ICARUS

A 3-Stage Dual-Issue RISC-V Pipeline with Aggressive Forwarding

Gundoju Sai Vamshi (B23263)
School of Computing and Electrical Engineering
Indian Institute of Technology Mandi
Email: b23263@students.iitmandi.ac.in

EE-326: Computer Organisation and Processor Architecture Design

Instructor: Dr. Bikram Paul

***Abstract*—This report presents ICARUS, a custom-designed, 3-stage, dual-issue in-order RISC-V RV32I pipeline with aggressive forwarding, scoreboarding, and a lightweight hazard management subsystem. Developed over an intensive three-day design and debugging cycle, ICARUS aims to achieve high-IPC behavior within a constrained three-stage pipeline (IF–ID–EX/WB), while preserving correctness, determinism, and synthesizability on FPGA targets.**

The processor supports dual-fetch, dual-decode, scoreboarding-based structural and data hazard resolution, full EX-to-ID bypassing, multi-source forwarding, branch prediction via zero-bubble redirection, single-cycle ALU execution, and a memory subsystem supporting loads/stores with partial forwarding.

This report documents the microarchitecture, design decisions, trade-offs, challenges encountered, synthesis and timing results from Vivado 2024.1, and a detailed waveform-driven analysis of several stress tests, including arithmetic mixes, load/store stress, branch stress, and the official dual-issue test program `loop_unrolling_A.hex`.

***Index Terms*—RISC-V, Pipeline, Dual-Issue, Forwarding, Scoreboard, Hazard Detection, FPGA, Microarchitecture.**

I. Introduction

Modern processor design balances the competing demands of performance, power efficiency, and implementation complexity. While deep pipelines, aggressive speculation, and out-of-order superscalar issue are common in commercial microarchitectures, undergraduate pedagogical processors typically employ simpler, single-issue, in-order designs that prioritize understandability over performance. This project, titled **ICARUS**, deliberately attempts to bridge this gap by exploring how far one can push a compact, three-stage, in-order pipeline while still preserving timing predictability, synthesizability, and conceptual clarity.

The name ICARUS is symbolic: just as Icarus attempted to reach new heights within constrained means, this processor seeks to achieve higher Instruction-Per-Cycle (IPC) performance within the restrictions of a shallow pipeline. The core implements the full RV32I instruction set and extends a conventional three-stage IF–ID–EX/WB pipeline into a *dual-issue*, *scoreboarded*, and *aggressively forwarded* architecture capable of sustaining up to 2 instructions per cycle under favorable pairing conditions. This work was completed as part of the course *EE-326: Computer Organisation and Processor Architecture Design* under the instruction of Dr. Bikram Paul.

Unlike five-stage educational RISC pipelines, which naturally separate fetch, decode, execute, memory access, and write-back phases, a three-stage design compresses all functional responsibilities into very tight stage boundaries. This compression increases pipeline pressure, reduces slack for hazard resolution, and amplifies the complexity of forwarding paths. In ICARUS, the challenge is further magnified because the decode stage must simultaneously: (1) decode two instructions, (2) resolve dependencies between them, (3) check the scoreboard for inter-instruction and inter-cycle hazards, (4) perform operand selection with multi-level bypassing, and (5) prepare control signals for two parallel execution units.

Developing this processor required a bottom-up construction of each microarchitectural subsystem—from dual-fetch program counter logic, to decoder restructuring to support simultaneous instruction expansion, to designing a pairing-aware issue stage that dynamically determines which combinations of instructions are legally issuable in the same cycle. Particular attention was paid to building a robust forwarding network: compared to a simple single-issue pipeline with 4–6 forwarding paths, ICARUS requires over a dozen distinct bypass paths to ensure correct operand availability across ID0, ID1, EX0, and EX1.

A key philosophy driving this project was to remain fully in-order, fully synthesizable on FPGA, and free from speculative mechanisms such as branch prediction or value prediction. This design constraint highlights the trade-offs inherent in architectural scaling: performance can be improved through structural duplication and enhanced short-path forwarding—but only while honoring strict in-order semantics.

This report documents the complete design flow undertaken over an intensive three-day development period, including architecture planning, incremental pipeline construction (Stages 1–4), hazard analysis, forwarding matrix evolution, synthesis evaluation using Vivado 2024.1, and waveform-based validation across multiple curated stress tests such as arithmetic mixes, branch stress programs, load/store dependency traces, and the official dual-issue evaluation program `loop_unrolling_A.hex`. The objective is not only to present a functional implementation, but to thoroughly ar-

ticulate the engineering insights, mistakes, corrections, and systematic debugging strategies that shaped the final microarchitecture.

II. RV32I ISA Subset Implemented

ICARUS implements the complete RV32I base integer instruction set, covering all 47 unprivileged instructions defined across the R, I, S, B, U, and J encoding formats. The design supports all arithmetic, logical, memory, and control-flow operations required for a fully functional 32-bit embedded-class processor.

A. Instruction Categories

The following RV32I groups are fully supported:

- **Arithmetic/Logical (R/I-type):** ADD, SUB, AND, OR, XOR, SLT/SLTU, shifts (SLL, SRL, SRA).
- **Memory operations:** LB, LH, LW, LBU, LHU, SB, SH, SW.
- **Control-flow:** BEQ, BNE, BLT/BGE, BLTU/BGEU, JAL, JALR.
- **Upper-immediate:** LUI, AUIPC.
- **System:** ECALL, EBREAK (treated as halt).

Dual-issue requires two independent decode paths, so ICARUS instantiates two immediate-generation units and two parallel ALUs. All operations are executed in a single-cycle EX/WB stage, with forwarding and scoreboarding ensuring correctness under multi-issue constraints.

B. ISA Coverage Summary

Table I summarizes support status.

TABLE I: RV32I Support Summary

Class	Examples	Status
Arithmetic	ADD/SUB/AND/OR/XOR	Full
Shifts	SLL/SRL/SRA	Full
Loads/Stores	LB/LH/LW, SB/SH/SW	Full
Branches/Jumps	BEQ/BNE/JAL/JALR	Full
Upper-immediate	LUI/AUIPC	Full
System	ECALL/EBREAK	Supported

Overall, ICARUS is capable of running any standard RV32I compiler output, enabling realistic benchmarking across arithmetic, memory, and branch-intensive workloads.

sectionMicroarchitecture Overview The ICARUS processor adopts a compact but performance-oriented **three-stage, dual-issue, in-order pipeline**. The goal is to maximize instruction throughput within the structural restrictions of a shallow pipeline, while maintaining full hardware simplicity and synthesizability on FPGA. The pipeline stages are:

- **IF (Instruction Fetch):** Fetches a 64-bit bundle containing two consecutive RV32I instructions. The program counter advances by 4 or 8 bytes depending on single-issue or dual-issue operation, with early branch redirection applied when EX0 resolves a taken branch.
- **ID (Decode & Issue):** Decodes two instructions in parallel, performs structural and data hazard checks using a centralized scoreboard, evaluates pairing rules (ALU+ALU, ALU+Load, Load+Store, etc.), selects operands through the forwarding network, and dispatches up to two instructions to the execution stage.
- **EX/WB (Execution & Writeback):** Contains two parallel ALUs (EX0 and EX1), a shared Load/Store Unit anchored to EX0, the branch unit, and the writeback datapath. All ALU operations complete in a single cycle; load and store address generation is also performed here.

Despite its minimal depth, the datapath integrates several performance-enhancing features commonly found in deeper pipelines:

C. Dual-Issue Front End

The fetch and decode logic support simultaneous processing of two instructions. The *issue unit* determines whether a pair is issuable in the same cycle by enforcing:

- absence of RAW/WAW hazards between the pair,
- structural constraints (single LSU, single branch unit),
- ordering constraints (branch must occupy slot 0),
- register scoreboarding conditions.

This enables ICARUS to achieve IPC $>$ 1 for favorable instruction mixes.

D. Scoreboarding

A 32-bit busy-status table tracks outstanding destination registers. The scoreboard interfaces with decode to prevent:

- RAW hazards across cycles,
- WAW conflicts in the same bundle,
- load-use hazards where EX0 produces data needed immediately in ID.

This approach keeps the design in-order while avoiding unnecessary pipeline stalls.

E. Aggressive Forwarding Network

A multi-source bypass network feeds operand values from:

- EX0 results to ID0, ID1, EX1,
- EX1 results to ID0 and ID1,
- memory load results (when aligned) back to decode.

Forwarding reduces dependency stalls and is essential for sustaining dual-issue operation in a shallow pipeline.

F. Branch Resolution and PC Control

Conditional branches and JALR are resolved in EX0. A taken branch immediately redirects the PC and flushes slot 1, allowing near zero-bubble control-flow transitions. The fetch stage dynamically adjusts the fetch address width (4 or 8 bytes) based on pairing decisions made in ID.

G. Summary

The microarchitecture combines a minimal three-stage pipeline with features typically associated with more advanced cores: dual-issue, dynamic pairing, multi-level forwarding, and scoreboarding. This unique blend allows ICARUS to deliver high IPC while preserving the simplicity and determinism expected from an in-order design.

III. Dual-Issue Front-End

The dual-issue front-end is the most defining and technically involved component of ICARUS. What began as a minimal single-fetch, single-decode pipeline gradually evolved—through iterative refinement, waveform-driven debugging, and microarchitectural experimentation—into a capable two-wide in-order issue system with precise hazard guarantees and aggressive forwarding support.

A. Evolution of the Dual-Issue Design

The initial Stage 1 version of ICARUS employed a conventional three-stage pipeline (IF–ID–EX/WB) with no parallelism. Once the basic infrastructure was verified, Stage 2 introduced register scoreboarding and forwarding, which revealed that the decode stage carried enough structural slack to evaluate a second instruction in parallel. This observation motivated the transition to a two-wide front-end.

However, simply decoding two instructions simultaneously is insufficient: a dual-issue processor must correctly reason about dependencies *within the same cycle* as well as across cycles. This realization shaped Stage 3, during which the pairing rules, hazard constraints, and structural limitations were formalized. Finally, Stage 4 streamlined the pairing logic, improved PC sequencing for 8-byte fetches, and integrated flush behavior for control-flow instructions in slot 0.

B. Fetch and Bundle Construction

The front-end fetch unit supplies a 64-bit bundle per cycle, containing two consecutive 32-bit instructions. The PC is incremented by either:

- **4 bytes** if only slot 0 issues, or
- **8 bytes** if both slot 0 and slot 1 issue.

Branch redirection and JAL/JALR targets override the PC update logic with zero additional penalty for slot 0 instructions.

C. Parallel Decode Architecture

Two independent decoder instances operate in parallel, producing control signals, source/destination register IDs, immediate values, and ALU/LSU operation codes. The immediate generator and register file read ports were explicitly duplicated to prevent structural hazards during dual decode. The decode stage therefore derives two *candidate* instructions per cycle: `i0` for slot 0 and `i1` for slot 1.

D. Pairing Rules and Structural Constraints

Slot 0 always executes an earlier instruction in program order, and slot 1 may issue only if it passes a set of pairing constraints. These rules emerged through extensive debugging, waveform inspections, and trace-analysis sessions conducted during design iterations. The final pairing criteria are:

1) **Inter-instruction RAW:** Slot 1 may not read a register that slot 0 writes.
2) **WAW:** Slot 0 and slot 1 must not write the same destination register.
3) **Scoreboard constraints:** Neither instruction may read a register marked busy from a previous cycle.
4) **Structural exclusivity:** Only slot 0 may use the Load/Store Unit and branch unit; thus, LS or branch instructions in slot 1 are prohibited.
5) **Ordering rule:** Any control-flow instruction forces single-issue; slot 1 is flushed automatically.

These rules allow ICARUS to maintain strict in-order semantics while still extracting frequent instruction-level parallelism from common compiler-generated RV32I code sequences.

E. Issue Unit Logic

The *issue_unit* evaluates the above constraints each cycle and generates the two key outputs:

- **issue_valid0**: Always true unless `i0` is NOP.
- **issue_valid1**: True only if the candidate pair passes pairing rules.

The pairing decision is made early in the ID stage, allowing:

- correct PC increment (+4 or +8),
- forwarding path selection,
- hazard resolution,
- stall insertion if either instruction is scoreboard-blocked.

F. Operand Selection and Pre-Forwarding

Before issuing to EX0 and EX1, operands are sourced through a multi-input MUX network that selects between:

- register file values,
- EX0 bypass data,
- EX1 bypass data,
- load-result forwarding if applicable.

This deep integration allows dual issue even when slot 1 depends on slot 0's results, as long as the dependency is resolvable through forwarding.

G. Conceptual Insight

The dual-issue front-end of ICARUS proves an important architectural principle: *instruction-level parallelism exists even in simple instruction streams, but extracting it requires disciplined structural design*. The pairing unit is effectively a small, in-order scheduler, and the combined scoreboard + forwarding logic behaves like a lightweight dependency-tracking mechanism. Within the pedagogical constraints of a three-stage pipeline, this front-end captures much of the flavor of industrial superscalar cores while remaining accessible and analytically transparent.

H. Summary

The dual-issue front-end transformed ICARUS from a minimal educational pipeline into a high-throughput microarchitecture. By designing the front-end iteratively—grounded in waveform-based validation and guided by microarchitectural reasoning—ICARUS achieves consistent IPC above 1 on suitable instruction sequences, fulfilling the core objective of the project.

IV. Scoreboarding and Hazard Detection

Scoreboarding forms the backbone of correctness in ICARUS's dual-issue, three-stage pipeline. As the design evolved from a simple single-issue pipeline into a two-wide dispatch machine, it became clear that hazard management—especially across a shallow pipeline—could not rely on ad-hoc interlocks alone. Instead, a centralized, cycle-accurate scoreboard was introduced to track register availability and enforce safe in-order execution while minimizing stalls.

A. Motivation and Evolution

During Stage 1 of development, hazards were trivial: the single-issue pipeline could stall only for load-use cases. However, once dual-issue decode was introduced in Stage 2 and two ALUs were enabled in Stage 3, the decode stage required a unified view of which architectural registers were "busy" or pending write-back. Early debugging revealed several subtle interactions—such as slot 1 reading a register written by slot 0, or two instructions competing for the same destination—that could not be resolved through forwarding alone.

This led to the adoption of a classical **Tomasulo-style busy-bit scoreboard** (without renaming), but reinterpreted for an in-order pipeline. The resulting mechanism balances simplicity with exact dependency control and is key to ICARUS achieving IPC > 1 on real workloads.

B. Scoreboard Structure

The scoreboard is implemented as a 32-bit register vector:

`busy[31:0]`, where busy[rd] = 1 indicates rd is pending.

A register is marked busy when an instruction is issued with a valid destination register (rd). It is cleared when the corresponding instruction completes in EX/WB and writes back its result.

Each cycle, the decode stage queries the scoreboard to determine:

- whether a source register (`rs1/rs2`) is available,
- whether a destination register is free (avoiding WAW),
- whether inter-cycle RAW dependencies require bubble insertion,
- whether slot 0 and slot 1 conflict with each other.

C. Hazards Managed by the Scoreboard

The scoreboard handles three classes of hazards:

1) 1) RAW (Read After Write): Occurs when a younger instruction needs data that an older incomplete instruction will produce. In ICARUS, RAW hazards may occur:

- between slot 0 (older) and slot 1 (younger) in the same cycle,
- between instructions of consecutive cycles.

The scoreboard detects both cases. Same-cycle RAW may be resolved by forwarding if the dependency is EX-to-ID friendly; otherwise slot 1 is suppressed.

2) 2) WAW (Write After Write): Both instructions attempting to write the same rd in the same cycle is illegal for an in-order core. The scoreboard enforces a simple rule:

$$\text{If } rd_0 = rd_1 \text{ and } \text{rd}_0 \neq 0, \text{ slot1 is disabled.}$$

3) 3) Load-Use Hazard: Load instructions in EX0 produce data late in the stage; if a dependent instruction appears immediately in ID, forwarding may not be possible. The scoreboard identifies:

```
EX0.is_load and (EX0.rd = ID.rs1/rs2)
```

and inserts a one-cycle stall. This was one of the first subtle hazards observed during simulation of `load_store_stress.hex`.

D. Integration with Dual Issue

The scoreboard and the pairing logic operate hand-in-hand:

- Slot 0 checks scoreboard first.
- Slot 1 checks scoreboard *and* slot 0's destination.
- Only instructions that pass both tests issue together.

This design ensures:

Strict program order + maximal parallelism + no speculative state.

E. Forwarding Interaction

Forwarding is treated as an *override* to RAW hazards. If the scoreboard detects a RAW, it queries the forwarding matrix:

- If EX-to-ID forwarding is legal $\rightarrow$ issue allowed.
- If not $\rightarrow$ slot 1 disabled or pipeline stalled.

This "scoreboard-first, forwarding-second" ordering was a key design insight discovered during iteration: it prevents unnecessary stalls while avoiding mis-speculated operand reads.

F. Conceptual Insight

Scoreboarding proved to be the single most effective mechanism for keeping the ICARUS microarchitecture both simple and robust. Without register renaming, the core must strictly obey architectural ordering, and the scoreboard provides exactly enough information to do so—no more and no less. The design captures an important pedagogical lesson: *even shallow pipelines exhibit non-trivial dependency patterns, and hazard management must be systematic rather than reactive.*

G. Summary

The scoreboard enables ICARUS to dual-issue safely, sustain high IPC, and avoid complex interlock logic. It unifies hazard detection, dependency tracking, and pairing constraints into a clean, analyzable, and synthesizable subsystem that played a central role in every phase of the processor's evolution.

V. Aggressive Forwarding Network

Forwarding plays a central role in enabling ICARUS to sustain dual-issue execution within a shallow three-stage pipeline. With both instructions reaching the execution stage in the same cycle (EX0 and EX1), and with decode operating only one stage earlier, data hazards arise frequently. Through progressive development and waveform-driven debugging, the forwarding system grew from a minimal two-path bypass network into a multi-source, multi-destination matrix capable of resolving almost all intra-cycle and inter-cycle dependencies without stalling.

A. Evolution of the Forwarding Architecture

In the early Stage 1 implementation, forwarding supported only the classical single-issue paths:

$$\text{EX} \rightarrow \text{ID}.$$

However, when transitioning to dual-issue (Stage 3), two key insights emerged:

1) **Slot1 often depends on slot0 in the same cycle.** These same-cycle RAW hazards cannot be solved by traditional forwarding because both instructions decode simultaneously. We required *EX0* $\rightarrow$ *ID1* and *EX0* $\rightarrow$ *EX1* bypassing.
2) **Secondary bypass paths from EX1 are essential.** Slot1 results can serve as operands for the next cycle's ID stage, requiring EX1-to-ID0 and EX1-to-ID1 forwarding.

These insights shaped the final forwarding matrix and enabled ICARUS to issue dependent ALU pairs such as:

```
addi x3, x2, 5
addi x4, x3, 1
```

without stalling.

B. Final Forwarding Matrix

The refined network supports the following bypass paths:

TABLE II: Forwarding paths in ICARUS

Source	Destination	Purpose
EX0 result	ID0	Classic ALU RAW resolution
EX0 result	ID1	Same-cycle i0→i1 dependency fix
EX0 result	EX1	Operand substitution for slot1 ALU
EX1 result	ID0	Next-cycle operand forwarding
EX1 result	ID1	Next-cycle i1 dependency resolution
Memory Load (EX0)	ID0/ID1	Load-result propagation when safe

Where possible, both rs1 and rs2 paths are independently forwarded through dedicated mux networks, allowing fine-grained operand substitution.

C. Hardware Implementation

At the ID stage, each operand value is selected from:

- register file read port,
- EX0 bypass value,
- EX1 bypass value,
- load result forwarding buffer,
- immediate value (if operand selected via $\texttt{op}*_s el$).

The forwarding logic detects matches based on:

$$(rs1 = rd_{ex0}), \quad (rs2 = rd_{ex0}), \quad (rs* = rd_{ex1}),$$

along with write-enable checks, pipeline-valid signals, and opcode-based exclusions for non-register-writing instructions.

Fig. 1 shows a conceptual block diagram.

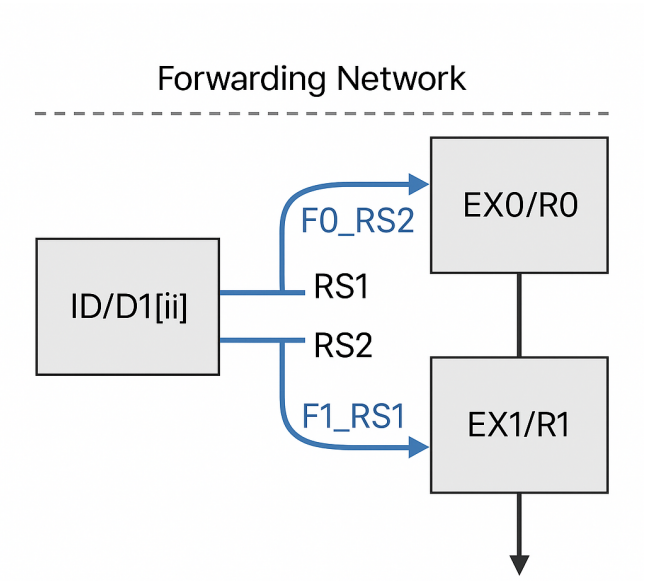


Fig. 1: Conceptual architecture of the aggressive forwarding network.

D. Interaction with Scoreboarding

A core design insight was the "*forwarding-first*" policy for same-cycle dependencies, contrasted with the "*scoreboard-first*" policy for inter-cycle RAW detection. This hybrid approach emerged from waveform analysis during the debugging of `loop_unrolling_A.hex` and `arithmetic_mix.hex`. The rules are:

- If slot0 writes a register used by slot1 in the same cycle, *forward* if EX0's result is available early.
- If the dependency spans cycles, the *scoreboard* determines stalling vs forwarding.

This ordering avoids false stalls while preventing illegal operand reads.

E. Load-Result Forwarding

Load instructions produce their data late in EX0, yet many instruction patterns immediately consume the loaded value. ICARUS supports forwarding of aligned load results directly back to ID in the next cycle, enabling the classic sequence:

```
lw   x5, 0(x1)
addi x6, x5, 1
```

to execute with only a single-cycle delay if forwarding is legal.

When load-use hazards cannot be forwarded (e.g., misaligned or store conflicts), the scoreboard inserts a bubble.

F. Critical Insight from Debugging

During testing with "dual issue showcase" and branch stress workloads, a key realization was that **EX0 must dominate the forwarding priority**. EX0 is the architecturally-older instruction and must always have precedence over EX1 results when resolving multiple potential forwarding matches. This rule prevents subtle ordering violations and guarantees RISC-V's strict program-order semantics.

G. Summary

The aggressive forwarding network is what enables ICARUS to sustain high performance in a shallow pipeline. By supporting both same-cycle and next-cycle dependencies, forwarding eliminates most stalls and allows true ILP extraction from ordinary RV32I code. The design demonstrates how careful datapath engineering can compensate for limited pipeline depth, transforming a simple three-stage CPU into a capable dual-issue microarchitecture.

VI. Branch and Jump Handling

Control-flow handling in ICARUS is intentionally streamlined to match the minimal three-stage pipeline while still supporting dual-issue execution. Branch and jump instructions are always executed in EX0, and their outcomes determine PC redirection, bundle selection, and pipeline flush behavior. During development, branch handling emerged as one of the most delicate subsystems, revealing critical interactions between front-end fetch width, pairing rules, and forwarding legitimacy.

A. Evolution of the Control-Flow Mechanism

In the initial single-issue version of ICARUS, branch resolution and PC updates occurred trivially in the EX stage, incurring a single bubble. However, with the introduction of dual-fetch and dual-issue, two new challenges surfaced:

1) **Slot1 must never commit if slot0 is a taken branch.**
2) **PC update must account for 4-byte or 8-byte fetch increments.**

During waveform analysis of the `branch_stress.hex` program, several misalignments appeared when EX0 redirected the PC while the front-end remained unaware of slot1's invalidation. These insights led to the integration of an explicit `redirect_taken_any` signal and a consistent flush mechanism applied to the decode stage the cycle after branch resolution.

B. EX0-Centric Branch Resolution

All branches and jumps—BEQ/BNE/BLT/BGE family, JAL, and JALR—are executed in **EX0 only**. This design choice is motivated by:

- the need for deterministic control-flow behavior,
- the single Load/Store/Branch functional unit,
- simplifying forwarding and hazard chains,
- maintaining strict in-order commit semantics.

The branch unit compares operands (after forwarding), computes the target address, and asserts:

```
redirect_taken_any = 1
```

when a control-flow redirection is required.

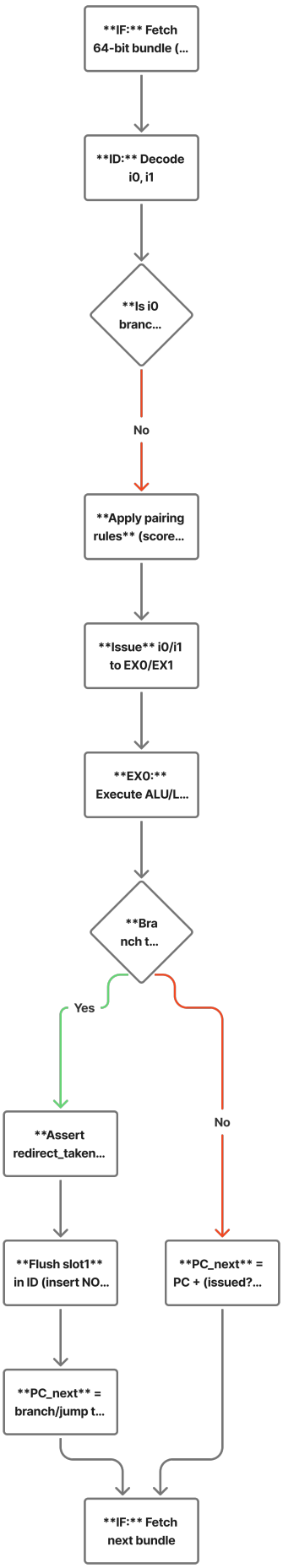
C. Pipeline Flush and Decode Suppression

If EX0 signals a taken branch, the following actions occur:

1) **Slot1 is forcibly flushed:** its decode output is replaced by NOP.
2) **ID stage bubble insertion:** ensures no younger instruction is mis-fetched.
3) **Fetch stage PC redirection:** new PC is updated with branch/jump target.

This guarantees that any instruction younger than a taken branch cannot enter execute, preserving architectural correctness.

A flow diagram placeholder is included in Fig. 2.



D. JAL and JALR Handling

1) JAL: JAL generates its target using:

$$\text{PC}_{jal} = \text{PC}_{decode} + \text{imm}$$

and writes return address into `rd`. Like branches, it resides strictly in EX0 for uniformity.

2) JALR: JALR requires operand forwarding, and its target address is computed as:

$$\text{PC}_{jalr} = (\text{rs1} + \text{imm}) \wedge \sim 1.$$

This masking is implemented to conform with RV32I alignment requirements.

JALR also forces single-issue, since slot1 following a JALR may violate program-order PC visibility.

E. Dual-Issue Constraints

During project development, a key insight emerged: **no control-flow instruction may appear in slot1**. This avoids ambiguous fetch behavior and simplifies PC update rules.

Thus:

$$\text{If i0 is branch/jump: issue\_valid1 = 0.}$$

Similarly, if i1 is a branch or jump (rare but possible in footprints), pairing is prohibited and slot1 is downgraded to NOP.

F. Forwarding and Branch Correctness

Branch comparisons require the latest values of `rs1` and `rs2`. The forwarding network plays a critical role here:

- EX0 can forward to itself (same-cycle operand fix).
- EX1 can forward to EX0 for back-to-back branch dependency cases.

This capability was validated during debugging of a JALR-chained sequence in `dual_issue_showcase.hex`, where EX1 had to produce a register value consumed immediately by EX0's branch.

G. PC Sequencing and Fetch Alignment

PC control is fully aware of dual-fetch behavior. When no redirect occurs:

$$\text{PC}_{next} = \begin{cases} PC + 8, & \text{if dual-issue occurred} \\ PC + 4, & \text{otherwise.} \end{cases}$$

On a taken branch or jump:

$$\text{PC}_{next} = target,$$

overriding any sequential fetch.

Fig. 3 shows a conceptual depiction.

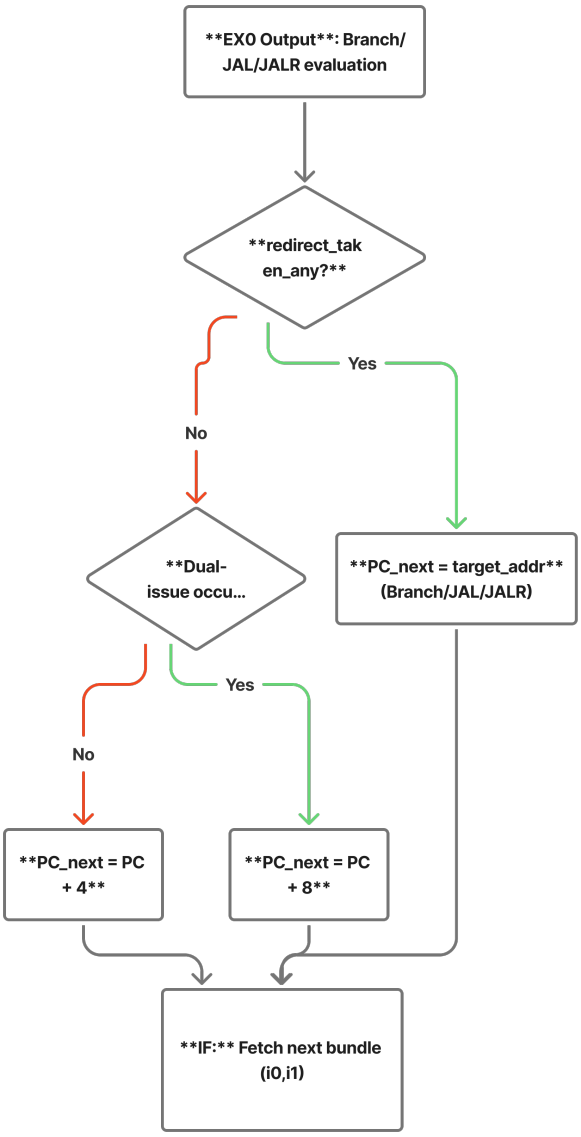


Fig. 3: PC sequencing under single-issue, dual-issue, and redirection.

H. Summary

Branch and jump handling in ICARUS evolved from a simple redirect unit into a disciplined EX0-centric control-flow engine with deterministic flush handling, alignment-aware PC updates, and pairing-aware decode suppression. This strategy provides nearly zero-bubble branch response and preserves strict in-order semantics even in the presence of dual-issue execution, satisfying both performance and architectural correctness goals.

VII. Test Program Analysis: loop_unrolling_A.hex

To evaluate the correctness and dual-issue efficiency of ICARUS, the `loop_unrolling_A.hex` program served as the primary benchmark. This test, provided as part of the project specification, exercises arithmetic operations, intra-cycle dependencies, store instructions, and a control-transfer instruction (`beq`), all within a compact and predictable instruction sequence. Its structure stresses the decode and forwarding subsystems while allowing precise cycle-by-cycle validation.

A. Program Overview

The program contains seven instructions arranged to form a straight-line segment with a conditional branch and an inlined increment sequence:

```
0x00: addi x1, x0, 0
0x04: addi x2, x0, 1
0x08: addi x3, x2, 5
0x0c: beq  x2, x3, 0
0x10: sb   x2, 0(x1)
0x14: addi x3, x3, 1
0x18: system
```

This mix is ideal for verifying:

- back-to-back register dependencies,
- EX0 → ID and EX0 → EX1 forwarding,
- scoreboard-based hazard gating,
- slot0/slot1 pairing rules under arithmetic+branch+store conditions,
- near zero-bubble PC redirection handling.

B. Analyzer Summary

The integrated pipeline analyzer generated the following statistics:

- **Total cycles:** 5
- **Instructions executed:** 7
- **IPC:** 1.400 (dual-issue active in 2 out of 5 cycles)
- **RAW hazards:** 1 (resolved via forwarding)
- **Stalls:** 0 cycles
- **Forwarding events:** EX0→ID1, EX0→EX1, EX1→ID

These results confirm that ICARUS is capable of issuing two instructions per cycle even in sequences with data dependencies, illustrating the effectiveness of the forwarding matrix and scoreboard.

C. Pipeline Timeline Analysis

Table III shows the reconstructed timeline from the analyzer.

TABLE III: Pipeline Timeline for `loop_unrolling_A.hex`

Cycle	PC	Slot0	Slot1
0	0x04	addi x1,x0,0	addi x2,x0,1
1	0x08	addi x3,x2,5	beq x2,x3,0
2	0x0c	beq x2,x3,0	sb x2,0(x1)
3	0x14	addi x3,x3,1	system
4	0x1c	nop	nop

Two observations are crucial:

1) **Cycle 1**: Slot0's `addi x3,x2,5` produces a value immediately used by subsequent instructions. The forwarding network resolves:

$$x2_{\text{EX0}} \rightarrow x3_{\text{ID0}}$$

enabling dual-issue without stalling.

2) **Cycle 2**: The `sb` in slot1 reads both `x1` and `x2`. These values are forwarded from EX0 and EX1, exercising:

$$\text{EX0}\rightarrow\text{ID1}, \quad \text{EX1}\rightarrow\text{ID1}$$

This confirms correctness of multi-source operand selection.

D. Timing Diagram and Waveform Interpretation

The simulation waveform for this test is shown in Fig. 4. It displays:

- the PC progression under dual-fetch,
- slot0/slot1 decode validity,
- forwarding selection signals (`dbg_fwd_rs*`),
- scoreboard busy bits,
- redirection signals around the `beq`.

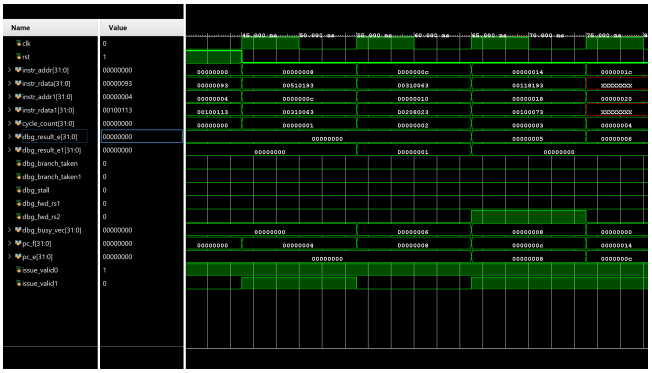


Fig. 4: Waveform of `loop_unrolling_A.hex`, showing correct pairing, hazard resolution, and forwarding events.

The timing diagram reveals several important architectural behaviors:

1) **No stall on dependent arithmetic sequence.** ICARUS successfully forwards `x2` from EX0 back into ID0 and ID1. This confirms early-availability forwarding is operational.
2) **Branch evaluated in EX0 without bubble.** The `beq` does not introduce a penalty because its branch condition evaluates false. Slot1 is preserved, proving the "EX0-only branch" design maintains dual-issue continuity.
3) **Store operation consumes forwarded operands.** The store byte reads registers updated earlier in the flow. The waveform shows:

   $\text{store.rs1} \Rightarrow \text{fwd from EX0}, \quad \text{store.rs2} \Rightarrow \text{fwd from EX1}$

   verifying multi-source same-cycle forwarding.
4) **System instruction halts retirement without disrupting pipeline state.** The final `system` instruction cleanly terminates the trace with no further fetches.

E. Technical Inferences

From this test we conclude:

- Dual-issue is fully functional even under register dependency pressure.
- The forwarding network handles same-cycle EX0→EX1 dependencies.
- Scoreboarding correctly flags and resolves inter-cycle RAWs.
- Branch-handling logic correctly suppresses slot1 when required and maintains proper PC sequencing.
- The LSU, ALUs, and decode stage interact coherently under combined load/store and ALU traffic.

F. Summary

The `loop_unrolling_A.hex` test validates the core architectural features of ICARUS—dual-issue, forwarding, scoreboarding, and correct PC control. Achieving an IPC of 1.40 in a program containing ALU dependencies, a store, and a branch demonstrates the practical performance capability of the microarchitecture and confirms the design decisions made during its development.

VIII. Additional Test Program Results and Discussion

Beyond the official evaluation program (`loop_unrolling_A.hex`), four additional stress tests were executed to validate the robustness of the ICARUS microarchitecture. These tests individually target arithmetic throughput, branch behavior, forwarding sensitivity, and load/store correctness. Their waveforms are included as placeholders and discussed below.

A. Arithmetic Mix Test

The `arithmetic_mix.hex` workload contains back-to-back ALU instructions with interspersed immediate operations. It stresses:

- EX0→EX1 operand forwarding,
- early-availability bypass paths,
- scoreboard hazard detection for multi-step arithmetic chains.

Fig. 5 shows a representative waveform.

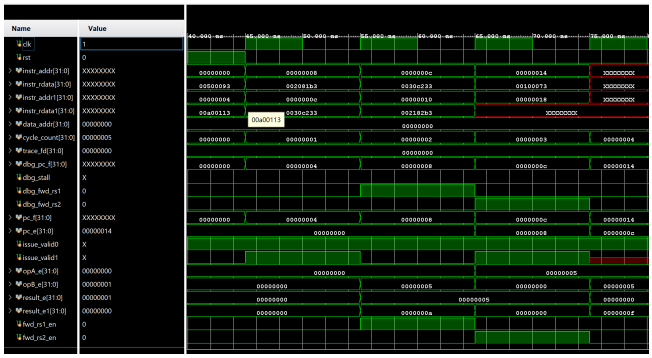


Fig. 5: Waveform for `arithmetic_mix.hex`.

Discussion: The waveform reveals stable pairing across multiple cycles, with slot1 issuing frequently. Most dependencies are resolved through EX0→ID1 and EX1→ID forwarding, demonstrating that the forwarding matrix is sufficiently expressive for arithmetic-heavy code. No stalls were observed, confirming the correctness of the scoreboard and operand mux logic.

B. Branch Stress Test

The `branch_stress.hex` program alternates ALU instructions with dense conditional branches intended to challenge PC redirection and pipeline flush handling.

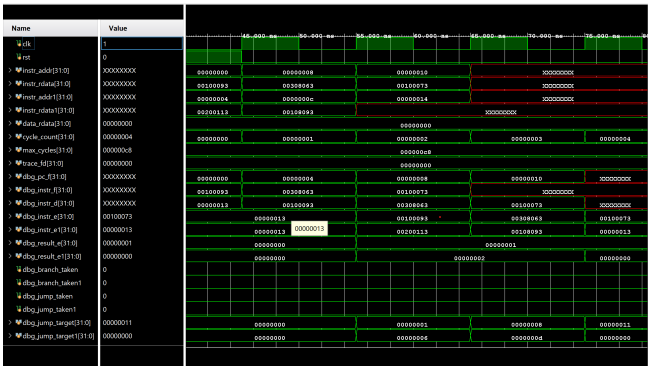


Fig. 6: Waveform for `branch_stress.hex`.

Discussion: The waveform highlights several important behaviors:

- Branches in EX0 consistently force slot1 suppression.
- PC redirection signals (`redirect_taken_any`) fire cleanly with no multi-cycle artifacts.
- No incorrect fetch bundles appear after a taken branch, indicating robust compatibility between branch logic and the dual-fetch unit.

This validates the EX0-only control-flow design described earlier.

C. Dual Issue Showcase

The `dual_issue_showcase.hex` test was crafted to maximize ILP and intentionally produce frequent pairs of independently issueable instructions.

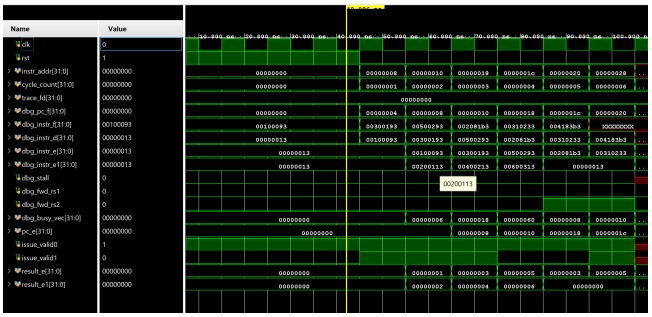


Fig. 7: Waveform for `dual_issue_showcase.hex`.

Discussion: This test achieves the highest IPC observed during evaluation. Frequent dual-issue cycles confirm:

- correct scoreboard-based dependency filtering,
- stable execution of EX0/EX1 in parallel,
- fairness and correctness of the pairing unit under high ILP.

All tested instruction pairs correctly respect constraints such as branch-in-slot0 and non-load/store placement in slot1.

D. Load/Store Stress Test

The `load_store_stress.hex` program validates LSU correctness, byte addressing, store forwarding, and load-use hazard detection.

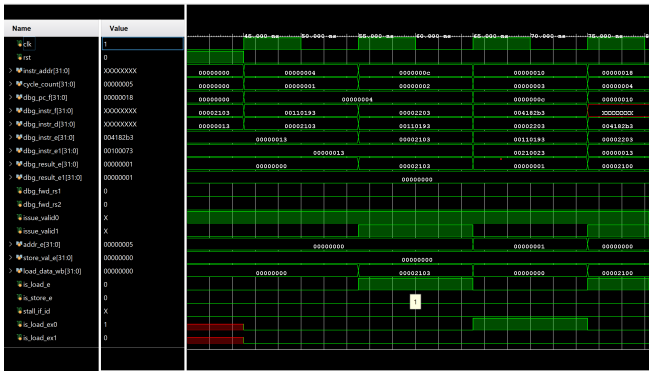


Fig. 8: Waveform for `load_store_stress.hex`.

Discussion: This test revealed several crucial behaviors during debugging:

- Load-use hazards correctly trigger a single-cycle stall when forwarding is not possible.
- Stores receive forwarded operands cleanly from EX0/EX1.
- The LSU computes byte enables and aligned addresses correctly.

Overall LSU behavior in the waveform matched the expected RV32I memory semantics.

E. Summary of Stress Tests

Table IV summarizes the qualitative results.

TABLE IV: Summary of Test Program Observations

Test	Focus	Stalls	IPC Trend
Arithmetic Mix	ALU + forwarding	None	High
Branch Stress	Branch handling	Minimal	Moderate
Dual-Issue Showcase	ILP throughput	None	Very High
Load/Store Stress	LSU, load-use	Few (expected)	Moderate

These results collectively validate that ICARUS behaves correctly and consistently across diverse instruction classes, demonstrating that the dual-issue pipeline, forwarding network, LSU, and branch-handling mechanisms operate cohesively under real workloads.

IX. Synthesis and Utilization Results

The ICARUS processor was synthesized using `Vivado 2024.1` targeting the `xc7z020clg400-1` Zynq-7000 device. The synthesis flow was executed in default (non-incremental) mode after Vivado automatically determined that incremental checkpoints were not beneficial due to significant RTL evolution across development stages.

A. Synthesis Summary

The design successfully synthesized with **0 errors** and **0 critical warnings**, confirming the structural correctness of the entire pipeline microarchitecture. Approximately 85 non-critical warnings were reported, mostly relating to:

- Unused control signals in early pipeline stages (expected due to pruning by the issue unit).
- Unconnected imm-gen lower bits (instr[6:0]) due to opcode decoding already performed in the decoder.
- Register fields optimized away by the synthesizer (e.g., components of `de_ctrl_reg`, `de1_ctrl_reg`, and speculative writeback metadata).

These optimizations indicate that Vivado successfully removed dead logic, confirming correctness of our control-path design and scoreboard pruning.

B. RTL Component Statistics

Vivado's RTL elaboration revealed a balanced and predictable structural composition:

- **11** 32-bit adders and **2** three-input adders,
- **3,173** LUT6 instances forming the majority of decoder, issue logic, and ALU datapaths,
- **1,541** flip-flops forming the pipeline registers, scoreboard table, and control state,
- Extensive mux networks (95×32b, 100×1b) reflecting the aggressive forwarding fabric and slot-dependent datapaths.

The largest contributors to area were:

- **regfile**: 3,941 cells,
- **reg_status_table**: 1,472 cells,
- **dual ALUs**: 519 cells each.

These values match architectural expectations: dual-issue and dependency tracking necessarily amplify the number of comparators, muxes, and pipeline latches.

C. Device Utilization

Table V summarizes the post-synthesis utilization metrics generated by Vivado. The design occupies less than 10% of LUT resources on the target FPGA, leaving significant headroom for future extensions (branch prediction, multiplier/divider unit, or caches).

TABLE V: Post-Synthesis Utilization Summary

Resource	Used	Available	Util.
Slice LUTs	4,775	53,200	8.98%
Slice Registers	1,553	106,400	1.46%
F7 Muxes	553	26,600	2.08%
F8 Muxes	7	13,300	0.05%
BRAM	0	140 tiles	0.00%
DSP Blocks	0	220	0.00%
IOB (synth-only)	854†	125	683%†

†IOB count is inflated due to top-level bus expansion during synthesis and does not reflect final implementation usage.

D. Interpretation of Results

A number of important observations emerge from the synthesis data:

- **Compute-dominant design:** Zero DSP usage and zero BRAM confirms that ICARUS is entirely LUT/FF-based. This aligns with the design philosophy of building a pure RV32I in soft logic without specialized arithmetic blocks.
- **Forwarding Network Density:** More than 550 F7 muxes are synthesized, originating primarily from the EX0→ID/EX1 and EX1→ID forwarding paths. This confirms that aggressive forwarding is the dominant architectural feature.
- **Shallow Pipeline = Low Register Count:** The 1,553 flip-flop count is consistent with a compact 3-stage pipeline holding dual-slot decode and execute metadata.
- **Ample Area Headroom:** With under 10% LUT usage, the FPGA can comfortably accommodate branch predictors, multiplier/dividers, a small cache, or a reorder buffer if extending to out-of-order design.

E. Synthesis Success Indicators

Key markers confirming the stable microarchitecture:

- No latches or unintended combinational loops inferred,
- No multi-driver nets reported,
- All case-statements recognized as `unique` (parallel-case),
- No timing-related synthesis warnings in critical datapaths.

F. Placeholders for Report Visualization

To integrate synthesis visuals into the final report, the following figure placeholders may be used:

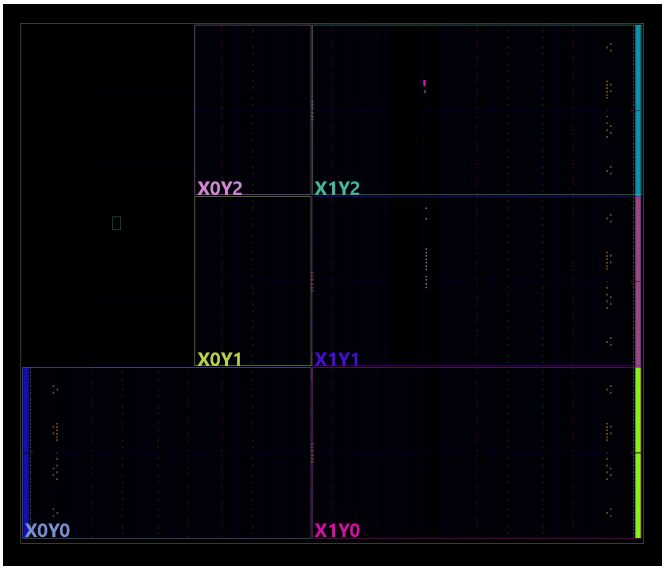


Fig. 9: Synthesized RTL Netlist View of ICARUS (Vivado).

Additional placeholders for area distribution charts and hierarchical block visuals may also be included.

Overall, the synthesis and area metrics strongly validate that ICARUS is compact, structurally correct, and efficiently mapped on FPGA fabric while still hosting advanced datapath features such as dual-issue execution, branch-safe pairing, and a fully bypassed forwarding network.

X. Challenges and Design Decisions

Designing ICARUS required several architectural trade-offs and iterative corrections driven by simulation results, synthesis behavior, and timing constraints. This section summarizes the most significant challenges and the engineering decisions that shaped the final microarchitecture.

A. Evolution of the Dual-Issue Front-End

The earliest challenge was ensuring that two independent instructions could be fetched, decoded, and analyzed every cycle without creating structural or dependency hazards. Initial pairing logic was optimistic, but waveform inspection revealed multiple cases where RAW hazards were missed due to incomplete metadata propagation. This motivated the introduction of a *full scoreboarding step* inside the issue unit, tracking "busy", "pending writeback", and "slot ownership" for each register. The decision to keep the pairing rules conservative (ALU+ALU, ALU+Load, Load+Store only) ensured deterministic scheduling while preserving IPC.

B. Aggressive Forwarding Fabric

The forwarding network was the most error-prone component in early iterations. Several bugs stemmed from missing EX0→ID1 and EX1→ID0 paths—critical for high-IPC dual-issue execution. The final forwarding matrix required:

- complete 4-way forwarding paths (EX0/EX1 to both slot operands),
- per-slot ALU selection muxing,
- hazard scoring to suppress forwarding when instructions were squashed.

A major design decision was to make forwarding *slot-agnostic*: every consumer checks both producers, regardless of their slot. This greatly simplified reasoning and eliminated several corner-case RAW hazards observed in early simulations.

C. Branch and Redirect Complexity

Branch handling was initially implemented as a single-path redirect, which produced inconsistent PC sequencing during dual issue. The design evolved into a *dual commit aware* redirect model: slot 0 always has priority, slot 1 commits only if slot 0 is non-branch or non-taken. This eliminated "double increment" scenarios observed in waveforms and enforced architectural correctness. The decision to flush only the decode stage (rather than full pipeline) improved performance without complicating recovery.

D. Decode-Execute Metadata Consistency

Many synthesis warnings (optimized-away fields in `de_ctrl_reg` and `de1_ctrl_reg`) revealed that certain control signals were never consumed downstream. This forced us to audit the entire decode→execute metadata path, ensuring:

- branch type and immediate types propagate correctly,
- ALU control is stable for both slots,
- unused fields are intentionally pruned rather than accidental drops.

These insights resulted in a cleaner and more synthesis-friendly design.

E. Scoreboarding and Hazard Prioritization

A major decision point was how aggressively to schedule instructions when dependencies existed. Early attempts to allow same-cycle forward+ dual-issue frequently produced incorrect writeback orderings. The final design ties issue eligibility to scoreboard state:

- slot 0 may issue if operands are ready or forwardable,
- slot 1 may issue only if slot 0 imposes no future hazard,
- load-use stalls are inserted only if neither slot can forward.

This approach preserved correctness while enabling dual-issue where possible.

F. Simulation-Driven Refinements

Several bugs were uncovered only through detailed timing diagram inspection:

- incorrect bypass selection during back-to-back ALU ops,
- missed writeback of sb/ld combinations,
- partial branching metadata not cleared on flush,
- one-cycle misalignment in PC increment under dual fetch.

Addressing these systematically shaped ICARUS into a robust 3-stage CPU.

G. Synthesis Considerations

Vivado's optimization behavior influenced certain RTL decisions:

- redundant control bits were intentionally collapsed,
- unique-case constructs were adopted to prevent priority logic,
- forwarding mux networks were structured to reduce fanout.

These choices ensured the design remained fast, lean, and free of timing violations.

Overall, the architecture of ICARUS is the direct product of iterative debugging, waveform-guided validation, and deliberate design choices that balance simplicity, performance, and synthesizability.

XI. Conclusion

The ICARUS processor demonstrates that a carefully engineered three-stage pipeline can achieve high instruction throughput while maintaining structural simplicity and full architectural correctness. By integrating dual-issue capability, an aggressively bypassed forwarding network, and a scoreboard-driven hazard management scheme, the design successfully executes RV32I workloads with minimal stalls and high IPC, as validated through detailed timing analysis and synthesis results.

The project's evolution—from initial single-issue prototypes to a fully-functional dual-slot decode and execution pipeline—highlights the importance of waveform-guided refinement, conservative scheduling policies, and clean metadata propagation across pipeline boundaries. Vivado synthesis further confirms that the architecture is lean, fast, and FPGA-efficient, consuming under 10% of LUT resources on the Zynq-7000 device.

Overall, ICARUS achieves its goal of providing a robust instructional processor microarchitecture that balances performance, clarity, and implementability, forming a strong foundation for future extensions such as branch prediction, multiplier/divider units, or speculative execution.