Multi-Issue and Modern Superscalar CPUs

Roadmap

1

In-order issue

Classic in-order dual-issue: hardware can issue multiple instructions, but software order strongly affects throughput.

2

Out-of-order issue

Rename, queues, ROB, and speculation let hardware find ready work dynamically.

3

Hardware cost

Issue width increases pressure on register ports, bypasses, schedulers, queues, and power.

4

Portability case

A real NEON optimization exposes different behavior across A53/A55/A510/A520/A73/A76.

What multi-issue tries to improve

$$\text{CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

Pipeline

Multiple instructions overlap across stages. Ideal single-issue CPI approaches 1.

Multi-issue

Multiple instructions may be issued in one cycle. Ideal CPI can be below 1.

Reality

Issue slots are only useful when operands, functional units, prediction, and memory all cooperate.

In-order multi-issue: width remains constrained by program order

Fetch → **Decode / issue pair** → **Execute in program order**

Hardware capability

The core may issue two instructions per cycle when the available slots can accept them.

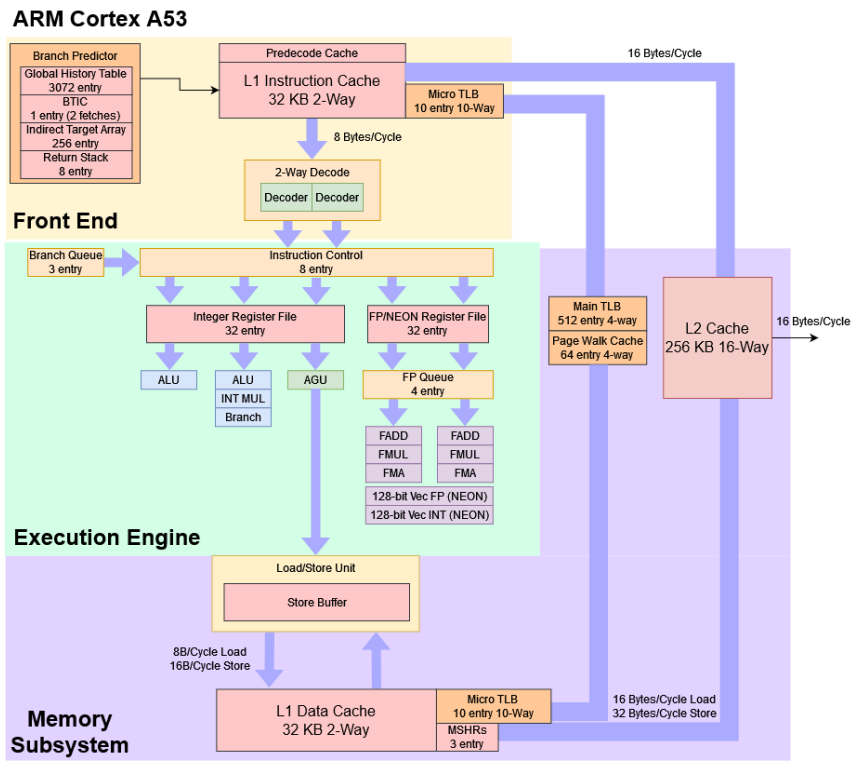
Software responsibility

Instruction order must expose independent work. Unfavorable ordering directly creates pipeline bubbles.

Key limitation

The core cannot skip over an older stalled instruction the way an OoO core can.

Cortex-A53: a classic in-order dual-issue core



Dual-issue, in-order

A53 is a low-power 8-stage core that can issue two instructions when operand dependencies and execution resources permit.

Visible in the diagram

Two-wide decode feeds instruction control; two dispatch paths then serve ALU, multiply, branch, address-generation, and FP/NEON resources.

Vector constraint

The FP queue feeds the vector units, so NEON-heavy kernels are sensitive to this side of the core.

Sources: Chips and Cheese, “ARM’s Cortex A53: Tiny But Important”; SoC Labs, Cortex-A53.

In-order issue is local and resource-constrained

Compatible pair

`ALU + load`, or two instructions that use different execution resources and independent operands.

Blocked pair

Two instructions require the same port, or the second instruction needs a value the first has not produced.

Pair / sequence	In-order issue result	Why
`add` + `ldr`	Often compatible	Different functional paths, if operands are ready.
`ldr` → dependent `add`	Bubble likely	The dependent instruction must wait for loaded data.
Two SIMD multiply-accumulates	Microarchitecture-dependent	Depends on SIMD pipelines and forwarding paths.

On an in-order core, equivalent computations written in different orders can produce different throughput.

Software scheduling: expose independent work

```
Loop:
    lw   t3, 0(t1)
    add  t3, t3, s0
    sw   t3, 0(t1)
    addi t1, t1, -4
    bne  t1, t2, Loop
```

The short loop has true dependencies: `lw → add → sw`, and `addi → bne`.

```
L:
    lw   t3,   0(t1)
    lw   t4,  -4(t1)
    lw   t5,  -8(t1)
    lw   t6, -12(t1)
    add  t3, t3, s0
    add  t4, t4, s0
    add  t5, t5, s0
    add  t6, t6, s0
```

Unrolling and renaming create independent values, giving the scheduler more freedom.

Out-of-order multi-issue: hardware performs dynamic scheduling

Fetch	Decode	Rename	Dispatch	Issue	Execute	Commit

In-order front end

Instructions are fetched and decoded along a predicted program path.

Out-of-order backend

Ready operations can execute before older stalled operations.

In-order commit

The architectural state still changes as if the original program order was followed.

What makes out-of-order multi-issue work?

Register renaming

Maps architectural registers to physical registers, removing WAR/WAW false dependencies.

Reservation stations

Hold operations until source operands are ready, then select ready operations for issue.

Load-store queue

Tracks memory ordering and decides when loads can safely execute around stores.

Reorder buffer

Records in-flight instructions in program order, enables precise exceptions, branch recovery, and in-order commit.

Register renaming: remove name conflicts

```
I1: add r1, r2, r3   ; r1 = a
I2: mul r2, r1, r4   ; r2 = a * c
I3: add r1, r5, r6   ; r1 = d + e
I4: sub r7, r2, r1   ; r7 = (a*c) - (d+e)
```

Without renaming

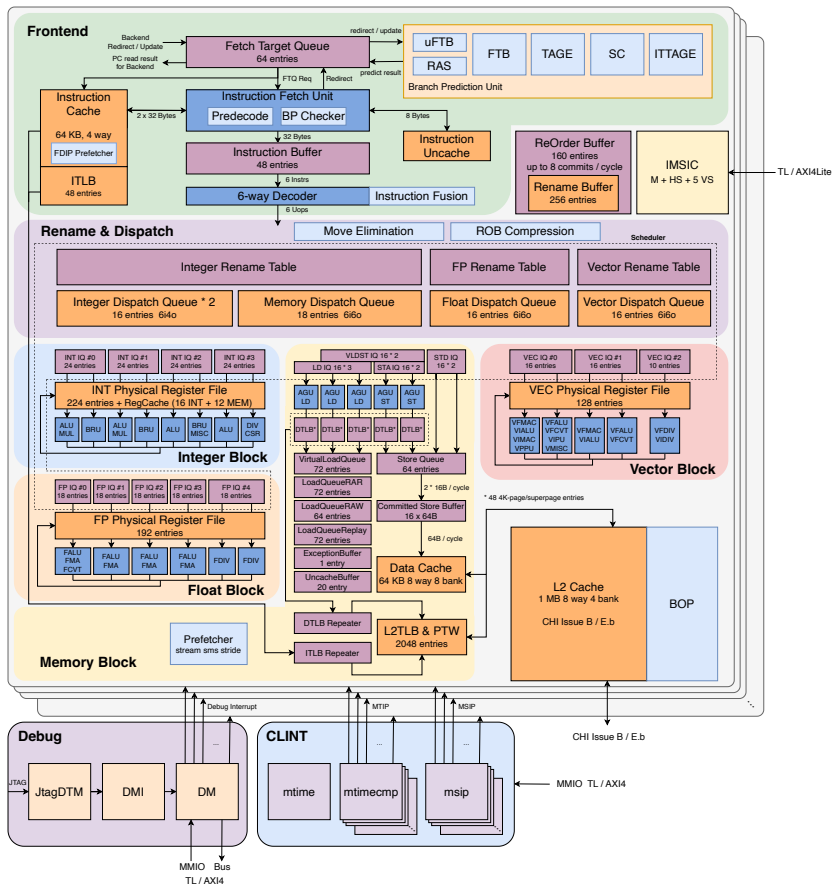
`I3` writes architectural `r1` while `I2` still needs the old `r1`. `I4` also needs the new `r1`. The architectural name alone is ambiguous.

Instruction	Source mapping	Destination mapping
`I1`	`r2 → p12`, `r3 → p13`	`r1: p10 → p40`
`I2`	`r1 → p40`, `r4 → p14`	`r2: p12 → p41`
`I3`	`r5 → p15`, `r6 → p16`	`r1: p40 → p42`
`I4`	`r2 → p41`, `r1 → p42`	`r7: p17 → p43`

What changes

`I2` reads the old value of `r1` from `p40`, while `I3` creates a new value of `r1` in `p42`. The WAR and WAW name conflicts disappear; the RAW value dependencies remain.

XiangShan Kunminghu: an open OoO core



Source: OpenXiangShan documentation, XiangShan processor overview.

Why use this example

Kunminghu is a documented open-source high-performance RISC-V core, so its frontend, backend, cache, and memory-system blocks can be inspected directly.

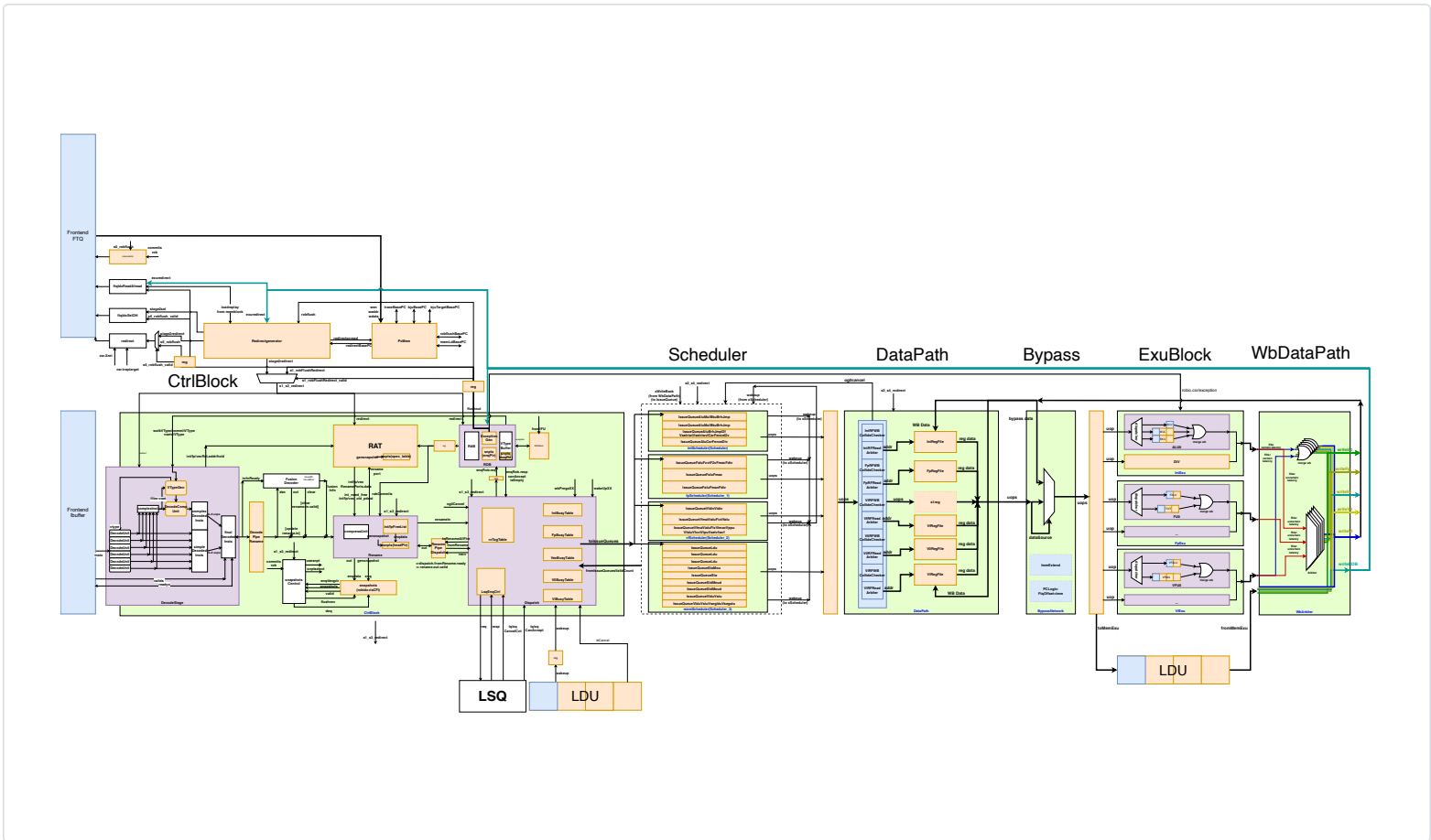
OoO pipeline blocks

The diagram exposes fetch, decode, rename, dispatch, issue, execution, writeback, and commit as concrete hardware structures.

Key takeaway

Useful issue width is an end-to-end property: queues, register state, execution ports, and memory ordering must all keep enough independent work available.

Kunminghu backend: rename, issue, execute, commit



Rename and dispatch

Architectural register names are mapped to physical state before uops enter issue queues.

Issue queues

Ready operations are selected from many in-flight uops, allowing execution to bypass older stalled instructions.

Commit

ROB-based commit preserves precise architectural state even when execution and writeback happen out of order.

Source: OpenXiangShan design documentation, Kunminghu backend.

The hardware cost of multi-issue

Register file ports

More simultaneous reads and writes increase area, energy, and timing pressure.

Bypass network

Every producer may need to forward to many consumers. Wider cores grow many more forwarding paths.

Wakeup / select

The scheduler must find ready operations every cycle, often under tight timing constraints.

Branch speculation

Wide cores must keep the front end busy, so prediction quality matters more.

Load-store machinery

Memory disambiguation and forwarding are needed to keep loads moving safely.

Power budget

A wider core can be faster, but it may be worse for area and energy efficiency.

Wide issue width consumes substantial resources and power

Rename state

Every in-flight destination needs a physical register, a rename-map entry, and recovery state for branch misspeculation.

Scheduler capacity

More issue width requires more entries, wakeup comparators, select logic, and result-broadcast paths.

Execution bandwidth

Integer, vector, load-store, and branch units must be replicated or carefully shared to sustain dispatch width.

Kunminghu example

The backend diagram makes rename tables, issue queues, execution blocks, and commit logic visible as separate resources.

Resource implication

Adding issue width requires more storage, comparators, write ports, result buses, and bypass paths. Those structures increase area, switching activity, and power.

Forwarding improves performance at hardware cost

With forwarding

A dependent instruction may receive a producer's result directly from the pipeline, before register writeback.

This reduces bubbles and lets tight dependency chains run faster.

Without forwarding

The consumer waits until the result is written back to the register file.

The pipeline may stall even after the producer has completed execution.

Producer executes → **Forward result** → **Consumer proceeds**

Forwarding paths cost wires, muxes, verification, and timing margin. Area-efficient cores may omit selected paths.

Case study: an Arm NEON scheduling portability issue

A real FFmpeg / H.266 NEON optimization changed instruction ordering. The patch improved performance on some Arm cores, had limited effect on others, and regressed on another core.

Workload

NEON multiply-long and multiply-accumulate sequence in hand-written AArch64 assembly.

Question

Should independent accumulators be interleaved, or should each accumulation chain be kept contiguous?

Microarchitectural dependency

The answer depends on forwarding, SIMD pipelines, and whether the core is in-order or out-of-order.

Source: quink, Arm NEON scheduling case study, Zhihu.

First identify the computation and dependencies

H.266 AArch64 NEON patch

```
smull v1.4s, v20.4h, v0.h[0]
smull v2.4s, v21.4h, v0.h[1]
smlal v1.4s, v22.4h, v0.h[2]
smlal v2.4s, v23.4h, v0.h[3]
smlal v1.4s, v24.4h, v0.h[4]
smlal v2.4s, v25.4h, v0.h[5]
smlal v1.4s, v26.4h, v0.h[6]
smlal v2.4s, v27.4h, v0.h[7]
```

Pseudocode view

```
v1 = v20 * v0_0;
v2 = v21 * v0_1;
v1 = v1 + v22 * v0_2;
v2 = v2 + v23 * v0_3;
v1 = v1 + v24 * v0_4;
v2 = v2 + v25 * v0_5;
```

The visible dependency is not between every line. It is two accumulator chains: one through `v1`, one through `v2`.

`smull`

Signed multiply long: multiply signed integers, then widen the result so the product does not overflow the original element width.

`smlal`

Signed multiply-add long: multiply, widen, then add the product into the destination accumulator.

Two equivalent orderings, different pipeline behavior

Interleaved accumulators

```
smull v1.4s, v20.4h, v0.h[0]
smull v2.4s, v21.4h, v0.h[1]
smlal v1.4s, v22.4h, v0.h[2]
smlal v2.4s, v23.4h, v0.h[3]
smlal v1.4s, v24.4h, v0.h[4]
smlal v2.4s, v25.4h, v0.h[5]
smlal v1.4s, v26.4h, v0.h[6]
smlal v2.4s, v27.4h, v0.h[7]
```

Motivation: separate dependent `v1` operations with `v2` work, so an in-order multi-issue core has independent instructions to fill issue slots.

Contiguous accumulation chain

```
smull v1.4s, v20.4h, v0.h[0]
smlal v1.4s, v22.4h, v0.h[2]
smlal v1.4s, v24.4h, v0.h[4]
smlal v1.4s, v26.4h, v0.h[6]
smull v2.4s, v21.4h, v0.h[1]
smlal v2.4s, v23.4h, v0.h[3]
smlal v2.4s, v25.4h, v0.h[5]
smlal v2.4s, v27.4h, v0.h[7]
```

Counterpoint: if the core has an FMA / accumulate forwarding path, the next `smlal` can consume the previous accumulator value directly.

An ordering that exposes more independent instructions may underperform when consecutive MAC instructions activate a specialized forwarding path.

SIMD MAC forwarding: the relevant hardware condition

Instruction family

Integer SIMD multiply / multiply-accumulate operations, including `SMULL`, `SMLAL`, `UMLAL`, `SMLSL`, `UDOT`, and `SDOT`.

When it is active

The MAC accumulator value can be forwarded directly to the next MAC instruction, reducing the effective dependency latency to about one cycle.

Same accumulator

Both consecutive instructions read from and write to the same destination / accumulator register.

Same element size

The destination element width must match across the two instructions.

Same register size

The destination register size must match, such as 128-bit with 128-bit or 64-bit with 64-bit.

This refines the scheduling intuition: interleaving `v1` and `v2` benefits generic multi-issue, while consecutive `smlal v1` can activate a specialized forwarding path.

A510 versus A520: the same schedule changes sign

Measured performance shift

The article reports a clear reversal: the contiguous-accumulator rewrite helps Cortex-A53/A55 by about 39%, is almost neutral on A73/A76, slows Cortex-A510, and helps again on Cortex-A520.
One A510 8x8 NEON case drops from 2.15x to 1.41x after the rewrite.

Arm manual evidence

Arm documentation for A55 and A520 describes a SIMD MAC accumulator forwarding path: compatible consecutive MAC instructions can receive the accumulator value directly.
The article notes that A510 documentation has no equivalent vector MAC forwarding section.

Why the result differs

When forwarding exists, a contiguous `smlal` chain can turn a dependency into a short one-cycle path. Without it, the next instruction must wait for the accumulator through the normal register path, creating bubbles.
The same assembly therefore exposes different hidden costs on different generations.

The scheduling rule is not universal. Instruction latency, functional-unit count, forwarding paths, branch behavior, and out-of-order capability jointly determine whether a hand schedule is an optimization or a regression.

Source: quink, Arm NEON scheduling case study, Zhihu; Arm optimization-guide excerpts cited in the article.

Forwarding changes the schedule that wins

With MAC forwarding

A55 and A520 can forward the accumulator value to the next compatible SIMD MAC instruction, so contiguous chains can issue with low effective latency.

Without that path

A510 must wait for the accumulator value to become available through the normal path, so contiguous chains can create bubbles.

Why this is fragile

The best order depends on latency, unit count, forwarding paths, branch behavior, and whether the core can schedule around stalls.

Portability implication

A kernel tuned deeply for one efficiency core can be neutral on a large OoO core and negative on another efficiency core.

Optimization target

For heterogeneous phones, the target is often not the single fastest kernel, but a version that avoids large regressions across big and little cores.

Source: quink, Arm NEON scheduling case study, Zhihu; Arm optimization-guide excerpts cited in the article.