

RESEARCH ARTICLE

OpenLS-DGF: An Adaptive Open-Source Dataset Generation Framework for Machine Learning Tasks in Logic Synthesis

Liwei Ni¹, Xingquan Li²¹ Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China² Independent researcher**Funding:** No specific funding was received for this work.**Potential competing interests:** No potential competing interests to declare.

Abstract

This paper introduces OpenLS-DGF, an adaptive logic synthesis dataset generation framework, to enhance machine learning (ML) applications within the logic synthesis process. Previous dataset generation flows were tailored for specific tasks or lacked integrated machine learning capabilities. While OpenLS-DGF supports various machine learning tasks by encapsulating the three fundamental steps of logic synthesis: Boolean representation, logic optimization, and technology mapping. It preserves the original information in both Verilog and machine-learning-friendly GraphML formats. The verilog files offer semi-customizable capabilities, enabling researchers to insert additional steps and incrementally refine the generated dataset. Furthermore, OpenLS-DGF includes an adaptive circuit engine that facilitates the final dataset management and downstream tasks. The generated OpenLS-D-v1 dataset comprises 46 combinational designs from established benchmarks, totaling over 966,000 Boolean circuits. OpenLS-D-v1 supports integrating new data features, making it more versatile for new challenges. This paper demonstrates the versatility of OpenLS-D-v1 through four distinct downstream tasks: circuit classification, circuit ranking, quality of results (QoR) prediction, and probability prediction. Each task is chosen to represent essential steps of logic synthesis, and the experimental results show the generated dataset from OpenLS-DGF achieves prominent diversity and applicability. The source code and datasets are available at <https://github.com/Logic-Factory/ACE/blob/master/OpenLS-DGF/readme.md>.

Corresponding author: Xingquan Li, lixq01@pcl.ac.cn

I. Introduction

Logic synthesis is a key phase in the electronic design automation (EDA) flow of digital circuits, translating high-level specifications into a gate-level netlist. Recently, there has been a trend towards adopting ML approaches for the EDA^[1] domain. Various machine learning methodologies have been proposed, demonstrating improvements in different

aspects of the logic synthesis process, including logic optimization^{[2][3][4][5][6]}, technology mapping^{[7][8][9]}, and formal verification^{[10][11]}. These machine learning-based techniques have shown their promise in improving the efficiency and quality of logic synthesis steps. In order to further develop these techniques, it is crucial to introduce more comprehensive and reliable datasets.

Previous benchmarks^{[12][13][14][15][16][17][18]} provide a foundation for testing, comparison, and enhancement, significantly advancing the development of EDA tools and methodologies. Moreover, logic synthesis datasets^{[19][20][11]} such as OpenABC-D, have been derived from these foundational benchmarks. However, these datasets are often tailored for specific tasks, limiting their use cases for diverse applications in machine learning. This paper underscores the need for a more versatile and adaptive dataset generation framework capable of supporting a variety of machine-learning tasks in logic synthesis. Such a framework should ideally possess the following attributes:

- **Diversity:** Generating a dataset covers a wide range of design types and categories, ensuring it can cater to a diverse array of use cases and applications;
- **Versatility:** Generating a dataset has the capacity to support various machine learning tasks, facilitating the sharing of the same dataset across different tasks;
- **Adaptivity:** Generating a dataset can adapt to different tasks, enabling the extraction of sub-datasets tailored to specific downstream tasks.

While the EDA flows like OpenLane^{[21][22]} are primarily aimed at facilitating the chip tape-out process, they do not inherently provide the specific needs for dataset generation. This further emphasizes the demand for a dedicated, adaptable dataset framework within the logic synthesis domain.

To address these limitations, we introduce OpenLS-DGF, an adaptive logic synthesis dataset generation framework designed to support a wide range of machine learning tasks within logic synthesis. The proposed framework covers the three fundamental stages of logic synthesis: Boolean representation, Logic optimization, and Technology mapping. The comprehensive workflow includes seven distinct steps, including the raw file generation and the dataset packing. OpenLS-DGF preserves all original information in the intermediate files, which are stored in both Verilog and ML-friendly GraphML formats. The verilog files offer semi-customization capabilities, enabling researchers to integrate desired intermediate steps and utilize previously generated verilog files. Furthermore, OpenLS-DGF includes a specialized circuit engine, which was developed to facilitate effective dataset packaging and extraction of adaptive sub-datasets for multiple tasks. This circuit engine can faithfully reconstruct the original Boolean circuit information, enabling a wide range of operations to be directly applied for further processing.

We generate the OpenLS-D-v1 dataset utilizing the above framework to facilitate multiple machine-learning tasks. OpenLS-D-v1 starts from 46 combinational designs from well-established benchmarks^{[16][17][18]}, including a diverse circuit type, such as arithmetic circuits, control circuits, and IP cores. It encompasses more than 966,000 Boolean circuits, each derived from 1,000 unique synthesis recipes. The breakdown of the dataset includes 7000 Boolean networks across 7 logic types, alongside 7000 ASIC and 7000 FPGA netlists. Moreover, QoRs are preserved in JSON format alongside their

corresponding Boolean circuits. To showcase the versatility of OpenLS-D-v1, we have implemented and tested four typical machine-learning tasks within logic synthesis: circuit classification, circuit ranking, QoR prediction, and probability prediction. Each task explores unique processes of logic synthesis, employing datasets that are directly extracted and specifically reformatted from OpenLS-D-v1. The experimental results substantiate the diversity and effectiveness of OpenLS-D-v1, confirming its value across various machine-learning applications and demonstrating the prominent diversity and applicability of OpenLS-DGF.

The contributions can be summarised as follows:

- We introduced OpenLS-DGF, an adaptive logic synthesis dataset generation framework that covers three pivotal stages: Boolean representation, logic optimization, and technology mapping. OpenLS-DGF also offers semi-customized capabilities, allowing the reuse of intermediate files for researchers to integrate additional steps as needed;
- We developed an adaptive circuit engine capable of loading multiple types of Boolean circuits without losing any information. This engine serves as a bridge between the framework and various downstream tasks, facilitating the extension of operations to generate desired features;
- We generated OpenLS-D-v1, an adaptive logic synthesis dataset generated using OpenLS-DGF, designed to support various machine learning tasks. This ensures that all feasible downstream tasks can derive their specific datasets directly from OpenLS-D-v1;
- We implemented four typical downstream tasks utilizing the OpenLS-D-v1 dataset to demonstrate its diversity and effectiveness. Moreover, the circuit ranking is a novel task in logic synthesis introduced by this study, highlighting improvements in technology mapping.

This paper is structured as follows: Section II provides the background and related works; Section III presents the details of OpenLS-DGF; Section IV introduce the generated OpenLS-D-v1 dataset and its key characteristics; Section Vformulates the selected four downstream tasks on OpenLS-D-v1 and gives the experimental results; Section VI gives the discussion, and Section VII draws the conclusion.

II. Background and Related Work

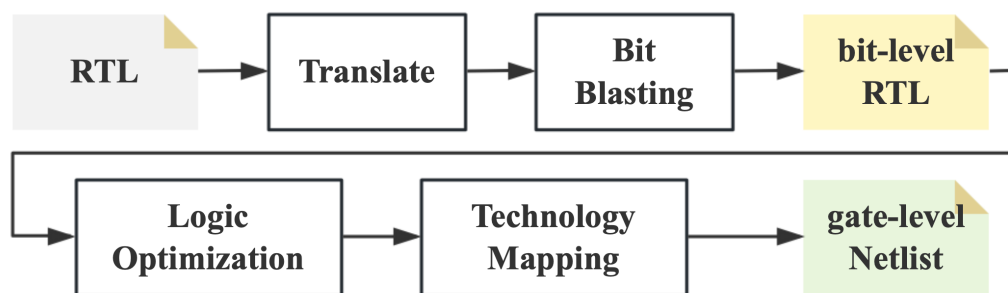


Figure 1. The Logic Synthesis flow.

Fig. 1 illustrates the essential steps of the logic synthesis flow. The following subsections will introduce the fundamental concepts and related works of logic synthesis.

A. Background

1) Boolean circuit and Functional completeness

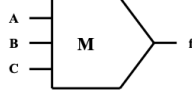
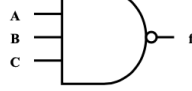
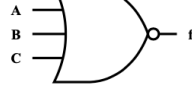
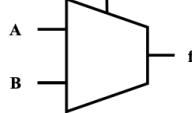
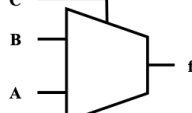
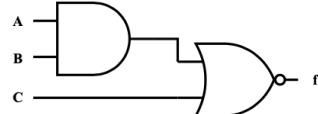
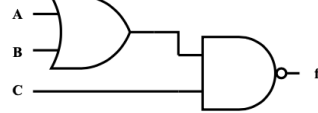
Logic Gate	Boolean Expression $f(A, B, C)$	Distinctive shape
Primitive Gates ¹
MAJ3	$A \cdot B + A \cdot C + B \cdot C$	
NAND3	$\overline{A \cdot B \cdot C}$	
NOR3	$\overline{A + B + C}$	
MUX21	$C \cdot B + \overline{C} \cdot A$	
NMUX21	$C \cdot A + \overline{C} \cdot B$	
AOI21	$\overline{A \cdot B + C}$	
OAI21	$\overline{(A + B) \cdot C}$	

Table I. The logic gates pool in this framework.

The **Boolean circuit** C is defined as a computational graph representation with specific Boolean function. It can be formulated by the following: $C = (V, E)$, $V = V^{PI} \cup V^{LG} \cup V^{PO}$, $(v_i \rightarrow v_j) \in E \mid v_i \in V, v_j \in V$, where V^{PI} represents the primary input nodes (PIs), V^{PO} represents the primary output nodes (POs), and V^{LG} represents the internal logic gates. Table I shows the used logic gates in this paper but the technology-dependent cells. Each edge $v_i \rightarrow v_j$ in E represents the connected signals between nodes.

Furthermore, the technology-independent Boolean circuits, also referred to as the Boolean network, primarily concentrate

on the topology and Boolean function. On the other hand, the technology-dependent Boolean circuit, which represents a gate-level netlist, incorporates physical attributes such as area, and timing. It should be noted that the sequential Boolean circuits are not discussed in this work.

Table II. The related functional complete set.	
Logic Circuit Type	Functional Complete Set
And-Inverter Graph (AIG)	NOT, AND2
Or-Inverter Graph (OIG)	NOT, OR2
Xor-And-Inverter Graph (XAG)	NOT, XOR2, AND2
Majority-Inverter Graph (MIG)	NOT, MAJ3
Primitive-Gate Graph (PRIMARY)	NOT, AND2, NAND2, OR2, NOR2, XOR2, XNOR2
Generic-Technology Graph (GTG)	{PRIMARY}, NAND3, NOR3, MUX21, NMUX21, AOI21, OAI21

Definition 1 (Functional completeness). A set of logical gates S is called functionally complete, if for any Boolean function f , there exists a circuit using only gates from S that can represent f .

As defined at Table II, the Boolean circuit employs a functionally complete set, thereby enabling the representation of any Boolean function through circuit type. Table II illustrates the Boolean networks utilized in this work, including AIG, OIG, XAG, MIG, PRIMARY, and GTG. While the gate-level netlists are involved after the technology mapping.

Boolean representation task: Different Boolean circuits, based on functional completeness, can exhibit varying performances across different stages of the EDA flow, a phenomenon known as the Boolean representation problem.

2. Logic Optimization and Technology Mapping

The Boolean equivalence^[23] asserts that the different Boolean circuit graph structure may lead to the same Boolean function. Moreover, it is the fundamental theory for logic optimization and technology mapping. The **logic optimization** algorithms aim to reduce the cost of the Boolean network to improve the desired criterion (area, timing, ...). Then, the optimized Boolean networks are translated into the gate-level netlists by **technology mapping** with the given standard cell library. This standard cell library encompasses a functionally complete set of gate-level netlists equipped with essential physical attributes required for technology mapping.

Circuit classification task: According to the Boolean equivalence theory, the Boolean networks derived from the same design have the same functionality. From this viewpoint, these Boolean networks are in the same class.

Probability prediction task: The functionally equivalent nodes within one Boolean network can be merged to reduce the size. By computing the probability of nodes, it is possible to effectively identify and filter the functionally equivalent nodes.

3. Static Timing Analysis

Static Timing Analysis (STA) is a critical technique used to ensure that the gate-level netlist meets specified timing requirements. STA involves calculating the delay for each signal path within the circuit. The total delay for any path is determined by the equation:

$$\text{path_delay} = \sum (\text{gate_delay}) + \sum (\text{wire_delay}),$$

where `gate_delay` represents the internal delay of each gate and `wire_delay` accounts for the delay of the wires between gates. The maximum path delay of the critical path (arrival time) is typically used to assess if the circuit can operate within the desired timing period.

QoR prediction task: Different logic optimization and technology mapping configurations can lead to different QoR results. If the QoR distribution is determined, it is possible to predict the related QoR.

4. Graph Neural Network (GNN)

GNNs are particularly adept at handling data structured in the form of graphs, offering important insights in applications where relationships and interactions are crucial. A GNN utilizes a multi-layered structure where each layer K updates a node's representation by aggregating features from its neighbors. This aggregation follows the update rule:

$$\begin{aligned} h_{N(v)}^k &= \text{AGGREGATE}_k(\{h_u^{k-1}, \forall u \in N(v)\}), \\ h_v^k &= \sigma(\mathbf{W}^k \cdot \text{CONCAT}(h_v^{k-1}, h_{N(v)}^k)). \end{aligned}$$

where h_v^k represents the embedding of node v in depth k ; the AGGREGATE is the differentiable aggregator function; $N(v)$ represents the neighborhood nodes of node v ; \mathbf{W} is the weight matrices and σ is the non-linearity function.

B. Related Work

Since the release of the "ImageNet" dataset^[24], the field of artificial intelligence has experienced significant advancements in computer vision. ImageNet has enabled a variety of applications, including image classification, segmentation, and detection. This progress has led to the development of innovative ML algorithms that are transforming fields such as autonomous driving, robotics, and natural language processing.

In recent years, the application of ML in logic synthesis has also seen considerable growth. The "OpenABC-D"^[49] dataset is generated by the OpenLane^[21] flow, which mainly produces intermediate files rather than being specifically designed for dataset generation. Existing datasets, such as those for probability prediction^[20] and node classification^[11], are typically tailored for specific tasks. However, there is a notable lack of a comprehensive dataset that can support multiple tasks.

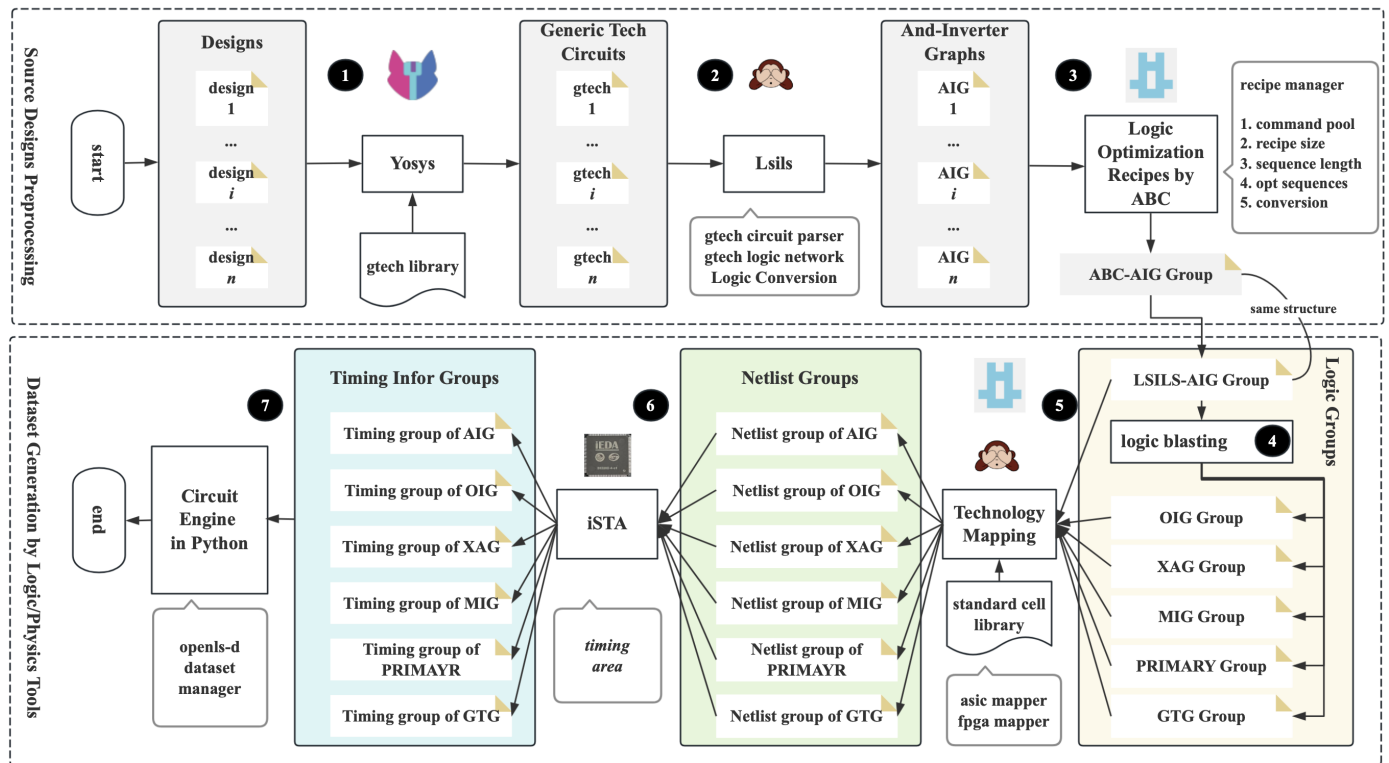


Figure 2. The adaptive logic synthesis dataset generation framework of OpenLS-DGF.

To address this gap, we introduce OpenLS-DGF, the logic synthesis dataset generation framework designed for various downstream tasks. OpenLS-DGF not only accommodates multiple tasks but also enables these tasks to share a common dataset. This approach standardizes the measurement of task performance, promoting consistency across different evaluations and enhancing comparability within the field.

III. OpenLS-DGF

In this section, we introduce the proposed OpenLS-DGF along with the circuit engine.

A. Overview

Fig. 2 illustrates the dataset generation flow of OpenLS-DGF. It covers the three fundamental steps in logic synthesis: Boolean representation, logic synthesis, and technology mapping. To streamline the process, all processes are integrated into the open-source platform, LogicFactory^[25], utilizing the TCL command environment. This framework involves 7 distinct steps, starting from the initial design input to the final dataset packaging. The first three steps (1-3) involve preprocessing the input design to generate the generic technology circuit and its optimized AIGs. The subsequent steps (4-6) are dedicated to producing intermediate Boolean circuits derived from these optimized AIGs, including logic blasting, technology mapping, and physical design. The final step (7) packages these Boolean circuits into PyTorch format data using a circuit engine, which facilitates efficient dataset management. This systematic methodology ensures that the design inputs are processed and transformed into a comprehensive dataset, ready for various logic synthesis applications.

Further details will be demonstrated in the subsequent sections.

B. Dataset Generation Steps

Step 1: Generic Technology Circuit Synthesis.

The first step involves synthesizing the generic technology circuit (GTG). Given that the source designs are provided in various formats such as Verilog, AIG, and BLIF, adopting GTG as the standardized representation for these inputs is crucial. This approach ensures uniform processing across different design types, facilitating more streamlined and consistent handling in subsequent stages of logic synthesis. We utilize Yosys^[26] served as the frontend parser for these format designs, consequently, the input source designs are translated to GTG using Yosys' "techmap" method. The generic technology cells defined in GTG correspond to the basic functionally complete set of RTL intermediate language (RTLIL) within Yosys. Besides the primitive gates, GTG includes several complex gates such as NAND3, MUX21 AOI21, and OAI21, which preserve the coarse-grained attributes similarities of the RTLIL.

Notably, similar generic technology cells are also utilized in commercial logic synthesis tools such as Design Compiler (DC) for their intermediate representations, which highlights the practical relevance of creating GTG for the input designs. We document the GTG in both Verilog and GraphML formats, facilitating further processing and exploration in subsequent processes.

Step 2: And-Inverter Graph Generation.

AIGs, composed solely of AND2 and INVERTER gates, are fundamental to most logic optimization techniques in logic synthesis due to their simplicity and structural directness. Open-source tools such as ABC^[27] and LSILS^[28] have implemented numerous logic optimization algorithms on AIG, including `rewrite`, `balance`, `refactor`, `resubstitution`, etc. To enable these optimizations, we convert the GTG generated in Step 1 into AIG. This conversion leverages the GTG parser and conversion method provided by LogicFactory. Additionally, we document each AIG in binary format alongside its corresponding Verilog and GraphML files.

Step 3: Logic Optimization Recipes.

Logic optimization aims to minimize the cost of Boolean circuits. It also generates different structural Boolean circuits with different optimization configurations (the optimization sequence). These Boolean circuit' variants can substantially influence the quality of results (QoR) during technology mapping and subsequent physical design processes. In this step, we utilize ABC to generate diverse structural Boolean circuits from a specific design's AIG. These Boolean circuits facilitate the exploration of the QoR distribution for a given design, which is crucial for the tasks related to QoR distribution.

We utilize a comprehensive set of optimization commands frequently employed in logic optimization:


```

    balance,
    rewrite, rewrite -l, rewrite -z, rewrite -l -z,
    refactor, refactor -l, refactor -z, refactor -l -z,
    resub, resub -l, resub -z, resub -l -z.

```

Additionally, the heuristic optimization sequence such as `resyn`, `resyn2`, along with sequence exploration tasks like BOILS^[29] and DRILLS^[30], are performed based on the above command pool. We generate 1000 distinct optimization sequences for each input design, with each sequence randomly composed of 10 commands from the command pool. To ensure equal selection probability, the `balance` command is repeated, appearing four times in the command pool. These 1000 distinct optimization sequences facilitate the exploration of the different sequences on the same design as well as the same sequence on different designs, enhancing the analysis of learning of their impact on design optimization. Following this step, we generated 1000 variant AIGs for each design, with each AIG indexed according to its corresponding optimization sequence.

Step 4: Logic Blasting.

Logic blasting is a process designed to transform Boolean networks into various formats. Table II shows the 6 logic types of Boolean network, including AIG, OIG, XAG, MIG, PRIMARY, and GTG. Despite the relative scarcity of optimization algorithms for these circuit types compared to AIG, logic blasting provides an avenue to potentially generate superior gate-level netlists through technology mapping for other logic types.

In this step, we utilize the LSILS tool to execute the logic blasting. Initially, the AIG groups generated by ABC are translated into corresponding AIG groups using the LSILS tool. Both AIG groups maintain identical structures for each corresponding item. Subsequently, each AIG in the LSILS AIG groups is converted into other logic types through the logic blasting method, which covers the AIG by the specific standard cell library. For example, the PRIMARY circuit consists of primitive gates {NOT, AND2, NAND2, OR2, NOR2, XOR2, XNOR2}. By utilizing standard cells composed of these gates, we can generate the PRIMARY circuit of the corresponding AIG. The gates with a larger area have a higher priority during the covering process.

The coverage of AIG, OIG, XAG, PRIMARY, and GTG are capable of generating the necessary supergate library during the covering algorithm by technology mapping. However, the functionally complete set S^{MIG} of the MIG circuit, {NOT, MAJ3}, are inadequate for generating a functionally complete supergate library due to the hardness of generating basic {"and2", "inverter"} set, which makes it complicated to cover the AIG to MIG by technology mapping. Instead, we employ a topological node-wise conversion method to achieve this conversion.

Theorem 1. *The logic blasting method preserves the dependency relationships of the original circuit.*

Proof. The logic blasting step relies on the mapping step, ensuring that nodes between the circuits, both before and after the blasting, can be precisely matched. Thus, they retain the same topological structure. The matched nodes preserve the dependency relationships. Additionally, each MAJ3 gate can represent an AND gate, and it is still feasible to meet Section

III-B. □

Lemma 1. *We can equip the logic optimization capability of AIG to the other logic types through logic blasting.*

Since most logic optimization algorithms are implemented on AIGs. According to Section III-B, we can generate similar Boolean circuits by logic blasting on AIGs. In this manner, the logic optimization capabilities of AIG are extended to other types of Boolean circuits. Following this step, we are able to generate corresponding groups of AIG, OIG, XAG, MIG, PRIMARY, and GTG Boolean circuits. All these Boolean circuits are written in Verilog and GraphML file formats.

Step 5: Technology Mapping.

For each of the 6 Boolean network groups, we generated a gate-level netlist using the same technology mapping algorithm provided by LSILS^[28], specifically through its “mapper_asic” and “mapper_fpga” methods, which are based on their respective template operation. For ASIC technology mapping, we employed the sky130^[31] standard cell library. Similarly, FPGA technology mapping was constrained to the LUT6 cell configuration. Given that certain tasks are exclusively relevant to ABC AIG, we will also employ ABC’s technology mapping algorithms for these specific instances. For ASIC technology mapping, we use the “amap” command, and for FPGA technology mapping, we apply the “if -K 6” command to accommodate the requirements. All resulting gate-level netlists, whether for ASIC or FPGA, are subsequently saved in both Verilog and GraphML file formats to ensure broad compatibility and facilitate downstream applications.

Step 6: Static Timing Analysis.

The primary goal of logic synthesis is to produce a better gate-level netlist that enhances the subsequent physical design steps. Assessing the performance of the existing Boolean circuit is crucial, as the timing information significantly influences the gate-level netlist’s performance and serves as a key metric for evaluating the results of logic synthesis. In this step, we utilize the static timing analysis (STA) tool provided by the open-source physical design tool, iEDA^[32], to compute the arrival time information for specific ASIC gate-level netlists. For FPGA netlists, the depth of the circuit provides a precise indicator for timing evaluation, offering a dependable metric for assessing the performance of the output LUT netlist. All acquired timing information is documented in JSON format, ensuring that it is accessible for further analysis.

Step 7: Dataset Packing

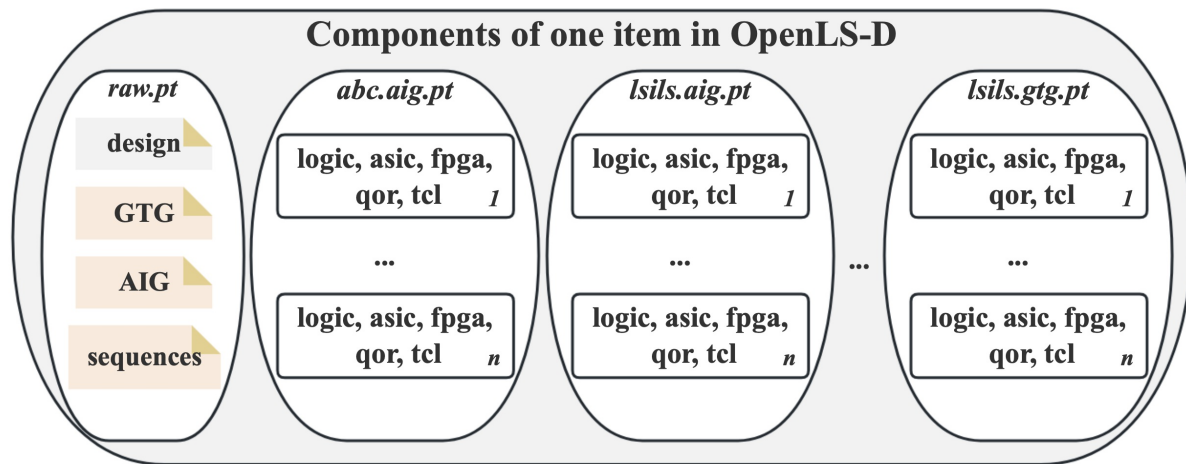


Figure 3. Components of an item in OpenLS-D-v1. For convenience, the logic types {OIG, XAG, MIG, PRIMARY} are abbreviated with ellipses.

To streamline the management of generated raw files, these files are organized and translated into the PyTorch files by different designs. This organization is executed based on the circuit engine, which will be discussed further in Section III-C.

Fig. 3 illustrates the components of an individual item in the assembled dataset. Each item is segmented into 8 PyTorch files: “raw.pt”, “abc.aig.pt”, “lsils.aig.pt”, “lsils.oig.pt”, “lsils.xag.pt”, “lsils.mig.pt”, “lsils.primary.pt”, and “lsils.gtg.pt”. The “raw.pt” file consists of 4 files: the source design, the transformed GTG circuit, the converted AIG circuit, and a fixed set of 1000 optimization sequences. The “abc.aig.pt” file contains optimized AIGs generated using the ABC tool with the generated optimization sequence. Additionally, the ASIC/FPGA gate-level netlists and their corresponding QoR (timing) are also stored. The accompanying “tcl” file aids in reproducing the respective intermediate files for further processing. Files such as “lsils.aig.pt”, “lsils.oig.pt”, “lsils.xag.pt”, “lsils.mig.pt”, “lsils.primary.pt” and “lsils.gtg.pt” share similar components with the “abc.aig.pt” file. However, the Boolean circuits and the technology mapping algorithms they utilize are specifically based on the LSILS tool.

All generated raw files undergo verification using combinational equivalence-checking tools. The files within the “raw.pt” can be checked using Yosys, while the files in the “abc.aig.pt” are checked by comparing the AIG circuits and their corresponding gate-level netlists through ABC. Similarly, the remaining files generated by LSILS are checked against their corresponding gate-level netlists in “abc.aig.pt”.

This structured approach to dataset management avoids the creation of excessively large or numerous PyTorch files for any single design. It allows for selective loading of task-related PyTorch files to derive sub-datasets as needed, enhancing efficiency. The flexibility of this framework allows for the regeneration of necessary files as required, either before or after certain steps. Additionally, the process can be tailored by inserting appropriate steps between the established ones, thus customizing the flow to better meet specific requirements.

C. Circuit Engine

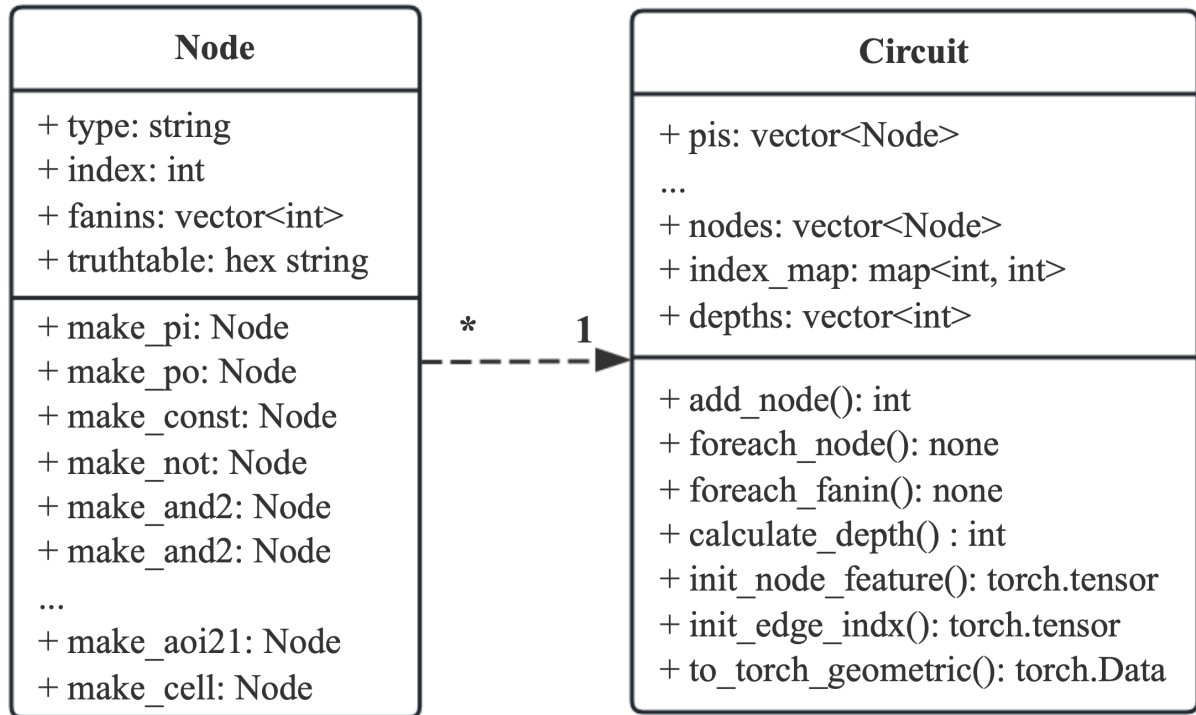


Figure 4. The UML class diagram of the generic circuit class.

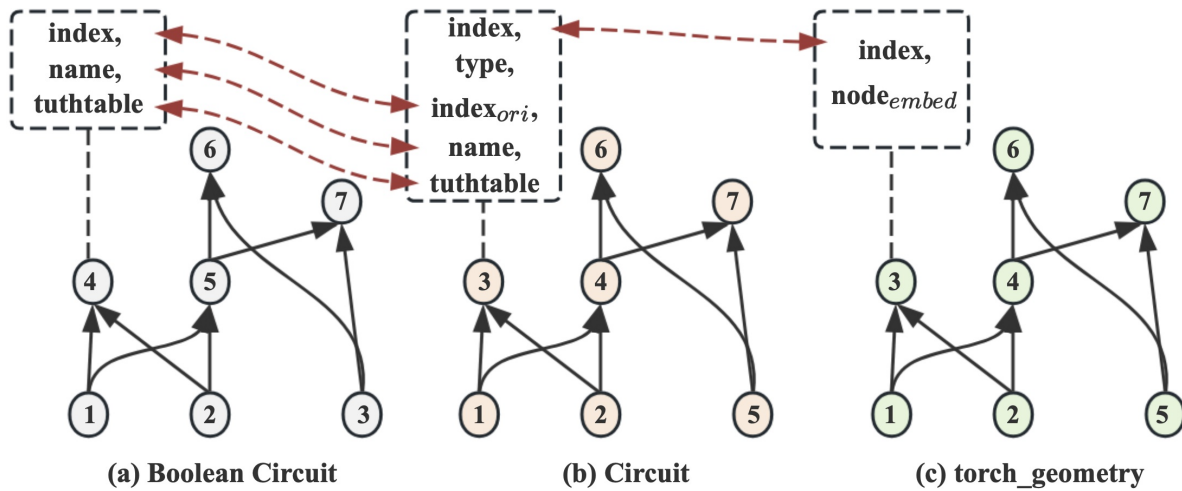


Figure 5. Node correspondence between the three types of graph: Boolean circuit, Circuit, and torch_geometry.

As mentioned in step (7) in Section III-B, the raw files generated are subsequently packaged into the dataset through the proposed circuit engine. It is primarily comprised of two main parts: the “Circuit” class, and operations for “Circuit”. Fig. 4 illustrates the UML diagram of the “Circuit” class within the circuit engine. Each node in the Circuit class has 5 attributes: `type`, `name`, `index`, `fanins`, and `truthtable`. For logic type circuits as listed in Table II, the node’s `type` and `name` correspond directly to their respective gate names such as “AND2”, “AOI21”; whereas for ASIC/FPGA gate-level netlists, the node `type` is designated as “CELL”, with the node `name` corresponding to the matched standard cell name. The `truthtable` is standardized to 64 bits to accommodate the function of each gate. Operations on the “Circuit” class include operators like “`to_torch_geometry(circuit: Circuit)`”, “`simulate(circuit: Circuit)`”, etc. The “`to_torch_geometry(circuit:`

`Circuit`” operator ensures consistency in node indices with the `Circuit` class, facilitating various operations that leverage the `torch_geometry`^[33] class. Meanwhile, the `simulate(circuit: Circuit)` operator allows for the simulation of the circuit using a range of input test cases, thereby providing valuable insights into the circuit’s behavior.

The aforementioned Boolean circuits are stored in two formats: Verilog and GraphML. We use the `load_graphml(path:str) → Circuit` operator to load the generated GraphML files into the `Circuit` class. The GraphML files are firstly loaded into a `NetworkX`^[34] graph, and the circuit is constructed by traversing the nodes and edges within this graph. Since the nodes and edges in the `NetworkX` graph are stored separately, we establish connected signals between nodes using the `add_fanin(fanins)` function of the `Circuit` class during the edge traversal after all nodes have been created. Moreover, users can easily introduce custom operations within the `Circuit` class to meet specific requirements, further utilizing this circuit engine.

Fig. 5 illustrates the node correspondence among the three types of graphs: the Boolean circuits, the proposed `Circuit`, and the `torch_geometry` graph. In this framework, the `Circuit` graph acts as a pivotal bridge, maintaining the original index ($index_{ori}$) from the input Boolean circuit while assigning a new index ($index$) for internal management. The `to_torch_geometry(circuit: Circuit)` operator ensures that the transformed `torch_geometry` graph retains the same node indices as the `Circuit` class. This design enables smooth interaction between the Boolean circuit structure and the ML-friendly `torch_geometry` graph, allowing for the efficient transition of features and operations between the two. Understanding the correspondence between these three types of graphs allows us to leverage their collective strengths to create a dataset with expanded possibilities and enhanced functionality.

IV. OpenLS-D-v1

Table III. Characteristics of the collected designs.

Design	#PI	#PO	#And	#Inv	#Edge	Depth
adder	256	129	1274	1781	5226	508
square	64	128	19499	24096	78151	445
div	128	128	27100	37726	108555	8406
multiplier	128	128	27753	31205	111242	524
max	512	130	3021	4021	12341	324
log2	32	32	32382	36027	129590	597
sqrt	128	64	32599	45647	130518	10384
sin	24	25	6604	7223	26446	273
bar	135	128	2891	3468	11820	26
cavlc	10	11	652	762	2623	26
int2float	11	7	208	242	845	23
i2c	177	128	994	1020	4148	23
priority	128	8	670	845	2696	384

voter	1001	1	10528	14208	42114	113
arbiter	256	129	11923	12173	47822	175
router	60	30	184	173	795	36
ctrl	7	26	112	119	483	11
mem_ctrl	1187	962	10001	10323	41691	58
ac97_ctrl	2339	2137	11129	13786	48414	23
steppermotordrive	28	27	133	145	567	22
ss_pcm	104	90	399	476	1722	14
usb_phy	132	90	438	465	1884	17
sasc	135	125	602	770	2637	16
simple_spi	164	132	826	989	3534	21
spi	254	238	3466	3672	14242	55
wb_conmax	2122	2075	45354	32111	185516	32
wb_dma	828	702	3644	4281	15742	41
fir	410	351	4134	5628	17022	159
des3_area	303	64	4862	4147	19544	49
iir	494	441	6302	8777	25813	193
systemcaes	927	672	9961	12880	41072	74
systemcdes	247	128	2636	3109	10728	46
usb_funct	1748	1556	13098	14481	54800	58
sha256	1943	1042	14677	16283	59752	198
dynamic_node	2708	2575	17402	22563	74297	57
fpu	632	409	27345	34160	110018	1938
aes	683	529	28655	22518	115418	44
aes_secworks	3087	2604	33953	36169	140730	71
aes_xcrypt	1975	1805	50426	43280	205032	79
tinyRocket	4561	4181	41800	50326	174934	157
tv80	636	361	9066	9333	36912	109
ethernet	10731	10422	65509	84899	282487	55
picosoc	11302	10797	71472	82817	306788	133
bp_be	11592	8413	75576	97621	317662	276
vga_lcd	17322	17063	103913	133019	449234	41
jpeg	4962	4789	119908	152637	488364	98
AVE	1882	1652	20762	24400	86129	574

In this section, we will provide a detailed overview of the structure and characteristics of the OpenLS-D-v1 dataset.

A. Data Source

1. Design Selection

Table III shows the source designs for OpenLS-D-v1 dataset generation. These designs are selected from established benchmarks, like IWLS2005^[16], IWLS2015^[17], and OpenCores^[18]. All the presented designs are combinational AIG, with the number of Primary Inputs (PIs) ranging from 7 to 17322, Primary Outputs (POs) from 1 to 17063, AND gates from 112 to 119908, Inverter gates from 119 to 152637, Edge signals from 483 to 488364, and depths from 11 to 10384.

It contains diverse types of designs, including arithmetic circuits, control circuits, and IP cores, making it a comprehensive resource for testing and benchmarking logic synthesis algorithms. In addition, new designs or internal steps can be added to update the generated dataset for specific demands incrementally.

2. Designs Diversity

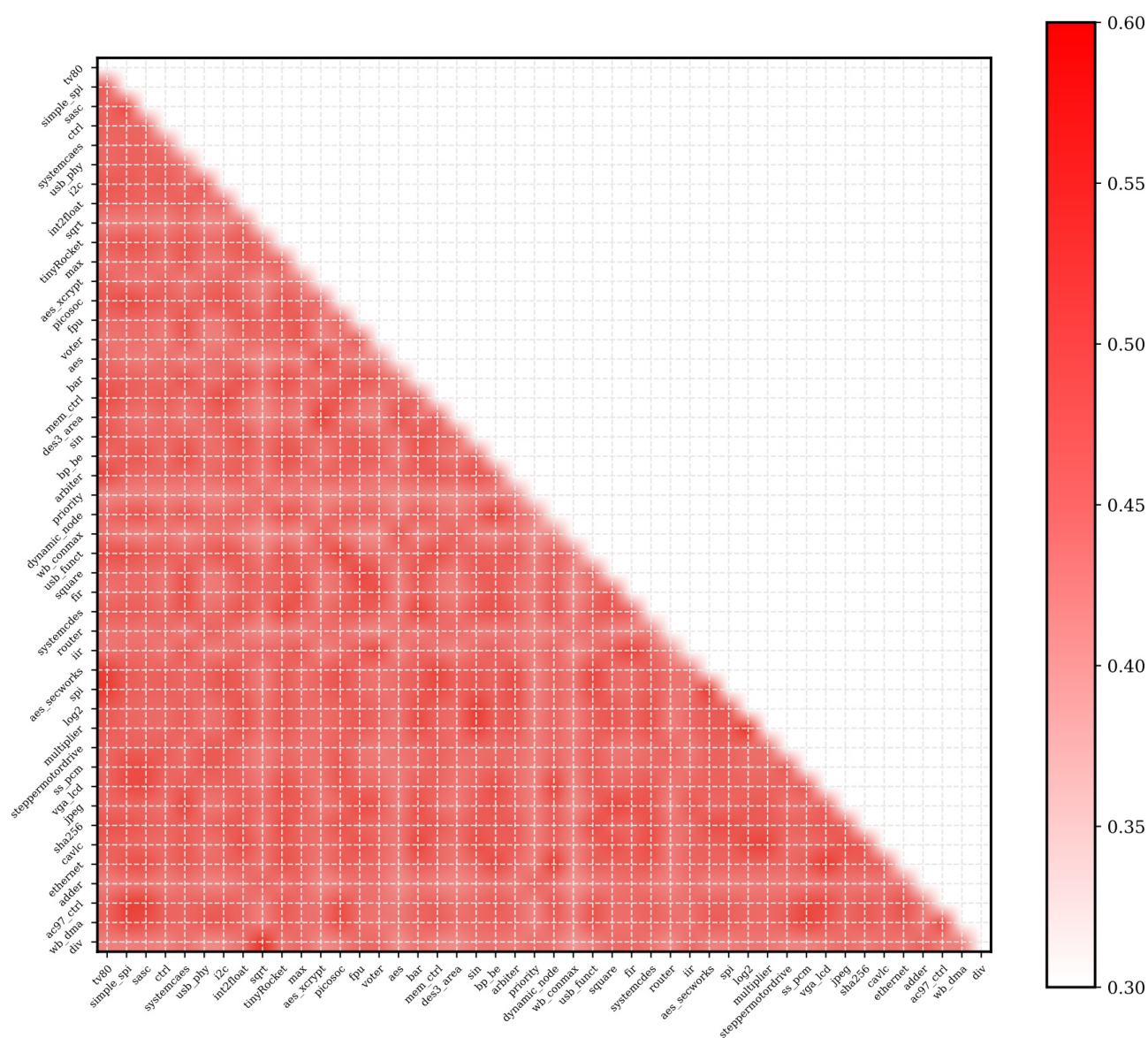


Figure 6. The graph embedding similarity of the source designs.

Fig. 6 presents the cosine similarity for the source designs as listed in Table III. Each design is represented by the graph

embedding computed using a combination of the heuristic features and features aggregated from Graph2Vec^[35]:

$$\begin{aligned} \text{feature}_1 &= [\text{pis}, \text{pos}, \text{ands}, \text{invs}, \text{edges}, \text{depth}], \\ \text{feature}_2 &= \text{graph2vec}(\text{circuit}, \text{dimension} = 128), \\ \text{feature} &= \text{concatenate}(\text{feature}_1, \text{feature}_2). \end{aligned}$$

The heuristic features provide a coarse-grained view of the design, while Graph2Vec provides a deeper insight into the internal structural intricacies. The combination of these features yields a robust graph embedding for each design.

Cosine similarity calculations are conducted using the “sklearn.metrics.pairwise.cosine_similarity” function. To enhance the visualization of this correlation, the similarity scores ranging from [-1, 1] are normalized to [0.3, 0.6]. Additionally, the diagonal and the top-right corner of the matrix are cleared to eliminate redundancy. The average cosine similarity score is around 0.44. This visualization distinctly highlights the variability across different design embeddings.

3. Dataset Generation

Table IV. The Characteristics comparison between the OpenABC-D^[19] and OpenLS-D-v1 Datasets.

	OpenABC-D	OpenLS-D-v1 (ours)
source	OpenCore IWLS	OpenCore IWLS EPFL
#designs	29	46
raw AIG		
raw GTech	x	
#recipes	1500	1000
#sequence length	20	10
#AIGs ^{abc} /design	30000 (1500 × 20)	1000
#AIGs ^{lsils} /design	x	1000
#OIGs ^{lsils} /design	x	1000
#XAGs ^{lsils} /design	x	1000
#MIGs ^{lsils} /design	x	1000
#PRIMARYs ^{lsils} /design	x	1000
#GTGs ^{lsils} /design	x	1000
#ASIC netlist/design	x	7000
#FPGA netlist/design	x	7000
#Total circuits	870k (30k × 29)	966k (21k × 46)

Table IV provides a detailed comparison between the OpenABC-D and OpenLS-D-v1 datasets. The OpenLS-D-v1 dataset includes over 966,000 Boolean circuits, structured into groups where each consists of 21,000 circuits generated from 1,000 distinct synthesis recipes. This collection includes 7,000 circuits across 7 Boolean network types, along with 7,000

ASIC and 7,000 FPGA netlists.

To optimize storage efficiency, we employed the “zstandard” compression tool^[36], which significantly reduced the storage footprint. The entire dataset generation process, including compression, was executed over approximately 76 hours for raw file generation and 65 hours for compression, using 32 threads on an Intel Xeon Platinum 8380 CPU with a 16T SEAGATE EXOS HDD. The raw files occupy about 410 GB, while the generated PyTorch files take up about 700 GB.

In comparison, although the OpenABC-D dataset involves generating a larger number of AIGs per design (30,000) from 1,500 recipes, this often results in redundancy within the generated AIGs (due to the same optimization sequence by sub-sequence extraction). Conversely, OpenLS-D-v1 adopts a more diverse approach by generating different types of Boolean networks for each design using varied logic types and synthesis sequences. This method provides a richer and more distinct dataset, better suited for various machine learning applications.

B. Dataset Characteristics

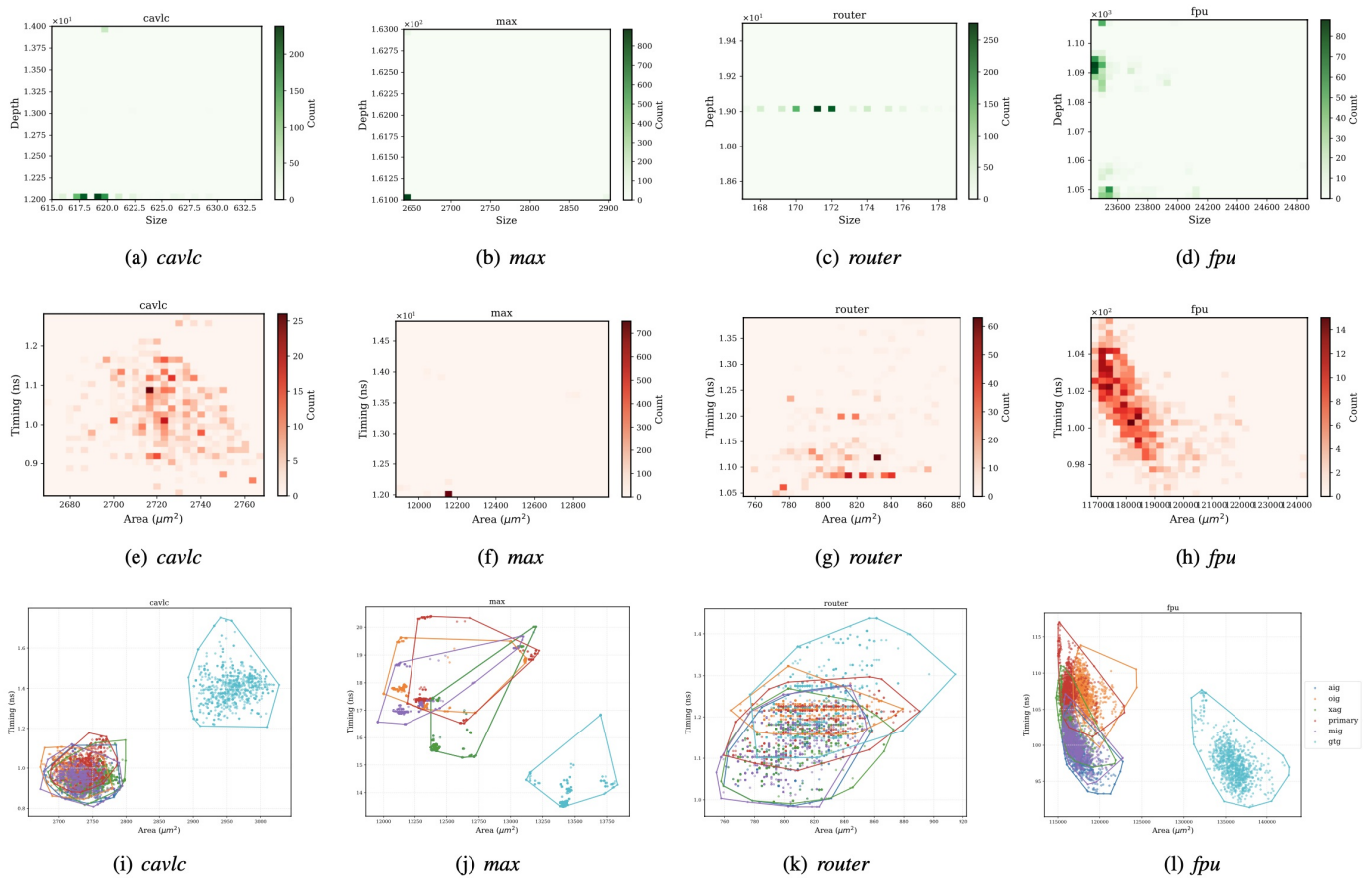


Figure 7. The QoR distribution for source designs. Each row illustrates the QoR distribution across different designs, and each column presents the three distinct types of QoR measures for a single design. Panels (a) to (d) show the node size and graph depth distribution of the optimization AIGs for the design; (e) to (h) show the area and arrival time distribution for the corresponding ASIC netlists; and (i) to (l) show the convex hulls of the QoR distribution for different types of Boolean networks.

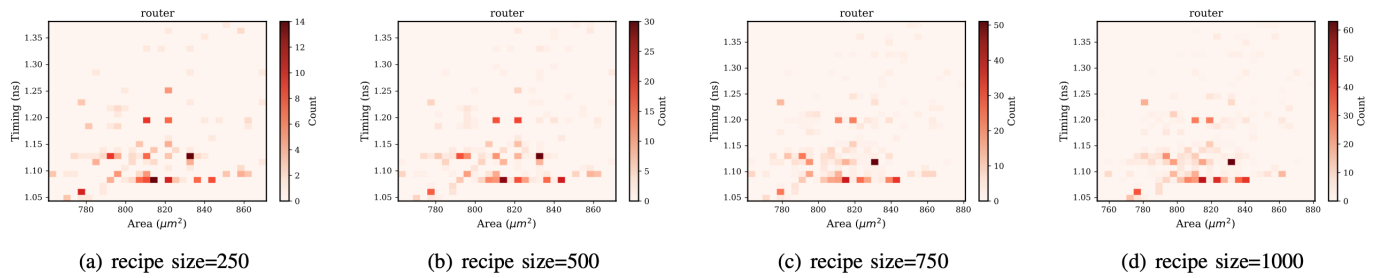


Figure 8. The distribution of node size and graph depth for one design with incremental recipe size of AIG sets.

The QoR within the OpenLS-D-v1 dataset is differentiated into two main categories: technology-independent and technology-dependent metrics. The technology-independent QoR corresponds to the Boolean circuits' structure, including the node size and maximum graph depth. Meanwhile, technology-dependent QoR relates to the physical attributes such as the total area and arrival time characteristics. The technology-independent QoR is applicable for types of Boolean circuits, while the technology-dependent QoR is specific to the gate-level netlists. Since the FPGA netlist's area and timing are generally related to the size and depth of its graph structure, we just explore the ASIC-specific characteristics here.

Fig. 7 illustrates the QoR distributions for various design parts, highlighting three types of distributions: technology-independent QoR for Boolean networks, technology-dependent QoR for ASIC netlists, and technology-dependent QoR for different logic types of Boolean networks' corresponding ASIC netlists. From this illustration, the following observations can be drawn:

Observation 1: Boolean networks with the same node size and graph depth can still have different QoR distributions of their corresponding ASIC netlist.

Logic optimization is more concerned about the local gain that can lead to global gain, it does not necessarily affect the size and depth of the optimized Boolean networks for many optimization operators. Although there are many similar QoR results of one AIG as shown in Fig. 7 (a) to (d), the area and timing distributions of their corresponding ASIC netlists are significantly dispersed. This variance is likely due to ASIC technology mapping's sensitivity to the local structures within the AIG. Thus, it suggests that the QoR distribution of the logic circuit is less indicative of performance compared to that of the corresponding gate-level netlist.

Observation 2: Different Boolean representations of one design may exhibit different behaviors.

The cut-based technology mapping method is particularly sensitive to the exploration space of the local structures, where different local structures constructed into a circuit can lead to significantly different physical properties. Consequently, the technology-dependent QoR of gate-level netlist varies among different logic types of Boolean networks with the same index in OpenLS-D-v1 of one design. Fig. 7 shows the total area and arrival time distribution for the provided logic types of Boolean networks. It highlights that there are non-overlapping regions between the convex hulls of distributions for different types of Boolean networks. This indicates that the QoR distributions for different types of Boolean networks are

not identical; in some cases, there is no overlap at all between the QoR of different Boolean representations. Furthermore, the difference between the Boolean networks' QoR distributions also varies across different designs.

Fig. 8 shows the QoR distribution under the incremental recipe size of the logic optimization of one design. From this, we can get:

Observation 3: After a certain number of optimized sequences are generated, a QoR interval typically forms, with new optimization sequences likely falling within this range.

Since logic optimization recipes primarily target AIGs, and other Boolean networks are translated from AIGs through the logic blasting method, the focus can remain on technology-independent QoR distribution for AIGs. Theoretically, these derived Boolean networks exhibit similar distributions under certain affine transformations. Fig. 8 not only showcases the node size and graph depth distribution for a selection of source designs but also highlights the incremental QoR changes across optimization sequence milestones at 250, 500, 750, and 1000 optimizations. This visualization underscores the initial observation that after a certain point, further optimizations tend to fall within a predictable range of QoR.

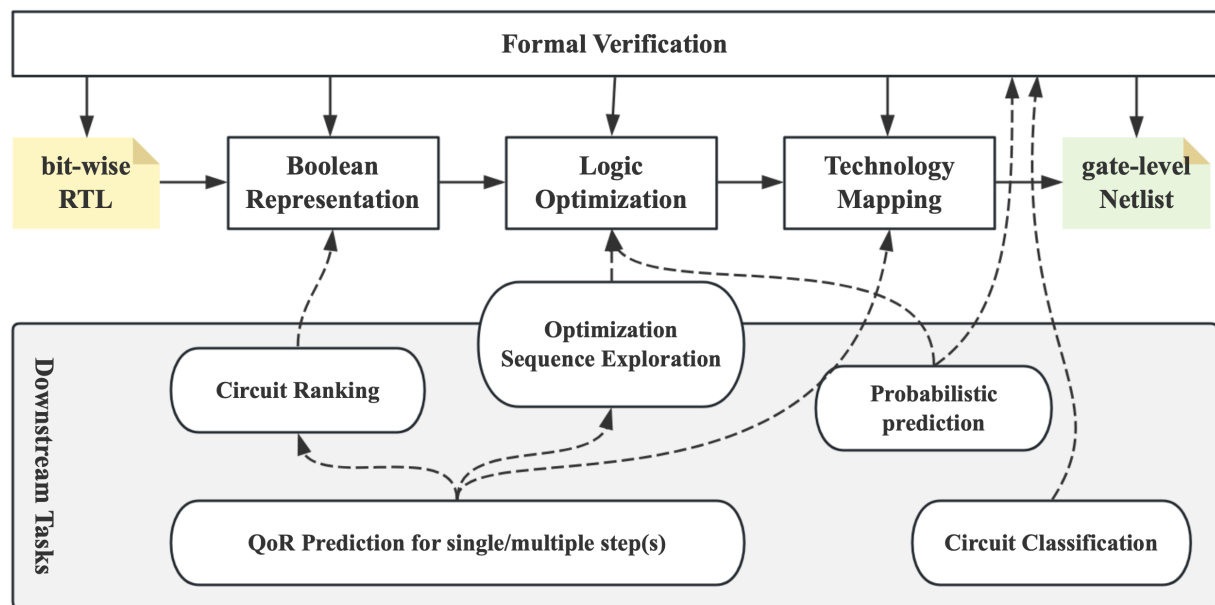


Figure 9. Illustration of the selected downstream tasks and their potential applications within logic synthesis flow.

V. Tasks Formulation and Experimental Results

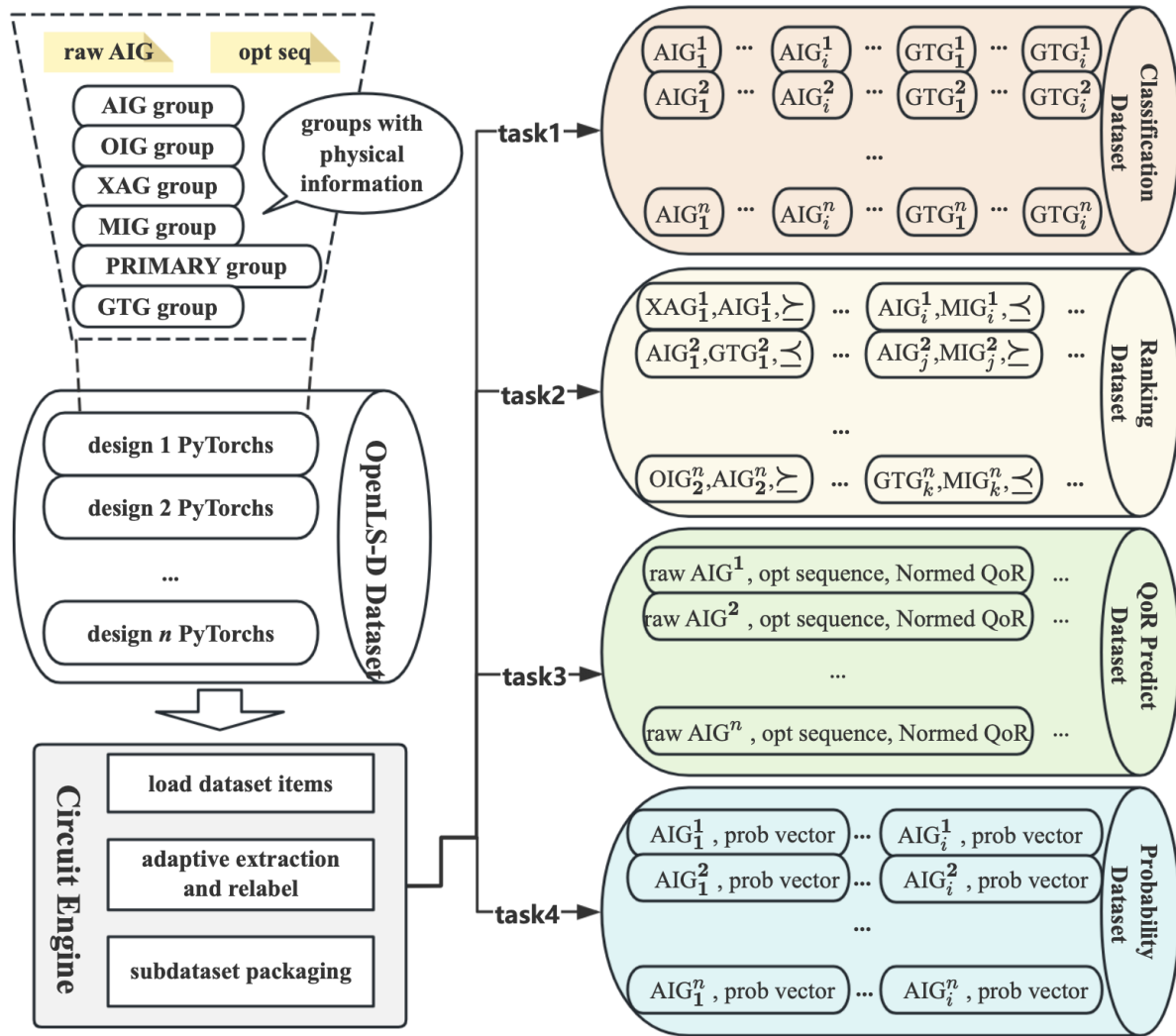


Figure 10. Adaptive sub-dataset extraction framework of OpenLS-D-v1.

In this section, we outline the formulation of the downstream tasks and present the experimental results for each. Fig. 9 illustrates the context of the four selected downstream tasks—circuit classification, circuit ranking, QoR prediction, and probability prediction—and their potential applications within the logic synthesis flow. Each of these tasks utilizes a unified adaptive sub-dataset extraction framework derived from OpenLS-D-v1.

A. Adaptive Dataset Extraction Framework

Fig. 10 illustrates the framework for adaptive dataset extraction from OpenLS-D-v1, tailored for various downstream tasks. Leveraging the capabilities of the previously mentioned circuit engine, we can simulate the behavior of the original Boolean circuits using the specially defined “Circuit” class. For a specific task within logic synthesis, the circuit engine initially loads the relevant portions of the dataset as input data for further processing. Subsequently, an adaptive function, “load_adaptive_subdataset(db:OpenLS-D-v1)”, systematically traverses each recipe of the Boolean circuits within each item to calculate and label the target items required for the task. These labeled items are then packaged into a sub-dataset dedicated to that specific task.

For example, if users wish to access only the “ABC-AIG” related data specifically, then merely the “raw.pt” and “abc.aig.pt” files are required for the process. Furthermore, Fig. 10 illustrates the items of the four sub-datasets tailored to the selected tasks. It is crucial to implement necessary operations within the “Circuit” class to cater to specific task requirements. The extraction of each sub-dataset depicted in Fig. 10 will be discussed in detail in the following subsections.

B. Environment Setup

The experimental environment for the following tasks is as follows: The hardware configuration: CPU (Intel Xeon Platinum 8380 CPU with 160 cores), Memory (512 GB RAM), GPU (NVIDIA A100 with 40 GB VRAM), while the software configuration: Operation System (Ubuntu 20.04.6), PyTorch (2.0.1), CUDA (12.0), torch_geometry (2.3.1), scikit-learn (1.2.2), pandas (1.5.3), and matplotlib (3.7.1). This high-performance setup provides a robust platform for conducting and evaluating experiments efficiently, ensuring the smooth handling of large datasets and complex computations. However, it is worth noting that the downstream tasks in this study are not resource-intensive and utilize only a fraction of the system’s capacity for training.

C. Task1: Circuit Classification

1. Problem Formulation

The circuit classification task represents the basic attribute of the circuit analysis tasks, and it can be formulated by the following: *Given a set of Boolean circuits $\{C_j\}_1^n$, classify these circuits into k ($k \leq n$) classes, and each class follows the property of the defined Boolean equivalence, thus, $\{C_i\} \equiv \{C_j\}$ in the same class, otherwise, $\{C_i\} \neq \{C_j\}$.*

2. Dataset Adaptation

Task 1 part of Fig. 10 shows the components of the circuit classification dataset. The different boolean representations of one design perform the same Boolean function, as shown in Fig. 3, thus, all these variations need to be in the same classes with the same label. In this task, the selected designs are labeled with continuous natural numbers from 0 to n , where n is the number of classes.

3. Solution: Circuit Graph Embedding Learning and Classification

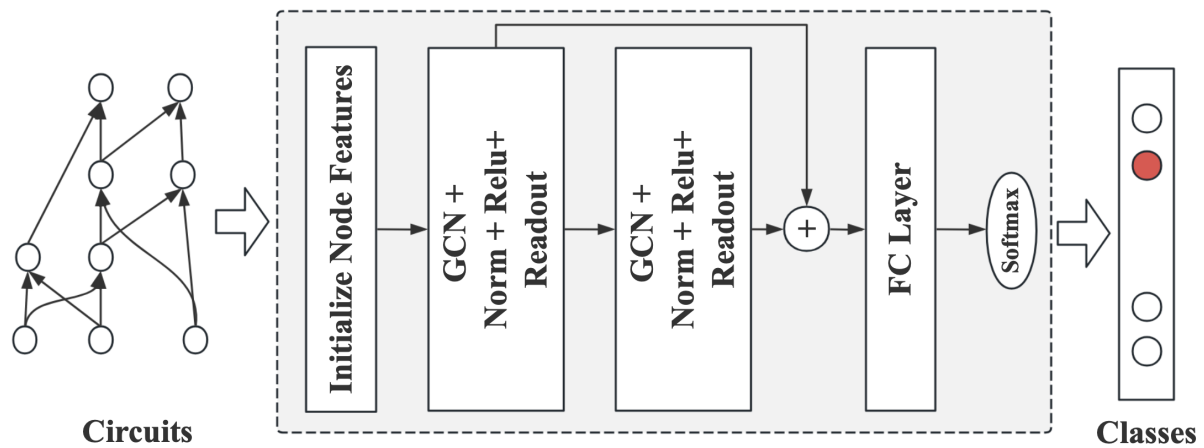


Figure 11. The GNN-based Model for the circuit classification task.

As shown in Fig. 11, a typical two-layer GCN-based graph embedding solution is given for this task. It comprises two steps: the preprocessing step, and the learning step. The first preprocessing initializes the node embedding by the node type and its truth table to embed sufficient circuit information. The following learning step aggregates the node's embedding to the graph embedding, then an MLP layer is used to predict the class number of current graph embedding. The cross-entropy function is used as the loss function here.

4) Experimental Results

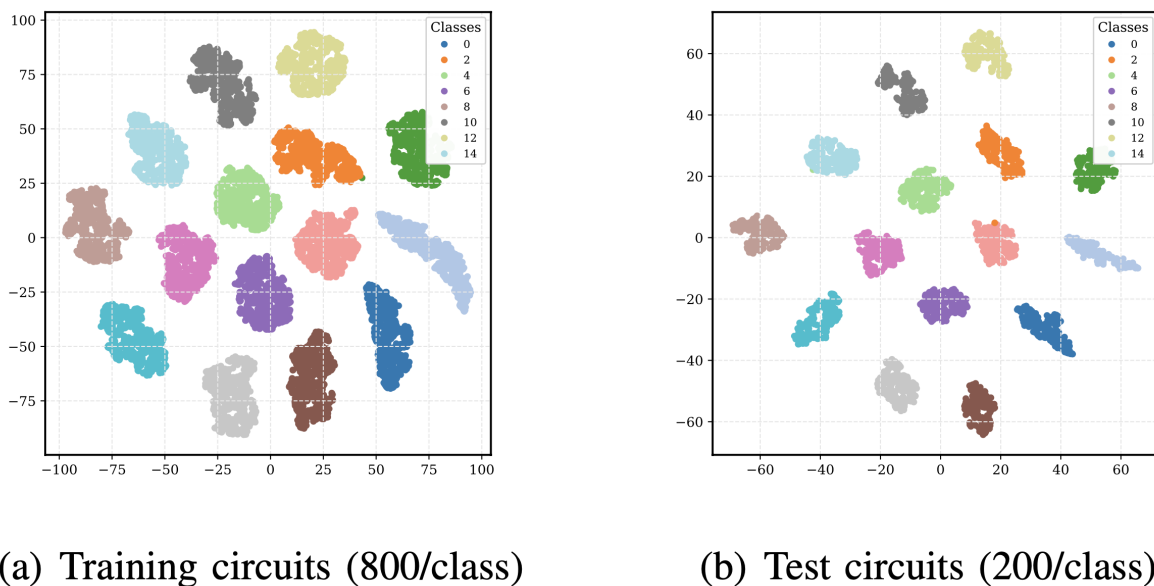


Figure 12. The t-SNE visualization for the circuit classification task with 15 designs. Labels to designs: 0 (router), 1 (usb_phy), 2 (cavlc), 3 (adder), 4 (systemcdes), 5 (max), 6 (spi), 7 (wb_dma), 8 (des3_area), 9 (tv80), 10 (arbiter), 11 (mem_ctrl), 12 (square), 13 (aes), 14 (fpu).

The experiment presented here is based on a selection of 15 designs and the reassigned labels from the OpenLS-D-v1 dataset as shown in the caption of Fig. 12. The 80% of each design's selected Boolean circuits are used for training, and

the remaining are used for validation.

The hyperparameters used in this experiment are as follows: input feature size of 64, hidden feature size of 128, learning rate of 0.0001, learning decay rate of 1e-5, and batch size of 16. By the 10th epoch, the test accuracy reaches approximately 99.8%. Fig. 12 presents the t-SNE visualization of the circuit classification results, clearly demonstrating distinct and independent distributions for each class. This task shows strong potential for analyzing circuit characteristics effectively.

D. Task2: Circuit Ranking

1. Problem Formulation

Different Boolean representations of one certain design can lead to different QoRs for their corresponding gate-level netlist. We can formulate this ranking problem by the following: *We can say that the Boolean circuits $C_0 \leq C_1$ only if the OoR of C_0 is better than C_1 , otherwise, $C_0 \geq C_1$. The QoR can be defined by the timing, area, power, or other criteria of the following EDA steps.*

However, the technology mapping, timing analysis, and other physical design steps are time-consuming. If we can find the best presentation of the current design, it can improve the efficiency of EDA. In this task, we only focus on the circuit ranking problem before the technology mapping step.

2. Dataset Adaptation

Task 2 part of Fig. 10 shows the components of the circuit ranking dataset. The partial order is defined as the following:

$$\begin{aligned}
 &Timing^{C_0} < Timing^{C_1} \Rightarrow C_0 \preceq C_1; \\
 &Timing^{C_0} > Timing^{C_1} \Rightarrow C_0 \succeq C_1; \\
 &Timing^{C_0} = Timing^{C_1}, \text{ then :} \\
 &\quad Area^{C_0} < Area^{C_1} \Rightarrow C_0 \preceq C_1; \\
 &\quad Area^{C_0} > Area^{C_1} \Rightarrow C_0 \succeq C_1; \\
 &\quad Area^{C_0} = Area^{C_1}, \text{ pass}
 \end{aligned}$$

Then, we construct the graph pairs and their partial order by the combination of the logic types as listed at Table II, and we only consider the $C_0 \leq C_1$ status as the $C_0 \geq C_1$ can be converted to $C_1 \leq C_0$. The tie condition of timing and area is also not considered.

3. Solution: Pair-wise Graph Ranking

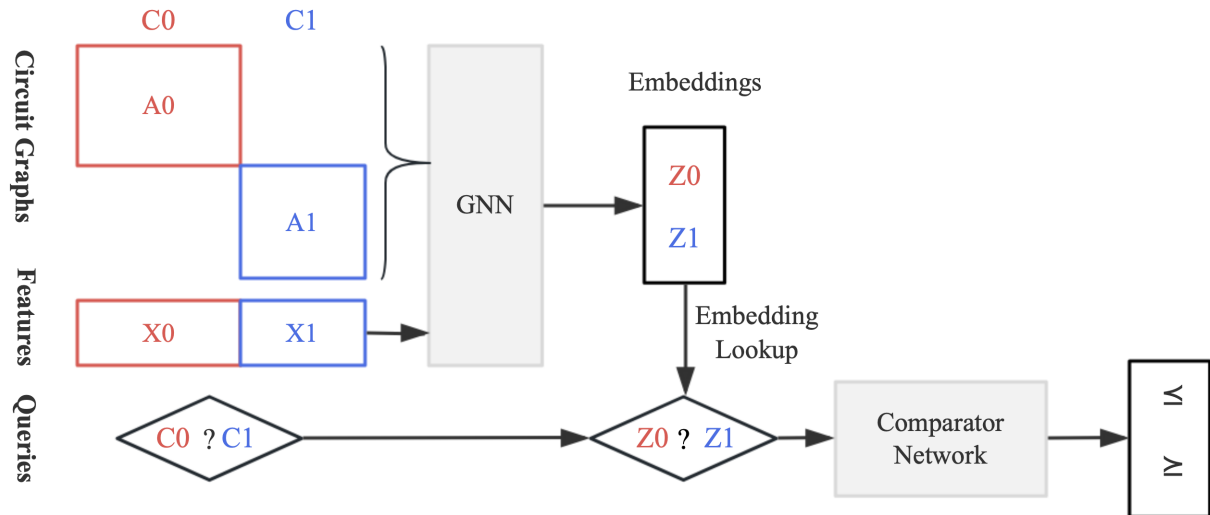
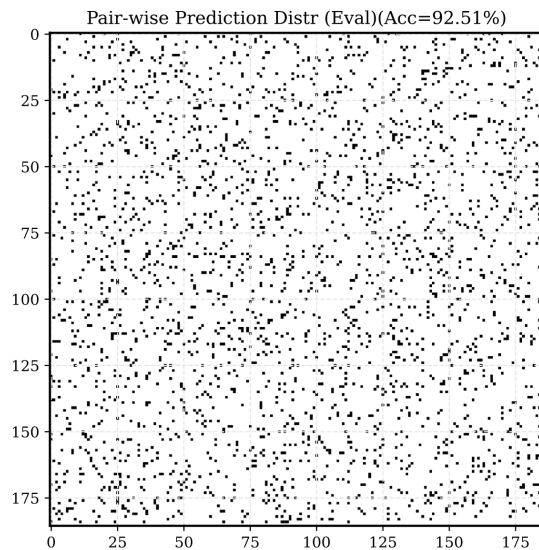


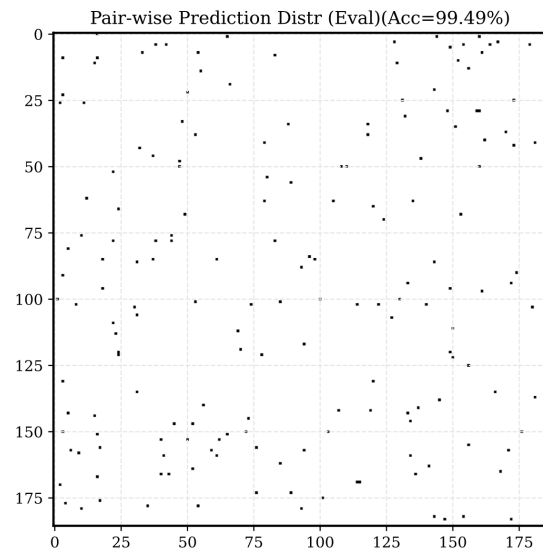
Figure 13. The pair-wise graph ranking model.

Fig. 13 illustrates the architecture of the pair-wise graph ranking model. We first combine these two Boolean circuits C_0 and C_1 into a block matrix. Then a GNN-based graph embedding will learn the combined embedding of these two circuits. Finally, the MLP-based comparison network will tell that $C_0 \leq C_1$ or $C_0 \geq C_1$; The binary cross entropy function is used as the loss function here.

4. Experimental Results



(a) Test, Epoch=1, Acc=92.51%



(b) Test, Epoch=20, Acc=99.49%

Figure 14. Pair-wise prediction distribution of the evaluation.

Table V. Performance comparison with different Graph Embedding Model (epoch = 20).

Metric	GCNConv	GraphSAGE	GINConv
BCE_loss	5.09	4.31	4.43
Accuracy	99.43%	99.49%	99.47%
Precision	0.9939	0.9949	0.9944
Recall	0.4993	0.4995	0.4969
F1-score	0.6647	0.6651	0.6627

The experiment presented here is based on a selection of 10 designs from the OpenLS-D-v1 dataset: *ctrl*, *router*, *int2float*, *ss_pcm*, *usb_phy*, *sasc*, *cavlc*, *simple_spi*, *priority*, and *i2c*. Using the previously defined partial order, approximately 120,000 pairs were extracted from OpenLS-D-v1 for analysis. Of each design's selected Boolean circuits, 70% were used for training, and the remaining for validation.

The hyperparameters used in this experiment are as follows: input feature size of 64, hidden feature size of 128, learning rate of 0.0001, learning decay rate of 1e-5, and batch size of 32. Fig. 14 illustrates the distribution of pair-wise prediction outcomes on the test dataset, with black nodes indicating incorrect predictions and white nodes indicating correct ones. The results indicate that all three models achieve high accuracy and effective pair-wise ranking predictions, demonstrating their validity for this application.

E. Task3: QoR Prediction

1. Problem Formulation

Fig. 8 and its corresponding observation 3 illustrate the motivation behind the QoR prediction task: once an adequate QoR distribution for a circuit is obtained, it is possible to make predictions about the inputs. *Given a known QoR distribution D , a Boolean circuit C , and an optimization sequence S , the objective is to predict the QoR of C with S .*

2. Dataset Adaptation

Task 3 part of Fig. 10 presents the profile of the sub-dataset used for QoR prediction. In the proposed framework, an ASIC gate-level netlist is generated for each optimized Boolean network. The Area and Timing are used as the QoR for the unoptimized Boolean networks and their respective optimization sequences. Consequently, each data item is organized as {unoptimized Boolean network, optimization sequence, Area, Timing}. Notably, due to the specific requirements of this task, different designs share the same optimization sequence within the same recipe index.

3. Solution: QoR Distribution Convergence Learning

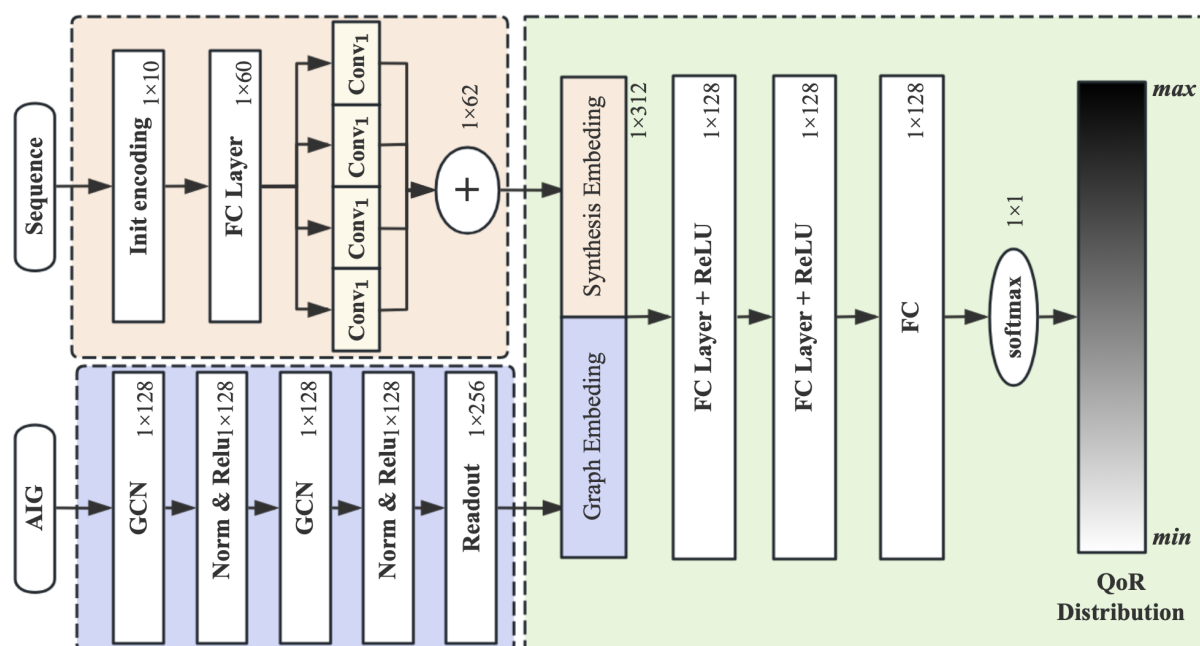


Figure 15. Convolutional networks for QoR synthesis formulation prediction.

Fig. 15 illustrates the proposed QoR prediction architecture. Each feature dimension is labeled in each layer. First, the input AIG and the optimization sequence are each embedded separately. The AIG is embedded using a standard GNN-based graph embedding approach, incorporating global mean and sum pooling as the readout layer. The optimization sequence, represented by numerically encoded synthesis recipes, is processed through a linear layer followed by four convolutional filters with dimensions $\{1 \times 14, 1 \times 15, 1 \times 16, 1 \times 17\}$, designed to extract relevant features. These two embeddings are then concatenated and fed into an MLP-based distribution learning module. Finally, a softmax activation function in the output layer predicts the position within the overall distribution, providing the QoR prediction.

4. Experimental Results

Table VI. MAPE Results for QoR Prediction (%)

Variant1 Seen Design, Unseen Recipe		Variant2 Unseen Design, Seen Recipes		Variant3 Unseen IC and Recipes	
Area	Timing	Area	Timing	Area	Timing
0.69	7.87	1.06	6.50	1.17	6.49

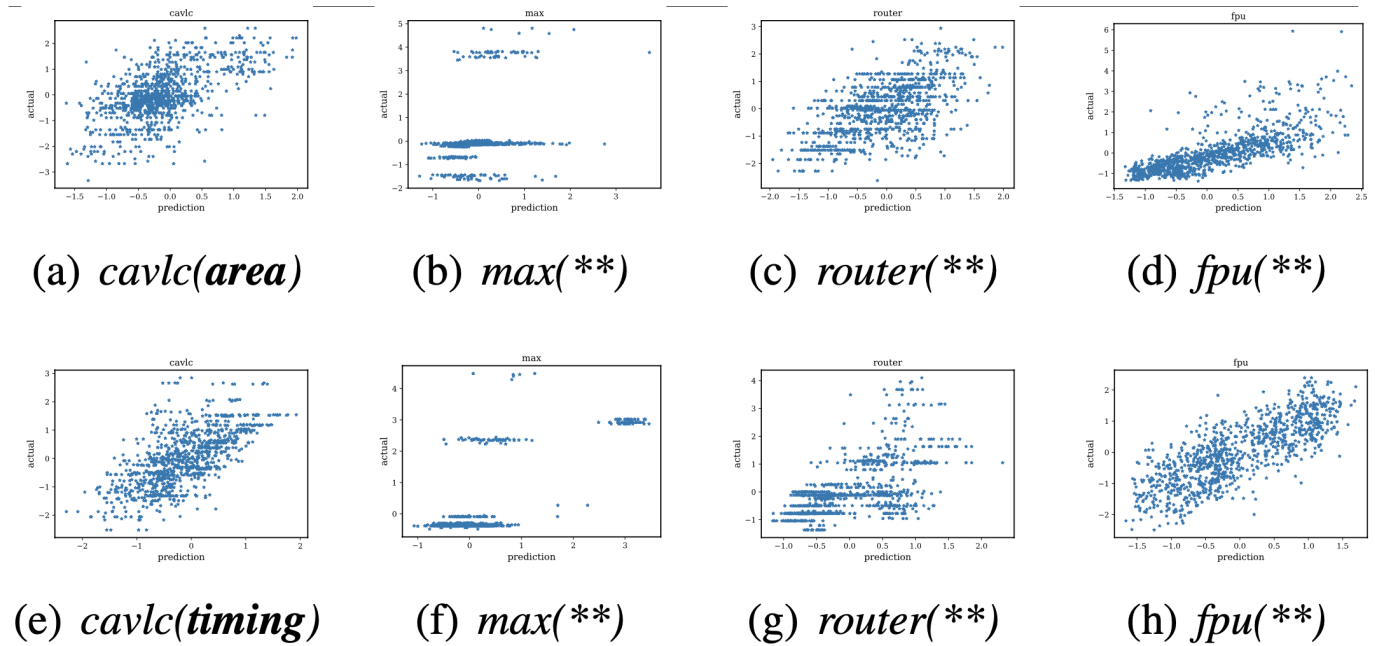


Figure 16. QoR predictions vs. ground truth of variant 1. (a)-(d) focus on the prediction of area, while (e)-(h) focus on timing.

There are three variants of the QoR prediction tasks, and the detailed configuration of each variant is outlined as follows:

- **Variant1:** Predicting QoR of synthesizing unseen Recipes. All designs and the AIG outputs of 700 synthesis recipes are used as the training set, and the remaining 300 recipes of each design are used as the test set;
- **Variant2:** Predicting QoR of synthesizing unseen Designs. We select 20 small designs as the training set, and the remaining 14 larger designs as the test set;
- **Variant3:** Predicting QoR on unseen Design-Recipe Combination. We randomly selected 70% of the synthesis recipes across all designs.

Table VI shows the mean absolute percentage error (MAPE) for the above three variant tasks. It indicates that the model achieved high predictive accuracy in area prediction, while timing prediction posed a challenge, with slightly lower accuracy compared to area prediction. This discrepancy may stem from the complexity of timing prediction, necessitating more refined feature engineering and model tuning. Overall, the model demonstrated good generalization performance with unknown circuit and optimization sequence pairs, indicating a certain level of robustness in the model. Fig. 16 visualizes the relationship between the prediction and ground truth.

F. Task4: Probabilistic Prediction

1. Problem Formulation

The Probabilistic Prediction task is a gate-level task that predicts the truth-table probability of the gate in the circuit^[20]. It can predict the logical probability of the gate without computing the truth table and can be formulated by: *Given a Boolean Circuit C, the logic probability of a gate v is defined as the frequency of the 1s in the gate's truth table.*

2. Dataset Adaptation

Task 4 part of Fig. 10 shows the dataset components of probability prediction. For each Boolean network in the OpenLS-D-v1 items, we label each Boolean network with a probability vector (the probability of each node) which is computed by random simulation of the Boolean network by “simulate(circuit:Circuit)” with enough activate vectors of PIs.

3. Solution: Node Embedding Learning

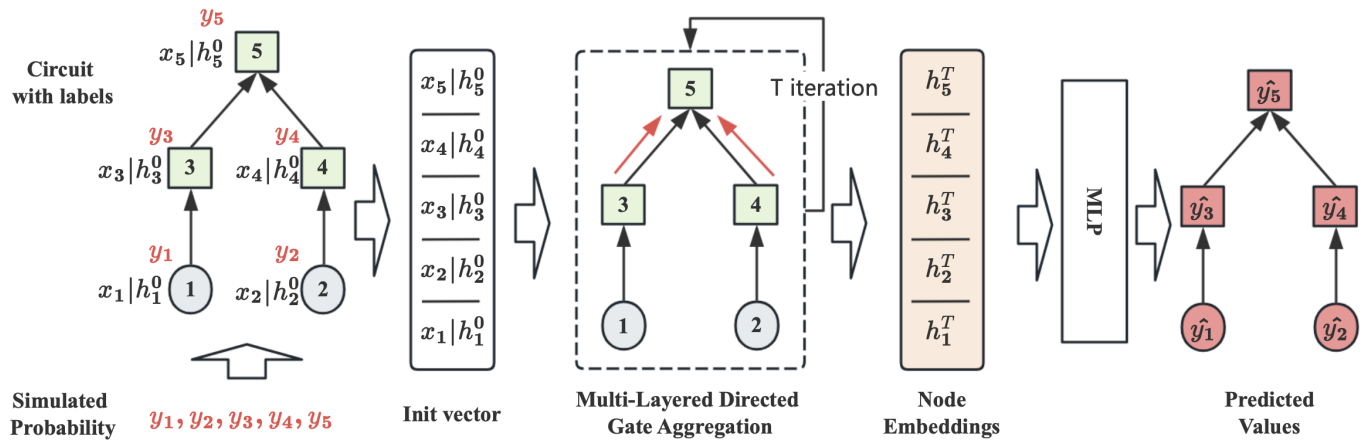


Figure 17. The node embedding model.

Instead of computational simulation of the circuit, the probability can be easily attained by the Node embedding-based method. Fig. 17 shows the node embedding learning-based probability prediction method. For a given Boolean network with the probability vector x_1, x_2, \dots, x_n , the first step is to initialize the node feature vector $h_1^0, h_2^0, \dots, h_n^0$ for the nodes; Then a T -layered directed gate aggregation is performed to learn the feature of each node. Finally, an MLP layer is to reduce each node's feature to 1 to predict the logic probability of each node. The loss function is the average prediction error (PE) loss, followed by DeepGate.

4. Experimental Results

Table VII. Average prediction error and time comparison by the different node embedding methods, while the comparison column represents DeepGate2/GraphSAGE.

	GraphSAGE		DeepGate2		Comparison	
	PE	Time(s)	PE	Time(s)	PE	Time(x)
100	0.011	0.050	0.0082	1.42	24% ↑	27.40 ↓
500	0.002	0.098	0.001	1.48	50% ↑	14.10 ↓
1000	0.0008	0.038	0.0002	2.20	75% ↑	56.89 ↓

The experiment presented here is based on a selection of 10 designs from the OpenLS-D-v1 dataset: *ctrl*, *router*, *int2float*, *ss_pcm*, *usb_phy*, *sasc*, *cavlc*, *simple_spi*, *priority*, and *steppermotordrive*. The recipe size of the dataset ranges from 100 to 1000, showing the scalability and adaptability of the dataset. A 70-30 split is used for training and validation, respectively.

The hyperparameters used in this experiment are as follows: input feature size of 64, hidden feature size of 128, learning rate of 0.001, learning decay rate of 1e-4, and batch size of 64 for fast training. The comparison between different methods can be seen in Table VII, where DeepGate outperforms the GraphSAGE model in various scales of the dataset.

VI. Discussion

Table VIII. The Tasks comparison between the logic synthesis-related Datasets.

Dataset	OpenABC-D ^[19]	Deepgate ^[20]	Gamora ^[11]	OpenLS-D-v1
Circuit Classification	-	x	x	-
Node Classification	x	x	-	-
QoR Prediction	-	x	x	-
Circuit Ranking	x	x	x	-
probability Prediction	x	-	x	-

The comparison table in Table VIII illustrates the comprehensive task support offered by the OpenLS-D-v1 dataset in contrast to other datasets such as OpenABC-D, DeepGate, and Gamora. Each dataset was developed with specific objectives, and they serve different roles in the enhancement of logic synthesis processes through machine learning.

OpenLS-D-v1 distinguishes itself by offering comprehensive task coverage, essential for developing versatile and generalized machine learning models within the logic synthesis domain. This wide-ranging support enables diverse experimental setups, paving the way for novel approaches to circuit design and analysis within a unified dataset framework. This broad task support enables researchers to explore new methodologies and improve existing processes.

In summary, OpenLS-DGF is capable of supporting a variety of machine learning tasks highlighting its potential as a general resource and standardized process in the field of logic synthesis. Additionally, the OpenLS-D-v1 dataset further enhances this by providing a versatile foundation for future research and innovation.

VII. Conclusion

In this paper, we begin by addressing the lack of a dataset generation flow specifically targeting multiple tasks in logic synthesis. To overcome this limitation, we propose OpenLS-DGF, an adaptive dataset generation framework tailored for machine learning tasks within logic synthesis. We highlight that the proposed solution framework can target multiple

machine-learning tasks in logic synthesis. We also generate the OpenLS-D-v1 dataset, created using OpenLS-DGF, and demonstrate its utility by implementing and evaluating four typical tasks on OpenLS-D-v1. The results of these tasks validate the effectiveness and versatility of our framework.

Future work will focus on enhancing the efficiency of the generation flow and benchmarking the specific machine-learning tasks for logic synthesis. Furthermore, we aim to integrate the machine learning models into the logic synthesis flow, contributing to an improved EDA flow for enhanced circuit performance.

Acknowledgment

The authors would like to express their gratitude to the teams behind the related open-source tools, including Yosys^[26], ABC^[27], LSILS^[28], OpenRoad^[22], iEDA^[32], and LogicFactory^[25].

Footnotes

¹ Primitive gates are composed of: NOT, BUFFER, AND2, NAND2, OR2, NOR2, XOR2, XNOR2.

References

- ¹ Huang G, Hu J, et al. "Machine Learning for Electronic Design Automation: A Survey." 2021. [Online]. Available: <https://arxiv.org/abs/2102.03357>.
- ² Rai S, Neto WL, et al. "Logic Synthesis Meets Machine Learning: Trading Exactness for Generalization." 2020. Available from: <https://arxiv.org/abs/2012.02530>.
- ³ Haaswijk W, Collins E, et al. "Deep Learning for Logic Optimization Algorithms." In: 2018 IEEE International Symposium on Circuits and Systems (ISCAS). 2018. pp. 1-4. doi:10.1109/ISCAS.2018.8351885.
- ⁴ Neto WL, Austin M, et al. "LSOracle: a Logic Synthesis Framework Driven by Artificial Intelligence: Invited Paper." In: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) 2019. p. 1-6. doi:10.1109/ICCAD45719.2019.8942145.
- ⁵ Pasandi G, Pratty S, Forsyth J. "AISYN: AI-driven Reinforcement Learning-Based Logic Synthesis Framework." 2023. Available from: <https://arxiv.org/abs/2302.06415>.
- ⁶ Ni L, Yang Z, et al. "Adaptive Reconvergence-driven AIG Rewriting via Strategy Learning." In: 2023 IEEE 41st International Conference on Computer Design (ICCD). IEEE; 2023. p. 336-343.
- ⁷ Neto WL, Moreira MT, et al. "SLAP: A Supervised Learning Approach for Priority Cuts Technology Mapping." In: 2021 58th ACM/IEEE Design Automation Conference (DAC), 2021. pp. 859-864. doi:10.1109/DAC18074.2021.9586230.
- ⁸ Liu J, Ni L, et al. "AiMap: Learning to Improve Technology Mapping for ASICs via Delay Prediction." In: 2023 IEEE 41st International Conference on Computer Design (ICCD). IEEE; 2023. p. 344-347.

9. ^aWang P, Lu A, et al. "EasyMap: Improving Technology Mapping via Exploration-Enhanced Heuristics and Adaptive Sequencing." In: *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*; 2023. p. 01-09. doi:[10.1109/ICCAD57390.2023.10323703](https://doi.org/10.1109/ICCAD57390.2023.10323703).
10. ^aAlrahis L, Sengupta A, et al. "GNN-RE: Graph Neural Networks for Reverse Engineering of Gate-Level Netlists." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. **41** (8): 2435-2448, 2022. doi:[10.1109/TCAD.2021.3110807](https://doi.org/10.1109/TCAD.2021.3110807).
11. ^{a, b, c, d}Wu N, Li Y, et al. "Gamora: Graph Learning based Symbolic Reasoning for Large-Scale Boolean Networks." 2023. Available from: <https://arxiv.org/abs/2303.08256>.
12. ^aBrglez F, Fujiwara H. "A neutral netlist of 10 combinational benchmark circuits and a targeted translator in FORTRAN." In: *ISCAS*; 1985 Jun.
13. ^aBrglez F, Bryan D, Kozminski K (1989). "Combinational profiles of sequential benchmark circuits." In: *1989 IEEE International Symposium on Circuits and Systems (ISCAS)*. pp. 1929-1934 vol.3. doi:[10.1109/ISCAS.1989.100747](https://doi.org/10.1109/ISCAS.1989.100747).
14. ^aYang S. "Logic Synthesis and Optimization Benchmarks User Guide Version 3.0." tech report microelectronics centre of north carolina. 1991.
15. ^aMcElvain K. *IWLS'93 Benchmark Set: Version 4.0. Distributed as a part of IWLS'93 benchmark set*, Tech. Rep., May 1993.
16. ^{a, b, c}Albrecht C. "IWLS 2005 benchmarks." In: *IEEE International Workshop for Logic Synthesis (IWLS)*; 2005.
17. ^{a, b, c}Amaru00fa L, Gaillardon P-E, De Micheli G. The EPFL combinational benchmark suite. In: *IEEE International Workshop on Logic & Synthesis (IWLS)*; 2015.
18. ^{a, b, c}OpenCores. Available from: <https://opencores.org/>.
19. ^{a, b, c, d}Chowdhury AB, Tan B, Karri R, Garg S. "OpenABC-D: A Large-Scale Dataset For Machine Learning Guided Integrated Circuit Synthesis." 2021. Available from: <https://arxiv.org/abs/2110.11292>.
20. ^{a, b, c, d}Li M, Khan S, Shi Z, Wang N, Yu H, Xu Q. "DeepGate: learning neural representations of logic gates." In: Oshana R, editor. *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*. ACM; 2022. p. 667-672. Available from: <https://dblp.org/rec/conf/dac/0019KSWY022.bib>.
21. ^{a, b}Shalan M, Edwards T (2020). "Building OpenLANE: A 130nm OpenROAD-based Tapeout-Proven Flow : Invited Paper." In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)* pp. 1-6.
22. ^{a, b}Ajayi T, Chhabria VA, et al. Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project. In: *Proceedings of the 56th Annual Design Automation Conference 2019. DAC '19*. New York, NY, USA: Association for Computing Machinery; 2019. doi:[10.1145/3316781.3326334](https://doi.org/10.1145/3316781.3326334).
23. ^aMishchenko A, Brayton R (2002). "Simplification of non-deterministic multi-valued networks." In: *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*. San Jose, California. p. 557-562. doi:[10.1145/774572.774654](https://doi.org/10.1145/774572.774654).
24. ^aDeng J, Dong W, Socher R, Li LJ, Li K, Fei-Fei L (2009). "ImageNet: A large-scale hierarchical image database." In: *2009 IEEE Conference on Computer Vision and Pattern Recognition* pp. 248-255. doi:[10.1109/CVPR.2009.5206848](https://doi.org/10.1109/CVPR.2009.5206848).

25. ^{a, b}Ni L. "LogicFactory: An Open-source Logic Synthesis Platform for Integrated Cross-tool Flow." Available from: <https://github.com/Logic-Factory/>.
26. ^{a, b}Wolf C. Yosys Open SYnthesis Suite. Available from: <https://yosyshq.net/yosys/>.
27. ^{a, b}Brayton R, Mishchenko A. "ABC: An Academic Industrial-Strength Verification Tool." In: Touili T, Cook B, Jackson P, editors. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010. p. 24-40.
28. ^{a, b, c}Soeken M, Riener H, et al. "The EPFL logic synthesis libraries." *arXiv*. Jun 2022. Available from: [arXiv:1805.05121v3](https://arxiv.org/abs/1805.05121v3).
29. ^aGrosnit A, Malherbe C, et al. "BOiLS: Bayesian Optimisation for Logic Synthesis." In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*; 2022. p. 1193-1196. doi:[10.23919/DATE54114.2022.9774632](https://doi.org/10.23919/DATE54114.2022.9774632).
30. ^aHosny A, Hashemi S, Shalan M, Reda S (2020). "DRiLLS: Deep Reinforcement Learning for Logic Synthesis." In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. pp. 581-586. doi:[10.1109/ASP-DAC47756.2020.9045559](https://doi.org/10.1109/ASP-DAC47756.2020.9045559).
31. ^aGoogle. SkyWater SKY130 PDK [Internet]. Available from: <https://github.com/google/skywater-pdk>.
32. ^{a, b}Li X, Tao S, et al. iEDA: An Open-Source Intelligent Physical Implementation Toolkit and Library [Internet]. 2023 [cited 2023]. Available from: <https://arxiv.org/abs/2308.01857>.
33. ^aFey M, Lenssen JE (2019). "Fast Graph Representation Learning with PyTorch Geometric". *arXiv*. Available from: <https://arxiv.org/abs/1903.02428>.
34. ^aHagberg AA, Schult DA, Swart PJ. "Exploring Network Structure, Dynamics, and Function using NetworkX." In: Varoquaux G, Vaught T, Millman J, editors. *Proceedings of the 7th Python in Science Conference, 2008; Pasadena, CA USA*. p. 11-15. Available from: <https://www.bibsonomy.org/bibtex/20f27dbabe1d5d95dafdead4d87c0bc16/alueschow>.
35. ^aNarayanan A, Chandramohan M, Venkatesan R, Chen L, Liu Y, Jaiswal S (2017). "graph2vec: Learning Distributed Representations of Graphs". *arXiv*. Available from: <https://arxiv.org/abs/1707.05005>.
36. ^aFacebook. Zstandard: A fast lossless compression algorithm. Available from: <https://github.com/facebook/zstd>.