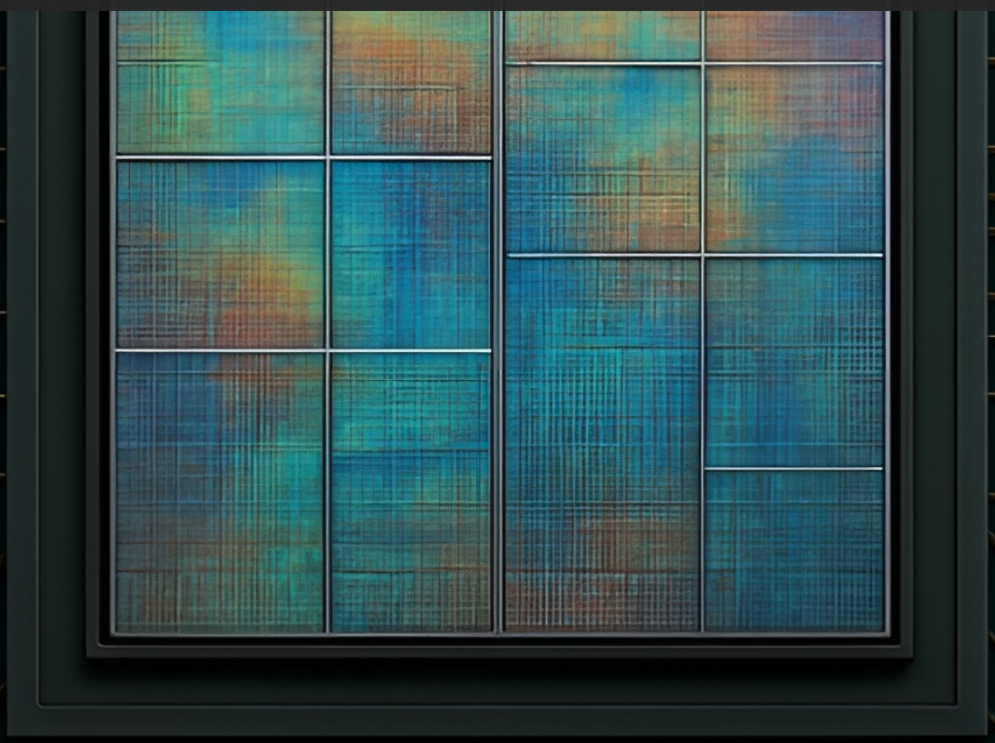
Study of Interlock Collapsing ALUs for RISC-V Microprocessors

Alexandre Gil Viudez



Delft University of Technology

TU Delft

Study of Interlock Collapsing ALUs for RISC-V Microprocessors

by

Alexandre Gil Viudez

Chair:	Dr. Ir. J.S.S.M. Wong
Core Member 2:	Prof. Dr. Ir. G.N. Gaydadjiev
Core Member 3:	Dr. C Gao
Project Duration:	12, 2024 - 10, 2025
Faculty:	Faculty of Electrical Engineering, Mathematics & Computer Science, Delft

TU Delft

Summary

The Interlock Collapsing ALU (ICALU) introduced in the 1990s by IBM sought to mitigate execution interlocks in processors [1]. These hazards occur when an instruction cannot be executed because it depends on the result of a previous instruction, effectively stalling the pipeline. ICALU aimed to collapse these instruction pairs into a single three-operand operation, solving them in parallel with a 3-1 ALU [1, 2, 3]. In order to not affect the machine cycle time, an optimized design was presented, extending the delay of traditional ALUs by only one Carry-Save-Adder stage. This project reevaluates ICALU in the context of contemporary RISC-V processors, where out-of-order engines dominate mainstream designs and presents its benefits and costs as a new potential way to improve processor performance.

Firstly, we performed a feasibility study which revisited IBM's assumptions, constraints and implementation and then mapped the RISC-V instructions that a potential ICALU could execute. It was followed by a trace analysis of CoreMark, Embench and SPEC CPU2017 integer to quantify the frequency and distribution of collapsible instruction pairs. The study showed that collapsible dependencies occur approximately once every eleven instructions (7.7%), half of them are adjacent and more than 92% are found within a distance of three instructions.

ICALU was then implemented in the state-of-the-art open source RISC-V core, BOOM, by incorporating a dedicated collapse detection logic and the three-input ALU with a focus on extending as few paths as possible. This served two purposes: first, to quantify the achievable instruction-level parallelism improvement of introducing ICALU through RTL simulation. Second, to evaluate the unavoidable design timing costs required to integrate ICALU into any current micro-architecture.

Cycle-accurate simulations revealed that ICALU introduces negligible benefits in 2-wide configurations (around 1%) but achieves measurable speedups of up to 7% in 3- and 4-wide versions of the core. Hardware counters were used to quantify the collapsed pairs the processor executed and committed, identifying 70-90% rate over theoretical results, ensuring these results represent ICALU's maximum potential uplift by exploiting almost all potential pairs found in the modern day code. The results demonstrate that performance gains grow with core width, indicating that conventional out-of-order engines become less effective at overcoming sequentiality imposed by execution interlocks.

The implementation process also revealed that in order to host ICALU, data-path widening is required from the issue queue to commit stages, hosting the equivalent of one extra micro-operation in the data-path. These trade-offs identified are unavoidable and universal to any micro-architecture that seeks to host ICALU.

This project concludes that introducing ICALU in modern RISC-V processors brings a 3-7% ILP uplift in integer workloads at the cost of increasing the delay of many paths in the instruction execution stages of the processor, commonly known as the back-end, which is frequently time critical.

If there is enough slack, ICALU can serve as a viable means to increase microprocessor performance, with more noticeable speed-ups in wider cores. Ultimately, this study presents both the advantages and limitations of integrating ICALU in modern RISC-V cores, leaving it to each implementation and designer to determine whether using the available slack or sacrificing cycle time is a worthwhile trade-off when incorporating ICALU in their core.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
ALU	Arithmetic Logic Unit
BOOM	Berkeley Out-of-Order Machine
CMOS	Complementary Metal–Oxide–Semiconductor
CPU	Central Processing Unit
CSA	Carry-Save Adder
CLA	Carry-Lookahead Adder
FPGA	Field-Programmable Gate Array
ILP	Instruction-Level Parallelism
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
ICALU	Interlock Collapsing ALU
OoO	Out-of-Order (execution)
ROB	Reorder Buffer
RTL	Register-Transfer Level (hardware model)
SoC	System-on-Chip
TB	Translation Block (QEMU)

1 Introduction

1.1. Context and Motivation

Modern CPU designs heavily rely on instruction-level parallelism (ILP) to speed up computation. By overlapping the execution of multiple instructions, processors make better use of their resources [4]. However, one of the main hurdles to achieving high ILP is the presence of execution interlocks, situations where an instruction depends on the result of a previous instruction. These interlocks force the processor to stall and issue these dependent instructions sequentially, greatly reducing the ILP [5].

In the early 1990s IBM proposed the Interlock Collapsing ALU (ICALU), an execution unit that allows dependent pairs of instructions that cause these interlocks to execute in one cycle by folding them into a single three-operand operation [1]. The researchers proved that an increase in the critical path of the machine was not necessary by applying an optimized design of a Carry-Save-Adder (CSA) + Carry-Lookahead-Adder (CLA) capable of executing 3-1 addition/subtraction and logical operations with common gates [1, 3]. Therefore, with the aim of removing execution interlocks, ICALU brought increases in ILP without degrading the frequency of the system, proving that it could boost computational performance. Early simulations showed up to double digit speedups for only a slight increase in silicon area [1, 2], establishing itself as a potential alternative to increase performance.

However, the idea never reached production, transistor budgets ballooned and full out-of-order scheduling soon became the standard way to raise ILP [6]. With the direction the industry took, ICALU was forgotten from mainstream design practice.

Since then, the drive for greater performance has pushed designers toward even bigger out-of-order engines, larger prediction tables and heavier speculation, yet each new generation yields diminishing single-thread returns [6, 7]. Conventional techniques to improve out-of-order machinery have been heavily stretched close to their practical limits. In this climate, schemes that focus on gaining extra ILP from a different approach than the one pursued over the past three decades are worth revisiting.

Meanwhile, RISC-V has emerged as a leading open ISA, supported by a rapidly growing ecosystem of commercial implementations and as an inviting platform for microarchitectural experimentation [8]. This setting raises a natural question: does a forgotten three-operand ALU still hold value in today's RISC-V cores or have modern pipelines and workloads left this concept obsolete?

1.2. Problem Statement and Research Goals

The main goal of this work is therefore to answer the following research question:

What are the benefits and costs of reintroducing an Interlock-Collapsing ALU into contemporary RISC-V processors?

Nevertheless, after more than 30 years, compiler optimizations and a different software landscape may have rendered obsolete the original distribution of collapse pairs found by IBM. Hence, in order to answer this question, it is needed to firstly start by understanding how collapsible instruction pairs appear

in contemporary workloads. After that, the goal is to confirm with realistic and accurate simulations, whether these opportunities translate into measurable ILP gains. However, as acknowledged by IBM, this improvement heavily relies on not increasing the delay of any path so processors that do not have slack in the modified path do not see their frequency reduced and the uplifts wasted. Therefore, once any ILP improvement has been demonstrated, its cost must be presented, giving a clear conclusion of the benefits and trade-offs, if any, of incorporating ICALU in the next generation of RISC-V microprocessors.

Hence, this work pursues the following specific goals:

- **Study** where, how frequently and under which **dependency patterns** collapsible execution interlock opportunities arise in modern workloads.
- **Demonstrate**, through a realistic system and accurate simulation that a modern processor that incorporates **ICALU increases ILP** on modern workloads.
- **Determine the costs**, specially in regards of timing, of incorporating ICALU in current-day RISC-V microprocessors.

1.3. Methodology Overview

The project follows a planned methodology, beginning with a feasibility study and culminating in cycle-accurate performance measurements. The process is summarized below.

1. **Feasibility of ICALU in modern RISC-V processors.** Before committing any engineering effort, it is required to confirm whether the constraints, assumptions and claims formulated by IBM for the original ICALU still apply, to ensure it can be replicated in modern designs. Furthermore, the potential instructions of RISC-V that could be used for an ICALU unit will be discussed.
2. **Obtain statistics of real-world workload traces.** QEMU will be instrumented to dump the dynamic instruction streams of the benchmarks chosen: CoreMark, Embench and the SPEC CPU 2017 integer benchmarks. A post-processing script will scan the traces and will dump metrics such as inter-pair distance and instruction frequency. This will answer the research goal of how collapse candidates appear in current workloads and will help build optimally the hardware prototype of the following stage.
3. **Incorporate ICALU in the BOOM core.** The Berkeley Out-of-Order Machine (BOOM) will be extended with ICALU support. This process will serve 2 purposes. The first one, to use this design to measure the ILP gains in the chosen workloads through RTL simulation. The second one, to identify the minimal trade-offs of incorporating ICALU in any modern processor. This process starts with a design exploration, focusing on producing the minimal potential timing trade-offs that later will be used for the final answer of this work.
4. **Measure ILP boost.** Using the Rocket Chip generator, a system-on-chip configuration incorporating a single BOOM core will be produced. The baseline and ICALU-enabled SoCs will be simulated in RTL with a full memory system running CoreMark and Embench. Due to the parameterizable nature of BOOM, different core widths (2,3,4) will be tested. Benchmark scores will be collected and compared against the baseline BOOM design, yielding representative performance data of incorporating ICALU. Linking the trace analysis results and the performance results of CoreMark and Embench, an extrapolation of SPEC potential speed-up will be presented, to reflect large-scale workloads.
5. **Integrate performance results with implementation lessons.** The experience of modifying BOOM will be used to highlight the minimal timing and complexity trade-offs required to host ICALU. This project will conclude with combining the aforementioned trade-offs linked to the measured performance results to provide the final answer on ICALU's consideration in modern RISC-V microprocessors.

1.4. Thesis Organization

The remainder of this thesis is arranged to take the reader from historical context through feasibility analysis, implementation and evaluation, before closing with the main conclusions, as outlined below.

Chapter 2 – Background
Revisits IBM's Interlock-Collapsing ALU (ICALU) concept, its design assumptions and performance studies and introduces the modern open-source ecosystem used in this work: RISC-V, Chipyard, QEMU, benchmark suites and the BOOM core.

Chapter 3 – Feasibility Study and Workload Characterization
Re-examines IBM's assumptions in the context of modern ISAs and technology, maps ICALU onto the RISC-V instruction set and presents a trace-based analysis of CoreMark, Embench and SPEC2017 to quantify the frequency, distance and patterns of collapsible pairs.

Chapter 4 – ICALU Integration into the BOOM Core
Describes the engineering process of incorporating ICALU into a state-of-the-art out-of-order RISC-V core. It establishes design requirements, explores architectural choices for collapse detection and execution and details the pipeline modifications required to support collapsed micro-operations.

Chapter 5 – Measurements and Results
Presents the experimental setup and RTL simulations comparing baseline and ICALU-enhanced BOOM cores across different widths. It reports performance uplifts, extrapolates to SPEC2017 and summarizes the unavoidable design trade-offs identified during integration.

Chapter 6 – Conclusions
Summarizes the main findings, highlights the contributions of this work and discusses future research directions.

2

Background

Revisiting IBM's work before diving into any discoveries about collapse opportunities or new micro-architectural changes is the first step of this work. It is necessary to understand what IBM proposed to see if it can be translated to modern-day processors 30 years later. Furthermore, background on the tools used in this project is presented, both to justify motivation and serve as a foundation for readers less familiar with the RISC-V environment. In other words, this chapter brings together IBM's pioneering Interlock Collapsing ALU work with an overview of the open RISC-V ecosystem and the Chipyard tool flow.

Section 2.1 summarizes IBM's papers regarding ICALU, from the design assumptions, details and implementation to the results of their simulation studies, showing the performance uplift reported by ICALU. Subsequently, in Section 2.2, RISC-V is introduced as an open platform that supports rapid hardware exploration, followed by a description of the Chipyard framework in Section 2.3. In this section, the benchmark suites used for the rest of this work appear in Section 2.3.1, while Section 2.3.2 presents QEMU for trace collection. Section 2.3.3 outlines the BOOM out-of-order core that will host the ICALU modifications. Section 2.4 presents the conclusion of this section, summarizing the key takeaways of the IBM papers in both design and performance studies.

2.1. IBM's Approach to Improve ILP: ICALU

Increasing instruction-level parallelism (ILP) is a crucial aspect of CPU design. IBM's work in the early 1990s aimed to tackle one of the key challenges to ILP: execution interlocks. This section revisits IBM's seminal work on the Interlock Collapsing ALU (ICALU) by first examining the problem they wanted to solve: execution interlocks, then it moves to ICALU's assumptions, constraints and design implementation [3, 9, 1] and ends with the results of IBM's simulations when researching the potential ICALU speedup [2].

2.1.1. Execution Interlocks and ICALU's Principle

Execution interlocks arise when an instruction must wait for the result of a previous one before it can execute. For instance, consider the sequence of instructions in Figure 2.1. The ADD operation produces a value in register `x1`, which is required by the subsequent SUB operation. The CPU cannot issue the SUB instruction until the ADD operation has completed, causing a stall in the pipeline. These interlocks inherently reduce ILP by forcing the processor to execute instructions sequentially [5].

These pairs are often referred to in this work as collapse pairs or producer-consumer pairs, with the ADD operation being the producer of a result that the SUB, the consumer, needs.

During the 1990s, single-issue pipelines were the dominant architecture. Multi-issue pipelines, also referred to as superscalar architectures, were emerging at the time, offering the ability to execute multiple instructions per cycle but were particularly affected by execution interlocks [1]. However, solutions to overcome this problem like complex out-of-order execution mechanisms with large instruction queues were expensive in terms of area and power and were not always fully effective against execution in-

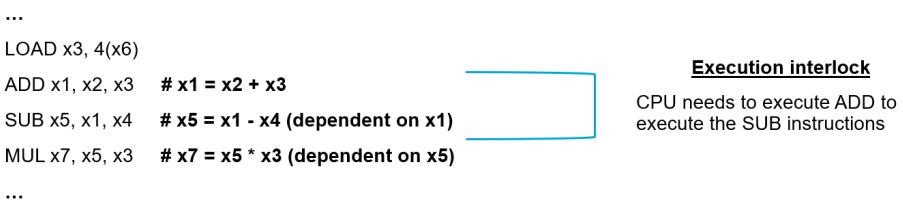


Figure 2.1: Execution interlock where a dependent SUB instruction must wait for the result of the preceding ADD.

terlocks or at increasing ILP [1]. Compared to out-of-order superscalar architectures, IBM sought a simpler, more cost-effective solution to mitigate interlocks and increase ILP: the Interlock Collapsing ALU. This potential innovation focused on collapsing data-dependent instructions to execute them in a single cycle, avoiding any pipeline stalls.

ICALU's Principle

IBM's ICALU addressed the interlock problem by merging two dependent ALU instructions into a single operation. Instead of stalling the pipeline, the ICALU executes both instructions in one cycle by using two ALUs, as shown in Figure 2.2. ALU1, a traditional ALU, computes the producer's result (e.g., `x1 = x2 + x3`). In parallel, ALU2, a special 3-1 ALU, computes the final result of the pair by calculating the equivalent three-operand operation (e.g., `x5 = x1 - x4`, which is equivalent to `x5 = x2 + x3 - x4`). This setup eliminates the need for the intermediate result to be written back to the register file before the consumer executes, providing the intermediate result of the producer-consumer pair and ensuring correct architectural behavior.

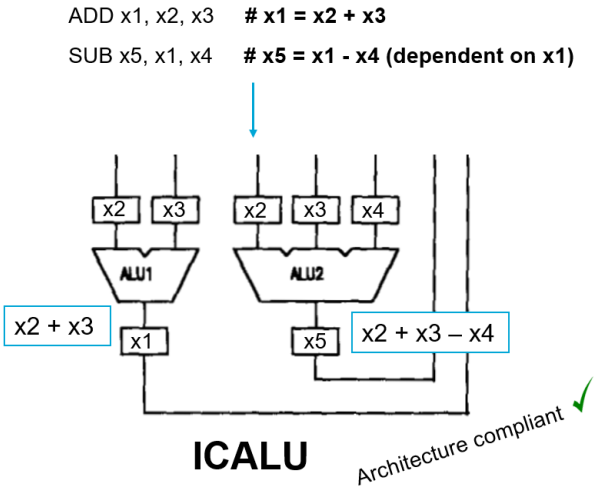


Figure 2.2: Conceptual view of the ICALU design with an example. Two dependent instructions (`ADD` and `SUB`) are executed in a single cycle using a traditional ALU (ALU1) and a special 3-1 ALU (ALU2) [1].

2.1.2. Design Constraints and Assumptions

Before presenting the ICALU implementation in detail, it is important to summarize the main design constraints that guided IBM's work, as well as the assumptions made to simplify and evaluate the design.

Design Constraints. The ICALU had to operate under several strict requirements [1]:

- **ISA compatibility:** The unit had to be fully compliant with any existing architecture, meaning no modifications to the specific instruction set or software where it would be deployed were allowed.
- **Technology envelope:** The design was restricted to standard CMOS logic, relying on a well-defined book-set of available gates:

  - 2-input gates (`XOR`, `XNOR`).
  - Multi-input gates (up to 8-way `AND`, `NAND`, `OR`, `NOR`).
  - Specialized 3-to-4-way gates (`AND-OR`, `AND-NOR`).

  These restrictions ensured that the ICALU could be implemented in any CMOS node without custom cells.

- **Cycle time budget:** Incorporating ICALU into any process could not increase the machine cycle time. In other words, ICALU had to fit within the same timing budget as a conventional ALU.

The designers set out to create a solution that imposed no changes on the ISA or software, was not restricted to custom gate types and most importantly guaranteed that any ILP increase would translate directly into performance gains without degrading machine frequency. This last point was critical: regardless of the available timing slack in a given implementation, ICALU could not lengthen the delay of any path.

In conclusion, IBM implemented ICALU aiming for a universal design capable of increasing ILP and performance independently of the underlying architecture, ISA, process node or implementation characteristics.

Assumptions. With these constraints in mind, IBM adopted a number of simplifying assumptions:

- **Conventional pre-add logic:** Tasks such as sign extension and inversion were assumed to be handled in the same way as in a traditional ALU; therefore, the design implementation would focus on providing new addition and post-addition logic inside the ALU.
- **One-stage delay model:** Each logical block of the book-set chosen was treated as a "one-stage" element, defined by the delay of the slowest gate in the library, the specialized 3-to-4-way gates. This made critical-path analysis more predictable.
- **Ignoring rare 4-to-1 cases:** Some consumer-producer pairs expand into expressions that require four operands. For example:

  ```
  sub x5, x1, x2     # x5 = x1 - x2
  add x6, x5, x5     # x6 = (x1 - x2) + (x1 - x2)
  ```

  This produces `x6 = x1 - x2 + x1 - x2`, a 4-1 pattern that would need specialized hardware. IBM assumed such cases were rare and consequently excluded them from ICALU execution.

2.1.3. ICALU's Design Implementation

Having set the constraints and assumptions, this section will talk about the design implementation done by IBM. Before going into the details, there are some design preliminaries to note:

Preliminaries. IBM targeted ICALU to execute collapses of instructions in the ESA/370 ISA. The supported instructions, listed in Figure 2.3 [9], include `ADD`, `SUB` and basic logical operations such as `AND`, `OR`, `XOR` and `NOT`. Certain special operations, for example `Load Positive Register (LPR)`, were also considered. However, with the aforementioned assumption of conventional pre-add logic remaining the same and solely focusing on the add-logic part of the ALU, such cases reduce to a standard add with an appropriately preprocessed operand [1]. In addition, ICALU was employed for address generation in loads and stores [1].

Therefore, the collapsible pairs can be grouped by operation pattern into four categories that cover combinations of arithmetic and logical pairs, as summarized in Figure 2.4 [3]. These categories specify what the three-operand ALU must realize once a producer and a consumer are fused. Even though the prototype targeted execution of ESA/370 ISA instructions, the small set of required arithmetic and logic primitives makes the scheme applicable to other ISAs and implementations and helps with the goal of making ICALU a universal solution.

Instruction	*Operation*	*h*
AR R1,R2	$(R1) \leftarrow (R1) + (R2) + h$	0
SR R1,R2	$(R1) \leftarrow (R1) + (\overline{R2}) + h$	1
LPR R1,R2;	$R2<0$; $(R1) \leftarrow 0 + (\overline{R2}) + h$	1
	$R2 \geq 0$; $(R1) \leftarrow 0 + (R2) + h$	0
LNR R1,R2	$R2 \geq 0$; $(R1) \leftarrow 0 + (\overline{R2}) + h$	1
	$R2<0$; $(R1) \leftarrow 0 + (R2) + h$	0
LR R1,R2	$(R1) \leftarrow 0 + (R2) + h$	0
LTR R1,R2	$(R1) \leftarrow 0 + (R2) + h$	0
LCR R1,R2	$(R1) \leftarrow 0 + (\overline{R2}) + h$	1
ALR R1,R2	$(R1) \leftarrow (R1) + (R2) + h$	0
SLR R1,R2	$(R1) \leftarrow (R1) + (\overline{R2}) + h$	1
CR R1,R2	$X \leftarrow (R1) + (\overline{R2}) + h$	1
CLR R1,R2	$X \leftarrow (R1) + (\overline{R2}) + h$	1
NR R1,R2	$(R1) \leftarrow (R1) \wedge (R2)$	0
OR R1,R2	$(R1) \leftarrow (R1) \vee (R2)$	0
XR R1,R2	$(R1) \leftarrow (R1) \forall (R2)$	0

Figure 2.3: List of the ESA/370 instructions used to collapse in ICALU [9].

Category 1: $\mathbf{A + B + \Gamma}$
Arithmetic followed by Arithmetic

Category 2: $\mathbf{B + (A\ LOP\ \Gamma)}$
Logical followed by Arithmetic

Category 3: $\mathbf{B\ LOP\ (A + \Gamma)}$
Arithmetic followed by Logical

Category 4: $\mathbf{B\ LOP\ (A\ LOP\ \Gamma)}$
Logical followed by Logical.

Figure 2.4: Categories of collapses based on the instructions of Figure 2.3 [3].

Implementation

With the execution scope defined, Figure 2.2 already showed a high-level view of the implementation. To briefly reintroduce it, ICALU uses two ALUs to execute a producer-consumer pair in one cycle. ALU1, a conventional two-operand unit, computes the producer's result. In parallel, ALU2, a specialized three-operand unit, consumes the producer's inputs together with the consumer's remaining operand and directly produces the final value. Taking into account that ALU1 is a traditional ALU and that in ALU2 the pre-add logic is assumed to be the same, the core contribution of IBM's work is the addition and post-addition logic of ALU2, which performs the equivalent of a three-operand add-logic operation, which was implemented with the following process:

Three-operand ALU (ALU2). IBM started with a simpler, non optimized, design and kept refining it until it met the constraints set before. To execute the mathematical operations of Figure 2.4, IBM started with a straightforward design that combined a carry-save adder (CSA), a carry-lookahead adder (CLA), multiplexers, explicit logic blocks to cover the four collapse categories, as depicted on the left side of Figure 2.5. At first glance, this naive structure is functionally complete but clearly clashes with one of the constraints of this design because it does not meet the same cycle-time budget as a traditional ALU.

As showed in Figure 2.5, the papers develop the strategy for how the datapath was optimized to reduce it to a CSA+CLA path. This process involved multiple boolean equations, operations and tricks. The explanation of these are out of scope but summarize this optimization process, three techniques enable this transitions from a long delay 3-1 ALU to the final CSA+CLA structure:

- Repurposing unused inputs on existing gates of the CSA or CLA structure to fold logical operations in the addition paths.
- Replacing multiplexers with transparency control signals by incorporating them also in the unused inputs of the CSA or CLA structures.

- Shifting certain logic from the adder data-path into the control-signal generation blocks.

The final organization, in the stage-counting model used by the authors, the design is formed by a CSA + CLA of 5 stages of delay structure instead of a typical CLA structure of 4 stages of delay. In order to meet the constraints set by the designers, this was argued not to affect the machine cycle time by comparing ICALU to subsystem architectures such as the RS/6000 multiply-add unit or the ESA/370 multiply unit. The papers highlighted that multi-operand additions had longer paths compared to the 5-stage CSA+CLA adder and therefore integrating one more stage of delay in the ALU would not affect machine cycle time [1].

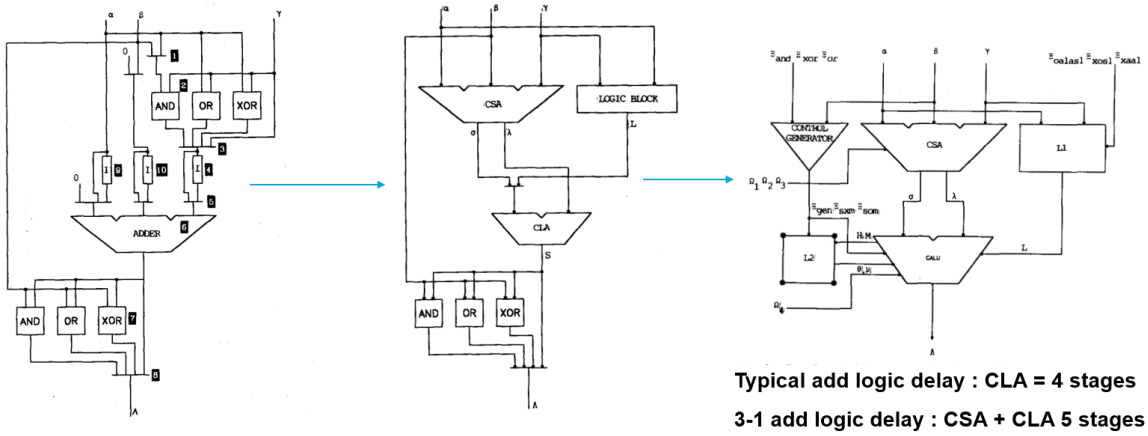


Figure 2.5: Optimization trajectory of the three-operand ALU. Left: functional but slower realization with explicit logic and multiplexers. Middle: integrated CSA/CLA with consolidated logic. Right: final organization driven by control generation and transparency, achieving a delay of `CSA + CLA`. A typical CLA add is four stages; a 3-to-1 add is 5 stages [3].

Condition codes. However, this explanation covered only the addition logic of the ALU and not the post-addition logic of this unit. In environments that maintain condition codes for results such as zero, carry and overflow, the papers describe how this CC generation can be performed in parallel so that it does not lengthen the ALU critical path [1]. The required flags are computed with dedicated side logic and control, consistent with the cycle-time constraint and the gate book set.

IBM's Research Conclusions

After describing the ICALU implementation in detail, IBM concluded with the following remarks:

1. **Cycle Time Impact:** A critical goal of ICALU was to avoid extending the processor's critical path. If any ILP increase came at the cost of increasing the delay of any path, ICALU would not be a universal design that, regardless of the slack of each stage in all implementations, would bring a performance boost. It was demonstrated that the 3-1 ALU could be implemented with one more stage using a CSA+CLA structure and that the additional stage did not increase the critical path in any machine, as other units had longer paths regardless [3].
2. **Area Overhead, Fan-In and Fan-Out:** While the design introduced some additional logic, the area overhead was assumed to be modest. Instead of focusing on precise transistor counts or custom designs, the feasibility of the design within the constraints of common CMOS technology was emphasized. The papers treated concerns about area, fan-in and fan-out as negligible, as ALUs could be considered, textually, "a commodity" [1].
3. **Complementary Nature of ICALU:** While ICALU was not designed to replace out-of-order execution or superscalar dispatch, it was positioned as a complementary solution. Because it can act as a normal ALU, with minimal area increase, a notable performance gain might be obtained. If a design consists of multiple ALUs, integrating an ICALU might only require replacing them with a slightly bigger 3-1 design [1].

These results established the feasibility of the design and highlighted its potential to enhance ILP in architectures that face interlock challenges in any ISA, software, process node or processor implemen-

tation. However, these conclusions only discussed the technical feasibility part of the design and not the actual potential improvements. This means that the most important part is left: to see if all this effort results in noticeable performance gains.

2.1.4. ICALU's Performance Studies

The evaluation of ICALU involved several studies, each refining its approach to simulate and measure the performance impact of this novel concept as the research progressed. Among the three main papers documenting these efforts, the most accurate, comprehensive and latest study is the one selected for the following explanation [2]. The other studies were simpler simulations of the later one and does not add value in this revision.

Simulation Setup

The simulation setup relied on a cycle-approximate approach, where instruction traces from SPEC CINT92 were fed into a dispatch environment with multiple functional units. While not fully cycle-accurate, this methodology effectively modeled real-world performance under certain key assumptions such as memory operation latencies, store buffering mechanisms and fixed branch prediction with 85% accuracy [2]. These assumptions captured nonidealities in a simplified manner. In the context of the 1990s, where simulation tools were far less developed than they are now, this yielded enough accuracy to validate performance potential.

The performance analysis involved five different models, each designed to test various configurations of execution unit availability and instruction issue widths, as shown on the left side of Figure 2.6. These models were evaluated under both in-order and out-of-order instruction issue scenarios to see where ICALU brought more performance gains [2]. Furthermore, register forwarding was also evaluated in some cases.

Configuration	Models				
	1	2	3	4	5
Inst. Cache Fetch BW	2	2	4	4	8
Data Cache Ports	1	2	2	2	4
Integer ALUs	2	2	2	4	8
Integer Mult/Div	1	2	2	4	8
F.P. Add/Mult/Div	1	2	1/1/1	4	8

Evaluation Cases for In–Order and Out–of–Order Instruction Issue	
Parallel issue of independent instructions only (Base Case)	A
Parallel issue with the addition of register forwarding	B
Parallel issue with the addition of both register forwarding and interlock collapsing (#ICALUs = #ALUs/2)	C

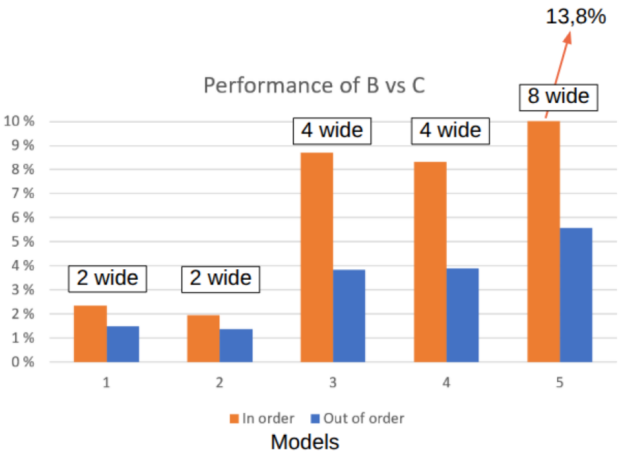


Figure 2.6: On the right side, on the horizontal axis, the models simulated by IBM with in-order and out-of-order variants. On the vertical axis, the ILP speedup with ICALU (C) vs without ICALU (B). On the left side, the parameters of each Model 1 to 5 simulated. Model 1 and 2 are 2-wide cores, Model 3 and 4 are 4-wide cores and Model 5 is an 8-wide core [2].

Results and Key Observations

The performance results, illustrated in Figure 2.6, demonstrated a consistent increase in parallelism across all models, with more pronounced improvements as resources were scaled up. While the original authors provided a detailed analysis of these results, some aspects of their evaluation must be analyzed in the context of current architectural trends because conclusions are slightly different.

For in-order designs, the performance uplift was around 2% for a 2-wide core and around 10% and 14% for wider cores. The latter results are not that relevant, as current transistor budgets require and allow out-of-order execution to extract ILP for more than 2-wide cores. It is not a realistic scenario to create a 4-wide core with in-order execution. Therefore, that 2% speedup for 2-wide in-order cores is what remains relevant for a modern-day reanalysis. The out-of-order (OoO) results were smaller but

still noticeable. The 2-wide version showed a 1.5% speedup, around 4% for a 4-wide core and 5.5% in an 8-wide core. There is a clear trend: the greater the width, the greater the performance uplift from incorporating ICALU. In more detail, it seems that when keeping the front-end bandwidth similar but increasing the back-end resources, the performance uplift remains constant, indicating this correlation is tied to front-end parameters only. This can be observed by comparing Model 1 vs 2 and 3 vs 4 of Figure 2.6, which have the same fetch bandwidth but different functional-unit resources.

The findings underscore ICALU's ability to uplift performance through the removal of certain execution interlocks. In today's landscape, however, these results must be approached cautiously. The heavy evolution of ISAs, software workloads and micro-architectures renders any direct conclusions from the historical analysis largely inapplicable to current designs. These results therefore motivate revisiting ICALU's applicability in the modern era and confirming that collapse opportunities still arise before committing engineering effort, which is the work later done in Section 3.3.

2.2. RISC-V: An Emerging Open Playground for Revisiting ICALU

Revisiting IBM's ICALU concept in today's systems needs an open and widely supported architectural platform. To fulfill the research goals within a reasonable time, it is not possible to develop the simulators and tools from scratch as IBM did in the 1990s and as explained in Section 2.1.4. RISC-V has emerged as an ideal foundation for this exploration. Introduced in the early 2010s at UC Berkeley, RISC-V is an open-source instruction set architecture (ISA) explicitly designed to be modular and extensible [10]. Unlike proprietary ISAs such as x86 or Arm, RISC-V's openness allows developers to freely modify or extend the instruction set [10]. This openness also extends beyond the ISA to designs, simulators and many other tools that are free, open-source and rapidly emerging.

This flexibility is built into its design: the base ISA can be paired with a selection of standard extensions (each identified by a letter) to tailor a processor for different needs. For example, the base 64-bit core (RV64I) can include the M extension for integer multiplication/division, A for atomic operations, F and D for single- and double-precision floating-point and C for 16-bit compressed instructions. These five extensions (I, M, A, F, D) are collectively referred to as the general-purpose "G" set [10]. A typical Linux-capable RISC-V processor is therefore denoted RV64GC, indicating a 64-bit core with the general-purpose extensions and the compressed instruction set [10]. This modular ISA approach means that a design can scale from minimal microcontrollers up to powerful multicore systems by simply enabling the appropriate extensions.

RISC-V's popularity and growth

In just over a decade, RISC-V has seen rapid adoption across industry and academia. Initially it gained traction in microcontrollers and other embedded devices due to its efficiency and zero licensing cost. Now, it is also proving itself in high-performance domains. Multiple groups have implemented 64-bit out-of-order RISC-V cores capable of running full operating systems at competitive speeds. For instance, SiFive's U8 Series was a 2019 milestone that demonstrated a quad-core RV64GC SoC running Linux [11]. Similarly, Alibaba's DAMO Academy announced the Xuantie 910, a 16-core, 2.5 GHz out-of-order processor built on RISC-V. The Xuantie 910 not only runs Linux, but was later open sourced, exemplifying the RISC-V community's ethos of shared innovation [12].

These developments show that RISC-V is not only for small-scale projects: it can deliver the complexity and performance needed for modern computing. It means that an idea like ICALU, which aims to exploit additional instruction-level parallelism and reach new levels of performance, has a place in RISC-V solutions and is not doomed to modest, simple embedded processors.

Openness fostering innovation

The open and modular nature of RISC-V makes it especially attractive for micro-architectural research and experimentation. Developers and researchers have a rich ecosystem of open-source tooling and hardware descriptions. For example, multiple high-quality RISC-V core implementations (in-order and out-of-order) are available for study and modification. Community-maintained compilers, operating systems and simulators have been built, lowering the barrier to trying new architectural ideas.

In short, RISC-V provides a plug-and-play ISA canvas, backed by industry support, on which novel concepts like the ICALU can be practically tested. This project leverages that openness: ICALU is

implemented in a state-of-the-art RISC-V core and standard RISC-V software (compilers, benchmarks, functional simulators) is used to evaluate it, all with minimal friction. The ability to work within a full open-source stack, from the ISA specification to the hardware RTL and through the operating system, is a key enabler for realizing the experimental setup in this thesis. All of this is smoothly connected through Chipyard, a framework of tools that facilitates this task and will be discussed in the next section.

2.3. Chipyard: A State-of-the-Art Platform for Micro-architecture Development

To implement and evaluate the ICALU concept in a modern setting, this project leverages Chipyard, an open-source framework for agile development of RISC-V systems-on-chip. Chipyard provides a "one-stop shop" with a collection of processor cores, peripherals and tools, all built around the Chisel hardware description language. It includes in-order and out-of-order RISC-V cores (such as Rocket and BOOM), memory systems and even accelerators, allowing designers to quickly assemble a full Linux-capable SoC [13]. Moreover, Chipyard integrates various development flows, from RTL simulation with Verilator to VLSI design flows, under a unified environment.

In this work, Chipyard serves as the backbone for all tasks, from compiling the workload to simulating the BOOM processor. It is thanks to Chipyard that this revisiting of ICALU can go much further than what IBM did in the 1990s and was presented in Section 2.1.4: instead of a cycle-approximate simulation, in a relatively short time this work produces a trace analysis and cycle-accurate results by doing RTL simulation of a real state-of-the-art core. The following subsections introduce the key components from the Chipyard ecosystem used in this project: the benchmark suites employed for evaluation [14, 15, 16], the QEMU functional simulator [17] and the BOOM core [18].

2.3.1. Benchmark Suites: CoreMark, Embench and SPEC 2017

Evaluating micro-architectural improvements requires workloads that represent core performance. In this project, three well-known CPU benchmark suites are used: CoreMark, Embench and SPEC CPU2017. These are selected as representative workloads, being the standard in industry and academia to evaluate core performance and, in turn, the potential improvement for ICALU. These workloads are used across the whole of this work, from the trace analysis to the cycle-accurate performance evaluation. Each serves a different purpose in terms of speed and comprehensiveness of evaluation.

CoreMark (developed by EEMBC) is a small but sophisticated benchmark designed specifically to test the basic performance of mainly embedded processor cores. It runs a mixture of simple algorithms (like list sorting, matrix manipulation, state machine processing and CRC calculation) to produce a single-number score [14]. CoreMark, being an embedded benchmark, can run on the order of millions of instructions, which is required for an RTL simulation.

Embench, developed by the Bristol/Embecosm group, is a collection of small embedded benchmarks intended as a more comprehensive replacement for older suites such as Dhrystone and CoreMark [15]. Unlike CoreMark, which condenses performance into a single score from a fixed set of kernels, Embench is composed of multiple real-world programs, each producing an individual runtime result. The suite includes applications such as cryptographic routines, compression/decompression, digital signal processing and integer arithmetic, covering a wider and more diverse set of operations than CoreMark alone. Similar to CoreMark, they retire on the order of millions of instructions, making them complete in reasonable time for RTL simulation. This SPEC-like compact benchmark suite provides a better view of how different workloads can affect ICALU. For this project, Embench complements CoreMark by adding variability and breadth in workload characterization, offering a richer perspective on where ICALU opportunities may arise. Both CoreMark and Embench therefore are used as fast workloads for the RTL simulation, together with the trace analysis.

On the other hand, SPEC CPU2017 is a comprehensive industry-standard benchmark suite aimed at measuring compute-intensive performance using real application workloads. The SPEC 2017 suite contains 43 programs drawn from real-world domains (such as C compilers, video compression, AI and physics simulations), split into integer and floating-point categories. In this thesis, in the SPEC CPU2017 suite, the integer benchmarks are used, as ICALU benefits the most for integer applications. Furthermore, the SPECrate suite is used; SPECspeed is the same but for multicore performance. Be-

cause the benefits of ICALU are in ILP, only single-core performance needs to be studied. Due to its complexity and use of full-length applications, SPEC CPU2017 is far more demanding to run; a full iteration can take years on RTL simulators; therefore, it is only considered in the trace analysis and extrapolations of the ILP improvement are obtained afterward from the trace analysis and other benchmark results.

Taken together, these three suites provide a balanced framework for evaluating ICALU. CoreMark and Embench are small enough to be run to completion on an RTL-simulated BOOM core, making them suitable for obtaining cycle-accurate performance comparisons between the baseline and ICALU-enabled designs. SPEC CPU2017, while far too demanding to execute in RTL simulations, acts as the comprehensive gold standard of core performance. By analyzing its traces, this work captures how collapsible pairs appear in large, complex workloads that more closely resemble real applications.

In conclusion, this strategy has two sides: trace analysis across all three suites is used to understand how collapsible opportunities arise in both embedded-scale and full-scale workloads, while RTL simulations of CoreMark and Embench quantify how such opportunities translate into ILP and performance improvements in practice. Although CoreMark and Embench lack the full complexity of SPEC, their results, interpreted alongside the SPEC trace analysis, ensure that the conclusions drawn reflect not only small embedded kernels but also the broader behavior of modern, compute-intensive workloads.

2.3.2. QEMU: A High-Speed Functional Simulator

QEMU is utilized in this project as a high-speed functional simulator for the RISC-V architecture. QEMU performs dynamic binary translation of target instructions to host instructions, enabling it to emulate a RISC-V machine at near-native speeds [17]. This makes it an ideal tool for tasks such as software debugging and workload exploration and, in the context of this work, gathering execution traces in RISC-V at native speed without any RISC-V hardware.

In the context of this thesis, by running the workloads from the previous section in QEMU, one can obtain traces and later identify how frequently collapse-worthy instruction sequences occur in typical programs. This information guides the design, for instance, confirming that there are sufficient dependent instruction pairs (ADD followed by SUB, etc.) to justify the complexity of an ICALU. The use of QEMU in this workflow is facilitated by Chipyard's tooling: it can launch the compiled workloads mentioned earlier on QEMU just as it does on real hardware, making it straightforward to obtain the instruction traces of any desired binary and, in turn, the selected representative benchmarks.

2.3.3. BOOM: A State-of-the-Art Open-Source Out-of-Order RISC-V Core

The RISC-V core chosen for this work is the Berkeley Out-of-Order Machine (BOOM), a family of open-source superscalar cores developed at UC Berkeley. Rather than a single fixed RTL file, BOOM is written in the Chisel hardware construction language as a *generator*: at elaboration time, the designer dials micro-architectural parameters such as decode width, functional-unit mix, queue sizes, cache options, etc. and Chisel emits a custom Verilog implementation. This flexibility is especially useful for experimenting with enhancements like the ICALU, since the core can be instantiated in various widths (2-, 3- or 4-wide issue) to gauge the ILP benefits across different superscalar aggressiveness [19].

Since its first release, it has evolved from an educational three-issue pipeline into *SonicBOOM*, a custom wide design that remains the highest-IPC open-source CPU ($\approx$6.2 CoreMark/MHz). Its third version, BOOM v3, nicknamed SonicBOOM, is, as of its release, "the fastest publicly available open-source core" in terms of IPC [20]. It was chosen for this project because of its parameterizable nature and its status as a state-of-the-art design.

BOOM is broadly modeled after the MIPS R10k and DEC Alpha 21264 micro-architectures. Like those classic designs, it implements explicit register renaming with a unified physical register file and a reorder buffer for in-order retirement [19]. Conceptually, the pipeline is broken into the usual stages: Fetch, Decode, Rename, Dispatch, Issue, Execute, Memory, Write back and Commit. In practice, some of these are combined (yielding a 7-stage core in the current implementation) to balance pipeline depth against cycle time. Figure 2.7 shows a high-level block diagram of the BOOM pipeline, more specifically, the 2-wide version.

The front end (fetch and branch prediction) supplies a continuous stream of instructions, while the

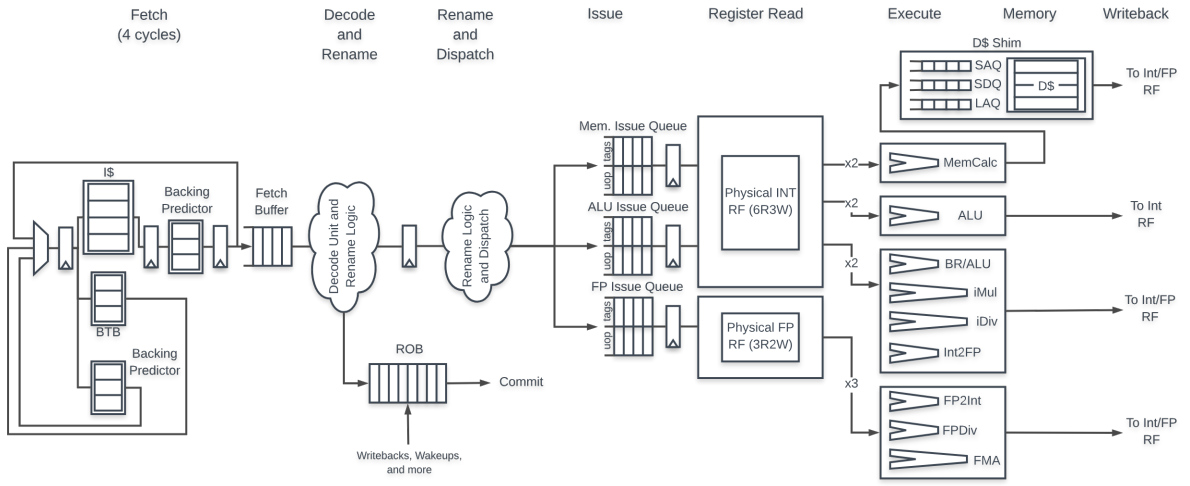


Figure 2.7: Simplified micro-architectural diagram of the BOOM core (third-generation SonicBOOM). The front-end (left) fetches and predicts instructions, while the back-end (right) handles out-of-order execution. Notably, BOOM employs multiple issue queues (Int, Mem, FP) feeding functional units and a unified physical register file with a reorder buffer for commit. Reproduced from [19].

back end contains everything from Decode/Rename onward, dealing with the out-of-order execution of instructions. However, as will be seen later in the implementation, all the modifications required for ICALU take place in the BOOM back-end stages. For the purposes of this study, the front end can be ignored. It is therefore necessary to understand how this back end works in order to properly dive into the ICALU modifications done in later chapters. Each back-end stage of BOOM, reflected in Figure 2.7, does the following:

- **Decode** drains the Fetch Buffer, expands each instruction into a single *micro-op* (UOP) for the back end and attaches branch speculation metadata.
- **Rename** maps architectural (logical) registers `x0–x31` onto a large pool of physical registers, removing false dependencies and creating room for out-of-order execution.
- **Dispatch** writes the renamed micro-ops into one of several issue queues (INT, MEM, FP). The ROB entry is also allocated here, providing precise trap and retirement bookkeeping.
- **Issue** is the heart of the OoO engine: a greedy scheduler selects ready UOPs whose operands are available and grants them a functional-unit slot; others continue to wait, accumulating wake-ups from bypasses or the register file.
- **Register Read** occurs the cycle after issue. Operands are pulled either from the unified physical register file or directly from the bypass network when producer and consumer are back-to-back.
- **Execute** hosts the ALUs, multipliers, dividers, branch units, and memory-address generators. Memory operations calculate their effective address here and enqueue it into the Load/Store Unit (LSU).
- **Memory** contains the LSU, split into a *Load Address Queue* (LAQ), *Store Address Queue* (SAQ) and *Store Data Queue* (SDQ). Loads fire when their address is known and no older store aliases it; stores wait until the ROB commits them.
- **Write back** returns ALU and load results to the physical register file, simultaneously forwarding them to any waiting consumers.
- **Commit** is managed by the ROB. When the head entry is marked “done,” architectural state is updated and store data in the SDQ is finally released to the cache.

Finally, every in-flight instruction carries a branch tag describing which unresolved branches it depends on. If a branch later proves mispredicted, the pipeline restores the precise register-rename state and replays only the affected instructions, ensuring forward progress with minimal flush cost. With this stage-wise context in place, the remainder of this chapter focuses on how the BOOM core can be built into

an SoC and the flexibility Chipyard brings into validating the changes made to the micro-architecture during development.

Rocket Chip: BOOM-based SoC. BOOM is delivered as a bare core; a full system is produced by inserting it into the *Rocket Chip* generator. Rocket Chip provides coherent caches, on-chip interconnect, peripherals and the TileLink infrastructure required to boot Linux [21]. All of those components are themselves parameterizable, so the same build description can generate a minimal bare-metal platform for fast testing or a full Linux-capable SoC for performance runs. For this project, the SoC contains one BOOM tile, a shared L2 cache, DDR memory models and standard UART and interrupt controllers, which is the basic and default configuration.

Simulation and Verification Environment. One major advantage of using BOOM (within the Chipyard framework) is the rich simulation and verification support available. The hardware design, once elaborated with Chisel, can be automatically generated into synthesizable Verilog and then simulated using industry-standard tools. For functional correctness testing and rapid iteration, Verilator is used as a fast open-source RTL simulator [22]. Its speed is faster than traditional RTL simulators, but it is still limited to simulating thousands of cycles per second. This is enough to execute embedded benchmarks that retire millions of instructions, on the order of hours.

The Chipyard build system integrates Verilator directly: with a simple *make* command, the entire BOOM-based SoC is compiled into a simulator binary and test programs can be run on it just like on a real chip. In fact, BOOM's developers provide a random torture test suite and the GNU RISC-V tests, which were used to ensure that the modified core still executes instructions correctly in all corner cases.

In summary, the BOOM core in Chipyard provides an ideal research vehicle: it is a modern, parameterized RISC-V out-of-order processor that is easy to instrument and modify, supported by comprehensive simulation infrastructure. This lets us integrate the interlock-collapsing ALU and thoroughly test its impact on both correctness and performance in multiple sizes of representative cores. Furthermore, the results of simulating this ICALU-enabled core with the selected workloads provide a robust and realistic ILP speedup. Instead of the cycle-approximate simulations IBM did in the 1990s, this work aims to generate a more robust conclusion by leveraging all these tools that result in accurate and realistic measurements.

2.4. Conclusion

In the 1990s, IBM presented ICALU: an innovative solution that fused dependent arithmetic and logic operations using a three-operand ALU. It was designed with a clear philosophy: to provide a boost in processor performance regardless of the implementation. This meant ICALU did not require changes to the ISA and software, used standard CMOS gates and above all ensured that any increase in ILP translated directly into performance by not lengthening the delay of any path.

IBM carried out cycle-approximate simulations on SPEC92 workloads to prove its benefits. In those experiments, ICALU improved performance by 1.5%, 4% and 6% for 2-, 4- and 8-wide out-of-order cores, with similar numbers for in-order cores, showing that its benefits grew with machine width.

To revisit ICALU three decades later, this project used the RISC-V ecosystem, which, with its flexible cores, fast simulators and broad benchmark suites, provides an even more realistic and accurate procedure to assess whether ICALU can still deliver meaningful gains in modern processors.

3

Feasibility Study and Trace Analysis

After understanding the fundamentals of ICALU and the tools used across this work, this chapter reexamines ICALU and studies how collapsible pairs appear in modern workloads before any micro-architecture changes. The feasibility analysis aims to ensure the assumptions and constraints IBM made for ICALU still hold and selects the instructions an ICALU unit could execute. Subsequently, CoreMark, Embench and SPEC2017 integer are analyzed to obtain statistics that later guide micro-architecture choices and fulfill the first research goal of understanding where, how often and under which dependency patterns collapsible execution interlock opportunities arise in modern workloads.

Section 3.1 revisits ICALU's original assumptions and constraints to ensure they still hold and the design is transferable to a modern RISC-V setting. The same section defines the set of instructions that a future ICALU unit in a RISC-V core will execute. Section 3.2 describes how traces are gathered and processed, explaining the instrumentation and post-processing used to analyze the workloads. Section 3.3 presents empirical results across CoreMark, Embench and SPEC2017 integer, reporting collapse rates, producer-consumer distances and opcode patterns and summarizing the main takeaways that will guide later micro-architectural choices.

3.1. Feasibility Study

It is required to determine whether, in today's technology and environment, if it is still technically possible to implement ICALU and to obtain empirical evidence that points to a potential performance uplift. Therefore, these are the first required steps before committing engineering effort to deploy ICALU in a core.

With regard to technical feasibility, this section reexamines the historical IBM ICALU assumptions and constraints presented in Section 2.1. It also presents the mapping of chosen instructions onto the RV64GC subset of RISC-V to be executed by the future ICALU unit. Finally, it uses results from Section 3.3 to show where and how collapses are found in modern code and to confirm whether there is room to exploit these collapses in a potential ICALU-modified core.

3.1.1. IBM's Assumptions, Constraints and Conclusions in Modern-Day Machines

IBM presented the original ICALU proposal with several concrete assumptions and constraints. Readers unfamiliar with those preliminaries are encouraged to consult Section 2.1 first, since many arguments here derive from that foundation. This list walks through these assumptions and constraints that need to be revisited:

Constraints to revisit

- **CMOS book-set compatibility.** IBM constrained the design to use only standard-cell primitives, specifically 2-input XOR/XNOR, up to 8-way AND/OR/NAND/NOR and 3-4-way AND-OR/AND-NOR gates [1]. IBM claimed this was a "common book-set" [1, 23], but no direct evidence was

found that this is true in modern process nodes. Some open-source libraries (e.g., SkyWater SKY130) include multi-input and complex gates [24], whereas others, such as the ASAP7 7 nm predictive library, explicitly exclude AOI/OAI and AO/OA layouts [25]. This constraint cannot be revalidated; implications are discussed when revisiting research conclusions.

Assumptions to revisit

- **Ignoring rare 4-to-1 cases.** Some consumer-producer pairs expand into expressions that require four operands. The trace analysis in Section 3.3 should confirm whether this remains rare to justify not adding specialized logic for those cases.

Other constraints and assumptions did not require revisiting. For example, the constraint that ICALU be ISA-compliant and require no ISA or software changes stands and is not needed to be reexamined. What remains is to revisit research conclusions:

Research conclusions to revisit

- **Architecture compliance.** ICALU consists of two ALUs to ensure architectural correctness, one 2-1 and one 3-1. This remains possible in a modern core and back-ends have not evolved in a way that makes it nonviable [26]. These blocks can replace typical functional units with the extra required logic.
- **Cycle-time assumptions.** IBM assumed a conventional 2-1 ALU was a 64-bit CLA with four stages of delay, while a 3-1 ALU composed of a CSA stage followed by the same CLA requires five stages [3]. Because the "book-set" claim could not be confirmed and since a *stage* was defined by the slowest available cell in that set, the universal claim that a 3-1 ALU is exactly one stage longer than a 2-1 ALU cannot be guaranteed across **any** node. However this does not affect the idea that ICALU is one stage longer than traditional ALUs but rather implies that only certain nodes will achieve that 25% delay increase only. To retain the one-extra-stage claim across nodes, a full re-derivation of the boolean equations that form ICALU with a widely available cell set is needed. This is left for future work. For the purposes of this study and because it still largely applies, ICALU is assumed to add one extra stage of delay compared to a conventional ALU.

  With this assumption in mind, it is needed to revisit if that extra stage will not affect the critical path of modern cores. In this context it can be concluded that if ALUs are currently typically time-critical with little slack, new solutions would likely have appeared. The fact that ALU architectures have changed little since the early 1990s [27] motivates IBM's claim that the ALU is not part of the overall critical path and an additional stage should not to affect machine cycle time.
- **Area and power overheads.** If ICALU overheads were small in the 1990s, current transistor densities make them even less significant. The hardware overhead for a 3-1 design remains minimal. What remains to be seen is whether integrating ICALU into modern micro-architectures introduces substantial additional logic outside the ICALU slice.
- **Condition-code generation.** RISC-V does not have condition codes [10], so generating zero, carry and overflow flags in parallel without lengthening the data path is irrelevant and can be ignored.
- **Complementary nature of ICALU.** IBM originally proposed ICALU as an alternative to out-of-order machines. Modern high-performance processors are out of order, so ICALU should aim to increase ILP rather than replace OoO.

IBM also suggested that compiler optimizations may alter the distance between producer-consumer pairs. In other words, they proposed higher optimization levels can move these pairs farther apart in program order. This could reduce ICALU's potential benefit, since closer pairs are more likely to stall successors, while distant ones give the OoO engine more flexibility to fill slots. The trace analysis revisits this claim to evaluate its impact before concluding the feasibility study.

In summary, most of the original constraints, assumptions and conclusions remain valid today, except for the cell libraries used in the original design. The inability to confirm a universal cell set used by

ICALU leaves the claim that a 3-1 ALU with a CSA+CLA structure always requires one additional stage compared to a 2-1 CLA-based ALU across all nodes to be applicable to only the nodes that have this used cells. A complete re-derivation with a modern common cell library is necessary and left for future work if this claim wants to be re-established. For this study, ICALU is assumed to extend a conventional ALU by one stage and IBM's implementation is used. The feasibility study is pending on confirming whether 4-1 collapses remain rare, how compiler optimization affects pair distances and whether enough collapsibility opportunities exist to motivate the next engineering phases, which is finished in the conclusion of this chapter found in Section 3.4.

3.1.2. Feasibility in RISC-V

After confirming that ICALU's design can be translated today, it is time to map it into the RISC-V ISA, which means identifying what instructions it will execute. IBM's ICALU focused on collapsing pairs of arithmetic and logical instructions in the ESA/370 instruction set. RISC-V, like most ISAs, includes many of the same operations. It is also modular [10]. For this work, the RV64GC set is the reference because it covers instructions needed for Linux-capable machines and is the ISA supported by BOOM.

Table 3.1 lists the subset of instructions considered for potential ICALU collapsing. They represent arithmetic and logical operations similar to IBM's original categories. Each row gives a RISC-V mnemonic, a brief description, a register form and whether it could act as the first and/or second instruction in a collapsed pair. Load and store instructions appear only in the sense that their *address generation* is collapsible; the memory access itself never creates a back-to-back arithmetic operation.

Table 3.1: Candidate RISC-V instructions for ICALU collapse. The table enumerates integer and floating-point instructions selected for potential fusion by the Interlock Collapsing ALU. For each mnemonic, the operation semantics appear alongside the canonical operand form and the collapse position (first and/or second in the producer-consumer pair).

Mnemonic	Operation	Register Form	Position
ADD	Addition	`rd = rs1 + rs2`	1 and 2
ADDI	Addition with immediate	`rd = rs1 + imm`	1 and 2
SUB	Subtraction	`rd = rs1 - rs2`	1 and 2
AND	Bitwise AND	`rd = rs1 AND rs2`	1 and 2
ANDI	Bitwise AND with immediate	`rd = rs1 AND imm`	1 and 2
OR	Bitwise OR	`rd = rs1 OR rs2`	1 and 2
ORI	Bitwise OR with immediate	`rd = rs1 OR imm`	1 and 2
XOR	Bitwise XOR	`rd = rs1 XOR rs2`	1 and 2
XORI	Bitwise XOR with immediate	`rd = rs1 XOR imm`	1 and 2
LB	Load byte	`rd = M[rs1 + imm]`	2
LH	Load halfword	`rd = M[rs1 + imm]`	2
LW	Load word	`rd = M[rs1 + imm]`	2
LD	Load double	`rd = M[rs1 + imm]`	2
FLW	Load single-precision float	`rd = M[rs1 + imm]`	2
FLD	Load double-precision float	`rd = M[rs1 + imm]`	2
SB	Store byte	`M[rs1 + imm] = rs2`	2
SH	Store halfword	`M[rs1 + imm] = rs2`	2
SW	Store word	`M[rs1 + imm] = rs2`	2
SD	Store double	`M[rs1 + imm] = rs2`	2
FSW	Store single-precision float	`M[rs1 + imm] = rs2`	2
FSD	Store double-precision float	`M[rs1 + imm] = rs2`	2

Many of these instructions map directly to ESA/370's arithmetic and bitwise logic (add, subtract and/or/xor). In RISC-V, some more CISC-like operations such as "Load Positive Register" or "Load Complement Register" do not exist and are expressed as multiple instructions. RISC-V also includes `*W` variants (e.g., `ADDW`) that operate on the lower 32 bits and sign-extend the result; handling them may require sign-extending partial results and is left out of this work. CSR and atomic instructions were considered but assumed to have low performance impact relative to added micro-architecture complexity.

In conclusion, from an ISA perspective, classic arithmetic and bitwise operations behave almost the

same, so the original IBM design applies. What remains unknown is how often these collapses appear in modern workloads; this is resolved with trace analysis.

3.2. Trace and Statistics Collection

One research goal is to understand where, how often and under which dependency patterns collapsible execution could occur, information crucial for any future work on ICALU. This goal serves two addition purposes: complete the feasibility analysis in Section 3.1 by confirming that enough collapse opportunities exist and complete some unanswered revisits and guide the micro-architecture changes to BOOM in Chapter 4. This section explains the methodology used to fulfill these goals. At a high level, QEMU is modified to record, for each chosen workload, every dynamic instruction's translation block context. The resulting trace is then scanned for pairs that satisfy the criteria of Section 3.1.2.

3.2.1. Workloads and Compilation

Workloads Section 2.3.1 presents the benchmark suites used in this work: CoreMark, Embench and SPEC CPU2017 integer. In summary, CoreMark provides a compact baseline of core performance, Embench adds diversity through a broad set of small real programs and SPEC captures large-scale application behavior and remains the gold standard for core performance evaluation. Combined, these suites form a complementary strategy: CoreMark and Embench are small enough to run to completion in RTL simulation, allowing cycle-accurate quantification of ICALU's impact, while SPEC, evaluated through trace analysis, reveals whether the same opportunities extend to complex, real workloads when linked to the CoreMark and Embench results.

Building the binaries The three suites were cross-compiled with the RISC-V GCC toolchain at `-O2`, unless otherwise specified. The resulting binaries can run on Chipyard's functional simulators such as QEMU or Spike. CoreMark and Embench are built in bare-metal form so they can later run on a Verilator simulation of BOOM, which is not feasible for SPEC. In SPEC's case, execution is restricted to functional simulators. However, because QEMU executes system calls on the host rather than the target, they do not appear in instruction traces; the traces collected remain equivalent to target-level execution and are comparable across collapse analyses. CoreMark is compiled to run for 100 iterations, the default Chipyard setting.

3.2.2. QEMU Instrumentation

QEMU is the functional simulator chosen to generate traces because it executes RISC-V binaries at near-native speed, allowing a single SPEC CPU2017 benchmark to complete in tens of minutes. That performance is essential, as each SPEC workload retires billions of instructions. Section 2.3.2 describes QEMU in more detail and the motivation for using it.

Translation blocks

Before starting with how QEMU was modified, it is important to understand why it is so fast, as it will influence the instrumentation work. Internally, QEMU translates guest basic blocks into host micro-ops and stores the sequence in a *translation block* (TB) cache. A TB contains a straight-line region of code that ends at a control-flow instruction. In other words, these are groups of instructions that terminate when a branch or jump is encountered. Once translated, the same TB can be executed many times without incurring translation overhead, enabling QEMU's high speed. An illustration of this process, where a guest RISC-V block is translated into a host (e.g. x86) block ending in the same control-flow instruction, is shown in Figure 3.1.

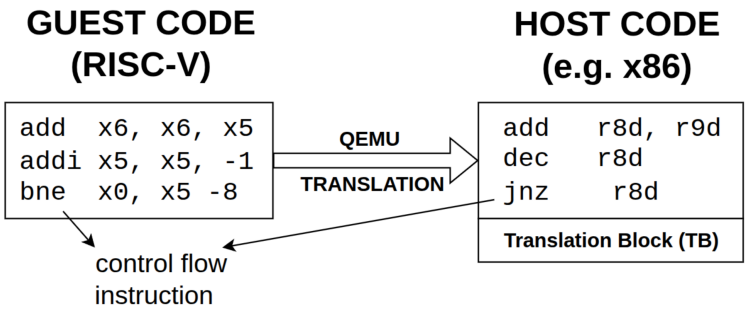


Figure 3.1: QEMU translates guest RISC-V basic blocks into host translation blocks (TBs). Both blocks end in a control-flow instruction. Once translated, the TB can be reused without additional overhead.

Trade-off between detail and turnaround time

A naive solution would log every dynamic instruction for offline processing. Other options would be to patch QEMU to process on the fly the statistics. Both approaches were ruled out because they degraded the workflow in three ways:

- **Disabling optimizations:** QEMU translates the guest code to host code once and then runs the TBs every time through fast paths (the control flow instructions jump to another TB, effectively removing QEMU's overhead). However, regardless of offline or online processing, it is required to disable this QEMU's fast translation path to log / process statistics. Tests showed that benchmarks that normally finish in minutes stretch to several hours for the case of *CoreMark* and to multiple days for the case of a single *SPEC* integer benchmark.
- **Processing Overhead:** However the main overhead is not about disabling the fast paths. Any processing that is done between instructions or TBs, introduces latency at every step of QEMU's pipeline. Although the slowdown per instruction may seem minor, preliminary results showed that scaling this cost across billions of operations quickly inflates runtime from a few hours to several weeks or even months, depending on the overhead.
- **Storage pressure:** On the other hand, when focusing on doing the analysis to an offline strategy, by logging every instruction to disk and processing the trace afterwards, causes a steep storage toll. Even aggressively compressed, full instruction logs can occupy multiple terabytes and impact performance heavily. This combination of vast data volumes and logging overhead makes the offline approach equally nonviable.

Therefore, it is more complicated than what it might originally seem to do such complex processing. Fortunately, initial experiments comparing inter-TB analysis (examining dependencies across TBs) with intra-TB analysis (dependencies only within TBs) on shorter benchmarks like CoreMark showed that intra-TB analysis captures around 90% of collapsible pairs compared to inter-TB analysis. This indicates data-dependent arithmetic chains rarely cross a branch or jump and most collapse opportunities already lie within a block.

These findings allow for a feasible approach to analyses the traces of the workloads as it is not needed to record the whole sequence of instructions to get good representative collapse statistics. On that basis, QEMU was instrumented to log, for every TB:

- the sequence of static instructions inside the block at translation time,
- a counter recording how many times the TB executed.

This approach preserves essential statistics information while keeping execution time and storage requirements manageable. Under the modified build, each SPEC integer benchmark completes in under two days and occupies a few megabytes. This shows how even when QEMU was instrumented to do the simple counter increase per TB executed, the overhead added for that task lengthens a benchmark execution from tenths of minutes to days. However, this is a manageable timeline and it was proceeded to patch QEMU with this focus. Statistics can be extracted later by running an offline script that uses as input the TB count information and TB content. This can always be rerun if any other parameter is

desired rather than repeating the costly simulation, also resulting in a much more flexible solution than online processing.

3.2.3. Trace Post-Processing

After QEMU writes TB information and counts, an offline Python script performs the following analysis, showed in Figure 3.2. In more detail, this is the post-processing pipeline applied once QEMU dumps TB traces and execution counts:

- **Scan instruction pairs** Each TB is scanned from first to last instruction. Starting with $N = 0$ (adjacent pairs), each instruction is checked against the one at offset $N + 1$. As can be seen in Figure 3.2, instruction 1 is compared to 2 for N = 0, 2 with 3 and so on.
- **Check collapsibility** For each comparison pair, the script checks whether (i) both instructions are eligible as producer and consumer (Table 3.1), (ii) the hazard rules are respected (Table 3.2) and (iii) neither instruction has already been marked as collapsed. If they are all met the pair is counted as a valid collapse.
- **Increase N and repeat** After scanning adjacent pairs ($N = 0$), the process is repeated with $N = 1, 2, \ldots, 7$. In other words, it starts scanning adjacent instructions, then those with a distance of 1, and so on, as depicted in Figure 3.2. This prioritizes closest pairs which have also been discussed to be most likely to stall and therefore obtain a performance uplift and aligns with how the collapse detection is done later in the the BOOM core modifications.

  This $N = 7$ limit is imposed because early experiments revealed unrealistic collapses at distances of tens of instructions, which would artificially inflate collapse rates without being exploitable in a real pipeline. This flow ensures that the analysis starts with the closest pairs which are those most likely to cause stalls and will be later prioritized in the BOOM design.
- **Weight and aggregate** When all potential combinations are compared, per-distance collapses are aggregated into TB-level statistics, then weighted by each TB's execution count and aggregated into benchmark-wide results as depicted at the bottom of Figure 3.2.

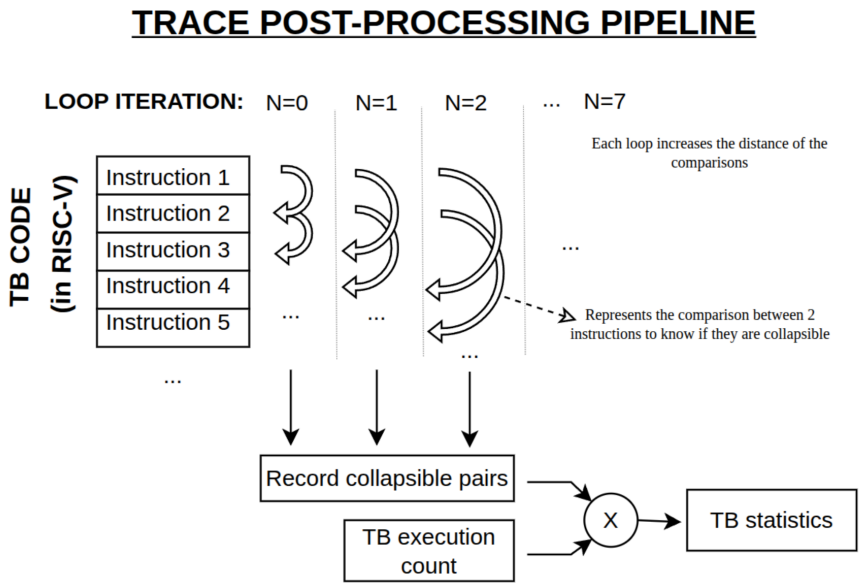


Figure 3.2: Post-processing pipeline. Each TB is scanned for collapsible pairs at increasing distances ($N = 0 \ldots 7$). Valid pairs are recorded, weighted by TB execution counts and aggregated into benchmark-level statistics.

The outcome is a single JSON file reporting the desired statistics. Because raw TB traces are small, additional metrics can be computed later by rerunning the Python script without a new QEMU run. These results in a flexible and manageable way of obtaining the collapse statistics required to do the trace analysis that will be later done.

Hazard rules and rule masking

When scanning if two instructions are collapsible, there a set of hazards that can occur that deem the collapse architecturally incorrect or in other words, result in a different program outcome than the correct one. The eight conditions in Table 3.2 describe every data dependency that could break a two-instruction collapse. Rules 1-4 are intrinsic to the definition of a safe three-operand execution and therefore stay enabled at all times in the trace analysis and micro-architecture collapse detection. However, there are some rules that can be avoided depending on the type of micro-architecture implemented.

For instance, rule 5 prevents a collapse when an instruction between the candidate pair reads the register that the consumer will later overwrite. Even though it might be able to be avoided with extra logic, in most cases this will result in architecturally incorrect results. Nevertheless, a processor that renames destinations removes that write-after-write hazard and therefore concern of architectural incorrectness that that collapse would have. Clearing the rule reveals collapses that would become illegal on a core without renaming. This happens also in rule 6 to 8 with renaming and reordering cores.

This can give a quick first-order view of how micro-architectural sophistication translates into extra ICALU opportunities and will be later explored for that purpose. The idea is to give an idea of how collapse opportunities change if a potential reader decides to implement in a simple in order core or a more sophisticated out of order core.

Rule	Condition	Snippet	Explanation	Maskability

1	Avoid circular chains	`add s0,s1,s2`
		`add s2,s0,s3`
		`add s4,s2,s5`
			Instruction depends on the previous one; if collapsed it creates a circular chain of dependencies	mandatory

2	Store must depend on address not data	`add s0,s1,s2` `lw s3,1(s0)`	`addrgen = 1 + s1 + s2` `s3` cannot be the collapse target	mandatory
3	Not a 4-1 operation	`add s0,s1,s2` `add s3,s0,s0`	`s3 = s1 + s2 + s1 + s2` (see Section 2.1)	mandatory

4	$\texttt{rd}_{\texttt{producer}}$ not redefined before consumer	`add s0,s1,s2`
		`add s0,s3,s4`
		`add s5,s0,s6`
			Collapse is with the second and third instruction	mandatory

5	$\texttt{rd}_{\texttt{consumer}}$ not read in between (WAW)	`add s0,s1,s2`
		`add s5,s7,s4`
		`add s5,s0,s6`
			Without renaming, collapse can be architecturally wrong	maskable (renaming)

6	$\texttt{rs1/rs2}_{\texttt{producer}}$ unchanged (WAR)	`add s0,s1,s2`
		`add s1,s3,s4`
		`add s5,s0,s6`
			Without renaming, collapse can be architecturally wrong	maskable (renaming)

7	$\texttt{rs1/rs2}_{\texttt{consumer}}$ constant after in between	`add s0,s1,s2`
		`add s4,s5,s6`
		`add s3,s0,s4`
			Collapse not possible if consumer is not executed after in between	maskable (OoO reorder)

8	$\texttt{rd}_{\texttt{consumer}}$ not used in between	`add s0,s1,s2`
		`add s1,s5,s4`
		`add s5,s0,s6`
			Collapse not possible if consumer is not executed after in between	maskable (OoO reorder)

Table 3.2: Hazard rules used during trace post-processing. Operands follow `opcode rd, rs1, rs2`. Producer is the first instruction of the pair; consumer is the second. Rules 5-8 can be masked when the micro-architecture provides register renaming or a reorder buffer; others are always enforced.

After defining the engineering needed to analyze modern workloads for collapsible candidates, Section 3.3 presents results that can guide future ICALU design and answer where collapses appear in current workloads.

3.3. Trace Analysis Results

Understanding the collapsibility opportunities an Interlock Collapsing ALU could fuse is the first relevant outcome of this study. Section 3.2 detailed trace generation and statistics collection; this section focuses on what the numbers reveal. The collapse metrics collected here serve three goals: identify where collapses map in real workloads, provide numerical evidence to judge ICALU's feasibility from Section 3.1 and supply design hints for efficient micro-architectural implementation in Chapter 4.

3.3.1. Metrics and Analysis Configurations

Before presenting the numbers this section reviews the experimental ground rules chosen for this specific trace analysis. The description that follows recaps the metrics specifically chosen to extract from the workloads and the different analyzed scenarios that appear in the results section and result in the conclusions.

Workloads chosen

All measurements are based on the three benchmark suites introduced in Section 3.2.1: CoreMark, Embench and SPEC CPU2017 int. CoreMark provides a compact measure of core performance, while Embench, being also compact, adds variety of workload styles through a broader set of small real programs. SPEC CPU2017 captures large-scale application behavior and remains the gold standard for core performance evaluation. CoreMark and Embench, due to its smaller size, later will be used to quantify how these opportunities translate into ILP improvements by modifying the BOOM core. Although CoreMark and Embench lack the full complexity of SPEC, their results interpreted alongside the SPEC trace analysis results that will be showed here ensure that the conclusions reflect not only small embedded kernels but also the broader behavior of modern, compute-intensive workloads.

In the following results, Embench is treated as a single benchmark due to the small size of each benchmarks that forms this suite. Some of these benchmarks compromise of a handful of basic blocks meaning analyzing them individually cannot give representative results. Therefore they are shown together and represent the average behavior of a wide variety of kernels. Moreover, CoreMark is run in the provided configuration in Chipyard, which is 100 iterations.

Metrics

Three headline metrics summarize each run; the full numerical dumps reside in Appendix A, while the body of the chapter shows only the tables relevant for each discussion.

- **Collapse rate**: percentage of collapses found compared to the amount of instructions the benchmark executes in a full run (collapses found / total instructions).
- **Distance histogram**: distribution of the gap, in instructions, between the producer and its consumer. A value of 0 marks two adjacent operations; a value of 1 means exactly one instruction sits between them and so on.
- **Mnemonic-pair frequency**: share of collapses over the total amount of collapses that a specific opcode combination has. Pairs are based on Table 3.1 but grouped on the following groups: *add* for all additions and subtractions, *logic* for all and or and xor operations and separate buckets for all loads and stores.

These three views jointly expose how many collapses opportunities exist, how far they are between each other and which operations are collapsed more often.

Analysis configuration

The trace data set is evaluated under one reference setup. This uses an optimization level `O2` for the benchmark compilation. All mandatory hazard constraints in Table 3.2 stay active except those whose conflicts are already solved by an out-of-order engine with register renaming, which is precisely the environment targeted by the SonicBOOM prototype. In summary, this means that when the trace analysis checks for collapses, it does not apply certain dependency rules (WAW, WAR..), leading to more collapses being architecturally legal. For further information in regards to what these hazard rules mean visit Section 3.2.3.

Additional configurations

As defined by the research goals, the main goal of this part is to properly understand how these collapses map into the current workloads. Therefore, it is not only about obtaining the metrics mentioned before but also understand the questions that have been asked in the revisit of IBM's work done in Section 3.1.1. Hence, asides from running the trace analysis for the aforementioned configuration, additional sweeps have been done in order to probe how software choices or micro-architectural decisions affect collapse opportunities. In more detail, the additional sweeps and their respective motivation are the following:

- **Effect of compiler optimization level.** Every benchmark is re-compiled at `O0`, `O1` and `O3`. The goal is to revisit ICALU's original suggestion in Section 3.1.1 that compilers often attempt to separate dependent instructions [2], changing the distance between the collapse pairs. This is relevant as it could affect the performance uplifts of ICALU in a microprocessor implementation, as closer execution interlocks are most likely to stall the back-end.
- **Relevance of 4-1 equivalent collapses.** As discussed in Section 2.1.2 certain collapses convert into 4-1 operations. These can be solved with specialized hardware, however, IBM claimed the frequency of these collapses is extremely low, diminishing the benefits of investing extra logic to solve them. This test re-examines this unanswered claim of Section 3.1.1 on wether workload evolution has not changed this landscape.
- **Filtering zero operand additions and subtractions.** Many modern CPUs already fuse sequences such as `add xN, xM, 0` or `sub xN, xM, 0` into a single move instruction using macro-op fusion. This sweep removes those trivial cases from the results so that the remaining collapse opportunities come *only* from true ICALU collapses. If the collapse rate drops sharply, much of the benefit could be obtained with a cheaper macro fusion block rather than a more complex ICALU incorporation. If it remains high, ICALU adds unique value and justifies the extra hardware and engineering effort.
- **Collapse opportunities in *in-order* or *no-renaming* micro-architectures.** As explained in detail in Section 3.2.3, some collapses might deem architecturally wrong if when executed, there are dependencies of the producer or consumer with instructions in between the pair. Some techniques like out-of-order execution and renaming can turn this illegal collapses architecturally possible. This study focuses on implementing ICALU in a OoO core and therefore some of these hazard rules, which can be found in Table 3.2, where not enforced in the main analysis process. However, this does not show the behavior for smaller super-scalar in-order cores. Therefore, in this comparison, stricter dependency rules are enabled when the trace analyzer finds a collapse with the aim of understanding how much micro-architectural decisions reduce the frequency of collapsible pairs.

3.3.2. Main Results

After stating the trace analysis pipeline in Section 3.2 and discuss the metrics and configurations in Section 3.3.1, the following section lists the results of the trace analysis. For each metric, its results are presented and discussed, shedding light into where, how frequently and under which dependency patterns collapsible execution interlock opportunities arise in modern workloads. The full results are found in the appendix Section A.1, where additional values that are not discussed are present. The main difference is each Embench result is shown and detailed distance histogram of all benchmarks too.

Collapse rate. Table 3.3 shows that every workload presents a meaningful pool of producer-consumer pairs. Measured across the whole suite of benchmarks, the unweighted mean of collapse rate sits at 8.9 %, while weighting by dynamic instruction count lowers the figure only slightly to 7.7 %. This solves one of the most important questions presented in this work in a clear matter: there are collapse opportunities in modern workloads for an ICALU execution unit to execute.

Furthermore, no single application area dominates: some such as *x264* reaches the top with more than 15.6 % yet others such as *exchange2*, barely clears 1 %. It can be observed that collapse potential is widespread rather than workload specific. This can be clearly proven when comparing benchmark *leela* and *exchange2*. They have a collapse rate of 11.7 % and 1.4 % respectively despite both being labelled AI benchmarks. This might indicate that ICALU success is not tied to specific workloads. However, this is a small example meaning this statement is more of a hint than a robust conclusion.

Table 3.3: CoreMark, Embench (suite aggregated) and SPEC CPU2017 int with application area (left) and share of collapsible pairs vs. executed instructions (right).

Benchmark	Application Area	Collapse rate [%]	Instr. executed
CoreMark	Embedded core benchmark	11.7	33.2 M
Embench	Embedded suite benchmark	8.8	46.1 M
SPEC CPU2017 int			
600.perlbench	Perl interpreter	5.3	1.29 B
602.gcc	GNU C compiler	5.9	0.05 B
605.mcf	Route planning	4.3	0.88 B
620.omnetpp	Discrete event simulation	8.0	0.43 B
623.xalancbmk	XML to HTML conversion via XSLT	6.3	0.70 B
625.x264	Video compression	15.6	0.49 B
631.deepsjeng	AI: alpha–beta tree search (Chess)	10.9	1.40 B
641.leela	AI: Monte Carlo tree search (Go)	14.3	1.43 B
648.exchange2	AI: recursive solution generator	1.6	2.10 B
657.xz	General data compression	13.4	0.43 B
Average unweighted †		**8.9**	
Average weighted †		**7.7**	

†Averages treat Embench as a single benchmark, using its average weighted collapse rate.

Distance of collapses. The distance profile in Table 3.4 shows the average instruction distance of the collapses. More than 52% of collapses occur when the two instructions sit next to each other and a further fifth appear with only one intervening instruction. Once the gap starts increasing the opportunity falls off rapidly; beyond a separation of four the share is negligible (<2%).

Short producer-consumer distances are critical because the closer a pair is, the higher the chances are to stall the consumer and other instructions in the back-end if the out-of-order engine cannot feed the execution units. The results shown in Table 3.4 confirms that the vast majority of collapsible pairs occur with a separation of at most three instructions maximizing the potential cycle savings and also helps guide micro-architecture choices made in the ICALU implementation in Chapter 4. When collapse detection is limited to a maximum of 3 instructions in between pairs, it is possible to capture, on average, 92.34% of the collapse opportunities.

Table 3.4: Distribution of instruction distance between the producer and the consumer in the collapse pair, expressed as percentage of all collapses and averaged across all benchmarks. A distance of zero means the two operations sit back to back in program order, higher distances imply additional instructions in between. Detailed information per benchmark in Table A.2

	Dist 0	**Dist 1**	**Dist 2**	**Dist 3**	**Dist 4**	**Dist 5**	**Dist 6**	**Dist 7**
Average share of collapses	52.24%	19.93%	11.72%	8.42%	4.29%	1.59%	0.95%	0.91%

Mnemonic-pair frequency. Figure 3.3 breaks down the eight most frequent opcode combinations. A clear trend appears: the typical pair marries an arithmetic operation (all *add* or *sub* instructions) with the address calculation of a following load or with another arithmetic pair. Logical operations surface much less often but are non-existent in a logic + memory pattern. This distribution suggests the relevance of incorporating the address generation in an ICALU slice in order to exploit interlock collapsing pairs. It also shows that removing logical operations keeps around 80% of the collapses.

3.3.3. Additional Results

The remaining results revisits the claims and assumptions from IBM presented in Section 2.1 and revisited in Section 3.1.1 that resulted in certain unanswered questions together with other questions presented. Each scenario discussed in this section has their motivation and explanation listed in Section 3.3.1, but each section will briefly remind it. If not mentioned, both the compiler optimization level and trace analyzer settings are set to the same values as Section 3.3.2

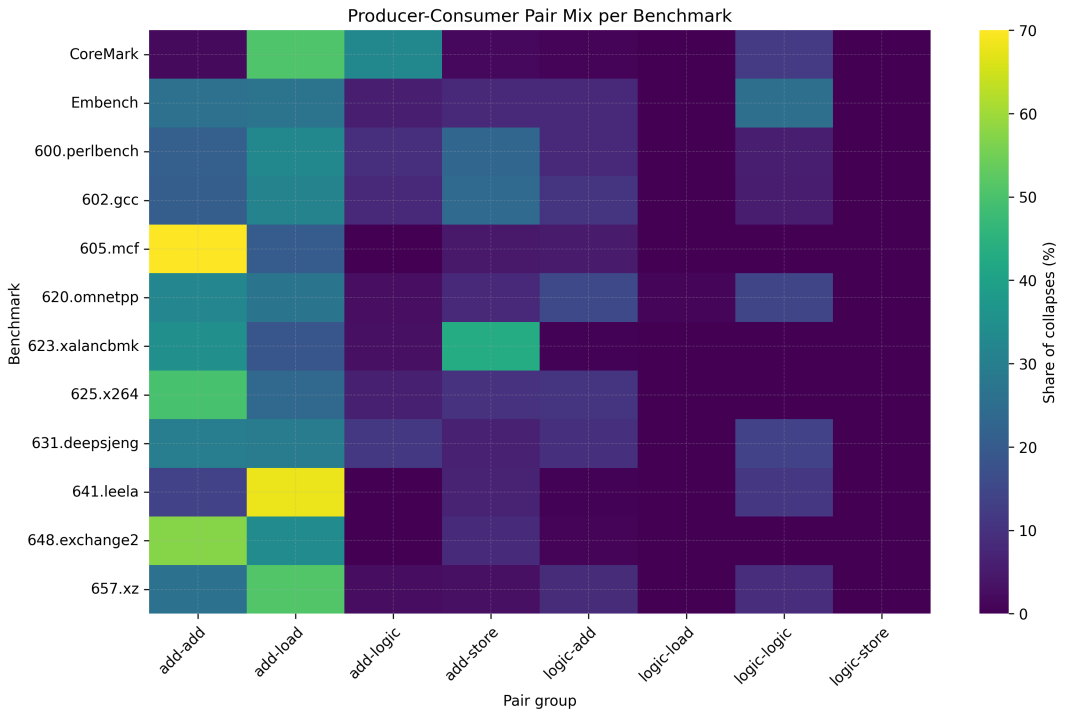


Figure 3.3: Heatmap of the frequency of producer-consumer pairs over the total amount of collapses per benchmark. On the Y axis, each benchmark tested. On the X axis, the pairs based on the grouping defined in Chapter 3.3.1. Each slot represents how often, over the total collapses, each pair group has been found.

Effect of compiler optimization level Table 3.5 shows the collapse rate and distance distribution for four compiler optimization levels. The overall collapse rate remains almost constant across `-O0` to `-O3`, with differences below 0.5%. The main change is in the distance breakdown: lower optimization levels have a higher share of back-to-back pairs (Dist 0), while higher levels shift more collapses into Dist 1 and Dist 2. At `-O0`, over 80% of collapses are adjacent, whereas at `-O3` this drops below 50%. This shift is consistent across all benchmarks and reflects compiler scheduling moving producer and consumer instructions further apart without removing the dependency. This confirms IBM's claims that compilers try to push instructions further in more speed oriented optimization levels to reduce pipeline stalls.

Table 3.5: Collapse density and pair distance across three compiler optimization levels. The four distance columns show the share of all detected collapses whose consumer follows the producer after 0, 1, 2 or 3 intervening instructions, respectively. A distance of 0 therefore marks back-to-back instructions, while higher distances indicate additional operations between producer and consumer.

Compiler level	Collapse rate	Dist 0	Dist 1	Dist 2	Dist 3
`-O0` (Minimum optimization)	7.3%	80.4%	15.1%	3.8%	2.7%
`-O1` (Restricted optimization)	7.5%	75.2%	17.2%	5.8%	3.7%
`-O2` (High optimization)	7.7%	62.2%	24.4%	5.3%	4.2%
`-O3` (Maximum optimization)	7.7%	46.0%	32.4%	12.3%	4.3%

Relevance of 4-1 equivalent collapses. Enabling the identification of collapses that result in 4-1 sequences does not shift the collapse rate in Table 3.6. Matter of fact, it was found that in most benchmarks there were less than 10 of these collapses across the billions of the instructions executed. The gain is statistically irrelevant and therefore supports IBM's original choice of ignoring this rare motif. Specialized hardware to handle this cases would offer no returns BOOM's modifications.

Filtering zero operand additions and subtractions. Removing zero-operand adds and subs operations inside the pair cuts the opportunity space to 6.9 % in Table 3.6. These would be pairs that

the decoding unit could fold through traditional macro-op fusion. In other words, roughly one point of the raw collapse pool can already be harvested by existing macro-op fusion techniques, but the bulk still demands a genuine three-operand unit. Therefore, this ICALU isolation pairs further motivates the usage of a dedicated ICALU that can exploit these collapse opportunities and these collapses found cannot be solved other simpler macro-op techniques.

Opportunities in in-order and no-renaming micro-architectures. Re-enabling all write-after-read and write-after-write rules while forbidding the out-of-order mask lowers the collapse rate to 6.7 % in Table 3.6. As previously explained, this scenario enforces stricter dependency rules in between the pairs and reduces the amount of collapses that are architecturally correct. This drop quantifies how many collapse opportunities drop in the scenario of deploying ICALU in cores with these micro-architecture features, as some collapses might deem architecturally incorrect. In conclusion, for simpler cores, the available collapse opportunities are slightly smaller, yet still non-trivial.

Table 3.6: Average collapse rate across all benchmarks under trace-analysis configurations (Section 3.3.1). Default references the configuration in Section 3.3.2.

Trace-analysis scenario	Avg. collapse rate (%)
Default rules	7.7 %
enabling 4-to-1 merges	7.7 %
Removing zero-operand add/sub	6.9 %
In order / no renaming scenario	6.7 %

3.4. Conclusion

The project started with a feasibility study that concluded that most of IBM's design assumptions and constraints hold true and that, after 30 years, the design is still applicable in modern cores. However, one of the constraints that limited the design to use a "common book-set" found across all process nodes could not be revalidated. These standard cells were not confirmed to be widely available in modern process nodes and a full re-derivation of ICALU's Boolean equations with widely available cells is needed to ensure that the claim of ICALU extending a traditional ALU formed by a 4-stage CLA to a 5-stage CLA+CSA, regardless of the fabrication process, is still true.

The study proceeded with a trace analysis of CoreMark, Embench and the SPEC2017 integer benchmarks. This analysis confirmed that collapsible ICALU pairs are a regular feature of modern workloads, appearing about one every eleven instructions or, in other words, at an average rate of 7.7%. Furthermore, it showed no evidence of pairs that convert into equivalent 4-1 operations, removing the need to have special hardware to handle these rare collapsed pairs. It also showed how compiler optimization levels increase the distance between instruction pairs. This concluded the aforementioned feasibility study by showing that there are enough collapse opportunities to commit engineering effort to modifying the BOOM core and by addressing assumptions and claims previously left unanswered.

Trace analysis also showed additional relevant information for the micro-architecture changes that later took place. It was observed that half of the collapsible pairs were adjacent in the instruction stream and more than 92% were separated by three or fewer instructions. Opcode mix was found to be diverse, with *add-add* and *add-load* combinations each accounting for 30% of the total pairs found. It was also found that removing logical opcodes from the mix keeps 80% of the collapse pairs, demonstrating that most of the operations are addition, subtraction or address generation.

Additional results showed that, by removing zero-operand add/sub instructions, the collapse rate stays constant. This means most of the collapse pairs found cannot be executed with simpler macro-op fusion techniques and motivates incorporating a specific ICALU unit to exploit them. Finally, the results showed that if a processor does not have register renaming and has an in-order architecture, collapse opportunities drop by 20%, shedding light on future work in simpler in-order cores.

4

ICALU Integration into the BOOM Core

After confirming it is feasible to implement ICALU in modern cores and using the information on how and in which shape collapse pairs appear in modern workloads, this chapter describes the engineering work done to implement ICALU in the BOOM core. This design process serves two purposes: evaluate the modified design through simulation to see whether there are ILP improvements when introducing ICALU and identify the minimal trade-offs required to implement ICALU in any modern-day processor, especially with regard to timing, to ensure performance uplifts are not lost to increases in machine cycle time regardless of the micro-architecture.

To fulfill this goal, the chapter is structured as follows: Section 4.1 establishes the goals and assumptions the design will follow, focusing on fulfilling the research goals. These ground rules set the stage for Section 4.2, which divides the implementation into 2 blocks and then explores the best potential solutions in the context of the aforementioned goals. After the exploration converges, Section 4.3 explains the details on how BOOM was modified, first at a high level and then with a deep dive. Section 4.4 presents the conclusions of this design process.

4.1. Goals and Assumptions

Before diving into the details of how the BOOM core was modified, this section presents the goals and assumptions of the implementation. Same as IBM's goal with the ICALU design, this integration in BOOM focuses on avoiding increases in the delay of any path. The philosophy established sought that any increase in ILP should not come at the cost of degrading the machine frequency regardless of the implementation (in other words, regardless of the available slack). Furthermore this design goals aim to keep the capture and execution rate of collapsible pairs as high as possible.

These design goals help identify the minimal non-negotiable trade-offs of incorporating ICALU in *any* modern-day processor and, at the same time, the realistic potential ILP improvements.

This does not mean these are the optimal choices for every design or even for BOOM. Because this is a *"Study of Interlock Collapsing ALUs"* and not *"How to Implement ICALU in BOOM"*, the aim of this implementation is to understand the non-negotiable timing trade-offs of incorporating ICALU, not to prescribe the best design choices for this or any core. Whether a specific processor or BOOM might avoid these trade-offs or be better served by different choices, is outside the scope of the following design process.

With those considerations in mind and after establishing the design philosophy, the main specific design goals in priority order are:

- **Maintain the delay of all paths.** As also defined by IBM in Section 2.1, ICALU and any required micro-architecture changes should avoid increasing any path's delay to avoid increasing machine cycle time. therefore, even if some paths have enough slack to absorb the added delay, the design goal will be to not increase the delay of any path and to isolate paths that *must* be extended, regardless of the slack available or the micro-architecture details.

- **No extra latency.** The pipeline depth cannot grow. Together with the previous point: ICALU should target increases in ILP that translate directly to performance uplifts and the goal is on uncovering any universal performance trade-off that affects this.
- **High capture rate.** Collapse logic should aim to catch as many pairs as possible to exploit the ICALU unit and reveal the maximum potential benefit.
- **Small area footprint.** Area overhead must remain as small as possible, provided it does not degrade the previous points.

Finally, the only assumption this study makes was introduced in Section 3.1.1. It is assumed that ICALU's design extends the delay of a traditional 2-1 ALU, formed by a 4-stage CLA, by adding a 3-1 ALU formed by a 5-stage CLA+CSA structure. This could not be proven to be universal across *all* modern process nodes. It is left for future work to revisit the 3-1 ALU's architecture to confirm it can be considered to have one more stage of delay across any process node.

4.2. Design Exploration

IBM's papers presented in Section 2.1 focus mainly on the execution-unit slice of ICALU and do not present how it was incorporated in any micro-architecture. Furthermore, processors have evolved compared to the 1990s. Therefore, as a first step, it is necessary to explore the micro-architecture changes required to incorporate ICALU in a modern processor, strictly following the design goals in Section 4.1.

At a high level, the line of attack is split into two blocks. The first is a *collapse detector* to find eligible producer-consumer pairs so the downstream pipeline can treat the duo as one equivalent operation. The second block is *collapse execution*, which completes the fused arithmetic in a single cycle. This includes the ICALU execution unit and the extra required back-end logic such as adjusting dispatch, issue, register read, write-back and commit so they can handle these collapsed instructions/micro-ops.

Section 2.3.3 describes in detail the micro-architecture of the BOOM out-of-order core used in this work. Readers unfamiliar with it and with modern out-of-order architecture in general, are encouraged to revisit that section, as the following assumes familiarity with these concepts.

At a high level and as depicted in Figure 4.1, the architecture is divided by two blocks. The front-end, formed by the fetch stage, which prepares and supplies instructions to the rest of the core and the back-end, compromising the decode to write-back stages, that is in charge of executing these instructions. In more detail, each stage can be summarized as follows:

- **Fetch:** Obtains instructions and performs branch prediction to feed the back-end.
- **Decode:** Expands fetched instructions into micro-ops the back-end can execute.
- **Rename:** Maps logical registers to physical ones, eliminating false dependencies.
- **Dispatch:** Places renamed micro-ops into the appropriate issue queue and ROB.
- **Issue:** Selects ready micro-ops from the queues for execution when operands are available.
- **Register Read:** Provides source operands from the register file.
- **Execute:** Executes the micro-op via ALU, branch and address-generation units.
- **LSU:** Manages loads and stores from the address generator.
- **Write-back:** Writes results to the register file and wakes up micro-ops in the issue queue.
- **Commit:** The ROB retires executed micro-ops and updates architectural state once speculation resolves.

4.2.1. Collapse Detection

The collapse detector, the unit capable of identifying which pairs can be folded into a three-operand operation, has to sit where it can catch the largest number of producer-consumer pairs, as required by Section 4.1. The collapse detection circuit is mainly simple comparators that produce a result. Hence, it is only a few logic levels deep and can easily be run in parallel and therefore its layout is the same no matter its location in the pipeline. This is further discussed and proven in the design description of

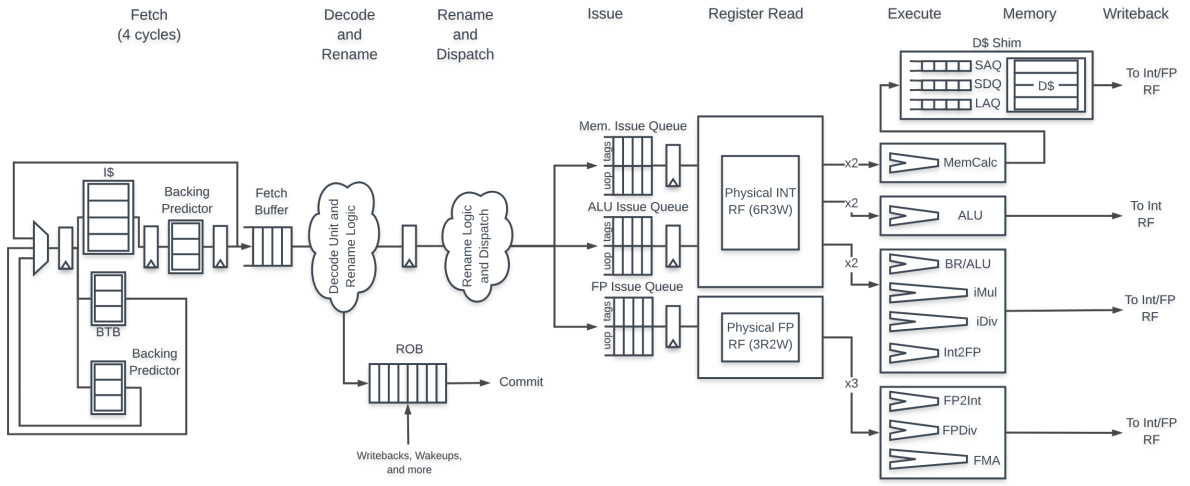


Figure 4.1: Simplified micro-architectural diagram of the BOOM core (third-generation SonicBOOM). The front-end (left) fetches and predicts instructions, while the back-end (right) handles out-of-order execution. BOOM employs multiple issue queues (Int, Mem, FP) feeding functional units and a unified physical register file with a reorder buffer for commit. Reproduced from [19].

Section 4.3. With that in mind, the real question and subsequently exploration is *where* in the pipeline should this logic be hosted to obtain the highest capture rate of collapsible pairs.

The BOOM pipeline shown in Figure 4.1 indicates that the issue stage is the last point where a micro-op can still be detected as collapsible, after that, the micro-op is already being executed and it is too late to be collapsed. Hence, any stage that includes or precedes the issue queue is therefore a potential candidate. This insight reduces the options to three placement groups that will be presented, each with its own strengths and weaknesses.

Detection at L1 cache

Principle: During the idle window where the L1 cache is waiting for a missed request, the previous refilled line is scanned instruction by instruction to look for producer-consumer pairs. A temporary decoder locates instruction boundaries, expands any compressed 16 bit instructions to their 32 bit equivalents and compares each instruction against the following with the later presented collapse detection logic. If a legal collapse is detected, the pair is tagged with metadata / merged before the line is written into the L1 array again.

Strengths

a) ***Long look ahead window*.** The scanner can see the whole 64 byte block, meaning even distant pairs can be detected.

b) ***One time calculation***. Re-scanning is only required when the line is evicted and later reloaded, which keeps dynamic energy low.

Weaknesses

a) ***Area overhead***. The RV64GC instruction set mixes 16 bit and 32 bit opcodes, the scanner has to additionally first detect instruction boundaries and expand compressed instructions. Furthermore, regardless of tagging or merging being used, extra storage is required for metadata that will later treat the pair as collapses. Doing a rough and simplistic estimation, in the case of tagging, assuming at least 4 bits per collapsed instruction are required (1 flag, 3 offset bits to locate produces) and with the best case collapse frequency of roughly 15% reported in Section 3.3, the storage cost could be simplistically assumed to be $\frac{0.15 \times 4}{32} \approx 1.9\%$ of the cache capacity.

   Another implementation would be to physically merge the pair, leveraging the fact that producer and consumer instructions have a common register in order to store some sort of collapse identifier / information. However this implementation quickly becomes nonviable because instructions

from the compressed instruction set do not provide spare room for extra encoding, all 16 bits are being used. Furthermore, in this scenario the design would need to restore the original instructions when a branch target lands inside a pair / a branch diverges flow inside a pair. Therefore, a merging scenario would require metadata that also, at a minimum, uses similar capacity as the tagging scenario proposed.

However, these are oversimplified calculations, these values should not be taken into account with a deeper design description. This simplistic explanation wants to bring a main take-away: supporting this collapse detection location would occupy a noticeable share of the cache array (at least of 2%).

b) ***Only static code collapses***. The method only observes instructions inside one cache line, limiting collapses to the ones that can be found in static code.

c) ***No collapses between lines***. This method misses collapses across cache lines.

Detection in the decode-rename / rename-dispatch stages

Principle: The front-end delivers a batch of micro operations every cycle. Collapse detection can be inserted on this path so that each incoming group is examined for producer-consumer pairs. Instructions remain in their original order and once a legal pair is found the consumer is annotated / merged so later stages recognize the collapse. However, a small look-back of the previous cycle instructions is required to capture all the collapses as even adjacent pairs, in program order, if they travel through different stages.

Strengths

a) ***Dynamic code collapses.*** This implementation can catch collapses across branches, as it scans the speculative code being feed through the front-end.

b) ***Simpler scanning logic***. Instructions have been expanded to 32 bits at this stage. Furthermore, RISC-V keeps the bits location of the opcodes and registers fixed for the 32 bit instruction, which results in simpler logic for collapse detection.

Weaknesses

a) ***Limited window length***. The maximum search distance equals the sum of the decode width (2, 3, 4, depending on the core size selected) multiplied by 2, as it looks back to the previous stage. Pairs that span more than this range are missed.

b) ***Non-profitable collapse in an idle back-end.*** In order to understand this complex point is best to start with an example. Consider a producer-consumer pair that is detected in that look-back window (interwindow collapse) and each micro-op travels in different stages. Imagine, in the baseline design, the producer reaches the issue queue and, because the execution units are free and its operands are ready, is issued in the very next cycle without stalling. The consumer would arrive one cycle later when the producer its already sent to execute. In an ICALU core with this proposed design, this collapse now requires holding the producer back for an extra cycle so that both micro-ops can be fused and issued together (forwarding is not possible as the micro-op needs to go through decode and rename). In the baseline design, the 2 instructions are executed in 2 cycles, one for each cycle. In the ICALU design the same result happens, but because the first cycle the producer is not executed and in the second cycle both collapsed micro-op are executed in parallel in the 3-1 ALU, the throughput is the same as baseline. This results in no performance improvement over the baseline in this specific scenario.

The conclusion is that if the producer of a pair that travels between different cycles never experiences a stall in the baseline design, the ILP improvement is not exploited, reducing the maximum potential performance improvement of ICALU.

Detection inside the issue queues

Principle Micro operations that wait for operands reside in separate integer and memory issue queues. While they remain in those structures the collapse detector compares each entry against the others and

flags / merges a pair when the destination register of one instruction matches a source register of another. In order to not violate timing, the micro-op needs to stay for at least a cycle. The issuing logic occurs at the beginning of this stage and therefore adding the collapsing detection logic before the issuing logic would add too much delay.

Strengths

a) ***Long look ahead window***. The effective window equals the queue depth rather than the fetch width, so increasing the queue immediately enlarges the search space for potential pairs.

b) ***Dynamic code collapses.*** For the same reasons in the previous proposal.

Weaknesses

a) ***One-cycle residency requirement***. A producer that enters the queue does it at the end of the cycle and it is decided if it issues in the beginning of following cycle. Similar to the previous point, this entry escapes detection / needs to wait to exploit the collapse pair, which reduces the number of collapses observed in practice. However, unlike the previous example, this occurs in all collapses and not in the interwindow ones.

b) ***Long hazard detection logic***. Even though instructions in the issue queues are age ordered, integer and memory operations occupy different structures. The collapse detector cannot see the exact sequence of instructions that lies between a candidate producer and consumer. Evaluating the dependency rules in Table 3.2 would then require to evaluate them for all the other instructions and across both queues to ensure program execution compliance, causing the comparator count to grow substantially. Moreover, the increased instructions in between the pair increase the chance of these hazards to be triggered and miss potential opportunities.

Decision

Table 4.1: Summary of pros and cons of each detector placement.

	L1 cache	**Decode-rename**	**Issue queues**
Pros	Long window One-time check	Dynamic collapses Simplest logic	Long window Dynamic collapses
Cons	Area overhead Static scope Line bound	Short window Some stalls missed	1-cycle stay Split queues

On Table 4.1 a summary of the pros and cons can be observed with the first conclusion being that no single placement meets all goals of Section 4.1. However, these pros and cons points do not take into account the results of the trace analysis. Thanks to this work shown in Section 3.3 it can be seen that more than 90 % of collapses occur with three or fewer intervening instructions, which limits the benefit of very long look-ahead windows. Furthermore, thanks to the that same trace analysis it was seen that collapse opportunities rarely cross control flow instructions, proving the pros and cons in regards to static and dynamic collapses to be low.

Table 4.2: Summary of pros and cons of each detector placement after applying the results of the trace analysis.

	L1 cache	**Decode-rename**	**Issue queues**
Pros	One-time check	Simplest logic	
Cons	Area overhead Line bound	Some stalls missed	1-cycle stay Split queues

When removing the aforementioned points found negligible through the trace analysis, Table 4.2 shows the final pros and cons table. In this final comparison, the issue-queue option does not bring a benefit

but suffers from a one-cycle residency requirement and more complex logic due to the split integer / memory structures and its extra search depth. Subsequently, the issue one option can be discarded.

The choice eventually narrows to either placing the detector in the L1 cache or in the decode-rename stage, which have a clear trade-off situation. The decode/rename implementation can be observed to have less collapse opportunities exploited due to situations where collapses that travel in different cycles and would not stall. The cache-resident design, however, demands a sizable wider tag array; even under optimistic assumptions the cache footprint could rise noticeably (> 2%). Based on the goals set in Section 4.1 both the other points are not a priority. The final trade-off is a cache that has uses a noticeable array allocated for storing collapse metadata vs a decode-rename implementation that can exploit less ILP improvements from certain collapses.

In this final trade-off of cache size vs collapse opportunities and aligning with the goals in Section 4.1, it has been chosen to implement the collapse detection logic in the decode/rename stage. The rationale behind has been the possibility of losing performance due to the reduced cache size vs reducing the potential performance uplift. This second option follows best the philosophy of trying to design ICALU in a way that only brings benefits and not drawbacks in performance. Consequently, the detector is integrated into the decode-rename stage and the remainder of this chapter proceeds with that placement.

4.2.2. Collapse Execution

After selecting the decode-rename stage as the most practical location for collapse detection, attention shifts to the second ICALU modification block mentioned before: how to integrate execution of these pairs within the micro-architecture. This exploration will be discussed in 2 sub-blocks, where the first one conditions the second one. This first block will consist into the layout of the execution unit inside the execution stage of BOOM. This will affect how the issuing ports behave, as execution units and issuing ports are statically linked. Hence, the second sub-block will be how the collapsed micro-ops are integrated into the issue queues.

Execution Unit

First of all, it will be discussed how to implement this ICALU functional unit. As can be seen in Figure 4.1, the BOOM core for the case of a 2-wide core incorporates 4 execution units. 1 is a memory micro-op execution unit that will handle the memory address generation. Additionally, there are 2 execution units for integer micro-ops. Finally, there is one separate pipeline for floating point micro-ops. Different to the previous exploration, there are not many options to integrate ICALU. Two possibilities exist, also depicted in Figure 4.2:

a) ***Option A: Additional 3-1 ALU functioning as an enhanced 2-1 ALU.*** In this approach, depicted in the center of Figure 4.2 the first integer execution unit is extended by attaching a 3-1 ALU alongside the existing 2-1 ALU. The new 3-1 ALU is designed to operate in two modes: when standard micro-operations are present, it functions as an additional 2-1 ALU, supporting regular arithmetic execution. When a collapsed micro-operation is detected, both the original 2-1 ALU and the 3-1 ALU are used together to execute the producer and the producer-consumer pair in parallel, leveraging a CSA+CLA structure. Integrating this dual-mode capability requires the following modifications to the micro-architecture:

   - 1 extra issue port to accommodate the increased throughput,
   - 2 additional read ports and 1 additional write port on the register file,
   - 1 extra write-back and wake-up paths.

   These additions are necessary because, in the non-collapsed case, the architecture essentially behaves as if it has two 2-1 ALUs (however, the second one does addition, subtraction and logic operations only!).

b) ***Dedicated 3-1 ALU only active for collapsed micro-operations.*** Alternatively, seen in the left side of Figure 4.2 a 3-1 ALU can be added as a dedicated execution unit that remains idle unless a collapsed operation needs to be processed. In this scheme, the 3-1 ALU is activated exclusively for collapsed micro-ops, while standard arithmetic operations continue to use the existing 2-1

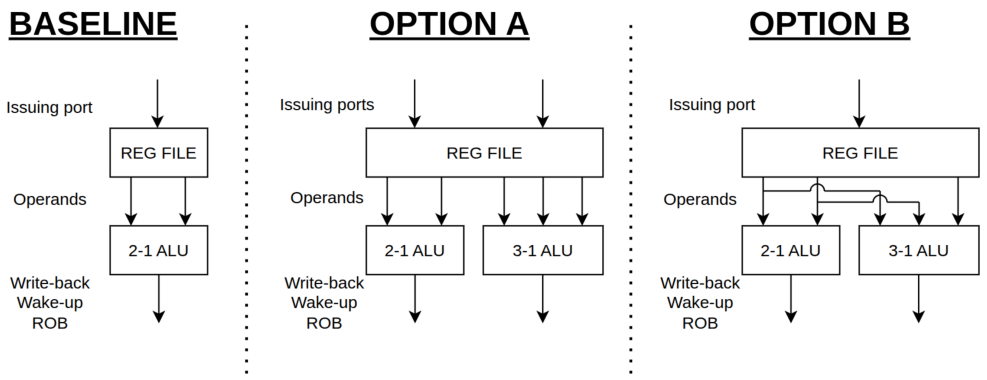


Figure 4.2: Comparison of potential ICALU placements. On the left side, the baseline BOOM ALU design. In the center Option A, which adds the 3-1 ALU of ICALU to work as an additional 2-1 ALU or together with the 2-1 to behave as ICALU. On the right the 3-1 can only work as ICALU an is idle if not.

ALUs. This strategy reduces the hardware modifications required as both 2-1 and 3-1 ALUs share their inputs, demanding:

- 1 additional read port and 1 additional write port on the register file,
- 1 extra write-back and wakeup paths.

When comparing these two integration strategies, it becomes clear that each presents a distinct set of trade-offs in terms of performance gains and micro-architectural impact. The first option, which involves extending the integer execution unit with an additional 3-1 ALU functioning as a flexible 2-1 ALU, can deliver the highest potential ILP as it effectively increases the ALU units by 1. However, this approach also introduces substantial changes to the delay paths of the processor. The increased number of issue ports and register file ports, along with the expanded write-back infrastructure, pushes the design timing substantially more than the second option. In contrast, the second approach, which adds a dedicated 3-1 ALU that is only activated when needed for collapsed operations, does not deliver the same level of performance uplift as the first strategy but limits the additional delay added when incorporating ICALU changes.

Decision The second approach is preferred because it minimizes the additional delay introduced to the issue logic and register file, making it more compatible with the goals outlined in Section 4.1. Although this option offers a lower potential performance uplift, it sacrifices less delay time for additional ILP. In line with the overall exploration philosophy of finding the minimal delay extension, it is preferred to accept a lower performance uplift if it helps maintain the delay of all paths. It should be noted, however, that this choice does not completely eliminate the impact on machine cycle time, as it has been seen some paths need to anyways be extended but rather, it keeps this impact as low as possible.

Issue queue exploration

Once the collapse detection and execution units are selected it remains to decide how the dispatch logic and issue queue handle these collapsed micro-operation waits before execution. Three architectural alternatives are available, each one balancing area, timing and design simplicity in a different way.

a) ***Option A: tagged pairs inside the existing queues.*** In this approach every micro-operation, hence every issue slot, carries metadata that links a consumer to its producer. The issuing logic must first connect the two related slots, then confirm that the operands of the pair are ready. These required link of pairs lengthens the issuing path. Additionally, extra logic is required to keep track of where the producer and consumers are.

b) ***Option B: merged pairs inside the integer queue.*** Here the producer and consumer are packed into one larger micro-operation that sits in the integer queue. When merged pairs are placed in the integer queue, it must expand its storing capabilities to include the memory queues fields.

For example, the integer queue slots now need to store fields such as is_store, is_fence or load_queue_index, etc. Therefore, area increases substantially due to increase of size in the queue, however timing remains baseline as the number of slots first issuing port has to check is the same.

c) ***Option C: dedicated collapse queue.*** A fourth queue holds only collapsed pairs stored as merged micro-operations. The original integer and memory queues shrink to keep the total slot count unchanged as micro-ops are redirected from the integer and memory queue to the collapse queue. Because execution units are statically linked to the issuing ports, the first integer execution unit hosts the ICALU and therefore the first integer issue port is formed of an integer or collapsed micro-op. The first integer issue port now receives wider request lines coming from the integer and the collapse queue so the priority encoder and multiplexer that choose between the request signals of the issue queue slots grows slightly. Figure 4.3 showcases this comparison. On the left side, the issuing logic that selects from the requests signals from the integer issue queue. On the right side, the issuing logic that would happen if Option C is implemented, which needs to track the request signals from the integer and collapsing queue. Area impact is limited as the slots added in the new queue can be removed in the other memory and integer queues as some of the micro-ops that would go to these queues are redirected to the new queue when collapsed.

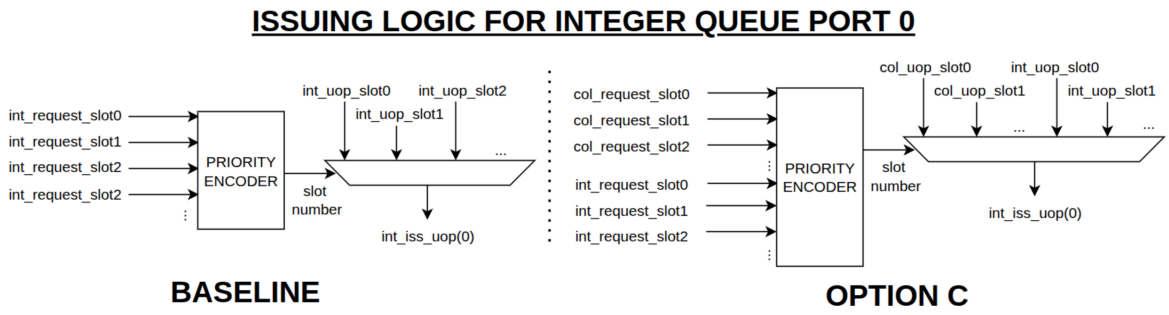


Figure 4.3: Diagram comparing issuing logic of the first integer issue port from baseline BOOM core vs when implementing Option C. Having a dedicated collapse queue but keeping the amount of issuing ports increases the amount of lines this issue port needs to track to decide which one to issue. xxx_request indicates the signal of the slot indicating it is available to be executed, being xxx the queue where it comes from. xxx_uop indicates the micro-op content and at the bottom the resulting issued micro-op. The whole structure is a priority encoder that selects from the request signals and muxes the content to be sent to the execution units.

Decision Based on the design goals in Section 4.1 of minimizing the increase of the delay of any path unless necessary, Option B is selected. Although it increases substantially the per-slot area by adding memory-related fields to the integer queue slots, it keeps the timing of the issue logic identical to baseline. Each issue port still scans the same number of slots and performs the same priority encoding, which keeps the critical wake-up select path short.

In contrast, Option A extends the issuing path with pair-linking logic, which risks lengthening the critical cycle. Option C is in principle the most area-efficient, since it avoids enlarging the integer queue slots, but in practice it slightly worsens the delay: the first integer issue port must arbitrate between two queues that have more equivalent slots (even after resizing the integer queue), enlarging the issuing mux/priority encoder, as depicted in Section 4.3.

Typically a distributed queue is usually chosen to improve timing performance, which might leave the reader thinking why the opposite is happening here. Often, when splitting queues, also a new issuing port is created, reducing the amount of slots each port has to track in the issuing process, reducing the delay of this logic. This often comes at the cost of having to carefully adjust the queue sizes. Nevertheless, based on the outcome of the last exploration, the same issuing port would host the integer queue slots and collapsed ones, different to the typical mentioned scenario. This placement increases the amount of request signals the issuing logic needs to process in that port, pressuring timing and resulting in a different scenario compared to the mentioned before.

Exploration conclusion Different design choices were presented. Following the design goals of Section 4.1 a solution has been obtained with minimal and specially unavoidable extensions of the micro-architecture paths. Instead of summarizing here the design choices, the beginning of the following section starts with a summary of the high level design which is the outcome of this design exploration.

4.3. Detailed Design Description

Once the design exploration has converged to the preferred solution, the next step is to explain how that solution is embedded into the BOOM core. The following subsections map every alteration required in BOOM. It starts with a high level overview of the design. Later, it does a deep dive into the changes applied in each stage. The flow of this design description follows the flow that an instruction or micro-op would have in the BOOM pipeline, from decoding until retirement. For the sake of simplicity, all the following explanations will take part in a 2-wide BOOM core if not mentioned. The RTL code generated is parameterizable and applies for the 2-, 3- and 4-wide core variants.

High level overview - the ICALU-enabled BOOM pipeline

Before going into every little detail, it is started with a high level view, showing the outcome of the design exploration. The high level micro-architecture diagram, in the scenario of a 2-wide core, can be viewed at a glance in Figure 4.4. Changes applied are shown in green. In more detail, the changes applied are:

1. Decode and rename. Right after the front-end delivers a 2-, 3- or 4-wide instruction bundle, a parallel collapse detection logic (shown in green, bottom-left of Figure 4.4) compares instructions within the current bundle (output of the fetch buffer) and the bundle decoded in the previous cycle (held in the register between decode+rename1 and rename2+dispatch) to identify eligible producer-consumer pairs for collapsing. This approach ensures the detection window spans up to $2\times$ the decode width, capturing both intra- and inter-cycle pairs. If an interwindow collapse is detected (that is, where the producer and consumer are split across cycles), dedicated control logic ensures that the producer is removed from the issue queue in the following cycle, forcing only the merged consumer (now a collapsed pair) proceed downstream.

2. Dispatch and Issue queue. When a micro-uop is found collapsible the consumer fields are extended to create a fused micro-op and its ”iq_type” field is changed so the dispatcher sends it into the int queue. This process is done in parallel while the rename stage is finishing the removal of false dependencies. From the perspective of the reorder buffer (ROB), collapsed micro-ops are indistinguishable from standard micro-ops when they are enqueued. Therefore, the ROB is basically unaware of any collapses. The dispatcher is then responsible for blocking the original producer (by not issuing it to the queue) and only sending the consumer, now a collapsed micro-op which encodes both micro-ops. Therefore, no significant modifications are needed to the ROB, aside from extending the write-back logic to clear the busy state for both original micro-ops when the collapsed operation completes. As explained in the design exploration of Section 4.2, the merged micro-op travels the issue queues the same way a traditional micro-op does.

3. Register file, one extra read/write port. Shown in Figure 4.1, because the fused operation reads three operands and writes two architectural results, the unified physical register file gains an extra read port and write port. This will also have implications compiled in Section 5.2. While the register file reads and writes its values, it also produces the control signals for the ICALU functional unit.

4. Execute and write-back / wake-up As seen in Figure 2.7, in the execution stage, the first integer pipeline lane is updated to include the 2-1 ALU + 3-1 ALU with a CSA/CLA structure replacing the baseline single 2-1 ALU. Logic for branch, jump and other non-collapsing operations remains unchanged, ensuring compatibility with standard operation modes. When a regular arithmetic micro-op (such as `ADD`, `OR`, `XOR`..) is issued, it ignores the 3-1 ALU, maintaining baseline behavior. When a collapsed pair is executed, both the 2-1 and 3-1 ALUs operate in parallel: the 2-1 ALU computes the producer

result and the 3-1 ALU simultaneously executes the producer-consumer sequence as a three-operand operation. The outputs of these parallel paths are then handled as follows:

- Both results are written back to the register file via the additional write ports.
- Fast bypass paths ensure that dependent operations can immediately access the most recent results, supporting back-to-back execution with minimal stalls.
- For collapsed address-generation pairs, a side-path enables results to feed directly into BOOM's existing Load/Store Unit (LSU), which is also widened to handle increased throughput. (the LSU is parameterizable, done automatically by BOOM)
- Write-back path updates the ROB, ensuring that both the producer and consumer micro-ops are marked as completed.

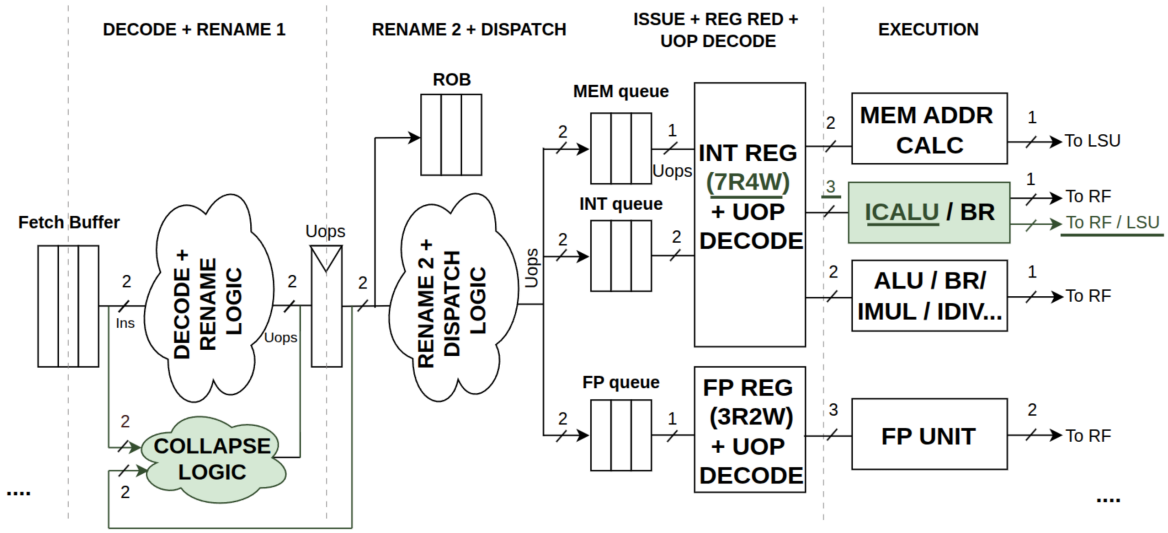


Figure 4.4: Final ICALU- BOOM pipeline for the case of a 2-wide core. Green boxes mark the new or modified logic introduced by this work.

4.3.1. Detailed Design

After establishing the high-level architecture and major design choices behind the ICALU integration, the following explanation describes in even more detail how each stage of the BOOM pipeline is adapted to support interlock-collapsing execution. The following sections follow the same order as the high-level explanation which is the flow of an instructions in the back-end of BOOM.

Decode, Rename and Dispatch: Collapse Detection

The first step of the ICALU implementation is detecting the collapses. On a high level, the design exploration of Section 4.2 concluded that collapse mechanism should be inserted in the decode and rename stages, where the front-end already supplies a two-instruction batch each cycle (when talking about a 2-wide BOOM configuration). At this point every instruction has been expanded to its 32-bit RISC-V form, so opcode and register addresses are fixed in the bit location of the instruction.

To uncover most of the producer-consumer pairs, the detector inspects the full window of micro-operations at a time. These consist of the current output of the fetch buffer and the output of the decode/rename stage, which contains the instructions that were in the fetch buffer in the previous cycle. The combined inter-cycle window guarantees that any pair separated by at most one fetch group is visible. In this window, every instruction is compared with each younger instruction for potential collapsibility. A pair is collapsible when every condition in Table 4.3 evaluates to true, as expressed in Equation 4.1.

$$\texttt{collapse\_valid} = S_{\text{match}} \wedge P_{\text{valid}} \wedge C_{\text{valid}} \wedge \ P_{\text{coll}} \wedge \ C_{\text{coll}} \wedge R_1 \wedge R_2 \wedge R_3 \wedge R_4. \tag{4.1}$$

The conditions for a pair to be collapsible can be summarized as follows:

Table 4.3: Boolean conditions evaluated in the decode/rename detector.A producer-consumer pair is considered collapsible only when every term in (4.1) is *true*.

Term	Condition
S_{match}	Source matches producer destination $(rs1_c = rd_p) \ \vee \ (rs2_c = rd_p)$
P_{valid}	Producer opcode collapsible (Table 3.1) $opcode_p \in \mathcal{I}_{\text{ICALU}}$
C_{valid}	Consumer opcode collapsible (Table 3.1) $opcode_c \in \mathcal{I}_{\text{ICALU}}$
P_{coll}	Producer not collapsed in the previous cycle $collapse_p[N-1] = \text{false}$
C_{coll}	Consumer not collapsed in the previous cycle $collapse_c[N-1] = \text{false}$
R_1	No circular chain (Rule 1, Table 3.2)) $\bigwedge_{rd_{mid} \in between} \neg\Big((rd_{mid} = rs2_c \ \vee \ rd_{mid} = rs1_c) \ \wedge \ (rs1_{mid} = rd_p \ \vee \ rs2_{mid} = rd_p)\Big)$
R_2	Store/load guard (Rule 2, Table 3.2) $rd_p \neq rs2_c$
R_3	Distinct consumer sources (Rule 3, Table 3.2) $rs1_c \neq rs2_c$
R_4	No redefinition between pair (Rule 4, Table 3.2) $\bigwedge_{rd_{mid} \in between} \neg(rd_{mid} = rd_p)$

- **RAW dependency:** The source register of the consumer matches the destination register of the producer, ensuring it is a collapse.
- **Valid mnemonics:** Both instructions have mnemonics listed in Table 3.1 which are the chosen ones to be collapsible (addition, subtraction, logic and loads / stores address generation).
- **Not previously collapsed:** The instruction has not been collapsed in any earlier cycle, preventing RAW chains to be marked as collapsible (ICALU can support 2 instructions merged at most).
- **Hazard rules satisfied:** The dependency rules from Section 3.2.3 and Table 3.2 are met. In an out-of-order core, only four such rules are required, as violating them would break correctness.

Once the predicates of Table 4.3 are evaluated and the collapsability of each instruction is obtained with Equation 4.1, each producer-consumer collapsability is built into a *collapsibility matrix*. The matrix has dimensions $2 * \textit{width} \times 4$. Pairs that fall outside the visible window never contribute, so the matrix contains only the entries that can actually be generated in hardware, therefore, let C_{ij} be the raw collapsibility flag between instructions i (producer) and j (consumer), with $i < j$ and $j < 2 * width$:

$$C_{ij} = \begin{cases} \texttt{collapse\_valid}(i,j), & 0 < j - i \leq 3, \\ 0, & \text{otherwise}, \end{cases} \tag{3.2}$$

The raw matrix is strictly upper-triangular and never stores a distance greater than three instructions in between the pair.

Figure 4.6 illustrates an example with of a window of instructions in the case of a 2-wide core. The left half shows the four-instruction window examined by the detector, the first 2 are the output of the

fetch buffer and the other 2 are the output of the fetch buffer in the previous cycle (already in rename + dispatch stage). The right side of the image shows the corresponding raw collapsibility matrix. There, it can be seen the matrix has dimensions $2 * \textit{width} \times 4$ and it has been applied Equation 3.2 to remove the flags that would fall out of the window. Each element represents the collapsibility of each pair. First element, C12, is the flag that indicates if instruction 1 and 2 are collapsible, C13 for instruction 1 and 3 and so on. Choosing a maximum distance of three instructions between the pairs is motivated by the results seen Section 3.3.2. There, it is shown than roughly 92 % of all collapses fall within this range, balancing performance and complexity.

In regards to delay, the maximum logic depth in *collapse_valid* of Equation 4.1 is set by R_1. In the worst case of 3 instructions in between, it can be implemented with a comparator, a OAI22 and a NAND3 giving R_1 a total of 3 logic levels. These are common gates similar to the proposed by IBM. The final AND-reduction over the nine predicates adds two more levels, resulting in a total critical depth of five gate levels used for this part.

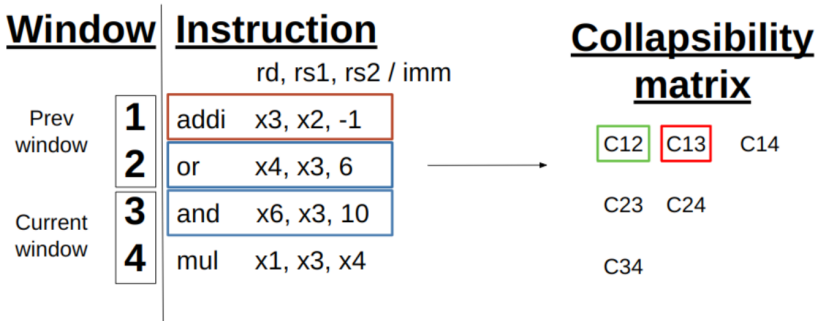


Figure 4.6: Four-instruction window (left) and the corresponding raw collapsibility matrix (right) for a 2-wide core. In this example instruction 1 can collapse with instruction 2 and 3 and the overlap collapse code prioritizes the closest collapses.

Overlap removal A given producer can have multiple younger instructions as potential consumers. To avoid conflicts, only the closest match is kept, since shorter producer-consumer distances are more likely to hide an execution interlock. This function is implemented as a distance-prioritized masking network, where it resolves multiple matches by priority.

The masked result is

$$\tilde{C}_{ij} = C_{ij} \ \wedge \bigwedge_{k=i+1}^{j-1} \neg C_{ik} \ \wedge \bigwedge_{m=i+1}^{j-1} \neg C_{mj}.$$

Thus $\tilde{C}_{ij}$ survives only if C_{ij} is true, there is no closer (fewer instructions in between) match in the same row for producer i and there is no closer match in the same column for consumer j.

In order to better understand this, consider the worst case scenario, a 4-wide core, which yields an 8×4 raw matrix (producers $i \in \{1, \ldots, 8\}$, distances $d \in \{1, \ldots, 4\}$ so $j = i + d$. The longest masking path occurs at the largest distance and away from edges, i.e., for the *fourth* instruction as producer ($i = 4$) and the *eighth* instruction as consumer ($j = i + 4 = 8$). The masked flag would be

$$\tilde{C}_{4,8} = C_{4,8} \wedge \neg C_{3,4} \wedge \neg C_{4,5} \wedge \neg C_{7,8} \wedge \neg C_{2,4} \wedge \neg C_{4,6} \wedge \neg C_{6,8} \wedge \neg C_{1,4} \wedge \neg C_{4,7} \wedge \neg C_{5,8}$$

In plain words, in order to collapse instruction 4 with instruction 8 it needs itself to be collapsible and any other collapse flag that includes instruction 4 or 8 and has less distance to be false. Terms that would reference consumers beyond the 8 entry window, such as $C_{8,9}$, $C_{8,10}$, $C_{8,11}$, are out of bounds and therefore zero by construction. In terms of delay for this logic, all negations are formed in parallel. In the worst case scenario, it this is left with $N = 10$ inputs that can be implemented with a balanced 3 input AND tree of 3 levels for the case of a 4-wide core and 2 levels for the 3-wide core.

In conclusion, this detection logic smoothly maps in parallel into the decode and rename stages, with a delay of 8 common gates. This is enough for any typical stage but even in the case a processor has a tighter back-end, part of this logic can be moved to previous stages (for instance, offloading calculations

of Figure 4.3 to fetch buffer at the cost of area), increasing area but avoiding timing impacts. Therefore, it can be concluded this does not affect the delay of these paths and no trade-off needs to be done.

Dispatch and Issuing

Once the final collapse matrix is obtained collapse information is encoded. With this encoded information the consumer is merged with the producer, hosting additional micro-op fields. Even though this might not be the optimal choice in terms of area, the design exploration done in Section 4.2 proved that this option is the only one capable of not increasing the timing of these stages and therefore meet the design goals of Section 4.1. Once the fused micro-op is obtained, the dispatcher will send it the int queue. This merging occurs by extending the fields of the micro-op based on the encoded information of the collapse matrix. For the original producer micro-op the field that indicates in which queue this micro-op goes, ”iq_type” is set to a null value, effectively not pushing that micro-op that is already in the bigger collapse micro-op. For the consumer, it is changed to go to the integer queue. This process is done while the rename work is finalized, effectively being in parallel.

From the perspective of the reorder buffer (ROB), collapsed micro-ops are indistinguishable from standard micro-ops when they are enqueued. Even though the ROB consumer has already been extended to host the collapsed micro-op, it still sees the producer. None of the additional fields for the collapsed micro-op are stored in the ROB, making it basically unaware of any collapses and only later seeing that the ICALU unit sends back two write-backs. Therefore, no significant modifications are needed to the ROB, aside from extending the write-back logic to clear the busy state for both original micro-ops when the collapsed operation completes. This ensures correct dependency tracking without introducing new critical paths.

Furthermore, additional logic kills the producer micro-ops that are found later to be collapsible with a consumer that is traveling through another cycle. The producer micro-op is allowed to sit in the issue queue (but not to be issued) instead of avoiding it dispatching it so the delay of the dispatcher is not extended. This way the micro-op is killed at the end of the cycle if it was found it was collapsible with a consumer that traveled one cycle before. Therefore, some additional logic exists to track the micro-ops that are found producers to avoid issuing them and kill them at the end of the cycle. Furthermore, the previous cycle micro-ops are stored so whenever this logic detects that the leader was dispatched the cycle before, instead of fetching it from the issue queue, which would add delay, it obtains it from this buffer. This might not be the optimal choice if there is slack, but proves this process can be done without extending the delay. Finally, the signal that stops the leader to issue is calculated the cycle before, ensuring it does not increase the delay.

Once a micro-operation enters the issue queue, it must wait until all its operands become ready. For standard instructions this typically means checking the two source registers are not busy anymore. However, a collapsed micro-operation can have up to three input operands, 2 from the producer and 1 from the consumer (the other operand of the consumer is the result of the producer). This introduces the need for an additional operand readiness check. Fortunately, BOOM and any micro-architecture already supports three-operand instructions in its floating-point issue queue, so this extension does not increase the critical path. This should be applicable to all RISC-V designs and therefore is not a concern.

Collapsed micro-operations are exclusively handled by the ICALU. As shown in Figure 4.4, the ICALU is statically assigned to the first integer execution unit and therefore to the first integer issue queue output port. As a result, only this port will issue collapsed micro-ops.

ICALU Execution and Write-back

Once the micro-op is issued, it is sent to decode its functional units control signals and to do the register read. In terms of register read, a collapsed instruction also requires fetching three source operands from the register file instead of the usual two. This is handled by reserving an additional read port assigned to the ICALU execution unit. This definitely increases the delay of this stage and is unavoidable due to the ICALU nature. Section 5.2 will compile all the trade-offs found across this design description to assess the timing implications of ICALU across all designs. Moving forward, before sharing any details about the functional units control signal generation it is important to define the execution slice itself.

As presented in Section 2.1, IBM's proposed ICALU in the early 1990s introduced a fused execution model composed of a 2-1 and a 3-1 ALU. By leveraging a CSA followed by a CLA it could execute a producer-consumer arithmetico-logic pair in a single cycle while the original producer was executed in parallel.

The implementation adopted in the execution unit of BOOM core is the exactly same one. The same types of operations are collapsed as those proposed by IBM: a combination of additions and subtractions optionally or logic operations (AND or, XOR). These include all type of instructions that require these operations, from address generation to register with register or register with immediates. Therefore, the implementation remains exactly the same as the 2-1 + 3-1 ALU proposed, with the same control signal calculation and modes. As such, the ICALU integrates seamlessly into a RISC-V pipeline without any changes required.

To provide more detail on how the ICALU proposal is integrated into the execution units of the BOOM core, Figure 4.7 illustrates the baseline BOOM integer execution unit. This unit consists of a two-operand ALU that handles both arithmetic and logic instructions and also supports branch and jump resolution in parallel. The ALU receives two operands from the register file and produces a single result, which is forwarded through the common execution response path. In addition, a bypass output enables back-to-back operations. This feature removes the need to wait for the write-back of results before reusing them and it allows unsupported consumer-producer instruction pairs to be issued sequentially without introducing additional delay cycles.

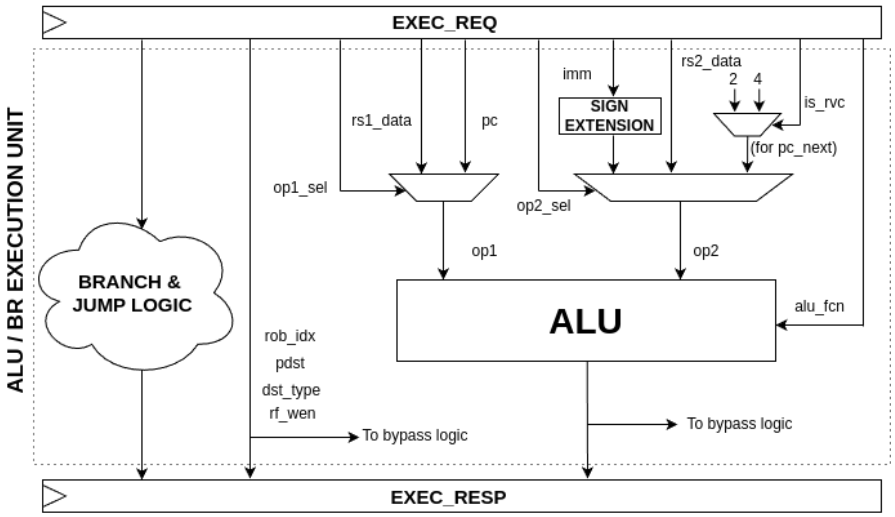


Figure 4.7: Baseline BOOM ALU. It incorporates a 2-1 ALU together with logic to handle branches and jumps

To support the collapsed execution of a producer-consumer pair, the enhanced design replaces this unit with the one shown in Figure 4.8. In this implementation, the original 2-1 ALU is complemented with a 3-1 ALU, which again is described as the one shown in Section 2.1. This new data-path accepts an additional third operand and computes the fused operation using a CSA+CLA pipeline. Both ALUs remain functional: if the issued instruction is a regular arithmetic or logic operation, it follows the same path as in the baseline, bypassing the 3-1 unit and executing in a single cycle. If the instruction is collapsed, the producer is computed in the 2-1 ALU and the consumer calculated in parallel in the 3-1 ALU, completing the entire operation in one cycle.

With this design in mind, at the previous stage, the register read and functional unit decoding stage, the requirements are similar to those described in IBM's original work. The decoding of the control signals for the execution unit is exactly the same as the one proposed by IBM. In terms of timing, the additional operation performed by the ICALU requires the register file to support one more register read port, since one operand of the fused micro-operation is already the result of the producer. Furthermore, the design requires one more bypass path to handle producer-consumer forwarding correctly. All of these will later be discussed in the implications of incorporating ICALU, in Section 5.2.

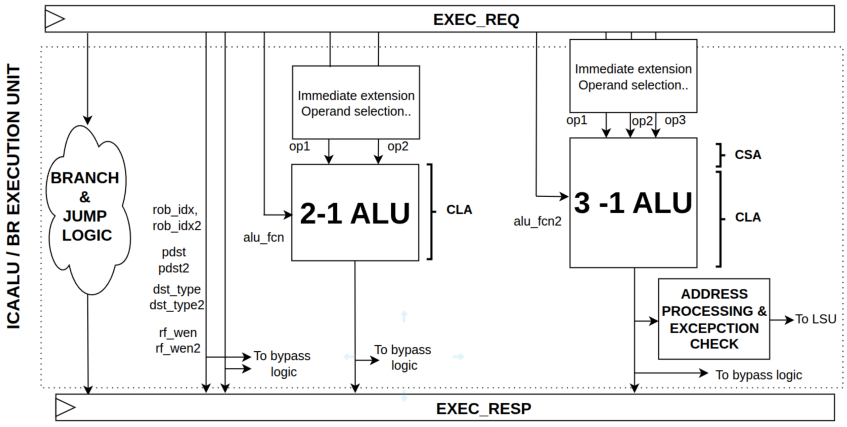


Figure 4.8: ICALU BOOM execution unit. Adds 3-1 ALU path for collapsed instruction execution compared to the baseline and address processing and exception check for the address generation.

The 3-1 ALU output is paired with a new address generation logic (exception checks and post-processing) for the pairs that have a load or store as their consumer. This block is exactly the same as the existing address generation unit logic. However, there are some problems with the delay increasing. This change has several implications:

- **Pre-addition logic:** Immediate extension and operand selection through multiplexers are introduced in the memory address generation path. These are normally not present in a memory address generation unit, adding delay to the input path of the LSU. However, these can mainly be can be moved to the previous stage to reduce the added delay if required.
- **Addition logic:** The adder in this path is a CSA-CLA, which increases the delay of the same memory address generation path. This was expected, as it was clear that ICALU requires an extra CSA stage of delay, but it not only affects the path of the ALU execution unit but also the memory address generation / LSU input path.
- **Dual input handling:** The LSU has to be extended to accept two instructions in parallel. In the BOOM core implementation is not a problem in terms of development, but in terms of timing this results in a longer path for the LSU input.

Even though some of these costs can be partially mitigated, the conclusion is clear: the memory address generation path is extended, pressuring the statement that ICALU can be included without machine cycle time increase.

Once instructions are executed, the ICALU produces two results. As shown in Figure 4.8, this design requires an additional bypass path to support back-to-back operations with a dependent consumer. In addition, one more write-back port is required to send the second result to the register file and another Reorder Buffer (ROB) write-back port is needed so that the ROB can correctly mark the corresponding micro-operation as complete and clear its busy status.

On the retirement side, the ROB is unaffected by the collapsing logic. Because the ROB enqueues instructions before any collapse occurs, it only observes that two results are produced at the same time but remains detached from the collapse mechanism. The maximum number of committed instructions per cycle stays the same, which is consistent with the fact that the processor still enqueues at a rate equal to the width of the decode stage. The ROB therefore does not require modifications, as it simply records that a collapsed instruction completed execution within that cycle.

Design conclusion: In summary, this section has described the complete set of modifications required to integrate ICALU into the BOOM core, from early-stage collapse detection to dispatch, execution and write-back. The proposed design adheres to the guiding goals of Section 4.1, isolating the unavoidable extensions to timing-critical paths.

The result is a fully functional ICALU-enabled BOOM pipeline that preserves compatibility with standard

operation while adding support for fused producer-consumer execution. The next step is to quantify whether these modifications translate into the performance uplifts suggested by the collapse statistics of Section 3.3 and to compile, in Section 5.2, the clear lessons learned in this implementation work about the trade-offs ICALU has in any modern processor.

4.4. Conclusion

With the results of the trace analysis, the BOOM core was modified to incorporate ICALU. This served two purposes: quantify the potential ILP improvement in a real core and identify unavoidable timing trade-offs. This is crucial, as ICALU's ILP increases should not come at the cost of a higher machine cycle time. The design process therefore aimed to create a solution that requires minimal timing trade-offs across any micro-architecture while exploiting as many collapsible pairs as possible.

After establishing the design goals, the project explored two blocks: collapse detection and execution. Several placements for collapse detection were considered: L1 cache, decode-rename and issue queues. Decode-rename was ultimately selected for its ability to maintain stage delay and avoid penalizing baseline performance, even though it slightly limited collapse opportunities relative to the cache approach.

For execution, both the layout of the execution units and the issue-queue structures were explored. Adding a flexible 3-1 ALU that could also act as an extra 2-1 ALU introduced substantial timing penalties due to extra register ports. A dedicated 3-1 ALU active only for collapsed micro-ops was chosen, reducing delay impact. Among issue-queue options, storing merged micro-op pairs inside the integer queue was selected over tagged pairs or a separate collapse queue, since it was the only one that preserved baseline timing despite higher area per slot. All decisions prioritized minimizing any increase in critical path delay, even if slack was available, to reveal the unavoidable paths that need to be extended and to understand ICALU's costs.

5

Measurements and Results

In order to finally address the main question of this work, this chapter focuses on presenting the outcomes of implementing ICALU into the BOOM core. On one hand, it aims to obtain the ILP improvements of running workloads on a baseline BOOM core vs. an ICALU-enhanced one. On the other hand, this chapter aims to extract the identified costs of implementing ICALU in any modern core. The results presented here help readers understand the benefits and required trade-offs of incorporating this potential innovation in a modern RISC-V processor.

Section 5.1 presents the experimental setup, verification process and performance results of the baseline BOOM core compared to the ICALU-enhanced version, followed by a discussion that projects the performance results to SPEC2017 using extrapolation. Section 5.2 summarizes the trade-offs identified when integrating ICALU into a modern out-of-order processor, highlighting the timing and design costs that accompany the observed gains. Section 5.3 presents the conclusions of the measurements and results obtained.

5.1. ICALU Performance Results in BOOM Core

Trace analysis in Section 3.3 confirmed that modern workloads expose an enough number of interlock collapse opportunities. While this result confirms that data-dependent instruction pairs continue to be common in modern code, it does not guarantee that an ICALU design in a modern processor will provide significant performance gains. Therefore, the next step is to determine whether those opportunities, when exploited in hardware, translate into real cycle savings and shorter benchmark execution.

This section explains the cycle-accurate performance results of implementing ICALU into the BOOM core, an out-of-order superscalar parameterizable core. The implementation work was done in Chapter 4. The following section begins with a description of the experimental setup before presenting the benchmark results of incorporating ICALU in BOOM.

5.1.1. Design Verification and Experimental Setup

Before presenting any results, it is important to first describe the hardware setup and the steps taken to ensure execution correctness, so that the results that follow are trustworthy.

Hardware Setup

The core used in this work is BOOM, an out-of-order superscalar parameterizable core, which was presented in Section 2.3.3. This core has been modified with all required changes to incorporate ICALU as presented in Chapter 4. The simulation results will be presented for the default 2-, 3- and 4-wide configurations.

The RTL model simulated is a SoC with a single BOOM core. The SoC parameters use the default, simplest configuration provided by Chipyard, the tool that generates these chips. This covers a cache hierarchy, UART and other necessary peripherals. The final parameters for the core, the SoC and the simulated system are presented in Table 5.1.

Table 5.1: Core and system parameters of the evaluated hardware configuration in this work. Three different systems are simulated, with varying core micro-architecture parameters.

Micro-architectural Parameters	**2-wide**	**3-wide**	**4-wide**
Fetch width [bytes]	8	16	16
Decode width [instr./cycle]	2	3	4
ROB entries	64	96	128
Int. IQ (entries / issue width)	20 / 2	32 / 3	40 / 4
FP IQ (entries / issue width)	16 / 1	24 / 1	32 / 2
Mem IQ (entries / issue width)	12 / 1	16 / 1	24 / 2
Physical registers (int / fp)	80 / 64	100 / 96	128 / 128
LSU queue entries (load / store)	16 / 16	24 / 24	32 / 32
L1 I$ (size, ways×sets, refill cyc.)	16 KiB, 4×64, 8	32 KiB, 8×64, 4	32 KiB, 8×64, 4
L1 D$ (size, ways×sets, refill cyc.)	16 KiB, 4×64, 8	32 KiB, 8×64, 4	32 KiB, 8×64, 4
L2$ (capacity, ways×sets, refill cyc.)	512 KiB, 8-way, 1024 sets, 40		
Main-memory model	DDR3, FRFCFS timing, 2 GiB, single channel		

Hardware Verification

Before running the benchmarks, the design underwent a thorough verification methodology using Chipyard's tools. This established functional correctness and robustness before moving to full-system performance evaluation. The aforementioned verification process was simulated through Verilator, the RTL simulator used also in the final results of this section.

The debugging process was carried out with existing and additional assertions and prints in the BOOM pipeline, helping spot and fix errors. Moreover, the simulations were run in co-simulation with Spike. Spike is another functional simulator, similar to QEMU, the one used for the trace analysis. Spike focuses more on functional correctness rather than speed, which is essential for verification. In this co-simulation mode, each committed instruction from the core was matched against Spike's execution, comparing the program counter, instruction result and register values. Any divergence halted the run and reported the mismatch, ensuring that ICALU did not alter architectural correctness and helped spot errors.

Finally, using this verification and debugging methodology, different bare-metal workloads were run under the aforementioned hardware setup, confirming the ICALU-enhanced system has functional correctness. These were:

- **GNU ISA tests**: Over 200 assembly ISA tests that come with the RISC-V GNU toolchain to ensure ISA compliance.
- **Chipyard micro-benchmarks**: Chipyard's 12 directed micro-benchmarks that, instead of focusing on ISA compliance, target micro-architecture correctness. These pushed common arithmetic, memory and control-flow corner cases.
- **Randomized Torture suite**: Long pseudo-random programs that run for days, stressing the rarest pipeline interactions.
- **CoreMark and Embench**: The benchmarks used for evaluating core performance were also verified with Spike co-simulation. Furthermore, CoreMark incorporates a self-checking CRC that ensures its execution was correct.

After completing successfully these stages it was confirmed that the ICALU modifications preserved correctness and ensured that the potential ILP improvements that are going to be shown are correct and trustworthy.

Software Setup

The two programs chosen as workloads for this study are CoreMark and the Embench Suite. Both benchmarks are compiled at `O2`. Embench is presented as a single benchmark even though it is composed of multiple benchmarks. The main reason is that each benchmark in the suite is a kernel consisting of a handful of basic blocks, providing an overly biased view of ICALU if analyzing each of them. However, all these kernels seen as a single benchmark present a good variety of workloads, yielding richer and more meaningful results. CoreMark is compiled for 100 iterations, the default setting provided by Chipyard.

The aforementioned SoC running the mentioned workloads is simulated using Verilator. This is an RTL simulator faster than traditional simulators and allows modern complex cores to retire millions of instructions per hour. However, this simulation does not allow long and complex benchmarks such as SPEC2017 to be run because it retires billions of instructions per run and still would take years to complete them. Therefore, the performance results for this benchmark are relegated to extrapolation, linking the trace analysis with the ILP improvements.

Metrics and Evaluation Methodology

Finally, it is required to understand the metrics used to quantify performance improvements and other supporting measurements. The following are the metrics that will be used to assess ICALU's potential improvement:

CoreMark and Embench Score For CoreMark, the simulator outputs the total number of processor ticks required to complete 100 iterations of the benchmark. In the case of Embench, the core is instrumented to report the cycles from benchmark start until the end. In both cases, the performance improvement is measured as a relative cycle reduction, computed using the following formula:

$$\text{Speedup} = \frac{T_{\text{baseline}} - T_{\text{ICALU}}}{T_{\text{baseline}}}$$

where T_{baseline} and T_{ICALU} denote the number of ticks/cycles reported by both benchmarks for the baseline and ICALU-enhanced designs, respectively.

Collapse Execution Success Rate In addition to performance scores, it is important to evaluate how effectively the ICALU modifications exploit the collapse opportunities previously identified in trace analysis. The speedup values will not be meaningful if it is not known whether if most of the collapse opportunities detected have been exploited. For this purpose, the *Collapse Execution Success Rate* is introduced. This metric compares the collapse rate observed in the modified BOOM core with the theoretical collapse rate derived from trace analysis.

Formally, the metric is expressed as:

$$\text{Collapse Execution Success Rate} = \frac{\text{Hardware Collapse Rate}}{\text{Theoretical Collapse Rate}}$$

where the Hardware Collapse Rate is measured from hardware counters as

$$\text{Hardware Collapse Rate} = \frac{N_{\text{collapsed}}}{N_{\text{total}}},$$

with $N_{\text{collapsed}}$ being the number of *committed* collapsed pairs and N_{total} the total number of *committed* instructions.

The Theoretical Collapse Rate is obtained from trace analysis as described in Section 3.3, adjusted to account for the maximum detection distance of three instructions enforced in the ICALU BOOM core (see Section 4.3.1).

In Section 4.2 it is noted that certain collapses that take place for micro-ops that are between different stages might not yield a performance uplift. The Hardware Collapse Rate does not count these collapses as successful in order to reflect the effectiveness of this ICALU implementation. In conclusion,

values closer to 100% for this metric indicate that ICALU is exploiting almost all theoretical collapses found in the workloads and that the results reflect ICALU's maximum potential.

5.1.2. Performance Metrics Results: Baseline vs. ICALU

Having defined the hardware configuration, its validation, the workloads used and the performance metrics in Section 5.1.1, this section presents the main results. The following tables compare the baseline and ICALU-enhanced BOOM cores across three issue widths configurations (2-, 3- and 4-wide). Each table reports the score of the benchmark execution, the relative speedup as defined earlier and the collapse execution success rate. Readers with any doubt about the meaning of the metrics or the platform configuration are encouraged to refer back to Section 5.1.1.

Table 5.2: CoreMark results across BOOM widths with and without ICALU. The table reports the baseline and ICALU scores, the relative speedup in percentage, and the collapse execution success, which represents the fraction of theoretical collapses successfully executed by ICALU for each core width.

Width	Baseline Score	ICALU Score	Speedup (%)	Succ. Rate (%)
2-wide	24.37	24.00	1.51	74.1
3-wide	18.47	17.16	7.12	90.6
4-wide	17.47	16.23	7.10	93.2

Table 5.3: Embench average results across BOOM widths with and without ICALU. The table reports the baseline and ICALU scores, the relative speedup in percentage, and the collapse execution success, which represents the fraction of theoretical collapses successfully executed by ICALU for each core width.

Width	Baseline Score	ICALU Score	Speedup (%)	Succ. Rate (%)
2-wide	38.22	37.69	1.38	71.0
3-wide	30.50	29.47	3.38	80.0
4-wide	26.89	26.02	3.23	82.6

Overall, the measured improvements from ICALU integration are modest but consistent. Across all benchmarks, the observed speedups range from approximately 1% to 7%, with higher values generally correlating with increased core width. Collapse execution success rates, representing the rate of collapse pairs successfully committed vs. the theoretical collapse opportunities found in the trace analysis of Section 3.3, consistently fall between 70% and 93%. This indicates that the implementation is capable of exploiting the majority of collapsible pairs found in modern workloads and that the results obtained are a good reflection of ICALU's potential to speed up ILP.

SPEC2017 integer extrapolated results As previously mentioned, SPEC2017 retires billions of instructions, making it unsuitable for RTL simulation with the available tools. Using the trace analysis and the previous results of CoreMark and Embench, the SPEC2017 integer results are extrapolated.

The average collapse rate for SPEC is 7.7%, which is lower than Embench (8.8%) and CoreMark (11.7%). If the observed relationship between collapse rate and achieved ILP speedup in Embench and CoreMark is taken into account, the extrapolated expected performance margin for SPEC is roughly 1% speedup for the 2-wide core. For the 3- and 4-wide cores, an extrapolated result of 3–5% is obtained. While this is a rough extrapolation, it provides a sufficiently accurate indication of how the gold standard of core performance could respond to the introduction of ICALU: a modest benefit that is not impressive but noticeable.

Similarities with IBM performance studies Interestingly enough, these results almost perfectly match the ones reported by IBM in the 1990s for SPEC92, presented in Section 2.1.4. Not only are the results almost the same but their behavior across micro-architecture parameters too. In both studies, higher core widths presented higher speedups when incorporating ICALU. Furthermore, in the IBM studies, it was also seen how increasing back-end resources while keeping the front-end resources constant left the performance uplift essentially unchanged, something also seen here when comparing the 3-wide and 4-wide core, which have the same front-end but different back-end resource size. Be-

cause it would be very unlikely for such specific behavior to be a mere coincidence, it can most likely be concluded that the performance uplifts from ICALU today vs. the 90s behave pretty much the same.

5.1.3. Results Behavior Discussion

The results presented here show patterns that need to be discussed in more detail. It can be clearly seen that higher core widths present substantially higher speedups and the reader might be left wondering the reason behind such behavior.

Before resolving why this behavior happens, it is important to go back to Table 5.1, where each configuration's micro-architecture parameters are presented. Note that the 2-wide core features a smaller front-end compared to the 3-wide and 4-wide cores, which have the same front-end and differ only in their back-end.

With this in mind, the reasons why this 2-wide core presents much lower results are:

- First, the 2-wide core has a reduced front-end throughput, which constrains the number of collapsible instruction pairs exposed to the collapse detection logic. As described in Section 4.3, the collapse detector's ability to find pairs depends on the front-end throughput. As a result, the success rate of collapse execution is slightly lower for the 2-wide core. However, this effect does not explain most of the gap in speedup, as the reduction in success rates for smaller cores is not very pronounced (around 10–15%).
- The second and main reason lies in the importance of execution interlocks. When comparing the 3- and 4-wide cores, since the front-end is identical, the score improvements when going wider must be attributed only to increased back-end throughput. However, in CoreMark, the baseline 3-wide and 4-wide cores show little difference in performance, 18.47M cycles vs. 17.47M cycles (5.7% speedup). This could indicate two things: either the back-end is unable to further exploit its additional resources (due to the presence of RAWs, for instance) or the core is front-end bottlenecked and no extra back-end resources can speed up the benchmark.

  However let's 2 scenarios, the 3-wide ICALU core vs. the 4-wide baseline core. In both scenarios the front-end is the same and the difference is that the first one the back-end has been changed to incorporate ICALU while maintaining the other resources and the second scenario has a wider back-end (bigger queues, reorder buffer, decode throughput, register file size, etc.). The results show that the first scenario executes in 17.16M cycles vs. 17.47M cycles of the second. In other words, when keeping the fornt-end the same, Coremark sees a bigger uplift when integrating ICALU than moving from a 3-wide back-end to a 4-wide one. This clearly indicates that CoreMark is bottlenecked by RAW dependencies and serves as a prove that ICALU is effective in alleviating this effect.

  Furthermore, Embench shows a different behavior that leads to the same conclusions. This benchmarks benefits from the extra back-end resources in the 3-wide vs. 4-wide comparison in the baseline results, at 30.50M cycles vs. 26.89M cycles (11.8% improvement). This indicates that the OoO engine is not bottlenecked as much by RAW dependencies and can exploit the additional resources. This is then reflected in the speedup results, showing a more modest uplift (about 3% in Embench vs. 7% in CoreMark).

In conclusion and putting everything together, ICALU demonstrates the greatest benefit in situations where execution is frequently stalled by RAW dependencies that traditional out-of-order scheduling cannot overcome. These dependencies have a higher impact as core width increases, since the frequency of them increases per decode bundle size. Briefly: the wider the core, the higher the ICALU uplift due to the higher presence of RAW dependencies and therefore higher removal of execution interlocks, with an even more clear correlation width the front-end throughput.

This trend naturally raises the question of how ICALU might perform in even wider cores, where these effects are likely to be even more pronounced. State-of-the-art proprietary designs easily go beyond a 4-wide core configuration. The SiFive P870 has a 6-wide decode stage [28] and other RISC ISAs reach up to 10-wide cores, such as Apple M4 [29] or ARM Cortex-X925 [30]. Seeing that IBM performance studies yield extremely similar results to the ones shown here, their studies present around a 6% improvement for an 8-wide core, which might help answer this question. Nevertheless, whether

this might be an even more appealing scenario in modern or future cores is outside the scope of this work and left as future work.

Main takeaways:

- The increase in instruction-level parallelism (ILP) from ICALU integration is negligible for the 2-wide core, with only about a 1% speedup.
- For 3- and 4-wide cores, ICALU delivers modest but consistent ILP improvements, yielding up to 7% speedup for CoreMark, around 3% for Embench and an extrapolated 3–5% improvement for SPEC based on collapse rate analysis and the other benchmark results. These SPEC results match almost perfectly IBM findings in the 1990s, described in Section 2.1.4, both in values and the behavior seen across micro-architecture parameters.
- The data show that ICALU is most effective when the rate of execution interlocks per cycle is higher. A clear trend indicates that this happens the wider the core is, more specifically, the higher is the front-end throughput, and explains why ICALU has higher speedups the higher the core widths is.

However, these found improvements might not be relevant if machine cycle time is affected. Before answering whether ICALU should be used in RISC-V designs, the next section presents the identified trade-offs of ICALU found when incorporating ICALU into BOOM.

5.2. ICALU's Trade-Offs in Modern Processors

As discussed in Section 2.1, IBM originally presented ICALU by claiming that its benefits in ILP came at no cost to machine cycle time, since the extra CSA stage would not place the ALU on the critical path. This assumption was crucial: if performance gains relied on lowering the frequency of any system, the approach would collapse on its own. However, as seen in the design process of Chapter 4, the implementation experience shows that this claim does not fully hold true in modern cores. Even when designing for minimal delay path increases, that section proved that ICALU introduced unavoidable implications to any microprocessor design.

The following section summarizes these trade-offs found in the implementation process, providing an overview of the real costs that accompany ICALU.

Timing

The main detrimental factor of a potential ICALU unit was its impact on machine cycle time. If incorporating ICALU would bring ILP improvements but decrease machine frequency, it would cancel the benefits and deem the solution pointless. IBM studies presented in Section 2.1 concluded that the only timing implication was the extension of a traditional ALU by adding an extra CSA stage. This was concluded not to affect machine cycle time, showing that the multiply–add structures already had longer delay paths.

Revisiting this claim, it holds today but for a different reason: if ALU designs have not changed substantially over the last decades [27], then they remain unlikely to define the overall critical path of the processor. Therefore, the lack of innovation in this area motivates the idea that, up to this day, an extra CSA adder in the ALU would not affect machine cycle time. It should be noted that it was not possible to confirm the claim that the IBM ICALU implementation would add only an extra CSA stage of delay. This is because the standard cells used by IBM in the 1990s to design ICALU, claimed to be universal across modern process nodes, were not found in certain modern process nodes. In conclusion, the logic equations for the 3-1 ALU need to be re-evaluated if the same claim of the ALU being one extra stage longer across any process node is to be true and for now this claim might be tied to specific nodes.

Nevertheless, this is not the only part of an ICALU-enabled implementation that needs to be revisited with regard to timing. IBM studies focused only on the ICALU execution unit slice but did not focus on the micro-architecture timing implications. The design exploration and implementation in Chapter 4, which aimed at identifying potential timing costs in any modern micro-architecture, showed that additional unavoidable paths arise when incorporating ICALU into any modern micro-architecture.

That chapter showed that collapse detection logic can be integrated without extending the paths in decode, rename and dispatch stages of a modern out-of-order core. By contrast, the execution of these collapses necessarily requires extending paths from the issuing stage to the retirement stage of the back-end of a processor. A collapsed pair ends up being equivalent to executing one extra micro-operation, which introduces additional work in several already tight paths. In more detail, it required an extra register read/write port, an extra wake-up path, an extra write-back to the ROB, an extra bypass path to the register file and one extra LSU path (incoming micro-op for the LSU). In other words: the equivalent to extending the datapath of the back-end to accommodate one more micro-operation from issuing to commit (with the exception that instead of two extra register read ports, only one is needed).

The origin of these newly identified timing implications is rooted in both the research context and the evolution of processor micro-architecture. Two primary factors explain the emergence of additional critical paths when integrating ICALU in modern designs:

- **Narrow Research Focus in Original Work.** IBM's original analysis considered only the timing implications within the execution unit, excluding broader architectural effects. The only other impact at that time was the need for additional register file ports when incorporating ICALU. However, the authors dismissed as follows: "While it is a real concern in machine design, it is not considered further in this paper as it pertains to a different machine cycle than the execution cycle being considered and it appears that it has not created undue problems in other implementations" [1], referring to the extra register ports. In our opinion, any proposal for a potential innovative solution should account for all downstream micro-architectural effects and focusing solely on the execution unit slice is insufficient for a complete assessment of ICALU or any potential improvement. Therefore, this different philosophy of the project that thinks these paths should be considered, added new timing implications.
- **Micro-Architectural Evolution Since the 1990s.** At the time of IBM's proposal, out-of-order execution engines were only beginning to appear and micro-architectures were significantly less complex than those found in modern cores. Structures such as bypass networks, wake-up logic and reorder buffers were absent. The introduction of these elements means that a 3-1 collapsed micro-operation now propagates additional demands throughout the back-end. As a result, integrating ICALU in contemporary designs unavoidably creates new timing-critical paths and resource contention, which were not present in the original studies.

Area & Power

Without a full place-and-route it is not possible to quantify exactly the area impact of implementing ICALU. What can be said is that the main logic additions come from two sources: (i) the 3-1 ALU datapath and (ii) the collapse detection logic. Both of these blocks are composed of simple logic structures (adders, comparators and a small amount of control) and the area impact is expected to be really small compared to the overall core. Historically, as noted in the IBM papers from the 1990s (see Section 2.1), ALUs were considered inexpensive, almost-commodity blocks and in today's context of multi-million-transistor cores this statement becomes even more true.

For register area usage, it depends on the micro-architecture implementation. When following the one proposed in this work, area increases substantially in parts such as the issue queue where new fields are hosted per slot. However, as noted earlier, most of these options were chosen because they gave zero delay increases and might not be the ones chosen if enough slack is available.

Regardless of that, assuming the current implementation is used and, in an extremely unrealistic worst-case scenario, the integer issue queue size is doubled and every pipeline register is doubled, it would still have a modest increase in area when integrating ICALU in BOOM and, most likely, any modern core. This is concluded based on the area usage of these stages in recent BOOM tape-outs [31], where it can be seen that the majority of the big structures have been untouched when incorporating ICALU (front-end, floating-point execution units and register file). However, as noted earlier, this remains only an informed expectation rather than a measurement without any place-and-route. It cannot be confirmed if the increase is specifically 1, 3 or 5%, but even with the over-pessimistic scenario it can be concluded that it is definitely not large.

Power and energy consumption are even harder to estimate without detailed synthesis and activity-based simulation. Power depends not only on transistor count but also on switching activity, critical

path balancing and layout choices. One can only argue indirectly: if the area overhead is modest, then the additional leakage is also modest and the dynamic energy of a few extra gates toggling per cycle is unlikely to dominate the overall execution unit power. This line of reasoning hints at a low impact on power, but, again, it must be emphasized that this is speculation, not analysis. Only a back-annotated power estimation flow could provide quantitative results.

5.3. Conclusion

With the BOOM core modified, a SoC consisting of one core was simulated, comparing baseline and ICALU-enhanced variants for both CoreMark and Embench. For the 2-wide core, the speedup was negligible, at around 1% in both benchmarks. In the 3- and 4-wide cores, speedup reached up to 7% in CoreMark and 3% in Embench. Linking the trace-analysis results of these benchmarks with the ILP improvements showed an extrapolated improvement of 3–5% in SPEC2017 integer. These results almost perfectly align with the IBM performance studies.

Hardware counters show around 70–90% of the pairs committed by ICALU compared to the number of collapse pairs found in the trace analysis. This indicates that the performance numbers are a good representation of the maximum uplift ICALU can bring. The results demonstrated ICALU's ability to accelerate dependency-bound back ends, with a clear trend that showed how wider cores and, more specifically, higher bandwidth front ends have higher performance uplifts.

On the other hand, the engineering process exposed unavoidable costs that any modern core will face when incorporating ICALU. Unlike IBM's claims in the 1990s, integrating ICALU today requires widening several stages of the back end of a modern out-of-order processor. From issue to commit, the datapath has to accommodate one extra micro-op. These 30 years of microprocessor development seem to have introduced very different back ends that appear incompatible with a timing-free ICALU.

6

Conclusions

6.1. Summary

In the 1990s, IBM presented ICALU: an innovative solution that fused dependent arithmetic and logic operations using a three-operand ALU. In Chapter 2 this concept was revisited, showing it was designed with a clear philosophy: to provide a boost in processor performance regardless of the implementation. This meant ICALU did not require changes to the ISA and software, used standard CMOS gates and, above all, ensured that any increase in ILP translated directly into performance by not lengthening the delay of any path.

IBM carried out cycle-approximate simulations on SPEC92 workloads to prove its benefits. In those experiments, ICALU improved performance by 1.5%, 4% and 6% for 2-, 4- and 8-wide out-of-order cores, with similar numbers for in-order cores, showing that its benefits grew with machine width.

In order to revisit ICALU three decades later, this project used the RISC-V ecosystem, which, with its flexible cores, fast simulators and broad benchmark suites, provides an even more realistic and accurate procedure to assess whether ICALU can still deliver meaningful gains in modern processors.

The project started with a feasibility study done in Chapter 3 that concluded that most of IBM's design assumptions and constraints hold true and that, after 30 years, the design is still applicable in modern cores. However, one of the constraints that limited the design to use a "common book-set" found across all process nodes could not be revalidated. These standard cells were not confirmed to be widely available in modern process nodes and a full re-derivation of ICALU's Boolean equations with widely available cells is needed to ensure that the claim of ICALU extending a traditional ALU formed by a 4-stage CLA to a 5-stage CLA+CSA, regardless of the fabrication process, is still true.

The study proceeded with a trace analysis of CoreMark, Embench and the SPEC2017 integer benchmarks. This analysis confirmed that collapsible ICALU pairs are a regular feature of modern workloads, appearing about one every eleven instructions or, in other words, at an average rate of 7.7%. Furthermore, it showed no evidence of pairs that convert into equivalent 4-1 operations, removing the need to have special hardware to handle these rare collapsed pairs. It also showed how compiler optimization levels increase the distance between instruction pairs. This concluded the aforementioned feasibility study by showing that there are enough collapse opportunities to commit engineering effort to modifying the BOOM core and by addressing assumptions and claims previously left unanswered.

Trace analysis also showed additional relevant information for the micro-architecture changes that later took place. It was observed that half of the collapsible pairs were adjacent in the instruction stream and more than 92% were separated by three or fewer instructions. Opcode mix was found to be diverse, with *add-add* and *add-load* combinations each accounting for 30% of the total pairs found. It was also found that removing logical opcodes from the mix keeps 80% of the collapse pairs, demonstrating that most of the operations are addition, subtraction or address generation.

Additional results showed that, by removing zero-operand add/sub instructions, the collapse rate stays

constant. This means most of the collapse pairs found cannot be executed with simpler macro-op fusion techniques and motivates incorporating a specific ICALU unit to exploit them. Finally, the results showed that if a processor does not have register renaming and has an in-order architecture, collapse opportunities drop by 20%, shedding light on future work in simpler in-order cores.

With the results of the trace analysis of Chapter 3, the BOOM core was modified to incorporate ICALU in Chapter 4. This served two purposes: to quantify the potential ILP improvement in a real core and to identify the unavoidable timing trade-offs. This point was seen as crucial, as the potential ILP increases of ICALU should not come at the cost of higher machine cycle time, which would cancel the benefit of ICALU. Therefore, the design process was aimed at creating a solution that would require the minimal trade-offs in timing across any micro-architecture implementation while exploiting the maximum possible number of collapsible pairs found by the processor.

After establishing the design goals, the project moved on to a design exploration by dividing the changes into two blocks: collapse detection and execution. Several placements for collapse detection logic were considered: the L1 cache, the decode-rename stage and the issue queue. The decode-rename stage was ultimately selected for its ability to maintain the delay of these stages and to avoid penalizing baseline performance, even though it slightly limited the number of collapse opportunities compared to the cache approach.

For the execution, both the layout of the execution units and the issue queue structures were explored. It was proposed to add a flexible 3-1 ALU that could also act as an extra 2-1 ALU, but it introduced substantial timing penalties due to the increased number of register ports. A dedicated 3-1 ALU active only for collapsed micro-ops was chosen, as it reduced the impact on delay. Among issue-queue options, storing merged micro-op pairs inside the integer queue was selected over tagged pairs or a separate collapse queue, since it was the only one that preserved the baseline timing despite higher area per slot. All the decisions prioritized minimizing any increase in critical path delay, even if there was available slack, to reveal the unavoidable paths that need to be extended and later to understand ICALU's costs.

With the BOOM core modified, a SoC consisting of one core was simulated in Chapter 5, comparing baseline and ICALU-enhanced variants for both CoreMark and Embench. For the 2-wide core, the speedup was negligible, at around 1% in both benchmarks. In the 3- and 4-wide cores, speedup reached up to 7% in CoreMark and 3% in Embench. Linking the trace-analysis results of these benchmarks with the ILP improvements showed an extrapolated improvement of 3–5% in SPEC2017 integer. These results almost perfectly align with the IBM performance studies.

Hardware counters show around 70–90% of the pairs committed by ICALU compared to the number of collapse pairs found in the trace analysis. This indicates that the performance numbers are a good representation of the maximum uplift ICALU can bring. The results demonstrated ICALU's ability to accelerate dependency-bound back ends, with a clear trend that showed how wider cores and, more specifically, higher bandwidth front ends have higher performance uplifts.

On the other hand, the engineering process exposed unavoidable costs that any modern core will face when incorporating ICALU. Unlike IBM's claims in the 1990s, integrating ICALU today requires widening several stages of the back end of a modern out-of-order processor. From issue to commit, the datapath has to accommodate one extra micro-op. These 30 years of microprocessor development seem to have introduced very different back ends that appear incompatible with an ICALU free of timing implications.

6.2. Main Contributions

This project makes the following key contributions:

- **Confirms ICALU's feasibility** in the context of modern RISC-V processors.
- Develops a trace-analysis methodology and **demonstrates** that **collapsible** producer-consumer **pairs are frequent** in contemporary workloads, appearing roughly once every eleven instructions, with most pairs separated by three or fewer instructions.
- **Proves ICALU boosts ILP** in the BOOM out-of-order core, showing that while the improvement

is negligible for 2-wide cores, it reaches up to 7% in 3- and 4-wide cores, with SPEC integer benchmarks having a theoretical speedup of around 3–5%.

- **Identifies** and analyzes the unavoidable **trade-offs** of bringing ICALU into modern processors, showing that back-end timing, datapath widening and extra register operations introduce costs in modern micro-architectures.

The project started by asking the following question:

What are the benefits and costs of reintroducing an Interlock-Collapsing ALU into contemporary RISC-V processors?

The results of this work show that introducing ICALU in modern RISC-V processors yields a 3–7% performance uplift on integer workloads, at the cost of increasing the delay of back-end stages from issue to commit by the equivalent of accommodating one extra micro-operation in the data-path.

If there is enough slack, ICALU can serve as a viable means to increase microprocessor performance, with more noticeable speed-ups in wider cores. However, these back-end paths are often timing critical and may not provide sufficient slack to accommodate the additional logic, which can diminish or even negate the observed ILP gains. Ultimately, this study presents both the advantages and limitations of integrating ICALU in modern RISC-V cores, leaving it to each implementation and designer to determine whether using the available slack or sacrificing cycle time is a worthwhile trade-off when incorporating ICALU in their core.

6.3. Future Work

The findings of this thesis open several promising directions for further research:

- **3-1 ALU re-evaluation.** IBM claimed ICALU extended the delay of a traditional ALU by one stage by assuming a fixed book-set found across all process nodes. These standard cells were not found in some modern process nodes and this claim can only be made in selected nodes that still have this cells. Further studies should re-evaluate, with a selection of cells found in modern-day process nodes, the equations of a 3-1 ALU that minimally extends the delay of the traditional ALU in order to conclude that ICALU increases the delay of a traditional ALU by one stage across *all* nodes.
- **Physical design analysis.** Perform a complete place-and-route implementation to quantify the timing, area and power impact of ICALU, moving beyond the informed estimates presented here. Such analysis would allow a wider and more accurate trade-off to be established beyond only the timing implications observed.
- **Evaluation in wider and future cores.** Results show that the performance uplift grows with core width, since out-of-order engines become less effective at hiding enforced sequentiality by RAW dependencies in code. There may exist a convergence point where ICALU's benefits scale enough to make it a viable option compared to widening a core or, in general, present it as a more viable and robust way of improving performance. Even though BOOM is the state-of-the-art academic core, modern industry cores are delivered with 6- to 10-wide designs, meaning there is potential for these cores to explore and consider integrating ICALU with different conclusions than the ones seen in this study. Interestingly enough, even though this work aims to revisit ICALU after 30 years, it may take even more years to revisit it and present it as a non-negotiable avenue to increase core performance.

References

[1] Stamatis Vassiliadis, James Phillips, and Bart Blaner. “Interlock collapsing ALU’s”. In: *IEEE Transactions on Computers* 42.7 (1993), pp. 825–839.

[2] Nadeem Malik, Richard J Eickemeyer, and Stamatis Vassiliadis. “Interlock collapsing ALU for increased instruction-level parallelism”. In: *ACM SIGMICRO Newsletter* 23.1-2 (1992), pp. 149–157.

[3] James Phillips and Stamatis Vassiliadis. “High-performance 3-1 interlock collapsing ALU’s”. In: *IEEE Transactions on Computers* 43.3 (1994), pp. 257–268.

[4] Acosta, Kjelstrup, and Torng. “An instruction issuing approach to enhancing performance in multiple functional unit processors”. In: *IEEE Transactions on Computers* 100.9 (1986), pp. 815–828.

[5] Peter M Kogge. *The architecture of pipelined computers*. CRC press, 1981.

[6] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[7] Liliana Margarita Espinosa Jimenez and Michael Opoku Agyeman. “A Study of Techniques to Increase Instruction Level Parallelisms”. In: *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control*. 2018, pp. 1–5.

[8] Krste Asanović and David A Patterson. “Instruction sets should be free: The case for risc-v”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[9] James E Phillips and Stamatis Vassiliadis. “Proof of correctness of high-performance 3—1 interlock collapsing ALUs”. In: *IBM Journal of Research and Development* 37.1 (1993), pp. 12–21.

[10] Andrew Waterman et al. “The RISC-V instruction set manual, volume I: User-level ISA, version 2.0”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54* (2014), p. 4.

[11] SiFive, Inc. *Incredibly Scalable High-Performance RISC-V Core IP*. Blog post introducing the SiFive U8-Series Core IP. Oct. 2019. url: `https://www.sifive.com/blog/incredibly-scalable-high-performance-risc-v-core-ip` (visited on 06/24/2025).

[12] Chen Chen et al. “Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension: Industrial product”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 52–64.

[13] A. Bachrach et al. “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs”. In: *Proc. IEEE Custom Integrated Circuits Conference (CICC)*. 2019, pp. 1–8.

[14] EEMBC. *CoreMark Benchmark*. `https://www.eembc.org/coremark/`. 2012.

[15] David Patterson et al. “Embench IOT 2.0 and DSP 1.0: Modern Embedded Computing Benchmarks”. In: *Computer* 58.5 (2025), pp. 37–47. doi: `10.1109/MC.2024.3511352`.

[16] J. L. Henning. “SPEC CPU2017 Benchmark Suite”. In: *Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2019, pp. 1–2.

[17] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator”. In: *FREENIX Track: 2005 USENIX Annual Technical Conference* (2005).

[18] C. Celio et al. “BOOM: A Portable, Parameterized Out-of-Order RISC-V Core”. In: *4th Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2018, pp. 1–8.

[19] C. Celio. *BOOM Microarchitecture Documentation*. Tech. rep. UCB/EECS-2020-BOOM. EECS Department, University of California, Berkeley, 2020.

[20] Jerry Zhao et al. “Sonicboom: The 3rd generation berkeley out-of-order machine”. In: *Fourth Workshop on Computer Architecture Research with RISC-V*. Vol. 5. International Symposium on Computer Architecture Valencia, Spain. 2020, pp. 1–7.

[21] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. EECS-2016-17. University of California, Berkeley, Electrical Engineering & Computer Sciences, Apr. 2016. url: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.pdf (visited on 06/19/2025).

[22] Wilson Snyder and contributors. *Verilator: Fast Verilog/SystemVerilog Simulator*. Converts Verilog/SystemVerilog designs into cycle-accurate C++ or SystemC models. 2003. url: https://www.veripool.org/verilator/ (visited on 06/20/2025).

[23] STAMATIS VASSILIADIS. “Recursive equations for hardwired binary adders”. In: *International Journal of Electronics Theoretical and Experimental* 67.2 (1989), pp. 201–213.

[24] *SkyWater SKY130 PDK Standard Cell Libraries*. https://skywater-pdk.readthedocs.io/en/main/. SkyWater Technology Foundry, 2020.

[25] Arizona State University and ARM Ltd. *ASAP7 Predictive 7nm FinFET PDK*. https://asap.asu.edu/asap/. 2016.

[26] Smruti R. Sarangi. *Basic Computer Architecture*. Tech. rep. Version 3.07. Department of Computer Science and Engineering, Indian Institute of Technology Delhi, June 2025.

[27] Mircea Vlăduțiu. *Computer arithmetic: algorithms and hardware implementations*. Springer Science & Business Media, 2012.

[28] SiFive. “P870 for Hot Chips: SiFive’s Six-Wide Out-of-Order RISC-V Core”. In: (2023).

[29] Apple Inc. “Apple M4: Wider Decode Execution Engines in New Performance Cores”. In: (2024).

[30] Arm Ltd. *Cortex-X925 Technical Overview*. Tech. rep. 2024.

[31] Jerry Zhao. *ADEPT EoP Party*. Technical Report. 8 pages. University of California, Berkeley, 2025. url: https://adept.eecs.berkeley.edu/wiki/_media/eop/adept-eop-jerry.pdf.

A

Appendix

In this appendix the full statistics in case more detail is desired.

A.1. Trace Analysis detailed Results

These are the detailed collapse rates, distances and percentage mixes for the main trace analysis done, corresponding to Section 3.3.2.

Table A.1: On the left, CoreMark and each Embench benchmark (with application area). On the right, share of collapsible pairs vs. executed instructions. Averages at the bottom treat Embench as a single benchmark using its weighted suite average.

Benchmark	Application Area	Collapse rate [%]	Instr. executed
CoreMark	Embedded core benchmark	11.7	33.18 M
Embench			
aha-mont64	Montgomery multiplication	8.9	1.92 M
crc32	CRC error checking (32-bit)	13.6	3.85 M
cubic	Cubic root solver	5.7	1.11 M
edn	Filter (general)	1.2	3.48 M
huffbench	Compression / decompression	9.6	2.70 M
matmult-int	Integer matrix multiply	1.2	3.27 M
minver	Matrix inversion	9.0	0.46 M
nbody	N-body simulation	3.1	0.07 M
nettle-aes	AES encrypt / decrypt	13.8	5.09 M
nettle-sha256	SHA-256 hash	13.8	4.10 M
nsichneu	Extended Petri net	0.0	2.24 M
picojpeg	JPEG codec	9.8	4.48 M
qrduino	QR codes	17.3	3.51 M
sglib-combined	Generic library algorithms (SGLIB)	7.3	2.72 M
slre	Regex engine	7.0	2.46 M
statemate	State machine (car window)	1.2	0.98 M
st	Statistics	7.1	0.08 M
ud	LU decomposition (integer)	6.1	2.32 M
wikisort	Merge sort	5.3	1.23 M
SPEC CPU2017 int			
600.perlbench	Perl interpreter	5.3	1.29 B
602.gcc	GNU C compiler	5.9	0.05 B
605.mcf	Route planning	4.3	0.88 B
620.omnetpp	Discrete event simulation	8.0	0.43 B
623.xalancbmk	XML to HTML conversion via XSLT	6.3	0.70 B
625.x264	Video compression	15.6	0.49 B
631.deepsjeng	AI: alpha–beta tree search (Chess)	10.9	1.40 B
641.leela	AI: Monte Carlo tree search (Go)	14.3	1.43 B
648.exchange2	AI: recursive solution generator	1.6	2.10 B
657.xz	General data compression	13.4	0.43 B
Average unweighted †		**8.9**	
Average weighted †		**7.8**	

† Averages treat Embench as a single benchmark, using its weighted suite average instead of the average of every Embench benchmark.

Table A.2: Distribution of instruction distance between producer and consumer in the collapse pair (share of all collapses within the benchmark). A distance of zero means the two operations sit back to back in program order; higher distances imply additional instructions in between.

Benchmark	Dist 0	Dist 1	Dist 2	Dist 3	Dist 4	Dist 5	Dist 6	Dist 7
CoreMark								
CoreMark	53.01%	25.45%	20.79%	0.07%	0.51%	0.09%	0.01%	0.07%
Embench								
aha-mont64	98.75%	0.00%	0.00%	0.00%	0.00%	0.75%	0.50%	0.00%
crc32	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
cubic	57.07%	27.84%	6.76%	4.15%	0.70%	3.49%	0.00%	0.00%
edn	33.63%	10.86%	38.10%	6.55%	10.65%	0.20%	0.00%	0.00%
huffbench	62.48%	36.77%	0.01%	0.41%	0.01%	0.15%	0.16%	0.00%
matmult-int	4.00%	4.90%	90.73%	0.00%	0.24%	0.12%	0.00%	0.00%
minver	33.34%	20.03%	14.65%	26.64%	0.00%	4.00%	1.33%	0.01%
nbody	9.53%	88.64%	0.35%	0.17%	1.31%	0.00%	0.00%	0.00%
nettle-aes	4.73%	1.34%	7.17%	41.67%	19.05%	5.55%	8.45%	12.04%
nettle-sha256	12.77%	16.47%	18.57%	9.16%	16.47%	11.09%	7.90%	7.56%
nsichneu	69.57%	26.09%	4.35%	0.00%	0.00%	0.00%	0.00%	0.00%
picojpeg	80.39%	6.66%	8.26%	0.29%	1.32%	2.25%	0.00%	0.84%
qrduino	76.79%	17.53%	0.08%	5.13%	0.05%	0.40%	0.00%	0.02%
sglib-combined	71.83%	24.84%	2.18%	0.78%	0.00%	0.02%	0.36%	0.00%
slre	92.32%	0.91%	0.19%	5.87%	0.00%	0.45%	0.26%	0.00%
statemate	50.03%	33.33%	0.00%	16.62%	0.00%	0.00%	0.00%	0.00%
st	50.46%	0.05%	0.02%	0.00%	49.47%	0.00%	0.00%	0.00%
ud	39.59%	23.96%	33.33%	2.08%	0.00%	1.04%	0.00%	0.00%
wikisort	31.09%	6.70%	1.30%	60.74%	0.00%	0.17%	0.00%	0.00%
SPEC CPU2017 int								
600.perlbench	79.11%	9.73%	5.97%	3.34%	0.25%	0.31%	0.93%	0.35%
602.gcc	75.51%	8.53%	6.50%	7.24%	0.90%	0.52%	0.21%	0.60%
605.mcf	57.93%	29.21%	7.39%	0.01%	0.00%	4.63%	0.83%	0.00%
620.omnetpp	52.09%	19.17%	15.91%	12.33%	0.38%	0.08%	0.01%	0.03%
623.xalancbmk	31.06%	34.91%	33.18%	0.20%	0.01%	0.24%	0.38%	0.01%
625.x264	26.94%	39.50%	11.36%	6.50%	9.84%	1.55%	2.98%	1.33%
631.deepsjeng	50.84%	21.20%	13.34%	8.23%	2.83%	3.02%	0.51%	0.03%
641.leela	55.78%	16.03%	4.28%	11.05%	5.50%	4.41%	1.27%	1.67%
648.exchange2	58.99%	11.23%	2.52%	17.48%	6.26%	1.16%	0.21%	2.16%
657.xz	47.68%	36.01%	4.32%	5.96%	3.01%	0.58%	2.06%	0.39%
Average	**52.24%**	**19.93%**	**11.72%**	**8.42%**	**4.29%**	**1.54%**	**0.95%**	**0.91%**

Table A.3: Producer to consumer group mix per benchmark. Each entry: share of total collapses inside the benchmark.

Benchmark	add-add	add-load	add-logic	add-store	logic-add	logic-load	logic-logic	logic-store
CoreMark								
CoreMark	1.86%	50.74%	32.78%	1.64%	0.71%	0.00%	12.27%	0.00%
Embench								
aha-mont64	1.99%	0.00%	0.25%	0.01%	1.49%	0.00%	96.26%	0.00%
crc32	33.33%	33.33%	0.00%	0.00%	0.00%	0.00%	33.33%	0.00%
cubic	50.73%	0.00%	23.49%	10.72%	13.81%	0.00%	1.25%	0.00%
edn	33.61%	45.47%	0.00%	20.91%	0.00%	0.00%	0.00%	0.00%
huffbench	32.22%	35.02%	0.00%	32.72%	0.03%	0.00%	0.00%	0.00%
matmult-int	4.46%	90.96%	0.00%	4.57%	0.00%	0.00%	0.00%	0.00%
minver	43.98%	28.01%	0.00%	28.00%	0.00%	0.00%	0.00%	0.00%
nbody	0.66%	88.81%	0.00%	10.45%	0.09%	0.00%	0.00%	0.00%
nettle-aes	0.73%	51.48%	0.13%	0.18%	10.24%	0.00%	37.23%	0.00%
nettle-sha256	21.60%	1.09%	0.17%	1.60%	25.97%	0.00%	49.58%	0.00%
nsichneu	30.43%	17.39%	0.00%	43.48%	8.70%	0.00%	0.00%	0.00%
picojpeg	37.41%	5.65%	2.58%	28.69%	12.83%	0.00%	12.83%	0.00%
qrduino	27.14%	20.93%	32.19%	0.12%	5.75%	0.00%	13.88%	0.00%
sglib-combined	58.84%	19.53%	4.50%	17.11%	0.02%	0.00%	0.00%	0.00%
slre	47.58%	34.53%	5.62%	10.85%	0.71%	0.00%	0.71%	0.00%
statemate	16.70%	66.51%	0.00%	16.73%	0.02%	0.00%	0.04%	0.00%
st	49.61%	0.51%	0.00%	49.84%	0.04%	0.00%	0.00%	0.00%
ud	12.50%	80.19%	0.00%	7.30%	0.00%	0.00%	0.00%	0.00%
wikisort	71.35%	2.74%	1.39%	12.41%	0.63%	0.00%	11.48%	0.00%
SPEC CPU2017 int								
600.perlbench	20.94%	32.56%	9.43%	23.14%	7.93%	0.00%	5.99%	0.00%
602.gcc	20.66%	31.30%	7.75%	23.99%	10.73%	0.00%	5.57%	0.00%
605.mcf	70.22%	20.19%	0.01%	4.63%	4.94%	0.00%	0.00%	0.00%
620.omnetpp	32.08%	26.97%	2.47%	7.71%	15.41%	0.93%	14.43%	0.00%
623.xalancbmk	34.77%	18.75%	2.80%	43.17%	0.41%	0.00%	0.10%	0.00%
625.x264	49.54%	23.38%	6.16%	9.94%	10.79%	0.00%	0.20%	0.00%
631.deepsjeng	29.41%	29.18%	11.32%	6.48%	9.71%	0.01%	13.88%	0.00%
641.leela	13.90%	67.91%	0.09%	6.82%	0.34%	0.00%	10.94%	0.00%
648.exchange2	57.33%	33.45%	0.07%	8.33%	0.81%	0.00%	0.02%	0.00%
657.xz	26.27%	50.86%	2.27%	2.91%	8.68%	0.00%	9.01%	0.00%
Average	**31.06%**	**33.58%**	**4.85%**	**14.48%**	**5.03%**	**0.03%**	**10.97%**	**0.00%**

A.2. Baseline vs ICALU Full Dump Results

Table A.4: Embench results for BOOM 2-wide with and without ICALU, shown in million of instructions. Speedup is relative percent; collapse success the effectiveness of exploiting collapsesg.

Benchmark	Baseline Score	ICALU Score	Speedup (%)	Succ. Rate (%)
aha-mont64	1.29	1.27	0.90	70.6
crc32	2.61	2.52	3.44	77.6
cubic	1.64	1.64	0.11	67.6
edn	1.85	1.84	0.22	66.0
huffbench	2.60	2.51	3.14	75.3
matmult-int	2.99	2.98	0.16	65.8
minver	0.38	0.38	0.93	70.5
nbody	0.23	0.23	0.09	66.5
nettle-aes	2.81	2.81	0.27	70.0
nettle-sha256	2.38	2.35	1.39	72.1
nsichneu	3.57	3.57	0.00	65.0
picojpeg	2.89	2.78	3.84	76.6
qrduino	3.54	3.50	1.29	75.0
sglib-combined	2.77	2.72	1.91	71.9
slre	1.63	1.62	0.66	69.3
st	0.12	0.12	0.09	68.3
statemate	1.23	1.23	0.01	65.5
ud	2.45	2.41	1.46	70.5
wikisort	1.24	1.20	2.73	72.6
total	**38.22**	**37.69**	**1.38**	**71.0**

Table A.5: Embench results for BOOM 3-wide with and without ICALU, shown in million of instructions. Speedup is relative percent; collapse success the effectiveness of exploiting collapsesg.

Benchmark	Baseline Score	ICALU Score	Speedup (%)	Succ. Rate (%)
aha-mont64	0.93	0.93	0.59	78.5
crc32	2.63	2.46	6.79	84.8
cubic	1.42	1.41	0.77	77.2
edn	1.28	1.24	2.81	77.0
huffbench	2.15	1.98	7.83	84.0
matmult-int	2.54	2.54	0.02	76.0
minver	0.29	0.29	2.23	79.5
nbody	0.23	0.23	0.02	76.0
nettle-aes	1.96	1.94	0.95	79.5
nettle-sha256	1.72	1.64	4.65	82.0
nsichneu	2.51	2.51	0.00	75.0
picojpeg	2.31	2.08	9.99	87.5
qrduino	2.85	2.82	0.93	82.5
sglib-combined	2.11	2.07	2.01	79.0
slre	1.10	1.03	5.99	81.8
st	0.12	0.12	0.00	77.8
statemate	1.17	1.17	0.15	75.3
ud	2.13	2.03	4.83	80.5
wikisort	1.05	1.00	5.22	80.5
total	**30.50**	**29.47**	**3.38**	**80.0**

Table A.6: Embench results for BOOM 4-wide with and without ICALU, shown in million of instructions. Speedup is relative percent; collapse success the effectiveness of exploiting collapsesg.

Benchmark	Baseline Score	ICALU Score	Speedup (%)	Succ. Rate (%)
aha-mont64	0.91	0.90	1.06	81.0
crc32	2.51	2.35	6.37	87.8
cubic	1.38	1.38	0.35	79.5
edn	1.03	1.01	1.56	78.5
huffbench	1.85	1.72	6.94	86.2
matmult-int	2.23	2.23	0.30	77.2
minver	0.26	0.25	2.06	81.5
nbody	0.23	0.23	0.31	77.8
nettle-aes	1.49	1.47	1.67	82.0
nettle-sha256	1.26	1.18	5.82	85.0
nsichneu	2.51	2.51	0.00	76.0
picojpeg	1.76	1.59	9.66	85.5
qrduino	2.76	2.72	1.42	84.8
sglib-combined	1.86	1.82	1.99	81.0
slre	0.91	0.88	4.10	82.8
st	0.12	0.12	0.17	79.0
statemate	1.08	1.08	0.25	76.8
ud	1.97	1.86	5.80	83.5
wikisort	0.78	0.74	5.06	82.8
total	**26.89**	**26.02**	**3.23**	**82.6**