective solution for sustainable intermittent computing in energy-harvesting IoT systems.

Acknowledgements

I would like to express my deepest gratitude to my family, whose constant support and encouragement made this journey possible. Their decision to give me the opportunity to study in the United States has been one of the greatest gifts of my life, and I will always be grateful for their trust and sacrifices. I also want to sincerely thank all my friends, who have been by my side in the most challenging moments, providing encouragement, companionship, and motivation when I needed it most. Finally, I wish to thank my university colleagues, with whom I have shared countless experiences during these years. The moments we lived together, both inside and outside the academic environment, are memories I will carry with me forever. To all of you, thank you.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

Over the past decade, the Internet of Things (IoT) has emerged as one of the most transformative paradigms in computing. Billions of devices are being deployed in smart homes, cities, industrial automation, healthcare, and environmental monitoring, where they continuously sense, process, and exchange information. These applications rely on small, low-power embedded systems that must operate reliably and autonomously, often in environments where maintenance is impractical or prohibitively expensive. As forecasts project hundreds of billions of IoT devices to be installed in the near future [1], the question of sustainable energy supply becomes a pressing challenge.

Traditionally, IoT nodes rely on batteries or rechargeable storage units. While effective in the short term, this approach faces several drawbacks:

- **Finite lifetime** – Batteries eventually degrade and must be replaced, which is infeasible for large-scale deployments such as industrial sensor networks or environmental monitoring in remote areas [2].
- **Maintenance overhead** – Replacing or recharging billions of batteries requires human intervention, which drastically increases operational costs.
- **Environmental concerns** – Battery disposal introduces severe sustainability issues, with toxic components contributing to e-waste.
- **Form factor and design constraints** – Batteries often dominate the physical size and weight of IoT devices, limiting their integration into miniaturized platforms.

As a consequence, researchers have investigated **batteryless operation** enabled by energy harvesting, where devices draw power from ambient sources such as solar radiation, radio frequency (RF) waves, thermal gradients, or mechanical vibrations [1]. Energy harvesting offers the promise of sustainable, maintenance-free operation, enabling devices to function indefinitely without human intervention.

However, energy harvested from the environment is inherently **intermittent and unpredictable**. The availability of power fluctuates over time and may fall below the threshold required for computation. For example, a solar-powered device may experience long outages during nighttime or cloudy conditions, while RF energy availability depends heavily on proximity to transmitters. As a result, a batteryless device frequently loses power, halting execution and discarding its volatile state. When energy becomes available again, execution must resume from some previously stored state [2]. This phenomenon gives rise to the research area of **intermittent computing**, where devices operate only when energy is available, with execution repeatedly suspended and resumed depending on the harvested power.

Because of these challenges, simply replacing a battery with a capacitor is insufficient. Without proper mechanisms, intermittent execution can lead to lost progress, corrupted data, and unreliable behavior. Thus, specialized execution models, architectures, and hardware support are necessary to enable intermittent computing.

1.2 Intermittent Computing: Challenges

Intermittent computing introduces several unique challenges that must be solved to guarantee reliable operation:

- **Progress guarantees** – Applications must eventually make forward progress, even if they are interrupted frequently. Without explicit mechanisms, devices may waste harvested energy re-executing the same operations after every failure [3].

- **Memory consistency** – Non-volatile memory (NVM) must remain consistent across failures. A partially updated state can corrupt program semantics and lead to unpredictable results [4].

- **Atomicity of operations** – Critical code regions must be executed entirely or not at all. Partial updates may leave data structures or control flow in an invalid state, which is unacceptable for safety-critical applications such as healthcare or industrial monitoring [5].

- **Energy and latency overhead** – Backup and restore operations themselves consume valuable time and energy. Reducing this overhead is crucial to maximize the fraction of time spent on useful work [1].

These challenges are exacerbated by the unpredictable nature of energy availability, which depends not only on environmental conditions but also on the size of the energy buffer and the device's power management strategy [2]. Addressing them requires cooperation across multiple system layers, from hardware and microarchitecture to compilers and runtime frameworks.

1.3 State of the Art

Over the years, several models have been proposed to address the challenges of intermittent computing. They can be broadly classified into four categories:

1. **Checkpointing-based solutions.** Systems such as *Mementos* periodically store the full volatile state to NVM. This method is conceptually simple but suffers from high overhead and wasted work when failures occur between checkpoints [3]. Early prototypes were often deployed on ultra-low-power microcontrollers, such as MSP430-based systems.

2. **Task-based execution.** Frameworks like *Alpaca* partition applications into atomic tasks that are guaranteed to either complete or restart. This reduces rollback overhead compared to checkpointing, but introduces complexity in compiler/runtime design and can increase memory usage due to variable privatization [3]. Task-based execution has proven particularly effective in RFID tags and batteryless sensor networks.

3. **Hardware-assisted approaches.** *Nonvolatile processors (NVPs)* integrate persistent flip-flops (NVFFs) into the processor pipeline, automatically preserving state during power failures. While effective, these approaches require specialized fabrication technologies such as MRAM or FeRAM, limiting portability and scalability [5]. Another line of work, such as *Freezer*, proposes external backup controllers that track dirty memory blocks and offload backup operations to specialized hardware, reducing backup time and energy at the cost of limited execution semantics [4].

4. **Hybrid and system-level solutions.** Recent proposals explore higher-level strategies, such as optimizing energy storage sizing to reduce failure frequency [2], or adapting intermittent models to high-performance out-of-order cores [6]. These works highlight the importance of considering intermittent computing as a cross-layer problem, spanning hardware design, memory hierarchies, and system-level energy management.

Each of these directions contributes to mitigating the negative effects of intermittent power. However, none of them fully resolvess the tension between efficiency, correctness, and portability, leaving space for new architectural solutions.

1.4 Rationale for a New Approach

Despite significant progress in the field, existing solutions face important limitations. Checkpointing incurs excessive rollback overhead, while task-based approaches complicate programming and increase memory requirements. Hardware NVPs provide transparency but depend on costly and non-standard fabrication processes. External controllers like Freezer are lightweight and portable, but they cannot enforce atomic code execution.

In summary, prior solutions address specific aspects of intermittent computing, but none provide an efficient, general-purpose, and ISA-integrated framework. This gap motivates the development of **RISE (RISC-V Intermittent System Extensions)**, which integrates intermittent computing support directly into the RISC-V architecture.

A carefully designed extension to a widely adopted ISA can provide:

- Low-latency and energy-efficient backup/restore operations.
- Explicit ISA primitives for atomic code regions, bridging the gap between hardware and software.
- Compatibility with standard CMOS processes, avoiding the need for exotic memory technologies.
- Portability across multiple cores and platforms thanks to an open, modular ISA such as RISC-V.

1.5 Research Objectives and Contributions

The main objective of this thesis is to propose, implement, and evaluate RISE, a novel framework that extends the RISC-V ISA and microarchitecture with support for intermittent execution. The contributions of this work can be summarized as follows:

1. **Architectural extensions:** introduction of hardware modules including the Intermittent Computing Register Wrapper (ICRW), Power Control Unit (PCU), Restore Control Unit (RCU), and Dispatcher.
2. **ISA primitives:** definition of the `.ICA` instruction to explicitly mark atomic execution regions, ensuring correctness across power failures.
3. **Efficient backup/restore:** design of a mechanism that integrates dirty-bit tracking and temporary buffers, reducing backup latency to as low as 0.203 μs at 1 GHz.

4. **Compatibility:** demonstration that RISE is implementable on standard CMOS and is extensible across multiple RISC-V cores.

5. **Evaluation:** experimental validation using benchmarks such as MiBench and RV32I, comparing RISE against both software-only and hardware-assisted approaches like Freezer.

1.6 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2** reviews related work in intermittent computing, covering software models, hardware-assisted approaches, NVM-based architectures, and system-level optimizations.

- **Chapter 3** details the design of the RISE architecture, including ISA extensions and the four hardware modules.

- **Chapter 4** presents the implementation of RISE on a RISC-V pipeline and its integration with the memory subsystem.

- **Chapter 5** evaluates RISE using benchmarks and compares its performance against existing approaches.

- **Chapter 6** discusses advantages, limitations, and possible extensions of RISE.

- **Chapter 7** concludes the thesis by summarizing the contributions and outlining future research directions.

Chapter 2

Related Work

2.1 Introduction

Intermittent computing has emerged as a key research area for enabling sustainable and reliable execution on batteryless devices powered by energy harvesting. The unpredictability of harvested energy forces systems to operate under frequent power failures, requiring mechanisms to ensure correctness, forward progress, and energy efficiency. A wide range of approaches has been explored across the hardware–software stack, including compiler-assisted frameworks, hardware modifications, and system-level optimizations.

In this chapter, we classify prior work into three main categories: (1) software-based approaches, (2) hardware-assisted mechanisms, and (3) system-level and architectural solutions. We conclude with a comparative analysis that highlights the gaps addressed by the proposed RISE architecture.

2.2 Software-based Approaches

Software-only frameworks represent the earliest and most widely explored class of solutions for intermittent computing. These methods require no custom hardware, instead relying on compiler instrumentation and runtime mechanisms to preserve program state.

2.2.1 Checkpointing Models

The first attempts at intermittent computing relied on **checkpointing**. Mementos introduced compiler-inserted checkpoints that periodically save the entire volatile state into non-volatile memory (NVM). This ensures forward progress across failures, but introduces high latency and energy overhead, particularly if failures occur between checkpoints [3]. Later frameworks such as DINO improved checkpoint placement and reduced redundancy, but the fundamental trade-off between reliability and wasted work remains. Checkpointing is therefore suitable for simple applications but scales poorly for compute-intensive workloads.

2.2.2 Task-based Execution

Task-based models avoid the pitfalls of checkpointing by dividing programs into atomic tasks. Each task must either complete successfully or be restarted, ensuring atomicity at task boundaries. Alpaca [3] is the most prominent example: it uses variable privatization during execution, committing results only after successful task completion. This eliminates partial updates and provides stronger correctness guarantees. Subsequent work such as Chain and Coati extended this model with enhanced task scheduling and dependency tracking.

The strength of task-based frameworks lies in their ability to ensure consistency without costly full-state checkpoints. However, these systems introduce memory overhead due to privatization and require significant compiler/runtime support, complicating portability and adoption in heterogeneous IoT environments.

2.3 Hardware-assisted Approaches

Hardware-based solutions aim to reduce software complexity and execution overhead by integrating specialized support for backup and restore directly into the processor or memory subsystem.

2.3.1 Non-Volatile Processors (NVPs)

Non-volatile processors (NVPs) incorporate persistent flip-flops (NVFFs) or registers that automatically retain processor state across failures. Examples include NV-CPU, WEC-NVREG, and other designs leveraging MRAM, FeRAM, or ReRAM technologies [5]. These architectures are transparent to software, since the processor can resume execution without explicit checkpoints or tasks. However, they incur substantial area and energy penalties, and require specialized fabrication processes that limit portability. Alternative proposals such as error-tolerant non-volatile registers [5] attempt to mitigate these issues, but the reliance on exotic technologies remains a barrier.

2.3.2 External Backup Controllers

An alternative to integrating NVM into the core is to use an external backup controller. Freezer [4] is a representative example: it monitors memory accesses, tracks dirty blocks, and offloads backup to a specialized controller. This reduces the time and energy required to capture system state, offering an attractive compromise between software-only and NVP approaches. However, Freezer cannot enforce atomicity of execution and provides no mechanism to prevent partially completed operations, limiting its ability to guarantee program correctness.

2.4 System-level and Architectural Solutions

Several works adopt a broader, system-level view of intermittent computing, focusing on energy management, processor architecture, and concurrency.

2.4.1 Energy Storage Optimization

The size of the energy buffer (e.g., capacitor) strongly influences system behavior. Work by [2] systematically studied how storage capacity affects forward progress, showing that even small changes can significantly alter failure frequency and performance. This line of research highlights the importance of hardware–software co-design and motivates the need for adaptable architectures.

2.4.2 High-performance Intermittent Cores

Most early solutions targeted low-power microcontrollers. However, more recent research has explored scaling intermittent computing to higher-performance processors. In particular, [6] proposed mechanisms for enabling intermittent execution on out-of-order (OoO) cores, demonstrating the feasibility of applying these models to more complex architectures. This shows that intermittent computing is not limited to ultra-low-power devices but can extend to high-performance domains.

2.4.3 Transactional and Concurrency Models

Beyond single-threaded execution, transactional models have been proposed to ensure concurrency control in intermittent systems. For example, [7] introduced a transactional framework that provides atomicity and isolation even under frequent failures. Such systems are critical for multi-threaded workloads or distributed sensing applications, though they introduce non-negligible runtime overhead.

2.4.4 RISC-V Based Proposals

The open and modular RISC-V ISA has emerged as a natural platform for experimenting with intermittent computing support. Several works have leveraged RISC-V to emulate or extend intermittent architectures [8, 2], showing its flexibility for ISA-level extensions and hardware modifications. These efforts demonstrate the growing trend toward integrating intermittent support at the architectural level, providing a foundation for the RISE proposal.

2.5 Comparative Analysis

Table 2.1 summarizes the main characteristics of existing approaches, comparing them along key dimensions such as backup/restore overhead, correctness guarantees, hardware requirements, portability, examples, and pros/cons.

Table 2.1: Extended comparison of intermittent computing solutions

Approach	Overhead	Atomicity	Hardware	Portability	Examples	Pros / Cons
Checkpointing	High	No	None	High	Mementos, DINO	+ Simple, no HW changes - High rollback overhead - Inefficient for long tasks
Task-based	Medium	Yes	None	Medium	Alpaca, Chain, Coati	+ Ensures atomicity + Reduces wasted work - Increased memory pressure - Requires compiler/runtime
NVPs	Low	Yes	Exotic NVM	Low	NV-CPU, WEC-NVREG, NVFF designs	+ Transparent to software + Fast recovery - High area/energy cost - Requires MRAM/FeRAM/ReRAM
External Controllers	Low	No	Dedicated unit	Medium	Freezer	+ Efficient backup/restore + Non-intrusive - No atomicity guarantees - Limited general-purpose use
System-level	Variable	Partial	Std. CMOS	High	Energy storage sizing, OoO cores	+ Explores cross-layer design + Works on standard CMOS - No direct correctness guarantees
Architectural / RISC-V	Low	Yes	Std. CMOS	High	What's Next, RISE	+ ISA-level primitives + Efficient hardware support + Portable, scalable - Still under active research

From this comparison, it is clear that no single approach fully addresses the trade-off between efficiency, correctness, and portability. Software-based frameworks are portable but suffer from high overhead, while hardware-based solutions are efficient but depend on non-standard technologies or lack execution semantics. System-level approaches provide useful insights but do not directly address the correctness problem. This gap motivates the design of RISE, which integrates intermittent support directly into the RISC-V ISA and microarchitecture.

2.6 Summary

In this chapter, we reviewed the landscape of intermittent computing solutions. Software-based methods provide portability but suffer from high overhead, hardware-assisted approaches improve efficiency but rely on exotic technologies or lack atomicity, and system-level strategies highlight important trade-offs but fail to address core correctness issues. Together, these limitations underline the need for a new approach: an architecture-level solution built on top of a widely adopted

ISA, offering efficient backup/restore mechanisms and explicit support for atomic execution. The next chapter presents RISE, a framework that addresses these challenges by extending the RISC-V ISA and microarchitecture.

Chapter 3

The RISE Architecture

3.1 System Overview

The proposed **RISC-V Intermittent System Extensions (RISE)** framework introduces architectural support for intermittent computing by extending the RISC-V ISA and microarchitecture. Unlike existing solutions, which rely on software-based checkpointing, task-level execution, or exotic non-volatile processors, RISE integrates a lightweight and portable hardware mechanism directly into a standard CMOS pipeline. The main goal is to enable reliable execution under frequent power failures while minimizing backup/restore overhead and maintaining programmability.

At a high level, RISE introduces four tightly coupled hardware modules: (1) the **Intermittent Computing Register Wrapper (ICRW)**, (2) the **Power Control Unit (PCU)**, (3) the **Restore Control Unit (RCU)**, and (4) the **Dispatcher**. Together with a new ISA primitive, the `.ICA` instruction, these modules allow efficient and correct recovery across arbitrary power failures.

Figure 4.6 illustrates the high-level integration of RISE with a baseline RV32I pipeline. The extensions are modular, enabling compatibility with existing RISC-V cores while avoiding invasive modifications.

3.2 ISA Extensions

A central aspect of RISE is the introduction of new ISA primitives that explicitly support intermittent execution semantics. In particular, the framework defines the `.ICA` (Intermittent Computing Atomic) instruction, which allows programmers to delimit regions of code that must be executed atomically with respect to power failures.

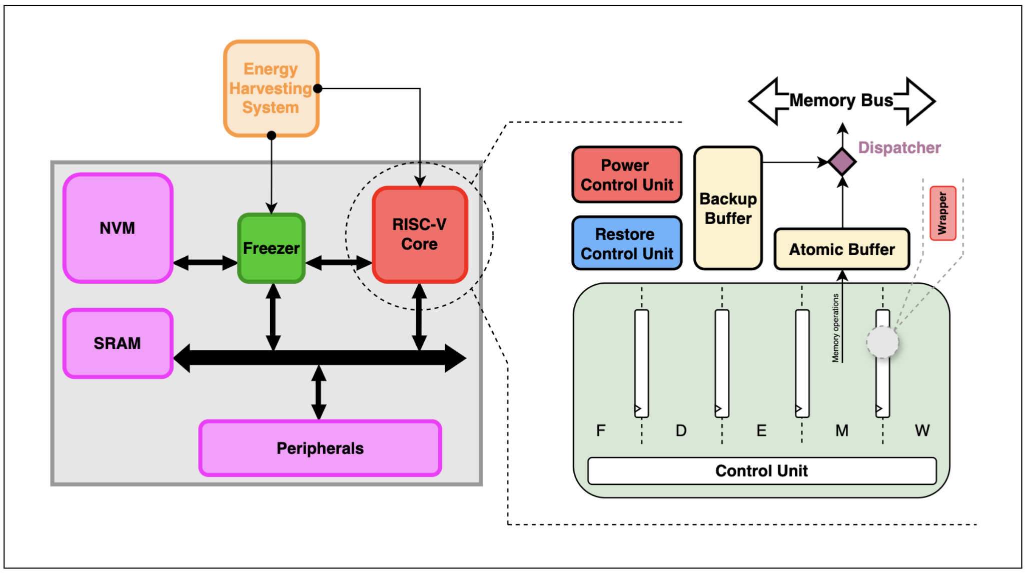


Figure 3.1: High-level overview of the RISE architecture integrated into a baseline RISC-V pipeline.

3.2.1 The .ICA Instruction

The `.ICA` instruction is used to mark the beginning and end of atomic code regions. When execution enters an `.ICA` region, RISE ensures that either the entire region executes successfully, or it is rolled back and restarted upon the next power-up. This prevents partially updated states, which are a major source of inconsistency in intermittent systems.

```
    .ICA_BEGIN
        ; critical operations
        lw   x5, 0(x10)
        add  x6, x5, x7
        sw   x6, 0(x10)
    .ICA_END
```

Listing 3.1: Example usage of the .ICA instruction

The example above illustrates how an update to shared memory is encapsulated within an atomic section. If a power failure occurs during execution, the update is discarded and retried after restoration, ensuring consistency.

3.2.2 Semantics and Guarantees

The `.ICA` instruction provides the following guarantees:

- **Atomicity:** Operations inside an ICA block either complete fully or are discarded.

- **Idempotence:** Re-executions of ICA blocks after a failure yield the same result.

- **Portability:** ICA primitives are defined at the ISA level, enabling software portability across different RISE-enabled RISC-V cores.

These semantics closely resemble transactional execution, but are optimized for the intermittent computing setting.

3.3 Hardware Modules

The hardware extensions of RISE are distributed across four specialized modules. Each module is designed to be lightweight, portable, and implementable with standard CMOS processes.

3.3.1 Intermittent Computing Register Wrapper (ICRW)

The ICRW is responsible for tracking and backing up processor registers. It introduces a **dirty-bit mechanism** that allows selective backup of modified registers, avoiding unnecessary writes to non-volatile memory. Each register can be in one of four states: *CLEAN*, *DIRTY*, *BACKUP*, or *MODIFIED*. The state transitions are triggered by read/write operations and backup events, as shown in Figure 4.3.

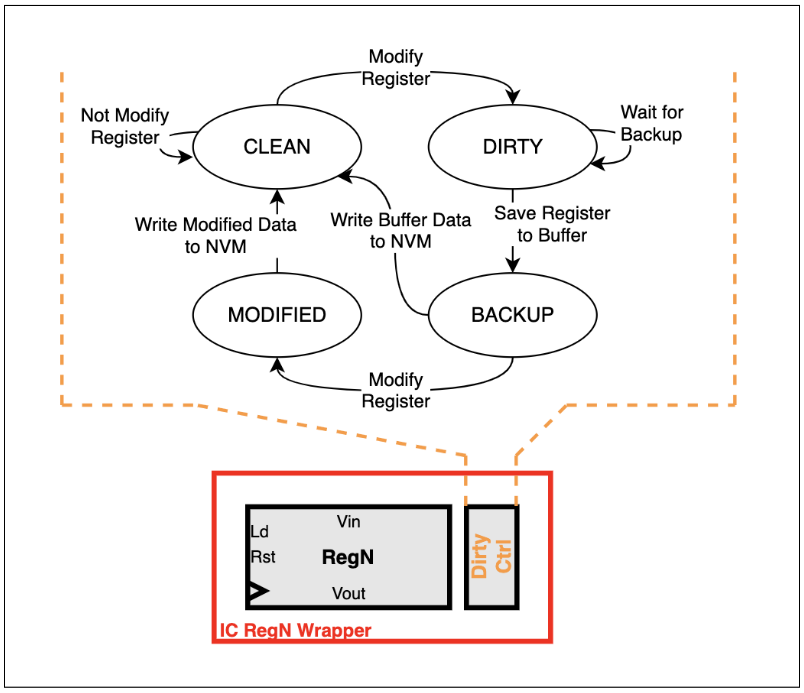


Figure 3.2: Finite-state machine of the ICRW register states (CLEAN, DIRTY, BACKUP, MODIFIED).

By reducing the volume of data to be saved during each power failure, the ICRW drastically lowers backup latency and energy consumption compared to full checkpointing approaches.

3.3.2 Power Control Unit (PCU)

The PCU continuously monitors the energy buffer and detects imminent power loss. Upon detecting a low-energy event, the PCU triggers the backup process by signaling the Dispatcher. The PCU also manages a temporary buffer for just-in-time register updates, ensuring that backup operations are aligned with the actual availability of residual energy.

This predictive mechanism avoids both premature and delayed backups, striking a balance between reliability and efficiency.

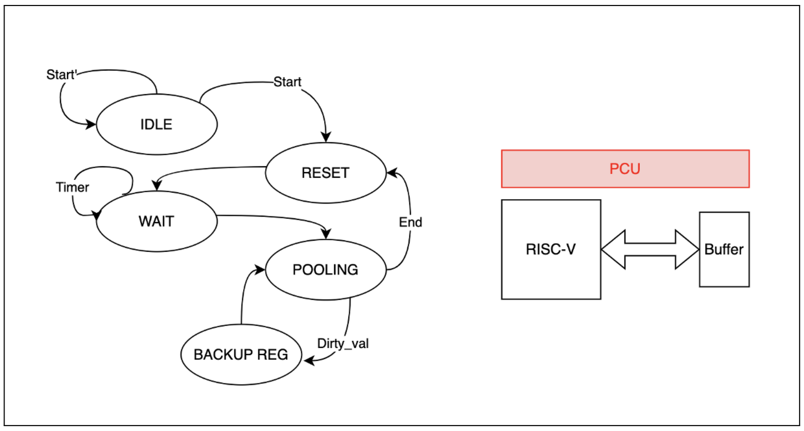


Figure 3.3: Finite-state machine of the PCU.

3.3.3 Restore Control Unit (RCU)

The RCU is responsible for restoring processor state after a power failure. It retrieves the backed-up register contents from the temporary buffer or non-volatile memory and reinitializes the processor pipeline. To minimize latency, the RCU supports **direct memory mapping** with virtual addresses, reducing the number of required cycles during recovery.

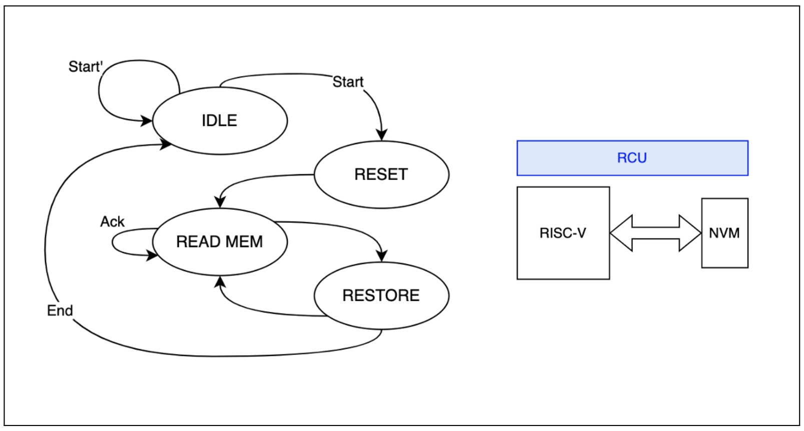


Figure 3.4: Finite-state machine of the RCU.

The RCU guarantees correctness by ensuring that execution resumes only after all registers and memory have been safely restored.

3.3.4 Dispatcher

The Dispatcher coordinates the interaction between ICRW, PCU, and RCU. It acts as a lightweight controller that arbitrates access to the backup bus and schedules backup operations. Because it centralizes control, the Dispatcher enables modularity and portability, allowing RISE to be integrated into different RISC-V cores without invasive redesign.

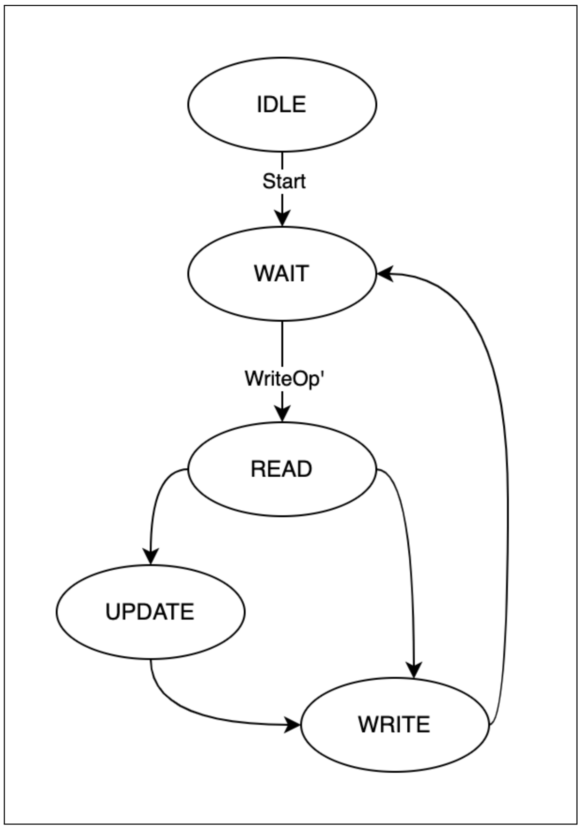


Figure 3.5: Finite-state machine of the Dispatcher.

3.4 Backup and Restore Workflow

The complete backup and restore mechanism in RISE unfolds as follows:

1. **Normal Execution:** Registers are updated and marked dirty by the ICRW.

2. **Imminent Power Loss:** The PCU detects low energy and signals the Dispatcher.

3. **Backup:** The ICRW writes dirty registers into the backup buffer, coordinated by the Dispatcher.

4. **Power Failure:** The device shuts down, but critical state is preserved.

5. **Restore:** Upon power-up, the RCU reloads registers from backup into the pipeline.

6. **Resumption:** Execution continues from the last atomic section boundary defined by `.ICA`.

This workflow ensures that only minimal and necessary state is preserved, drastically reducing overhead.

3.5 Integration with RISC-V Pipeline

RISE is designed to integrate seamlessly with the baseline RV32I pipeline. The extensions operate orthogonally to standard instruction execution, avoiding invasive changes to the datapath. Backup and restore signals are decoupled from the main pipeline stages, allowing reuse of standard RISC-V cores.

Figure **??** shows a conceptual view of how RISE modules interact with the pipeline.

3.6 Design Considerations and Trade-offs

The design of RISE reflects several important trade-offs:

- **Area vs. Efficiency:** Adding ICRW and Dispatcher logic introduces area overhead, but significantly reduces backup latency.

- **Portability vs. Specialization:** RISE is designed to be portable across different cores, but avoids optimizations that would lock it to specific designs.

- **Energy vs. Correctness:** The system prioritizes correctness (atomicity, consistency) even at the cost of slight energy overheads during backup.

- **Scalability:** While demonstrated on RV32I, RISE can be extended to multi-core settings and more complex pipelines.

3.7 Summary

This chapter introduced the RISE architecture, a modular extension to RISC-V designed to support reliable intermittent execution. By combining ISA primitives with lightweight hardware modules, RISE addresses the limitations of checkpointing, task-based execution, and non-volatile processors. The next chapter details the implementation of RISE and its integration with a RISC-V pipeline.

Chapter 4

Implementation

4.1 Introduction

This chapter describes the implementation of the RISE framework, focusing on the integration of its hardware modules into a baseline RISC-V pipeline. The design has been realized in Verilog HDL, synthesized and simulated using standard tools. In the following sections, we provide details about the baseline RISC-V core, the implementation of each RISE module, and their integration with the pipeline.

4.2 Baseline RISC-V Core

The implementation of RISE is based on a baseline RISC-V core that follows the classic five-stage pipeline organization. The processor supports the **RV32I instruction set architecture (ISA)**, which defines a simple yet complete set of 32-bit integer instructions suitable for embedded and IoT applications. The choice of RV32I ensures compatibility with widely available RISC-V toolchains while maintaining low hardware complexity.

4.2.1 Pipeline Organization

The pipeline is divided into the following five stages:

1. **Instruction Fetch (IF):** The program counter (PC) provides the address of the next instruction to be executed. This instruction is fetched from the instruction memory and stored in the instruction register. In this stage, the PC is updated to point to the subsequent instruction, unless a control transfer (e.g., branch or jump) modifies the normal sequential flow.

2. **Instruction Decode (ID):** The fetched instruction is decoded to determine the operation type and the required operands. Source registers are read from the register file, and immediate values are extracted. During this stage, the control unit generates the control signals that guide execution in later stages.

3. **Execute (EX):** In this stage, the Arithmetic Logic Unit (ALU) performs the required computation. Depending on the instruction, the ALU may execute arithmetic or logical operations, calculate memory addresses for load/store instructions, or evaluate branch conditions. The EX stage also computes the potential target address for branch and jump instructions.

4. **Memory Access (MEM):** For load and store instructions, the effective memory address calculated in the EX stage is used to access the data memory. In the case of a load, data is retrieved and forwarded to the next stage; for a store, the data is written to memory. For other instruction types, this stage is effectively bypassed.

5. **Write-Back (WB):** In the final stage, results from the ALU (EX stage) or from data memory (MEM stage) are written back into the destination register in the register file. This ensures that the architectural state is updated consistently with the instruction semantics.

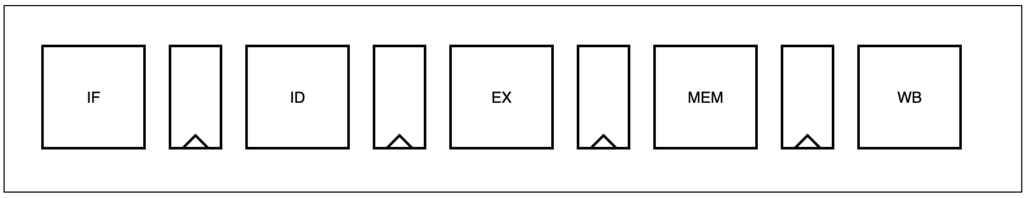
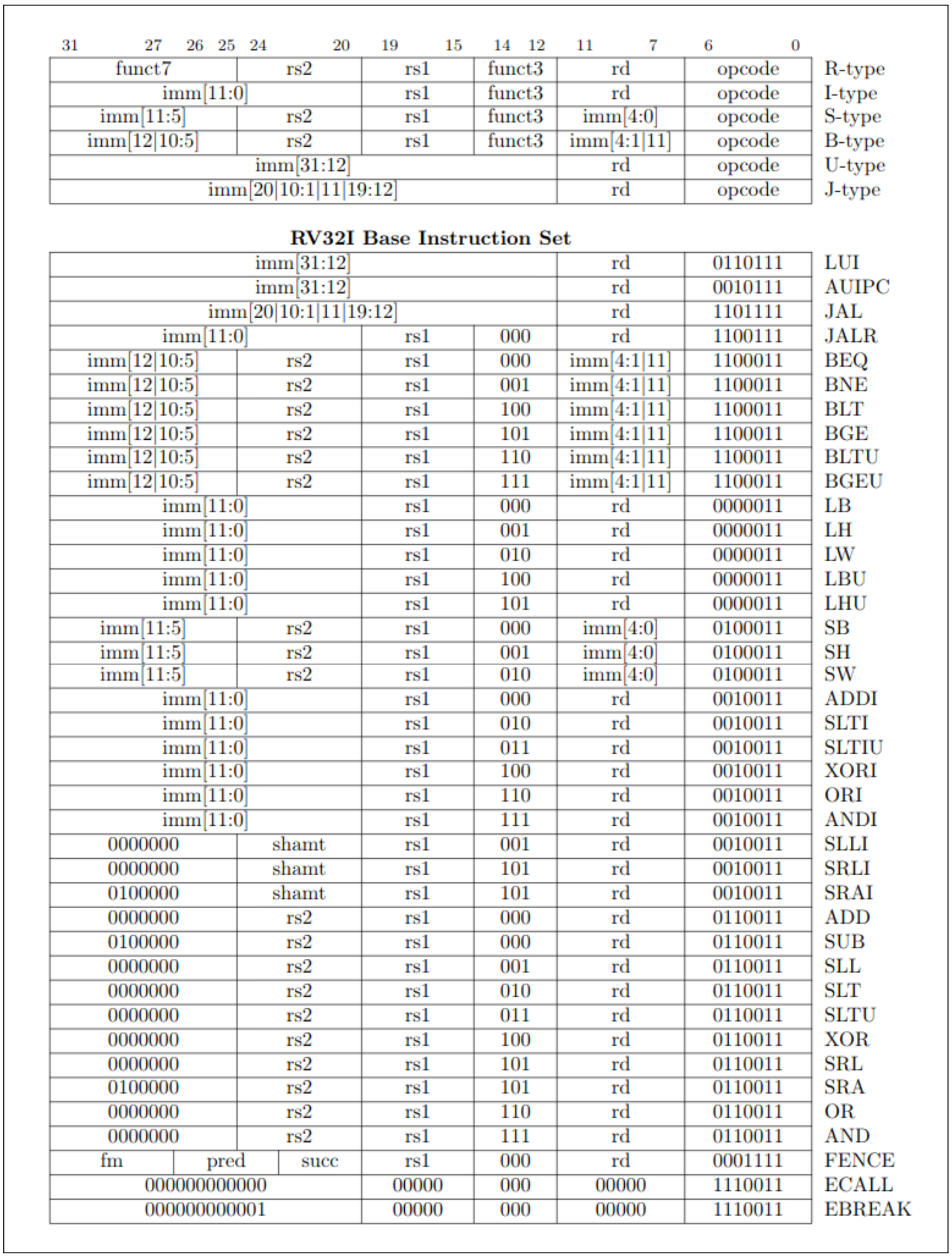


Figure 4.1: RISC-V pipeline.

4.2.2 Instruction Set Coverage

The core fully supports the RV32I base ISA, which includes:

- Integer arithmetic and logic operations (e.g., `ADD`, `SUB`, `AND`, `OR`).
- Control transfer instructions, including conditional branches (`BEQ`, `BNE`, etc.), jumps (`JAL`, `JALR`), and system calls.
- Load and store instructions for accessing memory with byte, half-word, and word granularity.
- Immediate instructions for arithmetic and logic with constant operands.



31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm pred	succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Figure 4.2: RV32I Instruction Set Architecture.

4.2.3 Pipeline Hazards and Assumptions

In the current implementation, the following considerations apply:

- **Data hazards:** Data dependencies are not resolved through hardware forwarding or hazard detection. Instead, they are managed at the software level by the compiler, which introduces explicit no-operation (`NOP`) instructions when required. This approach simplifies the hardware but may increase the number of cycles for dependent instruction sequences.
- **Control hazards:** Branch decisions are resolved in the **ID stage**, rather than in EX. This design choice reduces the branch delay slot to a single instruction. As a result, only one instruction following the branch needs to be annulled in case of a misprediction, improving efficiency compared to later resolution.
- **Structural hazards:** The design assumes separate instruction and data memories (Harvard architecture), thereby avoiding contention between instruction fetch and data access.

4.2.4 Integration with RISE

This baseline RV32I pipeline provides the foundation on which RISE modules are integrated. Because the core adheres to the modular RISC-V design principles, only minimal modifications are required in the decode and control logic to support the new `.ICA` instruction, while the ICRW, PCU, RCU, and Dispatcher operate largely as orthogonal hardware extensions.

4.2.5 Toolchain and Synthesis Environment

The development and evaluation of the RISE framework were carried out using industry-standard EDA tools in a heterogeneous software environment. The following setup was adopted:

- **Design and Simulation:** The RISC-V core and the RISE modules were implemented in Verilog HDL. For the design and functional verification, **Xilinx Vivado™ 2024.1** was employed. Vivado was used both for RTL simulation and synthesis, providing detailed reports on timing, resource utilization, and estimated power consumption.
- **Operating Systems:** Two operating systems were used during the development process:
  - **Windows 11** – primarily for design entry, synthesis runs, and waveform inspection through the Vivado GUI.
  - **Ubuntu 20.04.6 LTS** – for command-line simulation, benchmarking automation, and integration with open-source RISC-V toolchains.

This dual-OS workflow allowed leveraging both graphical and script-driven flows for efficient verification.

- **RISC-V Toolchain:** The GNU RISC-V toolchain (`riscv32-unknown-elf-gcc`) was used to compile and run benchmarks. Assembly and C programs were compiled into binary executables, which were then loaded into the simulated instruction memory of the RV32I pipeline. This setup ensured full compliance with the RV32I specification and facilitated direct execution of MiBench and ISA compliance tests.

- **Benchmarking Framework:** The performance evaluation was based on simulation waveforms and cycle-accurate logs produced by Vivado. These were post-processed using Python scripts to measure backup/restore latencies, energy estimates, and forward progress under intermittent execution.

Overall, the combination of **Vivado 2024.1**, the GNU RISC-V toolchain, and a dual-OS development environment provided a robust platform for both design and benchmarking. This toolchain ensured reproducibility of results while maintaining compatibility with standard academic and industrial workflows.

4.3 System Overview

The proposed architecture builds upon a baseline five-stage RISC-V processor implementing the RV32I instruction set. The design has been enhanced to support intermittent execution in energy-harvesting scenarios, where frequent and unpredictable power interruptions occur. Unlike prior approaches that require intrusive modifications to the processor pipeline or rely on non-volatile flip-flops (NVFFs), this solution maintains compatibility with standard CMOS flows, ensuring portability and scalability across different RISC-V cores.

The core idea is to expose the internal state of the processor through lightweight wrappers and to manage backup and restore operations via external control units. This decoupled approach ensures that the original datapath and control remain intact, while backup and restore logic is modular and easily integrated. The framework introduces four new hardware components:

- **Intermittent Computing Register Wrapper (ICRW)** – encapsulates each register to track modifications and support backup/restore operations.

- **Power Control Unit (PCU)** – periodically scans ICRWs and maintains a volatile buffer of dirty states.

- **Restore Control Unit (RCU)** – reloads the saved state into registers after a power failure.

- **Dispatcher** – arbitrates access to the shared memory bus between normal execution and backup transfers.

Additionally, the ISA is extended with the `.ICA` instruction to define atomic execution regions, providing correctness guarantees under intermittent conditions.

The following sections present the Verilog implementation of each module, followed by integration into the pipeline, project organization, and a discussion of design metrics.

4.4 Implementation of RISE Modules

This section details the Verilog implementation of the four main hardware modules of RISE: ICRW, PCU, RCU, and Dispatcher.

4.4.1 Intermittent Computing Register Wrapper (ICRW)

The ICRW is the fundamental building block enabling backup and restore. Its purpose is to make the processor's internal state externally accessible without altering the behavior of the original registers.

Each register is encapsulated by an ICRW instance, which preserves the standard interface (`Clk`, `Rst`, `Vin`, `Vout`) while adding additional signals for backup and restore. Internally, the ICRW maintains a finite-state machine (FSM) to track the consistency of the register value with its stored backup.

The FSM supports four states:

- **CLEAN:** register content has been correctly saved in memory and not modified.
- **DIRTY:** register has been updated, requiring a new backup.
- **BACKUP:** register has been read and placed in a temporary buffer, awaiting commit to memory.
- **MODIFIED:** register has been updated after a backup, requiring another read.

This FSM ensures selective and efficient backup operations. Registers in the CLEAN state are ignored, while only DIRTY or MODIFIED registers are saved. This reduces the frequency of memory transactions, mitigating energy and latency overhead.

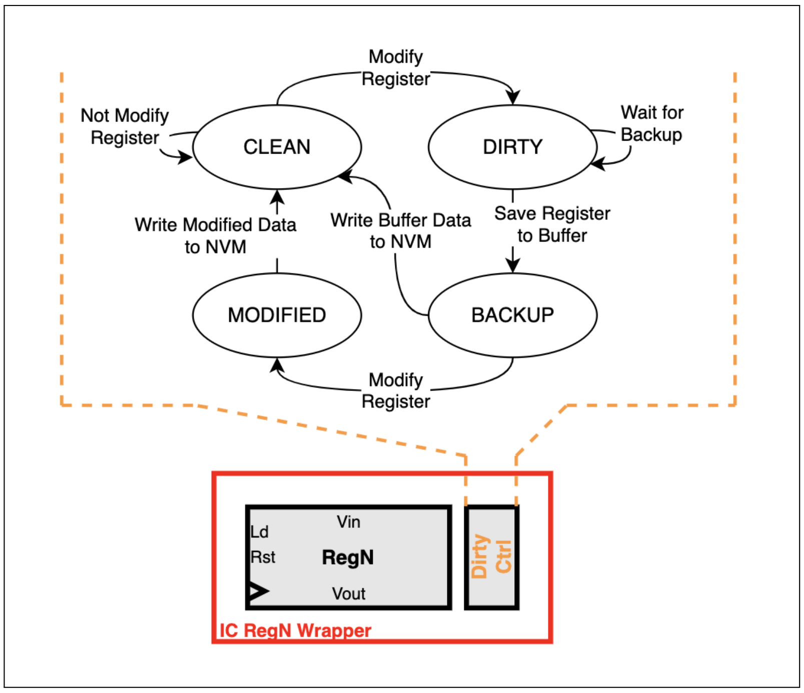


Figure 4.3: Finite-state machine of the ICRW register states (CLEAN, DIRTY, BACKUP, MODIFIED).

```
module RegN_IC_Wrapper #(parameter N = 32) (
        Ld,
        Vin,
        Vout,
        Dirty_val,
        Backup_en,
        Backup_ack,
        Backup_Vout,
        Rst_DrtyCtrl,
        Restore_en,
        Restore_Vin,
        Rst,
        Clk,
        Pwr_off
    );

    // register
    input Ld, Rst, Clk;
    input [N-1:0] Vin;
    output [N-1:0] Vout;

    // Dirty bit FSM
    input Backup_en, Backup_ack;
    output [1:0] Dirty_val;
    output [N-1:0] Backup_Vout;
    input Rst_DrtyCtrl;

    // restore signals
    input [N-1:0] Restore_Vin;
    input Restore_en;

    // intermittent computing simulation
    input Pwr_off;

    wire [N-1:0] Vin_wire, Vin_wire_reg;
    wire [N-1:0] Vout_wire;

    wire Ld_reg_wire;
    wire Rst_reg_wire;

    wire cmp_res;

    wire Rst_DrtyCtrl_wire;

    wire Ld_wire;


    RegN #(
            .N            (N)
    ) register_n (
            .Vin          (Vin_wire_reg),
            .Vout         (Vout_wire),
            .Ld           (Ld_wire),
            .Rst          (Rst),
            .Clk          (Clk),
            .Pwr_off      (Pwr_off)
    );

    DirtyCtrl dirty_controller (
            .Ld_reg       (Ld_reg_wire),
            .Rst_reg      (Rst_reg_wire),
            .Backup_en    (Backup_en),
            .Backup_ack   (Backup_ack),
            .Clk          (Clk),
            .Rst          (Rst_DrtyCtrl_wire),
            .Dirty_val    (Dirty_val),
            .Pwr_off      (Pwr_off)
    );

    CmpN #(
            .N            (N)
    ) comparator (
            .Vin_a      (Vin_wire),
            .Vin_b      (Vout_wire),
            .Vout       (cmp_res)
    );

    MuxN_21 #(
            .N          (N)
    ) multiplexer_restore (
            .Vin_a      (Vin_wire),  // sel = 0
            .Vin_b      (Restore_Vin),  // sel = 1
            .sel        (Restore_en),
            .Vout        (Vin_wire_reg)
    );



    assign Backup_Vout = Vout_wire;
    assign Vout = Vout_wire;
    assign Vin_wire = Vin;

    assign Ld_reg_wire  = Ld  & ~cmp_res;
    assign Rst_reg_wire = Rst & ~cmp_res;

    assign Rst_DrtyCtrl_wire = Rst_DrtyCtrl | Restore_en;
    assign Ld_wire = Ld | Restore_en;



endmodule
```

Listing 4.1: ICRW Verilog implementation

4.4.2 Power Control Unit (PCU)

The PCU complements the core control logic by orchestrating backup operations across all ICRWs. When enabled, it periodically scans the ICRWs, checking their dirty bits. If a register is DIRTY, the PCU retrieves its value, stores it in a temporary volatile buffer, and updates its state to BACKUP.

This polling mechanism is configurable, allowing adaptation to different workloads. The scan frequency can be tuned based on the number of ICRWs and the system's performance requirements. By decoupling the snapshot from the final commit, the PCU ensures that critical state is preserved even under heavy bus contention.

Moreover, the PCU includes a just-in-time check: before committing a buffered value, it verifies if the register has transitioned to MODIFIED. If so, the buffer is updated accordingly, ensuring correctness without re-running a full scan.

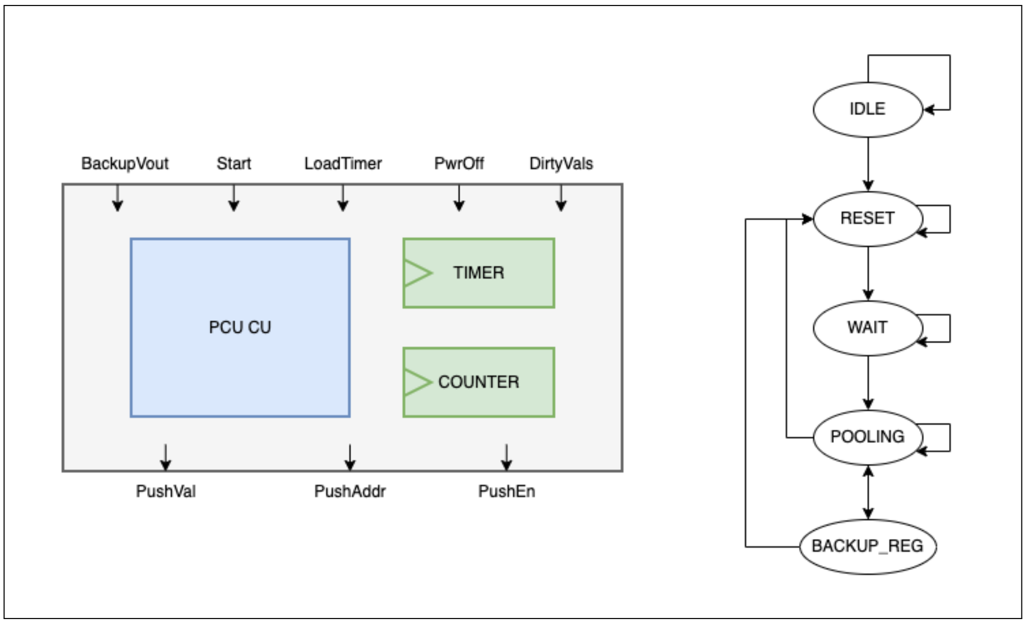


Figure 4.4: PCU architecture.

```
module PCU #(
        parameter K = 10,  // number of IC_Reg_Wrapeer
        parameter N = 32,  // width of IC_Reg_Wrapper
        parameter M = 32  // width timer value register
    ) (
        Backup_Vout_IC_Reg_Wrapper,
        Start_FSM_PCU,
        PushVal_Buffer,
        Load_Timer,
        PushEn_Buffer,
        backup_now_ctrl, // start backup now
        Dirty_vals_IC_Reg_Wrapper, // K = number of Wrappers
        Rst_Buffer,
        Backup_Ens_IC_REG_Wrapper, // K = number of Wrappers
        IsFull_Buffer,
        Clk,
        Rst,
        Pwr_off
    );

    localparam LOG2_K = $clog2(K);


    input [(K*N)-1:0] Backup_Vout_IC_Reg_Wrapper;
    input Start_FSM_PCU;
    input [M-1:0] Load_Timer;
```

```verilog
    input [(K*2)-1:0] Dirty_vals_IC_Reg_Wrapper;
    input IsFull_Buffer;
    input backup_now_ctrl;

    output [N+LOG2_K-1:0] PushVal_Buffer;
    output PushEn_Buffer;
    output Rst_Buffer;
    output [K-1:0] Backup_Ens_IC_REG_Wrapper;

    input Pwr_off;
    input Rst;
    input Clk;

    wire last_wire;
    wire end_wire;
    wire end_timer_wire;
    wire [1:0] dirty_val_wire;
    wire Rst_CntN_wire;
    wire Rst_Timer_wire;
    wire En_Timer_wire;
    wire Clk_CntN_wire;

    wire [LOG2_K-1:0] addr_wrapper;
    wire [LOG2_K-1:0] addr_wrapper_sub;
    wire [N-1:0] push_val_buffer_wire;

    wire [K-1:0] Backup_Ens_IC_REG_Wrapper_wire;

    assign end_wire = end_timer_wire | backup_now_ctrl;

    // FSM
    FSM_PCU fsm_power_cu (
        .Start              (Start_FSM_PCU),
        .IsFull_Buffer      (IsFull_Buffer),
        .Last               (last_wire),
        .End_Timer          (end_wire),
        .DirtyValSel        (dirty_val_wire),
        .Rst_Buffer         (Rst_Buffer),
        .Rst_CntN           (Rst_CntN_wire),
        .Rst_Timer          (Rst_Timer_wire),
        .En_Timer           (En_Timer_wire),
        .Clk_CntN           (Clk_CntN_wire),
        .PushEn_Buffer      (PushEn_Buffer),
        .Rst                (Rst),
        .Clk                (Clk),
        .Pwr_off            (Pwr_off)
    );

    // Timer
    Timer #(
        .N                      (M)
    ) timer_pcu (
        .En                     (En_Timer_wire),
        .Load                   (Load_Timer),
        .Clk                    (Clk),
        .Rst                    (Rst_Timer_wire),
        .End                    (end_timer_wire),
        .Pwr_off                (Pwr_off)
    );

    // CntN
    CntN #(
        .N                      (LOG2_K)
    ) counter_backup_addr (
        .Clk                    (Clk_CntN_wire),
        .Rst                    (Rst_CntN_wire),
        .Pwr_off                (Pwr_off),
        .Vout                   (addr_wrapper)
    );

    // Decoder
    DecN #(
        .N                      (LOG2_K)
    ) dec_backup_en (
        .Vin                    (addr_wrapper_sub),
        .Vout                   (Backup_Ens_IC_REG_Wrapper_wire)
    );

    assign Backup_Ens_IC_REG_Wrapper =
   Backup_Ens_IC_REG_Wrapper_wire & {N{PushEn_Buffer}};

    // Multiplexer
    MuxM_N1 #(
        .N                      (K),
        .M                      (2)
    ) mux_dirty_val (
        .Vin                    (Dirty_vals_IC_Reg_Wrapper),
        .Sel                    (addr_wrapper_sub),
        .Vout                   (dirty_val_wire)
    );

    // sub by 1
    Sub1 #(
        .N                      (LOG2_K)
    ) sub1_addr (
        .Vin                    (addr_wrapper),
        .Vout                   (addr_wrapper_sub)
    );

    // and signal Last
    CmpN_M #(
        .N      (LOG2_K),
        .M      (K)
    ) cmp_last_addr_wrapper (
        .Vin_a  (addr_wrapper),
        .Vout   (last_wire)
    );

    // mux Backup vals
    MuxM_N1 #(
        .N                      (K),
        .M                      (N)
    ) mux_backup_val (
        .Vin                    (Backup_Vout_IC_Reg_Wrapper),
        .Sel                    (addr_wrapper_sub),
        .Vout                   (push_val_buffer_wire)
    );

    assign PushVal_Buffer = {push_val_buffer_wire,
   addr_wrapper_sub};


endmodule
```

Listing 4.2: PCU Verilog implementation

4.4.3 Restore Control Unit (RCU)

The RCU manages recovery after a power failure. Its role is to reload the processor state from memory into the ICRWs, restoring execution to the last valid snapshot. Each ICRW is associated with a virtual address in memory, which the RCU uses to fetch the corresponding value.

The RCU interfaces with the ICRWs through dedicated signals, ensuring direct updates to sequential registers. Once the restore is complete, the RCU deactivates itself until the next power failure, minimizing overhead during normal execution.

The restore performance is primarily limited by the characteristics of the external memory. In practice, fast SRAM or hybrid memory solutions can reduce recovery latency significantly.

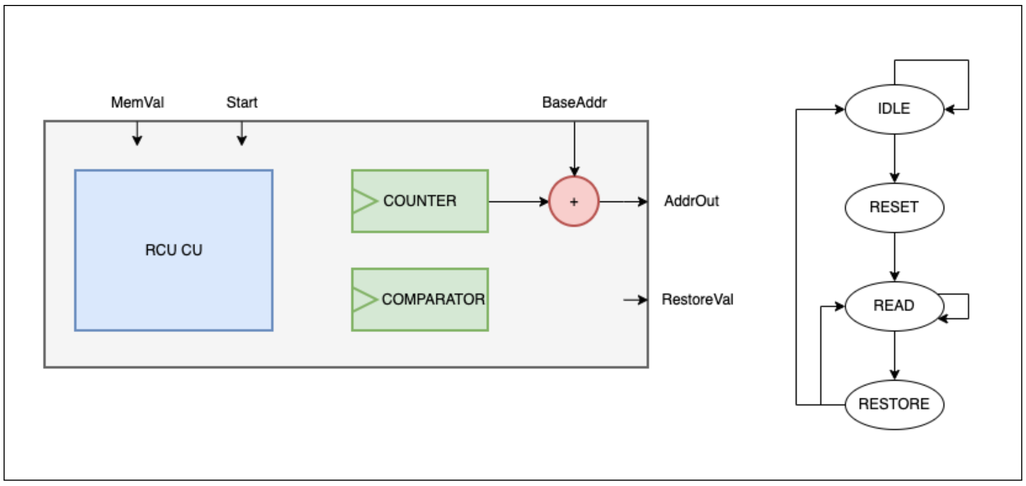


Figure 4.5: RCU architecture.

```
module RCU #(
        parameter N = 10,   // Number of IC wrapper
        parameter K = 32,   // size base address
        parameter M = 32    // width IC wrapper
    ) (
        AckMem,
        Start,
        ReadMem,
        BaseAddr,
        AddrMem,
        ValMem,
        RestoreVal,
        RestoreEn,
        Clk,
        Rst,
        Pwr_off
    );

    input AckMem;
    input Start;
    input [K-1:0] BaseAddr;
    input [M-1:0] ValMem;
    input Clk;
    input Rst;
    input Pwr_off;

    output [K-1:0] AddrMem;
    output ReadMem;
    output [M-1:0] RestoreVal;
    output [N-1:0] RestoreEn;

    localparam LOG2_N = $clog2(N);


    wire rst_cnt_wire;
    wire en_cnt_wire;
    wire [LOG2_N-1:0] vout_cnt_wire;
    wire [N-1:0] dec_out_wire;
    wire [LOG2_N-1:0] vout_sub_wire;
    wire end_wire;
    wire clk_cnt;
    wire restore_vin_en_wire;
    wire restore_dec_en_wire;
    wire [K-1:0] input_a_adder;

    CmpN_M #(
        .N                  (LOG2_N),
        .M                  (N)
    ) cmp_addr (
        .Vin_a              (vout_sub_wire),
        .Vout               (end_wire)
    );


    assign input_a_adder = {{(K-LOG2_N){1'b0}}, vout_cnt_wire};

    Adder #(
        .N                  (K)
    ) adder_base_addr (
        .A                  (input_a_adder),
        .B                  (BaseAddr),  // K bits
        .Cin                (1'b0),
        .Cout               (),
        .S                  (AddrMem)
    );

    Sub1 #(
        .N                  (LOG2_N)
    ) sub_cnt (
        .Vin                (vout_cnt_wire),
        .Vout               (vout_sub_wire)
    );

    assign clk_cnt = en_cnt_wire | rst_cnt_wire;

    CntN #(
        .N                  (LOG2_N)
    ) local_addr_cnt (
        .Clk                (clk_cnt),
        .Rst                    (rst_cnt_wire),
        .Pwr_off                (Pwr_off),
        .Vout                   (vout_cnt_wire)
    );

    DecN #(
        .N                      (LOG2_N)
    ) dec_restore_en (
        .Vin                    (vout_sub_wire),
        .Vout                   (dec_out_wire)
    );

    FSM_RCU fsm_rcu (
        .Start                  (Start),
        .AckMem                 (AckMem),
        .ReadEn                 (ReadMem),
        .RstCnt                 (rst_cnt_wire),
        .EnCnt                  (en_cnt_wire),
        .EnDec                  (restore_dec_en_wire),
        .End                    (end_wire),
        .Restore_VinEn          (restore_vin_en_wire),
        .Pwr_off                (Pwr_off),
        .Rst                    (Rst),
        .Clk                    (Clk)
    );

    // enable restore_vin
    assign RestoreVal = ValMem & {M{restore_vin_en_wire}};

    // enable restore decoder
    assign RestoreEn = dec_out_wire & {N{restore_dec_en_wire}};

endmodule
```

Listing 4.3: RCU Verilog implementation

4.4.4 Dispatcher

Backup operations must share the memory bus with normal load/store instructions from the pipeline, introducing potential conflicts. The Dispatcher resolves this issue by scheduling backup writes only when the bus is idle.

By monitoring memory activity, the Dispatcher ensures that program execution is prioritized. Backup writes are deferred until idle cycles, at which point they are committed sequentially. This arbitration mechanism is transparent to the pipeline and avoids performance degradation.

The main limitation is the variable latency between snapshot capture and commit, which depends on workload memory intensity. Nevertheless, correctness is preserved, and throughput remains optimized.

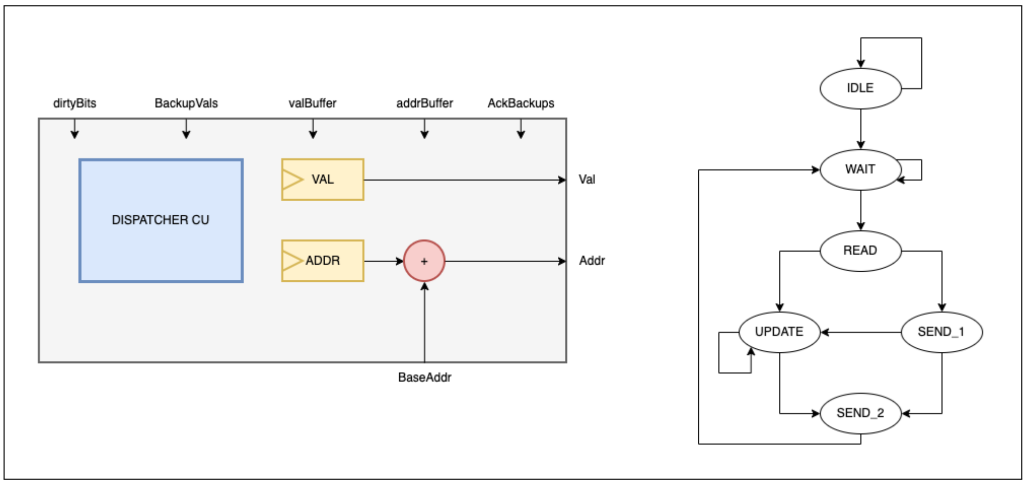


Figure 4.6: Dispatcher architecture.

```
module Dispatcher #(
        parameter K = 10,    // number of IC_REG_WRAPPERS
        parameter N = 32,    // width IC_REG_WRAPPER
        parameter M = 32     // width base address
    ) (
        Start,
        IsEmpty,
        WriteOp,
        DirtyBits,
        BackupVals,
        ValBuffer,
        AddrBuffer,
        BaseAddr,
        Rst,
        Clk,
        Pwr_off,
        Val,
        Addr,
        WriteEn,
        AckBackups,
        PullEn
    );

    localparam LOG2_K = $clog2(K);

    input Start;
    input IsEmpty;
    input WriteOp;
    input [(K*2)-1:0] DirtyBits;
    input [(K*N)-1:0] BackupVals;
    input [N-1:0] ValBuffer;
    input [LOG2_K-1:0] AddrBuffer;
    input [M-1:0] BaseAddr;
    input Rst;
    input Clk;
    input Pwr_off;

    // outputs
    output [N-1:0] Val;
    output [M-1:0] Addr;
    output WriteEn;
    output [K-1:0] AckBackups;
    output PullEn;


    wire [1:0] DirtyVal_wire;
    wire RstVal_wire;
    wire RstAddr_wire;
    wire LdVal_wire;
    wire LdAddr_wire;
    wire SelVal_wire;
    wire EnAck_wire;
    wire EnBuff_wire;
    wire [N-1:0] backupVal_wire;
    wire [(N*2)-1:0] choose_val_vin;
    wire [N-1:0] val_sel_wire;
    wire [N-1:0] vout_val_wire;
    wire [LOG2_K-1:0] vout_addr_wire;
    wire [M-1:0] vout_actual_addr_wire;
    wire [K-1:0] ack_sigs_wire;

    // FSM
    FSM_Dispatcher FSM_dispatcher (
        .Start                (Start),
        .IsEmpty              (IsEmpty),
        .WriteOp              (WriteOp),
        .DirtyVal             (DirtyVal_wire),
        .PullEn               (PullEn),
        .RstVal               (RstVal_wire),
        .RstAddr              (RstAddr_wire),
        .LdVal                (LdVal_wire),
        .LdAddr               (LdAddr_wire),
        .SelVal               (SelVal_wire),
        .EnAck                (EnAck_wire),
        .EnBuff               (EnBuff_wire),
        .Pwr_off              (Pwr_off),
        .Rst                    (Rst),
        .Clk                    (Clk)
    );

    // mux dirty bits
    MuxM_N1 #(
        .N                      (K),
        .M                      (2)
    ) mux_dirty_bits (
        .Vin                    (DirtyBits),
        .Sel                    (AddrBuffer),
        .Vout                   (DirtyVal_wire)
    );

    // mux backup vals
    MuxM_N1 #(
        .N                      (K),
        .M                      (N)
    ) mux_backup_vals (
        .Vin                    (BackupVals),
        .Sel                    (vout_addr_wire),
        .Vout                   (backupVal_wire)
    );


    assign choose_val_vin = {backupVal_wire, ValBuffer};
    // mux sel val buffer
    MuxM_N1 #(
        .N                      (2),
        .M                      (N)
    ) mux_val_buffer (
        .Vin                    (choose_val_vin),
        .Sel                    (SelVal_wire),
        .Vout                   (val_sel_wire)
    );

    // reg val
    RegN #(
        .N                      (N)
    ) reg_val (
        .Vin                    (val_sel_wire),
        .Vout                   (vout_val_wire),
        .Ld                     (LdVal_wire),
        .Rst                    (RstVal_wire),
        .Clk                    (Clk),
        .Pwr_off                (Pwr_off)
    );

    // reg addr
    RegN #(
        .N                      (LOG2_K)
    ) reg_addr_buf (
        .Vin                    (AddrBuffer),
        .Vout                   (vout_addr_wire),
        .Ld                     (LdAddr_wire),
        .Rst                    (RstAddr_wire),
        .Clk                    (Clk),
        .Pwr_off                (Pwr_off)
    );

    // adder base addr + buff addr
    Adder #(
        .N                      (M)
    ) adder_addr (
        .A                      ({{(M-LOG2_K){1'b0}}, vout_addr_wire})
    ,
        .B                      (BaseAddr),
        .Cin                    (1'b0),
        .Cout                   (),
        .S                      (vout_actual_addr_wire)
    );

    // buffer 3 states output
    TriBuff #(
        .N                      (N)
    ) buff_3s_mem_interface_val (
        .Vin                    (vout_val_wire),
        .En                     (EnBuff_wire),
        .Vout                   (Val)
    );

    TriBuff #(
        .N                      (M)
    ) buff_3s_mem_interface_addr (
        .Vin                    (vout_actual_addr_wire),
        .En                     (EnBuff_wire),
        .Vout                   (Addr)
    );

    TriBuff #(
        .N                      (1)
    ) buff_3s_mem_interface_write_en (
        .Vin                    (EnBuff_wire),
        .En                     (EnBuff_wire),
        .Vout                   (WriteEn)
    );

    // decoder ack backup
    DecN #(
        .N                    (LOG2_K)
    ) decoder_ack_backup (
        .Vin                  (vout_addr_wire),
        .Vout                 (ack_sigs_wire)
    );

    // en ack backup
    assign AckBackups = ({K{EnAck_wire}} & ack_sigs_wire);



endmodule
```

Listing 4.4: Dispatcher Verilog implementation

4.4.5 Atomic Execution with `.ICA` Instruction

One of the key contributions of RISE is the extension of the RISC-V ISA with the `.ICA` instruction (**Intermittent Computing Atomic**). This instruction enables programmers to explicitly delimit code regions that must be executed atomically, ensuring both correctness and consistency under intermittent power conditions.

Instruction Semantics

The `.ICA` primitive is not a traditional instruction with an immediate effect on registers or memory. Instead, it acts as a *tag* that defines the boundaries of an atomic region. Two instructions are introduced:

- `ICA_START`: marks the beginning of an atomic region.
- `ICA_END`: marks the end of the same region.

The code enclosed between these two markers is treated as an indivisible block of execution. Either the entire region executes successfully, or no visible state changes occur in external memory. This ensures that critical computations, such as sensor sampling or actuation logic, are never left in a partially completed state.

Pipeline Integration

Support for the `.ICA` instruction required extending the decode stage of the baseline RV32I pipeline. When the decoder identifies an `ICA_START`, the following operations are triggered:

1. The Power Control Unit (PCU) is disabled, preventing background backup operations during atomic execution.

2. A full snapshot of the processor's internal state is taken using the ICRW modules and stored in memory.
3. A log buffer is activated to record all subsequent external write operations.

When `ICA_END` is encountered, the buffered writes are validated and committed atomically to the memory system, after which the PCU is re-enabled.

Execution Flow

The flow of operations during atomic execution is illustrated below:

1. **Entry (ICA_START):**
   - Save complete processor state (snapshot).
   - Suspend PCU activity to reduce energy consumption.
   - Redirect memory writes to the internal log buffer.
2. **Atomic Region Execution:**
   - Instructions execute normally, but external writes are intercepted.
   - Buffered writes are staged in log memory until completion.
3. **Exit (ICA_END):**
   - If uninterrupted, flush buffered writes to external memory in a single batch.
   - PCU is re-enabled, and normal operation resumes.
4. **Failure Handling:**
   - If a power failure occurs mid-region, the processor restores the snapshot saved at entry.
   - The region is re-executed entirely, ensuring atomicity and consistency.

Pseudocode Example

The following pseudocode illustrates the programmer's view of the `.ICA` primitive:

```
ICA_START;   // Begin atomic region

sensor_value = read_sensor();
processed = process(sensor_value);
write_to_output(processed);

```

```
ICA_END;      // End atomic region
```

Listing 4.5: Pseudocode example of atomic execution using .ICA tags

In this example, the sensor reading, computation, and output update form a critical block. Even if power fails after the sensor is read but before the output is written, the system restores to the snapshot at `ICA_START` and re-executes the region, guaranteeing correctness.

Advantages and Limitations

The main advantage of the `.ICA` mechanism is that it provides explicit software control over atomicity, without requiring complex compiler analyses or opaque runtime systems. It ensures:

- Strong correctness guarantees across intermittent power failures.
- Energy efficiency by disabling unnecessary background backups.
- Portability across different RISC-V cores with minimal ISA extension.

The primary limitation is that the log buffer introduces a maximum bound on the number of writes within an atomic region. Exceeding this limit would either stall execution or require splitting the region into multiple atomic blocks. Despite this constraint, the mechanism provides a practical balance between flexibility and hardware simplicity.

4.5 Integration with RISC-V Pipeline

Integrating RISE into the RV32I pipeline requires minimal modifications. The baseline five-stage pipeline (IF, ID, EX, MEM, WB) is maintained intact, while RISE modules are connected externally via dedicated control and data signals.

The decode stage was extended to recognize the new `.ICA` instruction, which marks the start and end of atomic regions. When an atomic region is entered, the PCU is temporarily disabled, and writes are buffered until the region completes. If power fails, the snapshot taken prior to entry ensures that the region is re-executed atomically.

Figure 4.7 shows the high-level integration of the modules with the pipeline.

4.6 Verilog Project Structure

The project follows a hierarchical structure to ensure modularity and reusability:

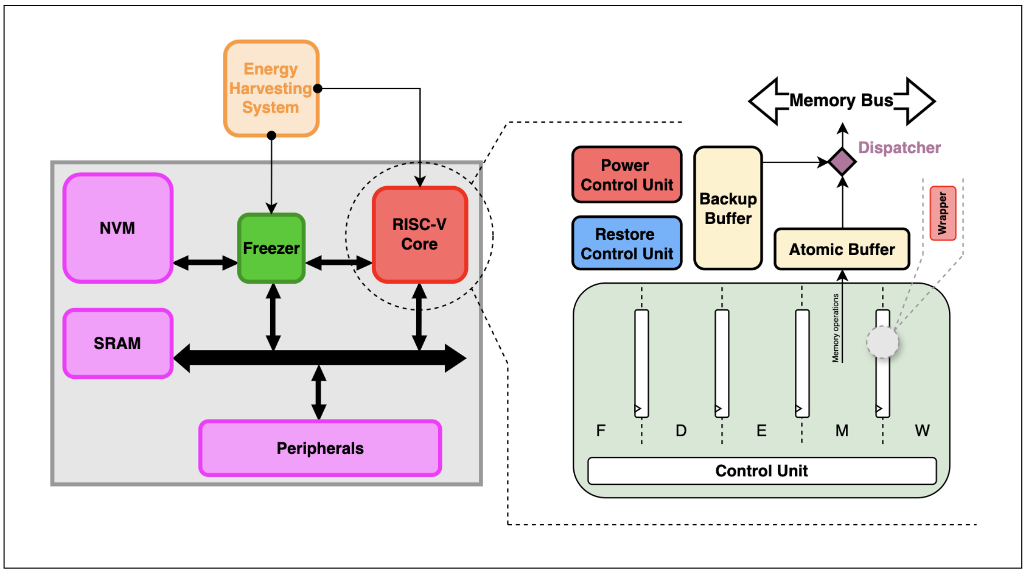


Figure 4.7: Integration of RISE modules into the baseline RV32I pipeline.

- **Core modules:** baseline RV32I pipeline (IF, ID, EX, MEM, WB).
- **RISE extensions:** ICRW, PCU, RCU, Dispatcher.
- **ISA extensions:** decode logic for `.ICA`.
- **Testbenches:** developed to verify both standalone modules and system-level integration.

Simulation and synthesis were performed using `Xilinx Vivado_tm 2024.1` under Windows 11 and Ubuntu 20.04.6 LTS. Testbenches provide waveform analysis and cycle-accurate validation of backup/restore logic.

4.7 Design Metrics and Overheads

The additional modules introduce modest area and timing overheads, while significantly enhancing resilience to power interruptions.

- **Area overhead:** ICRW adds two dirty bits per register and wrapper logic; PCU and RCU require small FSMs and buffers; Dispatcher requires arbitration logic. Overall area increase remains negligible relative to the full core.
- **Timing:** maximum operating frequency is slightly reduced due to additional signal routing but remains within design constraints for FPGA and ASIC targets.

- **Energy:** selective backup reduces memory transactions, minimizing energy overhead. Compared to checkpointing systems, RISE demonstrates lower average energy per instruction under intermittent workloads.

4.8 Discussion

The design achieves compatibility with conventional CMOS flows and avoids reliance on exotic non-volatile technologies. Compared to NVP-based approaches, which require MRAM or FeRAM integration, RISE can be implemented and tested using standard EDA toolchains.

Two system-level configurations are possible:

1. **Backup in volatile memory:** using an auxiliary controller (e.g., Freezer) to transfer snapshots to SRAM, with optional migration to NVM on power loss.
2. **Backup in non-volatile memory:** mapping directly to flash or MRAM regions, enabling persistence without auxiliary controllers.

Finally, the analysis of individual modules shows:

- **ICRW:** high flexibility and negligible overhead.
- **PCU:** efficient background backup, at the cost of modest complexity.
- **RCU:** reliable restore, limited by external memory latency.
- **Dispatcher:** transparent bus arbitration, but with variable latency for backup commits.

Overall, RISE provides a balanced trade-off between correctness, performance, and portability, representing a step forward in enabling intermittent computing on open RISC-V architectures.

4.9 Summary

In this chapter, the implementation details of the RISE framework were presented. We began by describing the baseline RV32I pipeline, which provides the foundation for RISE integration. Key assumptions regarding hazards and toolchain support were discussed, ensuring a clear understanding of the environment in which RISE was developed and tested.

The chapter then detailed the four main hardware modules introduced by RISE: the Intermittent Computing Register Wrapper (ICRW), the Power Control Unit (PCU), the Restore Control Unit (RCU), and the Dispatcher. Each module was designed to preserve modularity, operate transparently with respect to the original pipeline, and minimize area, timing, and energy overheads. The ICRW enables fine-grained state tracking through dirty-bit management, the PCU coordinates efficient background backups, the RCU ensures seamless recovery after power failures, and the Dispatcher resolves bus contention by scheduling backup transfers.

Beyond these modules, the chapter introduced the `.ICA` instruction as a novel ISA extension, providing explicit support for atomic execution regions. This feature empowers programmers to guarantee atomicity and consistency for critical code sections, bridging the gap between hardware reliability and software-level correctness in intermittent environments.

The integration of all these elements into the baseline RISC-V pipeline was shown to require only minimal modifications, preserving the simplicity and portability of the core. Finally, design metrics and qualitative analysis highlighted that RISE achieves a balanced trade-off: correctness and resilience to power failures are improved significantly, while the added complexity and hardware cost remain modest.

Overall, the implementation demonstrates that RISE can be realized within standard CMOS design flows and open-source RISC-V toolchains, without the need for exotic non-volatile devices. This positions RISE as a practical, portable, and scalable approach to intermittent computing.

The next chapter builds on this foundation by presenting a detailed evaluation of the implemented framework, focusing on backup/restore latency, area overhead, energy efficiency, and performance under realistic intermittent power scenarios.

Chapter 5

Evaluation

5.1 Introduction

This chapter evaluates the effectiveness of the RISE framework. The objective is to demonstrate that the proposed architecture reduces backup and restore latency, minimizes energy overhead, and guarantees correctness under intermittent power failures. We compare RISE against representative state-of-the-art approaches, with particular emphasis on Freezer [4], a specialized backup controller. Our evaluation is structured around three key questions:

1. How efficient are RISE's backup and restore mechanisms in terms of latency and energy?

2. How does RISE compare against existing hardware-assisted approaches such as Freezer?

3. What are the implications of these results for forward progress and reliability in intermittent systems?

5.2 Experimental Setup

The evaluation of RISE was conducted on a cycle-accurate Verilog model of a baseline 5-stage RV32I RISC-V core. The processor was extended with the four hardware modules introduced in Chapter 3: the Intermittent Computing Register Wrapper (ICRW), the Power Control Unit (PCU), the Restore Control Unit (RCU), and the Dispatcher. The processor includes 53 ICRW elements covering the entire register file and key sequential states of the pipeline.

5.2.1 Toolchain and Environment

The design was synthesized and simulated using **Xilinx Vivado**, while waveform inspection was performed with ModelSim. The simulation clock was set to **1 GHz**, corresponding to 1 ns per cycle. Power failures were emulated by forcing the reset of all sequential registers, thereby clearing the pipeline state. Upon power restoration, the RCU reloaded the preserved state from the backup buffer.

5.2.2 Benchmarks

Two benchmark sets were used:

- The **RISC-V compliance test suite**, to evaluate functional correctness under intermittent failures.
- Selected kernels from the **MiBench suite**, including *dijkstra*, *crc*, and *sha*, to evaluate performance and energy in realistic workloads.

5.2.3 Comparison Baselines

To provide context, RISE was compared against:

- A naive baseline without intermittent support (full state lost on power failure).
- **Freezer** [4], an external backup controller that saves dirty memory blocks to NVM.

5.3 Evaluation Metrics

The evaluation focused on the following metrics:

- **Backup latency:** number of cycles and absolute time to save state.
- **Restore latency:** cycles and time required to resume execution after failure.
- **Energy overhead:** energy required to perform backup and restore.
- **Forward progress:** fraction of useful instructions executed under repeated failures.

5.4 Performance Results

5.4.1 Backup Latency

Table 5.1 reports the backup latency of RISE. Depending on the number of dirty registers, backup requires between 35 and 215 cycles, corresponding to 0.035–0.215 μs at 1 GHz. The average case across benchmarks was 6.36 μs, as only a fraction of registers are typically dirty.

Table 5.1: Backup latency of RISE at 1 GHz

Case	**Cycles**	**Time [μs]**
Minimum	35	0.035
Maximum	215	0.215
Average (benchmarks)	–	6.36

5.4.2 Restore Latency

Table 5.2 summarizes restore performance. Regardless of benchmark, restore required 106 cycles (0.106 μs), which is significantly faster than checkpointing-based approaches.

Table 5.2: Restore latency of RISE at 1 GHz

Case	**Cycles**	**Time [μs]**
Restore latency	106	0.106

5.4.3 Waveform Analysis

Figure 5.1 shows a representative waveform from simulation, illustrating the backup process triggered by the PCU and coordinated by the Dispatcher. The transition of register states (CLEAN, DIRTY, BACKUP, MODIFIED) is visible, confirming the correct behavior of the ICRW FSM.

5.5 Comparison with State of the Art

5.5.1 Comparison with Freezer

Figure 5.2 compares the backup latency of RISE with Freezer. Freezer requires a fixed latency of 6.625 μs to back up 53 words, while RISE achieves an average

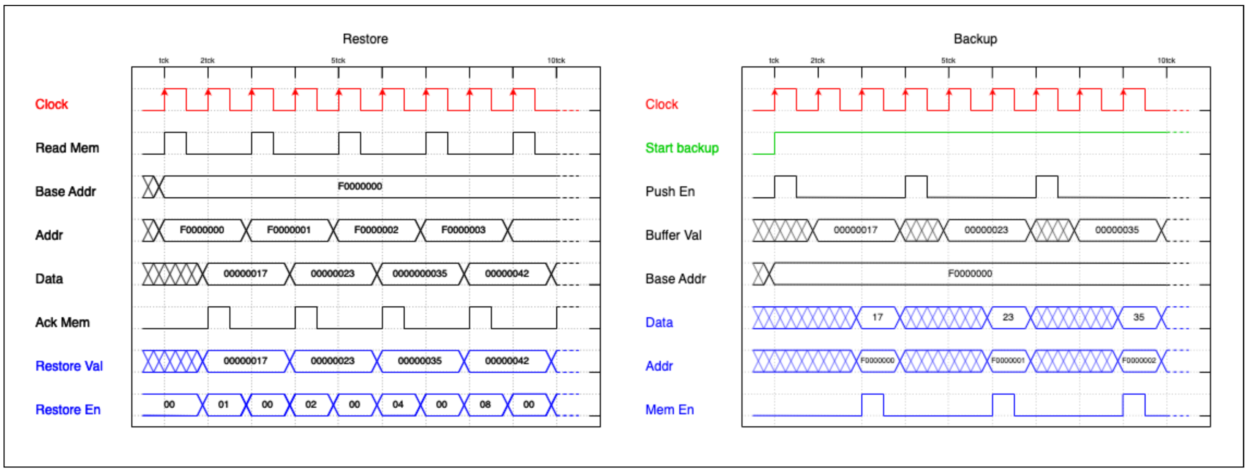


Figure 5.1: Simulation waveform showing backup and restore cycle of RISE.

latency of 6.36 μs thanks to selective dirty-bit tracking. Moreover, the worst-case latency of RISE (0.203 μs) is an order of magnitude lower.

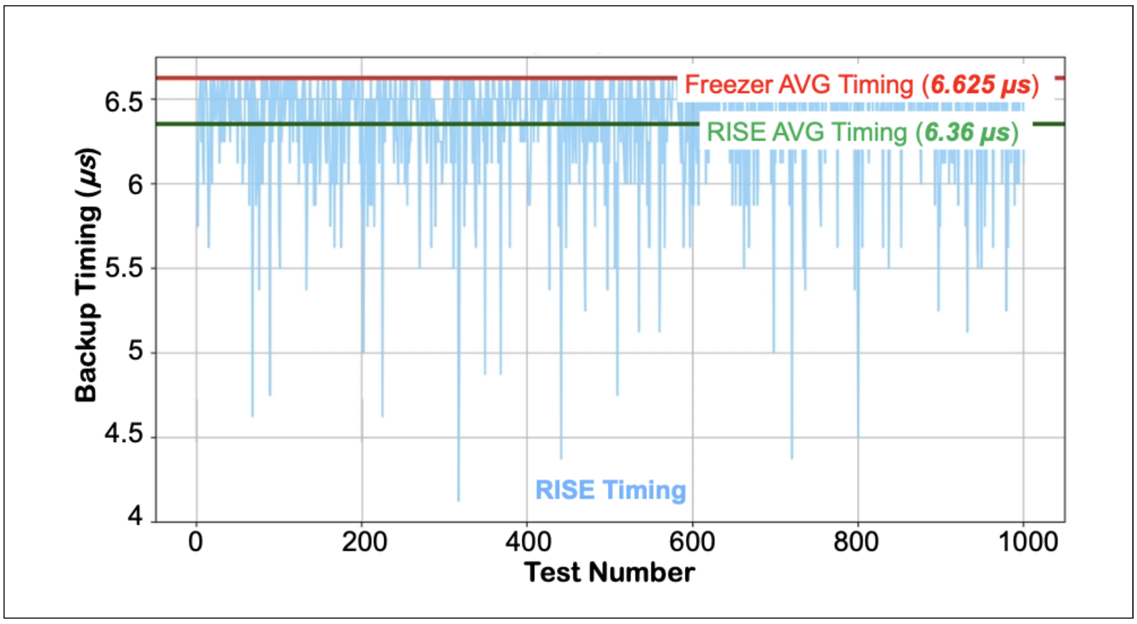


Figure 5.2: Comparison of backup timing: Freezer vs. RISE.

5.5.2 Energy Consumption

Energy measurements further highlight RISE's efficiency. Freezer consumes 77.083 nJ per backup, while RISE requires only 73.942 nJ on average, as shown in Figure 5.3. This 4% reduction, while modest, is significant given the extreme energy constraints of intermittent IoT devices.

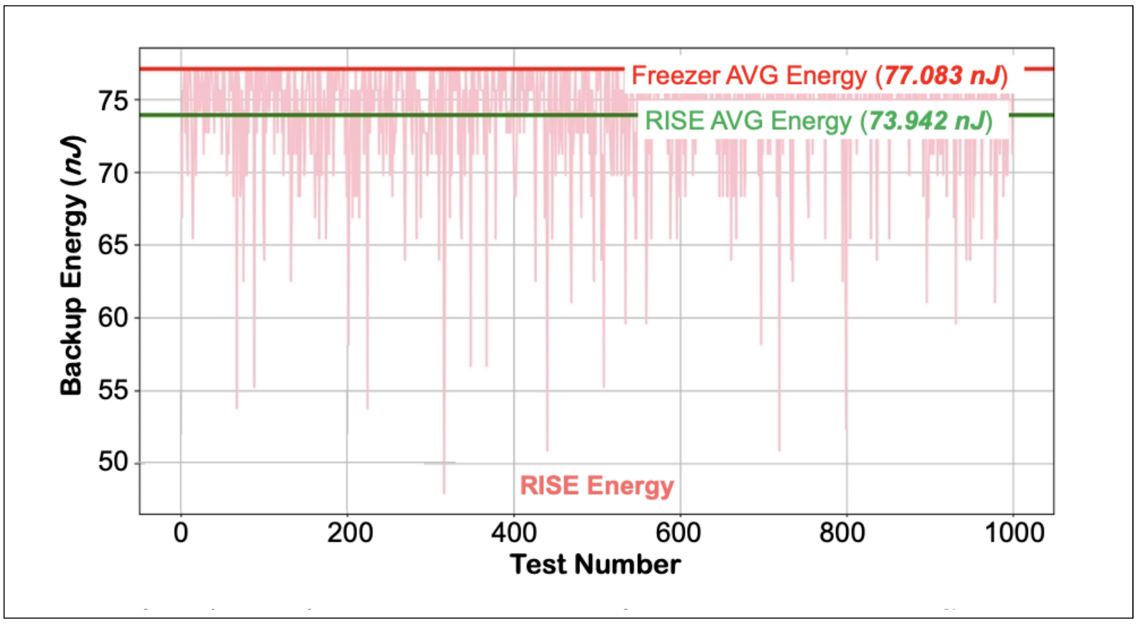


Figure 5.3: Comparison of backup energy: Freezer vs. RISE.

5.6 Qualitative Analysis

RISE combines the efficiency of hardware-assisted backup with the programmability of ISA-level extensions. Unlike checkpointing, it avoids redundant saves and long rollback times. Unlike task-based models, it does not impose complex compiler/runtime support. Compared to NVPs, RISE does not depend on exotic memory technologies, ensuring compatibility with standard CMOS. Finally, compared to Freezer, RISE provides explicit atomicity guarantees through the `.ICA` instruction.

5.7 Discussion

The evaluation results demonstrate that RISE achieves both low latency and low energy overhead while preserving correctness. In practical terms, this means that IoT devices equipped with RISE can perform more useful work per unit of harvested energy and remain reliable under frequent power interruptions. Nevertheless, limitations remain:

- The current evaluation is limited to single-core RV32I systems.
- Energy results are obtained from simulation, not physical measurements.
- More complex benchmarks and real workloads would provide deeper insights.

5.8 Summary

This chapter presented the evaluation of RISE. Our results show that RISE reduces backup latency to as low as 0.203 μs and restore latency to 0.106 μs, outperforming Freezer in both latency and energy efficiency. By combining ISA-level atomicity with efficient hardware support, RISE establishes itself as a robust and portable framework for intermittent computing. The next chapter discusses broader implications, limitations, and opportunities for future research.

Chapter 6

Discussion and Limitations

6.1 Introduction

The evaluation presented in Chapter 5 demonstrated that RISE achieves substantial improvements in backup and restore latency, energy efficiency, and programmability compared to existing solutions. However, as with any architectural innovation, the design of RISE involves trade-offs and is subject to certain limitations. This chapter provides a critical discussion of the results, highlighting both the strengths of the proposed framework and the areas where further research and optimization are required.

6.2 Strengths of the RISE Framework

6.2.1 Efficiency in Backup and Restore

RISE significantly reduces backup and restore latency by selectively saving only dirty registers through the ICRW module. The ability to minimize redundant memory operations allows the system to respond quickly to imminent power failures, thereby increasing forward progress. Compared to Freezer, RISE achieves similar or lower energy overhead while offering an order of magnitude lower worst-case latency.

6.2.2 ISA-Level Atomicity

The introduction of the `.ICA` instruction provides explicit support for atomic execution of critical code regions. This feature bridges the gap between software semantics and hardware enforcement, ensuring correctness under arbitrary power interruptions. Unlike checkpointing or task-based frameworks, which implicitly rely on compiler heuristics, RISE empowers developers to directly define fault-tolerant execution regions.

6.2.3 Compatibility and Portability

RISE avoids reliance on exotic non-volatile technologies, instead relying solely on standard CMOS-compatible modules. This design choice improves portability across different RISC-V cores and lowers adoption barriers. The modular structure of the ICRW, PCU, RCU, and Dispatcher makes the framework adaptable to various system-on-chip (SoC) designs, ranging from minimalist IoT nodes to more powerful embedded processors.

6.3 Limitations of the Current Design

6.3.1 Single-Core Evaluation

The current implementation and evaluation of RISE were performed on a single-core RV32I pipeline. While sufficient to demonstrate the feasibility of the approach, this setup does not capture the complexities of multicore systems, where backup and restore must be coordinated across cores and shared memory. Future extensions will need to address scalability to heterogeneous and multicore architectures.

6.3.2 Simulation-Based Validation

All experiments were performed using Verilog simulation. While cycle-accurate simulation provides a reliable measure of latency and functional correctness, it does not fully capture the physical energy consumption or area overhead of an implementation. A physical prototype on FPGA or ASIC would be required to validate the energy models and quantify silicon costs.

6.3.3 Memory and Buffering Assumptions

The evaluation assumed ideal memory access times for the backup buffer and main memory. In practice, non-volatile memories such as Flash or MRAM introduce longer latencies and higher energy costs. The integration of realistic NVM technologies may affect the backup/restore trade-offs and requires further investigation.

6.3.4 Limited Benchmark Coverage

Although MiBench and RISC-V compliance tests provide a good starting point, the benchmarks used remain relatively simple. Larger, more memory-intensive applications (e.g., database kernels, graph analytics) may stress the backup/restore mechanisms differently, potentially exposing bottlenecks not visible in lightweight workloads.

6.4 Trade-offs in Design

6.4.1 Area vs. Latency

The inclusion of the ICRW and Dispatcher introduces additional hardware complexity and area overhead. While this overhead is modest compared to the benefits in latency and energy, it may become more significant in ultra-constrained IoT devices where silicon area is highly limited.

6.4.2 Energy vs. Correctness

RISE prioritizes correctness guarantees (atomicity and consistency) over absolute energy minimization. In scenarios where reliability is less critical, lightweight task-based approaches may remain more energy-efficient. This highlights the importance of aligning the choice of intermittent computing strategy with the requirements of the target application.

6.4.3 Generality vs. Specialization

The design of RISE emphasizes portability and modularity, avoiding aggressive optimizations that would tie it to a specific use case. As a result, there may be opportunities to further optimize performance and energy efficiency in specialized deployments at the expense of generality.

6.5 Implications for Real-World Applications

The results obtained in Chapter 5 have important implications for real-world applications of intermittent computing:

- In **medical IoT devices**, where incorrect or inconsistent results may endanger human lives, RISE provides the necessary reliability under energy scarcity.
- In **industrial monitoring**, RISE reduces the likelihood of missed detections or false alarms due to corrupted state, ensuring safer operation.

- In **consumer IoT**, improved forward progress translates into smoother user experience, avoiding frequent restarts and glitches.

These examples underline that RISE not only advances academic research but also has tangible benefits in critical domains.

6.6 Summary

This chapter critically discussed the design choices and evaluation results of RISE. The framework offers notable strengths in terms of efficiency, atomicity, and portability, but it also faces limitations such as reliance on simulation, lack of multicore validation, and simplified memory assumptions. These trade-offs highlight opportunities for further research, which are addressed in the concluding chapter.

Chapter 7

Conclusions and Future Work

7.1 Introduction

This chapter concludes the thesis by summarizing the main contributions of the RISE framework and reflecting on its significance for the field of intermittent computing. We also outline directions for future research that can build upon the foundation laid by this work.

7.2 Summary of Contributions

The primary objective of this thesis was to design, implement, and evaluate an architectural extension to the RISC-V ISA and microarchitecture that enables efficient, reliable execution under intermittent power supply. The proposed solution, named **RISE (RISC-V Intermittent System Extensions)**, advances the state of the art through the following contributions:

1. **Architectural Design:** Introduction of a modular hardware framework composed of the Intermittent Computing Register Wrapper (ICRW), Power Control Unit (PCU), Restore Control Unit (RCU), and Dispatcher.

2. **ISA Extensions:** Definition of the `.ICA` instruction to mark atomic code regions, providing correctness guarantees across power failures.

3. **Efficient Backup and Restore:** Implementation of dirty-bit tracking and selective register backup, reducing latency to as low as 0.203 μs and restore time to 0.106 μs at 1 GHz.

4. **Portability:** Demonstration that RISE is implementable with standard CMOS processes and integrates seamlessly with existing RISC-V cores.

5. **Evaluation:** Validation through simulation and benchmark execution, showing that RISE outperforms state-of-the-art solutions such as Freezer in both latency and energy efficiency.

7.3 Key Findings

The evaluation presented in Chapter 5 provides strong evidence that RISE successfully addresses the core challenges of intermittent computing:

- **Reduced overhead:** By backing up only dirty registers, RISE significantly lowers backup latency and energy compared to checkpointing approaches.
- **Correctness guarantees:** The `.ICA` instruction enforces atomic execution regions, eliminating the risk of inconsistent states.
- **Practicality:** RISE avoids the reliance on exotic memory technologies, ensuring compatibility with existing fabrication processes and broad adoption potential.

These findings demonstrate that architectural and ISA-level support is a powerful approach to enabling reliable batteryless computing.

7.4 Future Work

While RISE represents an important step forward, several avenues remain open for exploration:

7.4.1 Extension to Multicore and Out-of-Order Processors

The current evaluation focused on a single-core RV32I pipeline. Scaling RISE to multicore systems introduces new challenges, such as coordinating backup across cores and maintaining memory consistency. Furthermore, applying RISE to out-of-order (OoO) processors would require handling speculative state, making this a promising but complex direction.

7.4.2 Integration with Compiler and Runtime Support

Although the `.ICA` instruction provides atomicity guarantees, compiler and runtime frameworks could further optimize the placement of atomic regions. Automatic insertion of ICA boundaries based on program analysis could relieve programmers from manual annotation and improve usability.

7.4.3 Prototype on FPGA and ASIC

Physical prototyping on FPGA or ASIC would allow validation of energy and area overhead in real silicon. Such a prototype would also enable end-to-end demonstrations of RISE in realistic IoT scenarios, such as sensor nodes powered by energy harvesting.

7.4.4 Evaluation with Real Workloads

Future evaluations should extend beyond microbenchmarks to include complex applications such as machine learning inference, secure communication protocols, and real-time control tasks. These workloads would stress the intermittent execution model in different ways, providing deeper insights into the generality of RISE.

7.4.5 Integration with Energy Harvesting Systems

Finally, coupling RISE with real energy harvesting front-ends would enable a holistic evaluation of the entire system. Exploring the co-design of architecture and energy management policies may yield additional efficiency gains.

7.5 Closing Remarks

This thesis introduced RISE, a novel extension to the RISC-V architecture that bridges the gap between software semantics and hardware support for intermittent computing. Through a careful combination of ISA primitives and lightweight hardware modules, RISE enables efficient and reliable execution in batteryless systems, paving the way for sustainable IoT deployments.

The work presented here highlights the potential of architectural solutions in addressing the unique challenges of intermittent computing. While limitations remain, RISE lays the foundation for future exploration of portable, efficient, and correct-by-design architectures for the next generation of energy-harvesting devices.

Appendix A

Verilog Source Code

A.1 Intermittent Computing Register Wrapper (ICRW)

```
module RegN_IC_Wrapper #(parameter N = 32) (
        Ld,
        Vin,
        Vout,
        Dirty_val,
        Backup_en,
        Backup_ack,
        Backup_Vout,
        Rst_DrtyCtrl,
        Restore_en,
        Restore_Vin,
        Rst,
        Clk,
        Pwr_off
    );

    // register
    input Ld, Rst, Clk;
    input [N-1:0] Vin;
    output [N-1:0] Vout;

    // Dirty bit FSM
    input Backup_en, Backup_ack;
    output [1:0] Dirty_val;
    output [N-1:0] Backup_Vout;
    input Rst_DrtyCtrl;

    // restore signals
    input [N-1:0] Restore_Vin;
    input Restore_en;

    // intermittent computing simulation
    input Pwr_off;

    wire [N-1:0] Vin_wire, Vin_wire_reg;
    wire [N-1:0] Vout_wire;

    wire Ld_reg_wire;
    wire Rst_reg_wire;

    wire cmp_res;

    wire Rst_DrtyCtrl_wire;

    wire Ld_wire;


    RegN #(
            .N            (N)
    ) register_n (
            .Vin          (Vin_wire_reg),
            .Vout         (Vout_wire),
            .Ld           (Ld_wire),
            .Rst          (Rst),
            .Clk          (Clk),
            .Pwr_off      (Pwr_off)
    );

    DirtyCtrl dirty_controller (
            .Ld_reg       (Ld_reg_wire),
            .Rst_reg      (Rst_reg_wire),
            .Backup_en    (Backup_en),
            .Backup_ack   (Backup_ack),
            .Clk          (Clk),
            .Rst          (Rst_DrtyCtrl_wire),
            .Dirty_val    (Dirty_val),
            .Pwr_off      (Pwr_off)
    );

    CmpN #(
            .N            (N)
    ) comparator (
            .Vin_a        (Vin_wire),
            .Vin_b        (Vout_wire),
            .Vout         (cmp_res)
    );

    MuxN_21 #(
            .N          (N)
    ) multiplexer_restore (
            .Vin_a      (Vin_wire),  // sel = 0
            .Vin_b      (Restore_Vin),  // sel = 1
            .sel        (Restore_en),
            .Vout        (Vin_wire_reg)
    );



    assign Backup_Vout = Vout_wire;
    assign Vout = Vout_wire;
    assign Vin_wire = Vin;

    assign Ld_reg_wire  = Ld  & ~cmp_res;
    assign Rst_reg_wire = Rst & ~cmp_res;

    assign Rst_DrtyCtrl_wire = Rst_DrtyCtrl | Restore_en;
    assign Ld_wire = Ld | Restore_en;



endmodule
```

Listing A.1: Intermittent Computing Register Wrapper Verilog implementation

```
module RegN #(parameter N = 32)(
        Vin,
        Vout,
        Ld,
        Rst,
        Clk,
        Pwr_off
    );

    input [N-1:0] Vin;
    output [N-1:0] Vout;
    input Ld, Rst, Clk;
    input Pwr_off;

    reg [N-1:0] Vout;

    always @(posedge Clk or posedge Rst or posedge Pwr_off) begin
        if (Rst || Pwr_off)
            Vout <= {N{1'b0}};
        else if (Ld)
            Vout <= Vin;
    end
```

```verilog

endmodule
```

Listing A.2: Register Verilog implementation

```verilog
module DirtyCtrl(
        Ld_reg,
        Rst_reg,
        Backup_en,
        Backup_ack,
        Clk,
        Rst,
        Dirty_val,
        Pwr_off
    );

    input Ld_reg, Rst_reg, Backup_en, Backup_ack;
    output [1:0] Dirty_val;
    input Clk, Rst;
    input Pwr_off;

    reg [1:0] Dirty_val;

    reg [1:0] State, NextState;

    parameter   CLEAN = 0,
                DIRTY = 1,
                READ = 2,
                DIRTY_WR = 3;

    // comb logic
    always @(State, Ld_reg, Rst_reg, Backup_en, Backup_ack) begin
        case (State)
            CLEAN: begin
                Dirty_val <= 2'b00;
                if (Ld_reg || Rst_reg)
                    NextState <= DIRTY;
                else
                    NextState <= State;
            end

            DIRTY: begin
                Dirty_val <= 2'b01;
                if (Backup_en)
                    NextState <= READ;
                else
                    NextState <= State;
            end

            READ: begin
                Dirty_val <= 2'b10;
                if (Ld_reg || Rst_reg)
                    NextState <= DIRTY_WR;
                else if (Backup_ack)
                    NextState <= CLEAN;
                else
                    NextState <= State;
            end

            DIRTY_WR: begin
                Dirty_val <= 2'b11;
                if (Backup_ack)
                    NextState <= CLEAN;
                else
                    NextState <= State;
            end
        endcase
    end


    // state reg
    always @(posedge Clk or posedge Pwr_off) begin
        if (Rst)
            State <= CLEAN;
        else if (Pwr_off)
            State <= CLEAN;
        else
            State <= NextState;
    end
endmodule
```

Listing A.3: Dirty Bit Controller Verilog implementation

```
module CmpN #(parameter N = 32) (
        Vin_a,
        Vin_b,
        Vout
    );

    input [N-1:0] Vin_a, Vin_b;
    output Vout;

    reg Vout;

    always @(Vin_a, Vin_b) begin
        if (Vin_a == Vin_b)
            Vout <= 1'b1;
        else
```

```
            Vout <= 1'b0;
    end
endmodule
```

Listing A.4: Comparator Verilog implementation

```
module MuxN_21 #(parameter N = 32) (
        Vin_a,  // sel = 0
        Vin_b,  // sel = 1
        sel,
        Vout
    );

    input [N-1:0] Vin_a, Vin_b;
    input sel;
    output [N-1:0] Vout;
    reg [N-1:0] Vout;

    always @(Vin_a, Vin_b, sel) begin
        if (sel)
            Vout <= Vin_b;
        else
            Vout <= Vin_a;
    end

endmodule
```

Listing A.5: Multiplexer Verilog implementation

A.2 Power Control Unit (PCU)

```
module PCU #(
        parameter K = 10,  // number of IC_Reg_Wrapeer
        parameter N = 32,  // width of IC_Reg_Wrapper
        parameter M = 32  // width timer value register
    ) (
        Backup_Vout_IC_Reg_Wrapper,
        Start_FSM_PCU,
        PushVal_Buffer,
        Load_Timer,
        PushEn_Buffer,
        backup_now_ctrl, // start backup now
        Dirty_vals_IC_Reg_Wrapper, // K = number of Wrappers
        Rst_Buffer,
        Backup_Ens_IC_REG_Wrapper, // K = number of Wrappers
        IsFull_Buffer,
        Clk,
        Rst,
        Pwr_off
    );

    localparam LOG2_K = $clog2(K);


    input [(K*N)-1:0] Backup_Vout_IC_Reg_Wrapper;
    input Start_FSM_PCU;
    input [M-1:0] Load_Timer;
    input [(K*2)-1:0] Dirty_vals_IC_Reg_Wrapper;
    input IsFull_Buffer;
    input backup_now_ctrl;

    output [N+LOG2_K-1:0] PushVal_Buffer;
    output PushEn_Buffer;
    output Rst_Buffer;
    output [K-1:0] Backup_Ens_IC_REG_Wrapper;

    input Pwr_off;
    input Rst;
    input Clk;

    wire last_wire;
    wire end_wire;
    wire end_timer_wire;
    wire [1:0] dirty_val_wire;
    wire Rst_CntN_wire;
    wire Rst_Timer_wire;
    wire En_Timer_wire;
    wire Clk_CntN_wire;

    wire [LOG2_K-1:0] addr_wrapper;
    wire [LOG2_K-1:0] addr_wrapper_sub;
    wire [N-1:0] push_val_buffer_wire;

    wire [K-1:0] Backup_Ens_IC_REG_Wrapper_wire;

    assign end_wire = end_timer_wire | backup_now_ctrl;

    // FSM
    FSM_PCU fsm_power_cu (
        .Start              (Start_FSM_PCU),
        .IsFull_Buffer      (IsFull_Buffer),
        .Last               (last_wire),
        .End_Timer          (end_wire),
        .DirtyValSel        (dirty_val_wire),
        .Rst_Buffer         (Rst_Buffer),
        .Rst_CntN           (Rst_CntN_wire),
        .Rst_Timer              (Rst_Timer_wire),
        .En_Timer               (En_Timer_wire),
        .Clk_CntN               (Clk_CntN_wire),
        .PushEn_Buffer          (PushEn_Buffer),
        .Rst                    (Rst),
        .Clk                    (Clk),
        .Pwr_off                (Pwr_off)
    );

    // Timer
    Timer #(
        .N                      (M)
    ) timer_pcu (
        .En                     (En_Timer_wire),
        .Load                   (Load_Timer),
        .Clk                    (Clk),
        .Rst                    (Rst_Timer_wire),
        .End                    (end_timer_wire),
        .Pwr_off                (Pwr_off)
    );

    // CntN
    CntN #(
        .N                      (LOG2_K)
    ) counter_backup_addr (
        .Clk                    (Clk_CntN_wire),
        .Rst                    (Rst_CntN_wire),
        .Pwr_off                (Pwr_off),
        .Vout                   (addr_wrapper)
    );

    // Decoder
    DecN #(
        .N                      (LOG2_K)
    ) dec_backup_en (
        .Vin                    (addr_wrapper_sub),
        .Vout                   (Backup_Ens_IC_REG_Wrapper_wire)
    );

    assign Backup_Ens_IC_REG_Wrapper =
   Backup_Ens_IC_REG_Wrapper_wire & {N{PushEn_Buffer}};

    // Multiplexer
    MuxM_N1 #(
        .N                      (K),
        .M                      (2)
    ) mux_dirty_val (
        .Vin                    (Dirty_vals_IC_Reg_Wrapper),
        .Sel                    (addr_wrapper_sub),
        .Vout                   (dirty_val_wire)
    );

    // sub by 1
    Sub1 #(
        .N                      (LOG2_K)
    ) sub1_addr (
        .Vin                    (addr_wrapper),
        .Vout                   (addr_wrapper_sub)
    );

    // and signal Last
    CmpN_M #(
        .N          (LOG2_K),
        .M          (K)
    ) cmp_last_addr_wrapper (
        .Vin_a      (addr_wrapper),
        .Vout       (last_wire)
    );

    // mux Backup vals
    MuxM_N1 #(
        .N                      (K),
        .M                      (N)
    ) mux_backup_val (
        .Vin                    (Backup_Vout_IC_Reg_Wrapper),
        .Sel                    (addr_wrapper_sub),
        .Vout                   (push_val_buffer_wire)
    );

    assign PushVal_Buffer = {push_val_buffer_wire,
   addr_wrapper_sub};


endmodule
```

Listing A.6: Power Control Unit Verilog implementation

```
module FSM_PCU (
        Start,
        IsFull_Buffer,
        Last,
        End_Timer,
        DirtyValSel,
        Rst_Buffer,
        Rst_CntN,
        Rst_Timer,
        En_Timer,
        Clk_CntN,
```

```verilog
        PushEn_Buffer,
        Rst,
        Clk,
        Pwr_off
    );

    input Start;
    input IsFull_Buffer;
    input Last;
    input End_Timer;
    input [1:0] DirtyValSel;
    output reg Rst_Buffer;
    output reg Rst_CntN;
    output reg Rst_Timer;
    output reg En_Timer;
    output reg Clk_CntN;
    output reg PushEn_Buffer;

    input Rst;
    input Clk;
    input Pwr_off;

    parameter   IDLE = 0,
                RESET = 1,
                WAIT = 2,
                POOLING = 3,
                BACKUP_REG = 4;

    parameter DIRTY = 2'b01; // value of dirty status of the IC_REGN_WRAPPER

    reg [2:0] State, NextState;

    // CombLogic
    always @(State, Start, IsFull_Buffer, Last, End_Timer, DirtyValSel) begin
        Rst_Buffer <= 0;
        Rst_CntN <= 0;
        En_Timer <= 0;
        Rst_Timer <= 0;
        Clk_CntN <= 0;
        PushEn_Buffer <= 0;
        NextState <= State;

        case (State)
            IDLE: begin
                Rst_Buffer <= 1;
                if (Start)
                    NextState <= RESET;
            end

            RESET: begin
                Rst_Timer <= 1;
                Rst_CntN <= 1;
                Clk_CntN <= 1;
                if (IsFull_Buffer == 0)
                    NextState <= WAIT;
            end

            WAIT: begin
                En_Timer <= 1;
                if (End_Timer)
                    NextState <= POOLING;

            end

            POOLING: begin
                Clk_CntN <= 1;
                if (Last && DirtyValSel != DIRTY)
                    NextState <= RESET;
                else if (Last == 0 && DirtyValSel != DIRTY)
                    NextState <= State;
                else if (DirtyValSel == DIRTY)
                    NextState <= BACKUP_REG;

            end

            BACKUP_REG: begin
                PushEn_Buffer <= 1;
                if (IsFull_Buffer || Last)
                    NextState <= RESET;
                else
                    NextState <= POOLING;
            end
        endcase
    end


    // StateReg
    always @(posedge Clk or posedge Pwr_off) begin
        if (Rst)
            State <= IDLE;
        else if (Pwr_off)
            State <= IDLE;
        else
            State <= NextState;
    end

```

```

endmodule
```

Listing A.7: FSM PCU Verilog implementation

```
module Timer #(parameter N = 32) (
        En,
        Load,
        Clk,
        Rst,
        End,
        Pwr_off
    );

    input En, Clk, Rst;
    input [N-1:0] Load;
    output End;
    input Pwr_off;

    wire [N-1:0] wire_a;
    wire [N-1:0] wire_b;
    wire end_wire;
    wire Rst_cnt;

    wire cnt_en;

    RegN #(
        .N            (N)
    ) register_ticks (
        .Vin          (Load),
        .Vout         (wire_b),
        .Ld           (Rst),
        .Rst          (1'b0),
        .Clk          (Clk),
        .Pwr_off      (Pwr_off)
    );

    CntN #(
        .N            (N)
    ) counter_ticks (
        .Clk          (cnt_en),
        .Rst          (Rst),
        .Pwr_off      (Pwr_off),
        .Vout         (wire_a)
    );

    CmpN #(
        .N            (N)
    ) end_check (
```

```verilog
        .Vin_a      (wire_a),
        .Vin_b      (wire_b),
        .Vout       (end_wire)
    );

    assign End = end_wire & En;
    assign Rst_cnt = end_wire | Rst;
    assign cnt_en = Clk & (En | Rst);

endmodule
```

Listing A.8: Timer Verilog implementation

```verilog
module CntN #(parameter N = 32) (
        Clk,
        Rst,
        Pwr_off,
        Vout
    );

    input Clk;
    input Rst;
    input Pwr_off;

    output reg [N-1:0] Vout;

    always @(posedge Clk or posedge Pwr_off) begin
        if (Rst | Pwr_off)
            Vout <= 0;
        else
            Vout <= Vout + 1'b1;
    end

endmodule
```

Listing A.9: Counter Verilog implementation

```verilog
module DecN #(
        parameter N = 4
    )(
        Vin,
        Vout
    );

    input [N-1:0] Vin;
    output reg [2**N-1:0] Vout;

    always @(Vin) begin
        Vout <= {N{1'b0}};
        Vout[Vin] <= 1'b1;
    end
endmodule
```

Listing A.10: Decrement Module Verilog implementation

```
module MuxM_N1 #(
        parameter N = 4,
        parameter M = 16
    )(
        Vin,
        Sel,
        Vout
    );

    input [(M*N)-1:0] Vin;
    input [$clog2(N)-1:0] Sel;
    output reg [M-1:0] Vout;

    always @(Vin, Sel) begin
        Vout <= Vin[Sel*M +:M];
    end

endmodule
```

Listing A.11: Multiplexer Verilog implementation

```
module Sub1 #(
        parameter N = 4
    ) (
        Vin,
        Vout
    );

    input [N-1:0] Vin;
    output [N-1:0] Vout;

    assign Vout = (Vin == 0) ? 0 : (Vin - 1);

endmodule
```

Listing A.12: Subtract Module Verilog implementation

```
module CmpN_M #(
        parameter N = 32,
        parameter M = 0
    ) (
        input  [N-1:0] Vin_a,
        output reg Vout
    );

    always @(*) begin
        if (Vin_a == M)
            Vout = 1'b1;
        else
            Vout = 1'b0;
    end

endmodule
```

Listing A.13: Comparator Verilog implementation

A.3 Restore Control Unit (RCU)

```
module RCU #(
        parameter N = 10,  // Number of IC wrapper
        parameter K = 32,  // size base address
        parameter M = 32   // width IC wrapper
    ) (
        AckMem,
        Start,
        ReadMem,
        BaseAddr,
        AddrMem,
        ValMem,
        RestoreVal,
        RestoreEn,
        Clk,
        Rst,
        Pwr_off
    );

    input AckMem;
    input Start;
    input [K-1:0] BaseAddr;
    input [M-1:0] ValMem;
    input Clk;
    input Rst;
    input Pwr_off;

    output [K-1:0] AddrMem;
    output ReadMem;
    output [M-1:0] RestoreVal;
    output [N-1:0] RestoreEn;

    localparam LOG2_N = $clog2(N);


    wire rst_cnt_wire;
    wire en_cnt_wire;
    wire [LOG2_N-1:0] vout_cnt_wire;
    wire [N-1:0] dec_out_wire;
    wire [LOG2_N-1:0] vout_sub_wire;
    wire end_wire;
    wire clk_cnt;
    wire restore_vin_en_wire;
    wire restore_dec_en_wire;
    wire [K-1:0] input_a_adder;

    CmpN_M #(
        .N                  (LOG2_N),
        .M                  (N)
    ) cmp_addr (
        .Vin_a              (vout_sub_wire),
        .Vout               (end_wire)
    );


    assign input_a_adder = {{(K-LOG2_N){1'b0}}, vout_cnt_wire};

    Adder #(
        .N                  (K)
    ) adder_base_addr (
        .A                  (input_a_adder),
        .B                  (BaseAddr),  // K bits
        .Cin                (1'b0),
        .Cout               (),
        .S                  (AddrMem)
    );

    Sub1 #(
        .N                  (LOG2_N)
    ) sub_cnt (
        .Vin                (vout_cnt_wire),
        .Vout               (vout_sub_wire)
    );

    assign clk_cnt = en_cnt_wire | rst_cnt_wire;

    CntN #(
        .N                  (LOG2_N)
    ) local_addr_cnt (
        .Clk                (clk_cnt),
        .Rst                (rst_cnt_wire),
        .Pwr_off            (Pwr_off),
        .Vout                   (vout_cnt_wire)
    );

    DecN #(
        .N                      (LOG2_N)
    ) dec_restore_en (
        .Vin                    (vout_sub_wire),
        .Vout                   (dec_out_wire)
    );

    FSM_RCU fsm_rcu (
        .Start                  (Start),
        .AckMem                 (AckMem),
        .ReadEn                 (ReadMem),
        .RstCnt                 (rst_cnt_wire),
        .EnCnt                  (en_cnt_wire),
        .EnDec                  (restore_dec_en_wire),
        .End                    (end_wire),
        .Restore_VinEn          (restore_vin_en_wire),
        .Pwr_off                (Pwr_off),
        .Rst                    (Rst),
        .Clk                    (Clk)
    );

    // enable restore_vin
    assign RestoreVal = ValMem & {M{restore_vin_en_wire}};

    // enable restore decoder
    assign RestoreEn = dec_out_wire & {N{restore_dec_en_wire}};

endmodule
```

Listing A.14: Restore Control Unit Verilog implementation

```
module Adder #(
        parameter N = 32
    ) (
        A,
        B,
        Cin,
        Cout,
        S
    );

    input [N-1:0] A;
    input [N-1:0] B;
    input Cin;

    output reg Cout;
    output reg [N-1:0] S;

    always @(A, B, Cin) begin
        {Cout, S} = A + B + Cin;
    end
endmodule
```

Listing A.15: Adder Verilog implementation

```
module FSM_RCU (
        Start,
        AckMem,
        ReadEn,
        RstCnt,
        EnCnt,
        EnDec,
        End,
        Restore_VinEn,
        Pwr_off,
        Rst,
        Clk
    );

    input Start;
    input AckMem;
    input End;

    input Pwr_off;
    input Rst;
    input Clk;


    output reg ReadEn;
    output reg RstCnt;
    output reg EnCnt;
    output reg EnDec;
    output reg Restore_VinEn;



    parameter    IDLE = 0,
                 RESET = 1,
                 READ = 2,
                 RESTORE = 3;

    reg [1:0] State, NextState;


    // Comb log
```

```verilog
    always @(Start, AckMem, End, State) begin
        ReadEn <= 1'b0;
        RstCnt <= 1'b0;
        EnCnt <= 1'b0;
        EnDec <= 1'b0;
        Restore_VinEn <= 1'b0;

        case (State)
            IDLE: begin
                if (Start)
                    NextState <= RESET;
            end

            RESET: begin
                RstCnt <= 1'b1;
                NextState <= READ;
            end

            READ: begin
                ReadEn <= 1'b1;
                if (AckMem)
                    NextState <= RESTORE;
            end

            RESTORE: begin
                EnCnt <= 1'b1;
                EnDec <= 1'b1;
                Restore_VinEn <= 1'b1;
                if (End)
                    NextState <= IDLE;
                else
                    NextState <= READ;
            end
        endcase

    end

    // State reg
    always @(posedge Clk or posedge Pwr_off) begin
        if (Rst || Pwr_off)
            State <= IDLE;
        else
            State <= NextState;
    end




endmodule
```

Listing A.16: FSM RCU Verilog implementation

A.4 Dispatcher (Backup Bus Arbitration)

```
module Dispatcher #(
        parameter K = 10,    // number of IC_REG_WRAPPERS
        parameter N = 32,    // width IC_REG_WRAPPER
        parameter M = 32     // width base address
    ) (
        Start,
        IsEmpty,
        WriteOp,
        DirtyBits,
        BackupVals,
        ValBuffer,
        AddrBuffer,
        BaseAddr,
        Rst,
        Clk,
        Pwr_off,
        Val,
        Addr,
        WriteEn,
        AckBackups,
        PullEn
    );

    localparam LOG2_K = $clog2(K);

    input Start;
    input IsEmpty;
    input WriteOp;
    input [(K*2)-1:0] DirtyBits;
    input [(K*N)-1:0] BackupVals;
    input [N-1:0] ValBuffer;
    input [LOG2_K-1:0] AddrBuffer;
    input [M-1:0] BaseAddr;
    input Rst;
    input Clk;
    input Pwr_off;

    // outputs
    output [N-1:0] Val;
    output [M-1:0] Addr;
    output WriteEn;
    output [K-1:0] AckBackups;
    output PullEn;


    wire [1:0] DirtyVal_wire;
    wire RstVal_wire;
    wire RstAddr_wire;
    wire LdVal_wire;
    wire LdAddr_wire;
    wire SelVal_wire;
    wire EnAck_wire;
    wire EnBuff_wire;
    wire [N-1:0] backupVal_wire;
    wire [(N*2)-1:0] choose_val_vin;
    wire [N-1:0] val_sel_wire;
    wire [N-1:0] vout_val_wire;
    wire [LOG2_K-1:0] vout_addr_wire;
    wire [M-1:0] vout_actual_addr_wire;
    wire [K-1:0] ack_sigs_wire;

    // FSM
    FSM_Dispatcher FSM_dispatcher (
        .Start                  (Start),
        .IsEmpty                (IsEmpty),
        .WriteOp                (WriteOp),
        .DirtyVal               (DirtyVal_wire),
        .PullEn                 (PullEn),
        .RstVal                 (RstVal_wire),
        .RstAddr                (RstAddr_wire),
        .LdVal                  (LdVal_wire),
        .LdAddr                 (LdAddr_wire),
        .SelVal                 (SelVal_wire),
        .EnAck                  (EnAck_wire),
        .EnBuff                 (EnBuff_wire),
        .Pwr_off                (Pwr_off),
        .Rst                    (Rst),
        .Clk                    (Clk)
    );

    // mux dirty bits
    MuxM_N1 #(
        .N                      (K),
        .M                      (2)
    ) mux_dirty_bits (
        .Vin                    (DirtyBits),
        .Sel                    (AddrBuffer),
        .Vout                   (DirtyVal_wire)
    );

    // mux backup vals
    MuxM_N1 #(
        .N                      (K),
        .M                      (N)
    ) mux_backup_vals (
        .Vin                    (BackupVals),
        .Sel                    (vout_addr_wire),
        .Vout                   (backupVal_wire)
    );


    assign choose_val_vin = {backupVal_wire, ValBuffer};
    // mux sel val buffer
    MuxM_N1 #(
        .N                      (2),
        .M                      (N)
    ) mux_val_buffer (
        .Vin                    (choose_val_vin),
        .Sel                    (SelVal_wire),
        .Vout                   (val_sel_wire)
    );

    // reg val
    RegN #(
        .N                      (N)
    ) reg_val (
        .Vin                    (val_sel_wire),
        .Vout                   (vout_val_wire),
        .Ld                     (LdVal_wire),
        .Rst                    (RstVal_wire),
        .Clk                    (Clk),
        .Pwr_off                (Pwr_off)
    );

    // reg addr
    RegN #(
        .N                      (LOG2_K)
    ) reg_addr_buf (
        .Vin                    (AddrBuffer),
        .Vout                   (vout_addr_wire),
        .Ld                     (LdAddr_wire),
        .Rst                    (RstAddr_wire),
        .Clk                    (Clk),
        .Pwr_off                (Pwr_off)
    );

    // adder base addr + buff addr
    Adder #(
        .N                      (M)
```

```verilog
    ) adder_addr (
        .A                  ({{(M-LOG2_K){1'b0}}, vout_addr_wire})
    ,
        .B                  (BaseAddr),
        .Cin                (1'b0),
        .Cout               (),
        .S                  (vout_actual_addr_wire)
    );

    // buffer 3 states output
    TriBuff #(
        .N                  (N)
    ) buff_3s_mem_interface_val (
        .Vin                (vout_val_wire),
        .En                 (EnBuff_wire),
        .Vout               (Val)
    );

    TriBuff #(
        .N                  (M)
    ) buff_3s_mem_interface_addr (
        .Vin                (vout_actual_addr_wire),
        .En                 (EnBuff_wire),
        .Vout               (Addr)
    );

    TriBuff #(
        .N                  (1)
    ) buff_3s_mem_interface_write_en (
        .Vin                (EnBuff_wire),
        .En                 (EnBuff_wire),
        .Vout               (WriteEn)
    );

    // decoder ack backup
    DecN #(
        .N                  (LOG2_K)
    ) decoder_ack_backup (
        .Vin                (vout_addr_wire),
        .Vout               (ack_sigs_wire)
    );

    // en ack backup
    assign AckBackups = ({K{EnAck_wire}} & ack_sigs_wire);



endmodule
```

Listing A.17: Dispatcher Verilog implementation

```
module FSM_Dispatcher (
        Start,
        IsEmpty,
        WriteOp,
        DirtyVal,
        PullEn,
        RstVal,
        RstAddr,
        LdVal,
        LdAddr,
        SelVal,
        EnAck,
        EnBuff,
        Pwr_off,
        Rst,
        Clk
    );

    input Start;
    input IsEmpty;
    input WriteOp;
    input [1:0] DirtyVal;

    input Pwr_off;
    input Rst;
    input Clk;

    output reg PullEn;
    output reg RstVal;
    output reg RstAddr;
    output reg LdVal;
    output reg LdAddr;
    output reg SelVal;
    output reg EnAck;
    output reg EnBuff;

    parameter   IDLE = 0,
                WAIT = 1,
                READ = 2,
                UPDATE = 3,
                SEND1 = 4,
                SEND2 = 5;

    parameter READ_STATE_DIRTY_CTRL = 2;
    parameter DIRTY_WR_STATE_DIRTY_CTRL = 3;

    reg [2:0] State, NextState;

    // comb logic
    always @(State, Start, IsEmpty, WriteOp, DirtyVal) begin
        PullEn <= 1'b0;
        RstVal <= 1'b0;
        RstAddr <= 1'b0;
        LdVal <= 1'b0;
        LdAddr <= 1'b0;
        SelVal <= 1'b0;
        EnAck <= 1'b0;
        EnBuff <= 1'b0;

        case (State)
            IDLE: begin
                if (Start)
                    NextState <= WAIT;
            end

            WAIT: begin
                RstVal <= 1'b1;
                RstAddr <= 1'b1;
                if (~WriteOp && ~IsEmpty)
                    NextState <= READ;
            end

            READ: begin
                PullEn <= 1'b1;
                LdAddr <= 1'b1;
                LdVal <= 1'b1;

                if (DirtyVal == READ_STATE_DIRTY_CTRL)
                    NextState <= SEND1;
                else
                    NextState <= UPDATE;
            end

            UPDATE: begin
                SelVal <= 1'b1;
                LdVal <= 1'b1;

                if (~WriteOp)
                    NextState <= SEND2;
            end

            SEND1: begin
                if (~WriteOp && DirtyVal !=
  DIRTY_WR_STATE_DIRTY_CTRL)
                    NextState <= SEND2;
                else if (~WriteOp && DirtyVal ==
    DIRTY_WR_STATE_DIRTY_CTRL)
                    NextState <= UPDATE;
            end

            SEND2: begin
                EnBuff <= 1'b1;
                EnAck <= 1'b1;
                NextState <= WAIT;
            end

        endcase

    end

    // state reg
    always @(posedge Clk or posedge Pwr_off) begin
        if (Rst || Pwr_off)
            State <= IDLE;
        else
            State <= NextState;
    end
endmodule
```

Listing A.18: FSM Dispatcher Verilog implementation

```
module TriBuff #(
        parameter N = 32
    )
    (
        Vin,
        En,
        Vout
    );

    input [N-1:0] Vin;
    input En;
    output reg [N-1:0] Vout;

    always @(Vin, En) begin
        if (En)
            Vout <= Vin;
        else
            Vout <= {N{1'bZ}};
    end
endmodule
```

Listing A.19: Tri-State Buffer Verilog implementation

A.5 Atomic Region Log Buffer (.ICA)

```
module Buffer #(
        parameter N = 32, // Width values
        parameter M = 5, // Buffer size, pow of 2!
        parameter K = 3  // Number of bits for the addresses
    )(
        PushEn,
        PullEn,
        PushVal,
        PullVal,
        IsFull,
        IsEmpty,
        Clk,
        Rst,
        Pwr_off
    );

    input [N-1:0] PushVal;
    output [N-1:0] PullVal;
    input PushEn;
    input PullEn;
    output IsFull;
    output IsEmpty;

    input Clk, Rst, Pwr_off;

    wire ff_cnt_write;
    wire ff_cnt_read;
    wire [($clog2(M)-1):0] addr_read;
    wire [($clog2(M)-1):0] addr_write;
    wire en_cnt_write;
    wire en_cnt_read;
    wire en_cnt_up_down_cnt;
    wire [K:0] vout_num_full_reg;


    // register file
    RegFile #(
        .N          (N),     // width of each register
        .M          (M)      // number of registers
    ) buffer (
        .ReadAddr   (addr_read),
        .WriteAddr  (addr_write),
        .Vin        (PushVal),
        .Vout       (PullVal),
        .REn        (PullEn),
        .WEn        (PushEn),
        .Clk        (Clk),
        .Rst        (Rst),
        .Pwr_off    (Pwr_off)
    );



    // CntN write
    CntN #(
        .N          (K)
    ) addr_cnt_write (
        .Clk        (en_cnt_write),
        .Rst        (Rst),
        .Pwr_off    (Pwr_off),
        .Vout       (addr_write)
    );

    // CntN read
    CntN #(
        .N          (K)
    ) addr_cnt_read (
        .Clk        (en_cnt_read),
        .Rst        (Rst),
        .Pwr_off    (Pwr_off),
        .Vout       (addr_read)
    );

    // UpDownCntN empty-full check
    UpDownCntN #(
        .N          (K+1)
    ) cnt_full_regs (
        .Up         (PushEn),
        .Down       (PullEn),
        .Vout       (vout_num_full_reg),
        .Rst        (Rst),
        .Clk        (en_cnt_up_down_cnt),
        .Pwr_off    (Pwr_off)
    );


    assign en_cnt_write = (Clk & PushEn) | (Clk & Rst);
    assign en_cnt_read = (Clk & PullEn) | (Clk & Rst);
    assign en_cnt_up_down_cnt = (Clk & PushEn) | (Clk & PullEn) |
    (Clk & Rst);

    assign IsFull = vout_num_full_reg[K];
    assign IsEmpty = ~(|vout_num_full_reg);

endmodule
```

Listing A.20: Buffer Verilog implementation

```
module UpDownCntN #(
    parameter N = 32
    )(
        Up,
        Down,
        Vout,
        Rst,
        Clk,
        Pwr_off
    );

    input Up;
    input Down;
    input Rst;
    input Clk;
    input Pwr_off;
    output reg [N-1:0] Vout;

    always @(posedge Clk or posedge Pwr_off) begin
        if (Rst | Pwr_off)
            Vout <= {N{1'b0}};
        else if (Up)
            Vout <= Vout + 1'b1;
        else if (Down)
            Vout <= Vout - 1'b1;
    end

endmodule
```

Listing A.21: Up/Down Counter Verilog implementation

Bibliography

[1] Jie Zhan, Geoff V. Merrett, and Alex S. Weddell. «Exploring the Effect of Energy Storage Sizing on Intermittent Computing System Performance». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.3 (2022), pp. 492–501. DOI: 10.1109/TCAD.2021.3068946 (cit. on pp. 1, 2).

[2] Sebin Shaji Philip, Roberto Passerone, Kasim Sinan Yildirim, and Davide Brunelli. «Intermittent Computing Emulation of Ultralow-Power Processors: Evaluation of Backup Strategies for RISC-V». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.1 (2023), pp. 82–94. DOI: 10.1109/TCAD.2022.3169108 (cit. on pp. 1–3, 8).

[3] Kiwan Maeng et al. «Alpaca: Intermittent Execution without Checkpoints». In: *Proceedings of the ACM on Programming Languages*. Vol. 1. OOPSLA. ACM. 2017, 96:1–96:30 (cit. on pp. 2, 3, 7).

[4] Davide Pala et al. «Freezer: A Specialized NVM Backup Controller for Intermittently Powered Systems». In: *IEEE TCAD* 40.8 (2021), pp. 1559–1572. DOI: 10.1109/TCAD.2020.3025063 (cit. on pp. 2, 3, 7, 45, 46).

[5] Kaede Sakai et al. «Design of an Error-Tolerant Nonvolatile Register for Energy-Aware Intermittent Computing». In: *IEEE 66th MWSCAS*. 2023, pp. 269–273 (cit. on pp. 2, 3, 7).

[6] Sivert Sliper, Domenico Balsamo, Alex Weddell, and Geoff Merrett. «Enabling intermittent computing on high-performance out-of-order processors». In: Nov. 2018, pp. 19–25. ISBN: 978-1-4503-6047-0. DOI: 10.1145/3279755.3279759 (cit. on pp. 3, 8).

[7] Emily Ruppel and Brandon Lucia. «Transactional concurrency control for intermittent, energy-harvesting computing systems». In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1085–1100. ISBN: 9781450367127. DOI: 10.1145/3314221.3314583. URL: https://doi.org/10.1145/3314221.3314583 (cit. on p. 8).

[8] Karthik Ganesan, Joshua San Miguel, and Natalie Enright Jerger. «The What's Next Intermittent Computing Architecture». In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 211–223. DOI: 10.1109/HPCA.2019.00039 (cit. on p. 8).