RECODE: RE-CONDITIONED DECODING FOR GENERATIVE LOGIC SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Generative neural models are a promising approach to logic synthesis, yet on difficult sub-circuits they still lag behind strong classical optimizers such as `&deepsyn`. How best to spend inference-time compute on these models is an open question: existing search, typically Monte Carlo tree search over decoder tokens, runs from a single fixed encoding of the input, and so can only refine the circuits that one representation makes likely. We show that re-encoding a circuit that is functionally identical but structurally different yields different outputs. Re-Conditioned Decoding (ReCoDe) exploits this by iteratively re-encoding a model's own intermediate outputs as new decoder conditioning, with a population parallelizing the search. We evaluate on several thousand IWLS fanout-free windows at $K = 8$, 10 and 12 inputs, the latter two larger than prior neural synthesis has handled. Under exact equivalence, ReCoDe reduces the size of the hardest windows by 16–22% relative to a single decode at every width, outperforms token-level search given ten times the budget, and is the only neural method to surpass `&deepsyn`.

1 INTRODUCTION

Logic synthesis is a core optimization problem in electronic design automation: given a Boolean circuit, the goal is to find a smaller or faster circuit that computes the same function. In this work we focus on size minimization for And-Inverter Graphs (AIGs), a standard representation in which circuits are encoded as directed acyclic graphs of two-input AND gates and complemented edges. Reducing the number of AND nodes is a central proxy for reducing area and downstream implementation cost. Classical synthesis systems such as ABC have long relied on carefully engineered rewriting and resynthesis flows, with scripts such as Resyn2 serving as strong practical optimization heuristics (Brayton & Mishchenko, 2010).

Practical synthesis tools optimize locally, repeatedly replacing a small sub-circuit, or window, with a cheaper implementation of the same function (Brayton & Mishchenko, 2010). Their core rewriting step operates on four-input cuts (Mishchenko et al., 2006), since the number of Boolean functions grows doubly exponentially with the number of inputs, although larger windows expose more optimization potential (Zhu et al., 2023). A natural unit is the fanout-free window, whose internal nodes feed nothing outside it, so it can be replaced without disturbing the rest of the design (Zhu et al., 2023). Synthesis may be exact, preserving the function precisely, or approximate. Approximate logic synthesis (ALS) accepts a bounded error rate in exchange for smaller or faster circuits (Meng et al., 2020; 2023).

Recent neural approaches formulate circuit synthesis as a generative modeling problem: the Circuit Transformer (CT) for equivalence-preserving synthesis (Li et al., 2025), and the Generative Transformer for Approximate Circuits (GTAC) for approximate synthesis under an error constraint (Wang et al., 2025). CT is an encoder–decoder Transformer (Vaswani et al., 2017) trained on pairs of input AIGs and Resyn2-optimized equivalent AIGs. CT serializes an input AIG into a sequence of circuit tokens: primary inputs, inverted primary inputs, AND gates, and NAND gates. It then encodes this sequence and decodes an optimized output AIG token-by-token. A constrained decoding mechanism ensures that generated outputs remain logically equivalent to the input. The strongest published inference setting, CT+MCTS, augments this decoder with Monte Carlo tree search (MCTS) over decoder-token choices. This allows CT to spend additional inference compute exploring alternative token continuations and selecting smaller generated circuits (Li et al., 2025).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

However, CT+MCTS spends its search budget within a single application of the model. The input AIG is encoded once, producing a fixed encoder representation, and all MCTS rollouts are conditioned on that same representation. Increasing the MCTS budget therefore widens search over token continuations, but it does not change the latent state from which decoding begins. This creates a structural limitation: CT+MCTS can explore more outputs reachable from the original encoding, but it cannot condition the decoder on the intermediate optimized circuits it discovers.

We study a different inference-time use of a trained generative model. Instead of treating it as a one-shot generator, we treat it as a learned mutation operator. After the model generates an optimized candidate, we feed that candidate back through the encoder and decode again. Repeating this process lets each decode condition on a fresh encoding of the model's own output (Section 3.1).

This repeated re-encoding allows the model to make larger and more coherent moves through circuit space than token-level search from a fixed representation alone.

We call this Re-Conditioned Decoding (ReCoDe) and instantiate it as a simple population-based genetic algorithm (GA). Starting from an input AIG, the GA generates an initial population of stochastic decodes, selects the smallest candidates as elites, re-encodes those elites, and decodes new children from each one. The population parallelizes the traversal across multiple re-encoding chains, while truncation selection biases the search toward smaller circuits. ReCoDe leaves the trained model, or backbone, unchanged and alters only how it is used at inference time. This lets us ask a precise question: for a fixed trained generative model, is inference compute better spent widening token-level search from one encoded circuit, or traversing a sequence of newly encoded intermediate circuits?

Our contributions are as follows. First, we introduce Re-Conditioned Decoding (ReCoDe), an inference-time search framework for generative models in logic synthesis that is agnostic to both the backbone and the synthesis setting. It re-encodes a model's own intermediate outputs as new decoder conditioning. Second, the resulting sub-circuit optimizer handles 10- and 12-input windows, larger than prior neural synthesis has addressed, and is the only neural method to surpass the strong classical optimizer `&deepsyn`: it reduces the hardest windows by 16–22% relative to a single decode and outperforms token-level search given ten times the budget. Third, we thoroughly analyze the mechanism behind ReCoDe through ablations and a latent-space visualization of where each method searches, and evaluate against classical, approximate and neural baselines on several thousand IWLS fanout-free windows.

2 Related Work

2.1 Neural approaches to circuit generation and logic synthesis

Recent work has begun to formulate circuit synthesis as a neural generation problem rather than solely as a hand-engineered sequence of rewriting passes. Early neural circuit-generation work studied synthesis from formal specifications. Schmitt et al. (2021) trained neural models to synthesize circuits from temporal-logic specification patterns, and Schmitt et al. (2023) studied circuit synthesis with pre-trained language models. Cosler et al. (2023) extended this line of research to iterative circuit repair, repeatedly applying a Transformer to a circuit/specification pair to repair candidates against formal specifications. These works demonstrate that neural sequence models can generate or repair circuits, but their objective is specification satisfaction rather than size minimization of a given combinational AIG under equivalence.

The most directly relevant work is the Circuit Transformer (CT) of Li et al. (2025), an encoder–decoder Transformer trained on pairs of input AIGs and output AIGs optimized by Resyn2, a standard synthesis script from the ABC logic-synthesis and verification framework (Brayton & Mishchenko, 2010). CT uses a constrained decoder to produce circuits that are logically equivalent to the input, and pairs this decoder with MCTS over decoder decisions to further improve circuit size at inference time. In the published CT+MCTS-K setting, the input circuit is encoded once, and the search budget is spent exploring token-level decoder choices conditioned on this fixed encoded representation. GTAC (Wang et al., 2025) applies the same generative formulation to approximate synthesis, with a decoder mask that keeps outputs within an error constraint.

2.2 Latent-space search and iterative refinement

Latent-space search with learned generative models is well established outside logic synthesis. In molecular design, early work trained variational autoencoders (VAEs) (Kingma & Welling, 2013) on molecular string representations and optimized molecular properties in the learned latent space before decoding improved candidates (Gómez-Bombarelli et al., 2018). Subsequent work developed graph-aware latent spaces for molecules, including the Junction Tree VAE of Jin et al. (2018), and combined learned molecular representations with evolutionary or local search procedures (Jensen, 2019; Nigam et al., 2021). These methods demonstrate the general principle that a learned generative model can define a smoother search space over discrete structured objects, allowing optimization to propose larger and more coherent edits than direct mutation in the original representation.

Denoising autoencoders provide an example in which repeated reconstruction steps define a Markov chain over the data space (Bengio et al., 2013). Similarly, Creswell et al. (2017) formulate a Markov chain Monte Carlo (MCMC) sampling process for generative autoencoders that is equivalent to iteratively decoding and encoding samples. Neural proposal kernels have also been studied for learned MCMC samplers (Song et al., 2017). This literature provides useful terminology for viewing a generative backbone as a learned proposal operator over feasible circuits, with truncation selection biasing the population toward smaller AIGs.

2.3 Learned mutation operators in evolutionary search

Evolutionary algorithms traditionally rely on hand-designed mutation and crossover operators (Holland, 1975; Koza, 1992). Recent work has instead begun to use learned generative models as variation operators inside evolutionary loops. FunSearch uses a large language model to propose program mutations that are selected by an external evaluator (Romera-Paredes et al., 2024). Evolution Through Large Models (ELM) uses language models as mutation operators for code in a quality-diversity search loop (Lehman et al., 2023). LMX studies language-model crossover for genetic improvement (Meyerson et al., 2024), and Eureka uses large language models to generate reward functions through an evolutionary search process (Ma et al., 2024). In these systems, the learned model acts as a domain-informed proposal distribution rather than as a standalone predictor.

Our work follows this learned-operator perspective but in a more specialized and formally constrained setting. Instead of using a general-purpose language model to mutate programs or text, we use task-specific encoder–decoders trained to map an input AIG to a smaller feasible one. The operator is therefore already biased toward circuit-size improvements that satisfy the synthesis constraint. Our instantiation of ReCoDe wraps this operator in a population loop: each generation re-encodes the current elite circuits, decodes new feasible candidates, and retains the smallest outputs. This differs from ordinary evolutionary circuit optimization, where mutation and crossover are typically hand-designed, and from CT+MCTS, where inference compute is spent on token-level search from a single encoded input.

Learned evolutionary operators have also been studied outside the LLM setting. For example, Shem-Tov et al. (2025b) survey and introduce deep-learning-based operators for evolutionary algorithms, including neural crossover and Transformer-based mutation, and Shem-Tov et al. (2025a) propose BERT Mutation for genetic programming. In protein design, Hong & Kortemme (2024) use learned protein models to bias mutation inside an evolutionary multiobjective optimizer. These works support the broader idea that learned models can provide more coherent variation operators than random local mutation. Our contribution is to apply this idea to generative logic synthesis, across backbones and synthesis settings, and to analyze the mechanism of repeated re-encoding.

3 Method

3.1 Preliminaries: AIGs, generative backbones, and token-level search

An AIG represents a Boolean circuit as a directed acyclic graph of two-input AND gates with complemented edges. We use #AND as the size objective. A valid output Y must be feasible for the original circuit X_0: in exact synthesis it must be equivalent, $Y \equiv X_0$, and in approximate synthesis its error rate, the fraction of input patterns on which any output differs from X_0, must stay within a budget ϵ.

We consider generative encoder–decoder backbones trained to map an input AIG to a smaller feasible one, such as the CT for exact synthesis (Li et al., 2025) and GTAC for approximate synthesis (Wang et al., 2025). At inference time, the backbone serializes the input circuit, encodes it into a hidden representation $h(X_0)$, and decodes an output AIG token-by-token. A per-token mask restricts decoding to feasible continuations: exactly equivalent ones for CT, and ones within the error budget for GTAC.

Token-level search, such as the CT+MCTS-K inference procedure (Li et al., 2025), spends additional compute on tree search over decoder-token choices. The original AIG is encoded once, and all K MCTS rollouts are conditioned on the same hidden representation $h(X_0)$. Thus CT+MCTS increases search width within one decoding pass:

$$Y \sim p_\theta(\cdot \mid h(X_0)).$$

It does not re-encode the intermediate circuits discovered during search.

Our method changes this inference-time use of the backbone. After producing a feasible candidate Y_1, we feed Y_1 back through the encoder and decode again from $h(Y_1)$. Along a single chain, this induces the sequence

$$X_0 \xrightarrow{\text{encode}} h(X_0) \xrightarrow{\text{decode}} Y_1 \xrightarrow{\text{encode}} h(Y_1) \xrightarrow{\text{decode}} Y_2 \to \cdots$$

ReCoDe runs many such chains in parallel through a population of candidates (illustrated in Figure 5, Appendix B).

3.2 Our instantiation

The framework above establishes that the search moves the conditioning by re-encoding, but leaves open how candidates are proposed and selected, which backbone generates them, and how each decode is sampled. We instantiate each as follows.

Search method. We use a population-based genetic algorithm with truncation selection (Algorithm 1). An initial population of P stochastic decodes of X_0 is formed; each generation keeps the T smallest candidates as elites, re-encodes each one, and decodes C children per elite from its fresh encoding, until B decodes have been spent. Re-encoding is the only structural difference from token-level search: both apply the same trained backbone the same number of times, but CT+MCTS encodes only the original input, whereas ReCoDe encodes TG distinct intermediate circuits over G generations. We use two configurations, ReCoDe-1000 (B=1000, P=200, T=20, C=10) and ReCoDe-100 (B=100, P=20, T=4, C=5), each running G=4 generations after the initial population.

Algorithm 1 ReCoDe algorithm

Require: AIG X_0; backbone encoder–decoder; budget B; population $\mathcal{P}$ of size P, elite set E of size T, C children per elite; sampling strategies $\mathcal{S}$.
Mutate$(x, s) :=$ Backbone.DECODE(Backbone.ENCODE(x); s) ▷ per-token masks keep outputs feasible for X_0
$\mathcal{P}_0 \leftarrow \{\text{MUTATE}(X_0, s_p)\}_{p=1}^{P}$, $b \leftarrow P$ ▷ s_p, s_c cycle through $\mathcal{S}$
while $b + TC \le B$ **do**
  $E \leftarrow$ top-T of $\mathcal{P}$ by #AND
  $\mathcal{P} \leftarrow \{\text{MUTATE}(e, s_c) : e \in E,\ c = 1, \ldots, C\}$; $b \leftarrow b + TC$
end while
return $\arg\min_{y \in \bigcup_g \mathcal{P}_g} \#\text{AND}(y)$

Backbones. For equivalence-preserving synthesis we use the released Circuit Transformer at 8 inputs (Li et al., 2025) and retrain it from scratch at 10 and 12 inputs, where no pretrained model exists. For approximate synthesis we additionally use GTAC's released checkpoint (Wang et al., 2025), a separately trained model with its own circuit serialization.

Sampling strategies. A backbone's decoder is deterministic given its conditioning and prefix, so under greedy decoding an elite's C children would collapse to a single circuit. Each decode therefore draws a strategy from a set $\mathcal{S}$, cycled across batched decoder calls and also used to build the initial population. For equivalence-preserving synthesis, $\mathcal{S}$ comprises sampling at temperature τ=0.8 and τ=1.2, top-k sampling with k=3 at τ=1.0, and greedy decoding. Under an error constraint this set leaves part of the budget unspent, so for both backbones we use higher temperatures: $\tau \in \{1.5, 2, 3\}$ and top-k with k=8 at τ=2.

4 Results

4.1 Experimental setup and baselines

Benchmarks. All circuits are two-output fanout-free windows extracted from the IWLS 2023 contest designs (Mishchenko, 2023; Zhu et al., 2023), and the metric is #AND, which Li et al. (2025) report as average size; lower is better. For equivalence-preserving synthesis we use three widths: $K = 8$ (n = 2,600), and $K = 10$ and $K = 12$ (n = 3,000 each), the latter two beyond the widths prior neural synthesis has addressed. Each benchmark is split into four equal-size difficulty buckets by the input's #AND, fixed before any method runs; at $K = 8$ these are *easy* ($\leq$ 15), *med-low* (16–22), *med-high* (23–28) and *hard* ($\geq$ 29). For approximate synthesis we use 100 of the 8-input windows, 25 per bucket, under an error constraint of 0.1: the error rate, the fraction of the 256 input patterns on which any output differs, must not exceed it on any circuit.

Models. At $K = 10$ and $K = 12$ we train CT from scratch on 15M random circuits, selecting checkpoints on dev splits disjoint from test (1,590 and 1,597 windows respectively) and never on reported numbers.

Baselines. We compare against the classical scripts `resyn2` and `&deepsyn` (Brayton & Mishchenko, 2010); the approximate synthesizers ALSRAC (Meng et al., 2020) and HEDALS (Meng et al., 2023), run with an error constraint of 0 in the equivalence-preserving setting and 0.1 otherwise; and the neural models CT (Li et al., 2025) and GTAC (Wang et al., 2025). For each backbone we report a single greedy decode, and for CT also its published token-level search, CT+MCTS-100. To test whether token-level search simply needs more width, we also run CT+MCTS-1000, ten times the published budget. Each MCTS rollout extends a single shared search tree, so rollouts must run one after another, and ten times the published budget is impractical at benchmark scale. We therefore evaluate CT+MCTS-1000 on a fixed random subset of five circuits per bucket. Finally, CT+MCTS+Re-Encode isolates the mechanism: it re-encodes the best output of CT+MCTS at each iteration but keeps no population, and is approximately matched to CT+ReCoDe-1000 on total decoder paths and unique encoder uses. SAT-based exact synthesis would give an optimality reference, but it times out even on our easiest 8-input windows.

Evaluation. Because windows have at most 12 inputs, we verify every reported circuit exhaustively over all 2^K input patterns, a complete check of equivalence or of the error rate; at $K = 8$ we also confirm equivalence with ABC's combinational equivalence checker. Within a table, every method runs on the same windows. In Figure 1, CT+ReCoDe-1000 results are means over three seeds at every width. Area and delay are measured after mapping to the Nangate 45nm library with ABC.

4.2 Experiment 1: Equivalence-preserving synthesis across window widths

Figure 1 compares all methods under exact equivalence. At $K = 8$, CT+ReCoDe-1000 gives the smallest circuits in every bucket, and its margin over CT+MCTS-100 grows with difficulty, from 0.03 AND on easy windows to 2.15 on hard ones. At the published budget, CT+ReCoDe-100 matches CT+MCTS-100 at $K = 8$, within 0.10 AND in every bucket, while running 5.6$\times$ faster (8 s against 45 s per window) because its decodes batch across the population; at the wider widths it trails on hard windows, where the larger budget is needed. With CT retrained at $K = 10$ and $K = 12$, CT+ReCoDe-1000 is again smallest in every bucket. It is the only neural method below `&deepsyn` in every bucket and on the mean at all three widths (Figure 1D), and relative to a single decode it shrinks hard windows by 22%, 16% and 20% at $K = 8$, 10 and 12. After technology mapping on a

A

$K = 8$ $n = 2{,}600$

Method	Easy	Med-low	Med-high	Hard
`resyn2`	12.73	17.00	22.68	31.84
ALSRAC	12.73	17.01	22.66	31.83
HEDALS	13.22	18.18	24.94	35.83
`&deepsyn`	12.46	16.17	20.80	27.99
CT (single-pass)	12.63	16.93	23.55	35.12
CT+MCTS-100	12.36	15.89	20.52	29.60
CT+ReCoDe-100	12.35	15.91	20.46	29.50
CT+ReCoDe-1000	**12.33**±.00	**15.76**±.01	**19.88**±.01	**27.45**±.08

B

$K = 10$ $n = 2{,}947$

Method	Easy	Med-low	Med-high	Hard
`resyn2`	15.82	16.83	18.16	20.22
ALSRAC	15.86	16.85	18.15	20.18
HEDALS	16.89	17.84	19.49	23.30
`&deepsyn`	15.52	16.46	17.65	19.08
CT (single-pass)	15.77	16.93	18.57	22.44
CT+MCTS-100	15.50	16.31	17.64	19.22
CT+ReCoDe-100	15.50	16.41	17.64	19.71
CT+ReCoDe-1000	**15.41**±.00	**16.23**±.01	**17.41**±.01	**18.75**±.06

C

$K = 12$ $n = 2{,}928$

Method	Easy	Med-low	Med-high	Hard
`resyn2`	18.95	21.17	23.32	26.46
ALSRAC	18.97	21.21	23.28	26.46
HEDALS	20.15	22.65	24.98	29.40
`&deepsyn`	18.66	20.75	22.14	24.27
CT (single-pass)	19.16	21.81	24.15	29.96
CT+MCTS-100	18.58	20.72	22.36	25.09
CT+ReCoDe-100	18.67	20.85	22.45	25.81
CT+ReCoDe-1000	**18.51**±.01	**20.55**±.01	**21.90**±.02	**24.09**±.06

D

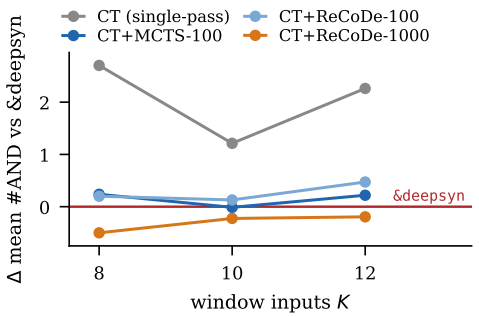


Figure 1: **Equivalence-preserving synthesis across window widths.** Mean #AND per difficulty bucket. Lower is better, and the best method in each bucket is in bold. **A–C**: 8-, 10- and 12-input windows. **D**: difference from `&deepsyn` in mean #AND at each width. Below zero is smaller. CT+ReCoDe-1000, run with three seeds at every width, is shown as mean ± standard deviation over those seeds.

A

Method	Easy	Med-low	Med-high	Hard	Mean	Error
Original (input)	13.44	18.88	25.44	34.04	22.95	0.0000
HEDALS	**3.80**	**2.96**	**5.24**	**5.64**	**4.41**	0.0724
ALSRAC	5.52	5.24	7.56	8.00	6.58	0.0707
CT (single-pass)	12.36	16.60	21.80	27.92	19.67	0.0082
CT+ReCoDe-100	8.36	9.20	11.84	12.36	10.44	0.0623
CT+ReCoDe-1000	5.88	6.40	8.08	8.80	7.29	0.0802
GTAC (single-pass)	9.68	14.52	20.60	19.04	15.96	0.0345
GTAC+ReCoDe-100	7.72	8.80	14.36	13.88	11.19	0.0636
GTAC+ReCoDe-1000	6.44	6.48	10.32	9.44	8.17	0.0791

B

Area (μm^2)	Delay (ps)
15.58	101.13
3.70	**39.14**
5.21	48.97
15.03	91.04
8.58	70.15
6.57	61.48
11.29	77.66
8.34	62.61
6.56	53.98

Figure 2: **Approximate synthesis under an error constraint of 0.1**, on 100 8-input windows. **A**: mean #AND per difficulty bucket, overall mean and mean error rate. **B**: area and delay after technology mapping, rows as in **A**.

200-window subset at $K = 8$, it also has the smallest area of any method, 14.65 against 14.76 μm^2 for `&deepsyn`, which keeps the lowest delay because our objective is size rather than depth (Table 4).

At $K = 10$ and $K = 12$ the backbone sometimes fails to decode: a single CT decode fails to finish a circuit on 21% and 28% of windows, leaving the input unchanged (Table 3). The failure belongs to the decoder rather than to the search, and ReCoDe-1000 reduces it to 1.8% and 2.4%. We exclude, for every method, the remaining 53 and 72 of the 3,000 windows, on which CT+ReCoDe-1000 returns the input unchanged; on all windows (Table 2), CT+ReCoDe-1000 is at parity with `&deepsyn`. At $K = 8$ we report all windows.

Table 1: **Search budget and mechanism** at $K = 8$, on five windows per bucket. **A**: CT+ReCoDe-1000 against token-level search with the same budget of 1000 decodes. **B**: the five hard windows one by one, adding CT+MCTS+Re-Encode, which re-encodes without a population.

A

Bucket	CT+ MCTS-1000	CT+ ReCoDe-1000	Δ
Easy	11.60	11.60	0.00
Med-low	15.80	**15.60**	−0.20
Med-high	20.60	**20.20**	−0.40
Hard	25.40	**24.20**	−1.20

B

Circuit	Input	CT+ MCTS-1000 (N=1, K=1000)	CT+MCTS +Re-Encode (N=80, K=12)	CT+ ReCoDe-1000 (B=1000)
0	33	26	28	26
1	41	28	27	27
2	29	21	**18**	19
3	30	26	26	25
4	35	26	24	24
Mean		25.4	24.6	**24.2**

4.3 Experiment 2: Generality across backbones and synthesis settings

To test whether our method's performance depends on a specific neural backbone or on exact equivalence, we run the same search on GTAC (Wang et al., 2025), a separately trained model with its own serialization, and on CT, both under an error constraint of 0.1 and with the sampling strategies of Section 3.2 (Figure 2). Re-encoding shrinks the mean circuit $2.70\times$ for CT and $1.95\times$ for GTAC relative to a single decode, and area and delay fall with size for both backbones. The dedicated approximate synthesizers remain ahead, HEDALS at 4.41 and ALSRAC at 6.58 mean #AND against 7.29 for CT+ReCoDe-1000. The result therefore shows that the gain transfers across backbones and synthesis settings, not that ReCoDe leads approximate synthesis.

4.4 Experiment 3: Mechanism analysis

Increasing the token-level search budget. If CT+MCTS-100 were under-budgeted, ten times the budget should close the gap. However, on a fixed subset of five windows per bucket it does not. At the same budget of 1000 decodes, CT+ReCoDe-1000 matches CT+MCTS-1000 on easy windows and is smaller in the other three buckets, by 1.20 AND on hard ones (Table 1A). The same compute goes further in ReCoDe than in wider token-level search.

Re-encoding without a population. ReCoDe differs from token-level search in two ways: it re-encodes intermediate circuits, and it keeps a population. CT+MCTS+Re-Encode keeps only the first. It runs MCTS with K=12 rollouts, re-encodes the best output and repeats for N=80 iterations, approximately matching CT+ReCoDe-1000 in decoder paths and unique encoder uses. On the five hard windows it lowers the mean from 25.4 to 24.6, two thirds of the way to CT+ReCoDe-1000 at 24.2, and on one window it finds a smaller circuit than CT+ReCoDe-1000 (Table 1B). On these windows, re-encoding accounts for most of the gain, and the population adds the rest by exploring several re-encoding chains at once.

Latent-space visualization. Figure 3 projects the encodings of every circuit each method visits on two hard windows. The circuits CT+MCTS visits stay within one part of the space, while ReCoDe's children occupy largely distinct regions, and the CT+MCTS+Re-Encode chain, which re-encodes but has no population, lands mostly among them. This matches the ablation: token-level search densifies the region reachable from one encoding, while re-encoding relocates it. Because t-SNE preserves neighbourhoods but not global distances, we read the separation as qualitative. Its quantitative counterpart is that, among exactly equivalent circuits, distance in the encoder's representation correlates with structural distance (Pearson $r = 0.61$ on a hard window), so re-encoding an equivalent circuit changes the conditioning rather than returning to it.

Tracing the winning lineage of CT+ReCoDe-1000 on a hard window shows the compounding directly (Figure 4). Following parent pointers from the final circuit back to the input, each candidate is decoded from the re-encoding of its parent rather than sampled from the input's encoding, and the circuit shrinks from 29 to 18 AND. ReCoDe thus composes several applications of the backbone rather than drawing many one-step samples from one encoding.

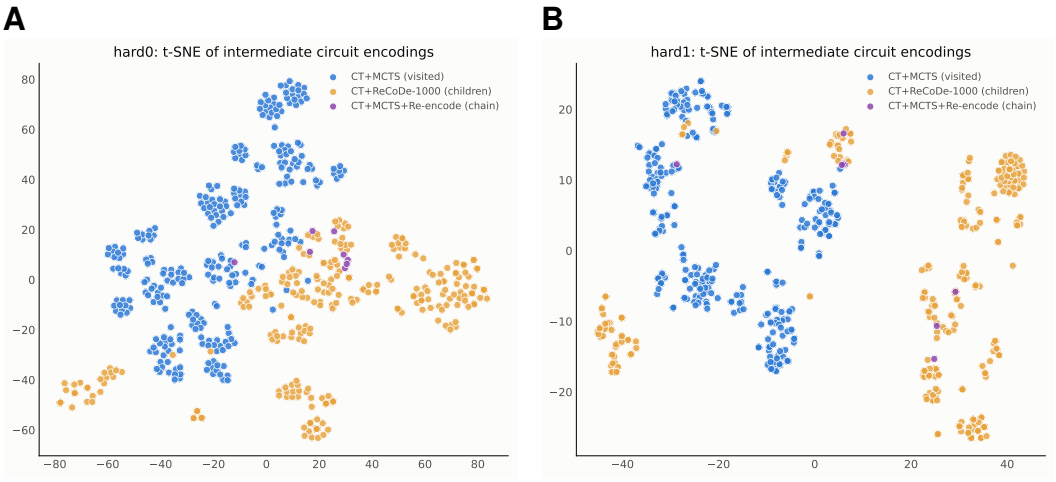


Figure 3: **Latent-space visualization**: t-SNE of the encodings of every circuit visited on two hard windows (**A**, **B**), for CT+MCTS (blue), ReCoDe (orange; labelled CT+ReCoDe-1000 in the legend) and CT+MCTS+Re-Encode (purple).

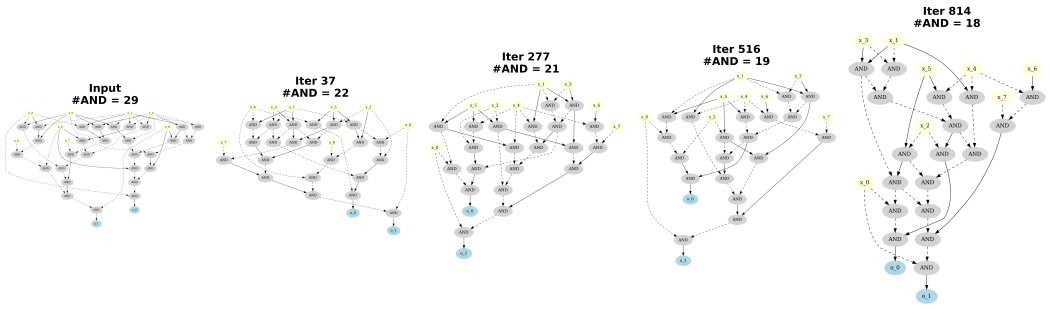


Figure 4: **Lineage of the best CT+ReCoDe-1000 circuit** on a hard window, from the input (29 AND) to the result (18 AND). Gray nodes are AND gates, yellow primary inputs and blue outputs; dashed edges are inverted.

5 Discussion

Our results suggest that, for generative synthesis models, the encoder conditioning is itself a useful search dimension. A backbone trained to map an input AIG to a smaller feasible one learns a distribution over improvements conditioned on the input's structure. Token-level search samples that distribution more thoroughly, but every rollout stays anchored to $h(X_0)$. Re-encoding an intermediate output instead yields a new $h(Y_i)$ from which the decoder proposes further transformations, so improvements compound across steps rather than all having to be found in one decoding pass. The mechanism needs only an encoder that accepts the model's own outputs, and the gain appears with two separately trained backbones, under exact equivalence and under an error constraint.

Limitations. As an inference-time search, ReCoDe is slower than a single decode, although batching across the population makes it faster than tree search at the same number of decodes. The neural backbones also sometimes fail to decode. At $K = 10$ and $K = 12$ we exclude the windows on which CT+ReCoDe-1000 fails (Section 4.2), and on all windows it is at parity with `&deepsyn` rather than ahead. The budget comparison and the ablation use five windows per bucket, because tree search at ten times the published budget is too slow for the full benchmark.

Future work. The search cost could be reduced by repeatedly fine-tuning the backbone on the candidates that ReCoDe produces, so that a single decode recovers part of the gain. Backbones could

also be trained for repeated re-encoding, and learned steps could alternate with classical rewriting passes inside a full flow that extracts, optimizes and stitches back windows.

6 Conclusion

We introduced Re-Conditioned Decoding (ReCoDe), an inference-time search framework that treats a generative model's conditioning representation as a search dimension. Rather than widening token-level search from a single fixed encoding, ReCoDe re-encodes the model's own intermediate outputs as new decoder conditioning, so each step can reach circuits that the previous encoding made unlikely, with a population running several such chains at once. On several thousand IWLS fanout-free windows at 8, 10 and 12 inputs, it reduces the size of the hardest windows by 16–22% relative to a single decode, outperforms token-level search given ten times the budget, and is the only neural method to surpass `&deepsyn`. The results transfer to a second backbone and to approximate synthesis, where re-encoding reduces circuit size by 1.95–$2.70\times$ over a single decode. In our ablations, re-encoding accounts for most of the improvement and the population for the rest. Our results suggest that for generative synthesis models used for logic synthesis, inference compute may be better spent changing the conditioning rather than searching over token space.

AI use statement

We used large language models (LLMs) to help draft and polish the writing, find related work, support research ideation and execution, and generate synthetic datasets. All AI-assisted work was reviewed and verified by the authors, who take responsibility for the final content of the paper.

Reproducibility statement

Section 4.1 describes the benchmarks, difficulty buckets, models, baselines and evaluation protocol, and Section 3.2 and Algorithm 1 give the search configurations and sampling strategies. Every reported circuit is verified exhaustively over all input patterns. We will release our code, the retrained 10- and 12-input Circuit Transformer checkpoints, and the benchmark windows upon acceptance.

References

Yoshua Bengio, Li Yao, Guillaume Alain, and Pascal Vincent. Generalized denoising auto-encoders as generative models. In *Advances in Neural Information Processing Systems*, volume 26, 2013.

Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *Computer Aided Verification*, pp. 24–40. Springer, 2010.

Matthias Cosler, Frederik Schmitt, Christopher Hahn, and Bernd Finkbeiner. Iterative circuit repair against formal specifications. In *International Conference on Learning Representations*, 2023.

Antonia Creswell, Kai Arulkumaran, and Anil Anthony Bharath. Improving sampling from generative autoencoders with markov chains. *arXiv preprint arXiv:1610.09296*, 2017.

Rafael Gómez-Bombarelli, Jennifer N. Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D. Hirzel, Ryan P. Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS Central Science*, 4(2):268–276, 2018.

John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.

Lu Hong and Tanja Kortemme. An integrative approach to protein sequence design through multiobjective optimization. *PLOS Computational Biology*, 20(7):e1011953, 2024. doi: 10.1371/journal.pcbi.1011953.

Jan H. Jensen. A graph-based genetic algorithm and generative model/monte carlo tree search for the exploration of chemical space. *Chemical Science*, 10(12):3567–3572, 2019.

Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *International Conference on Machine Learning*, pp. 2323–2332, 2018.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution through large models. In Wolfgang Banzhaf, Penousal Machado, and Mengjie Zhang (eds.), *Handbook of Evolutionary Machine Learning*, Genetic and Evolutionary Computation, pp. 331–366. Springer Nature, Singapore, 2023. doi: 10.1007/978-981-99-3814-8_11.

Xihan Li, Xing Li, Lei Chen, Xing Zhang, Mingxuan Yuan, and Jun Wang. Circuit transformer: A transformer that preserves logical equivalence. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=kpnW12Lm9p.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representations*, 2024.

Chang Meng, Weikang Qian, and Alan Mishchenko. Alsrac: Approximate logic synthesis by resubstitution with approximate care set. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE, 2020.

Chang Meng, Zhuangzhuang Zhou, Yue Yao, Shuyang Huang, Yuhang Chen, and Weikang Qian. Hedals: Highly efficient delay-driven approximate logic synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(11):3491–3504, 2023.

Elliot Meyerson, Mark J. Nelson, Herbie Bradley, Adam Gaier, Arash Moradi, Amy K. Hoover, and Joel Lehman. Language model crossover: Variation through few-shot prompting. *ACM Transactions on Evolutionary Learning and Optimization*, 4(4):1–40, 2024. doi: 10.1145/3694791.

Alan Mishchenko. Iwls 2023 programming contest. https://github.com/alanminko/iwls2023-ls-contest, 2023.

Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In *Proceedings of the 43rd annual Design Automation Conference*, pp. 532–535, 2006.

AkshatKumar Nigam, Robert Pollice, Mario Krenn, Gabriel dos Passos Gomes, and Alán Aspuru-Guzik. Beyond generative models: Superfast traversal, optimization, novelty, exploration and discovery (STONED) algorithm for molecules using SELFIES. *Chemical Science*, 12(20):7079–7090, 2021.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Frederik Schmitt, Christopher Hahn, Markus N Rabe, and Bernd Finkbeiner. Neural circuit synthesis from specification patterns, 2021.

Frederik Schmitt, Matthias Cosler, and Bernd Finkbeiner. Neural circuit synthesis with pre-trained language models. In *Workshop on Design Automation for CPS and IoT*, 2023.

Eliad Shem-Tov, Moshe Sipper, and Achiya Elyasaf. BERT mutation: Deep transformer model for masked uniform mutation in genetic programming. *Mathematics*, 13(5):779, 2025a. doi: 10.3390/math13050779.

Eliad Shem-Tov, Moshe Sipper, and Achiya Elyasaf. Deep learning-based operators for evolutionary algorithms. In Stephan M. Winkler, Wolfgang Banzhaf, Ting Hu, and Alexander Lalejini (eds.), *Genetic Programming Theory and Practice XXI*, Genetic and Evolutionary Computation, pp. 51–66. Springer, 2025b. doi: 10.1007/978-981-96-0077-9_3.

Jiaming Song, Shengjia Zhao, and Stefano Ermon. A-NICE-MC: Adversarial training for MCMC. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Jingxin Wang, Shitong Guo, Wenhui Liang, Ruicheng Dai, Ruogu Ding, Xin Ning, and Weikang Qian. Gtac: A generative transformer for approximate circuits. *arXiv preprint arXiv:2601.19906*, 2025.

Xuliang Zhu, Ruofei Tang, Lei Chen, Xing Li, Xin Huang, Mingxuan Yuan, Weihua Sheng, and Jianliang Xu. A database dependent framework for k-input maximum fanout-free window rewriting. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE, 2023.

A Additional results

Table 2 repeats Figure 1B and C on all 3,000 windows. Tables 3 and 4 give decode-failure rates, and size, area and delay after technology mapping.

Table 2: Mean #AND on all 3,000 windows at $K = 10$ and $K = 12$, counting each window on which the search fails at its input size. Lower is better; best per column in bold.

	$K = 10$					$K = 12$				
Method	Easy	Med-low	Med-high	Hard	All	Easy	Med-low	Med-high	Hard	All
`resyn2`	15.83	16.83	18.15	20.12	17.73	18.94	21.16	23.32	26.72	22.54
ALSRAC	15.87	16.85	18.14	20.03	17.72	18.96	21.20	23.29	26.72	22.54
HEDALS	16.89	17.84	19.49	23.57	19.45	20.16	22.65	24.99	29.91	24.43
`&deepsyn`	15.52	16.46	17.64	**18.95**	17.14	18.65	20.75	22.14	**24.37**	**21.48**
CT (single-pass)	15.77	16.93	18.57	22.83	18.52	19.17	21.82	24.18	30.53	23.93
CT+MCTS-100	15.50	16.32	17.64	19.69	17.29	18.57	20.73	22.41	25.95	21.92
CT+ReCoDe-100	15.50	16.41	17.65	20.28	17.46	18.69	20.87	22.49	26.65	22.17
CT+ReCoDe-1000	**15.41**	**16.24**	**17.42**	19.32	**17.10**	**18.54**	**20.55**	**21.94**	25.01	21.51

Table 3: Fraction of windows on which the decoder exhausts its step budget without closing all outputs, so that the search returns its input unchanged ($n = 3{,}000$ per width).

	$K = 10$				$K = 12$			
Bucket	CT (single)	CT+ MCTS-100	CT+ ReCoDe-100	CT+ ReCoDe-1000	CT (single)	CT+ MCTS-100	CT+ ReCoDe-100	CT+ ReCoDe-1000
Easy	11.2%	0.8%	1.9%	0.1%	8.8%	1.1%	2.1%	0.8%
Med-low	14.0%	1.2%	2.9%	0.1%	19.2%	1.7%	6.4%	0.7%
Med-high	20.1%	0.8%	3.1%	0.3%	32.7%	4.8%	8.8%	1.3%
Hard	38.0%	6.3%	14.0%	6.5%	50.9%	11.7%	19.7%	6.8%
All	20.8%	2.3%	5.5%	1.8%	27.9%	4.8%	9.3%	2.4%

Table 4: Size, area and delay under exact equivalence on 200 8-input windows, after mapping to Nangate 45nm with ABC. Lower is better; best per column in bold.

Method	Size (#AND)	Area (μm^2)	Delay (ps)
Original (input)	22.91	15.69	101.30
`resyn2`	20.70	15.30	90.71
ALSRAC	20.72	15.29	92.69
HEDALS	22.91	15.69	101.30
`&deepsyn`	19.18	14.76	**87.46**
GTAC (single-pass)	23.02	15.75	101.50
CT (single-pass)	21.55	15.93	95.80
CT+MCTS-100	19.16	14.94	91.30
CT+ReCoDe-100	19.16	14.91	91.35
CT+ReCoDe-1000	**18.43**	**14.65**	91.69

B ADDITIONAL FIGURES

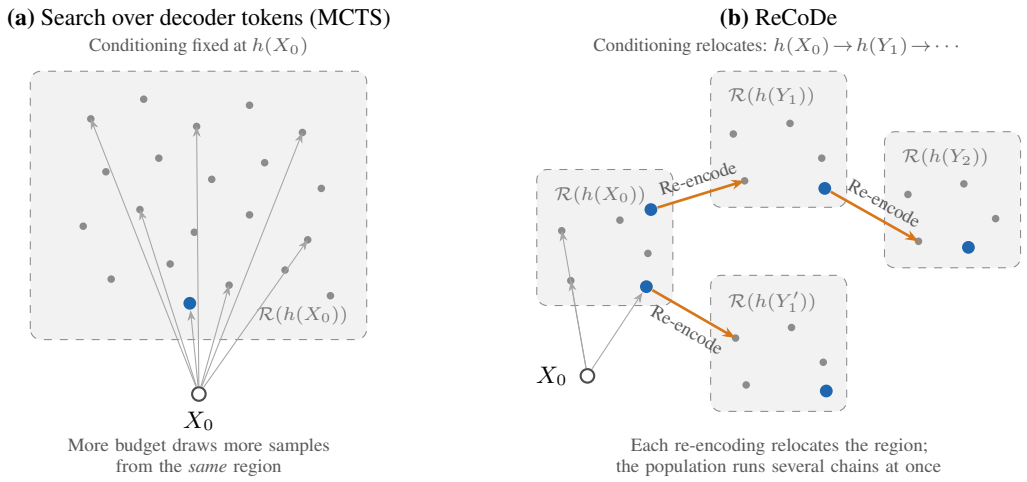


Figure 5: Two ways to spend inference compute under a decoder whose mask M fixes which outputs are *legal*. The conditioning does not change that set; it shapes the prior over it, and so decides which part is realistically reachable, written $\mathcal{R}(h)$. **(a)** Token-level search draws more samples from one fixed conditioning, refining $\mathcal{R}(h(X_0))$ without leaving it. **(b)** Re-encoding sets a new conditioning and hence a new reachable region, so improvements compound; a population traverses several chains at once. Orange marks a re-encoding step, blue an elite seeding the next region.