Expressivity Analysis

Temporal Nonlinearity vs Depth

Where Should Nonlinearity Live?

ElmanProofs Contributors

January 2026

All theorems formalized in Lean 4 with Mathlib.
Source: `ElmanProofs/Expressivity/`

Contents

Abstract

Every sequence model must answer a fundamental question: where should nonlinearity live? The answer determines computational limits that no amount of scaling can overcome.

This document develops the theory of *recurrence linearity* and its consequences. We prove that models with linear temporal dynamics—Mamba2, Fast Linear Attention, Gated Delta Networks—are mathematically constrained in ways that models with nonlinear temporal dynamics are not. A D-layer linear-temporal model has composition depth D, regardless of sequence length. An E88-style model with nonlinear recurrence has composition depth $D \times T$, where T is the sequence length. The gap is multiplicative.

The consequences are concrete. Functions like running parity and threshold counting are provably impossible for linear-temporal models at any depth. E88 computes them with a single layer. The separation is not empirical—it is mathematical, verified in Lean 4 with no gaps in the proofs.

We also confront the tension between theory and practice: despite E88's provably greater expressivity, Mamba2 achieves better perplexity on language modeling benchmarks. This apparent contradiction resolves once we distinguish what an architecture *can compute* from what it *learns efficiently*. The theoretical hierarchy concerns ultimate limits; training dynamics determine what happens within those limits.

The document traces a journey from the fundamental question—where should nonlinearity live?—through the mathematical machinery of linear recurrence, the impossibility results, E88's escape via tanh saturation, the circuit complexity perspective, output feedback and emergent tape, and finally to the practical implications. The reader emerges with a complete understanding of the expressivity hierarchy among modern sequence models.

Introduction

Every sequence model faces a choice: where should nonlinearity live?

This is not a hyperparameter. It is a fundamental architectural decision with mathematical consequences. The answer determines which functions a model can compute, which it cannot, and why chain-of-thought reasoning works when it does.

Consider the three dominant approaches. Transformers apply nonlinearity between layers: the sequence flows through attention, then through a feedforward network, then through attention again. Time is handled all at once through the attention mechanism; depth accumulates vertically. State-space models like Mamba2 make a different choice: nonlinearity still flows between layers, but now time flows *linearly* within each layer. The state h_t is a linear function of h_{t-1}. This enables parallel computation through associative scans, but it constrains what can be computed. The third path—the one we will examine most closely—places nonlinearity in time itself. In E88, the state $S_t =$

$\tanh(\alpha S_{t-1} + \delta v_t k_t^\top)$ involves a nonlinear function of the previous state. Every timestep adds a layer of composition depth.

These three choices lead to three different computational classes. Our central result makes this precise.

Theorem 1 (**Composition Depth Gap**). *For a D-layer model processing sequences of length T: Linear temporal dynamics yields composition depth D. Nonlinear temporal dynamics yields composition depth $D \times T$.*

The implications unfold from here. Tasks like running parity—computing $x_1 \oplus x_2 \oplus ... \oplus x_t$ at each position—require T sequential decisions. A D-layer model with linear temporal dynamics provides only D levels of composition. When $T > D$, running parity is not merely difficult; it is impossible by theorem.

This document develops the theory and its consequences. We begin with the mathematical machinery of linear recurrent systems and prove what they cannot compute. We then show how E88's tanh saturation escapes these limitations, creating stable fixed points that act as permanent memory. The complete hierarchy emerges:

$$\text{Linear SSM} \subsetneq \text{TC}^0 \text{ (Transformer)} \subsetneq \text{E88} \subsetneq \text{E23 (UTM)}$$

Linear state-space models fall *below* TC^0 in circuit complexity—they cannot even compute parity. Transformers sit exactly at TC^0: constant depth, unbounded fan-in. E88 exceeds TC^0 because its effective depth grows with sequence length. And E23, with its explicit tape memory, achieves full Turing completeness.

We will also confront the gap between theory and practice. Despite E88's provably greater computational power, Mamba2 often achieves better perplexity on language modeling benchmarks. This is not a contradiction; it reveals the difference between what an architecture *can* compute and what it *learns efficiently*. The theoretical hierarchy tells us about ultimate limits; training dynamics determine what happens within those limits.

The story has a final twist. When a model writes output and reads it back, the output stream becomes an *emergent tape*. Chain-of-thought, scratchpad computation, autoregressive self-conditioning—these are all instances of the same phenomenon. With output feedback, even limited architectures can achieve bounded Turing machine power:

$$\text{E88+Feedback} \equiv \text{Transformer+CoT} \equiv \text{DTIME}(T)$$

Chain-of-thought works because it provides working memory, not because it enables magical reasoning.

All theorems in this document are mechanically verified in Lean 4 with Mathlib. This is mathematical certainty, not argument by plausibility. We invite verification: clone the repository, run `lake build`, and confirm that every proof compiles without gaps.

The journey begins with the question of where nonlinearity should live. By the end, we will understand the hierarchy of sequence models and know, with mathematical precision, what each can and cannot do.

Mathematical Foundations

Before we can prove what linear systems cannot do, we need the language to describe what they *are*. The central concept is *recurrence linearity*: whether the new state depends linearly or nonlinearly on the previous state.

Linear Recurrent Systems

The simplest recurrent model updates its hidden state by a matrix multiplication plus an input term.

Definition 2 (**Linear RNN**)**.** A linear RNN has dynamics $h_t = Ah_{t-1} + Bx_t$ and output $y_t = Ch_t$, where $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{k\times n}$.

This innocuous-looking equation has a powerful consequence. By unrolling the recurrence, we obtain a closed form.

Theorem 3 (**State as Weighted Sum**)**.** *For a linear RNN starting from $h_0 = 0$: $h_T = \sum_{t=0}^{T-1} A^{T-1-t} B x_t$.*
[1]

Proof. Expand the recurrence: $h_1 = Bx_0$, $h_2 = ABx_0 + Bx_1$, and so on. The general term is a weighted sum of past inputs, with weights given by powers of A. ■

This closed form is the key to everything that follows. The state h_T is a *linear combination* of inputs. The output $y_T = Ch_T$ is therefore also linear in the input sequence. No matter how long the sequence, no matter how cleverly we choose A, B, C, the output remains additive and homogeneous: $y(\alpha x + \beta z) = \alpha y(x) + \beta y(z)$.

The Impossibility Results

Linearity is a severe constraint. Consider the threshold function: output 1 if the sum of inputs exceeds some threshold τ, output 0 otherwise. This is discontinuous—it jumps from 0 to 1 at the boundary. Linear functions are continuous. Therefore:

Theorem 4 (**Threshold Impossibility**)**.** *No linear RNN computes thresh_τ: output 1 if $\sum_t x_t > \tau$, else 0.*
[2]

Proof. A linear output has the form $y(x) = g \cdot x$ for some fixed vector g. At inputs above threshold, $y = 1$ regardless of the exact sum. But if $y = 1$ for all x with $\sum x_i > \tau$, and the output is linear, then $g = 0$. This contradicts $y = 1$ for any input. The function cannot be realized. ■

The argument for XOR is different. XOR is not continuous in the right sense—it is a Boolean function. But it fails linearity in a more algebraic way.

Theorem 5 (**XOR Is Not Affine**)**.** *No affine function equals XOR on $\{0, 1\}^2$.*

[1] Lean formalization: `LinearCapacity.lean:72`. See `linear_state_is_sum`.
[2] Lean formalization: `LinearLimitations.lean:107`. See `linear_cannot_threshold`.

[3]

Proof. Suppose $f(x, y) = ax + by + c$ agrees with XOR. Then $f(0, 0) = 0$ gives $c = 0$. $f(0, 1) = 1$ gives $b = 1$. $f(1, 0) = 1$ gives $a = 1$. But then $f(1, 1) = a + b + c = 2 \neq 0 = \mathrm{XOR}(1, 1)$. ■

Depth Does Not Help

A natural response is to add layers. Perhaps a deep stack of linear-temporal layers can overcome the limitation? This hope is unfounded.

Definition 6 (**Multi-Layer Linear-Temporal Model**)**.** A D-layer model with linear temporal dynamics within each layer and nonlinear activations (ReLU, GeLU) between layers.

Theorem 7 (**Depth Cannot Compensate**)**.** *For any $D \geq 1$, a D-layer linear-temporal model cannot compute threshold, parity, or XOR.*

[4]

The intuition is geometric. Each layer aggregates features linearly across time. Stacking adds vertical depth—nonlinearity between layers—but not horizontal depth—nonlinearity through time. These are orthogonal directions. Depth is nonlinearity in the wrong dimension.

The Composition Depth Gap

We can now state the central technical result precisely.

Theorem 8 (**Composition Depth Gap**)**.** *For a D-layer model processing sequences of length T: Linear temporal dynamics: total composition depth D. Nonlinear temporal dynamics: total composition depth $D \times T$.*

[5]

The depth of a computation is the longest chain of nonlinear operations from input to output. In a linear-temporal model, only the interlayer activations are nonlinear—D of them. In a nonlinear-temporal model, each timestep adds nonlinearity *within* a layer, for a total of $D \times T$.

Architecture Classification

Which architectures fall where? The classification follows directly from the recurrence equation.

Proposition 9 (**Classification**)**.** Linear in h*: MinGRU, MinLSTM, Mamba2 SSM. All have the form $h_t = A(x_t)h_{t-1} + b(x_t)$—linear in h, even if A and b depend on the input.*

Nonlinear in h*: E1, E88, standard RNN, LSTM, GRU. All have the form $h_t = \sigma(Wh_{t-1} + Vx_t)$—the previous state passes through a nonlinearity.*

[3] Lean formalization: `LinearLimitations.lean:218`. See `xor_not_affine`.
[4] Lean formalization: `MultiLayerLimitations.lean:231`. See `multilayer_cannot_running_threshold`.
[5] Lean formalization: `RecurrenceLinearity.lean:229`. See `e1_more_depth_than_minGRU`.

Input-dependent gating does not change the classification. In Mamba2, $A(x_t)$ and $B(x_t)$ depend on the current input, but the recurrence $h_t = A(x_t)h_{t-1} + B(x_t)x_t$ is still linear in h. The state at time T is still a weighted sum of inputs.

We have established the mathematical foundation. Linear recurrent systems are constrained to compute linear functions of their input history. No amount of depth escapes this constraint—depth adds nonlinearity between layers, not through time. The next section examines the consequences: which functions are provably impossible for linear-temporal models, and why this matters for practical architectures.

The Linear-Temporal Limitation

Armed with the mathematical machinery, we can now catalogue what linear-temporal models cannot do. The results are not merely theoretical curiosities—they apply directly to Mamba2, Fast Linear Attention, Gated Delta Networks, and every architecture built on linear state-space foundations.

The Architectures in Question

To see why these limitations matter, we must first recognize how widespread linear temporal dynamics are. The modern sub-quadratic revolution rests on a shared insight: if the state evolution is linear in the hidden state, the entire sequence can be processed in parallel via associative scan.

Mamba2's core recurrence is $h_t = Ah_{t-1} + Bx_t$, which unfolds to $h_T = \sum_t A^{T-t}Bx_t$. The matrices A and B may depend on the input at each step, but the state remains a linear function of past states. This is what enables the parallel scan—and what constrains the computation.

Fast Linear Attention computes the output as $\text{Output} = q \cdot \left(\sum_i k_i \otimes v_i\right)$, a linear combination of key-value outer products. Gated Delta Networks update state as $S' = S + (v - Sk)k^\top$, which is linear in S. Despite their different motivations—selective state spaces, linear attention, delta rule learning—they share the same fundamental constraint.

The Threshold Barrier

Running threshold is perhaps the simplest function that linear-temporal models cannot compute. The task: output 1 when the cumulative sum of inputs exceeds a threshold, output 0 otherwise.

[6] Lean formalization: `RecurrenceLinearity.lean:148,171`. See `e1_is_nonlinear_in_h, mamba2_is_linear_in_h`.

Theorem 10 (**Running Threshold**)**.** *No D-layer linear-temporal model computes running threshold. The output of such a model is a continuous function of its inputs; threshold is discontinuous.*

[7]

The proof is almost too simple. A linear combination of inputs is continuous. Threshold has a jump discontinuity. Continuous functions cannot equal discontinuous ones. Adding layers does not help—the composition of linear-temporal layers with nonlinear activations produces a continuous output, not a discontinuous one.

This impossibility extends to any function with a hard decision boundary. Binary classification, exact counting, flip detection—all require the output to jump between discrete values. Linear-temporal models can only approximate such jumps with smooth transitions.

The Parity Barrier

Running parity presents a different obstacle. The task: at each position, output the XOR of all inputs so far. Unlike threshold, parity is not about discontinuity—it is about the algebraic structure of XOR.

Theorem 11 (**Running Parity**)**.** *No linear-temporal model computes $y_t = x_1 \oplus ... \oplus x_t$. Parity violates the affine identity: $f(0,0) + f(1,1) = 0 \neq 2 = f(0,1) + f(1,0)$.*

[8]

The proof invokes the same algebraic argument we saw for XOR on two inputs. Affine functions satisfy $f(a) + f(b) = f(c) + f(d)$ when $a + b = c + d$. Parity does not. Therefore parity is not affine, and linear-temporal outputs are affine.

Parity has a distinguished role in complexity theory. It is the canonical function separating AC^0 from TC^0—adding threshold gates to constant-depth circuits allows parity to be computed. For linear-temporal models, parity is impossible at *any* depth.

The Capability Boundary

We can summarize the landscape in a single table. The separation is stark: functions that seem computationally trivial—parity, threshold, state machine simulation—are provably beyond the reach of linear-temporal architectures.

Task	**Why Impossible**	**D-layer Linear**	**1-layer E88**
Running threshold	Discontinuous	No	Yes
Running parity	Non-affine	No	Yes
FSM simulation	State count	Limited	Full

[7] Lean formalization: `ExactCounting.lean:344`.
[8] Lean formalization: `RunningParity.lean:200`.

The rightmost column anticipates the next section. E88, with a single layer of nonlinear temporal dynamics, computes all three. The contrast is not empirical—it is mathematical. These are theorems, not benchmarks.

When Linear Suffices

The limitations we have catalogued do not doom linear-temporal models. For many practical tasks, linear suffices.

Natural language has a characteristic depth distribution. Most sentences require parsing depth 2–5; complex center-embedded clauses push to 7–10; the extreme tail reaches 20–25. A 32-layer model with $D = 32$ exceeds most natural language requirements. The linear-temporal scan processes the entire sequence in parallel, achieving throughput that sequential nonlinear recurrence cannot match.

The limitation matters when depth is constrained (embedded systems, latency-sensitive applications), when tasks inherently require temporal decisions (counting, parity, state tracking), or when algorithmic reasoning is needed (following explicit procedures, simulating automata). For these problems, no amount of linear-temporal depth suffices. The gap is fundamental.

We have established the boundary. Linear-temporal models—Mamba2, FLA, GDN, and their relatives—are mathematically constrained. Threshold, parity, and state tracking lie beyond their reach, regardless of depth. The constraint is not a bug to be fixed; it is a consequence of the design choice to make time flow linearly.

The question becomes: how does E88 escape? What is it about $S_t = \tanh(\alpha S_{t-1} + \delta v_t k_t^\top)$ that crosses the boundary we have just drawn?

E88: Escaping the Linear Barrier

The answer to the question posed at the end of the previous section lies in two properties: *matrix state* and *tanh saturation*. Together, they give E88 capabilities that linear-temporal models provably lack.

Matrix State

Where Mamba2 maintains a state vector of dimension n, E88 maintains a state *matrix* of dimension $d \times d$.

Definition 12 (**E88 State Update**)**.**

$$S_t := \tanh(\alpha \cdot S_{t-1} + \delta \cdot v_t k_t^\top)$$

where $S \in \mathbb{R}^{d \times d}$ is matrix state, $\alpha \in (0, 2)$ is the retention coefficient, and $v_t k_t^\top$ is the rank-1 outer product update from current input.

The capacity difference is immediate. For $d = 64$, each E88 head stores $64^2 = 4096$ values. Mamba2 stores 64–256 total across its state dimension. An 8-head E88 maintains 32,768 state values versus Mamba2's few hundred—a gap of two orders of magnitude.

But capacity alone does not explain the expressivity gap. A large linear system is still linear. The crucial ingredient is what happens inside the tanh.

Tanh Saturation and the Bifurcation

The tanh function is bounded: its output lies in $(-1, 1)$. As the input grows large, tanh approaches ± 1 and its derivative approaches zero. This *saturation* regime is where E88's power emerges.

Theorem 13 (**Bifurcation at $alpha = 1$**). *Consider the scalar map $f(S) = \tanh(\alpha S)$. For $\alpha \leq 1$, zero is the unique fixed point and it is globally attracting. For $\alpha > 1$, two nonzero fixed points $\pm S^*$ appear, and zero becomes unstable.*

[9]

This bifurcation is the mathematical core of E88's memory capability. When $\alpha > 1$, the system has *bistability*: two stable states that the dynamics can settle into and remain in. A sufficiently strong input can push the state from one basin to the other, where it stays until pushed again.

In contrast, linear systems have no such structure. The recurrence $S_t = \alpha S_{t-1}$ with $|\alpha| < 1$ decays exponentially to zero. With $|\alpha| \geq 1$, the state explodes. There is no stable nonzero memory.

Latching: The Memory Mechanism

The bifurcation theorem tells us that stable nonzero states exist. Latching tells us that E88 can use them.

Theorem 14 (**E88 Latches; Linear Decays**). *In E88: once $|S|$ approaches 1, the state persists indefinitely. The tanh derivative vanishes at saturation, so perturbations have vanishing effect.*

In linear systems: $S_t = \alpha^t S_0 \to 0$ for $|\alpha| < 1$. Without continuous reinforcement, the state decays. There is no permanent memory.

[10]

This is the mechanism behind E88's ability to "remember" a binary fact. Once the state saturates near $+1$ or -1, it stays there. The model has latched onto a decision. In a linear-temporal model, the same decision would gradually fade.

Computing Parity

[9] Lean formalization: `AttentionPersistence.lean:212`.
[10] Lean formalization: `TanhSaturation.lean:360`.

With latching in hand, we can show that E88 computes running parity—the function that linear-temporal models provably cannot compute.

Theorem 15 (**E88 Computes Parity**)**.** *With $\alpha = 1$ and $\delta = 2$: an input of 0 preserves the sign of S (since $\tanh(S)$ has the same sign as S), while an input of 1 flips the sign (since $\tanh(S + 2)$ crosses zero when S is negative). The sign of S encodes the running parity.*

[11]

The construction is elegant. We use the sign of the state—positive or negative—to encode even or odd. Each 0-input preserves the sign. Each 1-input flips it. The state sign is the running XOR.

This is impossible for linear-temporal models. We proved it. Yet a single-layer E88 achieves it with a simple parameter choice.

Computing Soft Threshold

Threshold requires a similar but distinct mechanism.

Theorem 16 (**E88 Computes Soft Threshold**)**.** *Accumulated positive signal drives S toward $+1$. Accumulated negative signal drives S toward -1. The transition between regimes is a soft step function, approaching a hard threshold as the accumulation grows.*

[12]

A linear-temporal model cannot produce a step. E88 can: the tanh saturation creates an increasingly sharp transition as the input accumulates.

Head Independence

E88's multi-head structure provides an additional dimension of capacity.

Theorem 17 (**Parallel State Machines**)**.** *An H-head E88 is equivalent to H independent state machines. Head h's update depends only on its own state $S^{(h)}$ and its own input projection. The heads do not interact within a layer.*

[13]

Each head can track a different binary fact, a different parity, a different threshold. The H-head system has H independent bistable bits plus the full matrix state within each head.

The Separation

We can now state the formal separation between E88 and linear-temporal models.

[11] Lean formalization: `TanhSaturation.lean:720`.
[12] Lean formalization: `TanhSaturation.lean:424`.
[13] Lean formalization: `MultiHeadTemporalIndependence.lean:129`.

Property	**E88**	**Linear-Temporal**
State	Matrix d^2	Vector n
Dynamics	Nonlinear (tanh)	Linear
Fixed points	$0, \pm S^*$	Decay or explosion
Latching	Yes	No
Threshold/Parity	Yes	No
Composition depth	$D \times T$	D

The gap is not a matter of degree. It is a qualitative difference in computational class. Functions computable by a 1-layer E88 are provably impossible for any D-layer linear-temporal model. Both the matrix state capacity and the tanh nonlinearity are required: a large linear system cannot latch, and a small nonlinear system lacks the capacity to track complex state.

We have answered the question: E88 escapes the linear barrier through tanh saturation. The bifurcation at $\alpha = 1$ creates bistable fixed points. Latching exploits this structure for permanent memory. Parity and threshold, impossible for linear-temporal models, become achievable with simple parameter choices.

The next question is: how far does this escape extend? E88 exceeds linear-temporal models. Does it exceed Transformers? Does it reach Turing completeness? The answer requires placing these architectures in the landscape of circuit complexity.

Two Paths Beyond Linear

E88 is not the only architecture to escape linear-temporal limitations. E23 takes a different route—explicit tape memory rather than implicit saturation dynamics. The comparison illuminates what is possible and what is practical.

E23: The Tape-Based Approach

E23 maintains explicit memory slots, accessed through attention-like addressing.

Definition 18 (**E23 Dynamics**). *Read*: $r_t = \sum_{i=1}^{N} \alpha_i h_{\text{tape}}^{(i)}$, an attention-weighted sum over N tape slots.

Update: $h_{\text{work}'} = \tanh(W_h h_{\text{work}} + W_x x_t + r_t + b)$, incorporating the read result.

Write: $h_{\text{tape}}^{((j))'} = \big(1 - \beta_j\big) h_{\text{tape}}^{(j)} + \beta_j h_{\text{work}'}$, selectively updating tape slots.

With hard attention (winner-take-all addressing), this becomes a Turing machine simulator.

Theorem 19 (**E23 is UTM-Complete**). *E23 with N tape slots and hard attention simulates any Turing machine using at most N tape cells.*

The proof constructs the simulation explicitly: each tape slot holds one cell, the attention mechanism implements head movement, and the working state encodes the finite control. The correspondence is exact.

E88: The Saturation-Based Approach

E88 achieves its expressivity without explicit tape. The matrix state and tanh saturation create implicit structure.

The approaches differ in their source of power. E23′s memory is explicit, addressable, and permanent. Writing to tape slot j leaves the content there indefinitely; it never decays. E88′s memory is implicit, distributed across the matrix entries, and dynamically stable. Saturation prevents decay, but the stability is emergent rather than designed-in.

The capacity formulas differ accordingly. E23 with N slots of dimension D has $N \times D$ capacity. E88 with H heads of dimension d has $H \times d^2$ capacity. For typical parameters, E88′s capacity is larger, but it is less flexibly addressable.

The Trade-offs

Why would anyone choose E88 over E23, given that E23 is Turing-complete and E88 is not?

The answer lies in training dynamics. E23′s tape creates memory bandwidth pressure: $N \times D$ reads and writes per step, with discrete addressing decisions at each step. Hard attention is non-differentiable; replacing it with soft attention loses the exactness that made E23 Turing-complete.

E88′s matrix operations, by contrast, are naturally differentiable. The tanh is smooth everywhere. The gradient flows through the recurrence without discontinuities. The operations are also GPU-friendly: dense matrix multiplications achieve high utilization, while E23′s variable addressing patterns create irregular memory access.

E23 is Turing-complete but hard to train. E88 is sub-UTM but trainable. The mechanisms that give E23 theoretical power—hard addressing, explicit tape—are exactly what make it difficult to optimize.

The Expressivity-Trainability Frontier

This trade-off is a recurring theme in neural architecture design. Greater expressivity often comes with harder optimization. The frontier is not empty: E88 finds a point that exceeds linear-temporal models while remaining differentiable.

[14] Lean formalization: `E23_DualMemory.lean:730`.

The choice depends on the task. For problems requiring exact symbolic manipulation —compiler verification, theorem proving, arbitrary algorithm execution—E23′s Turing completeness is necessary. For problems where approximate computation suffices and training efficiency matters—language modeling, retrieval, pattern recognition—E88′s position on the frontier is more favorable.

We have seen two escape routes from linear-temporal limitations. E23 achieves full Turing completeness through explicit tape, at the cost of training difficulty. E88 achieves bounded superiority through implicit saturation, while remaining differentiable. Both exceed what Mamba2 and linear attention can compute. Neither dominates the other across all dimensions.

The next question is: where do these architectures sit in the classical hierarchy of computational complexity? The answer requires connecting neural network expressivity to circuit complexity.

The Computational Hierarchy

The pieces are now in place to state the complete picture. We have linear-temporal models that cannot compute parity. We have E88 that can. We have E23 that simulates Turing machines. How do these relate to each other, and to classical complexity theory?

The Strict Hierarchy

Theorem 20 (**Strict Computational Hierarchy**)**.**

$$\text{Linear RNN} \subsetneq \text{D-layer Linear-Temporal} \subsetneq \text{E88} \subsetneq \text{E23}$$

Each inclusion is proper: there exist functions computable at each level but not the previous one.

[15]

The witnesses are concrete. A depth-$(D+1)$ function separates D-layer linear-temporal from $(D+1)$-layer. Running parity separates any linear-temporal from E88. Arbitrary Turing machine computation separates E88 (bounded) from E23 (unbounded).

Two Barriers, Two Escapes

The linear-temporal models face two fundamental barriers. Understanding them reveals why E88 succeeds where they fail.

Theorem 21 (**Affine Barrier**)**.** *Linear-temporal outputs are affine functions of their inputs. XOR is not affine:* $\text{XOR}(0,0) + \text{XOR}(1,1) = 0 \neq 2 = \text{XOR}(0,1) + \text{XOR}(1,0)$*. Therefore running parity is impossible.*

[15] Lean formalization: `MultiLayerLimitations.lean:365`.

[16]

E88 escapes through sign encoding. The parity is stored in whether S is positive or negative. This is not an affine function of the input—it involves the nonlinear tanh.

Theorem 22 (**Continuity Barrier**)**.** *Linear-temporal outputs are continuous. Threshold is discontinuous. Therefore running threshold is impossible.*

[17]

E88 escapes through saturation. The tanh approaches a step function as the input accumulates. While technically continuous, the transition becomes arbitrarily sharp—a soft threshold that approximates hard threshold arbitrarily well.

E88′s Constructions

The escape is constructive. We can write down the parameters that compute parity and threshold.

Theorem 23 (**E88 Computes Parity**)**.** *With $\alpha = 1$ and $\delta = 2$: input 0 preserves the sign of S, while input 1 flips the sign. The state sign encodes the running parity.*

[18]

The construction: input 0 gives $S' = \tanh(S)$, preserving sign. Input 1 gives $S' = \tanh(S + 2)$. When $S < 0$, this crosses zero to positive; when $S > 0$ and moderate, it grows toward $+1$. The dynamics implement XOR.

Theorem 24 (**E88 Computes Soft Threshold**)**.** *Accumulated positive inputs drive S toward $+1$. Accumulated negative inputs drive S toward -1. The transition steepens as accumulation grows.*

[19]

The construction: each input adds to the running sum inside the tanh. As the sum grows large and positive, $\tanh$ saturates toward $+1$. As it grows large and negative, toward -1. The sign indicates whether the sum exceeds the threshold.

The Complete Capability Table

We summarize what each computational class can and cannot do.

Capability	**Linear RNN**	**D-layer Lin.**	**E88**	**E23**
Running parity	No	No	Yes	Yes
Running threshold	No	No	Yes	Yes
Binary latching	No	No	Yes	Yes
Arbitrary FSM	No	No	Yes	Yes

[16] Lean formalization: `LinearLimitations.lean:218`.
[17] Lean formalization: `ExactCounting.lean:344`.
[18] Lean formalization: `TanhSaturation.lean:720`.
[19] Lean formalization: `TanhSaturation.lean:424`.

UTM simulation	No	No	No	Yes

These are not empirical findings. They are theorems, verified in Lean 4. The "No" entries have proofs of impossibility; the "Yes" entries have explicit constructions.

The hierarchy is now established. Linear-temporal models form a proper subclass of E88, which forms a proper subclass of E23. The separations are witnessed by specific functions: parity separates linear from E88, and unbounded computation separates E88 from E23.

But there is a more familiar landmark in computational complexity: TC^0, the class of constant-depth threshold circuits. Where do our architectures sit relative to this classical boundary? The answer is surprising.

Practical Implications

The theorems we have established are mathematically precise. But mathematics alone does not build systems. When does the hierarchy matter for practice?

Matching Architecture to Task

The fundamental insight translates into a design principle: match the architecture's computational class to the task's requirements.

Definition 25 (**Task Classification**)**.** *Pattern aggregation*: tasks that combine information from multiple positions without sequential dependencies. Examples include language modeling (predicting the next token), retrieval (finding relevant documents), and classification (mapping sequence to label). Linear-temporal models suffice.

Sequential decision tasks: tasks where each decision depends on previous decisions. Examples include counting (tracking a running sum), state tracking (simulating a finite automaton), and parity computation. Temporal nonlinearity is mathematically required.

The classification is not always obvious from the task description. "Summarize this document" seems like pattern aggregation, but if the summary requires tracking which points have been covered, it becomes sequential. "Answer this question" may be aggregation or may require multi-step reasoning. The depth of the required computation determines the architectural need.

When Linear Suffices

For most natural language processing with $D \geq 32$ layers, linear-temporal models suffice. This is not a claim about expressivity—we have proved they are strictly less expressive. It is a claim about the distribution of natural language tasks.

Linguistic complexity follows a heavy-tailed distribution. The vast majority of sentences require parsing depth 2–5. Complex center-embedded constructions push to depth 7–10. The extreme tail—legal documents, nested conditionals, deeply recursive structures—may reach 20–25.

A 32-layer model with linear-temporal dynamics provides 32 levels of composition. This exceeds the natural language distribution almost everywhere. The theoretical gap between linear-temporal and E88 is real but rarely exercised by natural text.

The linear approach also has practical advantages. Parallel scan processes the entire sequence simultaneously, while nonlinear recurrence must proceed step by step. For training at scale, this throughput difference dominates.

When the Gap Matters

The theoretical gap becomes practical under specific conditions.

Depth-constrained deployment: When layers are limited by latency or compute budget, the $D = 32$ assumption fails. A 4-layer model on device has only 4 levels of composition. Tasks requiring 5 or more become impossible for linear-temporal architectures.

Algorithmic reasoning: When the task is explicitly algorithmic—counting objects, tracking state across a narrative, following a procedure—the linear-temporal limitation applies directly. These are exactly the tasks where "running parity" style computations are needed.

State machine simulation: Parsing context-free grammars, simulating finite automata, tracking dialogue state—all require maintaining discrete state that persists and updates. Linear-temporal models approximate but cannot exactly compute such state.

> *Adding layers is curvature in the wrong dimension. Depth adds nonlinearity* between *layers; temporal nonlinearity adds it* through *time. These are orthogonal. A 64-layer Mamba2 still cannot compute running parity. A 1-layer E88 can.*

Separating Benchmarks

If we want to measure the expressivity gap empirically, we need benchmarks designed to exercise it.

Benchmark	**E88**	**Linear-Temporal**
Running parity	$\approx 100\%$	$\approx 50\%$ (random)
Running threshold	Sharp transition	Smooth sigmoid
FSM simulation	Exact match	Degrades with state count

On running parity, the prediction is stark: E88 can achieve near-perfect accuracy with appropriate training, while linear-temporal models cannot exceed chance regardless of training budget. The impossibility is mathematical.

Design Principles

The theory yields practical guidance.

Match architecture to task. Use linear-temporal models for pattern aggregation where throughput matters. Use nonlinear-temporal models for sequential decision tasks where correctness matters.

Accept that depth does not substitute for temporal nonlinearity. If a task requires temporal decisions, adding layers does not help. The solution is architectural, not parametric.

Recognize saturation as a feature. E88's tanh saturation is not numerical instability—it is the mechanism enabling persistent memory. Attempts to "fix" saturation by clipping or normalization may destroy the expressivity advantage.

Consider hardware alignment. E88's matrix operations achieve high GPU utilization. Explicit tape memory (E23) creates irregular access patterns. The choice of escape route affects not just expressivity but efficiency.

The practical implications are clear. For most language tasks at scale, linear-temporal models suffice and train faster. For algorithmic tasks, constrained deployments, and exact computation requirements, the theoretical hierarchy predicts empirical outcomes. The art is recognizing which regime applies.

The next section places these results in the classical framework of circuit complexity, connecting our architectural hierarchy to TC^0 and beyond.

Circuit Complexity and the Inverted Hierarchy

The hierarchy we have established among neural architectures has a classical counterpart in circuit complexity. The correspondence is illuminating—and the conclusion is surprising. The popular ranking of architectures by "power" inverts when we measure computational expressivity.

The Boolean Circuit Hierarchy

Circuit complexity measures computational power by the resources a circuit needs. The classes form a nested hierarchy:

$$\mathrm{NC}^0 \subsetneq \mathrm{AC}^0 \subsetneq \mathrm{TC}^0 \subseteq \mathrm{NC}^1 \subseteq \mathrm{P}$$

NC^0 consists of constant-depth circuits with bounded fan-in gates. AC^0 allows unbounded fan-in AND and OR. TC^0 adds threshold gates (MAJORITY). NC^1 allows logarithmic depth with bounded fan-in. P is polynomial-time computation.

The famous separation $AC^0 \subsetneq TC^0$ comes from parity: PARITY requires super-polynomial size in AC^0 but has polynomial-size TC^0 circuits. Adding threshold gates makes parity

easy. This is exactly the separation we have been studying—now placed in classical context.

Where the Architectures Fall

The question is: where do Transformers, SSMs, and recurrent networks sit in this hierarchy?

Theorem 26 (**Transformers Are TC⁰-Bounded**)**.** *A Transformer with D layers and saturated (hard) attention can be simulated by a TC⁰ circuit of depth $O(D)$.*

[20]

The proof observes that attention computes weighted averages, which are threshold operations, and feedforward layers compute bounded-depth compositions of activations. The depth is constant (in the number of layers, not in the sequence length), so the circuit is TC⁰.

This means Transformers live at TC⁰—exactly at the boundary where threshold gates give power over AC⁰. They can compute parity. But their depth is constant.

Theorem 27 (**Linear SSMs Are Below TC⁰**)**.** *State-space models with nonnegative gates (Mamba, Griffin, RWKV) cannot compute PARITY.*

[21]

The proof uses eigenvalue analysis. Nonnegative matrices have nonnegative dominant eigenvalues. The state evolves monotonically in sign. But parity requires alternating sign with each 1-bit input. The dynamics cannot oscillate in the required way.

Linear SSMs fall *below* TC⁰—they cannot even compute the function that separates AC⁰ from TC⁰.

Theorem 28 (**E88 Exceeds TC⁰**)**.** *E88 with T timesteps has effective depth $D \times T$. For any constant bound C, there exists sequence length T such that $D \times T > C$.*

[22]

E88′s depth grows with sequence length. It is not constant-depth. Therefore it exceeds TC⁰ by definition.

The Inverted Ranking

We can now complete the picture:

Architecture	Circuit Class	PARITY	Depth
Linear SSM	< TC⁰	No	D

[20] Lean formalization: `TC0Bounds.lean:152`.
[21] Lean formalization: `TC0VsUnboundedRNN.lean:152`.
[22] Lean formalization: `TC0VsUnboundedRNN.lean:127`.

Transformer	**TC^0**	Yes	D
E88	> TC^0	Yes	$D \times T$
E23	**RE**	Yes	Unbounded

The popular ranking—"Transformers are more powerful than SSMs, which are more powerful than RNNs"—is based on training efficiency and benchmark performance. For *expressivity*, the ranking inverts completely.

E88, the "old" architecture descended from Elman networks, exceeds Transformers in computational class. Linear SSMs, the "modern" efficient architectures, fall below Transformers. The RNN that looked like a step backward is actually a step forward in computational power.

The naive ranking conflates trainability with expressivity. Transformers train efficiently via parallelism. SSMs train efficiently via parallel scan. But E88, while harder to train, computes functions that neither can compute.

What This Means

The circuit complexity perspective clarifies the stakes. TC^0 is a well-studied computational class with known limitations. Transformers live there. Any function outside TC^0 is forever beyond the reach of Transformers, no matter how large or how well-trained.

E88's escape to beyond TC^0 is not a minor improvement—it crosses a classical complexity barrier. The separation is not empirical; it is the same separation that complexity theorists have studied for decades.

We have placed our architectural hierarchy in classical complexity theory. Linear SSMs fall below TC^0. Transformers sit at TC^0. E88 exceeds TC^0 by achieving depth that grows with sequence length. The popular ranking inverts.

But computational class is not the whole story. What happens when models are allowed to write output and read it back? The answer involves an emergent tape.

Output Feedback and Emergent Tape

The hierarchy we have established assumes fixed internal state. But in practice, models often operate differently: they write output, then read it back as input. This simple change has profound consequences.

The Emergent Tape

When a model's output becomes part of its input, the output stream functions as external memory. The model has created an *emergent tape*.

Consider a Transformer with chain-of-thought prompting. The model generates tokens, which are prepended to the context, which the model reads on the next step. The generated text is not just output—it is working memory. Similarly, an RNN in autoregressive mode reads its own previous outputs. The "scratchpad" techniques in language models exploit exactly this mechanism.

Theorem 29 (**Emergent Tape**)**.** *A model with output feedback and T computation steps has effective memory capacity $O(T)$ bits, achieving $\mathrm{DTIME}(T)$ computational power regardless of fixed state dimension.*

[23]

The proof is constructive: simulate a T-step Turing machine by encoding the tape in the output stream. The model reads the tape, makes a transition, writes the updated tape. T steps suffice for T-bounded computation.

This theorem explains why chain-of-thought works. It is not that writing out reasoning steps clarifies the model's "thinking." It is that the written steps provide memory that the model's fixed state cannot. The improvement is computational, not psychological.

Access Patterns

The emergent tape equalizes computational class but not efficiency. The difference is in how the tape is accessed.

An RNN with feedback has sequential access. To read position p in the output history requires processing p timesteps. Each random access costs $O(T)$.

A Transformer with chain-of-thought has random access. Attention can look at any position in the context with $O(1)$ cost. The same computation that requires k sequential passes for an RNN can be done in one pass by a Transformer.

Both achieve $\mathrm{DTIME}(T)$—bounded Turing machine computation. But for algorithms that require multiple random accesses, the Transformer is polynomially more efficient.

The Extended Hierarchy

With output feedback, we can state the complete memory hierarchy.

Theorem 30 (**Strict Hierarchy with Feedback**)**.**

$$\text{Fixed Mamba2} \subsetneq \text{Fixed E88} \subsetneq \text{E88+Feedback} \equiv \text{Transformer+CoT} \subsetneq \text{E23}$$

[24]

Each separation has a witness.

[23] Lean formalization: `OutputFeedback.lean:282`.
[24] Lean formalization: `OutputFeedback.lean:498`.

Mamba2 < E88: Running parity. We have proved this at length.

E88 < E88+Feedback: Palindrome detection. Recognizing whether the input is a palindrome requires comparing position i with position $T - i$. Fixed state cannot store the entire input. With feedback, the model can write the input and read it backward.

CoT < E23: The halting problem. $\mathrm{DTIME}(T)$ is bounded computation; E23 with unbounded tape achieves all recursive functions. There exist problems solvable by E23 that no bounded-tape model can solve.

Why Chain-of-Thought Works

The emergent tape principle demystifies chain-of-thought reasoning.

> *Chain-of-thought works because it provides working memory, not because it enables magical reasoning ability. The scratchpad is computationally necessary for multi-step algorithms. A model computing A then B then C, where each depends on the previous, needs somewhere to store intermediate results.*

Without CoT, a Transformer has fixed internal state: the residual stream, the attention patterns. Complex multi-step computations overflow this state. The model hallucinates or takes shortcuts.

With CoT, the output provides unbounded (up to context length) working memory. The same model can now execute the algorithm correctly because it has somewhere to put the intermediate values.

This is not about "showing work" for interpretability. It is about computational necessity.

We have seen how output feedback changes the landscape. Models with emergent tape memory achieve bounded Turing machine power, regardless of their base architecture. The differences between architectures collapse at this level—E88+Feedback equals Transformer+CoT equals $\mathrm{DTIME}(T)$.

But what if we want something between fixed state and full feedback? Multi-pass RNNs offer an intermediate point.

Multi-Pass RNNs: A Middle Path

Between fixed state and full output feedback lies an intermediate option: re-processing the input multiple times. Multi-pass RNNs trade computation for improved access patterns, achieving a form of soft random access without generating output.

The Multi-Pass Idea

An RNN with fixed state processes the sequence once, left to right. But suppose we let it process the sequence twice, or three times, or k times. Each pass can carry information forward, building on what previous passes learned.

Theorem 31 (**k-Pass Random Access**)**.** *A k-pass RNN can access position p in a sequence of length T using 3 passes: (1) mark positions on the first pass, (2) locate the target position on the second pass, (3) retrieve the value on the third pass. Therefore k passes provide $\lfloor k/3 \rfloor$ effective random accesses.*

[25]

The construction mimics how humans scan a document. First pass: note where relevant information appears. Second pass: find the specific location needed. Third pass: retrieve the content. Each "random access" costs three sequential passes.

The Multi-Pass Hierarchy

More passes means more access capability, forming a strict hierarchy.

Theorem 32 (**Strict Multi-Pass Hierarchy**)**.**

$$\mathrm{MULTIPASS}(1, T) \subsetneq \mathrm{MULTIPASS}(2, T) \subsetneq \cdots \subsetneq \mathrm{DTIME}(T^2)$$

where $\mathrm{MULTIPASS}(k, T)$ is the class of functions computable by a k-pass RNN on sequences of length T.

[26]

The limit $\mathrm{DTIME}(T^2)$ comes from the observation that $O(T)$ passes, each taking $O(T)$ time, gives $O(T^2)$ total computation—matching quadratic attention.

E88 Multi-Pass: Depth Through Time

When we apply multi-pass to E88 specifically, the temporal nonlinearity compounds across passes. Each pass adds T to the compositional depth, creating a multiplicative advantage over linear-temporal models.

Theorem 33 (**E88 Multi-Pass Depth**)**.** *An E88 RNN with k passes over a sequence of length T achieves compositional depth $k \times T$. Each pass contributes T nested tanh applications, accumulating across passes to create $k \times T$ total depth.*

[27]

This is the fundamental expressivity advantage. A linear-temporal model like Mamba2 collapses temporally at each layer, giving effective depth k regardless of sequence length. E88's tanh nonlinearity prevents this collapse: the state at the end of pass i is a T-fold nested composition of tanh, and pass $i + 1$ applies tanh T more times on top of that.

[25] Lean formalization: `MultiPass.lean:878`.
[26] Lean formalization: `MultiPass.lean:958`.
[27] Lean formalization: `E88MultiPass.lean:149`.

Theorem 34 (**E88 Exceeds Linear-Temporal Multi-Pass**). *For any $k > 0$ and $T > 1$, E88 with k passes has compositional depth $k \times T$, which strictly exceeds the depth k of a linear-temporal k-pass RNN.*

[28]

The gap is multiplicative in sequence length. For typical sequences ($T \approx 1000$), even a single-pass E88 has 1000× the compositional depth of a linear-temporal model's single pass.

Tape Modification and Learned Traversal

Between passes, the RNN can modify working memory—not just carrying state forward, but actively transforming an external tape. This enables iterative refinement algorithms that progressively build solutions.

Theorem 35 (**Tape Modification Operations**). *A multi-pass RNN with tape modification can perform: (1) insertions that grow working memory, (2) deletions that shrink/clean working memory, (3) rewrites that modify intermediate results, (4) content-based head movement for adaptive traversal.*

[29]

The tape serves as external memory. On pass 1, the RNN might mark positions of interest. On pass 2, it inserts computed values at those positions. On pass 3, it reads the augmented tape and deletes temporary markers. Each pass sees the modified tape from the previous pass, enabling progressive computation.

Remark . Learned traversal patterns allow data-dependent access. Rather than always scanning left-to-right, the RNN can learn to move backward when it encounters certain symbols, skip forward to find targets, or bounce between positions. This converts sequential access into *adaptive* sequential access—still not random, but no longer rigid.

With unbounded tape modification across passes, the computational power approaches a Turing machine. Each pass executes one TM step: read current cell, update state, write to tape, move head. The multi-pass architecture thus sits between streaming RNNs (no tape) and full Turing machines (arbitrary computation).

Transformer vs Multi-Pass: The Memory-Parallelism Trade-off

The comparison reveals a fundamental trade-off in sequence processing.

Model	**Access**	**Time Complexity**	**Memory Bandwidth**
Transformer+CoT	Random $O(1)$	$O(T)$ parallel	$O(T^2)$
RNN+Feedback	Sequential	$O(T)$	$O(T)$
k-Pass RNN	Soft random	$O(kT)$	$O(kT)$

[28] Lean formalization: `E88MultiPass.lean:212`.
[29] Lean formalization: `MultiPass.lean:1075`.

A Transformer achieves $O(1)$ random access through attention, at the cost of $O(T^2)$ memory bandwidth (the attention matrix). An RNN with feedback has only sequential access but linear bandwidth. A k-pass RNN interpolates: $\lfloor k/3 \rfloor$ random accesses with $O(kT)$ bandwidth.

Theorem 36 (**E88 vs Transformer Depth Comparison**)**.** *For sequence length T and Transformer depth D, an E88 single-pass has compositional depth T. When $T > D$ (typical: $T > 32$), E88 exceeds Transformer depth. With k passes, E88 achieves depth $k \times T$, which grows linearly with sequence length while Transformer depth remains constant at D.*

[30]

The contrast is sharp. Transformers have *constant depth* regardless of sequence length —this is the defining property of TC^0. E88 has *depth proportional to T*—this exceeds TC^0 for long sequences. Multi-pass E88 with k passes achieves depth $k \times T$, creating a computational class that can grow arbitrarily large.

For problems requiring a bounded number of random accesses, multi-pass RNNs match Transformers computationally while using $O(d)$ memory vs $O(T^2)$. For problems requiring unbounded random accesses, Transformers win through parallelism. For algorithmic tasks requiring deep sequential composition, E88 multi-pass provides depth that no fixed-layer Transformer can match.

RNN k-Pass vs Transformer CoT: A Formal Comparison

Chain-of-thought (CoT) adds computation tokens to Transformers, letting them "think step by step." How does this compare to multi-pass RNNs, which iterate over the input k times?

Theorem 37 (**Main Comparison: Memory vs Depth**)**.** *For RNN with k passes over sequence length T, state dimension d, and Transformer with D layers:*

1. *RNN achieves $T/3$ soft random accesses with $k = T$ passes (vs T full random accesses for Transformer)*
2. *RNN uses $O(k \times T)$ total operations (vs $O(D \times T^2)$ for Transformer with attention)*
3. *RNN uses $O(d)$ memory (vs $O(T^2)$ for Transformer attention matrices)*
4. *RNN has sequential depth $k \times T$ (vs constant depth D for Transformer)*

[31]

The key insight: RNN k-pass trades parallelism for memory efficiency. A Transformer can process all T positions simultaneously (parallelism T), but must store $O(T^2)$ attention weights. An RNN processes one position at a time (parallelism 1), but uses only $O(d)$ state memory.

[30] Lean formalization: `E88MultiPass.lean:238`.
[31] Lean formalization: `MultiPass.lean:2004`.

Observation . Transformer CoT doesn't change the depth class. Adding C CoT tokens increases the effective width to $T + C$, enabling C additional "reasoning steps" in parallel. But the circuit depth remains D—constant in the sequence length. The Transformer is still TC^0, regardless of how many CoT tokens we add.

For E88 specifically, the multi-pass depth advantage is even more pronounced:

Theorem 38 (E88 Multi-Pass Hierarchy). *For $k > 0$ and $T > D$:*

$$\text{Linear-temporal } k\text{-pass} \subsetneq \text{E88 } k\text{-pass}$$

with compositional depth k vs $k \times T$. Furthermore, for $k \times T > D$:

$$\text{TC}^0(\text{Transformer}, \text{depth } D) \subsetneq \text{E88 } k\text{-pass } (\text{depth } k \times T)$$

E88 exceeds both linear-temporal multi-pass (by factor T) and Transformers (when depth $k \times T > D$).

[32]

The Extended Hierarchy

We can now place multi-pass RNNs in the complete picture.

$$\text{Mamba2} \subsetneq \text{E88} \subsetneq \text{E88-MULTIPASS} \subsetneq \text{E88+Feedback} \equiv \text{Transformer+CoT} \subsetneq \text{E23}$$

Multi-pass E88 sits between single-pass E88 and E88 with full output feedback. It exceeds single-pass because multiple passes enable random-access-like capabilities. It falls short of full feedback because the number of passes is fixed at architecture time, not adaptive at runtime.

The practical implications follow from the theory. For tasks where $k < T/D$ passes suffice, multi-pass RNNs are more memory-efficient than Transformers. For tasks requiring full random access, Transformers win. For tasks requiring compositional depth $> D$ (deep sequential reasoning), E88 multi-pass exceeds what any fixed-layer Transformer can compute.

Multi-pass processing offers a practical middle ground. When the number of required random accesses is known and bounded, multi-pass RNNs achieve the necessary computation with less memory overhead than full attention. The hierarchy continues to refine: each architectural choice trades off computation, memory, and capability in different ways.

But all these theoretical results face a sobering empirical reality. Despite E88's provably greater computational power, Mamba2 outperforms it on language modeling benchmarks. The next section confronts this gap between theory and practice.

The Theory-Practice Gap

[32] Lean formalization: `E88MultiPass.lean:435`.

The theorems in this document are airtight. E88 computes functions that linear-temporal models cannot. The hierarchy is strict. Yet when we train these models on language modeling, the empirical ranking inverts the theoretical one. This tension deserves careful examination.

The Empirical Results

In experiments with CMA-ES hyperparameter optimization at 480M parameters, the results surprised us:

Architecture	Best Loss	Best Configuration
Mamba2	1.271	d_state=96, depth=25
FLA-GDN	1.273	depth=17, heads=24
E88	1.407	heads=68, d=16, depth=23
Transformer	1.505	heads=8, depth=13

Mamba2—the architecture we proved cannot compute parity—achieved the best loss. E88—provably more expressive—placed third. The empirical ranking exactly inverts the theoretical hierarchy.

The Ablation That Reveals Everything

We conducted an ablation study: replace E88's tanh with linear recurrence, converting it to a linear-temporal model. If temporal nonlinearity matters for language modeling, this should degrade performance.

It did not. The loss was unchanged within measurement noise.

The theoretical separation—running parity, exact threshold, state machine simulation—does not manifest in the language modeling objective. Whatever natural language requires, it is not the capabilities that distinguish E88 from Mamba2.

Two Types of Efficiency

The resolution lies in distinguishing two notions of efficiency.

Definition 39 (**Sample vs Wall-Clock Efficiency**)**.** *Sample efficiency*: examples needed to learn a function class.

Wall-clock efficiency: forward and backward passes per unit wall-clock time.

E88's sequential recurrence processes one timestep at a time. Mamba2's parallel scan processes the entire sequence simultaneously. On modern hardware, Mamba2 achieves roughly 4× higher throughput—four times as many tokens processed per second.

In fixed wall-clock training time, Mamba2 sees 4× as many examples. If both architectures can learn the target function (language modeling), the one that sees more examples learns better. The training dynamics dominate the expressivity advantage.

When Theory Predicts Practice

The theory-practice gap closes under specific conditions.

When the task requires expressivity: For running parity, Mamba2 cannot converge regardless of training budget. Any loss curve plateaus at random-chance accuracy (50%). E88, given sufficient training, converges to near-perfect accuracy. The impossibility theorem manifests as an unbreakable floor.

When training budget is unlimited: Given infinite time, E88's expressivity advantage should eventually manifest even for tasks that both can approximate. We rarely have infinite time.

Property	**Theory Predicts**	**Empirical Observation**
Running parity	E88 > Mamba2	Mamba2 stuck at 50%
Language modeling	E88 $\geq$ Mamba2	Mamba2 > E88

Interpreting the Gap

> *Expressivity determines what can be computed with unlimited resources. Benchmark performance measures what is learned in fixed time. The gap between them is not a flaw in the theory—it is information about the task.*

The language modeling benchmark apparently does not require the capabilities that separate E88 from Mamba2. This could mean:

Natural language does not require temporal nonlinearity. The distribution of natural text may lie within what linear-temporal models can approximate, even though they cannot exactly compute running parity.

The benchmarks do not measure where it matters. Perplexity averages over all predictions. Rare cases requiring temporal nonlinearity—complex state tracking, deep nesting, multi-step reasoning—may be overwhelmed by common cases requiring only pattern matching.

The theory is about expressivity, not learnability. A function may be computable by an architecture but unreachable by gradient descent from random initialization. Expressivity is necessary but not sufficient for learning.

Lessons from Experiments

Several empirical findings guide practical design.

Many small heads outperform few large heads. For E88, 68 heads with 16-dimensional state outperformed configurations with fewer, larger heads. The diversity of independent state machines matters more than the capacity of each one.

Dense architectures outperform sparse at current scales. Mixture-of-experts and sparse attention show promise at very large scales, but at 480M parameters, dense computation wins.

Hardware alignment matters. State dimensions that are multiples of 8 achieve efficient CUDA execution. A state dimension of 68 may theoretically allow more capacity than 64, but 64 runs faster.

Theoretical power does not equal empirical performance. This is perhaps the central lesson. Expressivity is one factor among many. Trainability, throughput, initialization, and optimization dynamics all contribute to final performance.

The theory-practice gap is not a contradiction. Theory tells us what is possible with unlimited resources. Practice tells us what happens with finite resources on specific tasks. E88′s expressivity advantage is real and provable; Mamba2′s training advantage is real and measurable. Which matters depends on the task.

The next section examines where the expressivity advantage is likely to matter: tasks whose composition depth exceeds what linear-temporal models can provide.

Formal Verification

The theorems in this document are not arguments. They are proofs—mechanically verified in Lean 4, checked by computer, with no gaps.

This distinction matters. Mathematical claims in machine learning often rest on intuition, approximation, or empirical validation. Our claims rest on formal proof. If the Lean type checker accepts the proof, the theorem is true. There is no wiggle room for subtle errors or unstated assumptions.

The Verification Approach

Each theorem corresponds to a Lean definition and proof. The proof must satisfy Lean's type checker, which verifies that every step follows from the axioms and previously proven results. Mathlib provides the mathematical foundations: real analysis, linear algebra, topology.

The proofs are constructive where possible. When we say E88 computes running parity, we provide the parameter values and prove they work. When we say linear-temporal models cannot compute threshold, we provide the mathematical obstruction.

Verification Status

Result	**Source File**
Linear state as weighted sum	`LinearCapacity.lean`

Linear cannot threshold/XOR	`LinearLimitations.lean`
Running parity impossibility	`RunningParity.lean`
Multi-layer limitation	`MultiLayerLimitations.lean`
Tanh saturation and latching	`TanhSaturation.lean`
E88 computes parity	`TanhSaturation.lean`
Exact counting separation	`ExactCounting.lean`
TC^0 circuit bounds	`TC0VsUnboundedRNN.lean`
Output feedback / emergent tape	`OutputFeedback.lean`
Multi-pass RNN hierarchy	`MultiPass.lean`
DFA simulation bounds	`ComputationalClasses.lean`

All core expressivity theorems compile without `sorry` statements. The proofs are complete.

What Verification Guarantees

Formal verification provides certainty about what it checks:

Logical validity: Every proof step is a valid inference.

Type correctness: All mathematical objects have the stated types.

Explicit hypotheses: The assumptions of each theorem are stated precisely.

Completeness: There are no gaps—every lemma invoked is itself proven.

Formal verification does *not* guarantee:

Relevance: The theorems might not matter for practice.

Applicability: Real systems might not match the formalized abstractions.

Optimality: A proven bound might not be tight.

Efficiency: An existence proof does not provide an algorithm.

Our theorems concern idealized mathematical models. Real neural networks have finite precision, training noise, and optimization dynamics. The formalization captures expressivity—what functions the architecture *can* compute—not what it will learn.

How to Verify

Clone the repository: `github.com/ekg/elman-proofs`.

Run `lake build`.

If the build completes without error, every theorem in this document has been checked. The source code is the proof. The compiler is the verifier.

The formal foundation is solid. The theorems are true. The remaining sections explore their implications for practical deployment and the composition depth required by various tasks.

Composition Depth in Practice

The theoretical gap between linear-temporal models and E88 is real. But does it matter for tasks people actually care about? The answer depends on how much composition depth those tasks require.

The Distribution of Depth

Human-generated text follows a heavy-tailed distribution in composition depth. Most text is simple; some is moderately complex; a small fraction requires deep sequential reasoning.

Domain	**Typical Depth**	**Maximum**	$D = 32$ **Sufficient?**
Syntactic parsing	2–3	7	Yes
Semantic composition	3–4	10	Yes
Discourse structure	4–6	15	Yes
Simple programs	2–5	10	Yes
Recursive algorithms	10–30	100	Partially
Interpreters	50–200	unbounded	No
Formal proofs	100+	unbounded	No

For syntax and simple semantics, a 32-layer model provides more than enough depth. For recursive code execution or proof verification, no fixed-depth model suffices.

The Uncanny Valley of Reasoning

This distribution creates a distinctive failure mode. Large language models produce fluent, confident output across all complexity levels. But when the required depth exceeds what the architecture provides, the output degrades—not obviously wrong, just subtly broken.

Definition 40 (**The Depth Barrier**)**.** If a function requires composition depth N and a model has capacity $D < N$, the model cannot compute the function. Failures manifest as: correct for small instances, degraded for medium instances, random for large instances.

The model does not announce that it has exceeded its depth. It confabulates. The output looks like reasoning but misses steps, inverts implications, or hallucinates connections. This is the *uncanny valley*: systems that appear intelligent but fail in ways that surprise us.

The failure modes are predictable from the theory. *State tracking*: after D updates, the model loses track of accumulated state. *Long reasoning chains*: the model skips steps or inverts causality. *Negation blindness*: running parity of negations—a simple counting task—is impossible for linear-temporal models regardless of depth.

Where Depth Matters

> *Natural language follows a heavy-tailed distribution. Most text requires depth 2–5. Occasional complex text requires 10–30. Rare deep reasoning requires 50+. Linear-temporal models perform well on the bulk of the distribution. E88's advantage manifests in the tail—exactly where reasoning fails and chain-of-thought becomes necessary.*

This explains why linear-temporal models achieve good perplexity on language modeling benchmarks: they handle the bulk of the distribution well. It also explains why they fail unexpectedly on reasoning tasks: those tasks live in the tail where the architectural limitation bites.

Practical Guidance

A rough heuristic for architecture selection based on task depth:

Depth ≤ 10: Any modern architecture works. Syntax, simple semantics, pattern matching.

Depth 10–30: A $D = 32$ model handles most cases. Complex discourse, moderately nested code.

Depth 30–100: E88 or chain-of-thought required. Multi-step proofs, recursive algorithm traces.

Depth > 100: External tools required. Compilers, theorem provers, debuggers.

The boundary at $D = 32$ is not magical—it reflects current practice where 32-layer models are common. As models scale, the boundary shifts, but the qualitative picture remains: beyond some depth, fixed-depth linear-temporal models fail.

The composition depth perspective clarifies when the theoretical hierarchy matters. For the bulk of natural language, linear-temporal models suffice. For the tail of complex reasoning, they do not. The appendix provides specific examples across task types.

Appendix: Composition Depth Reference

This appendix provides concrete estimates for the composition depth required by various tasks. Use it to predict where linear-temporal models will succeed and where they will fail.

Depth by Task Type

Task	**Depth**	$D = 32$**?**	**Notes**
Simple word prediction	1–2	Yes	Context matching
Relative clause resolution	4	Yes	Binding dependency
Triple center-embedding	8	Yes	Nested clause parsing
50 negations	50	No	Running parity problem
n-digit addition	n	If $n \leq 32$	Carry propagation
10×10 multiplication	100	No	Full digit-by-digit
5-step deduction	6	Yes	Each step builds on previous
50-step proof	51	No	Sequential dependency chain
`fib(10)` evaluation	18	Yes	Recursive call depth
`fib(50)` evaluation	99	No	Deep recursion tree
20-iteration loop	21	Yes	State update per iteration
100-iteration loop	101	No	Exceeds typical depth
100-char DFA simulation	100	No	State transition per char

The pattern is clear: tasks with sequential dependencies accumulate depth linearly with sequence length. When that length exceeds D, linear-temporal models fail.

Architecture Selection by Domain

Domain	**Linear SSM Sufficient?**	**Recommendation**
Casual conversation	Yes	Mamba2 for throughput
Technical writing	Yes	Mamba2 for throughput
Mathematical proofs	Partially	E88 or chain-of-thought
Simple code completion	Yes	Mamba2 for throughput
Complex algorithm tracing	No	E88 with external tools
Formal verification	No	E88 + proof assistant

The Decision Procedure

Given a task, estimate its composition depth. Compare to the available model depth D.

If depth $\leq D$: any architecture works; choose based on throughput.

If depth $> D$ but bounded: E88 or chain-of-thought can help.

If depth unbounded: external tools are necessary.

The theory does not tell us what a model *will* learn. It tells us what a model *cannot* learn. Use the tables to identify tasks where the theoretical limits bind.

Conclusion

We began with a question: where should nonlinearity live? The answer, we have seen, determines fundamental computational limits.

Transformers place nonlinearity between layers. State-space models like Mamba2 do the same, but process time linearly within each layer. E88 places nonlinearity in time itself, with $S_t = \tanh(\alpha S_{t-1} + \delta v_t k_t^\top)$ compounding across timesteps.

These choices create a strict hierarchy:

$$\text{Linear SSM} \subsetneq \text{TC}^0 \text{ (Transformer)} \subsetneq \text{E88} \subsetneq \text{E23 (UTM)}$$

Linear-temporal models fall below TC^0—they cannot compute even parity. Transformers sit at TC^0, with constant depth. E88 exceeds TC^0, its depth growing with sequence length. E23 achieves full Turing completeness through explicit tape.

The separation is witnessed by concrete functions. Running parity: impossible for any linear-temporal model, achievable by single-layer E88. Threshold: impossible for linear (continuous functions cannot equal discontinuous ones), achievable by E88 via saturation. The proofs are complete, verified in Lean 4, with no gaps.

Yet theory is not practice. Mamba2 outperforms E88 on language modeling despite being provably less expressive. The resolution: expressivity determines what *can* be computed with unlimited resources; benchmarks measure what is learned in fixed time on specific tasks. The theory tells us about limits; training dynamics tell us what happens within those limits.

The practical implications follow from the theory. For tasks whose composition depth is bounded by $D = 32$, linear-temporal models suffice—and train faster. For algorithmic reasoning, state tracking, and any task requiring temporal decisions, the linear-temporal limitation bites. Depth adds nonlinearity in the wrong dimension; only temporal nonlinearity provides depth through time.

Chain-of-thought, the emergent tape, and output feedback all work because they provide working memory—not magical reasoning ability. When a model writes output and reads it back, it creates external storage that overcomes fixed state limitations. This is computation, not cognition.

The reader now understands the hierarchy of sequence models. Linear-temporal architectures are fast and sufficient for most natural language. Nonlinear-temporal architectures are slower but strictly more powerful. The choice depends on the task. For pattern aggregation, linear suffices. For sequential reasoning, nonlinearity is mathematically required.

The question of where nonlinearity should live has an answer: it depends on what you need to compute. And now we know, with mathematical certainty, what each choice can and cannot do.

References

The formal proofs are available in the ElmanProofs repository (`github.com/ekg/elman-proofs`):

- `LinearCapacity.lean` — Linear RNN state as weighted sum of inputs
- `LinearLimitations.lean` — Core impossibility results: threshold, XOR, parity
- `MultiLayerLimitations.lean` — Multi-layer extension of impossibility results
- `TanhSaturation.lean` — Saturation dynamics, bifurcation, latching
- `ExactCounting.lean` — Threshold and counting impossibility/possibility
- `RunningParity.lean` — Parity impossibility and E88 construction
- `E23_DualMemory.lean` — E23 tape-based memory formalization
- `MatrixStateRNN.lean` — E88 matrix state formalization
- `MultiHeadTemporalIndependence.lean` — Head independence theorem
- `E23vsE88Comparison.lean` — Direct comparison of E23 and E88 capabilities
- `AttentionPersistence.lean` — Bifurcation and fixed point analysis
- `OutputFeedback.lean` — Emergent tape memory and CoT equivalence
- `TC0Bounds.lean` — TC^0 circuit complexity bounds for Transformers
- `TC0VsUnboundedRNN.lean` — Hierarchy: Linear SSM < TC^0 < E88
- `ComputationalClasses.lean` — Chomsky hierarchy for RNNs
- `MultiPass.lean` — Multi-pass RNN computational class with tape modification (2000 lines)
- `E88MultiPass.lean` — E88 multi-pass depth hierarchy and random access theorems
- `RecurrenceLinearity.lean` — Architecture classification by recurrence type

Document generated from ElmanProofs Lean 4 formalizations.
All core expressivity theorems mechanically verified.
To verify: clone the repository and run `lake build`.