A Resource Sharing Approach for Logic Synthesis Based on Monte Carlo Tree Search

Qirui Yang*, Kang Zhao†, Wenjian Yu*, Yun Shao‡, Yong Xiao‡
*Department of Computer Science and Technology, Tsinghua University, Beijing, China
†School of Integrated Circuits, Beijing University of Posts and Telecommunications, Beijing, China
‡Shenzhen Giga Design Automation Co., Ltd., Shenzhen, Guangdong, China
Email: yqr19@mails.tsinghua.edu.cn, zhaokang@bupt.edu.cn, yu-wj@tsinghua.edu.cn, {yshao, yxiao}@giga-da.com

2023 International Symposium of Electronics Design Automation (ISEDA) | 979-8-3503-0451-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISEDA59274.2023.10218521

***Abstract*—How to reduce the area during multiple logic synthesis algorithms is a key problem. However, it is very hard because there is very small optimization space left for the logic synthesis. To settle this issue, this paper integrates the resource sharing technique into logic synthesis. Resource sharing is a key area-reduction approach in high-level synthesis (HLS), which is an NP-hard problem. Until now, most existing algorithms for resource sharing is greedy approach, which cannot obtain good result especially for the large circuits with numerous function units (FU). This paper proposes a novel algorithm based on Monte Carlo tree search (MCTS) for resource sharing, and uses it to reduce the resources at the early stage of the logic synthesis. The experimental results show the better area with short and stable runtime (0.5 second for MCTS). Comparing to the traditional greedy approach, the proposed algorithm can reduce the number of MUXs' inputs by about 44% and reduce the area up to 20%.**

***Index Terms*—Monte Carlo tree search, Resource Sharing, Logic Synthesis**

I. Introduction

Logic synthesis plays a central role in the design automation of VLSI circuits. With the help of logic synthesis tools, a hardware designer is freed from tedious and error-prone low-level circuit design, and can focus on architectural and algorithmic level issues. Logic synthesis generally includes two stages: technology-independent circuit optimization, followed by technology-dependent optimization. In the former, the circuit is optimized in a manner that is agnostic to the target technology, whether that be a standard-cell ASIC, FPGA, or other IC media. During this stage, area is a important metrics.

However, it is very hard to obtain a best solution to achieve the lowest resource utilization. There are two reasons. First, there is very small space left for logic synthesis step because RTL design has been determined by the high-level synthesis (HLS) stage, which is the upper reaches of logic synthesis. Second, it is a NP-hard problem to achieve a good trade-off between latency and area. To settle this issue and obtain a good area result, this paper considers to utilize the resource sharing technique in HLS and integrates it with logic synthesis. Resource sharing problem consists in minimizing the number of registers and functional units (FU) used in a design. One of the main objectives of the problem is to perform sharing between conditional branches.

In previous works, Wakabayashi and Tanaka proposed a global scheduling algorithm based on condition vectors to optimize across the basic blocks and all possible execution paths [1]. And based on this work, Kim also proposed a scheduling resource sharing algorithm by analyzing and transforming the data-flow graph (DFG) to share resources between the conditional branches [2]. Other researchers like Memik also work on the DFG and proposed a global resource sharing algorithm [3]. These algorithms could obtain better timing performance through resource sharing, without considering the circuit consumption. In addition, Raje proposed another generalized resource sharing method based on an area-cost function [5]. Since minimizing the multiplexers inputs is one of the main goals of resource sharing, Raje also proposed a technology library to permute inputs to minimize the multiplexers inputs. However, this technology is unlikely to find the optimal solution because minimizing the number of inputs is an NP-hard problem.

Another and more general method for solving the resource sharing problem is based on the Greedy Input Permutations on Commutative Operations Algorithm (For simplicity, we call it Greedy algorithm) in [5]. It often produces simple and relatively stable results, but still can fail to find the optimal solution. In this work, inspired by the recent applications of AI methods to logic synthesis area, such as in selecting the architectures of adder and multiplier [6], we propose an AI-based heuristic algorithm to reduce the area consumption of circuits and better solve the resource sharing problem.

Generally, resource sharing can be done either on the netlist or on the source code represented by hardware description language (HDL) such as VHDL, Verilog and so on. We chooses to accomplish resource sharing through the latter to convert the source code to a resource shared form. It means this algorithm can be easily transplanted to different EDA tools because it is a source-to-source algorithm. Specifically, the heuristic algorithm we proposed is based on Monte Carlo tree search (MCTS), which solve the resource sharing problem with upper confidence bound (UCB).

This work is partially supported by National Key R&D Program of China (2022YFB2901100), and NSFC (No. 62090025).

II. Problem Formulation

In this section, the resource sharing problem is formulated through the use of a simple source pseudocode, followed by a matrix description. Monte Carlo tree search (MCTS) is applied to the problem using the source-to-source method. A simple verilog-style pseudocode example is presented in Fig. 1.

```
if i == 0: // Conditional Expression
  out = a + b // Conditional Branch 1
else if i == 1:
  out = b + c // Conditional Branch 2
else
  out = a + c // Conditional Branch 3
```

Fig. 1. A simple pseudocode example

Codes in a basic block controlled by the same conditional statement is referred to as a *conditional branch*. And resource sharing occurs only between conditional branches whose conditional statement are mutually exclusive. Since these branches are not accessed at the same time, multiple conditions can share one or several functional units (FUs). This is known as *timing irrelevant*. If several conditional branches are timing relevant, it means that the FUs in those conditional branches must work simultaneously in one cycle. With the exception of special corner cases where a FU can perform various computation several times in parallel, it is concluded that the same FUs cannot be shared because a FU can only provide computational services for one conditional branch at a time.

Examining the pseudocode in Fig. 1, it can be seen that all three conditional branches are timing irrelevant and require an addition operation. As a result, these branches can share one adder instead of each having an independent adder. To accomplish this conversion process, the input to the adder must be selected according to different conditional branches.

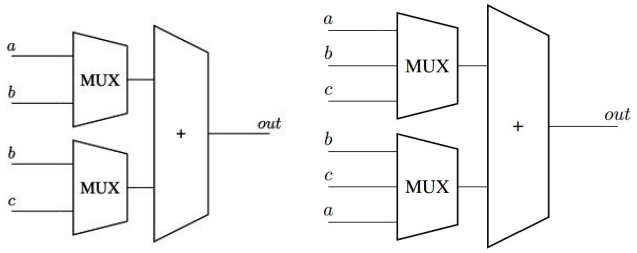
The codes in Fig. 2 show two different input selection schemes. The two schemes have the same resource sharing effect because both schemes use two multiplexers and one adder. However, as shown in Fig. 3, the area of the circuits are different because one uses two-input multiplexers while the other one uses three-input multiplexers. Since the area of the multiplexer is positively related to the number of its inputs, the number of the multiplexer inputs largely is positively correlated with the area. During the experiment, such designs are compiled by Design Compiler, and the results prove our statements above.

```
add_0 = a if i != 1 else b
add_1 = b if i == 0 else c
out = add_0 + add_1
```

```
add_0 = a if i == 0 else (b if i == 1 else c)
add_1 = b if i == 0 else (c if i == 1 else a)
out = add_0 + add_1
```

Fig. 2. Schemes using two-input/three-input multiplexers

The analysis above leads to the conclusion that the minimum number of FUs used is fixed and the area can be minimized by reducing the number of MUX inputs. As a result, the problem shifts from minimizing the area to minimizing



(a) Two-input MUX implementation (b) Three-input MUX implementation

Fig. 3. Circuits using two-input/three-input MUX for the example shown in Fig. 2

both the number of MUXs and their inputs. Assuming a case with n conditional branches, m shareable FUs, and p MUXs, minimizing the circuit area is equivalent to minimizing the sum of FUs and MUXs area. The area of a MUX can be described as $F(Inputs_{MUX})$, where F is the MUX area estimation function positively correlated with the number of its inputs. This function can be found in many technology libraries, as mentioned in [7]. In summary, the total area of circuits can be expressed as the $Area_{tot}$ in (1).

$$Area_{tot} = m \times (Area_{FU}) + \sum_{i=0}^{p} F(Input_{MUX}) \quad (1)$$

Since the number of FUs is fixed, the $Area_{FU}$ in (1) is also fixed. Therefore, the resource sharing problem is equivalent to minimizing $F(Input_{MUX})$.

Let's review the code in Fig. 1. Suppose a matrix A is built up. Its columns stand for the inputs (a, b, and c) and its rows stand for the three conditional branches. Then the element A_{ij} in the matrix represents how many inputs j does conditional branch i has. Besides, for inputs with quantities greater than 1, we treat these inputs as distinct for sharing purposes (e.g. $2a+b+c = a_1+a_2+b+c$). The matrix A can be described as in Fig. 4

Then let's focus on the matrix's columns, the vector-or and vector-and operations are introduced for the algorithm description. And the vector-or operations on the matrix's columns are actually selecting the inputs of MUXs. For instance, selecting column a and b in Fig. 4 for vector-or operations means choosing a and b as two inputs of the MUX. And the result of the column a vector-or b is a vector with all elements equal to 1. This means all conditional branches can spare at least one input, either a or b, for the MUX. As a result, the implementation of a two-input MUX is possible. This situation is defined as *covered*. If the vector-or result does not cover all branches, it must be added to the matrix as a new column and repeat the vector-or process until covered.

$$\begin{array}{l} \\ conditional\ branch\ 1 \\ conditional\ branch\ 2 \\ conditional\ branch\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \end{array}$$

Fig. 4. The matrix form for Fig. 1

$$\begin{pmatrix}1\\0\\1\end{pmatrix} \wedge \begin{pmatrix}1\\1\\0\end{pmatrix} = \begin{pmatrix}1\\0\\0\end{pmatrix} \qquad \begin{pmatrix}1&1&0\\0&1&1\\1&0&1\end{pmatrix} \rightarrow \begin{pmatrix}1&0\\0&1\\0&1\end{pmatrix}$$

(a) column a vector-and column b (b) column a vector-share column b

Fig. 5. The results of vector-and and vector-share operations for the example shown in Fig.4

It may also be found that when columns a and b in Fig. 4 are selected for conditional branch 1, both input a and b can be provided. In this case, one input must be chosen and the other one kept for future selection. For convenience, the column further to the left is usually selected, which means to provide input a and keeps input b in this case. The vector-and operation is then performed to obtain the remaining input for each branch. The result of the column a vector-and column b is shown in Fig. 5(a), and indicates that conditional branch 1 provides input a for the MUX and keeps input B for future selections.

In this process, two columns are selected from the matrix and subjected to vector-or and vector-and operations. The results are only added to the matrix as new columns when the result of the vector-or operation has elements equal to zero or the result of the vector-and operation has an element equal to one. This operation, which includes a vector-or and a vector-and, is defined as a single *vector-share* operation (see example shown in Fig. 5(b)). It is noted that the number of vector-share operations and the number of MUXs inputs are proportional, as the Vector-Share process selects inputs for the MUXs. As a result, the problem of minimizing $F(Input_{MUX})$ is equivalent to the Problem 1.

***Problem** 1:* Each row of matrix A has q ones and zeros otherwise. What is the minimum number of vector-share operations required to make the matrix A an empty matrix?

III. Methodology

For greedy algorithm, it selects two columns whose result after a vector-or operation is most likely to result in the greatest number of ones and repeats this step until further sharing is not possible. Although the algorithm finds the optimal solution at each step, the problem lacks the property of optimal substructure, meaning that an optimal solution cannot be constructed from optimal solutions of its subproblems. This is also why the near-optimal solutions from the greedy algorithm are always far from optimal. However, heuristic algorithms like MCTS can overcome the limitations of local optimal and find better near-optimal solutions.

In this section, we first propose a preliminary algorithm (Alg. 1) to handle simple cases such as the one in Fig. 1, and then propose a complete and universal algorithm (Alg. 2) to solve complex scenarios. Here, we only consider the elementary arithmetic, but the algorithm also applies to other operations.

A. MCTS Based MUXs Inputs Selection Algorithm

According to the analysis in Section. II, minimizing $F(Input_{MUX})$ is equivalent to minimizing $N_{vec-share}$. The process of obtaining codes in Fig. 2 after sharing codes in Fig. 1 can be abstracted as the Algorithm. 1. This algorithm can be applied to solve the case where each branch has only one operation and each branch has the same number of that operation.

Algorithm 1 MCTS Based MUXs Inputs Selection Algorithm

Input: A matrix $A_{m\times n}$ with each row has q ones.
Output: A set of the inputs of each MUX R.

1: $R \leftarrow \phi$
2: **while** $A \neq \phi$ **do**
3: Select and remove two columns $\vec{c_0}$ and $\vec{c_1}$ from A
4: $\vec{v_{or}} \leftarrow VectorOr(\vec{c_0}, \vec{c_1})$
5: $\vec{v_{and}} \leftarrow VectorAnd(\vec{c_0}, \vec{c_1})$
6: **if** $\vec{v_{or}}$ has zero **then**
7: Add $\vec{vor}$ into A as a new column
8: **end if**
9: **if** $\vec{v_{and}}$ has one **then**
10: Add $\vec{v_{and}}$ into A as a new column
11: **end if**
12: Add $(\vec{c_0}, \vec{c_1})$ to R
13: **end while**

In Algorithm. 1, one of the most important steps is to select $\vec{c_0}$ and $\vec{c_1}$ in line 3. Since the greedy algorithm cannot find the optimal solution, a heuristic algorithm called Monte Carlo tree search (MCTS) is applied to do the space exploration. In a Monte Carlo tree, each node stores a matrix state, and its children represent the new matrix state after a vector-share operation in present state. Comparing to the traditional MCTS with UCB algorithm in [4], we made the following modifications:

- **Expansion Stage**: For the matrix of the current node, select two columns to apply Algorithm. 1 to generate a new matrix as a child of the current node.
- **Simulation Stage**: In the simulation, Algorithm. 1 is randomly applied to two selected columns of the matrix until it can no longer be shared. This process is repeated multiple times, and the average number of vector-share operations used is recorded as the result of the simulation. The fewer operations used, the more efficient the resource sharing is and the closer it is to the optimal strategy.
- **Backpropagation Stage**: Passing the results of the simulation to the ancestors allows MCTS to find shared solutions that have more potential to reach the optimal policy right from the ancestors.

Unlike conventional MCTS, which finds the decision for the next step, the modified algorithm can directly find the final sharing solution. It only needs to traverse from the root of the MCT downward to the leaves to find the relatively optimal sharing solution. As shown in Figure. 6, the three child nodes of the root node are simulated to get the number of vector-shares required for completion of the sharing. After comparison, it is determined that a minimum of 1 vector-share is needed and the result is backpropagated to the root node, updating its minimum number of vector-shares required to 2. Then the search process is repeated and giving priority to the child node which need fewer vector-shares for access expansion. But it should be noted that, due to the implementation

of the UCB algorithm, MCTS reduces its interest in specific child nodes after repeated visits, and instead visits nodes that may not appear optimal at the moment, but have the potential to reach optimality in the future.

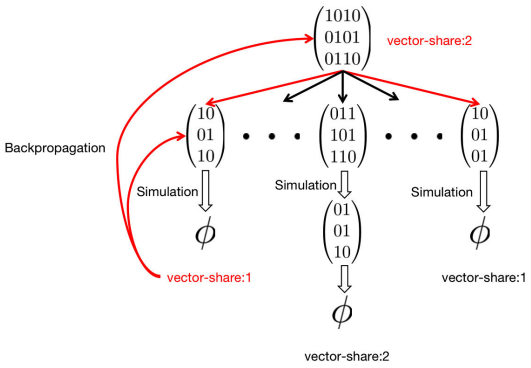


Fig. 6. An instance of MCTS

B. Universal MUXs Inputs Selection Algorithm

Last subsection we discussed how to process a simple case. However, in most cases, each branch has different kinds of operations and different numbers of operations. This subsection will introduce the categorization and processing of various operations, the processing of cascading operations, and the processing of different numbers of operations to construct a more realistic and universal problem. We solve the complex problem by splitting it into several steps by reusing Algorithm. 1.

- **Operations Categories**: The operations can be divided into two categories. We define the operations that can exchange two inputs as *exchangeable operations* (such as addition and multiplication), and those that cannot be exchanged *non-exchangeable operations* (such as subtraction and division). For exchangeable operations, it's easy to apply Algorithm. 1. But for non-exchangeable operations, such as subtraction, we can only choose subtrahends or minuends to apply Algorithm. 1.
- **Cascade Operations**: Some operations like division and multiplication can form cascade operations. It may form cases like:

$$a_0 \times a_1 \div a_2 \dots a_n \div a_{n+1} \tag{2}$$

We refer to this form of operations as *cascade operations*. To share these operations, a left-to-right recursive scheme is used. For the cases in (2), part 1 is considered to be $a_0 \times a_1$, and part 2 is considered to be $a_2 \dots a_n \div a_{n+1}$, resulting in a division. This idea can be applied to all other cascade operations from different conditional branches, and all the part 1 can be shared. After sharing, the part 1 can be seen as a new register r_0, and (2) becomes:

$$r_0 \div a_2 \dots a_n \div a_{n+1} \tag{3}$$

This process is repeated until the cascade operation becomes either an exchangeable operation or a non-exchangeable operation.

- **Alignment**: Sometimes, conditional branches may have varying amounts and types of operations. In such cases, only the conditional branches that can be shared are chosen rather than all of the branches. After several steps, some branches may no longer be shareable, meaning the rows they represent become a zero vector. These branches are then removed, indicating that they cannot be shared any further. This process is referred to as *alignment*.
- **Replacement**: For the case where there are multiple operations in a single conditional branch, after sharing one kind of operations, we treat the results of these operations as new registers to be shared for subsequent sharing. We refer to such an operation as replacement. The existence of this operation allows us to further exploit the shared results of different operations instead of processing them in a fragmented way.

The parts above help to build more complex and universal scenarios, and help to design a more generic algorithm. First we share cascade operations, then we can get a transformed cases who only consists exchangeable and non-exchangeable operations. Then we share the rest operations following the order of DMSA (division, multiplication, subtraction, and addition) since non-exchangeable operations can only choose one input to share, which means they're harder to share. After that, we can obtain complete MUXs inputs selection solutions. The optimized algorithm is shown in Algorithm. 2.

Algorithm 2 Universal MUXs Inputs Selection Algorithm

Input: A universal case S
Output: A set of the inputs of each MUX R.
1: $R \leftarrow \phi$
2: $P \leftarrow [\div, \times, -, +]$
3: Share cascade operations of S
4: **for all** $t \in P$ **do**
5: Extract operations t from S
6: Generate matrix A
7: Call Algorithm 1 with alignment and get the result r
8: Add r to R
9: Replace operations t with new registers
10: **end for**

IV. Experimental Results

We have implemented the proposed MCTS-based algorithm (Algorithm. 2), the greedy algorithm from [5], and a brute force search approach with Python. The brute force search involves enumerating all possible resource sharing solutions and selecting the optimal one. Although it will certainly find the optimal solution, the time cost is substantial. Experiments are conducted to compare them based on Synopsys Design Compiler (DC), a commercial logic synthesis tool. The open NangateOpenCellLibrary [8] is selected as the library, and all test cases are written by Verilog. All the experiments are conducted on a Ubuntu 16.04 LTS system, and the CPU is Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz. We also set the MCTS searching Time to 0.5 second, which is acceptable in most scenarios.

Considering the lack of test cases, we opted to automatically generate test cases for conducting the experiment. For

simplicity, it was assumed that each branch has only one assignment statement and that all assignments in different branches are made to the same output. Two sets of test cases of different sizes were randomly generated. For small-size cases, the number of conditional branches m ranged from 2 to 7, the number of input variables n ranged from 3 to 7, and the number of operations c involved in each branch ranged from 2 to n. In small-size cases, the time consumption of the brute force search was barely acceptable, so we selected the MCTS, greedy algorithm, and brute force search for comparison. For large-size cases, m ranged from 7 to 10 and n ranged from 3 to 12. In such cases, the time consumption of the brute force search was unacceptable, so we only compared the MCTS and greedy algorithms. We generated 10 small-size configurations and 12 large-size configurations, and automatically generated 200 test cases for each configuration as a group. To simplify the problem and focus solely on the selection of MUX inputs, our conditional control statements were all statements with the equal sign such as $state == 0$, and all inputs were set to 32 bits wide.

After generating the test cases, the original codes were processed using the MCTS and greedy algorithms to generate processed codes. These processed codes were then compiled using DC to obtain area information, with the resource sharing function of DC being disabled using the command *set hlo_resource_allocation none*. The original and processed codes were also compared through Multisim simulation for equality vilidation.

To better visualize the optimization results, we plot the average area of each group in Fig. 7(a) and 7(b). The greater the number of rows (m) and columns (n) of a matrix, the more difficult it is to share resources. On the other hand, the denser the matrix is (i.e., the greater c), the easier it is to share. Therefore, to quantify the size of cases, a size parameter α is also defined as follows.

$$\alpha = m \times n \times (n - c) \tag{4}$$

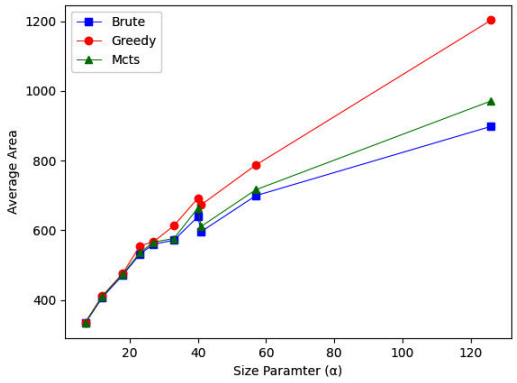
From Fig.7(a) we see that, the MCTS algorithm yields results that have an average circuit area almost indistinguishable from the optimal solution obtained through the brute force search. Meanwhile, the greedy algorithm has an average of 20% extra area. In Fig.7(b) for the larger size test cases, the MCTS algorithm continues to perform better than the greedy algorithm. Detailed result data can be found in Table I.

TABLE I
THE AVERAGE NUMBERS OF MUX INPUTS AND SYNTHESIZED AREAS OBTAINED WITH THE GREEDY ALGORITHM AND THE PROPOSED MCTS BASED ALGORITHM, FOR TWO SETS OF TEST CASES.

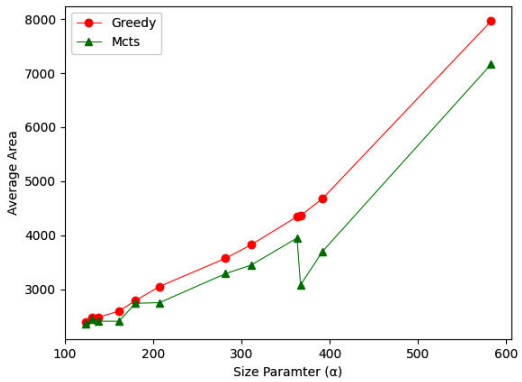
Cases	N_{MUX_Inputs}			$Area$		
	Greedy	MCTS	Reduction	Greedy	MCTS	Reduction
Small	47.1	33.6	28.7%	634.7	589.3	7.2%
Large	98.7	64.3	34.9%	3751.8	3357.3	10.5%

V. CONCLUSION

In this study, we have demonstrated through theoretical analysis and experimental results that MCTS, a heuristic algorithm, can effectively solve resource sharing problems and outperforms traditional algorithms such as greedy. In the future, we plan to further improve the algorithm by incorporating real-world cases and expanding the experimental settings (e.g. changing the bit width and the number of branches), and balance the trade-off between area and timing.



(a) The results of small size test cases



(b) The results of large size test cases

Fig. 7. The experimental results of automatically generated test cases

REFERENCES

[1] Wakabayashi, Kazutoshi, and Hirohito Tanaka. "Global scheduling independent of control dependencies based on condition vectors." Proceedings of the 29th ACM/IEEE Design Automation Conference. 1992.

[2] Kim, Taewhan, Jane W-S. Liu, and C. L. Liu. "A Scheduling Algorithm for Conditional Resource Sharing." ICCAD. 1991.

[3] Memik, Seda Ogrenci, et al. "Global resource sharing for synthesis of control data flow graphs on FPGAs." Proceedings of the 40th annual Design Automation Conference. 2003.

[4] Browne, Cameron B., et al. "A survey of monte carlo tree search methods." IEEE Transactions on Computational Intelligence and AI in games 4.1 (2012): 1-43.

[5] Raje, Salil, and Reinaldo A. Bergamaschi. "Generalized resource sharing." Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design. 1997.

[6] Jiawen Cheng, Yong Xiao, Yun Shao, Guanghai Dong, Songlin Lyu, and Wenjian Yu, "Machine-learning driven architectural selection of adders and multipliers in logic synthesis," ACM Transactions on Design Automation of Electronic Systems, 2022

[7] R. Camposano and R. A. Bergamaschi, "Redesign using state splitting," in Proceedings of The European Design Automation Conference, (Glasgow, Scotland), pp. 157-161, IEEE, March 1990.

[8] https://si2.org/open-cell-library/