Size-Optimized Depth-Constrained Large Parallel Prefix Circuits

Shiju Lin
sjlin@cse.cuhk.edu.hk
The Chinese University of Hong Kong

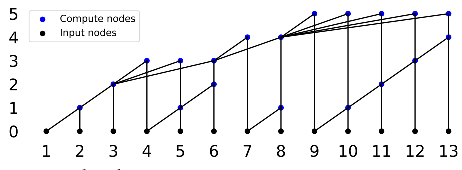
Bentian Jiang
jiangbentian@huawei.com
Huawei Design Automation Lab, Hong Kong

Weihua Sheng
sheng.weihua@huawei.com
Huawei Design Automation Lab, Hong Kong

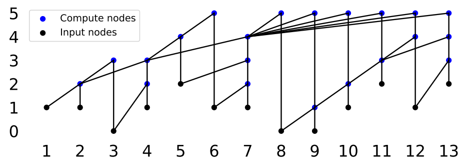
Evangeline F.Y. Young
fyyoung@cse.cuhk.edu.hk
The Chinese University of Hong Kong

ABSTRACT

Binary adders are a critical building block in integrated circuit (IC) design. In addition to the widely used 32/64/128-bit adders, large (1024/2048 bits) adders are important in applications such as cryptography. However, most current adder design methods target regular bitwidths, and cannot efficiently generate large adders with good performance. In practice, adders are often integrated into circuits such as a multiplier-accumulator (MAC), resulting in complex non-uniform input arrival times. To address these challenges, we propose a new algorithm for efficiently generating high-quality adders for non-uniform input arrival times. It is based on a novel divide-and-conquer-friendly problem formulation, and can effectively generate and maintain the most useful adder structures through dynamic programming. Experimental results show that it outperforms the current state-of-the-art methods in both quality and runtime. The adders generated by our algorithm have 2.8%, 8.3%, and 10.3% reductions in delay, area, and power, respectively, compared to those generated by a commercial synthesis tool.



(a) *size* = 19, *depth* = 5; **Input depths** = 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0



(b) *size* = 21, *depth* = 5; **Input depths** = 1, 1, 0, 1, 2, 1, 1, 0, 0, 1, 2, 1, 2

Figure 1: Our parallel prefix circuits for (a) uniform input depths, and (b) non-uniform input depths

1 INTRODUCTION

Chip functions can generally be divided into the following categories: datapath operators, memory elements, control structures, and special-purpose cells (I/O, power distribution, clock generation and distribution, and analog/RF)[10]. Common datapath operators include adders, subtractors, multipliers, comparators, shifters, counters, etc. Among these datapath operators, adders are the most fundamental and widely used. They are not only used for additions, but also an essential component of subtractors ($a-b$ is computed by $a+(-b)$), comparators (a and b are compared by comparing $a+(-b)$ and 0), and multipliers (typically designed with a compressor tree followed by an adder). The wide application of adders in datapath operators makes adder design an important and rewarding problem.

In addition to common bitwidths such as 32 and 64, which are widely used in modern computing platforms, extremely large adders are also common in certain applications, such as RSA chips in cryptography (see [11] for a 1024-bit RSA chip design). However, most existing adder design algorithms are either unable to generate adders for large bitwidths or are unable to design adders that meet the desired performance, power, and area (PPA) requirements. For example, the state-of-the-art adder design algorithm [6, 7] is search-based and can only generate adders up to 128 bits. The algorithm by Liu et al. [2] is refinement-based and cannot effectively minimize area, although it scales well to large bitwidths. New algorithms are needed to design large high-quality adders.

However, large bitwidth is not the only challenge in modern adder design. In practice, adders are often part of a complex circuit such as a multiplier-accumulator (MAC), which requires co-optimization of both units and introduces non-uniform input arrival times for adders. For example, Figure 1a and Figure 1b show two prefix circuits for adder design with and without the non-uniform input arrival times. The presence of non-uniform input arrival times greatly increases the complexity of adder design, resulting in both lower quality and lower efficiency of current adder algorithms.

To overcome these challenges, we propose a new method for designing high-quality parallel prefix circuits under non-uniform input arrival times. Parallel prefix circuits are a typical formulation of the adder design problem and has been widely adopted [1–3, 5–9, 12]. Our main contributions are summarized as follows.

- We propose a new divide-and-conquer-friendly problem formulation for the design of parallel prefix circuits, which enables the efficient divide-and-conquer methodology for the complex prefix circuit design problem.
- Based on the divide-and-conquer-friendly formulation, we develop a dynamic programming algorithm that can effectively identify and efficiently maintain the best candidate structures during divide and conquer.

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3655935

- The framework of the method is very flexible and can be naturally extended to consider non-uniform input arrival times.

We evaluate our algorithm by comparing our prefix circuits with those generated by the state-of-the-art works and comparing our prefix adders with those generated by a commercial synthesis tool. Experimental results show that our prefix circuits achieve the minimum size in every test case, and our prefix adders outperform the commercial tool in delay, area, and power by 2.8%, 8.3%, and 10.3%, respectively.

2 PRELIMINARIES

2.1 Parallel Prefix Adders

The binary addition problem is defined as follows. Given two n-bit integers a and b (a_i and b_i represent the ith bit of a and b respectively, $i = 1, ..., n$), calculate $s = a + b$. s is an integer of $n + 1$ bits, and s_{n+1} is the carry.

Parallel prefix adders are constructed based on binary signals *generate* and *propagate*, $p_{i,j}$ and $g_{i,j}$ ($1 \le i \le j \le n$). $g_{i,j}$ is 1 if and only if adding the ith to jth bits of a and b generates a carry. For example, we say that $110 + 011$ ($= 1001$) *generates* because a carry is produced. $p_{i,j}$ is 1 if and only if adding the ith to jth bits of a and b propagates a carry (i.e., generating a carry-out when a carry-in is received). For example, we say that $110 + 001$ ($= 111$) *propagates*. $g_{i,j}$ and $p_{i,j}$ can be computed by

(1) $p_{i,i} = a_i \oplus b_i$ and $g_{i,i} = a_i \cdot b_i$
(2) $p_{i,j} = p_{i,k} \cdot p_{k+1,j}$ for any integer $k \in [i, j-1]$
(3) $g_{i,j} = g_{k+1,j} + g_{i,k} \cdot p_{k+1,j}$ for any integer $k \in [i, j-1]$

Our goal is to compute $g_{1,i-1}$ for every $i \le n+1$, which can be used to compute s: $s_i = g_{1,i-1} \oplus a_i \oplus b_i$. Parallel prefix adders group the computations of $g_{i,j}$ and $p_{i,j}$ by choosing a common merging point k: $(g_{i,j}, p_{i,j}) = (g,p)_{i,j} = (g,p)_{i,k} \circ (g,p)_{k+1,j} = (g_{k+1,j} + g_{i,k} \cdot p_{k+1,j},\ p_{i,k} \cdot p_{k+1,j})$, where $\circ$ defines a binary associative operator. This allows the problem to be transformed into a parallel prefix circuit design problem given in the following subsection.

2.2 Parallel Prefix Circuits

A parallel prefix circuit takes n inputs x_i, and computes n outputs $y_i = x_1 \circ x_2 \circ ... \circ x_i$, $i = 1, ..., n$ in parallel, where $\circ$ is a binary associative operator. In the context of adders, x_i is $(g,p)_{i,i}$ and y_i is $(g,p)_{1,i}$. We use $x_{i,i} = x_i$ to represent the inputs and $x_{i,j}$ to represent the node computing $x_i \circ ... \circ x_j$. We use $x_{i,j} = x_{i,k} \circ x_{k+1,j}$ to denote that $x_{i,j}$ is computed from its left fanin $x_{i,k}$ and right fanin $x_{k+1,j}$. $depth(x_{i,j})$ denotes the depth of node $x_{i,j}$. The depth of an input node is 0 if not specified, and the depth of a non-input node $x_{i,j}$ with $x_{i,j} = x_{i,k} \circ x_{k+1,j}$ is recursively defined as $\max\{depth(x_{i,k}), depth(x_{k+1,j})\} + 1$. The *depth* and *size* of a prefix circuit are the maximum depth of its nodes and the number of nodes in it (excluding input nodes). The problems of prefix circuit design can be formally defined as:

Problem 1. $P1(N, D)$: Given two integers N and D, construct an N-bit prefix circuit of depth D with minimized size.

Problem 2. $P2(N, D, Q)$: Given integers N, D and Q_i ($1 \le i \le N$) where $depth(x_i) = Q_i$ (x_i is the input node at bit i), construct an N-input prefix circuit of depth D with minimized size.

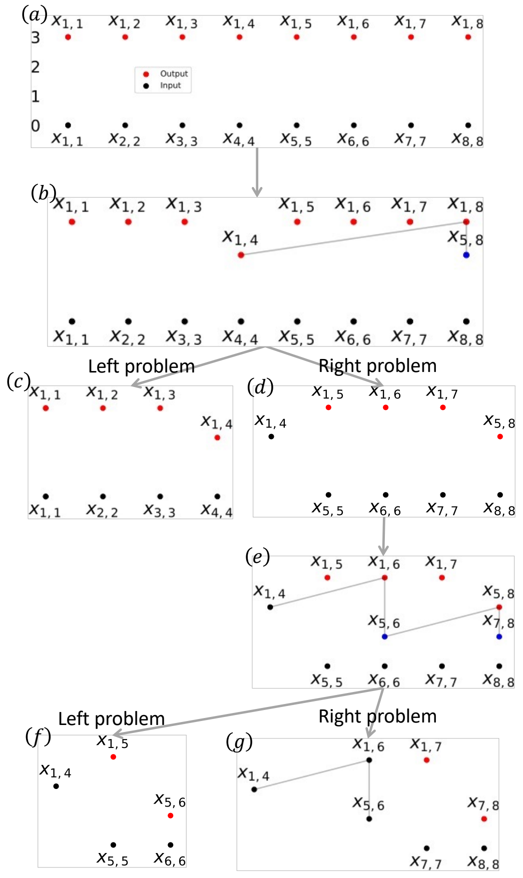

(a)
3
2
1
0
Output
Input
$x_{1,1}$ $x_{1,2}$ $x_{1,3}$ $x_{1,4}$ $x_{1,5}$ $x_{1,6}$ $x_{1,7}$ $x_{1,8}$
$x_{1,1}$ $x_{2,2}$ $x_{3,3}$ $x_{4,4}$ $x_{5,5}$ $x_{6,6}$ $x_{7,7}$ $x_{8,8}$
(b)
$x_{5,8}$
Left problem
Right problem
(c)
(d)
(e)
$x_{5,6}$
$x_{7,8}$
(f)
(g)

Figure 2: A divide and conquer example. The black and red dots are the given inputs and target outputs respectively.

Problem 2 is more general and practical than Problem 1, because adders are usually used together with other components, and Problem 2 can dynamically generate a better structure according to the input arrival times. We will first present our solution for Problem 1, followed by a natural generalization to Problem 2.

3 ALGORITHM

3.1 Motivating Example

In this subsection, we illustrate how to use divide and conquer for Problem 1 using a simple example. After explaining this example, we summarize some observations that lead to the development of a divide-and-conquer-friendly auxiliary problem formulation.

The example, $P1(8, 3)$, is to construct an 8-input prefix circuit of depth 3, as shown in Figure 2. The sub-figures in Figure 2 are numbered from (a) to (g) and are marked in the upper left corner. The black and red dots are the inputs and outputs respectively. The outputs in Figure 2(a) are placed at depth 3 to indicate the depth constraint of 3. Note that we may require an output to be computed at a smaller depth. For example, in Figure 2(b), $x_{1,4}$ is required to be computed at depth 2 (which is smaller than 3) to ensure that $x_{1,8}$ can meet the depth constraint.

Starting from Figure 2(a), we choose to compute $x_{1,8}$ by $x_{1,4} \circ x_{5,8}$ and obtain Figure 2(b), which can be divided into two subproblems shown in Figure 2(c) and (d). The left subproblem in Figure 2(c) takes inputs x_1 to x_4 and computes $x_{1,i}$ for each $i \in [1, 4]$, while the right subproblem in Figure 2(d) takes inputs $x_{1,4}$ and x_5 to x_8, and computes $x_{1,i}$ for every $i \in [5, 7]$ and $x_{5,8}$. Note that $x_{1,4}$ is the prefix result of x_1 to x_4 and is useful for computing $x_{1,i}$ for $i \in [5, 7]$. We continue with the example in Figure 2(d) and choose to compute $x_{5,8} = x_{5,6} \circ x_{7,8}$ and $x_{1,6} = x_{1,4} \circ x_{5,6}$ (Figure 2(e)), and convert the remaining problem into two problems as shown in Figure 2(f) and Figure 2(g). The left problem in Figure 2(f) takes inputs $x_{1,4}$, x_5 and x_6 to compute $x_{1,5}$ and $x_{5,6}$, and the right problem in Figure 2(g) takes inputs $x_{1,4}$, $x_{5,6}$, x_7 and x_8 to compute $x_{1,7}$ and $x_{7,8}$.

This example shows that the prefix circuit construction problem can be broken down recursively into subproblems sharing common input and output patterns. For input, there are m bits in front with non-zero and decreasing input depths followed by n bits with zero input depths. For output, we need to compute the prefix result for each of the last n bits (except the last bit, for which $x_{m+1,m+n}$ instead of $x_{1,m+n}$ should be computed) under some depth constraints. To illustrate this, for the original problem shown in Figure 2(a), $m = 0$ and $n = 8$; for the subproblem shown in Figure 2(d), $m = 1$, $n = 4$ and the input depth of the first bit, $x_{1,4}$, is 2; for the subproblem shown in Figure 2(g), $m = 2$, $n = 2$ and the input depths of the first two bits, $x_{1,4}$ and $x_{5,6}$, are 2 and 1 respectively.

3.2 Auxiliary Problem Definition

This subsection introduces an auxiliary problem formulation $AP1$ that (i) has a divide-and-conquer-friendly structure, leading to an efficient dynamic programming solution, and (ii) can help solve Problem 1, as $P1$ is a special case of $AP1$.

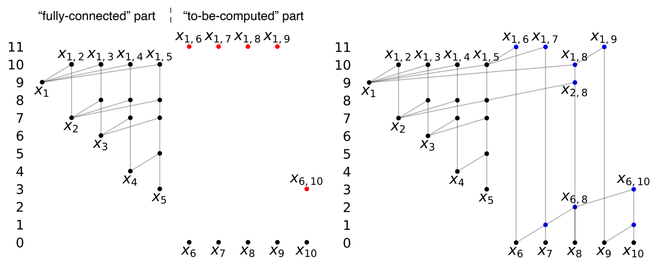
Problem $AP1(n, d, [p_0, ..., p_m])$: Given a circuit G with the following properties, find the minimum number of nodes that need to be added in order to compute $x_{m+1,m+n}$ at depth d and $x_{1,i}$ at depth p_0 for each $i \in [m + 1, m + n - 1]$.

1. G has $m + n$ inputs with $depth(x_i) = p_i$ for $i \in [1, m]$ and $depth(x_i) = 0$ for $i \in [m + 1, m + n]$.
2. $p_0 > p_1 > p_2 > ... > p_m \geq d$.
3. G has node $x_{i,j}$ computed by $x_i \circ x_{i+1,j}$ for any i and j s.t. $1 \leq i < j \leq m$.

We say that the first m bits are "fully connected" according to property 3, because $x_{i,j}$ exists for any pair of i and j satisfying $1 \leq i < j \leq m$. This "fully-connected" part carries rich information that is important to the "to-be-computed" part (i.e., the last n bits). Variable m is not explicitly included as a parameter in Problem $AP1$ because it can be inferred from the number of elements in $[p_0, ..., p_m]$. $depth(x_{i,j}) = depth(x_i) + 1$ $(1 \leq i < j \leq m)$ can also be inferred from property 3.

We use $AP1(5, 3, [11, 9, 7, 6, 4, 3])$ in Figure 3a as an example for illustration. The first $m = 5$ inputs have depths 9, 7, 6, 4, 3 and the depths of the last $n = 5$ inputs are all 0. It also has all $x_{a,b} = x_a \circ x_{a+1,b}$ for any $1 \leq a < b \leq 5$. $AP1(5, 3, [11, 9, 7, 6, 4, 3])$ is the minimum number of new nodes that need to be added in order to compute $x_{6,10}$ at depth 3, and compute $x_{1,6}$, $x_{1,7}$, $x_{1,8}$ and $x_{1,9}$ at depth 11. These target outputs are marked red in Figure 3a. The answer of $AP1(5, 3, [11, 9, 7, 6, 4, 3])$ is 9, and a feasible solution of adding 9 nodes is shown in Figure 3b.

Solutions to $P1(N, D)$ can be obtained by solving $AP1(N, D, [D])$.


"fully-connected" part
"to-be-computed" part
11
10
9
8
7
6
5
4
3
2
1
0
$x_{1,2}$ $x_{1,3}$ $x_{1,4}$ $x_{1,5}$
$x_{1,6}$ $x_{1,7}$ $x_{1,8}$ $x_{1,9}$
x_1
x_2
x_3
x_4
x_5
$x_{6,10}$
x_6 x_7 x_8 x_9 x_{10}
$x_{2,8}$
$x_{6,8}$

(a) $AP1(5, 3, [11, 9, 7, 6, 4, 3])$ **(b) Possible solution**

Figure 3: $AP1(5, 3, [11, 9, 7, 6, 4, 3]) = 9$ **(blue nodes)**

3.3 Overview of the Algorithm

To solve Problem $AP1$, we design an algorithm that first enumerates the computations of two output nodes and then divides the remaining problem into two subproblems. The enumeration is necessary for the divide and conquer and important for the final quality. We will illustrate our algorithm with an example in this subsection and present more technical details in the following subsections.

The example is $AP1(5, 3, [9, 7, 6, 4, 3])$ in Figure 4 (a), whose fully-connected and to-be-computed parts are bits 1 to 4 and bits 5 to 9 respectively. The target outputs we want to generate are $x_{1,5}$, $x_{1,6}$, $x_{1,7}$, $x_{1,8}$ and $x_{5,9}$, which are marked in red and placed at the corresponding maximum allowed depths.

First, we consider how to generate $x_{5,9}$ at depth 3. The computation of $x_{5,9}$ can be described by an integer variable $i \in [5, 8]$, which gives $x_{5,9} = x_{5,i} \circ x_{i+1,9}$. Two examples with $i = 6$ and $i = 7$ are shown in Figure 4 (b) and Figure 4 (c) respectively. This step completes target output $x_{5,9}$, but introduces two additional target outputs, $x_{5,i}$ and $x_{i+1,9}$.

Second, we consider how to derive the left subproblem. Take $i = 7$ in Figure 4 (c) as an example. We highlight the nodes that form the left subproblem in Figure 4 (d) and show the formal left subproblem in Figure 4 (e). It has the same fully-connected part as the original problem, and takes bits 5 to 7 as the to-be-computed part. The depth constraint of output node $x_{5,7}$ is 2, because it is a fanin of $x_{5,9}$ whose depth constraint is 3. The other two output nodes, $x_{1,5}$ and $x_{1,6}$, are directly inherited from the original problem and thus have an unchanged depth constraint of 9.

Third, we consider how to compute $x_{1,i}$ at depth 9. We use an integer variable $j \in \{1, 3, 4\}$ to describe the computation. The case of $j = 2$ is not considered because its resulting prefix circuits will be worse than those generated by setting $j \in \{1, 3, 4\}$. In Figure 4 (c), the three ways of computing $x_{1,7}$ with j equal to 1, 3 and 4 are shown in subfigures (f), (i) and (l) of Figure 4 respectively. The case $j = 1$ gives $x_{1,7} = x_{1,4} \circ x_{5,7}$, while $j = 3$ gives $x_{1,7} = x_1 \circ (x_2 \circ (x_{3,4} \circ x_{5,7}))$.

Lastly, we consider how to derive the right subproblem given i and j. For the examples in subfigures (f), (i) and (l), the relevant nodes for the right subproblems are highlighted in subfigures (g), (j) and (m) respectively. In Figure 4 (g), the right subproblem takes $x_{1,4}$ and $x_{5,7}$ as its fully-connected part, and bits 8 and 9 as the to-be-computed part. Nodes $x_{1,4}$, $x_{5,7}$, x_8 and x_9 in the original problem will become x_1, x_2, x_3 and x_4 in the right subproblem respectively, as shown in Figure 4 (h).

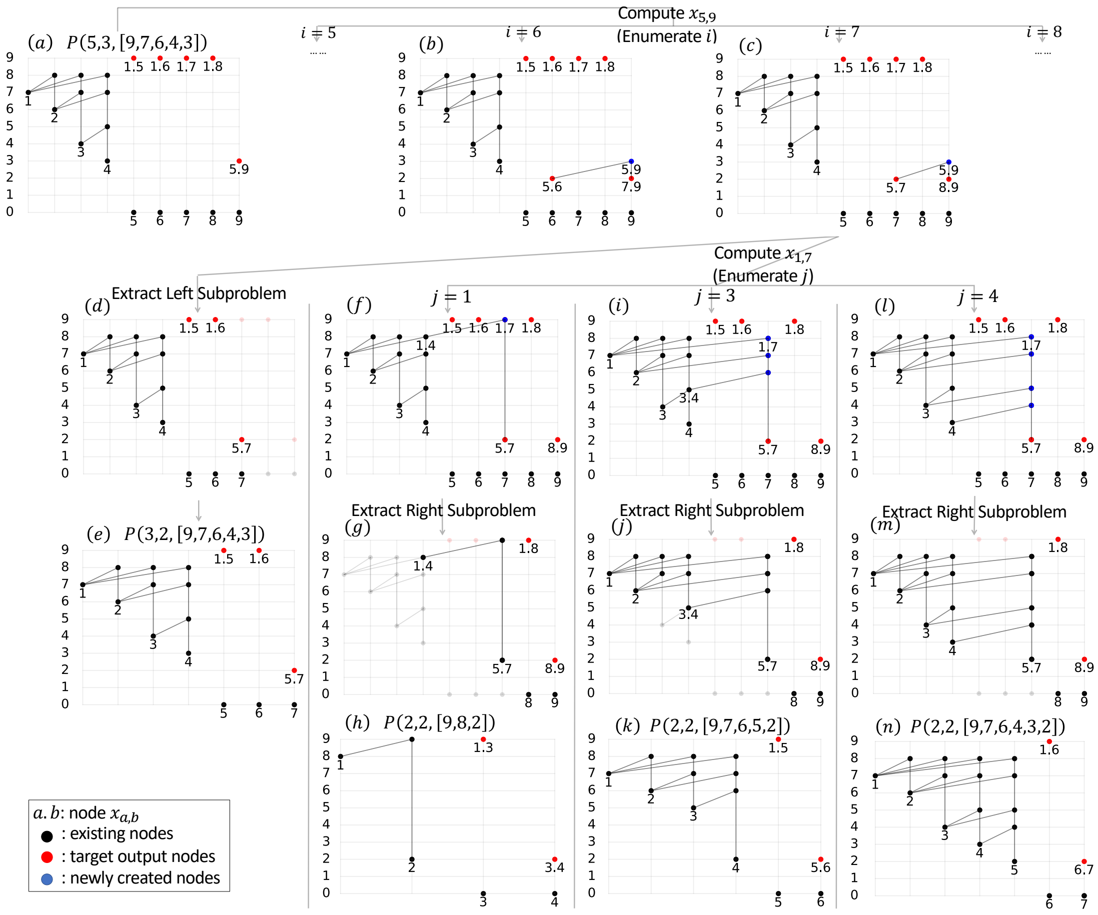


Figure 4: Flow of solving $P(5,3,[9,7,6,4,3])$

3.4 Enumeration

Enumerating i. Variable i determines how $x_{m+1,m+n}$ is computed: $x_{m+1,m+n} = x_{m+1,i} \circ x_{i+1,m+n}$. The construction of $x_{m+1,i}$ and $x_{i+1,m+n}$, which are the two fanins of $x_{m+1,m+n}$, will be done by the left and right subproblems respectively. Our algorithm enumerates i from $m+1$ to $m+n-1$, covering all cases of computing $x_{m+1,m+n}$.
Enumerating j. Variable $j \in [1,m]$ describes how $x_{1,i}$ is computed: $x_{1,i} = x_1 \circ (x_2 \circ (\ldots x_{j-1} \circ (x_{j,m} \circ x_{m+1,i})))$. This leads to the construction of $x_{j,i} = x_{j,m} \circ x_{m+1,i}$, and $x_{k,i} = x_k \circ x_{k+1,i}$ for every $k \in [1, j-1]$. We first compute $x_{j,i}$ and then $x_{j-1,i}, x_{j-2,i}, \ldots, x_{1,i}$ in order. In Figure 4 (f), j equals 1 and $x_{1,7} = x_{1,4} \circ x_{5,7}$ is computed. In Figure 4 (i), j equals 3 and we computes $x_{3,7}$, $x_{2,7}$ and $x_{1,7}$. In Figure 4 (l), j equals 4 and $x_{4,7}$, $x_{3,7}$, $x_{2,7}$ and $x_{1,7}$ are computed in order. This computing scheme ensures that a fully-connected subcircuit can be formed as the fully-connected part of the right subproblem.

However, we need to verify that this fully-connected subcircuit has depth-decreasing inputs (property 2 of $AP1$) in order to use it for the right subproblem. The necessary condition for this is for j to satisfy $j = m$ or $p_j + 1 < p_{j-1}$ (analysis omitted due to page limit). Fortunately, we are able to show that the restriction on j does not affect the quality of the algorithm, because the final prefix circuits of the unconsidered cases (j does not satisfy $j = m$ and $p_j + 1 < p_{j-1}$) are not better than those generated from the considered cases in terms of both size and depth, according to Theorem 1. Proof of Theorem 1 is not provided here due to page limit.

Theorem 1. *Given problem $AP1(n,d,[p_0,\ldots,p_m])$, let $S = \{m\} \cup \{j \in [1,m-1] : p_j + 1 < p_{j-1}\}$ and $T = \{1,\ldots,m\} - S$. For any integer $u \in T$, let v be the smallest integer in S s.t. $v > u$. The final prefix circuits generated by using $j = v$ are better than or as good as those generated by $j = u$ in terms of depth and size.*

3.5 Divide and Conquer

Given problem $AP1(n,d,[p_0,\ldots,p_m])$, the goal is to compute $x_{1,t}$ at depth p_0 for every $t \in [m+1, m+n-1]$ and to compute $x_{m+1,m+n}$ at depth d, according to the definition of $AP1$. The enumeration of i computes one target output $x_{m+1,m+n}$ at depth d, and introduces two more target outputs $x_{m+1,i}$ and $x_{i+1,m+n}$ at depth $d-1$. The enumeration of j computes $x_{1,i}$. After the enumeration step, the target outputs become $x_{m+1,i}$, $x_{i+1,m+n}$ and $x_{1,t}$ for every $t \in [m+1, i-1] \cup [i+1, m+n-1]$, which will be handled by the subproblems. The left subproblem computes $x_{m+1,i}$ and $x_{1,t}$ for $t \in [m+1, i-1]$, while the right subproblem computes $x_{i+1,m+n}$ and $x_{1,t}$ for $t \in [i+

Table 1: Comparison of Parallel Prefix Circuit Size

Bitwidth	32			64			128			256				512				1024				Average
Depth	5	6	7	6	7	8	7	8	9	8	9	10	11	9	10	11	12	10	11	12	13	Time (s)
[12]	N/A	**56**	**55**	N/A	N/A	**118**	N/A	N/A	**245**	N/A	N/A	N/A	**499**	N/A	N/A	N/A	**1010**	N/A				N/A
[6, 7]	**74**	**56**	**55**	**167**	126	**118**	**364**	276	250	N/A				N/A				N/A				>100
Ours	**74**	**56**	**55**	**167**	**125**	**118**	**364**	**272**	**245**	**773**	**575**	**504**	**499**	**1614**	**1190**	**1033**	**1010**	**3327**	**2437**	**2105**	**2035**	**<1**

$1, m+n-1]$. According to the target outputs of the left subproblem, its to-be-computed part is the $(m+1)$-th to the i-th bits of the original problem. Its fully-connected part is the same as that of the original problem, which provides information of the first m bits. Precisely, the left subproblem is $AP1(i-m, d-1, [p_0, ..., p_m])$. $AP1(3, 2, [9, 7, 6, 4, 3])$ in Figure 4 (e) is the left subproblem for $i = 7$.

The right subproblem is more complex. According to the problem division described above, the to-be-computed part of the right subproblem is the $(i+1)$-th to the $(m+n)$-th bits of the original problem. As we want to compute $x_{1,t}$ for every $t \in [i+1, m+n-1]$, the information of the first i bits of the original problem is important. We use the first m bits of the original problem, which is fully connected, and $x_{m+1,i}$ to form the fully-connected part of the right subproblem. We have to ensure that this part is fully connected, which is the third property of problem P described in Section 3.2. The way we construct this fully-connected part for the right subproblem is related to j. We use $x_1, x_2, ..., x_{j-1}, x_{j,m}$ and $x_{m+1,i}$ as input nodes. We have shown in the last subsection that these input nodes have decreasing input depths. Next, we verify that these inputs are fully connected. There are in total $j+1$ input nodes, and the first j nodes are already fully connected because they are from the fully-connected part of the original problem. It remains to show that the last node $x_{m+1,i}$ has connection with each of the first j nodes. This is exactly what the enumeration of j does. $x_{j,i} = x_{j,m} \circ x_{m+1,i}$ is first computed, followed by $x_{k,i} = x_k \circ x_{k+1,i}$ where k runs from $j-1$ to 1. Precisely, the right subproblem is $AP1(m+n-i, d-1, [p_0, ..., p_{j-1}, depth(x_{j,m}), d-1])$.

3.6 Time and Space Complexity

We analyze the algorithm with $D \leq 2\lceil \log_2 N \rceil$, as optimal solutions of $P1(N, D)$ for $D > 2\lceil \log_2 N \rceil$ can be obtained from [12].

Theorem 2. *The time and space complexities of our algorithm for $P1(N, D)$ with $D \leq 2\lceil \log_2 N \rceil$ are $O(N^4 \log_2 N)$ and $O(N^3)$.*

The proof is not provided due to page limit. The key of analysis is to note that $[p_0, ..., p_m]$ in $AP1(n, d, [p_0, ..., p_m])$ only has $O(2^D)$ (which equals $O(N^2)$ when $D = 2\lceil \log_2 N \rceil$) combinations, given that $p_0 > p_1 > ... > p_m >= d$ (property 2 of $AP1$). In practice, it is common to require depth to be the minimum possible (i.e., $D = \lceil \log_2 N \rceil$) for timing closure. The time and space complexities in this scenario are reduced to $O(N^3 \log_2 N)$ and $O(N^2)$ respectively.

3.7 Bitwise Input Depth Constraints

This subsection addresses Problem 2 defined in Section 2.2. Compared to Problem 1, N additional integers Q_i representing the input depths are given in Problem 2. Previously, we defined $AP1$ to help solve $P1$. Similarly, we define $AP2$ as follows to help solve $P2$.

Problem $AP2(l, r, d, [p_0, ..., p_m])$: Given a circuit G with the following properties, find the minimum number of nodes that need to be added in order to compute $x_{m+1,m+r-l+1}$ at depth d and $x_{1,i}$ at depth p_0 for every $i \in [m+1, m+r-l]$.

(1) G has $m+r-l+1$ inputs with $depth(x_i) = p_i$ for $i \in [1, m]$ and $depth(x_i) = Q_{l+i-m-1}$ for $i \in [m+1, m+r-l+1]$.
(2) $p_0 > p_1 > p_2 > ... > p_m \geq d$.
(3) G has node $x_{i,j}$ computed by $x_i \circ x_{i+1,j}$ for all $1 \leq i < j \leq m$.

$P2(N, D, Q)$ can be solved by computing $AP2(1, N, D, [D])$. Problem $AP2(l, r, d, [p_0, ..., p_m])$ record by l and r the bit range of its to-be-computed part in the original $P2$ (i.e., the to-be-computed part corresponds to bits l to r of the original $P2$). Their input depths can thus be obtained, which are Q_l to Q_r. Suppose we are solving $P2(N = 12, D = 9, Q = [3, 2, 1, 2, 0, 1, 1, 0, 0, 1, 1, 2])$, for example, and $AP2(l = 7, r = 11, 3, [9, 7, 6, 4, 3])$ is one of the many $AP2$ instances that need to be computed. In this example, $l = 7$ and $r = 11$ indicate that bits 5 to 9 (to-be-computed part) of $AP2$ are bits 7 to 11 of the original Problem 2 we are solving. Therefore, the depths of x_5, x_6, x_7, x_8, x_9 are $Q_7, Q_8, Q_9, Q_{10}, Q_{11} = 1, 0, 0, 1, 1$.

$AP2(l, r, d, [p_0, ..., p_m])$ can be computed using the same algorithm as computing $AP1$, with consideration of the bit range l and r. Precisely, the left and right subproblems of $AP2(l, r, d, [p_0, ..., p_m])$ are $AP2(l, i-m+l-1, d-1, [p_0, ..., p_m])$ and $AP2(i-m+l, r, d-1, [p_0, ..., p_{j-1}, depth(x_{j,m}), d])$.

4 EXPERIMENTAL RESULTS

The experiments were conducted on a Linux machine with 32GB memory and a 2.6GHz CPU. Our adders are functionally correct verified by a commercial tool.

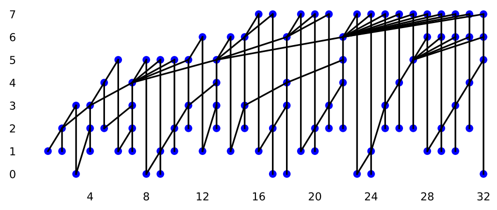
4.1 Size Comparisons of Parallel Prefix Circuits

Table 1 shows parallel prefix circuit size comparisons under different bitwidth and depth requirements between our algorithm and the previous state-of-the-art works [12] [6, 7]. The results of [12] are taken from the paper, which is complete. Since [6] is a preliminary version of [7], we combine their results by taking the better ones[1] and show them in one line in Table 1. An "N/A" entry in Table 1 indicates that the algorithm cannot generate a prefix circuit under that constraint. Notably, our algorithm is able to generate the minimum-size results for all cases, including many cases where the previous works fail. The available results (i.e., non-"N/A" entries) of [12] are claimed to be optimal, and our work does not show any suboptimality experimentally in the comparison with it.

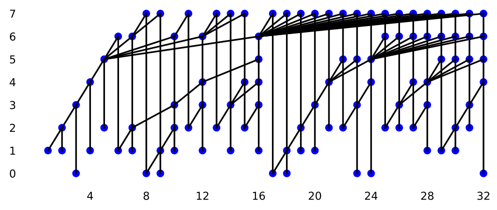
4.2 Non-Uniform Input Depths

It is difficult to make a comprehensive comparison for non-uniform input depths because there are too many possible input depths. We choose to show the prefix circuits generated by our method

[1]The results of [6] and [7] are obtained by running the provided source code and from the paper, respectively.



(a) Size=60, depth=7; prefix circuit by our method



(b) Size=70, depth=7; prefix circuit by [6]

Figure 5: Comparison under non-uniform input depths between (a) our method and (b) the state-of-the-art method [6]

and [6] in Figure 5. The input depths are obtained randomly by using "rand()%3" for bits 1-32 in order with random seed = 0 in C++. The prefix circuit sizes of our method and [6] are 60 and 70 respectively. In addition to the numbers, we can also visually see why our method is better in Figure 5 by focusing on bits 23-24, which arrive very early at depth 0. Our method takes advantage of this and lets them immediately do some computations at depth 1, while the work [6] does not take advantage of this point, and lets them wait until depths 3 and 4 for the first computation. This shows that our algorithm has a better awareness of the input depths.

4.3 Adder Comparison

To demonstrate the usefulness of our method in practice, we further compare the adders generated by our method to those generated by a commercial synthesis tool (CST). We implement our adder architectures in a standard way using alternating OAI/AOI gates [10], and use CST for one iteration of size-only (structure-preserving) synthesis followed by two iterations of incremental synthesis. The reference CST adders are obtained by synthesizing $y = a + b$ in the same flow. The NanGate 15nm Open Cell Library[4] (typical, NLDM) is used for synthesis and the target delay is set to 0.

Table 2 shows the delay, area and power results of our adders and CST adders. For each bitwidth, we explore 100 adders generated by our method and show the results of the best-delay adders in Table 2. Our method can effectively reduce the average delay, area and power of 64/128/256/512/1024-bit adders by 2.8%, 8.3% and 10.3% respectively. We further show the area-delay results of all 100 adders generated by our method in Figure 6. It clearly shows that our method has consistently and significantly better area than CST, and has better delay in most cases.

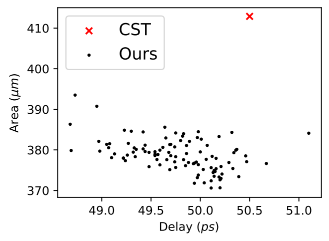
5 CONCLUSIONS

This paper presents a novel divide-and-conquer-friendly problem formulation for designing parallel prefix circuits, followed by an efficient dynamic programming approach to the problem. The method is fast (polynomial run time), flexible (able to consider non-uniform input depths), and of high quality (outperforming state-of-the-art academic and commercial tools). Notably, this work can generate high-performance large-scale (thousands of bits) adders, which are rarely studied in the literature but have important applications like RSA chips in cryptography.

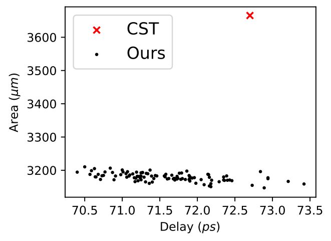
Table 2: PPA Comparison of Our Method and CST

Bitwidth	Delay (ps)			Area (μm^2)			Power (mW)		
	CST	Ours	Imp	CST	Ours	Imp	CST	Ours	Imp
64	41.4	40.6	1.8%	197.1	190.9	3.1%	129.1	122.3	5.3%
128	50.5	48.7	3.6%	412.9	386.3	6.4%	265.9	235.5	11.4%
256	56.7	55.0	3.0%	851.8	795.7	6.6%	536.7	494.3	7.9%
512	64.7	63.1	2.5%	1799.3	1575.4	12.4%	1098.7	939.0	14.5%
1024	72.7	70.4	3.2%	3665.4	3194.4	12.9%	2191.8	1918.6	12.5%
Average			**2.8%**			**8.3%**			**10.3%**

Imp: improvement computed by $(CST - Ours)/CST$



(a) 128 bits



(b) 1024 bits

Figure 6: Area-delay of adders by CST and this work

REFERENCES

[1] Hao Geng, Yuzhe Ma, Qi Xu, Jin Miao, Subhendu Roy, and Bei Yu. 2022. High-Speed Adder Design Space Exploration via Graph Neural Processes. *IEEE TCAD* (2022).
[2] Jianhua Liu, Shuo Zhou, Haikun Zhu, and Chung-Kuan Cheng. 2003. An Algorithmic Approach for Generic Parallel Adders. In *Proc. IEEE/ACM ICCAD.*
[3] Yuzhe Ma, Subhendu Roy, Jin Miao, Jiamin Chen, and Bei Yu. 2019. Cross-Layer Optimization for High Speed Adders: A Pareto Driven Machine Learning Approach. *IEEE TCAD* 38, 12 (2019), 2298–2311.
[4] Mayler Martins, Jody Maick Matos, Renato P. Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. 2015. Open Cell Library in 15nm FreePDK Technology. In *Proc. ACM ISPD.*
[5] Rajarshi Roy, Jonathan Raiman, Neel Kant, Ilyas Elkin, Robert Kirby, Michael Siu, Stuart Oberman, Saad Godil, and Bryan Catanzaro. 2021. PrefixRL: Optimization of Parallel Prefix Circuits Using Deep Reinforcement Learning. In *Proc. ACM/IEEE DAC.*
[6] Subhendu Roy, Mihir Choudhury, Ruchir Puri, and David Z. Pan. 2013. Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures. In *Proc. ACM/IEEE DAC.*
[7] Subhendu Roy, Mihir Choudhury, Ruchir Puri, and David Z. Pan. 2014. Towards Optimal Performance-Area Trade-Off in Adders by Synthesis of Parallel Prefix Structures. *IEEE TCAD* (2014).
[8] Subhendu Roy, Mihir Choudhury, Ruchir Puri, and David Z. Pan. 2016. Polynomial Time Algorithm for Area and Power Efficient Adder Synthesis in High-Performance Designs. *IEEE TCAD* (2016).
[9] Subhendu Roy, Yuzhe Ma, Jin Miao, and Bei Yu. 2017. A learning bridge from architectural synthesis to physical design for exploring power efficient high-performance adders. In *Proc. IEEE/ACM ISLPED.*
[10] Neil Weste and David Harris. 2010. *CMOS VLSI Design: A Circuits and Systems Perspective* (4th ed.). Addison-Wesley Publishing Company, USA.
[11] Chingwei Yeh, En-Feng Hsu, Kai-Wen Cheng, Jinn-Shyan Wang, and Nai-Jen Chang. 2006. An 830mW, 586kbps 1024-bit RSA chip design. In *Pro. DATE*, Vol. 2.
[12] Haikun Zhu, Chung-Kuan Cheng, and Ronald Graham. 2006. On the Construction of Zero-Deficiency Parallel Prefix Circuits with Minimum Depth. *ACM Trans. Des. Autom. Electron. Syst.* 11, 2 (apr 2006), 387–409.