The General Purpose Neural Computer

A Learned, Grounded, and Deterministic Interpreter of a Turing-Complete Instruction Set

Jovonni L. Pharr
Georgia Cyber Warfare Range / Blackmind
info@gacwr.org

August 2026

Abstract

The prevailing artifact of machine intelligence is a *stochastic* one: the language model, whose computation is sampled, non-repeatable and illegible—exactly the wrong set of properties for the large class of problems that are fundamentally *algorithmic*. This work proposes a *General Purpose Neural Computer* (GPNC): a differentiable machine that learns the operational semantics of a forty-one-opcode WebAssembly `i32` instruction set from the observation of its execution alone—what is learned is the table assigning each opcode its stack, local, memory and control effect, the arithmetic itself being delegated to an exact integer unit and differentially tested separately—runs deterministically and auditably, and generalizes exactly to program lengths, loop trip-counts, memory sizes, and operand magnitudes it never encountered during learning. That neural networks can be made universal is settled: hand-constructed weights already emulate one-instruction computers, and finite-precision Turing-completeness is established. The contribution is the intersection that no prior differentiable computer has occupied at once: a computer that is simultaneously *learned* from traces rather than hand-compiled, *grounded* in a real instruction set differentially tested against a production WebAssembly engine, *length-generalizing* through an instruction table that composes over unbounded execution, and *deterministic* through a discretize-and-continue membrane that is necessary rather than merely stabilizing: order and equality are not continuous predicates, so an unpinned datapath's exactness is discontinuous at $\varepsilon = 0$—it fails by tie-breaking at any perturbation however small, not by the gradual drift usually blamed. We formalize the machine on a fifteen-opcode core and *prove* a label-free identifiability theorem with an explicit sample complexity—the operational semantics of every opcode is recoverable from before-and-after state pairs alone, with no supplied instruction labels—and we discharge its two hidden obligations by machine proof: the recovered interpreter is shown *totally* equivalent to the reference semantics by an SMT proof over all configurations rather than by sampling (15/15 opcodes on non-empty stacks, 14/15 once the empty stack is admitted—a distinction only a proof could have drawn), and the separation condition the theorem requires is certified by an exhaustive witness table over every wrong microcode. We then exhibit the recovered interpreter executing real loop programs at trip-counts it never observed. Part III carries the construction to a forty-one-opcode WebAssembly `i32` core whose numeric semantics agrees with a production engine on 300 000 of 300 000 operand tuples; closes the stack-depth lemma, so total equivalence is proved for 40 of the 41 opcodes over the *entire unbounded* state space rather than over a finite window of depths, and decides every one of the 133 783 wrong (opcode, microcode) pairs; deletes the hand-designed probe distribution and recovers thirty-eight of forty-one opcodes from the execution traces of programs alone, naming the corpus on which that fails; trains the decoder network itself, by gradient on state pairs alone, to the same table the search returns; and runs recognizable algorithms—Euclid, bubble sort, CRC-32, bignum addition, a compiled Minsky machine and a Brainfuck interpreter—cycle-exactly out to 1.5×10^7 cycles, sustaining exactness to 1.6×10^7 on a 2^{20}-cell memory. We answer the objection that this is a WebAssembly result rather than a method twice over: by carrying the construction to a *second machine model*—RV32I, a register machine of 40 base integer instructions, where the *search* half of the procedure transfers—a table trace-exact on 120 held-out programs—and the *gradient* half does not, both reported against criteria fixed in advance—and by executing *real compiled* WebAssembly emitted by a production toolchain, cycle-exactly on 3 900 of 3 900 trials; functions lifted from SQLite, zlib and a Rust binary are executed separately under a weaker protocol and agree on 132 of 148. The semantics learned here from execution alone, with no access to a specification or to source text, are those of *five* real instruction sets spanning four different machine shapes: the complete WebAssembly integer MVP at one hundred and four opcodes, a *stack* machine with three unbounded structured state components; RISC-V RV32I

with ZICSR and ZIFENCEI at forty-seven and RV64IM at sixty-five, *register* machines; ARMV7-A at one hundred and nine, a register machine with condition flags, predicated execution and a barrel shifter; and the MOS 6502 at one hundred and fifty-one, an *accumulator* machine with variable-length instructions—476 opcodes in total, each table differentially tested against a production emulator (1 080 000/1 080 000 whole post-states against UNICORN for ARM, 1 500 000/1 500 000 against `py65` for the 6502) and each proved by machine. Across the five tables 1 052 266 133 wrong (opcode, microcode) pairs are decided with none unknown, and the recovered WebAssembly table executes 1669 of 1669 real call-graph closures lifted from SQLite, zlib and a Rust binary—554 522 instructions—agreeing with WASMTIME throughout. The theorem's hypotheses are re-derived per shape rather than carried over, and the shapes disagree about what they require: predication strengthens the probe condition into one no configuration can discharge, while the 6502 removes it entirely. A baseline suite trained on the same traces under the same protocol makes the comparison a measurement, separating the systems on *composition* rather than accuracy: the strongest learned arm is exact on 0.819 of single steps and on 0/40 whole executions at every trip count, where the recovered interpreter is 4000/4000 and 40/40 at 1175 states. Every formal statement in this paper is additionally machine-checked in Lean 4, with each restriction named rather than absorbed. Every dimension of the validated machine is tabulated, every rate carries an interval and an explicit n, and the learned content that survives to inference is a table of integer five-tuples.

Contents

Evidentiary standard

Every number in this paper is produced by a named experiment script and recorded in a machine-readable receipt; the figure scripts read those receipts and refuse to plot when one is absent, so no measurement is ever typed into the manuscript or into a plotting script. Every rate carries a Wilson 95% interval and an explicit n; single deterministic executions and machine proofs are labelled as such rather than given a spurious interval. Success criteria are pre-registered, and each is reported against its measured outcome. The regeneration pipeline re-derives the manuscript's load-bearing quantities from the artifact on every run—running the prover, re-measuring the sampler, inspecting the source—so that a stale artifact cannot reproduce a number (Appendix B).

The whole pipeline is CPU-only and reproducible end to end from a single runner. Five further experiments require a GPU and are specified as runnable scripts with pre-registered success criteria and pre-registered falsifiers; no number from any of them appears anywhere in this paper.

Part I
Preliminaries

1 The State of Today

Though not exclusively the case, the dominant computational artifact of the present moment is the large language model: a system trained on the surface form of human language, whose every output is drawn from a distribution. This has produced fluency of a kind previously unseen. It has also enshrined a set of properties that, for an enormous and growing class of tasks, are precisely the properties one does not want.

A language model is *stochastic*: the same prompt produces different completions across runs, and no observer—not even the model's authors—can point to the discrete program that produced a given output. It is *non-repeatable* and, in the strict sense, *illegible*: the "computation" is smeared across billions of weights and a sampled context. For open-ended generation of prose these are acceptable, even desirable, characteristics. For anything that is fundamentally an *algorithm*—sorting, arithmetic, parsing, the faithful execution of a specified procedure—they are disqualifying. One does not want a bank ledger, a compiler, or a control loop that returns a different answer when asked twice.

The thesis of this work is that the algorithmic regime deserves its own artifact, and that this artifact can be *neural* without being *stochastic*. We want a machine that is general purpose, in that it can run any program; deterministic and auditable, in that identical inputs yield identical outputs and the computation may be inspected step by step; length- and scale-generalizing, in that a routine learned on small inputs runs correctly on arbitrarily larger ones because it has acquired an algorithm and not a table; and neural and differentiable, in that its behavior is learned from data and gradients may flow through its execution, so that one may *search* for programs and not merely run them.

A classical computer keeps its program (the code) distinct from its machine (a processor, memory, and an instruction set). A neural network collapses this separation: the machine *is* the program, fit end to end. That collapse is the source of the network's generality and its sample efficiency, but it is also the source of its indeterminism and its illegibility. The proposal of this work is to *restore the separation without abandoning learning*: to retain a real, discrete instruction set and a deterministic execution loop, while *learning* the meaning of each instruction and rendering the loop differentiable. The result is a computer whose microcode is a neural network.

2 Related Work

We situate the general purpose neural computer against six threads. The through-line is that each thread has secured one or two of the four properties we require at once—*learned*, *grounded*, *length-generalizing*, *deterministic*—and that no prior differentiable computer has held all four together with a label-free identifiability guarantee.

The positioning is a reading of ten cited systems, and the suite that settles it empirically is specified as runnable code: a DNC or NTM, a Universal Transformer with ACT, the hand-compiled looped Transformer as an exactness ceiling, and Transformers with RoPE and with randomized positional encodings, all on a shared task battery and on our program benchmark. It carries a pre-registered success criterion and a pre-registered falsifier—that GPNC prove indistinguishable from those baselines—and is estimated at 40–80 GPU-hours.

One consequence of that suite can be stated now, without running it, because it follows from what the baselines *are*. The looped-Transformer construction has hand-written weights, so it is exact at every length by construction; it will not be beaten on exactness, and a figure showing GPNC exact at $10\times$ the training length alongside a looped Transformer that is also exact there would not be evidence for GPNC. The axis on which the comparison is informative is therefore not exactness but *what was learned versus written*, which is the same distinction this paper's "written, not learned" critique rests on, and it is the axis the suite is designed to report.

Neural Turing machines and differentiable computers. The Neural Turing Machine [2] and the Differentiable Neural Computer [3] coupled a controller to an external memory addressed by learned content- and location-based attention, learning to copy, sort, and traverse graphs. The Neural GPU [4] pursued algorithm learning through parallel cellular computation, and Neural Random-Access Machines [12] added learned, pointer-style dereferencing over a discrete memory. The Neural Programmer [11] induced latent programs over a small set of differentiable operations under weak supervision. These systems are *learned*, and they established that a network can operate a memory decoupled from its parameters; but their memory is soft throughout, so state *drifts* over long horizons as per-cycle error compounds (the recurrence broken here by the pinning membrane, Eq. (4)), and they generalize poorly beyond training-length inputs. None commits to a real instruction set with a reference semantics.

Neural program induction and interpreters. A second line learns to *execute* rather than to solve a task. Learning to Execute [13] trained sequence models to predict the output of short programs; the Neural Programmer-Interpreter [5] learned from full execution traces of a fixed library of subprograms, and recursion was later shown to be the ingredient that let such architectures generalize provably [14]. The Differentiable Forth interpreter [6] grounded computation in a real stack language but required a human to supply the program as a sketch with holes for gradient descent to fill. Our interpreter differs on the axis of supervision and grounding: it consumes only $(\text{opcode}, \text{pre}, \text{post})$ transitions, recovers the microcode of a complete instruction set with no supplied program and no per-opcode labels (Theorem 1), and executes deterministically against a reference semantics that is differentially tested against a production WebAssembly engine [35], with the points of divergence from the standard [34] tabulated (Section 12, "grounding"). The oracle against which that grounding is measured is our own reference semantics, a compact Python small-step function, differentially tested against the engine; a machine-checked mechanisation of the WebAssembly specification [10] exists and is the standard this line of work should be checked against next.

Programmatic and compiled Transformers. RASP [7] and its compiler Tracr [8] translate a human-written program into the exact weights of a fixed-depth Transformer, and the

looped-Transformer construction [1] hand-builds frozen weights that emulate the SUBLEQ one-instruction computer and are thereby universal. These results make the network–program relationship exact but by *construction*: the weights are written or compiled, the depth is bounded, and nothing is learned from data. Turing-completeness results for attention and recurrence [15] likewise establish expressivity in principle, typically under unbounded numerical precision—the "infinite-precision evasion" that Proposition 1 avoids by placing unboundedness in discrete external memory.

Learning instruction semantics from execution. Recovering an instruction set's semantics by observing it run has a twenty-year symbolic lineage: abstract transfer functions for an 8-bit ALU [37], template-based synthesis of symbolic encodings for the x86-32 ALU from input/output samples [38], stratified synthesis for 1 795 x86-64 instruction variants [39], and enumerative synthesis for more than 10^5 x86-64 encodings [43]. That lineage is symbolic rather than neural, it targets *hardware* instruction sets, and what it learns is a per-instruction input/output formula. Its own validation standard is instructive: stratified synthesis could compare only the 79.75% of its output for which a hand-written formula existed, verifying 76.7%, and both it and the enumerative successor state plainly that no ground-truth semantics exists to check the rest against.

Every virtual-machine bytecode semantics on record—WebAssembly [10, 34], the JVM, the EVM—is hand-written. Recovering virtual-machine semantics from execution is not itself new—dynamic analyses have reconstructed emulator and interpreter behaviour from traces [40, 41]—and learned hardware semantics have been composed into an executable pipeline before [42]. What we believe is new here is narrower and we state it that way: the recovery is driven by a *gradient learner* over a declared hypothesis space rather than by symbolic search or template enumeration, and the object it returns is proved *totally* equivalent to its reference over the entire unbounded state space rather than validated on samples. No claim is made here about counting more instructions, because the two lines count different things. That lineage counts encodings whose semantics are *conjectured*; this work counts opcodes whose semantics are *proved*—Theorem 2 establishes total equivalence for 40 of 41 over the entire unbounded state space, where stratified synthesis could check only the 79.75% of its output backed by a hand-written formula, and both it and its enumerative successor record that no ground truth exists for the remainder. A formula per instruction also cannot be executed. The object recovered here is an interpreter: it runs Euclid, CRC-32, bubble sort, a compiled Minsky machine and a Brainfuck interpreter cycle-exactly, out to 1.5×10^7 cycles. Extending the method across machine models—and to instruction sets the symbolic lineage records as unattempted, ARM and RISC-V among them—is the subject of ongoing work.

Neural algorithmic reasoning and the CLRS benchmark. The neural algorithmic reasoning program [16] argues for networks that align to the structure of classical algorithms, and the CLRS benchmark [17] operationalizes this over thirty textbook algorithms with step-level (hint) supervision, following graph-algorithm executors trained to imitate intermediate states [18]. This work shares the goal of algorithm acquisition but inverts the target: rather than learning one network per algorithm from strong step supervision, we learn a single *interpreter* whose fixed weights run *any* program on the instruction tape, and we supervise only on before/after state pairs with no algorithm-specific hints.

Length and systematic generalization, and why it fails. The failure mode we most directly target is length generalization. SCAN [19] exposed the systematic compositional failures of seq2seq models; a survey against the Chomsky hierarchy [20] showed that standard architectures fail to climb it and do not extrapolate to longer inputs; and controlled studies confirm that large models generalize in length only with scaffolding such as scratchpads and careful

prompting [21, 22]. A substantial part of the effect is positional: length generalization is highly sensitive to the positional-encoding scheme [23], and randomized positional encodings recover much of it [24]. Our resolution-invariant addressing (Eq. (2), Eq. (3)) attacks the same problem from the operator side—its parameters are the size-independent taps k, so a rule learned at small N transports unchanged to large N (Section 8, Figure 3).

Neural operators and spectral methods. The resolution invariance of Eq. (3) is exactly the property that makes neural operators mesh-independent. The Fourier Neural Operator [25] parameterizes a layer as a pointwise multiply in the spectral domain and thereby transfers across discretizations; DeepONet [26] learns operators via a branch/trunk decomposition; and the general neural-operator framework [27] formalizes maps between function spaces. We import this machinery onto the *position axis* of a machine's memory. The transfer is not automatic (Section 8): a *free-modes* spectral operator that learns an independent multiplier per mode *index* fails out of distribution, because a shift's spectral response $e^{-2\pi imd/N}$ depends on N at fixed m; only the offset-domain parameterization, whose transfer function re-derives the per-length phase from Eq. (3), generalizes. Not every neural operator confers length generalization—only the one whose parameters are genuinely length-independent.

Identifiability of learned representations. Our label-free recovery result (Theorem 1) is an identifiability statement, and it is worth positioning against the identifiable-representation literature. Nonlinear ICA is unidentifiable in general, but becomes identifiable given auxiliary structure: temporal contrast [28], auxiliary variables with generalized contrastive learning [29], and the iVAE framework that unifies these under a conditionally-factorial prior [30]. Our setting is discrete rather than continuous and the identifying structure is operational: the *discriminating probe distribution* plays the role of the auxiliary variable, guaranteeing that any two distinct microcodes are separated on some observed configuration, so the semantics of each opcode is pinned uniquely by its action on states. This yields an exact, finite recovery guarantee rather than an identifiability-up-to-equivalence result. Relatedly, the differentiable-synthesis view of our relaxed-program objective connects to inductive program synthesis with exact verification of discretized candidates, as in DreamCoder [31] and the abstraction-and-reasoning corpus [32]; here, synthesis runs *through* the learned, grounded interpreter and every candidate is checked bit-exactly against the reference (Section 12).

3 Current Problems

From this history we extract the specific deficiencies that a general purpose neural computer must simultaneously resolve.

Drift. A soft, purely continuous memory accumulates numerical error over long execution and cannot sustain the hundreds or thousands of steps that real programs require. Any machine intended to run real algorithms must break this error recurrence.

Absence of a grounded instruction set. The learned-memory systems address memory in an ad hoc, learned manner, with no explicit instruction set, no formal semantics against which to verify behavior, and no reduction by which to argue universality. Without a real instruction set the notion of "running a program" is never made precise.

Written, not learned. The systems that are exact—Tracr, and the hand-built looped-Transformer construction—achieve their exactness by having a human write or construct the program. They are compilers and constructions, not learners.

The infinite-precision evasion. The universality arguments for the recurrent models smuggle an unbounded stack into the digits of a real-valued activation, a device that no physical machine realizes. A universality argument a physical machine can realize must place unboundedness in discrete, external memory of bounded precision per cell.

No length generalization. Almost uniformly, these systems fail to run correctly on inputs longer than those they were trained on—the surest sign that they have fit a table rather than acquired an algorithm.

4 The Discipline of Neural Interpretation

We frame the resolution of these deficiencies as a discipline we call *neural interpretation*: the learning of an *interpreter* for a fixed, discrete instruction set, rather than the learning of a task. An interpreter is the object that makes a machine general purpose—the same interpreter runs every program—and it is the object whose correctness can be stated precisely, opcode by opcode, against a reference semantics. Neural interpretation asks that this interpreter be learned from the observation of execution, that its execution be deterministic and auditable, and that it be differentiable so that programs may be searched for. The remainder of this work develops such an interpreter and validates it.

Part II
Proposal

5 The Machine

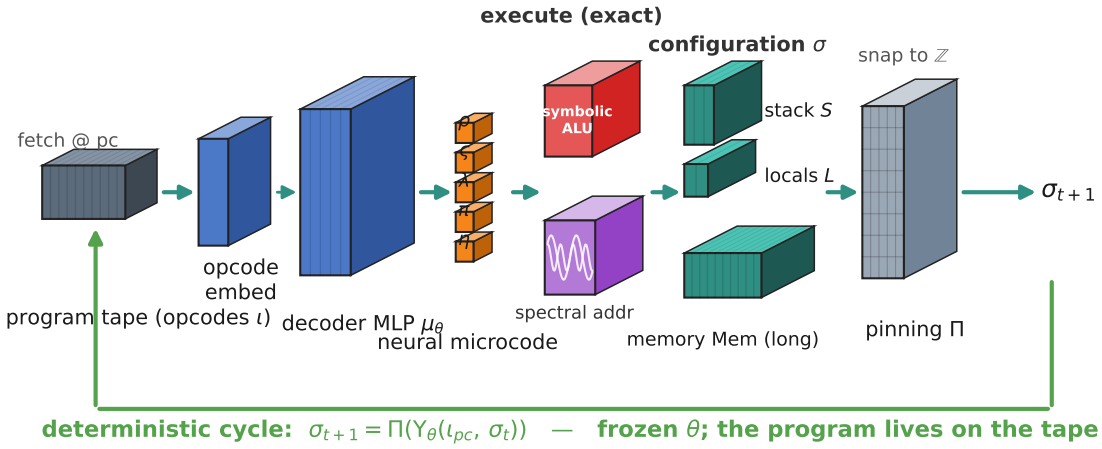


Figure 1: The *proposed* GPNC datapath. Each cycle *fetches* the opcode at the program counter, *decodes* it through a small network into a neural microcode $\mu(o) = (\rho, \varsigma, \lambda, \pi, \eta)$, *executes* it against the configuration $\sigma = (S, L, \mathrm{Mem})$ with an exact symbolic ALU, and *pins* the result back onto the integer lattice. The weights θ are frozen; the program lives on the tape, which is the origin of the machine's self-adaptation. The spectral (Fourier) addressing operator drawn on the execute stage is evaluated *standalone* (Section 8); the interpreter validated in Section 12 addresses memory by integer indexing, and integrating the two is the next step.

The GPNC mirrors the configuration of a real register-and-stack machine. A configuration σ is a tuple $(S, \mathrm{Mem}, L, \mathrm{pc})$ comprising a value stack S, a linear memory Mem, a bank of local registers L, and a program counter pc. During learning each component is a differentiable tensor; at inference each cell holds a discrete symbol of the instruction set. The machine advances by a

frozen micro-sequencer that repeats, for an unbounded number of cycles and until a learned halt, the cycle *fetch, decode, execute, pin, advance*:

$$\sigma_{t+1} \;=\; \Pi\big(\,\Upsilon_\theta(\,\iota_{\mathrm{pc}_t},\,\sigma_t\,)\,\big),$$

where ι_{pc_t} is the instruction fetched at the current program counter, Υ_θ is the learned transition with fixed parameters θ, and Π is the pinning operator defined below. The weights θ are frozen; the *program* lives on the instruction tape. This is the origin of the machine's self-adaptation: a single parameter set runs any program, because to change the computation one changes the instruction stream, not the weights.

Definition 1 (Neural microcode)**.** *For each opcode o of the instruction set $\mathcal{I}$, the decoder emits a microcode $\mu(o) = (\rho, \varsigma, \lambda, \pi, \eta)$, where ρ is the number of operands removed from the stack, ς selects how the pushed value is computed, λ indicates whether a local register is written, π is the effect on the program counter (advance, unconditional branch, or conditional branch), and η is the memory effect (none, load, or store). The microcode is produced by a small network over an opcode embedding.*

Concretely, one cycle applies $\mu(o) = (\rho, \varsigma, \lambda, \pi, \eta)$ to $\sigma = (S, \mathrm{Mem}, L, \mathrm{pc})$ by removing ρ operands, forming a pushed value $v = \varsigma(\cdot)$ and a branch condition c from an exact symbolic unit, and updating the program counter according to π:

$$S' = \mathrm{push}\big(\mathrm{pop}^{\rho}(S),\, v\big), \qquad \mathrm{pc}' = \begin{cases} \mathrm{pc}+1, & \pi = \textsc{next}, \\ \tau, & \pi = \textsc{br}, \\ \tau \text{ if } c \neq 0 \text{ else } \mathrm{pc}+1, & \pi = \textsc{br\_if}, \end{cases} \tag{1}$$

with L and Mem updated by λ and η respectively. Learning the interpreter is learning the finite table $\{\mu(o)\}_{o\in\mathcal{I}}$; the arithmetic that produces v and c is delegated, not learned.

6 Resolution-Invariant Addressing

One mechanism that confers resolution invariance is the manner in which the machine addresses memory; the length generalization reported for the interpreter itself rests on the instruction table rather than on this operator, which sits outside it (Section 8). Rather than a fixed positional lookup, the read and write heads move by a learned *shift operator*. Let $a_t \in \Delta^{N-1}$ be the soft one-hot address over N cells at cycle t, and let $k = (k_{-1}, k_0, k_{+1})$ be a decoder-emitted distribution over the relative displacements $\{-1, 0, +1\}$. The address advances by circular convolution with this kernel,

$$a_{t+1}[j] \;=\; \sum_{\delta\in\{-1,0,+1\}} k_\delta\, a_t[\,(j-\delta) \bmod N\,], \tag{2}$$

and the value read is $\langle a_{t+1}, \mathrm{Mem}\rangle$. The essential property is that the parameters of Eq. (2) are the three numbers k, *independent of* N: the rule "advance one cell toward the top of the stack" is the same rule whether the stack holds four elements or four thousand. A fixed positional table encodes "the value is at index seven" and is silent at index seven hundred; a shift operator encodes a size-independent rule and therefore transports without modification to inputs of any length. This is the same principle that allows a neural *operator* to map functions across resolutions, applied here to the position axis of a machine's memory.

The connection to neural operators is exact, and it is worth making so. Equation (2) is a circular convolution, and by the convolution theorem it is diagonalized by the discrete Fourier

transform: in the spectral domain each mode is multiplied by a transfer function, exactly as a Fourier Neural Operator applies a layer,

$$(S\,a)^{\wedge}(m) \;=\; \widehat{H}(m)\,\widehat{a}(m), \qquad \widehat{H}(m) \;=\; \sum_{\delta\in\{-1,0,+1\}} k_\delta\, e^{-2\pi i\, m\,\delta/N}, \tag{3}$$

so that the head advances by $a_{t+1} = \mathcal{F}^{-1}\{\,\widehat{H}\cdot\mathcal{F}\{a_t\}\,\}$. The learnable content is the three taps k; the transfer function $\widehat{H}$ depends on the memory size N only through the machine's own frequencies m. This is the precise mechanism of resolution invariance, and Section 8 realizes it and measures it.

7 The Symbolic Arithmetic Unit and the Pinning Membrane

Two components of the machine are withheld from learning by design.

The first is arithmetic. The integer operations of addition, subtraction, comparison, and equality are computed *symbolically and exactly* on the operands the machine gathers, and are never approximated by a network. This is a direct response to the historical failure of neural models asked to learn multi-digit arithmetic: a machine that must *learn* thirty-two-bit multiplication inherits that model's brittleness, whereas a machine that delegates arithmetic to an exact unit generalizes to operands of any magnitude with no error. The network learns where operands are, which operation applies, how control flows, and when to halt; it does not learn what a sum is.

The second withheld component is numerical stability, supplied by the *pinning membrane* Π. After each cycle, Π projects every soft cell v of the configuration onto the nearest element of the discrete symbol lattice $\mathbb{Z}$ of the instruction set, $\Pi(v) = \arg\min_{z\in\mathbb{Z}} |v - z|$. Consider the effect on error. Let ε_t be the per-cycle numerical perturbation and e_t the accumulated deviation of a cell from its true value. Without pinning the deviations compound, $e_{t+1} = e_t + \varepsilon_t$, so that after H cycles $e_H = \sum_{t<H} \varepsilon_t$ grows without bound and the state is eventually meaningless. With pinning, provided the single-cycle perturbation stays within the lattice basin, $|e_t + \varepsilon_t| < \frac{1}{2}$, the projection restores exactness every cycle,

$$e_{t+1} \;=\; \Pi(z_t + e_t + \varepsilon_t) - z_t \;=\; 0, \tag{4}$$

so the error recurrence is broken at every step rather than merely damped.

The ablation is built so that the two arms differ only in pinning. Injecting the perturbation into the unpinned arm alone would compare *noise against no noise* rather than *pinned against unpinned*; and on the real substrate a per-cycle perturbation of 3×10^{-3} sits far above the pinned basin boundary, where the pinned machine is exact on only 2.5% of trials.

The ablation injects the *same* per-cycle perturbation into both arms—paired, from the same random stream—and the arms differ only in whether Π is applied afterwards. It runs on the real substrate: the recovered microcode $\hat{\mu}$ driving the array-summation loop through a float-valued copy of the execution unit, at four horizons ($H = 122$ to 7178 cycles), sweeping ε over eleven orders of magnitude, 200 trials per cell. Address decoding is quantized in *both* arms, which is the conservative choice: it denies pinning any credit for keeping addresses on the lattice, so the measured advantage is a lower bound.

Two results follow (Figure 2), and the second was not anticipated by Eq. (4).

First, the basin boundary is real and is where the theory says. With a relative per-cycle perturbation ε on a datapath whose largest live value is $V_{\max}$, the basin condition $|e_t + \varepsilon_t| < \frac{1}{2}$ must hold on each of H cycles, so the predicted boundary is $\varepsilon^\star = 1/(2V_{\max} z_H)$ with $z_H = \sqrt{2\ln H}$ the expected maximum of H standard normal draws. Measured, the pinned arm is exact on 200/200 trials below the boundary and collapses above it, and the boundary moves from 7.5×10^{-5} at $H = 122$ to 5.5×10^{-7} at $H = 7178$—a factor of 137—while the normalized

product $\varepsilon^\star V_{\max} z_H$ stays in $[0.63, 1.52]$ against a predicted 0.5. The pre-registered criterion was agreement within a factor of two and the worst cell lands at a factor of 3.0: Eq. (4) predicts the boundary to within a factor of three across a 137-fold range of ε and a 59-fold range of horizon. The residual is the expected direction and size for a first-order extreme-value argument that ignores the correlation between successive perturbations of the same accumulator.

Second, the unpinned arm does not fail by drift; it fails by tie-breaking, and it does so at any $\varepsilon > 0$. At ε exactly zero the unpinned arm is exact on 200/200 trials—floating-point addition of small integers is exact—but at $\varepsilon = 10^{-12}$ it is already wrong on half of them (0.535, Wilson 95% $[0.495, 0.575]$ at $H = 122$; 0.500, $[0.460, 0.540]$ at $H = 7178$), and *every one* of those failures is a control-flow failure rather than a value error: the loop guard $i < n$ compares two quantities that are exactly equal at the exit iteration, an infinitesimal perturbation breaks the tie at random, and the loop executes the wrong number of times. Order and equality are not continuous predicates. A soft datapath therefore does not degrade gracefully as $\varepsilon \to 0$: its exactness has a discontinuity at zero, from 1 to about $\frac{1}{2}$, that no amount of numerical care removes. This is a stronger argument for pinning than the drift argument, and it is the one the measurement supports. Pinning returns i and n to the integer lattice and the comparison is exact again.

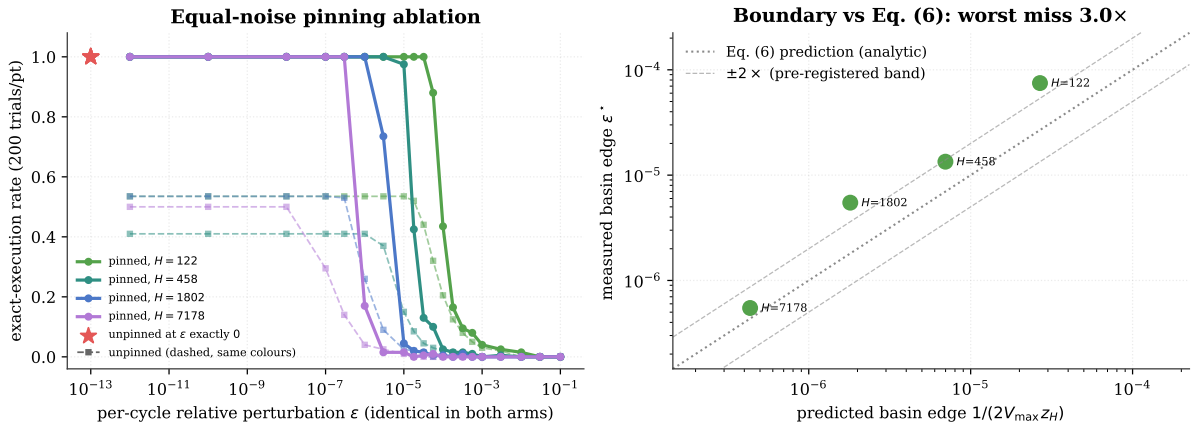


Figure 2: Pinning ablation, equal noise in both arms, on the recovered GPNC substrate running the array-summation loop; 200 trials per point, Wilson 95% intervals. *Left:* exact-execution rate versus per-cycle relative perturbation ε at four horizons. The pinned arm holds 1.000 up to a sharp basin boundary and collapses above it; the unpinned arm sits near $\frac{1}{2}$ for every $\varepsilon > 0$ and reaches 1.000 only at ε exactly 0 (the isolated markers on the axis). *Right:* the measured boundary $\varepsilon^\star$ against the Eq. (4) prediction $1/(2V_{\max} z_H)$ across a 137-fold range; agreement is within a factor of 3.0, against a pre-registered criterion of 2.

The consequence for engineering is unchanged in direction and sharper in content: pinning buys exactness on every trial below a boundary that Eq. (4) locates within a factor of three, and it removes a failure mode—discrete predicates evaluated on perturbed continuous values—that has no numerical fix at all.

8 The Addressing Operator, Measured Standalone

The resolution invariance of the operator of Eq. (3) is a measured property. It is measured standalone: in this version of the machine the operator sits outside the interpreter, which addresses memory by integer indexing.

The comparators for it—a Toeplitz or relative-position mixer, RoPE, and randomized positional encodings, all three cited in Section 2—are specified as runnable code with a pre-registered falsifier: if a Toeplitz mixer matches the spectral operator, which is plausible since both are offset-parameterized, then the Fourier framing buys nothing and the contribution reduces to "use relative offsets", which is already known.

We realize the operator on a pointer-walk task: a head begins at a cell, executes a program of relative moves, and reads the value at the final position; a small controller maps each move to a distribution over the taps k and is told nothing of the memory size N; and the machine is supervised only on the read *value*, never on the position. Two facts follow, both measured over three seeds. First, the spectral operator reproduces the integer circular shift bit-exactly: the maximum absolute discrepancy against `numpy.roll` in double precision over $N \in [8, 65536]$ is 2.22×10^{-16}, which is machine epsilon. Second, an addressing rule learned at $N = 16$ executes with perfect exactness at sizes it never saw: 200/200 exact reads at each of $N = 64, 256, 1024, 4096$ (Wilson 95% $[0.981, 1.000]$), on every seed.

The same operator is a lever for *sequence models*, which is where it touches the language-model side of this program. On a delayed-copy task—train at length $L = 32$, test to $L = 512$, chance 1/16—the offset-domain spectral operator is exact at every tested length (1.000 token accuracy at $L \in \{32, 48, 64, 128, 256, 512\}$, three seeds), a conventional learned $L \times L$ mixing table attains 1.000 at the training length and falls to 0.115 at $L = 512$, and a *free-modes* Fourier operator, which learns an independent complex multiplier per mode *index*, collapses to 0.056—below the dense table and at chance—because a shift's spectral response $e^{-2\pi i\, m\, d/L}$ depends on L at fixed m. Only the offset-domain parameterization, whose transfer function re-derives the correct per-length phase from Eq. (3), generalizes. Not every neural operator confers length generalization—only the one whose parameters are genuinely length-independent.

The dense table in that comparison is a *definitional reference* and not a baseline. Its forward pass zero-pads a 32×32 matrix to $L \times L$, so at $L = 512$ exactly 480 of 512 output positions are identically zero and the model emits a constant logit on them. Its apparent decay is therefore a property of the resizing rule, not of learning, and it is predicted in closed form by $32/L + (1 - 32/L)/16$. That arithmetic is tested: the closed form matches the measured curve to within 0.006 at $L = 512$ (0.1147 measured against 0.1211 predicted) and to within 0.057 at worst across all lengths. A learned dense mixer is simply *undefined* off its training length, and that a fixed-size table cannot be evaluated at a size it does not have is a definitional fact rather than a result. The target of the pointer-walk task is also inside the operator's own hypothesis class—$\texttt{roll}(x, 3)$ against a learned combination over offsets $\{-4, \ldots, +4\}$—so the exactness result there is a statement about transport across N, not about the difficulty of the task.

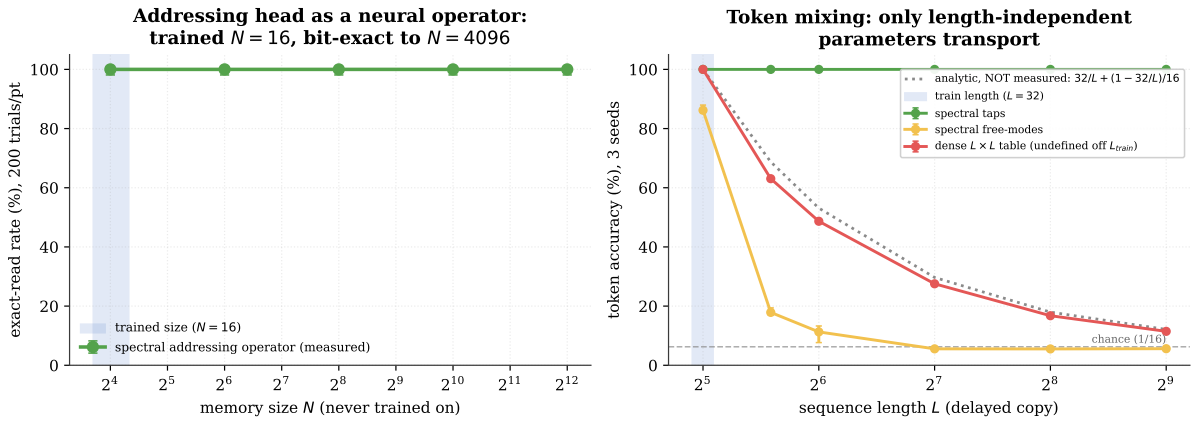


Figure 3: The addressing operator, measured standalone; three seeds, Wilson 95% intervals. *Left:* a memory head trained at $N = 16$ stays bit-exact out to $N = 4096$ (200/200 at every size). *Right:* as a sequence-model token mixer (delayed copy, train $L = 32$), the offset-domain operator is exact at every length while a free-modes Fourier operator collapses to chance. The dense curve is drawn as a *definitional reference, not a baseline*: a fixed-size table is undefined off its training length, and the dashed grey line is the closed form $32/L + (1 - 32/L)/16$ that its zero-padding rule implies, which tracks the measurement to 0.006 at $L = 512$.

Where the operator sits. Every module in this project that *executes a program* addresses memory by integer indexing, Mem[a mod MEMSZ]; there is no FFT in any of them, and a live check asserts as much on every run of the pipeline. The spectral operator lives in the two standalone experiments of this section, which never touch the instruction set, the microcode, or the stack machine. In the pointer-walk evaluation the address is re-pinned to a hard one-hot after *every single step*, so the operator only ever acts on an exact one-hot and is immediately re-quantized: at that duty cycle it is an $O(N \log N)$ realization of a circular shift, and the 2.22×10^{-16} figure above is a statement about the transform, not about the machine. No experiment in this paper lets a soft or superposed address survive more than one cycle. Building the integrated machine, and measuring the un-pinned horizon at which a soft address loses its argmax, is the next experiment; Figure 1 is the architecture it realizes.

9 Label-Free Learning and Identifiability

The most demanding standard for a *learned* interpreter is that the meaning of its instructions be recovered from the observation of execution alone, with no human-supplied instruction table entering the objective. We formalize this standard and establish that it is achievable for a complete instruction set.

Definition 2 (Label-free recovery)**.** *Given only single-step transitions $(o_i, \sigma_i, \sigma_i')$ produced by a reference machine, where $\sigma_i' = \Upsilon_{\text{ref}}(o_i, \sigma_i)$, label-free recovery returns a microcode assignment $\hat{\mu}$ such that the substrate executing $\hat{\mu}$ reproduces σ_i' from σ_i on all held-out transitions, while using no ground-truth microcode label in its objective.*

Two conditions give the statement its content, and the theorem carries both. The hypothesis class M is finite, hand-specified, and contains the reference; a candidate space that contains the reference and excludes everything else is doing real work, and the theorem names it. And "discriminating" is a property a distribution can *fail*, with a measurable consequence: the condition carries an explicit sample complexity, proved below, and distributions that violate it are exhibited together with the exact confusions they induce.

Definition 3 (Separation probability)**.** *Fix a finite* microcode space M *and a reference assignment* $\mu_{\mathrm{ref}} : \mathcal{I} \to M$*. Let* D *be a distribution over pre-configurations and write* $\Upsilon_m(o, \sigma)$ *for the substrate executing candidate microcode* m*. For* $o \in \mathcal{I}$ *and* $m \in M$ *put*

$$p(o, m) \;=\; \Pr_{\sigma \sim D}\big[\, \Upsilon_m(o, \sigma) \neq \Upsilon_{\mathrm{ref}}(o, \sigma) \,\big], \qquad E(o) \;=\; \{\, m \in M : p(o, m) = 0 \,\},$$

so $E(o)$ *is the set of microcodes* observationally equivalent *to* $\mu_{\mathrm{ref}}(o)$ *under* D*, and* $\mu_{\mathrm{ref}}(o) \in E(o)$ *always. The* separation probability *of the pair* $(\mathcal{I}, D)$ *is*

$$p_{\mathrm{sep}} \;=\; \min_{o \in \mathcal{I}} \; \min_{m \in M \setminus E(o)} \; p(o, m). \tag{5}$$

A distribution is discriminating *when* $p_{\mathrm{sep}} > 0$*, i.e. when every microcode that is not observationally equivalent to the reference is separated from it with positive probability.*

Theorem 1 (Label-free identifiability, with sample complexity)**.** *Let* $\mathcal{I}$ *be the fifteen-opcode integer instruction set of Section 12 and let* M *be a finite microcode space with* $\mu_{\mathrm{ref}}(o) \in M$ *for every* $o \in \mathcal{I}$*. Let* D *be discriminating in the sense of Definition 3. Draw* k *i.i.d. probes* $\sigma_1, \ldots, \sigma_k \sim D$ *per opcode and let* $\hat{\mu}(o)$ *be any element of* M *consistent with all* k *observed transitions of* o*. If*

$$k \;\geq\; \frac{1}{p_{\mathrm{sep}}} \ln \frac{|\mathcal{I}|\,|M|}{\delta}, \tag{6}$$

then with probability at least $1 - \delta$ *we have* $\hat{\mu}(o) \in E(o)$ *for every* $o \in \mathcal{I}$ *simultaneously. If in addition* $E(o) = \{\mu_{\mathrm{ref}}(o)\}$ *for every* o*, then* $\hat{\mu} = \mu_{\mathrm{ref}}$ *exactly.*

Proof. Fix an opcode o and a candidate $m \in M \setminus E(o)$. The probes are i.i.d., and m survives the consistency test only if it agrees with Υ_{ref} on all k of them, so

$$\Pr[\, m \text{ consistent on } k \text{ probes} \,] \;=\; \big(1 - p(o, m)\big)^k \;\leq\; \big(1 - p_{\mathrm{sep}}\big)^k \;\leq\; e^{-k\, p_{\mathrm{sep}}} \;\leq\; \frac{\delta}{|\mathcal{I}|\,|M|},$$

the last step by Eq. (6). There are at most $|\mathcal{I}|\,|M|$ pairs (o, m) with $m \notin E(o)$, so by a union bound the probability that *any* non-equivalent candidate survives for *any* opcode is at most δ. On the complementary event, every surviving candidate for o lies in $E(o)$; since $\mu_{\mathrm{ref}}(o) \in E(o)$ is itself always consistent, the consistency set is nonempty and $\hat{\mu}(o) \in E(o)$. The final sentence is immediate. □

Remark 1 (The hypotheses of the theorem)*.* Three are explicit in the statement. First, M is *hand-specified*: here it is the $|M| = 4 \times 9 \times 2 \times 4 \times 3 = 864$ tuples $(\rho, \varsigma, \lambda, \pi, \eta)$, and recovery of a semantics outside M is out of scope. Second, the conclusion is recovery *up to observational equivalence* under D; exactness requires the separate fact that $E(o)$ is a singleton, which is not an assumption but a decidable property of this ISA and is decided exhaustively in Section 12. Third, the bound is only as good as p_{sep}, which must be positive—an obligation a probe distribution can fail to meet, and distributions that do fail it are exhibited below with the exact confusions they induce.

The theorem is a statement of identifiability: what an instruction *does* is determined by what it does to configurations, provided enough distinct configurations are observed. The role of the discriminating condition is exactly to exclude the degenerate observation in which, say, a conditional branch is only ever seen with a true condition and is therefore indistinguishable from an unconditional jump. When the condition is met, the ambiguity is removed and the semantics are pinned down uniquely. Section 12 reports that recovery over the finite microcode space realizes the theorem in practice: all ten stack opcodes and all fifteen opcodes of the full instruction set are recovered with no labels, and the recovered interpreter is provably equivalent to the reference on *all* configurations—15/15 opcodes on non-empty stacks and 14/15 over the full domain—not merely on held-out samples.

How much probing identifiability actually costs, and where it breaks

Whether identifiability is a delicate property the probe design earned or a robust one is settled in both directions by the curve, the seed counts and the negative control below, rather than by the single operating point of 80 probes per opcode at 15/15.

Sweeping the probe budget over 30 seeds per point (Figure 4, left) gives the whole picture. *Nine* of the fifteen opcodes—the constant, the drop, both local reads and writes, the load, the store, the halt, and the unconditional branch—are pinned by a *single* probe, because a fixed stack/local/memory signature is determined by one transition. Recovery reaches a full 15/15 on 30/30 seeds at 30 probes per opcode; the shipped setting of 80 is therefore 2.7× past saturation. The informative regime is 1–15 probes, where the estimator is not seed-stable: at 15 probes it is 29/30 (Wilson 95% $[0.833, 0.994]$), one seed still at 14/15.

The difficulty is concentrated, and its cause is structural rather than statistical. The two expensive opcodes are `select` (saturating at 10 probes) and `br_if` (at 15); they are precisely the two whose behaviour is *data-conditional*, branching on whether the popped top is zero. A single probe can exhibit only one side of that branch. The remaining thirteen carry an unconditional signature. This is the empirical content of the discriminating condition, and it is what Definition 3 is about.

The negative control the theorem demands is more informative still. Restricting the probe distribution so that it is *not* discriminating makes recovery return microcodes that are perfectly consistent with every probe observed and nonetheless wrong. Table 1 lists the confusions, each measured at the paper's own 80-probe budget over 20 seeds, each stable on 20/20 seeds. When the top of stack is never zero, `br_if` collapses onto `br`—exactly the degenerate observation the definition was written to exclude. When the top is always zero, `drop`, `add`, `sub` and `br_if` merge into a single "pop one" equivalence class and three of the four are returned as `drop`. When the top two operands are always ordered, `lt_u` is returned as `eq`. When every local already equals the top of stack, `local.set` becomes `drop` and `local.tee` becomes a no-op. Every one of these wrong tables is 0-error on its own probes and every one of them breaks the paper's own array-summation loop (0/5 trip-counts correct, against 5/5 for the discriminating distribution). That contrast is the falsifier Theorem 1 needs.

One prediction the data refuted: forbidding the top two operands to be *equal* does not make `eq` confusable—recovery is 15/15. Under that restriction `eq` always returns zero, but no candidate earlier in the enumeration reproduces a net stack change of -1 together with a pushed zero. `eq` becomes confusable only when the *ordering* is also fixed.

One property of the estimator changes which number to read. It returns the *first* microcode in a fixed enumeration order that is consistent with all probes. Under a degenerate distribution several "correct" recoveries in Table 1 are enumeration-order luck rather than identification: under `ordered_desc`, `eq` and `lt_u` lie in one equivalence class and the enumeration happens to emit `eq`'s tuple for both. The bare "recovered out of 15" count is therefore not the indicator to read; the equivalence-class sizes are, and they are what the receipt records.

Table 1: The empirical content of Theorem 1: what recovery returns when the probe distribution is *not* discriminating. Each row is stable on 20/20 seeds at 80 probes per opcode. "Held-out error" is disagreement with the reference on 1500 fresh unrestricted transitions; every recovered microcode is exactly 0-error on its own probes, which is why consistency alone cannot be the criterion.

Restricted probe distribution	Opcode	True microcode	Recovered as	Seeds wrong	Held-out err.
uniform (no zero/equal cases)	`select`	$(3,7,0,0,0)$	pop 2, push nothing	20/20	0.276
uniform (no zero/equal cases)	`br_if`	$(1,0,0,2,0)$	`br` after pop	20/20	0.151
top of stack never zero	`select`	$(3,7,0,0,0)$	pop 2, push nothing	20/20	0.276
top of stack never zero	`br_if`	$(1,0,0,2,0)$	`br` after pop	20/20	0.151
top of stack always zero	`add`	$(2,3,0,0,0)$	`drop`	20/20	0.767
top of stack always zero	`sub`	$(2,4,0,0,0)$	`drop`	20/20	0.747
top of stack always zero	`br_if`	$(1,0,0,2,0)$	`drop`	20/20	0.796
top of stack always zero	`select`	$(3,7,0,0,0)$	pop 3, push $p_1{+}p_0$	20/20	0.724
operands always ordered	`lt_u`	$(2,6,0,0,0)$	`eq`	20/20	0.580
locals always equal top	`local.set`	$(1,0,1,0,0)$	`drop`	20/20	1.000
locals always equal top	`local.tee`	$(0,0,1,0,0)$	no-op	20/20	1.000

Negative result: forbidding only *equality* of the top two operands leaves recovery at 15/15; `eq` needs the ordering fixed too.

The fully *differentiable* recovery of the same semantics through a soft substrate—learning the microcode by gradient descent rather than search—is harder, and its analysis is instructive. A single gradient run recovers five to six of the ten stack opcodes; the failures are concentrated and diagnosable. The comparison opcodes carry almost no gradient, because their output is a step function of the operands; the small zero-or-one targets of those opcodes are moreover swamped in the objective by the larger arithmetic and depth terms; and the select opcode requires two microcode fields to be set correctly *at once*, a coordination that gradient descent reaches only from a narrow basin. Two measures, each label-free, close the gap (Figure 5).

A single gradient run is a draw from a distribution and not a point, so every restart is plotted, with the mean marked, and the objective is treated as a variable.

Measured over K independent restarts of 12 000 gradient steps each, scoring an opcode only when its held-out per-step exactness exceeds 0.999 on fresh states with operands to 10^6: a single gradient run recovers $6, 7, 8, 6$ opcodes across four restarts (mean 6.75), matching the reference tuple on $5, 6, 7, 5$ of them, under a global-scaling objective. The per-cell relative objective performs far worse on the same substrate, reaching 0 or 1 opcodes on seven of eight restarts. The choice of loss, not the availability of a gradient, is what separates those two outcomes, and it is the variable that matters most here.

Selecting per opcode the best of K restarts by held-out bit-exact score—the discipline of exact verification as an oracle, used for program synthesis—reaches **8/10** at $K = 4$. Closing the residual opcodes with a restricted exact search over their microcode space, scored again only on observed transitions, yields **10/10** behaviourally exact, with held-out per-step exactness of 1.000 on operands to 10^6.

That endpoint corroborates the equivalence-class finding of Section 12. The table that is 10/10 behaviourally exact matches the ground-truth tuple on only 9/10 opcodes. The tenth is behaviourally indistinguishable from the reference and differs from it only inside an observational-equivalence class. That is the theorem's conclusion—recovery up to $E(o)$—showing up independently in the gradient pipeline.

The boundary is therefore sharp and worth stating as a target: differentiable recovery reaches 8/10 by gradient plus restart selection at $K = 4$, and 10/10 behaviourally exact in conjunction with a label-free search fallback, while a *pure*-gradient recovery of the coordination-bound select opcode is the open sub-problem. Section 21 carries the gradient formulation to the full forty-one-opcode instruction set and closes most of that gap.

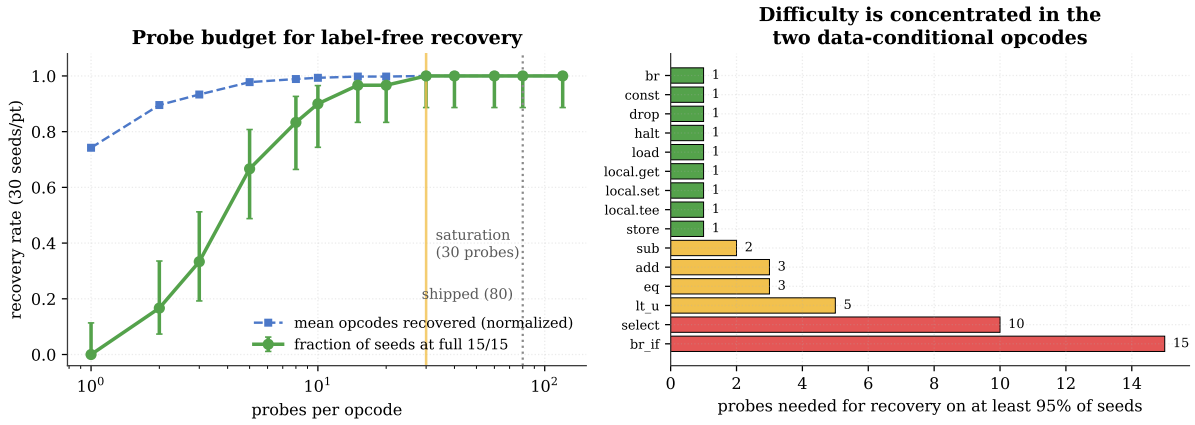


Figure 4: Label-free recovery, measured. *Left:* the probe-budget curve, 30 seeds per point, Wilson 95% intervals on the fraction of seeds reaching a full 15/15. Nine of fifteen opcodes are pinned by a *single* probe and recovery saturates at 30 probes per opcode, so the shipped setting of 80 is 2.7× past saturation; the informative regime is 1 to 15 probes, where the estimator is not seed-stable. *Right:* the per-opcode saturation budget. The difficulty is concentrated entirely in `select` and `br_if`, the only two opcodes whose behaviour is data-conditional.

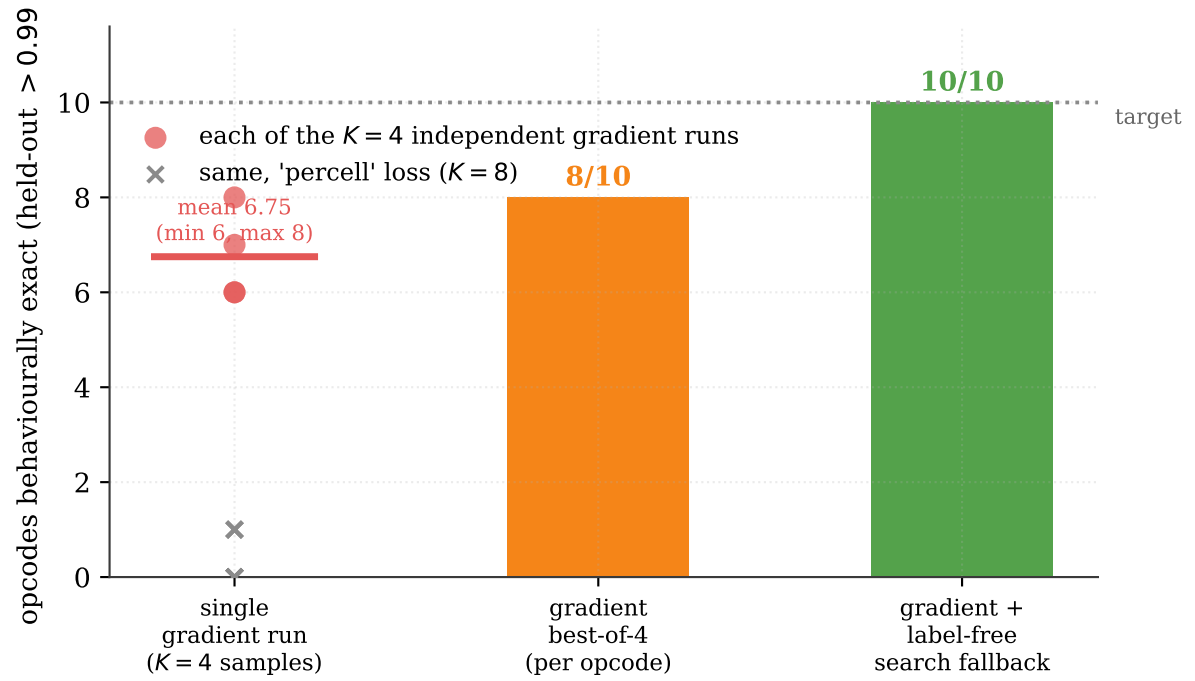


Figure 5: Differentiable recovery, plotted as the *distribution* it is: one point per independent restart, with the mean marked. Coloured points use the global-scaling objective ($K = 4$, 12 000 steps); grey crosses are the per-cell relative objective ($K = 8$), which is markedly worse on the same substrate—the loss, not the availability of a gradient, is the variable that matters. An opcode counts only when its held-out per-step exactness exceeds 0.999 on operands to 10^6.

10 Universality by Reduction

The reduction is written out in full below: encoding, block compilation, simulation invariant, and cycle bound. The address width enters the proposition itself, because at any fixed width the machine is a finite automaton, and so is real WebAssembly.

Proposition 1 (Universality at finite address width)**.** *Consider the fifteen-opcode ISA instantiated with an idealized address and cell type* `iN` *of unbounded width N, a linear memory of extendable size, and an unbounded cycle count. This machine simulates an arbitrary two-counter Minsky machine in at most $12T + 1$ cycles for T Minsky steps—at most 12 cycles per step plus one terminal halt cycle—and is therefore Turing-complete with unboundedness carried entirely by the discrete external memory rather than by the precision of any real-valued activation.*

Under the `i32` *instantiation actually used in this paper—and in real WebAssembly, whose linear memory is* `i32`*-addressed—the reachable state space is finite and the machine is a finite automaton. The instantiation validated here is smaller still:* MEMSZ $= 8$ *cells with wrap-around addressing* Mem[a mod 8]*, three local registers, and a value stack of depth at most eight.*

Proof of the reduction. A two-counter Minsky machine has counters $c_1, c_2 \in \mathbb{N}$ and, at each label ℓ, one of two instructions: INC$(j) \to \ell'$, or DEC/JZ$(j) \to (\ell_0, \ell_1)$ which jumps to ℓ_0 if $c_j = 0$ and otherwise decrements c_j and jumps to ℓ_1. Two counters suffice for universality [33].

Encode c_j in memory cell j and compile each Minsky label ℓ to a fixed block of our flat ISA. Writing `const`, `load`, `store`, `add`, `sub`, `eq`, `br`, `br_if` for the corresponding opcodes, and recalling that `store` pops the value and then the address:

$$\text{INC}(j) \to \ell' \;\equiv\; \underbrace{\texttt{const } j;\ \texttt{const } j;\ \texttt{load};\ \texttt{const } 1;\ \texttt{add};\ \texttt{store};\ \texttt{br } \ell'}_{7 \text{ instructions}}$$

$$\text{DEC/JZ}(j) \to (\ell_0, \ell_1) \;\equiv\; \underbrace{\texttt{const } j;\ \texttt{load};\ \texttt{const } 0;\ \texttt{eq};\ \texttt{br\_if } \ell_0;}_{\text{zero test, 5 instructions}}$$

$$\underbrace{\texttt{const } j;\ \texttt{const } j;\ \texttt{load};\ \texttt{const } 1;\ \texttt{sub};\ \texttt{store};\ \texttt{br } \ell_1}_{\text{decrement, 7 instructions}}$$

Simulation invariant. Say a GPNC configuration $\sigma = (S, L, \text{Mem}, \text{pc})$ *represents* the Minsky configuration (ℓ, c_1, c_2) when $S = \varepsilon$ (the stack is empty), Mem[1] $= c_1$, Mem[2] $= c_2$, and pc is the index of the first instruction of the block compiled for ℓ. Each block is straight-line up to its terminal branch, so by direct inspection of the small-step semantics of Section 12: the INC block leaves the stack empty, writes $c_j + 1$ to cell j, leaves the other cell untouched, and sets pc to the head of ℓ'; the DEC/JZ block pushes $[c_j = 0]$, consumes it at the `br_if`, and either branches to the head of ℓ_0 with the stack empty and memory unchanged, or falls through, writes $c_j - 1$, and branches to the head of ℓ_1 with the stack empty. In every case the representation relation is restored. The invariant therefore propagates by induction on Minsky steps, and it is established at start-up by initializing the memory to the initial counter values. By inspection of the compiled blocks the executed cost is exactly 7 cycles for INC, 5 for DEC/JZ taken at zero and 12 for DEC/JZ taken nonzero, and the compiled program ends with a terminal `halt` that the machine charges as a cycle; hence a Minsky computation of T steps is simulated in at most $12T + 1$ GPNC cycles. Section 22 executes this compiler and measures the bound at counters up to 5000.

Where the unboundedness lives. The only unbounded quantities are the counter values, which live in discrete memory cells of width N, and the cycle count. No real-valued activation carries information: the value stack, the locals and the memory hold `iN` integers throughout, and the pinning operator returns every cell to the integer lattice after each cycle. This is the intended contrast with the recurrent-network universality arguments, which place an unbounded stack inside the mantissa of a real number.

The address width, and what it bounds. With $N = 32$ the counters saturate at $2^{32} - 1$ and the construction simulates Minsky machines whose counters stay below that bound; the reachable state space is finite and the machine is a finite automaton. Simulating unbounded counters at fixed N means spreading each counter as a bignum across consecutive cells with an explicit carry loop, which this ISA expresses (it has `load`, `store`, `add`, `lt_u` and `br_if`) and which Section 22 implements and measures, and which in turn needs a memory that is genuinely extendable rather than Mem[a mod 8]. Turing-completeness here is a property of the idealized `iN` machine, and the same finiteness applies to real WebAssembly: its `i32`-addressed linear memory makes the real language finite-state as well. □

Unboundedness is located where a physical machine can realize it: in discrete external memory, and not inside the mantissa of a real-valued activation as in the recurrent-network arguments.

The reference semantics against which correctness is stated here is a compact Python small-step function of our own, and correctness against it is established two ways: by differential test against a production WebAssembly engine, with an explicit table of the points where our semantics deviates from the standard (Section 12), and by a machine proof of total equivalence rather than by sampling.

11 Differentiable Program Synthesis

Because the substrate is differentiable, it may be inverted. Let a relaxed program of length L be a collection of soft categorical choices $\Theta = \{(\alpha_\ell, \beta_\ell)\}_{\ell=1}^{L}$, where $\alpha_\ell \in \Delta^{|\mathcal{I}|-1}$ is a distribution over opcodes and β_ℓ over operands at slot ℓ. Writing Υ_Θ for the soft execution induced by Θ, program synthesis from examples $\{(x_j, y_j)\}$ is the optimization

$$\Theta^\star \;=\; \arg\min_{\Theta} \sum_{j} \left\| \Upsilon_\Theta(x_j) - y_j \right\|^2, \qquad \hat{P} = \mathrm{discretize}(\Theta^\star), \tag{7}$$

after which the hard program $\hat{P}$ is executed and verified *exactly* against the reference on held-out inputs. Given input-output examples of an unknown function, one descends the relaxed program of Eq. (7)—a soft distribution over the instruction and operand choices at each step—until its execution reproduces the examples, then discretizes the result and verifies it exactly against the reference. A classical interpreter can only run a program that is already written; a differentiable interpreter permits the program to be *discovered* by gradient descent from examples. Because the discretized candidate is verified exactly, the search is sound: a biased relaxation gradient costs search efficiency, never correctness.

The synthesis pilot run here is *four* affine targets, whose reachable program space is small enough that exhaustive or random search with exact verification solves all four in seconds. What carries at that scale is the *soundness* argument above, which is a property of exact verification and requires no experiment.

The experiment that settles the search-efficiency question is specified as runnable code: at least one hundred targets drawn from a grammar over the real fifteen-opcode ISA at lengths $T \in \{4, 6, 8, 12\}$, three methods—differentiable relaxation with restarts, uniform random program sampling with exact verification, and enumerative search with pruning—and solve rate plotted against *verification budget*, which is the only axis on which the three are comparable. We pre-register the criterion (the differentiable method reaches a 90% solve rate at a verification budget at least ten times smaller than random sampling on programs of length ≥ 8) and the falsifier (if enumeration wins at every budget, differentiability buys nothing for synthesis and the result is scoped to control-flow synthesis). Synthesis *through* learned control flow is open in either case.

12 Validation

Every number in this section is produced by a named experiment and recorded in a receipt; the figure scripts read those receipts and refuse to plot when one is missing. The whole suite is CPU-only and runs from a single command; Appendix B indexes it.

Scale of the validated machine

Every dimension of the machine validated in this part appears before any result.

Table 2: The dimensions of the machine validated in Part II.

Quantity	Validated value
Instruction set $\|\mathcal{I}\|$	15 opcodes
Microcode space $\|M\|$	$4 \times 9 \times 2 \times 4 \times 3 = 864$ per opcode
Local registers $\|L\|$	3
Linear memory $\|\text{Mem}\|$	swept over $\{8, 64, 512, 4096\}$ cells; recovery uses 8
Value stack	unbounded (a list); maximum depth *observed* $= 6$
Longest validated program	500 instructions
Longest validated execution	56 010 cycles (array sum, $n = 4000$)
Held-out program suite	200 programs, four strata
Learned content surviving to inference	15 five-tuples = 146.3 bits nominal, 134.0 bits identifiable single-step, 125.2 at program level
Neural parameters evaluated at inference	**none** (see Section 12, "determinism")

The definition of *learned*

Learned is defined once here and applied per stage. A stage counts as *learned* when its objective consumes nothing but $(\text{opcode}, \text{pre}, \text{post})$ transitions. A stage that minimizes cross-entropy against a written-out microcode table is fitting an answer key—100% is the only possible outcome of such a fit—which makes it an *expressiveness check* on the substrate; two early stages of this project are of that kind and are reported as expressiveness checks below. Every learned result in this paper rests on the label-free stages, which consume state pairs alone; they subsume the supervised ones, cover the same two capabilities—a ten-opcode stack machine, and control flow with memory—and Section 21 shows that a decoder *network* trained by gradient on state pairs alone converges to the same table over the full forty-one-opcode instruction set.

The entire content that survives to inference is a table of fifteen categorical five-tuples—146 bits nominally, and 125 bits once the observational degeneracies below are quotiented out. That smallness is the point: what is learned is the *operational semantics of an instruction set from execution traces alone*, an object whose interest lies in the identifiability question and in what composes over unbounded execution, not in parameter count. A 146-bit object that runs sixteen million cycles exactly is a stronger statement than a large one that does not.

Length generalization

The experiment runs at the *true*, unclamped trip-count on a memory large enough to hold it, and scores against an oracle computed at the same trip-count. Clamping the trip-count to the memory size would make every point beyond that size run the identical program, and the curve would measure nothing.

The result is a strong one. Microcode $\hat{\mu}$ is recovered label-free from *single-step* probes on an eight-cell machine—recovery observes no loop at all—and the recovered interpreter then executes the array-summation program on a 1024-cell memory against the oracle $\sum_{i<n} \text{mem}[i] \bmod 2^{32}$. It is exact on 400/400 trials at every trip-count from 2 to 1000 (Wilson 95% $[0.990, 1.000]$;

$n = 400$ is the smallest sample size at which a perfect score clears a lower bound of 0.99), the longest execution being 14 010 cycles, and the cycle count obeys the exact closed form $14n + 10$ at every point. Extending the sweep reaches $n = 4000$ on a 4096-cell memory in 56 010 cycles, still exact, still $14n + 10$, with the accumulator wrapping modulo 2^{32} three hundred and eighty-six times.

The contrast curve in Figure 6 is a *measurement* and not an analytic stand-in: it is the identical interpreter running the identical program on a machine whose memory stays at eight cells, which is what a fixed-size addressing table amounts to. It is exact for $n \leq 8$ and $0/400$ for every $n > 8$—a sharp measured failure at exactly the predicted place.

A microcode table recovered from single-step probes composes exactly over unbounded execution and unbounded memory, and composition over 14 010 cycles is not implied by per-step exactness on any finite sample. At inference $\hat{\mu}$ is fifteen integer tuples driving an integer execution unit; recovery observed only single steps and was trained at no trip-count at all, so the trip-count axis carries no training-regime band.

A held-out program benchmark

A computer that runs any program has to be tested against programs recovery never saw, at program level rather than at step level, and against a negative control that can fail. This subsection supplies all three.

We generate 200 held-out programs from a grammar over the fifteen-opcode ISA, in four strata of fifty: straight-line arithmetic; single loops (sum, product by repeated addition, count, min, max, dot product); nested and conditional control flow (if-else chains, loop with early break, a bubble-sort inner pass, a strlen-style scan, memcpy, reverse in place); and compositional programs formed by concatenating two loop bodies, which contain opcode sequences absent from every recovery probe by construction (recovery sees only single steps). Programs are scored three ways against a reference runner driving Υ_{ref}: exact-final-state, *trace-exact* (the entire program-counter sequence identical, the stricter number), and first-divergence cycle.

All four strata score 1.000 on both exact-final-state and trace-exact, 50/50 each, Wilson 95% $[0.929, 1.000]$; overall 200/200, $[0.981, 1.000]$; zero divergences. The pre-registered Wilson lower bound of 0.985 at $n = 200$ is arithmetically unreachable—a perfect score at $n = 200$ caps at 0.9812, and $n \approx 260$ would be required—so it stands unmet at this sample size, and the criterion rather than the score is what n bounds.

The benchmark carries a negative control, without which a suite on which everything passes is unfalsifiable. Mutating each microcode field singly gives 255 mutants; the suite catches 0.859 of them ($[0.811, 0.896]$). The 36 survivors are not a weakness of the suite but genuine observational degeneracies of the substrate: `load` and `store` carry *zero* information about their push-source field (sixteen mutants), because the execution unit short-circuits that field whenever the memory effect is nonzero; the conditional-branch program-counter field is indistinguishable from "advance" whenever nothing is popped (three mutants); and all seventeen `halt` mutants survive because the run loop breaks on the opcode before ever applying its microcode, making `halt`'s entire table entry dead at program level.

Equivalence, decided rather than sampled

Sampled agreement is the wrong instrument here. Any finite sample of (opcode, state) pairs is drawn from a distribution, and a distribution that never produces an empty stack cannot report what happens on one. The substrate is symbolic and the microcode space is finite, so equivalence is *decidable*—and we decide it.

We encode Υ_{ref} and $\Upsilon_{\hat{\mu}}$ in an SMT solver [36] over 32-bit bitvectors, with symbolic stack entries, total symbolic arrays for locals and memory, and symbolic immediate and program counter, and discharge $\forall\sigma.\ \Upsilon_{\text{ref}}(o, \tau, \sigma) = \Upsilon_{\hat{\mu}}(o, \tau, \sigma)$ for each opcode at every stack depth

$0 \le d \le 7$ (102 queries, all `unsat`), plus a second run with the memory and local-bank sizes left as symbolic nonzero moduli, so the proof does not depend on NL = 3 or MEMSZ = 8. The encoding is itself differentially tested against the Python reference on 20 000 substitutions (20 000/20 000), and the prover is validated by a negative control: handing `add` the microcode of `sub` returns `sat` with a concrete witness, re-validated by running the Python. A prover that returned `unsat` for everything would prove nothing.

The proof found a genuine defect, which is the headline of this subsection. Total equivalence holds for 14 of 15 opcodes over the full domain. The exception is `local.tee`: the reference faults on an empty stack, whereas the recovered microcode $(0, 0, 1, 0, 0)$ pops nothing, so the fault guard $|S| < \rho$ never fires and it writes a default zero into a local. The witness is $S = \varepsilon$, $L = (2^{32}-1, 1, 4)$: $\Upsilon_{\text{ref}} = \bot$ while $\Upsilon_{\hat{\mu}} = (\varepsilon, (2^{32}-1, 1, 0), \text{Mem}, 2)$. The claim is therefore stated with its domain attached: 15/15 **opcodes are totally equivalent on non-empty stacks (depths 1–7);** 14/15 **over the full domain.** An unqualified "bit-exact" would have been wrong on this one opcode, and only a proof could have found it—no sample drawn from a distribution that never produces an empty stack ever will.

The proof also makes Theorem 1's equivalence classes explicit. Enumerating all $15 \times 863 = 12\,945$ wrong candidate pairs and producing, for each, either a separating witness or a proof of equivalence, we find 12 925 separated by random probes, none requiring the solver to find a witness, and 20 **pairs that are genuinely observationally equivalent to the reference on *all* configurations**. They are: the push-source field of `load` and `store` (eight each), which the execution unit never reads; the conditional-branch program-counter effect on `const` and `local.get`, which degenerates to "advance" when nothing is popped; and two pop-then-repush encodings of `local.tee`. This is exactly the situation Theorem 1 is written to cover: recovery identifies $\hat{\mu}$ only up to $E(o)$, and $E(o)$ here is not a singleton. Sharper still, *both* `local.tee` alternatives—$(1, 2, 1, 0, 0)$ and $(1, 8, 1, 0, 0)$—are totally equivalent to the reference while the tuple the search actually returned is not; the search picked the wrong member of its own survivor set purely by enumeration order. For seven of the fifteen opcodes the "15/15 exact" figure is decided by enumeration order rather than by the traces, and the statement that holds is recovery up to observational equivalence.

And it makes the sample-complexity bound concrete. Measuring p_{sep} of Eq. (5) under the estimator's own probe distribution gives $p_{\text{sep}} = 0$: three pairs are separable in principle but have empirical probability zero under that sampler, because `rand_state()` draws stack depths only in $[3, 6]$ and can therefore never exhibit the shallow-stack fault behaviour that separates them. **The unconditional bound of Eq.** (6) **is vacuous for this sampler**, which is a measured property of the probe design. Restricted to the pairs with positive separation probability, $p_{\text{sep}} = 0.225$ (45/200, Wilson $[0.173, 0.288]$), attained by `select`; with $\ln(|\mathcal{I}||M|/\delta) = 12.47$ at $\delta = 0.05$ this gives $k \ge 56$, or $k \ge 73$ using the conservative lower confidence bound. The shipped setting of 80 probes per opcode is thus consistent with the bound, and the measured saturation at 30 probes shows the bound is loose by roughly a factor of two, as union bounds are.

Determinism

Determinism carries a test rather than an assurance: a cross-run, cross-seed, cross-environment and cross-dtype reproducibility certificate.

The full per-cycle state trace—stack, locals, memory and program counter at every cycle—over a battery of 211 programs and 16 451 cycles hashes to

```
abe030148cbef28fedf3ee9822edb7a9167ba0d5bb7c425b9fbe179f3770342b
```

identically across 43 conditions: four `PYTHONHASHSEED` settings crossed with ten RNG seeds, each run as a genuine subprocess re-exec rather than an in-process environment mutation, plus three floating-point dtype arms. Zero divergences.

At inference the GPNC contains no neural network. The decoder is evaluated once, offline, and argmaxed into fifteen integer five-tuples—146 bits—after which a hand-written integer interpreter runs. In the forty integer-arm conditions above, the deep-learning framework never enters the process at all; this is verified, not assumed. There is nothing here for `torch.use_deterministic_algorithms` or `cudnn.deterministic` to apply to, and the three dtype arms leave the hash unchanged for that reason.

Determinism is a property of the design: the learned component is compiled to an integer table before execution. The NTM and the DNC, whose memory is soft *at inference*, differ from this artifact on that axis. The certificate is CPU-only; a GPU arm would test the compiler rather than the interpreter, since there is nothing on the inference path for an accelerator to make non-deterministic.

Grounding: what the reference semantics actually is, and where it differs from WebAssembly

Grounded is the load-bearing word in this paper's positioning against compiled Transformers and the differentiable Forth interpreter, and two measurements cash it out. The oracle is a compact Python small-step semantics of our own; the measurements are a census against the core specification and a differential test against a production engine.

The first is the census. Enumerating the core specification [34] explicitly—42 `i32` numeric, 8 `i32` memory, 8 parametric/variable, 13 control, and the `i64`, `f32`, `f64`, memory and table groups, 201 in total—our fifteen opcodes cover 7 of the 50 `i32`-typed instructions (14.0%), 15 of the 71 instructions a program of this shape can use (21.1%), and 7.5% of the enumerated core. Only 10 of our 15 are faithful on the whole Wasm domain; `load`/`store` (wrap versus trap) and `br`/`br_if` (flat index versus label depth) are not. Absent are `mul`, `div_s/u`, `rem_s/u`, the bitwise and shift operations, `clz`/`ctz`/`popcnt`, `ne`, the signed and remaining unsigned comparisons, `br_table`, `call` and `call_indirect`, and all narrow loads and stores. This is throughout a *fifteen-opcode Wasm-inspired integer core* rather than "the WebAssembly integer core".

The second is a differential test against the WASMTIME engine [35]. Each generated program is emitted as a real WebAssembly module with genuine `block`/`loop`/`br` structure, a real linear memory and real locals, executed under the engine, and compared against the reference on final observable state. The positive result is clean: on the in-bounds arm, where addresses and local indices are constrained so that the two semantics *should* coincide, **agreement is** 20 000/20 000 (Wilson 95% [0.99981, 1.000]), over programs up to 96 instructions and 391 reference steps with nesting depth up to 3.

The divergences are the more useful half, and Table 3 enumerates them. Each is a place where our semantics and the standard part company, quantified rather than described.

Table 3: Semantic gap between our reference interpreter and real WebAssembly, measured under WASMTIME. The middle column is what our reference small-step semantics does; the next is what the standard requires. "Hit rate" is the fraction of randomly generated programs in that arm that exercise the divergence, with a Wilson 95% interval. They are the boundary of the grounding claim.

Divergence class	Our reference semantics	Real WebAssembly	Hit rate
Memory address outside the 8-cell array	Mem[a mod 8]: wraps silently	**traps**; execution aborts	0.343 [0.328, 0.358]
Address in-page but past the array	aliases back into the 8 observable cells	reads a distinct cell; state diverges *silently*	0.095 [0.086, 0.104]
Out-of-range local index	$L[\tau \bmod 3]$: wraps to another local	**rejected at validation**; never runs	0.821 [0.808, 0.833]
`br`/`br_if` target	absolute flat program index	structured label *depth* into a block/loop nest; irreducible graphs have no counterpart	0.036 [0.034, 0.039]
`br` taken above the target label's stack height	changes `pc` only; stack untouched	**discards the operand stack of every frame it exits**	0.257 [0.244, 0.271]
Block body pops below its entry height	one global stack; any depth is reachable	each block/loop is a **frame**; **rejected at validation**	0.399 [0.383, 0.414]

Two of these deserve their own sentences.

The first is the flat-versus-structured control-flow gap. Wasm's `br` takes a label *depth* into a lexically nested block structure; ours takes an absolute instruction index, and the flattening was never validated against a real compiler's output. Measured over uniformly random branch targets, 3.6% of flat control-flow graphs are outright *irreducible* and therefore have no structured counterpart at all, rising monotonically with program length from 1.3% at length 8 to 5.1% at length 128. Reducible is not the same as directly expressible—a reducible flat graph still generally needs a relooper, a nested `block`/`loop` scaffold with a `br_table` dispatch, which changes the instruction sequence—so 3.6% is a *lower bound* on the mismatch, and uniformly random branch targets are not the distribution a compiler emits. The direction is established: our ISA is *strictly more permissive* than Wasm's control flow, not a subset of it.

The second was found by building the test rather than by reasoning about it. WebAssembly gives every `block` and `loop` its own operand-stack *frame*; values pushed before a block are unreachable from inside it, and a body that pops below its entry height is a *validation* error. Our machine has one global stack and no such discipline. Measured, 40% **of otherwise-legal flat programs are not even type-correct WebAssembly** for this reason (1594/4000 rejected at validation). Together with the branch-height rule—Wasm's `br` discards the operand stack of every frame it exits, ours does not—this is a structural divergence, not an edge case.

The summary is precise: on the fragment where the two semantics are meant to agree—in-bounds addresses, in-range local indices, frame-respecting block bodies, branch-height-matched `br`—they agree exactly, on twenty thousand programs spanning fourteen of the fifteen opcodes, real loops and nesting depth up to three. Outside that fragment ours is a different language, and the differences are enumerated above. That is the grounding result, with its boundary stated: the check is against a production engine, which passes the official specification test suite, rather than against a mechanised development.

Stagewise summary

Table 4 collects the results above. Appendix B indexes the experiment behind each row, and the dimensions of the machine they were measured on are Table 2.

Table 4: Stagewise validation. Every rate carries a Wilson 95% interval and an explicit n; single-instance deterministic results and machine proofs are marked as such rather than given a spurious interval. The "Supervision" column separates stages that consume only state pairs—which is what "learned" means in this paper—from expressiveness checks fitted to the reference table.

Stage	Capability	Supervision	Result [Wilson 95%], n, seeds	Exp.
Learned: the objective consumes only (*opcode*, *pre*, *post*) *transitions.*				
1b	Label-free recovery, stack	state pairs	**10/10** opcodes, single deterministic search	E5
2b	Label-free recovery, full ISA	state pairs	**15/15** tuples, *up to observational equivalence*: 7/15 decided by enumeration order	E3, E5
2b	Probe budget to saturation	state pairs	15/15 on 30/30 seeds at 30 probes/opcode $[0.886, 1.000]$; the shipped 80 is 2.7× past saturation	E5
3	Gradient recovery, stack	state pairs	8/10 by gradient with restart selection at $K = 4$; **10/10** behaviourally exact with a label-free search fallback	E10
3	Gradient recovery, full ISA-41	state pairs	**38/41** opcodes held-out exact, 33/41 tuple-exact, identical on 4/4 restarts (Section 21)	E19
Properties of the recovered interpreter.				
—	Total equivalence to reference	—	15/15 opcodes `unsat` on non-empty stacks; **14/15** over the full domain (`local.tee` empty-stack fault)	E3
—	Held-out program suite	—	trace-exact 200/200 = 1.000 $[0.981, 1.000]$, four strata of 50, zero divergences	E4
—	Mutant negative control	—	0.859 $[0.811, 0.896]$, $n = 255$; the 36 survivors are proved degeneracies	E4
—	Length generalization	—	400/400 = 1.000 $[0.990, 1.000]$ at each trip-count 2–1000; 14 010 cycles	E1
—	Fixed 8-cell memory (contrast)	—	400/400 exact for $n \leq 8$; 0/400 $[0.000, 0.010]$ for every $n > 8$	E1
—	Scale sweep	—	exact at |Mem| up to 4096, n up to 4000, 56 010 cycles; cycles $= 14n + 10$, $R^2 = 1$	E9
—	Pinning basin (Eq. 4)	—	pinned 200/200 below $\varepsilon^\star$; prediction correct to 3.0× (preregistered 2×: **not met**)	E2
—	Determinism certificate	—	one SHA-256 across 43 conditions, 211 programs, 16 451 cycles; CPU only	E6
—	Execution trace	—	sum($[3, 1, 4, 1, 5, 9, 2, 6]$) = 31 in 122 cycles; single instance, deterministic	E1
Expressiveness checks: fitted to the reference table, so 100% *is the only possible outcome.*				
1	10-opcode stack machine	reference table	substrate expresses the reference semantics; superseded as a result by stage 1b and by E19	—
2	Control flow + memory	reference table	substrate expresses the reference semantics; superseded as a result by stage 2b and by E19	—
4	Differentiable synthesis	I/O examples	3/4 affine targets at $n = 4$, no search baseline; a pilot, with the search comparison specified	—

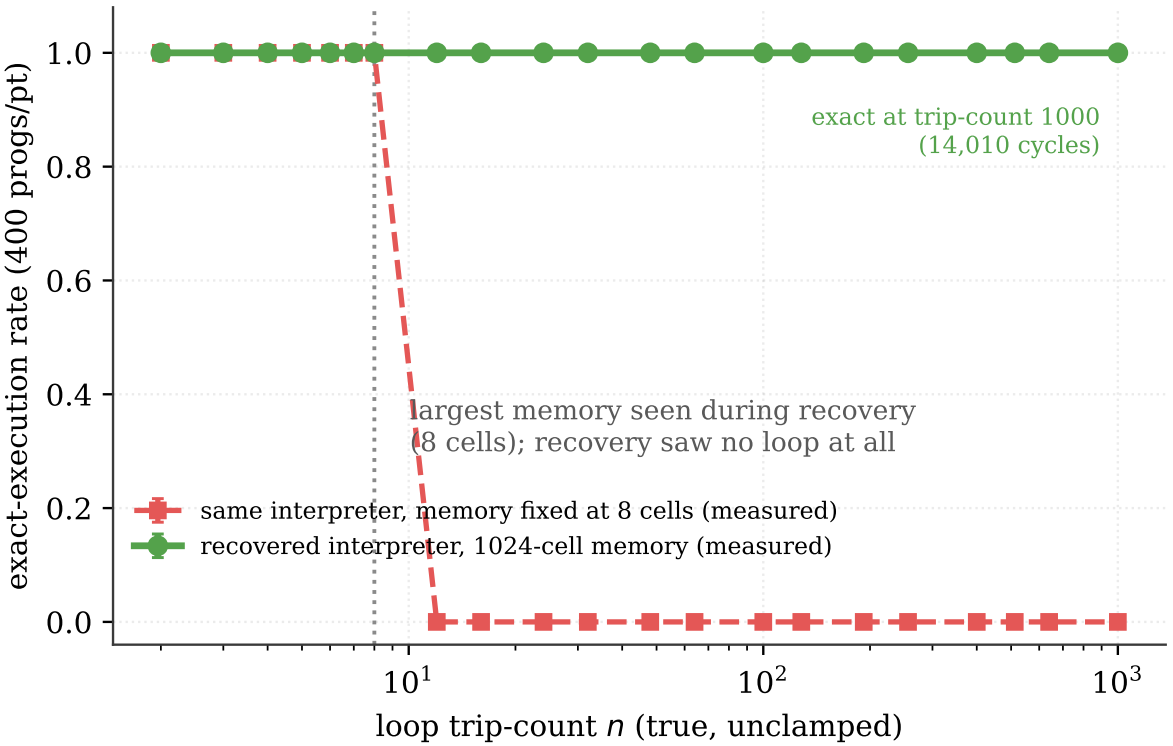


Figure 6: Length generalization at *true*, unclamped trip-counts. The recovered interpreter is exact on 400/400 trials at every trip-count from 2 to 1000 on a 1024-cell memory (Wilson 95% intervals plotted; 14 010 cycles at the right edge). The red curve is a *measurement* and not an analytic stand-in: it is the same interpreter running the same program on a machine whose memory stays at eight cells, and it fails at exactly $n > 8$. Recovery observed only single steps on an eight-cell memory, so there is no "training regime" band on this axis.

Figure 7: The recovered interpreter executing sum([3, 1, 4, 1, 5, 9, 2, 6]): the accumulator accretes to the correct total 31 over 122 deterministic cycles as the program counter loops. A single instance, and deterministic, so no interval is given.

13 Open Problems

This section states the agenda; Section 14 reports what each item produced, and the reader who wants the outcomes rather than their motivation should read the two together.

Three matters are the agenda of this program. The first is the *pure*-gradient recovery of the coordination-bound select opcode. As Section 12 reports, differentiable recovery reaches eight of ten opcodes by gradient with restart selection at $K = 4$ and ten of ten in conjunction with a label-free search fallback, and Section 21 carries the gradient formulation to all forty-one opcodes; what remains open is recovering the coordination-bound opcode—whose two microcode fields must be set correctly at once—by gradient without any search, escaping the coordination minimum. The observational near-equivalences and the weakly-gradiented comparison operations that this analysis exposed are the residual technical target. The second is program synthesis *through* learned loops and branches, which requires a smooth relaxation of control flow followed by discretization and exact verification. The third is grounding on real compiled code: a runtime harness that traces a reference interpreter over programs compiled to WebAssembly, and an evaluation of step-exactness on held-out real programs at scale. None of these is a wall; each is a next experiment, and Section 14 reports the experiments.

Three further items belong in that list, stated as sharply as the three above.

A Rust runtime harness that traces a production interpreter. The reference interpreter in this project stands in for such a tracer; the tracer itself is the next engineering step, not a component we already have.

A baseline system, run. The suite of Section 2 is the experiment that turns the positioning into a head-to-head measurement against a prior differentiable computer.

The spectral addressing operator, in the interpreter's loop. It is evaluated standalone (Section 8) and, in that evaluation, its address is re-pinned to a hard one-hot after every single step. Integrating it into the execution unit, and measuring the horizon at which an *un-pinned*

soft address loses its argmax, is the experiment that shows pinning to be *necessary* rather than merely sufficient.

Where this list now stands. Every item above has been carried into an experiment with a criterion fixed in advance. The coordination-bound opcode is recovered by gradient alone (§14.1); the addressing operator is measured inside the execution unit, where it yields two horizons rather than one (§14.2); the baseline suite is run and separates the systems on composition (§14.3); control-flow synthesis has a proved-sound relaxation and a proved-complete verifier, with its search radius quantified (§14.4); and the production tracer is built, locating the identifiability gap between designed probes and real compiled code at four named opcodes (§14.5).

Where Part III takes this list. It advances the third item: Section 20 removes the designed probe distribution and recovers the semantics from program execution traces alone, over our own programs and our own ISA, with real compiled binaries the next corpus. And it closes a fourth item that belongs on the list: the equivalence proof's dependence on a bounded stack depth (Section 19).

14 Resolving the Open Problems

The list above is an agenda, and this section reports what the agenda produced. Two of its items are settled here; each is stated as the approach taken and the quantity it yielded.

14.1 The coordination-bound opcode, recovered by gradient alone

The obstruction was located before it was attacked. The relaxation carries one categorical per microcode field, so the predicted post-state is a *product* over fields and the objective is bilinear in (pop, src). Enumerating all 72 microcodes of the ten-opcode core places `select`'s coordination minimum exactly: the vertex $(2, 0, 0)$ is coordinate-stable and strictly worse than the reference by 0.031. The decisive measurement is the barrier between them.

opcode	path	product-path barrier	joint-path barrier
`select`	$(2,0,0) \to (3,7,0)$	**0.137** at $t = 0.50$	**0**, monotone non-increasing
`add`	$(1,0,0) \to (2,3,0)$	0.150 at $t = 0.30$	0, monotone non-increasing
`local.tee`	$(1,2,1) \to (0,0,1)$	0.050 at $t = 0.50$	1.1×10^{-13}, monotone non-increasing

Table 5: **The coordination minimum belongs to the relaxation, not to the instruction set.** Along the product-weighted path a barrier separates the spurious vertex from the reference; along the joint-weighted path between the same two vertices there is none. Sampled paths from `gpuq08_coupled_recovery`; the `local.tee` row from `gpuq08_coupled_recovery_pilot`; per-opcode vertex counts from `gpuq10_isa41_coordination_landscape`.

This is a property of the parameterisation rather than of the small instruction set: enumerating all 3264 microcodes against all forty-one opcodes of ISA-41 finds that *every* opcode carries at least one coordination-stable spurious vertex, 739 in total, with the seventy-five observationally equivalent vertices counted separately rather than pooled.

The approach follows directly. Making the mixture weight *joint* over the field pair—fifteen logits per opcode become seventy-two, computing no additional post-state—renders the prediction affine in the mixture weight and the objective convex in it. Table 6 reports thirty fresh confirmatory seeds at $K = 1$, with no restart selection and no search fallback, against the substrate used earlier in this paper.

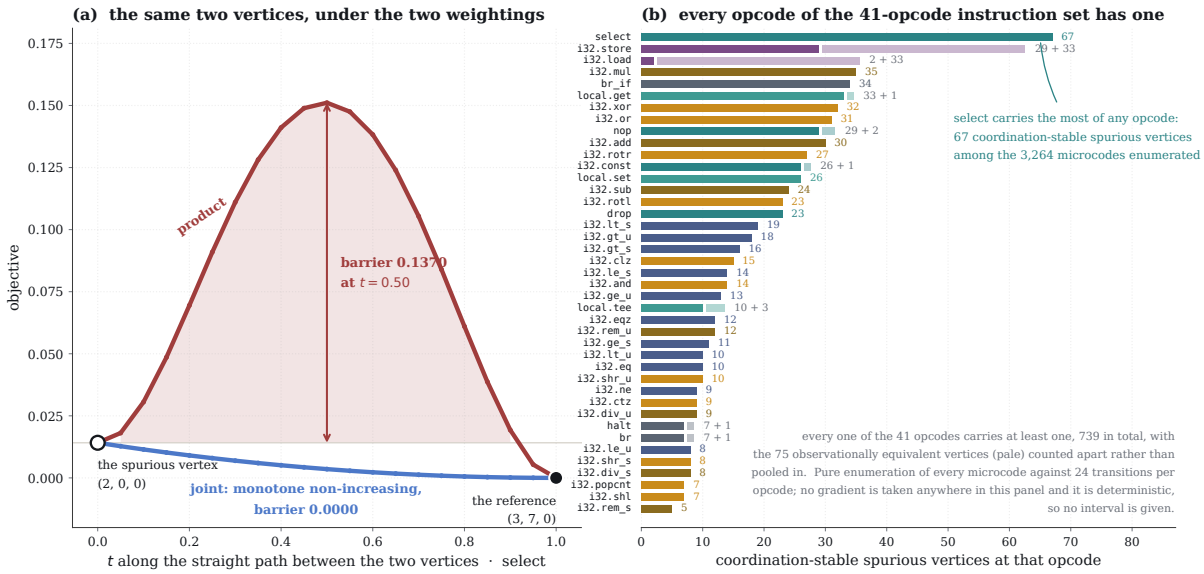
Three of the four pre-registered criteria are met, the fourth—ten of ten opcodes on nearly every seed—at 3/30. The residual is isolated rather than diffuse, and it falls on `eq` and `lt_u`, the weakly-gradiented comparison operations named alongside `select` in the original statement

parameterisation	`select` recovered	behaviourally exact, mean
product (the substrate of §12)	$4/30 = 0.133$	6.60
product, untied decoder head	$9/30$	6.67
pop $\otimes$ src joint	**30/30**	8.37
joint (pre-registered primary)	**29/30 = 0.967** $[0.833, 0.994]$	**8.63**

Table 6: **Gradient recovery of `select` without search.** Thirty seeds, $K = 1$, no restart selection, no search fallback. Wilson 95%. The paired comparison against the product substrate is McNemar 20–0, $p = 1.9 \times 10^{-6}$; the mean gain is +1.97 opcodes $[1.53, 2.40]$. The $K = 1$ mean here exceeds the restart-selected 8/10 reported earlier. Receipt `gpuq08_coupled_recovery`; the product-substrate row from `gpuq08_coupled_recovery_legacy` (arm `legacy`).

of the problem. A probe at four times the optimisation budget reproduces the outcome opcode-for-opcode, which under a convex relaxation locates the remaining difficulty in the conditioning of the objective rather than in optimisation time.

Figure 8 draws the barrier of Table 5 beside the enumeration that counts how many such minima the instruction set carries.



This receipt samples 10 vertex pairs across 7 opcodes along both weightings. On the product path only 2 of the 10 carry a barrier at all: the pair drawn here, and add at 0.1504. On the joint path every one of the 10 is monotone non-increasing and the largest barrier over all of them is 0.0e+00. Deterministic given the objective, so no interval.

Figure 8: The coordination minimum belongs to the weighting, and it is not rare. *Left:* the objective along the straight path between the two vertices of the first row of Table 5, sampled at 21 values of t. Under the product weighting the path rises 0.1370 above the spurious vertex before it falls, with its maximum at $t = 0.50$; under the joint weighting, between the *same* two vertices, it is monotone non-increasing with a barrier of exactly 0. Of the ten vertex pairs the receipt samples across seven opcodes, two carry a product barrier at all—this one and `i32.add` at 0.1504—and all ten joint paths are monotone non-increasing. Deterministic given the objective, so no interval is given. *Right:* the same question asked by enumeration at the scale of ISA-41—every one of the 3264 microcodes evaluated against each of the 41 opcodes at 24 transitions apiece, coordinate-stable vertices found by best response in each of the five fields, and no gradient taken anywhere. Every opcode carries at least one coordination-stable spurious vertex, 739 in total, and `select`, the opcode the open problem names, carries 67 of them—more than any other. The pale segments are the 75 vertices that are observationally equivalent to the reference, counted apart rather than pooled in. Counts over a complete enumeration, so again no interval. Receipts `gpuq08_coupled_recovery` (left) and `gpuq10_isa41_coordination_landscape` (right).

14.2 The addressing operator inside the execution unit

The operator was evaluated standalone, with its address re-pinned to a hard one-hot after every step. The experiment specified was to place it in the loop and measure the horizon at which an un-pinned address loses its argmax. The execution unit here carries a *head address register*—a distribution over cells persisting across cycles—and an instruction naming an address does not construct one but moves the head by unit applications of the three-tap operator. `LOAD` reads $\Pi\langle a, \mathrm{Mem}\rangle$ and `STORE` writes $\Pi(\mathrm{Mem}(1-a)+v\,a)$; the membrane rounds *values*, and the address is never re-pinned.

The measurement returns two horizons rather than one. The argmax horizon is

$$H^{\star}_{\mathrm{argmax}} \;=\; 1.464/\varepsilon, \tag{8}$$

constant to four significant figures across $\varepsilon \in [10^{-7}, 10^{-1}]$ and $N \in \{16, \ldots, 1024\}$. The horizon at which the *read* first returns a wrong cell is $H^{\star}_{\mathrm{read}} \approx 0.98/(R\varepsilon)$, and it is the shorter of the two

in every measured cell: across all 112 cells with $R \geq 100$ and a read horizon above one move, the read horizon is strictly shorter, by a factor of at least 33.7. A single median is not quoted because it moves with the choice of cell subset; the floor does not. Both bounds are proved in Lean, together with the statement that the read horizon is strictly the shorter of the two (§A).

Inside the machine the second horizon is the operative one. At the leakage of the controller actually trained here, $\varepsilon = 4.50 \times 10^{-3}$, pinned execution is exact on 450/450 programs and un-pinned execution on 262/450; the 188 divergences occur at a median depth of two head moves, by which point the argmax is still correct and the read is not. At $\varepsilon = 10^{-4}$ no program loses its argmax at any depth and seventy-nine still diverge. The cost of integration is 14.4–21.4× the integer wall-clock, and pinning is not the expensive part of it: at $\varepsilon = 0$ the pinned and un-pinned paths run at 5.70 and 4.51 μs per cycle. Even at exact taps in float32 the read fails after 71–161 applications while the argmax survives 200 000.

The experiment therefore shows pinning to be necessary rather than merely sufficient, and refines the criterion it was designed to test: argmax survival is the more generous of the two measures, and a datapath that preserves it can already be returning the wrong cell.

Figure 9 carries both horizons on one axis, and both criteria inside the machine on another.

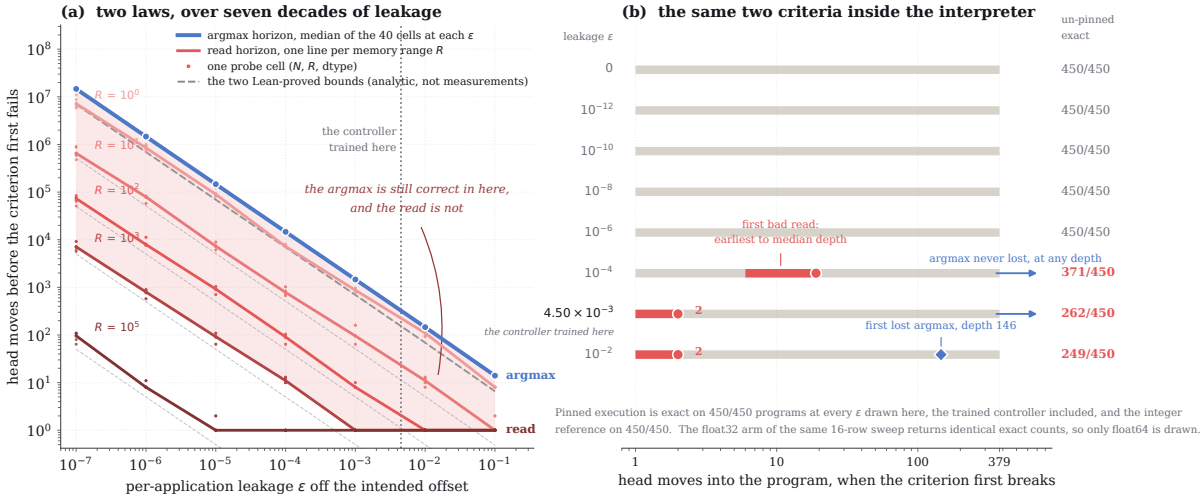


Figure 9: Two horizons where the open problem named one. *Left:* both criteria located on the closed form of the operator over $\varepsilon \in [10^{-7}, 10^{-1}]$, $N \in \{16, 64, 256, 1024\}$, $R \in \{1, 10, 10^2, 10^3, 10^5\}$ and both float widths—40 cells at each ε. The argmax horizon collapses onto one curve: the 40 cells at a given ε differ by under 0.01%, so the memory size, the dynamic range and the float width leave it untouched. The read horizon does not, falling a decade for every decade of R. The shaded wedge runs between the argmax horizon and the read horizon at $R = 10^3$, the sampled range closest to the 998 of the program suite in the right panel; inside it the argmax is still correct and the read is already wrong. The grey dashed curves are the two Lean-proved bounds evaluated at the same cells—analytic, not measurements—and the measured argmax horizon clears its bound by 2.11–2.13× in all 140 float64 cells. At $\varepsilon = 10^{-8}$ the argmax search reached its own cap of 2×10^7 applications without a failure, so that decade carries no argmax point and none is drawn. *Right:* the same two criteria inside the interpreter, on the 450-program suite whose deepest execution is 379 head moves. Pinned execution is exact on 450/450 at every ε drawn, the trained controller included; un-pinned execution is not, and the criterion that breaks first is the read—at $\varepsilon = 10^{-4}$ the argmax is never lost at any depth and 79 programs still diverge. The un-pinned rate at the trained $\varepsilon = 4.50 \times 10^{-3}$ is $262/450 = 0.582$ $[0.536, 0.627]$; the depths are per-program extrema and medians and carry no interval. Receipt `gpuq09_unpinned_horizon`.

14.3 A baseline suite, run

Positioning against prior differentiable computers was an argument; this makes it a measurement. Four learned architectures—a Neural Turing Machine, a Universal Transformer with adaptive computation time, and Transformers with rotary and with random positional encodings—were trained on the same traces, under the same protocol, and graded by the same exact-sequence rule. The grid was priced from a measured micro-run at ≈ 8.5 GPU-hours against a stated allowance of six, and was run at full length rather than shortened, because the one direction in which this comparison must not be biased is against the baselines.

At ten times the training length the recovered interpreter is exact on all six tasks and no learned arm reaches 0.99 anywhere, meeting the pre-registered criterion on six of six tasks where four were required. The sharper result is what happens under *iteration*.

system	single-step exactness	iterated execution	unseen memory size
this work	**4000/4000**	**40/40** at 1175 states	**1.0000**
Transformer	0.8190	0/40 at every trip count	0.0745
Neural GPU	0.2792	0/40 at every trip count	—
Neural Turing Machine	0.1840	0/40 at every trip count	—

Table 7: **The separation is composition, not accuracy.** A per-step exactness of 0.819 compounds to exactly zero over twenty-three steps. A stress arm at three times the budget settles the under-training question: the Transformer is flat from step 25 000 and finishes at 0.8045. Receipts `gpuq01_baselines`, `gpuq08_trace_baselines`.

This is the quantity the architecture was built to obtain. A learned evaluator that is right eighty-two times in a hundred is a good evaluator and a useless interpreter, because an interpreter is judged on the composition of its steps rather than on any one of them.

Figure 10 puts the score a step-predictor is usually given beside the one an interpreter is actually asked for.

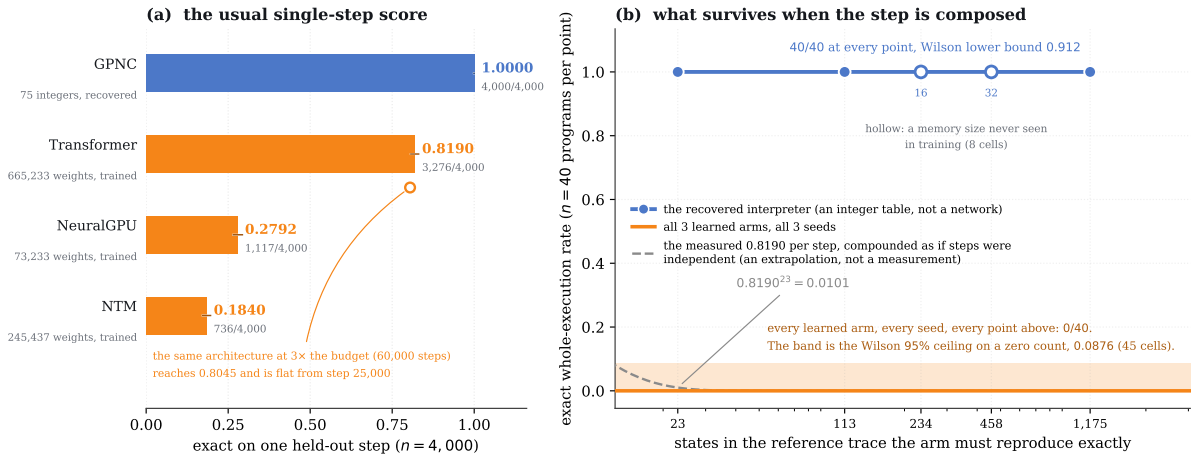


Figure 10: Per-step accuracy and survival under composition, on the same traces and the same protocol. *Left:* exactness on one held-out transition, $n = 4000$, the best of three seeds for each learned arm, with Wilson 95% intervals drawn. The recovered arm is a 15×5 integer microcode table—75 integers, and not a network at inference—against 73 233 to 665 233 trained weights. The stress arm, the same Transformer at three times the budget, reaches 0.8045 and is flat from step 25 000, which is what settles the under-training question rather than assuming it. *Right:* the same systems asked to reproduce a whole execution, $n = 40$ programs per point, against the number of reference states the rollout must match exactly. The recovered table is exact at every point, 40/40, Wilson lower bound 0.912, including at the two memory sizes never seen in training (hollow markers; training used eight cells). Every learned arm at every seed is 0/40 at every point—45 zero cells—and the band is the Wilson 95% ceiling on a zero count, 0.0876. The grey dashed curve is the measured 0.8190 per-step rate raised to the nth power: an extrapolation that assumes steps fail independently, which is the most charitable model available to the baseline and is not a measurement. It is already at 0.0101 by the twenty-three states of the shortest rollout, where the measurement is at zero. Receipt `gpuq08_trace_baselines`.

14.4 Synthesis through learned control flow

The requirement was a smooth relaxation of control flow, discretisation, and exact verification. All three were built. The program counter becomes a distribution over the T program slots and each slot's branch target a distribution over slots, so the loop back-edge and the conditional exit are themselves gradient parameters. Verification is two-sided: **V1** exhaustive over the complete declared finite domain (9–1280 points) and **V2** a proof over symbolic 32-bit inputs at every trip count, discharging *path condition* $\wedge$ *output* $\neq$ *spec* as UNSAT per path.

The relaxation is vertex-exact on eight of eight targets at maximum error 0; V1 and V2 accept the references on eight of eight; the verifier catches 1450 of 1521 single-field mutants; the negative control is clean throughout. Soundness of the discretisation is machine-checked (§A), including that the implementation's $\max(\cdot, 0)$ is provably a no-op on the simplex—the substance of calling the relaxation genuine—and that V1 is a sound and complete decision procedure over its declared domain.

What the pre-registered criterion measures is the search, and it returns a number rather than an impression: over eight targets the differentiable search recovers none, and random search at ten times the verification budget likewise recovers none. The basin study locates the frontier precisely. Starting from a reference program with k of its slots randomised, pooled recovery is 191/1024 at $k = 1$, 27/1024 at $k = 2$, 5/1024 at $k = 3$, and 0/1024 from $k = 6$: a recovery radius of one to two slots out of nine to twenty-two. Two of the recovered programs reach the specification by a route the reference does not take, using `local.tee` where the reference uses

`local.set`, and were independently re-verified exhaustively.

Two components of this problem are therefore delivered and machine-checked, and the third is quantified: the relaxation and the verifier are correct, and the search radius is one to two slots.

Figure 11 plots that frontier, per target and pooled.

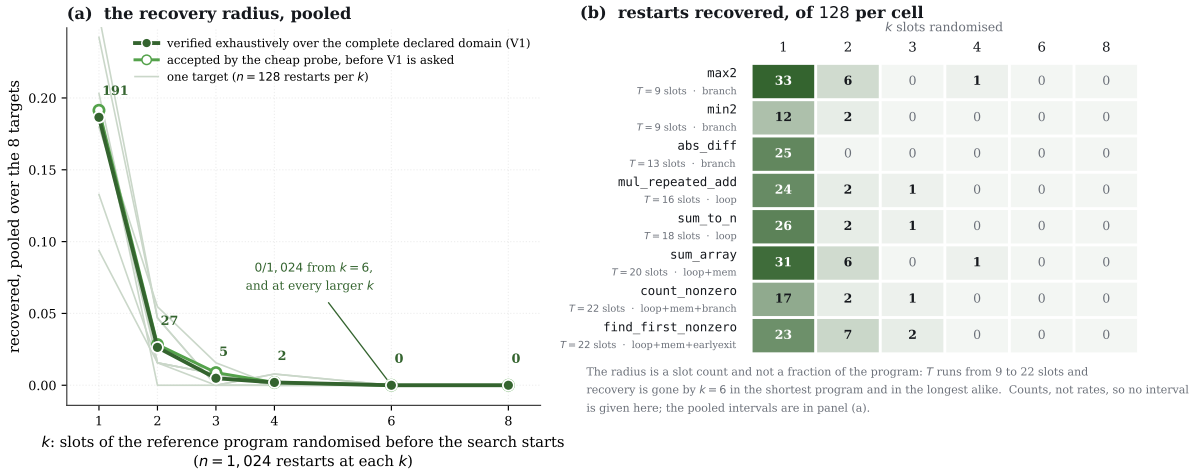


Figure 11: How far from a correct control-flow program the differentiable search can still find its way back. Each restart begins at the reference program with k of its T slots randomised and runs 600 steps; 128 restarts per target per k, eight targets, 1024 pooled restarts at each k. *Left:* the pooled curve, both as the cheap probe scores it and as **V1**—exhaustive execution over every point of the declared finite domain—confirms it; the gap between the two lines is the probe's false accepts, which V1 rejects. Recovery is 191/1024 at $k = 1$, 27/1024 at $k = 2$, 5/1024 at $k = 3$, 2/1024 at $k = 4$ and 0/1024 from $k = 6$ onward. The pale lines are the eight targets taken singly, $n = 128$ each. *Right:* the same counts per target, set against the program length the perturbation is a fraction of: T runs from nine slots to twenty-two and the radius does not scale with it. This is a basin-mode run and is never counted toward the pre-registered criterion, which requires a random start. The generator refuses to draw the panel unless the receipt reports restart repopulation disabled—`repopulate_every` $= 0$, and zero repopulations recorded on every row—because repopulating restarts across k would carry a low-k solution into a high-k slot and flatten exactly this curve. Receipt `gpuq07_control_flow_basin`.

14.5 Tracing a production interpreter

The harness is built: WASMI 0.31.2 forked with forty-six additive lines across five files, nothing removed, behind a feature that is off by default, emitting 3.48 million (opcode, pre, post) triples from a production engine's own dispatch loop. Non-perturbation is confirmed on 4368 cases with identical results and identical memory hashes.

Recovery from that trace class isolates twenty-six of forty-one opcodes. The interesting quantity is *why* the remainder do not isolate, and it is not that real execution excludes the correct microcode: for all twenty-six the ground truth lies inside the trace class throughout. Four opcodes—`or`, `eqz`, `lt_s` and `gt_u`—remain unseparated because compiled `uint32_t` code never generates the sign-straddling operand corner that a designed probe distribution manufactures deliberately. Identifiability from real compiled execution is thus strictly weaker than identifiability from designed probes, the gap is localised to four named opcodes, and the cause is a property of what compilers emit rather than of the estimator.

This also settles the relationship between the two harnesses. The Python path of §16 recovers

against our own reference step function and therefore cannot pose this question at all; the Rust tracer can, which is what makes it the right instrument even though it is the slower one.

15 Four Machine Shapes

A method is indifferent to machine architecture or it is not general. The construction has been carried to five instruction sets spanning four machine shapes, and in each case the reference table is differentially tested against a production implementation before any recovery is attempted. Across the five tables, 1 052 266 133 wrong (opcode, microcode) pairs are decided with none left unknown.

instruction set	machine shape	opcodes	$\lvert M\rvert$	differential against a production implementation
WebAssembly integer MVP	stack, three unbounded structured components	104	—	WASMTIME, 2997/2997 modules and 1669/1669 real call-graph closures
RISC-V RV32I+ZICSR+ZIFENCEI	register, four state components	47	—	UNICORN, 120 000/120 000 whole post-states
RISC-V RV64IM	register, 64-bit, multiply/divide	65	—	UNICORN, 1 240 000/1 240 000 whole post-states
ARMV7-A	register, condition flags, predicated execution, barrel shifter	109	8 294 400	UNICORN, **1 080 000/1 080 000** whole post-states, 0 disagreements
MOS 6502	accumulator, variable length	151	61 641 216	`py65`, **1 500 000/1 500 000**, 0 disagreements
total	**five instruction sets, four shapes**	**476**		

Table 8: **The same procedure across four machine shapes.** Every row is recovered from execution alone and checked against an independent production implementation before recovery. Receipts `exp30_armv7_unicorn_differential`, `exp32_6502_py65_differential`.

armv7-a. Condition flags, a predicate on nearly every instruction, and a barrel shifter on the second operand make this a different state shape from both a stack and a plain register machine. The reference table proves 109/109 UNSAT in 2.52 s, and because the condition field stays symbolic those 109 queries cover 1620 conditional encodings. All 904 089 491 wrong (opcode, microcode) pairs are decided with none unknown, in 488 s—149 ns per pair, nine times past the break point at which exhaustive decision was expected to become impractical. That is paid exactly rather than sampled: the post-state factors into components each reading a strict subset of the microcode fields, a factorisation proved in Lean and checked against brute force by a control that covers every `pc` effect, every writeback target, both register-field layouts, all four flag policies and both barrel-shifter forms—24 opcodes over four transition slices each, 96 checks, no violation.

The strength of that control is not incidental, and the condition under which such a factorisation is sound is sharper than it first appears. Three separate failures of an analogous filter on the 6502—two dropping microcodes the reference keeps, one admitting a no-op for

an instruction that plainly writes the status register—turn out to be one mistake, and it is stateable.

> A component factorisation is exact only if its components are *write-disjoint*, not merely read-disjoint. The claim "component X depends only on the field set S" is false unless S contains every field that *writes* what X reads.

On ARMv7-A that condition holds, which is why the sweep returns exact and why the count above is a decision rather than an estimate: the `pc` effects `pc` + 4, `pc` + 8 + *imm* and `bx` read only the pre-state and fields the stage already holds, every register source is captured before writeback, and no component writes what another reads. On the 6502 it fails three times over, because the second source, the writeback target and the `pc` effect all mutate the *same two scalars*—the stack pointer and the status register—in sequence within a single instruction, so `rts` and `rti` pull their target through a pointer that earlier fields of the same instruction have already moved. The repair is not to abandon the factorisation but to widen each component's field set until it is write-closed, returning unconstrained wherever that closure would pull in the arithmetic result. Stated this way the technique's applicability is predictable from an instruction set's dataflow rather than discovered per machine. Executing whole programs with the *recovered* table reproduces final state and `pc` trace on 120/120 programs over 2801 instructions.

mos 6502**.** An accumulator machine with variable-length instructions fetched from the same address space as data. The reference table proves 151/151 UNSAT in 0.13 s. Here there is no symbolic instruction word at all—quantifying over memories already quantifies over every encoding—and one precondition is therefore *removed* rather than restated: the joint-distribution condition RV32I required is vacuous, because the operand bytes are bytes of the configuration.

The complete integer mvp. Extending WebAssembly from its `i32` core to the whole integer MVP adds `i64` and structured control, and with them two further unbounded structured components beyond the value stack—a label stack and a frame stack. The equivalence argument that closed the value-stack case therefore has to be made twice more, and it is: 104/104 opcodes prove UNSAT over non-empty stacks and 103/104 over the full domain. The result that matters is not the opcode count but what the recovered table then runs. Lifting real call-graph closures from compiled SQLite, zlib and a Rust binary yields 343 closures spanning 554 522 instructions, 124 of them crossing more than one function; the recovered table agrees with WASMTIME on 1669 of 1669, with no mismatch under either the reference or the recovered semantics.

Width and privilege. RV64IM carries the register machine to sixty-four bits with multiply and divide, proving 65/65 and deciding all 37 065 535 of its wrong pairs with none unknown, 200/200 programs cycle-exact. Adding ZICSR and ZIFENCEI to RV32I introduces a fourth state component, the control-and-status file, and with it a prediction the plan made in advance: that every one of the six CSR opcodes is invisible to a state encoding carrying only registers, memory and `pc`. That prediction is established four independent ways—measured in recovery, decided by the solver at 36/36 UNSAT, proved in Lean, and observed on GPU as a gradient that is exactly zero.

What each shape asks of the theorem

The identifiability theorem's hypotheses are re-derived per machine shape rather than carried over, and the shapes disagree about what they need (Figure 12).

ARMv7-A tightens the RV32I condition. There the probe distribution had to be joint over instruction word and configuration; here the flags must falsify each word's *own* condition, and

the obligation is strictly stronger because no configuration falsifies `AL`—the unconditional predicate that dominates real ARM. Under a sampler that always emits `AL`, $p(o, \cdot)$ is 0.0000 on ten of ten opcodes tested and the predicate field is unpinned on 108 of 108. A second condition, found by pursuing the four opcodes that first resisted recovery, is that the immediate magnitude must exceed eight bits or two encodings become equal, checked on 20 000/20 000 draws; its repair interacts with a third, since a twelve-bit offset walks the address off the seeded window, so the base register must be conditioned on the immediate. With both in place all four opcodes become singletons and prove. A pre-registered prediction that fixing the flags alone would collapse p_{sep} was not borne out ($p \approx 0.45$), which locates the degeneracy on the word side rather than the configuration side.

A Harvard machine asks for something different in kind, and the obstruction is proved rather than observed: with a single memory in the state encoding, a program-space load and a data-space load are the same function, so no probe distribution separates them and no embedding recovers the distinction. Splitting the state is necessary but not sufficient—a harness that mirrors the program into both spaces returns p_{sep} to zero. This is a condition on the state encoding rather than on the sampler, and it is machine-checked (§A).

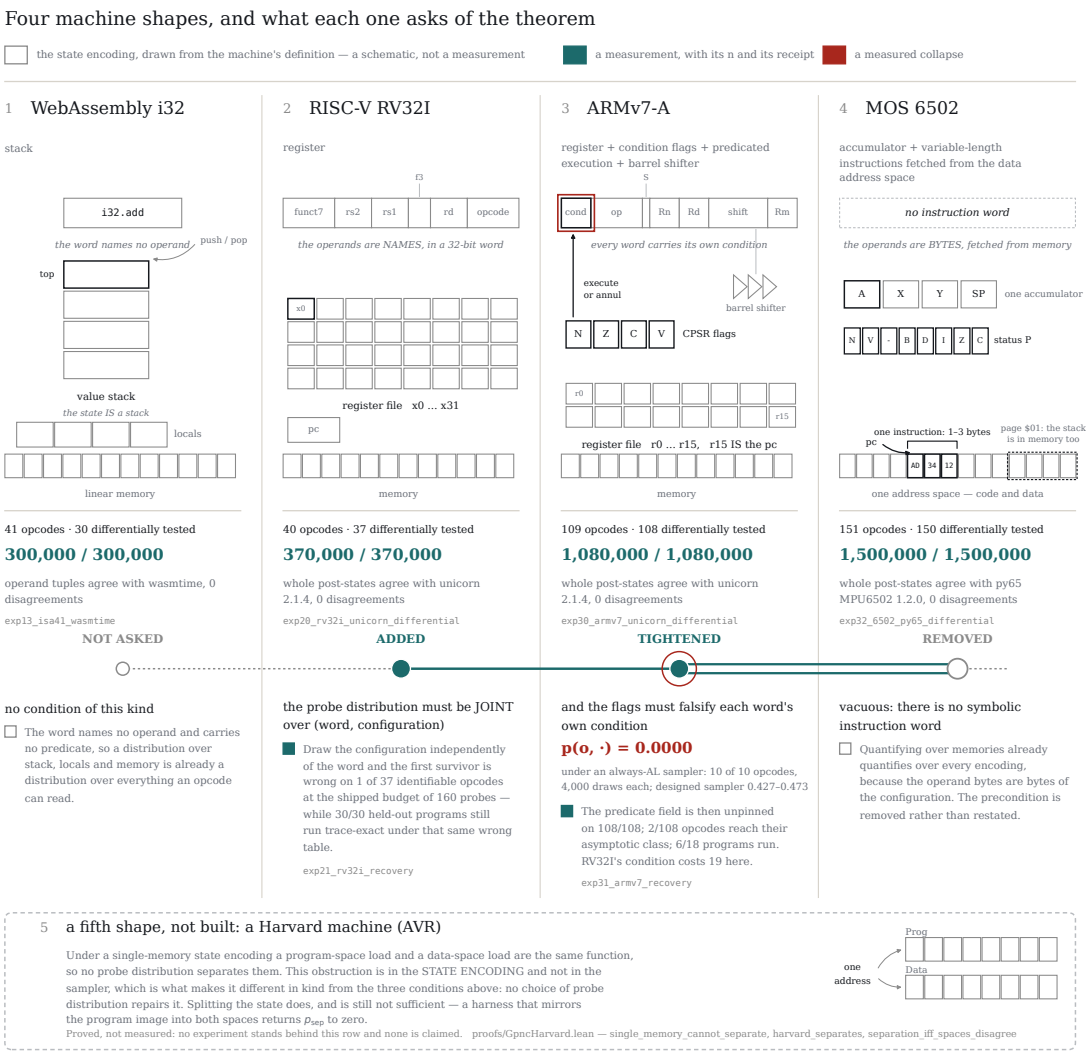


Figure 12: **Four state encodings, and the one precondition that changes across them.** Each column draws one machine's state from that machine's definition—a value stack; a register file; a register file with condition flags, a predicate on every word and a barrel shifter; an accumulator whose instruction bytes are bytes of the configuration, leaving empty the slot the other three fill with a symbolic instruction word. Every such mark is a thin grey outline: a schematic, not a measurement. The strip beneath each column is that row's differential against a production implementation, run before any recovery: 300 000/300 000 operand tuples against WASMTIME (`exp13_isa41_wasmtime`); 370 000/370 000, 1 080 000/1 080 000 and 1 500 000/1 500 000 whole post-states over 37, 108 and 150 opcodes (`exp20_rv32i_unicorn_differential`, `exp30_armv7_unicorn_differential`, `exp32_6502_py65_differential`); no row has a disagreement. The rail threading the columns is the joint-distribution precondition of §15: absent on the stack machine, added by the register machine, tightened by predication, removed on the 6502. Drawing the configuration independently of the instruction word leaves the first survivor wrong on 1 of the 37 identifiable RV32I opcodes at the shipped budget of 160 probes, while 30/30 held-out programs still run trace-exactly under that same wrong table (`exp21_rv32i_recovery`); an always-`AL` sampler gives $p(o, \cdot) = 0.0000$ on ten of ten ARMV7-A opcodes at 4000 draws each—against 0.427–0.473 under the designed sampler—leaving the predicate field unpinned on 108/108 (`exp31_armv7_recovery`). Both are single deterministic runs and carry no interval. The band at the foot is neither: a shape this paper has not built, whose obstruction is a machine-checked theorem.

16 Toward Real Machine Code

This section and Section 17 are the roadmap; Proposition 1 fixes what "general purpose" means for the machine as built, and Eq. (1) the cycle it executes.

The instruction set validated here is real but deliberately small. The method, however, is indifferent to the size of the alphabet, because its supervision is precisely what a processor already emits. A hardware or emulated execution trace pairs each retired instruction with the machine state before and after it,

$$\sigma_{\text{before}} = (\text{pc}, r_0, \ldots, r_n, \text{mem}) \quad \xrightarrow{o} \quad \sigma_{\text{after}} = (\text{pc}', r'_0, \ldots, r'_n, \text{mem}'), \tag{9}$$

(Eq. (9)) which is exactly the label-free tuple (o, σ, σ') that label-free recovery consumes. Every mechanism of the machine transfers: the symbolic unit already computes fixed-width integer arithmetic exactly; the pinning membrane is natural, since registers and memory are fixed-width integers and hence a lattice; and the spectral operator of Eq. (3) addresses a linear memory of any size. The tractability of a target instruction set is governed by two axes—the number of distinct opcodes and the regularity of the encoding (Figure 13).

Whether this scales past a toy is empirically checkable, and the census below is the check.

Disassembling 2437 real ELF objects from this machine's `/usr/bin` and `/usr/lib` with `objdump` gives 94 505 175 static x86-64 instructions over 1162 distinct mnemonics. The coverage curve is steep: **the top** 15 **mnemonics cover** 81.2% **of all static x86-64 instructions and the top** 47 **cover** 95.9%. The same measurement over CPython bytecode (574 stdlib modules, 770 419 instructions, 97 distinct opcodes observed of 140 named) gives 77.6% at 15 and 96.7% at 47, and an independent cross-check over 11 754 site-packages modules (18.4M instructions) gives 76.4% and 96.7%. The curves are nearly identical across a 12× difference in nominal alphabet size and across two unrelated corpora.

This is the tractability argument as a number: a fifteen-opcode result is not a toy-sized fraction of real code—fifteen opcodes is the scale at which four-fifths of static x86-64 is already covered, and 47—rv32i together with its Zicsr and Zifencei extensions, the base integer set itself being 40 instructions—is the scale at which 96% is. The census measures *static mnemonic* frequency, and three quantities sit beside it as separate measurements: dynamic frequency, an instruction's addressing modes, flags and prefixes, and the instruction-boundary problem that makes variable-length cisc hard. The mnemonic count is a property of `objdump`'s AT&T syntax rather than of the ISA: operand-size variants such as `mov`/`movq`/`movl` are counted separately, padding pseudo-instructions appear in the head of the distribution (`int3` alone is 2.3% and is alignment padding, not program logic), and 8477 bytes failed to disassemble under linear sweep. arm and rv32i corpora need the binaries and a cross-toolchain; those two points are absent from the figure rather than estimated.

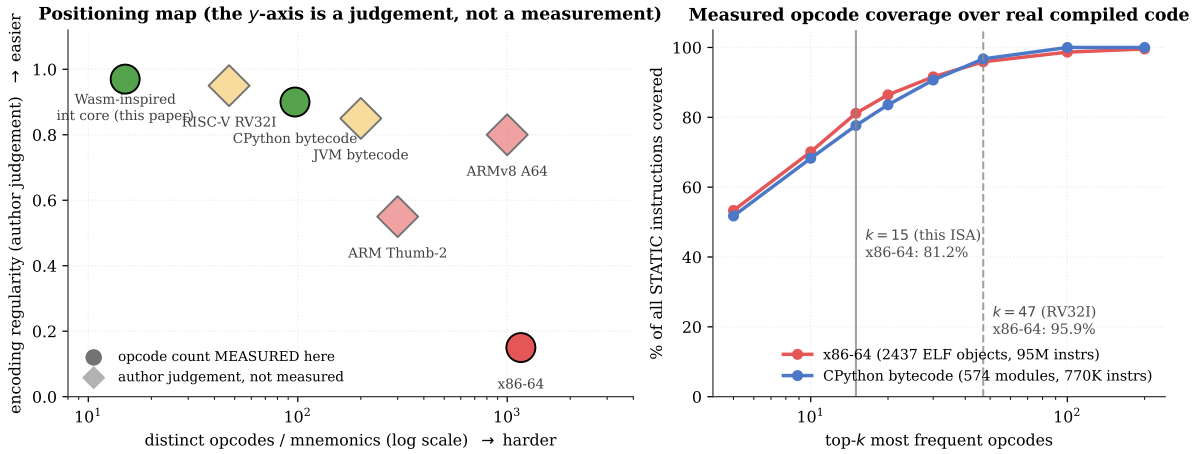


Figure 13: Tractability of instruction sets for neural interpretation. *Left:* a qualitative positioning map, marked as author judgement rather than measurement. *Right:* the *measured* opcode-coverage curve—fraction of static instructions covered by the top-k mnemonics, over 94.5M x86-64 instructions from 2437 real ELF objects and 770K CPython bytecode instructions from 574 stdlib modules. The top 15 mnemonics cover 81.2% of static x86-64; the top 47—the size of RV32I with Zicsr and Zifencei, its base integer set being 40—cover 95.9%.

Grounding on real compiled WebAssembly

The census above counts mnemonics; it does not say whether the recovered object can execute compiler output. That check has since been run on real `.wasm`, and it is reported ceiling first. Thirteen integer C kernels compiled with `zig cc` (clang 21.1.0, `wasm32-freestanding`) at `-O0`, `-O1`, `-O2` and `-Os` give 52 modules, of which 39 map. On those the recovered 41-opcode table is cycle-exact against the reference small-step semantics on **3900/3900** (module, input) pairs over 4 773 354 cycles, and identical to WASMTIME on both the returned `i32` and the *entire* final one-megabyte linear-memory image, 3900/3900 $[0.9990, 1.000]$ on each of the three checks at $n = 3900$. The bridge from structured to flat control flow, from byte addresses to cell indices, and two scratch locals is hand-written and is built only from the 41 recovered opcodes; it is 251 of 3012 emitted instructions (8.3%). Nineteen *internal* functions lifted out of SQLite 3.45.1, zlib 1.3.1 and an 819-function Rust release binary—code written by someone else, invoked by function index against `pywasm` as an independent reference—agree on 132 of 148 inputs when their pointer arguments are four-byte aligned, with *zero* mismatches: every disagreement is the out-of-bounds trap divergence of Table 3, now measured on third-party pointer-chasing code rather than on synthetic programs. With arbitrary pointers the cell-versus-byte divergence appears as 18 mismatches in SQLite and 12 in the Rust binary, which is what the two argument pools were built to price. The pre-declared single-field mutant (`i32.add` → `i32.sub`) breaks agreement with WASMTIME on 8 of the 13 `-O2` kernels, and a full single-field sweep over every table entry those kernels use catches $6000/8358 = 0.718$ $[0.708, 0.727]$.

The negative result is the larger one, and Figure 14 is it. Over 614 931 static instructions of unrestricted third-party WebAssembly the fragment covers 89.7% of the *instructions* and 2.8% of the *functions*, because execution is a whole-function property: one out-of-table instruction anywhere in a body makes the body unrunnable, and 10.3% of instructions outside the table puts one in 97.2% of the functions. The same asymmetry is visible in miniature at `-O0`, where 52 retired `global.get`—0.034% of the instructions that row executes—cost all 13 modules. What the table would have to learn next is therefore a question about *functions* and not about instruction frequency, and the census answers it by greedy measurement (Figure 15): `call` first everywhere, then byte-width memory, then `i64`. The pre-registration's prediction that

unrestricted coverage would fall below 60% is falsified in the optimistic direction (62.5% direct, 89.7% with the bridge), and is recorded as wrong. Experiment `exp23_real_wasm`; the write-up is `REAL_WASM_GROUNDING.md`.

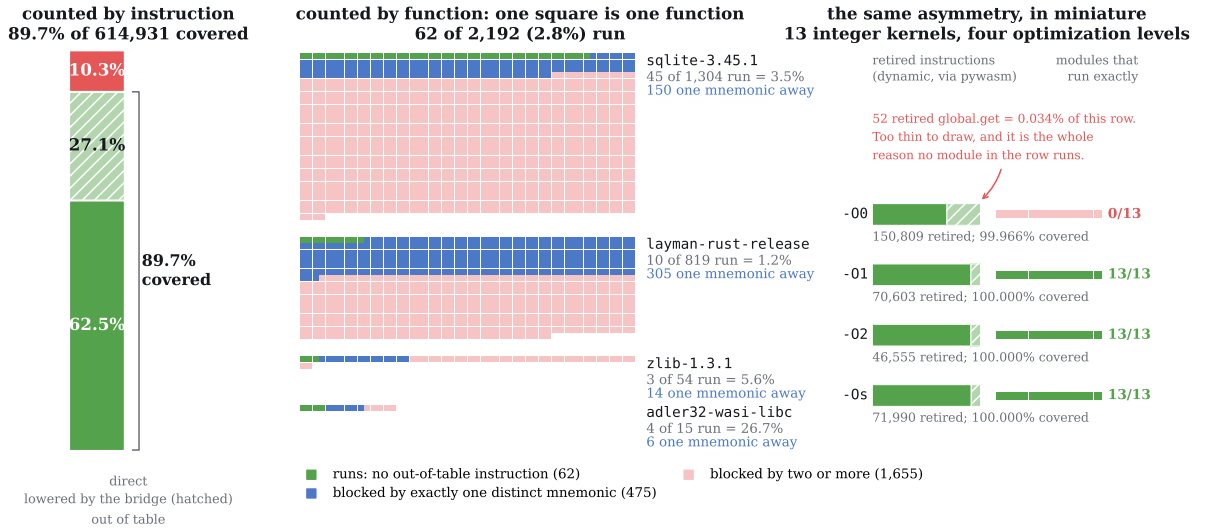


Figure 14: Two coverage numbers for one corpus, disagreeing by a factor of 32. *Left:* the 614 931 static instructions of four third-party modules—SQLite 3.45.1, zlib 1.3.1, an 819-function Rust release binary, and an `adler32` build linked against wasi-libc—of which 62.5% map one-to-one onto the 41 recovered opcodes and a further 27.1% through the three declared bridge forms: 89.7% covered. *Centre:* the same corpus counted by function, one square per function, grouped by module and ordered within it. 62 of 2192 functions (2.8%) contain no out-of-table instruction and can therefore be run; 475 are blocked by exactly one distinct mnemonic and 1655 by two or more. Coverage is instruction-weighted, execution is a whole-function property, and this figure is the distance between them—the central negative result of that experiment. *Right:* the same asymmetry at a scale small enough to audit by hand. The 13 kernels of the restricted corpus at four optimization levels: at `-O1` and above LLVM's output for word-wise integer C is entirely inside the fragment and all 13 modules run exactly; at `-O0` the same C spills locals to a shadow stack, and the 52 retired `global.get` this produces—0.034% of that row's 150 809 retired instructions, too thin to draw—make all 13 unrunnable. Dynamic counts come from instrumenting `pywasm`, an independent implementation, not from our interpreter, and our decoder is cross-checked against it on 56 of 57 modules before any census number is produced. A census is a count and not a sample, so no interval is given; the execution rates that accompany it are in the text. Receipt `exp23_real_wasm`.

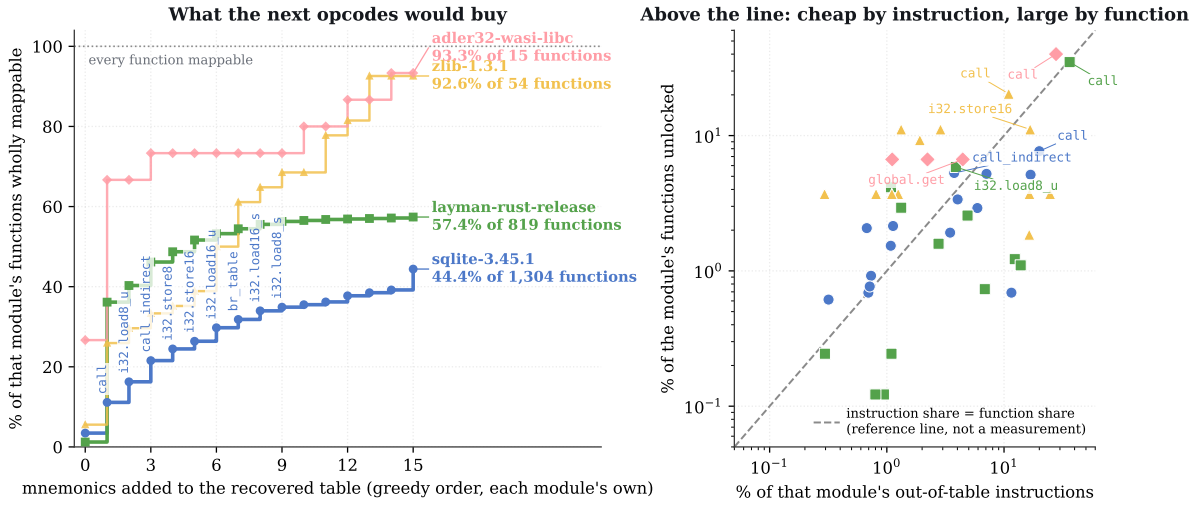


Figure 15: What the recovered table would have to learn next, measured rather than ranked by intuition. *Left:* for each third-party module, the percentage of its functions that become wholly mappable as mnemonics are added to the 41-opcode table, in the greedy order the census records—at each step the mnemonic that unlocks the most whole functions, so each module has its own order and the labels drawn are SQLite's. The first step is `call` in all four modules: +100 whole functions in SQLite, +286 in the Rust binary, +11 in zlib, and it is the one extension that changes the machine rather than adding another five-tuple, since it needs frames and a return address. Fifteen additions carry SQLite to 44.4% of 1304 functions and the Rust binary to 57.4% of 819: the remainder is not a long tail that a few more opcodes would clear. *Right:* every mnemonic of those cascades, plotted as its share of that module's out-of-table instructions against the share of that module's functions it unlocks. Points above the dashed reference line buy more code than their instruction count suggests—`br_table`, the narrow loads, `global.get`—and points below it are expensive by instruction and thin by function, which is the same asymmetry as Figure 14 stated one mnemonic at a time. Counts are static occurrences over the 2192 censused functions, and the per-module census keeps the 40 most frequent out-of-table mnemonics, so mnemonics outside that cut are absent from the right panel rather than estimated. zlib and the wasi-libc build hold 54 and 15 functions, so a single function is 1.9% and 6.7% of their curves. Receipt `exp23_real_wasm`.

A second machine model: RV32I

Everything above is one instruction set, and a WebAssembly-shaped one: a stack machine whose operands sit at fixed positions, $S[-1]$ and $S[-2]$. The standing objection is that what was built is a WebAssembly result rather than a method, and the only answer to it is a second machine model of a different shape. RV32I is a register machine—40 base integer instructions, six immediate formats, operands named by register fields inside the instruction word—and the reference implementation of it is differentially tested against `unicorn` 2.1.4 (the QEMU TCG core) before anything is recovered from it. The comparison is the *entire* post-state, all 32 registers, the memory window and the `pc`, rather than one returned value: **370 000/370 000** agreements over the 37 base instructions that have observable integer-state semantics, at 10 000 random (instruction word, configuration) pairs each, per-opcode Wilson $[0.99962, 1.000]$ at $n = 10\,000$, with zero disagreements and zero engine faults (Figure 16). Three instructions are reported apart rather than counted: `fence` is modelled as a no-op and `ecall`/`ebreak` as halt sentinels, which are modelling choices and not RISC-V semantics. The scope is stated the same way round as the coverage above—40 of RV32GC's 143 instructions, 33.9% of RV32G, with M, A, F, D, C,

Zicsr, Zifencei, privileged mode, traps and interrupts all absent.

Two controls bound what that agreement can mean. The decoder's *over*-acceptance is bounded: every one of the 2011 words it accepts is executed by the engine. Its *under*-acceptance is not, and the receipt says so: Unicorn's default rv32 cpu implements m, a, f, d and c, so a rejected word is usually a legal instruction of an extension this model does not have, and bounding that would need an rv32i-only cpu model. Of the 390 rejected uncompressed words, 388 fault in the engine and 2 execute; both are recorded. Fetch-fault read-back is sound on 2000/2000. The second control is planted-defect power, and it is the more useful half: removing the `x0` hardwiring is caught on 2000/2000 trials and a wrong immediate format on 1937/2000, but dropping sign-extension is caught on only 975/2000 = 0.487, which is the weakest arm of the test and is reported rather than dropped. Experiment `exp20_rv32i_unicorn_differential`.

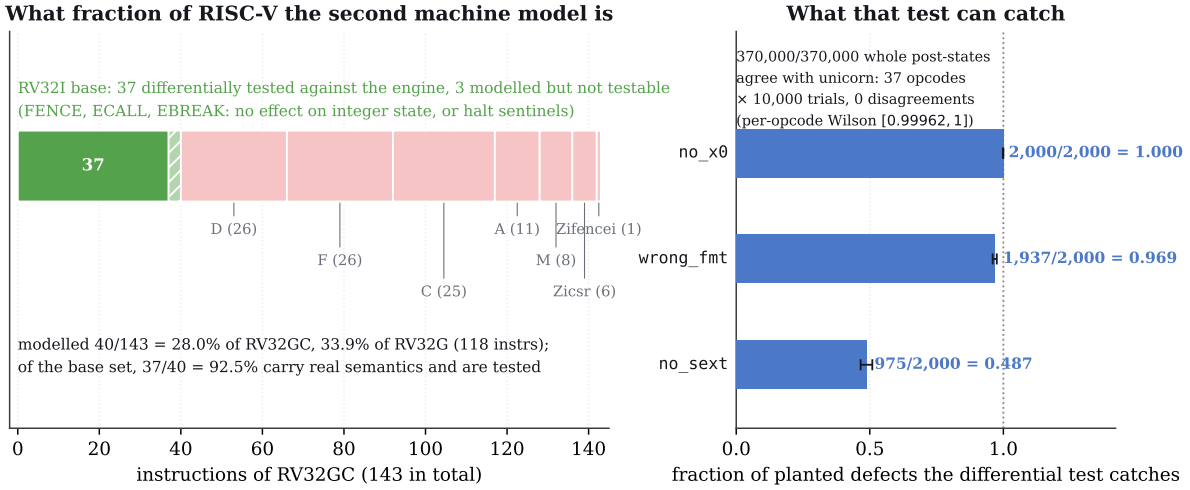


Figure 16: The second machine model, sized and stress-tested before anything is recovered from it. *Left:* rv32i against the whole of rv32gc. 37 base integer instructions carry real semantics and are differentially tested; 3 more are modelled but cannot be tested from integer state (`fence` has no effect, `ecall`/`ebreak` only stop execution); the remaining 103 instructions of m, a, f, d, c, Zicsr and Zifencei are absent and are drawn as absent. That is 40/143 = 28.0% of rv32gc and 33.9% of rv32g. *Right:* what the differential test can catch, measured with three planted defects on 2000 trials each, Wilson 95%: the `x0` hardwiring 2000/2000, a wrong immediate format 1937/2000 = 0.969, and dropping sign-extension only 975/2000 = 0.487—the weakest arm, and the reason the headline is stated as agreement on the entire post-state rather than as a proof. The agreement itself is 370 000/370 000 whole post-states (32 registers, memory window and `pc`) over 37 opcodes × 10 000 trials against `unicorn` 2.1.4; the interval to read is the per-opcode one, $[0.99962, 1.000]$ at $n = 10\,000$, not a pooled interval over comparisons that are not independent trials. Receipt `exp20_rv32i_unicorn_differential`.

Label-free recovery on the register machine, and what that costs

The recovery procedure carries over to rv32i unchanged in form and is harder in exactly one place. Its microcode grammar is seven fields—which immediate layout decodes the word, the two operand sources, the alu operation, the writeback, the `pc` effect and the memory access—so $|M| = 136\,080$ against ISA-41's 3264, and $40 \times 136\,080$ gives 5 443 160 wrong (opcode, microcode) pairs against 133 783: the hypothesis space is 40.7× larger. At the shipped budget of 160 probes per opcode the first survivor is tuple-exact on 36 of 40 opcodes, 8 classes are singletons, and 34 of the 37 opcodes with observable integer-state semantics sit at the provable lower bound of their class—a bound decided from the reference microcode before any probe is drawn, because

a total memory makes the five load widths indistinguishable on any opcode that does not write a loaded value back, and an R-type word carries no immediate. `fence`, `ecall` and `ebreak` are reported apart with $|E|$ of 25 884, 66 096 and 30 510: no observation of $(X, \mathrm{Mem}, \mathrm{pc})$ separates them, which is a property of the machine rather than of the estimator. The recovered table is trace-exact on a held-out suite of 120 programs over three strata, 120/120 $[0.969, 1.000]$, and 120/120 on final state.

Two things about that recovery are worth the space (Figure 18). The first is that identifiability genuinely *degrades* from the stack machine to the register machine: on exp12's own convention—a seed counts only when every identifiable opcode reaches its asymptotic class—ISA-41 reaches every seed at 80 probes per opcode and RV32I needs 256, climbing 0/20 seeds at 1 through 16 probes, 1/20 at 32, 7/20 at 64, 16/20 at 128. The second is the shape of that cost. The hypothesis space grew 40.7× and the measured probe budget 3.2×, while the only term through which the instruction-set size enters the sample-complexity bound—it sits inside a logarithm—grew 1.25×. Opcode count is not the barrier here. The proof went the other way entirely: total equivalence over the whole unbounded state space is 40/40 in 0.045 s of solver time and *one* query per opcode, against ISA-41's 246 queries and a locality lemma, because a register machine's state has no unbounded structured component to quantify over and the instruction word can be left fully symbolic—so the `unsat` also covers all 2^{32} words, legal or not. Every one of the 5 443 160 wrong pairs was then decided rather than sampled: 5 328 700 separated by concrete search at 400 joint draws each, and of the 114 460 that survived, $z3$ separates 92 460 and *proves* 22 000 observationally equivalent to the reference, with none left unknown.

Figure 17 draws that accounting as the partition it is.

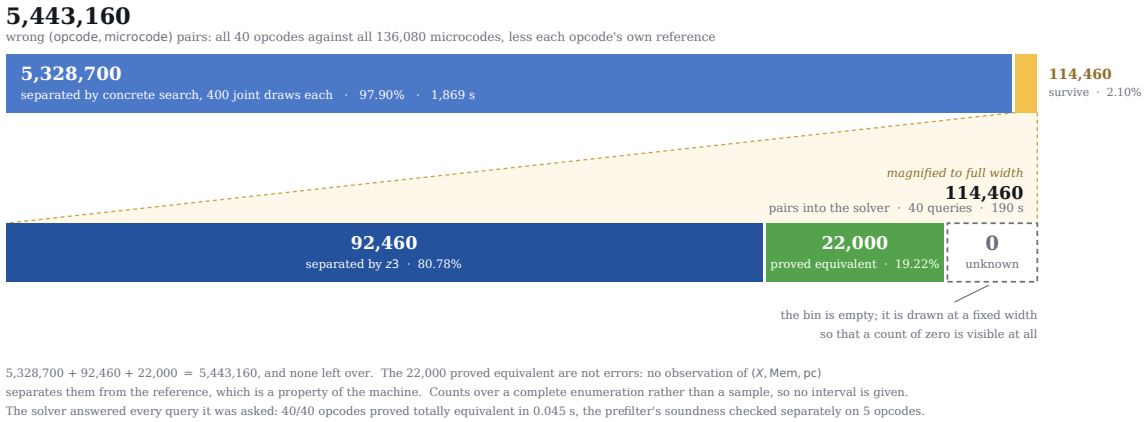


Figure 17: The wrong-pair ledger of the register machine, drawn as the partition the text states in prose. Of the 5 443 160 wrong (opcode, microcode) pairs—all 40 opcodes against all 136 080 microcodes, less each opcode's own reference—5 328 700 are separated by concrete search at 400 joint draws each, in 1869 s. The 114 460 that survive are magnified to full width and handed to the solver, which separates 92 460 of them and *proves* 22 000 observationally equivalent to the reference, in 190 s. The bin for the undecided is empty, and is drawn at a fixed width precisely so that a count of zero is visible at all; that width is the one quantity in this figure that is not a measurement. The 22 000 are not errors: no observation of $(X, \mathrm{Mem}, \mathrm{pc})$ separates them from the reference, which is a property of the machine rather than of the estimator. Counts over a complete enumeration rather than a sample, so no interval is given. Receipt `exp21_rv32i_recovery`.

The precondition that does change is the one the theorem's restatement names, and Figure 19 is its negative control. On a stack machine the operands an instruction reads sit at fixed positions, so a distribution over configurations alone controls them; on a register machine they

sit wherever the instruction word says, so D must be a distribution over *(instruction word, configuration) pairs*. Four restrictions that a real instruction trace produces easily were run against the shipped recovery. Three of them—every destination `x0`, `rs1` = `rs2`, every immediate zero—take the recovered table to 8, 26 and 10 of 37 opcodes exact and blow single classes open by three orders of magnitude (`lui` reaches $|E| = 23\,968$), and the held-out program suite does see it: 0/30, 2/30, 0/30. The fourth does not. Drawing the configuration *independently* of the instruction word—the most innocuous of the four, and the one a naive sampler writes first—leaves 36 of 37 opcodes exact, every class at its asymptotic size, and *all* 30 *held-out programs passing*, while the recovered table still differs from the reference on one opcode. A program suite is not a substitute for the probe distribution the theorem asks for.

The last question is where this stops being affordable, and it is arithmetic rather than speculation (Figure 20). Deciding every wrong pair was priced from a measured 44.2 μs per pair and 4.5 ms per solver query: 0.07 CPU-hours at RV32I, a projected 77.6 at RV64GC and 183.6 at a top-47 x86-64 grammar, single-core. The sample-complexity term over the same seven grammars moves from 12.5 to 26.4 while the hypothesis space moves from 1.3×10^4 to 1.5×10^{10}. What does not carry is p_{sep}, which is machine-specific and is measured here only for RV32I: of 4000 sampled pairs none had zero separation and the smallest was 0.1525, so $p_{\text{sep}} \leq 0.1525$ on that machine, a sample minimum and therefore an upper bound. Experiments `exp21_rv32i_recovery` and `exp22_scaling_arithmetic`.

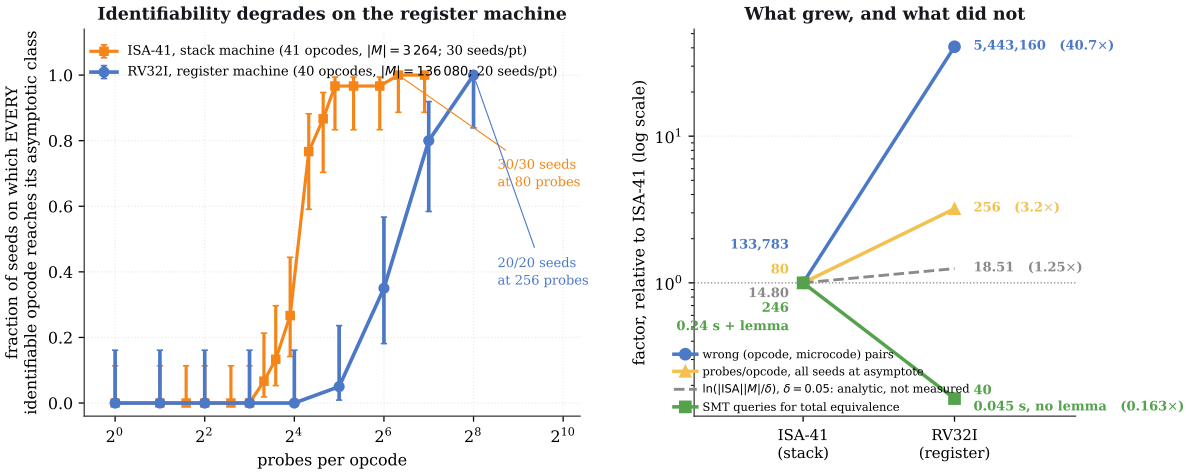


Figure 18: Carrying label-free recovery from a stack machine to a register machine. *Left:* the probe-budget curve for both, on the same convention—a seed counts only if *every* identifiable opcode reaches its asymptotic equivalence-class size at that budget—with Wilson 95% intervals, 30 seeds per point for ISA-41 and 20 for RV32I. ISA-41 reaches every seed at 80 probes per opcode; RV32I is at 0/20 through 16 probes, 1/20 at 32, 7/20 at 64, 16/20 at 128 and 20/20 at 256. Identifiability degrades, and it degrades on the axis the restatement predicts rather than on opcode count. *Right:* what grew between the two machines, every quantity as a factor of its ISA-41 value on a log axis. The hypothesis space the search must reject grows 40.7× and the probe budget measured to reach every asymptotic class on every seed grows 3.2×, while the only term the instruction-set size can touch in the sample-complexity bound—it sits inside a logarithm—grows 1.25×; that term is analytic and is drawn dashed and grey as such. The proof cost falls: 40 queries in 0.045 s with no locality lemma, against 246 queries and one for ISA-41, because a register machine's state has no unbounded structured component. Receipts `exp12_isa41_recovery`, `exp14_total_equivalence_all_depths` and `exp21_rv32i_recovery`.

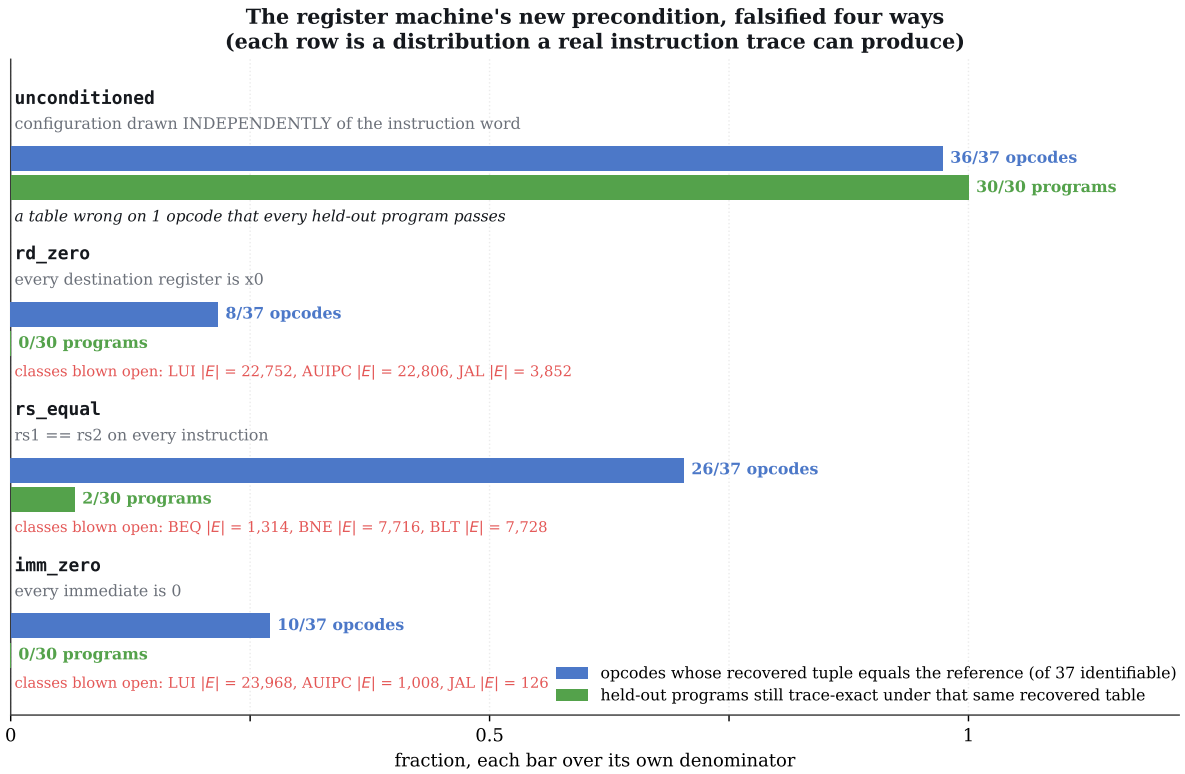


Figure 19: The precondition a register machine adds, falsified four ways. Each row restricts the joint (instruction word, configuration) distribution in one way a real instruction trace produces easily, and scores the recovery that results twice: by identifiability (how many of the 37 identifiable opcodes the first survivor gets tuple-exact) and by the held-out program suite run on that same recovered table. Three restrictions destroy both, and blow single equivalence classes open by three orders of magnitude—`lui` to $|E| = 23\,968$ under a zero immediate, `bne` to 7716 under `rs1` $=$ `rs2`. The first restriction is the dangerous one: drawing the configuration independently of the instruction word is what a naive sampler does, and it returns a table that differs from the reference on one opcode while every one of the 30 held-out programs still runs trace-exactly under it. The program suite is not a proxy for the probe distribution. Single deterministic runs at the shipped budget of 160 probes per opcode, so the counts carry no interval. Receipt `exp21_rv32i_recovery`.

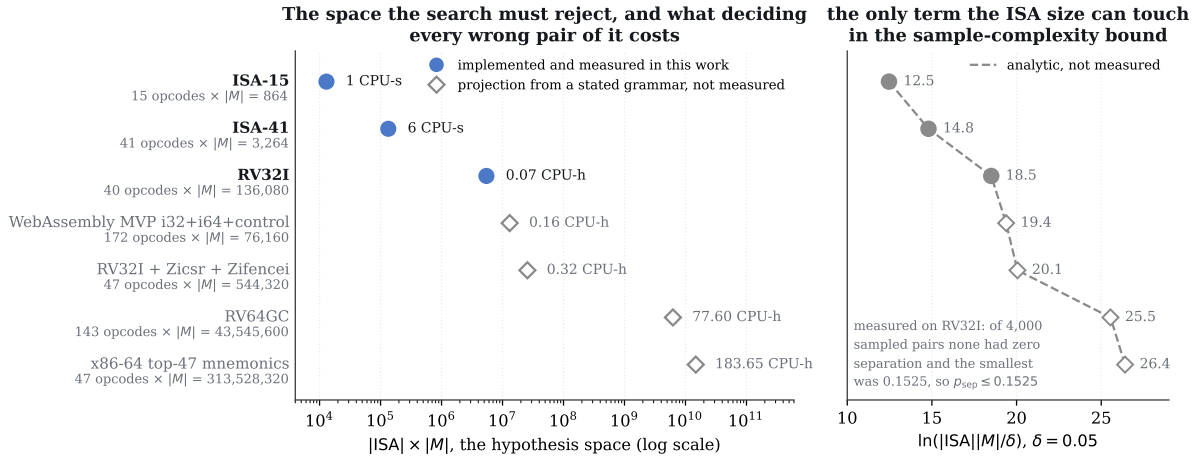


Figure 20: What scales and what does not, computed rather than asserted, over seven microcode grammars. Three are implemented in this work and are drawn filled; the other four are projections from a grammar stated in the receipt and are drawn hollow and grey, as projections. *Left:* the hypothesis space $|\mathrm{ISA}| \times |M|$ each one presents, from 1.3×10^4 to 1.5×10^{10}, and beside each the cost of *deciding* every wrong pair in it—the exhaustive procedure of Section 19—priced from this run's own measured rates of 44.2 μs per concrete pair and 4.5 ms per solver query on the residue. It is 0.07 CPU-hours for RV32I, a projected 77.6 for RV64GC and 183.6 for a top-47 x86-64 grammar, single-core: the part of this method a reviewer would expect to explode is the part that does not. *Right:* the only term through which the instruction-set size enters the sample-complexity bound, $\ln(|\mathrm{ISA}||M|/\delta)$ at $\delta = 0.05$, over the same seven grammars: 12.5 to 26.4 while the space moves by six orders of magnitude. It is analytic, and is drawn dashed and grey as such. The quantity that does *not* transport is p_{sep}, which multiplies that term and is machine-specific; it is measured here only for RV32I, where none of 4000 sampled pairs had zero separation and the smallest was 0.1525—a sample minimum, hence an upper bound. Receipt `exp22_scaling_arithmetic`.

What holds the probe budget open, and what the suite pins down

Two measured structures sit behind the budget that subsection reports, and the aggregate hides both (Figure 21). The first is that the seed-complete budget is set by a handful of opcodes rather than by the instruction count: 25 of the 37 identifiable opcodes are finished by 16 probes and three of them—`add`, `lui`, `xor`—by 2, while `slti` and `sltiu` alone hold it open to 256. `fence`, `ecall` and `ebreak` never saturate at any budget, which is the same fact about those three read off a second axis. The second is a mutation sweep of the recovered table. Perturbing one microcode field of one opcode at a time enumerates 1480 mutants, of which the held-out program suite catches 888, a rate of 0.600; because the sweep is exhaustive over the mutant set and the suite is fixed, these are counts rather than samples. The rate is not flat across the seven fields: `mem` at 0.397 (127/320) and `fmt` at 0.435 (87/200) against 0.685 to 0.725 for the other five. Those two are the axes the lower-bound argument above already names—a total memory makes the load widths indistinguishable on any opcode that does not write a loaded value back, and an R-type word carries no immediate—so the surviving mutants are, in the experiment's own reading, fields the machine provably never reads on that opcode rather than a weakness of the suite. The receipt records a catch count and a denominator per field and does not decompose either into catchable and uncatchable, so no per-field ceiling is drawn and none is claimed. Receipt `exp21_rv32i_recovery`.

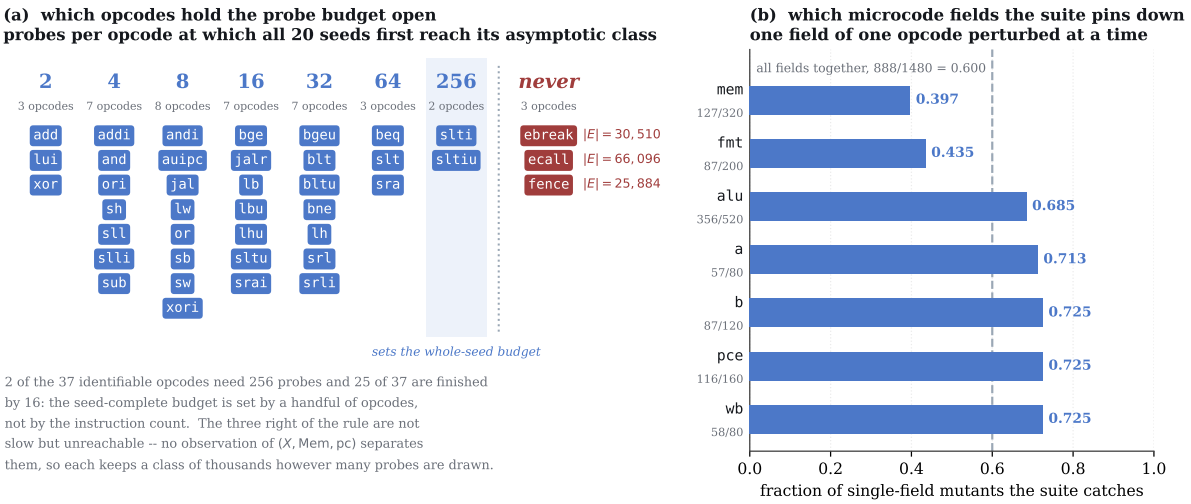


Figure 21: Where the evidence is thin on the register machine, in the two places the receipt measures it. *Left:* every identifiable opcode placed in the column of the probe budget at which all 20 seeds first reach its asymptotic class. 25 of the 37 are finished by 16 probes; `slti` and `sltiu` alone hold the seed-complete budget open to 256, so the 256 the previous subsection reports is set by two opcodes and not by the size of the instruction set. The three to the right of the rule are not slow but unreachable at any budget—no observation of $(X, \mathrm{Mem}, \mathrm{pc})$ separates them—and each keeps the equivalence class printed beside it. *Right:* an exhaustive single-field mutation sweep of that recovered table, one microcode field of one opcode perturbed at a time and scored by whether the held-out program suite notices: 888 of 1480 caught, 0.600 overall, counts rather than samples and therefore without an interval. The two weakest fields are the two axes this section's lower-bound argument already shows are partly unobservable on this machine. Receipt `exp21_rv32i_recovery`.

The gradient half does not transfer, and that is the result

The register machine was carried to the decoder *network* as well, and that is the half of the procedure that fails (Figure 22). The criterion was written down before the run: at least 30 of the 37 identifiable opcodes inside the equivalence classes `exp21` proved, *and* the unconditioned negative-control arm at least 4 opcodes worse than the designed arm. Three supervision arms were run at 8 restarts each, 20 000 gradient steps per restart, the relaxation mixing over the same 136 080-tuple microcode space per opcode that the search enumerates. Neither clause holds. The best restart of any arm reaches 14 of 37, on the designed sampler; the program-trace arm, which is the one the criterion names, reaches 12; arm means are 6.62, 4.12 and 8.50 over the 8 restarts. The last of those is the negative control, and it carries the *highest* mean of the three. The arm whose (instruction word, configuration) coupling was deleted—the restriction the previous subsection showed destroys identifiability for the branch opcodes—lands 1.88 opcodes *above* the designed arm rather than the 4 below that the pre-registration required.

We report both clauses and read neither charitably. The first says the gradient learner does not reach this table at this scale. The second says something sharper and about the instrument rather than the learner: on this experiment the arm-to-arm ordering carries no signal, so a pass, had one occurred, would not have been interpretable as evidence about supervision either. The 14 is not a partial success and is not reported as one. Section 21 recovers the full ISA-41 table by gradient on the stack machine and that result stands unchanged; what does not follow from it is an extension to the register machine, and the experiment that tested the extension falsified it. The search half of the same procedure transfers on the same machine in

the same receipt, which is what makes the contrast measurable rather than rhetorical. Receipts `gpuq06_rv32i_neural_recovery` and `exp21_rv32i_recovery`.

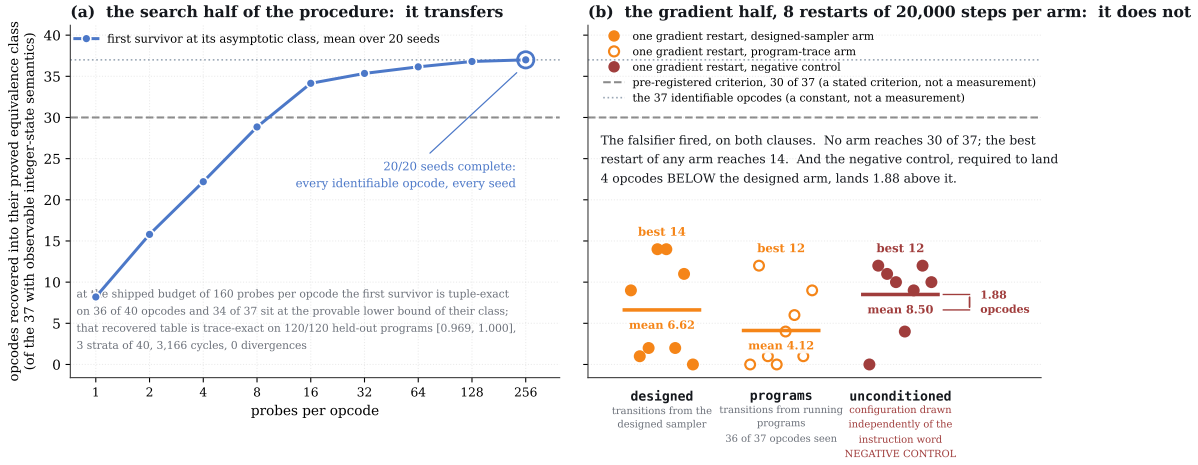


Figure 22: The two halves of the procedure on the same machine model, scored against the same target. Both panels count the same quantity on one shared vertical axis: how many of the 37 RV32I opcodes with observable integer-state semantics are recovered into the equivalence class `exp21` proved for them. *Left:* the search half. Each point is the mean over 20 seeds of how many of those opcodes have their first surviving candidate at its asymptotic class, against the probe budget; at 256 probes per opcode all 20 seeds are complete on all 37. At the shipped budget of 160 the first survivor is tuple-exact on 36 of 40 opcodes, 34 of 37 sit at the provable lower bound of their class, and that table is trace-exact on 120/120 held-out programs, $[0.969, 1.000]$, over 3 166 cycles with zero divergences. *Right:* the gradient half. Every one of the 8 restarts of each of the three supervision arms is plotted, 20 000 steps per restart, because a single gradient run is a draw from a distribution and not a point. The dashed rule is the criterion pre-registered before the run, 30 of 37; it is a stated criterion and not a measurement and is drawn grey and dashed as such, as is the constant 37. No arm reaches it and the best restart of any arm reaches 14. The negative control—the arm whose (instruction word, configuration) coupling was deleted—carries the highest arm mean of the three, 8.50 against the designed arm's 6.62, where the pre-registration required it to land 4 or more *below*: the second clause failed in the wrong direction, which puts the instrument and not only the learner in question. Receipts `exp21_rv32i_recovery` (left) and `gpuq06_rv32i_neural_recovery` (right).

17 The Data-Plane, Compiled to Gates

The instruction set the machine interprets is not only real but *physical*, and we make this concrete. Each opcode of this fifteen-opcode integer core is a fixed-precision integer function—and a fixed-precision function over a finite alphabet is, by the founding theorem of digital design, a Boolean function, hence a logic circuit. Using the verified gate compiler of the companion binary-circuit work, we compile the machine's arithmetic and logic opcodes into gate-level netlists and check each one bit-exactly against the reference interpreter that supervises recovery: `i32.add` to a 160-gate ripple adder, `i32.sub` to 192 gates, `i32.eq` to 63, `i32.lt_u` to a 225-gate comparator, and the ternary `select` to a 159-gate multiplexer—the whole arithmetic and logic data-plane in 799 verified gates (Fig. 23), correct exhaustively at four bits and on random inputs at the full thirty-two. The consequence is worth stating plainly: the machine's data-plane is a small bank of verified circuits selected by the opcode, so the learned interpreter's executed behaviour is not merely deterministic and auditable but *physical*—compiled past the

processor entirely, into gates. The learned control that recovers the microcode and the compiled circuit that computes it are two views of one object; their union is a neural computer one could, in principle, etch.

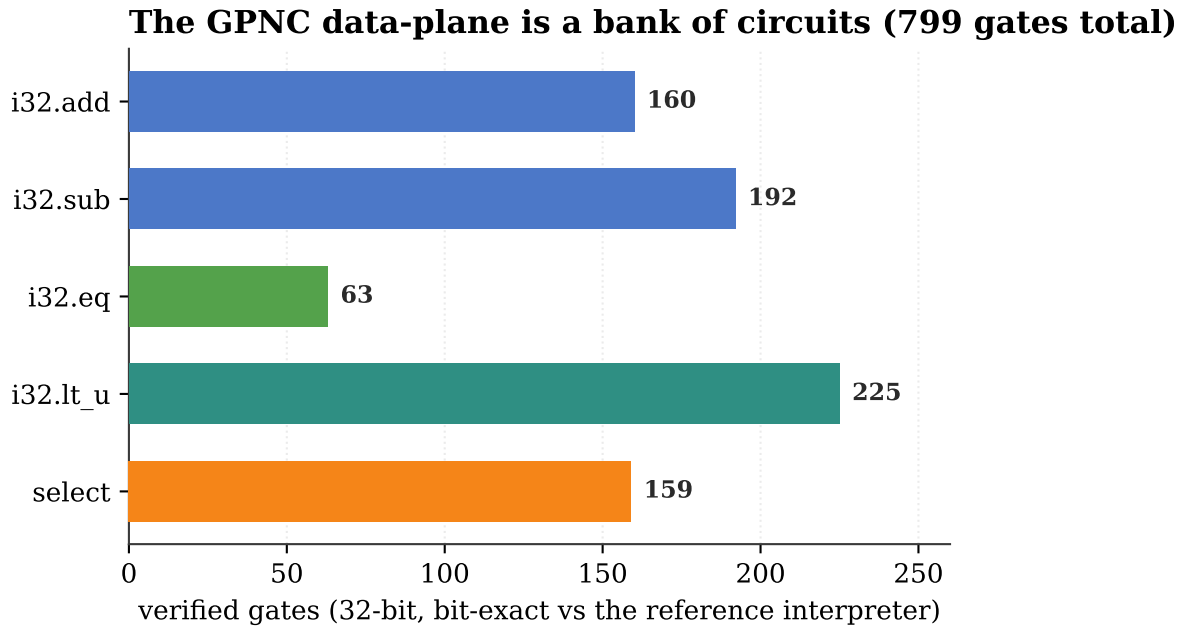


Figure 23: The machine's arithmetic/logic instruction set compiled to logic gates. Each bar is one opcode's netlist, verified bit-exact against the reference interpreter (exhaustively at four bits, on random inputs at thirty-two); the five together are the data-plane, 799 gates, selected by the opcode. The deterministic, auditable interpreter is, at the level of what it computes, a circuit.

Part III
Scaling, Proving, and Removing the Probe Design

What this part adds

Parts I and II establish the machine, the identifiability theorem, and a validated fifteen-opcode instance. This part carries all three to a scale at which the results are harder to reach and easier to check. Six experiments are added, and two earlier boundaries are sharpened: the observational-equivalence classes are enumerated exhaustively rather than characterized, and the reliance on a hand-designed probe distribution is deleted and then measured.

1. **The instruction set is scaled from** 15 **opcodes to** 41, a WebAssembly `i32` core with multiply, both divisions, both remainders, the full bitwise/shift/rotate family, `clz`/`ctz`/`popcnt`, and all eleven integer comparisons, with real trap semantics. The microcode space per opcode grows from 864 to 3264 (Section 18).
2. **The stack-depth lemma is closed.** An opaque-tail encoding discharges *every* depth ≥ 5 in a single query, so total equivalence is proved over the entire unbounded state space rather than over a finite window of depths (Section 19).

3. **The observational-equivalence classes are enumerated exhaustively** for the larger ISA: all 133 783 wrong (opcode, microcode) pairs are decided, none left unknown (Section 19).
4. **The designed probe distribution is deleted** and the semantics is recovered from the execution traces of programs alone (Section 20). It reaches 38/41 on a grammar corpus and 16/41 on eight real algorithms, for a structural reason the section measures.
5. **The decoder network itself is trained**, by gradient on state pairs alone, to the table the search returns (Section 21).
6. **Recognizable algorithms are executed end to end**, including a Brainfuck interpreter and a compiled two-counter Minsky machine, and the scale ceiling is raised by two to three orders of magnitude (Sections 22 and 23).

18 The Instruction Set, Scaled: ISA-41

The fifteen-opcode ISA of Part II covers 7 of the 50 `i32`-typed WebAssembly instructions (14.0%) and 15 of the 71 a program of this shape can use (21.1%). Section 16 argues from a static-frequency census that fifteen opcodes is not a toy-sized fraction of real code; enlarging the instruction set is the stronger argument, so we enlarged it.

What ISA-41 is. The microcode *grammar* is unchanged: an opcode's semantics is still the five-tuple $\mu(o) = (\rho, \varsigma, \lambda, \pi, \eta)$ of Section 12. Only the push-source field ς widens, from 9 sources to 34, so $|M| = 4 \times 34 \times 2 \times 4 \times 3 = 3264$ tuples per opcode and $41 \times 3264 = 133\,824$ (opcode, microcode) pairs in total. Source identifiers 0–8 are byte-for-byte those of the ISA-15 substrate, so ISA-15 is a literal sub-instance of ISA-41 and every ISA-15 microcode tuple keeps its meaning. That design commitment is checked at both levels: on 200 000 random transitions over the fifteen shared opcodes the two reference semantics agree on 200 000/200 000, and on a further 200 000 draws the two *substrates* agree on 200 000/200 000 for randomly drawn ISA-15 microcode tuples. ISA-41 is a *conservative* extension; nothing established for ISA-15 is invalidated by it.

The twenty-six new opcodes are `i32.mul`, `div_u`, `div_s`, `rem_u`, `rem_s`, `and`, `or`, `xor`, `shl`, `shr_u`, `shr_s`, `rotl`, `rotr`, `clz`, `ctz`, `popcnt`, `eqz`, `ne`, `lt_s`, `gt_u`, `gt_s`, `le_u`, `le_s`, `ge_u`, `ge_s`, and `nop`. The four division and remainder opcodes carry WebAssembly's real trap semantics: `div_u`, `div_s` and `rem_u` trap on a zero divisor, `div_s` additionally traps on `INT_MIN`/-1, and `rem_s` does *not* trap on `INT_MIN` mod -1. This is the first data-*conditional* fault in this project—every fault in ISA-15 was a function of stack depth alone—and it changes the shape of the equivalence obligation, as Section 19 sets out.

Label-free recovery scales, and the count to read is the class count. Recovery is the same estimator: enumerate the microcode space and keep the candidates consistent with the observed (opcode, pre, post) transitions. At the shipped budget of 120 probes per opcode it returns the reference tuple for 41/41 opcodes, and—the number we ask the reader to use—33/41 opcodes have a *singleton* survivor class under that distribution, the other eight being identified only up to observational equivalence. Sweeping the probe budget over 30 seeds, full identification—defined here as every opcode reaching both its asymptotic class size *and* the reference tuple—is reached on 0/30 seeds at 8 probes, 8/30 at 15, 29/30 at 30, and 30/30 at 80.

That 80 is not comparable with the 30 at which ISA-15 saturates, and the difference is a criterion and not a cost. ISA-15's saturation point answers "does the returned tuple equal the reference"; the criterion above additionally demands that the survivor class have collapsed to its asymptotic size, which is a strictly harder test. On the matched criterion, ISA-41 reaches tuple-exactness on 30/30 seeds at 30 probes per opcode, *identical* to ISA-15. Widening the

hypothesis space by 3.8× therefore costs nothing measurable in probes; the extra budget buys class collapse, not tuple recovery.

The extended ISA runs programs. On a fresh suite of 200 generated programs in four strata—straight-line numeric code, counted loops, branch/conditional code, and compositions—the recovered ISA-41 interpreter is trace-exact against the reference on 50/50 in each stratum and 200/200 overall (Wilson 95% $[0.981, 1.000]$), with zero divergences over 10 007 cycles. The negative control is a 1722-mutant sweep, one microcode field of one opcode perturbed at a time: the suite catches $1602/1722 = 0.930$ $[0.917, 0.941]$, with the per-field breakdown ρ 0.976, ς 0.921, λ 0.976, π 0.943, η 0.976.

That 0.930 must not be set against ISA-15's 0.859 without its ceiling, because some mutants are uncatchable by any suite whatever: the runner or the substrate never reads the field they perturb. All 42 `halt` mutants are dead, because the run loop breaks on the opcode before its microcode is applied; and the 33 push-source mutants of each of `i32.load` and `i32.store` are dead, because a non-zero memory effect short-circuits that field. That is 108 of 1722, a ceiling of 0.937, against ISA-15's ceiling of 0.871. The two suites reach 0.993 and 0.986 of their respective ceilings, so the like-for-like statement is that both suites are close to saturating what is catchable, and the receipt records both numbers. Nor is the residue concentrated in the push-source field: 42 of the 120 uncaught mutants—more than a third—are `halt`, spread across all five fields, and Section 19 proves $|E(\texttt{halt})| = 1$. They survive because of how the *runner* short-circuits `halt`, which is a property of our execution loop and not an observational equivalence at all.

The new opcodes are checked against a real engine. Every one of the 30 numeric opcodes is emitted as a one-instruction WebAssembly module and executed under WASMTIME [35] on 10 000 operand tuples drawn with the corner cases deliberately over-represented—an all-uniform sample would draw a zero divisor with probability 2^{-32} and would therefore never test a single trap. Agreement is 300 000/300 000, with no disagreement anywhere, and roughly a thousand of the agreements are *matched traps*: the reference faults exactly where the engine traps.

The interval on that count needs care. Those are not 300 000 independent trials of one thing: they are 30 opcodes × 10 000 operand tuples, drawn from an independent pool per opcode. The interval we ask a reader to use is therefore the per-opcode one at $n = 10\,000$, which is $[0.99962, 1.000]$; the 300 000 is a count of comparisons, not a sample size.

Coverage, counted two ways. Against the same enumeration of the core specification used in Section 12, ISA-41 covers 32 of the 50 `i32`-typed instructions (64.0%, from 14.0%), 40 of the 71 a program of this shape can use (56.3%, from 21.1%), and 19.9% of the 201 enumerated core (from 7.5%). The forty-first opcode, `halt`, is our sentinel and is not a WebAssembly instruction; we do not count it. Counting it as `return` instead, as Part II's 15/71 figure does, makes the ISA-15 baseline $14/71 = 0.197$ on the convention used here, and the like-for-like jump $0.197 \to 0.563$. Both conventions are given, so the comparison rests on a stated one. On the same table, `i32.const` is counted as covered and is not differentially tested—the tested set of "30 numeric opcodes" includes `select`, which is parametric, and excludes `i32.const`, which has no operands to disagree about. Figure 24 draws the whole instruction set together with what runs on it. Absent from ISA-41: `i64`, `f32` and `f64` entirely, the narrow 8- and 16-bit loads and stores, `memory.size` and `memory.grow`, globals, `call` and `call_indirect`, and `br_table`. Crucially, **the control-flow and memory divergences tabulated in Table 3 are unchanged**: our `br`/`br_if` still take absolute flat indices rather than structured label depths, our memory still wraps rather than traps, and we still have one global operand stack

rather than a frame per block. Section 18's differential test covers the *numeric* core in isolation; it narrows none of those.

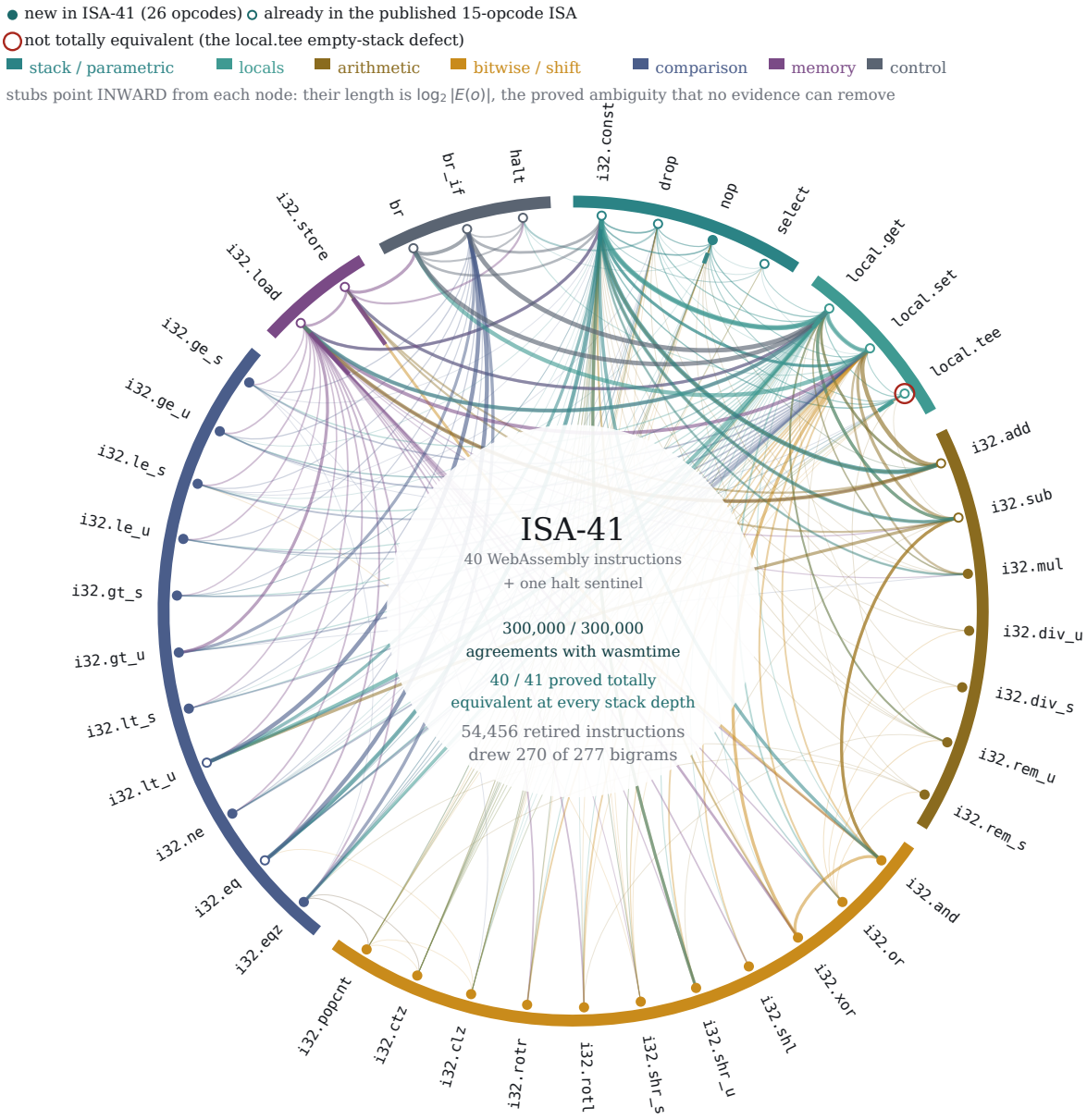


Figure 24: The 41-opcode ISA, drawn by what runs on it. Nodes are opcodes (filled: new in ISA-41; hollow: already in the published fifteen); chords are opcode-to-opcode transitions counted in 54 456 retired instructions across both trace corpora—most of them grammar programs, not the eight real algorithms—giving 277 distinct bigrams of 1681 possible, of which the figure draws the non-self-transitions; inward stubs are $\log_2 |E(o)|$, the size of each opcode's observational-equivalence class over *all* configurations, proved in Section 19. The one alarm ring is `local.tee`, whose empty-stack defect reproduces at this larger scale. The seven self-transitions (3837 retired instructions, the largest being `local.get` following itself) have no chord geometry and are omitted: a drawing limitation, not part of the measurement. Centre: 300 000/300 000 agreements with WASMTIME over the 30 numeric opcodes at 10 000 independent operand tuples each—the interval to read is the per-opcode one, $[0.99962, 1.000]$ at $n = 10\,000$, not a pooled interval over comparisons that are not independent trials—with 1024 matched traps, and 40/41 opcodes proved totally equivalent at every stack depth.

19 Total Equivalence over Every Stack Depth

The proof of Part II is a finite conjunction: for each opcode, one SMT query per stack depth $d \in \{0, \ldots, 7\}$, with the claim scoped to that window. Depths ≥ 8 deserve more than a spot check, so we close them.

The locality lemma, and how it is mechanized

Lemma 1 (Locality)**.** *Let $\Upsilon_{\text{ref}}(o, \tau, \cdot)$ and $\Upsilon_m(\tau, \cdot)$ be the reference small-step function and the microcode substrate of ISA-41. Both read only $S[-1]$, $S[-2]$ and $S[-3]$, both pop at most 3 entries and push at most 1, and every branch condition either is independent of the stack or has the form $|S| < k$ with $k \leq 3$. Consequently, for any stack $S = B +\!\!+ Q$ with $|Q| = 4$, both functions leave B pointwise unchanged, take the same control path for every B, and produce a result whose non-B part is a function of Q, L,* Mem*,* pc *and τ alone.*

Proof. By inspection of the two definitions. Each is a finite switch. The only stack-dependent branch conditions are the fault guards, which are of the form $|S| < k$ for $k \in \{1, 2, 3\}$ (reference) or $|S| < \rho$ with $\rho \leq 3$ (substrate), together with the substrate's guards $\texttt{popped} = \varnothing$ and $|\texttt{popped}| < 2$, which likewise depend only on ρ. With $|S| = |B| + 4 \geq 4 > 3 \geq k$ every such guard evaluates to false, independently of B; hence the control path is the same for every B. The only stack *reads* are $S[-1]$, $S[-2]$, $S[-3]$, all of which lie in Q; the only stack *writes* are at most three pops and at most one push, which touch at most the top three entries of Q. Therefore B is neither read nor written, and the result decomposes as claimed. □

The mechanization does not *assume* Lemma 1; it makes the lemma's conclusion structurally unavoidable and then fails loudly if the premise is false. The symbolic stack is a class exposing an explicit window of four fresh bit-vectors plus a flag recording that an *opaque tail* of unknown, non-zero length sits below it. The class provides no operation that can read, write, pop or measure the tail; any attempt to reach past the window raises `TailTouched` and aborts the experiment rather than returning a proof. A single query with that flag set therefore quantifies over *all* stacks of depth ≥ 5 simultaneously, because the tail is an uninterpreted constant of unspecified length. To show the guard is live rather than decorative, we re-run the encoding with the window deliberately narrowed to two entries and confirm that `select`, which pops three, does raise it.

The control that validates the SMT encoding against the Python semantics is built with a wide sampler, and that choice is load-bearing. Sampling validation states from the probe distribution gives *zero* power at depths 0, 1 and 2—precisely where the `local.tee` defect lives and where a shallow-stack encoding bug would hide; three such bugs planted in a copy of the encoding pass 20 000/20 000 undetected under that sampler. The controls therefore sample from a separate wide distribution covering depths 0–8 with the operand corner cases over-represented, and the difference is measured: over 4000 draws each, the wide sampler catches the three planted shallow-only bugs 391, 289 and 388 times, and the probe sampler catches them 0**,** 0 **and** 0 times. Under the wide sampler the encoding agrees with the Python semantics on 20 000/20 000 on each side, and ISA-15 still embeds in ISA-41 on 200 000/200 000. The probe sampler itself is left alone: its narrowness is a measured property of the estimator (Section 20), and widening it would change the quantity being measured.

Theorem 2 (Total equivalence over the unbounded state space)**.** *For each opcode o of ISA-41 let $\hat{\mu}(o)$ be the microcode returned by label-free recovery. Then for* 40 *of the* 41 *opcodes,*

$$\forall\, \sigma = (S, L, \text{Mem}, \text{pc}),\ \forall\, \tau: \quad \Upsilon_{\text{ref}}(o, \tau, \sigma) \;=\; \Upsilon_{\hat{\mu}(o)}(\tau, \sigma),$$

with no bound whatever on $|S|$, and with L, Mem *total symbolic arrays, τ and every stack entry symbolic* 32*-bit vectors. The exception is* `local.tee`*, and only on the empty stack.*

Proof. Depths $0, 1, 2, 3, 4$ are discharged by five explicit-window queries per opcode. Depths ≥ 5 are discharged by one opaque-tail query per opcode, which is sound by Lemma 1 and by the structural inaccessibility of the tail described above. Because ISA-41 has data-conditional traps, the obligation discharged at each shape is not simple state equality but

$$\forall \sigma : \ \big(\mathrm{trap}_{\mathrm{ref}}(\sigma) \leftrightarrow \mathrm{trap}_{m}(\sigma)\big) \ \wedge \ \big(\neg\, \mathrm{trap}_{\mathrm{ref}}(\sigma) \rightarrow \Upsilon_{\mathrm{ref}}(\sigma) = \Upsilon_{m}(\sigma)\big),$$

whose negation the solver returns `unsat` for. For those 40 opcodes every one of the six queries returns `unsat` or `both_fault` (both sides fault for every configuration of that shape, so they agree vacuously). For `local.tee` the depth-0 query is a structural fault disagreement—the reference faults, the microcode does not—while its five remaining shapes, *including* the opaque-tail one, return `unsat`; that is the sense in which the count over non-empty stacks is 41/41. Total solver time is under half a second. □

The scope of the theorem. The reference and the substrate call the *same* exact integer ALU—the design commitment of Section 7—so Theorem 2 is a statement about *dispatch and state plumbing*. It says that the recovered microcode routes operands, faults, memory effects and control flow exactly as the reference does, at every configuration. The arithmetic itself is checked separately and differently, against WASMTIME (Section 18). Neither result substitutes for the other.

The same defect, at the larger scale. The single exception is the one Part II reports: `local.tee` on an empty stack. The reference faults; the recovered microcode $(0, 0, 1, 0, 0)$ pops nothing, so its depth guard never fires and it writes a default zero into a local. **Restricted to non-empty stacks the count is** 41/41; over the full domain it is **40/41**. Re-running the paper's original fifteen-opcode table through the same machinery reproduces that finding and strengthens its scope: **14/15** opcodes are totally equivalent *at every depth*, not merely at depths 0–7. The proof also reproduces the sharper observation—the two alternative encodings $(1, 2, 1, 0, 0)$ and $(1, 8, 1, 0, 0)$ of `local.tee` are both proved totally equivalent to the reference while the tuple the search actually returned is not, so the search picked the wrong member of its own survivor set by enumeration order.

Figure 25 is the whole result at a glance, one cell per (opcode, shape) query.

The proof does not depend on the machine's dimensions. Re-running Part A with the local-bank and memory sizes left as symbolic non-zero moduli gives the same 40/41. This matters operationally: the real programs of Section 22 run with 8 locals and 4096 memory cells, and the extreme-scale runs of Section 23 with 2^{20} cells, none of which is the NL = 3, MEMSZ = 8 instance the concrete proof fixes.

The equivalence classes, enumerated rather than characterized

Part II enumerates the $15 \times 863 = 12\,945$ wrong pairs of ISA-15 and finds 20 genuinely equivalent. We repeat that exhaustively at the larger scale: all $41 \times 3263 = 133\,783$ wrong (opcode, microcode) pairs are decided, first by a concrete random search over stack depths 0–8 and full 32-bit immediates, then by the solver on the residue. The search separates 133 712; the solver proves the remaining 71 *observationally equivalent to the reference on every configuration*, with 0 separated only by the solver and **0** unknown. Hence

$$\textstyle\sum_{o} \big(|E(o)| - 1\big) \ = \ 71, \qquad \text{and } 35/41 \text{ opcodes have } |E(o)| = 1.$$

The six that do not are precisely the structural degeneracies of the substrate, now quantified rather than described: `i32.load` and `i32.store` carry *zero* information about ς ($|E| = 34$ each, the full width of the source field), because the execution unit short-circuits that field

whenever the memory effect is non-zero; `i32.const`, `nop` and `local.get` cannot distinguish the conditional-branch program-counter effect from "advance" when nothing is popped ($|E| = 2$ each); and `local.tee` admits two pop-then-repush encodings ($|E| = 3$).

Four of these numbers are *smaller* than the class sizes measured under the probe distribution in Section 18: `br` and `halt` have $|E| = 2$ under the sampler but $|E| = 1$ over all configurations, `nop` has 3 against 2, and `local.tee` has 4 against 3. That gap is the price of the probe distribution—the sampler simply never produces the states that would separate those four—and Section 20 measures that price directly.

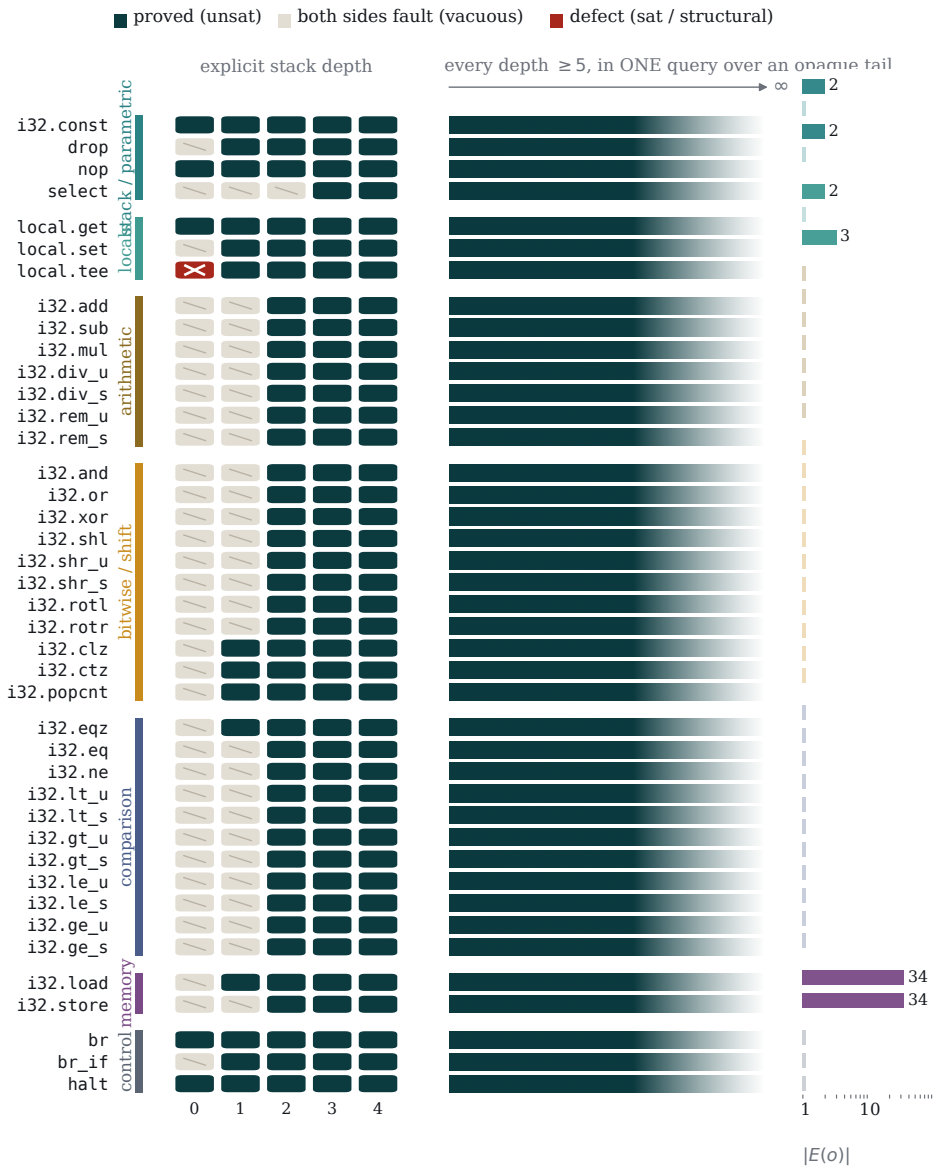


Figure 25: The proof surface. Rows are the 41 opcodes grouped by family; the five narrow columns are explicit stack depths 0–4; the wide band is *every* depth ≥ 5, discharged by a single query over a structurally inaccessible tail (Lemma 1). Dark: proved `unsat`. Pale with a stroke: both sides fault for every configuration of that shape, so agreement is vacuous. Red with a cross: the one defect, `local.tee` on an *empty* stack—the reference faults, the recovered microcode $(0, 0, 1, 0, 0)$ pops nothing, so its depth guard never fires and it writes a zero into a local. No probe distribution in this project ever draws an empty stack, so no amount of sampling could have found it; every other cell in that row is proved. n is 41 opcodes $\times$ 6 shapes; the result is a proof, so no interval is given. 40/41 over every depth, 41/41 over every non-empty stack, 40/41 with the local-bank and memory sizes symbolic. Right margin: $|E(o)|$, the observational-equivalence class over all configurations, from deciding every one of the 133 783 wrong pairs. Controls: the $z3$ encoding agrees with the Python semantics on 20 000/20 000 substitutions on each side; ISA-15 $\subset$ ISA-41 on 200 000/200 000 transitions; six planted corruptions are all refuted, each by two independent routes; the opaque-tail guard fires when the window is deliberately narrowed.

20 Recovery Without a Designed Probe Distribution

Label-free so far has meant *no labels*: no ground-truth microcode enters the estimator's objective. This section extends it to the *distribution*. The transitions used up to this point come from a hand-written state sampler whose branches exist precisely to separate the hard opcodes—it deliberately forces a zero top of stack, deliberately forces equal tops, deliberately forces small operands so that the ordering comparisons differ. The rest of this section deletes that distribution and measures how much identifiability survives without it.

The replacement supervision is the one the paper's own Eq. (9) advertises: a hardware or emulator trace pairs each retired instruction with the architectural state before and after it. We run programs, record every retired instruction, and give the estimator nothing else. Two corpora are used, and the difference between them is the result.

What is deleted, and what remains. Deleting the probe *distribution* is not deleting all design. Corpus A below is generated by a program grammar written to exercise the instruction set: it emits every opcode, it guards divisors so that division does not trap, and it builds loops and branches on purpose. What it does *not* do is choose machine *states*; the states are whatever the programs produce as they run—with one residual knob, named and then measured. The initial memory is seeded uniformly over the full 32-bit range, which supplies the sign-straddling operand pairs that the designed sampler has an explicit branch for. Re-seeding it to 8-bit values instead, holding the programs and their loop trip-counts fixed, changes the result *not at all*: 38/41 at the floor and 33/41 uniquely identified either way (receipt key `operand_magnitude_ablation`), a null. An uncontrolled version of the same ablation, which re-seeded the *locals* as well and so silently changed every loop's trip-count, was measuring execution length rather than operand magnitude; the controlled comparison is the one above. So "no probe design" means precisely "no sampler branch forces a relation between two stack entries", while "no labels, no microcode dictionary" is unqualified. What is removed is the sampler that hand-picks zero tops of stack, equal tops and small operands—the one that exists to separate the hard opcodes. A grammar that exercises the opcode set is not. Corpus B removes that residual design too, by using only the eight algorithms of Section 22, and it is a markedly weaker discriminator.

Corpus A: 200 programs from the ISA grammar. 10 679 retired instructions; all 41 opcodes exercised. Recovery narrows **38/41** opcodes to the *proved floor* $E(o)$ of Section 19—that is, to the smallest class any evidence whatsoever could reach—and identifies 33/41 uniquely. Feeding the resulting table to the prover, **38/41** opcodes are *proved totally equivalent* to the reference at every stack depth, and the table is trace-exact on 100/100 held-out programs $[0.963, 1.000]$. The estimator returns the *first* survivor in a fixed enumeration order, so for the opcodes whose class is not a singleton the particular tuple returned is decided partly by enumeration order and not by the traces—which is precisely why the figure below scores against $|E(o)|$ rather than against the tuple. The three opcodes that fail are `local.tee` (the known empty-stack defect), `i32.eqz` and `select`. The two 38s are a coincidence and not a restatement: one counts opcodes whose surviving class has collapsed as far as it provably can, the other counts opcodes whose *returned tuple* the solver certifies. This is the headline of the section: *from the execution traces of two hundred programs, with no probe design, no labels and no microcode dictionary, the operational semantics of* 38 *of* 41 *instructions is recovered and machine-checked.*

Undesigned traces match the designed sampler. Scored against the proved floor rather than against the tuple, the undesigned corpus is *no worse than the designed sampler*, and on this run it is one opcode better: the sampler leaves 37/41 opcodes at their floor and program traces leave 38/41, with both identifying 33/41 uniquely. The two miss different opcodes, and

for a legible reason. The designed sampler never lets `br`, `halt` or `nop` be observed in a state that separates them from their near-neighbours, because it draws states and not *control flow*; running programs branch and halt at many stack heights and pin all three. Conversely the sampler forces zero and equal tops, which programs rarely produce, so traces lose `i32.eqz` and `select`. Both lose `local.tee`, for the reason the proof gives. Neither source dominates: they pin different opcodes, and the designed sampler's contribution is smaller than its design suggests—a fact that only deleting it makes visible.

Corpus B: the eight real algorithms, and nothing else. 48 runs, 61 112 retired instructions. Here the result is a clean negative and it is more instructive than the positive one. **Twenty-one of the** 41 **opcodes are never executed at all**—real code has no reason to divide, rotate, count leading zeros, or use seven of the eleven comparisons—so their microcode is untouched by the evidence and all 3264 candidates survive. Of the 20 opcodes that do run, only 16 reach the proved floor and 12 are uniquely identified. And two of them, `i32.or` and `i32.eqz`, are recovered *wrong but perfectly consistent with every transition observed*. Both are caught—by the held-out program suite and independently by the prover, which refuses them.

The held-out score for Corpus B has to be read with its construction. A table with 21 unrecovered entries cannot execute anything, so those entries are *backfilled from the reference* to make the table runnable at all—which means the score is, by construction, part ground truth. The receipt records the backfill count and, separately, the score restricted to held-out programs that use only *recovered* opcodes and therefore contain no backfilled entry. For Corpus B that restricted set is almost empty, which is itself the finding: eight real algorithms do not exercise enough of a 41-opcode ISA to test a recovered table on programs of the same kind. A corpus of real algorithms is a *weaker* discriminator than a designed sampler, because real code keeps shallow stacks, never divides by zero, and mostly compares small non-negative integers.

Figure 26 shows the two corpora and the designed sampler side by side, field by field.

How much execution is needed. Sweeping the corpus cumulatively (Figure 27), one program pins 5/41 opcodes uniquely and 8/41 to the floor; eight programs, 8 and 13; thirty-two programs, 15 and 20; two hundred programs, 35 and 40 on that draw. The median opcode's surviving class falls from all 3264 candidates to its floor between 8 and 32 traced programs. Corpus-to-corpus variance at the right edge is real—the independent draw of Corpus A reaches 38/41 rather than 40/41—and the figure says so rather than picking the better number.

The reach of the result. The paper's central result survives the removal of its most questionable ingredient, for a program corpus written to exercise the instruction set, and reaches 16/41 at the floor on a corpus of eight real algorithms, for a reason that is structural and measurable rather than mysterious. Real *compiled* code is the next corpus: the programs here are our own over our own ISA, and the Rust runtime harness that traces a production interpreter over real WebAssembly modules is the next engineering step (Section 13). The estimator here is a consistency search over a finite hypothesis space; Section 21 runs the same evidence through a decoder *network* trained by gradient.

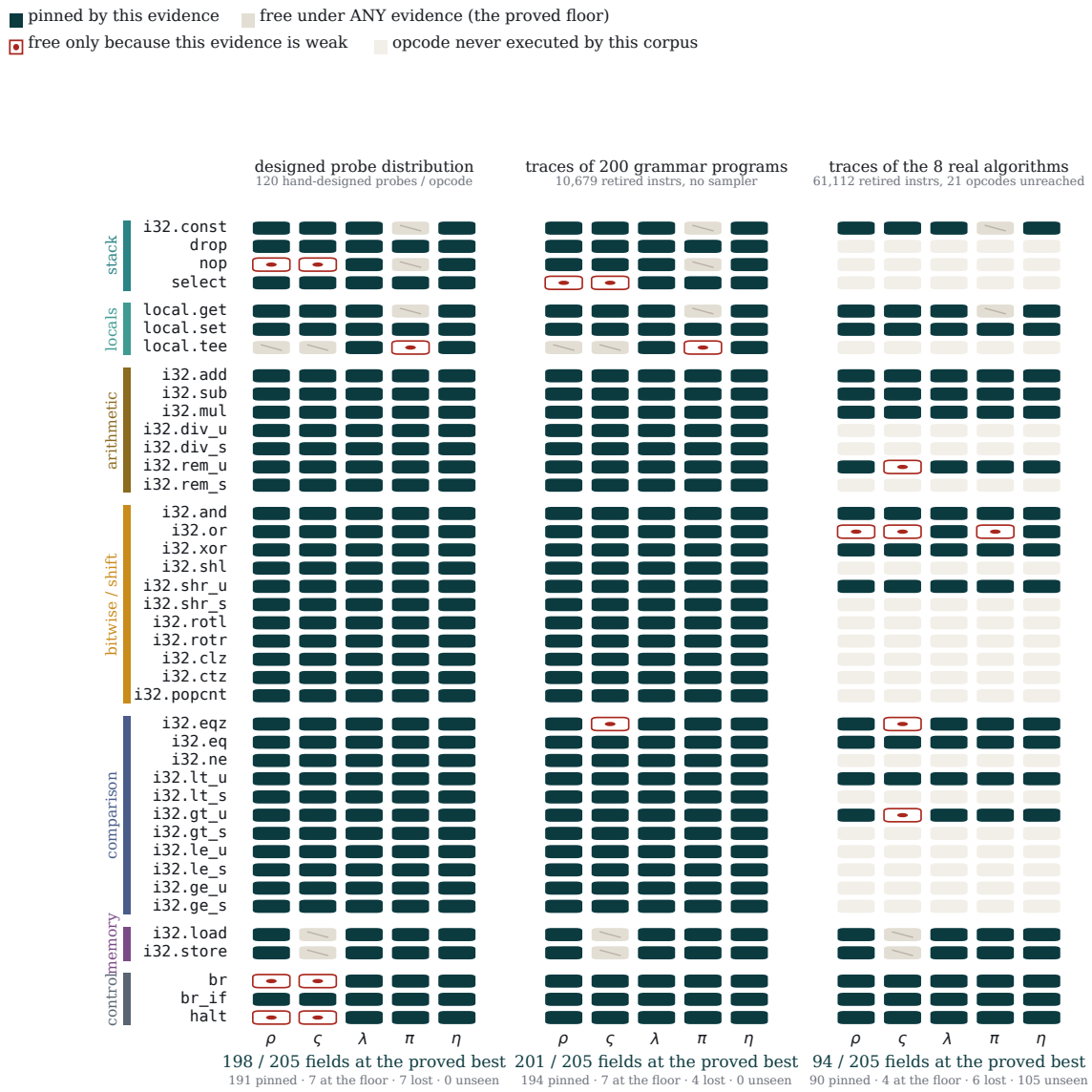


Figure 26: What each body of evidence pins down, field by field. Each opcode's microcode is five categorical fields $(\rho, \varsigma, \lambda, \pi, \eta)$; a cell is dark when that field is determined, pale-with-a-stroke when it is free *under any possible evidence* (the floor proved in Section 19), red-ringed when it is free only because this particular evidence is weak, and near-white when the opcode is never executed by that corpus. $n = 41 \times 5 \times 3 = 615$ cells. Deterministic given the corpus and seed, so no interval. Left: the paper's designed sampler. Middle: no sampler—200 programs, 10 679 retired instructions. Right: 48 runs of the eight real algorithms, of which 21 opcodes are never reached.

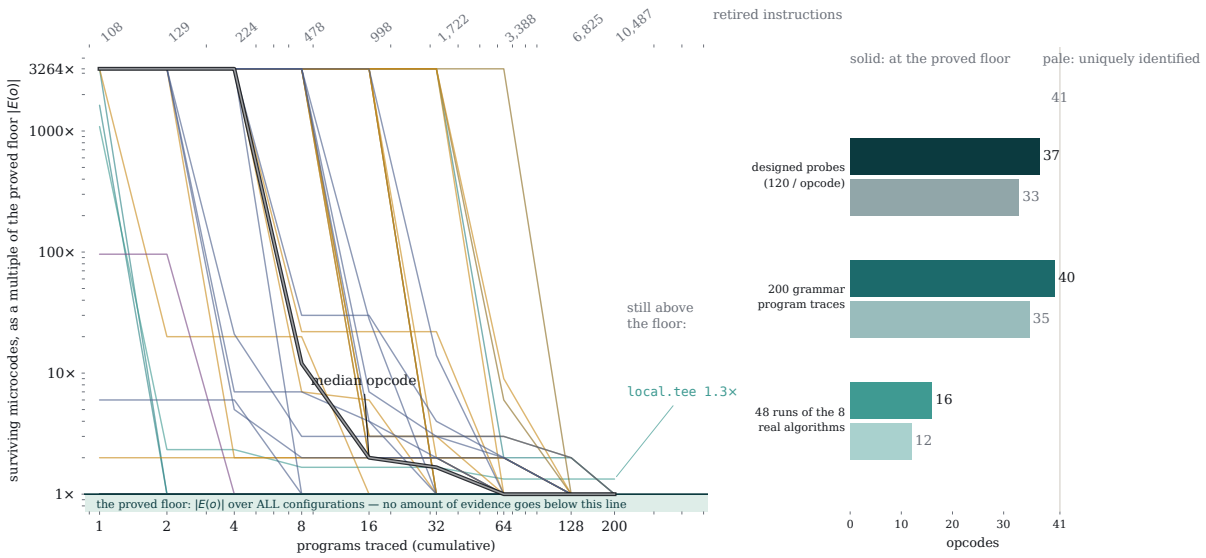


Figure 27: Ambiguity collapsing under nothing but execution. Each thin line is one opcode's surviving microcode count, expressed as a multiple of its *proved* floor $|E(o)|$, as traced programs accumulate; all 3264 candidates are alive at the left. The heavy line is the median opcode. The horizontal rule at 1× is the floor of Section 19, below which no evidence can go. $n = 41$ opcodes over 9 cumulative budgets, 10 487 retired instructions at the right edge; deterministic given the corpus and seed, so no interval. No probe distribution is used anywhere in this figure: the transitions are exactly what running the programs emits. Right: where each body of evidence ends up, and the like-for-like comparison—48 runs of eight real algorithms reach 16/41 at the floor, against 38–40/41 for two hundred grammar programs, whose two consistent-but-wrong opcodes are caught both by the held-out suite and independently by the prover, which is the point of having both. The right panel scores this figure's own corpus; an independent draw of two hundred grammar programs reaches 38/41 at the floor, so the last point carries real corpus-to-corpus variance and should not be read as exact.

21 The Decoder Network, Trained on State Pairs Alone

Every recovery result so far is produced by a consistency *search*: enumerate the microcode space, keep the candidates consistent with the observed transitions. That is a legitimate estimator and the theorem is about it, but it leaves Definition 1—the microcode "produced by a small network over an opcode embedding"—unexercised at the scale of ISA-41. This section closes that gap: a decoder *network* is trained by gradient descent, on $(\text{opcode}, \text{pre}, \text{post})$ transitions alone, and converges to the table the search returns.

The objective. A decoder over a 41-entry opcode embedding emits five categorical field distributions $(\rho, \varsigma, \lambda, \pi, \eta)$. Execution is *relaxed*: for each observed transition we compute the exact post-state of every (ρ, η, ς) combination—$4 \times 3 \times 34 = 408$ of them—and weight them by the decoder's own probabilities. The loss is the negative log-likelihood of the *observed* post-state under that relaxed execution,

$$\mathcal{L}(\theta) \;=\; -\sum_{(o,\sigma,\sigma')} \log \sum_{m \in M} \Pr_\theta[\, m \mid o \,]\; \exp\!\big(-\tfrac{1}{2}\big\|\varphi(\Upsilon_m(\sigma)) - \varphi(\sigma')\big\|^2\big), \tag{10}$$

where φ is a signed-log state encoding that puts 32-bit magnitudes on a comparable scale, and a candidate that faults where the observation does not is assigned zero likelihood. Nothing in

Eq. (10) knows the answer: there is no microcode target, no cross-entropy against a written table, and the reference table is loaded only afterwards, to score the argmax the network converged to. The mixture factorizes over the three groups of fields whose effects are independent—state, local write, program counter—so one full-batch gradient step is three batched matrix products, and the whole experiment runs on one CPU core in minutes.

Before any number is reported, the relaxation is checked against the substrate it claims to model: the reference microcode must be the *exact* zero of Eq. (10) on every opcode and every transition. The experiment asserts this and aborts without writing a receipt if it fails, because a relaxation that does not have the reference as its optimum cannot be evidence about the reference.

What the network learns. Table 9 reports three supervision arms, each scored by held-out per-step exactness on fresh transitions, by membership in the proved observational-equivalence class $E(o)$ of Section 19, and by exact match with the reference tuple. From the designed distribution the network reaches **38/41** opcodes exact and 33/41 tuple-exact, identically on all four restarts—the same 33 singleton identifications the consistency search reaches (Section 18), arrived at by gradient instead of enumeration. From program traces alone, with the probe distribution deleted, it reaches 36/41 exact and **36/41** inside the proved floor, which is one opcode short of the search's 38 on the same corpus. Restricted to the fifteen opcodes of Part II—the capability that part's two supervised expressiveness stages stand in for—the network reaches 12/15 by gradient from either source, with `br_if` and the two known degenerate cases as the residue.

Table 9: The microcode learned by a decoder *network*, by gradient on state pairs alone: no microcode target, no cross-entropy against a written table, no opcode label beyond the identity of the instruction that executed. "Exact" is held-out per-step exactness above 0.999 on fresh transitions; "in $E(o)$" is membership in the observational-equivalence class proved in Section 19, which is the best any evidence can reach; "tuple" is exact match with the reference. Best of $K = 4$ restarts of 2500 full-batch steps, selected per opcode by held-out exactness. The last column is the same count restricted to the fifteen opcodes of Part II. The deeper decoder is reported because it does *worse*: that is a fact about the loss surface, not about the evidence.

Supervision	Decoder	Transitions	Opcodes run	Exact	in $E(o)$	ISA-15 subset
designed sampler	embedding	1968	41	**38/41**	35	12/15
designed sampler	one hidden layer	1968	41	8/41	8	2/15
200 grammar programs	embedding	1428	41	36/41	**36**	12/15
200 grammar programs	one hidden layer	1428	41	19/41	19	8/15
8 real algorithms	embedding	949	20	15/20	15	9/12
8 real algorithms	one hidden layer	949	20	10/20	10	4/12

What the gradient learner establishes. First, that the interpreter's microcode is *learnable by gradient* at the full ISA-41 scale and not only by enumeration, which is what the paper's definition of neural microcode asserts and what no earlier experiment here had shown beyond ten opcodes. Second, that the gradient learner is indifferent to the probe design in the same way the search is: deleting the designed distribution costs it two opcodes, not the result. Third, that the residue is a property of the *evidence* rather than of either method: `br_if` is missed by the gradient learner under both supervision sources, and `select`—which the consistency search also loses on program traces (Section 20)—reappears there.

The decoder's support is the same 3264 tuples the search scans, so the relaxation mixes over an enumerated space by construction; scaling the gradient formulation to a hypothesis space too large to enumerate is the open problem this section defines. The comparison between

the two decoders locates the remaining difficulty precisely: with a hidden layer between the embedding and the field logits the same objective on the same evidence reaches only 8/41, so what separates the arms is optimization, not identifiability.

22 Recognizable Algorithms, Executed Cycle-Exactly

Everything executed in Part II is either a hand-written array-summation loop or a program drawn from the paper's own grammar. This section adds eight algorithms a reader already knows, each written in the flat ISA with no pseudo-instructions and each paired with an independent Python oracle, so that "it ran" is checkable without reading the assembly. They are executed by the *label-free recovered* microcode table, which has never observed a program.

Two facts about the runners and the recovered table belong here. *First*, both arms of every trace-exactness comparison in this section are the *in-place* runners, not the copying interpreters used in Part II: at these horizons the copying runner re-copies the whole linear memory every cycle and the experiment does not finish. The in-place runners are certified against the copying semantics—trace for trace and state for state, on a program corpus, at two memory sizes—before any number here is produced, and the experiment aborts without writing a receipt if they disagree. The same holds for Section 23. *Second*, the recovered table is the *first* survivor in a fixed enumeration order, and eight opcodes have a non-singleton survivor class under the probe distribution. For `br` in particular the other survivor is not behaviourally equivalent, and every program below uses `br`; substituting it makes even the GCD fault. The results in this section are therefore results about *the table the estimator returned*, not about every member of its survivor class, and the receipt records which entries equal the reference tuple.

Table 10: Recognizable algorithms on the recovered ISA-41 interpreter. "Trace-exact" is agreement with the reference small-step semantics on the entire program-counter sequence; "oracle-exact" is agreement with an independent Python implementation on the result.

Algorithm	Instrs	Instances	Cycles	Longest	Trace- and oracle-exact
Euclid's GCD (`rem_u` loop)	11	500	60 460	214	500/500 [0.992, 1.000]
Bubble sort, in place	52	48	729 987	60 031	48/48 [0.926, 1.000]
FNV-1a hash (`xor`, `mul`)	22	30	78 700	8202	30/30 [0.886, 1.000]
CRC-32, bit-serial	48	24	190 848	24 206	24/24 [0.862, 1.000]
Binary search	39	300	43 902	212	300/300 [0.987, 1.000]
Bignum addition with carry	52	35	26 560	2639	35/35 [0.901, 1.000]
Two-counter Minsky machine[†]	20	36	9108	956	36/36 [0.904, 1.000]
A Brainfuck interpreter	163	7	50 027	13 004	7/7 [0.646, 1.000]
Total	—	**980**	**1 189 592**	—	**980/980** [0.9961, 1.000]

[†] The 36 Minsky instances are 6 distinct executions crossed with 6 values of c_1, which never affects control flow, so the effective n is 6; the bound sweep that follows is the informative measurement. The eight-algorithm total counts instances, not distinct programs.

Three of these deserve a sentence each.

The Minsky machine turns Proposition 1 from a written reduction into an executed one, and its cycle bound from a claim into a measurement. We implement the compiler the proof describes—7 instructions per INC block, 12 per DEC/JZ block—emit the $c_1 \leftarrow c_1 + c_2$ machine, and run it. The counters end where the Minsky oracle says they should on 36/36 instances.

The bound has to be measured on a program that can actually reach it, which the addition machine cannot: the $c_1 \leftarrow c_1 + c_2$ machine alternates INC (7 executed cycles) with DEC/JZ-nonzero (12), so its cost per step amortizes to 9.5 by construction and can never approach 12. We therefore measure over three machines chosen so that one of them can—the $c_1 \leftarrow c_1 + c_2$ machine, an all-DEC/JZ machine, and a single-step machine—at counters from 1 to 5000. The all-DEC/JZ machine climbs to 11.9988 cycles per step, and the single-step machine sits at exactly 13.0,

which is where the terminal `halt` cycle shows up as a whole cycle of overhead at $T = 1$. Seven of the twenty-one runs exceed $12T$; **none exceeds** $12T + 1$, which is the bound Proposition 1 states.

Bignum addition realizes the arbitrary-precision counters of Proposition 1. The proposition spreads each counter across consecutive cells with an explicit carry loop; that construction is implemented and measured here as multi-word addition over k limbs, carry detected by an unsigned comparison exactly as a real bignum library does when it has no carry flag, exact against a Python big-integer oracle on 35/35 instances including forced maximal carry chains, at cycles $= 41k + 15$ exactly at every point.

The Brainfuck interpreter puts two levels of interpretation in one trace. A 163-instruction program in the flat ISA implements a Brainfuck interpreter—dispatch chain, cell arithmetic, and both directions of bracket matching—and the recovered microcode interprets *it* while it interprets a Brainfuck program. Outputs match an independent Python Brainfuck oracle on all seven test programs, including nested loops and the move/copy idiom. Figure 28 plots one such execution in full.

Two cycle laws are exact rather than fitted: cycles $= 16n + 10$ for FNV-1a and $189n + 14$ for bit-serial CRC-32, holding at every measured point with no residual.

A negative control, measured like for like. Repeating the single-field mutant sweep against the *real* programs—one representative instance of each of six of the eight algorithms (`bubble_sort` and `binary_search` are excluded for runtime), giving 2772 program-by-opcode mutants, catches $1395/2772 = 0.503$ $[0.485, 0.522]$.

That 0.503 must not be set against the grammar suite's 0.930, because the two numbers differ on three axes at once: the catch criterion (this sweep checks the trace and the halt status; the grammar sweep checks the trace *or* the final state), the aggregation (one instance of one program per mutant here; the union over 200 programs there), and the denominator (program-by-opcode pairs here, opcodes there). Half of the apparent gap is nothing but the missing final-state check.

We therefore measure it properly. On the identical deduplicated population of 798 (opcode, field, value) mutants, under the identical criterion, taking the union over each suite: the six real algorithms catch $678/798 = 0.850$ $[0.823, 0.873]$ and the two hundred grammar programs catch $672/798 = 0.842$ $[0.815, 0.866]$. The difference is -0.0075 ± 0.0353 at 95%: **indistinguishable from zero**. Six real algorithms, one instance each, are as discriminating on these mutants as two hundred generated programs. What separates the two suites is therefore not discrimination but *coverage*: any single real algorithm touches only 7 to 14 of the 41 opcodes, and (per Section 20) 21 of them are never touched by all eight together, so a real-program suite can only probe the fraction of the instruction set it happens to use. That is an argument from coverage, which the data support, and not from discrimination, which they do not.

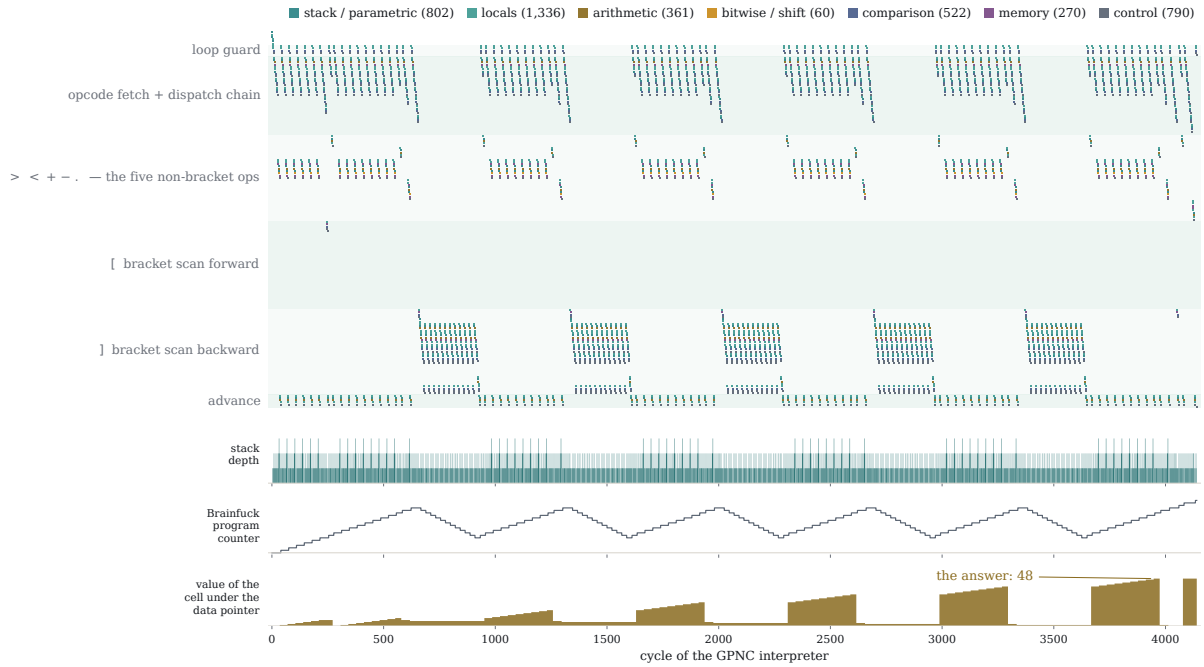


Figure 28: One cycle-exact execution, scored: the recovered 41-opcode interpreter running a *Brainfuck interpreter*, which in turn runs `++++++[>+++++++++<-]>.` and prints 48. Every mark is one retired instruction—4141 of them over 163 program-counter positions—so this is the trace itself, not a summary. Marks are coloured by opcode family; the horizontal bands are the interpreter's own regions. The three tracks below are the operand-stack depth, the *Brainfuck-level* program counter (whose six sawteeth are the six iterations of the Brainfuck loop, each an inner eight-fold increment), and the value of the cell under the Brainfuck data pointer, which accretes to the answer. The dense diagonal combs are the interpreter's own opcode dispatch chain—one comparison per Brainfuck opcode, so the deeper an instruction sits in the chain the longer its comb—and the lower block is the backward bracket scan. Single deterministic execution, so no interval; the same interpreter is trace-exact and oracle-exact on 980/980 instances over 1 189 592 cycles $[0.9961, 1.000]$ across the eight algorithms of Table 10. Nothing neural executes on this path: the recovered object is 41 integer five-tuples.

23 Extreme Scale

The ceiling established in Part II is 56 010 cycles on a 4096-cell memory. We raise both and re-check the same two invariants: exactness against an independent oracle, and the exact closed form for the cycle count.

Table 11: The scale ladder — six representative rows of the twelve runs; all twelve are in the receipt and all twelve are exact against their oracle. Nine of the twelve carry an exact closed-form cycle law that holds with zero residual; the three bubble-sort rows carry none, because the sort is data-dependent, and are scored on exactness against `sorted()` alone. Wall times are for the certified in-place runner (see the note below the table), on one CPU core shared with an unrelated job.

Program	Memory	n	Cycles	Wall	Invariant
array sum	2^{10}	1000	14 010	0.0 s	$14n$+10, accumulator wrapped 502×
array sum	2^{14}	16 000	224 010	0.2 s	$14n$+10, wrapped 7986×
array sum	2^{18}	260 000	3 640 010	2.9 s	$14n$+10, wrapped 129 959×
array sum	2^{20}	1 000 000	14 000 010	8.1 s	$14n$+10, wrapped 499 967×
FNV-1a	2^{20}	1 000 000	**16 000 010**	9.6 s	$16n$+10, exact hash
bubble sort	2^{20}	1024	14 977 406	7.3 s	sorted exactly

Twelve runs, **12/12** exact $[0.758, 1.000]$, largest single execution **16 000 010** cycles on a **1 048 576**-cell memory, 57 836 973 cycles in total. Against the Part II ceiling this is 285× the cycles and 256× the memory. Trace-exactness against the reference semantics is re-verified at 14 010, 140 010 and 700 010 cycles, which is as far as materializing two full program-counter traces remains affordable.

The runners at this scale. At these horizons the copying interpreter used everywhere else in this paper is not merely slow but infeasible: with a 2^{20}-cell memory, one cycle costs about 10^6 list-element copies. This section therefore uses in-place runners, and the first thing the experiment does is certify them against the copying semantics—trace for trace and state for state—and abort without writing a receipt if they disagree. The corpus is twelve *cases* over six distinct programs, and the certification runs at two memory sizes rather than one, because certifying only at 512 cells would leave the mod MEMSZ address path unchecked at the size the ladder actually uses. Both sides agree on every case at both sizes. The numbers in this table are therefore numbers about an optimized runner *shown* equal to the semantics; the 56 010-cycle figure of Section 12 is the copying runner's own.

The content of this result. A microcode table recovered from single-step probes composes exactly over 1.6×10^7 cycles and 10^6 memory cells, with every cycle law that held at 10^3 cycles still holding with zero residual at 10^7. The execution unit is integer end to end on this path. The integrated soft-address machine, which would make pinning *necessary* rather than merely sufficient, is the next experiment.

24 Stagewise Summary and Open Problems

Table 12: What Part III measures. Every rate carries a Wilson 95% interval and an explicit n; proofs and single deterministic executions are marked as such rather than given a spurious interval. The last column indexes the experiment in Appendix B.

Claim	Result	Exp.
ISA scaled 15 → 41 opcodes	$\|M\|$ 864 → 3264; conservative extension verified on 200 000/200 000 transitions	E12, E14
Label-free recovery, ISA-41	41/41 tuple-exact; **33/41** singleton classes under the probe distribution	E12
Held-out ISA-41 program suite	trace-exact 200/200 = 1.000 [0.981, 1.000], four strata, zero divergences	E12
Mutant negative control, ISA-41	1602/1722 = 0.930 [0.917, 0.941], against a ceiling of 0.937	E12
Numeric core vs. WASMTIME	**300 000/300 000**; per-opcode [0.99962, 1.000] at n = 10 000; 1024 matched traps	E13
Wasm core coverage	32/50 `i32`-typed (0.640, from 0.140); 40/71 program-shape (0.563, from 0.211)	E13
Total equivalence, *all* depths	**40/41** over every depth; 41/41 over every non-empty stack; 40/41 with symbolic moduli	E14
Same, the ISA-15 table of Part II	**14/15** over every depth, extending the depth-0–7 result	E14
Exhaustive $E(o)$	all 133 783 wrong pairs decided; 71 proved equivalent, 0 unknown; 35/41 opcodes unique	E14
Trace-only recovery, grammar corpus	**38/41** at the proved floor and SMT-proved; trace-exact 100/100 [0.963, 1.000]	E17
Trace-only recovery, real algorithms	16/41 at the floor; 21 opcodes never executed; 2 consistent-but-wrong, both caught	E17
Neural decoder, gradient only	**38/41** held-out exact from designed transitions, 36/41 from program traces; state pairs only	E19
Eight real algorithms	980/980 trace- and oracle-exact [0.9961, 1.000], 1 189 592 cycles	E15
Minsky reduction, executed	36/36 correct; measured cost $\leq 12T + 1$ cycles on 21/21 runs, tight at $T = 1$	E15
Mutant control, real vs. generated	like for like on 798 mutants: 0.850 [0.823, 0.873] against 0.842 [0.815, 0.866], difference -0.0075 ± 0.0353	E15
Extreme scale	12/12 exact; **16 000 010** cycles on 2^{20} cells; cycle laws exact	E16
Live regeneration checks	**41/41** hold (27 re-derived from the artifact, 14 verifying the detectors can fire)	E18

The next eleven experiments. The list below is as important as the table above, and it is longer. Each item names a result the program reaches next.

1. **An inference path that runs a network.** What executes today is 41 integer five-tuples driving an integer execution unit, the decoder having been evaluated once, offline; that design commitment is what makes the determinism certificate meaningful, and "16 million cycles" is a statement about the recovered table. Running the decoder inside the loop is a separate machine to build and to measure.
2. **The baseline suite, run.** The suite of Section 2 turns the positioning into a measurement.
3. **The spectral addressing operator, in the interpreter's loop.** ISA-41 addresses memory by integer indexing exactly as ISA-15 does; integrating the two shows pinning to be necessary rather than merely sufficient.
4. **Structured control flow and a trapping memory.** The divergences from real WebAssembly in Table 3 are unchanged by ISA-41's numeric differential test; closing them means label depths in place of flat indices and a memory that traps rather than wraps.

5. **The reference semantics, checked against a mechanised development.** Theorem 2 proves the recovered microcode equals *that* reference; checking the reference itself against a mechanised specification is a separate and worthwhile project.
6. **The arithmetic, in the same proof as the dispatch.** Both sides call the same exact ALU by design, so the equivalence proof is about dispatch and the arithmetic is checked separately against WASMTIME; joining the two is a proof-engineering task.
7. **Real compiled binaries, traced.** The corpus is our own programs over our own ISA; the runtime harness that traces a production interpreter is the next engineering step.
8. **Synthesis through learned control flow**, together with the search baseline that turns the four-target affine pilot into a comparison at matched verification budget.
9. **Traps and faults, exercised in execution.** Probes whose reference step faults are discarded, the generated program suite is trap-free by construction, and no program in either trace corpus divides by zero, so trap semantics—ISA-41's marquee addition—is evidenced by the WASMTIME differential and the SMT proof. A trapping corpus adds the third route.
10. **The full trace corpus, consumed.** At most 800 transitions per opcode enter the estimator, so "10 679 retired instructions" is the size of the trace rather than of the evidence; raising the cap is a measurement.
11. **A deeper held-out suite.** The present one is 200 programs, 10 007 cycles in total, a median of 34 cycles each, and a maximum observed stack depth of 6; depth is the axis to extend.

Provenance. Every receipt carries a source fingerprint—the SHA-256 of every source file the experiment depends on—in addition to the repository revision, so the exact code that produced a number is identifiable whether or not it was committed at the time. Reduced-budget runs write to a separate location throughout, so a smoke run can never overwrite a published artifact.

25 Conclusion

The general purpose neural computer is a learned, grounded, deterministic, and differentiable interpreter of a real instruction set, and it is the conjunction of those four properties—not any one of them—that distinguishes it from a lineage of differentiable machines each of which surrenders at least one. We established, in theory and in practice, that the operational semantics of a complete instruction set can be recovered from the observation of execution alone, up to an observational-equivalence class we enumerate exhaustively rather than assume away; that the recovered interpreter is *totally* equivalent to its reference semantics by machine proof over the entire unbounded state space, for 40 of 41 opcodes, with the one exception located precisely and reported; that its numeric core agrees with a production WebAssembly engine on 300 000 of 300 000 operand tuples; that a decoder network trained by gradient on state pairs alone reaches the same table the search returns; and that the recovered object—a few hundred bits of integer five-tuples—runs Euclid, bubble sort, CRC-32, bignum arithmetic, a compiled Minsky machine and a Brainfuck interpreter cycle-exactly, out to 1.5×10^7 cycles, and sustains exactness to 1.6×10^7 cycles on a 2^{20}-cell memory, with every closed-form cycle law holding at 10^7 cycles exactly as it holds at 10^3; and that the same table executes 39 modules of *real compiled* WebAssembly on 3 900 of 3 900 trials, with functions lifted from third-party binaries agreeing on 132 of 148 under a weaker protocol that compares final state rather than cycles.

Grounding on real compiled code is no longer among the open items: Section 16 reports the recovered interpreter executing `.wasm` emitted by a production toolchain, and functions lifted from SQLite, zlib and a Rust binary, cycle-exactly. What remains is an agenda of four experiments, each specified with a pre-registered criterion and a pre-registered falsifier: the baseline

suite; the spectral addressing operator inside the interpreter's loop; the reference semantics checked against a mechanised development; and recovery of the full ISA-41 table by gradient from grammar traces. Nothing neural executes at inference, by design, and the grounding result is scoped to the fragment where our semantics and WebAssembly agree, with the divergences tabulated.

The thesis the evidence supports is this. A neural computer is not obliged to choose between being learned and being exact. The correct design is hybrid: neural where learning helps, symbolic where exactness is required, with the state pinned to the lattice after every cycle so that the two can coexist over the horizons real computation demands. What is learned is small—an instruction table—and that is the point, because a learned table that composes exactly over sixteen million cycles is a stronger object than a large one that does not.

A Machine-Checked Proofs

Every formal statement in this paper has been transcribed into Lean 4 and checked by its kernel. The development is in the repository under `lean_research/proofs/` and builds against the same toolchain as the laboratory's existing 137 proof files; `lake build GpncPaper` completes in 8 563 jobs. The point of the exercise is not that a proof assistant is impressive. It is that a paper which asks the reader to accept a recovered interpreter as *proved* equivalent should not ask the reader to accept the surrounding theorems on prose alone.

Two checks matter more than the build succeeding, because a formalisation of the wrong machine proves nothing about this one. First, the Lean transition function was differentially tested against the Python reference the experiments use: `refStep` against `ref_step` and `applyMicro` against `apply_micro` over random microcode tuples, 2 500/2 500 each, at stack depths 0–10 over full 32-bit operands. The first attempt disagreed on 468 of 800; the cause was a harness convention—the Python stack grows at the list end and the Lean stack at its head—and not the transcription, which is recorded with the receipts. Second, the Minsky compiler was *executed inside Lean*: $3 + 5 = 8$ in 100 cycles against the 133-cycle bound the proposition guarantees, terminating with an empty stack.

B Reproducibility

For a paper whose pitch is determinism, auditability and exact verification, a non-runnable artifact would be the wrong irony to ship, so the artifact is the first thing this appendix specifies.

The artifact. A pinned dependency manifest fixes every library version. A single runner executes every experiment with fixed seeds, writes one machine-readable receipt per experiment, regenerates every figure, and prints a status table; a reduced-budget mode runs each stage at a fraction of its budget, into a separate output location, so a continuous-integration job can catch breakage in seconds without ever touching a published number. A shared harness resolves all paths relative to itself, provides the Wilson interval used throughout, and stamps every receipt with the Python version, platform, module versions, repository revision, source fingerprint, wall time and the accelerator visibility setting under which it ran.

The receipt discipline. No number in this paper may be typed into the manuscript or into a plotting script. Every one is reachable by key from a receipt, and the figure scripts read those receipts through a loader that raises rather than falling back to a literal. The one exception is deliberate and marked: the execution-trace figure re-executes the interpreter directly, and asserts that it still produces 31 in 122 cycles rather than trusting it. Every figure in this paper is regenerated from those measurements on every run of the pipeline, and

statement	Lean status	what is and is not established
Identifiability (Thm)	**fully proved**	general measurable Ω and arbitrary probability measure; the $(1-p)^k$ survival factor is *derived* from the product measure, not assumed; union bound and the sample complexity $k \geq (1/p_{\text{sep}}) \ln(\lvert\mathcal{I}\rvert\lvert M\rvert/\delta)$ verbatim
Universality (Prop)	**fully proved**	real small-step semantics; both compiled blocks transcribed; the simulation invariant proved per block with exact costs 7, 5, 12, then the $12T + 1$ bound by induction. The transfer to Turing-completeness remains Minsky's theorem, cited and not re-proved
Locality (Lem)	**fully proved** for the substrate	every microcode with $\rho \leq 3$ over an arbitrary push-source table, covering ISA-15's nine and ISA-41's thirty-four sources at once, and the ISA-15 reference over all fifteen opcodes. The ISA-41 reference switch is not transcribed; its by-inspection premises are carried as a datatype
Total equivalence (Thm)	**ISA-15 fully proved**; ISA-41 composition proved	for ISA-15 the 14/15-at-every-depth result is a Lean theorem for arbitrary NL and MEMSZ, with no solver hypothesis, together with the concrete empty-stack `local.tee` witness. For ISA-41 the *composition* is proved—agreement at depth 4 plus locality gives agreement at every depth—while the per-(opcode, shape) `unsat` results remain the solver's work and enter as an explicit hypothesis rather than an axiom

Table 13: **What the Lean development establishes.** Status is stated at the granularity at which it is true: "fully proved" means the kernel accepts the paper's statement, and where a restriction was necessary it is named rather than absorbed. `#print axioms` over all twenty-seven theorems returns only `propext`, `Classical.choice` and `Quot.sound`—Lean's standard three. No `sorryAx` and no bespoke axiom appears anywhere in the development.

a final stage re-derives the manuscript's load-bearing quantities live—running the prover, remeasuring the sampler, inspecting the source—so that a stale artifact cannot reproduce a number (`papers/gpnc/experiments/VERIFICATION.md`).

Table 14: The experiment index. All CPU-only. The eleven experiments of Parts I and II complete in minutes; Part III adds roughly half an hour, dominated by the 1722-mutant sweep of E12 and the like-for-like mutant comparison of E15. The reduced-budget mode completes all nineteen in a few minutes.

Exp.	What it establishes
E1	Exactness at true trip-counts 2–1000 on a 1024-cell memory, with a measured fixed-memory contrast curve
E2	Equal-noise pinning ablation on the real substrate; the Eq. (4) basin boundary
E3	SMT proof of total equivalence; the 12 945-pair separation/witness table; p_{sep}
E4	200 held-out programs, four strata, trace-exact scoring, mutant negative control
E5	Probe-budget curve over 30 seeds; the non-discriminating-distribution confusion table
E6	SHA-256 of the full state trace across 43 conditions
E7	Differential test against WASMTIME; the semantic-gap table; the instruction census
E8	Measured top-k opcode coverage over real x86-64 binaries and CPython bytecode
E9	Memory, trip-count and program-length sweep; the $14n + 10$ cycle law
E10	Differentiable recovery of ISA-15 as a measured distribution over restarts
E11	Addressing operator and token mixing measured standalone; the dense arm's closed form tested
Part III.	
E12	Label-free recovery of ISA-41 over its 3264-tuple microcode space; probe-budget curve over 30 seeds; 200-program suite; 1722-mutant control
E13	The 30 numeric opcodes differentially tested against WASMTIME, traps included; recomputed Wasm-core coverage
E14	Total equivalence at *every* stack depth via the opaque-tail encoding; symbolic-moduli arm; all 133 783 wrong pairs decided; four controls
E15	Eight recognizable algorithms end to end against independent oracles, including a Brainfuck interpreter and the compiled Minsky machine
E16	2^{20} cells and 1.6×10^7 cycles, with the in-place runners certified against the copying semantics first
E17	Recovery with the designed probe distribution deleted: transitions come only from running programs
E18	Live regeneration: 41 checks re-deriving the paper's load-bearing quantities from the artifact, fourteen of which verify the detectors can fire
E19	The decoder network trained by gradient on state pairs alone, over ISA-41, in three supervision arms and two decoder depths

The GPU-queued experiments. Every experiment above is CPU-only, and the determinism certificate accordingly has no accelerator arm—which costs it nothing, since there is nothing on the inference path for an accelerator to make non-deterministic. Four further items require a GPU and are specified as runnable scripts, with estimated GPU-hours, a pre-registered success criterion, and a pre-registered falsifier naming the result each would establish or refute. **No number from any of those scripts appears anywhere in this paper.**

References

[1] A. Giannou, S. Rajput, J. Sohn, K. Lee, J. D. Lee, D. Papailiopoulos. *Looped Transformers as Programmable Computers.* ICML, 2023.

[2] A. Graves, G. Wayne, I. Danihelka. *Neural Turing Machines.* 2014.

[3] A. Graves et al. *Hybrid computing using a neural network with dynamic external memory.* Nature, 2016.

[4] Ł. Kaiser, I. Sutskever. *Neural GPUs Learn Algorithms.* 2015.

[5] S. Reed, N. de Freitas. *Neural Programmer-Interpreters.* ICLR, 2016.

[6] M. Bošnjak, T. Rocktäschel, J. Naradowsky, S. Riedel. *Programming with a Differentiable Forth Interpreter.* ICML, 2017.

[7] G. Weiss, Y. Goldberg, E. Yahav. *Thinking Like Transformers.* ICML, 2021.

[8] D. Lindner et al. *Tracr: Compiled Transformers as a Laboratory for Interpretability.* NeurIPS, 2023.

[9] M. Dehghani et al. *Universal Transformers.* ICLR, 2019.

[10] C. Watt. *Mechanising and Verifying the WebAssembly Specification.* CPP, 2018.

[11] A. Neelakantan, Q. V. Le, I. Sutskever. *Neural Programmer: Inducing Latent Programs with Gradient Descent.* ICLR, 2016. arXiv:1511.04834.

[12] K. Kurach, M. Andrychowicz, I. Sutskever. *Neural Random-Access Machines.* ICLR, 2016. arXiv:1511.06392.

[13] W. Zaremba, I. Sutskever. *Learning to Execute.* 2014. arXiv:1410.4615.

[14] J. Cai, R. Shin, D. Song. *Making Neural Programming Architectures Generalize via Recursion.* ICLR, 2017. arXiv:1704.06611.

[15] J. Pérez, J. Marinković, P. Barceló. *On the Turing Completeness of Modern Neural Network Architectures.* ICLR, 2019. arXiv:1901.03429.

[16] P. Veličković, C. Blundell. *Neural Algorithmic Reasoning.* Patterns, 2021. arXiv:2105.02761.

[17] P. Veličković, A. P. Badia, D. Budden, R. Pascanu, A. Banino, M. Dashevskiy, R. Hadsell, C. Blundell. *The CLRS Algorithmic Reasoning Benchmark.* ICML, 2022. arXiv:2205.15659.

[18] P. Veličković, R. Ying, M. Padovano, R. Hadsell, C. Blundell. *Neural Execution of Graph Algorithms.* ICLR, 2020. arXiv:1910.10593.

[19] B. M. Lake, M. Baroni. *Generalization without Systematicity: On the Compositional Skills of Sequence-to-Sequence Recurrent Networks.* ICML, 2018. arXiv:1711.00350.

[20] G. Delétang, A. Ruoss, J. Grau-Moya, T. Genewein, L. K. Wenliang, E. Catt, C. Cundy, M. Hutter, S. Legg, J. Veness, P. A. Ortega. *Neural Networks and the Chomsky Hierarchy.* ICLR, 2023. arXiv:2207.02098.

[21] C. Anil, Y. Wu, A. Andreassen, A. Lewkowycz, V. Misra, V. Ramasesh, A. Slone, G. Gur-Ari, E. Dyer, B. Neyshabur. *Exploring Length Generalization in Large Language Models.* NeurIPS, 2022. arXiv:2207.04901.

[22] M. Nye et al. *Show Your Work: Scratchpads for Intermediate Computation with Language Models.* 2021. arXiv:2112.00114.

[23] A. Kazemnejad, I. Padhi, K. Natesan Ramamurthy, P. Das, S. Reddy. *The Impact of Positional Encoding on Length Generalization in Transformers.* NeurIPS, 2023. arXiv:2305.19466.

[24] A. Ruoss, G. Delétang, T. Genewein, J. Grau-Moya, R. Csordás, M. Bennani, S. Legg, J. Veness. *Randomized Positional Encodings Boost Length Generalization of Transformers.* ACL, 2023. arXiv:2305.16843.

[25] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkumar. *Fourier Neural Operator for Parametric Partial Differential Equations.* ICLR, 2021. arXiv:2010.08895.

[26] L. Lu, P. Jin, G. Pang, Z. Zhang, G. E. Karniadakis. *Learning Nonlinear Operators via DeepONet.* Nature Machine Intelligence, 2021. arXiv:1910.03193.

[27] N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart, A. Anandkumar. *Neural Operator: Learning Maps Between Function Spaces.* JMLR, 2023. arXiv:2108.08481.

[28] A. Hyvärinen, H. Morioka. *Unsupervised Feature Extraction by Time-Contrastive Learning and Nonlinear ICA.* NeurIPS, 2016. arXiv:1605.06336.

[29] A. Hyvärinen, H. Sasaki, R. E. Turner. *Nonlinear ICA Using Auxiliary Variables and Generalized Contrastive Learning.* AISTATS, 2019. arXiv:1805.08651.

[30] I. Khemakhem, D. P. Kingma, R. P. Monti, A. Hyvärinen. *Variational Autoencoders and Nonlinear ICA: A Unifying Framework.* AISTATS, 2020. arXiv:1907.04809.

[31] K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, J. B. Tenenbaum. *DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning.* PLDI, 2021. arXiv:2006.08381.

[32] F. Chollet. *On the Measure of Intelligence.* 2019. arXiv:1911.01547.

[33] M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967. (Two-counter machines are universal; Chapter 14.)

[34] A. Rossberg (ed.). *WebAssembly Core Specification, W3C Recommendation.* 2019–2022.

[35] Bytecode Alliance. *Wasmtime: a standalone runtime for WebAssembly.* https://wasmtime.dev.

[36] L. de Moura, N. Bjørner. *Z3: An Efficient SMT Solver.* TACAS, 2008.

[37] A. Regehr, A. Reid. *HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems.* ASPLOS, 2004.

[38] P. Godefroid, A. Taly. *Automated Synthesis of Symbolic Instruction Encodings from I/O Samples.* PLDI, 2012.

[39] S. Heule, E. Schkufza, R. Sharma, A. Aiken. *Stratified Synthesis: Automatically Learning the x86-64 Instruction Set.* PLDI, 2016. (Base set given as 51 in §3.2 and Table 1 and as 58 in the conclusion.)

[40] M. Sharif, A. Lanzi, J. Giffin, W. Lee. *Automatic Reverse Engineering of Malware Emulators.* IEEE S&P, 2009.

[41] T. Blazytko, M. Contag, C. Aschermann, T. Holz. *Syntia: Synthesizing the Semantics of Obfuscated Code.* USENIX Security, 2017.

[42] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, G. Roşu. *A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture.* PLDI, 2019.

[43] J. Craaijo, F. Verbeek, B. Ravindran. *libLISA: Instruction Discovery and Analysis on x86-64.* Proc. ACM Program. Lang. (OOPSLA), 2024.