SymDiReC: A Neuro-Symbolic Divide-Retrieve-Conquer Framework for Enhanced RTL Synthesis and Summarization

Prashanth Vijayaraghavan, Apoorva Nitsure, Luyao Shi, Charles Mackin,
Ashutosh Jadhav, David Beymer, Ehsan Degan, Vandana Mukherjee

IBM Research, San Jose, CA, USA
`{prashanthv,apoorva.nitsure,luyao.shi,charles.mackin}@ibm.com`
`{ashutosh,beymer,edehgha,vandana}@us.ibm.com`

Abstract

Register-Transfer Level (RTL) synthesis and summarization are central to hardware design automation but remain challenging for Large Language Models (LLMs) due to rigid HDL syntax, limited supervision, and weak alignment with natural language. Existing prompting and retrieval-augmented generation (RAG) methods have not incorporated symbolic planning, limiting their structural precision. We introduce **SymDiReC**[1], a neuro-symbolic framework that decomposes RTL tasks into symbolic subgoals, retrieves relevant code via a fine-tuned retriever, and assembles verified outputs through LLM reasoning. Supporting both Verilog and VHDL without LLM fine-tuning, SymDiReC achieves ∼20% higher Pass@1 rates for synthesis and 15–20% Rouge-L improvements for summarization over prompting and RAG baselines, demonstrating the benefits of symbolic guidance in RTL tasks.

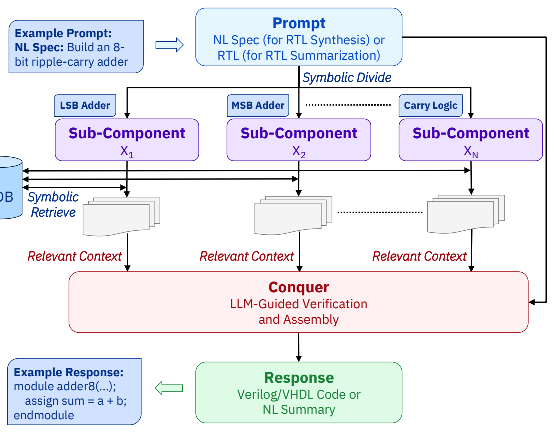


Figure 1: Overview of our SymDiReC framework for RTL synthesis and summarization.

1 Introduction

Register-Transfer Level (RTL) synthesis and summarization are central tasks in Electronic Design Automation (EDA). RTL synthesis translates high-level natural language specifications into synthesizable hardware modules, while RTL summarization produces concise natural language explanations of existing hardware code. For example, given the natural language (NL) specification "build an 8-bit ripple-carry adder" a system must generate a correct Verilog/VHDL module that composes full-adders and propagates carries; conversely, given such a module, it should explain its functional structure (e.g., LSB/MSB adders and carry logic). While this example is relatively simple, real-world RTL designs often involve multi-stage pipelines, control logic, and hierarchical modules with complex timing and data dependencies. These characteristics make monolithic generation brittle and error-prone, motivating the need for modular decomposition, targeted retrieval of reusable RTL components, and explicit integration and verification. As illustrated in Figure 1, both synthesis and summarization require preserving strict HDL syntax, modular structure, and precise functional semantics, distinguishing them from general-purpose code generation and summarization.

While Large Language Models (LLMs) have shown increasing promise in code generation, their performance on Hardware Description Languages (HDLs) like Verilog and VHDL remains limited due to rigid syntax, sparse annotated data, and semantic divergence from natural language (Vijayaraghavan et al., 2024b; Zhao et al., 2025). Recent prompting strategies such as Chain-of-Thought (CoT) (Wei et al., 2022), CoDes (Vijayaraghavan et al., 2024a), and ReAct (Yao et al., 2023) enhance step-by-step reasoning yet struggle with RTL-specific tasks due to domain mismatch and absence of structural priors. Retrieval-Augmented Generation (RAG) methods (Lewis et al., 2020; Petroni et al., 2021; Ho et al., 2025; Ping et al., 2025; Yao et al., 2024) reduce hallucinations and

[1]Short for Neuro-**Sym**bolic **Di**vide–**Re**trieve–**C**onquer Strategy

improve factuality by grounding LLMs with external context. However, they often rely on expensive instruction-tuning, target only Verilog, and address either synthesis or summarization in isolation. Moreover, neither prompting nor RAG pipelines typically incorporate symbolic scaffolds, which can explicitly capture hardware intent.

Symbolic planning and neuro-symbolic approaches have demonstrated strong benefits in enhancing interpretability and structure-awareness in generation tasks (Zhou et al., 2022; Pan et al.). Models like Logic-LM (Pan et al.) and Code-as-Symbolic-Planner (Chen et al., 2025) leverage symbolic scaffolding to guide generation, but these techniques have not been extended to RTL workflows. We introduce **SymDiRec**, a neuro-symbolic *Divide–Retrieve–Conquer* framework tailored for RTL synthesis and summarization across both Verilog and VHDL. SymDiRec consists of three symbolic reasoning-driven stages: (a) **Divide** via symbolic decomposition, where an LLM breaks a high-level RTL task into modular subcomponents with natural language and symbolic representations (e.g., Boolean or dataflow logic); (b) **Retrieve** using a domain-adapted symbolic retriever fine-tuned on the RTL-IR dataset, which incorporates both symbolic and textual cues to fetch semantically relevant RTL fragments; and (c) **Conquer** via LLM-guided verification and assembly, where retrieved candidates are aligned with symbolic intent and assembled into a final code block or summary. By integrating symbolic logic into every stage, SymDiRec improves retrieval precision, output consistency, and verification; all without requiring full LLM fine-tuning. Empirical results demonstrate that SymDiRec achieves roughly 20% higher Pass@1 accuracy in synthesis and 15–20% improvement in Rouge-L for summarization over strong RAG and prompting baselines. These results highlight the value of symbolic reasoning in bridging the gap between natural language, logic, and RTL semantics. Our key contributions are as follows:

SymDiRec Framework: We propose a novel neuro-symbolic Divide–Retrieve–Conquer pipeline for RTL tasks, integrating symbolic decomposition, retrieval, and verification in a unified architecture.

Symbolic Guidance for Retrieval and Verification: We show that symbolic logic enables more precise retrieval and structurally consistent outputs, outperforming natural language methods.

Cross-Language RTL Evaluation: We evaluate SymDiRec on both Verilog and VHDL benchmarks for synthesis and summarization, demonstrating generalizability across RTL domains.

Lightweight and Modular Reasoning: Our approach avoids full LLM fine-tuning by adapting only the retriever, maintaining competitive performance relative to several strong baselines.

2 Related Work

Transformer-based code models such as Megatron-LM (Shoeybi et al., 2019), StarCoder (Li et al., 2023a), CodeGen (Nijkamp et al., 2022), CodeLlama (Roziere et al., 2023), and Granite (Mishra et al., 2024) underpin much of the progress in multi-language generation and code reasoning. Prompting techniques, including Chain-of-Thought (CoT) (Wei et al., 2022), CoDes (Vijayaraghavan et al., 2024a), and CoT with self-verification (Ping et al., 2025), deliver structured reasoning for hardware-related tasks. ReAct prompting (Yao et al., 2023) adds iterative refine and act cycles to improve correctness further. Retrieval-Augmented Generation (RAG) grounds LLM outputs with external context (Lewis et al., 2020; Petroni et al., 2021). Recent extensions such as self-reflective and corrective RAG (Asai et al., 2023; Yan et al., 2024) and document-level prompting (Zhou et al., 2022) reduce hallucinations and streamline domain adaptation. Frameworks such as REDCODER (Parvez et al., 2021) have applied RAG for code summarization and generation in general-purpose languages, illustrating benefits of dual retrieval and generation.

In hardware design, modular RAG and reasoning strategies are emerging. VerilogCoder uses a task and circuit relation graph and AST-based debugging to exceed 90% pass rates on Verilog benchmarks (Ho et al., 2025). HDLCoRe and HDLdebugger add hardware-aware prompt decomposition plus evidence filtering (Ping et al., 2025; Yao et al., 2024). ComplexVCoder implements a two-stage approach with intermediate representations and domain-specific RAG (Zuo et al., 2025). Multi-level summarization models like CodeV (Zhao et al., 2025) improve Verilog generation via instruction tuning. All these methods rely on expensive model tuning or fine-tuning and typically focus only on Verilog or on one task, either synthesis or summarization.

Efforts specifically targeting VHDL are limited. The VHDL-Eval benchmark (Vijayaraghavan et al., 2024b) and CoDes for VHDL (Vijayaraghavan et al., 2024a) highlight consistent weaknesses of LLMs on VHDL, underscoring the need for methods that can handle both synthesis and summarization. Our approach, SYMDIREC, uniquely addresses these gaps. It incorporates symbolic logic as a structured intermediate representation, enhancing both decomposition and retrieval. Unlike graph-only RAG frameworks, symbolic logic captures functional intent, allowing more precise retrieval and verification. SYMDIREC is among the few systems evaluated on both Verilog and VHDL, and the first to jointly tackle synthesis and summarization across both languages. The neuro-symbolic combination composed of symbolic decomposition, retriever tuning for RTL semantics, and LLM-guided verification surpasses prior art in performance while maintaining interpretability and modular reasoning.

3 Neuro-Symbolic Divide-Retrieve-Conquer (SYMDIREC) Framework

3.1 Overview

We introduce **SYMDIREC**, a unified neuro-symbolic framework designed for two complementary tasks in register-transfer level (RTL) design: (i) *synthesis*, where natural language (NL) problem statements or specifications are translated into their corresponding RTL code; and (ii) *summarization*, where RTL modules are converted into interpretable NL explanations augmented with symbolic logic. By integrating symbolic reasoning into our pipeline, SYMDIREC ensures semantically meaningful decomposition, retrieval, and verification. The symbolic representations act as an intermediate scaffold that enhances both retrieval precision and output correctness, especially in the context of more challenging RTL semantics. SYMDIREC follows a three-stage architecture (refer Figure 2) shared across both synthesis and summarization:

Divide (Symbolic Decomposition): The input, either a natural language specification or RTL code, is decomposed into smaller, semantically meaningful sub-components. Each sub-component is annotated with a brief textual description and a symbolic logic representation that captures its functional behavior.

Retrieve (Symbolic Querying): For each sub-component, the corresponding symbolic logic is used to retrieve relevant RTL snippets or symbolic summaries from a structured knowledge base. The retriever is trained to understand the alignment between symbolic logic and RTL structures.

Conquer (LLM Verification and Assembly): An LLM evaluates the retrieved candidates based on their alignment with the symbolic logic and description, selects the top candidate for each sub-component, and assembles them into a coherent RTL implementation or summary.

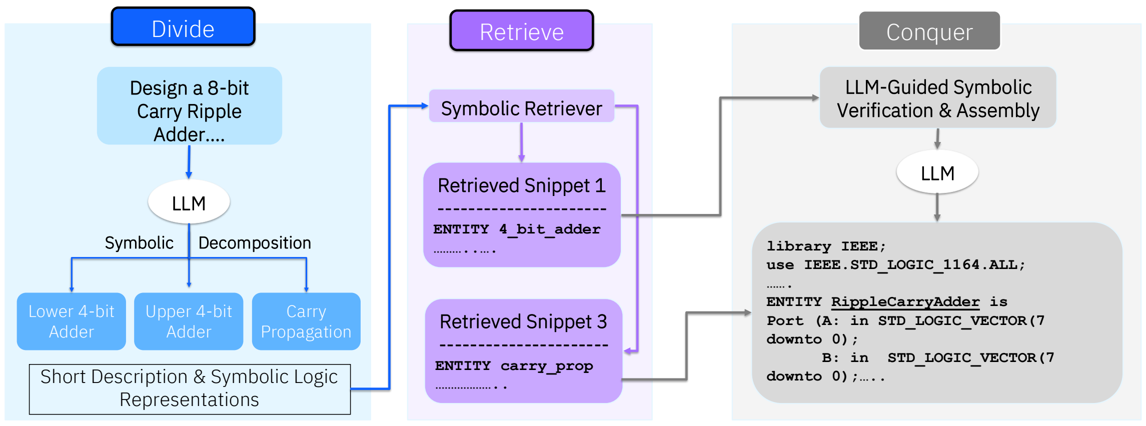
3.2 Divide: Symbolic Decomposition

The **Divide** stage decomposes the input into a set of interpretable sub-components, each represented by a symbolic logic expression and a corresponding textual or structural unit. This decomposition step provides structure and granularity to the task, enabling downstream retrieval and verification at a finer granularity. Formally, for an input X, the decomposition function is defined as:

$$f_{\text{DIV}}(X) = \{(x_1, \phi_1), \ldots, (x_N, \phi_N)\}$$

where x_i is the i^{th} sub-unit and ϕ_i is its associated symbolic logic representation. The number of sub-units N is dynamic and may vary with the complexity of the input; it is governed by prompting strategies or syntax-driven partitioning mechanisms. For synthesis tasks, the input X is a natural language specification. We use a pretrained LLM to decompose this specification into a sequence of short NL descriptions $\{x_i\}$, each denoting a distinct functional sub-task (e.g., counter, comparator, multiplexer). For each x_i, we prompt the same LLM to produce a corresponding symbolic logic expression ϕ_i that captures the intended behavior in a logic-based form (e.g., temporal, Boolean, or dataflow expressions). Optionally, we provide some in-context examples to encourage structured and consistent output formats grounded in symbolic hardware semantics. For summarization tasks, the input X is RTL code. We first parse the code into its abstract syntax tree (AST) and segment it into functional blocks $\{x_i\}$, such as 'always' blocks, modules, or combinational logic segments. For each block x_i, we prompt an LLM to abstract its functional intent into a concise symbolic representation ϕ_i, typically capturing control logic, state transitions, or data transformations. We encourage structural consistency in these symbolic forms using prompt-based templates grounded in RTL semantics.

LLM Guidance and Symbolic Abstraction: In both directions, symbolic abstraction relies on prompting a pretrained LLM to reason about hardware functionality and generate interpretable symbolic expressions. While LLMs may not always



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
…….
ENTITY RippleCarryAdder is
Port (A: in STD_LOGIC_VECTOR(7
downto 0);
      B: in  STD_LOGIC_VECTOR(7
downto 0);…..
```

Figure 2: Illustration of our SYMDIREC framework for 8-bit carry ripple adder.

produce complete or formal logic, their output often provides high-quality approximations that capture key semantic elements of the underlying subcomponent. These symbolic sketches serve as anchors for downstream retrieval and alignment. This decomposition process ensures that each sub-unit is semantically meaningful and structurally aligned with RTL design principles, enabling targeted retrieval and robust composition in later stages.

3.3 Retrieve: Symbolic Retriever

The **Retrieve** stage enriches each sub-component (x_i, ϕ_i) with relevant RTL code snippets or symbolic/NL summaries. To leverage both the textual description x_i and its formal symbolic logic ϕ_i, we design a joint embedding retriever that matches paired queries (x_i, ϕ_i) against a structured knowledge base $\mathcal{K}$. We denote them as:

$$f_{\text{RET}}(x_i, \phi_i) = R_i \;=\; \text{TopK}_{y \in \mathcal{K}}\, \text{score}(x_i, \phi_i; y).$$

3.3.1 Knowledge Base

We construct a repository of S indexed entries: $\mathcal{K} = \{(y_j, d_j, \phi_j)\}_{j=1}^{S}$, where each entry contains: (a) y_j: an RTL code snippet (VHDL/Verilog) or NL summary, (b) d_j: a short NL explanation of y_j, and (c) ϕ_j: symbolic logic representation.

3.3.2 Joint Retriever Architecture

We implement a dual-encoder with three transformer encoders: $e_x : \mathcal{X} \to \mathbb{R}^D$, $e_\phi : \Phi \to \mathbb{R}^D$, $e_y : \mathcal{Y} \to \mathbb{R}^D$, where e_x embeds NL fragments x, e_ϕ embeds symbolic logic ϕ, and e_y embeds candidate entries y. To form a joint query representation, we concatenate and project:

$$q_i = W_q \big[e_x(x_i) \,\|\, e_\phi(\phi_i)\big] \;\in\; \mathbb{R}^D,$$

with learned projection matrix $W_q \in \mathbb{R}^{D \times 2D}$. Retrieval then ranks each entry y_j by cosine similarity:

$$\text{score}(x_i, \phi_i;\, y_j) = \cos\big(q_i,\; e_y(y_j)\big).$$

3.3.3 Training Objective

We fine-tune all three encoders on our RTL-IR dataset of aligned triplets $\{(x_p, \phi_p, y_p)\}_{p=1}^{S}$. In each batch of size B, the positive example (x_p, ϕ_p, y_p) is contrasted against in-batch negatives $\{y_q\}_{q \neq p}$. We minimize the multiple-negatives ranking loss and explore both dense (continuous embeddings) and sparse (term-weighted) variants for e_x and e_ϕ; implementation and hyper-parameter details appear in the Appendix B, along with dataset statistics and ablations.

3.3.4 Inference: Retrieving Sub-components

At inference time, each decomposed query (x_i, ϕ_i) is encoded as q_i and we compute the cosine similarity score between the computed query and candidate $y_j \in \mathcal{K}$. The retriever returns the top-k candidates as:

$$R_i = f_{\text{RET}}(x_i, \phi_i) = \text{TopK}_{j \in [1,S]}\, \text{score}(x_i, \phi_i; y_j),$$

yielding $R_i = \{r_{i,m}\}_{m=1}^{k}$. Each $r_{i,j}$ is either an RTL snippet (for synthesis) or a NL summary (for summarization). Table 4 presents a qualitative example of an 8-bit ripple-carry adder, including symbolic decompositions into Boolean/logical expressions for each submodule and the retrieved Verilog and VHDL code snippets tied to these submodules.

3.4 Conquer with LLM-Guided Verification and Assembly

The **Conquer** stage integrates retrieved candidates into a finalized output. Given the original input X

Method	LLM	Pass@1		ROUGE-L	
		Verilog	VHDL	Verilog	VHDL
Vanilla Prompting	GPT-4o	0.543	0.285	43.1	39.3
	Llama-3	0.385	0.226	40.2	34.1
CoDes	GPT-4o	0.602	0.348	46.9	43.2
	Llama-3	0.435	0.274	43.5	39.5
ReAct Prompting	GPT-4o	0.616	0.353	46.1	43.0
	Llama-3	0.437	0.291	42.9	38.8
VRAG-CodeBERT	GPT-4o	0.688	0.487	53.2	50.3
	Llama-3	0.527	0.396	47.4	44.5
VRAG-FT	GPT-4o	0.719	0.531	57.0	52.8
	Llama-3	0.569	0.439	50.5	48.1
RTLCoder (open-source)	Mistral	0.625*	-	-	-
	GPT-4o/Llama-3	-	-	-	-
CodeV (instruction-tuned)	CodeQwen	0.532*	-	-	-
	GPT-4o/Llama-3	-	-	-	-
SYMDIREC (ours)	GPT-4o	$\mathbf{0.805}_{\pm 0.020}$	$\mathbf{0.634}_{\pm 0.022}$	$\mathbf{62.5}_{\pm 0.015}$	$\mathbf{56.6}_{\pm 0.018}$
	Llama-3	$\mathbf{0.652}_{\pm 0.022}$	$\mathbf{0.545}_{\pm 0.020}$	$\mathbf{56.1}_{\pm 0.018}$	$\mathbf{50.8}_{\pm 0.015}$
SYMDIREC-GT (oracle)	GPT-4o	**0.902**	**0.842**	**70.2**	**64.7**
	Llama-3	**0.807**	**0.721**	**63.3**	**57.9**

Table 1: RTL synthesis (Pass@1) and summarization (ROUGE-L) performance across methods and LLMs. For our SYMDIREC, mean ± standard deviation over five independent runs is reported. Results are statistically significant vs. the strongest baseline (paired t-test, $p < 0.01$). * indicates results reported in the original papers.

and the retrieved sets $\{R_i\}_{i=1}^N$, the final artifact $\hat{Y}$ is produced by: $\hat{Y} = f_{\text{CONQ}}\big(X, \{R_i\}_{i=1}^N\big)$, where $\hat{Y}$ is either the synthesized RTL module or the summarized NL description. Given $R_i = \{r_{i,m}\}_{m=1}^k$ are the top-k candidates for each sub-component i, from Section 3.3, we prompt the generator LLM to assign an alignment score by conditioning on the sub-component x_i and its associated symbolic logic representation $\phi_i \quad \forall m \in \{1, k\}$ as: $\hat{\alpha}_{i,m} = \text{verify\_score}\big(r_{i,m}, x_i, \phi_i\big) \in [0, 1]$; These scores reflect both functional correctness and fidelity of retriever results to the symbolic specification. We select the highest-scoring candidate for each sub-component. This yields a verified set of sub-modules or summaries $\{\hat{r}_i\}_{i=1}^N$. The final assembly invokes the LLM conditioned on X and the verified candidates. This step produces a coherent RTL design or comprehensive NL summary, ensuring consistent naming, module connectivity, and logical integrity.

4 Experiments

This section outlines our experimental setup, including RTL benchmarks (in both VHDL and Verilog), a diverse suite of baseline models, and evaluation metrics designed to assess the effectiveness, generalization, and efficiency of the proposed SYMDIREC framework. Refer Appendix A for full implementation and dataset details. Our evaluation is structured around the following research questions: **(RQ1) Effectiveness of SYMDIREC:** How effective is the SYMDIREC framework for RTL synthesis and summarization, compared to existing baseline approaches? **(RQ2) Impact of Symbolic Logic:** To what extent do symbolic logic representations improve retrieval quality and downstream RTL generation or summarization? **(RQ3) Hyperparameter Sensitivity:** How do key hyperparameters affect the performance of SYMDIREC?

4.1 Benchmarks and Baselines

We evaluate SYMDIREC on two standard RTL benchmarks: **Verilog-Eval**, comprising 156 functional Verilog tasks from HDLBits (Liu et al., 2023) with testbenches, and **VHDL-Eval**, containing 202 VHDL tasks translated from Verilog-Eval or public tutorials (Vijayaraghavan et al., 2024b), with analogous verification procedures. Our comparisons include zero-shot prompting, intermediate plan-based CoDes (Vijayaraghavan et al., 2024a), iterative ReAct (Yao et al., 2023), Vanilla RAG (VRAG-CodeBERT), and RAG fine-tuned on the **RTL-IR** dataset (VRAG-FT). We also include recent domain-specialized models RTLCoder (Liu et al., 2024) and CodeV (Zhao et al., 2025). Fi-

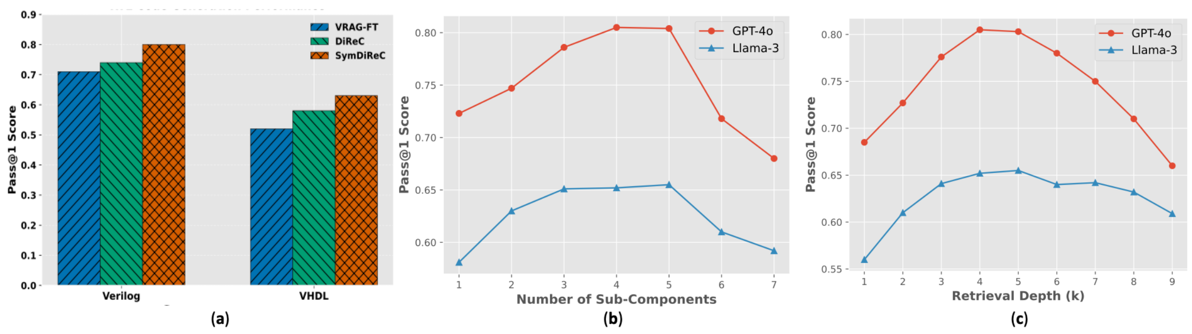


Figure 3: Ablation results: Performance with varying (a) number of sub-components and (b) chunking strategy.

RTL-IR Dataset Statistics	
# Total Size	∼ 50.5k
# Text-to-Code Pairs	∼ 8k
# FEC Pairs	∼ 13.5k
# Code-to-Summary Pairs	∼ 6.5k
# Partial-to-Complete Code Pairs	∼ 22.5k

Table 2: Dataset statistics of RTL-IR used for model finetuning and retrieval enhancement.

nally, we evaluate SymDiReC and its oracle variant using ground-truth snippets (SymDiReC-GT). The RTL-IR dataset is a curated collection of RTL code and annotations used to finetune RAG models, comprising text-to-code, functionally equivalent code (FEC), code-to-summary, and partial-to-complete code pairs. Table 2 summarizes the dataset statistics. Detailed dataset descriptions, benchmarks, and additional examples are provided in Appendix A, C. Baselines use GPT-4o and Llama-3 (70B), evaluated with Pass@1 for synthesis and Rouge-L for summarization. Results report mean ± std over 5 runs with paired t-tests.

4.2 Evaluation Metrics

For RTL synthesis, we use the metric Pass@1, which denotes the proportion of first-attempt RTL designs that successfully pass the self-checking testbenches provided in the Verilog-Eval and VHDL-Eval benchmarks. For code summarization, we employ Rouge-L, measuring the longest common subsequence (LCS) between generated and reference summaries and computing an F-measure to assess fluency and coherence (Lin, 2004).

5 Results & Discussion

5.1 Overall Performance of SymDiReC

Table 1 compares SymDiReC against strong baselines, including prompting-only methods and retrieval-augmented generation (RAG) strategies. Our neuro-symbolic pipeline consistently outperforms all alternatives, demonstrating the effectiveness of symbolic decomposition, a domain-adapted retriever, and LLM-guided verification.

5.2 Effectiveness of SymDiReC (RQ1)

Vanilla Prompting serves as the zero-shot lower bound for each model, showing the weakest performance across both synthesis and summarization tasks. In contrast, SymDiReC-GT represents an approximate upper bound for RAG-based methods, as it always includes the ground-truth among the top-k retrieved candidates. Despite having the correct solution in context, the failure of SymDiReC-GT in specific experimental cases indicates that the LLM can still struggle to filter out distractors within the retrieved candidates or may lack sufficient RTL-specific reasoning capabilities. Chain-of-Descriptions (CoDes) and ReAct prompting yield comparable performance. ReAct demonstrates a modest edge (∼5-8%) relative improvement in Pass@1, likely because its iterative reasoning/action loop allows correction pathways that simplistic descriptive chains lack. Among RAG strategies, VRAG-FT with a finetuned retriever consistently outperforms VRAG-CodeBERT, achieving ∼10%-15% relative improvement in Pass@1 and ∼5%-8% in Rouge-L. This suggests that while generic code retrievers benefit RTL tasks, task-specific tuning further enhances retrieval quality by aligning natural language queries more effectively with RTL semantics. SymDiReC outperforms all baselines by significant margins: up to ∼80% relative improvement over the best prompt-based method and up to ∼20% over RAG-only baselines in Pass@1. For summarization, SymDiReC yields up to ∼35% gains over prompting and ∼10% over RAG in Rouge-L, highlighting the benefits of symbolic planning and RTL-aware retrieval; the remaining ∼10–15%

gap to SymDiReC-GT indicates room to improve retriever precision and LLM alignment.

5.3 Impact of Symbolic Logic (RQ2)

To isolate the benefit of symbolic logic, we compare three pipeline variants: VRAG-FT, which applies a fine-tuned retriever on NL queries; DiReC, which uses the same Divide–Retrieve–Conquer structure but uses NL-only queries for retrieval; and our full SymDiReC, which incorporates symbolic decomposition with modular retrieval and LLM-guided verification. We find that SymDiReC achieves $\sim$10-15% relative improvements in Pass@1 (refer Figure 3(a)) and Rouge-L versus both VRAG-FT and DiReC. These findings demonstrate that symbolic representations serve as a strong scaffolding mechanism, enabling more precise retrieval, reducing noise, and thereby enhancing both synthesis and summarization outcomes.

5.4 Hyperparameter Sensitivity (RQ3)

We perform ablations to explore how key hyperparameters, namely the number of sub-components (N) and retrieval depth (k), affect SymDiReC's performance. Figures 3(b) and (c) summarize the results. Increasing N from 2 to 6 leads to consistent improvements in Pass@1; however, performance starts to decline when N exceeds 6, likely due to excessive fragmentation that results in semantically weaker sub-units. Importantly, the optimal value of N is *task-dependent*. Simpler combinational tasks (e.g., adders, multiplexers) benefit from smaller decompositions ($N \approx$ 3–4), while multi-stage sequential designs (e.g., counters, FSMs) achieve better performance with slightly larger N ($N \approx$ 4–5), reflecting their increased structural complexity. For retrieval depth, performance improves as k increases up to 5, but plateaus and eventually drops when k becomes too large. This decline is likely caused by additional noise introduced by less relevant retrievals. Across benchmarks, a default configuration of $N = 4$ and $k = 5$ provides a strong balance between synthesis accuracy, summarization quality, and computational efficiency, while allowing task-specific tuning when appropriate.

6 Error Analysis

We perform a detailed error analysis to understand the limitations of SymDiReC in RTL synthesis and summarization. Our investigation focuses on three major sources of errors: (a) **Symbolic Decomposition Errors:** Approximately 8-10% of sub-components have incomplete or inconsistent symbolic expressions, particularly for multi-bit comparators or sequential elements. These errors correlate with lower retrieval precision, reducing Pass@1 performance by up to 5-7% for affected designs; (b) **Retrieval Mismatches:** Even with symbolic scaffolds, around 12-15% of retrieved candidates only partially match the intended behavior or contain distractors. This results in a 3-6% drop in Pass@1 accuracy and 2-4 Rouge-L points in summarization; and (c) **LLM Assembly & Verification Failures:** When retrieved candidates are correct, the LLM occasionally fails to integrate them properly (signal misalignment, missing connections, or carry propagation issues), observed in roughly 6-8% of sub-components. These failures contribute to a remaining gap of $\sim$10-15% between SymDiReC and the oracle SymDiReC-GT in synthesis and 6–8 Rouge-L points in summarization. Qualitative inspection shows that most failures occur in hierarchical designs or uncommon module patterns. The SymDiReC-GT results suggest that improved retrieval precision and symbolic reasoning could close a significant portion of the performance gap.

7 Conclusion

We presented SymDiReC, a neuro-symbolic Divide–Retrieve–Conquer framework for RTL synthesis and summarization across Verilog and VHDL. By integrating symbolic decomposition, domain-adapted retrieval, and LLM-guided verification, SymDiReC effectively bridges the gap between formal hardware semantics and large language model generation. Unlike prior approaches that rely heavily on instruction tuning or overlook symbolic intent, our method introduces structured intermediate reasoning to improve both retrieval relevance and generation correctness. Empirical results demonstrate consistent improvements over prompting- and RAG-based baselines, with gains in synthesis accuracy and summarization quality. This work underscores the utility of symbolic planning in program synthesis and opens new directions for interpretable and modular neuro-symbolic systems in hardware design automation and beyond.

Limitations

While SymDiReC demonstrates strong performance in RTL synthesis and summarization, several limitations remain. Symbolic decomposition depends on the LLM's ability to generate well-formed symbolic expressions. Smaller or less capable models may produce incomplete or inconsistent decompositions, diminishing the symbolic scaffolding benefits and yielding performance similar to natural language-only queries. The LLM-guided verification in the CONQUER stage can also fail to align retrieved candidates with the intended logic, particularly when top-k retrievals include distractors or partially matching snippets. Decomposition granularity is sensitive: overly fine segmentation fragments the input, producing weak sub-units, while overly coarse segmentation may reduce retrieval precision. The current pipeline is restricted to single-file RTL designs and does not support hierarchical or multi-file projects, where cross-module dependencies are common. Scaling to such designs may require advanced AST processing, block- and function-level chunking, and multi-level summarization strategies. Even with the correct solution retrieved, the LLM may fail to select it, highlighting challenges in noise filtering, candidate ranking, and reasoning under imperfect retrieval. Addressing these issues: improving symbolic reasoning, retrieval alignment, and hierarchical abstraction, will be essential to extending SYMDIREC to complex real-world hardware design scenarios.

Ethics Statement

We use publicly available datasets (e.g., VerilogEval) and our curated RTL-IR dataset, which is sourced from permissively licensed GitHub repositories (MIT, BSD, Apache-2.0); license metadata is provided in the supplementary material. No private or sensitive data was used; outputs are intended for research and developer-assist purposes only. Potential risks include generating hardware designs that may be incorrect or unsafe if deployed without verification. Our system is intended as a developer-assist tool, and all outputs should be validated using standard testbenches and human review before real-world use.

References

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*.

Yongchao Chen, Yilun Hao, Yang Zhang, and Chuchu Fan. 2025. Code-as-symbolic-planner: Foundation model-based robot planning via symbolic code generation. *arXiv preprint arXiv:2503.01700*.

Elastic. 2024. Machine learning: Natural language processing with elastic. https://www.elastic.co/guide/en/machine-learning/current/ml-nlp-elser.html. Accessed: March 21, 2025.

Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2021. Splade v2: Sparse lexical and expansion model for information retrieval. *arXiv preprint arXiv:2109.10086*.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model pre-training. In *International conference on machine learning*, pages 3929–3938. PMLR.

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.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Raymond Li, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, and 1 others. 2023a. Starcoder: may the source be with you! *Transactions on Machine Learning Research*.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023b. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*.

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. 2024. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, and 1 others. 2024. Granite code models: A family of open foundation models for code intelligence. *CoRR*.

Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. 2023. Mteb: Massive text embedding benchmark. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *EMNLP-Findings*.

F Petroni, A Piktus, A Fan, PSH Lewis, M Yazdani, ND Cao, J Thorne, Y Jernite, V Karpukhin, J Maillard, and 1 others. 2021. Kilt: a benchmark for knowledge intensive language tasks. In *NAACL-HLT*, pages 2523–2544. Association for Computational Linguistics.

Heng Ping, Shixuan Li, Peiyu Zhang, Anzhe Cheng, Shukai Duan, Nikos Kanakaris, Xiongye Xiao, Wei Yang, Shahin Nazarian, Andrei Irimia, and 1 others. 2025. Hdlcore: A training-free framework for mitigating hallucinations in llm-generated hdl. *arXiv preprint arXiv:2503.16528*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992.

Stephen Robertson, Hugo Zaragoza, and 1 others. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

Aivin V. Solatorio. 2024. Gistembed: Guided in-sample selection of training negatives for text embedding fine-tuning. *arXiv preprint arXiv:2402.16829*.

Prashanth Vijayaraghavan, Apoorva Nitsure, Charles Mackin, Luyao Shi, Stefano Ambrogio, Arvind Haran, Viresh Paruthi, Ali Elzein, Dan Coops, David Beymer, and 1 others. 2024a. Chain-of-descriptions: Improving code llms for vhdl code generation and summarization. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, pages 1–10.

Prashanth Vijayaraghavan, Luyao Shi, Stefano Ambrogio, Charles Mackin, Apoorva Nitsure, David Beymer, and Ehsan Degan. 2024b. Vhdl-eval: A framework for evaluating large language models in vhdl code generation. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pages 1–6. IEEE.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Shi-Qi Yan, Jia-Chen Gu, Yun Zhu, and Zhen-Hua Ling. 2024. Corrective retrieval augmented generation. *arXiv preprint arXiv:2401.15884*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. 2024. Hdldebugger: Streamlining hdl debugging with large language models. *ACM Transactions on Design Automation of Electronic Systems*.

Dun Zhang. 2023. stella_en_400m_v5. https://huggingface.co/dunzhang/stella_en_400M_v5. Hugging Face model card, accessed on March 21, 2025.

Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Muxin Song, Yinan Xu, Ziyuan Nan, Mingju Gao, Tianyun Ma, Lei Qi, and 1 others. 2025. Codev: Empowering llms with hdl generation through multi-level summarization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.

Jian Zuo, Junzhe Liu, Xianyong Wang, Yicheng Liu, Navya Goli, Tong Xu, Hao Zhang, Umamaheswara Rao Tida, Zhenge Jia, and Mengying Zhao. 2025. Complexvcoder: An llm-driven framework for systematic generation of complex verilog code. *arXiv preprint arXiv:2504.20653*.

A Datasets

A.1 RTL-IR

A.1.1 Data Collection and Preprocessing

The dataset was curated from publicly available VHDL/Verilog repositories on GitHub. We filtered repositories based on permissive licensing and selected VHDL/Verilog projects with meaningful comments and README descriptions. The preprocessing pipeline involved:

- Extracting comments and README documentation.
- Using in-context learning (ICL) with Granite-13b-Instruct to refine problem statements and code summaries.
- Applying various transformations to generate functionally equivalent code pairs.

Text-to-Code (TC) Pairs To construct TC pairs, we extracted natural language descriptions from comments in VHDL/Verilog files and relevant portions of README documentation. Since raw comments may be unstructured, ICL with Granite-13b-Instruct was used to generate structured problem statements. These statements were validated to ensure clarity and relevance to the corresponding VHDL/Verilog code.

Code-to-Summary (CS) Pairs CS pairs were created by mapping VHDL/Verilog code to textual summaries. Code files with well-commented structures were prioritized, and ICL was employed to convert detailed comments into concise summaries. To assess summary quality, we manually annotated 100 examples, classifying them into:

- *Good* (clear, precise, and informative).
- *Acceptable* (partially informative but useful).
- *Bad* (incomplete or misleading).

Overall, 84% of summaries were classified as *good* or *acceptable*, while 16% were *bad*. The latter were treated as “hard negatives.”

Functionally Equivalent Code (FEC) Pairs FEC pairs were generated by applying different transformation strategies to create variations of functionally identical VHDL/Verilog code. The transformations include:

- **Type-2:** Renaming identifiers while maintaining functional equivalence. We extracted and renamed entity, architecture, process, and port names using an LLM-based renaming strategy. Single-character identifiers were replaced with LLM-suggested alternatives, while complex identifiers underwent abbreviation, permutation, or transformation to maintain readability.
- **Type-3:** Modifying statement order and introducing functionally inert code, ensuring variation while preserving functionality. Reordering declarations and restructuring conditional logic introduced additional diversity.
- **Type-4:** Back-translation between VHDL and Verilog using GHDL and ICARUS Iverilog. This process altered variable names and introduced intermediate signals, capturing functionally equivalent structures while minimizing lexical similarities.

Figure 4 illustrates these transformation types with examples.

Partial-to-Complete Code (PC) Pairs PC pairs were created by extracting partial VHDL/Verilog snippets from larger codebases and pairing them with their complete versions. To ensure lexically diverse representations, Type-2 transformations were applied to a subset of the complete versions. Snippet extraction was limited to code sections containing fewer than 1024 tokens, capturing function declarations and entity definitions along with relevant contextual comments.

A.1.2 Quality Control Measures

To ensure dataset integrity and usefulness, the following quality control measures were applied:

- **Compilation Validation:** All functionally transformed code underwent compilation tests to ensure correctness.
- **Testbench Execution:** Available testbenches from GitHub were executed to verify functional equivalence.
- **Manual Review:** Code summaries were manually reviewed, with low-quality summaries marked as "hard negatives."

These measures enhance dataset reliability, ensuring it serves as a strong benchmark for VHDL code generation and summarization tasks.

B Training & Evaluation of Retriever

Retrieval We fine-tune models from three categories: sparse, dense, and hybrid retrievers, using the RTL-IR dataset. The sparse retriever is BM25 (Robertson et al., 2009). For dense models, we fine-tune top performers from the MTEB Leaderboard (Muennighoff et al., 2023) (GTE-Qwen-1.5b (Li et al., 2023b), Stella-400m(Zhang, 2023), GIST-Large (Solatorio, 2024)), as well as CodeT5+ (Wang et al., 2021) and Sentence Transformer (ST) (Reimers and Gurevych, 2019) for code embeddings. Hybrid methods include SPLADE (Formal et al., 2021) and ELSER(Elastic, 2024). All, except ELSER, are fine-tuned on the RTL-IR training set. We evaluate on the held-out test set.

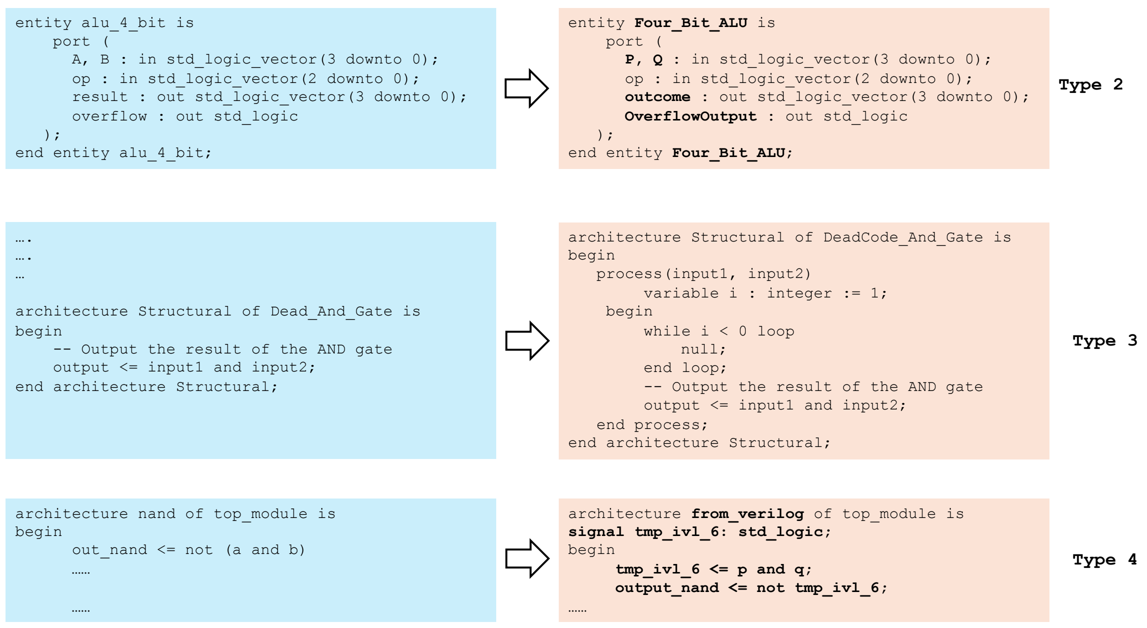
B.1 Evaluation Metric

Retrieval: We employ Normalized Discounted Cumulative Gain (NDCG) to assess ranking quality, rewarding highly relevant results appearing earlier in the list. NDCG@1 measures the relevance of the top-ranked result, while NDCG@10 evaluates ranking effectiveness across the top 10 positions, assigning higher weights to top-ranked items.

B.2 Performance of Retrievers

Table 3 presents the performance of various retrieval methods on the RTL-IR test set using NDCG@1 and NDCG@10 metrics. The results indicate that dense retrieval methods consistently outperform hybrid and sparse approaches, as RTL-IR requires identifying semantically relevant matches beyond surface-level lexical overlaps. Among the evaluated models, CodeT5+ achieves the highest performance, with an NDCG@1 of 0.657 and an NDCG@10 of 0.872, demonstrating its strong ability to retrieve relevant VHDL/Verilog code snippets. This performance advantage can be attributed to CodeT5+'s pre-training on VHDL/Verilog code before fine-tuning on RTL-IR.

Text embedding models such as Stella-400m (NDCG@1 = 0.656) and GTE-Qwen-1.5b (NDCG@1 = 0.644) follow closely, despite being fine-tuned solely on the RTL-IR dataset. Their effectiveness is linked to their large parameter and embedding sizes (e.g., Stella-400m with an embedding size of 8192), enabling better generalization. However, CodeT5+'s code-specific training appears to compensate for its smaller embedding size, leading to superior retrieval performance.



```
entity alu_4_bit is
    port (
      A, B : in std_logic_vector(3 downto 0);
      op : in std_logic_vector(2 downto 0);
      result : out std_logic_vector(3 downto 0);
      overflow : out std_logic
   );
end entity alu_4_bit;
```

```
entity Four_Bit_ALU is
    port (
      P, Q : in std_logic_vector(3 downto 0);
      op : in std_logic_vector(2 downto 0);
      outcome : out std_logic_vector(3 downto 0);
      OverflowOutput : out std_logic
   );
end entity Four_Bit_ALU;
```

```
….
….
…

architecture Structural of Dead_And_Gate is
begin
    -- Output the result of the AND gate
    output <= input1 and input2;
end architecture Structural;
```

```
architecture Structural of DeadCode_And_Gate is
begin
   process(input1, input2)
        variable i : integer := 1;
    begin
        while i < 0 loop
            null;
        end loop;
        -- Output the result of the AND gate
        output <= input1 and input2;
   end process;
end architecture Structural;
```

```
architecture nand of top_module is
begin
      out_nand <= not (a and b)
      ……

      ……
```

```
architecture from_verilog of top_module is
signal tmp_ivl_6: std_logic;
begin
     tmp_ivl_6 <= p and q;
     output_nand <= not tmp_ivl_6;
……
```

Figure 4: VHDL Samples of different transformation strategies applied using the three categories of code clones – Type 2, Type 3 and Type 4.

Methods	NDCG@1	NDCG@10
BM25	0.434	0.570
ELSER	0.485	0.664
SPLADE	0.577	0.688
GTE-Qwen-1.5b	0.644	0.864
Stella-400m	0.656	0.866
GIST-Large	0.616	0.802
CodeT5+	**0.657**	**0.872**
ST	0.556	0.665

Table 3: Evaluation of sparse, hybrid, and dense retrievers on RTL-IR test set.

C Benchmarks and Baselines

C.1 Benchmarks

Verilog-Eval: 156 functional Verilog tasks sourced from HDLBits (Liu et al., 2023), each with a self-verifying testbench. The tasks span a variety of RTL constructs, including combinational logic, sequential modules, counters, comparators, and simple finite-state machines.

VHDL-Eval: 202 VHDL tasks translated from Verilog-Eval problems or drawn from public VHDL tutorials (Vijayaraghavan et al., 2024b), also with testbenches for functional verification. The suite maintains a similar functional diversity to Verilog-Eval.

C.2 Baselines

We compare SymDiRec against several baselines, including recent domain-specialized models:

Vanilla Prompting (ZS): Zero-shot, natural language to RTL code / summary generation.

Chain-of-Descriptions (CoDes) (Vijayaraghavan et al., 2024a): Uses intermediate textual plans (descriptions) to guide LLM synthesis.

ReAct Prompting (Yao et al., 2023):Iterative reasoning-and-action loops for stepwise generation and refinement.

Vanilla RAG (VRAG-CodeBERT): Uses a generic CodeBERT retriever without RTL-specific fine-tuning.

VRAG-FT: RAG with a retriever fine-tuned on our RTL-IR dataset of aligned (NL, symbolic logic, RTL) triplets.

RTLCoder (Liu et al., 2024): An open-source LLM trained specifically for RTL generation, designed to be efficient and locally deployable.

CodeV (Zhao et al., 2025): Instruction-tuned Verilog generation LLMs using a multi-level summarization strategy, shown to improve code generation by first summarizing then generating.

SymDiRec: Our full neuro-symbolic pipeline, combining symbolic decomposition, retrieval, and LLM-guided verification.

SymDiRec-GT (Oracle): An upper-bound variant that retrieves using ground-truth symbolic snippets, to assess ideal retrieval conditions.

C.3 LLM Settings and Evaluation

We run all methods using two LLMs: a proprietary model (GPT-4o) and an open-source model (Llama-3, 70B). Synthesis correctness is measured via self-verifying testbenches using Pass@1, and summarization quality is scored using ROUGE-L against reference summaries. For statistical robustness, we perform five independent runs per setting and report mean and standard deviation; paired t-tests (e.g., comparing SYMDIREC vs VRAG-FT or vs other baselines) are computed and reported in the main results table.

D Implementation Details & Computation Cost

Our system is implemented in Python using the PyTorch framework, enabling flexible model development and efficient training. Retriever fine-tuning is performed on two NVIDIA V100 GPUs, which allows for effective processing our RTL-IR dataset. We orchestrate LLM queries using LangChain, and index high-dimensional vector representations with Milvus, a high-performance vector database offering both in-memory and GPU-accelerated similarity search. In contrast, traditional search solutions such as Elasticsearch (Elastic, 2024) and Elastic's Learned Sparse Encoder Representations (ELSER) (Elastic, 2024) serve as baselines; while Elasticsearch excels in full-text search, its semantic retrieval is limited compared to Milvus and ELSER.

Our SYMDIREC pipeline processes multiple prompts per task in parallel, achieving an average turnaround time of approximately 5–10 seconds. This efficiency demonstrates the practical viability of our approach for RTL code synthesis and summarization tasks. These implementation choices align with recent literature on retrieval-augmented generation (Lewis et al., 2020; Guu et al., 2020) and domain-specific fine-tuning strategies. The integration of advanced indexing via Milvus and query orchestration using LangChain not only outperforms traditional retrieval methods but also substantially enhances the overall performance of our system.

Task	Symbolic Decomposition	Retrieved Context

8-bit Ripple Carry Adder	**LSB Half-Adder:** $S_0 = A_0 \oplus B_0$, $C_1 = A_0 \wedge B_0$
	Bits 1–7 Full-Adders: $S_i = A_i \oplus B_i \oplus C_i$, $C_{i+1} = (A_i \wedge B_i) \vee (B_i \wedge C_i) \vee (A_i \wedge C_i)$
		Half-Adder (LSB)
		Verilog:
		`module half_adder(input a, b, o`
		`utput sum, carry);`
		`  assign sum = a ^ b;`
		`  assign carry = a & b;`
		`endmodule`
		VHDL:
		`entity half_adder is`
		`  port(a, b: in std_logic;`
		`  sum, carry: out std_logic);`
		`end entity;`
		`architecture rtl of half_adder is`
		`begin`
		`  sum <= a xor b;`
		`  carry <= a and b;`
		`end architecture;`
		Full-Adder (bits 1–7)
		Verilog:
		`module full_adder(input a, b, cin,`
		`output sum, cout);`
		`  assign sum = a ^ b ^ cin;`
		`  assign cout = (a & b) | (b & cin) | (a & cin);`
		`endmodule`
		VHDL:
		`entity full_adder is`
		`  port(a, b, cin: in std_logic; sum,`
		`  cout: out std_logic);`
		`end entity;`
		`architecture rtl of full_adder is`
		`begin`
		`  sum <= a xor b xor cin;`
		`  cout <= (a and b) or (b and cin) or (a and cin);`
		`end architecture;`

Table 4: Qualitative example for 8-bit ripple carry adder. Symbolic decomposition shows Boolean/logical expressions for each submodule. Retrieved context contains modular Verilog and VHDL code snippets corresponding to these submodules. All submodules passed simulation/testbench.