A Differentiable von Neumann Computer

Architecture, Cross-Implementation Verification, and the Open Synthesis Question

Jacob Valdez
jacobfv123@gmail.com

August 15, 2026

Abstract

A conventional processor is a switch statement, and that discreteness is what makes machine code invisible to gradient descent. We replace every discrete choice in a von Neumann computer with a temperature-scaled softmax: all 32 ALU operations execute and are combined by weight, the program counter is a distribution over instruction addresses, register selection is a convex blend, and memory access is attention. The result is a complete machine — ALU, register file, flags, call stack, cache with learned eviction, content-addressed memory, paged RAM, a sector-granular disk, an arbitrated bus, a raster GPU, peripheral ring buffers, and a framebuffer — that is differentiable with respect to its own program and that executes exact discrete machine code as $\tau \to 0$.

Our central claim is that this soft machine and its discrete counterpart are the same machine, and we establish it by construction rather than by assertion. Four independent implementations of the instruction set exist — a reference C virtual machine, a Rust virtual machine, a batched C++ simulator, and the differentiable JAX machine — held in agreement by 59 conformance vectors across four suites. Recursive `factorial(5)` returns 120 in exactly 49 cycles on all four, including on the soft machine at $\tau = 10^{-3}$. Differential testing across the four implementations found six classes of defect that no single implementation could have surfaced, because each was internally consistent; we report the methodology and the defects, since both generalise beyond this system. The machine runs real software: `tcc`, a self-hosting C compiler, occupies 40,942 of 65,536 instruction slots and, running on the machine, compiles a C program in 1.26 M cycles to output byte-identical to the host-built compiler's; *tensor-os* adds a preemptive scheduler, thirteen system calls, and a damage-tracking windowing compositor with five applications in 20,256 slots.

We do not report a synthesis result. The differentiable training path is implemented and verified end to end by fourteen checks across nine stages, but at the budgets shipped here the learned kernels are wrong — scoring below the trivial baseline of returning an input unchanged — and the experiment that would settle whether gradients earn their cost against discrete search is written and unrun. We state what the construction did establish, what it cost, and what remains open.

1 Introduction

A conventional processor is a switch statement. It decodes an opcode, jumps to one branch, and discards the other thirty-one. That discreteness is what makes machine code invisible to gradient descent: there is no derivative of "execute `ADD` instead of `XOR`", because nothing continuous separates the two. Every technique that has learned programs at scale therefore works around

the machine rather than through it — enumerating candidates, solving constraints, or generating source text with a model that never executes anything.

This paper takes the other route. We replace every discrete choice in a von Neumann computer with a temperature-scaled softmax and ask what that costs. All 32 ALU operations execute on every cycle and are combined by weight; the program counter is a probability distribution over instruction addresses rather than an index; register selection is a convex blend; memory access is attention over an address space; bus arbitration is a softmax over priority-weighted requests. A single temperature τ interpolates between two regimes. At $\tau \to \infty$ every selector is uniform and gradient reaches everywhere. At $\tau \to 0$ every softmax collapses to a one-hot vector and the machine executes exact, discrete machine code.

1.1 What This Paper Reports

The construction is the easy half. The hard half is establishing that the soft machine and a real machine are the *same* machine, because the whole premise depends on it: a program discovered in the soft regime is worthless unless it is a program in the discrete one. We treat that as a property to be checked rather than a claim to be made. Four independent implementations of this instruction set exist — a reference C virtual machine, a Rust virtual machine, a batched C++ simulator, and the differentiable JAX machine — and 59 conformance vectors in four suites hold them in agreement (Section 4). Recursive `factorial(5)` returns 120 in exactly 49 cycles on all four, the soft machine at $\tau = 10^{-3}$ included.

Building four implementations of one specification and differential-testing them against each other turned out to be the most productive thing we did. It surfaced six classes of defect that no single implementation could have found, because each implementation was internally consistent and passed every test written for it (Section 4.4). Four backends had independently chosen four different random number generators. A straight-through estimator had its operands inverted, giving it exactly zero gradient — a defect whose symptom would have been a training run that annealed normally and then silently stopped learning. We report the methodology as a result, because the situation that produced these defects — one specification, several implementations, tests written per implementation — is not unusual.

We also report that the machine runs real software, at a scale that settles whether it is a toy. `tcc`, a C compiler written in the subset of C it compiles, occupies 40,942 of the paper profile's 65,536 instruction slots and, *running on the machine*, compiles a C program in 1.26 M cycles to output byte-identical to the host-built compiler's (Section 5). *tensor-os* boots on the same machine with a preemptive scheduler, IRQ dispatch, a page allocator, thirteen system calls, four device drivers, and a damage-tracking windowing compositor running five applications, in 20,256 slots.

1.2 What This Paper Does Not Report

It does not report a program synthesis result. The differentiable training path described in Section 7 is implemented and each of its links is verified end to end — declare a kernel, train its instruction-memory logits, crystallise, extract discrete machine code, cross-verify the extracted program on the Rust virtual machine, and splice it back into hand-written assembly — but at the budgets shipped in this repository the learned kernels are wrong, in the specific sense that they score below the trivial baseline of returning an input unchanged and fault on some of their examples (Section 7.7). The experiment that would settle whether gradients earn their

cost against pure discrete search is written and deliberately unrun, because answering it on inadequate compute would produce a number worse than silence.

We state this early and repeat it where it is relevant. The reframing is not modesty: a paper that reported the machine, the verification, and the software while implying a synthesis result would be making its weakest claim carry its strongest section. Section 7.7 gives the measured state of the learning question and Section 10 gives what would settle it.

1.3 Contributions

1. **A complete differentiable von Neumann machine** (Section 3). ALU, register file, flags, call stack, cache with soft-LRU eviction, content-addressed memory, paged RAM with a soft TLB, a sector-granular disk with write-ahead journalling, an arbitrated bus, a raster GPU, peripheral ring buffers, and a framebuffer — every one a pure differentiable function of arrays, with no discrete index anywhere in the machine state.
2. **A verification methodology, and what it found** (Section 4). Four independent implementations, 59 conformance vectors, and an equivalence check that pins the soft machine at $\tau = 10^{-3}$ to the reference virtual machine's register file within 10^{-4}. Six classes of divergence that single-implementation testing cannot surface, including one in the fault semantics that no vector could express until the vector format grew a way to say "this program is supposed to fail".
3. **Two software artefacts in the 10^4-instruction range** (Section 5). A self-hosting C compiler and a preemptively multitasking operating system with a windowing GUI, both running on the machine, together with the bounded negative result that stops the compiler's fixed point from closing.
4. **Three architectural findings the construction forced** (Section 6). The instruction set cannot express a signed comparison at full width, and the defect appeared independently in three places including the evaluation ground truth. The machine's information capacity, not its instruction budget, bounds what can be loaded into it. And a device command whose cost is independent of the area it covers inverts the usual raster cost model, so a compositor on this machine is text-bound rather than fill-bound.
5. **The measured state of the learning question** (Section 7.7, Section 10). Including the scale argument that forces a hybrid design, and a correction to a claim about vanishing gradients that this repository believed for some time and that is wrong.

1.4 Organisation

Section 2 places the machine among neural computers, discrete relaxations, and program synthesis. Section 3 is the specification and Section 4 the verification methodology that holds it to four implementations. Section 5 covers the two software artefacts and Section 6 the cost model and the three findings they forced. Section 7 describes the training method as designed and implemented, marks which parts have been exercised, and ends with the current state of the synthesis question. Section 8 keeps the formal claims the construction supports and names the one it assumes. Section 9 gives the scale argument for a hybrid design, Section 10 what remains, and Section 11 and Section 12 close.

2 Background and Related Work

2.1 Neural Computers

The Neural Turing Machine (Graves et al., 2014) attaches a differentiable external memory to a controller network. A read is $\mathbf{r} = \sum_i w_i \mathbf{M}_i$ for a weighting $\mathbf{w}$ produced by content-based addressing — a softmax over cosine similarity between a query key and each memory row, sharpened by a scalar — composed with location-based addressing via an interpolation gate, a convolutional shift, and an exponent. The Differentiable Neural Computer (Graves et al., 2016) adds dynamic allocation and temporal linking. The Neural GPU (Kaiser and Sutskever, 2016) learns algorithms on a two-dimensional cellular grid, the Neural Programmer-Interpreter (Reed and de Freitas, 2016) composes learned subroutines with an explicit call stack, and differentiable Forth (Bošnjak et al., 2017) fills learnable holes in a program sketch.

We reuse the NTM's addressing wholesale for one component of the memory hierarchy (Section 3.8.2) and cite it as such. The difference is everywhere else. In an NTM the *controller* is a neural network and the computational primitives are whatever that network learns to approximate. Here the primitives are exact: `ADD` is a differentiable ripple-carry adder over soft bits, not a learned approximation to addition, and it is bit-exact against three independent discrete implementations at $\tau \to 0$ (Section 4). Consequently the artefact a training run would produce is not a set of network weights that must be carried along at inference; it is a sequence of instructions that runs on an ordinary virtual machine at $\mathcal{O}(1)$ host operations apiece. Differentiable Forth is the closest prior work in this respect, but its holes sit inside a hand-written Forth program in a small stack language, whereas here the entire machine — cache, bus, disk, framebuffer — is the differentiable object and the language is a 32-opcode load/store instruction set.

2.2 Relaxing Discrete Choices

Every discrete decision in this machine is relaxed by the same two devices: the temperature-scaled softmax and the temperature-scaled sigmoid. These are the Gumbel-softmax / concrete relaxation (Jang et al., 2017; Maddison et al., 2017) without the Gumbel noise, since the categorical variables here are optimised directly rather than sampled. Below the temperature at which the pathwise gradient becomes numerically useless, the straight-through estimator (Bengio et al., 2013) carries signal past the discretisation. The operand order of that estimator is load-bearing and is the subject of one of the defects in Section 4.4.

2.3 Program Synthesis

Enumerative search is complete and exponential in program length. Constraint solving in the counterexample-guided style (Solar-Lezama et al., 2006) scales with the specification but requires one to exist. Type-directed synthesis (Polikarpova et al., 2016) prunes aggressively but needs rich types, which machine code does not have. Neural-guided search (Balog et al., 2017) learns a prior over the functions of a domain-specific language and hands the result to a search procedure. Library learning (Ellis et al., 2021) discovers reusable abstractions by alternating between solving and compressing. Large language models generate programs as text (Chen et al., 2021; Li et al., 2022), which works well and inherits the cost of the model that produced them. Learned search has also produced genuinely new artefacts in adjacent spaces — faster matrix multiplication (Fawzi et al., 2022) and better sorting routines (Mankowitz et al., 2023).

The position this machine occupies is that a program is neither enumerated nor generated as text but *optimised directly* as a tensor, then crystallised into discrete instructions. Whether that is a good position is precisely the question this paper does not answer; Section 7.7 reports what happened when we tried, and Section 10 states the experiment that would settle it.

2.4 Grokking and the Motivating Hypothesis

The motivation for building a differentiable computer at all comes from the grokking literature. Networks trained far past the point of fitting their training data can undergo a delayed transition to generalisation (Power et al., 2022), and the solution on the far side of that transition is often a legible algorithm rather than a lookup table — a transformer trained on modular addition implements a discrete Fourier transform (Nanda et al., 2023). The transition itself has been characterised as one between a memorising and a generalising circuit, with the delay set by their relative efficiency (Liu et al., 2022; Varma et al., 2023). Separately, much of a large model's behaviour appears to be steerable within low-dimensional subspaces of its activations (Li et al., 2023; Sharma et al., 2024), which is at least suggestive that the load-bearing computation is small relative to the network that hosts it.

The hypothesis that follows — if the computation a network converges to is a small algorithm, it may be cheaper to represent it as one — is stated carefully in Section 9 and is not established by this paper. We mention it here only because it explains why one would build this machine rather than a better search procedure.

3 The Differentiable Machine

This section is the specification. Every operation below is implemented in JAX (Bradbury et al., 2018) as a pure function of arrays and is differentiable with respect to the program. The organising rule is stated once and applies everywhere: *a discrete choice becomes a probability mass, and conditional execution becomes multiplication.* Nothing branches.

3.1 System Overview

Figure 1 shows the machine. A CPU core holds the register file, the ALU, the program counter, the flag vector, and an L1 cache. A system bus arbitrates between four masters — CPU, DMA engine, GPU, and peripherals — by a softmax over priority-weighted requests. Hanging off the bus are a content-addressed memory in the style of an NTM, main RAM behind two-level page tables, a sector-granular disk, a raster GPU with its own memory and a framebuffer, and a bank of peripheral ring buffers.

The invariant that makes any of this worth doing is that at $\tau \to 0$ every soft selector becomes one-hot and the machine executes exact discrete semantics. That invariant is a checked property rather than a design intention, and Section 4 says how it is checked.

The machine is parameterised by a *profile*. All figures in this paper use the `paper` profile: $N_\mathrm{r} = 16$ registers of $W = 32$ bits, $A = 2^{16}$ instruction slots, $N_\mathrm{s} = 4096$ call-stack entries, a $N_\mathrm{c} = 256$-line cache of $L = 8$ words, $N_\mathrm{m} = 256$ NTM slots, 64 MB of RAM, a 64 MB disk at 4 KB sectors, 64 K words of GPU memory, a 640×480 RGB display, and six peripherals with 256-slot rings. A second `dev` profile shrinks the address space to 2^8 slots and RAM to 16 KB so that the whole machine fits on a laptop; it is what the conformance suite runs on, and it differs from `paper` in sizes only. No implementation may hard-code a size.

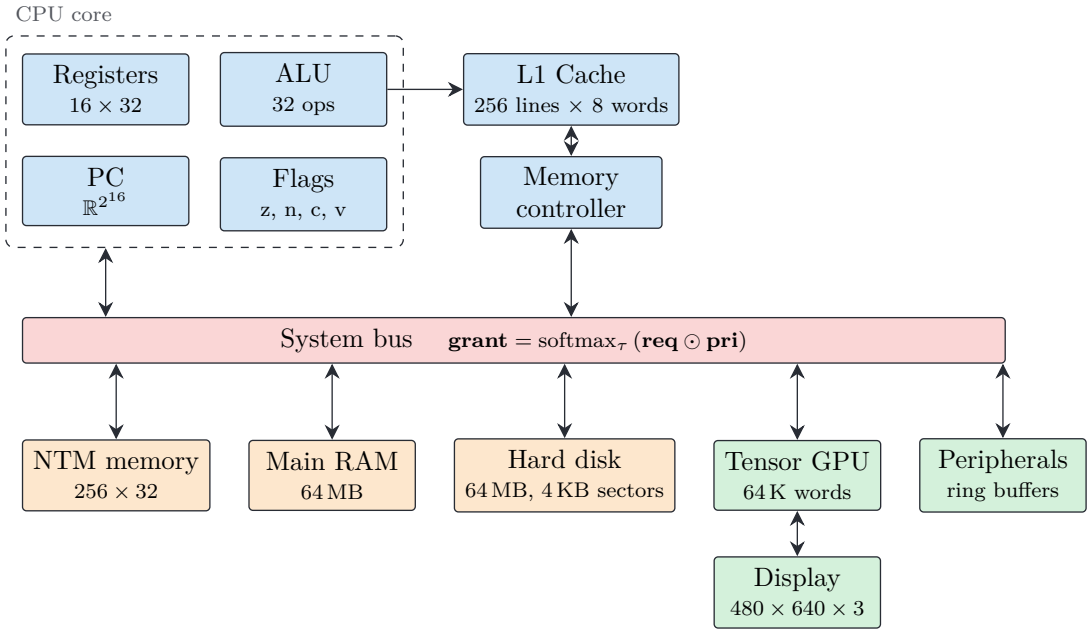

Figure 1: The machine on the `paper` profile. Every component is a differentiable tensor operation: the ALU computes all 32 operations and weights them by the opcode distribution, the program counter is a distribution over 2^{16} instruction addresses, and the bus arbitrates by a softmax over priority-weighted requests. At $\tau \to 0$ every selector becomes one-hot and the whole system executes exact discrete program semantics — which is a property checked against three independent implementations (Section 4), not a design intention.

3.2 Notation

Let $\tau > 0$ denote the temperature. The temperature-scaled softmax is

$$\operatorname{softmax}_\tau(\mathbf{x})_i = \frac{\exp(x_i/\tau)}{\sum_j \exp(x_j/\tau)}, \tag{3.1}$$

and for binary signals the soft indicator is

$$\sigma_\tau(x) = \operatorname{sigmoid}(x/\tau) = \frac{1}{1+\exp(-x/\tau)}. \tag{3.2}$$

Proposition 3.1 (Temperature limits)**.** *Fix $\mathbf{x} \in \mathbb{R}^n$ with a unique maximising index i^*. Then* $\operatorname{softmax}_\tau(\mathbf{x}) \to \operatorname{onehot}(i^*)$ *as $\tau \to 0^+$ and* $\operatorname{softmax}_\tau(\mathbf{x}) \to (1/n, \ldots, 1/n)$ *as $\tau \to \infty$. Likewise $\sigma_\tau(x) \to \mathbf{1}[x > 0]$ as $\tau \to 0^+$, with value $\frac{1}{2}$ at $x = 0$, and $\sigma_\tau(x) \to \frac{1}{2}$ as $\tau \to \infty$.*

Proof. Write $\operatorname{softmax}_\tau(\mathbf{x})_i = \left(\sum_j \exp((x_j - x_i)/\tau)\right)^{-1}$. For $i = i^*$ every exponent $x_j - x_{i^*}$ is negative except $j = i^*$, so each term vanishes as $\tau \to 0^+$ and the sum tends to 1; for $i \neq i^*$ the $j = i^*$ term diverges and the reciprocal tends to 0. As $\tau \to \infty$ every exponent tends to 0 and the sum to n. The sigmoid statements are the two-element case. The set of $\mathbf{x}$ with a tied maximum has Lebesgue measure zero. □

Two representational conventions are fixed here and assumed everywhere after. **Words are soft bit vectors** in $[0,1]^W$, least significant bit first, with element i carrying weight 2^i and element 31 the sign bit. Arithmetic is differentiable logic over these bits, not arithmetic on a scalar,

Symbol	Dimensions	Description
$\mathbf{R}^{(t)}$	$\mathbb{R}^{16\times 32}$	Register file, 16 registers of 32 soft bits
$\mathbf{PC}^{(t)}$	$\mathbb{R}^{2^{16}}$	Program counter as a distribution over instruction addresses
$\mathbf{F}^{(t)}$	$\mathbb{R}^{4}$	Flags: zero, negative, carry, overflow
$\mathbf{SP}^{(t)}$	$\mathbb{R}^{4096}$	Stack pointer as a soft position distribution
$\mathbf{S}^{(t)}$	$\mathbb{R}^{4096\times 2^{16}}$	Call stack: each slot a saved return-address distribution
$\mathbf{C}^{(t)}$	$\mathbb{R}^{256\times 8\times 32}$	L1 cache, 256 direct-mapped lines of 8 words
$\mathbf{M}^{(t)}$	$\mathbb{R}^{256\times 32}$	NTM-style content-addressed memory
$\mathbf{A}^{(t)}$	$\mathbb{R}^{2^{24}\times 32}$	Main RAM, 64 MB behind two-level page tables
$\mathbf{D}^{(t)}$	$\mathbb{R}^{16384\times 1024\times 32}$	Hard disk, 64 MB at 4 KB sector granularity
$\mathbf{G}^{(t)}$	$\mathbb{R}^{2^{16}\times 32}$	GPU memory, 64 K words
$\mathbf{V}^{(t)}$	$\mathbb{R}^{480\times 640\times 3}$	Display framebuffer, RGB in $[0,1]$
$\mathbf{B}^{(t)}$	$\mathbb{R}^{6\times 256\times 32}$	Peripheral ring buffers, with soft head and tail

Table 1: The machine state on the `paper` profile. Every component is an array of reals, so the entire state is one differentiable object and there is no discrete index anywhere in it — not in the program counter, not in the stack pointer, not in a cache tag. Three soft scalars accompany the tuple: a halted indicator, a cycle counter, and a fault vector in $\mathbb{R}^7$. The `dev` profile has the same fields at 2^8 instruction slots and 16 KB of RAM, which is what makes a whole machine state 3.58 MB there against 2.1 GB here (Section 9.1).

which is what makes `AND`, `XOR`, `SHL` and the carry and overflow flags simultaneously meaningful and differentiable. **Selectors are distributions**: register indices, opcodes, addresses, the stack pointer and the program counter are probability vectors, never integers.

3.3 Machine State

The complete machine state at time t is

$$S^{(t)} = (\mathbf{R}^{(t)}, \mathbf{PC}^{(t)}, \mathbf{F}^{(t)}, \mathbf{SP}^{(t)}, \mathbf{S}^{(t)}, \mathbf{C}^{(t)}, \mathbf{M}^{(t)}, \mathbf{A}^{(t)}, \mathbf{D}^{(t)}, \mathbf{G}^{(t)}, \mathbf{V}^{(t)}, \mathbf{B}^{(t)}), \tag{3.3}$$

together with a soft halted indicator, a cycle counter, and a fault vector in $\mathbb{R}^7$ whose components are the six fault kinds plus the cycle-limit stop. Table 1 gives dimensions.

Every component is an array of reals. There is no Python-level integer anywhere in the state, which is what allows the whole tuple to be registered as a JAX PyTree that survives `jit`, `vmap` and `scan`. A frozen dataclass holding one Python `int` would retrigger tracing on every step; a machine whose program counter is an integer cannot be differentiated with respect to where it is pointing.

3.4 Instruction Encoding

Each instruction is the tuple

$$\mathbf{I} = (\mathbf{op}, \mathbf{r}_d, \mathbf{r}_{s1}, \mathbf{r}_{s2}, \mathbf{imm}), \tag{3.4}$$

with $\mathbf{op} \in \mathbb{R}^{32}$ a distribution over opcodes, $\mathbf{r}_d, \mathbf{r}_{s1}, \mathbf{r}_{s2} \in \mathbb{R}^{16}$ register selectors, and $\mathbf{imm} \in [0,1]^{32}$ an immediate carried as soft bits. The widths sum to $N_{\text{op}} + 3N_{\text{r}} + W = 32 + 48 + 32 = 112$

parameters per instruction, and a program is one array $\mathbf{I}_{\text{mem}} \in \mathbb{R}^{A \times I}$. At low temperature each of the four distributions concentrates on a single index and the tuple is literally a machine instruction; an assembled program is one-hot in every selector field and is therefore exactly a $\tau \to 0$ machine program. Appendix A lists the 32 opcodes.

3.5 Instruction Fetch

Fetch is an inner product of the program counter distribution with the entire instruction memory,

$$\mathbf{I}^{(t)} = \left(\mathbf{PC}^{(t)}\right)^{\top} \mathbf{I}_{\text{mem}}. \tag{3.5}$$

When **PC** is one-hot this selects one row exactly. When it is spread the machine executes a blend — part of the instruction at address 7 and part of the one at address 12 — and both rows receive gradient. This is the operation that lets a gradient reach the program, and it is also by a wide margin the most expensive thing the machine does.

Remark 3.2 (Cost of soft fetch). Equation (3.5) costs $\mathcal{O}(AI)$ multiply–accumulates per instruction: $65{,}536 \times 112 \approx 7.3\,\text{M}$ on the `paper` profile, the great majority of the machine's per-cycle cost and about three orders of magnitude more than a hard indexed fetch. Section 3.12 breaks this down and Section 10 treats reducing it as the most concrete piece of unclaimed engineering in the design.

3.6 Register File

3.6.1 Soft Read

A read is a convex combination of all registers,

$$\text{RegRead}(\mathbf{R}, \mathbf{r}_{s1}) = \mathbf{r}_{s1}^{\top} \mathbf{R} \in \mathbb{R}^{W}, \tag{3.6}$$

exact whenever $\mathbf{r}_{s1}$ is one-hot.

3.6.2 Soft Write

A write blends the new value into every register in proportion to the destination selector,

$$\mathbf{R}' = \mathbf{R} + w\,\mathbf{r}_d \otimes (\mathbf{v} - \text{RegRead}(\mathbf{R}, \mathbf{r}_d)), \qquad \mathbf{R}'_j = (1 - w\,d_j)\,\mathbf{R}_j + w\,d_j\,\mathbf{v}, \tag{3.7}$$

where d_j is the j-th component of $\mathbf{r}_d$ and $w = \mathbf{op}^{\top}\mathbf{m}_{\text{write}}$ is the total opcode mass on instructions that write a register. This is the smallest clean instance of the general pattern. With $w = 1$ and $\mathbf{r}_d = \text{onehot}(j)$ the update sets register j to $\mathbf{v}$ and leaves every other register bit-for-bit untouched, because $d_k = 0$ for $k \neq j$ makes their blend coefficient exactly 1. With a spread $\mathbf{r}_d$ every register moves a little way toward $\mathbf{v}$. With $w = 0$ — the instruction is a `STORE` or a branch — the register file is computed and then multiplied by zero. The same device gates flag writeback, memory access, stack motion and device transactions: the mask $\mathbf{m}$ changes, the shape does not.

3.7 Arithmetic Logic Unit

A conventional ALU is a multiplexer over one selected operation. This one computes all N_{op} operations on every cycle and takes their opcode-weighted sum (Figure 2).

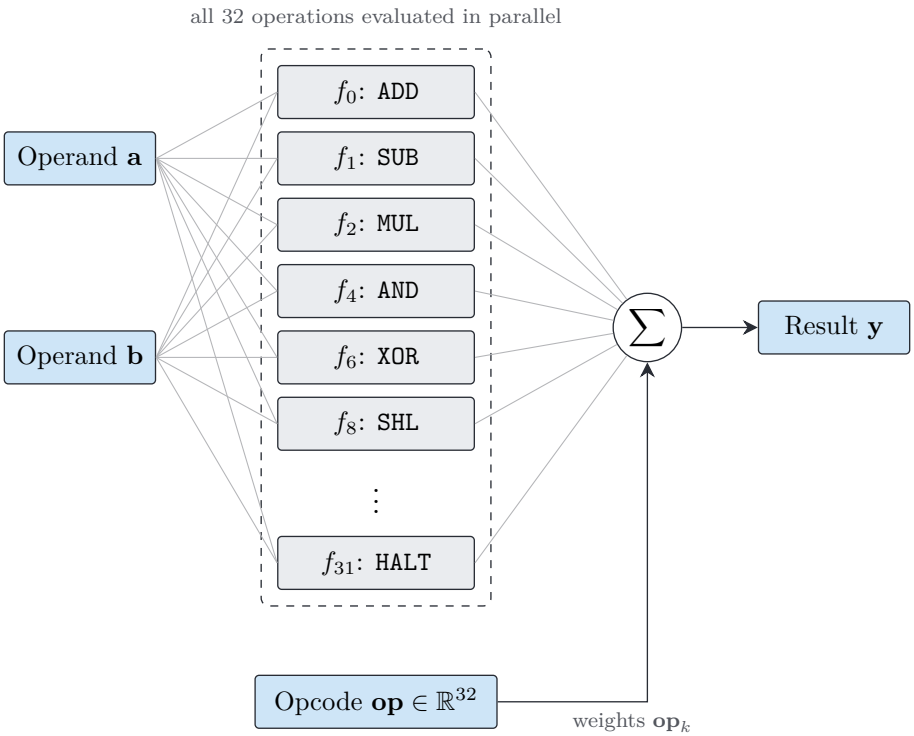

Figure 2: All 32 operations are evaluated on every cycle and the opcode distribution *weights* their outputs rather than *selecting* among them (Equation (3.10)). At $\tau \to 0$ the weighting collapses to ordinary operation selection, so the same datapath is a differentiable mixture during training and an exact multiplexer at deployment. The fan-in on the right is what makes an opcode learnable: because every f_k is computed, $\partial \mathcal{L}/\partial \mathbf{op}_k$ exists even for operations the current program does not use (Theorem 3.4).

3.7.1 Primitive Operations

The family $f_k : \mathbb{R}^W \times \mathbb{R}^W \to \mathbb{R}^W$ for $k = 0, \ldots, 31$ follows Appendix A. Operations 0–15 are combinational: `ADD` and `SUB` are differentiable ripple-carry chains over the soft bits; `MUL` is a shift-and-add network retaining the low W bits; `DIV` is stabilised as $\texttt{rs1}/(\texttt{rs2} + \varepsilon)$; `AND`, `OR`, `XOR` and `NOT` are the identities below; `SHL`, `SHR` and `SRA` are the interpolated permutations of Section 3.7.3; `MOV` and `LDI` are projections; `CMP` produces flags and no result; and `CMOVZ`, `CMOVN` blend two operands by a flag. Operations 16–31 address memory, the stack, control flow, and devices, and are defined in the subsections that own those datapaths.

3.7.2 Bitwise Operations

For soft bits $a_i, b_i \in [0, 1]$,

$$(a \wedge b)_i = a_i b_i, \quad (a \vee b)_i = a_i + b_i - a_i b_i, \quad (a \oplus b)_i = a_i + b_i - 2a_i b_i, \quad (\neg a)_i = 1 - a_i. \tag{3.8}$$

Proposition 3.3 (Bitwise correctness)**.** *On inputs in* $\{0, 1\}^W$*, Equation (3.8) reduces to the corresponding Boolean operations, and each expression is multilinear, hence the unique multilinear extension of that operation to* $[0, 1]^W$*.*

Proof. Each expression is degree one in each argument, so it is determined by its values at the four vertices $(0, 0), (0, 1), (1, 0), (1, 1)$, which are checked directly against the truth tables.

Uniqueness of the multilinear extension follows from the same counting. □

3.7.3 Shifts

A shift by a known amount is a permutation matrix: $\text{ShiftLeft}(\mathbf{a}, s) = \mathbf{P}_L(s)\,\mathbf{a}$ with $\mathbf{P}_L(s)_{ij} = 1$ iff $j = i - s$, and zero fill at the boundary. Two things must then be softened. First the shift amount is recovered from the low five bits of the second operand, $s = \sum_{i=0}^{4} 2^i b_i$, which is a real number rather than an integer. Second the permutation is interpolated over the 32 candidates,

$$\mathbf{P}_L(s) = \sum_{k=0}^{31} w_k(s)\,\mathbf{P}_L(k), \qquad w_k(s) = \text{softmax}_\tau\left(-|s-k|\right)_k . \tag{3.9}$$

This costs $\mathcal{O}(W^2)$, which is negligible at $W = 32$. `SRA` replaces the zero fill with the sign bit; `SHR` does not. A shift by 32 is a shift by 0, since only five bits are read.

3.7.4 Output

The ALU output is the opcode-weighted sum of every primitive,

$$\text{ALU}(\mathbf{op}, \mathbf{a}, \mathbf{b}) = \sum_{k=0}^{N_{\text{op}}-1} \mathbf{op}_k \cdot f_k(\mathbf{a}, \mathbf{b}). \tag{3.10}$$

Remark 3.4 (Gradient flow through the opcode). Equation (3.10) is the mechanism by which an opcode is learned rather than chosen. Because every f_k is evaluated, $\partial\mathcal{L}/\partial\mathbf{op}_k$ exists for every k and reports how the loss would change if operation k carried more weight — including for operations the current program does not use. A discrete multiplexer supplies no such quantity. Note also what this makes cheap: adding a thirty-third opcode costs $\mathcal{O}(W)$ per cycle, against the $\mathcal{O}(AI)$ of Theorem 3.2. Opcodes are nearly free; address space is not.

3.7.5 Flags

The four flags are soft indicators of the ALU result:

$$\begin{aligned} F_{\text{z}} &= \sigma_\tau(-\|\mathbf{y}\|_1/\varepsilon)\,, & F_{\text{c}} &= \sigma_\tau(\text{CarryOut}(\mathbf{a},\mathbf{b},\mathbf{op}))\,, \\ F_{\text{n}} &= \sigma_\tau(y_{31})\,, & F_{\text{v}} &= \sigma_\tau(\text{Overflow}(\mathbf{a},\mathbf{b},\mathbf{y},\mathbf{op}))\,. \end{aligned} \tag{3.11}$$

For addition, CarryOut is the final carry c_{32} of the ripple chain and Overflow is the soft form of $(a_{31} = b_{31}) \wedge (a_{31} \neq y_{31})$, both built from Equation (3.8) so that they are themselves differentiable. Carry on `SUB` and `CMP` follows the ARM convention: set when there is no borrow. Flags are written back by the same gated blend as Equation (3.7), with $w_{\text{flags}} = \mathbf{op}^\top \mathbf{m}_{\text{flags}}$, so operations that do not set flags leave them alone without a branch.

The overflow flag exists and is computed. What the instruction set does not provide is any way for a program to *read* it, which turns out to matter a great deal; Section 6.2 is that finding.

3.8 Memory Hierarchy

The hierarchy is deliberately not uniformly soft (Figure 3). Registers and cache stay fully differentiable. RAM and disk harden as τ falls. The reason is partly cost — a fully soft read of an N-cell array is $\mathcal{O}(N)$ — and partly regularisation: a program whose lower-level accesses must be nearly one-hot to work at all is a program whose access pattern survives extraction to real hardware.

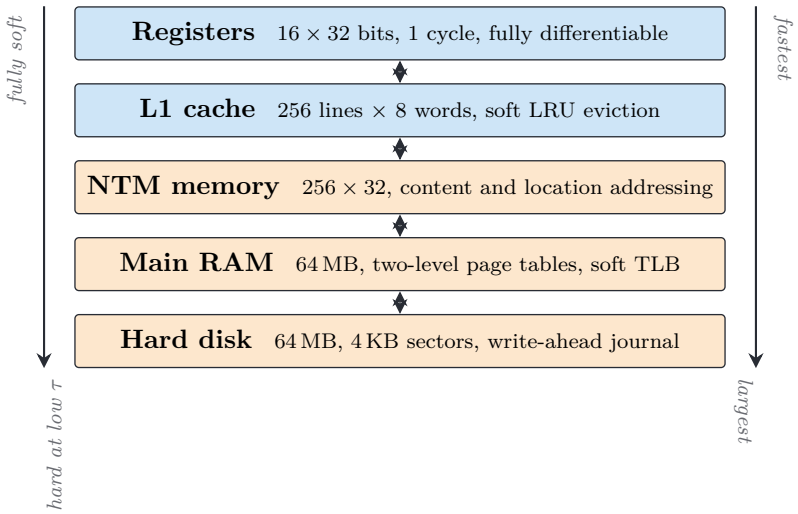

Figure 3: The hierarchy is deliberately not uniformly soft. Registers and cache stay fully differentiable; RAM and disk approach hard addressing as τ falls. The reason is twofold. A fully soft read of an N-cell array costs $\mathcal{O}(N)$, so soft access at the bottom of the hierarchy is unaffordable — which is also why the disk transfers whole 4 KB sectors rather than words. And a program whose lower-level accesses must be nearly one-hot to work at all is a program whose access pattern survives extraction to real hardware, so the hardening is a regulariser as well as a budget.

3.8.1 Cache

The cache is $N_{\mathrm{c}} = 256$ direct-mapped lines of $L = 8$ words, each carrying a tag, a valid bit, a dirty bit, and an LRU counter. An address decomposes into tag, index and offset, and the lookup scores every line at once:

$$h_i = v_i \exp\!\big(-\|\mathbf{tag}_i - \mathbf{t}\|^2/\tau\big)\exp\!\big(-|i - \mathbf{idx}|^2/\tau\big), \qquad \mathbf{hit} = \mathrm{softmax}_\tau\left(\mathbf{h}\right), \quad p_{\mathrm{hit}} = \sum_i h_i. \tag{3.12}$$

As τ falls the index term suppresses all but one line and the tag term becomes an equality test, recovering ordinary direct-mapped behaviour. A read is a doubly-soft selection — over lines by **hit**, then within the line by a soft offset — and the miss path, weighted by $1 - p_{\mathrm{hit}}$, falls through to the level below. A write updates the selected line and its dirty bit by the blend of Equation (3.7). Eviction chooses a victim by $\mathbf{evict} = \mathrm{softmax}_\tau\left(-\boldsymbol{\ell}\right)$, writes it back if dirty, and updates the counters.

The eviction policy is therefore not fixed. The LRU counters are ordinary differentiable state, and nothing prevents a training signal from reshaping what "least recently used" means for a particular workload.

3.8.2 Content-Addressed Memory

One level of the hierarchy is an NTM memory (Graves et al., 2014) of $N_{\mathrm{m}} = 256$ slots, imported wholesale. Content addressing produces $\mathbf{w}^c_i \propto \exp\!\big(\beta\,\mathrm{cosine}(\mathbf{k}, \mathbf{M}_i)\big)$; an interpolation gate blends this with the previous weighting; a convolutional shift and a sharpening exponent γ follow. Reads are $\mathrm{MemRead}(\mathbf{M}, \mathbf{w}) = \mathbf{w}^\top \mathbf{M}$ and writes are the usual erase-then-add, $\mathbf{M}'_i = \mathbf{M}_i(1 - w_i\mathbf{e}) + w_i\mathbf{a}$. We claim no novelty here; the component is present because content addressing is the one memory primitive that a soft machine gets for free and a discrete one does not have at all.

3.8.3 Main RAM and Virtual Memory

Main RAM is 64 MB behind two-level page tables: a 32-bit virtual address splits 10/10/12 into two table indices and an offset, the walk reads $\mathbf{PT}_1$ then $\mathbf{PT}_2$ by soft indexing, and a page table entry carries a frame number and permission bits. A 16-entry TLB reuses the cache lookup of Equation (3.12) verbatim. In practice the paging machinery is exercised by *tensor-os* (Section 5.5) only as far as the instruction set allows, since no instruction manipulates a page table base register.

3.8.4 Disk

The disk is 64 MB in 4 KB sectors, $S_{\text{sector}} = 1024$ words each, 16,384 sectors. Transfers are whole sectors, never words, and this is a design decision worth defending rather than an inherited convention: a soft sector selection costs $\mathcal{O}(N_{\text{d}})$, so word-granular soft access to the disk would be ruinous, and making the sector the unit of transfer teaches a program locality by making the alternative unaffordable. Writes go through a write-ahead journal, so a sector update is atomic under a cycle-limit stop.

3.9 Control Flow

3.9.1 Sequential Execution

Advancing the program counter is a cyclic rotation of a distribution,

$$\mathbf{PC}^{(t+1)} = \text{Shift}(\mathbf{PC}^{(t)}, +1), \qquad \text{Shift}(\mathbf{p}, k)_i = p_{(i-k) \bmod A}. \tag{3.13}$$

A rotation is a permutation: it is exact at every temperature, conserves probability mass, and has the permutation matrix as its Jacobian. There is no approximation anywhere in "go to the next instruction", which is a piece of luck the design leans on constantly — it means that error accumulating in the program counter comes from branches alone.

3.9.2 Conditional Branches

A branch blends the target against the sequential successor,

$$\mathbf{PC}^{(t+1)} = c \cdot \mathbf{target} + (1 - c)\ \text{Shift}(\mathbf{PC}^{(t)}, +1), \tag{3.14}$$

with the condition c a smooth polynomial in the flags. Writing $z = F_{\text{z}}$ and $n = F_{\text{n}}$, the six conditions are

$$\begin{aligned} &\texttt{JEQ} : z, && \texttt{JNE} : 1 - z, && \texttt{JLT} : n(1 - z), \\ &\texttt{JGE} : 1 - n, && \texttt{JLE} : n + z - nz, && \texttt{JGT} : (1 - n)(1 - z). \end{aligned} \tag{3.15}$$

These are not discretisations of a branch rule; they are the rule, and they are what all four implementations compute. Note what is absent: no condition mentions F_{v}. Section 6.2 is the consequence.

3.9.3 Subroutine Calls

`CALL` writes $\text{Shift}(\mathbf{PC}^{(t)}, +1)$ into the call stack at the soft position **SP**, rotates **SP** forward, and jumps. `RET` rotates **SP** back and restores **PC** from the stack. The stack is a circular buffer of $N_{\text{s}} = 4096$ slots and **SP** is a distribution over positions, so "push" is a blended write and

"increment the stack pointer" is a rotation, exactly as in Equation (3.7) and Equation (3.13). The stack is separate from RAM and is reachable only by these four instructions; no instruction can read **SP** into a register. Section 5.5 reports what that costs an operating system.

3.10 Bus and Memory-Mapped I/O

3.10.1 Arbitration

Four masters — CPU, DMA, GPU, peripherals, with priorities $3.0, 2.0, 1.0, 0.5$ — contend for the bus, and the grant is

$$\mathbf{grant} = \mathrm{softmax}_\tau\left(\mathbf{req} \odot \mathbf{pri}\right). \tag{3.16}$$

Because the priorities are softmax logits their scale matters as well as their order. Ungranted requests are held to the next cycle.

3.10.2 Address Decoding

A physical address is routed to a region by a product of sigmoid gates,

$$w_{\text{region}} = \sigma_\tau(\mathbf{addr} - \text{base}) \cdot \sigma_\tau(\text{base} + \text{size} - \mathbf{addr}), \tag{3.17}$$

normalised across regions. The seven region bases in Appendix B are spaced far more widely than the regions are large; that spacing is a design choice, not an accident, because it keeps the gates well separated at moderate temperature, where an address is a blurred number and adjacent regions would otherwise bleed into one another.

3.11 Devices

3.11.1 GPU

The GPU is a raster device with its own 64 K-word memory and six commands — `NOP`, `CLEAR`, `PLOT`, `FILL_RECT`, `BLIT`, `FLUSH` — selected by the immediate field of a `GPU` instruction (Appendix C). As with the ALU, all six execute and the six resulting device states are blended by the soft selector, which is what lets a gradient tell a program that it wanted `FILL_RECT` and issued `PLOT`.

Two things follow that are easy to miss. The GPU operations are natively differentiable, so gradient can in principle flow through dense tensor work invoked from inside a program — this is what the hybrid design of Section 9 would eventually need. And `FILL_RECT` paints an arbitrary rectangle in a single instruction, so its cost is independent of the area it covers; Section 6.4 reports what that does to the shape of programs written for this machine.

3.11.2 Peripherals and Display

Six peripheral ports — console, keyboard, mouse, timer, entropy source, disk controller — are backed by ring buffers $(\mathbf{d}, \mathbf{head}, \mathbf{tail})$ whose head and tail are soft position distributions. Enqueue is a blended write followed by a rotation of **tail**; dequeue is the mirror. The predicates are soft: $\text{full} = \sigma_\tau\left(\mathbf{head}^\top \mathrm{Shift}(\mathbf{tail}, +1) - \tfrac{1}{2}\right)$ and $\text{empty} = \sigma_\tau\left(\mathbf{head}^\top \mathbf{tail} - \tfrac{1}{2}\right)$. Reading an empty ring returns the sentinel `0xFFFFFFFF`.

The framebuffer is $\mathbf{V} \in \mathbb{R}^{480 \times 640 \times 3}$ with components in $[0, 1]$. A memory-mapped write at byte offset o from the framebuffer base decomposes as $y = o \div 3W$, $x = (o \bmod 3W) \div 3$, $c = o \bmod 3$. Section 5.5 drives every pixel of a windowing desktop through this surface.

Component	Complexity	Multiply–accumulates	Share
Instruction fetch	$\mathcal{O}(A \cdot I)$	7,340,032	98.1%
Cache lookup and read	$\mathcal{O}(N_c \cdot L \cdot W)$	65,536	0.9%
PC update	$\mathcal{O}(A)$	65,536	0.9%
Content-addressed memory	$\mathcal{O}(N_m \cdot W)$	8,192	0.1%
ALU, all 32 operations	$\mathcal{O}(N_{op} \cdot W)$	1,024	$< 0.1\%$
Register read ($\times 2$)	$\mathcal{O}(N_r \cdot W)$	1,024	$< 0.1\%$
Register write	$\mathcal{O}(N_r \cdot W)$	512	$< 0.1\%$
Total		$\approx 7.48\,\mathrm{M}$	

Table 2: Per-instruction cost on the `paper` profile, counting multiply–accumulates. Soft instruction fetch — attention over the whole 2^{16}-slot address space — dominates everything else by two orders of magnitude, and is roughly a thousand times the cost of a hard indexed fetch. The consequence is that program length is nearly free and address space is not, which inverts the usual intuition; the `dev` profile's 2^8 slots make fetch $256\times$ cheaper and change nothing conceptual. Reducing this term — hierarchical instruction addressing, sparse attention over the program — is the single highest-value optimisation available (Section 10.3).

3.12 Cost Model

Table 2 gives the per-instruction cost on the `paper` profile. One row dominates. Soft instruction fetch is $\mathcal{O}(AI)$ and everything else together is under two percent of it, which has a consequence that runs against every intuition carried over from real hardware: *program length is nearly free and address space is not.* Adding instructions to a program costs nothing per cycle, because the fetch already touches all A rows. Enlarging `address_bits` costs every cycle of every program, which is why the configuration validator refuses more than 24 of them.

4 Verification: Four Implementations of One Machine

The premise of Section 3 is that the soft machine and a discrete machine are the same machine. That is a falsifiable statement about two artefacts, and it deserves to be checked rather than argued. This section describes how, and reports what checking it found — which turned out to be more interesting than the check itself.

4.1 Four Implementations

The instruction set and the machine profiles live in a language-neutral specification from which the opcode tables, the register maps and the memory map are generated into each language. Four independent implementations consume it, each for its own reason:

- `tcvm`, a reference virtual machine in dependency-free C11. It is the specification made executable — deliberately simple, readable in one sitting — and it is the tiebreaker when two implementations disagree.
- A Rust virtual machine, with the assembler, disassembler, debugger and command-line tooling, where memory safety and a usable interface matter more than throughput.
- A batched C++ simulator running at roughly 10^9 instructions per second, for generating data and rollouts where the JAX path would be needlessly soft and slow.
- The differentiable JAX machine of Section 3, evaluated at $\tau \to 0$.

The redundancy was not built for verification. Each implementation exists because it does a job the others do badly. Verification is the byproduct, and the byproduct is the part of this work most likely to transfer: the configuration that produced the defects below — one specification, several implementations, tests written per implementation — is not unusual.

4.2 The Conformance Suite

A conformance vector is a program, an initial state, and an expected final state:

```
{"name": "add_carry",
 "program": ["ldi r1, #0xFFFFFFFF", "ldi r2, #1", "add r3, r1, r2", "halt"],
 "initial": {"registers": {}},
 "expect":  {"registers": {"r3": 0}, "flags": {"carry": 1, "zero": 1},
             "cycles": 4}}
```

There are 59 vectors in four suites (Table 3). Every backend ships a runner that consumes them; a cross-language harness executes all backends against all vectors and asserts agreement, so adding an opcode means adding vectors and updating four implementations before the build is green.

What the suites pin is *values, not shapes*. The `devices` suite exists because the instruction set describes behaviour it does not make numerically exact — what a timer reads, what an empty ring returns, what the entropy source produces — and a specification that only constrains types permits four backends to be individually reasonable and mutually incompatible. `devices.json` therefore names the generator (`xorshift32`, shifts 13/17/5, seed `0x2545F491`) and pins its first three outputs.

The `faults` suite required a change to the vector format itself. A vector's default contract is that the program completes; there was no way to write down "this program is supposed to fail, in this way, after this many cycles". Fault behaviour was consequently the least-protected part of the instruction set, which is exactly the arrangement that had already allowed four backends to choose four random number generators. An optional `expect.fault` field inverts the default: the named fault kind becomes required and completing cleanly becomes the failure. Its cycle counts are its content — `pop_empty_stack_underflows` pins `cycles` to 1, where the reference implementation had reported 2.

4.3 The Equivalence Check

The differentiable machine is held to the same standard as the three discrete ones, plus one more. Running Section 3's machine at $\tau = 10^{-3}$ on an assembled program must reproduce the reference virtual machine's register file to within 10^{-4}. This is the single most important test in the repository, because it is what licenses the entire premise: a program found in the soft regime is a program in the discrete one only if the two regimes agree.

The sharpest instance is a recursive factorial. Compiled to 63 instructions, `factorial(5)` returns 120 in *exactly* 49 *cycles*, identically on `tcvm`, on the Rust virtual machine, on the C++ simulator, and on the JAX machine at $\tau = 10^{-3}$. Cycle-exact agreement is a much stronger statement than agreement on the answer. It requires the four implementations to agree about when a comparison sets flags, what `CALL` pushes, when the stack pointer moves, whether the halting instruction is counted, and how many times the loop body ran. The program exercises recursion five deep, conditional branching, stack frame construction and teardown, multiplication, comparison, `CALL`/`RET` with link register preservation, and callee-saved register discipline — and any disagreement in any of these shows up as a cycle count that is not 49.

Suite	File	Vectors	What it pins
ALU	`alu.json`	27	Arithmetic, logic, shifts, and the flags they produce
Control	`control.json`	17	Branches, loops, memory, the stack, subroutine linkage
Devices	`devices.json`	7	Peripheral ports, entropy source, timer, console — behaviour the ISA describes but does not make numerically exact
Faults	`faults.json`	8	Programs that are supposed to fail, and after how many cycles
Total		**59**	

Table 3: The 59 conformance vectors. All four implementations — the reference C virtual machine, the Rust virtual machine, the batched C++ simulator, and the differentiable JAX machine at $\tau \to 0$ — pass all 59, and a cross-language harness asserts pairwise agreement rather than only per-backend success. Every vector runs on the `dev` profile, so the suite is cheap enough to be a precondition for merging. The `faults` suite required an `expect.fault` field that inverts the runner's default contract; before it existed, fault semantics were the least-protected part of the instruction set (Section 4.2).

4.4 Six Divergences

Differential testing across the four implementations found six classes of defect. Each was invisible from inside any one implementation, because each implementation was internally consistent and passed every test written against it.

1. **Four different random number generators.** The reference used `xorshift32`, Rust `xorshift64*`, C++ `splitmix64`, and JAX a Numerical Recipes linear congruential generator. All four passed every one of the original 44 vectors, because random numbers look equally random until two traces are compared. This is the cleanest illustration of the general point: a property that no single implementation can be wrong about is a property that every implementation can be different about.
2. **A timer off-by-one, twice.** The vector written to pin down an ordering bug in the Rust cycle counter caught the identical bug in the C++ simulator on its first run. Two implementations had independently made the same mistake, which means the mistake is a property of the specification's ambiguity rather than of either author.
3. **Fault cycle accounting.** Rust and C++ did not count a faulting instruction; the reference rolled back only faults raised at fetch time, so it over-reported by one cycle on stack underflow, stack overflow, and unmapped access. This is the divergence that could not be expressed as a vector at all until the format grew `expect.fault`.
4. **An inverted straight-through estimator.** The correct surrogate is $\mathbf{s} + \mathrm{sg}(\mathbf{s} - \mathbf{h})$ — soft plus the stopped difference, giving hard values forward and soft gradients backward. The implementation had the operands the other way round, which evaluates to *soft* in the forward pass and routes the gradient through an arg max, and therefore has exactly zero gradient everywhere. The failure mode is silent: a training run would have annealed normally, crossed into crystallisation, and then stopped learning while continuing to produce plausible output and a believable-looking program. No output check finds this; comparing two implementations of the same surrogate does.
5. **An unencodable pseudo-operation.** `INC` was documented as `ADD rd, rd, #1`, but `ADD`

ignores its immediate: this instruction set has no add-immediate form. The specification now defines `INC` as `ADD rd, rd, rs` with a register holding one, and requires assemblers to reject a bare `INC` rather than invent an encoding.

6. **Two defects in the kernel extraction path**, found only by running extracted programs on the Rust virtual machine rather than trusting JAX. Extracted programs carried live values in operand fields their opcode never reads, which one assembler accepts through an escape hatch and the reference assembler rejects; and a non-zero exit status caused by a program *faulting* was misread as the assembler refusing the program. Section 7.6 describes the canonicalisation that fixes the first.

Defects 1–3 and 5 are specification defects wearing implementation clothing: in each case the specification permitted more than one behaviour and every implementation picked one. Defect 4 is different in kind, and is the reason this section precedes Section 7 rather than following it. It sat in the one component whose failure produces no wrong answers at all.

4.5 What This Does Not Cover

Agreement is only ever as strong as the vector set. Two device behaviours illustrate the gap. `FILL_RECT` consumes both of its register operands for geometry and therefore has no colour argument; the reference virtual machine takes the colour from GPU memory word 0 as a latch, while the JAX machine paints with the colour last latched by `CLEAR` or `PLOT`. `BLIT` has a similar disagreement about where its length comes from. Both readings are defensible, both are documented, and neither is pinned by any of the seven device vectors — which are about the timer, the entropy source, the rings and the console. The suite is a floor, not a ceiling, and every unpinned behaviour is a place where four implementations are free to differ.

The check also covers only the semantic half of the transfer question. It establishes that a program means the same thing on four implementations of this instruction set. It says nothing about whether a program that is correct under 32-bit integer semantics behaves the same in the presence of real timing, real cache behaviour, and real interrupt latency on hardware that is not this machine.

5 Software on the Machine

Two pieces of software were written to find out whether the machine of Section 3 is a computer or a demonstration. Both are hand-written and neither is learned; that is the point of them. A machine that runs five-instruction snippets proves nothing about a machine, and the failure modes that appear at 10^4 instructions — calling conventions, stack discipline, device latency, budget exhaustion — are the ones that tell you what the design actually is. Table 4 summarises both.

5.1 `tcc`: A Self-Hosting C Compiler

`tcc` is a C compiler written in the subset of C that it compiles. It is 8,265 lines and 206,974 bytes across nine `.c` files and five headers, and its pipeline is conventional: a lexer, a preprocessor with hideset-based macro expansion and a full integer constant-expression evaluator, a recursive-descent parser that performs type checking and scope resolution as it goes, an optimiser doing

	`tcc`	*tensor-os*
Source	8,265 lines C	2,674 lines C + 250 assembly
Instruction slots used	40,942	20,256
of 65,536 available	62.5%	30.9%
of which data image	—	1,054 (5%)
Representative run	1.26 M cycles	34,911,348 cycles
what it does	compiles a C program	scripted desktop session
Heap / frame stack	51,680 B / 3,636 B	17,440 B globals
Session detail, tensor-os		
Context switches		2,050
Frames presented		194
Glyphs rendered		13,233
GPU rectangles issued		102,608
System calls		9,758
Dropped input events		0
Pages leaked at halt		0

Table 4: Two independent hand-written programs in the 10^4-instruction range on the `paper` profile. `tcc`'s figure is the hosted image of stage 2 — the compiler running *on* the machine — whose output is byte-identical to the host-built compiler's (Section 5.3). *tensor-os*'s figures come from a scripted session driving keyboard and mouse input through a five-window desktop. Neither artefact was cut to fit: in both cases the binding constraint is cycles rather than slots (Section 6.4).

constant folding, dead code elimination and peephole rewriting, and a code generator with a spilling allocator over eight temporary registers.

The subset is real C and its boundaries are worth stating so the claim is not oversold. It supports `char` (signed and unsigned, packed one byte per byte), `int`, `unsigned`, pointers, arrays, `struct`, `enum`, `typedef` including self-referential structs, all control flow including `switch` with fall-through, all compound assignments, short-circuit evaluation, `sizeof`, casts, function definitions with mutual recursion and up to 32 arguments, brace-list and string initialisers with address constants, and a preprocessor with the conditional directives. It does not support floating point, `long` or `short`, `union`, bitfields, varargs, function pointers, `goto`, designated initialisers, compound literals, structs passed by value, variable-length arrays, token pasting, or separate compilation — it has no linker, which is why self-hosting feeds it a unity build. Each omitted construct is rejected with a diagnostic naming it rather than mis-compiled.

5.2 The Calling Convention

Arguments zero to three arrive in `r0`–`r3` and the rest on the stack; `r0` carries the return value; `r4`–`r11` are callee-saved expression temporaries; `r12` holds a constant zero established once at start-up; `r13`, `r14` and `r15` are the stack pointer, frame pointer and link register. Frames live in RAM and grow upward behind a 40-byte header — saved link register, saved frame pointer, and eight callee-saved slots — with parameter k at $fp + 40 + 4k$.

One detail is forced by Section 3.9.3 and is worth recording. The machine has two stacks: a hardware LIFO reachable only through `PUSH`, `POP`, `CALL` and `RET`, with a depth fixed by the profile, and RAM. Because no instruction can read the stack pointer into a register, the hardware stack cannot host call frames. `tcc` therefore keeps the hardware stack depth at exactly one at all times: a callee's first instruction pops the return address that `CALL` pushed, into the link

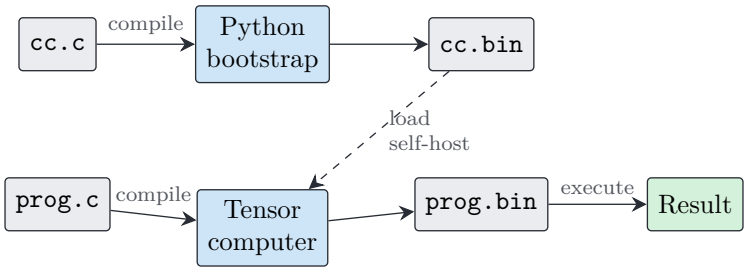

Figure 4: The bootstrap chain. A host build produces `tcc`'s binary; that binary, loaded onto the tensor computer, compiles a C program; the result executes on the same machine. The self-hosting claim is precisely that the dashed arrow and the second compile step involve the *same artefact* — and the measured claim is stronger still, since the assembly that comes back is byte-identical to the host-built compiler's output for the same input (Section 5.3). The fixed point, in which the machine compiles `tcc`'s own source, does not close, for the information-capacity reason in Section 5.3.1.

register, and everything else lives in a RAM frame. A program that recurses ten thousand deep still has one word on the hardware stack.

5.3 Bootstrap

Figure 4 shows the chain and Table 4 the measurements. Self-hosting is a sequence of stages, and the section states how far it goes rather than resting on the word alone.

Stage 0 — holds.
The host C compiler builds `tcc`.

Stage 1 — holds.
`tcc` compiles all of its own source, fed in as a single translation unit because it has no linker, into 39,786 tensor instructions, which assemble and run.

Stage 2 — holds.
`tcc` compiles a C program *while running on the machine*. The hosted image is 40,942 instructions inside the `paper` profile's 65,536 slots; it compiles a recursive factorial in 1.26 M cycles using 51,680 bytes of heap and 3,636 bytes of frame stack; and the assembly that comes back is *byte-identical* to the host-built compiler's. Running that assembly leaves 120 in `r0`. The regression suite checks this for all 19 test programs and for seven of `tcc`'s own nine source files, up to `sema.c` at 27,962 bytes.

Stage 3 — does not hold.
The fixed point: the machine compiling `tcc`'s entire source.

Stage 2 is the sentence worth writing. It is not that a compiler exists for this instruction set; it is that a real compiler runs on this machine and emits the same bytes as the host toolchain. The nine source files the hosted build includes are byte for byte the nine the host compiler builds — the compiler's source is not conditional on being hosted. Only the entry point differs, because the machine has no shell to pass `argv`.

5.3.1 Why the Fixed Point Does Not Close

Stage 3 fails on *information capacity, not compiler size*, and the result is more useful stated than omitted.

```c
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

int main(void) {
    return factorial(5);
}
```

Figure 5: The validation program. Compiled by the self-hosted compiler running on the tensor computer, it yields 63 instructions; executing them runs 194 instructions and leaves 120 in `r0`. The same source appears as a conformance vector, where it settles cycle-exact agreement across all four implementations (Section 4.3).

A program image is the only channel into the machine: the loader takes a program and nothing else, and the console input ring starts empty. The machine is Harvard, so a program carries no data section, and every initialised global and every string literal must be materialised into RAM at start-up by generated `LDI`/`STORE` pairs — one pair per non-zero word, which is two instruction slots for four bytes. Therefore 65,536 instruction slots carry at most $\lfloor 65{,}535/2 \rfloor \times 4 = 131{,}070$ bytes of arbitrary data *even if the compiler occupied none of them.* `tcc`'s source is 206,974 bytes. The measured image is 143,027 instructions, 2.2× the budget.

The consequence to state plainly is that no amount of shrinking the compiler closes this gap; the bound does not mention the compiler. Three changes would, and none is a change to instruction semantics: a data section in the program format, a disk-loading option on the loader, or eighteen address bits instead of sixteen. This is a fact about the machine's I/O surface, and it belongs in the paper as a bounded negative result rather than being left out.

Stage 2 concedes two further things, both about I/O rather than compilation. The input file travels inside the program image, over a read-only memory built by a host tool, with `fopen`, `fread`, `fseek`, `ftell` and `fclose` implemented over it — what is missing is a medium, not a file layer. And every output stream is the console port, so diagnostics land on the same stream as the generated assembly.

5.4 Validation

Figure 5 is the program the compiler compiles in stage 2, and "`factorial(5)` returns 120" on its own proves nothing. What makes it the right test is the list of machine features one result exercises: recursion five frames deep, conditional branching on a signed comparison, stack frame construction and teardown in RAM, multiplication, `CALL` and `RET` with link register preservation across a nested call, and callee-saved register discipline. A defect in any one of them changes the answer, the cycle count, or both — which is why the same program is also the sharpest conformance vector.

5.5 *tensor-os*

The second artefact exercises everything the compiler does not touch: the timer, the interrupt controller, the keyboard and mouse rings, the disk, the GPU and the framebuffer. *tensor-os* boots on the machine, identically on the C and Rust backends. It exists in two builds sharing a machine, a memory map and their first seven system call numbers: an assembly kernel of 1,853

instructions that boots and halts cleanly in 303,644 cycles, and a larger C build that adds the graphical stack. The figures below are the C build.

Its kernel is a round-robin scheduler over eight task slots, an interrupt dispatcher reading the pending and mask registers through a vector table, a bitmap page allocator with a real free path, a bump heap, four device drivers, and thirteen system calls. Above that sits a graphical stack written in C — 2,674 lines of it, against 250 lines of assembly for boot and the context switch: a damage-tracking compositor with z-order and a software cursor, a retained-mode toolkit of seven widget kinds (label, button, text field, checkbox, list, scrollbar, colour swatch), a 5×7 bitmap font, and five applications — a clock, a text editor that saves to and loads from a disk sector, a file browser, a paint canvas, and a task manager that reads the other tasks' control blocks live. It occupies 20,256 of 65,536 instruction slots, of which 1,054 — five percent — are not code but the data image, materialised by the `LDI/ STORE` pairs of Section 5.3.1.

A scripted session driving keyboard and mouse input through the desktop runs 34,911,348 cycles, performs 2,050 context switches, presents 194 frames, renders 13,233 glyphs, issues 102,608 GPU rectangles and 9,758 system calls, drops zero input events, and returns every allocated page before halting. The compositor never draws an occluded pixel: there is no back buffer and there cannot usefully be one, since a full screen is 921,600 bytes against 262,144 bytes of GPU memory, so the back buffer is the draw list and raising a window is a repaint rather than a copy.

5.5.1 Preemption, and What the ISA Costs It

The scheduling is genuinely preemptive but constrained, and the constraint is architectural. The instruction set has no trap instruction and no interrupt delivery, so userland reaches the kernel by calling a preemption point at loop back edges. What makes this preemption rather than cooperation is that the timer decides and the task cannot decline: the call transfers control unconditionally, and the scheduler returns to a different task if the slice is spent. A task that never reaches a back edge is a task that never yields, which is the honest limit of the arrangement.

Two further limits follow directly from Section 3. Carry and overflow are not saved across a context switch, because no instruction can move them into a register. And per-task call stacks are impossible, because the stack pointer is unreadable, so the hardware stack cannot be saved and restored — the same fact that forces `tcc`'s one-deep discipline in Section 5.2. A trap instruction and a readable stack pointer are the two additions that would lift both, and neither changes the semantics of any existing instruction.

6 Cost Model and Architectural Findings

Writing 60,000 instructions of software for a machine teaches you things about it that specifying the machine does not. This section reports the measured behaviour of the discrete implementations and three findings that the construction forced — one about the instruction set, one about the machine's I/O surface, and one about its cost model. All three share a shape, which Section 6.5 names.

6.1 Measured Behaviour

The discrete implementations run at conventional speeds: the batched C++ simulator sustains roughly 10^9 instructions per second, and a three-instruction loop executed 100,000 times on the reference virtual machine takes 300,005 cycles — three per iteration plus the entry and the halt,

which is the arithmetic a reader should be able to do in their head and get right. Twenty-five assembly programs, covering iterative and recursive arithmetic, sorting, searching, string and memory routines, and raster graphics, execute cycle-identically on the C and Rust backends and are checked against declared final register and memory states. The native test tree is 126 tests passing, and the reference virtual machine builds warning- free under two compilers and runs clean under the address and undefined-behaviour sanitisers. *tensor-os* boots and halts cleanly in 303,644 cycles on it, and one cross-backend discrepancy survives in that run: the low word of the timer reads 158,304 against Rust's 158,303 at the same point — a live instance of Section 4.5's point that the suite is a floor.

The differentiable path is slower by the margin Table 2 predicts. On the `dev` profile with a batch of eight and twelve machine steps, a gradient step costs 0.45 s at a peak resident set of 3.43 GB with rematerialisation enabled on the execution scan; without it the same configuration reaches 8.77 GB and runs out of memory. Rematerialisation cost nothing in wall clock here — it *saved* 60 seconds of compile time, because the un-checkpointed backward pass is a much larger program for the compiler to handle.

6.2 Finding 1: The ISA Cannot Express a Signed Comparison

Equation (3.15) builds every conditional branch from the zero and negative flags alone. `JGE` is $1 - n$ and `JLT` is $n(1 - z)$. There is no $N \neq V$ form, no branch on overflow, and no instruction that can read the overflow flag into a register — Section 3.7.5 computes F_{v} faithfully and then offers a program no way to see it.

The consequence is that `CMP a, b` followed by `JGE` decides $a \geq b$ only while $a - b$ cannot overflow. It is wrong on roughly a quarter of all 32-bit input pairs. The verification is a single line: $\max(\texttt{0x7FFFFFFF}, -1)$ returns -1, halting cleanly, with a plausible answer.

The repair is the standard identity. For $d = a - b$, the subtraction overflows exactly when $\operatorname{sgn}\big((a \oplus b) \wedge (a \oplus d)\big)$, and signed $a < b$ is $N \neq V$; so the sign bit of $d \oplus \big((a \oplus b) \wedge (a \oplus d)\big)$ is the comparison at any width. `XOR` sets flags, so an ordinary `JGE` still branches on it:

```
      sub  r3, r1, r2     ; d = a - b, may overflow
      xor  r4, r1, r2     ; a ^ b
      xor  r5, r1, r3     ; a ^ d
      and  r4, r4, r5     ; sign bit = overflow of a - b
      xor  r3, r3, r4     ; sign bit = (a < b), signed, at any width
      jge  done
      mov  r1, r2
  done:
      halt
```

Eight instructions instead of four, verified across 128 boundary and random cases with zero mismatches. `tcc` emits the same five-instruction comparison sequence for every ordering operator, compiles $\leq$ as $\neg(b < a)$ because the zero flag afterwards belongs to the exclusive-or'd word rather than to $a - b$, and biases both operands by 2^{31} for the unsigned case.

Three things about this finding are worth more than the fix.

First, it appeared *independently in three places*: in `tcc`'s code generator, via a calling-convention document that was itself wrong; in a listing in an earlier draft of this paper; and — most damagingly — in the reference implementations of `maximum` and `minimum` in the evaluation harness. Those references are ground truth for every evaluation number. A wrong reference does not fail. It silently redefines correct, and any accuracy figure computed against it is a figure

about the wrong task.

Second, the reason it is easy to miss is that the wrong program *halts cleanly with a plausible answer*. There is no fault, no exception, and no range in which the answer is obviously garbage — it is simply the other operand, about a quarter of the time, on inputs a uniform sampler almost never draws.

Third, and most interesting for the rest of this paper: this is a case where the machine's own instruction set constrains what the optimal program can be. The optimal signed maximum on this machine is eight instructions, not four, and any claim that a synthesised program "matches the optimum" is a claim measured against that number. A learned program would have to discover the same constraint, and would have no reason to be told about it.

6.3 Finding 2: Information Capacity, Not Program Size

Section 5.3.1 states the bound: because the machine is Harvard and a program image is the only channel into it, initialised data costs an `LDI`/`STORE` pair per word, so 65,536 instruction slots carry at most 131,070 bytes of arbitrary data even with a zero-size compiler. What makes this a finding rather than a limitation is that it is the wrong quantity from the perspective anyone would start with. The natural question about a self-hosting compiler is whether the compiler fits. It does: 40,942 of 65,536 slots, with room to spare. The binding quantity is how much *information* the machine can be handed, and that is a property of the I/O surface, not of the program.

The same fact prices *tensor-os*'s data image at 1,054 slots, five percent of its budget, for a font, a string table and a set of initialised globals that a machine with a data section would carry for free.

6.4 Finding 3: Area Is Free and Text Is Expensive

`FILL_RECT` paints an arbitrary rectangle in one instruction. A four-pixel fill and a 400×300 fill cost the same. This inverts the cost model every raster graphics technique is designed around, where filling area is the expensive thing and one optimises by touching fewer pixels.

The measured consequence is that a compositor on this machine is *glyph-bound*: about 1,300 cycles per character, measured during the first full-screen composite and counting everything the compositor does around it. The scripted session of Table 4 spends the great majority of its 34.9 M cycles on its 13,233 glyphs. Nothing in *tensor-os* was ever cut to fit the instruction budget — at 30.9% of it, slots were never the question — and every optimisation that mattered was about text: drawing glyph rows as runs of ink so that five lit pixels are one fill rather than five plots, clamping clip rectangles inline rather than calling a helper that costs eighteen instructions to compute a maximum four times per fill, and setting the colour latch once for glyphs entirely inside the clip.

Two scheduling changes in the same system make the point from the other side. Bounding the damage-rectangle merge — merging only when the union is barely larger than the two rectangles combined — took a repaint from 800,000 cycles to 10,000, and the whole session from 104.7 M cycles to 88.6 M. Having idle tasks yield their remaining slice, at a cost of one context switch of about 120 instructions, took it to 29.5 M and took the latency from a mouse click to pixels from five million cycles to three hundred thousand. A cost model that is unfamiliar in one direction is unfamiliar in the other: the wins are where a raster programmer would not look for them.

6.5 What the Three Have in Common

Each finding is a place where the machine's own structure decides what a correct or efficient program looks like, and in each case the structure was invisible until something real was written. Section 6.2 is the instruction set deciding that the optimal comparison is eight instructions. Section 6.3 is the I/O surface deciding that a compiler that fits cannot be fed. Section 6.4 is a device command deciding that a graphical system is bound by text rather than by area. Theorem 3.2 is the fourth instance and the largest: soft instruction fetch deciding that program length is free and address space is not, which is the reverse of the constraint every programmer carries in from real hardware.

This matters for the synthesis question of Section 7 in a specific way. A learned program is optimised against this cost model, not the one a human would assume, and none of these four facts is written down anywhere the optimiser can read. They are what the search would have to discover.

7 Training: Method, Implementation, and Current State

Section 3 makes the machine differentiable. This section is about why that is not sufficient, what the training method does about it, and what happened when we ran it. The last subsection is the important one and is written last on purpose.

7.1 The Core Difficulty

The loss landscape over programs does not resemble the landscapes deep learning is tuned for. It has vast plateaus, because the overwhelming majority of programs fail identically and their losses are therefore equal; narrow ridges, because a program correct on some inputs is one instruction from a program correct on none; isolated peaks, because correct programs are sparse; and deceptive basins, because a program that exploits the training distribution scores well until the distribution moves. The governing fact is that one changed opcode changes the semantics entirely. Relaxation buys a continuous path between two programs; it does not make the space between them meaningful.

7.2 Temperature Dynamics

The temperature is the exploration knob. The shipped schedule is exponential, $\tau(t) = \tau_0 \exp(-\lambda t)$ from $\tau_0 = 10$ to $\tau_{\min} = 0.01$ over 1,000 warm-up, 8,000 annealing and 1,000 crystallisation steps (Figure 6, Appendix D).

7.2.1 Where the Gradient Actually Dies

An earlier version of this work stated that gradients vanish below a fixed temperature, and repeated the figure $\tau = 0.1$ as though it were a constant. That was wrong, and correcting it is more useful than quietly restating it.

The original measurement was real but narrow: on a six-step program addressing a full 256-slot space, 209 of 28,672 instruction-memory entries carry non-zero gradient at $\tau = 1$ and exactly zero at $\tau = 0.1$. On a different object the curve is different. Table 5 measures a four-instruction kernel — 448 parameters, batch 8, random initialisation — inside a frozen host program, and the gradient norm spans five orders of magnitude and is *not monotone in* τ. It peaks near $\tau = 1$

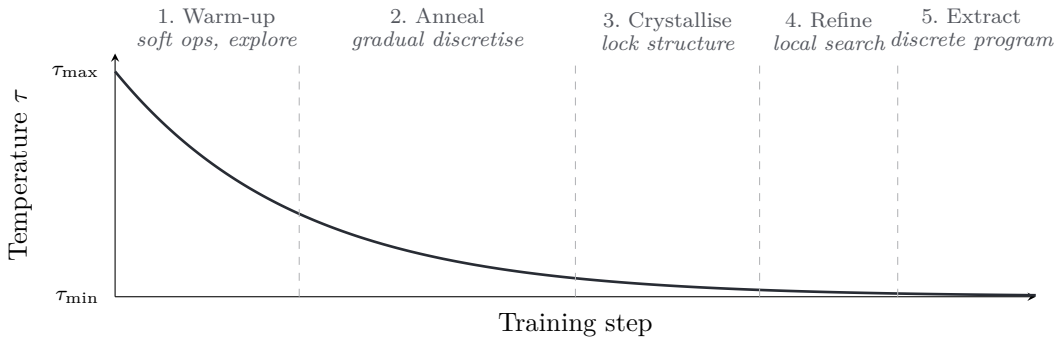

Figure 6: The five phases, over the shipped exponential schedule from $\tau_{\max} = 10$ to $\tau_{\min} = 0.01$ (Appendix D.2). Training moves from exploration in soft program space to exploitation of discrete structure. Two things the curve does not show: phases 4 and 5 are not gradient-based at all — the temperature continues to fall but the optimiser has been replaced by discrete local search — and the *backward* pass is held at a temperature floor of 0.05 throughout phase 3, without which crystallisation silently ends learning (Section 7.3). Where the pathwise gradient actually stops being usable is program-dependent and non-monotone in τ, so the phase boundaries are nominal rather than principled (Table 5).

τ	10.0	3.0	1.0	0.3	0.1	0.03	0.01
$\|\nabla_{\mathbf{I}_{\text{mem}}} \mathcal{L}\|$	2.21e-05	6.72e-03	**9.93e-01**	2.31e-01	5.29e-01	2.34e-04	9.29e-06
Entries $> 10^{-8}$ (of 448)	439	442	443	441	441	439	439

Table 5: Gradient norm with respect to instruction memory, measured on a four-instruction kernel (448 parameters, batch 8, random initialisation) inside a frozen host program. The norm spans five orders of magnitude and is *not monotone in* τ: it peaks near $\tau = 1$ and collapses in both directions, because at high temperature every selector is a uniform blur and nothing prefers anything. The second row is the diagnostic that matters — the *support* barely moves across the whole sweep, so what dies is magnitude, not coverage. A different program gives a different curve; on a six-step program over a full 256-address space, 209 of 28,672 entries are non-zero at $\tau = 1$ and exactly zero at $\tau = 0.1$. Neither is a constant, and an earlier claim in this work that gradients vanish below a fixed temperature was wrong (Section 7.2.1).

and falls off in both directions. The hot end is not a regime of abundant gradient: at high temperature every selector is a uniform blur and nothing prefers anything, so the norm is small there too.

Note also what does *not* change across the sweep. The count of entries carrying gradient above 10^{-8} moves only from 439 to 443 of 448. It is the magnitude that dies, not the support — which is why an optimiser with per-parameter scaling can be misled about how much signal is present.

The structural claim the schedule actually rests on is weaker than the one we withdrew, and is still sufficient: for any given program there is a temperature below which the pathwise gradient stops being usable, the annealing phase is where the gradient does its work, and everything colder is the straight-through path plus discrete search. The practical instruction is to measure per task and not to assume; the repository ships a script that re-measures for any kernel.

7.3 The Five Phases

Warm-up holds $\tau \gg 1$. The program is a differentiable mixture of all programs, gradient reaches every path, and the object being optimised corresponds to no valid discrete program at

all. **Annealing** decays the temperature; distributions peak, credit assignment sharpens as fewer operations contribute, and the learning rate decays alongside. **Crystallisation** holds $\tau \ll 1$: selectors are effectively one-hot, pathwise gradients are numerically dead, and the straight-through estimator carries signal past the discretisation. **Refinement** abandons gradients entirely — extract by arg max, then hill-climb by per-position substitution over the opcode and register fields, with restarts. Some improvements are not reachable along continuous paths at all: insertion and deletion of instructions are moves a fixed-width logit array cannot express, and they exist only on the discrete side. **Extraction** emits discrete machine code and verifies it (Section 7.6).

The straight-through surrogate is $\mathbf{s} + \mathrm{sg}(\mathbf{s} - \mathbf{h})$: hard forward, soft backward. Written with the operands reversed it evaluates to *soft* forward and routes the gradient through an arg max, giving it exactly zero gradient everywhere. That inversion was present in this repository and is defect 4 of Section 4.4. It is worth restating here because of where its symptom would have appeared: precisely at the crystallisation boundary this subsection describes. A run would have annealed normally and then flatlined, with no error and a believable-looking program. The shipped configuration additionally holds the *gradient* pass at a temperature floor of 0.05 even after the forward pass has hardened, for the same reason.

7.4 Credit Assignment and Curriculum

Gradient from a final loss must traverse the whole execution trace, so the method supplies denser signal in four ways. The reward is shaped rather than binary: a bitwise output distance carrying almost all the weight, a partial credit term for the fraction of output components correct, a small bonus for halting sooner, a penalty for out-of-bounds access and division by zero, and a term for halting naturally rather than by cutoff. The shipped weights are 1.0, 0.2, 0.05, 0.5 and 0.2 respectively; weight the auxiliary terms much more heavily and the optimiser writes a program that halts immediately. Auxiliary losses can supervise intermediate execution checkpoints, and an optional critic over machine states supplies per-step feedback. An outcome-supervised policy gradient is available and is off by default: gradient descent through the soft machine is the primary signal.

The curriculum (Figure 7) advances along four independent axes — program length, input size, task complexity, and which level of the memory hierarchy the task requires — promoting a stage when accuracy holds above a threshold across a window and backing off when it collapses. It is implemented and disabled by default.

Two further mechanisms are described in the design and *not* implemented, and we mark them rather than let the list read as uniformly available: a library of frozen subroutines grown across curriculum stages in the style of library learning (Ellis et al., 2021), and a two-level policy that chooses between emitting a primitive and calling a library entry. Both are the natural answer to credit assignment over long horizons and neither has been built. Population-based training (Jaderberg et al., 2017) is implemented and off by default, since it costs N times the compute.

7.5 Pathological Programs

Programs that never halt yield no useful gradient, and are handled by a hard step cutoff with a penalty, a penalty for making no progress, and a reward for halting naturally. The degenerate optima the optimiser actually reaches are more interesting: returning a constant, returning the training-set mean, and returning an input unchanged. On the maximum task, returning

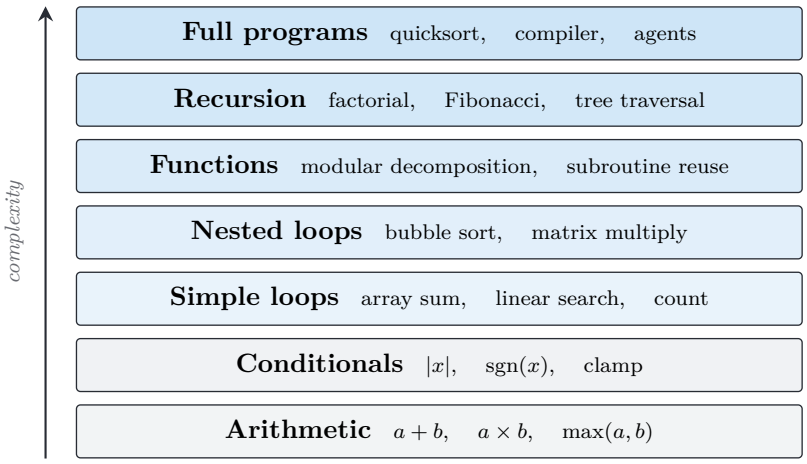

Figure 7: The curriculum ladder as designed. Each rung builds on the capabilities of the one below, and advancement is gated on accuracy sustained across a window rather than on a step count, with backoff when it collapses. The other three axes of Section 7.4 — program length, input size, and how far down the memory hierarchy the task reaches — vary *within* each rung. This is a design, not a result: the curriculum is implemented and disabled by default, and nothing in this paper has been trained past the bottom rung (Section 7.7).

either input unchanged already scores about 0.5, which sets the bar any reported score must clear before it means anything. The countermeasures are input distribution design, complexity regularisation, and held-out verification, and a trivial-solution detector is on by default.

7.6 What Runs End to End: The Kernel Path

The version of this pipeline that has been run end to end is the *kernel* version, and the distinction is the subject of Section 9. Instruction memory is one $\mathbb{R}^{A\times I}$ array; all of it except a small budget of rows is written as frozen machine code at a logit scale that decodes one-hot at every temperature, and the remaining rows are the optimiser's parameters. The host program runs at $\tau \to 0$ as ordinary machine code, and inside it sits a subroutine of a handful of instructions whose contents are learned.

Nine stages and fourteen checks verify this path, and each check prints the number behind it rather than the word PASS: that the hole is real and splicing touches nothing else; that gradients reach instruction memory; that they collapse as τ falls, non-monotonically; that the loss decreases; that crystallisation produces legal discrete instructions; that extraction round-trips against the soft machine; that the extracted program runs on the *discrete Rust virtual machine* and agrees with JAX; that it can be spliced into a larger hand-written program that still runs; and that the evolutionary loop never loses its incumbent. All fourteen pass.

One extraction detail is not obvious and broke the cross-check twice. A program that came out of a softmax has an opinion about all five fields of every instruction, *including the fields its opcode never reads*. An `ADD` carrying 32 immediate bits is the normal shape of an extracted kernel. Those bits are dead at $\tau \to 0$ but not harmless: one assembler accepts them through an escape hatch and the reference assembler rejects them. Extraction therefore canonicalises, zeroing every field the opcode ignores. That cannot change behaviour at $\tau \to 0$, and rather than take it on trust the round-trip check scores the canonical discrete program against the soft program it came from, example by example, so a canonicalisation that changed anything would

surface as a round-trip failure.

7.7 Current State of the Synthesis Question

Nothing in this repository synthesises a correct non-trivial program.

The budgets shipped here are toy-sized on purpose — a few minutes on a laptop, none of them a search run — and at those budgets the learned kernels are wrong. A representative run on the four-instruction maximum kernel, 120 gradient steps at batch 8, moves the mean loss from 0.804 over the first tenth of training to 0.192 over the last tenth; discrete local search then moves the score from 0.4970 to 0.4980 over 456 candidates. The crystallised program assembles, and it is not a maximum. It is four instructions that score 0.438 on a task where returning either input unchanged scores about 0.5, and on four of eight examples it faults on an unmapped load. At 30 steps the same script produces a three-instruction program that runs in five cycles, agrees with JAX on all four examples, and leaves 0 where the host expected 23. That is the pipeline working and the search not.

The genuinely strong result in that run is a different one. JAX and the Rust virtual machine agree on all eight examples *including the four that fault* — same fault kind, same address, same cycle. That is what establishes that the extracted object is machine code rather than a JAX artefact, and it is the check that would have caught a pipeline that produced plausible-looking output for the wrong reason.

Two further things are true and should be said in the same breath. First, every knob that separates this from a serious attempt is a configuration value, not a code change: gradient steps of 10^4–10^5 rather than 10^2, batch 64 rather than 8, eight to thirty-two random restarts rather than one, populations of hundreds rather than tens, and kernel budgets of tens to low hundreds of instructions. Second, the experiment that would settle the central methodological question — pure gradient against pure evolution against the hybrid, at accounted compute, on the same task — is written and deliberately unrun. Answering "does the hybrid beat either alone" on the compute available here would produce a number worse than silence. Section 10 treats it as the first thing to do.

8 Formal Properties

8.1 The Temperature-Parameterised Loss Family

Write θ for the program parameters — opcode logits, register selector logits, immediates — and $\mathcal{L}_\tau(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D}}\big[\ell(f_\tau(x;\theta), y)\big]$ for the loss of the machine run at temperature τ. Everything below is a statement about this family indexed by τ.

Theorem 8.1 (Continuity in temperature)**.** *Suppose the per-example loss ℓ is bounded and continuous, the machine is run for a fixed number of steps, and every primitive is continuous in its arguments. Then $\mathcal{L}_\tau(\theta)$ is jointly continuous on $(0,\infty)\times\mathbb{R}^d$, and for every θ whose selector logits have no ties, $\lim_{\tau\to 0^+}\mathcal{L}_\tau(\theta) = \mathcal{L}_0(\theta)$, the loss of the discrete program obtained by taking the* arg max *of each selector.*

Proof sketch. Equation (3.1) and Equation (3.2) are continuous in $\tau > 0$ and in their arguments; a fixed number of steps composes finitely many continuous maps; bounded convergence carries the pointwise limit under the expectation. The limit is Theorem 3.1, whose exceptional set — tied selector logits — has measure zero. Appendix E.1 gives the argument in full. □

This is the formal content of the claim that the soft machine and the discrete machine are the same machine. It is worth being clear about how little it proves on its own: it says the soft loss converges to the discrete loss, not that descending the former finds a minimum of the latter. Section 4 carries the empirical version of the same statement, and it is the stronger of the two, because it is bit-exact and holds for programs the theorem's hypotheses do not obviously cover.

Remark 8.2 (Non-convexity). Continuity buys nothing about convexity. Every discrete program of length n is a candidate local minimum in the $\tau \to 0$ limit, so the count of local minima grows as N_{op}^n at least. This is why a single run commits to a basin during warm-up and why multiple random restarts are not optional (Section 10).

Proposition 8.3 (Gradient scaling)**.** *The softmax Jacobian is* $\partial \mathbf{op}_{k'}/\partial z_k = \tau^{-1}\mathbf{op}_{k'}(\delta_{kk'} - \mathbf{op}_k)$. *Its magnitude therefore carries a factor* τ^{-1}*, while the factor* $\mathbf{op}_{k'}(\delta_{kk'} - \mathbf{op}_k)$ *tends to zero as the distribution saturates.*

The two factors pull in opposite directions, and which one wins is not settled by the algebra: the τ^{-1} growth is bounded below by how fast the selector saturates, which depends on the logits, which depends on the program and the data. This is the formal reason Table 5 is non-monotone, and the reason no fixed temperature can be quoted as the point where gradients die.

8.2 Expressiveness

Theorem 8.4 (Turing completeness in the limit)**.** *At* $\tau = 0$*, with the address space and RAM taken to be unbounded, the instruction set of Appendix A is Turing complete.*

Proof. The instruction set provides conditional branching on the result of a comparison, indexed load and store over an addressable memory, and integer addition and subtraction. These suffice to simulate a two-counter machine: represent each counter as a word in memory, increment and decrement by `ADD` and `SUB` against a register holding one, and test for zero with `CMP` and `JEQ`. Two-counter machines are universal, so the instruction set is. □

Two qualifications belong in the statement rather than in a footnote. First, the physical machine has 2^{16} instruction slots and 64 MB of RAM, so it is a linear bounded automaton, exactly as every real computer is; the theorem is about the instruction set, not about the configuration. Second, we do *not* rest this on Section 5. A self-hosting C compiler is evidence of practical expressiveness — that programs of realistic size and structure can be written and will run — and it is strong evidence for that. It is not a completeness proof, because the C subset `tcc` accepts is itself bounded and because "compiles a large program" and "simulates any Turing machine" are different claims. The simulation argument above carries the theorem; the compiler carries the practical claim.

8.3 Limits on Learnability

There are $(N_{\text{op}}N_{\text{r}}^3 2^W)^n$ syntactically distinct programs of length n, so identifying one from input/output examples needs $\Omega(n \log N_{\text{op}})$ bits and hence $\Omega(n)$ examples at any fixed output width. That counting bound fixes the units and is not the binding constraint. The binding constraint is that synthesis in general is not tractable for any method. Inverting a one-way function is a synthesis task — "write a program mapping this output to a preimage" — so if some efficient procedure synthesised arbitrary programs from examples, standard cryptographic

hardness assumptions would fail. We do not dress this as a theorem, because the honest version has no content beyond the observation that program synthesis contains arbitrary computation. Its consequence for the design is real, though, and it is the reason the method in Section 7 is hybrid rather than purely gradient-based: no relaxation makes an intractable search tractable, and the most a relaxation can do is change which easy instances are easy.

Theorem 8.5 (Description-length generalisation)**.** *Fix a prefix-free encoding of programs and let $|P|$ be the code length in bits of P. For a loss bounded in $[0, 1]$ and a sample of size m drawn i.i.d., with probability at least $1 - \delta$ simultaneously for all programs P,*

$$\mathcal{L}_{\text{test}}(P) \leq \mathcal{L}_{\text{train}}(P) + \sqrt{\frac{|P| \ln 2 + \ln(1/\delta)}{2m}}. \tag{8.1}$$

Proof. Assign each program prior mass $2^{-|P|}$, which sums to at most one by Kraft's inequality, apply Hoeffding's inequality at confidence $\delta 2^{-|P|}$ to each, and take a union bound. Appendix E.2. □

The contrast with a neural network is the point of quoting it. There the complexity term is a function of weight norms and is estimated; here it is literally the size of the emitted binary, and it is known exactly before the program is ever evaluated. Whether that handle is *useful* depends on whether short programs are found at all, which is Section 7.7's open question.

Conjecture 8.6 (Algorithmic programs generalise in length)**.** *A program implementing a size-independent algorithm — a loop whose trip count is read from the input — has test loss on inputs of size $m > n$ exceeding its loss at size n by $\varepsilon(m, n) \to 0$, while a program exploiting size-specific structure, such as an unrolled loop, does not.*

We state this as a conjecture and offer no evidence for it. It is the reason the curriculum of Section 7.4 varies input size within a stage rather than fixing it, and it is straightforwardly testable the moment anything on this machine learns a loop.

8.4 What Annealing Would Require

The annealing schedule rests on a statement we have not proved and should therefore state as an assumption rather than a theorem.

Assumption 8.7 (Basin persistence)**.** For a local minimum P^* of the discrete loss $\mathcal{L}_0$ there exists $\tau^* > 0$ such that for all $\tau < \tau^*$ the set of parameters whose arg max is P^* contains a local minimum of $\mathcal{L}_\tau$.

If this holds, discrete local minima lift to continuous ones once the temperature is low enough, and an annealing schedule slow enough to track the moving minimum ends in a discrete local minimum. If it fails — if some discrete optima have no soft counterpart at any positive temperature — then annealing systematically cannot reach them, and the gradient path is not merely inefficient but incomplete. We know of no proof either way, and the condition that would make a schedule provably sufficient (that the iterates maintain approximate stationarity relative to the temperature scale at every step) is not checkable online. It is a design guide, not a guarantee, and this section prefers to say so than to prove something weaker while implying something stronger.

9 The Hybrid Position

9.1 The Scale Argument

There is a straightforward reason the training method of Section 7 learns small kernels inside frozen host programs rather than whole programs end to end, and it is arithmetic rather than preference.

One `paper`-profile machine state is 2.1 GB, dominated by RAM and the framebuffer, both $\mathcal{O}\left(2^b\right)$ in their address widths. Reverse-mode automatic differentiation keeps one state per step, so an 8 GB accelerator affords roughly *three* steps of a backward pass. Against that: *tensor-os* boots in 303,644 cycles and a single compilation run is 10^6 to 10^7 instructions. Backpropagating through a real workload on this machine is infeasible by about six orders of magnitude, and no implementation effort closes a gap of that size — rematerialisation on the execution scan buys a constant factor, and the `dev` profile's 3.58 MB state buys another, but the deficit is 10^6.

The design that follows is hybrid: discrete infrastructure with small learned holes. Instruction memory is one array; nearly all of it is frozen machine code at a logit scale that decodes one-hot at every temperature, and a handful of rows are parameters. The compiler, the operating system and the graphical stack run at $\tau \to 0$ as ordinary machine code, and inside them sit subroutines of four to a hundred instructions whose contents are learned. This is tractable for the same reason the whole-program version is not: the differentiated object is small, the frozen host costs a forward pass and no backward one, and the credit assignment horizon is the length of the kernel rather than the length of the program.

This is the one part of the training story that runs end to end (Section 7.6), and Section 7.7 is honest about what it has produced so far, which is not a correct program. The claim being made here is narrower and survives that: whatever the eventual answer about gradients, *whole-program* end-to-end differentiation on a machine of this shape is not the route, and any method that works on it will be one that differentiates a small part of a large discrete computation.

9.2 The Motivating Hypothesis

The reason to want learned code at all — rather than a better neural policy — is a hypothesis about where computation is cheapest to represent.

Proposition 9.1 (Grokking hypothesis, informal)**.** *If extended training past the point of fitting yields sparse, legible circuits, then the computation those circuits perform may be more efficiently represented as imperative code than as the dense tensor operations that host it.*

The evidence for the antecedent is real but thin, and consists mostly of two observations: a transformer trained on modular addition converges to a discrete Fourier transform (Nanda et al., 2023), and the memorisation-to- generalisation transition looks like competition between two circuits of different efficiency (Varma et al., 2023). What code expresses natively and gradient descent tends to reach for only under pressure is the same short list every time — composition, recurrence, recursion, and explicit branching. Sorting is the standing example: compare, swap, repeat is three concepts and no amount of interpolation is a shorter description of it.

What would falsify the hypothesis, in the form this paper could test it, is stated in Section 10: if gradient annealing on this machine does no better than discrete search at the same accounted compute, the differentiability is elaborate scaffolding around a search that did not need it. What would support it is a learned program that is shorter than the network it replaces *and* generalises further, on a task where both are trained honestly. Neither experiment has been run here.

We are deliberately not tabulating a projected efficiency comparison against language-model agents. Such a table was in an earlier draft of this work. Every figure in its learned-program column was a projection, and a projection placed beside measured numbers reads as a measurement.

9.3 When Programs Should Win, and When They Should Not

Programs should be the better representation when the task has algorithmic structure, when compositionality matters, when generalisation across input size is required, when inference efficiency is paramount, or when the artefact must be traceable. Networks should be the better representation when pattern recognition dominates, when smooth interpolation is what is wanted, when data is abundant relative to structure, and when the task has no clean algorithmic description — which is most perception.

The position is therefore not that code replaces networks. It is that the boundary between them is currently drawn by what is differentiable rather than by what each is good at, and a machine on which both are differentiable moves the boundary somewhere it can be argued about. Whether it moves far is the open question, and Section 3.11.1's differentiable device commands are the mechanism by which it could: a learned program that calls into dense tensor work, with gradient flowing through the call.

10 What Remains

The list below is ordered by dependency and by cost, not by ambition. Each item is small enough to fail cheaply, and each has an outcome that would change what is worth doing next.

10.1 Answer the Methodological Question First

Run pure gradient annealing, pure discrete search, and the hybrid against each other on the same tasks at *accounted* compute — rollouts, meaning one program executed on one example — and report all three. The harness is written. What makes this first is that it is the cheapest experiment that can invalidate the premise: if annealing does no better than discrete search on the same budget, the differentiable machine is scaffolding around a search that did not need it, and every subsequent item changes.

A companion measurement costs almost nothing and should accompany it. Table 5 shows that where the gradient is useful is program-dependent and non-monotone; the schedule should be fitted per task from that curve rather than inherited.

10.2 Close the Scale Gap, or Locate It

Everything learned in this repository is at a program length of at most a few instructions. Everything the machine runs that was written by hand is three orders of magnitude larger: 40,942 instructions for the compiler, 20,256 for the operating system. Nothing has been trained anywhere between those two points, and the useful next measurement is not a success but a *ceiling*: the kernel length at which the method stops working, on a task whose correct answer is known. A curve of success rate against kernel budget, from four instructions to a few hundred, is a more informative result than any single learned program, and it is what would tell us whether hierarchical decomposition is worth building.

10.3 Reduce the Cost of Softness

Table 2 makes soft instruction fetch 98% of the machine's per-instruction cost and roughly a thousand times a hard indexed fetch. This is the most concrete unclaimed engineering in the design and it is self-contained: hierarchical instruction addressing — page tables for the program, so that fetch is attention over pages then over a page — would change what is affordable by about two orders of magnitude without touching instruction semantics. Sparse or learned addressing is the same idea with a weaker guarantee. The corresponding problem for the content-addressed memory is that attention over N cells does not scale, which is why $N_{\mathrm{m}} = 256$ is a budget rather than a design choice.

10.4 Four Instruction Set Additions the Construction Identified

Writing software on this machine identified four changes, none of which alters the semantics of an existing instruction.

1. **A branch on $N \neq V$, or an instruction that reads the overflow flag into a register.** Either one makes a signed comparison four instructions rather than eight (Section 6.2). The flag is already computed; nothing can see it.
2. **A trap instruction.** Without one, preemption in *tensor-os* is a call at loop back edges rather than an interrupt, and a task that never reaches a back edge never yields (Section 5.5.1).
3. **A readable stack pointer.** Without one, per-task call stacks are impossible and `tcc` must keep the hardware stack exactly one deep (Section 5.2).
4. **A data section in the program format**, or a disk-loading option, or two more address bits. Any of the three closes the information capacity bound that stops the compiler's fixed point (Section 6.3) and returns the five percent of *tensor-os*'s budget currently spent materialising its own globals.

10.5 Extend the Conformance Suite Where It Is Thin

Section 4.5 names two device behaviours that four implementations are currently free to differ about because no vector pins them. The general form of the task is to find the rest of them: every place where the specification constrains a type but not a value is a place where implementations have already diverged or will. Two defects found while adding fault vectors remain uncharacterised, and the full training test suite — over twenty-five minutes of wall clock — has not been run end to end since the kernel and evolutionary modules landed.

10.6 And Then the Things This Machine Was Built For

If and only if Section 10.1 comes out favourably, the programme the architecture was designed around becomes worth attempting, in this order: classical algorithms with known optimal lengths, so that correctness and optimality are both checkable; programs that correctly *invoke* frozen pretrained weights resident in tensor memory, which is the first test of whether a learned program can respect an interface it did not design; joint learning of program structure and weights through the differentiable GPU commands of Section 3.11.1; and only then anything resembling an agent. Each phase has a cheap failure mode, and the right posture is to take it.

11 Discussion and Limitations

11.1 Limitations of the Approach

Softness is expensive, and one term dominates. Soft instruction fetch costs $\mathcal{O}(AI)$ per instruction — three orders of magnitude above hard addressing, 98% of the machine's per-cycle cost (Table 2). Attention over the content-addressed memory does not scale either, which caps it at 256 slots. Both have candidate fixes (Section 10.3) and neither has been attempted.

The scale gap is unmeasured, not merely unclosed. Between the handful of instructions anything has been trained to produce and the 10^4 instructions the machine demonstrably runs, there is no data at all. We do not know where the method stops working, and until Section 10.2 is run the honest statement is that the practical ceiling is unknown rather than high or low.

The loss landscape moves during training. Long-horizon discrete optimisation is unstable on its own; annealing adds a second source, because the surface being descended is being deformed on a schedule chosen in advance. The number of local minima grows exponentially in program length (Theorem 8.2), and Theorem 8.7 — the statement that would license annealing — is assumed rather than proved.

Extraction is verified semantically and not physically. Four implementations agreeing on 59 vectors establishes that a program means the same thing on four implementations of this instruction set. It says nothing about real timing, real cache behaviour, or real interrupt latency on hardware that is not this machine, and the efficiency argument of Section 9 depends on that second half.

The primitives are fixed, and that is in tension with the contribution. The ALU computes exact operations chosen by a human from a set chosen by a human. Perhaps some should be learned, as a differentiable neural computer learns its addressing. But exactness is precisely what makes extraction to real hardware possible and distinguishes this machine from prior neural computers, so the two cannot both be maximised. We have no principled answer to where the line goes.

11.2 Limitations of the Evidence

The verification result is strong and narrow: agreement among four implementations on the vectors that exist, with Section 4.5 giving two behaviours where the vectors do not exist and the implementations already differ. A conformance suite is a floor. The software results are strong and hand-written: `tcc` and *tensor-os* establish that the machine sustains real programs, establish nothing whatever about learning, and were not produced by any method this paper proposes.

The synthesis evidence is a negative result and should be read as one. We do not know whether the learned kernels are wrong because the budgets are three orders of magnitude too small, because the relaxation optimises something that is not a relaxation of what we want, or because the task is simply hard — and we decline to guess, because the experiment that distinguishes these is written and could be run.

11.3 On Interpretability

It is tempting to treat interpretability as a settled advantage of learned code, and we want to state the narrower version that we think survives.

Reading a two-hundred-instruction program with no names, no types, no comments, and no structure a human would recognise is not obviously easier than reading a small network's activations. Compiler output is readable in exactly that sense and is not, in practice, read. The enthusiasm for solutions no human would have written cuts directly against the claim: a program is interesting precisely when its structure is unfamiliar, which is precisely when it is hard to follow.

What does survive is that learned programs are *traceable* and *verifiable* in ways networks are not. One can single-step them, assert on intermediate machine state, run them against adversarial inputs with complete coverage of a bounded input space, and prove properties of bounded fragments. Section 4 is what that looks like in practice, applied to hand-written code. That is a real and useful property. It is not the same as "we can read the code", and this paper does not claim the latter.

11.4 Broader Implications and Ethics

If the hypothesis of Section 9.2 holds, the consequence is capable agents at a fraction of current inference cost, deployable without connectivity, and inspectable by stepping. If it fails, the negative result has content: it would locate where program synthesis breaks down and say something about the relationship between networks, programs, and the computations both perform. The machine remains useful either way, as a substrate on which questions about that relationship can be asked concretely — which is why the specification, the conformance suite, and the four implementations were worth building carefully regardless of how the hypothesis turns out. They are the part that stays true.

Three ethical considerations are worth naming rather than gestured at. Cheaper capable agents lower the cost of harmful automation exactly as they lower the cost of useful automation, and the interaction between "far cheaper per action" and the volume of actions that becomes economical is the specific risk, not a general one. A program whose structure no human would write is, by construction, a program whose failure modes no human has anticipated; fast, cheap, locally deployable and unfamiliar is a combination that needs evaluation methodology, and we have none to offer beyond Section 4's. And the traceability argument above is a genuine safety argument, provided it is made in its narrow form: what can be verified is behaviour on inputs one thought to test and properties of fragments one thought to state, which is more than can be verified of a policy network and less than the word "interpretable" suggests.

12 Conclusion

12.1 What Was Built

A complete von Neumann computer as differentiable tensor operations, with no discrete index anywhere in its state and exact discrete semantics at $\tau \to 0$; four independent implementations of its instruction set held in agreement by 59 conformance vectors, with recursive `factorial(5)` returning 120 in exactly 49 cycles on all four, soft machine included; and two hand-written artefacts in the 10^4-instruction range — a C compiler that compiles itself while running on the machine to byte-identical output, and an operating system that preemptively schedules five graphical applications.

Two of those are results rather than artefacts. Differential testing across four implementations found six classes of defect that no single implementation could surface, including an

inverted straight-through estimator with exactly zero gradient whose only symptom would have been a training run that annealed normally and then silently stopped learning; we regard that methodology as the most transferable thing in this paper. And building real software forced three findings about the machine — that its instruction set cannot express a signed comparison at full width, that its information capacity rather than its instruction budget bounds what can be loaded into it, and that a device command costing the same at any area makes a compositor text-bound rather than fill-bound. Each is a fact about the machine that no amount of specifying it would have surfaced.

12.2 What Was Not Established

Nothing here synthesises a correct non-trivial program.

The differentiable path is complete and each of its links is verified end to end — declare a kernel, train its instruction-memory logits, crystallise, extract machine code, cross-verify on the discrete Rust virtual machine, embed into hand-written assembly, all fourteen checks passing. What is absent is a *result* from it. At the budgets shipped the learned kernels are wrong, and the experiment that would settle whether gradients earn their cost against pure discrete search is written and unrun. "The pipeline exists and the answer does not" is the accurate summary, and it is a better one than either half alone.

Four questions remain open, and each has an experiment attached (Section 10): whether gradients are useful for program synthesis at all; where the practical ceiling on learned program length lies, given that nothing has been trained anywhere between a handful of instructions and the 10^4 the machine runs; whether the cost of softness can be reduced by hierarchical addressing; and whether an extracted program preserves its behaviour on hardware that is not this machine.

12.3 Closing

The central hypothesis of this work — that computation which a network converges to as a small algorithm may be more efficiently represented as code — is a hypothesis, and this paper does not upgrade it. What the paper offers instead is the substrate on which it becomes a question one can ask precisely, together with the evidence that the substrate is what it claims to be: the same machine as a real one, verified four ways, running a compiler that compiles itself and an operating system that draws windows.

We think that is worth reporting on its own terms. A differentiable computer whose learning question is open is more useful than a learning result whose machine is unverified, and the parts built carefully here — the specification, the conformance suite, the four implementations — stay true whichever way the hypothesis goes.

References

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In *International Conference on Learning Representations*, 2017.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable Forth interpreter. In *International Conference on Machine Learning*, pages 547–556, 2017.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. Software, http://github.com/jax-ml/jax, 2018.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Luc Morales, Luke Hewitt, Luke Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 835–850, 2021.

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538 (7626):471–476, 2016.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with Gumbel-softmax. In *International Conference on Learning Representations*, 2017.

Łukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. In *International Conference on Learning Representations*, 2016. arXiv:1511.08228.

Kenneth Li, Oam Patel, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. Inference-time intervention: Eliciting truthful answers from a language model. In *Advances in Neural Information Processing Systems*, 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

Ziming Liu, Ouail Kitouni, Niklas Nolte, Eric J. Michaud, Max Tegmark, and Mike Williams. Towards understanding grokking: An effective theory of representation learning. In *Advances in Neural Information Processing Systems*, 2022.

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *International Conference on Learning Representations*, 2017.

Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin Riedmiller, Oriol Vinyals, and David Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. In *International Conference on Learning Representations*, 2023. arXiv:2301.05217.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 522–538, 2016.

Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets. *arXiv preprint arXiv:2201.02177*, 2022.

Scott Reed and Nando de Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations*, 2016. arXiv:1511.06279.

Mrinank Sharma, Meg Tong, Tomasz Korbak, David Duvenaud, Amanda Askell, Samuel R. Bowman, Newton Cheng, Esin Durmus, Zac Hatfield-Dodds, Scott R. Johnston, Shauna Kravec, Timothy Maxwell, Sam McCandlish, Kamal Ndousse, Oliver Rausch, Nicholas Schiefer, Da Yan, Miranda Zhang, and Ethan Perez. Towards understanding sycophancy in language models. In *International Conference on Learning Representations*, 2024.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, 2006.

Vikrant Varma, Rohin Shah, Zachary Kenton, János Kramár, and Ramana Kumar. Explaining grokking through circuit efficiency. *arXiv preprint arXiv:2309.02390*, 2023.

A Instruction Set Reference

Table 6 lists the 32 opcodes in five classes. Opcodes `0x00`–`0x0F` are combinational ALU work, `0x10` and `0x11` address memory, `0x12` and `0x13` the hardware stack, `0x14`–`0x1C` control flow,

and `0x1D`–`0x1F` devices and machine control. Every one of them is evaluated on every cycle and weighted by the opcode distribution (Equation (3.10)); the branch conditions in the table are the exact soft polynomials of Equation (3.15), not discretisations of them.

Three encoding facts are not visible in the table. Instructions are the 112-parameter tuple of Equation (3.4), so a program is one $\mathbb{R}^{A\times I}$ array. Branch targets in the immediate field are absolute instruction indices, since the machine is Harvard and instruction memory is addressed separately from data. And there is no add-immediate form: `ADD` ignores its immediate entirely, which is why `INC` expands to a register-register add against a register holding one, and why an assembler must reject a bare `INC` rather than invent an encoding (Section 4.4).

Table 7 lists the assembler pseudo-operations. These are not opcodes, expand to real instructions, and consume no encoding space.

B Memory Map

A soft address is routed to a region by the product of sigmoid gates in Equation (3.17), normalised across regions. Table 8 lists the seven regions of the `paper` profile.

The region bases are spaced far more widely than the regions are large — RAM occupies 64 MB at base `0x00000000` while the next region starts at `0x08000000`, and the 64 KB MMIO window sits alone in a 256 MB gap. That spacing is a design choice rather than an accident. At moderate temperature an address is a blurred number, and gates on adjacent regions would otherwise overlap: a store intended for the framebuffer would partially land in GPU memory. Wide separation keeps the decode well-conditioned in exactly the regime where training operates.

B.1 Memory-Mapped Control Registers

Table 9 lists the ten control registers at offsets from the MMIO base. The status and pending words are soft values in $[0, 1]$ rather than booleans, which is not a rounding convenience: a half-completed disk operation is a legal state during training, and a program polling on `DISK_STATUS` receives a gradient telling it whether polling longer would have helped.

Within each peripheral's window, offset 0 is the queued count, offset 4 pops on read and pushes on write, and offset 8 is the capacity.

C GPU Command Format

The implemented GPU is a raster device. Table 10 lists the six command selectors carried in the immediate field of a `GPU` instruction, with their arguments in `rs1` and `rs2`. Coordinates pack x and w into the low 16 bits and y and h into the high 16; colour is `0x00RRGGBB`, generalised across the configured channel count.

Two commands take state rather than an argument, because two register operands are already spent on geometry. `FILL_RECT` has no colour operand and `BLIT` no length operand; each reads a latch the program sets by an ordinary store beforehand. This is how a command-list GPU normally works — set state, then draw — and it is also the one place where the four implementations are documented as differing about which latch, since no conformance vector pins it (Section 4.5).

All six commands execute on every `GPU` instruction and the six resulting device states are blended by the soft selector, exactly as the ALU blends its primitives. At $\tau \to 0$ the selector

Opcode	Mnemonic	Class	Semantics
0x00	ADD	alu	rd $\leftarrow$ rs1 + rs2
0x01	SUB	alu	rd $\leftarrow$ rs1 $-$ rs2
0x02	MUL	alu	rd $\leftarrow$ rs1 $\times$ rs2 (low 32 bits)
0x03	DIV	alu	rd $\leftarrow$ rs1$/$(rs2 $+ \varepsilon$), stabilized
0x04	AND	alu	rd $\leftarrow$ rs1 $\wedge$ rs2
0x05	OR	alu	rd $\leftarrow$ rs1 $\vee$ rs2
0x06	XOR	alu	rd $\leftarrow$ rs1 $\oplus$ rs2
0x07	NOT	alu	rd $\leftarrow \neg$rs1
0x08	SHL	alu	rd $\leftarrow$ rs1 $\ll$ rs2[0:5]
0x09	SHR	alu	rd $\leftarrow$ rs1 $\gg$ rs2[0:5] (logical)
0x0A	SRA	alu	rd $\leftarrow$ rs1 $\gg$ rs2[0:5] (arithmetic)
0x0B	MOV	alu	rd $\leftarrow$ rs2
0x0C	LDI	alu	rd $\leftarrow$ imm
0x0D	CMP	flags	**F** $\leftarrow$ flags(rs1 $-$ rs2)
0x0E	CMOVZ	select	rd $\leftarrow F_{\mathrm{z}}$? rs1 : rs2
0x0F	CMOVN	select	rd $\leftarrow F_{\mathrm{n}}$? rs1 : rs2
0x10	LOAD	memory	rd $\leftarrow$ mem[rs1 + imm]
0x11	STORE	memory	mem[rs1 + imm] $\leftarrow$ rs2
0x12	PUSH	stack	**S**[**SP**] $\leftarrow$ rs1; **SP**++
0x13	POP	stack	**SP**$--$; rd $\leftarrow$ **S**[**SP**]
0x14	JMP	branch	**PC** $\leftarrow$ imm
0x15	JEQ	branch	**PC** $\leftarrow$ imm if F_{z}
0x16	JNE	branch	**PC** $\leftarrow$ imm if $1 - F_{\mathrm{z}}$
0x17	JLT	branch	**PC** $\leftarrow$ imm if $F_{\mathrm{n}}(1 - F_{\mathrm{z}})$
0x18	JGE	branch	**PC** $\leftarrow$ imm if $1 - F_{\mathrm{n}}$
0x19	JLE	branch	**PC** $\leftarrow$ imm if $F_{\mathrm{n}} + F_{\mathrm{z}} - F_{\mathrm{n}}F_{\mathrm{z}}$
0x1A	JGT	branch	**PC** $\leftarrow$ imm if $(1 - F_{\mathrm{n}})(1 - F_{\mathrm{z}})$
0x1B	CALL	branch	push **PC** + 1; **PC** $\leftarrow$ imm
0x1C	RET	branch	pop into **PC**
0x1D	GPU	device	GPU command, selector in imm, arguments in rs1, rs2
0x1E	IO	device	Peripheral transfer, port and direction in imm
0x1F	HALT	system	Halt execution

Table 6

Pseudo-op	Expansion	Meaning
`NOP`	`MOV r0, r0`	No operation
`INC`	`ADD rd, rd, rs`	Increment by rs, which must hold 1
`DEC`	`SUB rd, rd, rs`	Decrement by rs, which must hold 1
`NEG`	`SUB rd, r0, rs1`	Two's complement negate (requires r0 == 0)
`CLR`	`LDI rd, #0`	Zero a register

Table 7

Region	Base	Size	Contents
RAM	`0x00000000`	`0x04000000`	Main RAM, 64 MB
NTM	`0x08000000`	`0x00000400`	NTM content-addressed memory
MMIO	`0x10000000`	`0x00010000`	Memory-mapped control registers
GPU	`0x20000000`	`0x00040000`	GPU memory, 64 K words
Framebuffer	`0x30000000`	`0x000E1000`	$480 \times 640 \times 3$ RGB
Disk	`0x40000000`	`0x04000000`	Disk window, 64 MB
Peripherals	`0x50000000`	`0x00010000`	Peripheral ring buffers

Table 8

is one-hot and five results are discarded — which is what lets a gradient tell a program that it wanted `FILL_RECT` and issued `PLOT`.

C.1 The Command Queue

A wider compute command set — matrix multiply, convolution, reductions, elementwise operations — was described in an earlier design and is issued through the memory-mapped queue at the MMIO base rather than through the `GPU` opcode's immediate field. It is *not* in the instruction set implemented here, and no result in this paper uses it. We mention it only because the distinction between the two command paths has confused readers of the design documents, and because Section 9.3's hybrid argument depends on the compute path eventually existing.

D Machine Profiles and Training Defaults

D.1 The Two Profiles

Two machine profiles exist and every implementation must accept both. All architectural figures in this paper are the `paper` profile; every conformance vector runs on `dev`, and so does every example in the differentiable path, because a `paper`-profile machine state is 2.1 GB against `dev`'s 3.58 MB (Section 9.1). The two differ in sizes only.

D.2 Training Defaults

Table 12 lists the shipped defaults. The temperature schedule is exponential from τ_0 to $\tau_{\min}$ across the warm-up, annealing and crystallisation step counts given there, with adaptive rescheduling enabled: the schedule may extend the annealing phase when the discrete and soft losses have not yet converged, so the phase boundaries are the nominal ones rather than guaranteed ones. The boundary that reimplementations get wrong is between annealing and

Offset	Register	Meaning
0x0000	DISK_CMD	0 = idle, 1 = read sector, 2 = write sector
0x0004	DISK_LBA	Sector number
0x0008	DISK_ADDR	RAM address for DMA
0x000C	DISK_STATUS	Soft completion flag
0x0010	GPU_CMD	GPU command latch
0x0014	GPU_STATUS	GPU busy flag
0x0020	TIMER_LO	Cycle counter, low word
0x0024	TIMER_HI	Cycle counter, high word
0x0030	IRQ_MASK	Soft interrupt mask
0x0034	IRQ_PENDING	Soft pending interrupts

Table 9

Selector	Command	Arguments
0	NOP	—
1	CLEAR	Clear framebuffer to color in *rs1*
2	PLOT	Write pixel: *rs1* = packed (x, y), *rs2* = color
3	FILL_RECT	*rs1* = packed (x, y), *rs2* = packed (w, h)
4	BLIT	Copy from GPU memory at *rs1* to framebuffer at *rs2*
5	FLUSH	Present the framebuffer

Table 10

crystallisation, where the forward pass hardens but the backward pass is held at the gradient temperature floor; without that floor, crystallisation silently ends learning for the reason given in Section 7.3.

E Deferred Proofs

E.1 Proof of Theorem 8.1

Proof of Theorem 8.1. Fix the step count T. Write the machine's one-step transition as $\mathrm{Step}_\tau : \mathbb{R}^D \times \mathbb{R}^d \to \mathbb{R}^D$, mapping a state and the program parameters to a successor state, and $f_\tau(x;\theta) = \pi\big(\mathrm{Step}_\tau^T(\iota(x),\theta)\big)$ for an input embedding ι and an output projection π, both fixed linear maps.

Continuity. Every constituent of Step_τ is one of: Equation (3.1), Equation (3.2), a fixed permutation (Equation (3.13)), a bilinear contraction (Equation (3.5), Equation (3.6), Equation (3.10)), or a polynomial in its arguments (Equation (3.8), Equation (3.7), Equation (3.15)). The first two are jointly continuous on $(0,\infty)\times\mathbb{R}^n$: the map $(\tau,\mathbf{x}) \mapsto \exp(x_i/\tau)$ is continuous there and the denominator is bounded away from zero. The rest are continuous unconditionally. Composition of finitely many jointly continuous maps is jointly continuous, so Step_τ^T and hence f_τ are. Composing with the continuous bounded ℓ and taking a finite or dominated expectation preserves continuity, giving joint continuity of $\mathcal{L}_\tau$.

The limit. Fix θ with no ties among the selector logits at any address. By Theorem 3.1 each softmax converges pointwise to the one-hot vector at its arg max and each soft indicator to the corresponding Heaviside value as $\tau \to 0^+$. Under these one-hot selectors, Equation (3.5) selects a single instruction row, Equation (3.6) a single register, Equation (3.7) writes one register and leaves the others fixed, Equation (3.10) returns a single primitive, and Equation (3.14) takes one

Parameter	`paper`	`dev`
Registers, word bits	16, 32	16, 32
Address bits (slots)	16 (65,536)	8 (256)
Stack depth	4096	64
Cache lines × words	256 × 8	16 × 4
NTM slots	256	64
RAM	64 MB	16 KB
Disk (sector size)	64 MB (4 KB)	64 KB (512 B)
GPU memory	65,536 words	1024 words
Display	640 × 480	32 × 24
Peripherals (ring slots)	6 (256)	6 (16)
Machine state size	2.1 GB	3.58 MB

Table 11: The two profiles. The configuration validator refuses more than 24 address bits, because soft fetch costs $\mathcal{O}(2^b)$ per instruction (Theorem 3.2).

of its two arguments — that is, Step_0 is the discrete transition function. Induction on the T steps carries the limit through the composition, since each Step_τ is continuous in the state uniformly on compacta and the state space of a fixed-length run is bounded. Bounded convergence then gives $\mathcal{L}_\tau \to \mathcal{L}_0$ under the expectation. The excluded set — parameters with a tied maximum in some selector — is a finite union of hyperplanes and so has Lebesgue measure zero. □

E.2 Proof of Theorem 8.5

Proof of Theorem 8.5. Fix a prefix-free binary encoding of programs and let $|P|$ be the code length of P in bits. By Kraft's inequality, $\sum_P 2^{-|P|} \le 1$, so $p(P) = 2^{-|P|}$ is a sub-probability distribution on the program space, chosen before the sample is drawn.

Fix P. The training loss is the mean of m i.i.d. terms bounded in $[0,1]$ with expectation $\mathcal{L}_{\text{test}}(P)$, so Hoeffding's inequality gives $\mathbb{P}\big[\mathcal{L}_{\text{test}}(P) - \mathcal{L}_{\text{train}}(P) > \varepsilon\big] \le \exp(-2m\varepsilon^2)$. Set $\varepsilon_P = \sqrt{(|P| \ln 2 + \ln(1/\delta))/2m}$, so that $\exp(-2m\varepsilon_P^2) = \delta\, 2^{-|P|}$.

A union bound over the countable program space gives failure probability at most $\sum_P \delta\, 2^{-|P|} \le \delta$, so with probability at least $1 - \delta$ the bound holds simultaneously for every P — including one chosen after seeing the sample, which is what makes the statement usable. □

Remark E.1. The bound is uniform over programs, so it applies to a program produced by any search procedure, gradient-based or otherwise. Its complexity term is the code length of the emitted binary and is known exactly, which is the contrast with a norm-based bound for a neural network drawn in Section 8.3. It is also loose in the usual way: for $m = 10^4$ and a 100-instruction program at 112 bits each, the square-root term is about 0.6, which is vacuous for a loss in $[0,1]$. The bound bites for short programs and large samples, which is precisely the regime this machine is designed to produce and has not yet reached.

Parameter	Default	Note
Optimisation		
Optimiser	Adam, $\beta = (0.9, 0.99)$	
Learning rate	0.05 (`dev`), 0.03 (`paper`)	
Batch size (episodes)	16 (`dev`), 64 (`paper`)	memory-bound, not statistics-bound
Gradient clipping	1.0	not optional — a program that starts branching on garbage produces gradients orders of magnitude above the previous step's
Weight decay / L_2	0 / 10^{-4}	
Temperature (Section 7.2)		
τ_0 / $\tau_{\min}$	10.0 / 0.01	
Warm-up / anneal / crystallise	1000 / 8000 / 1000 steps	
Refinement steps	500 (`paper`), 0 (`dev`)	discrete local search
Schedule form	exponential	
Gradient temperature floor	0.05	holds the *backward* pass above the temperature at which the softmax Jacobian is numerically zero, even after the forward pass has hardened
Shaped reward (Section 7.4)		
Output distance	1.0	bitwise
Partial credit	0.2	
Efficiency	0.05	
Safety	0.5	out-of-bounds access, division by zero
Halting	0.2	
Optional components, default state		
Policy gradient	off	gradient through the soft machine is the primary signal
Discrete local search	on, 2 restarts	cheap, and what turns a nearly-correct crystallised program into a correct one
Imitation warmstart	off	`enabled: false` is a from-scratch run
Curriculum	off	threshold 0.95 over a 50-episode window
Population	off	size 4; N times the compute

Table 12: The library defaults, as written out in the repository's base configuration. These are defaults and not the settings of any reported run: this paper reports no training result (Section 7.7), and the budgets the shipped example scripts use are two to three orders of magnitude below every one of the step, population and restart counts a serious attempt would need.