DeepRTL2: A Versatile Model for RTL-Related Tasks

Yi Liu[1,2*], **Hongji Zhang**[1,2*], **Yunhao Zhou**[1,2],
Zhengyuan Shi[1,2], **Changran Xu**[1,2], **Qiang Xu**[1,2]
[1]The Chinese University of Hong Kong
[2]National Technology Innovation Center for EDA
{yliu22,zyshi21,qxu}@cse.cuhk.edu.hk
{hongjizhang183,yunhaoz.cs,xxuchangran}@gmail.com

Abstract

The integration of large language models (LLMs) into electronic design automation (EDA) has significantly advanced the field, offering transformative benefits, particularly in register transfer level (RTL) code generation and understanding. While previous studies have demonstrated the efficacy of fine-tuning LLMs for these generation-based tasks, embedding-based tasks, which are equally critical to EDA workflows, have been largely overlooked. These tasks, including natural language code search, RTL code functionality equivalence checking, and performance prediction, are essential for accelerating and optimizing the hardware design process. To address this gap, we present **DeepRTL2**, a family of versatile LLMs that unifies both generation- and embedding-based tasks related to RTL. By simultaneously tackling a broad range of tasks, DeepRTL2 represents the first model to provide a comprehensive solution to the diverse challenges in EDA. Through extensive experiments, we show that DeepRTL2 achieves state-of-the-art performance across all evaluated tasks.

1 Introduction

The rapid advancement of large language models (LLMs) has had a profound impact on various domains (Singhal et al., 2023; Bran and Schwaller, 2024), including electronic design automation (EDA). Recently, LLMs have shown remarkable potential in automating and enhancing tasks related to the generation and understanding of register transfer level (RTL) code (Liu et al., 2024; Zehua et al., 2024; Zhao et al., 2024; Liu et al., 2025). These models are capable of generating RTL code from high-level natural language instructions or summarizing the functionality of existing RTL code, thereby substantially improving the efficiency of hardware design workflows. While the application of LLMs to these generation-based tasks has yielded impressive results, their full potential at the RTL stage remains underexplored, particularly in embedding-based tasks that are equally crucial to the design process.

Embedding-based tasks like natural language code search, RTL code functionality equivalence checking, and performance prediction are vital for accelerating and optimizing the hardware design process. Natural language code search allows designers to quickly query large RTL codebases with simple natural language descriptions, enabling efficient identification and reuse of relevant modules, thus reducing search time. Moreover, verification and optimization are two key time-consuming bottlenecks in hardware design. RTL code functionality equivalence checking can significantly reduce the time spent on verification by quickly assessing whether two designs are functionally equivalent. Performance prediction tasks, such as power, performance, and area (PPA) estimation, enable early evaluation of RTL design efficiency. Accurate performance predictions can guide RTL code optimization, minimizing the need for time-intensive trial-and-error. Together, these tasks enhance code reuse, verify functionality, and provide early performance feedback, resulting in a more streamlined and efficient design workflow. Previous methods have attempted to apply machine learning solutions for hardware design verification (Vasudevan et al., 2021) and performance prediction (Fang et al., 2023), but they are typically design-specific, lack generalizable representations for RTL designs, or do not operate directly at the RTL stage.

In this paper, we introduce **DeepRTL2**, a family of versatile LLMs designed to address both generation- and embedding-based tasks related to RTL. By unifying these tasks in a single model, DeepRTL2 offers a comprehensive solution to the multifaceted challenges inherent in EDA. Unlike previous work, which has primarily focused on

[*]These authors contributed equally.

Findings of the Association for Computational Linguistics: ACL 2025, pages 6485–6500
July 27 - August 1, 2025

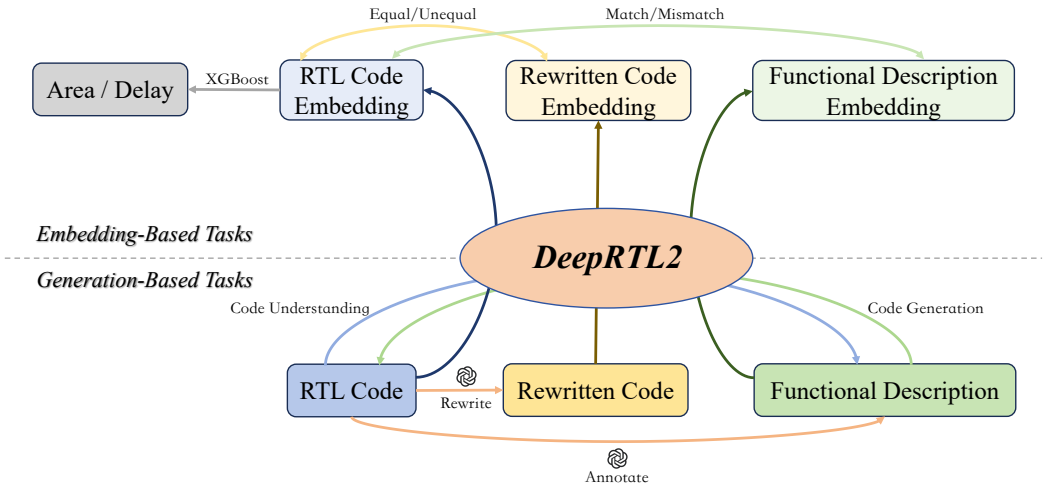

Figure 1: The overview of DeepRTL2. It can handle both generation- and embedding-based tasks at the RTL stage. For generation-based tasks, it performs RTL code generation and understanding. For embedding-based tasks, it uses cosine similarity scores between the embeddings of RTL code and functional descriptions to assess their match, enabling natural language code search. Additionally, cosine similarity between RTL code embeddings and rewritten code embeddings is used for functionality equivalence checking. Furthermore, a prediction model, such as XGBoost, can be applied to predict area and delay metrics based on code embeddings.

generation, DeepRTL2 is the first model to provide a unified framework for handling a broad range of critical EDA tasks, including code generation, understanding, natural language code search, functionality equivalence checking, and performance prediction. Figure 1 provides an overview of our model. To achieve this, we have carefully curated a comprehensive dataset and developed new benchmarks for each task, with a particular focus on the embedding-based tasks, for which no existing datasets or benchmarks are available. We have adopted state-of-the-art decoder-only models, such as Llama-3.1 (Dubey et al., 2024) and DeepSeek-Coder (Guo et al., 2024), as our base models for fine-tuning, given their superior performance over other architectures in the open-source LLM space. To enable these models to handle both generation- and embedding-based tasks, we adapt the generative representational instruction tuning (GRIT) approach (Muennighoff et al., 2025) for fine-tuning, ensuring that DeepRTL2 can effectively manage the diverse tasks at the RTL stage. Through extensive experimentation, we demonstrate that the DeepRTL2 series achieves state-of-the-art performance across all evaluated tasks.

2 Related Works

2.1 Register Transfer Level in EDA

Register transfer level (RTL) is a key abstraction in EDA that describes the flow of data between registers and the operations performed on this data. It is typically expressed using hardware description languages (HDLs), with Verilog being the most widely used HDL in the industry. Thus, throughout this paper, we use the terms RTL code and Verilog code interchangeably. In modern hardware design, engineers usually begin with specifications in natural language, which are then manually translated into HDLs before synthesizing the circuit elements (Blocklove et al., 2023). RTL serves as an intermediary between high-level design specifications and low-level implementation details, enabling designers to describe intricate digital systems while retaining flexibility for synthesis into gate-level representations. Within EDA workflows, RTL plays a crucial role in various phases, including functional verification, performance estimation, synthesis, and optimization. Efficient handling of RTL code is essential for minimizing design time, improving performance, and ensuring correctness.

2.2 LLMs for RTL

With the rapid development of artificial intelligence (AI), there has been increasing interest in leveraging these technologies to automate and enhance RTL-based design workflows (Chen et al., 2024). A key area of focus has been the use of LLMs for RTL code generation and understanding, which has shown great promise in improving hardware design efficiency (Thakur et al., 2023; Liu et al., 2023; Lu et al., 2024). Recent works have fine-tuned open-source LLMs to generate

high-quality RTL code from natural language descriptions (Chang et al., 2024; Liu et al., 2024; Thakur et al., 2024; Zehua et al., 2024; Zhang et al., 2024; Zhao et al., 2024), achieving significant improvements in the automation of hardware design process. Additionally, models like DeepRTL (Liu et al., 2025) have extended these capabilities by introducing RTL code understanding tasks, *i.e.*, summarizing the functionality of existing code, which facilitates collaboration and comprehension among hardware designers. Despite the great success achieved in these generation-based tasks, prior research has largely overlooked embedding-based tasks, which are equally critical for addressing challenges in EDA. Embedding-based tasks, such as natural language code search, RTL code functionality equivalence checking, and performance prediction, are essential for improving the efficiency of code reuse, verification, and optimization within hardware design workflows. Unlike generation-based tasks, which focus on producing new RTL code, embedding-based tasks involve understanding and analyzing existing designs, providing valuable insights into design reusability, correctness, and performance. Meanwhile, even if some studies have applied machine learning techniques for hardware design verification (Vasudevan et al., 2021) and performance prediction (Fang et al., 2023), these efforts are either design-specific, lack generalizable representations for RTL designs, or do not operate directly at the RTL stage. In contrast, this work introduces DeepRTL2, a versatile model capable of handling both generation- and embedding-based tasks, achieving superior performance across all evaluated tasks despite its versatility.

2.3 Embedding Capabilities of Decoder-Only LLMs

Compared to bidirectional encoders like BERT (Devlin, 2018) and encoder-decoder architectures like T5 (Raffel et al., 2020), decoder-only LLMs have demonstrated superior performance across a range of language tasks (Brown et al., 2020). However, their potential for text embedding tasks was largely overlooked until recently. In recent years, several studies have focused on adapting decoder-only LLMs for language embedding tasks (Jiang et al., 2023; Wang et al., 2023; BehnamGhader et al., 2024; Springer et al., 2024; Lei et al., 2024; Lee et al., 2024). Notably, Muennighoff et al. (2025) introduce the GRIT training strategy, which employs a multi-task training objective function to enable a single decoder-only LLM to both generate content and encode text into fixed-length vectors. Despite their success on various language embedding benchmarks, these models primarily focus on general embedding tasks, which limits their effectiveness on specialized tasks like RTL embedding-based tasks. To the best of our knowledge, there is no model that has been specifically trained for RTL embedding, despite its critical role in optimizing hardware design workflows. DeepRTL2 is the first model explicitly designed for RTL embedding-based tasks, outperforming general-purpose text embedding models on our benchmarks.

3 Dataset

Previous research has predominantly focused on generation-based tasks, resulting in a notable gap in available datasets for the embedding-based tasks considered in this paper. Moreover, the availability of RTL code is limited even for generation-based tasks, due to the proprietary nature of hardware designs. To fill this gap, we have curated a comprehensive dataset tailored to support both generation- and embedding-based tasks at the RTL stage. Furthermore, we have established new benchmarks specifically for the embedding-based tasks, which have been largely neglected in previous work.

3.1 Generation-Based Tasks

3.1.1 RTL Code Generation

RTL code generation involves automatically synthesizing RTL code from user-defined natural language descriptions, streamlining hardware design and enabling a more accessible development process. To construct a high-quality dataset for this task, we follow the data construction pipeline proposed in DeepRTL (Liu et al., 2025), given its demonstrated effectiveness in generation-based tasks. The process begins by collecting `.v` files from GitHub[1] using the keyword `Verilog`. Each file is then segmented into individual Verilog modules, with each module representing a distinct functional unit. To ensure dataset quality and reduce redundancy, we remove modules that are predominantly composed of comments or lack structurally complete `module` and `endmodule` declarations. Additionally, we apply MinHash and Jaccard similarity metrics (Yan et al., 2017) to eliminate duplicates. To further refine the dataset, we employ the Stagira Verilog parser (Chen et al., 2023) to filter out mod-

[1] https://github.com/

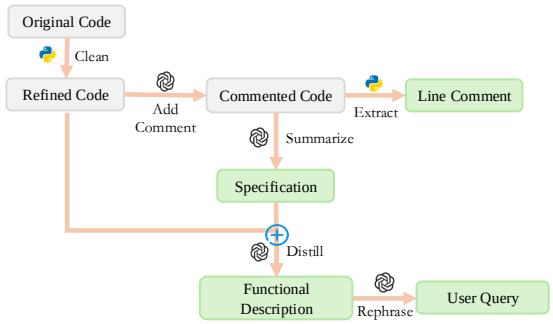

Figure 2: The annotation process for the RTL code generation/understanding dataset. After obtaining the high-level functional description, we prompt GPT-4o to rephrase it into a user query format, which is then used to construct the natural language code search dataset.

ules containing syntax errors, ensuring that only syntactically valid Verilog code is retained.

For annotation, we adopt the chain-of-thought (CoT) prompting strategy used in DeepRTL, leveraging GPT-4o (Hurst et al., 2024), a state-of-the-art LLM, to generate structured and informative annotations. Specifically, we first query GPT-4o to insert line-level comments into the Verilog modules, then extract line-level descriptions, pairing individual lines of RTL code with corresponding natural language explanations. Next, we prompt GPT-4o to generate a detailed specification for each module, comprising a summary of the module's functionality and a comprehensive explanation of its implementation process. By integrating these specifications with the module code, we construct high-level functional descriptions—succinct one-sentence summaries that capture the core functionality of each Verilog module. The resulting dataset consists of Verilog modules enriched with line-level comments, detailed specifications, and succinct high-level functional descriptions, facilitating both generation and understanding tasks in RTL design. Figure 2 provides an overview of the annotation process. For specifics on the prompts used, please refer to DeepRTL. To ensure the quality of the generated annotations, we have conducted human evaluations, as detailed in Appendix A.

To further expand the training dataset and improve model performance, we augment our dataset with open-source Verilog datasets from RTL-Coder (Liu et al., 2024), MG-Verilog (Zhang et al., 2024), and DeepCircuitX (Li et al., 2025). These datasets provide additional RTL designs with diverse structures and functionalities, while also incorporating different annotation strategies. The diversity in annotations improves the model's adaptability to varying description styles, enhancing its robustness across various RTL-related tasks.

3.1.2 RTL Code Understanding

RTL code understanding focuses on summarizing the functionality of existing Verilog code, enhancing collaboration and comprehension among hardware designers. The dataset for this task is derived from the RTL code generation dataset, with Verilog code as input and corresponding natural language description as output. In the absence of a standardized benchmark for this task, we build upon the benchmark introduced in DeepRTL, which originally comprises 100 Verilog designs. To improve evaluation reliability and ensure broader coverage, we extend this benchmark to include 500 high-quality Verilog modules with diverse functionalities. Each module is annotated by professional hardware designers with a concise summary of its functionality along with a detailed description of the specific operations involved in its execution. This extended benchmark establishes a more robust and comprehensive foundation for evaluating RTL code understanding capabilities.

3.2 Embedding-Based Tasks

3.2.1 Natural Language Code Search

Natural language code search refers to the process of querying a large codebase using natural language to find relevant code snippets. It involves embedding both the user query and each code snippet into vectors, then calculating their similarity. The snippet with the highest similarity score is considered the best match for the user's requirements. This task is particularly crucial for hardware design, as it enables code reuse, improves efficiency, and accelerates the transition from user specifications to RTL code. For this task, we reuse the dataset and benchmark from the RTL code understanding task. However, since the functional descriptions in the understanding dataset often contain specific identifiers, introducing the risk of data leakage, and are too complex for direct use in practical code search, we employ GPT-4o to rephrase the descriptions into a user query format, as shown in Figure 2. The rephrasing ensures that the new descriptions meet the following conditions: (1) no references to specific identifiers, (2) retention of the core functionality and high-level logic, and (3) clarity and simplicity, resembling how a user would query for relevant code based on its functionality. After this rephrasing process, we obtain the natural language

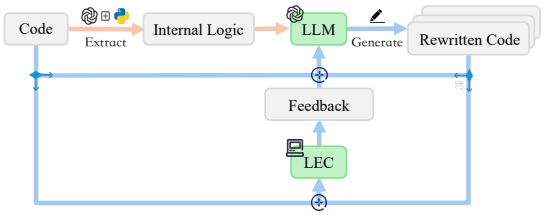

Figure 3: The feedback-driven code rewrite process.

code search dataset and benchmark in the format $\{(\text{user\_query}_i, \text{RTL\_code}_i)\}_{i=1}^{n}$. For details on the prompt used to rephrase the functional descriptions, please refer to the Appendix B.

3.2.2 Functionality Equivalence Checking

Functionality equivalence checking is a critical verification step in hardware design, ensuring that different RTL implementations exhibit identical behavior despite structural differences. To construct a dataset for this task, we develop a feedback-driven CoT prompting strategy using GPT-4o, as shown in Figure 3. Given a Verilog module, we first prompt GPT-4o to introduce significant modifications to its internal logic while preserving its intended functionality. We then use Yosys (Wolf et al., 2013) to perform logic equivalence checking (LEC), which verifies whether the original and modified designs are functionally equivalent. Based on Yosys feedback—classified as equivalent, inequivalent, or syntax error—we iteratively refine the modifications. Specifically, we incorporate the original design, rewritten design, and verification results into the prompt to guide GPT-4o in generating alternative implementations. This process is repeated for two to three rounds per design, ensuring a diverse set of functionally equivalent and inequivalent pairs. The resulting dataset consists of paired RTL designs, where some maintain functional equivalence while others introduce subtle variations. Since only implementation details differ, distinguishing equivalent from inequivalent designs presents a significant challenge for models. Additionally, we adapt RTLLM v2.0 (Lu et al., 2024), a Verilog generation benchmark, to construct a new benchmark for functionality equivalence checking. Applying the same feedback-driven CoT strategy to its 50 verified Verilog designs, we generate multiple alternative implementations, expanding our benchmark to 400 code pairs. This benchmark provides a diverse and well-validated resource for evaluating functionality equivalence checking. For further details on this process, please refer to the Appendix C.

3.2.3 Performance Prediction

Performance prediction plays a crucial role in the early stages of hardware design, enabling designers to estimate key circuit characteristics before physical implementation. Accurate predictions allow for informed architectural decisions, reducing design iterations and improving overall efficiency. Among the commonly used PPA metrics, delay and area are the primary focus in early-stage evaluations, as accurate power estimation requires detailed workload to specify the circuit's dynamic behavior, which is unavailable at the RTL stage. In this work, we construct a performance prediction dataset by synthesizing and mapping RTL designs into netlists using Yosys (Wolf et al., 2013) with the SkyWater 130nm technology library (Google, 2021). We then utilize open-source ABC (Brayton and Mishchenko, 2010) tool to extract delay and area metrics, where delay metric is reported by the static timing analysis, and area metric reflects the total logic footprint, which directly impacts manufacturing cost. This process provides a dataset that captures essential performance characteristics of RTL designs, facilitating learning-based performance estimation. For a comprehensive summary of all dataset statistics, please refer to the Appendix D.

4 Methodology

4.1 Model Training

We choose Llama-3.1 (Dubey et al., 2024) and DeepSeek-Coder (Guo et al., 2024) as the base models for training. Specifically, we fine-tune meta-llama/Llama-3.1-8B-Instruct[2] and deepseek-ai/deepseek-coder-6.7b-instruct[3]. Our training consists of two stages. In the first stage, we follow the curriculum learning strategy adopted by DeepRTL (Liu et al., 2025) and train the base model solely on RTL code generation and understanding data. In the second stage, we incorporate embedding data into the training set and train the model on both RTL code generation/understanding and embedding tasks, utilizing the training framework of GRIT (Muennighoff et al., 2025).

4.1.1 First-Stage Training

Following DeepRTL, we apply a curriculum learning strategy in the first stage of our training pipeline, which can be further divided into four sub-stages:

[2] https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct

[3] https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct

Model	syntax			function		
	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10
GPT-3.5	56.50%	69.72%	71.75%	30.10%	39.59%	41.40%
GPT-4o	72.00%	77.31%	78.53%	49.70%	56.80%	58.84%
o1-preview	**76.20%**	**83.71%**	**84.00%**	**50.00%**	**60.86%**	**62.52%**
CodeV-CodeLlama	47.70%	74.96%	82.20%	22.00%	39.49%	45.74%
CodeV-CodeQwen	51.50%	77.71%	82.17%	23.10%	44.54%	52.22%
CodeV-DeepSeek	57.60%	80.23%	83.25%	30.00%	49.63%	54.74%
DeepRTL-220m	60.69%	78.81%	80.88%	28.79%	45.86%	49.66%
DeepRTL-16b	63.79%	74.82%	80.05%	38.91%	47.24%	51.72%
Llama-3.1	32.40%	57.01%	62.76%	14.60%	26.04%	30.16%
DeepSeek-Coder	59.30%	72.38%	74.67%	31.40%	39.59%	42.57%
DeepRTL2^{1st}-Direct (Llama)	54.48%	63.52%	67.99%	16.28%	28.78%	32.76%
DeepRTL2^{1st}-Direct (DeepSeek)	60.60%	73.12%	75.70%	32.50%	44.42%	47.96%
DeepRTL2^{1st} (Llama)	67.90%	77.53%	79.52%	43.70%	49.98%	50.00%
DeepRTL2^{1st} (DeepSeek)	63.50%	76.74%	80.10%	39.70%	51.96%	54.70%
DeepRTL2 (Llama)	68.30%	81.31%	83.36%	33.70%	49.57%	52.90%
DeepRTL2 (DeepSeek)	71.60%	80.58%	81.75%	38.50%	52.62%	55.99%

Table 1: The performance evaluation for RTL code generation using the pass@k metric, with k set to 1, 5, and 10. The best results among all models are bolded, and the best results among open-source models are underlined.

Model	F1
text-embedding-3-small	0.189
text-embedding-3-large	0.290
GritLM-7B	0.269
DeepRTL2$^{no\text{-}hard}$ (Llama)	**0.476**
DeepRTL2$^{no\text{-}hard}$ (DeepSeek)	0.464
DeepRTL2 (Llama)	0.463
DeepRTL2 (DeepSeek)	0.453

Table 2: The performance evaluation for natural language code search using the F1 metric. The best result is bolded, and the second-best result is underscored.

training with line-level data, module-level data with specifications, module-level data with high-level descriptions, data with varying prompts. For details on these sub-stages, please refer to Appendix E.

4.1.2 Second-Stage Training

Following GRIT, in the second stage of training, we combine the generation/understanding and embedding tasks. For the generation/understanding training, we reuse the high-quality data from the fourth sub-stage of the first-stage training. For the embedding task, we employ contrastive learning to learn contextualized representations that preserve the semantic information of the original text and code. Details for constructing the contrastive learning training set can be found in Appendix F. In the embedding part of the second-stage training, we first use data that does not contain hard negatives and then incorporate data with hard negative samples. For more details on the loss functions at different sub-stages, please refer to Appendix G. For additional details on the hyperparameters and hardware resources used, please refer to Appendix H.

4.2 Model Evaluation

For RTL code generation, we utilize the latest version of the widely adopted RTLLM v2.0 benchmark (Lu et al., 2024), which contains 50 designs paired with corresponding natural language descriptions and testbenches. To measure Verilog generation accuracy, we use the pass@k metric, which estimates the proportion of problems that can be solved at least once within k attempts:

$$\text{pass@}k := \mathop{\mathbb{E}}_{\text{problems}} \left[\frac{1 - \binom{n-c}{k}}{\binom{n}{k}} \right] \quad (1)$$

where $n \geq k$ represents the total number of trials for each problem, and c denotes the number of trails that pass the functional check. In our experiments, we set $n = 20$ to mitigate randomness in results. The pass@k metric is reported for both syntactical and functional accuracy. Following RTLCoder (Liu et al., 2024), we evaluate performance across multiple generation temperatures (0.2, 0.5, and 0.8) and report the best performance across these settings.

For RTL code understanding, we use the benchmark constructed in Section 3.1.2. To evaluate the model's performance, we apply both traditional machine translation metrics—BLEU (Papineni et al.,

Model	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L	METEOR	Emb. Sim.	GPT Score
GPT-3.5	3.34	28.20	10.46	25.11	20.36	0.740	0.510
GPT-4o	4.59	29.26	11.48	25.74	22.78	0.761	0.549
o1-preview	3.73	28.00	10.39	24.98	20.48	0.748	0.535
CodeV-DeepSeek	3.05	25.14	9.78	23.25	20.23	0.705	0.495
CodeV-CodeQwen	2.80	24.91	8.27	22.75	21.07	0.747	0.499
DeepRTL-220m	13.06	37.56	19.85	34.72	34.37	0.806	0.600
DeepRTL-16b	12.85	37.43	19.34	34.63	33.09	0.802	0.597
Llama-3.1	2.68	25.37	10.39	23.75	17.16	0.730	0.430
DeepSeek-Coder	2.56	24.52	7.72	22.45	22.83	0.756	0.571
DeepRTL2^{1st}-Direct (Llama)	11.28	34.29	16.35	33.63	27.73	0.754	0.580
DeepRTL2^{1st}-Direct (DeepSeek)	12.07	36.37	17.78	33.78	28.56	0.767	0.602
DeepRTL2^{1st} (Llama)	13.34	37.74	19.54	**34.76**	33.46	0.798	0.594
DeepRTL2^{1st} (DeepSeek)	13.53	37.52	19.68	34.68	33.28	0.814	0.612
DeepRTL2 (Llama)	13.84	**37.97**	20.69	34.42	**34.75**	0.813	0.603
DeepRTL2 (DeepSeek)	**13.96**	37.93	**20.73**	34.34	34.74	**0.820**	**0.616**

Table 3: The performance evaluation for RTL code understanding. BLEU-4 refers to the smoothed BLEU-4 score, while Emb. Sim. represents the embedding similarity metric. The best results are highlighted in bold, and the second-best results are underscored.

2002), ROUGE (Lin, 2004), and METEOR (Banerjee and Lavie, 2005)—which primarily assess lexical similarity, as well as the embedding similarity and GPT score metrics introduced in DeepRTL (Liu et al., 2025), which focus on semantic similarity. This combination of evaluation metrics provides a comprehensive assessment of the model's ability to understand RTL code, capturing both surface-level and deeper, semantic-level understanding. For further details on how to compute these metrics, please refer to the Appendix I.

For natural language code search, we utilize the benchmark introduced in Section 3.2.1. To assess the model's ability to retrieve relevant code from a large codebase based on a user's query, we follow the bitext mining setting from MTEB (Muennighoff et al., 2022). In our evaluation process, the inputs consist of two sets: the first set contains functional descriptions, while the second set consists of Verilog code snippets. For each description in the first set, the best matching code snippet in the second set is identified using cosine similarity. We report F1 score, precision, and recall for each model, with F1 serving as the primary evaluation metric for natural language code search.

For functionality equivalence checking, we utilize the benchmark introduced in Section 3.2.2. To evaluate the models' ability to check functional equivalence, we follow the pair classification setting from MTEB (Muennighoff et al., 2022). In this evaluation, the inputs consist of several pairs of RTL codes. For each pair, the model assigns a binary label: 1 for "functionally equivalent" and 0 for "functionally inequivalent". The binary label is determined by calculating the cosine similarity of their embeddings and comparing the similarity score to a predefined threshold. For each model, we first identify the optimal accuracy threshold and compute the accuracy score. We then determine the best F1 threshold and report the F1, precision, and recall scores. Finally, we calculate the average precision score based on the similarity scores of the code pairs and their corresponding ground-truth labels. Average precision is the primary evaluation metric for RTL code functionality equivalence checking, with other metrics also reported.

For performance prediction, we use the dataset introduced in Section 3.2.3. This task aims to test the expressive power of code embeddings for predicting performance metrics, such as area and delay, at the early stage of RTL design. To achieve this, we first encode each code snippet into a fixed-length vector and create a new dataset in the format $\{(\text{code\_embedding} \in \mathbb{R}^p, \text{area} \in \mathbb{R}, \text{delay} \in \mathbb{R})_i\}_{i=1}^n$, where p is the embedding dimension and n is the dataset size. The dataset is then split into training and test sets at an 80:20 ratio. In this paper, we use XGBoost (Chen and Guestrin, 2016) as the regression model, training separate models for area and delay prediction. The trained models are evaluated on the test set using r2_score, mean absolute percentage error (MAPE) and root relative squared error (RRSE), with their formulas provided below:

$$\text{r2\_score}(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \quad (2)$$

$$\text{MAPE}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100\% \quad (3)$$

$$\text{RRSE}(y, \hat{y}) = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2}} \quad (4)$$

5 Experimental Results

5.1 Generation-Based Tasks

For comparison, we select several baseline models: the state-of-the-art commercial models, OpenAI's GPT-3.5, GPT-4o, and o1-preview, which represent the most advanced general-purpose LLMs currently available. We also include the CodeV series (Zhao et al., 2024), a collection of leading open-source models specifically designed for RTL code generation, as well as the original DeepRTL models (Liu et al., 2025), which have shown strong performance in both RTL code generation and understanding. All these models have demonstrated excellent capabilities in Verilog generation-based tasks (Liu et al., 2025), making them strong baselines for evaluating the performance of DeepRTL2. Additionally, we report the performance of base models, Llama-3.1 and DeepSeek-Coder, to show the effectiveness of our dataset construction and training strategy.

Table 1 reports the pass@k results for RTL code generation across different models, with k set to 1, 5, and 10. The results show that o1-preview outperforms all other models, likely due to its design for addressing complex tasks, including programming. The DeepRTL2 models, however, achieve the best performance among all open-source models, with results comparable to GPT-4o. The performance improvement from base models to DeepRTL2 highlights the effectiveness of our dataset construction process and training strategy. Furthermore, DeepRTL2 outperforms the original DeepRTL models, likely due to the incorporation of additional open-source datasets, aside from data sourced from GitHub, and the inclusion of more diverse problem formulations that enhance DeepRTL2's generalization ability. Given that DeepRTL2 is a multi-task model and the generation benchmark may overlap with the training data used by OpenAI's models, these results highlight DeepRTL2's impressive performance for this task.

Table 3 presents the results for RTL code understanding. Since the CodeV-CodeLlama model outputs random messages for this task, we exclude it from the comparison. The results show that DeepRTL2 models significantly outperform all other models, including the previous state-of-the-art DeepRTL models, underscoring its strong capabilities in RTL code understanding. Notably, DeepRTL2 surpasses GPT-4o by a substantial margin, despite the fact that its training data is annotated using GPT-4o. The main reason is that during benchmark testing, all models, including GPT-4o, are required to generate high-level functional descriptions directly from RTL code. As shown in Appendix A, CoT-based annotations are more accurate than direct annotations. This enhanced annotation quality contributes to DeepRTL2's superior performance in RTL code understanding.

5.2 Embedding-Based Tasks

Since none of the existing models are specifically designed for RTL embedding-based tasks, the baselines used for the generation-based tasks, *e.g.*, CodeV series and DeepRTL models, perform poorly in this setting. These models show near-zero performance, with an F1 score close to 0 on the natural language code search task and an average precision of approximately 0.5 on the functionality equivalence checking task. Therefore, we select state-of-the-art general-purpose embedding models as baselines for comparison. These include OpenAI's text embedding models (text-embedding-3-small, text-embedding-3-large) (Neelakantan et al., 2022) and open-source models like GritLM-7B (Muennighoff et al., 2025).

Table 2 presents the F1 scores for the natural language code search task. The results show that our DeepRTL2 models outperform all baseline models by a significant margin, demonstrating the effectiveness of our dataset and training strategy for this task. For the full evaluation results on natural language code search, please refer to Appendix J.

Table 5 presents the average precision scores for the functionality equivalence checking task. The results show that DeepRTL2 models outperform all other baselines, demonstrating their effectiveness in capturing functional relationships between RTL modules. The full evaluation results are in Appendix J. It is important to emphasize that our embedding-based verification is not intended to replace the traditional verification process, but rather to serve as an efficient preliminary step that can significantly streamline the verification flow.

Table 4 presents the results for performance prediction on area and delay. Our DeepRTL2 series models outperform the baseline models across all metrics. These results highlight that the code embeddings generated by the DeepRTL2 models are more expressive for predicting performance-related metrics such as area and delay.

Model	Area			Delay		
	r2_score	MAPE	RRSE	r2_score	MAPE	RRSE
text-embedding-3-small	0.603	5.568	0.630	0.608	0.883	0.626
text-embedding-3-large	0.699	4.446	0.548	0.699	0.705	0.548
GritLM-7B	0.651	3.878	0.591	0.651	0.726	0.591
DeepRTL2$^{no\text{-}hard}$ (Llama)	0.510	2.828	0.700	0.735	0.471	0.515
DeepRTL2$^{no\text{-}hard}$ (DeepSeek)	**0.805**	2.947	**0.445**	0.743	0.449	0.507
DeepRTL2 (Llama)	0.759	1.966	0.490	**0.773**	0.469	**0.476**
DeepRTL2 (DeepSeek)	0.773	**1.598**	0.476	0.772	**0.448**	0.478

Table 4: The performance evaluation for performance prediction on area and delay using r2_score, MAPE and RRSE metrics. The best results among all models are bolded, and the second-best results are underscored.

Model	Average Precision
text-embedding-3-small	0.565
text-embedding-3-large	0.498
GritLM-7B	0.541
DeepRTL2$^{no\text{-}hard}$ (Llama)	0.518
DeepRTL2$^{no\text{-}hard}$ (DeepSeek)	0.481
DeepRTL2 (Llama)	**0.667**
DeepRTL2 (DeepSeek)	0.591

Table 5: The performance evaluation for RTL code functionality equivalence checking using the average precision metric. The best result among all models is bolded, and the second-best result is underscored.

5.3 Ablation Studies

In this section, we conduct ablation studies to demonstrate the effectiveness of different dataset components and training strategies. In the first training stage, we adopt a curriculum learning strategy, where the model is progressively trained on line-level data, module-level data with specifications, module-level data with high-level descriptions, and data with varying prompts. While the benefits of curriculum learning have been shown in DeepRTL (Liu et al., 2025), we extend this analysis with additional comparisons. Specifically, we compare our first-stage model (DeepRTL2^{1st}) with a variant trained without curriculum learning (DeepRTL2^{1st}-Direct), both focused on generation-based tasks. As shown in Table 1 and Table 3, the incorporation of curriculum learning significantly improves performance for both code generation and understanding tasks. When we further introduce the second-stage training, *i.e.*, GRIT-based fine-tuning, the performance improves even more, demonstrating the effectiveness of both curriculum learning and GRIT-based fine-tuning strategies.

In the second training stage, we combine contrastive learning and curriculum learning to ensure that our model performs effectively on embedding-based tasks. Specifically, we start with data that excludes hard negatives and gradually introduce hard negative samples, which improves overall performance. To evaluate this strategy, we compare DeepRTL2 with and without hard negatives (DeepRTL2$^{no\text{-}hard}$) in Tables 2, 4, and 5. Since hard negatives primarily influence contrastive learning, these comparisons focus on embedding-based tasks, with negligible impact on generation-based performance. The results show a minor drop in natural language code search accuracy but substantial gains in functionality equivalence checking and performance prediction. Despite the small accuracy decrease in the natural language code search task, DeepRTL2 still outperforms powerful baseline embedding models. This improvement in functionality equivalence checking and performance prediction justifies our decision to integrate hard negatives into the training process.

6 Conclusion

In this work, we present DeepRTL2, a novel family of LLMs that unifies both generation- and embedding-based tasks at the RTL stage, offering a comprehensive solution to the diverse challenges in EDA. By addressing critical tasks including RTL code generation, understanding, natural language code search, functionality equivalence checking, and performance prediction, DeepRTL2 significantly improves the efficiency of hardware design workflows. To develop DeepRTL2, we have curated a comprehensive dataset and established new benchmarks specifically designed for these tasks, particularly the embedding-based ones, for which no suitable resources previously existed. Furthermore, we adapt the GRIT approach to fine-tune the model, enabling it to manage both generation- and embedding-based tasks effectively. Extensive experimentation demonstrates that DeepRTL2 achieves state-of-the-art performance across all evaluated tasks, advancing the application of LLMs in hardware design.

Limitations

There are two main limitations in our work. First, due to the multi-task nature of our model and constraints in time and computing resources, we may not have employed the most optimal training strategy and hyperparameter settings to maximize performance across all tasks. Second, performance prediction directly at the RTL stage is challenging, as RTL designs typically lack detailed information about delay and area metrics. Although our model outperforms others in the evaluation, a significant gap remains in achieving accurate predictions. We hypothesize that incorporating the control data flow graph (CDFG) of RTL designs, which offers a more structured representation of the design's behavior, may facilitate better learning of performance characteristics. In future work, we plan to explore how incorporating CDFG into the DeepRTL2 model series could improve the model's ability to predict performance metrics more accurately.

Acknowledgements

This work was supported in part by the Hong Kong Research Grants Council (RGC) under Grant No. 14212422, 14202824, and C6003-24Y, in part by Huawei Technolgies Co. Ltd. under grant No. N2-2c-TH2420350, and in part by National Technology Innovation Center for EDA, China.

References

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72.

Parishad BehnamGhader, Vaibhav Adlakha, Marius Mosbach, Dzmitry Bahdanau, Nicolas Chapados, and Siva Reddy. 2024. Llm2vec: Large language models are secretly powerful text encoders. *arXiv preprint arXiv:2404.05961*.

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.

Andres M Bran and Philippe Schwaller. 2024. Transformers and large language models for chemistry and drug discovery. In *Drug Development Supported by Informatics*, pages 143–163. Springer.

Robert Brayton and Alan Mishchenko. 2010. Abc: An academic industrial-strength verification tool. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, pages 24–40. Springer.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Kaiyan Chang, Kun Wang, Nan Yang, Ying Wang, Dantong Jin, Wenlong Zhu, Zhirong Chen, Cangyuan Li, Hao Yan, Yunhao Zhou, et al. 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.

Lei Chen, Yiqi Chen, Zhufei Chu, Wenji Fang, Tsung-Yi Ho, Ru Huang, Yu Huang, Sadaf Khan, Min Li, Xingquan Li, et al. 2024. Large circuit models: opportunities and challenges. *Science China Information Sciences*, 67(10):1–42.

Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794.

Xiangli Chen, Yuehua Meng, and Gang Chen. 2023. Incremental verilog parser. In *2023 International Symposium of Electronics Design Automation (ISEDA)*, pages 236–240. IEEE.

Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

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

Google. 2021. Skywater pdk. Accessed: 2025-02-11.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Ting Jiang, Shaohan Huang, Zhongzhi Luan, Deqing Wang, and Fuzhen Zhuang. 2023. Scaling sentence embeddings with large language models. *arXiv preprint arXiv:2307.16645*.

Chankyu Lee, Rajarshi Roy, Mengyao Xu, Jonathan Raiman, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. 2024. Nv-embed: Improved techniques for training llms as generalist embedding models. *arXiv preprint arXiv:2405.17428*.

Yibin Lei, Di Wu, Tianyi Zhou, Tao Shen, Yu Cao, Chongyang Tao, and Andrew Yates. 2024. Meta-task prompting elicits embedding from large language models. *arXiv preprint arXiv:2402.18458*.

Zeju Li, Changran Xu, Zhengyuan Shi, Zedong Peng, Yi Liu, Yunhao Zhou, Lingfeng Zhou, Chengyu Ma, Jianyuan Zhong, Xi Wang, et al. 2025. Deepcircuitx: A comprehensive repository-level dataset for rtl code understanding, generation, and ppa analysis. *arXiv preprint arXiv:2502.18297*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

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*.

Yi Liu, Changran XU, Yunhao Zhou, Zeju Li, and Qiang Xu. 2025. DeepRTL: Bridging verilog understanding and generation with a unified representation model. In *The Thirteenth International Conference on Learning Representations*.

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

Niklas Muennighoff. 2022. Sgpt: Gpt sentence embeddings for semantic search. *arXiv preprint arXiv:2202.08904*.

Niklas Muennighoff, Hongjin SU, Liang Wang, Nan Yang, Furu Wei, Tao Yu, Amanpreet Singh, and Douwe Kiela. 2025. Generative representational instruction tuning. In *The Thirteenth International Conference on Learning Representations*.

Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2022. Mteb: Massive text embedding benchmark. *arXiv preprint arXiv:2210.07316*.

Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pretraining. *arXiv preprint arXiv:2201.10005*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

Karan Singhal, Shekoofeh Azizi, Tao Tu, S Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Tanwani, Heather Cole-Lewis, Stephen Pfohl, et al. 2023. Large language models encode clinical knowledge. *Nature*, 620(7972):172–180.

Jacob Mitchell Springer, Suhas Kotha, Daniel Fried, Graham Neubig, and Aditi Raghunathan. 2024. Repetition improves language model embeddings. *arXiv preprint arXiv:2402.15449*.

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

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.

Shobha Vasudevan, Wenjie Joe Jiang, David Bieber, Rishabh Singh, C Richard Ho, Charles Sutton, et al. 2021. Learning semantic representations to verify hardware designs. *Advances in Neural Information Processing Systems*, 34:23491–23504.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023. Improving text embeddings with large language models. *arXiv preprint arXiv:2401.00368*.

Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys - a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97.

Ziqi Yan, Jiqiang Liu, Gang Li, Zhen Han, and Shuo Qiu. 2017. Privmin: Differentially private minhash for jaccard similarity computation. *arXiv preprint arXiv:1705.07258*.

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

Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Celine Lin. 2024. Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pages 1–5. IEEE.

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

A Human Evaluation of Generated Annotations

To evaluate the reliability and accuracy of GPT-4o-generated annotations, we conduct a human evaluation focusing primarily on the accuracy of high-level functional descriptions, as this is the most challenging and critical aspect of the generation-based tasks. We randomly sample 200 annotated RTL modules and ask professional hardware designers to verify the correctness of the generated descriptions. The human evaluation results show that approximately 90% of these annotations are accurate. In comparison, when we test direct annotations, *i.e.*, generating high-level functional descriptions directly from the original code, the accuracy drops significantly to 70%. This significant difference further demonstrates the effectiveness of the CoT-based annotation strategy.

Additionally, GPT-4o is employed for rewriting RTL code in the functionality equivalence checking task. For this task, we address concerns about accuracy by using EDA tools to verify the functionality equivalence of the rewritten code against the original code. Therefore, all the data collected for this task is validated as ground truth, ensuring the quality and correctness of the rewritten RTL code.

B Prompt For Rephrasing Descriptions

Figure 4 shows the instruction given to GPT-4o to rephrase the code descriptions into their corresponding user query formats.

C Code Rewrite Instructions

Figure 5 illustrates the code rewrite instructions provided to GPT-4o for constructing the functionality equivalence checking dataset. The leftmost column presents the instruction used during the initial rewrite process, where only the original RTL code is available. The subsequent three columns represent instructions based on previously rewritten code, corresponding to the following cases: (1) equivalent rewritten code, (2) inequivalent rewritten code, and (3) rewritten code with syntax errors. Notably, in addition to the code itself, we also include the functional description and specification from Section 3.1.1. This additional context helps the model better understand the intended functionality, leading to improved accuracy in rewriting the code while preserving its functionality.

```
I will provide a Verilog code snippet that defines a module, along with its
corresponding natural language description. The description may reference specific
identifiers in the code, such as variable names and module names. Your task is to
rephrase the description by following these steps:
1. Remove any references to specific identifiers (e.g., module names, variable names,
and specific signals).
2. Retain the core functional meaning and high-level logic of the description.
3. Ensure the rephrased description is concise and suitable for a query, resembling
how a user would search for the relevant code based on its functionality.
4. Avoid unnecessary technical details and long-winded explanations. Keep it simple
and to the point.

Please return only the rephrased description.
code:
{code}

text:
{text}
```

Figure 4: The instruction for rephrasing the code description into the user query format.

D Dataset Statistics

Table 6 presents the overall statistics for all datasets used across the evaluated tasks. Except the performance prediction datasets, all datasets listed in this table are utilized for model training. For the performance prediction datasets, we split them in an 80:20 ratio, creating a training set with 15,000 samples and a test set with 3,766 samples. For performance prediction, we regress area and delay based on code embeddings, without tuning the model.

E Details of First-Stage Training

In Section 3.1.1, we construct a dataset consisting of Verilog modules enriched with line-level comments, detailed specifications, and succinct high-level functional descriptions. These three levels of annotations correspond to the first three sub-stages of our first-stage training pipeline. In the first sub-stage, we train the model using line-level data, where each line of Verilog code is paired with a corresponding natural language comment. The

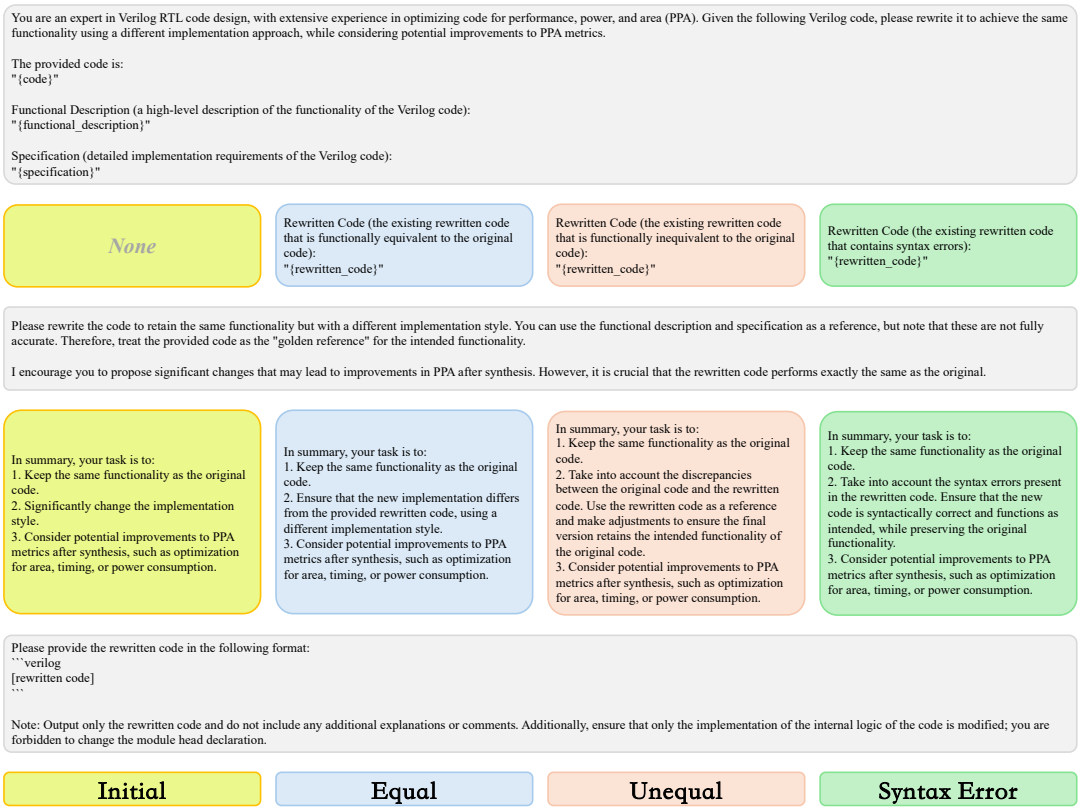

Figure 5: The code rewrite instructions used to construct the functionality equivalence checking dataset.

second sub-stage utilizes module-level data with specifications, providing more detailed descriptions of the Verilog modules. The third sub-stage focuses on module-level data with high-level functional descriptions, offering a broader functional overview of the code. To further refine the dataset and adapt it to a wider range of scenarios, we introduce a fourth sub-stage, where GPT-4o generates varying prompts based on the high-quality data from the third sub-stage. These varying prompts represent different problem descriptions used to generate Verilog code. We find that incorporating this sub-stage improves the model's performance and robustness, as it allows the model to better generalize across a wide range of code generation tasks.

F Contrastive Learning Training Set Construction

In the second-stage training, we apply contrastive learning to enable the model to (1) determine whether a Verilog module matches a given functional description; and (2) assess whether two Verilog code snippets are functionally equivalent.

To construct a dataset for contrastive learning, we first prompt GPT-4o to rewrite Verilog code snippets from the natural language code search training set. The rewrite process is illustrated in Figure 3. After several iterations, we combine the original natural language code search training set with their rewritten code snippets, resulting in four types of new data samples:

- type a: {original_text, original_code}
- type b: {original_text, original_code, equivalent_code}
- type c: {original_text, original_code, inequivalent_code}
- type d: {original_text, original_code, equivalent_code, inequivalent_code}

Since the format of an original data sample in the natural language code search training set is {original_text, original_code}, the four types of data samples correspond to the following scenarios:

- type a corresponds to the case where all rewritten code snippets contain syntax errors.
- type b corresponds to the case where all rewritten code snippets, free of syntax errors, are functionally equivalent to the original code.
- type c corresponds to the case where all rewritten code snippets, free of syntax errors, are not functionally equivalent to the original code.

Task	Description	Source	Count
RTL Code Generation/Understanding	Line Level	DeepRTL2	341310
	Module Level (Detailed Specification)	DeepRTL2	45519
		MG-Verilog	10035
		DeepCircuitX	32809
	Module Level (High-Level Description)	DeepRTL2	46876
		RTLCoder	25001
		MG-Verilog	10037
		DeepCircuitX	38179
Natural Language Code Search	N/A	DeepRTL2	59700
Functionality Equivalence Checking	Equal Pairs	DeepRTL2	9532
	Unequal Pairs	DeepRTL2	23330
Performance Prediction	Area	DeepRTL2	18766
	Delay	DeepRTL2	18766

Table 6: The overall dataset statistics for all evaluated tasks.

- type d corresponds to the case where some rewritten code snippets, free of syntax errors, are functionally equivalent to the original code, while others are not.

For all four types of data samples, we convert them into contrastive learning samples as follows:

- type a:
  - {"query": original_code, "pos": original_text, "neg": None}
  - {"query": original_text, "pos": original_code, "neg": None}
- type b:
  - {"query": original_code, "pos": original_text, "neg": None}
  - {"query": original_text, "pos": original_code, "neg": None}
  - {"query": original_code, "pos": equivalent_code, "neg": None}
  - {"query": equivalent_code, "pos": original_code, "neg": None}
- type c:
  - {"query": original_code, "pos": original_text, "neg": inequivalent_code}
  - {"query": original_text, "pos": original_code, "neg": None}
- type d:
  - {"query": original_code, "pos": original_text, "neg": inequivalent_code}
  - {"query": original_code, "pos": equivalent_code, "neg": inequivalent_code}
  - {"query": original_text, "pos": original_code, "neg": None}
  - {"query": equivalent_code, "pos": original_code, "neg": inequivalent_code}

In each of the contrastive learning samples above, the key "pos" refers to the positive instance of the query code/text, while the key "neg" refers to the hard negative instance. In the embedding part of the second-stage training, we first use samples colored blue that do not contain hard negatives and then incorporate samples colored purple with hard negative instances.

G Training Loss Function

In the second stage of training, we combine generation/understanding and embedding tasks. For generation/understanding, we reuse high-quality data from the fourth sub-stage of the first training stage. For the embedding tasks, we apply contrastive learning to learn contextualized representations that preserve the semantic information of text and code. In the embedding part of the second-stage training, we first use data without hard negatives and later incorporate data with hard negatives. The embedding loss function is defined as follows:

$$E_i^+ = \exp\left(\frac{\sigma(f_\theta(x_i), f_\theta(x_i^+))}{\tau}\right) \quad (5)$$

$$S_i^+ = \sum_{j=1}^{M} \exp\left(\frac{\sigma(f_\theta(x_i), f_\theta(x_j^+))}{\tau}\right) \quad (6)$$

$$S_i^- = \sum_{j=1}^{M} \exp\left(\frac{\sigma(f_\theta(x_i), f_\theta(x_j^-))}{\tau}\right) \quad (7)$$

Model	Precision	Recall	F1 (Main Metric)
text-embedding-3-small	0.173	0.241	0.189
text-embedding-3-large	0.273	0.340	0.290
GritLM-7B	0.255	0.320	0.269
DeepRTL2$^{no\text{-}hard}$ (Llama)	**0.469**	**0.497**	**0.476**
DeepRTL2$^{no\text{-}hard}$ (DeepSeek)	<u>0.456</u>	0.489	<u>0.464</u>
DeepRTL2 (Llama)	0.450	<u>0.493</u>	0.463
DeepRTL2 (DeepSeek)	0.443	0.481	0.453

Table 7: The full performance evaluation results for natural language code search. The best results among all models are bolded, and the second-best results are underscored.

$$\mathcal{L}_{emb1} = -\frac{1}{M}\sum_{i=1}^{M}\log\left(\frac{E_i^+}{S_i^+}\right) \tag{8}$$

$$\mathcal{L}_{emb2} = -\frac{1}{M}\sum_{i=1}^{M}\log\left(\frac{E_i^+}{S_i^+ + S_i^-}\right) \tag{9}$$

where M is the batch size, x_i is the i-th training sample, f_θ is the embedding function (in this paper, we use position-weighted mean pooling method introduced in SGPT (Muennighoff, 2022) to obtain sentence embeddings), τ is the temperature hyperparameter, and σ is the similarity function (typically cosine similarity). x_i^+ is the positive instance of the i-th training sample, while x_i^- is the hard negative of the i-th training sample. $\mathcal{L}_{emb1}$ represents the embedding loss when no hard negative is available for each training sample, while $\mathcal{L}_{emb2}$ corresponds to the embedding loss when a hard negative instance is present for each sample.

For generation/understanding, we adopt the traditional next-token cross-entropy loss:

$$\mathcal{L}_{gen} = -\frac{1}{N}\sum_{i=1}^{N}\log P(f_{\theta,\eta}(x^{(i)})|f_{\theta,\eta}(x^{(<i)})) \tag{10}$$

where η is the language modeling head used for generation-based tasks. In the second-stage training, we first use $\mathcal{L}_1 = \mathcal{L}_{emb1} + \mathcal{L}_{gen}$ as the loss function, then switch to $\mathcal{L}_2 = \mathcal{L}_{emb2} + \mathcal{L}_{gen}$.

H Hyperparameters

All experiments are conducted on a cluster equipped with eight NVIDIA A800 GPUs, each with 80GB of memory. Tables 9 and 10 present the hyperparameter settings used in the first-stage and second-stage training, respectively.

I Understanding Evaluation Metrics

To evaluate the models' understanding capabilities of RTL code, we apply both traditional machine translation metrics—BLEU (Papineni et al., 2002), ROUGE (Lin, 2004), and METEOR (Banerjee and Lavie, 2005)—which primarily assess lexical similarity, as well as the embedding similarity and GPT score introduced in DeepRTL (Liu et al., 2025), which focus on semantic similarity. These metrics measure the similarity between the generated descriptions and the ground truth summaries.

Specifically, BLEU measures the proportion of n-grams (sequences of n words) in the generated text that also appear in the reference text. It calculates the overlap of n-grams (typically up to a length of 4), with higher scores assigned to more matches. BLEU is precision-focused and rewards the accurate use of words or phrases in the generated descriptions. In our evaluation, we report the smoothed BLEU-4 score to address zero counts in higher-order n-grams, which helps to avoid penalizing models for small discrepancies.

ROUGE is a recall-based metric that evaluates the proportion of n-grams in the reference summary that are present in the generated summary. For our evaluation, we report ROUGE-1 (unigram overlap), ROUGE-2 (bigram overlap), and ROUGE-L (longest common subsequence).

METEOR combines both precision and recall while also accounting for synonymy, stemming, and word order. It computes unigram precision and recall and applies a penalty for word order mismatches. For calculating these traditional machine translation metrics, we directly use the corresponding functions from Python libraries nltk (for BLEU and METEOR) and rouge (for ROUGE).

In contrast to the lexical metrics, embedding similarity and GPT score evaluate semantic similarity by assessing how well the generated descrip-

Model	Average Precision (Main Metric)	Accuracy	F1	Precision	Recall
text-embedding-3-small	0.565	0.581	0.646	0.525	0.840
text-embedding-3-large	0.498	0.544	0.647	0.478	**1.000**
GritLM-7B	0.541	0.613	0.661	0.503	0.960
DeepRTL2$^{no\text{-}hard}$ (Llama)	0.518	0.594	0.661	0.497	0.987
DeepRTL2$^{no\text{-}hard}$ (DeepSeek)	0.481	0.581	0.658	0.497	0.973
DeepRTL2 (Llama)	**0.667**	**0.681**	**0.723**	**0.575**	0.973
DeepRTL2 (DeepSeek)	0.591	0.619	0.708	0.552	0.987

Table 8: The full performance evaluation results for RTL code functionality equivalence checking. The best results among all models are bolded, and the second-best results are underscored.

Hyperparameter Name	Value
finetuning_type	lora
per_device_train_batch_size	4
gradient_accumulation_steps	4
lr_scheduler_type	cosine
warm_up_ratio	0.1
learning_rate	5e-5
epochs	3

Table 9: Hyperparameters selected for the first training stage of DeepRTL2.

Hyperparameter Name	Value
finetuning_type	full
per_device_embedding_batch_size	4
per_device_generative_batch_size	4
gradient_accumulation_steps	8
lr_scheduler_type	linear
warmup_ratio	0.03
learning_rate	2e-5
epochs	1
temperature (τ)	0.02

Table 10: Hyperparameters selected for the second training stage of DeepRTL2.

tion captures the underlying meaning of the RTL code, rather than focusing solely on surface-level word matches. Embedding similarity computes the cosine similarity between the embeddings of the generated description and the ground truth summary, derived from OpenAI's text-embedding-3-large model. This metric rewards models for producing descriptions that are semantically closer to the reference, even if the wording differs. The GPT score, based on GPT-4o, quantifies the semantic coherence between descriptions by assigning a similarity score between 0 and 1, where 1 indicates perfect alignment. Unlike lexical metrics, the GPT score focuses on semantic accuracy rather than exact word matching. For the prompt used in calculating the GPT score, please refer to DeepRTL.

Together, these metrics offer a comprehensive evaluation of both lexical precision and semantic accuracy, providing a holistic view of the model's understanding of RTL code.

J Full Evaluation Results

J.1 Natural Language Code Search

The full evaluation results for natural language code search are presented in Table 7. Results show that the DeepRTL2 models significantly outperform all baseline models across all metrics. Specifically, DeepRTL2 (Llama) and DeepRTL2 (DeepSeek) achieve F1 scores of 0.463 and 0.453, respectively, surpassing the best baseline model, GritLM-7B, which scores 0.269. The higher precision and recall scores for the DeepRTL2 models indicate that they are more effective at retrieving relevant code snippets based on user queries, highlighting the strength of our dataset and training framework. These results confirm that DeepRTL2 excels in natural language code search, demonstrating its superior ability to handle hardware-specific queries compared to the baseline models.

J.2 Functionality Equivalence Checking

The full evaluation results for RTL code functionality equivalence checking are presented in Table 8. Results show that the DeepRTL2 models outperform all baseline models across all metrics. Specifically, DeepRTL2 (Llama) achieves the highest performance with an average precision score of 0.667, F1 score of 0.723, and accuracy of 0.681. In comparison, the best-performing baseline model, GritLM-7B, achieves an average precision of 0.541, an F1 score of 0.661, and accuracy of 0.613. Moreover, DeepRTL2 (DeepSeek) also shows strong performance, with an average precision of 0.591 and an F1 score of 0.708. The significantly higher precision and recall scores for DeepRTL2 models indicate their superior capability in identifying functionally equivalent RTL code compared to the baseline models. These results confirm that DeepRTL2 excels in functionality equivalence checking, demonstrating its effectiveness in hardware-specific tasks over general-purpose models.