Locality by Representation: Certified Learned Modular Multiplication at 2048 Bits from a 37,767-Parameter Carry-Save Montgomery Automaton

Jonathan Washburn
Recognition Physics Institute
washburn.jonathan@gmail.com

July 25, 2026

Abstract

Learned models of arithmetic characteristically fail to generalize in length: a network that adds or multiplies perfectly at its training width collapses a few digits beyond it. We argue that this failure belongs to the *representation* rather than to arithmetic, and that choosing the representation removes it. Written in a carry-save Montgomery basis, every step of modular multiplication $(a \cdot b) \bmod p$ becomes a translation-invariant local digit map on two redundant bit arrays: two 3:2 compressions, a parity read at position zero, and a right shift. The function a network has to represent then has constant-radius dependencies at every operand width, so width is a property of the input array and not of the learned map.

We build the whole of modular multiplication from four such local cells totalling 37,767 parameters. The tick cell has 32,800 of them: two trained embedding tables over the exact radius-1 local alphabet of the carry-save Montgomery tick, one mapping each of the 2^{14} state windows to two output bits and one mapping each of the 2^4 quotient inputs to two logits, with a call signature matching a convolutional tick. Trained by supervised filling against the closed-form truth table from random initialization with a logit margin of 10, it is enumeration-certified exact on its complete alphabets (16 quotient configurations and 16,384 state windows). Three further cells (a carry-save fold, an MSB-first comparator, and a conditional-subtract prep) close the pipeline, so every bit on the inference path is produced by trained weights. All four shipped cells are enumeration-certified exact over their complete finite input spaces.

The value-level induction that composes the cells (evenness, the bounded representative, the Montgomery congruence, the two-pass master theorem, and fold termination) is machine-checked in a proof assistant and depends on no axioms beyond that assistant's three standard ones. On the SAIR Modular Arithmetic Challenge, an independently specified benchmark with prime moduli to 2048 bits, the pipeline attains accuracy 1.0 on all ten scored tiers, the maximum the ranking metric admits, using roughly 61–71 seconds of inference inside the organizers' own sandbox. Randomizing any single cell breaks the pipeline on at least one of two row sets at every seed tried, and we show why the second row set is necessary: random operands never request the conditional subtraction, so they cannot test the two cells that decide it.

1 Introduction

Modular multiplication $(a \cdot b) \bmod p$ for large primes is the computational core of public-key cryptography, lattice reduction, and number-theoretic search. Classical implementations reduce it to schoolbook multiplication with Barrett or Montgomery reduction, or decompose it by the

Chinese remainder theorem. All of these are exact circuits, and they are exact for any parameter values, because nothing in them is learned.

Learned arithmetic behaves differently, and worse. The characteristic failure is not accuracy at the training width but collapse just beyond it. A model trained to add sixteen-digit numbers is typically hopeless at twenty digits, and the standard remedies are positional: index hints, reversed digits, position coupling, and specialized embeddings [11, 13, 14]. These extend the usable range. They do not make the map width-independent, and at cryptographic widths the gap is four orders of magnitude, not a factor of two.

The thesis of this paper is that length generalization in arithmetic is a question about the representation one writes the arithmetic in, and that for modular multiplication a representation exists in which the question does not arise. The obstruction in the ordinary binary basis is not the multiplication. It is that a single step of the computation has unbounded reach: a carry entering at the bottom of a w-bit accumulator can change the top bit, so the map from one state to the next couples all w positions. Any architecture with a bounded receptive field must therefore approximate a function it cannot represent, and the approximation degrades exactly as w grows.

Two changes of basis remove both non-local dependencies. Montgomery form [1, 2] replaces the reduction decision, which naively requires comparing a w-bit accumulator against p, by a parity read at a single position. Carry-save form [5, 6] holds the accumulator as two equal-weight bit vectors whose sum is the value, so an addition never propagates a carry: the carry is emitted one position up and carried in the second vector. What remains, after both changes, is a digit map whose output at position i depends on the inputs at positions $i-1$, i, and $i+1$, plus two scalars broadcast from position zero. That map is translation-invariant and has radius one at every width.

This is a statement about the function class, so it has consequences a network can inherit. We show that a learned table over that finite local alphabet implements the map exactly and that the cell transfers zero-shot from a training alphabet independent of width to a deployment width of 2120, sustaining exactness over 4,224 sequential applications of itself. We close the pipeline with three further local cells so that no arithmetic outside the network touches the operands, certify each of the four shipped cells by exhaustive enumeration against its truth table, and machine-check the induction that composes them.

Our contributions are four.

1. A carry-save Montgomery formulation of modular multiplication in which each step is a radius-one, translation-invariant digit map, stated and proved as Theorem 1 (Section 4).

2. A 37,767-parameter pipeline of four learned local cells that implements it, with measured zero-shot width transfer to 2120 and measured exactness on 2048-bit primes (Sections 5 and 8).

3. Certification rather than estimation: each of the four shipped learned cells is enumerated against its exact local truth table over its entire finite input space, and the value-level composition is machine-checked in a proof assistant with no axioms beyond that assistant's standard three (Section 7).

4. A benchmark result on an independently specified evaluation: accuracy 1.0 on all ten scored tiers of the SAIR Modular Arithmetic Challenge, the maximum its ranking metric admits, with seeded per-cell random-weight controls and an account of which rows can test which cell (Section 8).

2 Related work

Learned arithmetic and length generalization. Neural GPUs [7] learned binary addition and multiplication as convolutional cellular automata and observed that a translation-invariant local update is what allows training short and running long. Our tick cell is in that lineage, specialized to a representation in which modular multiplication, and not just addition, becomes local: once the map is finite and local, an embedding table over the complete alphabet is a natural exact realization. The representation thesis has concurrent support: Wei argues that the long-range carry dependency of integer multiplication is an artifact of the chosen computational layout, and that a two-dimensional outer-product grid reduces long multiplication to 3×3 local updates that a 321-parameter cellular automaton generalizes $683\times$ in length [3], and derives locality, symmetry, and stability postulates under which a convolutional automaton achieves exact addition from length 16 to 10^6 [4]. Those works treat plain addition and multiplication and stop at measured exactness. We differ on both axes that matter here. The operation is modular multiplication with a per-instance prime modulus, where the reduction decision rather than the carry is the non-local obstruction, and our carry-save Montgomery form localizes it to a single parity read. And exactness is not only measured but certified: every learned cell is enumerated against its truth table, and the composition to $(a \cdot b) \bmod p$ at every width is machine-checked.

Recurrent easy-to-hard extrapolation [8, 9] scales the number of iterations of a learned block; we scale iteration count and operand width together, and reach exactness rather than high accuracy. Transformer studies document the failure we start from, with models that master arithmetic at training length degrading sharply beyond it [10, 11]. The RASP-L account [12] predicts that transformers length-generalize exactly when the target is expressible as a short position-uniform program. Our result is the convolutional analogue, made constructive: we choose a representation that turns the target into a constant-radius position-uniform map, and then a small cell learns it exactly. Grokking work [15, 16] trains transformers on modular arithmetic over small fixed moduli and interprets the learned circuit post hoc; we operate at 2048-bit moduli that vary per instance, and the learned object is a digit-level automaton designed to be auditable rather than discovered to be one. Algorithmic-reasoning benchmarks [17] measure step-supervised algorithm imitation; our supervision is likewise on the transition map, with the gate on closed-loop exactness.

Hardware arithmetic. Carry-save form and 3:2 compression are classical multiplier structure [5, 6], and bit-serial Montgomery multiplication with a redundant accumulator is standard in hardware [1, 2]. The contribution here is not the automaton, which is textbook, but the observation that its locality is exactly the property that makes a learned version width-stable and, being finite and local, certifiable.

3 Problem setting

The SAIR Modular Arithmetic Challenge [18] asks for a learned model that computes $(a \cdot b) \bmod p$ for prime p, evaluated in eleven tiers. Ten scored tiers run prime moduli from a few bits to 2048 bits; a diagnostic tier runs pure multiplication with moduli chosen larger than the square of the widest operand, and is excluded from scoring. Submissions are ranked on the highest tier reaching 90% accuracy, then on mean accuracy across the scored tiers.

The rules constrain what may produce the answer. The emitted digits must be determined by trained parameters, randomizing weights must collapse accuracy, and hand-coded arithmetic over the operands on the inference path is prohibited. The interface enforces part of this structurally: three preprocessing hooks each see only their own argument, so no single point in the submitted code holds a, b, and p simultaneously, and a pipeline-owned decoder is the only code that turns

emitted digits into the answer. Per-argument work is permitted, including operand reduction and constants that depend on the modulus alone. Evaluation runs offline in a container with four CPUs, eight gigabytes of memory, and a total wall-clock budget of 300 seconds for all eleven tiers, with any tier interrupted mid-way scoring zero.

Two consequences shape the design. First, the model must be exact and not merely accurate, since a tier is a hundred independent rows and a per-row failure rate of one percent costs the tier. Second, the cost of a design is the number of sequential steps times the cost of a step, because the pipeline is a scan and the budget is wall-clock.

4 Locality by representation

4.1 Bit-serial Montgomery form

Fix an odd modulus p and a bit count n with $2^n > 4p$. For inputs X, Y the bit-serial Montgomery product processes the bits of X from the least significant upward, maintaining an accumulator t:

$$t \leftarrow \frac{t + x_i Y + q_i p}{2}, \qquad q_i = (t + x_i Y) \bmod 2, \tag{1}$$

where x_i is the i-th bit of X. Two facts make this well posed, and both are proved formally in Section 7. Since p is odd, the choice $q_i = (t + x_i Y) \bmod 2$ makes the numerator even, so the division by two is exact and discards nothing. And the accumulator stays below $p + Y$ throughout the scan, so no comparison against p is required mid-scan.

After n ticks the accumulator satisfies $2^n t \equiv XY \pmod p$, which is the Montgomery congruence: the scan computes $XY 2^{-n} \bmod p$ rather than $XY \bmod p$. A second pass against the constant $R^2 = 2^{2n} \bmod p$ removes the factor, and a single conditional subtraction brings the result from $[0, 2p)$ into $[0, p)$.

The essential point is where the reduction decision went. A direct binary scan would have to compare a w-bit accumulator against p at each step, a global operation. Equation (1) instead reads q_i as the parity of the low position: one bit, at one place, no comparison.

4.2 Carry-save redundancy

Equation (1) still hides a non-local operation, because the two additions $t + x_i Y$ and $+ q_i p$ propagate carries the length of the word. We remove it by never resolving the carries. Represent the accumulator redundantly as a pair of bit vectors (A, B) standing for the value $\mathrm{val}(A, B) = \sum_j (A_j + B_j) 2^j$. A 3:2 compressor maps three equally weighted bit vectors to two while preserving the represented sum:

$$\mathrm{compress}(a, b, d) = \big(a \oplus b \oplus d,\ 2\ \mathrm{maj}(a, b, d)\big), \tag{2}$$

where maj is the bitwise majority. The sum bit stays at position j and the carry bit moves to position $j + 1$, so no information travels further than one position, and val is invariant.

A tick in this representation is: compress $(A, B, x_i Y)$, read the parity at position zero, compress the result with $q_i P$, then shift both vectors down by one. Each of the two compressions is local by (2), the shift is local, and the parity read is a single position broadcast as a scalar.

4.3 The locality theorem

Write shl for the shift that moves a bit from position j to $j+1$, and let A', B' be the accumulator planes after one tick.

Cell	Parameters	Radius	Role
Tick	32,800	1	One carry-save Montgomery step, plus the quotient bit
Fold	3,394	2	Carry-save pair to binary, iterated
Subtract-prep	1,282	0	Forms the two's-complement pair for the final subtract
Compare	291	n/a	MSB-first comparator state machine
Total	37,767		

Table 1: The four learned cells. Radius is the receptive field of the spatial cells in positions; the comparator is a scan over a three-state machine and has no spatial extent. The tick is realized as embedding tables over its radius-1 alphabet.

Theorem 1 (Locality of the carry-save Montgomery tick)**.** *Let p be odd with bit planes P, let Y be the multiplicand planes, let (A, B) be the accumulator planes, and let $x \in \{0, 1\}$ be the current multiplier bit. Define*

$$q \;=\; A_0 \oplus B_0 \oplus (x \wedge Y_0).$$

Then there are fixed Boolean functions f_A, f_B, independent of the width w and of the position i, such that for every i

$$A'_i \;=\; f_A\big(W_i\big), \qquad B'_i \;=\; f_B\big(W_i\big), \qquad W_i = \big(A_{i-1..i+1},\, B_{i-1..i+1},\, Y_{i-1..i+1},\, P_{i-1..i+1},\, q,\, x\big),$$

with out-of-range indices read as zero. That is, one tick is a translation-invariant local map of radius one over a finite input alphabet, together with two scalars broadcast from position zero.

Proof sketch. Both compressions in (2) are position-local with a carry-out of one position, so after the two compressions the value at position i depends only on inputs at positions $i-1$ and i. The subsequent shift by one moves the dependency window to $\{i-1, i, i+1\}$. Invariance in i holds because (2) is applied identically at every position. For the quotient bit: after the first compression the low position holds $A_0 \oplus B_0 \oplus (x \wedge Y_0)$, and its partner bit at position zero is zero because a carry can only be emitted upward, so the parity of the represented value at position zero equals that expression, giving q as a function of exactly four bits. The formal statement and a complete proof are machine-checked as `tickBits_correct` (Section 7). □

Theorem 1 is the entire reason the rest of the paper works. It says that the function a network must represent is a fixed finite object which does not grow with the operand width. Width lives in the length of the arrays, which a scan over the local alphabet handles by construction, not in the map. Two properties follow that we exploit directly: a cell trained on the local alphabet is the right cell at every width, and the map, being finite, can be enumerated.

5 The learned pipeline

The full computation is two Montgomery passes followed by an exit that canonicalizes the redundant result and performs the conditional subtraction. Four learned cells cover it, and nothing else on the path touches operand state. Table 1 summarizes them.

Tick cell (TableTick). Two embedding tables over the exact radius-1 local alphabet of Theorem 1: an `nn.Embedding`(2^{14}, 2) for the state map (14 bits from the radius-1 window of (A, B, Y, P) together with the broadcast scalars q and x) and an `nn.Embedding`(2^4, 2) for the quotient head (the four bits that determine q). The call signature matches a convolutional tick: at each position the window is packed into an index, the table is looked up, and the two output logits are written back as the next carry-save planes; the state is thresholded to bits at the end

of every tick. Parameter count is $2^{14} \cdot 2 + 2^4 \cdot 2 = 32{,}800$. Because the alphabet is complete and finite, exactness of the shipped weights is certifiable by enumeration of those 16,384 + 16 configurations (Section 7).

Fold cell. The learned local compressor $(A, B) \mapsto (A \oplus B,\ \mathrm{shl}(A \wedge B))$, two width-3 convolutions over 32 channels with a smooth activation and a width-1 projection. Iterating it drives the carry plane to zero and leaves the binary representation of the same value, which is how a redundant pair becomes an answer. Its termination is the subject of Theorem 2 below.

Compare cell. A three-state machine (equal, greater, less) scanned from the most significant position down, deciding $q = [\, t \geq p \,]$ for the conditional subtraction. Twelve transitions, learned as a small multilayer perceptron over the state one-hot and the two bits being compared.

Subtract-prep cell. Given the value planes, the two's complement of the modulus, and the compare bit, it forms the carry-save pair whose sum is the conditionally subtracted result. Every convolution here is width-1, so its radius is zero.

5.1 Fold termination

The fold chain is the one place where a fixed iteration count is tempting and wrong. Each fold step moves every carry up exactly one position, so the number of steps required is the length of the longest carry run in the value being canonicalized. After a Montgomery pass that run is short, around two dozen positions, because the accumulator is dense and a carry meets a zero quickly. After the conditional subtraction it can span the entire word: subtracting p from $t = p + k$ for small k means adding the two's complement of p, and the sum is $2^w + k$, so a carry travels from the bottom of the word to the top through an unbroken run of ones. That is precisely the case where the answer is small.

The correct bound is therefore the width, and it is a theorem rather than a measurement.

Theorem 2 (Fold termination)**.** *Let* $\mathrm{foldStep}(A, B) = (A \oplus B,\ 2(A \wedge B))$. *Then* val *is invariant under* foldStep*, the carry word after* k *iterations is divisible by* 2^k*, and consequently if* $\mathrm{val}(A, B) < 2^W$ *then* $\mathrm{foldStep}^{[W]}(A, B)$ *has carry word zero and first component equal to* $\mathrm{val}(A, B)$.

The implementation iterates until the carry plane is empty, capped at the width. This is faster than running the bound, since the common case converges in about two dozen steps, and it returns the same bits: once the carry plane is zero the compressor is at a fixed point, so the remaining iterations would only confirm it. Rows batched together stay independent, because the loop exits when every row has converged and a row's value stops changing at its own convergence.

5.2 Width schedule and cost

For a modulus of L bits the pipeline uses $n = 64\lceil (L + 2)/64 \rceil$ ticks at array width $w = n + 8$, so the invariant $n \geq L + 2$ required by $2^n > 4p$ holds with room to spare. Both quantities are functions of L alone, which is what keeps a row's answer independent of the other rows in its batch. The granularity of 64 wastes at most 63 ticks and keeps the tensors a convenient shape. Moduli requiring more than 2,176 ticks are outside the declared capacity and are declined rather than guessed.

A row therefore costs $2n$ sequential ticks, each a table lookup over w positions, plus three fold chains and a w-step comparator scan. At the top tier, $L = 2048$ gives $n = 2112$ and $w = 2120$: 4,224 sequential applications of a cell whose training alphabet does not depend on width.

Cell	Alphabet enumerated	Configurations	Result
Tick, q	4 bits	16	exact
Tick, state	radius-1 window of (A, B, Y, P) with q, x	16,384	exact on both planes
Fold	width-7 window of (A, B)	16,384	exact on both output planes
Subtract-prep	(V, P^c, q), radius 0	8	exact
Compare	full transition table	12	exact; scan checked on 800 pairs

Table 2: Exhaustive enumeration against the exact local truth tables. Every configuration of every listed alphabet was checked; the verdict in each row is over the complete space, not a sample. All four rows are the shipped cells.

6 Training

Every parameter is trained; none is hand-initialized or solved for.

The tick cell is trained by supervised filling of its two embedding tables against the closed-form truth table of the radius-1 carry-save Montgomery map. Gradient descent from random initialization with cross-entropy plus a hinge requiring a logit margin of 10 reaches exact agreement on every configuration of both alphabets: all 16 quotient inputs and all 16,384 state windows. The target is the Boolean map of Theorem 1, so the training set is the complete local alphabet rather than sampled trajectories, and width never enters the loss. The margin term keeps logits away from the decision boundary under thousands of sequential ticks: the state is thresholded back to bits at every tick, so a logit that merely has the right sign is enough for one step, and a logit comfortably away from zero is what keeps the sign right after four thousand steps.

The three exit cells are trained by per-transition supervision against their own exact local maps. The fold and subtract-prep cells are small convolutions; the comparator is a multilayer perceptron over its twelve transitions. Each is trained until it matches its truth table exactly on the enumerated alphabet reported in Section 7.

7 Certification

Measured exactness on samples bounds a failure rate; it does not establish zero. Two properties of this pipeline let us do better. Each learned map is local over a finite alphabet, so it can be enumerated exhaustively rather than sampled. And the composition of the cells into $(a \cdot b) \bmod p$ is a value-level induction that can be checked mechanically. We carry out both.

7.1 Finite enumeration of the local cells

For a cell of radius r over m input planes, the complete local input alphabet has $2^{m(2r+1)}$ elements, times any broadcast scalars. Where that number is small enough to enumerate, checking every element against the exact truth table is a proof of exactness for that cell at every width and every input, not an estimate. Table 2 reports the enumeration for all four shipped cells.

7.2 Value-level induction, machine-checked

Enumeration certifies one step. The composition of steps into $(a \cdot b) \bmod p$ is a separate obligation, and we discharge it in a proof assistant against a standard mathematics library. The development defines the value-level tick and accumulator and proves the following, all sorry-free:

- `tick_even`: for odd p the pre-shift value $t + (t \bmod 2)p$ is even, so the right shift is an exact division and drops nothing.

	Run 1	Run 2
Mean accuracy, scored tiers 1–10	1.0	1.0
Highest tier at or above 90%	10	10
Scored tiers at 100/100	10 of 10	10 of 10
Deterministic	yes	yes
Inference wall-clock (s, of 300)	71.2	60.6
Load wall-clock (s)	2.1	1.7

Table 3: Two independent runs of the official evaluator in the official sandbox (`modchallenge-sandbox`, 4 CPU, 8g), different random seed each time. Seeds: `a1b2c3d4e5f60718293a4b5c6d7e8f90` (Run 1) and `f0e1d2c3b4a5968778695a4b3c2d1e0f` (Run 2). Accuracy 1.0 and tier 10 are the maxima the ranking metric admits. **MEASURED**

- `acc_lt`: the accumulator remains below $p+Y$ for the whole scan, so no mid-scan comparison is needed.
- `pow_mul_acc_modEq` and `monpro_spec`: after n ticks, $2^n \cdot \mathrm{acc} \equiv XY \pmod{p}$.
- `mulmod_correct`: the two-pass pipeline with $R^2 = 2^{2n} \bmod p$ and a final conditional subtraction returns exactly $(a \cdot b) \bmod p$, under the headroom hypothesis $4p < 2^n$.
- `fold_sum_invariant`, `fold_carry_dvd`, `fold_terminates`: Theorem 2.
- `tickBits_correct`: the carry-save bit-level automaton implements the value-level tick, which is the bridge between the enumerated local maps and the value-level induction.

Each of these depends on no axioms beyond the assistant's standard `propext`, `Classical.choice`, and `Quot.sound`, verified with an axioms print of each theorem. Composed with Table 2, the certified local cells iterated on the schedule proved here compute $(a \cdot b) \bmod p$ exactly at every width and input, with the finite enumerations as the only leaves outside the proof assistant.

8 Results

8.1 Benchmark

We evaluate with the challenge's own evaluator inside the organizers' published sandbox image, a container with four CPUs, eight gigabytes of memory, no network, and a read-only filesystem. Each run generates 1,100 fresh problems from a random seed. Table 3 reports two runs.

The diagnostic tier scores 70 of 100 in every run, and the misses are deliberate. That tier is pure multiplication, so its generator selects a modulus larger than the square of the widest operand, reaching the Mersenne primes $2^{2203} - 1$, $2^{4253} - 1$, and $2^{9689} - 1$ for its top three sublevels. Those lie past the declared capacity of Section 5 and are declined rather than guessed. Covering them would roughly triple total runtime, and the tier is excluded from both ranking keys.

8.2 Width transfer

The tick cell is an embedding table over a width-independent local alphabet and is never trained at any deployment width. Exactness was verified at both edges of every rung of the width schedule from 8 to 2048 bits, and on operand edge cases (zero, one, and $p - 1$ on each side) at 64, 256, 1024, and 2048 bits. At the top rung a row is 4,224 sequential ticks at width 2120, with no drift. **MEASURED**

Randomized	Tier 1 (2–3 bit)	Tier 4 (32 bit)	Tier 7 (256 bit)	Small answers
nothing (trained)	100%	100%	100%	100%
tick cell	49%	0%	0%	0%
fold cell	45–62%	2%	2%	0%
compare cell	49–91%	0–60%	0–69%	40–100%
subtract-prep cell	49%	0%	0%	0%
restored	100%	100%	100%	100%

Table 4: Per-cell random-weight controls, three seeds, ranges shown where the seeds differ. **MEASURED** The small-answer set spans 256, 1024, and 2048 bits. The tier-1 residual is chance: the answer space there is one or two bits wide. Randomizing the tick embedding tables collapses to chance or zero on every row set. A hand-coded solver would be correct for any weights and would read 100% in every cell of the table.

8.3 Which cells the benchmark distribution exercises

Before the controls, one property of the evaluation distribution has to be stated, because it determines what a control can detect.

After the second Montgomery pass the value V_2 satisfies $V_2 < p + R^2$ and $V_2 \equiv ab \pmod{p}$. Hence V_2 is either the reduced answer or the answer plus p, and the second case requires $(ab \bmod p) < R^2$. For random operands the answer is a full-width residue, so that inequality essentially never holds: the conditional subtraction is never requested, and the comparator's decision is uniformly "do not subtract". We measured this directly. Across tiers 1, 4, and 7 the subtraction was requested on none of the rows sampled, while on small answers such as $a = b = 1$ it is requested on every row from 256 bits upward. **MEASURED**

Two consequences follow, and both are properties of the benchmark rather than of the model. The comparator and the subtract-prep cell are load-bearing for correctness in general, and the master theorem needs them, but they are inert on random rows. And a control that samples only random rows cannot distinguish a trained comparator from a randomized one that happens to emit the same constant. We therefore score the controls on two row sets: benchmark tiers, and an edge set of deliberately small answers at widths where $V_2 = \text{answer} + p$.

This is also the regime in which fold depth matters, for the same reason: subtracting p from $p + k$ produces a carry that crosses the entire word, which is why the bound in Theorem 2 has to be the width and not a measured typical depth.

8.4 Weight-perturbation controls

The rules name random-weight collapse as the operational test that capability resides in trained parameters. We re-initialize one cell at a time, leaving the surrounding loop untouched, and re-score both row sets. The randomization is seeded explicitly and reported over three seeds, because the outcome for an inert cell depends on which constant its random network emits (Table 4).

The tick, fold, and subtract-prep cells collapse on every row set at every seed. The comparator behaves as Section 8.3 predicts, and the pattern is worth reading rather than summarizing. On two of the three seeds the randomized comparator emits a constant "subtract", which is wrong on every random row and accidentally right on every small-answer row, giving 0% on tiers 4 and 7 and 100% on the edge set; on the third it is inconsistent and degrades both. The invariant across the table is the one that matters: no randomized cell, at any seed, is correct on both row sets, and the trained pipeline is correct on both. Reporting only random rows, or only one seed, would have made the comparator look either load-bearing or inert depending on the draw.

8.5 What remains exact arithmetic, and where

Stating this precisely is part of the claim. Outside the network, the pipeline computes: operand reduction $a \bmod p$ and $b \bmod p$; the Montgomery constant $R^2 = 2^{2n} \bmod p$; the two's complement $2^w - p$; the parity test for the even prime; the bit length used to select the width rung; and bit packing. Every one of these is a function of a single argument, none depends on both operands, and none computes or can compute the product. All operand-dependent state transitions are produced by trained weights: both Montgomery passes including the quotient read, all three fold chains, the comparator, the subtract-prep, and the $p = 2$ path, which reads a trained fold cell's carry rather than performing a Boolean conjunction. No exclusive-or, conjunction, addition, shift, comparison, or subtraction operator acts on operand state in the forward path, and no predicted tensor becomes an integer mid-computation.

9 Discussion

What the result is evidence for. The width transfer here is not a scaling curve that happens to stay flat. It is the behavior one should expect once Theorem 1 holds: the cell is trained on the same finite map it is later asked to apply, and the only thing that changes with width is how many positions the same local lookup sweeps. Read that way, the interesting quantity is not the extrapolation factor but the absence of any mechanism by which width could matter. The lesson generalizes to the extent that other arithmetic admits a similar change of basis, and the concurrent results on plain multiplication [3, 4] suggest the class is not small.

Cost. Locality buys exactness at the price of depth: the pipeline is $2n$ sequential steps, so it is quadratic in the modulus size overall, and a hand-written Montgomery routine is orders of magnitude faster. The comparison worth drawing is not against a classical circuit but against learned alternatives, where the relevant figure is that a 37,767-parameter model is exact at 2048 bits while much larger transformers are not exact at 20 digits.

Limitations. Two, stated plainly. The declared capacity stops at 2,174-bit moduli, which is a schedule choice rather than a property of the cell, but it is a limit. And the certification covers the cells and the composition, not the floating-point arithmetic of a particular runtime; exactness is preserved by thresholding to bits at every step and by the trained logit margin, which is measured rather than proved.

10 Conclusion

Modular multiplication is not hard for a neural network because it is arithmetic. It is hard in the basis that is usually chosen, where one step of the computation reaches across the whole word, and it stops being hard in a basis where one step does not. In carry-save Montgomery form every step is a translation-invariant local digit map, a learned embedding table over that finite alphabet implements the map exactly, and because the map is finite and local its exactness can be certified by enumeration rather than estimated by sampling. Four such cells, 37,767 parameters in total, all four enumeration-certified, compute $(a \cdot b) \bmod p$ exactly at 2048 bits, and score the maximum the SAIR ranking metric admits.

Artifacts

The submitted model, weights, verification record, and enumeration certificate are public at `https://huggingface.co/jonwashburn/sair-modmul-carrysave`. This PDF is at `https://`

recognitionphysics.org/Paperss/SAIR_CarrySave_Modular_Arithmetic.pdf. The machine-checked value-level induction and the enumeration script are included as ancillary files.

References

[1] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[2] Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[3] Z. Wei. On the mirage of long-range dependency, with an application to integer multiplication. *arXiv:2603.29069*, 2026.

[4] Z. Wei. On the spatiotemporal dynamics of generalization in neural networks. *arXiv:2602.01651*, 2026.

[5] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, 1964.

[6] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000. (3:2 carry-save and redundant addition.)

[7] Ł. Kaiser and I. Sutskever. Neural GPUs learn algorithms. In *International Conference on Learning Representations (ICLR)*, 2016. arXiv:1511.08228.

[8] A. Schwarzschild, E. Borgnia, A. Gupta, F. Huang, U. Vishkin, M. Goldblum, and T. Goldstein. Can you learn an algorithm? Generalizing from easy to hard problems with recurrent networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021. arXiv:2106.04537.

[9] A. Bansal, A. Schwarzschild, E. Borgnia, Z. Emam, F. Huang, M. Goldblum, and T. Goldstein. End-to-end algorithm synthesis with recurrent networks: Extrapolation without overthinking. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. arXiv:2202.05826.

[10] C. Anil, Y. Wu, A. Andreassen, A. Lewkowycz, V. Misra, V. Ramasesh, A. Slone, G. Gur-Ari, E. Dyer, and B. Neyshabur. Exploring length generalization in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. arXiv:2207.04901.

[11] N. Lee, K. Sreenivasan, J. D. Lee, K. Lee, and D. Papailiopoulos. Teaching arithmetic to small transformers. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2307.03381.

[12] H. Zhou, A. Bradley, E. Littwin, N. Razin, O. Saremi, J. Susskind, S. Bengio, and P. Nakkiran. What algorithms can transformers learn? A study in length generalization. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.16028.

[13] H. Cho, J. Cha, P. Awasthi, S. Bhojanapalli, A. Gupta, and C. Yun. Position coupling: Improving length generalization of arithmetic transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2405.20671.

[14] S. McLeish, A. Bansal, A. Stein, N. Jain, J. Kirchenbauer, B. R. Bartoldson, B. Kailkhura, A. Bhatele, J. Geiping, A. Schwarzschild, and T. Goldstein. Transformers can do arithmetic with the right embeddings. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2405.17399.

[15] A. Power, Y. Burda, H. Edwards, I. Babuschkin, and V. Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets. arXiv:2201.02177, 2022.

[16] N. Nanda, L. Chan, T. Lieberum, J. Smith, and J. Steinhardt. Progress measures for grokking via mechanistic interpretability. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2301.05217.

[17] P. Veličković, A. P. Badia, D. Budden, R. Pascanu, A. Banino, M. Dashevskiy, R. Hadsell, and C. Blundell. The CLRS algorithmic reasoning benchmark. In *International Conference on Machine Learning (ICML)*, 2022. arXiv:2205.15659.

[18] SAIR Foundation. Modular Arithmetic Challenge rules and evaluation specification, 2026. https://github.com/SAIRcompetition/modular-arithmetic-challenge