Scalable and Effective Arithmetic Tree Generation for Adder and Multiplier Designs

Yao Lai[1], **Jinxin Liu**[3], **David Z. Pan**[2], **Ping Luo**[1]
[1]The University of Hong Kong, [2]The University of Texas at Austin,
[3]Zhejiang University

Abstract

Across a wide range of hardware scenarios, the computational efficiency and physical size of the arithmetic units significantly influence the speed and footprint of the overall hardware system. Nevertheless, the effectiveness of prior arithmetic design techniques proves inadequate, as they do not sufficiently optimize speed and area, resulting in increased latency and larger module size. To boost computing performance, this work focuses on the two most common and fundamental arithmetic modules, adders and multipliers. We cast the design tasks as single-player tree generation games, leveraging reinforcement learning techniques to optimize their arithmetic tree structures. This tree generation formulation allows us to efficiently navigate the vast search space and discover superior arithmetic designs that improve computational efficiency and hardware size within just a few hours. Our proposed method, **ArithTreeRL**, achieves significant improvements for both adders and multipliers. For adders, our approach discovers designs of 128-bit adders that achieve Pareto optimality in theoretical metrics. Compared with PrefixRL, it reduces delay and size by up to 26% and 30%, respectively. For multipliers, compared to RL-MUL, our method enhances speed and reduces size by as much as 49% and 45%. Additionally, ArithTreeRL's flexibility and scalability enable seamless integration into 7nm technology. We believe our work will offer valuable insights into hardware design, further accelerating speed and reducing size through the refined search space and our tree generation methodologies. Codes are released at github.com/laiyao1/ArithmeticTree.

1 Introduction

Since the inception of computers, researchers have striven to boost computing speed and decrease hardware size. High computing speed is essential for a wide range of real-world applications, such as artificial intelligence [1], high-performance computing [2], and high-frequency trading [3], particularly for the recent applications of large language models like GPT [4]. Concurrently, the demand for smaller hardware has escalated due to the growth of wearable devices and IoT technology [5].

Hardware specialists have steadily miniaturized CMOS technology [6] to boost processor speeds and shrink chip sizes. However, as CMOS technology's scaling nears its fundamental physical limits [7], further miniaturization poses significant challenges. Therefore, exploring innovative circuit design has emerged as a vital alternative to drive performance enhancement and area reduction. Among the family of arithmetic modules for hardware architectures, adders and multipliers constitute two essential modules, playing a critical role in various computational operations. For example, basic addition and multiplication operations compute all convolution and fully connected layers

Corresponding to: Ping Luo (pluo@cs.hku.hk).

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

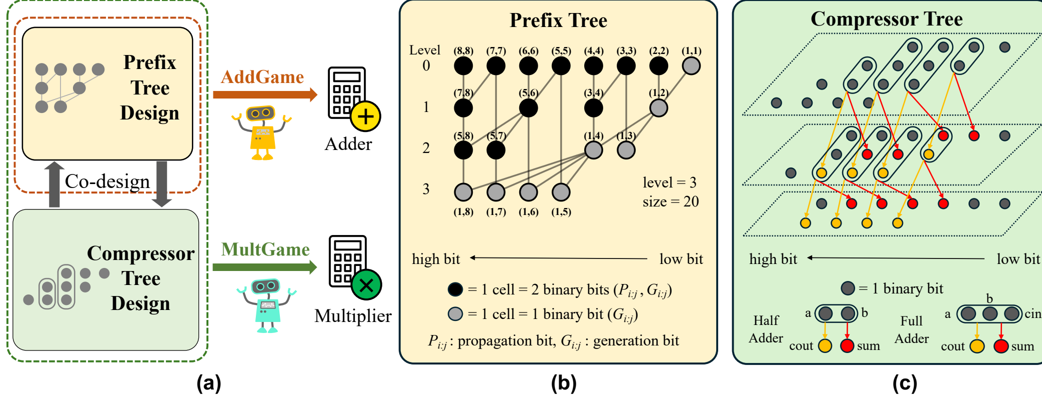

Figure 1: **(a) ArithTreeRL framework.** Two agents optimize prefix and compressor trees, respectively, modeling the tasks as AddGame for adders and MultGame for multipliers. **(b) Prefix tree. (c) Compressor tree.** Different tree structures lead to different qualities of adder and multiplier designs.

of deep learning models. Performance analysis of the ResNet model [8] reveals that the convolution operation, consisting solely of addition and multiplication, constitutes 98.4% of the overall GPU execution time during model inference. Under Amdahl's Law [9], an enhancement of 30% in addition and multiplication operation speeds could result in a 29% improvement in inference speed. Intriguingly, this improvement is comparable to the speedup typically seen with a generational upgrade in semiconductor process technology [10, 11]. Thus, designing more efficient and compact adders and multipliers is crucial for the overall advancement of hardware design.

Numerous arithmetic module design methods have been proposed in recent years. These techniques generally fall into one of three main categories: human-based [12, 13], optimization-based [14–16], and learning-based [17–19]. However, these methods either demand significant hardware expertise or get trapped in local optimal due to the vast design search space for adders and multipliers modules. For human-based methods, hardware experts have crafted a variety of arithmetic modules, such as the Sklansky adder [12] and the Wallace multiplier [13]. Nevertheless, designing new structures becomes increasingly challenging for humans as input bits increase. Optimization-based methods, like bottom-up enumerate search [14, 15] and integer linear programming [16], can enhance the quality of arithmetic designs by exploring a wider variety of structures. Despite their potential, the extensive search space poses a challenge, necessitating manually defined assumptions to limit the search scope for feasible computation. For example, Ma et al. [20] assumed the existence of semi-regular structures in adders, which may lead to locally optimal solutions. While learning-based approaches have emerged as a promising tool for automating hardware design in recent years [17–19], navigating the vast design space to find the optimal solution for arithmetic modules remains a formidable challenge. For example, the two primary components of an N-bit multiplier, the compressor tree and the prefix tree, have approximately $O(2^{N^2})$ and $O(2^{4N^2})$ design space [17], respectively. Consequently, the search space of a simple 16-bit multiplier is already comparable to that of the Go game (3^{361}) [21]. Meanwhile, such learning-based approaches also fail to consider the joint optimization of different components within arithmetic hardware [17, 18], thus easily leading to degenerated hardware with undesired performance bottleneck.

To resolve the above limitations and boost the performance, we formulate the arithmetic adder and multiplier design problems as two single-player tree generation games, AddGame and MultGame, respectively, as shown in Fig. 1a. The key insight is that by reframing the design problems into interactive tree generation games, we harness the power of progressive optimization algorithms, allowing us to explore the intricate design space of arithmetic units dynamically. Starting from an initial prefix tree, the player in AddGame sequentially modifies cells in the prefix tree, in the same spirit as tactical movements in board games. Our MultGame contains two parts, specifically for designing the compressor tree and the prefix tree of multipliers. The compressor tree design involves the player compressing all partial products with different compressors, similar to a match game. In contrast, the prefix tree design follows the same rules as the AddGame. Unlike the default design process depicted in Fig. 2a, the tree structures discovered in games are converted into specific Verilog codes [22], as illustrated in Fig. 2b. We demonstrate that the delay and area of arithmetic modules can be largely decreased by substituting the default designs with our discovered tree structures.

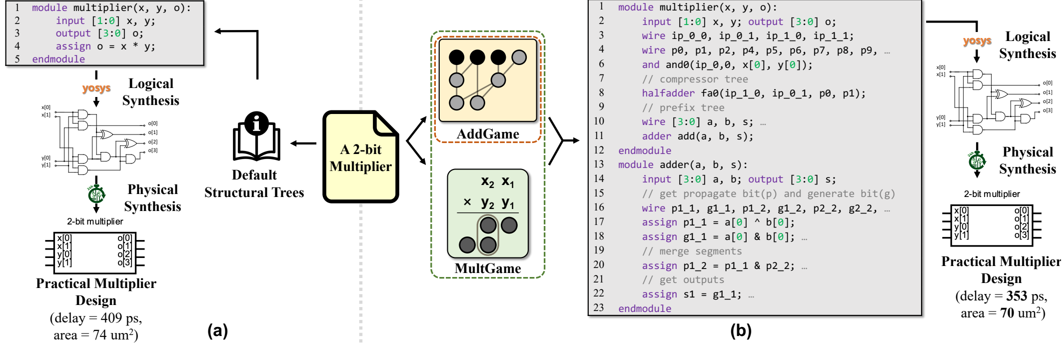

Figure 2: **Comparison of design processes. (a) Default design process.** The synthesis tool automatically generates a default multiplier when using multiplication commands (x*y) in Verilog HDL code. **(b) Enhanced design process in ArithTreeRL.** ArithTreeRL discovers an optimized multiplier structure and generates specialized Verilog HDL code for this improved structure, reducing delay and area after synthesis.

We propose **ArithTreeRL** (Arithmetic Tree Reinforcement Learning), a novel approach that utilizes customized reinforcement learning agents for optimizing arithmetic tree structures. In practical implementation, ArithTreeRL employs two distinct agents tailored to the specific characteristics of prefix and compressor tree optimization. For the prefix tree, appearing in both AddGame and MultGame, we utilize a Monte-Carlo Tree Search (MCTS) [23] agent to efficiently explore the large action space while preserving previous exploration experience. For the compressor tree, exclusive to MultGame, we take a Proximal Policy Optimization (PPO) [24] agent due to its superior exploration efficiency. To capture the global design for multiplier designs, we also designed an optimization curriculum as depicted in Fig. 1, iteratively running MCTS and PPO agents to refine the prefix and compressor trees.

This paper has three main **contributions**. Firstly, we model the arithmetic module design tasks as single-player tree generation games, *i.e.,* AddGame and MultGame, which inherit the well-established RL capabilities for complex decision-making tasks (arithmetic tree optimization). Secondly, we propose a co-designed framework that integrates prefix and compressor tree modules, enabling the discovery of optimal combinations that lead to global optimal multipliers. Thirdly, our experiments reveal that our designed 128-bit Pareto-optimal adders outperform the latest theoretical designs. Also, our designed adders achieved up to 26% and 30% reductions in delay and area compared to PrefixRL [17], and multipliers offer 33% and 45% improvements over RL-MUL [18] in the same metrics. These designs are ready for direct integration into synthesis tools, offering significant industrial benefits, and are flexible and scalable enough to be seamlessly adopted into 7nm technology.

2 Preliminaries

Adder Design. An N-bit adder can be constructed by cascading N 1-bit adders. However, this approach results in an $O(N)$ delay due to the sequential propagation of the carry signal from the lower bit to the higher bit. To address this issue, prefix adders have been proposed [25, 26]. Prefix adders are designed based on the principles of addition, with a focus on reusing and parallelizing intermediate signal bits. These signal bits can be divided into two categories: propagation bits $p_i = a_i \oplus b_i$ and generation bits $g_i = a_i \cdot b_i$, where $a_i, b_i \in \{0, 1\}, i \in \{1, 2, \dots, N\}$ represent the addends at the i-th bit, and '$\oplus$' and '$\cdot$' denote the logic XOR and AND operations, respectively [27]. These propagation and generation signals can be defined at both the individual bit level and across a range of bits. For an individual bit with index i, they are denoted by $P_{i:i} = p_i$ and $G_{i:i} = g_i$. When considering a range of bits, this range is treated as an interval identified by a tuple (i, j). Within each such interval, we have a single propagation signal $P_{i:j} = \prod_{k=i}^{j} p_k$ and a single generation signal $G_{i:j} = g_j + \sum_{k=i}^{j-1} P_{k:j} \cdot g_k$, where '$+$' represents the logic OR operation. Note that the computation of $P_{i:j}$ and $G_{i:j}$ is influenced solely by the input bits from position i to j. The $(N + 1)$ outputs of the adder can be calculated from the signal bits with the initial condition $G_{1:0} = 0$ by $c_{N+1} = g_N + p_N \cdot G_{1:N}$ and $s_i = p_i \oplus G_{1:i-1}$, where c_{N+1} is the carry-out bit and s_i is the i-th sum bit.

The prefix adder design aims to optimize a hierarchical tree structure that generates all intervals $(1, i)$ from the initial intervals (i, i), as shown in Fig. 1b. Signal bits for two adjacent intervals, (i, k) and

$(k+1, j)$, can be merged to form the larger interval (i, j) by the computations $P_{i:j} = P_{i:k} \cdot P_{k+1:j}$ and $G_{i:j} = G_{i:k} \cdot P_{k+1:j} + G_{k+1:j}$. This merging process generates a prefix tree where each cell represents an (i, j) interval with two signal bits. If an interval results from merging two others, its corresponding cell is the child node in the tree, and the merged intervals are its parent node. For example, the $(5, 8)$ cell is the child node of the $(5, 6)$ and $(7, 8)$ cells because it derived from them. A key advantage of this structure is that cells with no dependencies can be computed in parallel. Different tree structures can result in adders with varying delays and areas. When evaluating the theoretical quality of the prefix adder, We can use level (tree height) and size (number of cells) as theoretical metrics to substitute for practical metrics like delay and area.

Multiplier Design. An N-bit multiplier carries out the multiplication of two N-bit multiplicands, which can be regarded as the cumulative addition of N addends, involving a total of N^2 bits. Each addend represents a partial product with different powers of two weights, illustrated in Fig. 4b. Multipliers can be easily achieved by cascading $(N-1)$ N-bit adders or using a single N-bit adder $(N-1)$ times. However, both result in a large area or high delay. To mitigate this, the N^2 bits in the partial products can be added simultaneously by 1-bit adders, which can also be seen as a bit compression process because the number of bits gradually decreases. The compression process halts when the number of bits for each binary digit is reduced to two or fewer before feeding into a downstream adder, as illustrated in Fig. 1c. The process generates a compressor tree, describing a compression mechanism that merges N^2 bits into fewer bits by compressors such as half and full adders. Introducing an additional carry-in input distinguishes a full adder from a half adder, as shown in Fig. 1b, which affects the latency and area. The difference is crucial when configuring the compressor tree in multipliers to optimize for delays and area requirements. Upon completing the compression, the remaining bits are processed by a $2N$-bit prefix adder, designed to yield the globally optimal multiplier. In summary, adder and multiplier design tasks can be interpreted as a tree-based structural generation process to optimize hardware metrics while maintaining functionality.

3 Our Approach

We use reinforcement learning to solve the tree generation for adder and multiplier designs. The environments are modeled as single-player tree generation games: AddGame for adder design and MultGame for multipliers, as illustrated in Fig. 1a. Considering differences among the games, such as action space, we propose two types of agents: one by MCTS [28] and another by PPO [24].

3.1 AddGame

AddGame is modeled for designing prefix trees in adders and multipliers, as shown in Fig. 3. In this game, the player modifies the structures of given initial prefix trees by basic actions to optimize the adders' metrics. The state of the game is denoted as s, corresponding to the current prefix tree. In our evaluation, each state s is assessed on two theoretical metrics, level and size, and two practical metrics, delay and area. The player always chooses one action from two kinds of actions: (1) delete a cell (i, j), (2) add a cell (i, j), which (i, j) is the cell index as shown in Fig. 1b. A cell (i, j) $(i < j)$ can be deleted if the prefix tree does not have the cell (i, k) subject to $k > j$ and $i > 1$, and all deletable cells are marked in red in Fig. 3 and 5. A cell (i, j) can be added if it does not exist in the prefix tree. All positions where cells can be added are marked with '×'. A legalization operation [17] is always executed after one action to guarantee the feasibility of the prefix tree as Fig. 3. The game aims to maximize the performance score $R(s)$ of the adder s. This score is determined by a weighted combination of delay and area (using level and size when optimizing theoretical metrics).

Given the large action space, the agent for playing AddGame is based on an improved MCTS method, which has demonstrated its effectiveness in numerous game tasks [21, 29, 30]. Starting from the prefix trees in human-designed adders, the MCTS agent continuously cycles through four phases: *selection*, *expansion*, *simulation*, and *backpropagation*, and gradually builds a search tree in this process. Each node in the search tree represents one prefix tree.

In the *selection* phase, the agent selects the child node with state s that has the highest score $W(s)$, continuing until it encounters a node that has not been fully expanded. The scores for evaluating nodes are computed by the Upper Confidence bounds applied to Trees (UCT) [31], keeping the balance between exploration and exploitation. In the search tree, each node with the state s stores a visit count $N(s)$ and an action value $V(s)$. The visit count $N(s)$ records the number of visits to the

node s. The action value $V(s)$ is the weighted sum of the best performance score $\max R$ and average performance score $\overline{R}$ of all its descendant nodes, which can be formalized as:

$$V(s) = (1-\beta)\underbrace{\sum_{s' \in D(s)} R(s')/|D(s)|}_{\text{avg performance score}} + \beta \underbrace{\max_{s' \in D(s)} R(s')}_{\text{best performance score}} \tag{1}$$

where $D(s)$ represents all descendant nodes of the node s (including s itself), *i.e.,* all generated adders by a sequence of actions from adder s. $|\cdot|$ gives the number of nodes. $R(s')$ indicates the performance score of the adder of the state s', which is defined as $-$Delay $-$ αArea or $-$Size. α and β are sum weights.

We define the node score $W(s)$ with the state s as follows:

$$W(s) = \sqrt{\frac{\ln N(P(s))}{N(s)}} + cV(s) \tag{2}$$

where $P(s)$ is the parent node of s, $N(\cdot)$ is visit count function, and c is an adjustable parameter.

In the *expansion* phase, a random action is chosen from the unexplored actions available at the node identified in the selection phase and executed. It expands the search tree by adding a new node corresponding to the result after that action. In the *simulation* phase, a sequence of actions is taken until the performance scores of adders can no longer be improved (in theoretical metrics optimization) or the simulation exceeds the maximum steps (in practical metrics optimization). In the *backpropagation* phase, the last state s reached in the simulation phase is evaluated to get a performance score $R(s)$, which is then backpropagated to update the scores of all preceding nodes in the search tree.

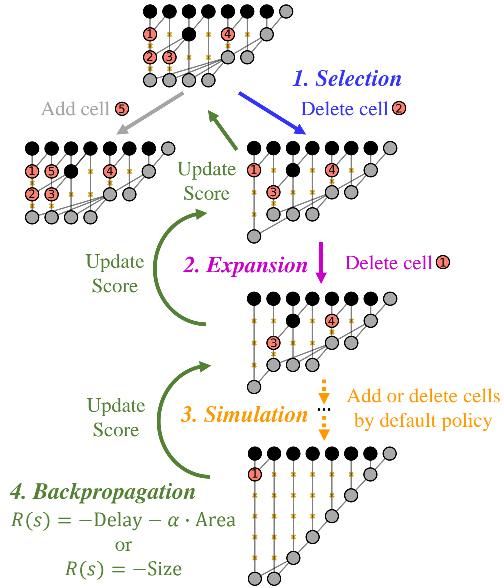

Figure 3: **Method for designing prefix trees with MCTS.** Four phases in the search process are executed iteratively, gradually building a search tree.

Pruning. To enhance efficiency, we implement pruning techniques to avoid the exploration of unnecessary sub-trees. When optimizing theoretical metrics, we restrict modifications to delete cell actions, as adding cells does not improve the design outcome. Furthermore, we impose an upper limit on the level metric to prevent the creation of structures with excessively high complexity. This upper limit, denoted as L, is set for each MCTS search and is gradually relaxed with each search iteration.

Two-level Retrieval. We adopt a two-level retrieval strategy to balance synthesis accuracy and computational efficiency. We divide the search into two stages because the full synthesis flow is highly accurate but time-consuming. A faster yet marginally less simulating accurate synthesis flow is employed in the first stage, eliminating the time-intensive steps such as routing. Only the top K adders identified in the first stage undergo full synthesis in the second stage.

3.2 MultGame

MultGame consists of two parts for jointly designing compressor and prefix trees in multipliers, as shown in Fig. 1. The part focused on the prefix tree design is identical to that in AddGame. Meanwhile, the part focused on the compressor tree design involves continually merging bits in partial products through compression actions, as depicted in Fig. 4. This process is similar to some match games like '2048' [32], where items are merged in a specific way to achieve high scores.

The compressor tree is built from scratch instead of starting from existing solutions for more design flexibility. The game state s_t at step t is represented by a vector representing the current compressor tree status. The player chooses one of two actions: (1) using a half adder or (2) using a full adder to

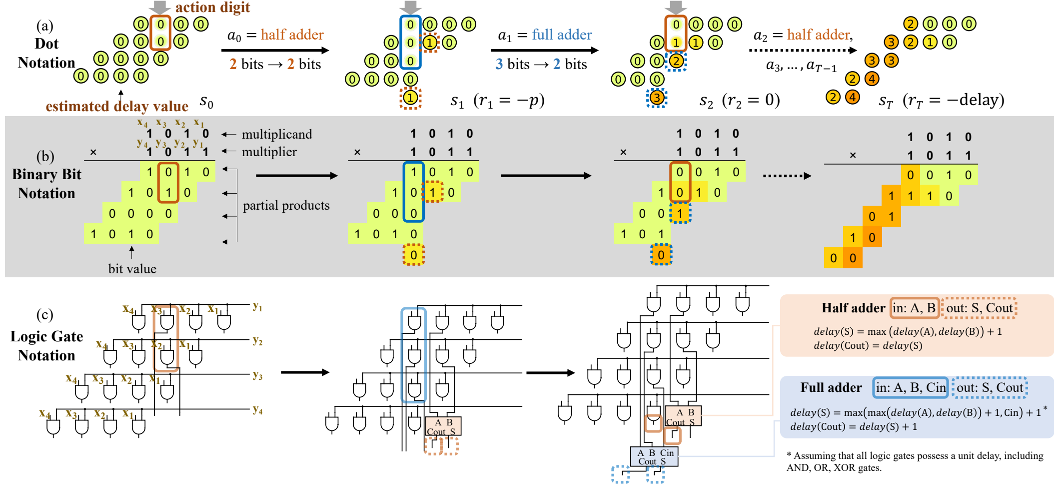

Figure 4: **Designing compressor trees with PPO.** Three representations are illustrated. **(a) Dot notation.** Each dot represents an output bit, with the number inside indicating the estimated delay for selecting adder input bits. The agent's actions involve adding full or half adders to compress the bits until each binary digit contains no more than two bits. The final reward, r_T, is defined as the inverse of the delay, encouraging designs with lower delays. **(b) Binary bit notation.** 0/1 are values of bits for the example multiplication. **(c) Logic gate notation.** The actual logic gate circuit design for each state.

compress bits at the action digit, which is defined as the lowest digit containing more than two bits, as indicated in Fig. 4a. Half and full adders compress two or three bits in the k-th digit and generate a carry-out bit in the $(k+1)$-th digit and a sum bit in the k-th digit. Rough delays for all bits are estimated, assuming a one-unit delay for all basic logic gates, as shown in the dots of Fig. 4a. To minimize the increase in total delay, the bits with the lowest estimated delays are selected as inputs for the adders. The game terminates at step T when all digits have two or fewer bits. A reward r_T is computed through the synthesis tools as the negative of the delay, denoted $r_T = -\text{delay}$. Moreover, a penalty term $-p$ is also applied to r_t if the action a_{t-1} uses a half adder, where $1 \leq t \leq T$. This penalty reflects that a full adder accepts three input bits (two addend bits and a carry-in bit) and produces two output bits (a sum bit and a carry-out bit), effectively reducing the bit count. In contrast, a half adder only processes two addend bits and outputs two bits, thus not contributing to a reduction in bit count. A half adder's lack of bit count reduction can lead to more adder modules, increasing the overall module area.

We train an RL agent with policy and value networks using the PPO method. Both networks are built by multi-layer perceptions (MLPs) [33] with three layers. The inputs comprise pre-defined features as Table 1, including action digit, max delay, number of half adders, eligible action type, and the estimated delays of bits. The policy and value networks contain $(64, 16, 2)$ and $(64, 8, 1)$ neurons in each layer. The last layer of the policy network is connected to a Softmax activation function [34] for choosing actions.

Table 1: **State features for policy and value network.**

Feature	**Size**	**Description**
Action digit	1	Digit for action.
Max delay	1	Maximum estimated delay value of all bits.
Number of half adders	1	Number of added half adders in action digit.
Mask for action	2	The mask for ensuring valid action.
Delay of action bits	3	The delays of bits for action.

When training, the objective function can be defined as follows for maximizing the game's cumulative reward:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[G_T\right] = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{i=0}^{T} \gamma^i r_i\right] \tag{3}$$

where $\tau = (s_0, a_0, s_1, r_1, a_1, ..., a_{T-1}, s_T, r_T)$ is a trajectory from the game episode, and π_θ denotes the policy parameterized by θ. G_T refers to the cumulative discounted reward from step 0 to step T. The discount factor γ adjusts the emphasis between immediate and future rewards. When implementing the PPO, the objective function for optimizing the policy network can be formalized as:

$$L(\theta) = \hat{\mathbb{E}}_t\big[\min\big(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\big)\big] \tag{4}$$

where $\hat{\mathbb{E}}_t[\cdot]$ indicates the empirical average over a finite batch of samples, and $r_t(\theta)$ denotes the probability ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$. Here, θ_{old} is the policy network parameters before the update. $\hat{A}_t = G_t - \hat{V}_t$ is an estimation of the advantage function at step t, and $\hat{V}_t$ is the value estimated by the value network. $\text{clip}(\cdot, 1-\epsilon, 1+\epsilon)$ is the function restricting results to the interval $[1-\epsilon, 1+\epsilon]$.

Simultaneously, the value network with parameters ϕ is updated by optimizing the following objective function $L(\phi) = \hat{\mathbb{E}}_t\big[\text{smooth_L1}(G_t, \hat{V}_t)\big]$, where smooth_L1$(\cdot)$ is the smooth L1 loss function [35].

Synthesis Acceleration. In RL-MUL [18], running synthesis tools proved to be a bottleneck, especially for scaling to multipliers with higher bit-widths. To address this, our enhancements to the synthesis flow yield a 10× speedup in reward computation without sacrificing accuracy. These modifications facilitate the design of multipliers up to 64-bit, expanding from the 16-bit limit in RL-MUL. Enhancements include activating the fast mode in the logical synthesis script and adopting direct code template-based generation of Verilog HDL code from our search results, moving away from the time-consuming EasyMAC [36] tool.

Co-design Framework. As shown in Fig. 1, we developed a joint design approach to optimize the multiplier's two primary components: the prefix and compressor trees. Our method involves an iterative process where each round involves optimizing the compressor tree with a fixed prefix tree and searching for an ideal prefix tree that aligns with the optimized compressor. This alternating optimization continues until the computational iterations conclude.

4 Experiments

We use the logic synthesis tool Yosys 0.27 [37] and the physical synthesis tool OpenROAD 2.0 [38] with Nangate45 [39] and ASAP7 [40] libraries to implement experiments. Both synthesis tools are open-sourced for result reproduction. All experiments are run on one GeForce RTX 3090 GPU and one AMD Ryzen 9 5950X 16-core CPU. Detailed settings are in Appendix A.3 and A.5. All designed modules have successfully undergone functional verification.

4.1 Adder Design

Theoretical Evaluation. As illustrated in Fig. 1b, prefix tree structures define the technology-independent theoretical metrics of level and size. Empirically, optimizing the level usually presents more challenges than size. Therefore, we set the search objective when optimizing theoretical metrics to find the optimal size for each specified upper bound level L. We begin our search with the Sklansky adder [12], which has a theoretical minimum level of $\log_2 N$. Starting with L set at this minimum, we incrementally increase it for each new iteration, using the smallest prefix tree identified in the previous round as the initial state. We limited the number of steps to 4×10^5 for each search iteration. For baselines, the results were obtained directly from the respective original publications. Table 2 shows that our method surpasses the state-of-the-art designs in [14]. Some discovered adder structures are presented in Fig. 5. Despite the exponentially growing search space, our MCTS method can enhance 128-bit adders, surpassing the designs from optimization-based methods. Notably, guided by Snir's theoretical lower bound for size at a given level [41], we were the first to discover an optimal 128-bit adder with 10 levels and a size of 244.

Practical Evaluation. Practical metrics, including the delay and area of hardware modules, are computed through synthesis tools for evaluation. We run 1000 full syntheses for adders in each method to ensure a fair comparison. Our ArithTreeRL method begins each search from one of three adders: Sklansky [12], Brent-Kung [43], and ripple-carry [44]. A two-level retrieval strategy is implemented by dividing the search into two stages: (1) 5000 fast syntheses. (2) 500 full syntheses with the top 500 adders selected from the first stage. Efficiency tests show that one full flow's computational load equals 10 fast flows. Thus, the proposed strategy achieves the same computational

Table 2: **Comparisons of discovered adders in size and area.** Smaller sizes are preferable.

Input Bit	Level	Theory Size Bound [41]	Sklansky Size [12]	Area Heuristic [42]	Best Known Size [14]	ArithTreeRL
64	6	120	192	169	**167**	**167**
64	7	119	-	138	**126**	**126**
64	8	118	-	120	**118**	**118**
64	9	117	-	**117**	**117**	**117**
64	10	116	-	**116**	**116**	**116**
128	7	247	448	375	**364**	**364**
128	8	246	-	304	276	**273**
128	9	245	-	284	250	**248**
128	10	244	-	257	245	**244**

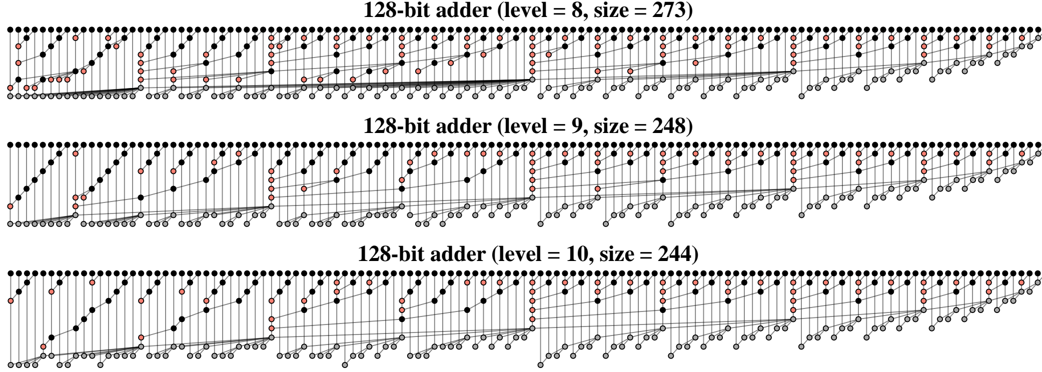

Figure 5: **Some first discovered prefix trees for 128-bit adders with the smallest sizes.**

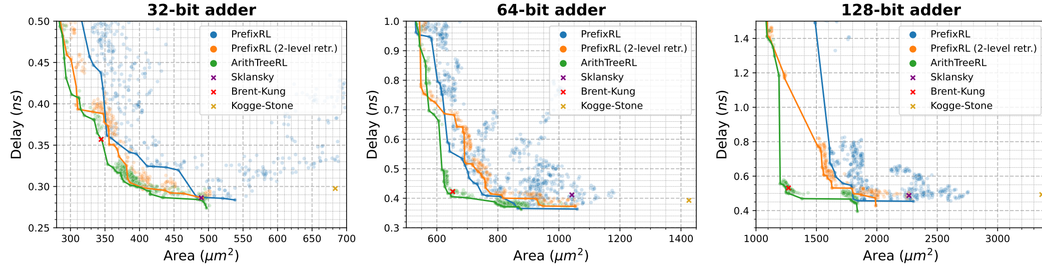

Figure 6: **Comparison of adders in delay and area.** Each point represents one adder and line segments connect Pareto-optimal adders. 'PrefixRL (2-level retr.)' is the raw PrefixRL method improved by our two-level retrieval strategy. Sklansky, Brent-Kung, and Kogge-Stone refer to human-designed adders. ArithTreeRL can significantly improve the delay and area, particularly for high-bit adders. Furthermore, it can discover adders with minimal delays. Our two-level retrieval strategy can effectively find superior designs.

volume with 1000 full flows. The state-of-the-art method PrefixRL [17] is implemented with optimal settings. In our results in Fig. 6, each prefix adder is represented by a 2D point based on its delay and area. It shows the significant improvement achieved when our two-level retrieval strategy is used in the PrefixRL method due to efficiency improvements that facilitate exploring an expanded sample corpus. Moreover, employing the MCTS method can lead to the discovery of more superior adders because this method effectively navigates through problems with vast state spaces, utilizing information stored during the search process. Overall, our approach can reduce the delay or area of adders by up to 26% and 30%, respectively, compared with PrefixRL, while maintaining the computational amount.

Visualization. The scores of the first actions after 400 search steps when optimizing the theoretical metrics are visualized as heatmaps in Fig. 7. In the selection phase, the action with the highest score is chosen. For example, the first action for the 8-bit adder is to delete the $(5, 7)$ cell with the highest score because this reduces the size of the adder. On the contrary, the action with the lowest score is to add the $(4, 7)$ cell because it augments both size and level.

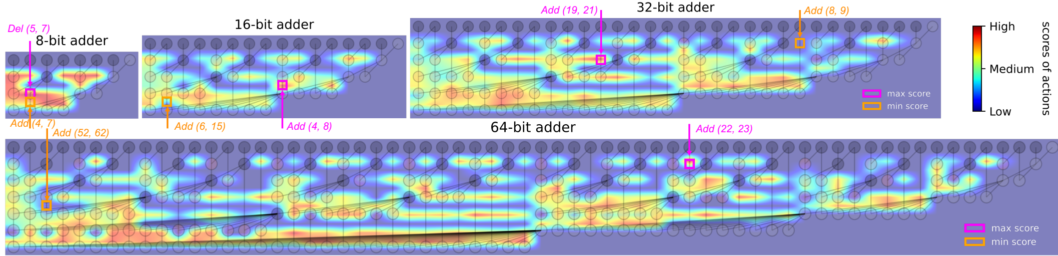

Figure 7: **Heatmap for first action scores.** The actions with the highest and lowest scores are marked.

Accuracy of Fast Flow. The time-consuming routing phase is removed in the fast flow of the two-level strategy. To evaluate the impact of this simplification, we tested the simulation accuracy of the fast flow against the full flow. The results in Table 3 indicate that the fast flow can still achieve an utterly accurate area estimation and over 95% accurate delay. Therefore, the fast synthesis flow can help improve efficiency without significantly losing accuracy.

Table 3: **Accuracy of fast synthesis flow.**

Bits of adders	32	64	128
Delay Acc. (%)	96.11±0.86	95.82±1.12	95.34±2.60
Area Acc. (%)	100.00±0.00	100.00±0.00	100.00±0.00

4.2 Multiplier Design

Practical Evaluation. Given the lack of a commonly adopted theoretical metric for multipliers, we use practical metrics for evaluation. Our multiplier design utilizes a co-design framework with three iterative search rounds, incorporating 900 steps for the compressor tree and 100 for the prefix tree each round. This yields 3000 steps, consistent with the search steps of other baseline methods in our experiments. As shown in Fig. 8 and Appendix Fig. 14, we compared the effectiveness of our method with several baselines, including the human-designed Wallace multiplier [13], optimization-based methods including GOMIL [16] and SA [45], the default multiplier given in the synthesis tool, and the learning-based method RL-MUL [18]. In our evaluation, we assessed the multipliers' performance by adjusting the expected delay parameter in the synthesis process. Subsequently, the resulting areas of each multiplier at different delays are depicted as a segmented line. Consistent with the RL-MUL [18] assessment approach, each method selects an optimal multiplier for comparison. Results for Wallace [13], GOMIL [16], SA [45], and RL-MUL [18] in 8/16 bits are referenced from the RL-MUL work. RL-MUL method is reproduced and tested in 32/64 bits. The results show that the co-design method, ArithTreeRL, outperforms the synthesis tool's baselines and default multipliers. This is because of the co-design framework, the restructured MultGame, and the improved synthesis flow. It can achieve second-best results even when only optimizing the compressor tree. Compared with the state-of-the-art RL-MUL method, our method can reduce the delay by up to 33% and the area by 45%. Furthermore, our method can reduce the delay of the default multipliers used in the Yosys tool [37] by up to 16% and the area by 35%. We also report the delays and areas in Table 4. Our method consistently achieves minimal delays for the delay minimization. When optimizing for a trade-off (delay + 0.001area), our approach achieves optimal or comparable results. Also, The multipliers designed by 45nm technology are compatible with the 7nm [40] without any modifications.

Efficiency. Due to the time-consuming nature of the full synthesis flow, we developed a synthesis flow that is over 10× faster while maintaining high simulation accuracy for adder design, as discussed in the method. The efficiency is shown in Fig. 9a. Additionally, we optimized the logic synthesis and HDL code generation processes in the synthesis flow for multiplier design. According to Fig. 9b, our improved fast flow can accelerate the process up to 20×.

5 Conclusion

Designing adder and multiplier modules is a fundamental and crucial task in computer science. We first model this task as a tree-generation process, conceptualizing it as a sequential decision-making

Table 4: **Numerical comparison of multipliers in delay (ns) and area** (μm^2)**.** (45nm)

	Num of bits	8-bit		16-bit		32-bit		64-bit	
Objective	Method	area	delay	area	delay	area	delay	area	delay
Min Delay	RL-MUL	496	0.7089	2271	1.1330	8767	2.0150	34810	2.6771
	PPO w/ raw flow	496	0.6921	2259	1.1277	8788	1.9437	34810	2.6355
	Default	555	0.6203	2499	0.8908	10637	1.0745	42128	1.3498
	PPO	692	0.5180	2551	0.7392	11329	0.9960	41237	1.2424
	ArithTreeRL	714	**0.4905**	2955	**0.7138**	11460	**0.9685**	39436	**1.2401**
Trade-off	RL-MUL	388	0.7691	1695	1.2668	7033	2.1932	28616	2.8891
	PPO w/ raw flow	388	0.7618	1687	1.2268	7036	2.0945	28609	2.8928
	Default	**367**	0.6837	1590	0.9997	6685	1.4170	26871	1.9403
	PPO	377	0.6558	1568	1.0135	6581	1.3856	26088	1.7941
	ArithTreeRL	384	**0.6420**	**1566**	**0.9487**	**6469**	**1.3262**	**26087**	**1.7038**

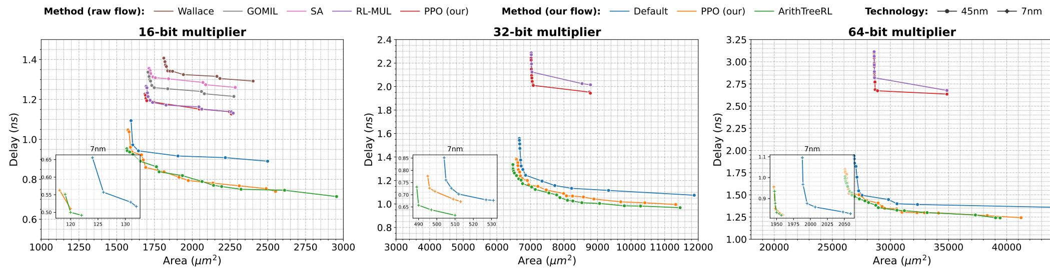

Figure 8: **Comparison of multipliers.** The designs were tested in 45nm and 7nm. Each segmented line represents the performance of one multiplier under different timing constraints. 'Method (our flow)' are methods with our improved flow. The 'Default' multipliers are those generated by the synthesis tool by default. 'ArithTreeRL' is our co-design method combining PPO and MCTS, while 'PPO (our)' optimizes only the compressor tree. We apply 45nm designs to the 7nm library without modifications, showcasing the transferability.

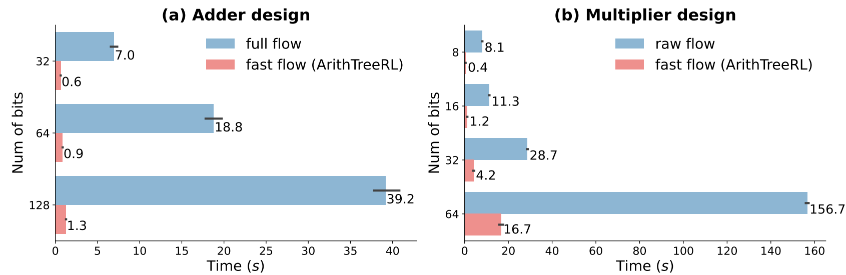

Figure 9: **Design flow time consumption.** (average of 1000 runs)

game. Then, we propose a reinforcement learning method to solve it, facilitating a scalable and efficient search for globally optimal designs. Through extensive experiments, our approach achieves state-of-the-art performance for adders and multipliers in terms of delay and area within the same computational resources. Moreover, our method has demonstrated transferability, as the designs we discovered can be applied to more advanced technology processes. This enhancement in basic arithmetic modules optimizes hardware performance and size, showing significant potential for boosting computationally intensive fields.

Limitations. This paper focuses exclusively on designing and optimizing adder and multiplier modules, which are fundamental components in computational systems. It does not explore other basic elements, such as exponentiation or more complex arithmetic units. However, our method is naturally extendable to other arithmetic operations, such as exponentiation. Future research could explore these extensions to unlock further designs across various hardware components.

Acknowledgments and Disclosure of Funding

We gratefully acknowledge Ronghao Lin of Sun Yat-sen University for his assistance with the introduction video. This paper is partially supported by the National Key R&D Program of China No.2022ZD0161000 and the General Research Fund of Hong Kong No.17200622 and 17209324.

References

[1] P. Yao, H. Wu, B. Gao, J. Tang, Q. Zhang, W. Zhang, J. J. Yang, and H. Qian, "Fully hardware-implemented memristor convolutional neural network," *Nature*, vol. 577, no. 7792, pp. 641–646, 2020.

[2] M. Haseeb and F. Saeed, "High performance computing framework for tera-scale database search of mass spectrometry data," *Nature computational science*, vol. 1, no. 8, pp. 550–561, 2021.

[3] R. Orús, S. Mugel, and E. Lizaso, "Quantum computing for finance: Overview and prospects," *Reviews in Physics*, vol. 4, p. 100028, 2019.

[4] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, pp. 27 730–27 744, 2022.

[5] J. Min, S. Demchyshyn, J. R. Sempionatto, Y. Song, B. Hailegnaw, C. Xu, Y. Yang, S. Solomon, C. Putz, L. E. Lehner *et al.*, "An autonomous wearable biosensor powered by a perovskite solar cell," *Nature Electronics*, pp. 1–12, 2023.

[6] M. T. Bohr and I. A. Young, "Cmos scaling trends and beyond," *IEEE Micro*, vol. 37, no. 6, pp. 20–29, 2017.

[7] Y. Taur, "Cmos design near the limit of scaling," *IBM Journal of Research and Development*, vol. 46, no. 2.3, pp. 213–222, 2002.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. IEEE, 2016, pp. 770–778.

[9] D. P. Rodgers, "Improvements in multiprocessor system design," *ACM SIGARCH Computer Architecture News*, vol. 13, no. 3, pp. 225–231, 1985.

[10] T. Hiramoto, "Five nanometre cmos technology," *Nature Electronics*, vol. 2, no. 12, pp. 557–558, 2019.

[11] S. Salahuddin, K. Ni, and S. Datta, "The era of hyper-scaling in electronics," *Nature Electronics*, vol. 1, no. 8, pp. 442–450, 2018.

[12] J. Sklansky, "Conditional-sum addition logic," *IEEE Transactions on Electronic computers*, pp. 226–231, 1960.

[13] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on electronic Computers*, pp. 14–17, 1964.

[14] S. Roy, M. Choudhury, R. Puri, and D. Z. Pan, "Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures," in *Proceedings of the Annual Design Automation Conference (DAC)*. ACM/IEEE, 2013, pp. 1–8.

[15] ——, "Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 33, no. 10, p. 1517, 2014.

[16] W. Xiao, W. Qian, and W. Liu, "Gomil: Global optimization of multiplier by integer linear programming," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 374–379.

[17] R. Roy, J. Raiman, N. Kant, I. Elkin, R. Kirby, M. Siu, S. Oberman, S. Godil, and B. Catanzaro, "Prefixrl: Optimization of parallel prefix circuits using deep reinforcement learning," in *Proceedings of the Annual Design Automation Conference (DAC)*. ACM/IEEE, 2021, pp. 853–858.

[18] D. Zuo, Y. Ouyang, and Y. Ma, "RL-MUL: Multiplier design optimization with deep reinforcement learning," in *Proceedings of the Annual Design Automation Conference (DAC)*. ACM/IEEE, 2023, pp. 1–8.

[19] H. Geng, Y. Ma, Q. Xu, J. Miao, S. Roy, and B. Yu, "High-speed adder design space exploration via graph neural processes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 8, pp. 2657–2670, 2021.

[20] Y. Ma, S. Roy, J. Miao, J. Chen, and B. Yu, "Cross-layer optimization for high speed adders: A pareto driven machine learning approach," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 38, no. 12, pp. 2298–2311, 2018.

[21] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[22] S. Palnitkar, *Verilog HDL: a guide to digital design and synthesis*. Prentice Hall Professional, 2003, vol. 1.

[23] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[25] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers (TC)*, vol. 100, no. 8, pp. 786–793, 1973.

[26] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, 1980.

[27] C. H. Roth Jr, L. L. Kinney, and E. B. John, *Fundamentals of logic design*. Cengage Learning, 2020.

[28] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, "Monte carlo tree search: A review of recent modifications and applications," *Artificial Intelligence Review*, vol. 56, no. 3, pp. 2497–2562, 2023.

[29] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[30] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R Ruiz, J. Schrittwieser, G. Swirszcz *et al.*, "Discovering faster matrix multiplication algorithms with reinforcement learning," *Nature*, vol. 610, no. 7930, pp. 47–53, 2022.

[31] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning (ECML)*. Springer, 2006, pp. 282–293.

[32] A. Dedieu and J. Amar, "Deep reinforcement learning for 2048," in *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[33] F. Murtagh, "Multilayer perceptrons for classification and regression," *Neurocomputing*, vol. 2, no. 5-6, pp. 183–197, 1991.

[34] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4.

[35] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision (ICCV)*. IEEE, 2015, pp. 1440–1448.

[36] J. Zhang, Q. Gao, Y. Guo, B. Shi, and G. Luo, "Easymac: design exploration-enabled multiplier-accumulator generator using a canonical architectural representation," in *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2022, pp. 647–653.

[37] C. Wolf, "Yosys open synthesis suite," 2016.

[38] T. Ajayi and D. Blaauw, "Openroad: Toward a self-driving, open-source digital layout implementation tool chain," in *Proceedings of Government Microcircuit Applications and Critical Technology Conference*, 2019.

[39] I. NanGate, "NanGate FreePDK45 open cell library," 2008.

[40] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm FinFET predictive process design kit," *Microelectronics Journal*, vol. 53, 2016.

[41] M. Snir, "Depth-size trade-offs for parallel prefix computation," *Journal of Algorithms*, vol. 7, no. 2, pp. 185–201, 1986.

[42] T. Matsunaga and Y. Matsunaga, "Area minimization algorithm for parallel prefix adders under bitwise delay constraints," in *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, 2007, pp. 435–440.

[43] Brent and Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers (TC)*, vol. 100, no. 3, pp. 260–264, 1982.

[44] P. Behrooz, "Computer arithmetic: Algorithms and hardware designs," *Oxford University Press*, vol. 19, pp. 512 583–512 585, 2000.

[45] P. J. Van Laarhoven, E. H. Aarts, P. J. van Laarhoven, and E. H. Aarts, *Simulated annealing*. Springer, 1987.

[46] J. K. Ousterhout *et al.*, *Tcl: An embeddable command language*. University of California, Berkeley, Computer Science Division, 1989.

[47] S. D. Compiler, "Synopsys design compiler," *Pages/default. aspx*, 2016.

[48] N. H. Weste and D. Harris, *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India, 2015.

[49] T. Han, D. A. Carlson, and S. P. Levitan, *VLSI DESIGN OF HIGH-SPEED, LOW-AREA ADDITION CIRCUITRY.* IEEE, 1987.

[50] J. Liu, S. Zhou, H. Zhu, and C.-K. Cheng, "An algorithmic approach for generic parallel adders," in *International Conference on Computer Aided Design (ICCAD)*. IEEE, 2003, pp. 734–740.

[51] J. P. Fishburn, "A depth-decreasing heuristic for combinational logic: or how to convert a ripple-carry adder into a carry-lookahead adder or anything in-between," in *Proceedings of the Annual Design Automation Conference (DAC)*. ACM/IEEE, 1991, pp. 361–364.

[52] R. Zimmermann, "Non-heuristic optimization and synthesis of parallel-prefix adders," in *proc. of IFIP workshop*. Citeseer, 1996.

[53] H. Zhu, C.-K. Cheng, and R. Graham, "Constructing zero-deficiency parallel prefix adder of minimum depth," in *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2005, pp. 883–888.

[54] F. S. Melo, "Convergence of q-learning: A simple proof," *Institute Of Systems and Robotics, Tech. Rep*, pp. 1–4, 2001.

[55] L. Dadda, "Some schemes for parallel multipliers," *Alta frequenza*, vol. 34, pp. 349–356, 1965.

[56] W. J. Townsend, E. E. Swartzlander Jr, and J. A. Abraham, "A comparison of dadda and wallace multiplier delays," in *Advanced signal processing algorithms, architectures, and implementations XIII*, vol. 5205. SPIE, 2003, pp. 552–560.

[57] D. J. Mankowitz, A. Michi, A. Zhernov, M. Gelmi, M. Selvi, C. Paduraru, E. Leurent, S. Iqbal, J.-B. Lespiau, A. Ahern *et al.*, "Faster sorting algorithms discovered using deep reinforcement learning," *Nature*, vol. 618, no. 7964, pp. 257–263, 2023.

[58] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[59] Z. Wang, J. Wang, Q. Zhou, B. Li, and H. Li, "Sample-efficient reinforcement learning via conservative model-based actor-critic," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, vol. 36, no. 8, 2022, pp. 8612–8620.

[60] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Proceedings of the AAAI conference on artificial intelligence (AAAI)*, vol. 32, no. 1, 2018.

[61] R. Yang, J. Wang, Z. Geng, M. Ye, S. Ji, B. Li, and F. Wu, "Learning task-relevant representations for generalization via characteristic functions of reward sequence distributions," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2022, pp. 2242–2252.

[62] J. Wang, R. Yang, Z. Geng, Z. Shi, M. Ye, Q. Zhou, S. Ji, B. Li, Y. Zhang, and F. Wu, “Generalization in visual reinforcement learning with the reward sequence distribution,” *arXiv preprint arXiv:2302.09601*, 2023.

[63] Z. Wang, T. Pan, Q. Zhou, and J. Wang, “Efficient exploration in resource-restricted reinforcement learning,” in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, vol. 37, no. 8, 2023, pp. 10 279–10 287.

[64] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.

[65] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” *arXiv preprint arXiv:1509.02971*, 2015.

[66] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, “Policy gradient methods for reinforcement learning with function approximation,” *Conference on Neural Information Processing Systems (NeurIPS)*, vol. 12, 1999.

[67] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, “Drills: Deep reinforcement learning for logic synthesis,” in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2020, pp. 581–586.

[68] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, “Exploring logic optimizations with reinforcement learning and graph convolutional network,” in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, 2020, pp. 145–150.

[69] Z. Wang, J. Wang, D. Zuo, J. Yunjie, X. Xia, Y. Ma, H. Jianye, M. Yuan, Y. Zhang, and F. Wu, “A hierarchical adaptive multi-task reinforcement learning framework for multiplier circuit design,” in *Forty-first International Conference on Machine Learning (ICML)*, 2024.

[70] Z. Wang, L. Chen, J. Wang, Y. Bai, X. Li, X. Li, M. Yuan, H. Jianye, Y. Zhang, and F. Wu, “A circuit domain generalization framework for efficient logic synthesis in chip design,” in *Forty-first International Conference on Machine Learning (ICML)*, 2024.

[71] D. Niu, Y. Dong, Z. Jin, C. Zhang, Q. Li, and C. Sun, “Ossp-pta: An online stochastic stepping policy for pta on reinforcement learning,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2023.

[72] Z. Jin, H. Pei, Y. Dong, X. Jin, X. Wu, W. W. Xing, and D. Niu, “Accelerating nonlinear dc circuit simulation with reinforcement learning,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 619–624.

[73] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi *et al.*, “A graph placement methodology for fast chip design,” *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.

[74] R. Cheng and J. Yan, “On joint learning for solving placement and routing in chip design,” *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 34, pp. 16 508–16 519, 2021.

[75] Y. Lai, Y. Mu, and P. Luo, “Maskplace: Fast chip placement via reinforced visual representation learning,” *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, pp. 24 019–24 030, 2022.

[76] Y. Lai, J. Liu, Z. Tang, B. Wang, J. Hao, and P. Luo, “Chipformer: Transferable chip placement via offline decision transformer,” in *ICML*. PMLR, 2023, pp. 18 346–18 364.

[77] Y. Shi, K. Xue, L. Song, and C. Qian, “Macro placement by wire-mask-guided black-box optimization,” *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[78] R. Zhong, J. Ye, Z. Tang, S. Kai, M. Yuan, J. Hao, and J. Yan, “Preroutgnn for timing prediction with order preserving partition: Global circuit pre-training, local delay learning and attentional cell modeling,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 15, 2024, pp. 17 087–17 095.

[79] Z. Geng, J. Wang, Z. Liu, S. Xu, Z. Tang, M. Yuan, H. Jianye, Y. Zhang, and F. Wu, “Reinforcement learning within tree search for fast macro placement,” in *Forty-first International Conference on Machine Learning (ICML)*, 2024.

[80] Z. Wang, Z. Geng, Z. Tu, J. Wang, Y. Qian, Z. Xu, Z. Liu, S. Xu, Z. Tang, S. Kai *et al.*, “Benchmarking end-to-end performance of ai-based chip placement algorithms,” *arXiv preprint arXiv:2407.15026*, 2024.

[81] H. Chen, K.-C. Hsu, W. J. Turner, P.-H. Wei, K. Zhu, D. Z. Pan, and H. Ren, "Reinforcement learning guided detailed routing for custom circuits," in *Proceedings of the 2023 International Symposium on Physical Design (ISPD)*, 2023, pp. 26–34.

[82] T. Qu, Y. Lin, Z. Lu, Y. Su, and Y. Wei, "Asynchronous reinforcement learning framework for net order exploration in detailed routing," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1815–1820.

[83] X. Du, C. Wang, R. Zhong, and J. Yan, "Hubrouter: Learning global routing via hub generation and pin-hub connection," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 36, 2024.

[84] S. A. Beheshti-Shirazi, A. Vakil, S. Manoj, I. Savidis, H. Homayoun, and A. Sasan, "A reinforced learning solution for clock skew engineering to reduce peak current and ir drop," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, 2021, pp. 181–187.

[85] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H.-S. Lee, and S. Han, "Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning," in *Proceedings of the Annual Design Automation Conference (DAC)*. ACM/IEEE, 2020, pp. 1–6.

[86] Y.-C. Lu, S. Nath, V. Khandelwal, and S. K. Lim, "Rl-sizer: Vlsi gate sizing for timing optimization using deep reinforcement learning," in *Proceedings of the Annual Design Automation Conference (DAC)*. ACM/IEEE, 2021, pp. 733–738.

[87] Z. Shi, M. Li, S. Khan, L. Wang, N. Wang, Y. Huang, and Q. Xu, "Deeptpi: Test point insertion with deep reinforcement learning," in *2022 IEEE International Test Conference (ITC)*. IEEE, 2022, pp. 194–203.

Appendix

A Method Details

A.1 Level Upper Bound

When optimizing adders in theoretical metrics, the search process is stratified based on a series of incremental level upper bounds, L. The initial bound is set to $\log_2 N$ and is incrementally raised in subsequent stages. For each new stage, the starting configuration state is the adder design with the minimum size obtained from the previous stage's search, as illustrated in Fig. 10.

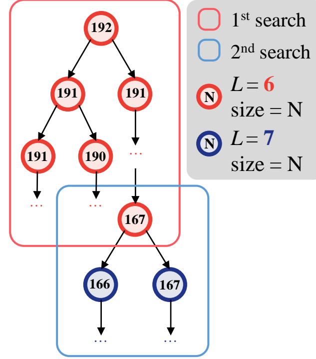

Figure 10: **Level upper bound *L* for optimizing theoretical metrics of adders.** The example is for 64-bit adder design. The search is divided into stages, and the level upper bound L increases one at a time. The initial state for each search is set to the best adder found in the last search iteration.

A.2 Two-Level Retrieval

In our two-level retrieval strategy, we implement a fast synthesis flow with minimal loss of precision. In the fast flow, we keep all other steps, including logic synthesis, clock tree synthesis, and placement, but remove the most expensive routing step. According to our efficiency test in Fig. 9, our fast synthesis flow without the routing step can speed up more than ten times. At the same time, the fast flow can still achieve highly accurate area measurements and 95% accurate delay estimations as detailed in Table 3. Thus, the fast synthesis flow can help search for as many adders as possible without losing accuracy. At the end of the first stage of two-level retrieval, we use the coordinate $(\text{area}, \text{delay})$ as the representative points for adders and compute all distances from these points to the Pareto boundary. We sort the distances in ascending order and use the K-th distance D as the threshold for selecting the adders to the second stage. As shown in Fig. 11, the K adders with the shortest distances to the Pareto boundary—constituting the top 10% in our efficiency settings—will be selected for full synthesis execution.

A.3 Synthesis Scripts

The logical and physical synthesis process for Yosys and OpenROAD is implemented using the Tcl scripting language [46]. We provide the complete Tcl scripts used for the logical synthesis in Fig. 12.

A.4 Cache, Save, and Recover Design

Our configuration allows the prefix and compressor trees to be easily saved and restored. The prefix tree is stored as an upper triangular matrix $A_{N\times N}$, where a cell (i,j) is marked with $A_{i,j} = 1$ if it exists; otherwise, $A_{i,j} = 0$ if it does not. The compressor tree is represented by a variable-length sequence $S = \{a_0, a_1, a_2, \ldots, a_{T-1}\}$ with each $a_i \in 0, 1$. Here, $a_i = 0$ represents the addition of a full adder, and $a_i = 1$ signifies the addition of a half adder. In the context of our game modeling, the matrix A and the sequence S together can completely reconstruct the prefix and compressor trees, respectively. Furthermore, both structures can be serialized into strings. These strings are then

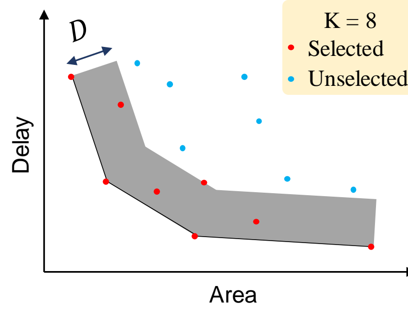

Figure 11: **Select adders in two-level retrieval.** After the first stage of the two-level retrieval process, each adder is represented by a 2D point based on its delay and area. When selecting the top *K* adders for the second stage, we sort them according to their distances from these points to the Pareto boundary. The *K* adders with the smallest distances are selected. The threshold distance, denoted as *D*, is defined by the distance of the *K*-th adder to the Pareto boundary.

```
read -sv input.v
synth -top main
flatten
opt
abc -constr ./abc_constr -fast -liberty
    library_typical.lib -D 100
write_verilog output.v
```

logic synthesis script

```
set_driving_cell BUF_X1
set_load 10.0 [all_outputs]
```

constraint file: abc_constr

Figure 12: **Scripts for logical synthesis.**

processed through a hash function to generate fixed-length values that serve as keys in our cache. This cache stores the results of previous syntheses, which helps in avoiding redundant synthesis runs.

A.5 Hyperparameter

Our hyperparameter configuration can be found in Table 5.

Table 5: Hyperparameter Configuration

Name	Description	Value
α	Weight for delay and area in Fig. 3	$0.01/0.001/0$ *
β	Sum weight in Eq. 1	0.01
c	Sum weight in Eq. 2	$10\sqrt{2}$
p	Penalty value for using half adders	0.1
γ	Discount factor in Eq. 3	0.8
ϵ	Gradient clip norm in Eq. 4	0.2
-	Batch size for PPO	64
-	Replay buffer size for PPO	$6N^2$
-	learning rate	0.001

* 0.01 for designing multipliers, 0 (ripple-carry adder as initial state) and 0.001 (others) for designing adders, where the unit of delay is ns, and the unit of area is μm^2.

A.6 Input Selection in Compressor Tree

In the compressor tree design, both half and full adders are utilized. When assigning input bits to a full or half adder, we prioritize the bits with minimal estimated delays. For instance, consider the case where we are selecting inputs for action a_0, and the available input bits have delays $\{0, 0, 0, 0, 1\}$. In this situation, the three bits with a delay of 0 would be chosen as inputs for a full adder to minimize the overall delay. The rationale behind this is that adders introduce additional delays, and our objective is to minimize the maximum delay across all bits. A more nuanced strategy is employed when inputs are fed into a full adder: the bit with the highest delay out of the three is connected to the carry input. For example, given input bits with delays $\{0, 0, 1\}$, the bit with a delay of 1 would be connected to the carry input of the full adder. This strategy is adopted because the delay from the carry input to the output bits involves only two logic gates, which is faster than the three logic gates' delay from the addend inputs to the outputs.

A.7 Strategy for Searching Multipliers

In the search process, each multiplier is tested on two boundary expected delay parameters (50 and 2×10^5). The average delay and area are then calculated from the results obtained at these two boundary conditions. The performance score for each multiplier is the weighted sum of the average delay and area. The multiplier with the highest score is selected for final evaluation.

A.8 Module Functionality Verification

Each module undergoes a rigorous testing protocol comprising 100 addition or multiplication operations to ensure the correctness and reliability of its functionality. For specific test bench details, please refer to our code.

B Supplementary Results

B.1 Adder Design

In addition to Fig. 5, we present some novel designs of the 128-bit adder discovered by our method in Fig. 13, which achieve minimal sizes under the given levels.

As illustrated in Fig. 6, we concurrently present the timeline for optimizing key performance metrics in the design of the adder. Our approach ensures sustained efficiency throughout the design process. The primary bottleneck remains in the simulation phase.

Table 6: **Time cost for Adder Design (hours).**

Method	32-bit	64 bit	128-bit
PrefixRL	1.74	4.36	11.62
PrefixRL (two-level retrieval)	1.88	3.79	8.05
ArithTreeRL	**1.71**	**3.68**	**7.34**

B.2 Multiplier Design

The design results of the 8-bit multiplier are reported in Fig. 14.

B.3 Correlation between Metrics

We investigated the correlation between theoretical and practical metrics to demonstrate the significance of optimizing theoretical metrics. For 64-bit and 128-bit adders, we sampled $6,000$ instances to assess their theoretical and practical metrics. Our results show a high correlation between two groups of metrics: level with delay and size with area, as illustrated in Fig. 15. Thus, structures with lower levels and smaller sizes are more likely to result in adders with lower delays and smaller areas.

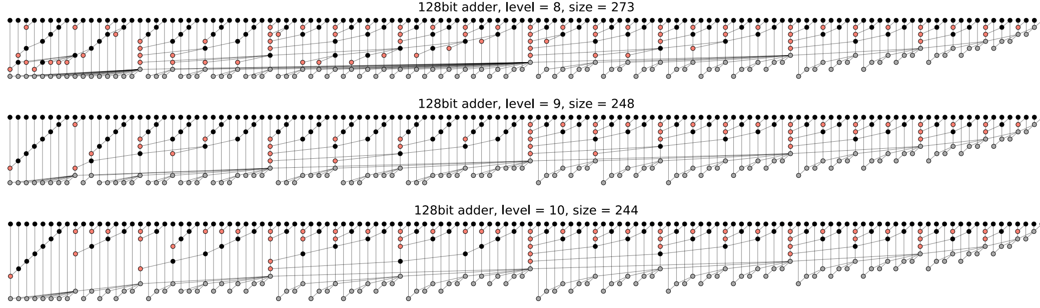

Figure 13: **Additional examples of 128-bit adders.** More structures of the 128-bit adder first discovered by our method are shown.

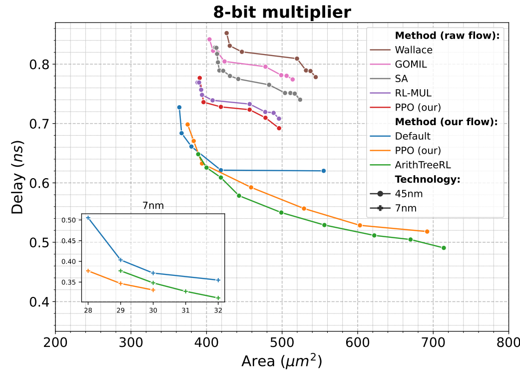

Figure 14: **Comparison of 8-bit multipliers.**

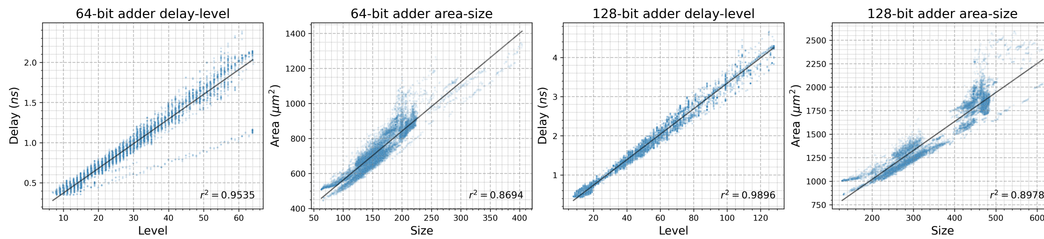

Figure 15: **Correlation of theoretical and practical metrics.** The fitted lines indicate strong correlations in delay-level and area-size. The data are derived from 6k adders for each.

B.4 Commercial Synthesis Tool Results

In addition to our tests on open-source tools, we also utilized a commercial synthesis tool, Synopsys Design Compiler 2020 [47], to demonstrate the generalizability of our approach. Table 7 presents the results of the multipliers designed by this tool. We did not incorporate timing constraints when testing the delay of the critical path. The technology library used was the Nangate 45nm library [39]. The speed of our designed multiplier still holds a significant advantage, illustrating our design approach's broad applicability and substantial potential.

B.5 Design Time

The overall design time is reported in Table 8, and the duration is within an acceptable range for the design process.

Table 7: **Results of a commercial synthesis tool.** All designs are the best-discovered multipliers with the OpenROAD tool. Corresponding Verilog codes are input into the Synopsis Design Compiler for synthesis.

Bits in multiplier	8-bit		16-bit		32-bit		64-bit	
Method	area (μm^2)	delay (ns)	area (μm^2)	delay (ns)	area (μm^2)	delay (ns)	area (μm^2)	delay (ns)
Default	314.1	1.30	1288.5	2.60	5203.2	4.88	20844.3	9.29
RL-MUL	313.9	1.47	1373.6	2.98	5757.3	5.86	23563.6	11.73
PPO	416.6	1.65	1734.9	3.19	7331.5	6.07	29545.7	11.92
ArithTreeRL	465.5	**1.20**	1866.8	**1.76**	7555.5	**2.34**	30134.1	**3.17**

Table 8: **Total design time.**

Module	Adder			Multiplier			
Bits	32	64	128	8	16	32	64
Time (h)	1.71	3.68	7.34	0.82	2.26	4.04	27.92

C Related Work

C.1 Computer Arithmetic

In the quest for high performance and low cost, computer arithmetic design plays a crucial role in computer hardware, one of the most fundamental fields in computer science [44]. Issues for study include number representation, arithmetic operations, and real arithmetic. The addition is the most common arithmetic operation and serves as a basic unit for many other operations, making it the most studied module. The most basic adder structure is the ripple carry adder, which propagates the carry bit from low to high bits. Due to its serial structure, both the delay and size are $O(N)$ for an N-bit addition. The carry look-ahead adder has been proposed to improve the delay by computing the carries for each digit simultaneously through an expanded formula. It can achieve $O(\log N)$ delay and $O(N \log N)$ size. However, due to the long internal delay of higher-valency gates used in the look-ahead adder [48], various prefix adders have been developed, including the Brent-Kung [43], Sklansky [12], Kogge-Stone [25], and Han-Carlson adders [49]. Most of these designs are variations of prefix adders. Although the minimal delay complexity is still $O(\log N)$, these adders can often have lower delays than the carry look-ahead adder because they use faster two-input logic gates [48]. Additionally, different prefix adders can strike a balance between delay and area, making them more suitable for actual hardware design.

Despite extensive research, human-engineered prefix adders encounter challenges in realizing Pareto-optimal designs. Notably, the dimensions of the Sklansky adder can be further minimized whilst maintaining its operational level, as indicated by Roy et al. [14]. Consequently, a plethora of optimization-oriented methodologies have been put forward [42, 50–52, 41, 53]. The heuristic algorithm proposed by Roy and colleagues [14] employs a bottom-up enumeration tactic, commencing with a binary adder and iteratively escalating the bit count inductively based on extant structures. To reconcile the disparity between theoretical and empirical metrics, Ma et al. [20] developed a training regimen for a predictive model to estimate actual metrics from theoretical ones. This model utilizes a Pareto active learning approach to selectively scrutinize adders, which exhibit latent high-performance metrics, for empirical validation via synthesis tools. Additionally, Geng et al. [19] have embraced graph neural networks to enhance the precision of the predictive model. However, these methodologies necessitate the pre-selection of a finite set of adders for prediction purposes, representing merely a fraction of the comprehensive feasible space and potentially overlooking superior adder configurations. The foray of reinforcement learning into the domain of adder design was pioneered by Roy et al. [17], integrating a novel approach to address design challenges. Nevertheless, the employed Q-network methodology [54] lacks exploration capabilities when applied to expansive problem domains. Moreover, it mandates complete synthesis for each adder design, a prohibitively time-intensive process when attempting to sample a vast array of adder configurations, thereby yielding suboptimal solutions.

In analog to adder design, foundational research on multipliers has also been rooted in manual methodologies. An N-bit multiplication fundamentally involves generating N partial products by deploying N^2 AND gates, which correspond to each pair of bits to be multiplied [48]. Subsequently, these partial products are accumulated to yield a $2N$-bit result. The most straightforward strategy employs N successive accumulation operations over N clock cycles, utilizing a serial approach that requires solely one adder and one register. Nevertheless, this method incurs a delay of $O(N \log N)$ with the employment of a logarithmic delay adder [44], indicating a super-linear increase relative to the bit count. To elevate computational efficiency, one may adopt a compressor tree structure to compress the partial products concurrently using full and half adders, finalizing the computation with a single $2N$-bit adder. Given that the compressor tree's height is roughly $O(\log N)$, the delay of the multiplier can be refined to $O(\log N + \log(2N)) = O(\log N)$. Although Wallace [13] and Dadda trees [55]—the predominant compressor trees—share a theoretical logarithmic delay, empirical delays vary [56], underscoring the impact of the specific tree structure on multiplier performance. Xiao et al. [16] translated the design of these trees into an integer linear problem, addressed via a combinatorial solver, yet they did not include practical metrics in their model. Zuo et al. [18] pioneered the use of reinforcement learning to refine the multiplier design. Their approach, which modifies the Wallace tree structure rather than constructing anew, narrows the state space due to the finite action sequence length. Moreover, the synthesis process remains laborious, presenting challenges in optimizing multipliers exceeding 16 bits. Furthermore, the technique has not considered the joint optimization of

the compressor and prefix trees within the multiplier, which poses a barrier to identifying a globally optimal design.

C.2 Reinforcement Learning

Reinforcement Learning (RL) has surpassed human performance in many domains, including the ancient game of Go [21], the complex strategy game StarCraft [29], optimizing sorting algorithms [57], and improving matrix multiplication techniques [30]. At its heart, RL involves training agents to make a series of decisions to achieve a goal, learning from interactions with their environment by trial and error to maximize a reward over time. There are two primary categories of RL methods: model-based and model-free. In model-based RL, agents use an explicit model of the environment to inform their decisions [58, 59]. Tools like Monte Carlo Tree Search (MCTS) [28], which simulate various future paths to aid decision-making, are often integrated with these methods. This combination has proven particularly potent for tasks requiring a long sequence of decisions. Conversely, model-free RL methods [60–63], such as DQN [64], DDPG [65], and policy gradient approaches [66], operate without an explicit model of the environment. A prominent example of model-free RL is Proximal Policy Optimization (PPO) [24]. This algorithm iteratively refines the agent's policy, optimizing a surrogate objective function to balance the need for stable policy updates with the desire for efficient exploration. This leads to high sample efficiency and reduced training times. Choosing the right RL method is crucial, as different tasks may require different approaches. By aligning the strengths of specific RL techniques with the demands of the task at hand, agents can navigate complex decision spaces with remarkable effectiveness.

Recent advancements have shown that reinforcement learning is a powerful tool at every hardware design phase, because circuit design and testing are fundamentally combinatorial optimization problems. These tasks aim to navigate a vast solution space for the most efficient configuration. Notable examples of prior achievements include logic synthesis [67–70], circuit simulation [71, 72], chip placement [73–80], chip routing [81–83], clock tree synthesis [84], circuit gate sizing [85, 86], and hardware testing [87], among others. As such, reinforcement learning's widespread success in various Electronic Design Automation (EDA) tasks highlights its remarkable capabilities and adaptability as a tool for hardware design optimization.

D Societal Impact

The advancements presented in this study have significant implications for various sectors reliant on high-performance computing and artificial intelligence. By optimizing the design of adders and multipliers, we can enhance the efficiency and reduce the physical footprint of hardware systems, leading to more powerful and compact devices. This can result in faster processing speeds and lower energy consumption, contributing to more sustainable technology practices. However, the societal impact extends beyond just technical improvements. As these optimized designs become more prevalent, they could reduce costs in producing advanced computational hardware, making high-performance computing more accessible to a wider range of industries and researchers. This democratization of technology could spur innovation and accelerate advancements in fields such as medicine, environmental science, and education. Nonetheless, the potential for job displacement in traditional hardware design roles should be considered, and efforts should be made to retrain and upskill workers to adapt to these technological advancements.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.

- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.

- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyper-parameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).

- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [NA]

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.