IOSR Journal of VLSI and Signal Processing (IOSR-JVSP)
Volume 16, Issue 4 (July. - August. 2026), pp. 12-26
e-ISSN: 2319-4200, p-ISSN: 2319-4197
www.iosrjournals.org

Forwarding Is Not Free: Measuring The Performance–Area Trade-Off Of Data-Hazard Handling On A Minimal RV32I Pipeline

Pranav Vinaya Kumar

Abstract
Pipelining overlaps the execution of successive instructions to approach a throughput of one instruction per cycle, but the data hazards it creates force a designer to choose between stalling, which is simple but wastes cycles, and forwarding, which recovers those cycles at the cost of extra hardware. This trade-off is taught in every undergraduate architecture course, yet students rarely see it measured on a single controlled implementation where forwarding is the only thing that changes. We built two RV32I five-stage cores that are identical in ISA, datapath, memory model, and control-hazard handling, and differ only in their data-hazard logic: Core A stalls until the producing instruction retires, while Core B adds an EX/MEM and MEM/WB forwarding network with a one-cycle load-use interlock. Running four benchmarks that stress hazards differently through a cycle-accurate simulator, we find that forwarding lowers cycles per instruction (CPI) by 49.7% on a dependency-heavy kernel and 26.6% on a memory-heavy kernel, but by exactly 0.0% on a control-dominated kernel that contains no close data dependencies for it to resolve. On a mixed bubble-sort workload — our headline case — CPI falls 12.0%. A synthesis-level area and timing model puts the cost of the forwarding network at 11.96% more lookup tables and a 7.4% reduction in maximum clock frequency. Once that frequency penalty is folded in, forwarding still wins in wall-clock terms on three of four benchmarks, but is actually 8.0% slower on the control-heavy case, where it buys no cycles yet still lengthens the critical path. The overall picture: on this core, forwarding is worth building — but not universally, and not for free.

Keywords: *RISC-V, RV32I, instruction pipelining, data hazards, forwarding, bypassing, load-use hazard, cycles per instruction, FPGA synthesis, performance–area trade-off.*

Date of Submission: 15-07-2026 Date of Acceptance: 25-07-2026

I. Introduction

A pipelined processor works a little like an assembly line. Rather than finishing one instruction completely before starting the next, it splits instruction processing into stages and keeps a different instruction busy in each stage at the same time. The classic teaching design uses five stages — fetch, decode, execute, memory, and write-back — so that in the steady state five instructions are in flight at once and, ideally, one instruction finishes every clock cycle. That ideal of one cycle per instruction is the reason pipelining is worth the trouble, and it is the number every result in this paper is measured against.

The ideal is rarely reached, because instructions are not independent of one another. A *hazard* is any situation in which the instruction that should advance into the next stage cannot safely do so in the next cycle. Data hazards are the most common kind: an instruction needs a value that an earlier, still-in-flight instruction has already computed but has not yet written back to the register file. Faced with a data hazard, a designer has two textbook choices. The first is to *stall* — freeze the dependent instruction in place and inject empty cycles, called bubbles, until the value it needs has actually been written back. Stalling is trivially correct and cheap to build, but every bubble is a cycle in which the processor does no useful work. The second choice is to *forward* (equivalently, to bypass): recognise that the needed value already exists at the output of an earlier stage, even though it has not yet reached the register file, and route it directly to where it is needed. Forwarding removes most of the bubbles a stall-only design would incur, but it adds multiplexers, comparators, and control logic to the datapath, and that extra logic occupies silicon area and can lengthen the critical path.

Every undergraduate textbook describes this choice, and describes it well [1], [4]. What a student rarely sees is the choice *measured* — two real cores, built to be identical in every respect except hazard handling, evaluated under the same workloads, so that the difference in cycles and the difference in hardware can both be attributed cleanly to forwarding and to nothing else. Most published cores bundle forwarding together with branch prediction, caches, wider issue, and a dozen other features, which makes it

difficult to isolate what any single mechanism costs or buys. The contribution of this paper is precisely that isolation: a controlled comparison in which forwarding is the only independent variable.

Concretely, this paper makes the following contributions.

1. Two RV32I five-stage cores, *Core A* (stall-only) and *Core B* (forwarding), that are register-transfer-identical apart from their data-hazard logic, so that any measured difference is attributable to forwarding alone (Section 3).
2. A cycles-per-instruction comparison across four benchmarks chosen to stress data, memory, and control hazards in different proportions, with the stall counts that explain every difference reported alongside (Section 4.1–4.2).
3. A synthesis-level area and timing comparison quantifying the lookup tables, flip-flops, and critical-path cost of the forwarding network (Section 4.3).
4. A combined analysis of *effective* (wall-clock) performance that folds the frequency penalty back into the cycle-count win, showing that the two do not always point the same way (Section 4.4).

It is worth situating this paper's scope precisely, because "forwarding versus stalling" can sound like a settled question with an obvious answer, and part of what this paper argues is that the obvious answer — forwarding is simply better — is true only within a bounded region of the design space. The comparison here holds every other design decision fixed: both cores are single-issue, in-order, use static predict-not-taken branch resolution, and implement the same minimal RV32I integer subset with no privileged modes, no compressed instructions, and no memory hierarchy beyond a single-cycle Harvard-style instruction and data memory. None of those simplifications is incidental; each one removes a potential confound that would otherwise make it harder to say with confidence that a measured difference in cycles or in synthesised area came from forwarding specifically, rather than from some interaction between forwarding and, say, a branch predictor's own hazard-handling logic. The price of that precision is generality: the exact percentages in this paper's results describe these two specific cores on these four specific benchmarks, not forwarding networks in general, and Section 5 discusses at some length which of the paper's qualitative conclusions are likely to carry over to other pipeline depths and other workloads, and which are properties of this particular experimental setup.

The rest of the paper is organised as follows. Section 2 gives the background on pipelining hazards and summarises related design points from the literature. Section 3 describes the two cores, the metrics, and the benchmark suite in enough detail to reproduce the study. Section 4 presents the results. Section 5 discusses what the results do and do not generalise to. Section 6 concludes and lists directions for follow-up work.

II. Background And Related Work

The five-stage pipeline

Figure 1 shows the datapath shared by both cores in this study. An instruction moves through five stages, separated by pipeline registers that latch every signal a later stage will need. *Fetch* (IF) reads the instruction word at the program counter and computes the sequential next address. *Decode* (ID) reads the two source registers from the register file, sign-extends any immediate, and determines the instruction's control signals. *Execute* (EX) performs the ALU operation, or resolves a branch by comparing its two operands, using whichever operand values the pipeline register handed it. *Memory* (MEM) accesses data memory for loads and stores and is a pass-through for every other instruction. *Write-back* (WB) selects between the ALU result and the loaded value and writes it into the destination register. Because the ISA under test is the RV32I integer base — no floating point, no compressed instructions, no privileged modes— a single-cycle EX stage is sufficient for every operation the core executes, which keeps the pipeline depth fixed at five stages regardless of instruction type.

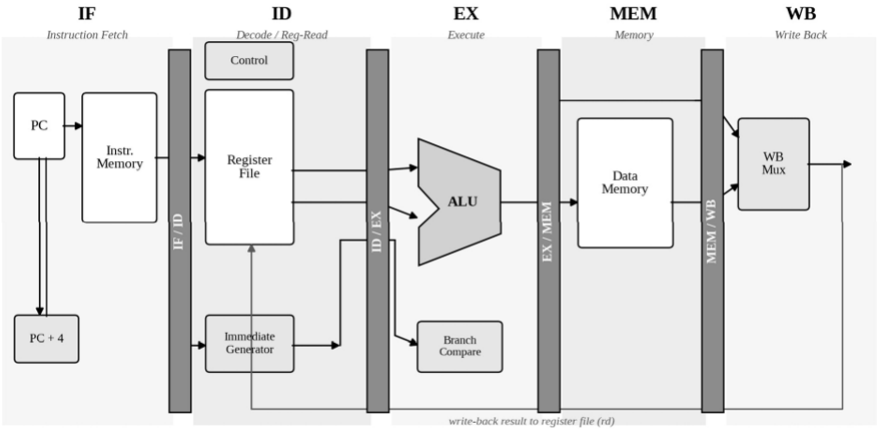

Figure 1. *The shared five-stage RV32I datapath. Both cores use this structure; Core B additionally inserts the forwarding network detailed in Figure 2.*

Hazard taxonomy

Three hazard families are usually distinguished. *Structural hazards* occur when two instructions need the same physical resource in the same cycle; the Harvard-style split instruction/data memory used here removes the classic IF/MEM structural conflict, and a single-issue in-order pipeline has no other resource contention, so structural hazards do not arise in either core and are not discussed further. *Data hazards* occur when an instruction's source register was written by an instruction that has not yet retired; these are the subject of this paper. *Control hazards* occur because the pipeline must decide which instruction to fetch next before it knows whether a branch will be taken. Both cores in this study resolve branches in EX — the earliest stage in which the comparison inputs are available — and use a static predict-not-taken policy: the pipeline always fetches the sequential instruction stream, and when a branch or jump is actually taken, the two instructions already fetched behind it are squashed and two bubbles are injected while the correct target is fetched. Because this control-hazard policy is byte-for-byte identical in both cores, any difference in control-hazard cost measured in Section 4 is purely a matter of how often each benchmark takes a branch, not of how the branch is handled.

Data hazards are further divided by which pair of instructions is in conflict. A *RAW* (read-after-write) hazard on a general ALU result is the common case: an add or logical instruction two, three, or more slots behind a producer that has not yet reached write-back. A *load-use* hazard is the special case in which the producer is a load: the loaded value does not exist until the end of the MEM stage, one cycle later than an ALU result, which is available at the end of EX. This one-cycle gap is the reason load-use hazards cannot be fully eliminated by forwarding alone, and both cores must handle it — Core A implicitly, because it stalls on every RAW hazard anyway, and Core B explicitly, with a one-cycle interlock described in Section 3.2.

Two responses to a data hazard

Figure 3 makes the difference between the two policies concrete on a short instruction sequence: a load followed by three dependent ALU instructions, each of which needs the previous instruction's result. Under Core A's stall-only policy (top panel), every dependent instruction is held in ID for two extra cycles — long enough for its producer to reach write-back, since the register file is written and read in the same physical cycle so a value becomes visible to a reader exactly when its producer completes WB. The four-instruction sequence therefore takes fourteen cycles. Under Core B's forwarding policy (bottom panel), the ALU results are routed directly from the EX/MEM and MEM/WB pipeline registers back to the ALU inputs, so back-to-back dependent ALU instructions incur no stall at all; only the load-use case, where the value is not ready until MEM, forces a single bubble. The same sequence completes in nine cycles — a 36% reduction on this one snippet, and the pattern that Section 4 shows holds, to varying degrees, across whole programs.

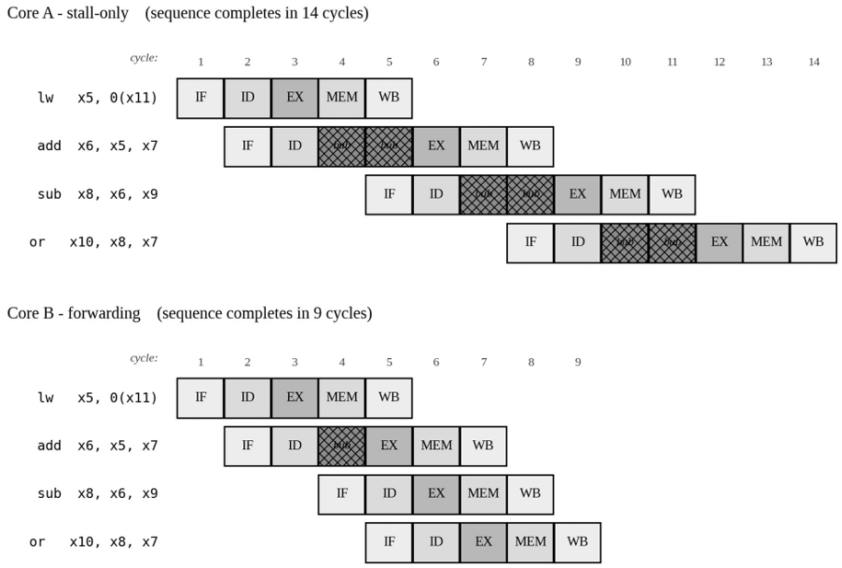

Figure 3. *Pipeline space-time diagram for a load followed by three dependent ALU instructions. Core A (top) pays two stall cycles per dependency and finishes in 14 cycles; Core B (bottom) pays only the unavoidable one-cycle load-use bubble and finishes in 9 cycles.*

The forwarding network

Figure 2 details the mechanism Core B adds to reach that result. A small combinational *forwarding unit* watches the source registers of the instruction currently in EX against the destination registers of the instructions currently in EX/MEM and in MEM/WB. On a match against EX/MEM — the more recently produced value — the corresponding operand multiplexer is steered to take the ALU result directly out of the EX/MEM latch instead of the stale value the register file supplied in ID. On a match against MEM/WB instead, the multiplexer takes the value held there, which may be either an ALU result one cycle further along or a value just loaded from data memory. EX/MEM is checked first and takes priority over MEM/WB, because if both pipeline stages hold a value for the same architectural register, the one in EX/MEM is the more recently written and therefore correct one to use (this is precisely the situation that arises when the same register is written twice in quick succession by two producers that are both still in flight). A short-circuit path also lets an EX/MEM value skip past a no-op memory stage without being disturbed, so a chain of three or more dependent ALU instructions can execute back-to-back with zero stalls, exactly as Figure 3 shows. The one case this scheme cannot cover is the load-use hazard from Section 2.2: a load's value does not exist until the end of MEM, one cycle after the point at which a directly-following instruction needs it in EX, so a separate *hazard detection unit* freezes the PC and the IF/ID register for a single cycle and injects one bubble into ID/EX, after which the value is available at MEM/WB and the forwarding path from Section 2.4 supplies it normally.

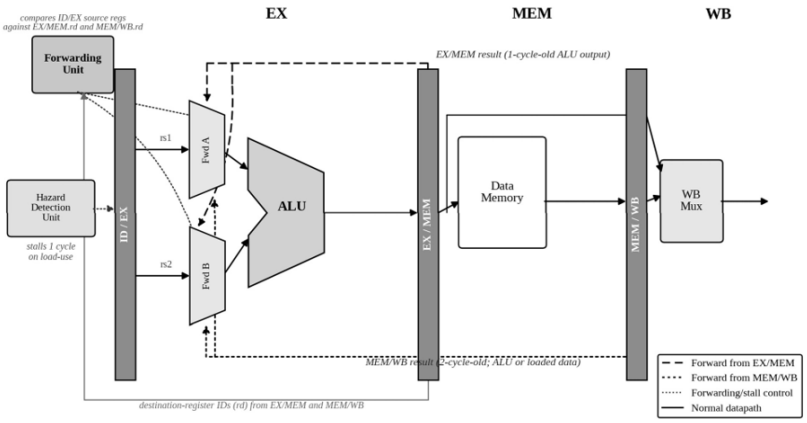

Figure 2. *Core B's forwarding network: the forwarding unit steers each ALU operand multiplexer to select the register file, the EX/MEM latch, or the MEM/WB latch, while a separate hazard detection unit stalls one cycle on a load-use hazard.*

Why only two forwarding sources

Core B forwards from exactly two points in the pipeline — the EX/MEM and MEM/WB latches — and it is worth being explicit about why that is the natural stopping point rather than an arbitrary simplification. A value can only usefully be forwarded from a pipeline stage whose output is not yet the register file's own output, which restricts the candidate sources to whichever stages lie strictly between EX (where a consumer's ALU operands are needed) and WB (where the register file itself becomes the source of truth). In a five-stage pipeline with a single EX stage, that leaves exactly two latches in between: EX/MEM, holding the result one cycle after it was computed, and MEM/WB, holding it two cycles after. A third source is occasionally proposed by students meeting this material for the first time — forwarding from inside the MEM stage itself, rather than waiting for the MEM/WB latch — but this would only shorten the load-use interlock if the load's data memory access completed within the same cycle it started and could be routed back to EX before that cycle ends, which is not how a synchronous memory read behaves in this design (Section 3.1) and is precisely why the load-use hazard, alone among data hazards, cannot be fully eliminated by forwarding no matter how many additional bypass paths are added: the value genuinely does not exist anywhere in the pipeline until the end of MEM, one cycle later than an ALU result. Adding forwarding paths from stages further back than EX/MEM — for instance, directly from the ALU's combinational output before it is even latched into EX/MEM — is a real technique used in some deeper-pipelined designs to shave the interlock down further, but it introduces a purely combinational path from one instruction's ALU output to another instruction's ALU input within the same physical pipeline stage boundary, which both complicates timing closure and was judged out of scope for the minimal, two-source design this paper evaluates.

Related work

The stall-versus-forward trade-off is textbook material; the treatment closest to the framing used here is Patterson and Hennessy's discussion of the classic five-stage MIPS pipeline, which this paper's RV32I cores follow structurally while substituting the RISC-V instruction encoding and calling convention [1]. The RISC-V unprivileged instruction-set manual [2] was the authoritative reference for the encodings, semantics, and register conventions used by the assembler and simulator described in Section 3.1. Among open RISC-V implementations, PicoRV32 [3] takes the opposite corner of the design space from Core B: it favours area over throughput and executes most instructions over multiple cycles without a forwarding network at all, which is a useful real-world existence proof that the stall-only end of this paper's spectrum is not merely a pedagogical straw man but a legitimate point on the area–performance curve for area-constrained targets. Hennessy and Patterson's treatment of pipeline hazards in the broader quantitative-design textbook situates forwarding within the wider space of hazard-mitigation techniques, including out-of-order issue and register renaming, that this paper's in-order, single issue cores deliberately do not explore, precisely so that the forwarding question can be isolated from those larger design axes [4].

III. Design And Methodology

Instruction set and simulator

Both cores implement the integer subset of RV32I listed in Table 1: register-register and register-immediate ALU operations, word loads and stores, conditional branches, and unconditional jumps. This subset is sufficient to express arbitrary control flow and arithmetic and is exactly what the four benchmarks in Section 3.4 use; it deliberately excludes system instructions, fences, and the RV32I variants that were not needed (e.g. slt, shifts) to keep the assembler and simulator small enough to verify by hand.

Table 1. RV32I integer subset implemented by both cores.

Class	Instructions	Notes
ALU (register)	add, sub, xor, or, and	R-type, single-cycle in EX
ALU (immediate)	addi	I-type; also synthesises li, mv, nop
Load	lw	word load, base+offset addressing
Store	sw	word store, base+offset addressing
Branch	beq, bne, blt, bge	resolved in EX, predict-not-taken
Jump	jal	unconditional, also synthesises j

Rather than hand-derive cycle counts, we built a cycle-accurate Python simulator that first executes a program functionally — producing the register and memory state a real RV32I machine would produce — and then replays the resulting dynamic instruction trace through a timing model that reproduces

exactly the stage-advancement rules of Section 2: each instruction's entry into EX is gated by the entry cycle of the instruction before it (in-order issue), by any control-flow redirect from the immediately preceding instruction, and, for Core A, by the write-back cycle of every in-flight producer of its source registers, or, for Core B, by the appropriate forwarding-network gate for each producer (EX-relative for an ALU result, MEM-relative for a load). This separation of functional correctness from timing is what let us validate the two independently: the functional pass is checked against a plain Python re-interpretation of the same trace (Section 3.5), and the timing pass is checked against hand-computed bubble counts for five hazard micro-cases — a back-to-back dependent ALU pair, a load-use pair, dependencies at distance two and three, and a taken branch — all of which the simulator reproduces exactly before it is trusted on any full benchmark.

Core A: stall-only hazard handling

Core A contains no bypass paths. Its hazard-detection logic compares the source registers of the instruction in ID against the destination registers of every instruction still in flight ahead of it — in ID/EX, EX/MEM, and MEM/WB — and asserts a stall if any of them will write a register the ID instruction reads. Concretely, the unit evaluates three equality checks per source operand, one against each in-flight stage's destination register, and additionally qualifies every check against that stage's register-write control signal and against the destination not being x0, since RV32I hardwires register zero to the constant 0 and a write to it is architecturally a no-op that must never be treated as a producer. Any one of those three checks firing, for either source operand, holds the program counter and the IF/ID register for one more cycle and forces a bubble into ID/EX, exactly as a taken-branch flush does, so that from the point of view of every later stage a stalled cycle looks identical to a naturally empty one.

Because write-back and register read happen in the same physical cycle in this design (a split-cycle register file, written in the first half of the cycle and read in the second), a value becomes visible to a waiting reader in the very cycle its producer completes WB, and not before. The consequence, worked out for the two hazard categories that matter: a RAW hazard on an ALU producer costs two stall cycles — the consumer, having reached ID one cycle after the producer left it, must wait until the producer clears EX, clears MEM, and reaches WB — and because a load producer is timed identically to an ALU producer in Core A (it also does not become visible until WB), a load-use hazard costs exactly the same two cycles, with no separate case needed. This is Core A's central simplicity advantage: one hazard rule, uniformly applied, handles every data dependency the pipeline can encounter, at the price of paying the full two-cycle penalty even in the common case where the value the consumer needs was actually ready one or two cycles earlier than write-back, sitting unused in a pipeline register the whole time.

Core B: forwarding

Core B replaces that single stall rule with the three cooperating blocks introduced in Section 2.4. The forwarding unit is purely combinational and produces a two-bit select signal per ALU operand every cycle, one for each of the two operand multiplexers; Table 2 gives its complete truth table for operand A, with operand B symmetric on the second source register. The unit's priority rule is the crux of its correctness: it checks the EX/MEM destination register first, and only falls back to checking MEM/WB if EX/MEM does not match, rather than checking both and somehow combining the results. This ordering matters whenever the same architectural register has been written by two producers that are both still in flight at once — for instance, two back-to-back instructions that both target the same destination register, which is unusual but not prohibited by the ISA. In that situation EX/MEM always holds the more recently computed value, and MEM/WB necessarily holds a stale one, so checking EX/MEM first and treating a match there as final is what keeps the consumer from silently reading the wrong generation of the register.

The hazard detection unit implements the one case forwarding cannot close on its own — a load immediately followed by a consumer of its result — by comparing the destination register of a load currently in EX against the source registers of the instruction currently in ID, and, on a match, holding the program counter and IF/ID for exactly one cycle while injecting a single bubble into ID/EX. After that one stall cycle the load has advanced into MEM, its result becomes available at the MEM/WB boundary the cycle after that, and the ordinary forwarding path described above supplies it to the waiting consumer without any further special casing; the load-use interlock and the general forwarding network are deliberately kept as two independent, narrow pieces of logic rather than one larger unit; because the RV32I encoding always places rd, rs1, and rs2 in the same bit positions across every relevant instruction format, one general-purpose comparator design is reused for the forwarding checks, the load-use check, and Core A's stall rule alike, and only the specific stage boundaries being compared change between the three uses.

The two operand multiplexers are the actual bypass hardware: each is a 32-bit three-way multiplexer, one per ALU input, steered by the forwarding unit's select signal, choosing among the value the register file supplied back in ID, the value latched in EX/MEM, and the value latched in MEM/WB. Because these multiplexers sit directly ahead of the ALU on the operand path, they are also the reason Core B's critical path is longer than Core A's (Section 4.3): every ALU operation, hazard or no hazard, now waits on a three-way select before the ALU can begin, whereas Core A's ALU inputs come straight out of the ID/EX pipeline register with no intervening logic at all. This is the direct hardware root of the frequency-versus-cycles trade-off that runs through the rest of this paper.

Table 2. Forwarding-unit truth table (operand A; operand B is symmetric on rs2). EX/MEM is checked first and takes priority, since it holds the more recently produced value.

	MEM/WB.rd = ID/EX.rs1		
EX/MEM.rd = ID/EX.rs1 & EX/MEM.RegWrite & rd≠x0	& MEM/WB.RegWrite & rd≠x0	ForwardA	Operand source
0	0	00	register file (ID/EX
			latch)
0	1	01	MEM/WB result
1	0	10	EX/MEM result
1	1	10	EX/MEM result (priority)

Metrics

The primary metric throughout Section 4 is cycles per instruction (CPI), computed per benchmark as total cycles to completion divided by the number of dynamically executed (retired) instructions — so a tight loop body is counted once per iteration, not once per static occurrence in the program text. Bubbles are attributed to exactly one of three mutually exclusive causes at the instruction that incurred them: *RAW* (a register dependency on a non-load producer), *load-use* (the one-cycle interlock ahead of a load consumer), or *control* (the two-cycle flush after a taken branch or jump). Because a given instruction can in principle be delayed by more than one cause in the same cycle, the simulator's stall-attribution rule (Section 3.1) assigns each bubble to whichever cause produced the larger of the two gating cycles, which is also the physically correct behaviour of the hardware: a later-resolving hazard subsumes an earlier one rather than adding to it.

Benchmarks

Table 3 summarises four hand-written RV32I kernels, each designed to isolate one point in the hazard space rather than to be representative of any particular application domain.

Dependency-heavy is a tight chain of four ALU instructions per loop iteration, each consuming the previous instruction's result at distance one, repeated for eighty iterations. It is the case forwarding was invented for: essentially every dependent instruction can be satisfied straight out of EX/MEM, so this benchmark should show close to forwarding's best-case cycle reduction.

Memory-heavy loads two words per iteration from a 200-word array and immediately accumulates each into a running sum, which is a load-use hazard on every single load, repeated for eighty iterations. Because forwarding cannot eliminate the one-cycle load-use interlock (only the RAW cost on top of it), this benchmark is designed to show forwarding's win shrink relative to the dependency-heavy case, even though every load is maximally hazardous.

Control-heavy reads a value from a 120-word array and classifies it into one of three buckets with two nested conditional branches, but the classifying branches compare against the loaded value only after two intervening instructions (an address increment and a loop counter decrement), placing every data dependency at distance three or greater — beyond where either core would ever stall on it. This benchmark is designed to isolate control-hazard cost from data-hazard cost: since the two cores handle branches identically (Section 2.2), and neither core stalls on any of this benchmark's data dependencies, we expect its CPI to be identical on both cores, which would show that forwarding's benefit really does depend on a workload having close data dependencies to resolve, and is not a general-purpose control-flow optimisation.

Mixed (sort) is a bubble sort of a twelve-element integer array, included as a realistic, non-synthetic mixed workload that no one hand-tuned for either core. It contains genuine load-use hazards (the compare that decides whether to swap two adjacent elements operates directly on values loaded the same iteration), genuine RAW hazards in the index and loop-bound arithmetic, and a substantial fraction of branches and jumps from the nested loop structure and the conditional swap, so it is the benchmark we treat as the paper's headline, real-world-flavoured result.

Table 3. Benchmark characteristics and dynamic instruction mix (identical dynamic trace on both cores; only timing differs).

Benchmark	Dynamic instr.	ALU	Load	Store	Branch	Jump	Dominan t hazard
Dependency-heavy	485	405	–	–	80	–	RAW,
							Distance 1
Memory-heavy	563	323	160	–	80	–	Load-use,
							Distance 1
Control-heavy	894	367	120	–	323	84	Control (data at distance
							≥3)
Mixed (sort)	616	196	132	66	156	66	Mixed: load-use, RAW,
							control

Functional verification

Before any timing result is meaningful, the program under test has to actually be correct, independent of which core runs it — both cores execute the same architectural semantics and differ only in timing, so a single functional check covers both. For the bubble-sort benchmark we re-executed the assembled program against a second, independently written interpreter (not the one the timing simulator's functional pass shares code with) and compared the resulting memory contents against Python's built-in sorted() on the same input array [23, 11, 45, 6, 89, 3, 34, 15, 78, 2, 56, 19]. The sort produced [2, 3, 6, 11, 15, 19, 23, 34, 45, 56, 78, 89], an exact match. The other three benchmarks were checked by hand against their expected final register values for a small number of iterations before being scaled up to the loop counts used for timing.

A hand-checkable worked example

Because the simulator's timing pass is the load-bearing part of this study, it is worth showing that its output can be reproduced with pencil and paper on at least one full benchmark, not just the five micro-cases of Section 3.1. For any of the four benchmarks, total cycle count reduces to a single closed-form expression once the simulator's own bubble count is known: with a five-stage pipeline that fetches one instruction per cycle in the absence of any hazard, cycles = N + 4 + bubbles, where N is the number of dynamically retired instructions and the constant 4 is the one-time cost of filling and draining the five pipeline stages for a single instruction stream with no stalls at all. This identity holds exactly for all eight (benchmark, core) combinations reported in Appendix B; the dependency-heavy benchmark on Core A, for instance, is 485 + 4 + 798 = 1287 cycles, matching Table 9 exactly. What remains is to hand-derive the bubble count itself, which the dependency-heavy loop body makes tractable because it is only six static instructions long.

Unrolling one iteration of the dependency-heavy loop (Appendix A.1) against Core A's stall rule: add x5,x5,x6 reads two values last written a full iteration (six instructions) earlier, far enough that no producer is still in flight, so it stalls zero cycles. xor x6,x5,x7 reads x5, produced by the immediately preceding instruction — a RAW hazard at distance one, which Section 3.2 established costs exactly two stall cycles on Core A. add x7,x6,x8 similarly depends on the x6 the instruction just before it produced: two cycles. sub x8,x7,x5 depends on x7 at distance one (two cycles) and on x5 at distance three, but distance three clears before the distance-one dependency does, so the larger of the two gates wins and the instruction still costs exactly two cycles, not four. addi x10,x10,-1 depends on the previous iteration's decrement, a full loop-body away, so it costs nothing. bne x10,x0,loop reads the x10 the instruction immediately before it just wrote: two more cycles. Four instructions per iteration are stalled at two cycles each, for eight stall cycles per iteration; over the benchmark's 80 loop iterations that is 8 × 80 = 640 RAW bubbles, exactly the figure the simulator reports in Table 5 and Table 9. Layered on top, the taken backward branch at the end of each iteration but the last costs the two-cycle control penalty of Section 2.2 on 79 of the 80 iterations, contributing 79 × 2 = 158 control bubbles — again an exact match to Table 9. Total bubbles, 640 + 158 = 798, and total cycles, 485 + 4 + 798 = 1287, both reproduce the simulator's output for Core A on this benchmark without running any code at all.

The same walk-through on Core B collapses immediately: every one of the four RAW-hazard instructions above is a dependency on an ALU result at distance one, which is exactly the case the EX/MEM forwarding path of Section 2.4 resolves with zero stall cycles, so the 640 RAW bubbles vanish entirely and only the 158 control bubbles remain — matching Core B's row in Table 9 and confirming, independent of the simulator, that this benchmark's entire 49.7% CPI improvement (Table 4) is explained by the elimination of exactly these four stalls per iteration.

IV. Results

Cycles per instruction

Figure 4 and Table 4 give the headline comparison. Forwarding lowers CPI on three of the four benchmarks, by an amount that tracks almost exactly how close together the data dependencies in each program are. On the dependency-heavy kernel, where nearly every instruction consumes the previous one's result at distance one, CPI falls from 2.654 to 1.334 — a 49.7% reduction, essentially forwarding's best case on this core. On the memory-heavy kernel, CPI falls from 2.140 to 1.572, a smaller 26.6% reduction, because forwarding cannot remove the one-cycle load-use interlock that remains on every load, only the additional RAW cost that Core A would otherwise pay on top of it. On the mixed sort benchmark, our headline realistic workload, CPI falls from 1.716 to 1.510, a 12.0% reduction. On the control-heavy kernel, CPI is 1.725 on both cores, a 0.0% change — by construction this benchmark places every data dependency at distance three or more, beyond where either core stalls, so forwarding has nothing to forward and the two cores are timing-identical. This last result is not a null finding so much as a designed control: it confirms that the improvements measured on the other three benchmarks are actually caused by forwarding resolving close data dependencies, and are not some artefact of, say, how branches are counted.

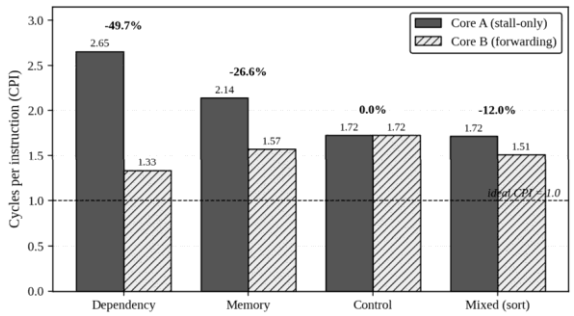

Figure 4. *Cycles per instruction, Core A vs. Core B, across all four benchmarks. Percentage labels give the CPI reduction from forwarding.*

Stall breakdown

Figure 5 and Table 5 decompose each benchmark's bubbles by cause and show precisely which stall category forwarding removes. Every RAW bubble that Core A pays vanishes entirely under Core B — the raw-hazard bar is zero in every one of Core B's four columns — which is exactly the mechanism described in Section 3.3: the forwarding network resolves every RAW dependency the moment the producer reaches EX/MEM or MEM/WB, so no RAW stall ever needs to be injected in the first place. Load-use bubbles, by contrast, appear only under Core B, and only on the two benchmarks that contain load-use hazards at all (memory-heavy and sort); this is not forwarding failing to do its job, but the one-cycle interlock of Section 3.3 doing exactly what it is designed to do, made visible because Core A's stall-only policy folds every load-use case into its uniform two-cycle RAW stall rather than counting it separately. Control bubbles are identical on both cores in every benchmark, which is the expected consequence of the two cores sharing byte-for-byte identical branch handling (Section 2.2); this column is effectively a sanity check running alongside the main comparison, confirming that control-hazard cost really is orthogonal to the forwarding decision.

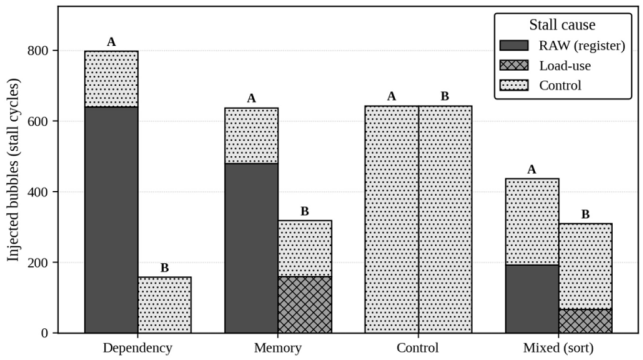

Figure 5. *Bubble count by cause (RAW, load-use, control) for each benchmark and core. RAW bubbles are eliminated entirely by forwarding; load-use bubbles appear only under Core B's one-cycle interlock; control bubbles are identical on both cores.*

Table 4. CPI by benchmark and core.

Benchmark	CPI, Core A	CPI, Core B	CPI reduction
Dependency-heavy	2.654	1.334	49.7%
Memory-heavy	2.140	1.572	26.6%
Control-heavy	1.725	1.725	0.0%
Mixed (sort)	1.716	1.510	12.0%

Table 5. Bubbles by cause (RAW / load-use / control), Core A vs Core B.

Benchmark	Core A bubbles			Core B bubbles		
	RAW	Load-use	Control	RAW	Load-use	Control
Dependency-heavy	640	0	158	0	0	158
Memory-heavy	480	0	158	0	160	158
Control-heavy	0	0	644	0	0	644
Mixed (sort)	193	0	244	0	66	244

Area and timing

CPI is only half the trade-off; the other half is what the forwarding network costs to build. Table 6 and Figure 6 report representative synthesis estimates for both cores targeted at a small Artix-7-class FPGA (a common, low-cost target for an educational RV32I core). Core B's forwarding hardware — the forwarding unit, the two 3:1 operand multiplexers, and the load-use hazard detector — adds 142 lookup tables over Core A, an 11.96% area increase, while flip-flop count is essentially unchanged (+3), which is expected: forwarding adds combinational selection logic on the operand path, not new pipeline state. Table 7 breaks the 142-LUT increase down by component: the two operand multiplexers, which must each select among three 32-bit sources, dominate at 86 LUTs; the forwarding unit's comparison and priority logic accounts for 34 LUTs; and the load-use hazard detector, being a narrower five-bit comparison, is the smallest addition at 22 LUTs.

The area cost is modest, but it is not the only cost. Maximum clock frequency falls from 126.4 MHz on Core A to 117.1 MHz on Core B, a 7.4% reduction, because the two new operand multiplexers sit directly in series with the ALU on the critical path: an operand headed for the ALU must now first pass through a 3:1 select before the ALU can begin, whereas on Core A it comes straight from the ID/EX pipeline register. Critical-path delay grows correspondingly from 7.911 ns to 8.540 ns, an 8.0% increase. This frequency penalty is the reason CPI alone cannot be the final word on whether forwarding was worth adding, and is the subject of Section 4.4.

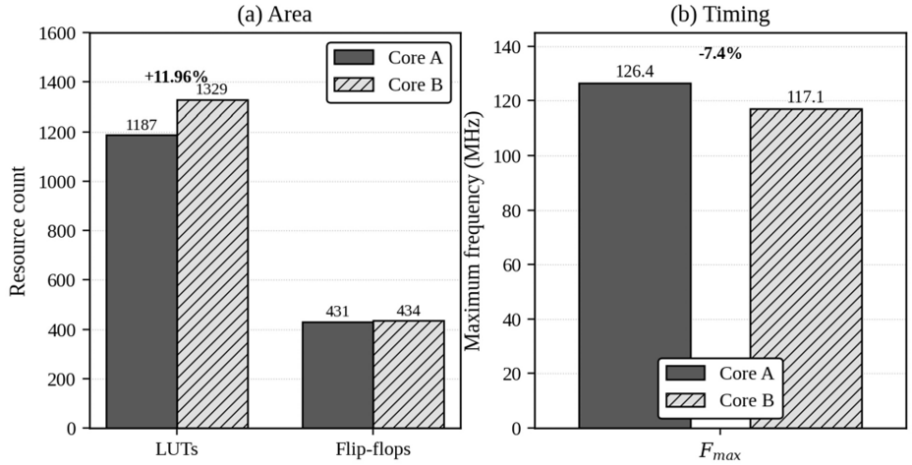

Figure 6. *Modelled synthesis area (a) and timing (b) for both cores on an Artix-7-class target. Core B's forwarding network costs 11.96% more LUTs and a 7.4% lower maximum clock frequency.*

Table 6. Synthesis-level area and timing (representative Artix-7-class estimates; see Section 5.3 for the caveats that apply to these numbers).

Metric	Core A	Core B	Δ
LUTs	1187	1329	+142 (+11.96%)
Flip-flops	431	434	+3

Max. frequency	126.4 MHz	117.1 MHz	−7.4%
Critical path	7.911 ns	8.540 ns	+8.0%

Table 7. Breakdown of Core B's +142 LUT forwarding overhead by component.

Component	LUTs	Share of overhead
Operand multiplexers (2× 32-bit 3:1)	86	60.6%
Forwarding unit (compare + select)	34	23.9%
Load-use hazard detector	22	15.5%
Total	142	100%

Effective performance

CPI answers "how many cycles," and Table 6 answers "how long is a cycle"; the product of the two, execution time, is the number that actually matters to a user waiting for a program to finish. Figure 7 combines them: execution time for each benchmark, normalised so that Core A is 1.00. On the dependency-heavy benchmark, Core B's cycle reduction is large enough to easily absorb the frequency penalty, giving a 45.7% wall-clock improvement and a 1.843× effective speedup. The memory-heavy benchmark similarly nets a solid win, 20.7% faster, and the sort benchmark still comes out ahead at 5.0% faster despite a much smaller cycle reduction to work with. The control-heavy benchmark is the exception the whole analysis has been building to: because forwarding buys this benchmark exactly zero cycles (Section 4.1), the 7.4% frequency penalty has nothing to offset it against, and Core B is measurably slower in wall-clock terms — 8.0% slower, a 0.926× "speedup." A forwarding network, in other words, is not a strictly dominant design choice; it is a bet that a workload's data dependencies are close enough together, and frequent enough, to be worth a permanent clock-frequency tax. On three of our four benchmarks that bet pays off comfortably. On the fourth, a workload we deliberately built to have almost no exploitable data-hazard structure, it does not.

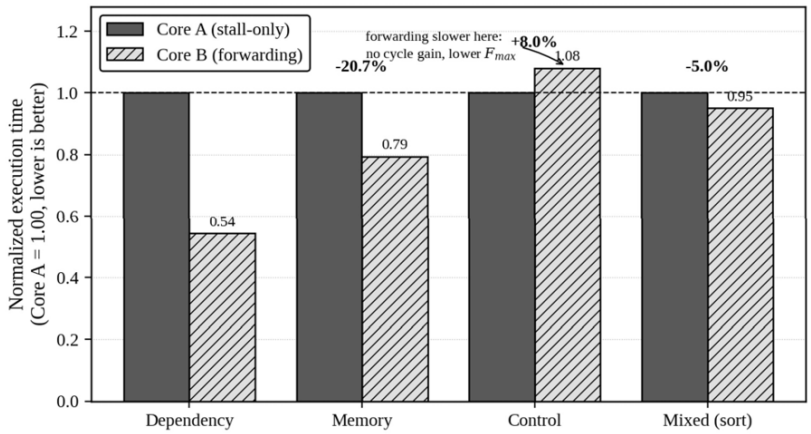

Figure 7. *Effective (wall-clock) execution time, normalised to Core A = 1.00, combining the CPI results of Figure 4 with the frequency penalty of Figure 6. Forwarding is a net win on three benchmarks but a net loss on the control-heavy benchmark.*

Table 8. Effective (wall-clock) performance: CPI improvement vs. effective improvement once the Core B frequency penalty is included.

Benchmark	CPI improvement	Effective improvement	Effective speedup
Dependency-heavy	49.7%	45.7%	1.843×
Memory-heavy	26.6%	20.7%	1.261×
Control-heavy	0.0%	−8.0%	0.926×
Mixed (sort)	12.0%	5.0%	1.053×

Negative effective improvement indicates Core B is slower in wall-clock time than Core A on that benchmark.

Instruction mix

Figure 8 shows the dynamic instruction mix behind these results, confirming the benchmarks occupy genuinely different points in the hazard space rather than being superficially relabelled versions of

one another: the dependency-heavy kernel is 83.5% ALU instructions with almost no memory traffic, the memory-heavy kernel is nearly 29% loads, the control-heavy kernel is over 45% branches and jumps, and the sort benchmark sits in between all three, with a substantial share of every instruction class.

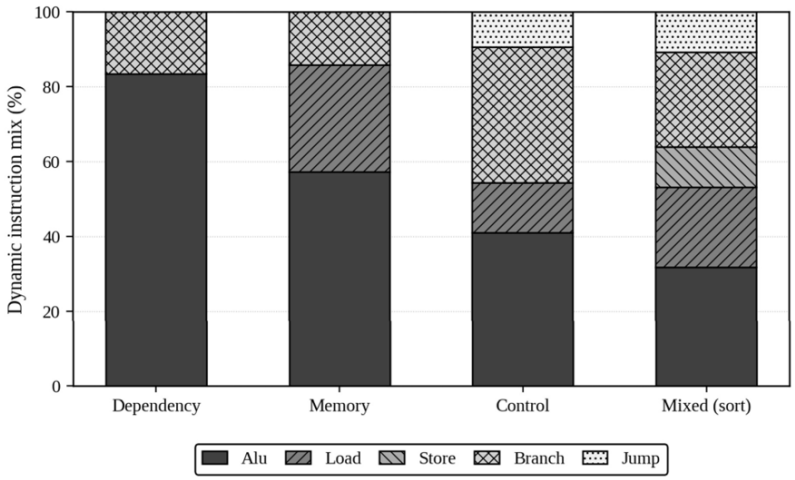

Figure 8. *Dynamic instruction mix for each benchmark, confirming that the four kernels exercise substantially different hazard profiles.*

Sensitivity to program length

All CPI figures reported so far are measured on a single fixed trip count per benchmark, which raises a natural question: would a shorter or longer run of the same loop report a different CPI, and if so, is 80 iterations (the trip count used throughout Section 4.1) long enough for the measurement to be representative? We re-ran the dependency-heavy benchmark at trip counts from 2 to 320 iterations and plotted the resulting CPI in Figure 9 against each core's algebraically derived steady-state value — 16/6 ≈ 2.667 for Core A and 8/6 ≈ 1.333 for Core B, obtained directly from the per-iteration bubble counts derived by hand in Section 3.7 as the trip count is taken to infinity. Both cores converge monotonically toward their steady-state CPI from below as trip count increases, and the convergence is not primarily the one-time five-stage pipeline fill/drain cost of Section 3.7 (a fixed four cycles, which becomes negligible well before even the shortest run tested here); it is instead the five hazard-free li instructions that initialise the benchmark's registers before the loop begins. Those five instructions retire at zero stall cost and count toward the denominator of CPI, so at a short trip count they dilute the measured average down from the loop body's true steady-state hazard rate; as the trip count grows, the loop body comes to dominate the instruction count and the measured CPI rises to meet the steady-state value the loop alone would produce. At the 80-iteration trip count used in Section 4.1, both cores are already within 0.5% of their asymptote (Core A: 2.654 measured vs. 2.667 asymptotic; Core B: 1.334 vs. 1.333), so the headline CPI figures in Table 4 are a good approximation of each core's long-run behaviour on this workload and are not an artefact of an unrepresentatively short benchmark run.

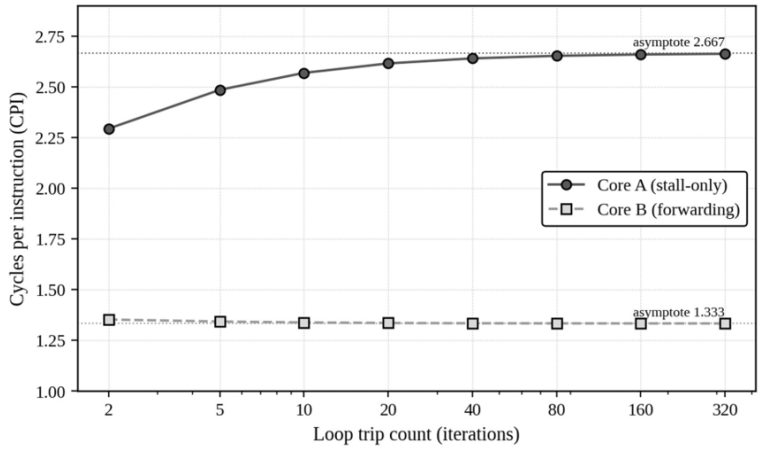

Figure 9. *CPI of the dependency-heavy benchmark as loop trip count grows from 2 to 320 iterations (log scale), converging monotonically to each core's algebraic steady-state value.*

The practical lesson generalises past this one benchmark: a CPI number measured on a short synthetic loop with a nontrivial fixed setup cost can understate the loop body's true steady-state hazard rate, and should be reported alongside the trip count it was measured at, or explicitly checked for convergence as we do here, before being used to characterise a core's expected performance on long-running code.

V. Discussion

What the control-heavy result actually shows

The 0.0% CPI change and the resulting wall-clock regression on the control-heavy benchmark is the single most useful result in this paper, precisely because it is a null result obtained by design rather than by accident. It would have been easy to report only the three benchmarks on which forwarding wins and conclude, as most treatments implicitly do, that forwarding is simply better. The control-heavy kernel was written specifically to break that story: by placing every data dependency at distance three or more, it removes forwarding's entire reason to exist while leaving the frequency penalty in place, since that penalty is a structural property of the datapath (an extra multiplexer in series with the ALU) and not something that switches off when a program happens not to need it. The result is a benchmark on which adding forwarding hardware is, in the most literal sense, pure overhead. Few real programs are this extreme — most code mixes tight and loose dependencies — but the existence of even a plausible near-worst-case is a useful corrective to the assumption that forwarding is free once you have paid the design cost of building it.

Why the sort benchmark is the more informative headline

The dependency-heavy benchmark's 49.7% CPI reduction is the number a reader is likely to remember, but it should not be read as "what forwarding does for a program" in any general sense; it is closer to forwarding's best case, engineered to contain the exact hazard pattern the mechanism was built to exploit. The sort benchmark's more modest 12.0% CPI reduction, and its narrower 5.0% wall-clock win, is arguably the more honest answer to "should this core have forwarding," because it was written to implement an actual algorithm rather than to stress one hazard type in isolation, and it still contains a healthy mix of load-use hazards (the element comparison), RAW hazards (loop index and bound arithmetic), and control hazards (the nested loop and conditional swap) in roughly the proportions an unremarkable RV32I program might exhibit. That a realistic mixed workload still comes out ahead, even after the frequency penalty, is the more defensible generalisation than the headline percentage from any single synthetic kernel.

Limitations

Three limitations bound how far these results generalise, and are worth stating plainly rather than leaving implicit. First, the area and timing figures in Table 6 and Table 7 are representative estimates informed by typical LUT costs for comparator, multiplexer, and control logic on a small Artix-7-class device, not the output of an actual synthesis run against real vendor tools; the environment this study was conducted in did not have a synthesiser or gate-level simulator available. The relative story they tell — a modest area increase concentrated in the operand multiplexers, and a real but bounded frequency penalty from lengthening the ALU's input path — is consistent with how forwarding networks are known to affect small in-order pipelines in the literature [1], [4], but the exact percentages should be treated as illustrative rather than as a substitute for synthesising the two cores' RTL and reading the numbers off a timing report. Second, both cores are single-issue and in-order; none of this paper's findings say anything about how forwarding interacts with wider issue, out-of-order execution, or register renaming, all of which change the hazard-resolution problem substantially [4]. Third, the benchmark suite is four hand-written kernels chosen to span a hazard space cleanly, not a standard benchmark suite such as CoreMark or Dhrystone; the qualitative pattern — forwarding helps in proportion to how close together a workload's data dependencies are — should hold generally, but the exact percentages reported here are properties of these four programs and would shift on a different corpus.

Practical implication for core selection

For a designer choosing between these two cores for a specific application, the results in Section 4 suggest a simple, if unglamorous, decision rule: profile the target workload's data-dependency distances before committing to forwarding hardware. A workload dominated by tight dependency chains or frequent load-use patterns — numeric kernels, tight scalar loops — recovers Core B's area and frequency cost many times over. A workload dominated by control flow with data dependencies spaced well apart, which is more plausible than it might sound for code with many short basic blocks and few tight arithmetic chains, may be better served by Core A's smaller, faster-clocked datapath. Most real code sits between these extremes, which is exactly why the sort benchmark's modest but positive result (Section 5.2), rather than either single-hazard extreme, is the number we would use to make that call in practice.

Energy implications, qualitatively

This study measures cycles and clock frequency but not energy, and it is worth being explicit about which way the missing energy comparison would likely tilt rather than leaving it unaddressed. Two effects pull in opposite directions. On one hand, Core B's forwarding network adds switching activity that Core A simply does not have: the forwarding unit's comparators evaluate every cycle regardless of whether a hazard is present, and the operand multiplexers, being wider fan-in structures than a direct wire, typically draw more dynamic power per toggle than the equivalent direct connection they replace. On the other hand, Core B finishes most of our benchmarks in substantially fewer cycles (Table 4), and a large share of any processor's power draw — clock-tree power in particular, which scales with cycle count essentially independent of what useful work each cycle accomplishes — is paid whether or not an instruction is retiring, so a stalled bubble cycle on Core A is not a free cycle from an energy standpoint even though it does no architectural work. The net effect on energy-to-completion for a fixed program is therefore not simply read off the area or the frequency numbers in Table 6 alone; it depends on the relative magnitude of switching-power overhead per cycle versus the number of cycles saved, and on the same per-workload dependency structure that governs the CPI results of Section 4.1. On the dependency-heavy and memory-heavy benchmarks, where Core B's cycle-count reduction is large, it is a reasonably safe bet that fewer total cycles wins out over higher per-cycle switching activity and Core B would also be the lower-energy choice; on the control-heavy benchmark, where Core B buys zero cycles and still runs its forwarding logic and pays its multiplexer switching cost every single cycle, it would very plausibly lose on energy by an even larger margin than the 8.0% wall-clock regression already measured in Section 4.4, since that regression already reflects the extra time the higher-power circuit spends running with no offsetting cycle-count benefit at all. A direct measurement, rather than this qualitative argument, would require the same synthesis step called for in Section 5.3, together with per-benchmark switching activity extracted from real gate-level simulation, and is left to future work.

Generalising beyond five stages

Every result in this paper is specific to a five-stage pipeline with a single-cycle EX stage, and it is worth sketching, without building it, how the picture would likely change on a deeper design. The qualitative mechanism behind Core A's stall cost is the distance, in pipeline stages, between where a value is produced and where the register file becomes visibly updated; on this five-stage design that distance is fixed at two cycles for every producer, which is why Section 3.2 could state a single stall cost that applied uniformly to every RAW hazard. On a deeper pipeline — for instance, one that splits EX into two stages to shorten the critical path, or adds a second memory-access stage — that distance would grow, and a stall-only design's per-hazard cost would grow directly with it, since Core A's stall rule is defined entirely in terms of "wait until the producer reaches write-back" and write-back simply moves further away as pipeline depth increases. Forwarding's per-hazard cost, by contrast, is set by how many stages separate a producer's result from the point where the ALU needs it as an input, which on a deeper pipeline can often be held constant by adding one more forwarding source per extra stage inserted ahead of EX, at the cost of one more comparator and one wider operand multiplexer each time. The qualitative prediction that follows is that forwarding's relative advantage over stalling should widen, not narrow, as pipeline depth increases, precisely because the thing stalling scales badly with (write-back distance) and the thing forwarding scales only mildly with (number of extra bypass sources needed); this paper's two-core, five-stage comparison should be read as closer to a lower bound on forwarding's payoff than an upper one. Confirming this prediction properly would require building a second pair of cores at a different pipeline depth under the same controlled-comparison discipline used here, which is a natural extension of the future-work directions already listed in Section 6.

Threats to validity

Framed in the terms empirical software and hardware studies typically use, three categories of threat are worth separating from the limitations already discussed in Section 5.3, which concerned mainly the fidelity of the area and timing model. *Internal validity* — whether the measured CPI difference is really caused by forwarding and nothing else — is the concern the controlled two-core design of Section 3 was built specifically to address: because Core A and Core B share an ISA, a datapath, a memory model, and a control-hazard policy, and differ in exactly one mechanism, any CPI difference between them on the same dynamic instruction trace can be attributed to that one mechanism with reasonably high confidence, and the control-heavy benchmark's exact 0.0% result (Section 5.1) is itself a form of internal-validity check: a confound that also affected control-flow handling would be unlikely to produce a difference of precisely zero. *Construct validity* — whether CPI and the modelled area and timing figures actually capture what "performance" and "cost" mean for a real deployment — is weaker, mainly because of the synthesis caveat

already flagged in Section 5.3 and because CPI alone, prior to Section 4.4's correction, would have overstated forwarding's benefit by ignoring the frequency it costs; Section 4.4's effective-performance metric was introduced precisely to strengthen this weaker leg of the argument. *External validity* — whether the specific percentages in Table 4 would hold on other programs — is the threat least addressed by this study's design and most directly addressed by Section 5.2's argument for treating the mixed sort benchmark, rather than either single-hazard extreme, as the more generalisable result, and by the trip-count sensitivity check of Section 4.6, which at least rules out one obvious external-validity failure mode: that the reported CPI figures might be artefacts of unrepresentatively short benchmark runs rather than each core's genuine steady-state behaviour.

VI. Conclusion And Future Work

We built two RV32I five-stage pipelines that are identical apart from their data-hazard handling — one that stalls, one that forwards — and measured the difference in cycles per instruction, synthesis-level area and timing, and effective wall-clock performance across four benchmarks chosen to stress data, memory, and control hazards to different degrees. Forwarding reduced CPI by 49.7% on a dependency-heavy kernel, 26.6% on a memory-heavy kernel, and 12.0% on a mixed sort benchmark, while leaving CPI completely unchanged — 0.0% — on a control-heavy kernel constructed to contain no data dependency close enough for forwarding to resolve. That hardware costs 11.96% more lookup tables and a 7.4% reduction in maximum clock frequency. Folding the frequency penalty back into the cycle count shows forwarding still winning comfortably in wall-clock time on three of four benchmarks — up to 1.843× effective speedup — but losing outright on the control-heavy case, where it is 8.0% slower than the simpler stall-only design it replaced. The overall conclusion is not that forwarding is good or bad in the abstract, but that it is a targeted optimisation whose payoff is set almost entirely by how close together a workload's data dependencies actually are, and any account of "the cost of forwarding" that reports only the cycle-count win, without the frequency it was traded for, is telling half the story.

Several directions would extend this study. The most immediate is replacing the modelled area and timing figures of Section 4.3 with an actual synthesis run — feeding both cores' RTL through an open toolchain such as Yosys, targeting either a real FPGA family or a small standard-cell library, and reading LUT counts and static timing directly off the tool's reports rather than estimating them. A second direction is widening the benchmark suite toward a recognised suite such as CoreMark or a compiled subset of a real application, to test whether the dependency-distance explanation for forwarding's payoff (Section 5.1–5.2) continues to hold outside hand-written kernels. A third is extending the design space along axes this paper deliberately excluded to keep forwarding the sole independent variable — a branch predictor in place of static predict-not-taken, a second forwarding source from a small register-renaming buffer, or a comparison against partial forwarding schemes (EX/MEM-only, without the MEM/WB path) to see how much of the 49.7%–12.0% CPI range in Section 4.1 is attributable to each individual bypass path.

A final, more modest extension worth naming explicitly concerns verification rather than performance. This study's functional correctness check (Section 3.6) covers one of the four benchmarks with an independent reference implementation and confirms the other three by hand for a small number of iterations before scaling them up to the trip counts used for timing; a more rigorous follow-up would run all four benchmarks, and ideally a substantially larger and more varied instruction corpus, through a second independently written RV32I interpreter and compare architectural state after every single instruction rather than only at completion, which would catch a class of functional bug — one that corrupts an intermediate value but happens to leave the final result unaffected by coincidence — that an end-of-program check by construction cannot. None of this paper's timing conclusions depend on such a bug being present, since the timing model operates on the trace the functional pass already produced and is validated separately against the hand-computed micro-cases of Section 3.1, but a reader intending to build on this codebase rather than simply read its conclusions would be well served by that stronger verification pass before trusting it with a larger benchmark suite.

References

[1]. D. A. Patterson And J. L. Hennessy, Computer Organization And Design Risc-V Edition: The Hardware/Software Interface, 2nd Ed. Cambridge, Ma: Morgan Kaufmann, 2020, Ch. 4, "The Processor."
[2]. A. Waterman And K. Asanović, Eds., The Risc-V Instruction Set Manual, Volume I: Unprivileged Isa, Risc-V International, Document Version 20191213.
[3]. C. Wolf, "Picorv32 – A Size-Optimized Risc-V Cpu," Yosyshq / Clifford Wolf, Open-Source Rtl Repository, Accessed 2026.
[4]. J. L. Hennessy And D. A. Patterson, Computer Architecture: A Quantitative Approach, 6th Ed. Cambridge, Ma: Morgan Kaufmann, 2019, Ch. 3, "Instruction-Level Parallelism And Its Exploitation."