Automating Hardware Design and Verification from Architectural Papers via a Neural-Symbolic Graph Framework

Haoyue Yang[1,*], Xuanle Zhao[1,*], Yujie Liu[1], Zhuojun Zou[1], Kailin Lyu[1], Changchun Zhou [2,†], Yao Zhu[3,†], Jie Hao[1,†]
[1] Institute of Automation, Chinese Academy of Sciences, Beijing, China
[2] Peking University, Beijing, China [3] Zhejiang University, Zhejiang, China
yanghaoyue2024@ia.ac.cn, ee_zhuy@zju.edu.cn

Abstract

The reproduction of hardware architectures from academic papers remains a significant challenge due to the lack of publicly available source code and the complexity of hardware description languages (HDLs). To this end, we propose **ArchCraft**, a Framework that converts abstract architectural descriptions from academic papers into synthesizable Verilog projects with register-transfer level (RTL) verification. ArchCraft introduces a structured workflow, which uses formal graphs to capture the Architectural Blueprint and symbols to define the Functional Specification, translating unstructured academic papers into verifiable, hardware-aware designs. The framework then generates RTL and testbench (TB) code decoupled via these symbols to facilitate verification and debugging, ultimately reporting the circuit's Power, Area, and Performance (PPA). Moreover, we propose the first benchmark, **ArchSynthBench**, for synthesizing hardware from architectural descriptions, with a complete set of evaluation indicators, 50 project-level circuits, and around 600 circuit blocks. We systematically assess ArchCraft on ArchSynthBench, where the experiment results demonstrate the superiority of our proposed method, surpassing direct generation methods and the VerilogCoder framework in both paper understanding and code completion. Furthermore, evaluation and physical implementation of the generated executable RTL code show that these implementations meet all timing constraints without violations, and their performance metrics are consistent with those reported in the original papers.

1 Introduction

"*This paper on a novel neural network accelerator is excellent*," your hardware director mused, "*but its implementation is closed-source. We must reproduce it to build our next-generation products upon it.*" This demand to reproduce synthesizable

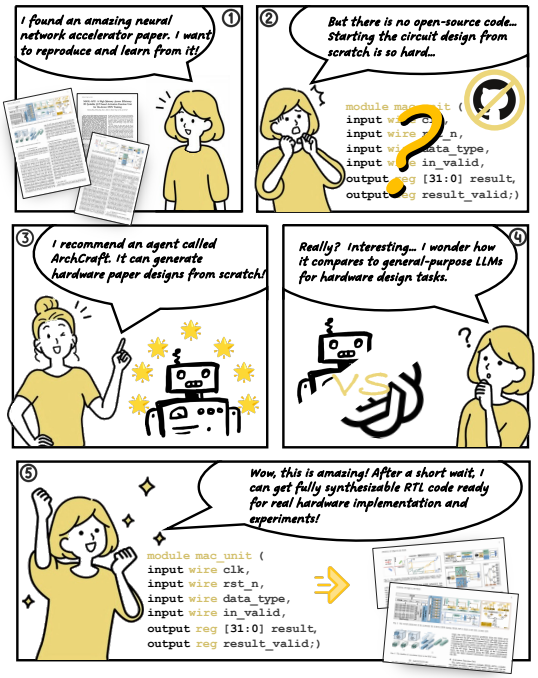

Figure 1: A real-world example. Archcraft assists in automating the process of transforming hardware design concepts into fully synthesizable RTL code, reducing manual intervention and accelerating the design process.

hardware designs from complex academic papers increasingly points to a fundamental challenge as shown in Figure 1: how can we design an automated system to seamlessly convert unstructured academic text, a 30-page PDF, into rigorous, synthesizable, and functionally correct code (Qian et al., 2023; Seo et al., 2025; Zhao et al., 2025c,b; Zhang et al., 2023a)?

Let us deconstruct the workflow of a human expert performing this task. First, the engineer thoroughly reads the paper to construct a high-level architectural blueprint to identify key modules, their hierarchy, and the data flow connections between them. Next, for each module, the engineer defines its precise functional specification, including interfaces like inputs and outputs, internal states, and, critically, concurrent behaviors

and timing logic. Finally, the engineer translates this structured, formalized understanding into RTL code and a corresponding TB. This process is, in essence, a complex, multi-step, domain-knowledge-dependent workflow.

Currently, approaches for any-to-any generative tasks typically fall into two paradigms: Implicit Neural Modeling and Agent Approaches. Implicit Neural Modeling approaches directly learn a neural representation from mass training data (Liu et al., 2024c, 2025c; Chang et al., 2024; Thakur et al., 2024; Wu et al., 2025). While Implicit Neural Modeling approaches show advantages in specific functional tasks and design space exploration, their extensibility is constrained by the scope of the training data. Consequently, it fails to understand the project-level architectural blueprint. The second, Agent Approaches (Zhao et al., 2025d; Ho et al., 2025; Wang et al., 2025), however, existing implementations often manifest as a Flat or Naive Agent paradigm. They fail to inherit the expert's actual mental model; instead, many rely on waveforms or TB as primary inputs. This is not true generation from scratch and fundamentally sidesteps the core challenge of translating high-level textual specifications into functional hardware logic.

This motivates us to explore an evolution of this paradigm: how can we design a "Structured Agentic Framework"? It should inherit the autonomy of the workflow while simultaneously embedding the human expert's mental model, enabling it to become more intelligent and reliable without training. Our core insight is: *a successful hardware generation framework must replace the LLM's general-purpose reasoning with a domain-specific, formalized intermediate representation*. Based on this, we propose ArchCraft, a novel, training-free, structured agentic framework. The core of ArchCraft is the Graphing Agent, which systematically constructs the first part of the human mental model, the architectural blueprint, through a rigorous process. Its execution process involves defining the graph's scope and theme, identifying its nodes and static links, planning its directionality and internal node attributes, and finally, adding global constraints. Moreover, we introduce the Symboling stage, which systematically translates the node logic and attributes defined during the Graphing stage into a rigorous symbolic representation. This symbolic blueprint serves as a precise intermediate language, greatly facilitating the subsequent coding stage and constraining the LLM to adhere to correct hardware logic without any domain-specific model fine-tuning. Subsequent Coding and Evaluating agents then generate the RTL, TB, and perform functional verification based on this foundation.

Furthermore, we constructed ArchSynthBench, the first benchmark specifically designed for synthesizing hardware from academic papers describing closed-source accelerators. The first key feature of ArchSynthBench is that it comprises 50 project-level circuits and around 600 circuit blocks. The second key feature of this benchmark is its complete set of evaluation indicators, which are meticulously structured into two levels, eight dimensions as detailed in Section 4.2: evaluating architectural understanding and implementation completeness. Experiments on this benchmark show that our structured agentic framework achieves superior performance, surpassing LLMs with direct generation and agent frameworks like VerilogCoder. Benefiting from the Graph-based process, our paper-level assessment reaches a score of 86.17. Meanwhile, the symbolic constraints from Symboling ensure code-level completeness, achieving 81.04. These results demonstrate the framework's effectiveness on the novel task of zero-shot generation of synthesizable hardware. In summary, the main contributions are as follows.

- We propose **ArchCraft**, a novel neural-symbolic graph-based framework that enables the complex task of transforming unstructured academic papers into synthesizable, project-level hardware implementations.
- Our proposed symbolic representation of functions and interfaces bridges the gap between RTL documentation and code, enabling the generation of non-open-source HDL code from models.
- We construct a benchmark, **ArchSynthBench**, to rigorously evaluate the replication accuracy and implementation quality of hardware synthesis. Our evaluation results show that previous methods struggle to reproduce functionally correct code, even for circuits.
- We further implement and evaluate the hardware design at the circuit level. The results show that the PPA of the synthesized implementation is consistent with that reported in the paper.

2 Related Work

2.1 LLMs for Research

LLM-agents are increasingly central to scientific discovery, from ideation to execution. Frameworks

decompose research into autonomous modules for tasks like mining, experimentation, and writing (Schmidgall and Moor, 2025; Schmidgall et al., 2025; Ghafarollahi and Buehler, 2025; Baek et al., 2024; Cui et al., 2025; Zhao et al., 2025a), enhancing efficiency and scalability. For reproducibility, AutoReproduce (Zhao et al., 2025c) uses a paper lineage algorithm to extract implicit knowledge, automating AI experiment reproduction with demonstrated superior performance. Collectively, these works show the potential of multi-agent LLM frameworks to transform the scientific process.

2.2 LLMs for Hardware Design

LLM application in hardware design is growing. Foundational datasets underpin this progress, such as PyraNet (Nadimi et al., 2024), CodeV (Zhao et al., 2024), and MG-Verilog (Zheng et al., 2024). Methodologies include direct generation and agent-based frameworks. Direct generation includes Chip-Chat (Blocklove et al., 2023) for interactive co-design, and structured prompting methods like HiVeGen (Tang et al., 2024), ROME (Nakkab et al., 2024), SynC-LLM (Liu et al., 2025b) and VeriMind (Nadimi et al., 2025) (using chain-of-thought) to improve correctness and precision. Agent-based frameworks include VerilogCoder (Ho et al., 2025), which integrates planning, checking, and simulation, and RTLSquad (Wang et al., 2025), a collaborative agent architecture for integrated design, implementation, and verification, advancing hardware automation.

3 ArchCraft

In this section, we introduce ArchCraft, a Neural-Symbolic graph-based framework designed to overcome the challenges of hardware reproduction stemming from the scarcity of open-source materials. Unlike previous approaches, such as specialized large model fine-tuning or circuit-level agents, which depend on implicit neural representations and entail costly training processes and input prerequisites, our ArchCraft embeds a Mental Model. The generated circuit code supports comprehensive evaluation from functional to physical implementation(Sections 4.3). Prompts for the Agentic framework are in Appendix B.

3.1 The Very Beginning

We utilize papers from the domain of architecture, particularly those related to ASICs, as input. In contrast to the industrial documents required in conventional workflows, academic papers represent a form of novel, rigorous, and high-quality documentation. Furthermore, compared to industrial documents used internally within engineering departments, academic papers are abundant and openly accessible to all in the relevant field for reading and study. However, a challenge with academic papers is that they can be more abstract, lacking descriptions of circuit-level design details. A more detailed comparison of these inputs is provided in the Appendix 5. This Neural-Symbolic graph-based Agentic framework is therefore proposed to address this challenge.

3.2 Graphing Phase

The Graphing phase is orchestrated by the Graphing agent, $\mathcal{A}_{\mathcal{G}}$, taking an academic paper P as input and produces a graph $\mathcal{G}$, where G consists of a set of nodes V and a set of edges $\mathcal{E}$:

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}) \tag{1}$$

Where $\mathcal{V}$ stands for nodes, $\mathcal{E}$ stands for edges. This transformation is decomposed into a rigorous four-step sequence, where each step contributes specific components to the final graph structure.

Scope and Theme First, the agent establishes the high-level context. This step defines the boundaries of the scope S and the primary objectives or themes T of the implementation, extracted directly from the paper's claims and contributions. Formally, this process can be described as:

$$\mathcal{A}_{\mathcal{G}_1} : P \mapsto (S, T) \tag{2}$$

where $\mathcal{A}_{\mathcal{G}_1}$ denotes the agent responsible for scope establishment.

Nodes and Static Links Based on the established scope S, the agent identifies the core components and their physical or hierarchical relationships. This step defines the set of nodes $\mathcal{V}$, representing hardware modules and the set of static, undirected edges $\mathcal{E}_{\text{static}}$, representing structural connections or module containment. Formally, this process can be represented as a mapping:

$$\mathcal{A}_{\mathcal{G},2} : (P, S) \mapsto (\mathcal{V}, \mathcal{E}_{\text{static}}) \tag{3}$$

where $\mathcal{A}_{\mathcal{G}_2}$ denotes the agent responsible for nodes and static links construction.

Direction and Internal Nodes With the nodes $\mathcal{V}$ defined, this step details the functional behavior and data pathways. The agent plans the graph's direction by defining directed edges $\mathcal{E}_{\text{dir}}$, representing dataflow and assigns a set of internal properties Φ

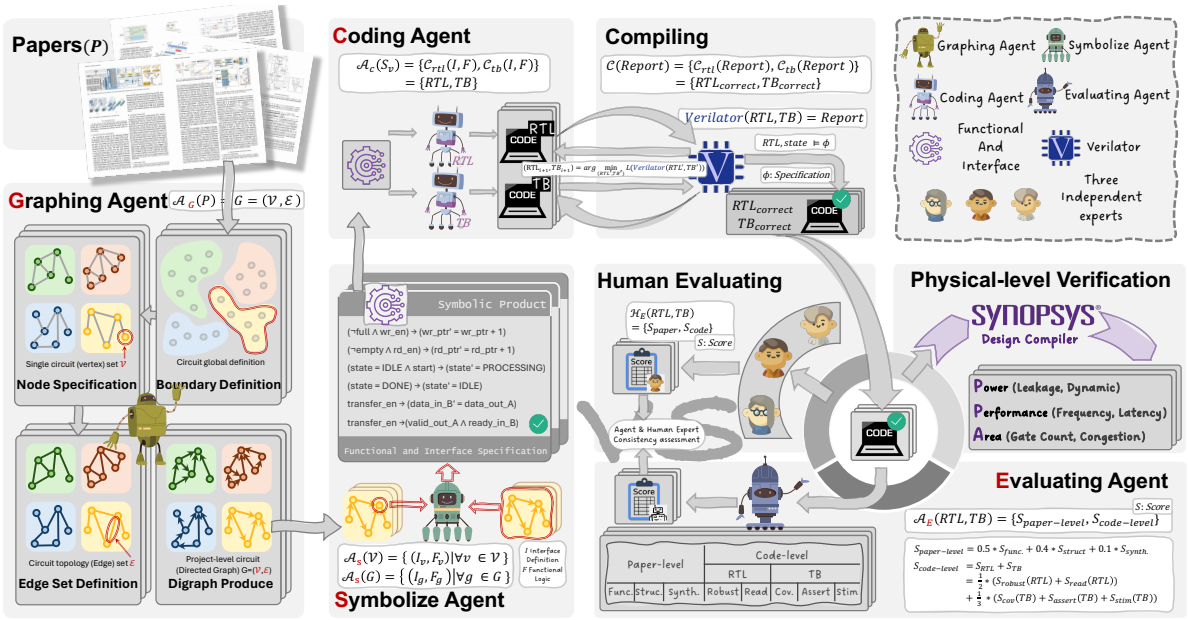

Figure 2: Overview of our ArchCraft framework with four phases. (1) Graphing Agent parses the paper into a formal architectural knowledge graph. (2) Symbolize Agent converts the graph nodes into a rigorous symbolic blueprint. (3) Coding Agent decoupledly generates synthesizable RTL and TBs from the symbolic blueprint. (4) Automatic execution of compilation and checking of grammar and logical issues, collects error reports, holds errors and related codes accountable, and collaboratively corrects them within a continuous feedback loop. Finally, both LLM and human experts evaluate the generated code to assess the overall effectiveness of the system. The generated RTL code is then synthesized into a hardware implementation for physical-level evaluation.

(e.g., functionality, parameters) to each node $v \in \mathcal{V}$. Mathematically, this process is represented by:

$$\mathcal{A}_{\mathcal{G}_3}(P, \mathcal{V}) \mapsto (\mathcal{E}_{\text{dir}}, \Phi) \tag{4}$$

in which $\mathcal{A}_{\mathcal{G}_3}$ is the agent for edge generation.

Global Constraints Finally, the agent extracts system-wide requirements that apply to the entire graph. This includes global constraints C, such as clocking schemes, bus standards, or top-level performance targets, which are critical for ensuring architectural coherence. This extraction task is formalized as the following mapping:

$$\mathcal{A}_{\mathcal{G}_4}(P) \mapsto C \tag{5}$$

Here, $\mathcal{A}_{\mathcal{G}_4}$ denotes the agent responsible for identifying and extracting the set of global constraints C from the entire paper P.

The final graph is $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. In this graph, the nodes $\mathcal{V}$ represent all registers and IO ports, while the edges $\mathcal{E}$ represent the dependency relationships between them. This composite graph, formed by the union of static edges $\mathcal{E}_{\text{static}}$ and directed edges $\mathcal{E}_{\text{dir}}$, is augmented by node attributes Φ and governed by global constraints C. This complete graph $\mathcal{G}$ serves as the formal architectural blueprint for all subsequent stages.

3.3 Symboling Phase

Following the construction of the architectural blueprint $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the Symbolization phase, executed by the Symbolize Agent $\mathcal{A}_{\mathcal{S}}$, translates the internal logic and attributes of each graph node into a formal symbolic representation. As illustrated in the provided Figure 2, the agent's operation is defined at two levels:

Graph-level Symbolization From a global perspective, the agent's task is to define the overall specification for design, which is itself hierarchical. If the intent is to produce a top-level model specification for the entire graph $\mathcal{G}$, we write this mapping as:

$$\mathcal{A}_{\mathcal{S}} : \mathcal{G} \mapsto (I_{\mathcal{G}}, F_{\mathcal{G}}), \qquad \text{where } \mathcal{G} = (\mathcal{V}, \mathcal{E}). \tag{6}$$

This top-level specification $(I_{\mathcal{G}}, F_{\mathcal{G}})$ is inherently defined by the composition of its internal parts.

Alternatively, the agent can be applied to a family of subgraphs (or subsystems) $\mathfrak{G} \subseteq \mathcal{P}(\mathcal{G})$, which represent the intermediate modules in the hierarchy. This is expressed as:

$$\mathcal{A}_{\mathcal{S}}(\mathfrak{G}) = \{(I_g, F_g) \mid g \in \mathfrak{G}\}. \tag{7}$$

This hierarchical decomposition continues until the process reaches the individual nodes, which require their own specific definitions.

Node-level Symbolization For every node $v \in \mathcal{V}$, the agent $\mathcal{A}_{\mathcal{S}}$ produces an interface definition I_v and a functional-logic specification F_v. This captures the complete symbolic product for each module. The per-node mapping can be expressed as:

$$\mathcal{A}_{\mathcal{S}} : \mathcal{V} \to \mathcal{I} \times \mathcal{F}, \qquad v \mapsto (I_v, F_v), \tag{8}$$

and the image of the whole node set is

$$\mathcal{A}_{\mathcal{S}}(\mathcal{V}) \;=\; \{(I_v, F_v) \mid v \in \mathcal{V}\}. \tag{9}$$

After the above two steps, we will get all levels of interfaces and functions of a project-level circuit design. This complete set of specifications, $\{(I, F)\}$, using the notation from the previous equations, this set is:

$$\{(I, F)\} = \mathcal{A}_{\mathcal{S}}(\mathcal{V}) \;\cup\; \mathcal{A}_{\mathcal{S}}(\mathfrak{G}) \;\cup\; \{(I_{\mathcal{G}}, F_{\mathcal{G}})\} \tag{10}$$

3.4 Coding and Compiling Phase

This phase transforms the interface I and functional specification F into synthesizable RTL code and its corresponding TB.

3.4.1 Coding

The Coding Agent, $\mathcal{A}_{\mathcal{C}}$ takes the symbolic interface I and functional logic F for each module as its input and generates two distinct code artifacts: the RTL implementation $\mathcal{C}_{rtl}$ and the TB $\mathcal{C}_{tb}$.

$$\begin{aligned} \mathcal{A}_{\mathcal{C}}(I, F) &= (\mathcal{C}_{rtl}(I, F), \mathcal{C}_{tb}(I, F)) \\ &= (RTL, TB) \end{aligned} \tag{11}$$

RTL and TB are generated decoupledly, ensuring that the functional hardware logic and its verification environment are both directly grounded in the same formal specification.

3.4.2 Compiling and Internal Feedback Loop

The generated code artifacts $\{RTL, TB\}$ are not assumed to be correct on the first attempt. They immediately enter a rigorous verification and refinement loop, as illustrated in the Figure 2.

Verification: The code is passed to an external simulator (e.g., Verilator) for compilation and execution, which produces a simulation $Report$.

$$Verilator(RTL, TB) = Report \tag{12}$$

Internal Reflection: The $Report$ is then fed back into a corrector function $\mathcal{C}\_\mathcal{F}(Report)$. This function analyzes the report to identify discrepancies and determines if the current code is correct $\{RTL_{correct}, TB_{correct}\}$.

$$\begin{aligned} \mathcal{C}\_\mathcal{F}(Report) &= (\mathcal{C}_{rtl}(Report), \mathcal{C}_{tb}(Report)) \\ &\stackrel{?}{=} (RTL_{correct}, TB_{correct}) \end{aligned} \tag{13}$$

Iterative Refinement: If errors are detected, e.g., $Report$ indicates failure, the framework activates an internal feedback loop. This iterative process, defined as $i+1$, seeks to find a corrected set of code (RTL', TB') that minimizes the error loss function L based on the output of the Verilator as:

$$\arg \min_{(RTL', TB')} L(Verilator(RTL', TB')) \tag{14}$$

This feedback cycle: Coding → Verilator → Report → Refinement repeats, continuously checking the RTL's state (Φ) against the original specification. The loop ends only when the code passes all the checks, resulting in verified artifacts $\{RTL_{correct}, TB_{correct}\}$.

4 ArchSynthBench

This section further introduces ArchSynthBench from three perspectives: the composition of the dataset (Section 4.1), the metrics and scoring methodologies for both machine and human expert evaluations on the benchmark (Section 4.2), and the physical implementation methods for the circuits (Section 4.3).

4.1 Data Source

ArchSynthBench consists of 50 distinct, closed-source hardware papers spanning 50 project-level circuits, and around **600** circuit blocks, which VerilogEval(Liu et al., 2023) has only 156 circuit blocks. This represents a significant expansion in project-level complexity over module-centric benchmarks like VerilogEval. Beyond its scale, ArchSynthBench introduces comprehensive criteria for evaluating agent workflows. The full paper list, along with detailed paper's theme, is available in Appendix Table 8 and Figure 10.

4.2 Evaluating Phase

Following recent LLM-as-judge approaches (Gu et al., 2024; Chen et al., 2024b), to quantitatively assess the quality of the generated artifacts, we introduce a comprehensive, two-tiered evaluation system orchestrated by an Evaluating Agent $\mathcal{A}_{\mathcal{E}}$, specific scoring criteria are set to measure the codes, detailed in Appendix Section D. This phase takes the verified RTL and TB as inputs and computes two distinct scores S, in Figure 2: $S_{paper-level}$ and $S_{code-level}$.

Paper-level Score The $S_{\text{paper-level}}$ score measures the fidelity of the generated code to the original academic paper. It assesses how well the implementation reproduces the concepts described in the source text. This score is defined as a weighted sum of three sub-metrics: Functionality $S_{\text{Func.}}$, Structure $S_{\text{Struct.}}$, and Synthesizability $S_{\text{Synth.}}$, as follow:

$$S_{\text{paper-level}} = 0.5 \cdot S_{\text{func.}} + 0.4 \cdot S_{\text{struct.}} + 0.1 \cdot S_{\text{synth.}} \quad (15)$$

Detailed definitions for these sub-metrics are available in Appendix Table 6.

Code-level Score The $S_{\text{code-level}}$ score assesses the intrinsic quality and robustness of the generated code itself, independent of the source paper. It is the sum of the RTL Score S_{RTL} and the TB Score S_{TB}. The S_{RTL} is the **average** of the design's Robustness $S_{\text{Robust.}}$ and Readability $S_{\text{Read.}}$. The S_{TB} is the **average** of the verification environment's Coverage $S_{\text{Cov.}}$, Assertions $S_{\text{Assert.}}$, and Stimulus $S_{\text{Stim.}}$. Detailed evaluation rubrics for these five sub-metrics are provided in Appendix 6, 7.

In addition, we have set up a penalty mechanism. If the generated code is not even in Verilog language, the code-level score will be further reduced by 10%. These machine-generated scores are then used in conjunction with human expert evaluations to perform a consistency assessment, ensuring the automated metrics align with human judgment on design quality and correctness.

4.3 Physical-level Verification

Beyond functional correctness, a crucial aspect of our evaluation is assessing the physical-level feasibility and efficiency of the generated hardware. This step moves from simulation to synthesis, providing tangible metrics for the design's real-world viability.

As illustrated in 2, the functionally correct RTL code is fed into an industry-standard logic synthesis tool, which synthesizes the RTL into a gate-level netlist, from which we extract the essential PPA metrics:

- **Power:** We analyze the estimated power consumption, including both static **Leakage** power and **Dynamic** power dissipated during operation.
- **Performance:** We evaluate the timing characteristics, primarily the operational **Latency**.
- **Area:** We measure the total design footprint, reported as the logical **Gate Count**, and assess the routing difficulty via the **Congestion** report.

This PPA analysis allows for a comparison against the results reported in the academic paper, providing the definitive measure of our framework's ability to reproduce not just the design's logic, but also its physical quality and efficiency.

5 Experiments

In this section, we describe the experimental setup and results in detail.

5.1 Experiment Settings

Baseline. We propose the novel task of generating hardware RTL code from academic hardware papers. Given the absence of agentic workflow baselines specialized for this task, we compare our framework directly against LLMs. These LLMs include OpenAI o3-mini (OpenAI, 2025), Gemini 2.0 Flash (Google DeepMind, 2025), and Qwen3-Coder-480B-A35B-Instruct-FP8 (Team, 2025), all of which generate RTL code in a single step from the raw paper input, devoid of any intermediate workflow or agentic collaboration, thus effectively acting as black-box generators.

For additional context and to evaluate the broader efficacy of our framework, we also benchmark its performance against existing work in the domain of automated software generation, notably ChatDev (Qian et al., 2023), Paper2Code (Seo et al., 2025), and VerilogCoder (Ho et al., 2025). All aforementioned baselines, along with our proposed ArchCraft framework, are evaluated using our custom ArchSynthBench benchmark. Crucially, the RTL and TB code generated by all baselines are evaluated using the identical standards and the o3-mini evaluation agent applied to ArchCraft on ArchSynthBench, ensuring a fair and consistent comparison.

Hardware Implementation Environment. We conduct comprehensive hardware synthesis experiments using Synopsys DC, a widely adopted ASIC synthesis tool. All experiments are performed using Synopsys DC at 200MHz on SIMC 28nm technology on a high-performance server equipped with Intel Xeon Gold CPUs and 256 GB RAM under CentOS 7.

5.2 Result

Main Result. Our experiment encompasses both paper- and code-level evaluation of all baselines, detailed code-level scores in all dimensions can be seen in the Appendix F. The results in Table

Method	LLM	Paper-level				Code-level			
		Func.	Struc.	Synth.	**WA**	RTL	TB	**Comp.**	**WA**
Direct	Gemini 2.0 Flash	15.00	28.00	27.50	21.45	36.83	25.11	✗	32.92
Direct	Qwen3-Coder-480B	27.50	28.13	28.75	27.88	46.46	43.82	✗	45.58
Direct	GPT-4o	27.22	34.72	34.44	30.94	41.11.33	31.20	✗	42.54
Direct	o3-mini	31.00	33.50	35.50	32.45	57.33	43.31	✗	52.66
ChatDev	GPT-4o	41.02	44.96	40.10	42.50	17.50	9.92	✗	14.97
VerilogCoder	o3-mini	41.79	38.21	46.07	40.79	43.57	00.00	✗	34.21
PaperCoder	o3-mini	70.11	69.26	63.07	69.07	50.41	33.93	✗	44.91
ArchCraft	Gemini 2.0 Flash	78.30	77.11	79.67	78.76	57.56	52.06	✓	55.72
ArchCraft	Qwen3-Coder-480B	77.28	80.28	70.13	78.57	66.71	68.40	✓	67.28
ArchCraft	GPT-4o	76.57	76.96	79.21	76.99	72.64	66.26	✓	70.51
ArchCraft	o3-mini	**85.62**	**86.51**	**86.17**	**86.18**	**86.03**	**71.06**	✓	**81.04**

Table 1: Overall evaluation scores on ArchAynthBench, 50 project-level large circuits, over 500 circuit blocks, for Paper-level and Code-level metrics. "Comp." denotes the Compilation Pass Status and "WA" denotes the Weighted Average score. The best performance is denoted in **bold**.

1 demonstrate that ArchCraft outperforms all the baselines, including direct LLM and other agent frameworks such as ChatDev. These results highlight the superior capability of ArchCraft in generating project-level, synthesizable RTL code directly from research papers. Detailed examples of original machine-based evaluation scores can be found in the Appendix Section G.

In both paper- and code-level evaluations, baselines using direct LLM generation consistently demonstrate poor performance. These methods only achieve around 30% structural clarity and synthesisability scores. These severe limitations stem from their black-box, non-iterative generation approach. At the paper-level evaluation, they struggle with the comprehensive understanding and conceptual planning of circuit architectures described in the input papers, resulting in their failing to capture the intended design logic. At the code level, the absence of integration toolchains leads to weak iterative refinement and compilation feedback loops, with generated code failing to compile and even failing logic synthesis.

Similarly, prior multi-agent software frameworks, such as ChatDev, PaperCode, and VerilogCoder, are also incapable of generating synthesizable RTL code. While these agent frameworks have been demonstrated to outperform black-box LLMs at the paper or code level, their inherent design presents limitations. They either lack the capability for project-level RTL code generation or impose stringent input requirements, rendering them ineffective at extracting circuit architectures from abstract paper inputs. Consequently, they underperform our ArchCraft in the paper-to-project-level task. Moreover, the design paradigms of ChatDev and PaperCode inherently lack crucial integration with EDA toolchains, which entirely preclude the compilation and iterative modification of the generated hardware descriptions.

In contrast, ArchCraft stably generates compilable RTL and TB code across all LLM backbones and achieves significantly higher scores across all evaluation metrics. This superior performance is attributed to ArchCraft's Neural-Symbol Graph-based framework, which is capable of sketching circuit structures from paper documents, combined with its compiler toolchain-integrated code traceability and error-correction feedback mechanisms. By analyzing fundamental circuit elements, integrating key error analysis, and implementing targeted corrective measures throughout the process, ArchCraft effectively bridges the gap between closed-source, high-level paper specifications and functionally verifiable, deployable hardware implementations. Due to space limitations, ArchCraft's rectifying analysis is shown in Appendix H.

Human Expert Assessments. To rigorously validate the accuracy of our automated evaluation metrics, we contract three independent human engineering experts to assess the code quality generated by the direct method, other frameworks, and our ArchCraft. The evaluation criteria and scoring methodology used by these human experts have the same dimension as those used in the machine-based evaluation assessment. As presented in Ta-

Method	LLM	Paper-level	Code-level
Direct	Gemini 2.0 Flash	22.17	37.03
Direct	Qwen3-Coder-480B	22.00	44.44
Direct	GPT-4o	26.54	47.73
Direct	o3-mini	29.67	50.74
ChatDev	GPT-4o	26.44	10.89
VerilogCoder	o3-mini	31.27	48.89
PaperCoder	o3-mini	49.68	47.33
ArchCraft	Gemini 2.0 Flash	75.24	70.11
ArchCraft	Qwen3-Coder-480B	76.90	78.54
ArchCraft	GPT-4o	75.59	71.47
ArchCraft	o3-mini	79.13	84.33

Table 2: Human expert evaluation scores from both paper and code levels.

Paper Title	(Success/Total)	Paper Title	(Success/Total)
FlightVGM	6 / 8	PBN	5 / 5
Flex-EGAI	4 / 5	SPARK	7 / 7
LC-MAC	3 / 4	ST-Purning	5 / 5
INSPIRE	4 / 6	MASL_AFU	5 / 7
bitWAVE	5 / 6	ViTA	7 / 8

Table 3: Synthesis success rates across part papers evaluated.

ble 2, the results unequivocally demonstrate that ArchCraft achieves the best ranking among all compared methodologies. For detailed original human evaluation scores and the Pearson correlation coefficient between the scores assigned by human experts and those derived from our machine-based evaluations, see the Appendix J and G.

Hardware Synthesis Evaluation. We conducted synthesis experiments on the RTL circuits for 60 modules, randomly sampled from ten papers in ArchSynthBench, using Synopsys's DC software. Due to space constraints, the PPA report for a subset of these circuits is presented in Appendix I. Table 3 summarizes the synthesis success rate, where success is defined as the proportion of successfully synthesized designs relative to all tested designs, excluding top-level designs and those with known syntactical errors. The results indicate that the majority of designs generated by our framework are synthesizable, meet timing constraints, and are free of violations, which confirms their structural integrity and practical feasibility for downstream ASIC implementation. The few observed synthesis failures were primarily attributed to syntactical issues or port definition mismatches, highlighting a clear direction for improving generation robustness in future work.

As a case study, Table 4 presents the post-synthesis PPA results made by ArchCraft with o3-mini as backbone, for designs derived from the SPARK (Liu et al., 2024a). All the implementations of the SPARK paper successfully pass synthesis with positive slack, confirming their feasibility for timing closure at the target frequency. Notably, the logic synthesizes a single PE that occupies approximately 4k μm^2, with a power consumption of 0.74 mW. While an area discrepancy exists due to specific design details compared to the original SPARK paper, our generated circuit design reproduces the relative scale and power characteristics expected for compute-intensive accelerators, considering the overall relationships between the various designs. Furthermore, the control and interface logic demonstrates sub-mW power consumption and compact area footprints, consistent with their functional roles within the SPARK architecture. These empirical results underscore the efficacy of our ArchCraft framework in faithfully translating high-level architectural descriptions into synthesizable RTL implementations with practical PPA characteristics.

Design	Power(μW)	Slack/CPD(ns)	Area(μm^2)
control_unit	170.1	3.92/0.93	954.68
im2col_pack_engine	1039.9	2.31/2.52	5758
interface	603.3	3.44/1.41	3945.11
memory_controller	939.7	2.85/2.06	6200.41
pe_array	5262.2	1.04/3.81	32345.91
spark_decoder	109.0	3.88/0.98	568.81
spark_encoder	324.4	3.47/1.38	1676.51

Table 4: Synthesis PPA results for SPARK designs using Synopsys DC. All the designs have no time violations. CPD is the abbreviation of Critical Path Delay.

6 Conclusion

This paper introduces ArchCraft, a framework that captures the architectural blueprint using formal graphs and defines functional specifications with symbols, transforming abstract architectural descriptions from academic papers into synthesizable Verilog projects with functional verification. In addition, we constructed ArchSynthBench, a benchmark specifically designed for synthesizing project-level circuits from closed-source academic papers describing especially for artificial intelligence accelerators. Extensive experimental results demonstrate that ArchCraft achieves state-of-the-art performance in machine evaluations, human expert assessments, and physical-level verification, while exhibiting robustness and practicality across different LLM backbones.We expect that the proposed ArchCraft will contribute to accelerating scientific progress by lowering the barriers to hardware replication and enhancing research reproducibility and innovation.

References

Sami Ben Ali, Silviu-Ioan Filip, and Olivier Sentieys. 2024. A stochastic rounding-enabled low-precision floating-point mac for dnn training. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE.

Jinheon Baek, Sujay Kumar Jauhar, Silviu Cucerzan, and Sung Ju Hwang. 2024. Researchagent: Iterative research idea generation over scientific literature with large language models. *arXiv preprint arXiv:2404.07738*.

Andrea Belano, Yvan Tortorella, Angelo Garofalo, Luca Benini, Davide Rossi, and Francesco Conti. 2025. A flexible template for edge generative ai with high-accuracy accelerated softmax & gelu. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*.

Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-chat: Challenges and opportunities in conversational hardware design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6. IEEE.

Oliver Cassidy, Marta Andronic, Samuel Coward, and George A Constantinides. 2025. Reducedlut: Table decomposition with" don't care" conditions. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 36–42.

Kaiyan Chang, Kun Wang, Nan Yang, Ying Wang, Dantong Jin, Wenlong Zhu, Zhirong Chen, Cangyuan Li, Hao Yan, Yunhao Zhou, and 1 others. 2024. Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6.

Chunyun Chen, Lantian Li, and Mohamed M Sabry Aly. 2024a. Vita: a highly efficient dataflow and architecture for vision transformers. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE.

Dongping Chen, Ruoxi Chen, Shilin Zhang, Yaochen Wang, Yinuo Liu, Huichi Zhou, Qihui Zhang, Yao Wan, Pan Zhou, and Lichao Sun. 2024b. Mllm-as-a-judge: Assessing multimodal llm-as-a-judge with vision-language benchmark. In *Forty-first International Conference on Machine Learning*.

Jiasong Chen, Zeming Xie, Weipeng Liang, Bosheng Liu, Xin Zheng, Jigang Wu, and Xiaoming Xiong. 2024c. Quantization-aware optimization approach for cnns inference on cpus. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 878–883. IEEE.

Haotian Cui, Yue Xu, Kuan Pang, Gen Li, Fanglin Gong, Bo Wang, and Bowen Li. 2025. Lumi-lab: a foundation model-driven autonomous platform enabling discovery of new ionizable lipid designs for mrna delivery. *BioRxiv*, pages 2025–02.

Mario Doumet, Marius Stan, Mathew Hall, and Vaughn Betz. 2024. H2pipe: high throughput cnn inference on fpgas with high-bandwidth memory. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 69–77. IEEE.

Xinkuang Geng, Siting Liu, Jianfei Jiang, Kai Jiang, and Honglan Jiang. 2024. Compact powers-of-two: An efficient non-uniform quantization for deep neural networks. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE.

Alireza Ghafarollahi and Markus J Buehler. 2025. Sciagents: automating scientific discovery through bioinspired multi-agent intelligent graph reasoning. *Advanced Materials*, 37(22):2413523.

Google DeepMind. 2025. Gemini - our most intelligent ai models. https://deepmind.google/models/gemini/.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, and 1 others. 2024. A survey on llm-as-a-judge. *arXiv preprint arXiv:2411.15594*.

Yufeng Gu, Alireza Khadem, Sumanth Umesh, Ning Liang, Xavier Servot, Onur Mutlu, Ravi Iyer, and Reetuparna Das. 2025. Pim is all you need: A cxl-enabled gpu-free system for large language model inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 862–881.

Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2025. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 300–307.

Kai Huang, Bowen Li, Siang Chen, Luc Claesen, Wei Xi, Junjian Chen, Xiaowen Jiang, Zhili Liu, Dongliang Xiong, and Xiaolang Yan. 2022. Structured term pruning for computational efficient neural networks inference. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(1):190–203.

Longwei Huang, Chao Fang, Qiong Li, Jun Lin, and Zhongfeng Wang. 2024. A precision-scalable risc-v dnn processor with on-device learning capability at the extreme edge. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 927–932. IEEE.

Zhirui Huang, Rui Ma, Shijie Cao, Ran Shu, Ian Wang, Ting Cao, Chixiao Chen, and Yongqiang Xiong. 2025. Tenet: An efficient sparsity-aware lut-centric architecture for ternary llm inference on edge. *arXiv preprint arXiv:2509.13765*.

Dongseok Im and Hoi-Jun Yoo. 2024. Lutein: Dense-sparse bit-slice architecture with radix-4 lut-based slice-tensor processing units. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 747–759. IEEE.

Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. 2024. Memory sharing with cxl: Hardware and software design approaches. *arXiv preprint arXiv:2404.03245*.

Zexi Ji, Hanrui Wang, Miaorong Wang, Win-San Khwa, Meng-Fan Chang, Song Han, and Anantha P Chandrakasan. 2023. A fully-integrated energy-scalable transformer accelerator supporting adaptive model configuration and word elimination for language understanding on edge devices. In *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE.

Yiyue Jiang, Andrius Vaicaitis, John Dooley, and Miriam Leeser. 2024. Efficient neural networks on the edge with fpgas by optimizing an adaptive activation function. *Sensors*, 24(6):1829.

Leonardo R Juracy, Alexandre M Amory, and Fernando G Moraes. 2022. A comprehensive evaluation of convolutional hardware accelerators. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 70(3):1149–1153.

Dongyun Kam, Myeongji Yun, Sunwoo Yoo, Seungwoo Hong, Zhengya Zhang, and Youngjoo Lee. 2025. Panacea: Novel dnn accelerator using accuracy-preserving asymmetric quantization and energy-saving bit-slice sparsity. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 701–715. IEEE.

Alireza Khataei and Kia Bazargan. 2025. Treelut: An efficient alternative to deep neural networks for inference acceleration using gradient boosted decision trees. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 14–24.

Daehan Kwon, Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gi-Moon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, and 1 others. 2022. A 1ynm 1.25 v 8gb 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep learning application. *IEEE Journal of Solid-State Circuits*, 58(1):291–302.

Eunji Lee, Yoonsang Han, and Gordon Euhyun Moon. 2024. Accelerated block-sparsity-aware matrix reordering for leveraging tensor cores in sparse matrix-multivector multiplication. In *European Conference on Parallel Processing*, pages 3–16. Springer.

Lei Lei and Zhiming Chen. 2024. A reconfigurable fused multiply-accumulate for miscellaneous operators in deep neural network. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

Wenjie Li, Aokun Hu, Gang Wang, Ningyi Xu, and Guanghui He. 2022. Low-complexity precision-scalable multiply-accumulate unit architectures for deep neural network accelerators. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 70(4):1610–1614.

Bo Liu, Anfeng Xue, Ziyu Wang, Na Xie, Xuetao Wang, Zhen Wang, and Hao Cai. 2022a. A reconfigurable approximate computing architecture with dual-vdd for low-power binarized weight network deployment. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 70(1):291–295.

Fangxin Liu, Ning Yang, Haomin Li, Zongwu Wang, Zhuoran Song, Songwen Pei, and Li Jiang. 2024a. Spark: Scalable and precision-aware acceleration of neural networks via efficient encoding. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1029–1042. IEEE.

Fangxin Liu, Ning Yang, Zhiyan Song, Zongwu Wang, Haomin Li, Shiyuan Huang, Zhuoran Song, Songwen Pei, and Li Jiang. 2024b. Inspire: Accelerating deep neural networks via hardware-friendly index-pair encoding. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6.

Jun Liu, Shulin Zeng, Li Ding, Widyadewi Soedarmadji, Hao Zhou, Zehao Wang, Jinhao Li, Jintao Li, Yadong Dai, Kairui Wen, and 1 others. 2025a. Flightvgm: Efficient video generation model inference with online sparsification and hybrid precision on fpgas. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 2–13.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. 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. 2024c. 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, Yao Lu, Wenji Fang, Jing Wang, and Zhiyao Xie. 2025b. Sync-llm: Generation of large-scale synthetic circuit code with hierarchical language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 17361–17376.

Ye Liu, Fei Wu, Neng Zhao, Qirong Zhang, Wenqiang Wang, Yutong Yang, Xiangting Li, Sixu Li, Zili Huang, Shuang Hao, and 1 others. 2022b. Nvp: A flexible and efficient processor architecture for accelerating diverse computer vision tasks including dnn. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 70(1):271–275.

Yi Liu, Hongji Zhang, Yunhao Zhou, Zhengyuan Shi, Changran Xu, and Qiang Xu. 2025c. Deeprtl2: A versatile model for rtl-related tasks. *arXiv preprint arXiv:2506.15697*.

Mukul Lokhande, Gopal Raut, and Santosh Kumar Vishvakarma. 2025. Flex-pe: Flexible and simd multiprecision processing element for ai workloads. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.

Binglei Lou, Richard Rademacher, David Boland, and Philip HW Leong. 2024. Polylut-add: Fpga-based lut inference with wide inputs. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 149–155. IEEE.

Yingchang Mao, Mingyu Shu, and Qiang Liu. 2024. Pbn: Progressive batch normalization for dnn training on edge device. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

Satvik Maurya and Swamit Tannu. 2022. Compaqt: Compressed waveform memory architecture for scalable qubit control. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1059–1077. IEEE.

Zhaoteng Meng, Lin Shu, Jianing Zeng, Zhan Li, Kailin Lv, Haoyue Yang, and Jie Hao. 2024. Masl-afu: A high memory access efficiency 2-d scalable lut-based activation function unit for on-device dnn training. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.

Zhiwen Mo, Lei Wang, Jianyu Wei, Zhichen Zeng, Shijie Cao, Lingxiao Ma, Naifeng Jing, Ting Cao, Jilong Xue, Fan Yang, and 1 others. 2025. Lut tensor core: A software-hardware co-design for lut-based low-bit llm inference. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 514–528.

Bardia Nadimi, Ghali Omar Boutaib, and Hao Zheng. 2024. Pyranet: A multi-layered hierarchical dataset for verilog. *arXiv preprint arXiv:2412.06947*.

Bardia Nadimi, Ghali Omar Boutaib, and Hao Zheng. 2025. Verimind: Agentic llm for automated verilog generation with a novel evaluation metric. *arXiv preprint arXiv:2503.16514*.

Andre Nakkab, Sai Qian Zhang, Ramesh Karri, and Siddharth Garg. 2024. Rome was not built in a single step: Hierarchical prompting for llm-based chip design. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, pages 1–11.

OpenAI. 2024. Hello gpt-4.

OpenAI. 2025. Openai o3-mini. `https://openai.com/index/openai-o3-mini/`. Accessed: 10 February 2025.

Zhewen Pan, Joshua San Miguel, and Di Wu. 2024. Carat: Unlocking value-level parallelism for multiplier-free gemms. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 167–184.

Gunho Park, Hyeokjun Kwon, Jiwoo Kim, Jeongin Bae, Baeseong Park, Dongsoo Lee, and Youngjoo Lee. 2025. Figlut: An energy-efficient accelerator design for fp-int gemm using look-up tables. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1098–1111. IEEE.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, and 1 others. 2023. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Ladan Sayadi, Somayeh Timarchi, and Akbar Sheikh-Akbari. 2023. Two efficient approximate unsigned multipliers by developing new configuration for approximate 4: 2 compressors. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(4):1649–1659.

Samuel Schmidgall and Michael Moor. 2025. Agentrxiv: Towards collaborative autonomous research. *arXiv preprint arXiv:2503.18102*.

Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Zicheng Liu, and Emad Barsoum. 2025. Agent laboratory: Using llm agents as research assistants. *arXiv preprint arXiv:2501.04227*.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. 2025. Paper2code: Automating code generation from scientific papers in machine learning. *arXiv preprint arXiv:2504.17192*.

Man Shi, Vikram Jain, Antony Joseph, Maurice Meijer, and Marian Verhelst. 2024. Bitwave: Exploiting column-based bit-level sparsity for deep learning acceleration. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 732–746. IEEE.

Jinwei Tang, Jiayin Qin, Kiran Thorat, Chen Zhu-Tian, Yu Cao, Caiwen Ding, and 1 others. 2024. Hivegen–hierarchical llm-based verilog generation for scalable chip design. *arXiv preprint arXiv:2412.05393*.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, 29(3):1–31.

Kevin Tien, Ken Inoue, Scott Lekuch, David J Frank, Sudipto Chakraborty, Pat Rosno, Thomas Fox, Mark Yeck, Joseph A Glick, Raphael Robertazzi, and 1 others. 2022. A cryo-cmos transmon qubit controller and verification with fpga emulation. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 13–16. IEEE.

Shreyas Kolala Venkataramanaiah, Jian Meng, Han-Sok Suh, Injune Yeo, Jyotishman Saikia, Sai Kiran Cherupally, Yichi Zhang, Zhiru Zhang, and Jae-Sun Seo. 2023. A 28-nm 8-bit floating-point tensor core-based programmable cnn training processor with dynamic structured sparsity. *IEEE Journal of Solid-State Circuits*, 58(7):1885–1897.

Bowei Wang, Qi Xiong, Zeqing Xiang, Lei Wang, and Renzhi Chen. 2025. Rtlsquad: Multi-agent based interpretable rtl design. *arXiv preprint arXiv:2501.05470*.

Chenyi Wen, Haonan Du, Zhengrui Chen, Li Zhang, Qi Sun, and Cheng Zhuo. 2024. Pace: A piece-wise approximate and configurable floating-point divider for energy-efficient computing. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE.

Jun-Shen Wu and Ren-Shuo Liu. 2023. Fm-p2l: An algorithm hardware co-design of fixed-point msbs with power-of-2 lsbs in cnn accelerators. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*, pages 407–414. IEEE.

Peiyang Wu, Nan Guo, Xiao Xiao, Wenming Li, Xiaochun Ye, and Dongrui Fan. 2025. Itertl: An iterative framework for fine-tuning llms for rtl code generation. In *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

Bai-Kui Yan and Shanq-Jang Ruan. 2022. Area efficient compression for floating-point feature maps in convolutional neural network accelerators. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 70(2):746–750.

Yifan Yang, Joel S Emer, and Daniel Sanchez. 2024. Trapezoid: A versatile accelerator for dense and sparse matrix multiplications. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 931–945. IEEE.

Zhewen Yu and Christos-Savvas Bouganis. 2024. Auto ws: Automate weights streaming in layer-wise pipelined dnn accelerators. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE.

Danqing Zhang, Baoting Li, Hang Wang, Xuchong Zhang, and Hongbin Sun. 2024. An efficient sparse-aware summation optimization strategy for dnn accelerator. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*.

Yuan Zhang, Lele Peng, Lianghua Quan, Yonggang Zhang, Shubin Zheng, and Hui Chen. 2023b. High-precision method and architecture for base-2 softmax function in dnn training. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(8):3268–3279.

Xuanle Zhao, Deyang Jiang, Zhixiong Zeng, Lei Chen, Haibo Qiu, Jing Huang, Yufeng Zhong, Liming Zheng, Yilin Cao, and Lin Ma. 2025a. Vincicoder: Unifying multimodal code generation via coarse-to-fine visual reinforcement learning. *arXiv preprint arXiv:2511.00391*.

Xuanle Zhao, Xianzhen Luo, Qi Shi, Chi Chen, Shuo Wang, Zhiyuan Liu, and Maosong Sun. 2025b. Chartcoder: Advancing multimodal large language model for chart-to-code generation. *arXiv preprint arXiv:2501.06598*.

Xuanle Zhao, Zilin Sang, Yuxuan Li, Qi Shi, Weilun Zhao, Shuo Wang, Duzhen Zhang, Xu Han, Zhiyuan Liu, and Maosong Sun. 2025c. Autoreproduce: Automatic ai experiment reproduction with paper lineage. *arXiv preprint arXiv:2505.20662*.

Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Ziyuan Nan, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, and 1 others. 2024. Codev: Empowering llms for verilog generation through multi-level summarization. *arXiv preprint arXiv:2407.10424*.

Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. 2025d. Mage: A multi-agent engine for automated rtl code generation. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, pages 1–7. IEEE.

Zhongyu Zhao, Rujian Cao, Ka-Fai Un, Wei-Han Yu, Pui-In Mak, and Rui P Martins. 2022. An fpga-based transformer accelerator using output block stationary dataflow for object recognition applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 70(1):281–285.

Mengying Zheng, Xianting Yang, Jiaxuan Tang, Yiting Zhu, Xinhui Liu, Jia Zhang, Zhengyuan Chen, Kai Liu, Yun Chen, and Yibo Jiang. 2024. Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation. *arXiv preprint arXiv:2404.09340*.

A Ethics Statement

The primary objective of this work is to automate the replication of experiments detailed in existing, publicly available research papers. While the methodologies themselves are drawn from the public domain, which generally implies transparency, it is important to acknowledge the potential for data leakage associated with the use of our system. Users should therefore be mindful of this possibility, particularly when the replication process might involve sensitive datasets or generate intermediate results that could inadvertently disclose information.

B Prompts

In this section, we provide a detailed exposition of the four phases within our framework. The specific prompts utilized at each stage of this process are presented in Section B.1 is Graphing Prompts, Section B.2 is Symbolize Prompts, Section B.3 is Coding Prompts, including original coding prompts and rectifying.

B.1 Graphing Phase

We show 4 steps' prompts in our Graphing Phase. Figure 3 is Define Graph Scope and Theme. Figure 4 is Define Graph Nodes and Static Links. Figure 5 is Plan Graph Direction and Internal Node Properties. Figure 6 is Add Global Constraints.

B.2 Symbolize Phase

We show 3 steps' prompts in our Graphing Phase. Figure 7 is Interface Protocol Design, Internal Logic Design,and Verification & Implementation Plan.

B.3 Coding Phase

In this section, we demonstrated the decoupling generation of RTL and TB code in Figure 8, and the prompt used in the rectifying process of the generated code in Figure 9.

Prompts for graphing papers. I

System Role Prompt:
You are an expert hardware engineer and RTL designer with a deep understanding of digital design and hardware implementation.
You will receive a research paper in <paper_format> format.
Your task is to create a detailed and efficient plan to implement the hardware described in the paper.
This plan should align precisely with the paper's hardware architecture, timing requirements, and performance metrics.
The plan must be clear, structured, and focused on hardware implementation details.
Task Prompt:
1. We want to implement the hardware described in the attached paper.
2. The authors did not release any official RTL code, so we have to plan our own implementation.
3. Before writing any Verilog code, please outline a comprehensive plan that covers:
- Key details from the paper's Hardware Architecture.
- Important aspects of Implementation, including module hierarchy, interfaces, timing requirements, and verification strategy.
4. The plan should be as detailed and informative as possible to help us write the final RTL code later.
Instruction Prompt:
The response should give us a strong roadmap for hardware implementation, making it easier to write the RTL code later.

Figure 3: Prompts for graphing papers.

Prompts for graphing papers. II

General Prompt:
Your goal is to create a concise, usable, and complete hardware system design for implementing the paper's method. Use appropriate hardware design practices and keep the overall architecture modular.
Based on the plan for implementing the paper's main method, please design a concise, usable, and complete hardware system. Keep the architecture modular and make effective use of standard hardware design patterns.
Format Prompt:
Please note that this is just an example and in the processing of the actual module, there is no need to strictly adhere to the file names and the number of modules therein. Please refer to the actual analysis results.
Action Prompt:
Follow the instructions of the node to generate the output and ensure that it adheres to the format example. Please remember that the file list and module names should be adjusted according to the actual designed functions and do not have to strictly follow the examples.

Figure 4: Prompts for graphing papers.

Prompts for graphing papers. III

General Prompt:
Your goal is to break down tasks according to PRD/technical design, generate a task list, and analyze task dependencies.
You will break down tasks, analyze dependencies.
You outline a clear PRD/technical design for implementing the paper's hardware method.
Now, let's break down tasks according to PRD/technical design, generate a task list, and analyze task dependencies.
The Logic Analysis should not only consider the dependencies between modules but also provide detailed descriptions to assist in writing the RTL code needed to implement the paper.
Format Prompt:
Please note that this is just an example and in the processing of the actual module, there is no need to strictly adhere to the file names and the number of modules therein. Please refer to the actual analysis results
Action Prompt:
Follow the node instructions above, generate your output accordingly, and ensure it follows the given format example.

Figure 5: Prompts for graphing papers.

Prompts for graphing papers. IV

General Prompt:
You write elegant, modular, and maintainable RTL code. Adhere to hardware design guidelines. Based on the paper, plan, and design specified previously, follow the "Format Example" and generate the code. Extract the hardware details from the above paper (e.g., clock frequency, data width, memory size, etc.), follow the "Format example" and generate the code. DO NOT FABRICATE DETAILS — only use what the paper provides. You must write ‘config.yaml‘.

Figure 6: Prompts for graphing papers.

Prompts for symbolize. I

System Role Prompt: You are an expert hardware interface designer. Your task is to take a high-level module definition and design its detailed physical interfaces and protocols.
Task Prompt: Based on the high-level plan and the module list, refine the interfaces for the module: [Module_Name_From_Graphing].
You must define:
Handshake Protocols: Specify the exact valid/ready or ack/req logic for each data port.
Memory Interfaces: If this module connects to memory, define the protocol (e.g., AXI-Lite, AXI-Stream, or simple BRAM interface) including all necessary signals (ar_addr, ar_valid, aw_addr, w_data, b_ready, etc.).
Detailed Timing: Specify any multi-cycle signal behaviors or interface timing constraints not covered by the global config.
Control Signals: Define any new control signals required for this specific module's operation (e.g., start_processing, op_mode, done_tick).

Prompts for symbolize. II

System Role Prompt: You are an expert RTL micro-architect. Your task is to design the internal control and data path logic for a given hardware module based on its function and interfaces.
Task Prompt: Design the internal micro-architecture for module:[Module_Name_From_Graphing].
You must provide two separate but related designs:
State Machine (FSM) Design:
Identify FSMs: List all required Finite State Machines.
Define States: For each FSM, list its states.
Define Transitions: Describe the logic that causes transitions between states.
Control Logic: Specify the control signals generated in each state.
Reset Behavior: Define the FSM's state upon rst_n assertion.
Data Path Design:
Data Flow: Describe the path data takes through the module, from input ports to output ports.
Key Components: List the necessary data path components.
Pipeline Stages: If pipelining is required, define the logic for each stage.
Data Formats: Specify any changes in data format.

Prompts for symbolize. III

System Role Prompt: You are an expert verification engineer and physical designer. Your task is to create a test plan and define implementation constraints for a given hardware module.
Task Prompt: Create the verification plan and define implementation constraints for module: [Module_Name_From_Graphing].
Verification Requirements:
Test Scenarios: Define key scenarios to test.
Coverage Points: List critical coverage points.
Test Vectors: Provide 2-3 example test vectors.
Implementation Constraints:
Module-Specific Constraints: Define constraints specific to this module that override or add to the global config .
Optimization Goals: Specify the priority for this block.

Figure 7: Prompts for symbolize.

Prompts for coding. I

System Role Prompt:
You are an expert researcher and software engineer with a deep understanding of experimental design and reproducibility in scientific research.
You will receive a research paper in format, an overview of the plan, a Design in JSON format consisting of "Implementation approach", "File list", "Data structures and interfaces", and "Program call flow", followed by a Task in JSON format that includes "Required packages", "Required other language third-party packages", "Logic Analysis", and "Task list", along with a configuration file named "config.yaml".
Your task is to write code to reproduce the experiments and methodologies described in the paper.
The code you write must be elegant, modular, and maintainable, adhering to Google-style guidelines.
The code must strictly align with the paper's methodology, experimental setup, and evaluation metrics.
Write code with triple quotes.
RTL Coding Prompt:
Based on the planning, design, and tasks of the thesis and the configuration file "config.yaml" specified earlier, write the code in accordance with the "Format Example".
We have done the file list. Next, you must only write the todo file name.
1. Only one file: Do your best to achieve this with only one file.
2. Complete code: Your code will be part of the entire project, so please implement complete, reliable, and reusable code snippets.
3. Follow the design: You must adhere to "Data Structures and Interfaces". Don't change any design. Do not use public member functions that do not exist in the design.
4. Carefully check that no necessary classes/functions are missing in this file.
5. Before using an external variable/module, please make sure to import it first.
6. Write down every detail of the code and do not leave any tasks to be done.
7. Reference configuration: You must use the configuration in "config.yaml". Do not forge any configuration values.

Prompts for coding. II

TB Coding Prompt:
You are an expert who can automate the RTL by using a fully autonomous toolchain. You are invited to teach us how to create chips by using a fully autonomous toolchain for digital layout generation across die sizes, process nodes, and foundry options.
Your specific task now is to write a complete and executable TB for the Verilog module named ‘module_name‘. You must leverage the provided detailed module analysis to infer the module's interface and write a comprehensive and accurate TB.
Here is the detailed analysis of the module and its context from our autonomous toolchain: $module\_analysis\_content$
In addition, here are a few things you should notice: $rules$

Figure 8: Prompts for coding.

Prompts for Rectifying. I

Error Finding System Role Prompt:
You are an experienced digital circuit engineer, proficient in Verilog/SystemVerilog design, simulation and debugging.
Your task is to analyze the provided Verilog simulation logs, identify the root cause of errors or warnings, and determine which file (such as' rtlfile.v 'or' testbenchfile.v ') is the main responsible for the problem.
Summarize the core lessons learned from this mistake in a concise sentence, with the focus on proposing a universal coding guideline to prevent similar mistakes from happening in the future.

Prompts for Rectifying. III

Error Analysis System Role Prompt:
You are an experienced digital circuit engineer, proficient in Verilog/SystemVerilog design, simulation and debugging.
Your task is to conduct an in-depth analysis of the provided Verilog simulation logs, RTL codes, and TB codes, identify the root causes of errors or warnings, and generate a detailed, fixe-oriented internal analysis result.
This analysis should clearly point out where the problem lies, why it occurs, and the potential methods and considerations for solving it (for example, if it involves width, it is necessary to consider how to correctly crop or expand, or whether the variable definition needs to be modified).
Please output your analysis results in JSON format, including the internal analysis text field.

Prompts for Rectifying. III

Error Fixing System Role Prompt:
You are an experienced digital circuit engineer, proficient in Verilog/SystemVerilog design, simulation and debugging.
Your task is to fix the Verilog code based on the provided original RTL/TBcode, simulation logs, and detailed internal analysis results .
Only modify the necessary code in the problematic files (RTL code and/or TB code) to solve the problem. If a file does not need to be modified, please return its original content. The code should be completely and correctly formatted in the verilog code block.
Please output your reply in JSON format.

Figure 9: Prompts for rectifying.

C Differences between Paper and Document

In this subsection, we provide a detailed comparison of the differences 5 between academic papers and industrial design documents as inputs to the ArchCraft framework. This distinction represents the foundational starting point for the entire ArchCraft process and the design motivation that led to the creation of its Neural-Symbolic Graph-based Framework.

D Scoring Criteria

To support the quantitative and qualitative analyses presented in the main text, this appendix provides the detailed evaluation rubrics used to assess the quality of the generated hardware designs. These rubrics are explicitly divided into two parts: criteria for the design implementation (RTL) and criteria for its verification environment (Testbench).

Table 6 details the evaluation rubric for the **RTL Design**. This rubric is structured around five core dimensions:

- **Functional Correctness:** Assesses whether the design accurately implements the required algorithms and logic.
- **Robustness:** Examines the design's handling of boundary conditions, exceptions, and timing issues (e.g., CDC).
- **Structural Fidelity:** Evaluates the code's modularity, structural clarity, and naming conventions.
- **Synthesis Compatibility:** Ensures the code is synthesizable and evaluates the reasonableness of its resource utilization.
- **Readability:** Assesses the quality of comments and documentation, which relates to design maintainability.

Correspondingly, **Table 7** presents the evaluation rubric for the **TB Design**. This rubric focuses on four key aspects:

- **Test Case Coverage:** Evaluates whether the stimulus adequately covers functional paths, boundaries, and corner cases.
- **Robustness:** Focuses on whether the assertions are accurate, effective, and stable under various stimuli.
- **Stimulus Generation Quality:** Assesses the use of directed, random, and constrained-random strategies.
- **Maintainability:** Examines the clarity, extensibility, and debuggability of the testbench code itself.

As mentioned in the main text, each dimension is scored out of 100 points. The specific questions listed within these tables provide the basis for fine-grained scoring and qualitative feedback by human evaluators.

E Paper in ArchSynthBench Details

In this subsection, we detail the corpus of academic papers that constitutes the **ArchSynthBench** benchmark. These papers serve as the high-level, abstract design specifications that our framework, ArchCraft, aims to interpret and implement. The complete list of these papers is presented in **Table 8**, which is organized to highlight the breadth and timeliness of our curated corpus.

The table provides three key pieces of information for each entry:

- **Year:** The publication year of the paper. This chronological organization (spanning 2022 to 2025) demonstrates that the benchmark is contemporary and built upon state-of-the-art research in hardware acceleration.
- **Paper:** The common acronym or a short name for the paper, along with its bibliographic citation key (e.g., `(Zhao et al., 2022)`). This provides full transparency and traceability, allowing researchers to reference the original design specifications used in our evaluation.
- **Theme** A set of descriptive terms summarizing the paper's core technical contribution, target application, or key architectural paradigm.

An analysis of the **Theme** column reveals the extensive diversity of our benchmark. The corpus was intentionally curated to move beyond simple, canonical designs and to reflect the complexity of the modern hardware landscape. The topics span:

- **Classic Acceleration Domains:** Such as CNN Accelerators (`CNN Accelerator`), Approximate Computing (`Approximate Computing / BNN`), and efficient inference (`Efficient NN Inference`).
- **Modern Neural Architectures:** Including dedicated hardware for Transformers (`FPGA Transformer`), Vision Transformers (`Vision Transformer / Accelerator`), and Large Language Models (`LLM Inference / PIM / CXL`).

Feature / Attribute		Academic Paper	Industrial Doc.
Accessibility	Publicly Searchable	✓	✗
	Abundant in Quantity	✓	✗
	Internally Confidential	✗	✓
Content Focus	Core Algorithm / Theory	✓	(✓)
	Rigorous Validation / Proofs	✓	✗
	Detailed Circuit-level Specs	✗	✓
	Implementation Details (APIs, etc.)	✗	✓
	Edge Cases	✗	✓
Form	Standardized Structure	✓	✗
	High-Level Abstraction	✓	✗
	Low-Level Granularity	✗	✓
Lifecycle	Static / Archived	✓	✗
	Dynamic / Living Document	✗	✓

Table 5: Feature-based Comparison of Academic Papers and Industrial Design Documents.

Functional Correctness	• Does the RTL correctly implement the key algorithms, logic, or functional points described in the logical analysis? Is the functionality complete?
	• Does it faithfully reproduce the architecture, dataflow, and bit-width details specified in the logical analysis?
Robustness	• Does the RTL handle all input cases, boundary conditions, and potential exceptions (e.g., divide-by-zero, overflow, illegal inputs)?
	• Is the reset logic clear and effective? Is Clock Domain Crossing (CDC) handled correctly (if applicable)?
	• Is the design complete, unambiguous, and synthesizable without errors?
Structural Fidelity	• Is the code structure clear? Does it adhere to the logical layering from the analysis (if mentioned)?
	• Is the module partitioning reasonable and the naming convention consistent?
Synthesis Compatibility	• Can the design be successfully synthesized by mainstream tools? Does it avoid non-synthesizable constructs?
	• Is the resource utilization reasonable and aligned with the goals set in the logical analysis?
Readability	• Does the code have sufficient comments? Are signal names clear and meaningful?
	• Does it include auxiliary documentation (e.g., parameter descriptions, structural comments)?

Table 6: Evaluation Rubric for RTL Design (100 points per dimension)

Test Case Coverage	• Does the TB generate sufficient and representative test cases to cover all functional paths of the RTL?
	• Does it cover boundary conditions, typical values, random values, and corner cases of the input space?
	• Does it consider various timing combinations of control signals (e.g., reset, enable)?
Robustness	• Do the assertions (‘assert‘) accurately reflect the RTL’s expected behavior?
	• Are assertions granular enough to effectively capture errors in outputs or internal states?
	• Are assertions robust, remaining stable and correct under varying or randomized stimulus?
	• Are there checks on key intermediate signals, not just the final outputs?
Stimulus Generation Quality	• Is the method for generating input stimulus logical and flexible?
	• Does it use a mix of directed, random, and constrained-random stimulus to enhance coverage?
	• Is the generation of clock and reset signals compliant with the design specificaåtion?
Maintainability	• Is the testbench code clear, well-structured, and easy to understand?
	• Is it easy to modify, extend with new test cases, or debug?
	• Is the code free of redundancy and unnecessary complexity?

Table 7: Evaluation Rubric for TB Design (100 points per dimension)

- **Specific Hardware Techniques:** Such as `LUT-based NN`, `Sparsity`, `Quantization`, and Accelerator-in-Memory (`AiM`).
- **Emerging Paradigms:** The benchmark even includes forward-looking topics like CXL-based memory systems (`Memory Systems / CXL`) and Quantum Computing control (`Quantum Computing / Control`).

In summary, the paper corpus detailed in Table 8 provides a rich, challenging, and representative set of tasks. This diversity is essential for robustly evaluating the capability of any high-level synthesis agent to generalize across different domains and effectively bridge the gap from abstract textual specifications to concrete hardware implementations.

F Code-Level Evaluation Scores

In Table 1, five key metrics are folded into RTL and TB, and we record detailed scores in all dimensions for code-level in Table 9.

G Original Scores

In this section, we provide the raw score lists from both machine and human evaluations as illustrative examples. Given that an exhaustive presentation of all circuit modules is infeasible, we selectively present the scores for nearly 100 circuit modules derived from the 10 papers in batch1 of ArchSynthBench, as evaluated by othermethod-o3-mini and archcraft-o3-mini.

Specifically, we provide the LLM evaluation results of Direct-o3-mini from Table 1, presented in Table 10, and the human evaluation results from Table 2, presented in Table 11; we also provide the LLM evaluation results of ArchCraft-o3-mini from Table 1, presented in Tables 12 and 13, as well as the human evaluation results from Table 2, presented in Table 14.

H Analysis

Rectifying Analysis. Our analysis focus on the mean and variance of feedback-based rectifica-

Year	Paper Name	
2022	OBS-TA (Zhao et al., 2022)	Efficient Compression (Yan and Ruan, 2022)
	LC-MAC (Li et al., 2022)	Comprehensive Evaluation (Juracy et al., 2022)
	ST-Purning (Huang et al., 2022)	Approx. Arch. (Liu et al., 2022a)
	COMPAQT (Maurya and Tannu, 2022)	NVP (Liu et al., 2022b)
	Cryo-CMOS Transmon (Tien et al., 2022)	GDDR6-Based AiM (Kwon et al., 2022)
2023	AFIES (Ji et al., 2023)	Accurate Binary-Stochastic (Zhang et al., 2023b)
	Efficient Multipliers (Sayadi et al., 2023)	High-Precision Softmax (Zhang et al., 2023b)
	FP Tensor Core (Venkataramanaiah et al., 2023)	FM-P2L (Wu and Liu, 2023)
2024	INSPIRE (Liu et al., 2024b)	PACE (Wen et al., 2024)
	PBN (Mao et al., 2024)	PolyLUT-Add (Lou et al., 2024)
	SPARK (Liu et al., 2024a)	RFMA (Lei and Chen, 2024)
	BitWave (Shi et al., 2024)	Precision-Scalable (Huang et al., 2024)
	MASL-AFU (Meng et al., 2024)	Trapezoid (Yang et al., 2024)
	AESA (Zhang et al., 2024)	ViTA (Chen et al., 2024a)
	AutoWS (Yu and Bouganis, 2024)	Carat (Pan et al., 2024)
	Memory Sharing with CXL (Jain et al., 2024)	Block-Sparsity (Lee et al., 2024)
	ENN (Jiang et al., 2024)	Quantization-aware (Chen et al., 2024c)
	CPoT (Geng et al., 2024)	LUTein (Im and Yoo, 2024)
	FPMAC (Ali et al., 2024)	TreeLUT (Khataei and Bazargan, 2025)
	H2PIPE (Doumet et al., 2024)	
2025	Flex-EGAI (Belano et al., 2025)	FIGLUT (Park et al., 2025)
	FlightVGM (Liu et al., 2025a)	TENET (Huang et al., 2025)
	PIM Is All You Need (Gu et al., 2025)	Panacea (Kam et al., 2025)
	RLUT (Cassidy et al., 2025)	Flex-PE (Lokhande et al., 2025)
	LUTTC (Mo et al., 2025)	

Table 8: List of Papers

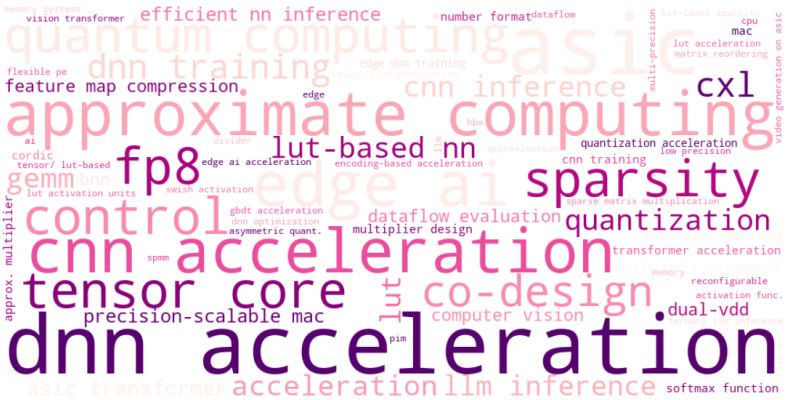

Figure 10: Theme of the papers.

Method	LLM	RTL		TB		
		Robust.	Read.	Cov.	Assert.	Stim.
Direct	Gemini 2.0 Flash	30.33	43.33	23.61	19.11	32.61
Direct	Qwen3-Coder-480B	42.08	50.83	46.74	41.74	42.99
Direct	GPT-4o	32.50	49.72	35.27	23.61	34.72
Direct	o3-mini	52.33	62.33	53.71	31.11	45.11
ChatDev	GPT-4o	17.08	17.92	17.63	1.11	11.03
VerilogCoder	o3-mini	34.62	52.50	00.00	00.00	00.00
PaperCoder	o3-mini	51.07	49.76	45.20	20.81	35.78
ArchCraft	Gemini 2.0 Flash	53.38	61.73	61.16	36.73	58.30
ArchCraft	Qwen3-Coder-480B	62.91	70.51	72.25	**64.44**	68.51
ArchCraft	GPT-4o	68.25	77.03	69.50	60.93	68.37
ArchCraft	o3-mini	**80.95**	**91.13**	**75.81**	62.58	**74.77**

Table 9: The code-level evaluation is based on five key metrics. The best performance is denoted in **bold**.

Design Name	Paper-level			Code-level				
	Fidelity	Structural	Synth.	RTL		TB		
				Sou.	Mai.	TC	AQ	SQ
Flex-EGAI	4.0	2.0	5.0	5.0	6.0	5.0	3.0	4.0
OBS-TA	4.0	4.0	3.5	6.0	6.5	4.6	5.0	4.5
INSPIRE	4.0	4.0	4.0	5.5	6.5	5.5	2.0	4.5
LC-MAC	4.0	4.5	4.0	5.5	6.5	5.0	2.0	4.0
PBN	3.0	3.5	2.5	4.5	6.0	4.5	0.0	2.0
SPARK	2.0	3.0	4.0	5.0	6.0	5.0	3.0	5.0
BitWave	3.0	4.0	3.5	4.5	6.0	5.0	3.0	4.5
FlightVGM	1.0	2.0	3.0	4.5	6.0	6.0	5.5	5.0
MASL-AFU	3.0	3.0	3.0	5.5	6.0	6.0	2.0	4.5
ST-Purning	3.0	3.5	3.0	5.5	6.0	6.0	4.5	6.0

Table 10: Machine Scoring Results Using o3-mini as the LLM in the Direct Method.

tion steps needed to correct code errors within the ArchCraft framework, utilizing Gemini 2.0 Flash, Qwen3-Coder-480B, GPT-4o (OpenAI, 2024) and o3-mini as backbone models. As Table 15 illustrates, Gemini 2.0 Flash required the most rectification iterations, averaging 7.58, indicating a significant need for corrections. Conversely, o3-mini showed the lowest average and minimal variance in rectification counts, suggesting it consistently produced more stable outputs. These findings collectively indicate that our framework not only enhances the performance of all LLM models through its correction process, but also that the RTL code generated by o3-mini possesses superior initial correctness and lower integration costs.

I Physical Implementation

In this section, we show the DC results of the final case study in the main text, three designs are shown. Figures 12, Figure 13, Figure 14 show the PPA of the pe_array, Figure 15, Figure 16, Figure 17 show the PPA of the spark_encoder, Figure 18, Figure 19, Figure 20 show the PPA of the spark_decoder.

J Pearson correlation

Furthermore, to ascertain the feasibility and reliability of our automated scoring system, we examine their agreement by calculating the Pearson correlation coefficient between the scores assigned by human experts and those derived from our machine-based evaluations, as illustrated in Figure 11, total Pearson r = 0.82, which demonstrates strong consistency.

Paper Name	Paper-level			Code-level				
	Fidelity	Structural	Synth.	RTL		TB		
				Sou.	Mai.	TC	AQ	SQ
INSPIRE	2.0	3.0	9.0	5.0	6.0	5.0	2.0	4.0
MASL-AFU	3.0	2.0	8.0	5.0	6.0	6.0	2.0	4.0
SPARK	2.0	2.0	9.0	5.0	5.0	4.0	3.0	5.0

Table 11: Human Expert Scoring Results Using o3-mini as the LLM in the Direct Method.

Paper Name	Design Name	Paper-level			Code-level				
		Fidelity	Structural	Synth.	RTL		TB		
					Sou.	Mai.	TC	AQ	SQ
Flex-EGAI	softex_streamer	7.0	8.0	8.0	9.0	8.0	8.0	5.0	8.0
	top	5.0	6.0	7.0	5.0	8.0	5.0	2.0	5.0
	tpu	9.0	8.0	8.0	8.0	9.0	7.0	6.0	7.0
	riscv_cluster	9.0	9.0	9.0	8.0	9.0	7.0	5.0	7.0
	softex_datapath	9.0	9.0	8.0	5.0	6.0	6.0	5.0	5.0
	softex_control	9.0	8.0	9.0	8.0	9.0	8.0	6.0	8.0
	memory_controller	8.0	8.0	8.0	8.0	9.0	7.0	5.0	7.0
OBS-TA	config	10.0	9.0	9.0	10.0	10.0	8.0	8.0	8.0
	obs_datapath	8.0	8.5	8.0	7.5	9.0	7.5	5.0	7.5
	top	6.0	7.5	8.0	7.0	7.5	8.0	7.0	7.5
	obs_controller	9.0	10.0	9.5	8.5	10.0	7.5	6.5	7.5
	nonlinear_block	6.0	6.0	7.5	5.0	7.5	6.0	5.0	6.0
	mem_controller	9.0	8.5	9.0	4.5	7.0	6.0	4.5	6.0
INSPIRE	accumulation_unit	9.0	9.0	9.0	8.5	9.5	8.0	5.0	8.0
	activation_encoder	9.0	9.0	9.0	8.5	9.0	9.0	8.0	7.5
	control_unit	9.0	9.5	9.0	5.5	6.5	4.5	3.0	5.5
	ip_pe	9.0	9.5	9.0	9.0	9.5	10.0	9.0	9.5
	ip_pe_array	9.5	9.5	10.0	9.0	9.5	8.0	7.0	7.5
	memory_interface	9.0	9.0	9.0	8.0	9.0	8.0	7.0	8.0
	top	8.0	9.0	8.5	7.5	9.0	7.0	5.0	7.5
	weight_buffer	9.0	9.0	10.0	5.5	6.0	4.5	5.0	4.5
LC-MAC	bit_brick	8.0	9.0	8.5	9.0	9.0	9.0	8.5	8.0
	rfu_core	7.5	9.0	8.5	8.0	8.0	8.5	6.0	7.5
	shift_add_unit	9.0	9.0	9.0	9.0	9.5	7.5	8.0	7.5
	bbu_unit	9.5	9.5	9.0	9.0	9.5	7.5	5.0	7.5
	top	9.0	9.0	9.0	8.0	9.5	9.0	6.0	8.5
	mode_controller	9.0	9.5	10.0	9.5	9.5	8.0	7.5	8.0
PBN	pbn_control_unit	9.0	9.0	9.0	8.5	9.0	8.5	5.0	7.5
	pbn_datapath	8.0	7.0	7.5	7.5	9.0	8.0	5.0	7.5
	pbn_fifo	10.0	9.0	10.0	9.0	9.5	8.5	8.0	9.0
	pbn_nps	8.0	9.0	8.0	7.5	9.0	7.0	5.0	7.0
	pbn_scs	9.0	9.0	8.5	8.0	9.0	7.0	5.0	7.0
	pbn_top	9.0	10.0	9.0	8.5	10.0	7.0	6.0	7.5

Table 12: Machine Scoring Results Using o3-mini as the LLM in the ArchCraft. I

Paper Name	Design Name	Paper-level			Code-level				
		Fidelity	Structural	Synth.	RTL		TB		
					Sou.	Mai.	TC	AQ	SQ
SPARK	control_unit	9.0	9.0	9.0	9.0	10.0	8.0	9.0	8.0
	im2col_pack_engine	9.0	9.5	10.0	9.0	9.5	7.5	5.0	8.5
	interface	8.5	8.0	9.0	8.0	8.0	6.0	6.0	7.0
	memory_controller	9.0	9.0	9.5	8.5	9.0	7.0	6.0	7.0
	pe_array	8.0	9.0	9.0	9.0	9.5	7.0	5.0	7.0
	spark_decoder	10.0	10.0	10.0	10.0	10.0	9.0	5.0	7.5
	spark_encoder	9.0	9.0	9.5	9.5	10.0	8.0	7.0	7.0
	top	9.0	9.0	9.0	8.0	9.0	7.5	7.0	7.0
BitWave	bce_array	9.0	9.0	8.5	8.5	9.5	9.0	8.0	9.0
	control_unit	9.0	8.5	9.0	8.0	9.0	8.0	7.5	8.0
	data_dispatcher	9.0	8.0	9.0	8.0	9.0	7.0	6.0	7.0
	memory_interface	8.0	9.0	9.0	7.5	9.0	7.0	6.0	7.0
	output_formatter	10.0	10.0	9.5	9.5	9.0	9.0	7.5	8.5
	top	8.0	9.0	9.0	7.5	8.5	8.0	5.0	8.0
	zcip_parser	9.5	9.5	10.0	9.0	9.0	8.0	7.0	8.0
FlightVGM	matrix_processing_engine	6.0	6.0	7.0	5.0	7.0	6.0	4.0	5.0
	cpu_scheduler	9.0	8.5	9.0	8.5	9.0	8.5	7.5	8.0
	top	5.0	4.0	7.5	6.0	7.0	6.0	4.0	5.0
	recovery_unit	9.0	8.5	9.0	9.0	10.0	8.5	8.0	8.5
	global_interconnect	7.0	9.0	9.0	7.5	10.0	9.0	9.0	9.0
	sparsification_unit	9.0	8.5	8.0	8.5	9.0	7.0	5.0	7.0
	dsp_e	8.0	8.5	8.0	7.5	9.0	6.0	6.0	7.0
	mmu_controller	9.0	8.5	9.0	8.5	9.5	8.0	7.5	7.5
	compute_core	6.0	4.0	6.0	7.0	8.0	7.0	5.0	6.0
	special_function_unit	8.0	8.0	7.5	7.5	8.5	7.0	5.0	6.0
MASL-AFU	bsearch_unit	9.0	8.0	9.0	9.0	8.0	8.0	7.0	8.0
	control_unit	9.0	9.0	9.0	8.0	8.5	7.5	7.0	7.5
	lut_unit	9.0	9.0	9.0	8.5	9.0	8.5	8.0	9.0
	mac_unit	9.0	9.0	8.0	8.5	9.5	8.5	8.0	7.5
	masl_afu_top	9.0	10.0	9.0	9.0	9.5	8.0	6.0	8.0
	masl_afu_unit	9.0	9.0	9.0	8.0	9.5	7.5	6.0	7.5
	memory_controller	9.0	8.5	9.0	8.0	9.0	8.0	6.0	8.0
	scalability_node	9.0	8.5	9.0	9.0	9.5	9.0	8.0	9.0
	shared_buffer	9.0	9.0	8.5	8.5	9.5	9.0	7.5	8.0
	top	9.0	9.5	9.0	9.0	10.0	6.5	5.0	6.0
ST-Pruning	accumulator	9.0	9.0	9.0	9.0	10.0	9.0	8.0	8.0
	config_interface	8.0	9.0	9.0	8.5	9.0	7.5	5.0	7.5
	control_unit	9.0	9.0	9.0	8.5	9.5	8.0	5.0	7.5
	fsde_encode	9.0	7.5	9.0	8.0	9.0	9.0	7.0	8.0
	gpe_array	9.0	8.5	8.0	5.5	6.0	4.0	4.0	4.0
	memory_controller	9.0	9.0	9.0	8.5	9.0	7.5	7.5	7.5
	top	8.0	9.0	8.5	7.5	9.0	7.0	6.0	7.5

Table 13: Machine Scoring Results Using o3-mini as the LLM in the ArchCraft. II

Paper Name	Design Name	Paper-level			Code-level				
		Fidelity	Structural	Synth.	RTL		TB		
					Sou.	Mai.	TC	AQ	SQ
INSPIRE	accumulation_unit	10.00	7.33	8.50	9.17	9.00	8.33	5.67	8.33
	activation_encoder	9.17	7.67	8.33	9.00	9.17	9.33	8.67	7.50
	control_unit	5.50	8.33	5.67	5.00	5.50	4.33	3.33	5.17
	ip_pe	9.17	9.67	9.00	9.33	9.17	10.00	9.33	9.67
	ip_pe_array	9.00	10.00	9.50	9.67	9.00	8.67	7.33	8.50
	memory_interface	7.67	8.50	9.17	8.33	8.67	7.50	6.17	7.33
	top	7.33	7.50	7.67	7.17	7.33	5.50	4.67	4.50
	weight_buffer	5.17	5.33	7.50	5.67	6.17	4.67	5.50	4.17
SPARK	control_unit	9.00	9.50	9.17	9.33	10.00	8.67	9.33	8.50
	im2col_pack_engine	10.00	9.33	10.00	9.17	10.00	7.33	5.17	8.67
	interface	8.33	7.50	8.17	8.00	8.33	6.67	6.50	7.17
	memory_controller	9.17	9.00	9.50	9.17	9.67	7.50	5.67	7.33
	pe_array	8.67	9.17	9.00	9.00	9.50	7.17	5.50	7.50
	spark_decoder	10.00	10.00	10.00	10.00	10.00	9.33	5.67	7.67
	spark_encoder	10.00	10.00	10.00	10.00	10.00	8.50	7.17	7.50
	top	8.50	8.17	8.33	8.67	8.00	7.67	7.33	7.67
MASL-AFU	bsearch_unit	9.33	8.50	9.17	9.00	8.67	8.33	7.67	7.50
	control_unit	9.17	9.00	9.50	9.67	9.33	7.50	7.17	7.67
	lut_unit	7.50	7.67	8.33	8.50	8.00	8.17	8.67	8.50
	mac_unit	9.00	9.50	8.17	8.33	10.00	8.67	8.50	8.33
	masl_afu_top	9.67	8.33	8.67	8.50	9.17	8.00	6.33	7.17
	masl_afu_unit	9.50	9.17	8.50	8.00	10.00	7.67	7.50	7.33
	memory_controller	8.67	8.50	9.33	9.50	9.00	8.33	7.17	7.67
	scalability_node	9.00	7.67	10.00	10.00	10.00	9.50	8.33	9.17
	shared_buffer	9.17	8.33	9.00	9.00	9.50	7.17	5.50	8.67
	top	8.50	9.00	8.67	8.17	9.00	7.50	5.67	7.17

Table 14: Overall Human Scoring Results Using o3-mini as the LLM in ArchCraft.

LLM Model	CD Num.	Mean	Variance
Gemini 2.0 Flash	92	7.58	8.52
Qwen3-Coder-480B	87	6.44	8.47
GPT-4o	70	4.95	9.26
o3-mini	75	2.07	7.64

Table 15: Rectifying phase iteration counts. CD Num. is the abbreviation of Circuit Designs Number

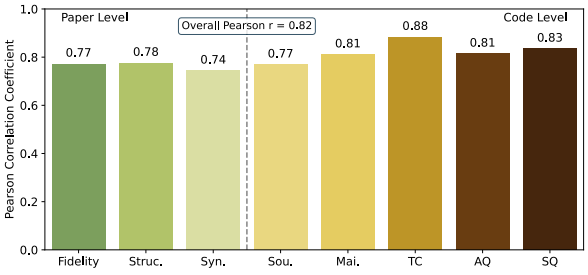

Figure 11: Pearson correlation between human and machine-based scores for ArchCraft

```
Global Operating Voltage = 1.98
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1uW


  Cell Internal Power  =   4.1925 mW   (80%)
  Net Switching Power  =   1.0696 mW   (20%)
                         ---------
Total Dynamic Power    =   5.2621 mW  (100%)

Cell Leakage Power     =  69.6676 nW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000            0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000            0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000            0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000            0.0000             0.0000  (   0.00%)
register           2.7187           0.1614        1.9938e-02             2.8801  (  54.73%)
sequential         0.0000           0.0000            0.0000             0.0000  (   0.00%)
combinational      1.4738           0.9083        4.9729e-02             2.3821  (  45.27%)
--------------------------------------------------------------------------------------------------
Total              4.1925 mW        1.0696 mW     6.9668e-02 uW          5.2622 mW
```

Figure 12: Reported power of the pe_array.

```
u_pe0/mult_54/U7/S (ADDHXL)                              0.22       1.11 f
u_pe0/mult_54/product[1] (pe_0_DW_mult_uns_0_DW_mult_uns_3)
                                                         0.00       1.11 f
u_pe0/U29/CO (ADDFX2)                                    0.24       1.35 f
u_pe0/U28/CO (ADDFX2)                                    0.15       1.50 f
u_pe0/U23/CO (ADDFX2)                                    0.15       1.66 f
u_pe0/U21/CO (ADDFX2)                                    0.15       1.81 f
u_pe0/U18/CO (ADDFX2)                                    0.15       1.96 f
u_pe0/U15/CO (ADDFX2)                                    0.15       2.11 f
u_pe0/U14/CO (ADDFX2)                                    0.17       2.28 f
u_pe0/U129/Y (XOR2X1)                                    0.15       2.43 r
u_pe0/U22/CO (ADDFX2)                                    0.25       2.68 r
u_pe0/U20/CO (ADDFX2)                                    0.14       2.82 r
u_pe0/U19/CO (ADDFX2)                                    0.14       2.96 r
u_pe0/U12/CO (ADDFX2)                                    0.15       3.11 r
u_pe0/U110/Y (AND2X1)                                    0.13       3.24 r
u_pe0/U104/Y (AND2X1)                                    0.13       3.37 r
u_pe0/U98/Y (AND2X1)                                     0.11       3.49 r
u_pe0/U95/Y (XOR2X1)                                     0.13       3.61 f
u_pe0/U13/Y (AOI22X1)                                    0.15       3.76 r
u_pe0/U16/Y (OAI2BB1X1)                                  0.05       3.81 f
u_pe0/accum_reg_15_/D (DFFRHQX1)                         0.00       3.81 f
data arrival time                                                   3.81

clock clk (rise edge)                                    5.00       5.00
clock network delay (ideal)                              0.01       5.01
clock uncertainty                                       -0.01       5.00
u_pe0/accum_reg_15_/CK (DFFRHQX1)                        0.00       5.00 r
library setup time                                      -0.15       4.85
data required time                                                  4.85
--------------------------------------------------------------------------
data required time                                                  4.85
data arrival time                                                  -3.81
--------------------------------------------------------------------------
slack (MET)                                                         1.04
```

Figure 13: Reported performance of the pe_array.

```
Current design is 'pe_array'.
design_vision> report_area

****************************************
Report : area
Design : pe_array
Version: Q-2019.12-SP5-1
Date   : Thu Jul 31 11:05:32 2025
****************************************

Library(s) Used:

    sc9tap_cm018mg_base_rvt_ff_typical_min_1p98
0-00eac0/arm/tsmc/cm018mg/sc9tap_base_rvt/r0p0/

Number of ports:                          350
Number of nets:                          1409
Number of cells:                          916
Number of combinational cells:            737
Number of sequential cells:               168
Number of macros/black boxes:               0
Number of buf/inv:                         76
Number of references:                       9

Combinational area:                20543.846578
Buf/Inv area:                        532.224007
Noncombinational area:             11802.067291
Macro/Black Box area:                  0.000000
Net Interconnect area:                 0.000000

Total cell area:                   32345.913869
Total area:                        32345.913869
```

Figure 14: Reported area of the pe_array.

```
Global Operating Voltage = 1.98
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1uW


  Cell Internal Power  = 273.2457 uW   (84%)
  Net Switching Power  =  51.1611 uW   (16%)
                         ---------
Total Dynamic Power    = 324.4068 uW  (100%)

Cell Leakage Power     =   2.6998 nW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000            0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000            0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000            0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000            0.0000             0.0000  (   0.00%)
register           0.2563       1.1524e-02        1.9077e-03             0.2679  (  82.57%)
sequential         0.0000           0.0000            0.0000             0.0000  (   0.00%)
combinational  1.6905e-02       3.9637e-02        7.9204e-04         5.6543e-02  (  17.43%)
--------------------------------------------------------------------------------------------------
Total              0.2732 mW     5.1161e-02 mW     2.6998e-03 uW          0.3244 mW
```

Figure 15: Reported power of the spark_encoder.

```
          (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  spark_encoder      Small                 sc9tap_cm018mg_base

  Point                                    Incr       Path
  ----------------------------------------------------------
  clock clk (rise edge)                    0.00       0.00
  clock network delay (ideal)              0.01       0.01
  input external delay                     0.70       0.71 f
  valid_in (in)                            0.00       0.71 f
  U32/Y (NAND2X1)                          0.23       0.94 r
  U47/Y (INVX1)                            0.24       1.18 f
  U26/Y (AOI22X1)                          0.15       1.34 r
  U25/Y (INVX1)                            0.04       1.38 f
  raw_reg_reg_5_/D (DFFRHQX1)              0.00       1.38 f
  data arrival time                                   1.38

  clock clk (rise edge)                    5.00       5.00
  clock network delay (ideal)              0.01       5.01
  clock uncertainty                       -0.01       5.00
  raw_reg_reg_5_/CK (DFFRHQX1)             0.00       5.00 r
  library setup time                      -0.15       4.85
  data required time                                  4.85
  ----------------------------------------------------------
  data required time                                  4.85
  data arrival time                                  -1.38
  ----------------------------------------------------------
  slack (MET)                                         3.47
```

Figure 16: Reported performance of the spark_encoder.

```
****************************************
Report : area
Design : spark_encoder
Version: Q-2019.12-SP5-1
Date   : Thu Jul 31 10:38:12 2025
****************************************

Library(s) Used:

    sc9tap_cm018mg_base_rvt_ff_typical_min_1p98v_0c
FB-00000-r0p0-00eac0/arm/tsmc/cm018mg/sc9tap_base_r
_1p98v_0c.db)

Number of ports:                           17
Number of nets:                            71
Number of cells:                           60
Number of combinational cells:             44
Number of sequential cells:                16
Number of macros/black boxes:               0
Number of buf/inv:                         17
Number of references:                      15

Combinational area:                558.835208
Buf/Inv area:                      113.097601
Noncombinational area:            1117.670410
Macro/Black Box area:                0.000000
Net Interconnect area:               0.000000

Total cell area:                  1676.505619
Total area:                       1676.505619
1
design_vision> report_clock
```

Figure 17: Reported area of the spark_encoder.

```
****************************************
Report : area
Design : spark_decoder
Version: Q-2019.12-SP5-1
Date   : Thu Jul 31 10:59:38 2025
****************************************

Library(s) Used:

    sc9tap_cm018mg_base_rvt_ff_typical_min_1p98
0-00eac0/arm/tsmc/cm018mg/sc9tap_base_rvt/r0p0/

Number of ports:                           12
Number of nets:                            25
Number of cells:                           18
Number of combinational cells:             12
Number of sequential cells:                 6
Number of macros/black boxes:               0
Number of buf/inv:                          6
Number of references:                       6

Combinational area:                149.688001
Buf/Inv area:                       39.916800
Noncombinational area:             419.126404
Macro/Black Box area:                0.000000
Net Interconnect area:               0.000000

Total cell area:                   568.814404
Total area:                        568.814404
```

Figure 18: Reported power of the spark_decoder.

```
          (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  spark_decoder      Small                 sc9tap_cm018mg_base_

  Point                                    Incr       Path
  ----------------------------------------------------------
  clock clk (rise edge)                    0.00       0.00
  clock network delay (ideal)              0.01       0.01
  input external delay                     0.70       0.71 r
  in_valid (in)                            0.00       0.71 r
  U13/Y (INVX1)                            0.09       0.80 f
  U12/Y (AOI32X1)                          0.13       0.93 r
  U11/Y (INVX1)                            0.04       0.98 f
  post_flag_reg/D (DFFRHQX1)               0.00       0.98 f
  data arrival time                                   0.98

  clock clk (rise edge)                    5.00       5.00
  clock network delay (ideal)              0.01       5.01
  clock uncertainty                       -0.01       5.00
  post_flag_reg/CK (DFFRHQX1)              0.00       5.00 r
  library setup time                      -0.15       4.85
  data required time                                  4.85
  ----------------------------------------------------------
  data required time                                  4.85
  data arrival time                                  -0.98
  ----------------------------------------------------------
  slack (MET)                                         3.88
```

Figure 19: Reported performance of the spark_decoder.

```
spark_decoder          Small             sc9tap_cm018mg_base_rvt_ff_typical_min_1p98v_0c


Global Operating Voltage = 1.98
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1uW


  Cell Internal Power  = 100.0763 uW   (92%)
  Net Switching Power  =   8.9223 uW    (8%)
                         ---------
Total Dynamic Power    = 108.9985 uW  (100%)

Cell Leakage Power     = 938.0424 pW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000            0.0000            0.0000            0.0000  (   0.00%)
memory             0.0000            0.0000            0.0000            0.0000  (   0.00%)
black_box          0.0000            0.0000            0.0000            0.0000  (   0.00%)
clock_network      0.0000            0.0000            0.0000            0.0000  (   0.00%)
register       9.5663e-02        2.8983e-03        7.2920e-04        9.8562e-02  (  90.42%)
sequential         0.0000            0.0000            0.0000            0.0000  (   0.00%)
combinational  4.4135e-03        6.0239e-03        2.0884e-04        1.0438e-02  (   9.58%)
--------------------------------------------------------------------------------------------------
Total              0.1001 mW     8.9223e-03 mW     9.3804e-04 uW         0.1090 mW
```

Figure 20: Reported area of the spark_decoder.