PrefixLLM: LLM-aided Prefix Circuit Design

***Abstract*—Prefix circuits are fundamental components in digital adders, widely used in digital systems due to their efficiency in calculating carry signals. Synthesizing prefix circuits with minimized area and delay is crucial for enhancing the performance of modern computing systems. Recently, *large language models* (*LLMs*) have demonstrated a surprising ability to perform text generation tasks. We propose *PrefixLLM*, that leverages LLMs for prefix circuit synthesis. *PrefixLLM* transforms the prefix circuit synthesis task into a structured text generation problem, termed the *Structured Prefix Circuit Representation* (*SPCR*), and introduces an iterative framework to automatically and accurately generate valid SPCRs. We further present a *design space exploration* (*DSE*) framework that uses LLMs to iteratively search for area and delay optimized prefix circuits. Compared to state-of-the-art, PrefixLLM can reduce the area by 3.70% under the same delay constraint. This work highlights the use of LLMs in the synthesis of arithmetic circuits, which can be transformed into the structured text generation.**

***Index Terms*—Prefix Circuit, Large Language Model, Structured Text Generation, Design Space Exploration**

I. Introduction

Computing carry signals efficiently is the most challenging aspect of designing digital adders [1]. In an adder, the carry signal must be computed for each bit position to compute the final sum result. For digital adders with large bit-widths, the delay in computing each carry signal is a bottleneck to the speed of the adder [2]. Prefix circuits are an effective solution to this challenge [3], [1]. Instead of computing the carry signal bit-by-bit, prefix circuits convert the carry signal computation at each bit position into computing *generate* and *propagate* signals over multiple bits for each bit position [1], [4]. Prefix circuits take advantage of the basic operator *prefix node* to compute generate and propagate signals in parallel, reducing the overall delay. Prefix circuits are classified based on their topological configurations, each providing different trade-offs between area and delay. Several classical topological configurations, such as Sklansky [5], Kogge-Stone [3], and Brent-Kung [1], are designed to balance the area and delay of prefix circuits, optimizing performance and resource utilization for specific constraints. In [6], it proposes a theoretical lower bound on the area of a prefix circuit for a given delay. Synthesizing a prefix circuit with the optimal trade-off between area and delay is still an open question due to the vast design space of topological configurations of a prefix circuit.

Various methods have been proposed for optimizing the trade-off between area and delay of a prefix circuit, with each approach offering unique advantages and limitations. These methods can be categorized into: classical prefix circuits, dynamic programming-based methods, and machine learning-based methods. Classical prefix circuits, such as Sklansky [5], Kogge-Stone [3], and Brent-Kung [1], feature highly regular structures that facilitate efficient carry signal computation. While these designs are straightforward and well-suited for general applications, they lack the flexibility to adapt to specific design constraints, such as unique area or delay requirements, limiting their applicability in specialized scenarios. The dynamic programming-based methods model prefix circuit synthesis as a divide-and-conquer problem, which can be addressed using dynamic programming [7], [8], [9]. By recursively dividing the problem into smaller sub-problems, dynamic programming explores the design space of prefix circuits systematically. However, the vast design space can result in low efficiency and high memory consumption of dynamic programming. Heuristics are used to prune the design space while preserving design quality. Despite these advances, designing effective heuristics that achieve a good trade-off between exploration efficiency and design quality is challenging.

Machine learning-based approaches model prefix circuit synthesis as a Markov Decision Process (MDP), which is then solved using *reinforcement learning* (*RL*) algorithms [10], [11]. For example, *PrefixRL* [10] uses *Deep Q Network* (*DQN*) algorithm to optimize the prefix circuit synthesis policy, while the work in [11] applies *Monte-Carlo Tree Search* (*MCTS*) algorithm. However, they require careful design of the state space and action space, as these significantly influence the policy training. This dependence on expert knowledge in RL is a significant barrier to widespread adoption and limits the generalizability of these RL-based approaches.

We propose a novel LLM-aided framework *PrefixLLM* that leverages large language models (LLMs) for synthesizing optimized prefix circuits without designing any heuristic or extra training. The main contributions are:

(1) We for the first time apply LLM to synthesize prefix circuits, which can achieves results that are comparable to state-of-the-art techniques in terms of area and delay;
(2) We transform the prefix circuit synthesis problem into a text generation task with a specific format, referred to as the *Structured Prefix Circuit Representation (SPCR)*, which aligns well with the strengths of LLMs.
(3) We propose a LLM-aided DSE framework to automatically search for prefix circuits with better area and delay.
(4) PrefixLLM framework can synthesize other types of arithmetic circuits.

In the remainder of this paper, Section II introduces some preliminaries. Section III-A introduces the LLM-aided framework for synthesizing optimized valid prefix circuits. Then, the experiment results are reported in Section IV. Finally, Section V concludes the paper.

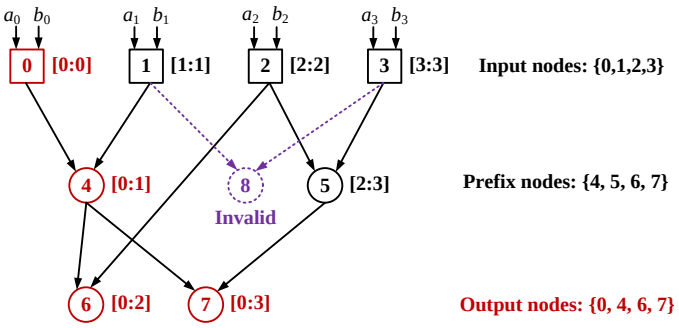


Fig. 1. A 4-bit valid prefix circuit with 4 input nodes (nodes 0-3) and 4 valid prefix nodes (nodes 4-7). Node 8 is an example invalid prefix node.

II. Preliminaries

A. Prefix circuit

A prefix circuit is a specialized type of digital circuit that can be used for generating carry signals in parallel, making it an essential component in digital adders [1], [12]. Thus, the prefix circuit plays a critical role in modern computational hardware, particularly in high-speed processors, where efficient addition of binary numbers is crucial.

In an n-bit digital adder with two operands $A = a_{n-1}a_{n-2}\cdots a_0$ and $B = b_{n-1}b_{n-2}\cdots b_0$, an n-bit prefix circuit, which can compute each carry signal c_i $(0 \le i \le n-1)$, is denoted as *Valid Prefix Circuit*. A valid prefix circuit is constructed using two types of basic operators: *input node* and *prefix node*, which are both to compute *propagate* and *generate* signals. For an n-bit prefix circuit, it contains n input nodes. The inputs of the i-th $(0 \le i \le n-1)$ input node are the two bits (a_i, b_i) from two operands A, B at each bit position. The i-th $(0 \le i \le n-1)$ input node is used to compute the propagate and generate signals for the single i-th bit. They are also generalized as the propagate and generate signals for the bit range $[i:i]$, denoted as $p_{i:i}$ and $g_{i:i}$ respectively, where:

$$p_{i:i} = a_i \oplus b_i,\ g_{i:i} = a_i b_i \tag{1}$$

For simplicity, we denote the bit range, which the propagate and generate signals of a node are computed for, as the bit range of the node. Fig. 1 illustrates an example of a 4-bit valid prefix circuit, featuring 4 input nodes (nodes 0-3) represented as squares, with the corresponding bit range displayed next to each input node. For each prefix node, it combines two groups of propagate and generate signals of two predecessor nodes for smaller ranges of bits, denoted as $(p_{i:j}, g_{i:j})$ and $(p_{k:l}, g_{k:l})$ where $i \le j \le k \le l$, to compute the propagate and generate signals for a larger range of bits. The formulations of the propagate and generate signals are shown in Eqs. 2, 3.

$$p_{i:j}p_{k:l} \tag{2}$$

$$g_{i:j} + p_{i:j}g_{k:l} \tag{3}$$

However, a prefix node in a valid prefix circuit can not combine any two groups of propagate and generate signals. A valid prefix circuit can be constructed only using *Valid Prefix Nodes*, which satisfy the condition in Eq. 4.

$$k = j + 1 \tag{4}$$

Then, the bit range of the prefix node is $[i : l]$ and the computed propagate and generate signals are $(p_{i:l}, g_{i:l})$. In Fig. 1, the valid prefix circuit uses 4 valid prefix nodes (nodes 4-7), denoted as circles. For example, node 5 combines two groups of propagate and generate signals for bit ranges $[2:2]$ and $[3:3]$, which satisfy Eq. 4, and the resulting bit range is $[2:3]$. Fig. 1, shows an example invalid prefix node 8.

According to [13], a generate signal $g_{i:0}$ is equal to the carry signal c_i. Thus, a valid prefix circuit should contain prefix nodes with all possibles ranges of bits, starting from 0. We denote such prefix nodes as *output nodes*. In Fig. 1, it has 4 output nodes $\{0, 4, 6, 7\}$, marked in red.

B. LLM for Structured Text Generation

Structured text is organized according to a predefined format, syntax, or set of rules, ensuring it is interpretable and processable by humans and machines. Examples include tables, configuration files, JSON, XML, and domain-specific formats. The generation of structured text is challenging due to the strict requirements for syntactic correctness, semantic consistency, and adherence to domain-specific constraints.

LLMs, such as *Generative Pre-trained Transformer* (*GPT*), have demonstrated remarkable capabilities in generating coherent and contextually relevant text across diverse applications. However, generating structured text presents unique challenges. Unlike freeform text, structured text demands precise formatting and logical consistency, making it prone to errors such as incomplete structures, misplaced elements, or logical contradictions. Addressing these challenges requires techniques such as *prompt engineering* and *iterative frameworks with refinement*, where the output is progressively improved through validation and feedback.

Iterative frameworks have emerged as a critical solution for structured text generation. By incorporating feedback loops, these frameworks validate the correctness of the generated text and iteratively refine it to meet the desired requirements. For instance, iterative mechanisms have been applied in translation refinement [14] and data cleaning [15], where the output quality improves with each iteration. The iterative process not only improves accuracy but also reduces the need for domain-specific expertise, enabling non-experts to achieve high-quality structured outputs. By combining the flexibility, scalability, and automation capabilities of LLMs with systematic validation and feedback mechanisms, iterative frameworks have proven to be effective in addressing the complexities of structured text generation across diverse domains.

III. LLM-aided Prefix Circuit Design

In this section, we will introduce our LLM-based framework for synthesizing a valid prefix circuit with optimized area and delay. In Section III-A, we propose a structured text-based representation of the prefix circuit called *Structured Prefix Circuit Representation (SPCR)* and transform the valid prefix circuit synthesis task into the SPCR. An iterative framework is proposed in Section III-A to generate a SPCR. In Section III-B, we propose an iterative design space exploration framework to

optimize the area and delay of the prefix circuit, i.e., iteratively synthesizing better prefix circuits.

A. LLM-aided Valid Prefix Circuit Synthesis

In this section, we will introduce our proposed iterative framework that uses LLM to synthesize valid prefix circuits. Our main idea is to transform the valid prefix circuit synthesis as a structured text generation task based on LLM. We propose a structured text representation of the prefix circuit, called *Structured Prefix Circuit Representation* (*SPCR*) as follows:

Definition 1. *The Structured Prefix Circuit Representation is a standardized text-based format for representing the nodes, connections, and computational ranges in a prefix circuit. Each line in the SPCR format corresponds to a single node in the circuit and contains the following elements:*

- ***Node Index**: A unique index for the node, denoted as an integer, e.g., 0, 1, 2, etc.*
- ***connectedNodes**: Specifies the two predecessor nodes that provide inputs to this node. The format is (left_node, right_node), where: left_node and right_node are indexes of the two nodes. Specially, for input nodes, this field is (None, None) as they have no predecessor nodes.*
- ***range**: The bit range [left_bound:right_bound] that the node computes propagate and generate signals for.*
- ***left_bound**: Start bit index of the range the node computes.*
- ***right_bound**: Ending bit index of the range node computes.*

In Fig. 2 (a), it shows an example valid 4-bit prefix circuit. The prefix circuit has 4 input nodes (nodes 0-3) and 5 prefix nodes (nodes 4-8). The bit range of each node is marked next to it. Among them, nodes 0, 4, 7, and 8 are output nodes, which are marked in red. The corresponding SPCR is shown in Fig. 2 (b), where each line corresponds to a node in the prefix circuit and the lines marked in red are output nodes. For example, node 5 has two predecessor nodes 1 and 2 denoted as connectedNodes= $(1, 2)$, and its bit range is denoted as range= $[1 : 2]$ with left_bound= 1 and right_bound= 2. For simplicity, we denote a SPCR, whose corresponding prefix circuit is valid, as a *valid SPCR*.

A prefix circuit has a one-to-one relationship with a corresponding SPCR. This relationship allows us to transform the valid prefix circuit synthesis problem into the LLM-aided valid SPCR generation problem. We generate valid SPCRs by using an LLM, which are then used to construct valid prefix circuits. As is described in Section II-B, generating the SPCR using LLM can be prone to errors. We propose an iterative framework for generating the valid SPCR based on LLM, which is finally converted into the corresponding valid prefix circuit. Fig. 3 shows the overall flow of the iterative framework to synthesize valid prefix circuit. Steps in the flow are:

- **Bit-Width Specification**: The framework begins with the specification of the bit-width n for the prefix circuit (e.g., 4-bit, 8-bit). This specification also determines that the valid prefix circuit should contain all n output nodes.
- **Partial Prefix Circuit**: The framework iteratively adds valid prefix nodes into the partial prefix circuit, until all n output nodes are generated. The partial prefix circuit is initialized with only n input nodes in this framework.
- **Checker**: validates whether the partial prefix circuit in the current iteration is valid. If it is valid, the framework terminates and output this valid prefix circuit; otherwise, it will output the lacked bit ranges of the current partial prefix circuit. In Fig. 3, the checker can determine that the partial prefix circuit in the current iteration is not valid and output the lacking bit ranges $[0 : 2]$, $[0 : 3]$.
- **SPCR Prompt**: Given the current partial prefix circuit and the lacking bit ranges, this step will generate a prompt, called *SPCR prompt*, with the format shown in Fig. 4. The initial part of the SPCR prompt contains the SPCR of the currrent partial prefix circuit and its lacked bit ranges. The next part is to ask the LLM for adding valid prefix nodes following two steps: the first is to ensure satisfying Eq. 4 and the other is to derive the corresponding bit range.
- **SPCR Response**: The SPCR response is the response to the SPCR prompt from the LLM, in which each line corresponds to the SPCR of a prefix node to be added. In Fig. 3, it shows an example SPCR response for the current partial prefix circuit, which represents a new prefix node 6 with two predecessor nodes 4 and 3, and bit range $[0 : 3]$.
- **Pruner**: The generated new prefix nodes, however, may be invalid. The pruner is used to prune all invalid prefix nodes from the SPCR response, which does not satisfy Eq. 4. Node 6 in the example SPCR response in Fig. 3 is invalid, such that it will be pruned by the pruner. After pruning, the valid prefix nodes in the SPCR response will be added into the current partial prefix circuit.

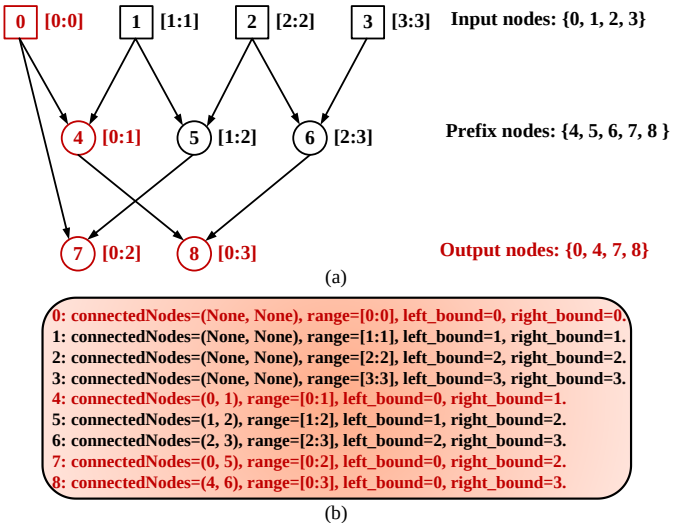


Fig. 2. (a) An example 4-bit valid prefix circuit with 4 input nodes and 5 prefix nodes; (b) the corresponding SPCR.

B. LLM-aided Design Space Exploration of Prefix Circuits

In this section, we will introduce an iterative DSE framework to optimize the area and delay of the prefix circuit. Building upon the prefix circuit synthesis framework in Section III-A, the iterative DSE framework optimize the area and

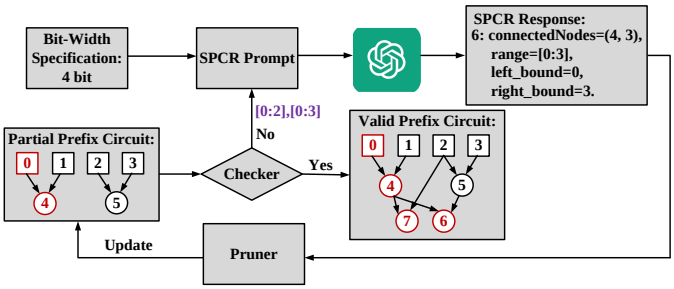


Fig. 3. Iterative framework for prefix circuit synthesis.

Generate the partial prefix circuit using the following steps:

1. **SPCR of the Current Partial Prefix Circuit**

   The initial circuit is provided, but it lacks certain nodes with specific bit ranges.

2. **Issue with Bit Ranges**

   Identify the missing bit ranges in the current partial prefix circuit.

To address the issue, follow these steps to add prefix nodes:

1. **Select two existing nodes** (represented as lines) in the partial prefix circuit:
   - The `right_bound` of the first node must equal the `left_bound` of the second node minus 1.
2. **Derive the range** for the new node:
   - The `left_bound` is taken from the first node, and the `right_bound` is taken from the second node.
   - The resulting range is defined as *`[left_bound of the first node : right_bound of the second node]`*.

Each newly added prefix node must be presented in the following format:

```
xxx: connectedNodes=(xxx, xxx), range=[xxx : xxx], left_bound=xxx, right_bound=xxx
```

Fig. 4. SPCR Prompt: the two orange frames have to be changed according to the current partial prefix circuit, and the remaining parts are fixed.

delay of prefix circuits. This framework incorporates a novel mechanism to use the *Sorted Prefix Circuit Pool* and the *DSE Prompt* for guiding the LLM toward discovering high-quality designs. The framework (see Fig. 5) operates as follows:

- **Area and Delay Evaluation**: Each valid prefix circuit is evaluated for area (as the number of nodes including input and prefix nodes) and delay (logic levels) Our DSE framework can incorporate other area and delay metrics from commercial EDA tools. These metrics quantify the trade-offs between resource efficiency and computational speed and are the basis for ranking the circuits.
- **Sorted Prefix Circuit Pool**: All prefix circuits are stored in a sorted prefix circuit pool, ranked in the descending order of area and delay. We apply the non-dominated sorting algorithm [16] to sort prefix circuits using the two metrics. Initially, when the pool is empty, we initialize it using classical prefix circuits, such as Kogge-Stone [3].
- **DSE Prompt**: As shown in Fig. 6, it combines the information from the sorted prefix circuit pool with the SPCR prompt of the current partial prefix circuit.
- **Prefix Circuit Synthesis Framework**: The core of the DSE framework is the prefix circuit synthesis framework. However, there is some difference for generating each SPCR response. Instead of querying the LLM using the SPCR prompt, the DSE framework uses the new DSE prompt to generate the SPCR response iteratively. This provides the LLM with prior prefix circuits and performance metrics and allows the LLM to detect patterns and trends from prior circuits, thereby improving its ability to generate better SPCR responses for the current partial prefix circuit with greater potential for minimizing area and delay.
- **Iteration Bound**: is the number of iterations of the DSE framework to limit the overall runtime.

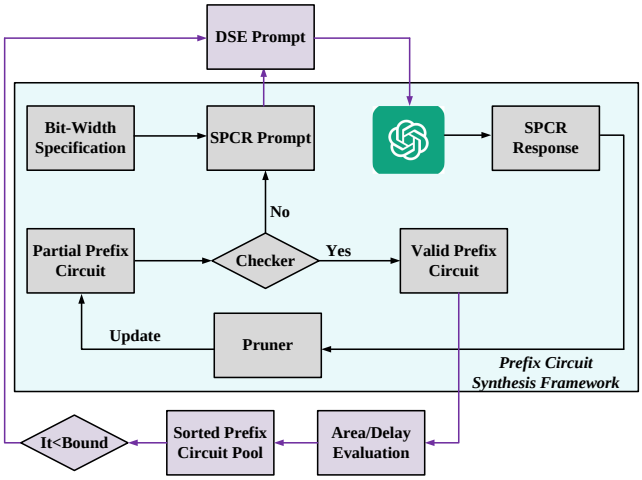


Fig. 5. Iterative DSE framework for optimizing prefix circuits.

Construct a n-bit prefix circuit with co-minimized area and delay

The following are examples of prefix circuits with their corresponding area and delay, where each line represents a node.

SPCR and area/delay of each partial prefix circuit in sorted prefix circuit pool

SPCR Prompt

SPCR of current partial prefix circuit

Missing bit ranges of the current partial prefix circuit

Fig. 6. DSE Prompt: the SPCR and the corresponding area and delay of each prefix circuit in the sorted prefix circuit pool; the SPCR prompt of the current partial prefix circuit; the remaining parts are a fixed template.

Our DSE framework can support more optimization modes:

- **Delay-Limited Area Minimization**: Minimizes prefix circuit's area under a given delay bound.
- **Area-Limited Delay Minimization**: Targets minimizing the prefix circuit's delay, while ensuring that the area remains within a upper bound.

For brevity, we do not explain these two optimization mode that enable a broader range of application-specific requirements, offering flexibility in DSE.

IV. Experimental Results

A. Experimental Setup

We implemented PrefixLLM, using Python. The framework leverages the *OpenAI o1 mini* model, accessed via the Python

OpenAI API, as the core LLM. The OpenAI o1 model was selected for its robust logical reasoning capabilities. In the DSE framework, we set the maximum number of iterations to 20 to balance computational efficiency and exploration. To control the length of the DSE prompt, we included the 10 best prefix circuits from the sorted prefix circuit pool, which were ranked based on their area and delay. In this section, we evaluated PrefixLLM in the depth-limited area minimization mode, where the objective is to minimize area given a predefined depth bound.

Experiments were conducted on 8-bit and 16-bit prefix circuits, and the results were compared with the state-of-the-art machine learning-based methods [11], and three classical prefix circuits Sklansky [5], Kogge-Stone [3] and Brent-Kung [1]. Note that the work in [11] is the most state-of-the-art open-source machine learning-based method, which can outperform the work in [8] and PrefixRL [10]. Both our PrefixLLM and the work in [11] need a initial prefix circuit to start the design space exploration, which is set as Kogge-Stone in this section. The Kogge-Stone is known for its minimal delay but incurs significant area overhead due to its exhaustive structure, the Brent-Kung offers a trade-off between area and delay with more compact designs at the cost of slightly larger delay, and the Sklansky has the minimal delay as well and smaller area than Kogge-Stone but often results in increased fanouts.

Finally, we use the *Synopsys Design Compiler* [17] to measure the actual area and delay of the synthesized prefix circuits using the *Nangate 45nm Technology* [18].

B. Comparison of Synthesized Prefix Circuits

Table I compares the area (number of nodes) for various prefix circuits under different input bit widths and delay bounds (number of logic levels). The evaluated circuits include Sklansky, Kogge-Stone, Brent-Kung, the state-of-the-art machine learning-based method [11], and the proposed PrefixLLM framework. The theoretical area bounds are also shown. To highlight the most efficient designs, the smallest area for each delay bound and bit-width is marked in bold.

For 8-bit prefix circuits, PrefixLLM synthesizes the best prefix circuit for each delay bound. At delay bound 4, PrefixLLM achieves the best prefix circuit with 20 nodes, with a reduction of area by 2 compared with [11]. Across all delay bounds, PrefixLLM achieves a total size of 86 nodes with a 2.27% reduction compared to the state-of-the-art method [11]. Additionally, PrefixLLM can outperform all of the three classical prefix circuits under the same delay.

For 16-bit circuits, PrefixLLM shows improvements over baselines. Across all delays other than 6, PrefixLLM achieves a size of 208 nodes, representing a 3.70% reduction compared to the 216 nodes of [11]. At delay bound 5, PrefixLLM synthesizes a prefix circuit with 47 nodes, outperforming Sklansky (48 nodes) and the state-of-the-art (58 nodes). At delay bounds 7, 8, and 9, PrefixLLM synthesizes circuits with the same areas as those generated by [11], close to the theoretical bound. Thus PrefixLLM balances area and delay efficiently, even for higher bit-width circuits.

Table I shows that PrefixLLM minimizes the area of prefix circuits for different delay (depth) bounds. Under strict delay bounds such as 4 for 8-bit circuits and 5 for 16-bit circuits, PrefixLLM outperforms [11].

TABLE I
COMPARISONS OF PREFIX CIRCUITS IN AREA AND DELAY.

Input Bit	Delay	Theory Area Bound [6]	Sklansky	Kogge Stone	Brent Kung	[11]	PrefixLLM
8	4	18	**20**	25	-	22	**20**
8	5	17	-	-	19	**18**	**18**
8	6	16	-	-	-	**17**	**17**
8	7	15	-	-	-	**16**	**16**
8	8	14	-	-	-	**15**	**15**
						88	**86 (2.27% ↓)**
16	5	41	48	65	-	58	**47**
16	6	40	-	-	-	**41**	44
16	7	39	-	-	42	**40**	**40**
16	8	38	-	-	-	**39**	**39**
16	9	37	-	-	-	**38**	**38**
						216	**208 (3.70% ↓)**

To validate the effectiveness of PrefixLLM, we use these prefix circuits to construct 8- and 16-bit digital adders and evaluate their area and delay by Synopsys Design Compiler using Nangate 45nm technology. The comparison under 8- and 16-bit cases are shown in Fig. 7 (a) and (b), respectively. *PrefixLLM* corresponds to our proposed work, *ML* corresponds to the machine learning-based work [11], *BK* corresponds to Brent-Kung, *KS* corresponds to Kogge-Stone, and *SK* corresponds to Sklansky. From the results, prefix circuits by PrefixLLM outperform classical prefix circuits for 8- and 16-bit adders. Compared with [11] for 8-bit adders, designs by PrefixLLM achieve the similar performance over area and delay, except the case under the delay bound 4 that the adder based on our prefix circuit can achieve smaller area. For 16-bit adders, our prefix circuits achieve the similar area and delay compared with those using prefix circuits from [11], except the case of delay bound 5. The adder based on our prefix circuit achieves more reduction in area compared to that using prefix circuit from [11] for the same delay bound.

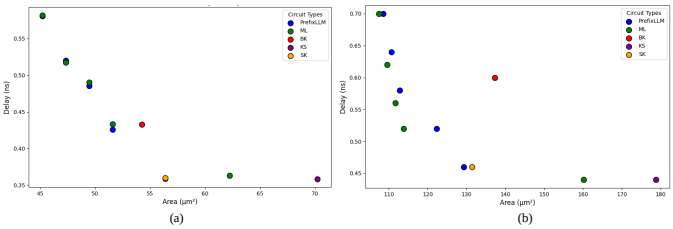
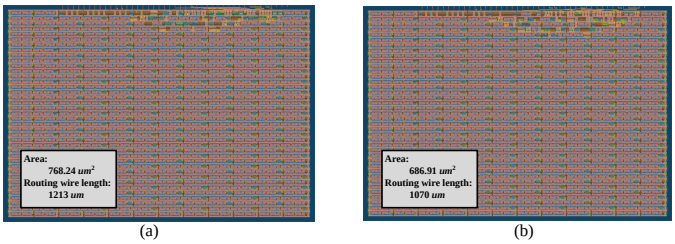


Fig. 7. Evaluation of area (μm^2) and delay (ns) of prefix circuit-based adders: (a) 8-bit; (b) 16-bit.

C. Chip Tapeout

To show the practicality and robustness of the prefix circuits, We took an 8-bit adder using the prefix circuit generated by our PrefixLLM and went through the Tiny Tapeout process [19]. The Tiny Tapeout process integrates OpenLane, which is an automated RTL to GDSII flow, and is based on the *SkyWater*



(a) (b)

Fig. 8. GDS of two 8-bit prefix circuit-adders based on (a) Kogge-Stone prefix circuit and (b) prefix circuit generated by PrefixLLM.

130nm open source technology. In the process, the designs are implemented on each Tiny Tapeout tile, which is about $160 \times 100 \mu m^2$.

In Fig. 8 (a) and (b), they show two generated GDS of a single Tiny Tapeout tile, on which two 8-bit adders are implemented using the Kogge-Stone prefix circuit and our PrefixLLM prefix circuit, respectively. In the lower left corner, the corresponding area and routing wire length of the two adders are displayed. From the result, we can see that the area of the adder based on our PrefixLLM prefix circuit ($686.91 \mu m^2$) achieves a 10.59% reduction over that of the Kogge-stone based adder ($768.24 \mu m^2$). Moreover, the routing wire length is another important metric, which has great effects over digital circuits' delay and power consumption. From Fig. 8, the routing wire length of the adder based on our PrefixLLM prefix circuit ($1070 \mu m$) is 11.79% smaller than that of the Kogge-stone based adder ($1213 \mu m$).

By leveraging the tiny tapeout process, we demonstrate the applicability of our PrefixLLM framework in generating prefix circuits that can seamlessly transition from theoretical synthesis to physical hardware realization.

V. Conclusion

By introducing the SPCR, we transformed the valid prefix circuit synthesis into a structured text generation problem, enabling efficient and automated prefix circuit synthesis using LLM. However, SPCR generation is inherently error-prone due to the strict structural and computational constraints of prefix circuits. To address this challenge, we proposed an iterative framework that ensures the automatic and correct generation of valid SPCRs using LLMs. Building on the iterative framework for valid SPCR generation, we developed an iterative DSE framework aimed at optimizing area and delay of prefix circuits. By utilizing the proposed sorted prefix circuit pool and the innovative DSE prompt, the framework guides the LLM to detect useful patterns from prior designs, facilitating the discovery of optimized designs.

References

[1] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 260–264, 1982.

[2] S. Hauck, M. M. Hosler, and T. W. Fry, "High-performance carry chains for fpga's," *TVLSI*, vol. 8, no. 2, pp. 138–147, 2000.

[3] 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*, vol. C-22, no. 8, pp. 786–793, 1973.

[4] W. Xiao, W. Qian, and W. Liu, "GOMIL: Global optimization of multiplier by integer linear programming," in *DATE*, 2021, pp. 374–379.

[5] J. Sklansky, "Conditional-sum addition logic," *IRE Transactions on Electronic Computers*, vol. EC-9, no. 2, pp. 226–231, 1960.

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

[7] J. Liu *et al.*, "An algorithmic approach for generic parallel adders," in *ICCAD*, 2003, pp. 734–740.

[8] S. Roy *et al.*, "Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures," in *DAC*, 2013, pp. 1–8.

[9] S. Lin *et al.*, "Size-optimized depth-constrained large parallel prefix circuits," in *DAC*, 2024, pp. 1–6.

[10] R. Roy *et al.*, "PrefixRL: Optimization of parallel prefix circuits using deep reinforcement learning," in *DAC*, 2021, pp. 853–858.

[11] Y. Lai *et al.*, "Scalable and effective arithmetic tree generation for adder and multiplier designs," in *NIPS*, 2024.

[12] D. Harris, "A taxonomy of parallel prefix networks," in *ACSSC*, vol. 2, 2003, pp. 2213–2217.

[13] G. Dimitrakopoulos and D. Nikolos, "High-speed parallel-prefix VLSI Ling adders," *IEEE Transcation on Computers*, vol. 54, no. 2, pp. 225–231, 2005.

[14] P. Chen *et al.*, "Iterative translation refinement with large language models," in *EAMT*, vol. 1, 2024, pp. 181–190.

[15] W. Ni *et al.*, "IterClean: An iterative data cleaning framework with large language models," in *ACM-TURC*, 2024, pp. 100–105.

[16] K. Deb *et al.*, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii," in *Parallel Problem Solving from Nature PPSN VI*, 2000, pp. 849–858.

[17] "Synopsys design compiler," 2012. [Online]. Available: http://www.synopsys.com

[18] "Nangate 45nm open cell library," 2008. [Online]. Available: http://www.nangate.com/

[19] M. D. Venn, "Tiny Tapeout: A shared silicon tapeout platform accessible to everyone." *TechRxiv*, 2024.