Booting an Operating System on a RISC-V Soft Core: A Development History from RV32/SERV to an Interrupt-Driven Zircon on NOEL-V

Filip Filmar
Independent
filmil@gmail.com

***Abstract*—We chronicle the multi-month bring-up of a full operating-system stack on a RISC-V soft core on a single low-cost FPGA: from a bit-serial RV32 bootstrap loader, through core, UART, and DRAM bring-up on a 64-bit GRLIB NOEL-V core, to M-mode firmware (OpenSBI), a supervisor-mode boot ladder, and finally a fully interrupt-driven console for Google's Zircon microkernel. The value of the account is less any single fix than the *pattern*: almost every failure traced not to the component under suspicion but to a non-standard choice in our own environment—an unconfigured scan input, a 32-bit core running 64-bit code, a byte-write strobe hardwired on, a device tree that under- or over-declared RAM, an interrupt-controller that was correct all along, and two RISC-V extensions (*Smrnmi*, *Sstc*) that were present in silicon but never enabled or advertised. We describe a three-vantage debugging methodology—isolated IP simulation, non-perturbing bare-metal probes, and a JTAG debug module that reads CSRs, arbitrary memory, and the kernel's own in-RAM log—that repeatedly separated real hardware faults from measurement artifacts and firmware omissions, and that made a low-cost, GPL-only toolchain sufficient to debug a proprietary core booting a production microkernel.**

***Index Terms*—RISC-V, NOEL-V, SERV, Zircon, Fuchsia, OpenSBI, PLIC, Smrnmi, Sstc, DDR3, FPGA, JTAG, bring-up, debugging.**

I. Introduction

Soft-core RISC-V on an FPGA is uniquely inspectable: RTL, firmware, and kernel are all editable, and a debug module can read any register or memory word on the live system. That same openness means a single observable symptom—a core that fetches garbage, a store that "never reaches the bus," a console that goes quiet—can originate in any layer, and the layers interact in ways that defeat single-layer reasoning. This paper is a development history of exactly that situation, assembled from the working notes of the bring-up. The complete design—the RTL, the RISC-V firmware and boot chain, and the Bazel build that ties them together—is available as open source at https://github.com/filmil/cocoapuffs-fpga.

The target is Zircon, the microkernel under Google's Fuchsia, running on a NOEL-V core (from the GRLIB IP library) synthesized to a Xilinx Artix-7 (xc7a200t) at 40 MHz. Reaching a booting kernel required climbing a long ladder, and the recurring lesson—which the developer's notes phrase as "why doesn't everyone hit this?"—is that *every* obstacle was a consequence of a non-standard decision in the build, not a defect in the widely used component we first blamed. We present the arc in the order it was climbed:

- §II: the RV32/SERV bootstrap that loads and launches the 64-bit core;
- §III: core bring-up—an X-propagation that poisoned the whole pipeline in simulation, a core accidentally built 32-bit, and a UART that worked in simulation but not on silicon;
- §IV: the DRAM subsystem—a byte-write strobe hardwired on, and a 128 MB aliasing bug that masqueraded as a memory-size limit;
- §V: M-mode firmware—OpenSBI wedged by heap corruption and by load-reserved/store-conditional that this core does not hold on uncached DRAM;
- §VI: the supervisor-mode boot ladder to a full Zircon boot, including a shim that trapped inside a bounds-checked C++ iterator and a data image placed on top of the boot stub;
- §VII: the interrupt-driven console, whose five-layer root cause is the technical climax; and
- §VIII: the debugging methodology that made all of it tractable.

A timeline of the bring-up, drawn from the project's commit history, appears in Appendix A.

II. The Bootstrap Architecture

The board carries no boot ROM and no bootloader flash. Bring-up therefore began not with the NOEL-V but with **SERV**, a bit-serial RV32 core small enough to bootstrap from block RAM. Early boards (`rv32_uart`, `rv32_ddr3`, `rv32_ddr3exec`) established, incrementally, that the RV32

core could drive the UART, reach DDR3, and *execute* from DDR3.

That RV32 core then became the permanent first stage. In the production design the SERV core boots, runs an Intel-HEX *uploader* over the serial port, and streams an arbitrary image into DRAM; it then writes a control register that switches the UART mux to the NOEL-V and releases the big core's reset. The NOEL-V begins executing the just-loaded image. This two-core hand-off is the substrate for everything that follows: every experiment in this paper is an image streamed to the SERV loader and launched on the NOEL-V. (An intermediate 64-bit core, *Muntjac*, was also explored; a simulation-timeout quirk made it a poor iteration vehicle, and effort concentrated on the NOEL-V.)

III. Bringing Up the Core

A. An X that poisoned the pipeline

The first milestone was simply getting the NOEL-V to run a boot program and print "halt." It did not: in simulation the core mispredicted branches, speculatively fetched address 0, and *dropped register writebacks*, so a `lui t0, 0xFF900` became `t0=0` and the UART store went to address 0 instead of `0xFF900000`. The cause was neither the fetch unit nor the store path but an *unconfigured scan interface*: the core's internal AHB slave record was assembled field by field and never assigned `testen`/`testrst`/`scanen` (and `endian`). NOEL-V derives its scan-mux controls from those inputs, so leaving them `'U'` propagated X across flops and RAMs throughout the core. Driving them from the parent bus fixed it, and the core printed "halt." A general lesson recorded at the time: in four-state simulation any undriven `std_logic` on a control path becomes X and spreads far; scan/test inputs are the worst offenders. Diagnose with a VCD scan for signals stuck at `x`/`u` after reset.

B. A 64-bit core that was 32-bit

The next symptom looked like a memory fault: a store to DDR3 "never reached the bus"; the bus-interface never asserted its request. Days of tracing the store path were wasted, because the store was a red herring. An instruction-disassembly trace showed *32-bit register widths*: the build had compiled `noelv_cfg_32.vhd` (`NV_XLEN=32`). An RV32 core running RV64 code *deadlocked on the first 64-bit word operation* (`addw`), several instructions *before* the store. The `cfg` generic (§III sidebar) selects the microarchitecture family—`0x300` is the dual-issue "GP" core intended for FPGA—but *not* XLEN, which is the separate global `NV_XLEN`. Switching to `noelv_cfg_64.vhd` produced 64-bit registers and a reachable, working store (AHB monitor: `WR 0x40000 hwdata=0xCAFEF00D`, then a matching read-back). Two further self-inflicted wrinkles surfaced in the tiny hand-written boot assembly: `lui t0, 0xFF900` sign-extends to an invalid high address on RV64 (use `li`+`slli`), and `lw` of a value with bit 31 set sign-extends and fails a naive compare (use `lwu`).

C. A UART that passed simulation and failed on silicon

With the core running, the on-board serial output was a constant square wave. The core wrote the correct bytes to the UART data register (confirmed on debug LEDs capturing the AHB write stream), and the testbench UART decoded "halt." from the transmitter in simulation—yet the GRLIB `apbuart_16550` transmitter garbled on the real FPGA. Rather than chase a vendored 16550, we switched to the Gaisler `apbuart` (same `uarti`/`uarto` interface, so no board rewiring), rewrote the boot code for its register map, and set the scaler for 115200 at the board clock. "halt." then appeared at 115200 8N1. A latent cross-domain bug was fixed in passing: a reset synchronizer for the NOEL-V's reset, released from the SERV clock domain.

IV. The Memory Subsystem

A. A byte write that wrote every byte

Aligned 32- and 64-bit stores worked; single-byte stores (`sb`) to DDR3 did not—and that broke OpenSBI, whose `sbi_memcpy`/`sbi_memset` are byte-wise. The failure manifested far downstream, as `sbi_domain_init` returning `SBI_EINVAL`: a byte-wise region-sort corrupted the root domain's memory-region table (two swapped entries came back zeroed), and the firmware hung. A raw bit-bang tracer patched into OpenSBI localized it, and `//bin/noelv_bytetest` reproduced it in isolation. The bug was in the AHB-to-Wishbone bridge, which hardwired the Wishbone byte-select strobes to all-ones and never captured the AHB `hsize`: because the NOEL-V replicates a sub-word datum across all lanes, an `sb` of `0xAA` wrote `0xAAAAAAAA`. Deriving the strobes from `hsize` fixed it; a focused testbench (`//ip/bridges`) now regression-tests byte, halfword, word, and burst writes against a lane-honoring model.

B. A 1 GB memory that looked like 128 MB

The board appeared limited to 128 MB of DRAM. The chip, the UberDDR3 controller, and the AHB decode were all 1 GB; the cap was an adapter that dropped byte-address bits, aliasing the 1 GB window every 128 MB. A one-line address-slice fix restored the full gigabyte, verified by a caches-off march test (`//bin/noelv_memtest1g`) that writes a distinct value to each 128 MB bank and reads all back, and confirmed end to end by a Zircon boot reporting ∼1.0 GB free after kernel init. This subsystem accreted a small suite of bare-metal tests—byte, data-section, read/write, atomic (AMO), and load-reserved/store-conditional probes—each of which paid for itself later.

TABLE I
THE SUPERVISOR-MODE BOOT LADDER.

#	What the image proves	Observable
0	OpenSBI loads and M-mode runs	OpenSBI banner
1	OpenSBI → S-mode hand-off; S-mode executes	"hello" payload
2	The Linux boot shim loads, runs in S-mode, parses the DTB	shim diagnostics
3	DTB `/chosen` initrd → shim finds the ZBI → physboot runs	physboot log
4	physboot relocates and starts the kernel → userboot	kernel/userboot log

V. M-Mode Firmware: OpenSBI

Once memory was trustworthy, OpenSBI became the focus. Beyond the byte-write heap corruption above, two core-specific issues blocked the supervisor hand-off. First, this NOEL-V does not hold `lr.d/sc.d` reservations on uncached DDR3, so OpenSBI's single use of compare-and-swap (in `atomic_cmpxchg`) spun forever; since the system is single-hart, a patch emulates CAS in an interrupt-safe critical section. Second, the core enters OpenSBI with `mseccfg.MML` set, which the feature detector treats as fatal; the vendored entry code clears it. With a handful of additional patches (a drained-putc console so diagnostics are legible on hardware, trusted-FDT parsing to keep RTL simulation tractable, and trap/init tracers), OpenSBI boots fully: banner, domain configuration, and a hand-off to a supervisor-mode payload at a fixed address with the hart ID in `a0` and the (expanded) device tree in `a1`.

VI. The Supervisor Boot Ladder

To keep failures localized, the Zircon bring-up was structured as a *ladder* of increasingly complete bootable images (Table I), each climbed only after the one below was green.

Rung 1 revealed a baud mismatch: a payload hard-coded a 50 MHz scaler, giving ∼92.6 kbaud on the 40 MHz board—the banner was transmitted, just unreadable at a 115200 capture. Rung 2 was subtler. The shim loaded, self-relocated, and ran, then trapped on an `unimp` that `addr2line` placed inside `devicetree::Match`: a hardened-libc++ `__builtin_trap` from a *bounds-checked* `std::array` iterator, tripped because our minimal device tree lacked nodes the matchers expected (a PLIC node, a fuller CPU topology). Completing the device tree cleared it. Rungs 3–4 fought placement and memory-map issues: physboot placed the data ZBI at address 0, overwriting the boot stub, until the ZBI load address and the declared `/memory` range were reconciled; caches had to be disabled for the uncached boot path; a null frame-pointer guard was needed in the backtrace walker; and the vector extension had to be disabled (the core has no V). The payload lands at a fixed supervisor address, and the device tree—patched on the host, since there is no U-Boot—carries the ZBI's location in `/chosen`.

The reward was a full boot: OpenSBI → shim → physboot → the kernel (complete init, memory-pressure subsystem, "starting user space") → `userboot`, terminating—correctly—at a deliberate `__builtin_trap` because the engineering ZBI carries no boot filesystem. This is the same terminus a reference emulator reaches for a filesystem-less image. An earlier build had one outstanding workaround—with address-space layout randomization enabled, `userboot` took an instruction-access fault at its randomized high base before running, so the configuration disabled ASLR—but that no longer reproduces: on the final build `userboot` self-relocates and runs to the same terminus with ASLR enabled, verified across two independent random placements. The fault had been collateral of a since-fixed problem (the interrupt-firmware and memory-map changes landed in between), and ASLR is now left on.

VII. The Interrupt-Driven Console

Zircon's early output is polled, but its steady-state console is a debug-log *dumper* thread that performs a *blocking, interrupt-driven* transmit: it writes a byte and, if the FIFO is full, enables the UART transmit interrupt and blocks until that interrupt reports space. On our board that thread never made progress; only the polled fallback (`kernel.debug_uart_poll=true`) produced late-boot output. Resolving this "last mile" took five independent fixes across four layers, and—as with every earlier stage—the component everyone suspects, here the platform interrupt controller (PLIC), was innocent.

A. Two software bugs on the delivery path

The device tree routed the UART interrupt with the modern `interrupts-extended = <&plic0 1>` form, but the shim's chosen-node matcher read only the legacy `interrupts` property, resolving the console IRQ to 0 so the kernel never unmasked PLIC source 1. Independently, the kernel PLIC driver's priority macro computed `base + 1 + irq`, writing source i's priority into source $i{+}1$'s slot and leaving source i at priority 0—below threshold, hence enabled but never delivered. Both were fixed; neither restored the console.

B. Exonerating the PLIC

A JTAG snapshot then showed the PLIC fully and correctly configured for source 1 in the supervisor context, yet `sip.SEIP`=0. Reading the claim register returned source 1, which we first read as "the PLIC identifies the interrupt but fails to drive `seip`"—a hardware bug. An *isolated* simulation of the `grplic_ahb` block, driven over an AHB bus-functional model exactly as the kernel drives it, refuted this: given the observed configuration and an asserted source, the block *does* assert `seip`, and the claim read *clears* it. The "`seip`=0 while pending" reading had been a *measurement artifact*—the claim read that produced the "source 1" datum had itself cleared the pending interrupt. The PLIC was correct throughout.

C. The core takes no interrupts

If delivery was fine, why no trap? A bare-metal probe armed the most trivial interrupt possible—a machine software interrupt—with `mstatus.MIE`=1 and its enable set, verified the pending and enable bits over JTAG, and observed *no trap*. The same held for timer and external interrupts. Reading the core RTL resolved it: this NOEL-V implements *Smrnmi*, and the integer unit masks *all* interrupts while `mnstatus.NMIE`=0—its reset value, which nothing in the boot chain ever changed. Setting it (`csrs 0x744, 8`) let the bare-metal probe take the full supervisor-external path and claim source 1. Smrnmi, not the PLIC, was the root cause of "no interrupts."

D. A wedge at the supervisor hand-off

Setting `NMIE` in firmware produced a new failure: the kernel emitted nothing. With interrupts unmasked, any M-mode interrupt still armed in `mie` fired the instant OpenSBI executed `mret` into supervisor mode—M-interrupts are always taken in a lower privilege—before the kernel installed its trap vector, wedging it at entry. Masking `mie` immediately before the `mret` (Listing 1) fixed it; the kernel's own supervisor interrupts are untouched, and OpenSBI re-arms M-interrupts on demand at its next ecall.

Listing 1. The two coupled firmware changes.

```
/* fw_base.S, _start_warm, after CSR_MIE is cleared: */
    csrsi   0x744, 8          /* mnstatus.NMIE = 1 */

/* sbi_hart.c, just before the mret to S-mode: */
    csr_write(CSR_MIE, 0);    /* no M-interrupt at S-entry
      */
```

E. A "stall" that was really a crawl

The kernel now booted and *took* timer interrupts (JTAG: `scause` showing the supervisor timer), yet appeared to hang in threading-level init. The debug-log ring, dumped over JTAG, told a different story: it was advancing, just extraordinarily slowly—timestamps put the boot at about nineteen minutes. The device tree declared `riscv,isa = "rv64imac"`, so Zircon could not tell the core implemented *Sstc* and armed every scheduler-tick timer through an SBI ecall (a supervisor→machine→supervisor round trip) instead of writing `stimecmp` directly. Advertising the extension (`riscv,isa = "rv64imac_sstc"`) let the kernel use the direct compare register; the boot dropped to about seventeen seconds and, with the polled fallback removed, the *entire* boot streamed over the interrupt-driven console to the expected terminus, past the init hook at which output had previously ceased.

F. The five-part fix

Table II collects the changes. They span four layers; no subset yields a fast, fully interrupt-driven console.

TABLE II
The five-part fix for the interrupt console.

#	Fix	Layer
1	Write each source's *own* PLIC priority register	Kernel PLIC driver
2	Resolve the console IRQ from `interrupts-extended`	Boot shim
3	Set `mnstatus.NMIE`=1 (unmask on this Smrnmi core)	M-mode firmware
4	Mask `mie` before the S-mode `mret`	M-mode firmware
5	Advertise `sstc` (fast direct-`stimecmp` timer)	Device tree

VIII. Debugging Methodology

Three vantage points, used throughout the arc, made the difference; a disagreement among them localizes a fault.

a) Isolated IP simulation: Simulating one block (the UART transmitter, the AHB bridge, the PLIC) with a bus functional model observes signals software cannot see and converts "probably a hardware bug" into a proof one way or the other. The PLIC exoneration is the sharpest example.

b) Non-perturbing bare-metal probes: Small OS-free programs arm one interrupt or exercise one memory pattern at a time. The interrupt probes deliberately read pending state *without* reading the claim register, precisely because an earlier measurement had been corrupted by that register's side effect.

c) A GPL-only JTAG debug module: The proprietary core's professional debugger refuses to attach, so we drove the GRLIB JTAG bridge directly from `openocd` (GPL) as a plain AHB master: ground-truth reads of CSRs, of live peripheral registers (PLIC, CLINT), and—most valuable—of the kernel's in-RAM debug-log ring, which recovered boot progress when nothing reached the wire. Its limits were charted the hard way (unreliable single-step; stepping into unloaded RAM wedges the bus), and respected.

IX. Lessons Learned

a) Suspect your own non-standard choices first: An unconfigured scan input, a 32-bit core running 64-bit code, a hardwired byte strobe, an aliasing address slice, an under-declared device tree—each broke a component that works for everyone else. The recurring question "why doesn't everyone hit this?" had the same answer every time: because everyone else's configuration is standard.

b) Beware self-perturbing measurements: The "`seip`=0 while pending" reading was an artifact of reading the claim register. On hardware with side-effecting registers, non-invasive probes are worth the extra care.

c) Extensions must be enabled and *advertised: Smrnmi* silently masked all interrupts until firmware enabled it; *Sstc*, present in silicon but absent from `riscv,isa`, silently forced a 100× slower timer path. Neither produced

a diagnostic; both produced "it doesn't work" or "it hangs."

d) Distinguish "stalled" from "slow": A log absent from the wire is not necessarily a log that is not being written. Recovering the in-RAM ring over JTAG and reading its timestamps turned a presumed deadlock into a measured, then eliminated, performance problem.

e) Climb a ladder: Structuring the bring-up as bootable rungs, each adding one layer, meant every new failure was localized to the layer just added—indispensable when a symptom can originate in any of five.

X. Conclusion

Booting a production microkernel on a RISC-V soft core, end to end on one FPGA, was a sequence of layer-crossing puzzles whose common thread was that the suspected component was almost never the culprit. The interrupt controller, the DRAM, the store path, and the UART core were each exonerated in turn; the real causes lay in configuration, in a bridge strobe, in firmware that failed to enable a masking extension, and in a device tree that failed to advertise a timer extension. A disciplined, multi-vantage methodology—isolating IP in simulation, probing bare metal without perturbing state, and reading ground truth (including the kernel's own log) over a GPL JTAG bridge—was what made a low-cost, fully open toolchain sufficient to find each cause among its several convincing decoys.

Appendix A
Timeline of the Bring-Up

Table III maps the bring-up to dates, drawn from the project's own commit history. The RV32/SERV foundation and the DDR3/UART bring-up span the first year; the NOEL-V, OpenSBI, and Zircon work is concentrated in a dense final month in which the boot ladder was climbed rung by rung to a full, and then interrupt-driven, boot.

TABLE III
Timeline of the bring-up.

Period	Milestone
2024 Q3	Project begins; SERV RV32 bit-serial core, UART, and DDR3 bring-up in simulation and on the FPGA
2024–25	RV32/SERV boards mature: Intel-HEX serial uploader, DDR3 access and execute-in-place
2026-03	Muntjac RV64 core evaluated as an intermediate
2026-06-14	NOEL-V core introduced
2026-06-19	NOEL-V runs: scan-input X-propagation fixed, RV32→RV64 config, "halt." on UART (16550→Gaisler `apbuart`)
2026-06-21	Dual-issue NV-HP core builds; OpenSBI `fw_payload` built
2026-06-26	Zircon boot ladder started; shim device-tree bounds-trap root-caused; JTAG debug module (`dsuen`) + `openocd`/ahbjtag
2026-06-27	OpenSBI boots fully; DDR3 byte-write bridge strobes fixed
2026-06-28–30	Faithful ZBI RTL simulation; a sim-only OpenSBI early-init timing hazard isolated
2026-07-05	Second-simulator (NVC) OpenSBI boot baseline
2026-07-07	DDR3 128 MB aliasing fixed (adapter address slice); UART 40 MHz clock
2026-07-08	L1 caches enabled in the boot stub (the Zircon LR/SC fix)
2026-07-09	Full Zircon boot to the expected no-bootfs terminus; PLIC node added
2026-07-10	Full 1 GB DDR3 verified; entire boot streamed on the polled serial console
2026-07-13	Interrupt-driven console solved (Smrnmi `NMIE` + `mie`-quiesce + `sstc`)