Instruction Flow Amplifier: A Novel Architecture for In-Order Processor Front-End Optimization

João Antônio Temochko Andre
Department of Electrical Engineering
Centro Universitário FEI
São Bernardo do Campo, Brazil
joao.temochko@ifsp.edu.br

Alexandre Beletti Ferreira
Department of Informatics
Federal Institute of Sao Paulo
Sao Paulo, Brazil
higuita@ifsp.edu.br

***Abstract*—This paper proposes the Instruction Flow Amplifier (IFA), a novel architecture designed to mitigate front-end bottlenecks in in-order processor cores. Using the Gem5 simulator, we evaluate Cycles Per Instruction (CPI), Instructions Per Cycle (IPC), branch prediction accuracy, and instruction cache misses to demonstrate front-end bottlenecks and quantify the performance gains achieved by the proposed architecture. Unlike traditional trace caches that primarily focus on instruction storage, the proposed IFA integrates a lightweight Dependency Analysis Unit (DAU) to enable pseudo-out-of-order dispatch within an energy-efficient in-order infrastructure. The experimental results demonstrate that the IFA acts as an effective instruction supply accelerator. In workloads dominated by tight loops (MatMult), the proposed architecture effectively saturates the pipeline, achieving a 28% IPC improvement. In control-heavy workloads (WikiSort), the IFA mitigates fetch latency through aggressive buffering, improving IPC by 13.4% with negligible impact on branch prediction accuracy.**

***Index Terms*—Computer Architecture, RISC-V, ARM, Front-End, Gem5, In-Order Processor cores.**

I. Introduction

Instruction fetch remains a fundamental performance bottleneck in modern processors [1]. In-order processor cores are particularly vulnerable to stalls, which completely halt execution when dependent instructions are encountered [2], [3]. Additionally, branch mispredictions and control dependencies fragment the instruction stream [2], while data dependencies serialize execution despite the presence of instruction-level parallelism.

Contemporary processor cores address these limitations through larger caches and wider fetch mechanisms, increasing complexity and power consumption without proportional performance gains [1], [4].

The performance of a processor, regardless of the efficiency of its execution core (back-end), is fundamentally limited by the rate at which useful instructions can be supplied. The instruction fetch subsystem, or front-end, is responsible for supplying the pipeline with a continuous stream of instructions while anticipating program control flow to minimize functional-unit idle time [1], [5].

Main memory access latency remains one of the primary obstacles to modern processor performance. In the front-end context, an instruction cache miss (L1-I miss) can completely stall the processor pipeline, exposing the core to hundreds of idle cycles [6]. To mitigate this effect, speculative prefetching techniques have become ubiquitous, aiming to load instruction blocks into the cache before they are required by the execution flow [1], [7], [8].

One of the most prominent approaches for mitigating memory-access latency is the *Fetch Target Queue (FTQ)*, originally proposed by Reinman et al. [5]. Instead of coupling branch prediction directly to instruction-cache access, the FTQ architecture allows the branch predictor to run ahead, generating target addresses and storing them in a queue.

In addition to the FTQ, which stores target addresses, advanced architectures employ *Instruction Byte Buffers (IBBs)* to temporarily store fetched instructions prior to decoding. In designs such as the Decoded Instruction Architecture (DIA) proposed by Santana et al., this concept is extended to store pre-decoded instructions or micro-operations, thereby removing decoding latency from the critical execution path [9].

Despite these latency benefits, most decoupled architectures, including FTQ-based designs, manage their buffers as strict FIFO (First-In, First-Out) queues. This introduces a critical limitation known as head-of-line blocking.

As characterized by Chacon and Gino, when the instruction at the head of the queue stalls due to a cache miss (Stalling Head Instruction), the entire instruction stream is interrupted, even if valid instructions are available in later queue positions [8]. In fully *out-of-order (OoO)* processors, this issue is mitigated through large instruction windows and register renaming. However, in in-order or embedded processors, where such structures are prohibitively expensive, the FIFO model renders decoupling ineffective under long-latency data dependencies.

This paper proposes the *Instruction Flow Amplifier (IFA)*, a lightweight front-end component that implements real-time register dependency analysis. Instead of relying on conservative scheduling policies, the IFA identifies independent instruction sets within an instruction window, enabling more efficient dispatch. The proposed design is inspired by classical instruction scheduling theory and out-of-order execution principles, adapted for practical single-core implementations.

The proposed architecture aims to answer the following research questions (RQs):

Manuscript received May 11, 2026; revised May 26, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.139

- RQ1: Can the IPC of in-order processors be increased through front-end optimization?
- RQ2: Which additional performance parameters can be improved by the proposed solution?
- RQ3: Does the proposed solution incur lower energy overhead compared to strategies?

The remainder of this paper is organized as follows: Section II presents the related work, Section III presents the architectural concept, Section IV presents the performance analysis, Section V discusses the performance and speedup metrics, Section VI evaluates the energy efficiency, and Section VII concludes the paper.

II. Related Works

A. Fetch Target Queue (FTQ)

Originally proposed by Reinman et al. [5], the FTQ serves as an interface between branch prediction and the memory subsystem. Because the bandwidth and operating speed of the branch predictor, which manipulates only target addresses, are typically higher than those of the instruction cache, the FTQ enables deeper control-flow lookahead. Previous studies have shown that FTQ-based architectures can better tolerate instruction cache misses (L1-I misses), since the queue maintains pending fetch requests ready for processing as soon as memory access becomes available [8], [10].

B. Instruction Byte Buffer (IBB) and Pre-Decoding

In addition to the FTQ, which stores target addresses, advanced architectures employ Instruction Byte Buffers (IBBs) to temporarily hold fetched instructions prior to decoding.

These structures allow in-order processors to maintain high functional-unit utilization, provided that the instruction buffer does not become empty. Oberoi argues that a truly high-performance front-end should be capable of fetching instruction fragments out of program order to maximize memory bandwidth utilization, although sequential reassembly is typically required before dispatch in conventional architectures [11].

C. Latency Tolerance: Runahead and iCFP

One approach for mitigating the impact of long-latency cache misses (L2/L3 misses) is runahead execution. Originally proposed as a mechanism to reduce the complexity of large instruction windows, this technique allows the processor to continue speculative execution after a blocking event, such as a load miss. During this phase, data dependencies are temporarily ignored to generate prefetches for future instructions and data. At the end of the blocking period, the speculative state is discarded and normal execution resumes [12]. Although effective at tolerating memory latency, runahead execution wastes energy by executing instructions whose results are eventually discarded.

D. Front-end Execution Architecture (FXA)

Shioya et al. proposed a distinct architectural approach known as the Front-end Execution Architecture (FXA). FXA introduces a small and energy-efficient in-order execution unit (IXU) immediately after the fetch stage, preceding a conventional out-of-order execution unit (OXU) [13].

The premise of FXA is that a significant portion of instructions (more than 50%) can be executed immediately in order if their operands are ready, thereby avoiding the complex logic associated with dynamic scheduling. Instructions that cannot be executed in the IXU are forwarded as NOPs to the OXU.

Although FXA improves the energy efficiency of high-performance processors by reducing pressure on OoO scheduling logic, it still depends on the presence of a complex back-end to handle unresolved dependencies and speculative execution scenarios. This differs from the proposed IFA architecture, which aims to optimize a purely in-order core [13].

E. The difference between IFA and other Architectures

Unlike the previously discussed approaches, the Instruction Flow Amplifier (IFA) transforms the decoupling buffer into an inspectable dispatch structure rather than a strict FIFO queue. This design enables the mitigation of head-of-line blocking without incurring the complexity and energy cost of a fully out-of-order core, addressing a gap identified in the literature on decoupled front-end architectures.

The IFA operates strictly within a bounded front-end instruction window. Unlike iCFP, the proposed architecture does not require complex re-execution buffers, and unlike FXA, it does not depend on a secondary out-of-order execution engine. Instead, the IFA exploits the visibility provided by the buffered instruction window to perform pseudo-out-of-order dispatch while locally resolving RAW (Read-After-Write) hazards, thereby maintaining high functional-unit utilization.

III. Architectural Concept

The proposed Instruction Flow Amplifier (IFA) acts as an intelligent front-end scheduler positioned between the fetch and decode stages of an in-order processor core. Unlike traditional trace caches, which are primarily designed to store execution paths, the IFA employs a dynamic instruction window to identify and schedule independent instructions, thereby enabling pseudo-out-of-order execution behavior within an energy-efficient in-order pipeline.

The proposed architecture consists of four main components: the Front-End Buffer (FEB), the Dependency Analysis Unit (DAU), the Dispatcher Unit (DU), and the Dynamic Barrier Synchronization (DBS) mechanism.

A. Front-End Buffer (FEB)

The Front-End Buffer (FEB) serves as a decoupling structure that mitigates latency between the instruction fetch stage and the execution back-end. Functionally, the FEB operates as a double-ended queue (deque) rather than a strictly ordered FIFO structure.

The buffer maintains a sliding window of fetched instructions, typically ranging from 16 to 32 entries. By buffering multiple instructions, the IFA exposes a deeper lookahead window to the control logic, enabling the hardware to analyze instructions beyond the head of the queue.

This structure absorbs front-end stalls caused by cache misses, ensuring a continuous supply of candidate instructions for dispatch.

B. Dependency Analysis Unit (DAU)

The Dependency Analysis Unit (DAU) is the central control structure responsible for ensuring data dependency correctness. It dynamically scans the instructions residing in the FEB to detect Read-After-Write (RAW) hazards.

At every clock cycle, the DAU performs the following operations:

- **Operand Comparison:** The DAU compares the source registers of younger instructions against the destination registers of older, unexecuted instructions within the instruction window.
- **Status Tagging:** Each instruction in the buffer is assigned a status flag (e.g., Ready or Stalled). An instruction is marked as Stalled if it depends on a result that has not yet been produced by the execution units.
- **Non-Blocking Selection:** By identifying independent instructions, the DAU enables the dispatcher to bypass stalled instructions and issue younger ready-to-execute operations, thereby hiding execution latency.

C. Dispatcher Unit (DU)

The Dispatcher Unit (DU) selects instructions from the FEB and forwards them to the execution pipeline. To preserve architectural correctness without requiring complex speculative recovery mechanisms, the DU employs a set of lightweight control barriers.

- **Selection Logic:** The dispatcher scans the instruction window for operations marked as Ready by the DAU and issues them according to the superscalar width supported by the processor core.
- **Control Barriers:** When a branch or jump instruction is encountered, the dispatcher imposes a soft synchronization barrier. Instructions located after the unresolved control-flow operation are temporarily prevented from being dispatched until the branch outcome is resolved. This mechanism avoids unsafe speculative execution while still allowing independent instructions preceding the branch to be reordered safely.

D. Dynamic Barrier Synchronization (DBS)

To preserve architectural correctness while avoiding the latency overhead associated with multi-phase control-state machines, the IFA employs a combinational control mechanism referred to as Dynamic Barrier Synchronization (DBS).

Instead of transitioning between explicit "performance" and "drain" operating modes, the DAU and DU cooperate dynamically to enforce strict instruction ordering only when required. Whenever a critical instruction is detected within the instruction window, such as a system call, atomic operation, or unresolved branch, the control logic immediately establishes a soft synchronization barrier.

This barrier prevents younger instructions from bypassing the critical operation. Consequently, pseudo-out-of-order dispatch is temporarily suspended, and older instructions are drained sequentially until the critical instruction reaches the head of the queue and is safely dispatched.

Once the critical operation is committed to the execution back-end, the barrier is automatically removed, restoring full dispatch capability without introducing additional pipeline state transitions or recovery latency.

IV. Performance Analysis

To evaluate the performance gains provided by the proposed IFA architecture, we used the Gem5 simulator [14] and benchmark suites from MiBench [15] and Embench to compare an in-order processor core (MinorCPU) operating with and without the IFA extension.

The evaluation considers Cycles Per Instruction (CPI), Instructions Per Cycle (IPC), instruction cache misses, branch prediction accuracy, and pipeline stall behavior.

The experimental results demonstrate measurable performance improvements across all evaluated workloads. Performance gains ranged from approximately 3% to 8% for MiBench workloads and reached up to 28% for selected Embench kernels. These improvements reduced execution stalls and increased effective pipeline utilization, contributing to higher overall energy efficiency.

A. The Gem5 Configuration

In the Gem5 simulator, we modified the MinorCPU implementation to emulate an in-order processor enhanced with the proposed IFA mechanism. The primary architectural modification was introduced in the Fetch2 stage. This design decision was motivated by three architectural constraints:

- **Availability of Semantic Information:** The DAU requires access to source and destination register identifiers to detect Read-After-Write (RAW) hazards. In the baseline MinorCPU architecture, Fetch1 performs raw cache-line retrieval, while instruction decoding occurs in Fetch2. Therefore, Fetch2 represents the earliest pipeline stage at which semantic information required by dependency analysis becomes available.
- **Front-End Decoupling:** The IFA Buffer acts as an instruction reservoir capable of absorbing latency bubbles caused by instruction cache misses. By intercepting decoded instructions immediately after Fetch2 and before dispatch, the IFA accumulates a pool of executable instructions, allowing the execution back-end to continue operation even when Fetch1 stalls waiting for memory.
- **Control-Flow Resolution:** Branch prediction validation and target calculation occur within Fetch2. Since the IFA implements control barriers to avoid unsafe speculative

dispatch, it must remain tightly coupled with branch-resolution logic to correctly halt and resume instruction issue based on control-flow resolution.

Consequently, the Fetch2 stage was modified to redirect decoded instructions into the FEB instead of forwarding them directly to the decode/dispatch queue, effectively transforming the baseline pipeline into a decoupled fetch-execute architecture.

The central implementation changes include modifications to the `hasDependency()` function, which detects register dependencies, and to the `evaluate()` function, which controls IFA activation and emulates the scheduling differences between the baseline and enhanced architectures.

B. Workload

Following methodologies adopted in prior work, we selected benchmark applications from MiBench and Embench to evaluate the proposed architecture. The workloads were divided into three categories: mathematical computation, validation, and sorting.

Mathematical Computation: We used BasicMath from MiBench, which performs computationally intensive mathematical operations such as cubic-function evaluation, integer square-root calculation, and angle conversion. We also evaluated MatMult from Embench, which performs dense matrix multiplication over non-zero matrix elements.

Validation: This category includes Blowfish from MiBench and CRC32 from Embench. Blowfish performs symmetric encryption and decryption using logical operations such as XOR and AND, while CRC32 computes cyclic redundancy checks using XOR and bit-shift operations.

Sorting: We evaluated WikiSort from Embench, which implements block merge sorting designed to operate efficiently without requiring significant auxiliary memory allocation.

C. Results

The experimental results demonstrate the effectiveness of the proposed IFA architecture across diverse workloads and instruction set architectures.

As shown in Tables I to X, performance improvements were consistently observed regardless of the target ISA. CPI, stall counts, and instruction-cache misses generally decreased, while IPC increased. These gains were achieved solely through front-end modifications, without altering execution back-end structures.

For example, in the BasicMath benchmark(Tables I and II), CPI was reduced by 7.53% in the RISC-V configuration, while IPC increased by 8.14%. Similarly, Blowfish exhibited a CPI reduction of 4.65% and an IPC improvement of 4.88%.

The most significant improvements were observed in newer Embench workloads optimized for embedded systems. These benchmarks highlight the effectiveness of the IFA under highly structured instruction-level parallelism.

In the MatMult benchmark (Tables III and IV), CPI was reduced by 22.3% in RISC-V and 13.57% in ARM while IPC increased by 28.8% in the RISC-V ISA and 15.70% in the ARM ISA.

TABLE I
RISC-V PERFORMANCE ANALYSIS (BASICMATH)

Metric	Base	IFA	Diff (%)
CPI	1.68	1.56	**-7.53%**
IPC	0.59	0.64	**+8.14%**
Stalls	66.5M	55.4M	**-16.73%**
ICache Miss	4.02M	3.81M	-5.23%
Branch Acc.	94.6%	93.5%	-1.21%
Mispredicts	0.95M	1.24M	+30.43%

TABLE II
ARM AARCH64 PERFORMANCE ANALYSIS (BASICMATH)

Metric	Base	IFA	Diff (%)
CPI	1.99	1.84	**-7.78%**
IPC	0.50	0.54	**+8.44%**
Stalls	66.6M	58.3M	**-12.44%**
ICache Miss	4.27M	4.09M	-4.29%
Branch Acc.	93.5%	93.9%	+0.36%
Mispredicts	0.90M	0.90M	+0.01%

In CRC32 (Tables V and VI), CPI decreased by 17.82% in RISC-V and 18.14% in ARM, IPC improved by 21.68% in RISC-V and 22.16% in ARM.

WikiSort (Tables VII and VIII) also demonstrated substantial gains, with CPI reduced by 11.84% in RISC-V and 13.57% in ARM, IPC increasing by 13.43% in RISC-V and 15.70% in ARM.

The Blowfish benchmark (Tables IX and X) showed smaller gains, the CPI reduced 6.01% in RISC-V and 4.65% in ARM, the IPC gains is 6.39% in RISC-V and 4.88% in ARM.

This tests demonstrated that IFA achieves substantial gains in benchmarks on both RISC-V and ARM ISAs, showing that the gains are ISA-independent. The results reveal that IFA excels at matrix multiplication but yields minimal gains in cryptography algorithms. This illustrates precisely how IFA boosts performance when benchmarks show lower instruction dependency, yet fails to improve when tests exhibit high dependency, is the case with Blowfish or BasicMath.

The results indicate that the IFA effectively mitigates front-end stalls, allowing the processor pipeline to sustain higher throughput while preserving architectural correctness. The improvements are particularly pronounced in workloads exhibiting predictable dependency structures and moderate branch-control complexity.

V. PERFORMANCE AND SPEEDUP METRICS

To quantify the overall performance gains achieved by the proposed architecture, we analyze the observed improvements using Amdahl's Law [16], [17].

The overall speedup of a processor is determined by the fraction of execution time affected by a given optimization and by the magnitude of the acceleration applied to that portion of the system. Since the IFA exclusively optimizes front-end behavior, the maximum achievable speedup remains bounded

TABLE III
RISC-V PERFORMANCE ANALYSIS (MATMULT)

Metric	Base	IFA	Diff (%)
CPI	1.33	1.03	**-22.3%**
IPC	0.75	0.97	**+28.8%**
Stalls	66.6M	58.3M	**-12.44%**
ICache Miss	448	449	0.00%
Branch Acc.	95,10%	95.32%	+0.22%
Mispredicts	1.95M	1.94M	-0.01%

TABLE IV
ARM AARCH64 PERFORMANCE ANALYSIS (MATMULT)

Metric	Base	IFA	Diff (%)
CPI	1.40	1.21	**-13.57%**
IPC	0.72	0.83	**+15.70%**
Stalls	137.42M	71.68M	**-47.84%**
ICache Miss	546	546	0.00%
Branch Acc.	95.04%	95.04%	0.00%
Mispredicts	1.95M	1.95M	0.00%

TABLE V
RISC-V PERFORMANCE ANALYSIS (CRC32)

Metric	Base	IFA	Diff (%)
CPI	1.22	1.00	**-17.82%**
IPC	0.82	1.00	**+21.68%**
Stalls	87.56M	579.32K	**-99.34%**
ICache Miss	411	420	+2.19%
Branch Acc.	99.97%	99.97%	0.00%
Mispredicts	17.52K	17.52K	-0.03%

TABLE VI
ARM AARCH64 PERFORMANCE ANALYSIS (CRC32)

Metric	Base	IFA	Diff (%)
CPI	1.30	1.06	**-18.14%**
IPC	0.77	0.94	**+22.16%**
Stalls	87.58M	17.97M	**-79.48%**
ICache Miss	522	528	+1.15%
Branch Acc.	99.97%	99.97%	0.00%
Mispredicts	17.56K	17.55K	-0.03%

TABLE VII
RISC-V PERFORMANCE ANALYSIS (WIKISORT)

Metric	Base	IFA	Diff (%)
CPI	1.24	1.09	**-11.84%**
IPC	0.81	0.91	**+13.43%**
Stalls	8.73M	3.40M	**-61.05%**
ICache Miss	505	516	+2.18%
Branch Acc.	96.81%	96.85%	+0.05%
Mispredicts	229.18K	225.94K	-1.41%

by the fraction of execution time spent stalled in the instruction supply pipeline.

The speedup model is expressed as follows:

$$S_{\text{speedup}} = \frac{1}{(1 - f_{\text{FE}}) + \frac{f_{\text{FE}}}{S_{\text{IFA}}}} \tag{1}$$

TABLE VIII
ARM AARCH64 PERFORMANCE ANALYSIS (WIKISORT)

Metric	Base	IFA	Diff (%)
CPI	1.36	1.17	**-13.57%**
IPC	0.74	0.85	**+15.70%**
Stalls	12.74M	6.18M	**-51.51%**
ICache Miss	636	643	+1.10%
Branch Acc.	96.34%	96.34%	0.00%
Mispredicts	229.65K	229.75K	+0.04%

TABLE IX
RISC-V PERFORMANCE ANALYSIS (BLOWFISH)

Metric	Base	IFA	Diff (%)
CPI	2.81	2.64	**-6.01%**
IPC	0.36	0.38	**+6.39%**
Stalls	210.60K	190.97K	**-9.32%**
ICache Miss	505	509	+0.79%
Branch Acc.	98.03%	98.05%	+0.02%
Mispredicts	579	573	-1.04%

TABLE X
ARM AARCH64 PERFORMANCE ANALYSIS (BLOWFISH)

Metric	Base	IFA	Diff (%)
CPI	3.01	2.87	**-4.65%**
IPC	0.33	0.35	**+4.88%**
Stalls	180.73K	168.16K	**-6.96%**
ICache Miss	592	586	-1.01%
Branch Acc.	98.05%	98.05%	0.00%
Mispredicts	589	589	0.00%

where:

- f_{FE} represents the fraction of execution time limited by front-end stalls;
- S_{IFA} represents the acceleration factor provided by the IFA for front-end operations.

To contextualize the measured improvements, we apply Amdahl's formulation to the empirical CPI reductions obtained during simulation.

The observed overall speedup is computed as:

$$\text{Speedup} = \frac{T_{\text{baseline}}}{T_{\text{IFA}}} = \frac{1}{1 - \text{TimeReduction}} \approx 1.081 \tag{2}$$

Using the BasicMath benchmark in the RISC-V configuration, the IFA reduced CPI by approximately 7.53%, corresponding to an overall speedup of nearly 1.08.

This result suggests that front-end stalls accounted for approximately 7.5% of the total execution time, validating the initial bottleneck hypothesis predicted by Amdahl's Law.

The results further demonstrate that relatively small front-end optimizations can produce measurable system-wide performance improvements, even without modifications to execution back-end structures such as ALUs, reorder buffers, or cache hierarchies.

Moreover, the measured gains indicate that the instruction supply subsystem remains a critical limiting factor in in-

order architectures, particularly in workloads exhibiting high instruction-level parallelism and frequent front-end stalls.

VI. Energy Efficiency

To validate the Energy Efficiency we use the concept of *Race to Halt* [16] to prove the IFA is better to FIFO (First In, First Out) front-end, many found in in-order processor's [3], [18].

We increased the processor's total power by adding the IFA. Clearly, FEB, DAU, and DU will consume more energy when the instruction comes from memory. However, with IPC gains and a decrease in CPI, stalls, and ICache misses, we managed to make processing faster, thus reducing it by approximately 6.5% for RISC-V and 7.5% for ARM (using metrics from the two benchmarks used).

To rigorously evaluate the energy benefits of the Instruction Flow Amplifier (IFA), we adopt a conservative "Race-to-Halt" model. We posit that the addition of the IFA Buffer and Dependency Analysis Unit (DAU) introduces a power overhead due to the extra transistor count and switching activity of the comparators.

"Previous implementations of front-end enhancements, such as Execution Drafting and the Front-end Execution Architecture (FXA), have demonstrated area overheads ranging from approximately 1% to 2.7% of the core area [13], [19].

Based on literature regarding lightweight out-of-order extensions for in-order cores which reports a 2.4% [20] area overhead for similar issue logic. We model a pessimistic scenario where the IFA increases the core's active power consumption (P_{active}) by **3.5%**.

This approach aligns with the 'Race-to-Halt' strategy described by Miyoshi et al. [21], where execution time reduction is prioritized to maximize the duration of low-leakage sleep states.

The net energy savings (ΔE) are calculated by comparing the baseline energy against [16] the IFA energy, which benefits from reduced execution time (T_{exec}) but suffers from the power penalty ($P_{\text{penalty}} = 1.035 \times P_{\text{base}}$):

$$\Delta E = \left(1 - \frac{(1.035 \times T_{\text{IFA}}) + (P_{\text{idle}} \times T_{\text{saved}})}{1.0 \times T_{\text{base}}}\right) \times 100 \quad (3)$$

1) Results Discussion: Using the empirical data from the BasicMath benchmark (RISC-V), where the IFA reduced execution time by **7.53%** ($T_{\text{IFA}} \approx 0.925 T_{\text{base}}$):

$$E_{\text{IFA}} \approx (1.035 \times 0.925) + (0.1 \times 0.075) \approx 0.965 \quad (4)$$

$$\Delta E \approx 1.0 - 0.965 = \mathbf{3.5}\% \quad (5)$$

Despite the modeled 3.5% power overhead, the system achieves a **3.5% net reduction in total energy**. This confirms that the performance gains from mitigating front-end stalls outweigh the cost of the additional logic, validating the IFA as an energy-efficient architectural enhancement.

This result demonstrates that the performance speedup provided by the IFA is sufficient to offset its logic overhead. By allowing the core to enter the sleep state earlier, the architecture effectively converts microarchitectural throughput gains into tangible battery life extension, validating the Race-to-Halt approach for this design [22], [23].

VII. Conclusion

This paper presented the Instruction Flow Amplifier (IFA), a lightweight front-end architecture that introduces pseudo-out-of-order dispatch capabilities into purely in-order processor cores without requiring modifications to the execution back-end.

By combining a Front-End Buffer (FEB), a real-time Dependency Analysis Unit (DAU), and a Dynamic Barrier Synchronization (DBS) mechanism, the proposed architecture mitigates the head-of-line blocking problem commonly found in FIFO-based decoupled front-end designs.

RQ1 — Can the IPC of in-order processors be increased through front-end optimization? The experimental results demonstrate that it is possible to improve the IPC of in-order processors exclusively through front-end optimization. Across all evaluated workloads and target ISAs (RISC-V and ARM AArch64), the IFA consistently improved IPC while reducing CPI and pipeline stalls. The observed IPC gains ranged from 4.88% in Blowfish (ARM AArch64) to 28.8% in MatMult (RISC-V), with CPI reductions reaching up to 22.3%. These results confirm that front-end stalls represent a significant and addressable bottleneck in in-order architectures.

RQ2 — Which additional performance parameters can be improved by the proposed solution? Beyond IPC improvements, the IFA positively affected several additional microarchitectural metrics. Pipeline stall counts were substantially reduced across all evaluated workloads, reaching reductions of up to 99.34% in CRC32 (RISC-V) and 61.05% in WikiSort (RISC-V). These results demonstrate the effectiveness of the DAU in bypassing stalled instructions while maintaining continuous pipeline utilization.

Instruction-cache miss rates remained stable or exhibited only minor variations, while branch prediction accuracy was preserved in nearly all evaluated scenarios. Importantly, no significant negative performance regressions were observed in IPC or CPI metrics, although branch prediction accuracy showed minor variation in the BasicMath workload, likely attributable to pipeline timing changes introduced by the IFA buffer.

RQ3 — Does the proposed solution incur lower energy overhead compared to strategies? Using the Race-to-Halt energy model [16] and assuming a conservative 3.5% active power overhead associated with the FEB, DAU, and DU structures, the IFA still achieved an estimated net energy reduction of approximately 3.5%.

These results indicate that the execution-time reduction provided by the proposed architecture compensates for the additional hardware cost, validating the IFA as an energy-

efficient enhancement suitable for embedded and low-power processors.

The current evaluation presents two primary limitations: i) the proposed architecture was validated exclusively through cycle-accurate simulation using Gem5. No RTL synthesis or physical implementation was performed, leaving area utilization, timing closure, and post-synthesis power measurements as open research questions; ii) although the selected benchmark suites are representative of embedded workloads, they do not cover real-time operating-system interference, multicore interactions, or multithreaded execution scenarios.

Future work will focus on three main directions: i) the first involves RTL implementation in SystemVerilog and FPGA-based validation using open-source RISC-V cores such as CVA6 or PicoRV32, enabling precise area, timing, and power measurements; ii) the second involves technology-scaling analysis using open-source process design kits to evaluate the overhead of the IFA in advanced semiconductor nodes below 14 nm; iii) future studies will investigate the behavior of the proposed architecture under embedded TinyML workloads, where instruction-level parallelism patterns may further amplify the performance gains observed in compute-intensive benchmarks such as MatMult.

References

[1] C. Kaynak, B. Grot, and B. Falsafi, "Confluence: Unified instruction supply for scale-out servers," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*, 2015.

[2] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*, 2008.

[3] M. B. Breughe, S. Eyerman, and L. Eeckhout, "Mechanistic analytical modeling of superscalar in-order processor performance," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, pp. 50:1–50:26, 2014.

[4] C. Arul Rathi, G. Rajakumar, T. Ananth Kumar, and T. S. Arun Samuel, "Design and development of an efficient branch predictor for an in-order risc-v processor," *Journal of Nano- and Electronic Physics*, vol. 12, no. 5, pp. 05 021–1–05 021–4, 2020.

[5] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA)*, ser. ISCA '99. IEEE Computer Society, 1999, pp. 16–27.

[6] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[7] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, 1995.

[8] G. A. Chacon, "Processor memory system design for performance and security," Dissertation, Texas A&M University, College Station, TX, 2023.

[9] O. J. Santana, A. Falcón, A. Ramirez, and M. Valero, "Dia: A complexity-effective decoding architecture," *IEEE Transactions on Computers*, vol. 58, no. 4, pp. 449–462, 2009.

[10] G. Reinman, B. Calder, and T. Austin, "Optimizations enabled by a decoupled front-end architecture," *IEEE Transactions on Computers*, vol. 50, no. 4, pp. 338–355, 2001.

[11] P. S. Oberoi, "Out-of-order front-ends," Preliminary Proposal, University of Wisconsin-Madison, Madison, WI, 2001.

[12] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An effective alternative to large instruction windows," *IEEE Micro*, vol. 23, no. 6, pp. 20–25, 2003.

[13] R. Shioya, M. Goshima, and H. Ando, "A front-end execution architecture for high energy efficiency," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 419–431.

[14] N. Binkert *et al.*, "The gem5 simulator," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011, pp. 1–7.

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization (WWC-4)*. IEEE, 2001, pp. 3–14.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012.

[17] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[18] S. Mashimo, A. Fujita, R. Matsuo, S. Akaki, A. Fukuda, T. Koizumi, J. Kadomoto, H. Irie, M. Goshima, K. Inoue, and R. Shioya, "An open source fpga-optimized out-of-order risc-v soft processor," in *International Conference on Field-Programmable Technology (ICFPT)*, 2019.

[19] M. McKeown, J. Balkind, and D. Wentzlaff, "Execution drafting: Energy efficiency through computation deduplication," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 432–444.

[20] R. Huerta *et al.*, "Socgpu: Simple out-of-order core for gpgpus," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[21] A. Miyoshi, C. Lefurgy, E. Hensbergen, R. Rajamony, and R. Rajwar, "Critical power slope: understanding the runtime-leakage tradeoff in deep submicron technologies," in *Proceedings of the 16th international conference on Supercomputing*, 2002, pp. 35–44.

[22] A. Bardizbanyan, M. Själander, and P. Larsson-Edefors, "Reconfigurable instruction decoding for a wide-control-word processor," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, 2011.

[23] A. Djupdal, M. Själander, M. Jahre, S. Aunet, and T. Ytterdal, "Optimizing energy efficiency in subthreshold risc-v cores," in *ResearchGate Preprint*, 2020.