SynC-LLM: Generation of Large-Scale Synthetic Circuit Code with Hierarchical Language Models

Shang Liu♦, Yao Lu♦, Wenji Fang, Jing Wang, Zhiyao Xie♣
The Hong Kong University of Science and Technology
{sliudx, yludf, wfang838, jwangjw}@connect.ust.hk, eezhiyao@ust.hk

Abstract

In recent years, AI-assisted integrated circuit (IC) design methods have shown great potential in boosting IC design efficiency. However, this emerging technique is fundamentally limited by the serious scarcity of publicly accessible large-scale circuit design data, which are mostly private IPs owned by semiconductor companies. In this work, we propose SynC-LLM, the first technique that exploits LLM's ability to generate new large-scale synthetic circuits. Our hierarchical circuit generation process includes three stages: 1) A directed graph diffusion model will learn to generate the *skeleton* of large circuits with sequential registers. 2) The expected function of the input cone of each sequential register will be annotated. Each cone, named *flesh*, consists of all combinational logic that controls the register value. 3) A level-by-level customized prompting technique will guide LLM to complete the design code of each cone. Experiments show that our generated circuits are not only valid and fully functional[1], but also closely resemble realistic large-scale designs and can significantly improve AI models' performance in multiple IC design tasks. The code and data are open-sourced in https://github.com/hkust-zhiyao/SynCircuitData.

1 Introduction

The advancement of artificial intelligence (AI) has led to unprecedented computational demand for hardware integrated circuits (ICs). However, the ever-increasing IC complexity and design costs are challenging traditional IC design methods. In recent years, AI-assisted IC design techniques have demonstrated great potential (Rapp et al., 2021; Chen et al., 2024; Xie et al., 2018; Liu et al., 2023b; Fang et al., 2025c,a,b). However, these AI-driven approaches are fundamentally constrained by the scarcity of circuit data. Such data scarcity is due to the proprietary nature of commercial IC designs as valuable intellectual properties (IPs) (Liu et al., 2025b,a). As a result, researchers only have access to small-scale and outdated circuit data, significantly hindering the development of new AI-driven IC design techniques.

♦ Equal contribution.
♣ Corresponding author.
[1]"Fully functional" means each generated circuit is a complete, self-contained design that is syntactically valid, simulatable, and will produce deterministic outputs for given inputs.

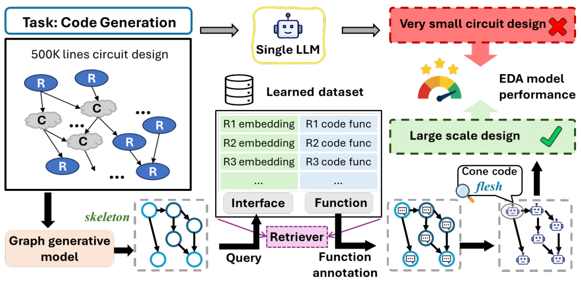


Figure 1: SynC-LLM is a novel hierarchical framework that generates large-scale and syntactically valid synthetic IC designs. SynC-LLM first learns both the sequential logic *skeleton* of realistic designs and then the detailed function *flesh* of all combinational logic. The generated synthetic IC designs can significantly enhance the performance of AI models in IC design tasks.

Most recently, LLMs are extensively applied in generating IC design in code format according to given design descriptions (Liu et al., 2023a; Chang et al., 2024; Liu et al., 2024a; Lu et al., 2024; Liu et al., 2024b; Li et al., 2025). However, as Table 1 shows, existing LLM solutions typically only generate very small-scale IC design components, from dozens to a few thousand gates after synthesis. Such a limited size implies the lack of scalability in LLM-assisted IC generation.

In this work, we propose **SynC-LLM**, a novel multi-stage framework that generates large-scale synthetic circuits. Our generated synthetic circuits reach millions of gates after synthesis, significantly exceeding the scale of existing works. Moreover, our generated synthetic circuits closely mimic realistic designs with complex functions. This ensures that ML models trained with our synthetic circuits apply to IC design tasks, such as the analysis and optimization of IC timing, power, and area.

SynC-LLM mainly comprises three stages to obtain large-scale hardware design code. In stage 1, a graph diffusion model will learn to generate the skeletal structure of the design. This skeleton is formulated as a directed graph that reflects the dependency and transition relationships among various digital states, capturing the overall logical functionality of the circuit. In stage 2, we utilize a retriever to annotate each register of the generated new skeleton with a functional description. The retriever is trained on real design data by aligning the skeleton's graph embeddings with the corresponding code function embeddings. This alignment effectively bridges the high-level structural information and the low-level code details, providing crucial contextual cues for subsequent code generation. In stage 3, each node's functional description is employed as a prompt input to an LLM to generate the corresponding cone code. In addition, we further introduce a logic-level control strategy within the prompt to regulate the timing length of the generated circuit. A syntax checker is applied to ensure that every code segment adheres to syntactic rules and embodies practical logical functionality. Finally, we integrate the cone codes according to the skeleton to obtain the complete design. This method not only facilitates data augmentation for downstream AI-assisted IC design tasks but also provides researchers with a robust benchmark dataset.

In summary, we propose a novel LLM-based approach for generating large-scale IC code. Contributions of SynC-LLM are summarized below:

- SynC-LLM proposes a new hierarchical generation strategy, considering both graph and natural language modalities. It first generates the overall sequential logic skeleton in graph modality and then generates combinational code. This strategy is highly scalable to support generating complex synthetic circuits.
- SynC-LLM proposes to generate the combinational code based on the skeleton structure. By providing structural function descriptions as context, LLMs generate more realistic combinational logic in synthetic circuits.
- SynC-LLM proposes layer-by-layer circuit generation prompts to enable precise control over path lengths and logic depths of generated synthetic circuits.
- Experiments demonstrate that SynC-LLM-generated circuits exhibit similar circuit statistics to real-world designs. Moreover, these synthetic circuits significantly improve the accuracy of AI-assisted IC design models.

Source Benchmark	**Design Scale (#K Gates) {Median, Max}**	**Design Scale (#K Lines) {Median, Max}**
RTLLM2.0 (Lu et al., 2024)	{1e-1, 2}	{5e-2, 5e-1}
VerilogEval (Liu et al., 2023a)	{1e-2, 2e-1}	{1e-2, 6e-2}
SynC-LLM	{ 22, 56}	{7, 17}

Table 1: Comparison of LLM-generated circuits. RTLLM and VerilogEval are de facto standard benchmarks in LLM for IC generation. SynC-LLM can produce much larger-scale hardware designs.

2 Related work

2.1 Data Scarcity in AI for IC Design

In recent years, AI-assisted IC design techniques have demonstrated remarkable potential in expediting the chip design process. Notable applications include automated chip design generation (Pei et al., 2024; Liu et al., 2023b), fast chip quality prediction (Fang et al., 2023; Xu et al., 2022; Zou et al., 2024), automated chip design planning (Bai et al., 2023), etc. A long-lasting bottleneck is the scarcity of open-source, large-scale circuit data, which significantly hampers data-driven AI advancements in IC design. Although several open-source circuits datasets (Chai et al., 2023; Jiang et al., 2023; Pan et al., 2023; Chowdhury et al., 2023; Liu et al., 2023a) have been proposed for various IC design tasks, these datasets mostly only help generate labels of *existing* open-source circuit designs, instead of generating brand-new circuit designs. The total number of valid circuits in the public domain is still very limited.

2.2 LLM for Circuits HDL Generation

Recently the automatic generation of IC design in hardware description language (HDL) based on LLMs has been widely explored (Lu et al., 2024; Liu et al., 2023b, 2024b; Pei et al., 2024; Liu et al., 2023a; Ho et al., 2024; Thakur et al., 2023). However, these methods are predominantly focused on relatively small-scale design tasks. In recent years, several works (Chhabria et al., 2021; Kim et al., 2021) have explored the generation of circuit datasets at the layout stage. However, these generative methods are limited to circuit layouts, without enforcing valid circuit designs. For tasks such as those involving logic synthesis (Xu et al., 2022; Fang et al., 2023, 2024), existing circuit generation methods are not directly applicable.

3 Problem Definition

Given a dataset of realistic circuit designs for training, SynC-LLM will learn to generate new synthetic designs of comparable scale. The generated synthetic designs will be in hardware description language (HDL) code format. Synthetic circuits should satisfy the following criteria:

1) *Syntax Validity*: The generated HDL code should first pass hardware syntax checking and can be successfully synthesizable into netlist by standard logic synthesis tools (e.g., Design Compiler).

2) *Structural Similarity*: After being synthesized into netlists, synthetic circuits should exhibit similar characteristics compared to realistic circuits.

3) *Data Augmentation Effectiveness*: The generated synthetic circuits should serve as augmented training data to enhance the performance of AI models in IC design tasks. In this way, they address the long-lasting data scarcity problem in AI-assisted IC design.

We respectfully clarify that specific generated circuit functions are not strictly required in our target application. The target of this work is to produce many large-scale synthetic circuits, providing training data for downstream AI-based IC tasks (such as power, performance, and area prediction in our experiment, which is fundamentally important in the digital chip design process). Therefore, this data generation application is different from LLM-assisted coding (e.g., Copilot), which strictly enforces correct functionalities. This application focuses more on the structural characteristics (such as timing features), similarity to realistic designs, diversity, and benefit to trained AI models.

4 SynC-LLM Overview

Circuit Modeling: Skeleton and Flesh. Digital IC is typically designed with HDL code such as Verilog. After parsing the structural information in HDL, each design can be modeled as a graph of word-level elements in both sequential and combinational logic (Fang et al., 2023). Let $S = \{s_1, s_2, \ldots, s_n\}$ represent the set of sequential cells (i.e., registers) and input/output (IO) ports. Such a collection S maintains and represents the circuit's state at each clock cycle. We refer to the graph G of all registers and IO ports[2] as the circuit design's ***skeleton***[3].

For any register s_i in this skeleton, its value depends on the values of all its parent registers. There is only combinational logic between its parent registers (as inputs) and s_i itself (as output). This multi-input and single-output subcircuit filled with combinational logic is referred to as the input **cone** of register s_i. The union of all input cones of all registers constitutes the design's ***flesh***.

Based on the formulation with *skeleton* and *flesh*, we propose a hierarchical generation scheme that incorporates both graph and text modalities to generate large-scale circuit code. As Figure 2 shows, the generation process comprises three stages:

Stage 1: Skeleton Generation. In the first stage (❶ in Figure. 2), a graph diffusion model is employed to first generate the IC design *skeleton*. This skeleton reflects the dependencies between all registers and IO ports, thereby capturing the overall logical state of the whole circuit.

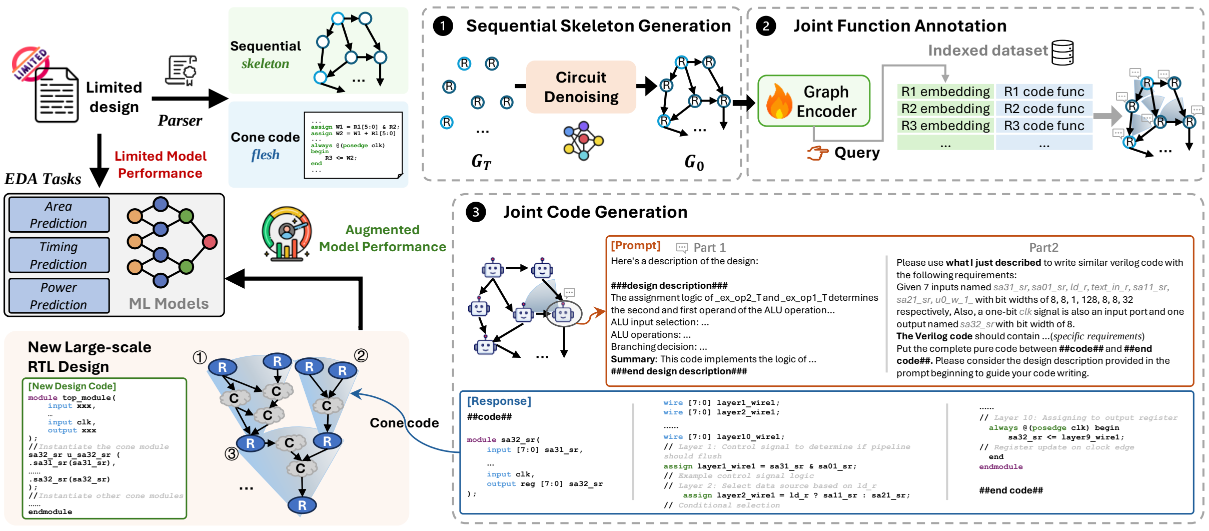
Stage 2: Function Annotation. In the second stage (❷ in Figure. 2), a multimodal encoder model will help annotate the expected functionality of the input cone of each register in the skeleton. These annotations will serve as contextual guidance for the subsequent generation of circuit *flesh*. The encoder is trained with self-supervised learning, aligning the register's skeleton-graph embeddings and the function description in text embeddings.

Stage 3: Cone Code Generation: In the third stage (❸ in Figure. 2), LLMs will generate the combinational logic in each cone in a layer-by-layer manner. Each register cone's annotated function description is used as context in our proposed LLM prompt. After the prompted LLM generates the cone code, a syntax checker is applied to ensure that every cone code is syntactically correct and embodies practical logical functionality.

Finally, the complete large-scale circuit is generated by integrating each register's cone codes into the skeleton's graph structure. This resulting design can both enhance downstream IC design tasks through data augmentation and serve as a benchmark, providing sufficient test data.

[2]For simplicity, we may only mention registers without explicitly discussing the very small portion of IO ports in S.

[3]Two registers are connected in the skeleton graph G if there is any combinational-logic path connecting them.



```
...
assign W1 = R1[5:0] & R2;
assign W2 = W1 + R1[5:0]
...
always @(posedge clk)
begin
    R3 <= W2;
end
...
```

[New Design Code]

```
module top_module(
    input xxx,
    …
    input clk,
    output xxx
);
//Instantiate the cone module
sa32_sr u_sa32_sr (
.sa31_sr(sa31_sr),
......
.sa32_sr(sa32_sr)
);
//Instantiate other cone modules
......
endmodule
```

[Prompt] Part 1

Here's a description of the design:

###design description###
The assignment logic of _ex_op2_T and _ex_op1_T determines the second and first operand of the ALU operation...
ALU input selection: ...
ALU operations: ...
Branching decision: ...
Summary: This code implements the logic of ...
###end design description###

Part2

Please use **what I just described** to write similar verilog code with the following requirements:
Given 7 inputs named *sa31_sr, sa01_sr, ld_r, text_in_r, sa11_sr, sa21_sr, u0_w_1_* with bit widths of 8, 8, 1, 128, 8, 8, 32 respectively, Also, a one-bit *clk* signal is also an input port and one output named *sa32_sr* with bit width of 8.
The Verilog code should contain ...(*specific requirements*)
Put the complete pure code between **##code##** and **##end code##.** Please consider the design description provided in the prompt beginning to guide your code writing.

[Response]

##code##

```
module sa32_sr(
    input [7:0] sa31_sr,
    …
    input clk,
    output reg [7:0] sa32_sr
);
```

```
wire [7:0] layer1_wire1;
wire [7:0] layer2_wire1;
......
wire [7:0] layer10_wire1;
// Layer 1: Control signal to determine if pipeline
should flush
assign layer1_wire1 = sa31_sr & sa01_sr;
// Example control signal logic
// Layer 2: Select data source based on ld_r
    assign layer2_wire1 = ld_r ? sa11_sr : sa21_sr;
// Conditional selection
```

```
......
// Layer 10: Assigning to output register
  always @(posedge clk) begin
      sa32_sr <= layer9_wire1;
// Register update on clock edge
  end
endmodule
```

##end code##

Figure 2: We propose a hierarchical generation scheme for large-scale circuit design with three stages. The integrated new designs can significantly enhance AI models in IC design. Stage 1 uses a graph diffusion model to create a skeleton capturing sequential dependency relations. Stage 2 annotates each register within the skeleton with functional descriptions via a retriever. Stage 3 leverages these annotations to guide LLM-based cone code generation.

5 SynC-LLM Methodology Details

5.1 Stage 1: Sequential Skeleton Generation

Each circuit in HDL code can be parsed and converted to a skeleton graph G, an attributed directed graph. In stage 1, SynC-LLM will learn the underlying distribution of G from real circuits, such that it can generate new synthetic skeletons that preserve similar structural characteristics.

Graph Diffusion Model: We adopt a diffusion-based generative model to learn the generation of the circuit skeleton. In the *forward process*, the original graph is gradually noised. Let $\mathbf{A}_0$ denote the adjacency matrix of the graph, and define a sequence $\{\mathbf{A}_t\}_{t=0}^{T}$ with the following transition:

$$q(\mathbf{A}_t|\mathbf{A}_{t-1}) = \mathcal{N}\left(\mathbf{A}_t; \sqrt{\alpha_t}\,\mathbf{A}_{t-1},\ (1-\alpha_t)\mathbf{I}\right),$$

where $\alpha_t \in [0,1]$ is a noise scheduling parameter and $\mathbf{I}$ is the identity matrix. In the *reverse process*, our diffusion model p_θ will recover $\mathbf{A}_0$ from $\mathbf{A}_T$ by iteratively applying:

$$p_\theta(\mathbf{A}_{t-1}|\mathbf{A}_t) = \mathcal{N}\left(\mathbf{A}_{t-1}; \mu_\theta(\mathbf{A}_t, t),\ \Sigma_\theta(\mathbf{A}_t, t)\right),$$

where $\mu_\theta(\cdot)$ and $\Sigma_\theta(\cdot)$ are learned functions predicting the mean and variance, respectively.

Edge Direction: We integrate the directional prediction directly into the reverse process. For any candidate edge between nodes u and v, we derive their latent node embeddings h_u and h_v. We then concatenate these embeddings and pass them through an MLP to generate the probability of an edge existence from u to v:

$$p(u,v) = \sigma\left(\mathrm{MLP}_E([h_u; h_v])\right)$$

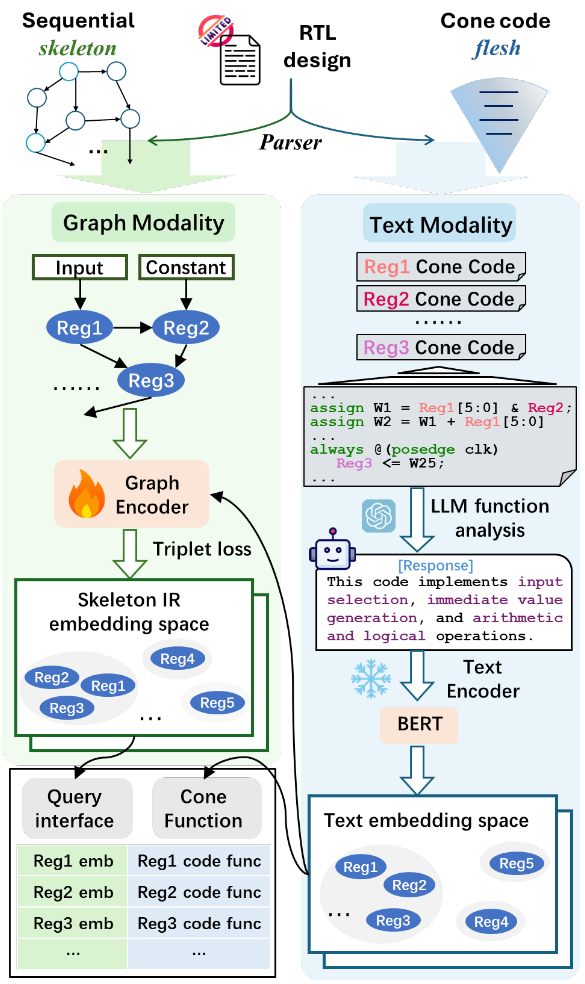
5.2 Stage 2: Function Annotation

After obtaining the circuit skeleton G in stage 1, stage 2 will annotate each register s_i in the skeleton with a function description. This annotation will describe the expected functionality of the cone *flesh* of register s_i. The annotation will later guide LLM to generate the HDL cone code in stage 3.

Motivation: In digital circuits, the function of each code *flesh* is related to the position of its corresponding register s_i in the graph skeleton G. For example, a high-bit-width register s_i near the graph's primary inputs is typically responsible for data reading, decoding, or handling of various instructions. In contrast, a high bit-width register s_i with a larger distance from inputs may perform extensive computational operations. These observations motivate our proposed function annotation on skeleton.

Recognizing that our collected real design datasets cover a wide range of functionalities, we introduce the retrieval-augmented approach (**retriever**) of reusing existing design data to annotate each register's expected function. The database is constructed and applied in the following ways:

Indexing: In the indexing stage, each circuit is decomposed into paired components: a register and its associated cone code. Each register is transformed into a graph embedding using our pretrained graph encoder. Along with their corresponding natural language function descriptions, these embeddings are stored in a dedicated database that serves as a reference for future retrieval tasks.



```
...
assign W1 = Reg1[5:0] & Reg2;
assign W2 = W1 + Reg1[5:0]
...
always @(posedge clk)
  Reg3 <= W25;
...
```

This code implements input selection, immediate value generation, and arithmetic and logical operations.

Figure 3: We propose a self-supervised multimodal graph encoder that links circuit skeleton features (graph modality) with functional semantics (text modality).

Retrieval & Inference: During the inference phase, register embeddings of the skeleton are obtained, which are then used to query the established database. A top-k search is performed based on the distance metric to retrieve the nearest entries which are expected functionally similar and can provide augmentative references to inform the new cone code generation.

Self-Supervised Multimodal Encoder: Since we query the function based on graph embeddings, we need to bridge the gap between the circuit's skeleton features and its cone function details. In other words, it is essential to learn a graph embedding space where registers with similar functions are mapped to nearby points.

We propose a self-supervised, multimodal contrastive learning technique to train the graph encoder as Figure 3 shows. The skeleton and corresponding cone codes are extracted firstly from circuit designs using a parser and we consolidate the skeletons of all designs into a single graph G_g, where different designs remain disconnected. The data processing flows in the graph and text modalities are explained as follows:

1. Graph Modality: Let $G_g = (V, E)$ denote the skeleton graph, where each node $v \in V$ is associated with an embedding h_v^g derived from a graph encoder which needs to be trained.

2. Text Modality: For each cone code corresponding with register v, we use commercial LLM to perform function analysis, thereby obtaining its natural language description (detailed prompt examples can be found in Appendix F). These descriptions are then transformed into a text embedding h_v^t using a pre-trained text encoder (e.g., BERT, text-embedding-ada-002).

To align the two modalities, we choose positive and negative samples for each register and utilize a triplet contrastive loss to train the graph encoder:

Positive and Negative Sample Selection: For each node v, we consider its text embedding h_v^t and compute the distances $\|h_v^t - h_j^t\|$ for all $j \in V \setminus \{v\}$. The k_p nodes with the smallest distances form the positive set for v: We define the positive set for each node v as:

$$P(v) = \{j \in V \setminus \{v\} \mid j \in \mathrm{NN}_{k_p}(v)\}.$$

where $\mathrm{NN}_{k_p}(v)$ denotes the set of the k_p nearest neighbors of v in the text embedding space. For the negative samples, we randomly select k_n nodes from the remaining nodes:

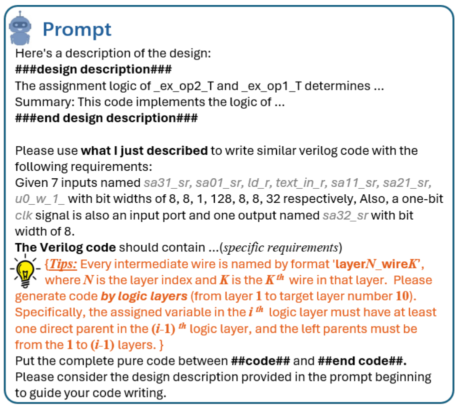
$$N(v) \subset (V \setminus (\{v\} \cup P(v)))\,.$$

Contrastive Alignment Loss: Our objective is to learn graph embeddings such that h_v^g is close to h_j^g for all $j \in P(v)$ and far from h_i^g for all $i \in N(v)$. This is enforced via a triplet loss. For each triplet (v, j, i) with $j \in P(v)$ and $i \in N(v)$, we define:

$$\mathcal{L}_{v,j,i} = \max\{0,\ d(h_v^g, h_j^g) - d(h_v^g, h_i^g) + \delta\},$$

where $d(\cdot,\cdot)$ denotes a chosen distance metric and δ is a margin hyperparameter. The overall loss is computed by averaging over all nodes and their corresponding positive-negative pairs:

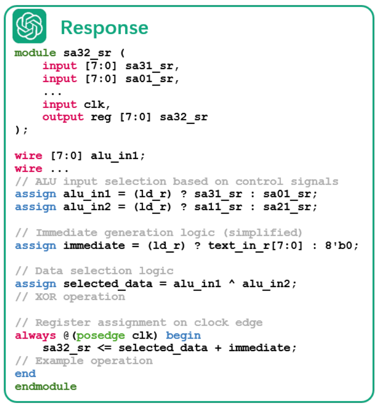
$$\mathcal{L} = \frac{1}{|V|}\sum_{v \in V}\frac{1}{|P(v)|\,|N(v)|}\sum_{j \in P(v)}\sum_{i \in N(v)}\mathcal{L}_{v,j,i}.$$



Here's a description of the design:
###design description###
The assignment logic of _ex_op2_T and _ex_op1_T determines ...
Summary: This code implements the logic of ...
###end design description###

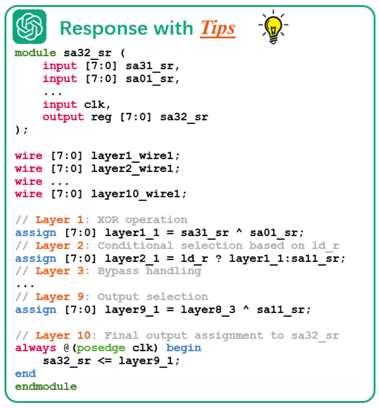
Please use **what I just described** to write similar verilog code with the following requirements:
Given 7 inputs named *sa31_sr, sa01_sr, ld_r, text_in_r, sa11_sr, sa21_sr, u0_w_1_* with bit widths of 8, 8, 1, 128, 8, 8, 32 respectively, Also, a one-bit *clk* signal is also an input port and one output named *sa32_sr* with bit width of 8.
The Verilog code should contain ...(*specific requirements*)
{*Tips:* Every intermediate wire is named by format '**layer*N*_wire*K***', where N is the layer index and K is the K^{th} wire in that layer. Please generate code ***by logic layers*** (from layer **1** to target layer number **10**). Specifically, the assigned variable in the i^{th} logic layer must have at least one direct parent in the $(i\text{-}1)^{th}$ logic layer, and the left parents must be from the **1** to **(*i*-1)** layers. }
Put the complete pure code between **##code##** and **##end code##.**
Please consider the design description provided in the prompt beginning to guide your code writing.

(a) Our prompt example for cone code generation



```
module sa32_sr (
    input [7:0] sa31_sr,
    input [7:0] sa01_sr,
    ...
    input clk,
    output reg [7:0] sa32_sr
);

wire [7:0] alu_in1;
wire ...
// ALU input selection based on control signals
assign alu_in1 = (ld_r) ? sa31_sr : sa01_sr;
assign alu_in2 = (ld_r) ? sa11_sr : sa21_sr;

// Immediate generation logic (simplified)
assign immediate = (ld_r) ? text_in_r[7:0] : 8'b0;

// Data selection logic
assign selected_data = alu_in1 ^ alu_in2;
// XOR operation

// Register assignment on clock edge
always @(posedge clk) begin
    sa32_sr <= selected_data + immediate;
// Example operation
end
endmodule
```

(b) Response without logic level control



```
module sa32_sr (
    input [7:0] sa31_sr,
    input [7:0] sa01_sr,
    ...
    input clk,
    output reg [7:0] sa32_sr
);

wire [7:0] layer1_wire1;
wire [7:0] layer2_wire1;
wire ...
wire [7:0] layer10_wire1;

// Layer 1: XOR operation
assign [7:0] layer1_1 = sa31_sr ^ sa01_sr;
// Layer 2: Conditional selection based on ld_r
assign [7:0] layer2_1 = ld_r ? layer1_1:sa11_sr;
// Layer 3: Bypass handling
...
// Layer 9: Output selection
assign [7:0] layer9_1 = layer8_3 ^ sa11_sr;

// Layer 10: Final output assignment to sa32_sr
always @(posedge clk) begin
    sa32_sr <= layer9_1;
end
endmodule
```

(c) Response with logic level control

Figure 4: Basic prompt with the retrieved functional specifications and I/O constraints instructs the LLM in generating the corresponding code. However, the LLM is not directly aware of the logic level and tends to produce code that is overly flat and lacks sufficient logical depth. To address this, we proposed a layer-by-layer generation prompt technique which results in new code that more accurately reflects the timing characteristics of real circuits.

5.3 Stage 3: Cone Code Generation

After each register in our circuit skeleton is annotated with a functional description in stage 2, these annotations will serve as a reference for LLM in stage 3 to generate the corresponding cone code, thus completing the circuit's *flesh*.

Basic Prompt: For every register s_i in the skeleton with known function annotation, parents, and associated bitwidth attributes, we can instruct the LLM to generate the cone code for s_i by providing I/O ports declaration and function reference.

Logic Level Control Technique: In real circuits, many cone codes exhibit a deep logic structure, characterized by potentially long paths from input to output, resulting in a slender structure. The length of these paths can partially reflect signal propagation delays, which is crucial in IC design.

In our approach, we propose a new customized prompt technique to query the LLM. Starting with the input and proceeding through a specified logic depth, the LLM is instructed to construct the circuit layer by layer. This generation method ensures that the resulting code exhibits a deeper hierarchical organization, thereby achieving the required logic level for realistic timing behavior. Specifically, this new prompt technique mainly consists of the following two aspects:

- *Variable Naming Convention:* Each variable is named based on its layer index following the convention `layerN_K`, where `N` denotes the variable's layer and `K` indicates its order within that layer. This systematic approach guarantees clarity and consistency in signal identification throughout the circuit.

- *Layer Consistency Enforcement:* To ensure that the layer number in the variable names reflects their actual position within the circuit, the prompt explicitly instructs the LLM that for any variable designated in the ith layer, at least one assignment variable must belong to the $(i-1)$th layer. Moreover, all additional assignment variables must originate from layers ranging between 0 and $i-1$.

Example Demonstration: As illustrated in Figure 4, in the absence of logic level control guidance within the prompt, the generated code only exhibits three logic layers from input to output. However, with the enhanced instructions, the response correctly produces a ten-layer logic circuit, also ensuring that the naming of each variable precisely matches its actual logic depth.

6 Experiments

In our experiment, we will first evaluate the similarity between synthetic circuits with real designs in multiple circuit statistics[4] Then we will evaluate if our newly generated synthetic circuits, as training data augmentation, can help boost ML model performance in IC design tasks.

	Degree MMD ↓	Cluster MMD↓	Orbit MMD ↓	Spectral MMD ↓
GraphRNN (You et al., 2018)	0.015	0.343	0.136	2.46
DVAE (Zhang et al., 2019)	0.043	0.458	0.371	0.036
SynC-LLM	**0.005**	**0.022**	**0.007**	**0.003**

Table 2: The directed register dependency graph similarity between the generated designs with the real datasets.

6.1 Experiment Setup

Circuit Design Compilation: Initially, we assembled a dataset consisting of 22 circuit designs extracted from open-source RTL repositories(Corno et al., 2000; Albrecht, 2005; Amid et al., 2020). This collection spans a diverse range of digital circuit modules, reflecting some of the highest-quality designs available in the open-source community. Dataset details are in Appendix A. It should be noted that although the dataset comprises only 22 designs, the training process of SynC-LLM utilizes several thousand cones extracted from these data. Moreover, the data employed for the timing task in the downstream application (using the register path) consists of tens of thousands of data points.

To obtain netlist-level labels serving downstream AI-based PPA prediction tasks, including design area, register slack, and total negative slack (TNS) prediction, we employed Synopsys Design Compiler™ 2021 with the NanGate 45nm technology library to get design labels. We randomly designated 7 of the designs as the test set, leaving the remaining 15 in the skeleton generative model training, retriever database building, and AI-based IC tasks model training.

Circuit Generative Methods. In contrast to our hierarchical multi-modal generation scheme, we implement generation baselines on circuit graph representation, motivated by an artificial netlist generator (Kim et al., 2021). We selected two representative graph generative models, including GraphRNN (You et al., 2018) and DVAE (Zhang et al., 2019), to generate new synthetic circuits as baselines. The detailed methodology and baselines settings are shown in Appendix D. For each generation method, we obtain 25 synthetic new designs for analysis and data augmentation. In addition, we provide a method complexity comparison in the Appendix C.

[4]Generating large-scale synthetic hardware designs is a novel research direction, and consequently, a standardized or mature evaluation framework for rigorously quantifying design similarity does not yet exist. We have made a concerted effort in our work to propose a set of reasonable and representative metrics from a hardware engineering perspective.

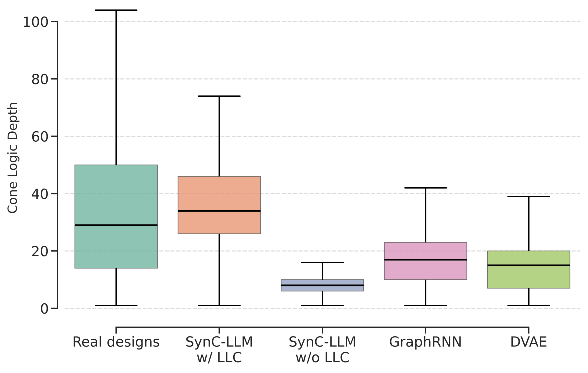


Figure 5: Statistics of the cone code's logic level depth. SynC-LLM with logic level control (LLC) generates cone code with much higher similarity to real circuits.

6.2 Synthetic Design Statistics

Register Dependency Graph (i.e., skeleton): The skeleton is parsed from the HDL code which represents the sequential logic characteristics of the design code. We compare the similarity in the register dependency graph between the generated circuits and the real designs using four metrics (Chen et al., 2023; Vignac et al., 2022): *1. Degree:* node connectivity distribution. *2. Cluster:* local connectivity via clustering coefficients. *3. Orbit:* recurring local subgraph patterns through orbit counts. *4. Spectral:* global structure extracted from the graph Laplacian eigenvalue. The statistics distribution similarity is calculated by the Maximum Mean Discrepancy (MMD).

$$\begin{aligned}\mathrm{MMD}^2(P,Q) &= \mathbb{E}_{x,x'\sim P}[k(x,x')] + \mathbb{E}_{y,y'\sim Q}[k(y,y')] \\ &\quad - 2\mathbb{E}_{x\sim P,y\sim Q}[k(x,y)]\end{aligned}$$

where P and Q denote two graph statistics distribution, and $k(\cdot,\cdot)$ denotes a kernel function.

Table 2 lists MMD scores of GraphRNN (You et al., 2018), DVAE (Zhang et al., 2019), and SynC-LLM. Lower scores are preferable as they indicate a closer match to the real design's skeleton. Results show that the hierarchical generation in SynC-LLM is more effective in capturing the circuit's global sequential skeleton structure.

Cone Logic Depth: We examined the logic depth of all the cones in the generated HDL code. Results are shown in Figure 5. SynC-LLM is able to generate code that more closely captures the cone logic depth distribution of real designs compared to GraphRNN (You et al., 2018) and DVAE (Zhang et al., 2019). Furthermore, we removed the logic level control (LLC) component

Model	Register Slack			TNS			Area		
	R → 1	MAPE ↓	RRSE ↓	R → 1	MAPE ↓	RRSE ↓	R → 1	MAPE ↓	RRSE ↓
No synthetic data augmentation	0.70	27%	0.83	0.81	50%	0.97	0.89	30%	0.62
GraphRNN (You et al., 2018)	0.70	27%	0.83	0.80	54%	0.97	0.84	44%	0.75
DVAE (Zhang et al., 2019)	0.69	29%	0.94	0.78	50%	0.97	0.84	61%	0.86
SynC-LLM w/o retriever	0.76	24%	0.76	0.85	47%	0.83	0.85	33%	0.70
SynC-LLM w/ retriever	**0.79**	**17%**	**0.73**	**0.95**	**42%**	**0.58**	**0.93**	**25%**	**0.43**

Table 3: AI model's accuracy in predicting the post-synthesis register slack, TNS, and area before synthesis. The basic training dataset consists of 15 real designs. The augmented dataset combines the basic dataset and 25 synthetic designs generated from SynC-LLM w/ retriever, SynC-LLM w/o retriever, GraphRNN (You et al., 2018), and DVAE (Zhang et al., 2019).

from the prompt used by SynC-LLM. Results show that SynC-LLM w/o LLC exhibits a noticeably shallower code structure compared to SynC-LLM and has a greater discrepancy from real designs. This observation indicates the effectiveness of our prompt technique.

Netlist Timing: We evaluate the circuit netlist statistics by calculating the ratio of Total Negative Slack (TNS) to the Number of Violated Paths (NVP). Results are shown in Figure 6. These metrics not only quantify the overall severity of timing violations but also reflect the distribution of delays across the circuit.

Figure 6 shows that graphs generated by GraphRNN (You et al., 2018) and DVAE (Zhang et al., 2019) exhibit very small TNS/NVP values, failing to capture the inherent delay characteristics of circuits. In contrast, our SynC-LLM w/ retriever demonstrates a more similar distribution to real designs. This suggests that SynC-LLM is more effective in modeling the timing behaviors present in real designs. In addition, after removing the retriever[5], the SynC-LLM w/o retriever shows a significant TNS/NVP value reduction. It proves the contribution of the function retrieval component for more realistic design generation.

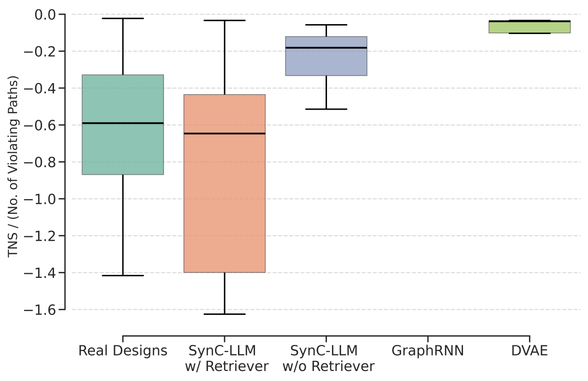


Figure 6: Netlist statistics. The synthetic designs generated by SynC-LLM w/ retriever exhibit *more similar TNS/NVP distribution* to those of the real designs compared to baselines. It should be noted that the TNS/NVP value of GraphRNN is very small, so it cannot be clearly displayed in this Figure.

6.3 Synthetic Designs as Training Data

In this section, we evaluate the effectiveness of generated synthetic circuits as training data augmentation in AI-assisted IC solutions. These AI application details can be found in Appendix G. This experiment is motivated primarily by the AI-assisted prediction of timing and area proposed in MasterRTL (Fang et al., 2023) and the fine-grained timing slack prediction by RTL-Timer (Fang et al., 2024). To rigorously assess model performance, we adopt three key metrics: the Pearson correlation coefficient (R), Mean Absolute Percentage Error (MAPE), and Root Relative Squared Error (RRSE).

Table 3 shows the model accuracy comparisons. Models augmented with synthetic data from SynC-LLM consistently surpass those trained solely on real designs, achieving superior performance across all evaluation metrics. Additionally, we removed the function annotation step (remove the retriever) from SynC-LLM in generating new designs. The SynC-LLM w/o retriever results indicate that the new designs even lead to a decrease in model performance compared with the basic training dataset. This may be due to the significant discrepancies between these designs from SynC-LLM w/o and real design characteristics, as evidenced by our analysis in Section 6.2.

7 Conclusion

SynC-LLM is the first work to leverage large language models (LLMs) for large-scale hardware syn-

[5]We directly prompt LLM to generate cone code without function annotation retrieval

thetic code generation, effectively addressing the scarcity of publicly available circuit design data. Our innovative three-stage process—comprising circuit skeleton synthesis, register function annotation, and level-by-level HDL code completion—not only produces valid large designs but also significantly enhances AI performance in various IC design tasks, marking a pivotal step forward in AI-assisted IC design techniques.

8 Acknowledgment

This work is supported by the Hong Kong Research Grants Council (RGC) CRF Grant C6003-24Y, GRF Grant 16200724, and ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK, Hong Kong SAR.

9 Limitations

We respectfully clarify that generating large-scale synthetic hardware designs is a novel research direction, and consequently, a standardized or mature evaluation framework for rigorously quantifying design similarity does not yet exist. Measuring the similarity between complex digital circuits—often containing over 20,000 nodes or 10,000 lines of code—is an inherently challenging task. Our analyses and augmentation training results indicate that our synthetic circuits bear significant resemblance to real designs. However, a comprehensive, systematic metric for evaluating circuit similarity and quality has yet to be established. Developing a rigorous framework to quantify would not only further validate our approach but also guide future efforts in synthetic circuit generation.

References

Christoph Albrecht. 2005. Iwls 2005 benchmarks. In *International Workshop for Logic Synthesis (IWLS): http://www. iwls. org*.

Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, et al. 2020. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4).

Chen Bai, Jiayi Huang, Xuechao Wei, Yuzhe Ma, Sicheng Li, Hongzhong Zheng, Bei Yu, and Yuan Xie. 2023. Archexplorer: Microarchitecture exploration via bottleneck analysis. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 268–282.

Zhuomin Chai, Yuxiang Zhao, Wei Liu, Yibo Lin, Runsheng Wang, and Ru Huang. 2023. Circuitnet: An open-source dataset for machine learning in vlsi cad applications with improved domain-specific evaluation metric and learning strategies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Chen-Chia Chang, Yikang Shen, Shaoze Fan, Jing Li, Shun Zhang, Ningyuan Cao, Yiran Chen, and Xin Zhang. 2024. Lamagic: Language-model-based topology generation for analog integrated circuits. *arXiv preprint arXiv:2407.18269*.

Lei Chen, Yiqi Chen, Zhufei Chu, Wenji Fang, Tsung-Yi Ho, Ru Huang, Yu Huang, Sadaf Khan, Min Li, Xingquan Li, et al. 2024. Large circuit models: opportunities and challenges. *Science China Information Sciences (SCIS)*.

Xiaohui Chen, Jiaxing He, Xu Han, and Li-Ping Liu. 2023. Efficient and degree-guided graph generation via discrete diffusion modeling. *arXiv preprint arXiv:2305.04111*.

Vidya A Chhabria, Kishor Kunal, Masoud Zabihi, and Sachin S Sapatnekar. 2021. Began: Power grid benchmark generation using a process-portable gan-based methodology. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–8. IEEE.

Animesh B Chowdhury, Shailja Thakur, Hammond Pearce, Ramesh Karri, and Siddharth Garg. 2023. Towards the imagenets of ml4eda. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE.

Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. 2000. Rt-level itc'99 benchmarks and first atpg results. *Design & Test of computers (ITC)*.

Wenji Fang, Wenkai Li, Shang Liu, Yao Lu, Hongce Zhang, and Zhiyao Xie. 2025a. Nettag: A multimodal rtl-and-layout-aligned netlist foundation model via text-attributed graph. *arXiv preprint arXiv:2504.09260*.

Wenji Fang, Shang Liu, Hongce Zhang, and Zhiyao Xie. 2024. Annotating slack directly on your verilog: Fine-grained rtl timing evaluation for early optimization. In *Proceedings of 2024 ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. ACM.

Wenji Fang, Yao Lu, Shang Liu, Qijun Zhang, Ceyu Xu, Lisa Wu Wills, Hongce Zhang, and Zhiyao Xie. 2023. Masterrtl: A pre-synthesis ppa estimation framework for any rtl design. In *Proceedings of 2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–9. IEEE.

Wenji Fang, Jing Wang, Yao Lu, Shang Liu, Yuchao Wu, Yuzhe Ma, and Zhiyao Xie. 2025b. A survey of circuit foundation model: Foundation ai models for vlsi circuit design and eda. *arXiv preprint arXiv:2504.03711*.

Wenji Fang, Jing Wang, Yao Lu, Shang Liu, and Zhiyao Xie. 2025c. Geneda: Unleashing generative reasoning on netlist via multimodal encoder-decoder aligned foundation model. *arXiv preprint arXiv:2504.09485*.

Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2024. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. *arXiv preprint arXiv:2408.08927*.

Xun Jiang, Yuxiang Zhao, Yibo Lin, Runsheng Wang, Ru Huang, et al. 2023. Circuitnet 2.0: An advanced dataset for promoting machine learning innovations in realistic chip design environment. In *International Conference on Learning Representations (ICLR)*.

Daeyeon Kim, Hyunjeong Kwon, Sung-Yun Lee, Seungwon Kim, Mingyu Woo, and Seokhyeong Kang. 2021. Machine learning framework for early routability prediction with artificial netlist generator. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1809–1814. IEEE.

Mengming Li, Wenji Fang, Qijun Zhang, and Zhiyao Xie. 2025. Specllm: Exploring generation and review of vlsi design specification with large language model. In *2025 International Symposium of Electronics Design Automation (ISEDA)*, pages 749–755. IEEE.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023a. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–8. IEEE.

Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024a. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, and Zhiyao Xie. 2025a. Towards big data in ai for eda research: Generation of new pseudo-circuits at rtl stage. In *Asia and South Pacific Design Automation Conference (ASPDAC)*.

Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2023b. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. *arXiv preprint arXiv:2312.08617*.

Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. 2024b. Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation. In *2024 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE/ACM.

Shang Liu, Jing Wang, Wenji Fang, and Zhiyao Xie. 2025b. Syncircuit: Automated generation of new synthetic rtl circuits can enable big data in circuits. *arXiv preprint arXiv:2509.00071*.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. Rtllm: An open-source benchmark for design rtl generation with large language model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 722–727. IEEE.

Jingyu Pan, Chen-Chia Chang, Zhiyao Xie, and Yiran Chen. 2023. Edalearn: A comprehensive rtl-to-signoff eda benchmark for democratized and reproducible ml for eda research. *arXiv preprint arXiv:2312.01674*.

Zehua Pei, Huiling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. 2024. Betterv: Controlled verilog generation with discriminative guidance. In *Forty-first International Conference on Machine Learning (ICML)*.

Martin Rapp, Hussam Amrouch, Yibo Lin, Bei Yu, David Z Pan, Marilyn Wolf, and Jörg Henkel. 2021. MLCAD: A survey of research in machine learning for CAD keynote paper. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE.

Clement Vignac, Igor Krawczuk, Antoine Siraudin, Bohan Wang, Volkan Cevher, and Pascal Frossard. 2022. Digress: Discrete denoising diffusion for graph generation. *arXiv preprint arXiv:2209.14734*.

Zhiyao Xie, Yu-Hung Huang, et al. 2018. RouteNet: Routability prediction for mixed-size designs using convolutional neural network. In *ICCAD*.

Ceyu Xu, Chris Kjellqvist, and Lisa Wu Wills. 2022. SNS's not a synthesizer: a deep-learning-based synthesis predictor. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, pages 847–859.

Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. 2018. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *International conference on machine learning*, pages 5708–5717. PMLR.

Muhan Zhang, Shali Jiang, Zhicheng Cui, Roman Garnett, and Yixin Chen. 2019. D-vae: A variational autoencoder for directed acyclic graphs. *Advances in neural information processing systems*, 32.

Jialv Zou, Xinggang Wang, Jiahao Guo, Wenyu Liu, Qian Zhang, and Chang Huang. 2024. Circuit as set of points. *Advances in Neural Information Processing Systems*, 36.

Source Benchmark	# of Designs	Design Scale (#K Gates) {Min, Median, Max}	Design Scale (#K Lines) {Min, Median, Max}
ITC'99 (Corno et al., 2000)	6	{9, 19, 45}	{5, 10, 10}
OpenCores (Albrecht, 2005)	8	{2, 6, 35}	{2, 8, 76}
Chipyard (Amid et al., 2020)	8	{12, 19, 52}	{3, 12, 24}

A Real Dataset Details

This dataset has adopted most of the widely adopted designs recognized by the community, considering the severe scarcity of high-quality open-source IC designs. High-quality digital HDL projects are exceptionally rare. Our 22 designs constitute the *most comprehensive open-source collection*, curated from three authoritative sources: academic benchmarks (ITC'99), industrial-grade repositories (OpenCores), and ecosystem platforms (Chipyard's RISC-V). In addition, this dataset represents significant diversity within the digital design space. It spans CPUs, DSPs, memory controllers, and synthesis benchmarks, with gate counts ranging from 2K–52K (ITC'99: 9–45K; OpenCores: 2–35K; Chipyard: 12–52K) and code complexity from 2K–76K lines.

Although it is possible to search for new IC designs from unverified public repositories (e.g., untested GitHub forks), these less validated designs lack the community's previous recognition and may not accurately reflect AI model's performance.

B SynC-LLM Utility

The key to SynC-LLM's **versatility** lies in its core capability: generating synthetic designs directly at the early hardware code stage. These synthetic codes serve as the initial input for the standard digital IC design flows (EDA flow). Consequently, the synthetic HDL generated by SynC-LLM can be seamlessly processed through established EDA toolchains for synthesis, placement, routing, and analysis. This enables circuit representations at different design stages, such as gate-level netlists or even physical layouts. As a result, critical features and labels corresponding to each task at different stages (e.g., post-synthesis timing information and post-placement congestion maps) can be extracted using standard tools. These features and labels can support AI models for any relevant task, including the prediction of IR drop, congestion, DRC violation, crosstalk, etc.

C Method Complexity and Cost

We demonstrate that SynC-LLM exhibits a significantly lower time and space complexity compared to the two state-of-the-art baselines (You et al., 2018; Zhang et al., 2019). We show the comparison of time and space complexity below:

	Generative Method	Space Complexity	Time Complexity
GraphRNN	Graph	$O(N)$	$O(N^2)$
DVAE	Graph	$O(N)$	$O(N^2)$
SynC-LLM (ours)	Graph+Text	$O(R^2)$	$O(R^2T)$

Table 4: Comparison of time and space complexity.

Specifically, both GraphRNN and DVAE are autoregressive models that generate graphs (i.e., circuits). Given that a circuit consists of N nodes and that these models must predict the edge connectivity between every pair of nodes, their time complexity is $O(N^2)$, and their space complexity is $O(N)$ for storing node embeddings.

In contrast, our SynC-LLM employs an edge-based generation strategy during the initial skeleton-generation phase, achieving a worst-case time complexity of $O(R^2T)$ and a space complexity of $O(R^2)$, where R represents the number of registers in the skeleton and T denotes the number of diffusion steps (we set T as 64 in experiment). Notably, at the word level, registers usually account for around 1% to 10% of the circuit (i.e., $R/N \approx$ 0.01 to 0.1). As a result, on our 4090 GPU platform, SynC-LLM is capable of generating a skeleton in mere seconds—using less than 10% of the time required by GraphRNN and DVAE. Moreover, during the text generation phase, the independent nature of each cone allows us to invoke the LLM API in parallel, with each cone being generated within tens of seconds. Overall, these factors underscore the superior scalability of SynC-LLM in generating large-scale circuits.

As for the Stage 2 cost in practice, we detail that, for the approximately 3,000 cone code segments, the entire function annotation process was completed in **less than 12 hours** with a computational cost of **less than $10 USD** using gpt-4o-mini. For the cone code generation phase in Stage 3, approximately 5000 cone codes for the 25 synthetic circuit designs were generated using 5 parallel gpt-4o-mini api around 8 hours, and cost around 20 $USD.

D Baselines and SynC-LLM Experiment Setting

For the baselines, as these node-ordering-based autoregressive approaches for GraphRNN and DVAE (You et al., 2018; Zhang et al., 2019) are not directly applicable to generating directed cyclic circuits, we preprocessed the training circuits by breaking cycles and generating the circuits in a topological node order.

For our SynC-LLM, we list the settings below:

- *In stage 1*, our approach is primarily based on EDGE (Chen et al., 2023), which is augmented with an additional edge direction and attribute predictor.
- *In stage 2*, for the text modality, we utilize GPT-4o-mini to perform the functional analysis of cone codes. The extracted function descriptions are then mapped into the text embedding space using the text-embedding-ada-002 Openai API. For each of the approximately 10K cone codes, we select the top 20 most similar data samples—determined by text embedding cosine similarity distances—as the positive samples, while 200 samples are randomly selected to serve as negatives. Concurrently, we leverage multiple encoder architectures—including GCN, GAT, and GraphSAGE—to encode the graph modality. These encoders are trained using a triplet loss formulation. We finally adopted the GraphSAGE to build the retriever. Please see some ablation studies in Appendix E.
- *In stage 3*, we employ GPT-40-mini to generate the detailed cone code for each register.

E Function Retriever Performance

In this section, we evaluate the retrieval performance based on the average distance between the text embeddings of a query and its retrieved entries. Specifically, for each query q, the top-K most similar samples $\{q_i^n\}$ in graph embeddings space are selected, and the mean distance between q and $\{q_i^n\}$ in text space is computed.

The *Nearest Labels* baseline represents the optimal retrieval where, for each data point, the top-K candidates are selected directly according to text embedding similarity, yielding the lowest and optimal average distance. In contrast, the *Random Retrieval* baseline is obtained by randomly sampling 1000 candidates, serving as a reference.

	Top-K average distance ↓			CCA→ 1
	K=1	K=10	K=20	
Random Retrival	0.739	0.739	0.739	-
Nearest Labels (optimal)	0.1673	0.175	0.179	-
GCN	0.181	0.194	0.201	0.975
GAT	0.193	0.202	0.208	0.972
GraphSage w/o logic level features	0.182	0.195	0.203	0.976
GraphSage w/ logic level features	**0.177**	**0.187**	**0.189**	**0.979**

Table 5: The retriever performance analysis. A lower Top-K average distance (text embedding distance) and $|CCA - 1|$ represent better retrieve quality. The results indicate that our approach effectively aligns the graph (*skeleton*) features with the text details (*flesh*).

Multiple graph-based models, including GCN, GAT, and GraphSAGE, are examined. The logic level features are used to initialize each node's representation[6], thereby helping the encoder to be more aware of the register positional information in the skeleton structure. For the GraphSAGE model, we further remove logic-level features for ablation study.

Top-K Results: Table 5 presents the performance at different values of K (i.e., $K = 1$, $K = 10$, and $K = 20$). The ground truth retrieval using Nearest Labels achieves average distances of 0.1673, 0.175, and 0.179 for $K = 1$, $K = 10$, and $K = 20$, respectively, which represent the optimal retrieval outcome. The Random Retrieve baseline yields a consistent average distance of 0.739 across different K values. Among the evaluated models, GraphSAGE with logic level features achieves the best performance with distances of 0.177, 0.187, and 0.189, whereas excluding these logic features leads to inferior performance (0.1823, 0.195, and 0.2033, respectively).

The experimental results indicate that our approach effectively aligns the graph (skeleton) features with the text details. The excellent performance of GraphSAGE, especially when enhanced with logic-level features, demonstrates that incorporating node positional information (via shortest path distances) improves the model's ability to retrieve semantically similar samples. Overall, the gap between the graph-based retrieval performance and the optimal is considerably narrowed, validating the efficacy of our proposed framework.

[6]The logic features are defined as the shortest path distances from the source node to the current node.

CCA Evaluation: We also evaluate the relationship between the graph embedding interface and the text embedding using Canonical Correlation Analysis (CCA). In CCA, given two sets of random variables, $X \in \mathbb{R}^p$ and $Y \in \mathbb{R}^q$, the aim is to determine the linear projections $\mathbf{a} \in \mathbb{R}^p$ and $\mathbf{b} \in \mathbb{R}^q$ that maximize the correlation between the projected variables:

$$\rho = \max_{\mathbf{a},\mathbf{b}} \frac{\mathbf{a}^\top \Sigma_{XY} \mathbf{b}}{\sqrt{\mathbf{a}^\top \Sigma_{XX} \mathbf{a}} \sqrt{\mathbf{b}^\top \Sigma_{YY} \mathbf{b}}}, \quad (1)$$

where Σ_{XX} and Σ_{YY} denote the covariance matrices of X and Y, respectively, and Σ_{XY} is the cross-covariance matrix between X and Y. The constraints $\mathbf{a}^\top \Sigma_{XX} \mathbf{a} = 1$ and $\mathbf{b}^\top \Sigma_{YY} \mathbf{b} = 1$ are imposed to normalize the projections.

The resulting CCA of 0.98 shown in Table 5 demonstrates that our learned embeddings are highly correlated across modalities.

F Detailed Prompt Examples

In this paper, we employ the LLM in two areas:
❶ **Cone Function Analysis**: We utilize LLMs to analyze functions associated with cone structures, which allows us to generate comprehensive index datasets capturing their key properties.

Note that we only use the following **prompt format**. It allows for a focused examination of specific functionalities without delving into trivial explanations, catering to an audience with advanced knowledge in HDL.

For the first example **Cone code 1**, it implements the basic logic selection and assignment functions, commonly found in the input stage of large circuits. These functionalities allow for the selection of specific input values and the execution of simple AND/OR operations, which are essential for processing various input conditions effectively.

For another example **Cone code 2**, it implements complex logic for managing control signals and data flow related to memory operations in a processor, focusing on program counter (PC) handling for branches, jumps, and instructions. It computes the next program counter value by evaluating various control signals, ensuring the correct PC is determined while checking for alignment and exception states. Additionally, it calculates the branch target based on the current PC and the type of instruction, accommodating compressed instructions and adjusting for immediate values. The code also includes error handling to check for misalignment, ensuring execution integrity. It determines the data to be written back to memory, contingent on valid instruction conditions, and integrates with control and status registers (CSRs) to capture exception-related values. This logic typically resides in the middle portion of a large circuit, positioned between the completion of input data processing and the generation of output data, serving as a critical decision-making hub that ensures the overall functionality and efficiency of the digital system.

Prompt

I am a seasoned Verilog code expert, and you are my assistant.
I will provide you with a snippet of Verilog code.
The code is {...}
Please analyze it and describe the specific functional logic it implements.
You may infer additional details based on your expert knowledge if necessary.
Since I am also an expert, please avoid explaining trivial or self-evident details.
Try to be concise. Output your explanation between the markers:
###func###
... (your functional description)...
###endfunc###

Cone Code 1

```
wire 05749;
assign 05749 = mem_reg_valid | mem_reg_replay;
wire mem_pc_valid;
assign mem_pc_valid = 05749 |
mem_reg_xcpt_interrupt;
wire [39:0] 00094;
assign 00094 = mem_pc_valid ? mem_reg_pc :
_coreMonitorBundle_pc_T;
reg [39:0] _coreMonitorBundle_pc_T;
always @(posedge clock)
_coreMonitorBundle_pc_T <= 00094;
```

Response

###func###
The provided Verilog code *snippet implements a logic for determining the validity of a program counter (PC) based on several signals.*
Signal Definitions:_05749_ is a wire that combines mem_reg_valid and mem_reg_replay using a logical OR operation. mem_pc_valid is another wire that is set to true if either _05749_ is true or if an interrupt exception (mem_reg_xcpt_interrupt) occurs.
PC Selection: _00094_ is a 40-bit wire that selects between mem_reg_pc and _coreMonitorBundle_pc_T based on the value of mem_pc_valid. If mem_pc_valid is true, _00094_ takes the value of mem_reg_pc; otherwise, it takes the value of _coreMonitorBundle_pc_T.
Register Update:_coreMonitorBundle_pc_T is a register that is updated on the rising edge of the clock. It takes the value of _00094_, thereby synchronizing the selected PC value with the clock.
In summary, this code evaluates and selects a valid program counter based on several input conditions, updating a register with this value on each clock cycle.
###endfunc###

❷ **Cone Generation**: By examining input-output

Cone Code 2

```
wire _00209_;
assign _00209_ = _take_pc_mem_T &  mem_ctrl_fp;
wire _00274_;
assign _00274_ = $signed(_mem_npc_a_T[63:39]) ==
$signed(25'h1ffffff);
wire _00273_;
assign _00273_ = $signed(_mem_npc_a_T[63:39])==
$signed(25'h0000000);
                        ......
assign _00158_ = _take_pc_mem_T & _05874_;
wire [63:0] mem_int_wdata;
assign mem_int_wdata = _00158_ ?  {
mem_br_target[39], mem_br_target[39],
mem_br_target[39], mem_br_target[39],
mem_br_target[39], mem_br_target[39],
mem_br_target[39], mem_br_target[39],
mem_br_target[39], mem_br_target[39],
mem_br_target[39], mem_br_target[39],
mem_br_target[39], mem_br_target[39],
mem_br_target[39], mem_br_target[39],
mem_br_target[39], mem_br_target[39],
mem_br_target[39], mem_br_target[39],
mem_br_target[39], mem_br_target[39],
mem_br_target[39], mem_br_target[39],
mem_br_target } : _mem_npc_a_T;

wire _00210_;
assign _00210_ = _00209_ &  mem_ctrl_wxd;
wire _05749_;
assign _05749_ = mem_reg_valid|mem_reg_replay;
wire mem_pc_valid;
assign mem_pc_valid = _05749_  |
mem_reg_xcpt_interrupt;
wire [63:0] _05824_;
assign _05824_ = _00210_ ?  io_fpu_toint_data :
mem_int_wdata;
wire [63:0] _00098_;
assign _00098_ = mem_pc_valid ?  _05824_ :
_csr_io_tval_a_T;
reg [63:0] _csr_io_tval_a_T;
always @(posedge clock)
_csr_io_tval_a_T <= _00098_;
```

Response

###func###
The provided Verilog code *implements logic for managing control signals and data flow associated with memory operations in a processor, specifically related to program counter (PC) handling for branches, jumps, and instructions.*
PC Management: The code computes mem_npc, which is the next program counter value based on various control signals (mem_ctrl_jal, mem_ctrl_jalr, etc.). ......
Branch Target Calculation: There are mechanisms to derive mem_br_target, which is calculated based on the current PC and the type of branch ......
Error Handling: Conditions such as mem_npc_misaligned indicate if the next PC is misaligned, which is a critical check for ensuring program execution integrity.
Data Writeback: The code determines mem_int_wdata, which is the data written back to memory based on the calculated branch target and the validity of the memory operation. ......
CSR Handling: There is a register _csr_io_tval_a_T that captures values related to exceptions or invalid operations, indicating integration ......
Overall, this snippet encapsulates complex decision-making logic for next instruction address computation, ensuring proper handling of control flow in a pipelined architecture.
###endfunc###

(I/O) relationships and existing functions, LLMs facilitate the creation of new cone codes, enhancing the complexity and scale of the hardware code.

In Figure 4, we illustrate one example of cone code generation, showcasing how the LLM leverages specified functional requirements and constraints to generate a more structured output. This example highlights the iterative process of refining the generated code by analyzing the dependencies and relationships within the design, ultimately leading to a more sophisticated representation of the intended hardware functionality. The layer-by-layer generation approach not only improves the logical depth of the code but also ensures that timing characteristics are accurately captured, thereby aligning the generated output more closely with the needs of complex circuit designs.

G Downstream AI-based IC tasks

We evaluate our synthetic RTL circuits on three different RTL-stage design quality prediction tasks: design-level TNS and area prediction, and register-level slack prediction. We use three representative ML models (Fang et al., 2023, 2024; Xu et al., 2022), covering both word-level and bit-level RTL representations. All models are implemented strictly following their respective reference papers (Fang et al., 2023, 2024; Xu et al., 2022):

- Model 1: SNS (Xu et al., 2022), is a transformer-based model designed for word-level RTL analysis. It takes tokenized RTL circuit paths as input sequences and uses a two-layer transformer with two attention heads (around 1.4M parameters in total). The model produces path-level predictions, which are then aggregated using an MLP to estimate design-level TNS and area. The model is trained using a regression objective with MAPE loss, following the original implementation in SNS.

- Model 2: MasterRTL (Fang et al., 2023), is a tree-based regression model that operates on bit-level RTL designs represented as Simple Operator Graphs (SOGs), which capture low-level Boolean operations. (1) For TNS prediction, it uses a two-stage pipeline. The first stage predicts path-level delays using a Random Forest model with 80 trees and a maximum depth of 20, utilizing features such as gate type, fanout count, and accumulated delay. The second stage refines these predictions using a design-level XGBoost regressor (45 trees, max depth 8), incorporating both SOG graph features and the outputs from stage one. (2) For area prediction, an XGBoost model

(45 trees, max depth 12) is trained using features including SOG operator count, operator types, and cell area information extracted from liberty files. Both tasks are trained with a standard squared error regression loss.

- Model 3: RTL-Timer (Fang et al., 2024), focuses on fine-grained timing slack estimation at the register level. For each register, it constructs its input cone and extracts hierarchical features from the design, cone, and path levels. To improve timing diversity, two representative paths, one the slowest and one randomly selected, are sampled per cone. The model uses an XGBoost regressor with 100 trees and a maximum depth of 45 to predict the slack delay for each register. Training is performed using regression with squared error loss.

H Synthetic Circuit Visualization

We present the visualization results of our generated circuits at different implementation stages. We first synthesize the generated circuits using Synopsys Design Compiler (DC) to produce gate-level netlists. Due to the large scale of the circuits, Figures 7 and 8 only show representative portions of the netlists. In these diagrams, arrows indicate the data flow directions, while rectangular blocks represent the standard cells in the netlist.

We further conducted Place & Route (P&R) for more rigorous comparison using the Innovus tool. Figures 9 and 10 display the layout diagrams of two synthesized circuits. To facilitate visual comparison and demonstrate that our synthetic circuits closely resemble real-world designs, we also provide the layout diagrams of realistic circuits (aes_core, Tinyrocket) as references in Figures 11 and 12.

From the figures, our generated circuits show minimal visual differences compared to the realistic reference circuits. Both EDA flows (DC and P&R) require standard cell libraries for implementation, for which we have adopted the open-source Nangate45 library.

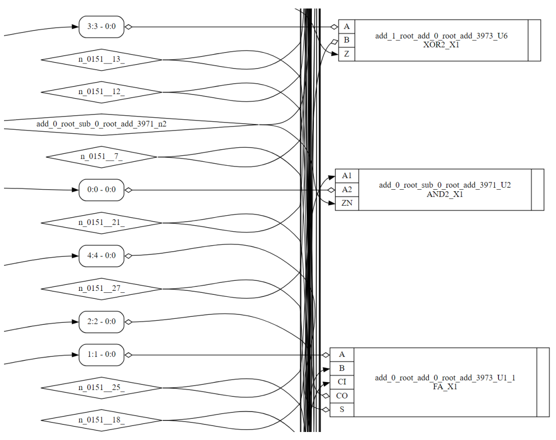


Figure 7: Partial gate-level netlist (Example 1).

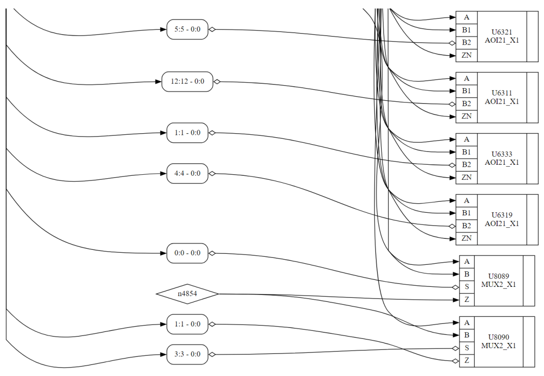


Figure 8: Partial gate-level netlist (Example 2).

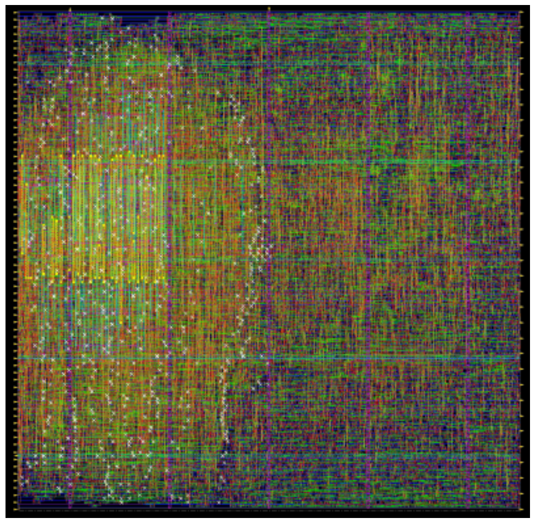


Figure 9: Layout view of **our synthesized** Example 1 after P&R.

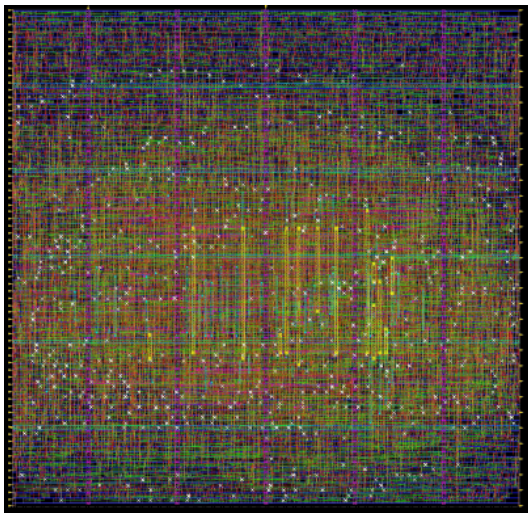


Figure 10: Layout view of **our synthesized** Example 2 after P&R.

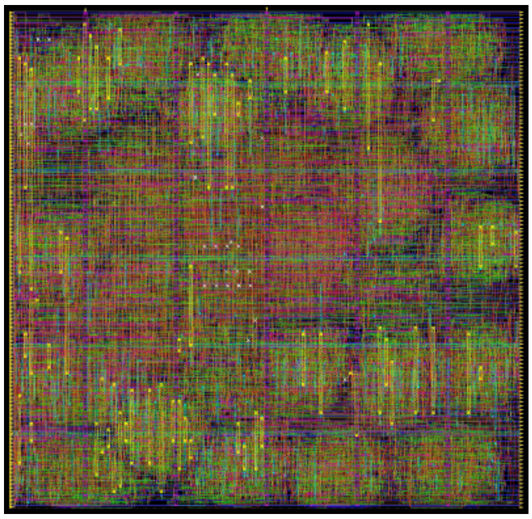


Figure 11: Layout of a **realistic** circuit (aes_core) for visual comparison with our synthetic designs.

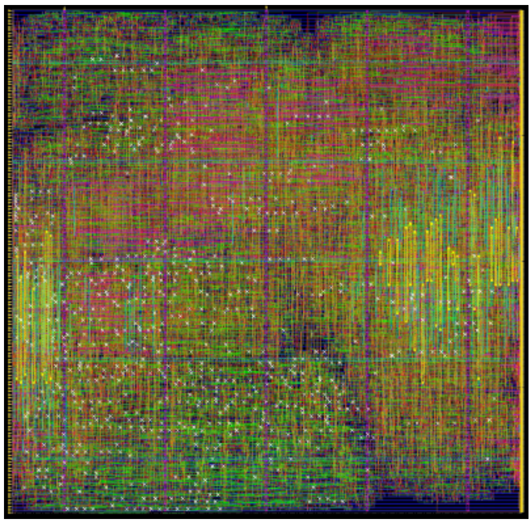


Figure 12: Layout of a **realistic** circuit (Tinyrocket) for visual comparison with our synthetic designs.