2023 IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS)
Phoenix, Arizona, USA, August 6-9, 2023

RELUT-GNN: Reverse Engineering Data Path Elements From LUT Netlists Using Graph Neural Networks

Kishore Pula, Aparajithan Nathamuni Venkatesan,
Ram Venkat Narayanan, Sundarakumar Muthukumaran, Ranga Vemuri and John Emmert

Digital Design Environments Lab, ECE Department
University of Cincinnati, Cincinnati, Ohio, USA
pulake@mail.uc.edu, nathaman@mail.uc.edu,
narayart@mail.uc.edu, muthuksr@mail.uc.edu, vemurir@ucmail.uc.edu, emmertj@ucmail.uc.edu

Functional reverse engineering of flattened Field Programmable Gate Array (FPGA) Look-Up Table (LUT) netlists to Register Transfer Level (RTL) representation is essential to understand, reconstruct and enhance the existing legacy designs. Recent advances in machine learning show promising results in solving EDA problems. In this paper, we propose a tool, RELUT-GNN that uses Graph Neural Networks (GNNs) to extract high-level functionality of data path elements from LUT-level netlists. For GNNs, the netlist structure is represented as a graph with FPGA leaf cells as nodes and the nets among them as edges. We extract features for each node and train the GNN to learn the structure of the netlist by aggregating their node features and their neighbors. The training dataset includes a comprehensive custom dataset consisting of various Operators, Shifters, Counters, FSMs, and their combinations of varying bit widths. The model is validated and tested on unseen real-world designs obtained from Opencores and ITC99. It is observed that RELUT-GNN achieved a combined accuracy of 97.12% for the classification of selected benchmarks from arithmetic and DSP cores and the ITC'99 benchmarks.

I. Introduction

With the globalization of Integrated Circuit (IC) manufacturing, the supply chain involves several third-party manufacturers and suppliers. Untrusted entities could counterfeit ICs, illegally overproduce ICs, steal Intellectual Property (IP), etc. This lack of trust motivates designers to develop hardware security techniques. Reverse Engineering (RE) is one such technique that aids in detecting IP infringement, and functionality verification [1]. FPGA-based designs can be reverse-engineered in two phases. The first phase is netlist extraction from the bitstream and the second phase is specification discovery where the functionality of the design is extracted from the netlist [2]. In this paper, we focus on the specification discovery part of the reverse engineering of FPGA netlists.

FPGA architecture consists of Configurable Logic Blocks (CLBs), programmable interconnects, LUTs, Carry modules, and input/output blocks. We assume that the flattened LUT-level netlist has been extracted from the FPGA bitstream using a tool such as Project X-ray [3]. Design hierarchy, bit-slicing information is assumed to be unavailable in the flattened LUT-level netlists. Cross-optimized logic across module boundaries can be absorbed in LUTs. These aspects make FPGA netlist reverse engineering a challenging problem.

In this paper, we extracted features specific to LUT netlists that capture structural and functional information about FPGA design elements. We then showed how GNNs can be applied to LUT netlists. RELUT-GNN achieved a combined accuracy of 97.12% for classifying datapath elements in real-world designs from opencores.org [4]. We created a toolchain that produces a high-level representation of identified components.

II. Related Work

Alrahis et al. [5] used GNNs to identify module boundaries from flattened gate-level netlists. The authors created custom benchmarks and extracted features such as gate type, gate degree, and neighborhood information for each node. The authors used GraphSAINT [6] platform for supervised node classification. GNN-RE is evaluated on selected 74X series, EPFL, and ISCAS-85 benchmarks. GNN-RE achieved an average accuracy of 98.82% in detecting sub-circuits. Subajit et al. [7] proposed a technique to identify state registers in the gate-level netlist using GNNs. In addition to the node features specified in GNN-RE [5], the authors extracted betweenness centrality and harmonic centrality measures for each node. The classification accuracy is further improved by applying strongly connected components algorithm. ReIGNN achieved 96.5% average balanced accuracy and 97.7% sensitivity.

Narayanan et al. [8] created a toolchain to reverse engineer FPGA netlists by using a novel carry chain analysis technique. The authors compared the subcircuits with golden library components to detect operators such as adders, subtractors, comparators, and ALUs. To detect sequential components, the

979-8-3503-0210-3/23/$31.00 ©2023 IEEE

authors created a flip-flop dependency graph and analyzed the data flow. In the case of real-world designs, the toolchain provides 34% to 100% gate coverage. Venkatesan et al. [9] organized elements such as LUTs, CARRYs, RAM blocks, and registers using proximity information on cells in the implemented design. For real-world designs, the algorithm groups elements with a Normalized Mutual Information (NMI) metric of 0.73. However, to our knowledge, no one has attempted to adapt GNN techniques to reverse engineer LUT-based netlists.

III. Datapath Elements Identification in LUT Netlists using GNNs

To detect datapath elements in the design, we created a rich custom dataset with varying bit widths. We extracted the structural and functional features of each design element in the LUT netlist. We trained RELUT-GNN on custom benchmarks and cross-validated it. Then, we selected the best-performing model to evaluate real-world designs. Lastly, we applied post-processing techniques to extract a high-level representation of the design.

A. Dataset Generation

We analyzed a wide variety of real-world benchmarks and identified a list of designs that most frequently appears. Some of the examples of such designs are adders, subtractors, multipliers, comparators, counters, etc. In addition to the designs specified in GNN-RE [5] and ReIGNN [7], We added the following custom designs to our training dataset.

1) ADD-SUB: This design contains one adder and one subtractor module, and the output of the design is chosen based on a select line. From this design, RELUT-GNN learns to differentiate between adder, subtractor, and add-sub modules. This design is most common in arithmetic cores [4].

2) MUL-AND-ACC: This design contains one multiplier and one adder where two words are multiplied and the result is added to an accumulator. From this design, RELUT-GNN learns to differentiate between adder and multiplier structures. This design is most common in DSP cores [4].

3) CROSS-OPTIMIZED: Cross-optimization in logic synthesis is defined as the concept of logic sharing across different modules. In real-world designs, the logic across module boundaries can be cross-optimized and absorbed into LUTs. This category of designs is generated by interspersing multiple operators such that each LUT can belong to one or more operators.

B. Netlist to Graph Conversion and Feature Extraction

We converted LUT netlists into undirected graphs. The nodes of the graph obtained are the design elements and the edges of the graph are the connections between the design elements. An example netlist is shown in Fig 1. We classified FPGA design primitives, thus IBUFs and OBUFs are not shown in the converted graph.

We extracted a comprehensive list of structural and functional features for each node in the graph. An example feature

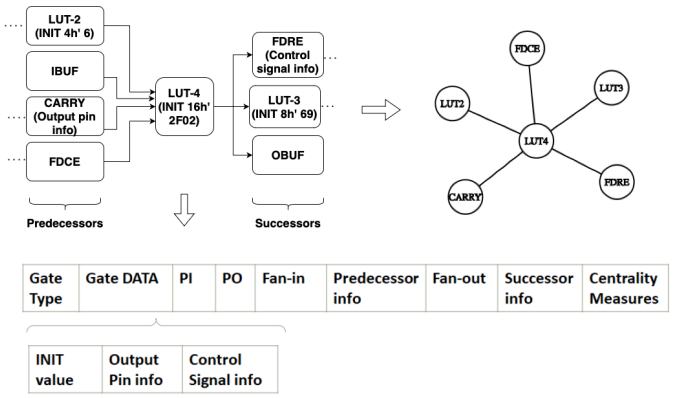

Fig. 1: FPGA netlist to graph conversion and feature extraction

vector of the LUT4 node is shown in Fig 1. The description of each feature is given below.

1) Gate type (gt): One hot encoding of the list of primitives in the Xilinx 7-series FPGA. Each bit location indicates a type of FPGA primitive. A '1' indicates a presence of a certain design element and a '0' indicates the absence of the same element.

2) Gate DATA (gd): Unlike ASIC netlists, it is not enough to capture just the gate type. In FPGA netlists, each primitive performs a certain function. i) LUT primitives are used in generating complex digital logic and the functionality information is attached in a format called INIT string. Thus, we captured the INIT string as a feature. ii) CARRY primitives perform fast carry logic which improves the performance of the overall design [10]. Narayanan et, al. [8] discussed in-depth how the carry chains can be used to extract high-level functionality. Thus capturing carry module information is critical. We captured the output pin information of the CARRY primitive in a binary encoded format since each output pin performs a certain function based on inputs. iii) For flops, we captured the control information such as the connectivity of the inputs and outputs of flops to other primitives. The number of combinational and sequential elements in the neighborhood of the flop is also found. iv) Each Shifter in FPGA is associated with its depth and it is captured as a feature.

3) PI, PO (io): This field gives the number of primary inputs (IBUFs) and outputs (OBUFs) in the one-hop neighborhood of the target node. Since we captured this information as a feature, there is no information loss by converting LUT netlists to undirected graphs.

4) Neighborhood Information (ni): This feature captures fan-in, fan-out, predecessor, and successor gate information of one-hop neighbors of the target node. Fan-in is the total number of predecessors of the target node and fan-out is the total number of successors of the target node. We captured the gate type and gate data information for all predecessors and successors of the target node.

5) Centrality Measures (cm): In graph theory, centrality measures give information about the data flow among nodes in the graph. This information is essential in identifying state

registers in the design. In addition to the measures mentioned in the ReIGNN [7], we added closeness and degree centrality of the node.

Definition 1: Closeness Centrality [11] is defined as how close the target node x is to all nodes in the design and is shown in equation 1, where $S(x, y)$ is the shortest path between nodes x and y.

$$C_C(x) = \frac{1}{\sum_{y \neq x} S(x, y)} \tag{1}$$

Definition 2: Degree centrality [11] is defined as how many edges are connected to the target node x and is shown in equation 2, where $deg(x)$ is the degree of node x and n is the total number of nodes in the design.

$$C_D(x) = \frac{deg(x)}{n - 1} \tag{2}$$

C. GNN Classification

In our experiment, we used a graph learning pipeline called GraphSAINT [6]. GraphSAINT uses a graph sampling-based inductive learning mechanism to extract subgraphs from the input graph. It then builds a complete GNN for each retrieved subgraph. Once the GNN model is trained using the GraphSAINT framework, the trained GNN model is evaluated on real-world designs. An example classification of flattened LUT netlist is shown in Fig 2.

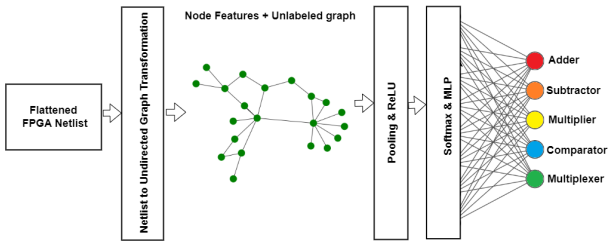

Fig. 2: Supervised node classification technique for identifying operators

D. RELUT-GNN Tool Flow

The RELUT-GNN tool shown in Fig 3 is built on top of the tool discussed in Narayanan et al. [8]. The GNN classification result is post-processed using techniques such as register grouping and majority voting algorithms. We used DANA [12] API from HAL [13] tool to group data registers. To more accurately identify nodes from adders and multipliers in the design, we applied a majority voting algorithm. Then the detected datapath elements are given to an RTL writer to generate a high-level representation of the netlist. This tool eliminates the necessity to establish module boundaries and the hassle of analyzing carry chains in the design.

IV. Evaluation

The tests were carried out on an Intel Core i7 processor with 16GB RAM. We used HAL [13] tool to parse the LUT netlist and extract features for each node. GraphSAINT framework [6] is used for developing GNN models. Python is used for writing scripts run by the HAL tool, employing the GraphSAINT framework, and developing the overall tool.

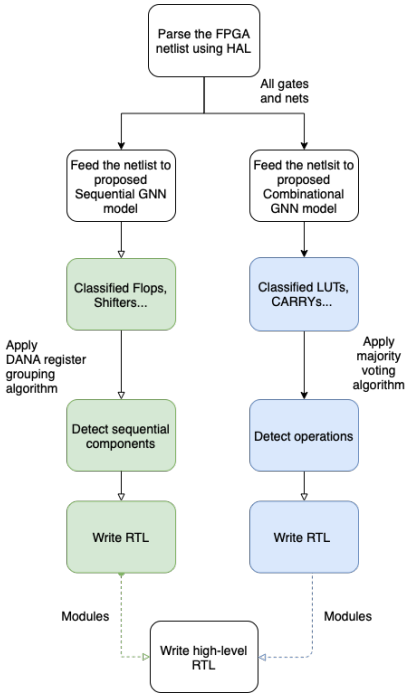

Fig. 3: RELUT-GNN tool workflow showing high-level functionality extraction of the design by integrating different techniques

A. Dataset and Feature Importance

Our original training dataset contains stand-alone, interconnected, and cross-optimized designs that are synthesized using Xilinx Vivado targeting Artix 7-series FPGAs [14]. We have set the *flatten_hierarchy* to *full*, *max_bram*, and *max_dsp* flags to *zero* while leaving other synthesis options as is. We removed all original net names that may have indicated the design information and replaced them with unique random names. The bit-width of operands in the designs varies from 8 to 128 bits.

We calculated the feature set importance by progressively adding one feature at a time. The prediction accuracy of the model is observed for the same. It is evident from Table I that the accuracy of the model is higher after including the neighborhood information in the feature vector.

TABLE I: Feature Importance

Feature Set	Subset Accuracy (%)
gt	42.8
gt + io	56.7
gt + io + gd	91.3
gt + io + gd + ni	97.8
gt + io + gd + ni + cm	97.1

B. GNN Model Parameters

For this experiment, a random walker sampler is selected to generate subgraphs. We employed two Graph Convolutional Network (GCN) layers with mean aggregators. Each layer has 256 hidden dimensions and a ReLU activation function is added to operate on these subgraphs. The Adam optimization approach is used to train the GNN. It uses an initial learning rate of 0.01 and a dropout rate of 0.1 for 1000 epochs. The final layer consists of five fully connected layers for combinational and four for sequential GNN models. A softmax activation function is used for single-label classification and Sigmoid for multi-label classification. Multi-label classification is useful in detecting cross-optimized modules. We used a 70/10/20 split for training/validation/test datasets.

C. Performance metrics

Our model is evaluated on a variety of real-world designs from OpenCores [4] that range from Arithmetic cores to DSP cores and ALUs. Table II shows the list of benchmarks used, the number of gates in the design, and the metrics such as precision, recall, f1 score, and sub-set accuracy for each benchmark. A "–" in the table shows the design doesn't contain that class.

The node classification outcome of the GNN model can be seen as follows: True positives (TP), the node is correctly classified as a positive outcome; False Positives (FP), the node is falsely classified as a positive outcome; similarly one can define True Negatives (TN) and False Negatives (FN). The column *Precision %* shows the ratio of TP to the total number of predicted positive outcomes (TP + FP), if the precision measure is high that means the model does not predict a negative outcome as a positive outcome. The column *Recall %* shows the ratio of TP to the total number of actual positive outcomes (TP + FN), if this measure is high that means the model correctly predicts a node as a positive outcome. The column *F1-Score %* shows the harmonic mean of precision and recall

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

which is the balanced measure of both metrics, if this measure is high that means the model performs well in all outcomes. Subset accuracy is defined as the number of correctly classified nodes to the total number of nodes in all classes.

D. Detection of Combinational Sub-circuits

We can infer from Table II that RELUT-GNN detects adder and subtractor structures well in all the designs. However, in the designs containing multiplier structures such as **Quadrature Oscillator**, **FIR filter**, and **PID Controller**, the accuracy of the adders are low i.e. 93.33%, 78.68%, and 91.3% respectively. This is because multipliers involve certain addition operations. The nodes which are part of the addition operation are incorrectly classified as multipliers. A majority voting algorithm can be used to increase the accuracy of the model. In LUT netlists, the multiplexer nodes contain logic from different operators. To detect them we employed a multi-label classification method that performed well on **fixed point arithmetic** and **Cordic polar2rect** designs. However, if the complexity of the design is increased, many operators can be cross-optimized which poses a limitation to GNN node classification. Benchmarks **Cordic polar2rect** and **Cordic rect2polar** contain add-sub modules and comparator structures which are detected with an accuracy of 98.87% and 96.51%. For designs containing multiplier structures, we generated a low-level representation of the design in terms of FPGA primitives instead of a high-level representation.

E. Detection of Sequential Sub-circuits

Table III shows the number of flip-flops in each benchmark and the same metrics as described in the previous section are calculated. The GNN model for sequential modules only applies to the sequential elements in the design. We collect features such as control signal information and centrality measures for each sequential element. The GNN model learns the structure and data flow in the design. It can be inferred from Table III that Benchmarks 1–5 show the detection of *Data registers, Counters,* and *Shifters*, Benchmarks 6–10 show the detection of single FSM controllers taken from ITC'99 [15]. Shifters in all the designs are identified with high accuracy and this is because FPGA shifter primitives such as SRL32, SRL64, etc. are unique compared to the FPGA flip-flop primitives. Counters in the benchmarks **DDS Synthesizer**, **FIR filter**, **CIC filter** are detected with good accuracy i.e. 97.3%, 94.78%, and 96.29% respectively. This shows that RELUT-GNN utilizes the structural similarity of counters and accurately classifies them. State registers are also identified with high accuracy for the designs **ITC b01**, **ITC b03**, **ITC b04**.

F. Comparison with State-of-the-art GNN Techniques

We synthesized the benchmarks using Xilinx Vivado for comparison with other state-of-the-art GNN techniques. When tested with EPFL, ISCAS-85 benchmarks, RELUT-GNN achieved an average accuracy of 99.7% in classify nodes which is slightly higher than the GNN-RE [5] accuracy. The reason for high classification accuracy is that the benchmarks contains stand-alone operations and the LUT netlist sizes are much smaller compared to ASIC netlists. We also compared RELUT-GNN performance with ReIGNN [7] for state register identification. As shown in Fig 4, RELUT-GNN achieved better accuracy for the benchmarks fsm (86.2%), gcm_aes (98.3%) and uart (91.8%) compared to ReIGNN. This can be explained by the fact that we are performing a four-way classification for the sequential circuits, and the counters are classified correctly in these designs. The registers in counter designs are misclassified as state-registers in ReIGNN. Also, the performance of other designs can be improved by applying structural analysis techniques.

The toolchain discussed in section III-D is used to produce a high-level representation of the designs and the functionality of the extracted design is verified using Cadence Jaspergold.

TABLE II: Detection of Combinational Datapath elements

No.	Benchmarks	Total no. of gates	Precision (%)					Recall (%)					F1-Score (%)					Subset Accuracy (%)
			Adder	Subtractor	Multiplier	Comparator	Mux	Adder	Subtractor	Multiplier	Comparator	Mux	Adder	Subtractor	Multiplier	Comparator	Mux	
1	32-bit ALU	171	93.18	97.62	–	–	86.84	97.62	95.35	–	–	100	95.34	96.47	–	–	92.96	95.1
2	DDS synthesizer	209	100	–	–	75	81.8	95.5	–	–	93.8	90	97.7	–	–	83.4	85.7	95.8
3	Fixed point arithmetic	220	100	100	–	98.63	100	100	100	–	100	96.77	100	100	–	99.31	98.36	99.54
4	CIC filter	332	100	100	–	–	–	97.46	100	–	–	–	98.71	100	–	–	–	98.38
5	Quadrature Oscillator	379	100	100	92.34	–	100	87.5	100	100	–	100	93.33	100	96	–	100	95.49
6	FIR filter	469	80	–	99.62	–	–	77.4	–	95.97	–	–	78.68	–	97.76	–	–	95.82
7	Hilbert transformer	871	100	95.8	–	–	–	84.1	100	–	–	–	91.3	97.87	–	–	–	97.05
8	PID Controller	1399	100	100	97.38	–	94.44	76.19	80.76	99.66	–	94.44	86.48	89.36	98.51	–	94.44	97.65
9	Cordic polar2rect	1602	99.56	99.61	–	84.61	100	99.27	98.98	–	100	76.92	99.42	99.29	–	91.66	86.95	98.87
10	Cordic rect2polar	2331	96.23	95.98	–	97.5	79.66	99.7	99.5	–	98.9	97.91	97.96	97.7	–	98.2	87.85	96.3
		Averages	96.90	98.63	96.45	88.94	91.82	91.47	96.82	98.54	98.18	93.72	93.89	97.59	97.42	93.14	92.32	97.00

TABLE III: Detection of Sequential Datapath Elements

No.	Benchmarks	Total no. of Flops	Precision (%)				Recall (%)				F1-Score (%)				Subset Accuracy (%)
			State register	Data register	Counter	Shifter	State register	Data register	Counter	Shifter	State register	Data register	Counter	Shifter	
1	Quadrature Oscillator	32	–	100	–	–	–	100	–	–	–	100	–	–	100
2	32-bit LFSR	38	–	95.8	–	100	–	92.34	–	100	–	94.04	–	100	97.76
3	DDS synthesizer	63	–	81	99.5	–	–	78.4	95	–	–	79.6	97.3	–	92.36
4	FIR filter	111	–	92.3	96.7	100	–	94.8	93	100	–	93.53	94.78	100	95.8
5	CIC filter	167	–	98.4	95.6	100	–	91.6	97	100	–	94.49	96.29	100	97.3
6	ITC b01	19	100	100	–	–	100	100	–	–	100	100	–	–	100
7	ITC b02	12	100	50	–	–	85.7	100	–	–	92.33	66.67	–	–	92.8
8	ITC b03	54	100	100	–	–	100	100	–	–	100	100	–	–	100
9	ITC b04	37	100	100	–	–	100	100	–	–	100	100	–	–	100
10	ITC b05	47	97.6	100	–	–	100	96.9	–	–	98.8	98.4	–	–	97.51
		Averages	99.52	91.75	97.27	100.00	97.14	95.40	95.00	100.00	98.23	92.67	96.12	100.00	97.35

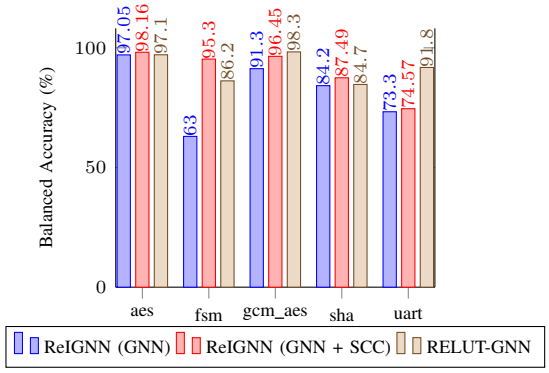

Fig. 4: ReIGNN vs RELUT-GNN performance [7]

V. Conclusions and Future Work

We proposed a tool chain RELUT-GNN that detects nodes belonging to datapath components such as adders, subtractors, multipliers, comparators, multiplexers, shift registers, counters, and FSMs in flattened LUT-level netlists using Graph Neural Networks. The correctness of the node classification on average is observed to be 97.12%. This paper presented a preliminary study on how cross-optimized modules pose a difficulty in LUT-based netlist to RTL reverse engineering. We intend to improve accuracy in netlists including cross-optimized modules and processors.

Acknowledgement

This work was supported in part by the National Science Foundation under IUCRC-1916762, and CHEST (Center for Hardware Embedded System Security and Trust) industry funding.

References

[1] M. Fyrbiak, S. Strauß, C. Kison, S. Wallat, M. Elson, N. Rummel, and C. Paar, "Hardware reverse engineering: Overview and open challenges," in *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, pp. 88–94, 2017.

[2] L. Azriel, J. Speith, N. Albartus, R. Ginosar, A. Mendelson, and C. Paar, "A survey of algorithmic methods in IC reverse engineering," *Journal of Cryptographic Engineering*, vol. 11, no. 3, pp. 299–315, 2021.

[3] "Project X-Ray." https://f4pga.readthedocs.io/projects/prjxray/en/latest/index.html.

[4] "The reference community for Free and Open Source gateware IP cores." https://opencores.org/.

[5] L. Alrahis, A. Sengupta, J. Knechtel, S. Patnaik, H. Saleh, B. Mohammad, M. Al-Qutayri, and O. Sinanoglu, "GNN-RE: Graph Neural Networks for Reverse Engineering of Gate-Level Netlists," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2435–2448, 2022.

[6] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "GraphSAINT: Graph sampling based inductive learning method," *arXiv preprint arXiv:1907.04931*, 2019.

[7] S. D. Chowdhury, K. Yang, and P. Nuzzo, "ReIGNN: State Register Identification Using Graph Neural Networks for Circuit Reverse Engineering," 2021.

[8] R. V. Narayanan, A. N. Venkatesan, K. Pula, S. Muthukumaran, and R. Vemuri, "Reverse Engineering Word-Level Models from Look-Up Table Netlists," *https://arxiv.org/abs/2303.02762*, 2023.

[9] A. Nathamuni-Venkatesan, R.-V. Narayanan, K. Pula, S. Muthukumaran, and R. Vemuri, "Word-Level Structure Identification In FPGA Designs Using Cell Proximity Information," in *2023 36th International Conference on VLSI Design and 2023 22nd International Conference on Embedded Systems (VLSID)*, pp. 389–394, 2023.

[10] S. Hauck, M. M. Hosler, and T. W. Fry, "High-performance carry chains for FPGAs," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pp. 223–233, 1998.

[11] A. Hagberg and D. Conway, "NetworkX: Network analysis with Python," *URL: https://networkx. github. io*, 2020.

[12] N. Albartus, M. Hoffmann, S. Temme, L. Azriel, and C. Paar, "DANA Universal Dataflow Analysis for Gate-Level Netlist Reverse Engineering," *Cryptology ePrint Archive*, 2020.

[13] S. Wallat, N. Albartus, S. Becker, M. Hoffmann, M. Ender, M. Fyrbiak, A. Drees, S. Maaßen, and C. Paar, "Highway to HAL: open-sourcing the first extendable gate-level netlist reverse engineering framework," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pp. 392–397, 2019.

[14] "Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide (UG953)." https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries.

[15] S. Davidson, "ITC-99 Benchmark Circuits - Preliminary Results," in *International Test Conference 1999. Proceedings (IEEE Cat. No.99CH37034)*, pp. 1125–1125, 1999.