

UC Santa Barbara

UC Santa Barbara Electronic Theses and Dissertations

Title

A RISC-V Implementation of the Sharing Architecture

Permalink

<https://escholarship.org/uc/item/7gr1d8j3>

ISBN

9798186386124

Author

Bui, Minh

Publication Date

2026-06-08

Peer reviewed|Thesis/dissertation

University of California
Santa Barbara

A RISC-V Implementation of the Sharing Architecture

A thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Electrical and Computer Engineering

by

Minh Bui

Committee in charge:

Professor Jonathan Balkind, Chair
Professor Timothy Sherwood
Professor Dmitri Strukov

June 2026

The Thesis of Minh Bui is approved.

Professor Timothy Sherwood

Professor Dmitri Strukov

Professor Jonathan Balkind, Committee Chair

June 2026

Acknowledgments

I would like to thank Professors Jonathan Balkind, Timothy Sherwood, and Dmitri Strukov for agreeing to be on my thesis committee, as well as providing valuable feedback that helped me arrive at the final version.

This thesis would not have been possible without the support of my advisor, Professor Jonathan Balkind. I am grateful to Jon for allowing me to join ArchLab at the end of my third year, when I was clueless about how to design architectures and hardware, and for the patience and mentorship that let me far outgrow what I thought I was capable of. Throughout this project he has always been available to help, which is much of the reason it grew into a processor capable of running real code. I have learned a lot from you, and thank you for being my advisor for three great years.

I am equally grateful to Dr. Sudhanva Gurumurthi, who mentored me over the nine months of my internship at AMD. Although his guidance did not feed directly into this thesis, the time I spent under his advisement was monumental to my growth as an engineer and a researcher. I learned a lot about how to efficiently approach problems, which enabled me to solve many of the issues encountered during the development of this thesis. Thank you for mentoring me and helping me grow.

I also want to thank the members of the ArchLab for helping me maintain a good balance between research and personal life, providing a friendly environment to grow in, and taking the time to answer any questions I had. I wouldn't have been able to complete this thesis without you.

Abstract

A RISC-V Implementation of the Sharing Architecture

by

Minh Bui

Master of Science in Electrical and Computer Engineering

University of California, Santa Barbara

As computing has evolved, general-purpose processors have been expected to support an increasingly diverse set of applications, from latency-sensitive single-threaded programs to throughput-oriented and memory-intensive workloads. These workloads can differ substantially in their demand for instruction-level parallelism (ILP), memory capacity, and execution resources. A fixed monolithic core must therefore be designed around one compromise point in a much larger design space. Scaling such cores also increases the cost and complexity of tightly coupled microarchitectural structures. Together, workload diversity and inflexible hardware architectures limit both performance and utilization. Prior work has addressed this mismatch through configurable and composable architectures that allow compute resources to be reorganized around workload demand [13, 1, 2]. By grouping or partitioning execution resources, these architectures aim to improve resource utilization and provide a more flexible alternative to fixed-core designs. However, this flexibility introduces new questions about coordination, cross-component communication, and scalability.

This thesis presents a RISC-V implementation of the Sharing Architecture [1]. The implementation realizes a fabric of small core-like superscalar out-of-order execution units, called “slices”, where multiple slices can cooperate on a single instruction stream as a composed virtual core. The design is configurable in slice count, issue width, and cross-slice interconnection topology.

Using bare-metal RISC-V tests and Embench-IoT workloads, this thesis evaluates how

slice count, issue width, and cross-slice interconnection topology affect IPC and pipeline bottlenecks. The evaluation identifies when additional slices improve performance and when coordination, memory ordering, branch handling, or network contention limit scaling.

Contents

Contents	vi
List of Figures	viii
List of Tables	ix
1 Introduction	1
1.1 Motivation	1
1.2 Thesis Question	2
1.3 Contributions	2
1.4 Organization	3
1.5 Artifact Availability	3
2 Background	4
2.1 Superscalar Out-of-Order Execution	4
2.2 Sharing Architecture Model	6
2.3 Scalar Operand Communication	7
2.4 RISC-V Bare-Metal Workloads	8
2.5 Metrics	9
3 Architecture and Implementation	12
3.1 Top-Level Organization	12
3.2 Slice Pipeline	15
3.3 Sequential Window Control and the Pipelined Variant	18
3.4 Register Rename	20
3.5 Scoreboard, Functional Units, and Writeback	24
3.6 Branch and Control Handling	27
3.7 Memory Ordering and LSU Arbitration	28
3.8 Scalar Operand Network	29
3.9 Implementation Status	32
4 Experimental Methodology	33
4.1 Workloads	33
4.2 Configurations	34
4.3 Build and Run Flow	36
4.4 Performance Counters	37

4.5	Correctness Tests	38
4.6	Threats to Validity	39
5	Evaluation	40
5.1	Correctness Summary	40
5.2	Full Butterfly Embench Scaling	42
5.3	Bottleneck Breakdown	45
5.4	Supplemental Direct-Topology Sensitivity	45
5.5	Memory Ordering Impact	47
6	Related Work	48
6.1	Scalable Out-of-Order Architectures	48
6.2	Operand Networks and On-Chip Interconnect	50
6.3	RISC-V Research Platforms	51
7	Discussion and Future Work	53
7.1	What the Current Design Shows	53
7.2	Improving the Scalar Message Boundary	55
7.3	Rename Path	55
7.4	Frontend Decoupling and Instruction Stream Ordering	56
7.5	Branch Recovery	57
7.6	Memory System	57
7.7	Full-System Support	58
8	Conclusion	59
A	Useful Commands	61
A.1	RISC-V Matrix Runner	61
A.2	Directed Memory-Ordering Regression	61
B	Counter Schema	62
	References	64

List of Figures

3.1	High-level organization of the current multi-slice processor. The dashed <code>top.sv</code> boundary encloses the present prototype coordination logic, with instruction and data memories shown outside it.	15
3.2	Overview of one slice and its current inter-slice interfaces.	16
3.3	Organization of the in-progress pipelined slice variant, with a register-staged frontend, fetch-time branch target buffer, and speculative rename. This variant passes the RV32UI <code>riscv-tests</code> suite but is not yet correct across the Embench workloads at higher speculation depths, and is not used for the evaluation in this thesis.	20
3.4	Global rename in the original Sharing Architecture, reproduced from Zhou and Wentzlaff [1]. The left side illustrates grouping slices around a master slice; the right side shows the multi-stage destination rename and broadcast process.	21
3.5	Fields in one <code>fu_scoreboard_entry_t</code> , the per-functional-unit scoreboard entry defined in <code>sharing_pkg.sv</code>	24
3.6	Scalar operand network organization with slice-side event sources, NoC connectors, and a butterfly scalar packet NoC, shown for a four-slice configuration.	30
3.7	Scalar message header, opcode values, and opcode-specific payload fields defined in <code>sharing_pkg.sv</code> . The redirect opcode is defined in the packet format but is not yet the primary frontend redirect mechanism.	31
5.1	RV32UI direct IPC heatmap	41
5.2	Per-workload Embench butterfly IPC	43

List of Tables

4.1	Collected configuration datasets used in the evaluation.	35
5.1	Outcome summary for checked-in evaluation datasets.	40
5.2	Per-workload average run size for the completed Embench butterfly dataset, with NUM_WAYS=2 and local scale factor 1.	44
5.3	Average bottleneck summary for the completed Embench butterfly dataset, with NUM_WAYS=2.	45
5.4	Average IPC for the direct-topology Embench smoke matrix. Each populated cell averages three passing workloads.	46
5.5	Legacy direct versus butterfly average IPC on the common passing Embench smoke subset, with NUM_WAYS=2.	47
B.1	Raw metrics CSV schema.	63

Chapter 1

Introduction

1.1 Motivation

General-purpose processors have become responsible for an increasingly wide range of applications whose behavior differs substantially in instruction parallelism, memory behavior, control flow, and resource demand. Modern systems-on-chip (SoCs) typically expose a fixed set of cores, and a process or thread is assigned to one of those cores regardless of how much of that core’s hardware it can use or whether it would benefit from additional resources. As a result, some workloads leave expensive resources idle, while others are limited by resources that cannot be expanded or reorganized at run time.

One way to improve performance for certain workloads is to build larger superscalar out-of-order cores that expose more single-thread performance. However, these cores depend on tightly coordinated structures for renaming, dependency tracking, wakeup, writeback, branch recovery, and retirement. As these mechanisms scale in width and capacity, their communication and control costs increase, and the resulting performance gains do not necessarily scale with the added complexity.

Configurable and composable architectures address this mismatch by allowing processor resources to be grouped or partitioned according to workload demand. Prior work has

explored this direction through clustered, tiled, and composable architectures, including RAW [4], Core Fusion [13], and the Sharing Architecture [1]. These designs differ in how much control they expose to software and how they coordinate communication between the composed resources.

This thesis studies that idea through a RISC-V implementation of the Sharing Architecture. The design composes multiple superscalar out-of-order slices into a larger virtual core that cooperates on a single instruction stream. The central question is whether additional slices can improve IPC enough to justify the coordination and communication overheads introduced by composition.

1.2 Thesis Question

This thesis evaluates the following question:

How do slice count, issue width, and scalar operand network topology affect the performance and communication bottlenecks of a RISC-V implementation of the Sharing Architecture?

1.3 Contributions

1. A configurable, synthesizable RTL implementation of a multi-slice RISC-V processor that composes local superscalar out-of-order execution slices into a larger virtual core.
2. A scalar message network model for cross-slice operand request/reply, deallocation, and branch/control traffic, with direct and staged topology options.
3. A bare-metal verification and benchmark flow using RISC-V architectural tests and Embench-IoT workloads.

4. IPC and bottleneck counter infrastructure for identifying how frontend, rename/window, issue, functional-unit, ROB, memory, branch, and scalar-network behavior affect performance.
5. An evaluation of slice scaling and topology effects across selected values of NUM_SLICES and NUM_WAYS.

1.4 Organization

Chapter 2 introduces the architectural background needed for the rest of the thesis. Chapter 3 describes the RTL design. Chapter 4 explains the benchmark and measurement flow. Chapter 5 presents the results. Chapter 6 positions the work against related architectures and interconnect approaches. Chapter 7 discusses design limitations and future work. Chapter 8 concludes the thesis.

1.5 Artifact Availability

The complete artifact for this thesis is open source and publicly available at <https://github.com/UCSBarchlab/sharing-arch>, released under the BSD 3-Clause license. The repository contains the configurable multi-slice RISC-V RTL, the bare-metal test harness, the `riscv-tests` and Embench-IoT benchmark flows, and the matrix-runner and summarization scripts used to produce the datasets, figures, and tables reported in Chapter 5.

Chapter 2

Background

2.1 Superscalar Out-of-Order Execution

Superscalar and out-of-order execution are related but distinct techniques for extracting instruction-level parallelism from a single instruction stream. A superscalar processor can fetch, decode, issue, or execute multiple instructions in the same cycle, depending on the width of the corresponding pipeline stages. An out-of-order processor may execute a younger instruction before an older instruction if its operands are ready and the required execution resources are available. Combining these techniques allows the machine to hide latency from long-running operations, such as memory accesses, while still presenting the architectural state as though instructions completed in program order [5, 7].

Register renaming is one of the mechanisms that makes this possible. The instruction set architecture defines a fixed set of architectural registers, but an out-of-order processor maps those registers onto a larger set of physical registers or producer tags. When an instruction is renamed, its destination is assigned a new physical location and its sources are translated to the current producers of the corresponding architectural registers. This removes false write-after-read and write-after-write dependencies, leaving only true read-after-write dependencies between instructions.

After rename, instructions wait in scheduling structures such as reservation stations, issue queues, or instruction windows. These structures track whether each instruction's source operands are ready and select ready instructions for execution when functional units are available. In a superscalar machine, the issue width determines how many ready instructions can be selected and sent to execution in a cycle. In many out-of-order designs, writeback from a completed producer wakes dependent instructions by broadcasting the producer's tag and, in some designs, the produced value. A scoreboard can serve a similar role by tracking which registers or tags are ready. In either case, the scheduler must observe producer completion quickly enough to expose useful parallelism.

Writeback is distinct from commit or retirement. At writeback, an instruction has produced a result that can be consumed by younger dependent instructions. That result may still be speculative if older instructions have not yet retired or if a preceding branch may redirect control flow. A reorder buffer tracks the program order of in-flight instructions so that the processor can retire instructions in order, update architectural state precisely, and discard younger work after exceptions or branch mispredictions.

The same mechanisms that improve single-thread performance also create scaling pressure. A wider or deeper out-of-order core needs more rename bandwidth, larger scheduling windows, more wakeup and select logic, more bypass and writeback paths, and larger retirement structures. These mechanisms are often tightly coupled: producer completion must be communicated to many possible consumers, branch recovery must invalidate younger speculative work, and retirement must preserve global program order. As a result, centralized out-of-order structures can become expensive in area, timing, and power as core width and instruction window capacity increase. This communication pressure is a main reason to consider architectures that distribute execution resources across smaller units and explicitly manage the communication between them.

2.2 Sharing Architecture Model

The Sharing Architecture was proposed as a way to make processor resources configurable below the granularity of a fixed core. Instead of exposing only a set of complete cores, the architecture organizes the chip as a fabric of smaller computational units and shared resources that can be grouped into Virtual Cores, or VCores [1, 2]. A VCore executes a single architectural instruction stream, but the hardware resources used to execute that stream may come from multiple slices. This distinguishes the model from a conventional multicore system, where each core normally runs an independent thread and the core boundary is the basic unit of allocation.

The basic compute unit in the Sharing Architecture is a slice. In the original design, a slice is a simple out-of-order processor-like unit with local execution resources that can be composed with other slices into a larger VCore. This thesis uses the same abstraction, but implements each slice as a superscalar out-of-order RISC-V execution unit. A single-slice configuration behaves like a small core, while a multi-slice configuration allows several slices to cooperate on the same instruction stream as one larger virtual core.

Composition creates a tradeoff between resource availability and coordination cost. Adding slices can increase the amount of instruction window capacity, issue bandwidth, execution bandwidth, and buffering available to a program. However, the composed VCore must still preserve the behavior of one sequential RISC-V instruction stream. Register dependencies, writeback, branch recovery, memory ordering, and retirement all require some communication or synchronization between slices. If this communication is too expensive, the additional execution resources may not translate into higher IPC.

The original Sharing Architecture also considers a broader system in which resources such as ALUs, cache banks, and fetch bandwidth can be provisioned for IaaS cloud workloads. The implementation in this thesis is narrower. It is a configurable, synthesizable RISC-V SoC research implementation focused on the processor slice fabric and scalar communication needed to compose slices into VCores. Shared caches, shared floating-point

units, cloud-market resource allocation, and production hypervisor support are outside the current scope. The purpose of the implementation is therefore not to reproduce the entire cloud-oriented system, but to study the core architectural question: when does slice composition improve performance, and when do the required communication paths become the bottleneck?

2.3 Scalar Operand Communication

When execution resources are distributed across slices, producer-consumer communication can no longer be treated as a purely local event. An instruction issued on one slice may produce a value or status update that is needed by an instruction, scheduler, or bookkeeping structure on another slice. A conventional centralized core often handles this kind of event with short bypass paths, common-data buses, or tightly coupled wakeup and writeback logic. In a sliced design, the same information must cross slice boundaries with enough latency and bandwidth to preserve useful instruction-level parallelism.

Scalar operand networks were proposed as a way to carry these fine-grain producer-consumer messages in partitioned ILP architectures. Rather than communicating only through memory or through a globally centralized register file, a scalar operand network transports register-sized values, producer tags, and readiness information between distributed execution units [3]. The original Sharing Architecture uses this idea for operand request and reply messages between slices [1]. The network lets a VCore be composed from multiple slices while avoiding a single monolithic broadcast structure for all cross-slice communication.

The important design question is where the boundary should be placed between specialized local hardware and general inter-slice transport. A direct all-to-all fabric can approximate the behavior of a centralized broadcast path, but its wiring and fanout grow quickly as more slices are added. A packetized or staged network can reduce direct fanout

and provide a more scalable structural model, but it introduces routing latency, buffering, arbitration, and possible contention. These costs matter because scalar communication is often on the critical path for dependent instructions.

This thesis models selected cross-slice events as scalar messages. Operand request and response messages move producer values from owner slices to consumer slices on demand. Deallocation messages communicate when resources associated with older instructions can be released. Branch and control messages communicate events that may require redirecting or flushing younger work. These events are natural candidates for packetization because they already cross slice boundaries and can be represented as small opcode-tagged messages.

Not every communication path is treated this way in the current implementation. Rename is deliberately kept on a direct specialized path. In the current implementation, rename-related communication occurs across slices in lock step, so multiple slices can need the rename path at the same time. Routing this traffic through the same modeled scalar network would likely make rename-network contention dominate CPI, hiding the effects of operand request/reply, deallocation, and control-message topology. The direct rename path is therefore a deliberate approximation and accepted design cost that this thesis treats separately, rather than a claim that rename communication is free.

2.4 RISC-V Bare-Metal Workloads

RISC-V is a useful target for this work because the base ISA is open, well-specified, and small enough to support focused microarchitectural experimentation [8]. The implementation in this thesis runs bare-metal RISC-V programs rather than a full operating system. A bare-metal program executes directly on the test harness with a simple linker script, startup code, local instruction and data memories, and a `tohost`-style pass/fail mechanism. This keeps the evaluation focused on instruction execution, slice coordination,

memory ordering, and scalar communication rather than privileged software, interrupts, virtual memory, or device emulation.

The first class of workloads is drawn from the RISC-V architectural test suites. These tests are short assembly programs that check individual ISA features such as integer arithmetic, branches, jumps, loads, stores, and sign-extension behavior. They are primarily correctness tests, but they are also useful performance probes because they isolate particular instruction classes and control-flow patterns. Passing these tests across multiple slice counts and issue widths gives confidence that the configurable design preserves the architectural behavior of a single RISC-V instruction stream.

The second class of workloads is Embench-IoT, a free and open benchmark suite designed for deeply embedded systems with no operating system, minimal C library support, and small memory footprints [10]. These programs provide more realistic mixes of integer computation, memory accesses, branches, and loop behavior than individual architectural tests, and the benchmark suite is maintained in an open-source repository. In this project, the Embench sources are compiled as freestanding RISC-V programs and linked into the same local memory and `tohost` harness used by the architectural tests.

This bare-metal scope is a deliberate limitation. It does not measure full system behavior, operating-system overhead, cache coherence, I/O, or virtual memory effects. Instead, it provides a controlled environment for studying the microarchitectural questions central to this thesis: how slice count, issue width, and scalar interconnect topology affect IPC and bottleneck behavior.

2.5 Metrics

The primary performance metric is instructions per cycle (IPC), defined as

$$\text{IPC} = \frac{\text{retired instructions}}{\text{cycles}}. \tag{2.1}$$

The retired instruction count measures instructions that have completed and retired in architectural order. This is different from the number of instructions fetched, issued, or executed, because speculative or flushed work may consume resources without changing architectural state. Cycle count measures the number of simulated processor cycles in the run interval reported by the harness. The reciprocal metric, cycles per instruction (CPI), is sometimes more convenient when discussing stalls: a higher CPI means that each retired instruction required more cycles on average.

Topology refers to the organization used to transport selected scalar messages between slices. In this thesis, direct topology models all-to-all communication with minimal modeled routing structure, while staged topologies model a more structured packet path with additional latency and potential arbitration. Network contention refers to cases where a scalar message is ready to move but cannot immediately advance because another message, stage, or destination is using the required transport resource. Contention can reduce IPC even when the program has useful instruction-level parallelism, because dependent work may be waiting on a remote operand response, deallocation, or control message.

The design also records bottleneck counters. These counters are diagnostic signals rather than perfect causal attribution, because several conditions can be true in the same cycle. For example, a slice may be waiting on operands while the frontend is also unable to provide useful work, or a full ROB may interact with memory-ordering stalls. The counters are therefore used to identify dominant trends and compare configurations, not to prove that each lost cycle has exactly one cause.

Important counter categories include frontend stalls, rename or instruction window stalls, issue stalls, functional-unit and operand-read activity, ROB occupancy and head blocking, branch redirects, memory stalls, and scalar-network send/receive behavior. These counters help distinguish workloads that are mostly limited by available throughput from workloads that expose latency on a critical dependency path. In the evaluation

chapters, IPC trends are interpreted together with these counters so that changes in performance can be related back to specific mechanisms in the multi-slice design.

Chapter 3

Architecture and Implementation

3.1 Top-Level Organization

The implementation is organized as a configurable RISC-V SoC built around an array of composable execution slices. The top-level module, `top.sv`, instantiates the slice array, the rename fabric, the scalar message network, frontend control, memory arbitration, and performance-counter aggregation. For simulation, the top level also instantiates local instruction and data memories used by the bare-metal test harness. For synthesis-oriented builds, those test memories can be removed and replaced with top-level instruction and data memory ports.

The two primary structural parameters are `NUM_SLICES` and `NUM_WAYS`. `NUM_SLICES` controls the number of slice instances in the composed processor. `NUM_WAYS` controls the number of instructions each slice can accept in a rename window, and maps to the RTL issue-width parameter `NUM_ISSUES`. Together, these parameters determine the amount of backend state and execution bandwidth available to the virtual core. The inter-slice communication parameters `SCALAR_NOC_TOPOLOGY` and `RENAME_NET_TOPOLOGY` select the modeled transport for scalar messages and rename traffic, respectively. `WB_BCAST_TOPOLOGY` remains as a compatibility alias in the build flow, but cross-slice operand movement is no

longer modeled as a writeback-broadcast network.

The slice vocabulary and global/local communication split follow the Sharing Architecture lineage [1, 2], but the implementation target is a synthesizable bare-metal RISC-V RTL model. The most important RTL modules are:

- `src/top.sv`: top-level SoC integration, slice instantiation, frontend coordination, memory arbitration, inter-slice fabrics, and counter aggregation.
- `src/rtl/sharing_pkg.sv`: global parameters, RISC-V instruction formats, rename-map types, functional-unit definitions, and scalar message formats.
- `src/rtl/sharing_slice.sv`: the parameterized superscalar out-of-order slice pipeline.
- `src/rtl/frontend/sharing_frontend.sv`: frontend sequencing and instruction-cache request/response coordination.
- `src/rtl/sharing_scoreboard.sv`: source readiness, issue selection, and functional-unit dependency tracking.
- `src/rtl/sharing_rob.sv`: in-order retirement and precise bookkeeping for each slice.
- `src/rtl/sharing_rename_fabric.sv`: the current direct primary/secondary rename-message fabric.
- `src/rtl/interconnect/sharing_scalar_noc.sv`: direct and staged packet transport for scalar cross-slice messages.
- `src/rtl/sharing_son.sv`: scalar message encode/decode at the slice boundary.

Conceptually, the design separates local slice work from global coordination. Each slice fetches, decodes, renames, schedules, executes, and retires a portion of the shared

instruction stream. The top level supplies the coordination needed for the slices to behave like one virtual core: it accepts rename windows only when all slices are ready, distributes rename information through the rename fabric, routes scalar messages through the scalar network, selects global branch redirects, and arbitrates data-memory requests by dynamic age. This organization preserves a single architectural RISC-V instruction stream while allowing the amount of participating slice hardware to scale with `NUM_SLICES`.

Several global structures remain in `top.sv` in the current implementation. These structures are intentional correctness and evaluation scaffolding: they make it possible to run bare-metal workloads and collect slice-scaling results before every control policy has been moved into slice-native logic. The long-term design goal is for the top level to become mostly structural, wiring together slices and interconnects while the control needed to participate in a composed VCore is implemented at the slice boundary. Reducing fixed global control is necessary for the architecture to provide true flexibility rather than simply replacing one monolithic core with another globally constrained structure.

Instruction fetch illustrates this boundary. Each slice presents an instruction memory request/response protocol, but the current design still coordinates window acceptance, redirects, and slice ordering through top-level logic for one composed group. A more complete Sharing implementation would include explicit grouping and ordering logic so that different sets of slices can form VCores and issue coherent instruction-memory requests without relying on a fixed global policy in `top.sv`.

Frontend and Instruction-Memory Path

The current implementation includes OpenHW Group’s `cv-hpdcache` RTL as a submodule [9]. HPDcache was originally developed as an open-source high-performance L1 data cache for RISC-V cores; in this project it is adapted for the instruction side through a wrapper layer, referred to in this work as HPIC. The wrapper fits the HPDcache request/response interface into the bare-metal frontend used by the slice array. This integration

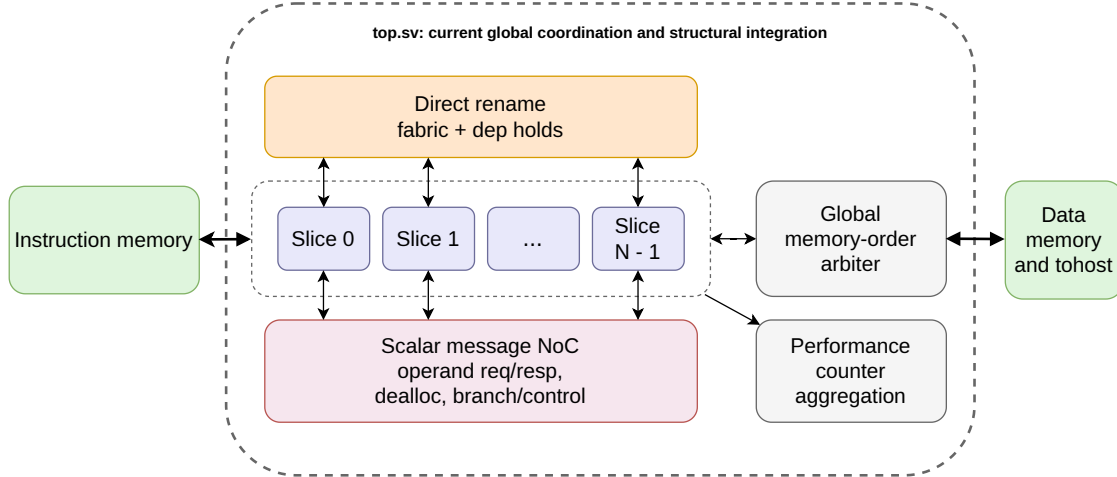


Figure 3.1: High-level organization of the current multi-slice processor. The dashed `top.sv` boundary encloses the present prototype coordination logic, with instruction and data memories shown outside it.

was initially developed as part of CS254 Advanced Computer Architecture. Data-memory behavior is modeled separately by the conservative shared-memory mechanism described later in this chapter.

3.2 Slice Pipeline

The slice pipeline is organized around two coupled forms of progress. The first is a window-admission path that accepts a group of fetched instructions, decodes them, performs global and local rename, and attempts to allocate them into the local scoreboard and ROB. The second is a per-functional-unit execution path that continues independently after an instruction has been admitted. This organization is what makes the implementation superscalar and out-of-order: a slice may accept multiple instructions from a window, and already admitted instructions do not have to wait for the frontend and rename FSM to return to a particular state before they can read operands, execute, or write back.

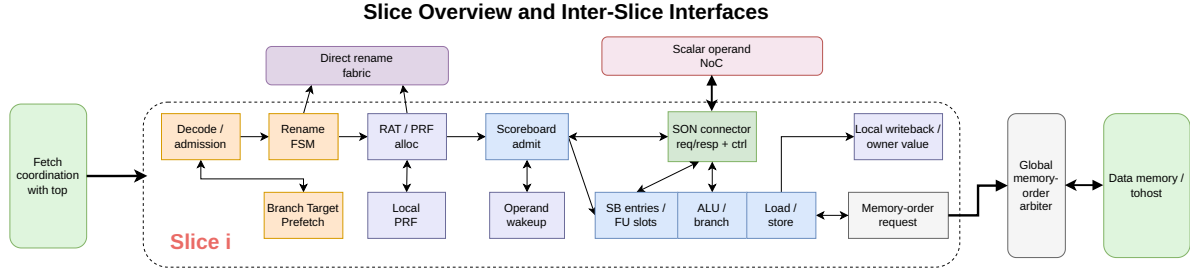


Figure 3.2: Overview of one slice and its current inter-slice interfaces.

At the front of the slice, the current implementation accepts a fetched window from the top-level/frontend interface only when the global window policy permits it. The slice then sends each issue way to its decoder, latches format-independent decoded fields, masks invalid ways after control-flow instructions, and records which destinations require rename. The control path then moves through the rename sequence:

```

S_RESP_FE
S_REQ_DECODE
S_RESP_DECODE
S_RENAME
S_BROADCAST_RECEIVE
S_PBROADCAST_RESPOND / S_SBROADCAST_RESPOND
S_RAT_REMAP
S_LOCAL_RENAME
S_SCOREBOARD_CHECK

```

The rename portion of this sequence first allocates logical registers for local destinations, exchanges destination mappings with the other slices, merges those mappings into the global architectural view, and then performs local physical register allocation and source lookup. The final state, `S_SCOREBOARD_CHECK`, is the admission point into the backend. For each valid issue way, the slice checks ROB space, memory-order restrictions, scoreboard structural hazards, same-window prefix ordering, and outstanding destination hazards. Accepted ways allocate scoreboard and ROB entries; unaccepted ways remain valid and retry in a later cycle. Once all valid ways from the current window have either been accepted or masked, the slice can return to the frontend response state and begin

processing the next window.

This backend should therefore be described as a scoreboard-based out-of-order prototype rather than a fully decoupled out-of-order machine. The scoreboard entries provide some buffering after rename, but accepted instructions are allocated directly into functional-unit slots rather than into a separate instruction scheduling queue (ISQ). Without an ISQ, the frontend and rename path can only move ahead as far as scoreboard and ROB resources allow, so structural or memory-order stalls can propagate back to window admission more directly than they would in a design with a larger scheduling window.

This mechanism is close in spirit to the CDC 6600 scoreboard, which was an important design reference for this work [6]. In both cases, a scoreboard records issue-time dependencies and functional-unit usage, holds waiting operations near the execution resources, and allows independent operations to proceed when their operands and units are available. The analogy is limited because this implementation also includes register renaming, local PRFs, ROB, and slice-level communication, but the shared idea is the use of scoreboard entries as a compact scheduling structure rather than a large decoupled ISQ.

Execution after admission is not controlled by this window FSM. Each scoreboard entry occupies a functional unit slot and advances through a smaller per-FU sequence:

`STAGE_READ_OPS -> STAGE_EXEC -> STAGE_WRITEBACK -> STAGE_IDLE`

In `STAGE_READ_OPS`, the scoreboard waits until the instruction’s local or remote operands are available. Local operands are read from the PRF, with a local bypass path for same-cycle writeback, while remote operands become ready only after an explicit scalar operand response returns from the owning slice. In `STAGE_EXEC`, ALU operations use fixed operation latencies, branch operations produce control-resolution events, and load/store operations wait for the conservative global data-memory arbiter. In `STAGE_WRITEBACK`, completed operations update the local PRF, wake dependent instructions on the same slice, and make the result available to future remote operand requests.

This separation means the slice can be processing a later fetch/decode/rename window while older instructions continue through functional units, wait on remote operands, or retire through the ROB. The implementation is still more conservative than a commercial out-of-order backend because several events remain globally synchronized, including rename broadcast, branch recovery, and memory ordering. Nevertheless, the pipeline is not in-order after admission: independent scoreboard entries can make progress while older entries wait on operands, memory, or writeback.

3.3 Sequential Window Control and the Pipelined Variant

The slice control path evaluated in this thesis is a single sequential finite state machine. The slice declares a twelve-entry `state_t` enumeration in `sharing_slice.sv`, and each fetched window advances through a fixed linear chain of ten of these states before the slice can accept the next window:

```
S_RESP_FE -> S_REQ_DECODE -> S_RESP_DECODE -> S_RENAME ->
S_BROADCAST_RECEIVE ->
S_PBROADCAST_RESPOND / S_SBROADCAST_RESPOND ->
S_RAT_REMAP -> S_LOCAL_RENAME -> S_REMOTE_OPERAND_REQ ->
S_SCOREBOARD_CHECK -> (back to) S_RESP_FE
```

A slice visits exactly one of `S_PBROADCAST_RESPOND` or `S_SBROADCAST_RESPOND` depending on whether it is acting as the primary or a secondary slice for the window. The remaining enumeration entry, `S_EXECUTE`, corresponds to the decoupled per-functional-unit execution path described in Section 3.2 rather than to an additional admission step; post-admission execution is the only part of the slice that is not driven by this sequential window FSM.

Because this chain is strictly sequential and is not overlapped across consecutive windows, the full per-window admission latency is paid for every window regardless of

how much independent work the backend could otherwise accept, heavily reducing IPC. This is the structural reason for the bottleneck behavior reported in Chapter 5: the slice FSM spends most of its time in the frontend, rename, and broadcast-rendezvous states (Table 5.3), and that residency grows with slice count as the lock-step admission chain must rendezvous across more slices.

A natural next step is to convert this control path into a register-staged pipeline so that distinct windows can occupy the decode, rename, broadcast, and scoreboard-admission stages concurrently. Preliminary work on such a variant exists, including a register-staged fetch path, a pipelined frontend module, a fetch-time branch target buffer, and speculative renaming with misprediction recovery; its organization is sketched in Figure 3.3. This pipelined slice now passes the full RV32UI `riscv-tests` suite, and on a warm-cache, branch-prediction-friendly custom workload it reaches a peak of roughly 1.9 IPC in the two-slice, two-way (2s2w) configuration. However, the Embench workloads remain incomplete on this variant: correctness problems surface as the branch speculation depth is increased, and resolving them is still in progress. Because these issues are unresolved and the broadcast, rename-visibility, and branch-redirect mechanisms are tightly coupled and fragile under reordering, we chose not to base this thesis on the pipelined variant. This thesis instead proceeds with the sequential FSM-based slice: all results in Chapter 5 use it, and completing the pipelined slice control path is identified as future work in Chapter 7.

Slice Overview and Inter-Slice Interfaces

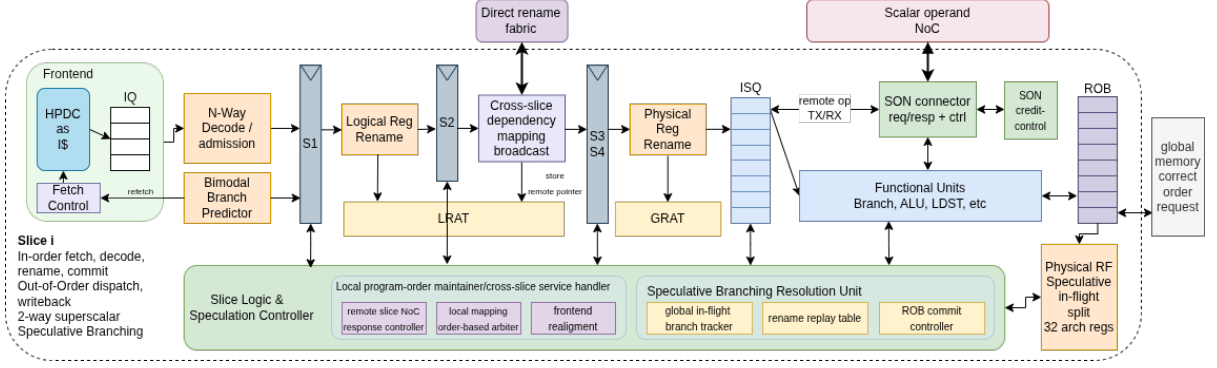


Figure 3.3: Organization of the in-progress pipelined slice variant, with a register-staged frontend, fetch-time branch target buffer, and speculative rename. This variant passes the RV32UI `riscv-tests` suite but is not yet correct across the Embench workloads at higher speculation depths, and is not used for the evaluation in this thesis.

3.4 Register Rename

As in the original Sharing Architecture, rename is split into a global and local view [1]:

- The GRAT maps architectural registers to a logical register and slice.
- The LRAT maps each logical register and slice to a local physical register file entry.
- The PRF stores local values and readiness state.

The implementation follows this two-level organization. Each slice keeps a copy of the global architectural map, represented as an architectural register to $\{\text{logical register, slice id}\}$ mapping. The logical register identifies a producer in the global slice namespace, while the slice id identifies which slice owns the corresponding physical storage. Each slice also keeps a private LRAT and PRF free list. The LRAT translates local logical registers to local PRF entries, and the PRF stores the actual data. Register zero is handled specially: instructions that write `x0` do not allocate a new rename mapping, and PRF entry zero is reserved so writes to `x0` never consume physical state.

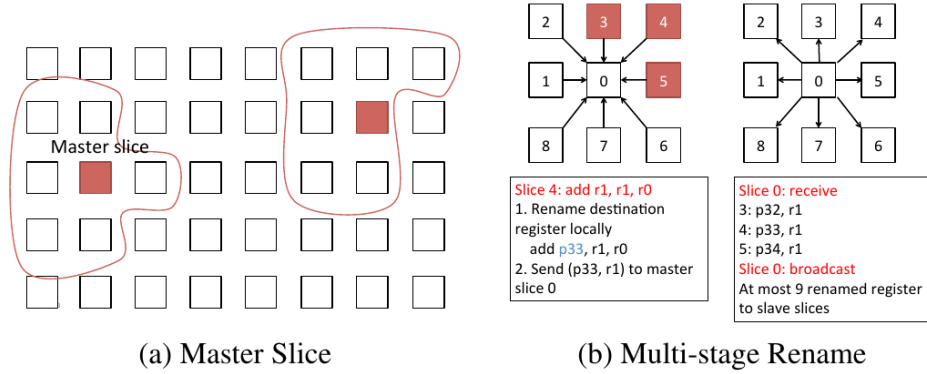


Figure 3.4: Global rename in the original Sharing Architecture, reproduced from Zhou and Wentzlaff [1]. The left side illustrates grouping slices around a master slice; the right side shows the multi-stage destination rename and broadcast process.

Rename is performed once per accepted frontend window. After decode determines which issue ways write architectural destinations, the `S_RENAME` state allocates logical registers for those destinations from the slice’s local logical-register free list. These allocations are not enough by themselves, because other slices in the same composed VCore must observe the same architectural-to-logical mapping before they can rename their source operands. The implementation therefore uses a primary/secondary broadcast protocol. The primary slice collects destination mappings from secondary slices, forms a window-wide broadcast map, and sends that map back to all slices. This protocol is wrapped by `sharing_rename_fabric.sv`, but its timing is still direct: it preserves the existing lock-step rename barrier rather than modeling a latency-tolerant packet network.

Once every slice has the broadcast map, `S_RAT_REMAP` merges the window’s destination mappings into the GRAT in program order. Program order is defined by the composed window position, `slice_id * NUM_ISSUES + way`. If two instructions in the same window write the same architectural register, the later writer naturally overwrites the earlier writer in the GRAT. The implementation also refines the old mapping stored in the ROB for same-window WAW chains: a younger writer’s old mapping may be the logical register allocated by an earlier writer in the same window, not only the mapping that existed

before the window was renamed.

Source rename uses the same program-order view. For each source operand, the slice begins with the previous-window GRAT mapping, then scans the current window's broadcast entries for the most recent earlier writer to that architectural register. This handles same-window RAW dependencies across both issue ways and slices. After the global logical source has been selected, `S_LOCAL_RENAME` performs the local part of rename: it allocates a destination PRF for local writes, updates the LRAT for the newly allocated logical register, and looks up local PRF indices for sources whose producing slice is the current slice. If a source is remote, the instruction carries the remote {slice, logical register, generation} tag into the scoreboard instead of reading a local PRF entry. The same rename step also emits a dependency-hold record for that source through the rename fabric. This hold is not the data request itself; it is a lifetime guarantee telling the owner slice not to recycle that logical-register generation until the later operand request has been serviced.

The generation field is an age tag for a reusable logical-register index. A logical register names a storage slot, not a unique dynamic producer, so packets and scoreboard entries use {slice, logical register, generation} as the producer identity. This lets delayed operand responses, deallocation messages, and retained completed values be ignored if they refer to an older use of the same slot.

The dependency hold is packaged with the rename broadcast because rename is the first point where the exact cross-slice RAW dependency and producer generation are known. Waiting until the consumer reaches `STAGE_READ_OPS` would reopen the lifetime race: the producer could commit, receive deallocation, and make the logical register available for reuse before the operand request reached the owner. Other approaches are possible but less attractive for this prototype. The ROB head could be held for a maximum scalar-network delay, but that delay is hard to calculate once topology, arbitration, and backpressure are included. The owner could require an explicit acknowledgment before

allowing commit progress, but this would put NoC round trips on the commit path and severely reduce IPC. A third option is a replay table of committed producer values, but that requires a bounded table size, overflow policy, and backpressure mechanism; without those, the table size is effectively another magic constant. Sending the hold with the existing rename exchange gives the owner early knowledge of remote consumers without turning operand request/reply into a global commit-acknowledgment path. This choice favors a simple architectural-correctness guarantee for mistimed request windows, while accepting that it increases the metadata carried by the rename network.

Commit completes the rename lifecycle. The ROB entry for each renamed instruction records the new logical/physical destination and the old mapping it replaces. When the instruction commits, the old mapping can be released. If that old mapping belongs to the same slice, the slice frees the old PRF entry, returns the logical register to its local free list, and invalidates the LRAT entry. If the old mapping belongs to another slice, the committing slice emits a cross-slice deallocation message through the scalar operand network so the owner slice can free the corresponding LRAT and PRF state. Deallocation is generation checked and is treated only as owner-side resource reclamation. It is not a cache-coherence event for remote operand data.

The current rename path is intentionally specialized and direct. It is not yet a slice-native, latency-insensitive scalar-message protocol. This choice is a deliberate approximation for the thesis experiments: rename is synchronized across slices and multiple slices can require the rename fabric at the same time, so routing rename through the same modeled scalar network as operand request/reply, deallocation, and branch/control traffic would likely make rename contention dominate CPI. Keeping rename on a direct path lets the scalar-network evaluation focus on operand request/reply, cross-slice deallocation, and control messages while still exposing rename synchronization as a separate bottleneck. The dependency-hold metadata described above is forwarded alongside the direct rename exchange so that the operand-request protocol has a correctness guarantee without turning

rename into the scalar data path.

3.5 Scoreboard, Functional Units, and Writeback

After rename, accepted instructions enter a local scoreboard. The current scoreboard is closer to the CDC-style organization described in Section 3.2 than to a modern reservation-station scheduler. Each scoreboard entry is tied to a functional-unit slot, and the scoreboard tracks whether that slot is busy, which operation it contains, which architectural and logical registers it references, and whether each source operand is ready. The design uses the usual Qj/Qk producer tags and Rj/Rk readiness bits. A separate result-status table records whether an architectural destination is currently pending and which local FU is expected to produce it.

Control			Destination			Sources						Producer and Wakeup State					Remote Slice		Execution	
busy	stage	op	fi	fi_lreg	fi_gen	fj	fk	fj_lreg	fk_lreg	fj_gen	fk_gen	qj	qk	qj_valid	qk_valid	rj	rk	qj_slice	qk_slice	cycles_remaining

Figure 3.5: Fields in one `fu_scoreboard_entry_t`, the per-functional-unit scoreboard entry defined in `sharing_pkg.sv`.

Admission from `S_SCOREBOARD_CHECK` is constrained by both backend capacity and dependency state. For each valid issue way, the scoreboard searches for a free FU of the required type: ALU, load/store, branch, or FPU. If no matching FU is free, the way reports a structural hazard and remains in the window-admission path until it can retry. The issue logic also preserves a program-order prefix within the current slice window: a later issue way cannot allocate if an earlier valid way stalls. This avoids losing the unaccepted instruction when the slice retries the current window.

When an instruction is allocated, the scoreboard records its destination logical register and the logical/source tags produced by rename, including the logical register generation. If a local source is waiting on an older local producer, the entry stores the producing FU id in Qj or Qk and clears the corresponding ready bit. Local producer matching uses

both logical register and generation, so a reused logical register cannot accidentally wake a consumer of an older generation. If a source is remote, the entry records the remote slice, logical-register tag, and generation. A remote source becomes ready only when a matching explicit operand response arrives, rather than from any stale cached value. Same-window RAW dependencies are therefore represented in the scoreboard even though rename has already assigned distinct logical and physical destinations.

The per-FU execution state then advances independently of the frontend FSM. In `STAGE_READ_OPS`, an entry waits until both operands are ready. Local operands are read from the PRF, with special handling for `x0` and for `AUIPC`. The implementation includes a limited local bypass path: if a source PRF is read in the same cycle that another FU writes the same PRF, the read sees the writeback result rather than the old PRF value. This is not a full dependent-result bypass network that allows a consumer to execute before the producer reaches writeback; dependent local consumers are woken when the producer writes back and clears their `Qj/Qk` dependency.

Remote operand handling now follows the demand-driven request/reply model used in the original Sharing Architecture more closely. When a FU reaches `STAGE_READ_OPS` with a remote source that is still not ready, the slice sends one `SCALAR_MSG_OPERAND_REQ` to the owning slice. The request carries the requester FU id, operand select bit, source logical register, and generation. A per-FU `remote_req_sent` bit prevents the same blocked operand from injecting a request every cycle. If a response returns while the instruction is still in the rename/admission path, the slice stores the value in the issue-window context and transfers it into the FU only if the source {slice, logical register, generation} still matches.

The owner slice services operand requests from a small queue instead of deasserting NoC receive-ready whenever a requested value is not yet available. The service logic scans for a request whose generation is ready in the current LRAT, available through a same-cycle writeback bypass, or present in the retained logical-register history. When the

data is available, the owner returns `SCALAR_MSG_OPERAND_RESP` to the requester, carrying the requester FU id, operand select, logical register, generation, and data. The requester latches the data beside the waiting FU and the scoreboard marks only that source ready. If a request is stale because its consumer was killed by control flow and the matching generation is no longer live or retained, the owner drains it without a response.

This path replaces the earlier remote writeback broadcast cache. Cross-slice writeback is no longer used to populate remote shadow values, and commit deallocation no longer invalidates such a cache. Correctness instead comes from three mechanisms: generation-qualified source tags, rename-time dependency holds that prevent premature logical-register reuse, and retained completed values for late same-generation requests after the owner PRF can be freed.

In `STAGE_EXEC`, the slice datapath drives the selected functional unit. Integer ALU operations execute through the local ALU instances with fixed latencies. Branch and jump operations compute control-resolution information that is sent toward the scalar control path and, in the current top-level design, also participates in direct frontend redirect handling. Load/store operations use the load/store FU to compute an address and then wait for the global data-memory arbiter described in the next section. Loads complete when a data memory response is returned; stores complete when the arbiter grants the write.

In `STAGE_WRITEBACK`, the scoreboard asserts a writeback event when the local write-back path is available. The slice writes nonzero destinations into the local PRF, marks the corresponding ROB entry complete, clears the functional unit slot, updates the owner-side logical-register value state, and wakes local dependent scoreboard entries. Remote slices receive the value only through an operand request to the owner slice and a matching response.

This organization exposes several bottlenecks that the evaluation counters track directly: scoreboard admission stalls, structural hazards, WAW stalls, operand wait

cycles, remote operand wait cycles, FU execution cycles, writeback stalls, remote operand request/response counts, and remote deallocation counts. These counters are important because the scoreboard provides real out-of-order progress after admission, but its limited buffering, per-FU allocation model, and dependence on scalar-network request/reply still allow communication and structural hazards to dominate CPI.

3.6 Branch and Control Handling

Branch handling is one of the most conservative parts of the current implementation. The design does not yet implement speculative checkpointing of rename, scoreboard, ROB, and memory state. Instead, after a surviving control-flow instruction is decoded, top-level control marks a branch as pending and prevents new backend windows from being accepted until that control-flow instruction resolves in the branch functional unit. Older admitted instructions may continue executing, but younger windows are not allowed to enter the backend speculatively past the unresolved branch.

The slice decode stage identifies control-flow instructions before scoreboard admission. Within a slice window, any issue way after a decoded control instruction is masked off. Across slices, the top level computes whether an earlier slice in the composed window contains a control instruction and passes that information into later slices. This effectively enforces a first-control-in-window policy: only instructions up to the first control-flow operation in program order survive backend admission. This policy is simple and avoids rollback complexity, but it serializes the frontend/rename path around branches and jumps.

The frontend still performs a limited prediction-style optimization. For conditional branches and JAL, the target immediate is known at decode, so the slice emits a predicted-target event. Conditional branches are currently treated with an always-taken policy, and JAL is always treated as taken. The top level can use that target to redirect instruction

fetch immediately, so the frontend may prefetch the predicted path while the backend waits for execute resolution. This does not make the backend speculative: `window_accept` remains blocked while the branch is pending. The optimization only reduces the amount of frontend refetch work after the branch resolves.

At execution, the branch FU compares operands or computes the jump target and emits a branch-resolution event. Conditional branches select either the branch target or the fall-through PC; JAL and JALR produce the link value and a taken target. JALR remains effectively unpredicted because its target depends on the source register value and is only known after operands are read. When the branch resolves, top-level control clears the pending-branch state. A recovery redirect is sent only if no prediction was recorded, or if the resolved restart PC differs from the predicted target. If the predicted target matches the resolved restart PC, the prefetched path can be kept.

Branch and control metadata have been moved partway into the scalar packet interface. The slice-local `sharing_son.sv` adapter can package decoded control events, predicted targets, and resolved branch results as scalar messages. However, this is not yet a fully slice-native branch recovery protocol. The current top level still selects the relevant branch event, maintains the single global pending-branch state, and fans out the final frontend redirect through direct control wires. A more complete implementation should represent redirects and flushes as scalar message classes with explicit acknowledgment or stall semantics so that `top.sv` becomes mostly structural rather than a control oracle.

3.7 Memory Ordering and LSU Arbitration

The original Sharing Architecture proposes distributed load/store queues and load/store sorting across slices, drawing on unordered load/store queue ideas such as late binding [1, 11]. The current RISC-V implementation instead uses a conservative global memory-ordering mechanism. Each memory operation receives a dynamic age, and a top-level

arbiter grants data-memory access to the oldest ready operation. Younger memory operations are blocked behind older pending memory operations across slices. This behavior is similar in spirit to an LSQ because it uses age information to preserve memory order, but it does not implement a full speculative load/store queue.

The main limitation is store handling. In a production out-of-order design, stores usually compute their address and data early but remain in a store queue until they are safe to commit. Holding stores until commit allows the processor to execute memory operations speculatively while still recovering precisely from branch mispredictions or exceptions. In this implementation, a store updates data memory as soon as the age arbiter grants the write. There is no commit-time store buffer that can later cancel or delay the update. The mechanism therefore preserves the ordering needed by the current bare-metal tests, but it is more conservative than a realistic LSQ and does not provide full speculative memory execution or precise store recovery.

3.8 Scalar Operand Network

The scalar network carries opcode-tagged messages for selected cross-slice events. This is a narrower implementation of the scalar operand communication problem described by Zhou and Wentzlaff [1] and by earlier scalar operand network work [3]. Direct topology preserves the old all-to-all behavior. Butterfly topology uses staged 2-by-2 switches with valid/ready backpressure and fair arbitration.

This packet interface is the part of the current design closest to the intended slice-native communication model. Each slice packages local semantic events into a `scalar_msg_t` packet containing an opcode, source metadata, payload, and destination mask. The external NoC routes these opaque packets; opcode decode happens near the receiving slice rather than in `top.sv`. Operand request/reply, deallocation, and branch/control messages therefore already use an opcode-and-destination packaged routing model, even

though final frontend redirect and some fetch coordination remain direct top-level control paths.

Scalar Operand Network (SON): slice packet adapter and scalar NoC

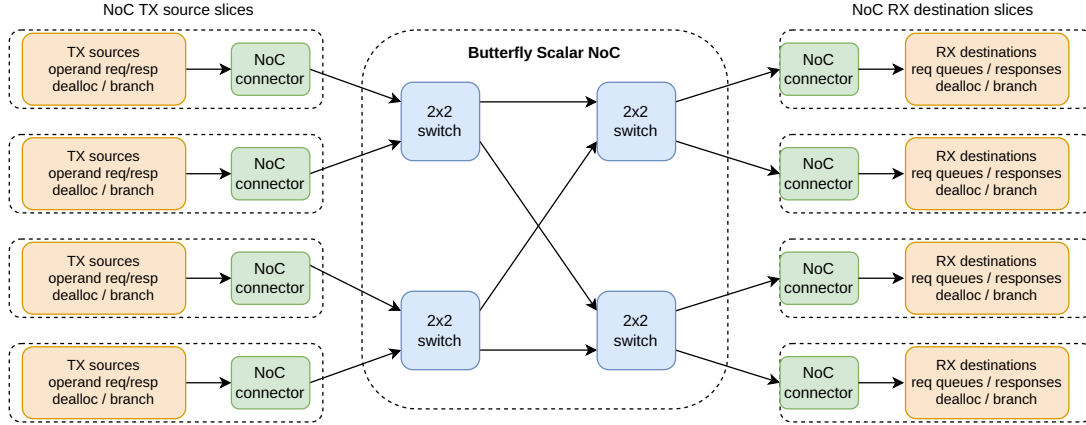


Figure 3.6: Scalar operand network organization with slice-side event sources, NoC connectors, and a butterfly scalar packet NoC, shown for a four-slice configuration.

NoC Connector

The NoC connector shown in Figure 3.6 is the slice-local `sharing_son.sv` adapter. On the transmit side, it observes local scalar event sources from the slice, packs them into a `scalar_msg_t`, and presents transmit valid, destination-mask, and packet signals to the scalar NoC. The ready signal from the NoC provides backpressure to the adapter. Operand requests target the producer slice, operand responses target the requester slice, deallocation messages target the old owner slice, and branch/control messages are currently sent to the primary slice that participates in the global branch control path.

scalar_msg_t						
opcode[2:0]		source slice	opcode-specific payload			

RESERVED0	DEALLOC	BRANCH	REDIRECT
3'd0	3'd1	3'd2	3'd3

OPERAND_REQ	OPERAND_RESP
3'd4	3'd5

Payload kind	Fields					
Operand request	requester FU	operand sel	lreg	gen		
Operand response	requester FU	operand sel	lreg	gen	data	
Deallocation, repeated per commit way	valid	target_slice	lreg	gen		
Branch/control	control_decoded	pred_valid	pred_target_pc	resolved	taken	restart_pc

Figure 3.7: Scalar message header, opcode values, and opcode-specific payload fields defined in `sharing_pkg.sv`. The redirect opcode is defined in the packet format but is not yet the primary frontend redirect mechanism.

On the receive side, the connector watches the per-source receive-valid and message channels delivered by the NoC. When a packet is valid, the connector decodes the opcode and raises the corresponding slice-local semantic valid signal. Operand request packets enter the owner-side request queue; operand response packets carry the returning value and destination FU context; deallocation packets raise a deallocation event when the embedded target slice matches the local slice; and branch packets expose the decoded control, prediction, and resolution fields.

In butterfly mode, each source keeps a pending packet and destination mask. A round-robin `sharing_arb` chooses one destination from that mask before the packet is injected into the NoC. Within the NoC, each 2-by-2 switch routes packets using the current stage's destination bit. If the two inputs want different outputs, both paths can advance in the same cycle. If both inputs want the same output, the switch uses the same stored-state round-robin arbiter to grant one input while the other input remains not-ready and is retried through valid/ready backpressure.

3.9 Implementation Status

At the current stage, the processor is a functional RTL prototype of a multi-slice, Sharing-inspired RISC-V core. The implementation supports configurable slice count and issue width, per-slice rename, scoreboard, PRF, ROB, and functional-unit structures, cross-slice operand communication, conservative branch recovery, and a correctness-first shared memory model. Bare-metal RISC-V tests and Embench-IoT workloads can be loaded through the hex-based instruction/data memory harness and evaluated using cycle-level IPC and bottleneck counters. For the present thesis experiments, the design should be viewed as a bare-metal RV32I-oriented core with partial RV64I support, intended to evolve toward a minimal RV64 Linux-capable target.

The scalar operand network has been refactored into an opcode-tagged packet interface with direct and butterfly routing models. Remote operands now use explicit request/reply traffic rather than all-slice writeback broadcast, and logical-register generations plus rename-time dependency holds protect against stale reused-register matches. The implementation is not yet a complete Linux-capable, fully speculative, or fully ISA-compliant target. Instead, it should be understood as an experimental platform for quantifying how slice count, issue width, communication topology, and global synchronization affect a Sharing-style microarchitecture.

Chapter 4

Experimental Methodology

4.1 Workloads

The evaluation uses bare-metal workloads that can run in the current RTL simulation harness without operating-system support. This choice matches the present implementation status: the processor can fetch, decode, execute, retire, and report `tohost` results for integer bare-metal programs, but it is not yet a privileged, interrupt-capable, Linux-capable system.

The first workload class is `riscv-tests`. The main correctness and IPC dataset uses the RV32UI integer user-mode tests, excluding `fence_i` because the current test memory model uses separate instruction and data memory images. These tests are short, but they are useful because they isolate integer, branch, load/store, and jump behavior while producing a simple pass/fail result. The repository also contains selected RV64UI bring-up runs and a small RV64I self-check, but the thesis results should treat RV32I as the primary measured ISA target.

The second workload class is Embench-IoT [10]. Embench is useful because it exercises longer control-flow and memory-access behavior than the ISA tests while still fitting a free-standing embedded execution environment. The current repository contains two Embench

datasets. The direct-topology issue-width sweep is a smoke subset containing `aha-mont64`, `crc32`, and `md5sum` at local scale factor 1. A separate butterfly-topology dataset attempts the full local-scale-factor-1 Embench set: `aha-mont64`, `crc32`, `depthconv`, `edn`, `huffbench`, `matmult-int`, `md5sum`, `nettle-aes`, `nettle-sha256`, `nsichneu`, `picojpeg`, `qrduino`, `sglib-combined`, `slre`, `statemate`, `tarfind`, `ud`, `wikisort`, and `xgboost`.

Embench uses scale factors to control benchmark runtime. Each benchmark defines a `LOCAL_SCALE_FACTOR` that is passed into its benchmark body, usually as the number of outer-loop repetitions, and the harness also provides a `GLOBAL_SCALE_FACTOR`. In the standard Embench flow, local scale factors are chosen to make different benchmarks run for comparable amounts of time. In this thesis harness, `EMBENCH_LOCAL_SCALE_FACTOR_OVERRIDE` copies the benchmark source into a generated build directory and rewrites that local factor before compilation. Setting the local factor to 1 is therefore a simulation shortcut: it keeps the benchmark code path and correctness check, but reduces the amount of repeated work so long RTL simulations can finish. Because a larger local scale factor primarily runs the same benchmark kernel for more outer-loop repetitions, increasing it mostly lengthens simulation time rather than adding qualitatively different program behavior. All Embench experiments reported below therefore use local scale factor 1.

4.2 Configurations

Table 4.1 summarizes the configuration sets currently available in the repository. The most complete dataset is the direct-topology RV32UI sweep, which covers 40 tests across four slice counts and four issue widths. The other datasets are useful for preliminary topology and Embench analysis, but they should not be overinterpreted as final full-factorial measurements.

Table 4.1: Collected configuration datasets used in the evaluation.

Dataset	Workloads	NUM_SLICES	NUM_WAYS	Topology
RV32UI direct matrix	40 RV32UI tests	1, 2, 4, 8	1, 2, 4, 8	direct
Embench direct smoke	3 Embench workloads	1, 2, 4	1, 2, 4, 8	direct
RV32UI butterfly smoke	add, beq, lw, sw	1, 2, 4, 8	2	butterfly
Embench butterfly request/reply fresh	19 Embench workloads	1, 2, 4, 8	2	butterfly

The full Embench butterfly sweep fixes `NUM_WAYS` at 2. This choice keeps the non-ideal-network evaluation focused on slice scaling rather than multiplying the already long Embench RTL simulations by another issue-width dimension. It also reflects the preliminary direct-topology sweeps: moving from two-way to four-way issue does not produce a consistent IPC improvement. In the RV32UI direct matrix, four-way issue is only slightly higher than two-way issue at one slice and is lower at two, four, and eight slices. In the direct Embench smoke set, four-way issue helps at one and two slices but is lower than two-way issue at four slices. Two-way issue is therefore a useful compromise point: it still exercises a superscalar slice, but leaves the full butterfly experiment small enough to run across the full Embench workload set and all slice counts.

All datasets keep `RENAME_NET_TOPOLOGY` on the direct path. This is intentional: rename is already synchronized across slices and is expected to dominate CPI if it is placed behind a contended packet network. The scalar network experiments therefore focus on operand request/reply, deallocation, and branch/control traffic. Packetized experiments set `SCALAR_NOC_TOPOLOGY` to select the scalar message transport. `WB_BCAST_TOPOLOGY` is still accepted as a legacy build alias, but it no longer denotes a separate cross-slice writeback broadcast fabric.

4.3 Build and Run Flow

The full-design test flow is driven by `tests/unit/full_design/Makefile`. RISC-V assembly tests are preprocessed with the local test header, linked into a freestanding ELF, and converted into separate instruction-memory and data-memory hex files. The local test header and linker script replace the default `riscv-tests` environment because the upstream test macros assume privileged ISA support for trap handling and test termination. The current core does not yet implement those privileged mechanisms, so the local header defines a simpler bare-metal pass/fail protocol: `RVTEST_PASS` and `RVTEST_FAIL` write a result word to a fixed `tohost` address near the end of the data memory. The cocotb harness monitors that word to determine whether the program completed correctly. Embench uses local board support under `tests/unit/full_design/embench_env`, links into the same memory map, and reports completion through the same `tohost` word in data memory.

Example RISC-V test command:

```
make -C tests/unit/full_design -s run-riscv-hex \
  RISCV_TEST=rv32ui/add \
  NUM_SLICES=2 NUM_WAYS=2 \
  SCALAR_NOC_TOPOLOGY=2 RENAME_NET_TOPOLOGY=0 \
  SIM=verilator TRACE=0
```

Example Embench command:

```
make -C tests/unit/full_design -s run-embench-hex \
  EMBENCH_BENCH=crc32 \
  EMBENCH_LOCAL_SCALE_FACTOR_OVERRIDE=1 \
  EMBENCH_TIMEOUT_CYCLES=2000000 \
  EMBENCH_DMEM_ADDR_WIDTH=16 EMBENCH_DMEM_SIZE=65536 \
  NUM_SLICES=2 NUM_WAYS=2 \
  SCALAR_NOC_TOPOLOGY=2 RENAME_NET_TOPOLOGY=0 \
  SIM=verilator TRACE=0
```

For larger sweeps, the repository uses three helper scripts in the full-design tools directory:

- `run_design_matrix.py`,
- `run_riscv_matrix.py`,
- `run_embench_suite.py`.

These scripts instantiate Verilator builds for the requested `NUM_SLICES`, `NUM_WAYS`, topology, and workload combinations, write one raw metric CSV per run, and append a combined CSV for aggregation.

4.4 Performance Counters

The harness records cycle count, retired instruction count, and IPC for every run. It also records operation mix and microarchitectural counters that explain scaling behavior rather than merely reporting final IPC:

- retired ALU, load/store, and branch counts,
- branch resolved, taken, and redirect counts,
- frontend and window-wait stalls,
- slice FSM occupancy,
- issue stall attribution,
- functional-unit read/operand/execute/writeback counters,
- ROB occupancy and head-blocked counters,
- remote operand request/response and deallocation sent/received counters.

The summary script `summarize_design_matrix.py` averages passing rows by `NUM_SLICES` and `NUM_WAYS`. The bottleneck script groups slice state counters into

frontend wait, decode, rename-network, issue, and execute categories. These state groups are exclusive within the slice FSM. Event counters such as operand wait, remote operand wait, ROB head blocking, and branch stalls are non-exclusive; they are reported as rates per retired instruction or per cycle.

4.5 Correctness Tests

The performance data is only meaningful if the configuration can complete the test and produce the expected `tohost` value. The main correctness source is the RV32UI matrix itself: every row in the direct RV32UI matrix passed. In addition, the repository contains directed tests for behaviors that are easy to hide in average IPC numbers:

- reset and frontend bring-up,
- RAT rename, local PRF allocation, and program-order-aware source rename,
- cross-slice RAT visibility,
- scoreboard inter-instruction and intra-window RAW/WAW behavior,
- explicit cross-slice operand request/reply behavior,
- reused logical-register stale-wakeup rejection,
- unmapped remote source handling,
- non-speculative branch taken and not-taken redirects,
- cross-slice load-before-store ordering through the global LSU arbiter.

The cross-slice memory ordering test is especially important because it exposed a real correctness problem in the earlier direct data-memory model. The current global age arbiter passes the checked-in butterfly configuration of this test.

4.6 Threats to Validity

The reported IPC values are RTL simulation results from Verilator, not frequency-normalized silicon performance. The memory system is a simplified test harness rather than a complete cache hierarchy, and the current LSU ordering mechanism is intentionally conservative. Rename remains on a direct specialized path, final frontend redirect is still direct top-level control, and the backend is not fully speculative. The slice window control path is also a single sequential FSM rather than a pipelined frontend; as explained in Section 3.3, a pipelined variant exists but is not yet verified well enough to produce trustworthy numbers, so all reported results deliberately use the slower sequential control path. These choices make the results useful for studying synchronization and communication bottlenecks, but they should not be interpreted as final processor performance.

The Embench data is also sensitive to timeout and scale-factor settings. The reported request/reply scalar-network runs use local scale factor 1 and a larger cycle timeout so that long-running rows can complete rather than being misclassified as failures. Results should still be read together with the exact timeout and benchmark list used for the sweep, especially when comparing against older runs that used the writeback-broadcast prototype.

Chapter 5

Evaluation

5.1 Correctness Summary

Table 5.1 summarizes the simulation runs used for the evaluation. The direct RV32UI matrix remains the broadest ISA-smoke dataset: all 640 runs pass, corresponding to 40 RV32UI tests evaluated across 16 direct-topology architectural configurations. The earlier writeback-broadcast butterfly runs exposed timeout and stale-value failures; the request/reply implementation removes those failures in the checked rows. The butterfly Embench dataset combines an 18-workload sweep with separately run `nettle-sha256` dependency-hold checks and passes all 76 butterfly rows.

Table 5.1: Outcome summary for checked-in evaluation datasets.

Dataset	Total rows	Pass	Running	Timeout	Fail
RV32UI direct matrix	640	640	0	0	0
Embench direct smoke	36	36	0	0	0
RV32UI butterfly smoke	16	16	0	0	0
Focused <code>nettle-sha256</code> direct/butterfly	8	8	0	0	0
Combined Embench butterfly request/reply dataset	76	76	0	0	0

Figure 5.1 summarizes the direct-topology RV32UI IPC matrix used as the broadest ISA smoke test. Each cell averages the same 40 passing RV32UI tests, and darker cells

indicate higher IPC. The heatmap makes the issue-width sensitivity visible: the best average IPC occurs at eight slices and one issue way, while wider issue stops helping once frontend, rename, and scoreboard admission overheads dominate.

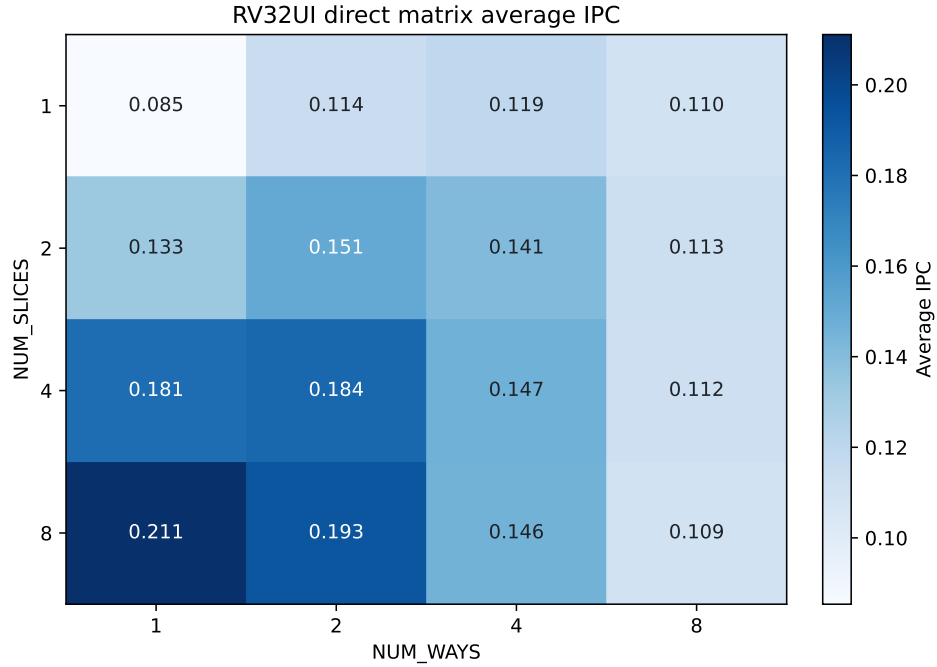


Figure 5.1: Average IPC heatmap for the direct-topology RV32UI matrix. Darker cells indicate higher IPC.

The rest of this chapter separates primary performance data from supplemental sensitivity checks. The primary performance data is the full Embench butterfly dataset, because it exercises the packetized, non-ideal scalar network. Direct topology approximates an ideal all-to-all scalar network, so direct-topology IPC is used only to isolate issue-width behavior and to provide a reference point for what happens when scalar communication is made less expensive. Those direct numbers are not the main evaluation target.

5.2 Full Butterfly Embench Scaling

Each slice count in the combined Embench butterfly dataset contains 19 workload rows, all of which pass. The `nettle-sha256` row comes from a separate dependency-hold matrix used to validate the request/reply fix, while the other 18 workloads come from the main butterfly sweep. Figure 5.2 reports per-workload IPC for the completed set. The full-set averages are 0.121, 0.121, 0.112, and 0.101 IPC for one, two, four, and eight slices respectively.

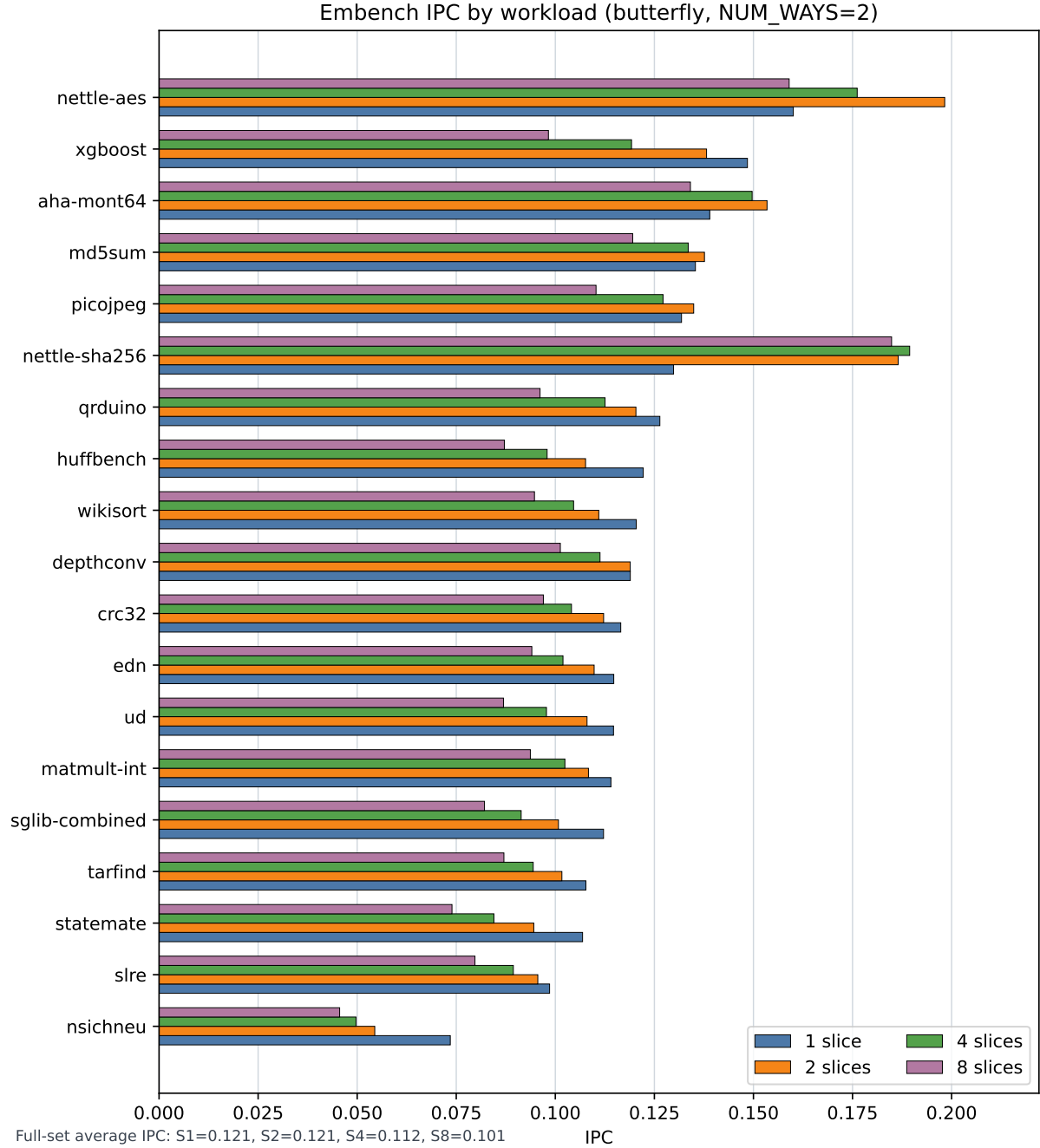


Figure 5.2: Per-workload IPC for the completed Embench butterfly dataset, with NUM_WAYS=2 and local scale factor 1.

Table 5.2 summarizes the same completed rows as a per-workload run-size check. Each row averages the one-, two-, four-, and eight-slice butterfly runs for that workload. Retired instruction counts are nearly fixed for a given workload because each slice count

runs the same benchmark binary at the same local scale factor, while cycle counts differ because NoC contention, branch flush alignment, ROB-head blocking, and remote operand timing change the exact execution schedule. The table should therefore be read as a simulation-length summary rather than as a claim that each configuration executes in exactly the same number of cycles.

Table 5.2: Per-workload average run size for the completed Embench butterfly dataset, with NUM_WAYS=2 and local scale factor 1.

Workload	Rows	Avg. retired instr.	Avg. cycles	Max cycles
xgboost	4	3,251,487	26,447,854	33,089,878
wikisort	4	1,408,622	13,181,220	14,868,183
qrduno	4	1,145,764	10,170,854	11,916,837
edn	4	869,447	8,316,834	9,242,466
picojpeg	4	984,528	7,858,171	8,927,733
matmult-int	4	765,714	7,357,297	8,172,011
tarfind	4	285,110	2,936,747	3,276,074
huffbench	4	276,602	2,708,104	3,174,276
crc32	4	210,023	1,964,511	2,165,471
sglib-combined	4	123,893	1,299,544	1,508,566
depthconv	4	62,811	560,274	620,253
md5sum	4	53,543	408,372	447,943
nettle-aes	4	66,314	385,552	417,044
slre	4	28,282	313,486	354,753
aha-mont64	4	26,842	186,899	200,173
nettle-sha256	4	9,407	55,858	72,450
ud	4	5,505	54,651	63,335
nsichneu	4	1,782	32,980	39,101
statemate	4	1,763	19,960	23,849

On this completed set, average IPC is roughly flat from one to two slices and then decreases at four and eight slices. The **nettle-sha256** row remains an important positive scaling case, but the full workload mix is dominated by global frontend/rename coordination, remote operand traffic, and ROB head blocking before the extra slice resources can pay for themselves.

5.3 Bottleneck Breakdown

Table 5.3 uses the same 19 passing Embench butterfly workloads as Figures 5.2 and Table 5.2. State percentages are exclusive slice FSM residency; event rates are cycles per retired instruction.

Table 5.3: Average bottleneck summary for the completed Embench butterfly dataset, with NUM_WAYS=2.

NUM_SLICES	Rows	IPC	FE	Rename	Issue	Remote wait	ROB block
1	19	0.121	37.9%	40.1%	8.6%	0.00	2.28
2	19	0.121	55.7%	24.8%	11.2%	4.70	6.18
4	19	0.112	65.5%	18.1%	10.3%	8.10	8.74
8	19	0.101	72.4%	14.8%	7.9%	12.45	12.10

The bottleneck counters explain why the current butterfly IPC curve does not scale on the larger completed set. The per-retired-instruction remote operand wait rate grows from 0.00 cycles at one slice to 12.45 cycles at eight slices. ROB head blocking also grows steadily, and the slice FSM spends an increasing fraction of its time waiting on the frontend path. These measurements match the qualitative expectation from Chapter 3: the current implementation exposes useful slice-level parallelism, but its global synchronization points and demand-driven cross-slice operand traffic grow quickly.

The rename percentage decreases as slice count rises, but this should not be read as rename becoming free. The percentage is relative to the measured slice FSM state total; other costs, especially frontend wait and remote operand wait, grow at the same time. Rename remains a direct, synchronized path and is still part of the fixed per-window cost paid before useful execution can proceed.

5.4 Supplemental Direct-Topology Sensitivity

The following direct-topology results are supplemental. They use a smaller Embench smoke subset and an idealized all-to-all scalar network. These rows predate the final

request/reply butterfly refresh, so the absolute IPC values should not be compared directly against Figure 5.2 without rerunning the same workloads on both topologies. Their role here is to show how the implementation responds when scalar communication is made less expensive than in the butterfly network.

Table 5.4 averages `aha-mont64`, `crc32`, and `md5sum` at local scale factor 1. For this small set, increasing issue width helps up to two or four ways, but the eight-way configuration declines despite exposing more nominal issue slots. The best smoke result is 0.299 IPC at four slices and two ways.

Table 5.4: Average IPC for the direct-topology Embench smoke matrix. Each populated cell averages three passing workloads.

NUM_SLICES	NUM_WAYS=1	NUM_WAYS=2	NUM_WAYS=4	NUM_WAYS=8
1	0.105	0.172	0.208	0.208
2	0.174	0.246	0.257	0.228
4	0.253	0.299	0.282	0.234

The important point is that `NUM_WAYS` is not a pure functional-unit count in this implementation. Increasing it also increases the amount of work that must move through the synchronized frontend, decode, rename, and scoreboard admission path. When those shared mechanisms dominate, extra ways can add pressure without retiring proportionally more instructions.

A second supplemental comparison keeps the workload subset fixed and changes only the scalar-network model. Table 5.5 is retained as a legacy smoke comparison between direct and butterfly routing on the common passing Embench smoke subset: `aha-mont64`, `crc32`, and `md5sum`, all at two-way issue. The one-slice result is identical because no cross-slice network is exercised. The eight-slice point is omitted because the checked-in direct Embench smoke matrix stops at four slices. A clean request/reply direct-versus-butterfly comparison would require rerunning the same workload set on both topologies.

Table 5.5: Legacy direct versus butterfly average IPC on the common passing Embench smoke subset, with NUM_WAYS=2.

NUM_SLICES	Direct IPC	Butterfly IPC	Relative change
1	0.172	0.172	0.0%
2	0.246	0.235	-4.3%
4	0.299	0.258	-13.6%

This result is expected for the current design. Direct routing approximates an idealized all-to-all fabric, while the butterfly network introduces staged routing, arbitration, and backpressure. The result is not a claim that butterfly is a poor topology choice; it shows that the present core still depends heavily on timely cross-slice operand and control communication, so a modeled network cost quickly becomes visible in IPC.

5.5 Memory Ordering Impact

The conservative LSU arbiter was added for correctness. The directed cross-slice load-before-store test passes in the checked-in two-slice, one-way butterfly configuration, retiring 8 instructions in 105 cycles. This test matters because the previous direct shared-memory behavior could allow a younger store from one slice to affect an older load from another slice in the same cycle.

No committed dataset currently isolates the conservative global age arbiter against an idealized or speculative LSQ-like memory system. Accordingly, this section makes only a limited claim: the global LSU arbiter fixes a real ordering bug and is visible as a source of serialization in memory-heavy programs, but the exact IPC cost of that choice needs a controlled before/after experiment.

Chapter 6

Related Work

6.1 Scalable Out-of-Order Architectures

Conventional superscalar out-of-order cores improve single-thread performance by exposing instruction-level parallelism through dynamic scheduling, register renaming, speculation, and bypassing [7]. These mechanisms are effective, but they become increasingly difficult to scale as issue width and instruction window size grow. Wakeup/select logic, bypass networks, physical-register access, load/store ordering, and branch recovery all place pressure on centralized structures. The result is the central tension that motivates this thesis: a fixed monolithic core can be too large for narrow or latency-bound phases, while still being too rigid to efficiently exploit wide phases when communication and synchronization dominate.

The closest prior work is the Sharing Architecture, which proposes composing multiple simple out-of-order slices into configurable virtual processors for IaaS clouds [1, 2]. Sharing directly targets workload diversity by allowing a core allocation to change at sub-core granularity rather than assigning each application to a fixed monolithic core. Its evaluation focuses on the architectural and resource-provisioning benefits of configurable slice groups. This thesis follows that architectural direction but studies a different point

in the design space: a RISC-V RTL implementation that exposes the concrete costs of rename coordination, scalar operand communication, branch/control handling, and conservative memory ordering in a working prototype.

Clustered out-of-order processors attack a related scalability problem by partitioning execution resources into smaller clusters [12]. Clustering reduces the cost of fully centralized issue and bypass structures, but it introduces a communication-placement trade-off: instructions must be placed near the values and functional units they need, or inter-cluster operand movement can erase the benefit of the partition. The slice organization in this thesis has a similar communication/locality concern, but the goal is not only to make a single wide core more scalable. The larger goal is to make the amount of core-like hardware assigned to a thread configurable while preserving enough out-of-order execution capability inside each slice.

Core Fusion also responds to software diversity by allowing multiple simpler cores to cooperate as a larger core when a program can benefit from additional single-thread resources [13]. This makes it an important point of comparison for configurable architectures. Core Fusion begins from a chip multiprocessor and adds mechanisms that let cores fuse for single-thread execution. By contrast, Sharing and the implementation in this thesis take a sub-core view: the basic unit is a slice containing the local machinery needed to participate in a shared instruction stream. That distinction matters because it changes where operand communication, rename visibility, and commit ordering must be handled.

TRIPS and WiDGET explore still different ways to distribute execution across many tiled or decoupled resources [14, 15]. These designs emphasize spatial execution and distributed communication rather than simply widening a conventional issue queue. They are relevant because they show that scalable instruction execution often requires exposing placement, communication, and synchronization as first-class design issues. The present thesis is narrower: it keeps a conventional RISC-V instruction stream and focuses on how a Sharing-style slice organization behaves when implemented as a cycle-level RTL core

with measurable scalar-network and synchronization costs.

6.2 Operand Networks and On-Chip Interconnect

Once execution resources are partitioned, scalar operand movement becomes an architectural problem rather than an incidental bypass-wire detail. A centralized bypass network or all-to-all crossbar can provide low-latency communication for a small number of producers and consumers, but its cost grows quickly as the number of execution sites increases. Distributed designs therefore need an explicit interconnect policy: which values are sent, where they are sent, how conflicts are arbitrated, and how much latency the scheduler or execution model can tolerate.

Scalar operand networks were proposed for this setting as an on-chip interconnect for ILP in partitioned architectures [3]. The key idea is to make short scalar values and dependence wakeups travel over a structured communication substrate rather than assuming a single global bypass path. Related tiled architectures such as Raw also treat on-chip communication as a first-order part of the machine organization rather than a hidden implementation detail [4]. These systems show that scalable execution is not only a question of adding functional units; it also depends on whether the operand network can deliver the right values at the right time.

The Sharing Architecture adopts scalar operand communication between slices [1]. A slice can execute part of an instruction stream while depending on values produced by other slices, so operand request, reply, and wakeup behavior must cross slice boundaries. This is closely aligned with the present thesis, but the implementation here separates two effects that are often collapsed together: the architectural value of slice-level sharing and the cycle-level cost of the interconnect used to realize that sharing.

Direct broadcast is useful as an idealized reference because it approximates an all-to-all scalar communication fabric. It avoids staged routing delay and contention in the model,

but it is not a scalable physical assumption. More structured networks, including meshes, trees, and butterflies, reduce wiring and fanout pressure by routing packets through intermediate stages. The cost is that packets now experience routing latency, arbitration, and backpressure. The butterfly scalar NoC in this thesis is used to expose those effects: if two packets need different outputs at a switch, both can advance, but packets competing for the same output serialize through round-robin arbitration.

The current RTL prototype packages operand requests, operand responses, deallocation, and branch/control events as opcode-tagged scalar messages with destination information. This is not yet a complete slice-native control system, since some frontend and rename coordination remains direct. It is, however, enough to measure how packetized scalar communication changes IPC, remote operand waiting, and ROB blocking relative to the direct topology. In that sense, the scalar NoC is both an implementation mechanism and an evaluation instrument for understanding where Sharing-style flexibility starts to incur communication overhead.

6.3 RISC-V Research Platforms

RISC-V has become a practical ISA target for architecture research because it is open, modular, and supported by a broad software and tool ecosystem. For this thesis, RISC-V provides a stable instruction format and a clear path from small bare-metal tests toward larger software targets [8]. The current implementation deliberately starts from a narrow point in that space: a bare-metal RV32I-oriented core with partial RV64I support. That scope makes it possible to focus on slice organization, rename behavior, scalar messages, and memory ordering before taking on the additional machinery required for a Linux-capable system.

Many RISC-V research platforms are valuable because they provide complete cores, SoCs, cache hierarchies, and tool flows that can run substantial software. The goal

of this thesis is different. It does not attempt to present a polished general-purpose RISC-V core or a competitive application processor. Instead, it uses RISC-V as the instruction-set substrate for studying a Sharing-style microarchitecture. The important artifact is therefore not only a core that executes instructions correctly, but also an implementation that exposes the coordination points that matter for configurable slices: global and local rename, demand-driven remote operand wakeup, cross-slice deallocation, conservative memory ordering, and topology-dependent scalar communication.

The project also benefits from reusable open-source RISC-V infrastructure. The HPDcache work provides an example of a high-performance open L1 cache design for RISC-V cores [9]. In this thesis repository, HPDcache is adapted through a wrapper so it can be integrated with the broader SoC direction, even though the main evaluation still uses a simplified memory harness. This distinction is important: the prototype is moving toward a more realistic memory hierarchy, but the results in Chapter 5 should be read as measurements of the current bare-metal research core, not of a final cache-backed system.

Benchmarking follows the same staged approach. Directed tests and RISC-V ISA tests establish that the core can execute expected instruction sequences and report pass/fail status through the custom `tohost` mechanism. Embench-IoT then provides a compact embedded-workload suite that is large enough to expose instruction mix, branch behavior, memory behavior, and communication bottlenecks without requiring an operating system [10]. The thesis adds matrix runners and counter aggregation around these workloads so that each result can be compared across `NUM_SLICES`, `NUM_WAYS`, and scalar network topology. In this way, the RISC-V platform is not only a correctness target; it is the measurement framework for isolating where a Sharing-inspired implementation gains or loses performance.

Chapter 7

Discussion and Future Work

7.1 What the Current Design Shows

The main result of the prototype is that a Sharing-style core can be made concrete in a small RISC-V RTL implementation. The design executes direct-topology RV32UI matrices, butterfly-topology smoke tests, and a completed Embench butterfly dataset through a real rename, scoreboard, ROB, functional unit, memory, and scalar-message path. This matters because many of the hardest issues in a composed single-thread core only appear once dependencies, commit, operand transport, and branch recovery interact cycle by cycle. The final operand request/reply protocol, dependency hold mechanism, and generation tags show that the stale-value correctness problem can be solved without returning to all-slice writeback broadcast as a coherence mechanism.

The performance result is workload-dependent rather than uniformly positive or negative. Adding slices does increase the number of local execution resources, and several workloads show their best butterfly IPC at more than one slice, including `nettle-sha256`, `nettle-aes`, `picojpeg`, `md5sum`, and `aha-mont64`. This supports the basic motivation for a Sharing-style design: some instruction streams can use more than one slice, at least up to a point. However, the full Embench average is roughly flat from one to two slices and

then falls at four and eight slices, so the extra resources are not automatically converted into throughput.

The bottleneck counters clarify why. Table 5.3 shows that the issue-state fraction remains modest, between 7.9% and 11.2% across the butterfly sweep. Even without a deeper issue queue to decouple frontend delivery from backend execution, the machine is not primarily stalled in issue waiting for compute resources. The much larger effect is frontend/global-control waiting: frontend residency rises to 72.4% at eight slices, while remote operand wait and ROB-head blocking also grow with slice count. In other words, the limiting factor is not that additional compute units are unavailable; it is that the current control, rename, operand, and commit paths cannot feed those units often enough.

This suggests that one of the central difficulties in this style of composable single-stream architecture is preserving a shared program-order view at the frontend and rename boundary without forcing all slices into an overly rigid rendezvous. In this prototype, that boundary is implemented with lock-step window admission. Under butterfly topology, unequal completion times across slices become especially costly: slices that finish their current window early wait for lagging slices before the next window can be accepted. This completion mismatch appears to be a larger bottleneck than backend issue availability in the measured eight-slice runs.

Some of this behavior is inherent to the model being studied. Remote operands are real communication events, and the butterfly network intentionally exposes distance and contention. A larger virtual core also has more points where a single unresolved dependency, branch, or retirement constraint can block useful work elsewhere. Other effects are artifacts of the current thesis core: the frontend and rename paths remain globally synchronized, branch recovery is conservative, and memory ordering is more serialized than a production load/store queue would be.

The lesson is therefore that slice composition is useful, but only when the workload has enough exploitable parallelism and the scalar/control boundary is efficient enough to

expose it. Correct operand transport and commit-safe dependency tracking make the architecture measurable; the next performance limit is reducing the frontend, rename, branch, remote-request, and memory ordering costs that prevent the added slices from staying busy.

7.2 Improving the Scalar Message Boundary

The slice-side SON adapter now handles more of the scalar-message boundary. Operand request/reply, deallocation, and branch/control traffic share the opcode-tagged packet path. The remaining work is to reduce the semantic fanout still visible in `top.sv`, move final branch redirect handling onto the packetized path, and improve request-queue fairness and capacity so the demand-driven operand protocol is not limited by small local buffers.

The most important next step is to make scalar traffic robust under contention. The current design is sufficient for the evaluated workloads, but a production version should use explicit request queues, credit or backpressure rules, and fair arbitration so that operand replies, deallocation, and control messages do not starve one another. This would also make it easier to study scalar-network topology as an architectural parameter rather than as a correctness risk.

7.3 Rename Path

Rename is kept direct in the thesis-core design because the current lock-step rename protocol would likely make a packetized rename network dominate CPI. Future work can study a more realistic rename transport together with a rename protocol designed to tolerate network latency and contention.

A useful intermediate step is to keep the same rename semantics but remove the assumption that every rename message arrives in the same cycle. That would require

buffering rename windows, carrying dependency metadata with the rename messages, and defining how much skew between slices is allowed before the frontend must stop admitting new work.

7.4 Frontend Decoupling and Instruction Stream Ordering

The current frontend and rename boundary is intentionally conservative: slices advance through fetched windows with a shared global notion of program order. This is broadly consistent with the Sharing Architecture model because a VCore executes one instruction stream, and cross-slice rename, operand, branch, and commit state must agree on which instructions are older or younger.

The high frontend residency in Chapter 5 does not by itself imply that future designs should let slices fetch arbitrary parts of the instruction stream independently. Fully out-of-order slice fetch would require additional sequence tags, speculation epochs, redirect acknowledgments, replay or discard rules, and careful merge logic so that branch recovery and precise state remain well-defined. Those mechanisms may be worth studying, but they are substantially more complex than simply moving the current lock-step interface onto a different network.

A nearer-term direction is a latency-tolerant but still ordered frontend protocol. For example, future work could add small decoupling queues after fetch/decode or rename, allow limited skew between slices while preserving a global window order, and acknowledge redirects explicitly before new speculative windows are admitted. This would keep the single-stream Sharing semantics while reducing the number of backend stalls that immediately bubble back into the frontend residency counter.

Preliminary work in this direction already exists. The register-staged pipelined slice variant described in Section 3.3 (Figure 3.3) replaces the sequential window FSM

with overlapped pipeline stages, adds a fetch-time branch target buffer, and renames speculatively with misprediction recovery. It passes the full RV32UI `riscv-tests` suite and reaches roughly 1.9 IPC on a warm-cache, branch-prediction-friendly workload in the two-slice, two-way configuration, which suggests that decoupling the frontend and rename paths has substantial headroom over the sequential control path used for the main evaluation. Making that variant correct across the Embench workloads at higher branch-speculation depths is the main obstacle to adopting it, and is the most direct continuation of this work.

7.5 Branch Recovery

Branch recovery is still one of the most conservative parts of the prototype. The current design uses top-level control to select redirects and to hold new frontend progress while control flow is unresolved. This keeps correctness simple, but it also contributes directly to the frontend residency seen at high slice counts.

Future work should move branch recovery closer to the scalar message boundary. Branch resolution, redirect, flush, and acknowledgment messages should be packetized so that slices can locally discard younger work while still agreeing on the global recovery point. A complete design also needs speculation epochs or redirect sequence numbers so that stale branch messages cannot affect work from a newer fetch stream.

7.6 Memory System

The current memory-ordering model is conservative and corrects known cross-slice ordering bugs, but it is not a realistic out-of-order load/store queue. Future work should add a commit-safe store buffer or store queue, reduce unnecessary serialization, and preserve precise behavior.

The key goal is to separate correctness ordering from unnecessary execution blocking. Loads should be able to pass older independent operations when it is safe, stores should become visible only at the correct commit point, and memory violations should trigger a precise replay or flush policy. Without this, memory ordering remains another source of slice completion mismatch.

7.7 Full-System Support

Linux boot remains a stretch goal. It likely requires privileged CSRs, exceptions/traps, timer and interrupt support, atomics or kernel configuration workarounds, a memory map, a device model, and possibly MMU or nommu/Linux support.

The nearer-term full-system milestone is not a complete operating-system boot, but a cleaner bare-metal platform boundary: a documented memory map, stable trap entry behavior, timer support, and a less ad hoc test termination path. Those pieces would make the evaluation flow more portable while leaving full Linux support as a later integration target.

Chapter 8

Conclusion

As general-purpose computing has evolved, workloads have become increasingly diverse in their behavior and resource demands. Fixed monolithic cores are therefore designed around a chosen compromise point in the microarchitectural design space, aiming to provide acceptable performance across common and expected uses. This compromise limits both utilization and performance, because a static core organization cannot match the changing needs of every program phase. Prior research in composable and configurable architectures has addressed this problem by allowing execution resources to be combined or partitioned more flexibly [13, 1, 2].

This thesis studies the Sharing Architecture [1] by implementing a RISC-V RTL prototype of a slice-based SoC. Each slice contains a superscalar, scoreboard-based out-of-order execution pipeline, and multiple slices can be composed to execute a shared instruction stream. The prototype turns the abstract idea of composable execution resources into concrete hardware mechanisms, including cross-slice rename coordination, scalar operand request and response messages, branch/control communication, conservative memory ordering, and a configurable scalar NoC that packages requests and responses for studying interconnect delay. Through this implementation, the thesis studies both the potential benefit of adding slices and the coordination costs that arise when those slices must

behave as one logical core.

The evaluation shows that this design point is functional, but that useful composition depends strongly on workload behavior and synchronization overhead. The prototype passes the selected RV32UI and Embench experiments, including the full butterfly Embench sweep, demonstrating that the slice, rename, memory, and scalar-message mechanisms can execute real bare-metal programs correctly. The performance results are more nuanced: several workloads benefit from additional slices, but IPC does not improve monotonically on average. Interestingly, the dominant bottleneck is not simply the added latency of scalar operand transport. Even with an idealized frontend/rename path and a more realistic packetized NoC for scalar messages, the largest limitation is frontend synchronization, along with ROB blocking, remote operand waiting, and conservative branch and memory handling.

Through implementation-based experimentation, this thesis concludes that Sharing-style configurability is a viable direction, but it cannot be achieved by adding slices alone. A scalable implementation must make the common coordination paths as slice-native as the execution resources: frontend admission, rename exchange, branch recovery, operand transport, and memory ordering must tolerate latency and contention without forcing all slices to wait in lock step. The prototype developed in this thesis provides a working basis for that next step, showing both how a RISC-V Sharing implementation can be built and where future designs must reduce synchronization to turn configurability into sustained performance.

Appendix A

Useful Commands

A.1 RISC-V Matrix Runner

```
python3 tests/unit/full_design/tools/run_riscv_matrix.py \  
  --suite rv32ui \  
  --slices 1 2 4 8 \  
  --ways 1 2 4 8 \  
  --xlen 32 \  
  --output-dir tests/unit/full_design/metrics/rv32ui-matrix \  
  --no-plot
```

A.2 Directed Memory-Ordering Regression

```
make -C tests/unit/full_design -s \  
  TESTCASE=test_cross_slice_load_before_store_order \  
  NUM_SLICES=2 NUM_WAYS=1 \  
  SCALAR_NOC_TOPOLOGY=2 RENAME_NET_TOPOLOGY=0 \  
  SIM=verilator TRACE=0
```

Appendix B

Counter Schema

The benchmark harness writes one raw metrics CSV row per completed simulation. Table B.1 groups the most important columns. Aggregated `perf_*` fields are sums across slices unless otherwise noted. Columns with the `slice_` prefix contain the corresponding per-slice values in slice order, separated by vertical bars.

Table B.1: Raw metrics CSV schema.

Column group	Meaning
<code>test, num_slices, num_ways</code> <code>cycles, retired, ipc</code>	Benchmark name and architectural configuration. Total cycles, retired instructions, and retired instructions per cycle.
<code>tohost, result</code>	Bare-metal termination word and pass/fail result decoded by the harness.
<code>retired_alu, retired_ldst,</code> <code>retired_branch</code>	Retired instruction mix grouped into ALU, load/store, and branch/control classes.
<code>branch_resolved,</code> <code>branch_taken,</code> <code>branch_redirect</code>	Branch/control events observed by the top-level control path.
<code>frontend_stall_cycles</code>	Explicit unresolved-control frontend stalls; distinct from <code>perf_state_resp_fe</code> , which is slice FSM residency.
<code>perf_window_*</code>	Global window admission stalls, including cycles where not all slices are simultaneously waiting and ready.
<code>perf_branch_*</code>	Cycles spent waiting on predicted or unpredicted branch/-control recovery conditions.
<code>perf_state_*</code>	Slice FSM residency for frontend response, decode, rename, rename-exchange states, RAT remap, local rename, scoreboard check, and execute. State names do not always imply current semantic broadcast behavior.
<code>perf_issue_*</code>	Scoreboard/ROB admission activity and stalls: ROB capacity, memory order, structural, WAW, and same-window prefix restrictions.
<code>perf_fu_*</code>	Functional-unit residency and stalls: operand wait, remote operand wait, execute, writeback, and writeback backpressure.
<code>perf_rob_*</code>	ROB nonempty/full cycles, head-blocked cycles, and accumulated occupancy.
<code>perf_remote_operand_*</code>	Scalar operand-network request/response counts, separated by sent and received messages.
<code>perf_remote_dealloc_*</code>	Cross-slice deallocation message counts, separated by sent and received.
<code>slice_retired, slice_alu,</code> <code>slice_ldst, slice_branch</code> <code>slice_perf_*</code>	Per-slice retired instruction and instruction-mix counts, encoded as pipe-separated values. Per-slice versions of the <code>perf_*</code> state, issue, FU, ROB, and remote message counters, encoded as pipe-separated values.

The summarized `ipc-combined.csv` files used for plots add workload and runner meta-data around the raw metrics: `benchmark`, `slices`, `ways`, `topology`, `status`, `returncode`, `seconds`, `log`, `metrics_csv`, and `results_xml`. The `metrics_csv` column points back to the raw row using the schema above.

References

- [1] Yanqi Zhou and David Wentzlaff. The Sharing Architecture: Sub-Core Configurability for IaaS Clouds. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 559–574, 2014.
- [2] Yanqi Zhou. *Configurable Architecture and Resource Provisioning for Future Clouds*. PhD thesis, Princeton University, 2018.
- [3] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 341–353, 2003.
- [4] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpfen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002. doi:10.1109/MM.2002.997877.
- [5] Robert M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.
- [6] James E. Thornton. Parallel Operation in the Control Data 6600. In *AFIPS Proceedings of the Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, volume 26, pages 33–40, 1964. Available: <https://cs.uwaterloo.ca/~mashti/cs850-f18/papers/cdc6600.pdf>.
- [7] James E. Smith and Gurindar S. Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE*, 83(12):1609–1624, 1995. doi:10.1109/5.476078.
- [8] Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, December 2019.
- [9] César Fuguet. HPDcache: Open-Source High-Performance L1 Data Cache for RISC-V Cores. In *Proceedings of the 20th ACM International Conference on Computing Frontiers (CF ’23)*, pages 377–378. Association for Computing Machinery, New York, NY, USA, 2023. Repository: <https://github.com/openhwgroup/cv-hpdcache>.

- [10] David Patterson, Jeremy Bennett, Palmer Dabbelt, Cesare Garlati, G. S. Madhusudan, and Trevor Mudge. Embench: An Evolving Benchmark Suite for Embedded IoT Computers from an Academic-Industrial Cooperative: Towards the Long Overdue and Deserved Demise of Dhrystone. RISC-V Workshop Zurich, 2019. Repository: <https://github.com/embench/embench-iot>.
- [11] Simha Sethumadhavan, Franziska Roesner, Joel S. Emer, Doug Burger, and Stephen W. Keckler. Late-Binding: Enabling Unordered Load-Store Queues. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 347–357, 2007.
- [12] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 275–286, 2003.
- [13] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 186–197, 2007.
- [14] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 422–433, 2003.
- [15] Yasuko Watanabe, John D. Davis, and David A. Wood. WiDGET: Wisconsin Decoupled Grid Execution Tiles. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 2–13, 2010.