

GB3 RISC-V Processor – Final Report

Liu Zihe (Zach) | zl559 | Group 6

Abstract

This report describes the design, implementation and evaluation of microarchitectural optimisations to the `picorv32` RISC-V processor core, deployed on the Lattice iCE40UP5K FPGA of the iCEBreaker board. The project is scoped to hardware-only changes, with all firmware compiled with `-march=rv32im`.

The baseline `picorv32` FSM (Fig. 8, Appendix B) executes most instructions in 3–5 cycles, yet profiling measures ~ 70 CPI (Table 2, Appendix B): the gap is almost entirely stall on instruction fetch from SPI flash. Amdahl's law guided the optimisation order: address instruction-fetch latency first, and only then core CPI and clock period. Three principal optimisations were implemented. First, an EBR-backed, parameterized instruction cache (currently 2-way set-associative, 1024 sets, 8 KB) was inserted between the CPU and SPI flash. Second, the SPI flash controller's reset defaults were upgraded from single-bit SPI to Quad-DDR fast read. Third, the system clock was raised from the 12 MHz crystal to 22.875 MHz via the on-chip PLL. Configuration-level decisions complement these.

Correctness and performance were verified at every step with an internal benchmark suite of microbenchmarks and application programs. The combined optimisations reduced $\text{CPI} \geq 10\times$ from ~ 70 to ~ 6 , and roughly $20\times$ in wall-clock time against the baseline. This report details my individual contributions to this team effort, the problems encountered and their technical resolutions, and concludes with my personal reflections.

Please refer to my [personal notes](#) for a technical deep dive of the `picorv32` microarchitecture, and summary microarchitecture concepts from [1] and [2].

1 Design Decisions and Project Management

Our task was to implement the PicoRV32 RISC-V processor on an iCE40UP5K FPGA board, optimising for performance, power, and area against the baseline PicoSoC implementation. To improve performance, we started by trying several methods to decrease execution time $T_{exec} = N \times CPI \times T_c$, where N is number of instructions, CPI is the average number of cycles needed to execute each instruction, and T_c is the clock time period. N could not be optimised since we were using a fixed, uncompressed ISA. We used the following methods to improve CPI and T_c .

1.1 Instruction Cache

From our benchmarking tests, we discovered that the instruction fetch from the flash cost ~ 128 cycles when using the default single SPI. For this, we implemented an instruction cache that sits between the SPI flash and the processor and caches recently fetched instructions to avoid repeated SPI fetches. We made our cache parameterised — switchable between number of sets, ways, and words per line — letting us sweep across parameters and keep the best performing configuration. We were able to reduce CPI for sequential code to ~ 5 with the cache implementation.

We built our instruction cache in the EBR (4Kb dual-port RAM blocks) since 26 of the 30 blocks were unused in the baseline implementation. A naive hit costs 2 cycles because the EBR read is synchronous; we cut this to 1 cycle by pre-reading the set off PicoRV32’s look-ahead interface (Sec. 2.1.2). On a miss, the request passes through to flash, the returned word is cached, then the CPU receives it. For the benchmark competition, we set the parameters to a 2-way set-associative cache with 1024 sets and 1 word per line, as we found this to show the lowest CPI. Our primary constraint was the number of available logic cells (LCs), which limited how large we could make the cache. Our final configuration used 27/30 EBR blocks and 4591/5280 LCs.

1.2 QDDR SPI

Next, we wanted to reduce the cold fetch CPI. To achieve this we enabled Quad SPI with double data rate (QDDR) directly on the hardware, so that spimemio shifts 4 bits per SPI clock, and double data rate clocks data on both the rising and falling edges of the SPI clock. This ensured our SPI optimisations were applied regardless of whether the flag was explicitly switched on in the C code, which allowed us to still perform well on benchmarks where it was turned off. With QDDR, we were able to get our cold fetch CPI from 64.03 down to 8.02.

1.3 Clock Frequency

To reduce T_c , we used the iCE40’s Phase Locked Loop to multiply the board’s 12 MHz crystal up to a faster system clock. The maximum clock frequency is determined by the critical (slowest) path through the design. Implementing the instruction cache increased our maximum frequency from around 16 MHz to 18.7 MHz, which was a substantial benefit in addition to the reduced CPI. Further improvements were achieved by disabling the unused IRQ logic, switching to the noabc9 synthesis strategy in the Yosys build, and running a seed sweep with `nextpnr --seed`, which bumped the clock frequency to 22.875 MHz. We also tried toggling a few more parameters like `BARREL_SHIFTER` and `TWO_CYCLE_ALU`, but found negligible shortening of the critical path and higher max clock frequency, so we did not implement them.

1.4 Verification

We performed functional verification of the instruction cache using simulations across a wide range of inputs and scenarios to show, beyond reasonable doubt, that the design was functionally correct. We defined functional coverage metrics to divide the search space into regions of high interest, and developed tests to achieve 100% coverage on these metrics. We used Verilator for simulation, Cocotb for testbench development and PyVSC for coverage tracking.

1.5 Project Management

The actual division of labour and execution of subgoals mostly followed the proposed plan, with the exception of pipelining:

Subgoal	Owner	Description
Baseline config changes	All	Baseline profiling and config settings: COMPRESSED_ISA=0, ENABLE_DIV=1, ENABLE_FAST_MUL=1, ENABLE_IRQ=0
Cache RTL design	PL, Z	Parameterised SETS $\times$ WAYS $\times$ WORDS_PER_LINE cache module: EBR tag/data arrays with full-width set writes, parallel tag compare, and the 1-cycle look-ahead hit via <code>mem_1a_addr</code> . Cache-geometry sweep settling on 2-way $\times$ 1024 sets.
Cache integration	All	Wire cache into PicoSoC as an address-decoded responder, forwarding misses to <code>spimemio</code> . Unit testbench sweeping multiple cache paths (hit/miss, eviction, reset invalidation, look-ahead path)
QDDR SPI flash	Z	Hardware reset defaults for fast read (INIT_MODE: dual/quad/QDDR), including a reset-time FSM that sets the flash QE bit so QDDR holds for any firmware
Clock	PL, Z	Clock raised to 22.875 MHz via PLL retuning and seed sweeps against the critical path.
Secret benchmark	GL	Design the team’s benchmark program flooding other team’s instruction cache and firmware defaults
Final verification	GL	Coverage-driven functional verification (Verilator + Cocotb + PyVSC)
Final evaluation	All	Run all benchmarks, final measurements (performance, power, area), competition spreadsheet entry and presentation prep.

Table 1: Subtask breakdown. PL is Prishim, GL is Galen, Z is Zach.

2 Individual Contribution: Design, Code and Testing

2.1 Instruction Cache

All (binary machine) code is stored in SPI flash¹, so *every* instruction fetch must access the SPI bus. A full single-bit read costs ~ 128 system clocks². Sequential fetches are cheaper (~ 64 system clocks) but still dwarf the 3–5 cycles an instruction takes to execute — which is why the baseline measures ~ 70 CPI on internal benchmarks even on sequential instructions. Loops re-fetch the exact same instructions every iteration, so the easiest fix is to exploit this temporal locality. A small cache built from EBR memory completes a repeated fetch in 2 system cycles instead of > 64 , collapsing CPI from ~ 70 to ~ 6 (all CPI figures are measured on our internal benchmark suite, Sec. 2.4).

Memory interface. The CPU requests a memory transaction by asserting `mem_valid` with an address, and the addressed device completes it by raising `mem_ready` (with `mem_rdata` for reads). Internally this is layered (Figure 1): the CPU main FSM never drives the bus itself, but raises request flags (`mem_do_rinst/rdata/wdata/prefetch`) to a small *bus FSM* — the sole driver of the external wires — which reports completion back as `mem_done`.

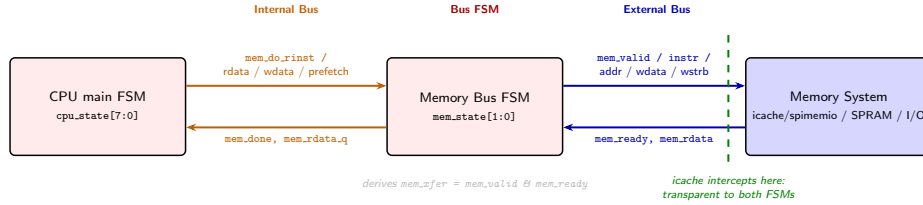


Figure 1: The three layers of the picorv32 memory subsystem, left-to-right from core to memory: the **internal bus** (orange, CPU FSM $\leftrightarrow$ bus FSM request handshake), the **bus FSM** (red, sole driver of the external bus), and the **external bus** (blue, fanned out by PicoSoC’s address decoder). The cache attaches at the external layer only (green), invisible to the two FSMs.

Cache placement. The cache is added as one more device on the flash range ($0x00020000 \leq \text{addr} < 0x02000000$), shown in Figure 10 (Appendix B). Cache hits answer on the external `mem_*` bus directly³, while cache misses fall through to the `spimemio` flash controller over the `spi_*` channel. Attaching the cache at the external layer means neither the CPU FSM nor the bus FSM is touched, and no core RTL changes are needed.

2.1.1 Structure and Operation

Structure. The cache module is fully parameterized — $\text{SETS} \times \text{WAYS} \times \text{WORDS_PER_LINE}$, with $\text{WAYS} = 1$ giving a direct-mapped cache and $\text{WAYS} > 1$ a set-associative one. The shipped configuration is **2-way $\times$ 1024 sets $\times$ 1 word = 8 KB**⁴. From [2], each 24-bit flash address splits into tag/index/word/byte fields (Figure 2, left). The index bits select a **set** (a row of the cache); within the set the line may sit in any of the **WAYS ways**, each way storing its **tag** so the hardware can compare this to the requested address tag and determine if a cache hit occurs. We use $\text{WAYS} = 2$ as this helps cache misses due to conflicts: in a direct-mapped cache, two hot instruction words sharing index bits evict each other on every alternate fetch, causing a high miss rate. A second way lets them both be stored in cache (Figure 2), but beyond 2–4 ways we measure the returns diminishing on our internal benchmarks. WORDS_PER_LINE stays at 1: a wider line exploits spatial-locality, but we find in practice this doesn’t improve performance even with benchmarks that require many cold fetches.

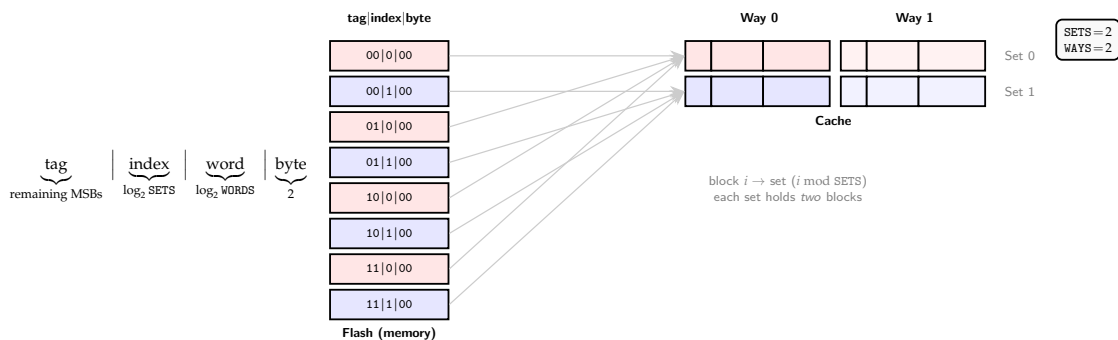


Figure 2: Left: the parametric 24-bit flash-address split. Right: 2-way set-associative mapping, scaled down to $\text{SETS} = 2$ for illustration (the shipped cache is the same organisation but with $\text{SETS} = 1024$), and with no word fields since we have 1 word per line. The index bits pick the set; the block may occupy either way, so two hot addresses sharing index bits (e.g. blocks 0 and 4) coexist instead of conflicting. Each way line stores {valid, tag, data} respectively.

Default Operation. Figure 3 shows the three cache states. Without look-ahead (§2.1.2), the default path with timing shown in (Figure 5a) is:

¹The 128 KB SPRAM holds data like heap/stack/global variables only and is not preloadable

²8 command + 24 address + 32 data = 64 SPI clock cycles ≈ 128 system cycles with the 2:1 SCK (SPI serial clock) divider.

³A note on naming: the bus wires carry picorv32’s `mem_*` names (`mem_valid/mem_ready/mem_rdata/mem_la_*`), while the cache’s own ports are named `cpu_*`. These are equivalent so from here on we write `cpu_*` throughout. Only `mem_instr/mem_wstrb` keep their bus names: they exist on the bus side only (the cache decodes purely by address).

⁴The shipped single-word build composes: tag [23:12] | index [11:2] | byte [1:0], no word field needed with 1 word per line

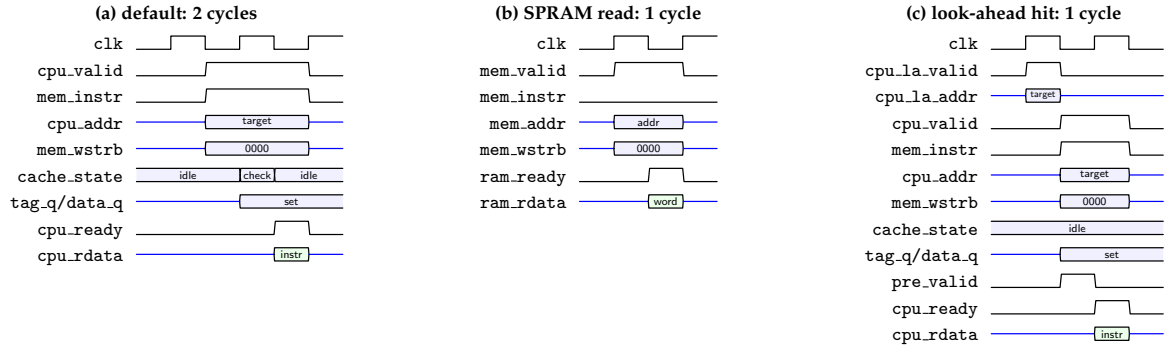


Figure 5: Cache hit timing; latency is counted from the rising edge of `cpu_valid` to that of `cpu_ready`. (a) The default 2-cycle hit — `cache_state` shows where each cycle goes: one in `idle` (the EBR read) and one in `check` (compare + registered ack) — one cycle slower than (b), SPRAM’s registered 1-cycle data ack (a plain bus access that never enters the cache, hence the `mem.*` names). (c) With the look-ahead, `cpu_la_valid/cpu_la_addr` stage the EBR read one cycle *before* `cpu_valid` (`pre_valid` marks the staged set), so the registered ack lands one cycle after the request — the FSM never leaves `idle`, and fetch now matches data-memory latency. `tag_q/data_q` is the EBR read-out: the set lands a cycle *after* the request in (a), but *together with it* in (c).

1. *Request cycle* (`idle`): `cpu_valid` rises; the cache latches the address fields into `cpu_addr` and searches for the index in the EBR. At the next rising clock edge the addressed set — all `WAYS` lines {`valid`, `tag`, `data`} — is captured into the EBR output registers `tag_q/data_q`.
2. *Compare cycle* (`check`): the requested tag is compared against all way tags stored in `tag_q`, in parallel. A hit selects the requested word from `data_q` and registers the acknowledge next rising edge. A miss enters `fill` state, which fetches the line word-by-word from `spimemio`, writes the rebuilt set back, and only then acknowledges.
3. *Acknowledge cycle*: `cpu_ready/cpu_rdata` are up — two cycles after the request: a 2-cycle hit.

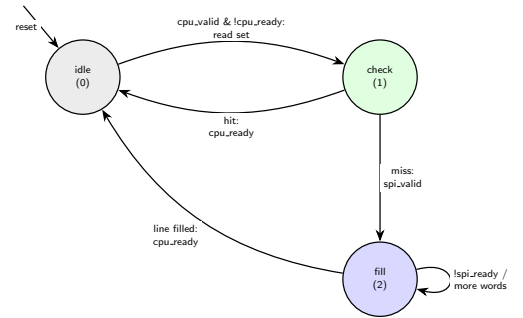


Figure 3: The cache FSM, default path (no lookaheads)

2.1.2 Look-Ahead Hit

The default cache hit takes two cycles: `idle` spends a full cycle just starting the EBR cache read, and only then does `check` compare and raises the registered `cpu_ready` a cycle later (Figure 5a). With lookaheads, we can bring cache hits down to 1 cycle.⁵

Pre-reading off the look-aheads. `picorv32` announces every fetch one cycle early: in the cycle before it raises `cpu_valid`, it raises its look-ahead bus (`mem_la_read`, with the target address on `mem_la_addr`). We exploit this early information and route these through `picosoc.v` into the cache as `cpu_la_valid/cpu_la_addr` (top rows of Figure 5c) and use them to start the EBR read one cycle early. Tracing Figure 5c left to right:

1. *Look-ahead cycle*: `cpu_la_valid` on: search for this indexed set in the EBR, and at the next rising clock edge capture the addressed set into `tag_q/data_q`, flagging it staged (`pre_valid`).
2. *Request cycle*: `cpu_valid` rises with the same `cpu_addr`. The corresponding requested set is already sitting on the EBR output. `idle` compares the request tag against all way tags held in `tag_q`, in parallel. A hit registers the acknowledge at the closing edge; a miss launches the `fill` from SPI directly. `check` is never entered (Figure 4).
3. *Acknowledge cycle*: `cpu_ready/cpu_rdata` are up, one cycle after the request.

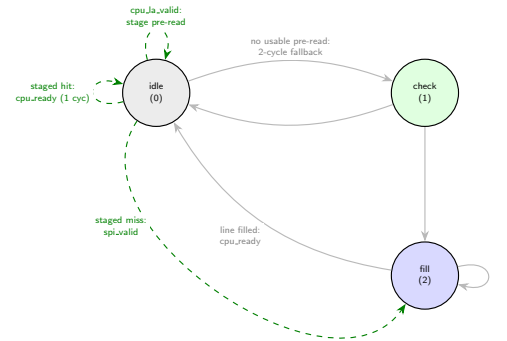


Figure 4: The look-ahead fast path (green dashed) extending the default FSM of Figure 3 (grey): a staged pre-read lets `idle` serve a hit in one cycle, or launch the `fill` directly — `check` is skipped entirely.

The EBR read still costs one cycle; the look-ahead merely pays it *before* the CPU starts waiting. Compare `tag_q/data_q` across panels: in (a) the set arrives a cycle *after* the request, in (c) simultaneously with request `cpu_valid`. On our internal benchmarks, with lookahead cache hit most CPI decrease ~ 1 cycle per instruction: e.g. `alu` 6.89 $\rightarrow$ 5.86 CPI and `hot` 6.70 $\rightarrow$ 5.75 CPI.

2.1.3 Verification

A testbench (`icache.tb.v`) drives the cache against a simulated golden flash model: every word returned for an address must be as in the golden model. The model mirrors `spimemio`’s real handshake. Tests cover edge cases like post-reset invalidation bits, eviction under colliding tags, random stress, and the look-ahead path. One run sweeps five cache geometries at varied flash latencies.

⁵A combinational acknowledge (`cpu_ready = (state==check) && hit`) would also give 1 cycle, but would affect the critical path (Sec. 2.3) hence lowering $f_{\max}$.

2.2 SPI Flash Fast-Read Modes

The `spimemio` module serves every instruction fetch. A read is a fixed sequence of phases — command, 24-bit address, mode + dummy bits, then data — driven by the FSM of Figure 6. Each byte is shifted out at rate $SCK = \text{half the system clock}$. The first byte of every transaction is the flash chip’s read opcode (the command). This tells the flash how the address, data etc will be transferred — `0x03` single-bit (1 bit per SCK), `0xBB` dual I/O (2 bits across more pins), `0xEB` quad I/O (4 bits per SCK), or `0xED` quad-DDR (4 pins clocked on both SCK edges — hence 8 bits transmitted per SCK). The controller also streams sequential reads: it keeps the command constant and advances word by word, so a sequential fetch costs only its data phase. Any *non-sequential* address raises *jump*, aborting the burst and re-paying the full command + address + dummy latency. *This is the source of the long fetch stall of SPI instruction fetches*. Figure 7 gives the per-word cost: ~ 64 SPI clocks single-bit, ~ 28 quad, ~ 23 qddr. Any sustained sequential streaming pays only the data phase, ~ 4 clocks per word in qddr.⁶

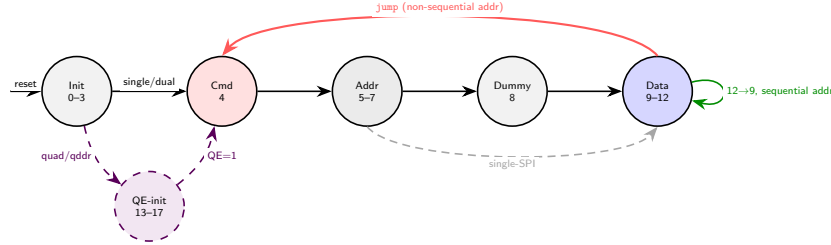


Figure 6: `spimemio` read FSM, grouped by phase (circled numbers are the RTL state numbers). States 0–3 run once after reset; each read then walks `Cmd` → `Addr` → `Dummy` → `Data`, and single-bit SPI skips the mode/dummy state (dashed). **Green:** sequential streaming — the Data phase loops (12 → 9), advancing the read address by 4 and keeping the flash command constant, so consecutive words cost only their data phase. **Red:** a jump (non-sequential fetch) aborts the burst and re-issues the command and address. **Violet dashed:** the QE-init detour we added — in the quad modes, states 13–17 set the flash’s QE bit before the first read command.

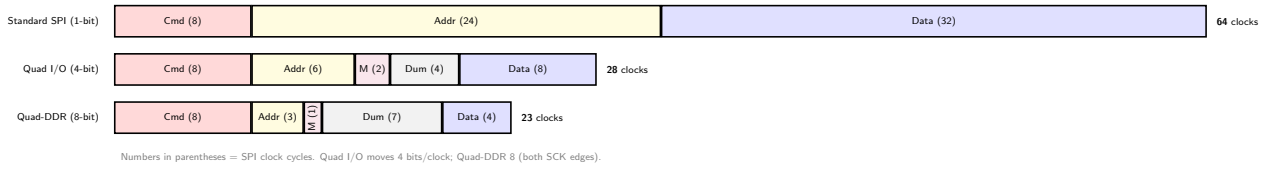


Figure 7: SPI flash read transaction overhead for one 32-bit word. Standard SPI transmits 1 bit per clock: 8 (command) + 24 (address) + 32 (data) = 64 SPI clocks (the `0x03` read has no dummy phase). Quad I/O sends the command single-lane (8 clocks) but transfers address and data 4 bits per clock, reducing the total to ~ 28 SPI clocks; Quad-DDR halves the address and data phases again, ~ 23 clocks. The `spimemio` controller keeps the flash command open and streams sequential words, so sustained sequential throughput drops to the data phase only — ~ 4 SPI clocks per word in qddr.

On reset the controller comes up in the slowest single-bit mode and never leaves it unless firmware writes its configuration register (`cfgreg` after boot). Hence we change the hardware reset default for the flash read mode to qddr mode, using a 2-bit `INIT_MODE` parameter that decodes at reset into the relevant `cfgreg` mode bits. Enabling the quad modes need the quad-enable (QE) bit inside the flash chip to be set⁷, hence we modify `spimemio`’s reset init chain to have an extra five states (13–17, violet in Figure 6) that write the QE bit over single-bit SPI at reset.

2.3 Clock Frequency

2.3.1 PLL Retuning

The board supplies a raw 12 MHz crystal; the iCE40’s hard Phase Locked Loop (PLL) (`SB_PLL40_PAD`) multiplies it up to the system clock:

$$f_{\text{out}} = f_{\text{ref}} \times \frac{\text{DIVF} + 1}{(\text{DIVR} + 1) \times 2^{\text{DIVQ}}} = 12 \times \frac{61}{1 \times 2^5} = 22.875 \text{ MHz}$$

The divider values come from Lattice’s `icep11` tool, which honours the physical constraints⁸. The PLL scales *everything*, including fetch-bound code: the flash clock $SCK = c1k/2$ rises in lockstep, so a fetch costs the same number of cycles but less wall-clock time. We increase the frequency until it reaches the limit set by the critical path.

2.3.2 Critical Path and f_{max}

The bottleneck for max frequency f_{max} is the logic critical path. `icetime` traces it to a single chain (~ 54 ns): a front half (~ 21 ns) involving `mem_addr` and `mem_ready`, and a back half (~ 33 ns) involving `mem_done`. Any logic placed in front of `mem_ready` is therefore added in series to this whole chain, while a register there cuts it into two shorter flip-flop-to-flip-flop paths. Hence memory acknowledges must stay registered, which was why the combinational cache hit was rejected, Sec. 2.1.2. Registering `mem_ready` completely increased f_{max} to ~ 24 MHz but increased CPI $\sim 1.2\times$. Synthesising with `-noabc9` and sweeping `nextpnr` seeds lift f_{max} further. Going beyond 22.875 MHz requires shortening the chain even further, likely pipelining, which would modify the chip core. This is significantly more risky hence was not implemented.

2.4 Internal Benchmark Suite

As a proxy for the unknown secret benchmarks, we wrote a two-layer suite (`benchmarks.c`): microbenchmarks that each isolate one operation class (ALU, shift, mul, div, branch, call, load/store, fetch) and application programs spanning realistic mixes of those operations (sorting, matmul, string search, ...). We wrap each operation and program with `rdcycle/rdinstret` and measure cycles, instructions, CPI. We simulate fetch-heavy programs using `hot` (tiny loops) vs `cold` (straight-line body larger than the cache size) which helped us find the best cache parameters. All CPI figures quoted in this report come from this suite.

⁶ ~ 128 , ~ 56 , ~ 46 and ~ 8 system cycles respectively, at the 2:1 SCK divider.

⁷ setting it from firmware is too late for a reset default, since the reset-vector fetch precedes any instruction

⁸ VCO within 533–1066 MHz, output ≥ 16 MHz

3 Problems Encountered and Technical Solutions

3.1 Cache Hardware Issues

The cache module initially synthesized and flashed to the board, but booting the chip had some serious issues.

Undefined EBR power-up. After FPGA flashing, EBR contents are undefined, while the cache's valid bits are synthesized to the EBR. Then a random power-up pattern can present as a valid entry whose tag happens to match, and the cache returns garbage as an "instruction". Simulation never showed this: an uninitialised Verilog array reads X, so the tag compare is *accidentally* false and every first access gracefully misses. The fix is an invalidation sweep: a post-reset state that walks all SETS rows to zero the valid bits before the first lookup.

Chip Freeze. The board froze instantly when a key was pressed during the all-LEDs-on phase in the flashing boot phase, yet a keypress during the all-off phase had normal behaviour. Since the keypress write to `reg_leds` flipped all ~15 driven pins `0xFF` → `0x00` at once, we speculate that the simultaneous switching of all pins corrupted a flash word, perhaps due to power supply issues. Without a cache this problem is fine: perhaps the corrupted word is simply re-fetched on the next loop. With the cache, maybe the corrupt word is stored under a valid tag and replayed on every later hit. Then the CPU repeatedly executes a corrupt instruction and therefore hangs. Making a more stable power supply with a powered USB hub changed nothing. I was unable to reproduce this behaviour in the simulation, yet the hardware kept failing, so I simply restricted the firmware to drive *one* LED at a time, never the all-on word. The problem was fixed, but to this day I'm still not sure the root cause of the issue.

3.2 Sim-Real Misalignment

Many issues that ran perfectly fine in simulation had problems when actually running on the chip, as some chip behaviour wasn't made explicit in the Verilog software.

Dummy cycles in the flash model. The default `spiflash.v` simulation model hardcodes 8 dummy cycles for every fast-read command, but the real W25Q128JV SPI flash requires 0 cycles for dual `0xBB` (the mode byte itself is the turnaround), 4 for quad `0xEB` and 7 for qddr `0xED`. Any mismatch makes the SPI controller latch misaligned garbage, hence we needed to change the default dummy cycles in `spiflash.v`. The fix uses different dummy counts matching `config_dummy`, and instruction data could be fetched correctly.

QE bit. The flash model ignores the QE bit entirely and completes quad commands regardless. Hence setting the QE bit correctly can only be validated on the board, via the flash status-register read.

Yosys PLL simulation stub. Yosys's simulation model of `SB_PLL40_PAD` was an empty placeholder stub, and drives neither `PLLOUTCORE` nor `LOCK`, so after adding the PLL the simulation system clock never toggled. In simulation we instead drive the system clock directly from the testbench clock, bypassing the PLL.

3.3 Race Conditions

The cache-`spimemio` boundary had a ton of handshake races, all of which we needed to address carefully. Many problems were because we did not distinguish between a signal held high, and single cycle pulse signal.

Fill-streaming handshake. `spimemio`'s `ready` signal is a constant signal held high while the word it currently holds matches the requested address. Initially our first fill loop activated on its rising edge, which caused race conditions: when `spimemio` advances its internal read address in the same cycle that the cache advances `spi_addr`, `ready` never falls. Hence the awaited rising edge never comes, and a word of the line is skipped. The fix captures on the high level itself and advances `spi_addr` in the same cycle a word is captured, so each word is taken exactly once.

Phantom re-latch. The cycle after a hit, `cpu_valid` is still high while `cpu_ready` falls, so `idle` state would re-accept the old request as a new request and serve it again. A single `!cpu_ready` term in the accept condition drops that incorrect logic.

Stale look-ahead addresses. The staged pre-read signals (Sec. 2.1.2) can contain a different address than the request that follows it. Comparing against the staged tag would false-hit; instead the compare always uses the CPU's actual request tag, so a wrong pre-read can only downgrade to a miss, never return wrong data.

4 Conclusions, Reflections and Lessons Learned

4.1 Results

Table 2 summarises the outcome: CPI fell from ~ 70 to ~ 6 ($\geq 10\times$) on every benchmark in the suite, with the PLL this reduces $16\text{--}28\times$ in wall-clock time. The instruction cache accounts for most of this CPI reduction — a repeated fetch costs 1–2 cycles instead of ≥ 64 , so any code with temporal locality (basically all code) runs at near core CPI. QDDR fast-read covers the weak spots of our cache: cold and non-sequential misses, shrinking the worst-case fetch from ~ 128 to ~ 46 system cycles and sequential streaming to ~ 8 . The PLL then scales everything that remains by $\sim 2\times$ in wall-clock.

Competition outcome. In the final cross-evaluation — nine secret benchmarks, one per team, each run on every team’s processor — our design achieved the *best overall normalised runtime of all eight groups* Table 3. The big wins were precisely the fetch-heavy benchmarks:

- **Secret 2** (> 5000 unique straight-line instructions executed once each, written explicitly to thrash any LRU cache) reduces to a pure sequential-streaming contest, which qddr’s data-phase throughput wins.
- **Secret 6** (~ 6 KB unrolled body that deliberately leaves the flash controller at its single-bit hardware reset default) ran $7\text{--}12\times$ faster on our chip than on most rivals
- **Secret 8** (a ~ 38 KB storm of short forward jumps, uncacheable by any group) rewarded our single-word lines: each miss fetches only the word needed, whereas a wide-line design fills a whole line and then jumps past it. One group with 16-word lines ran $20\times$ slower here.
- **secret 0’s** firmware forces every chip’s controller back to single-bit SPI, hence any fast-read default is useless, yet its small loop is cache-resident, so we still performed much better than baseline

The losses are equally informative:

- On every **cache-heavy benchmark** (secrets 0, 1, 3, 4, 5, 7) the winning design beat us by a consistent $\sim 1.4\text{--}1.7\times$. Once the hot loop fits in the cache, the contest reduces to $\text{CPI}_{\text{core}} \times T_c$, which we lost to other designs
- On **secret 5** — a benchmark tuned to fill the cache and punish direct-mapped designs — our 2-way associativity and qddr default were able to perform decently
- We ranked worst **power** (0.81 W, tied highest) likely due to extra look-ahead pre-reads, and the raised clock

4.2 Technical Reflections

Amdahl’s rules. The textbook optimisation for a multicycle core is to pipeline it. At ~ 70 CPI of fetch stall, that improvement would have been negligible. Cutting core CPI from 4 to 1 changes nothing while each instruction waits tens of cycles on flash. Hence we should always target the biggest bottlenecks first in $T_{\text{exec}} = N \times \text{CPI} \times T_c$.

Prioritize the critical path. Every shortcut that looked easy in isolation — the combinational cache hit, combinational acknowledges — fed the single longest flip-flop-to-flip-flop chain and would have increased the critical path and reduced f_{max} across the whole design. We should always design factoring in the critical path in mind.

Simulation and Hardware. The on-board failures of Sec. 3.2 were all because the simulation model was limited. A green simulation does not mean the design is flawless. In the future we should always note design decisions that cannot be captured by simulation alone.

4.3 Personal Reflections

I mean no exaggeration when I say this was the most useful module or thing I have done in the Cambridge Engineering Tripos so far. I came to Cambridge to do Computer Engineering, so the amount of computer architecture I picked up while poring through the Harris [2] and Patterson [1] textbook is enormous — see my [personal notes](#) for this project.

Reading the source code before modifying was honestly pretty painful - the repository was *huge*, and I spent a week tracing `picorv32`'s FSM and memory layers and pouring over the textbook. But this was really important for me to understand the biggest and easiest bottlenecks to solve, and also keep up and verify what AI was telling me to do. If not, I'll just be the blind following the blind.

Compared to the other project, this one really felt like a team effort. We often faced bugs we couldn't solve ourselves, but someone on the team understood the problem so we could patch each other's holes in code too. Verifying each others' code was helpful. My previous software engineering experience proved really helpful as we were often working on different versions of code, so I worked with the team a lot for rebase, merging etc.

4.4 Conclusion

This project took a fetch-bound multicycle core at ~ 70 CPI and, through an instruction cache, quad-DDR flash fast-read and PLL retuning, delivered a $\geq 10\times$ CPI reduction with hardware-only changes.

The highest value future work is pipelining: with fetch now cheap, core CPI (~ 4) is now the bottleneck, and restructuring the load/store path would simultaneously shorten the back half of the critical path (Sec. 2.3.2), unlocking higher PLL targets. However, there might not be enough logic-cell budget (pipelining would require ~ 1500 – 2000 LCs extra).

My biggest takeaways is not the $20\times$ performance improvements however, but rather the high level ideas of computer architecture, how to design and test hardware, and doing this all with parallel workflows in a team.

A AI Usage Declaration

We used AI in the following ways:

1. **Design:** AI was used to understand the source code, textbooks [2] chapters 6–8, write my personal technical deep dive notes, and suggest possible improvements. However final improvements chosen went through extensive team discussion and research before deciding on the optimization plan, design decisions, and division of labour.
2. **Implementation:** the specifications and high level design were human drafted. AI made the initial mockup of Verilog code, which of course ran into many bugs especially on the hardware side. The code was thereafter heavily modified by everyone on the team, and went through many rounds of iteration mostly by humans until our testbenches passed and the hardware was working.
3. **Writeup:** The prose and content was almost entirely handwritten by me except to correct formatting and technical accuracy with the source code.⁹ The diagrams, text alignment, and formatting was aided by AI, and improved on after discussion with teammates.

There is no material that is submitted verbatim.

B Supplementary Material

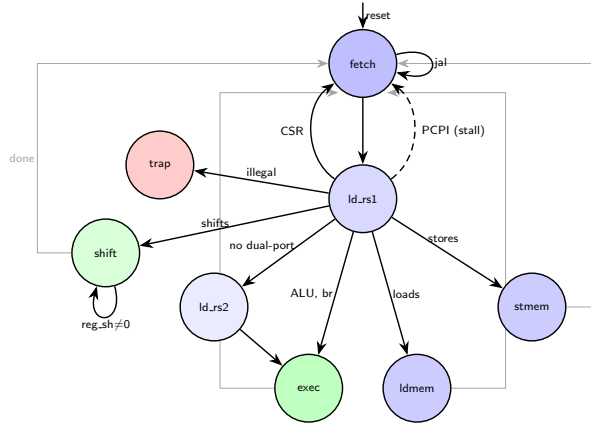


Figure 8: Baseline picorv32 FSM. Blue = memory/storage access (fetch, register read, load/store), green = compute (ALU, shift), red = error. Dispatch arrows (black) fan downward from `ld.rs1`. Return arrows (grey) route along the outer edges back to `fetch`, avoiding crossings.

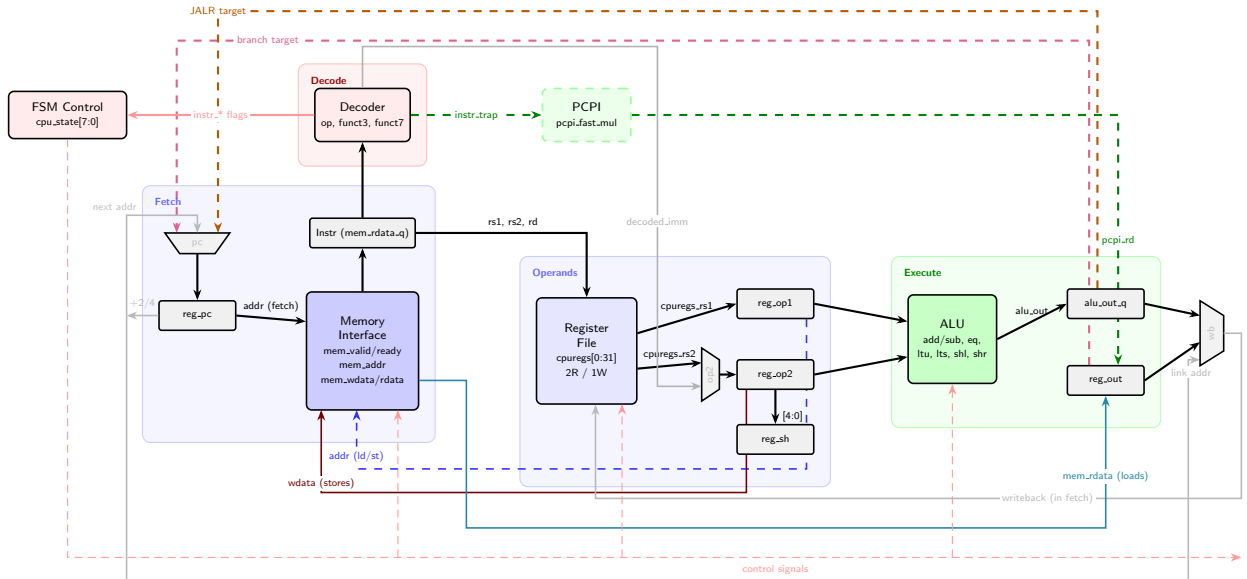


Figure 9: Simplified picorv32 datapath (v2). Three multiplexers control data routing: the **op2 mux** selects the ALU's second operand, the **writeback (wb) mux** selects what is written to the register file, and the **PC mux** selects the next instruction address. A **+2/4 fork** (+2 for COMPRESSED_ISA and +4 for full 32bit addresses) from `reg_pc` computes the next sequential address and feeds both the PC mux (as the default "next addr") and the writeback mux (as the "link addr" for JAL/JALR return addresses; note this is a separate addr, not the ALU). **Forward path** (black): PC → Memory Interface → Instruction Register → Decoder (one-hot flags) and Register File (`rs1/rs2/rd` indices). `reg_op1` receives `cpuregs.rs1`; `reg_op2` receives `cpuregs.rs2` or `decoded_imm` via the op2 mux. Both feed the ALU; its combinational output is registered into `alu_out.q` every cycle. **Feedback paths:** purple dashed = branch target (`reg_out` → PC mux); orange dashed = JALR target (`alu_out.q` → PC mux); teal = load data (`mem_rdata` → `reg_out`); red = store data (`reg_op2` → memory); blue dashed = data address (`reg_op1` + `decoded_imm` → memory); green dashed = PCPI (`pcpi_rd` → `reg_out`); grey = +2/4 fork (`reg_pc` → PC mux and writeback mux). Not shown: in the shift state, `reg_op1` is shifted in-place (1 or 4 bits/cycle) and `reg_sh` is decremented until zero.

⁹The complete source code is many thousands of lines long, and certain claims I make may not be wholly accurate, so I had to check with AI on the factual accuracy of certain behaviours in `picorv32`

Benchmark	Baseline (12 MHz)		Final (22.875 MHz)		Speedup
	CPI	Time (ms)	CPI	Time (ms)	
<i>Compute</i>					
alu	67.59	10421	5.86	474.3	22.0×
shift	67.91	9621	6.02	448.1	21.5×
mul	67.24	11488	5.31	476.5	24.1×
div	68.29	7057	8.29	449.4	15.7×
branch	74.88	18566	6.07	790.2	23.5×
call	70.96	24246	5.79	1038.3	23.4×
<i>Memory</i>					
memcpy	71.40	5534	5.72	232.6	23.8×
chase	71.39	5380	5.77	228.4	23.6×
<i>Fetch</i>					
hot	70.65	23550	5.75	1005.5	23.4×
cold	64.03	5494	5.10	229.8	23.9×
<i>Programs</i>					
bubble_sort	69.76	4396	5.45	180.4	24.4×
matmul	68.85	16769	5.55	709.7	23.6×
crc32	72.55	18594	5.40	726.1	25.6×
prime_count	87.58	3806	9.01	205.5	18.5×
fir	69.59	25100	5.49	1040.6	24.1×
strsearch	81.25	10619	5.61	384.8	27.6×

Table 2: Baseline vs final design performance snapshot, on internal benchmark suite.

Secret	Workload character	Flash mode set	Baseline (s)	Ours (s)	Best (s, group)	Our speedup
0	CORDIC sin/cos, small hot loop	single (!)	4.69	0.2207	0.01715 (4)	21.2×
1	10 × 10 integer matmul	none (default)	0.456	0.01901	0.0112 (5)	24.0×
2	5120-instr straight-line, anti-cache	qddr	0.0193	0.0104	0.0104 (6)	1.9×
3	sort + CRC + soft-mul matmul mix	qddr	2.48	0.6282	0.3844 (5)	3.9×
4	steady-state Kalman filter	qddr	0.121	0.02572	0.01581 (5)	4.7×
5	~1700-instr polynomial loop	none (default)	22.2	0.9602	0.6656 (5)	23.1×
6	ours: ~6 KB body + random RMW	none (default)	5.24	0.3268	0.3268 (6)	16.0×
7	16-tap FIR over 64 samples	qddr	0.0592	0.01081	0.01008 (1)	5.5×
8	~38 KB forward-jump storm	quad	0.286	0.02296	0.02296 (6)	12.5×
Normalised runtime (vs baseline, lower = better)			1.0	0.162	0.247 (1, next best)	
Power (W)			0.76	0.81	0.61 (3)	
Logic cells (ICESTORM.LC)			5280	4591	3670 (8)	
EBR blocks (ICESTORM.RAM)			30	27	4 (7)	

Table 3: Final competition results: nine secret benchmarks (one per team) cross-evaluated on all eight processors. “Flash mode set” is the read mode each firmware programs in its `setup_picosoc()` — *none* means the controller stays at the hardware reset default (qddr on our chip, single-bit SPI on an unmodified one); secret 0 actively forces single-bit SPI on every chip. Bold marks our outright wins; “Our speedup” is against the stock baseline processor. Our design took the best overall normalised runtime of all eight groups.

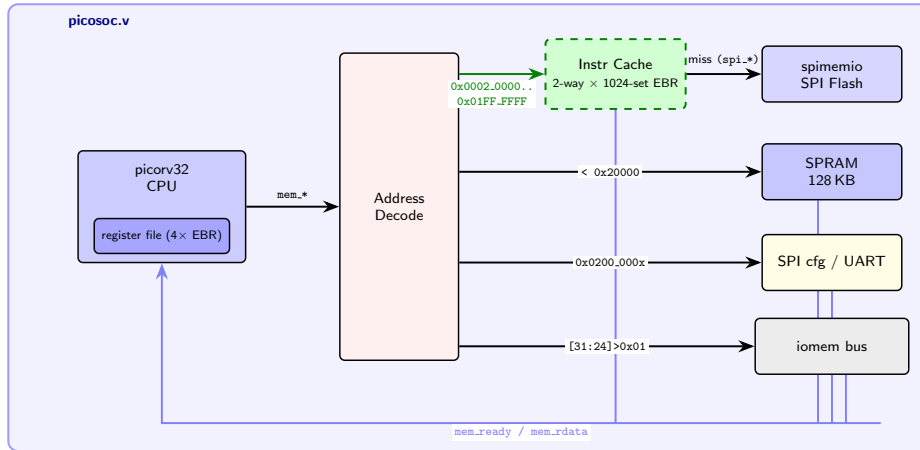


Figure 10: The cache as a memory bus device in `picosoc.v` — hits return on the `mem_*` bus, misses fall through to `spimemio` via `spi_*`.

References

- [1] Patterson, D.A. and Hennessy, J.L., *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed., Morgan Kaufmann, 2017.
- [2] Harris, S.L. and Harris, D.M., *Digital Design and Computer Architecture*, RISC-V ed., Morgan Kaufmann, 2022.