Intelligence In The Fence: Construct A Privacy and Reliable Hardware Design Assistant LLM

Shijie Li
University of Science and Technology of China
Hefei, Anhui, China
shijie_li@mail.ustc.edu.cn

Weimin Fu
Kansas State University
Manhattan, Kansas, USA
weiminf@ksu.edu

Yifang Zhao
University of Science and Technology of China
Hefei, Anhui, China
zhaoyifang@mail.ustc.edu.cn

Xiaolong Guo
Kansas State University
Manhattan, Kansas, USA
guoxiaolong@ksu.edu

Yier Jin
University of Science and Technology of China
Hefei, Anhui, China
jinyier@ustc.edu.cn

Abstract

Large language models (LLMs) have been widely used for code assistance in the software and hardware domains. The trend of training a local model for code generation is growing because of the security concerns of releasing proprietary data to third-party providers. However, due to the lack of high-quality training datasets in the hardware domain, researchers have to rely on commercial LLMs, facing the issue of training data leakage. This paper adheres to the principle of zero data upload to address data privacy concerns. Instead of commercial LLMs, we propose a localized and transparent solution leveraging local LLMs to synthesize data and eliminate data leakage risks. We propose an innovative approach to constructing high-quality data by interpretation. To enable efficient local deployment, we fine-tune compact open-source LLMs. The proposed training process and the new dataset structure help us locally train a hardware design assistant LLM named PrivacyGen. PrivacyGen outperforms GPT-4 in the VerilogEval [21] benchmark and performs similarly to GPT-4 in RTLLM [24] benchmark while exhibiting a significantly smaller model size and lower total cost of ownership.

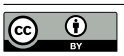
CCS Concepts

• **Hardware** → **Software tools for EDA**; • **Computing methodologies** → *Supervised learning by regression.*

Keywords

Domain-Specific Large Language Model, Verilog Code Generation, Dataset Construction, Data Privacy

ACM Reference Format:
Shijie Li, Weimin Fu, Yifang Zhao, Xiaolong Guo, and Yier Jin. 2025. Intelligence In The Fence: Construct A Privacy and Reliable Hardware Design Assistant LLM. In *Great Lakes Symposium on VLSI 2025 (GLSVLSI '25), June 30-July 2, 2025, New Orleans, LA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3716368.3735172



GLSVLSI '25, New Orleans, LA, USA

ACM ISBN 979-8-4007-1496-2/2025/06
https://doi.org/10.1145/3716368.3735172

1 Introduction

The advent of ChatGPT [26] has broadened the application of LLM intelligence beyond natural language processing to various fields, among which hardware code generation has become a valuable and popular scenario for LLMs. LLMs have been utilized for design verification [14], debugging [13], and generation. Specifically, efforts in hardware design generation can be categorized into three approaches: constructing complex prompts to guide general LLMs [4, 24], building foundational models through pretraining [12], and developing domain-specific LLMs through supervised fine-tuning (SFT) [9, 20–22, 33–35]. However, these approaches face challenges such as high training/fine-tuning costs, security & privacy concerns, and performance issues.

To be precise, constructing prompts to guide general LLMs in design creation faces all these issues despite offering a comprehensive solution. First, in terms of cost, prompt-based methodologies rely on complex prompts to enhance performance. However, these prompts may unintentionally introduce flaws or errors that LLMs may not identify and correct, leading to incorrect hardware design. It takes a lot of human labor to design and check intricate prompts. Additionally, all API usage incurs fees, making the Total Cost of Ownership (TCO) uncontrollable and leading to high costs [31]. Regarding security and privacy, transmitting hardware design parameters and source code in plaintext using commercial LLMs may leak sensitive data. Regarding performance, general LLMs are not explicitly designed for hardware code generation and lack specialized knowledge, often resulting in designs that cannot be synthesized or functionally verified [28].

On the other hand, building foundational models lacks the large-scale hardware domain pretraining dataset to support high-performance models [12]. Building domain-specific LLM via SFT has become a popular solution. For SFT tasks, there are four components: the foundational model (e.g., Llama [2]), the fine-tuning method (e.g., LoRA [17]), the fine-tuning dataset, and the training platform. In hardware-related tasks, since the foundational model, the fine-tuning method, and the training platform are consistent with those used in general LLM training, the primary focus is on

the fine-tuning dataset. However, constructing datasets in the hardware domain is particularly challenging. The high barriers to entry in the hardware field result in much less active open-source communities than their software counterparts. Emulating the software field's approach to building LLMs based on open-source knowledge becomes challenging. Solutions to this dataset problem and their limitations can be categorized into four approaches, each illustrated by typical works below.

- ChipNeMo [20]: NVIDIA possesses a vast hardware dataset as a commercial entity. Although not publicly accessible, it demonstrates the feasibility of implementing hardware domain-specific LLMs.
- VeriGen [33]: Utilizing code from GitHub and textbooks. Due to its unlabeled nature, it can only be a copilot, limiting its usage.
- RTLCoder [22]: Constructing a 27k dataset using randomly generated prompts and ChatGPT-3.5. Similar works in the general domain show poor LLM performance using only synthetic dataset [29]. Relying on online LLMs to build the dataset will increase the exposure of sensitive data. This method also violates OpenAI's restrictions on using ChatGPT-generated data for training, posing legal risks [27].
- Accelerator Generation [25]: Leveraging the enumerable nature of programmable circuits within accelerators, exhaustively listing all potential possibilities. However, it targets only one type of programmable circuit.

To address these shortcomings, we propose PrivacyGen, a specialized hardware code generation assistant that is a **fully localized**, **transparent**, and **open-source** solution. We developed a code interpreter using a local LLM and pre-configured templates to label hardware design datasets through interpretation and translation. This allows us to ensure privacy, avoid restrictions from commercial LLM providers such as OpenAI, and reduce costs. By hierarchically interpreting the code's functionality and implementation, we create a dataset with 25,000 high-quality labeled samples and then fine-tune the open-source LLM, Llama3. We validate the generated PrivacyGen on independent benchmarks, demonstrating state-of-the-art performance among open-source and commercial LLMs. The entire construction process of PrivacyGen can be conducted in a local environment without transmitting any data to external entities. Our main contributions are listed as follows.

- Compared to previous work that relies on commercial LLMs, we propose a privacy and reliable solution for LLM-based hardware design assistants. The framework proposed is localized and transparent, protecting data privacy.
- We propose a novel dataset construction approach using a hierarchical interpretation of hardware designs as labels. This method simplifies dataset creation, enabling offline LLMs for primary automation. Our dataset is based on real hardware data, and we annotate unordered datasets, maintaining the dataset quality.
- We fine-tune Llama-3-8B and Llama-3-70B using our dataset. Specifically, we use quantization fine-tuning for the Llama3 70B model. Benchmark testing shows that our model's performance, particularly on complex designs, is similar to that of GPT-4 and beyond that of other state-of-the-art open-source LLMs. We plan to open-source PrivacyGen to the research community to inspire further research and development in related areas.

2 Background

2.1 LLMs for Hardware Code Generation

Table 1 provides an overview of existing works on the automatic generation of RTL code based on LLMs. It summarizes the comparison between existing works and our work, including methodology, openness, performance of the model, and dependence on commercial models.

Table 1: LLMs for automatic RTL code generation.

Works	Method	Dataset Available	Synthesize Data Based on GPT	Outperform GPT-3.5
Chip-chat [4], RTLLM [24], AutoChip [35]	Prompt Engineering	N/A	N/A	N/A
VeriGen [33, 34]	Fine-tuning	Yes	No	No
RTLcoder [22]	Fine-tuning	Yes	Yes	Yes
VerilogEval [21]	Fine-tuning	No	Yes	Comparable
ChipNeMo [20]	Pretraing, RAG, Fine-tuning	No	No	Comparable
PrivacyGen(Our Work)	Fine-tuning	Yes	No	Yes

Some early works [4, 24, 35] adopted prompt engineering methods based on constructing prompts to guide general LLMs. These studies highlight the significant potential of LLMs in hardware-domain tasks but also raise concerns such as prompt engineering effort, data privacy, general LLM limitations, and cost considerations.

VeriGen [33, 34] collected hardware code from GitHub and textbooks to fine-tune the model directly. It filters the collected files to obtain Verilog files but does not contribute to enhancing the dataset's quality, thereby limiting its application area and performance. While PrivacyGen meticulously processes the raw dataset and generates high-quality labels of functionality description.

ChipNeMo [20] from NVIDIA built their own Verilog training dataset and used Domain-Adaptive Pre-Training, supervised fine-tuning, and retrieval-augmented generation (RAG) methods to enhance the Verilog generation capabilities of LLM. They have comparable performance with GPT-3.5, but their datasets and models are not publicly available. In contrast, PrivacyGen is open-source and outperforms GPT-3.5.

RTLCoder [22] constructed a $27k$ dataset and introduced a new LLM solution for generating RTL code, outperforming GPT-3.5. The synthetic dataset used for training relies on an external commercial model, GPT-3.5. Furthermore, it uses random prompts as labels and entirely synthetic data for training, which limits the diversity of data and model performance. However, PrivacyGen is established locally based on open-source tools under the protection of the firewall. We generate design descriptions using a localized code interpreter rather than commercial models. Moreover, our RTL dataset is derived from actual hardware projects, ensuring data diversity.

2.2 Data Privacy Concerns for Remote LLMs

The use of commercial LLMs in handling client data introduces significant risks to data privacy. When private or proprietary data is used to fine-tune or prompt a commercial LLM, the data will be exposed directly to the LLM provider. Furthermore, LLM providers

```python
from openai import OpenAI
client = OpenAI(api_key="0",base_url="http://0.0.0.0:8000/v1")
messages = [{"role": "user",
             "content": "Here is the sample of sensitive data"}]
result = client.chat.completions.create(messages=messages,
                                        model="meta-llama/Llama-3.2-1B")
```

(a) Client Making an API Request

```
INFO:     Started server process [3327070]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
[INFO|2025-03-04 13:19:40] >> ==== request ====
{
  "model": "meta-llama/Llama-3.2-1B",
  "messages": [
    {
      "role": "user",
      "content": "Here is the sample of sensitive data"
    }
  ]
}
INFO:     127.0.0.1:50098 - "POST /v1/chat/completions HTTP/1.1" 200 OK
```

(b) Backend Revealing Client Input

Figure 1: Client Input Exposure in Remote LLM Backend

can use the data input by clients to improve their models, which can pose a greater risk to the data of clients. Samsung Electronics has imposed a ban on the use of generative AI tools like ChatGPT, Google Bard, and Bing AI by its employees. The move follows an internal incident where an employee allegedly uploaded confidential source code to ChatGPT. Many other tech companies, including Apple, JPMorgan Chase, and Amazon, have also implemented similar restrictions to protect their intellectual property.

When utilizing the Web services or APIs of commercial LLMs, clients must send their queries to external providers. Regardless of the security measures implemented, LLMs inherently require plaintext input to process requests, and their output inevitably includes the original input content. As illustrated in Figure 1, when invoking an LLM deployed via Huggingface backend, the original input text remains visible in the system's backend.

Cloud service providers are increasingly supporting the remote deployment of LLMs, enabling these models to be hosted on cloud platforms where client-provided data is processed and stored. A critical concern associated with this approach is the potential lack of absolute data confidentiality. Although many cloud providers assert the implementation of robust security measures, there remains a risk of unauthorized access to sensitive data by internal actors, whether due to malicious intent or negligence. Additionally, inherent vulnerabilities within the cloud infrastructure may further expose user data to external threats, posing significant security risks.

2.3 Private LLM Inference

Fully Homomorphic Encryption (FHE) is a promising solution that enables computation on encrypted data without revealing the plaintext and only outputs the encrypted result. This property makes it suitable for LLM inference, ensuring that cloud service providers cannot access the plaintext of clients' data. CryptoNets [15] has successfully applied FHE to neural networks. However, existing FHE schemes, such as BGV [23] and CKKS [6], face significant challenges in LLM inference [7]:

- The computation of LLMs is already very expensive. However, FHE operations are significantly slower than their plaintext counterparts, leading to substantial computational overhead.
- FHE primarily supports addition and multiplication, whereas transformer networks require nonlinear transformations like GELU, softmax, and LayerNorm, which necessitate complex homomorphic approximations, leading to trade-offs in both efficiency and accuracy. [6]

Secure Multi-Party Computation (SMPC) enables multiple participants to collaboratively compute a function's output without revealing their individual input data. The input data is partitioned into multiple shares, with each participant holding a portion. In the context of large language model (LLM) inference, SMPC can be utilized to ensure that user input data remains hidden from the model provider. However, SMPC also faces several challenges:

- SMPC operations, such as secure additions and multiplications via secret sharing or garbled circuits, are much slower than plaintext operations. Additionally, secret sharing protocols require communication for each operation, making them $10\times-1000\times$ slower, depending on the protocol and network conditions [11].
- Certain complex nonlinear operations, such as activation functions and Softmax, are computationally inefficient within the SMPC framework [18].

Trusted Execution Environments (TEEs) provide a secure enclave for executing computations while ensuring confidentiality and integrity, making them a promising solution for deploying LLMs in untrusted environments. By isolating model execution from the host system and cloud provider, TEEs can protect user inputs and prevent unauthorized access to model parameters. However, it faces several critical challenges. First, TEEs have stringent memory and computational constraints, making it difficult to accommodate the large-scale operations required for LLM inference. Second, the lack of native support for hardware acceleration, such as GPUs or TPUs, significantly degrades the efficiency of neural network computations within TEEs [16].

2.4 Local LLM Deployment

With the open-sourcing of DeepSeek-R1 [8], the LLMs as powerful as GPT-o1 can be deployed locally, providing greater control over data privacy. However, deploying a large-size model poses substantial computational challenges. For example, deploying DeepSeek R1 671B for local inference requires approximately 1,500 GB of GPU memory. It renders such deployments impractical for users with constrained infrastructure. As a result, achieving an optimal balance between model size, computational efficiency, and deployment feasibility has become a crucial factor in the adoption of locally hosted LLMs. A small-scale local LLM is the most practical approach at present. Although small-scale models exhibit inferior overall performance, their capabilities in specific domains can be enhanced

to achieve performance comparable to or even surpassing that of large-scale general models.

2.5 Hardware Private Data

Unlike software, hardware designs typically require significant investment in research, verification, and fabrication, making their confidentiality paramount. Engineers and organizations want to leverage LLMs to assist and accelerate the hardware development process while ensuring that proprietary hardware data remains protected. A viable solution is to train and deploy an LLM in a secure, on-premises environment using private datasets. By conducting model training and inference locally, companies can maintain full control over data access and mitigate the risk of information leakage associated with cloud-based or third-party LLM services. Additionally, fine-tuning LLMs on domain-specific hardware design data enhances their ability to generate high-quality HDL code, optimize circuits, and automate verification tasks.

Since we do not have a private hardware dataset, we use open-source data to simulate the private data. We gather 182, 964 hardware designs from 4, 704 GitHub and OpenCore projects. In the following sections, we refer to these designs as the private dataset.

3 Methodology

Fig. 2 illustrates the overall structure of the PrivacyGen construction process. The proposed framework can operate entirely offline, ensuring that all processes, including dataset construction, model training, and model usage, are conducted inside the firewall.

3.1 Data Cleaning

As shown in process ❶, we preprocess the private dataset to construct a high-quality RTL code dataset. The preprocessing begins with deduplication, where each code snippet in the private dataset is tokenized, and a SHA-1 hash of the resulting token sequence is computed while disregarding whitespace and comments. Code snippets with identical hash values are identified as duplicates and subsequently removed. Following this deduplication process, we obtained a total of 68, 567 unique designs.

On the other hand, hardware designs include macros and cross-references. If the model assimilates these designs, the model may use instances of unknown modules when generating a hardware design. To ensure each design in the dataset is independent, we use Yosys [36]. A customized script is developed to perform abstract syntax tree construction and code generation, converting the hardware designs into an RTLIL intermediate representation. Macros and cross-references are then expanded into their target locations. Additionally, Yosys filters out syntactically incorrect designs and removes comments to ensure dataset quality. Consequently, the script utilizes the Yosys Verilog backend to export the design in Verilog. The size of the dataset was further reduced due to Yosys's limited language support, which cannot support all designs in the raw dataset.

After preprocessing, the dataset comprises 25, 199 unique hardware designs.

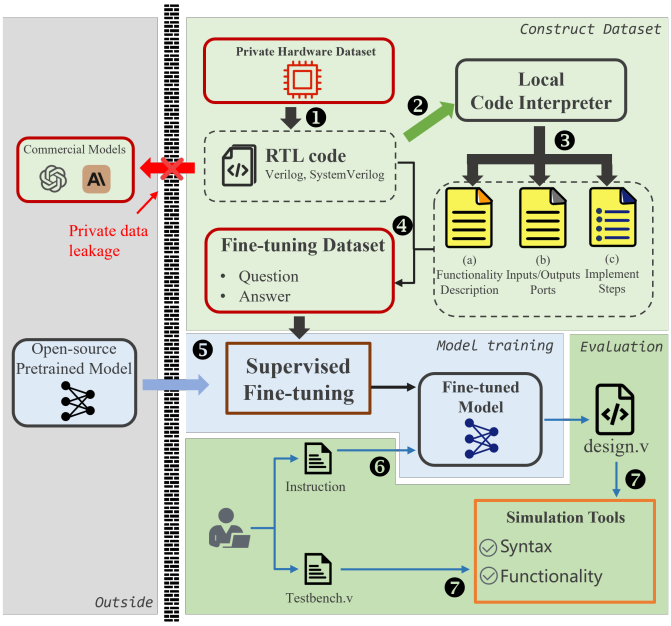


Figure 2: The Overall Framework of Constructing PrivacyGen.

3.2 Fine-tuning Dataset Constuction

The RTL code dataset alone cannot directly support LLM fine-tuning. To build an SFT dataset, we need to follow these steps: constructing inputs for the local LLM ❷, including hardware designs and detailed hierarchical prompts; organizing the three different outputs into a comprehensive raw dataset ❸; and building the fine-tuning dataset based on these results ❹.

To obtain a simple, yet comprehensive description of functionality, steps ❷ and ❸ accomplish the following tasks. First, they provide a high-level overview of the hardware design's functionality in natural language. Second, they specify the I/O ports of the hardware design, ensuring clear communication with other modules. Third, they offer a detailed natural language description of the hardware design process, including step-by-step explanations, algorithmic details, and design considerations. This comprehensive approach aids in understanding the purpose, operation, and internal structure of complex hardware designs.

The quality of data generated based on LLM is directly related to the performance of the LLM and the prompt design. Although the local LLM underperforms commercial models such as ChatGPT due to model size and training data scale, we still deploy open-source LLM locally for interpreting code to ensure that the entire process is carried out locally with the protection of the firewall. Compared to generating hardware design from design description (such as RTLCoder), summarizing functional description from code is easier, which local LLMs are competent for. Our designed prompt guides the LLM to perform multi-level analysis for a more accurate understanding of code functionality, bridging the performance gap.

We employed the Nemotron-4-340B Instruct [1], which is the best open model for synthetic data generation originating from NVIDIA. A typical prompt is shown in Fig. 3, which instructs Nemotron

[prompt]
Analyze the provided Verilog code to determine:
1. Circuit Functionality: Explain the purpose and overall function of the circuit described by the code.
2. Module Name and Ports: Identify and describe the module name, input, output, and inout ports, including their data types and significance.
3. Implementation Steps: Summarize the main processes, modules, or components in the code and how they interact to achieve the desired functionality.
<Insert Verilog code here>

```
module sqrt_pipelined
(
  input clk,
  input reset_n,
  input start,
  input [INPUT_BITS-1:0] radicand,
  output reg data_valid,
  output reg [OUTPUT_BITS-1:0] root
);
  /* Collapse for Space */
endmodule
```

Figure 3: Example of a Pipelined Square Root Module

to achieve the three tasks step by step according to the hardware design code.

To construct a comprehensive LLM fine-tuning dataset ❹, we create a dataset made up of Question-Answer pairs, where the "Question" serves as the input instructing the LLM to generate the hardware design, and the "Answer" is the Verilog implementation of the hardware design. In our fine-tuning dataset, Questions consist of the Functionality Description and I/O Ports. Meanwhile, Answers include the Implementation Steps and the preprocessed raw hardware design obtained in Subsection 3.1.

3.3 Fine-tuning and Evaluation

In Section 3.2, we build the fine-tuning dataset. We choose open-source pre-trained LLM as the base model and LoRA as the fine-tuning technique, which is more efficient and saves a lot of computing resources compared to Full Fine-tuning. To preserve the base model's performance [3], we selectively fine-tuned only the *Queries* and *Values* matrices within the self-attention blocks while keeping the *Key* matrices and other parameters unchanged.

For the evaluation, we utilize simulation to verify the syntax and functionality correctness of LLM's response. As shown in ❻ and ❼, the validation relies on LLM input and testbench design. So, for the sake of fairness and convenience of data comparison, we chose to use the testbench built by VerilogEval [21] and RTLLM [24].

4 Experiment

This section details the construction of the fine-tuning dataset. Next, two Llama models are fine-tuned using this dataset. Experiments are then set up to evaluate their performance.

4.1 Dataset Construction

We construct a 25k high-quality fine-tuning dataset following the process ❹. Fig. 3 4 5 show an example of generating data from the raw dataset, corresponding to processes ❷ and ❸.

Specifically, the Nemotron-4-340B Instruct is deployed locally, and prompt templates are applied. This AI system serves as a code interpreter, analyzing the code from the raw dataset.

For example, Fig. 3 shows a square root calculation module from the preprocessed raw dataset, along with the prompt template to instruct the code interpreter. Fig. 4 5 shows the Functional Description(1), Input/Output ports(2) and Implementation Steps(3).

1. Circuit Functionality
This is a pipelined square root computation circuit. It computes the integer square root of a given input (`radicand`) using an iterative subtraction-based algorithm. It incorporates a multi-stage pipeline to efficiently compute the square root over several clock cycles.

2. Module Name and Ports
Module Name: sqrt_pipelined
Inputs:
- `clk`: Clock signal to synchronize operations. Bit width: 1.
############Collapse for Space#########

Figure 4: Functionality and Ports of Module in Fig. 3

- **Functionality Description**: LLMs tend to interpret code line by line rather than provide high-level semantic information about functionality. But as shown in Fig. 4, with the assistance of developed prompt templates, Nemotron accurately abstracts that the module is a pipelined square root computation circuit.
- **Input and Output Ports**: Extracting ports is easy, but analyzing their role in the design is difficult. In Fig. 4, Nemotron correctly understands all port functionalities and identifies the association between the input/output port (radicand, root) and constant parameters based on internal design information.
- **Implementation**: Analyzing the implementation process requires LLM to be able to reasonably segment the code and summarize the function of each code snippet. Fig. 5 shows that Nemotron uses a hierarchical structure to detail the hardware design's conceptual elements in natural language. It describes the module implementation process and the adopted algorithm in five steps.

In total, the dataset contains 25, 199 items and 22, 124, 325 tokens.

4.2 Model Training

We selected Llama-3-8B and Llama-3-70B [2], which boasts advanced general capabilities and programming performance, as the foundational model for finetuning. Their context length is 8, 192, which is sufficient to accommodate all data instances in our dataset. We set the truncation length to 4, 096 in our fine-tuning.

The hardware, software, LoRA parameters, and deep learning optimizer configuration of the fine-tuning experimental platform are detailed in Table 2. Additionally, we employed DeepSpeed ZeRO-3 [30] to implement multi-GPU parallelism, distributing segments of model parameters, optimizer state, and gradients to three GPUs. For the Llama-3-8B model, we used BF16 mixed-precision for fine-tuning, which takes up 12GB of VRAM per GPU (36GB). For the

Table 2: Hardware and Software Platforms and Training Configuration

Hardware Platform			Software Platform			LoRA Configuration				Training Optimizer: Adam		
CPU	Memory	GPU	OS	Pytorch	Transformers	R	α	dropout	Learning Rate Warmup	β	ϵ	learning rate
Intel(R) Xeon(R) Silver 4310	512GB	3* Nvidia A100 40	Ubuntu 18.04.6 LTS	2.3.0	4.40.2	8	16	0.05	100 steps	(0.9, 0.999)	$1e-8$	$1e-4$

3. Implementation Steps
1. Initialization and Parameterization:
- `OUTPUT_BITS` is derived from `INPUT_BITS` to ensure the correct bit width for the square root.
##########Collapse for Space########
2. Pipeline Stage Implementation:
- The design uses a `generate` block to instantiate multiple pipeline stages (`OUTPUT_BITS - 1` stages).
##########Collapse for Space########
3. Mask Generation:
- A `genvar` loop generates masks for each stage:
##########Collapse for Space########
4. Control Logic:
- The `start_gen` register propagates the `start` signal through the pipeline to synchronize operations.
##########Collapse for Space########
5. Output Assignment:
- `data_valid` is set when the pipeline computation completes, based on the final stage of `start_gen`.
##########Collapse for Space########

Figure 5: Implement Steps of Module in Fig. 3

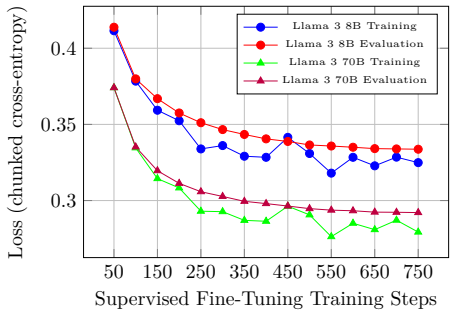


Figure 6: Training and Validation Loss During Supervised Fine-Tuning Training

70B version, we opt for quantization [10] for fine-tuning, mapping BF16 mixed-precision parameters to normalized float 4-bit (NF4) to optimize computational efficiency, only consuming 20GB of VRAM per GPU (a total of 60GB). We trained them for 3 epochs on the dataset and fixed the batch size at 96. Our platform can train models at a speed of 0.854 samples(Llama-3-8B) and 0.361 samples(Llama-3-70B-4bit) per second, and it took a total of 1 day, 19 minutes, 47 seconds and 2 days, 9 hours, 33 minutes, 11 seconds, respectively.

Fig. 6 presents the loss variations on the training and test sets during training. The loss represents the discrepancy between predicted outputs and actual values. The decrease in loss during training represents LLMs assimilating the information provided by the fine-tuning dataset. A plateau in the loss after 550 steps indicates that further training may not significantly improve the model's performance on unseen data. Despite employing quantization techniques, the 70B model consistently outperformed the 8B model.

4.3 Experiment Setup

We compare our fine-tuned models' performance on two representative benchmarks, VerilogEval [21] and RTLLM [24], with prior works. VerilogEval consists of two parts: Eval-Human with 156 questions and Eval-Machine with 143 questions. VerilogEval measures the Verilog generation accuracy using the *pass@k* [5] metric as shown in Equation 1.

$$pass@k = \frac{1}{N}\sum_{i=1}^{N}\left(1 - \frac{C_{n-c_i}^{k}}{C_n^k}\right) \quad (1)$$

where N is the total number of problems, n is the number of code candidates generated for each problem, and c_i is the number of candidates for the i-th problem that pass syntax or functionality checks. In our experiment, we set $n = 20$ and $k = \{1, 5, 10\}$. For hyperparameters, we set $top_p = 0.9$ and select the best performance among $temperature = \{0.2, 0.5, 0.8\}$.

RTLLM contains 29 RTL generation problems. Following the instructions, we generate five candidate Verilog codes for each problem and report the correctness rate. The problem is considered passed if at least one trial passes the syntax and function check. We also use the same version of Synopsys-VCS [32] with RTLCoder and the evaluation method mentioned in Subsection 3.3 in our experiments.

To evaluate the model's generalization ability on unseen data, we remove the data similar to the benchmarks. We check the similarity between the samples in our dataset and the test cases in the benchmarks based on the Rouge-L metric and remove the samples with Rouge-L greater than 0.5 from the dataset.

5 Experimental Results

5.1 PrivacyGen's evaluation on the benchmarks

Table 3 summarizes the comparison results of PrivacyGen and baselines on VerilogEval and RTLLM v1.1 benchmarks. The baselines include (1) closed-source or commercial models: GPT-3.5, GPT4, ChipNeMo [20], and VerilogEval[21]; (2) open-source models: VeriGen [34], Starcoder [19], RTLCoder [22], and our foundational models Llama-3-8B and 4-bit quantized Llama-3-70B.

In the VerilogEval benchmark, PrivacyGen-Llama3-70B-4bit outperforms all other methods, and PrivacyGen-Llama3-8B performs close to RTLCoder. PrivacyGen-Llama3-70B-4bit achieves 69.2% in Eval-machine and 49.1% in Eval-human in *pass*@1, significantly outperforming GPT-4. They are 9.2% and 5.6% higher than GPT-4, which has the best performance in baselines.

In the RTLLM benchmark, PrivacyGen-Llama3-8B fails one more case than RTLcodeer, but PrivacyGen-Llama3-70B-4bit achieves

Table 3: Comparison of PrivacyGen with baseline models in the VerilogEval [21] benchmark and RTLLM [24] v1.1 benchmark.

Model Type	Model	Params	VerilogEval [21](using *pass@k*)						RTLLM v1.1 [24] (using 5-*shot*)	
			Eval-Machine(%)			Eval-Human(%)				
			k=1	k=5	k=10	k=1	k=5	k=10	Syntax-VCS(%)	Func(%)
Closed-Source	GPT-3.5	175B	46.7	69.1	74.1	26.7	45.8	51.7	89.7	37.9
	GPT4	N/A	60	70.6	73.5	43.5	55.8	58.9	**100**	65.5
	ChipNeMo[20]	13B	43.4	N/A	N/A	22.4	N/A	N/A	N/A	N/A
	VerilogEval[21]	16B	46.2	67.3	73.7	28.8	45.9	52.3	N/A	N/A
Open-Source	Verigen[33]	16B	44.0	52.6	59.2	30.3	43.9	49.6	86.2	24.1
	Starcoder[19]	15B	46.8	54.5	59.6	18.1	26.1	30.4	93.1	27.6
	RTLCoder[22]	7B	62.5	72.2	76.6	36.7	45.5	49.2	96.6	48.3
Foundational Model	Llama-3-8B	8B	45.9	60.7	65.7	23.6	34.8	38.7	79.3	27.6
	Llama-3-70B-4bit	70B*4bit	65.5	71.7	74.2	41.2	46.7	48.7	86.2	51.7
PrivacyGen	PrivacyGen-Llama3-8B	8B	60.6	71.4	77.4	35.8	46.6	51.7	89.7	44.8
	PrivacyGen-Llama3-70B-4bit	70B*4bit	**69.2**	**80.4**	**86.5**	**49.1**	**59.6**	**63.3**	96.6	**72.4**

Table 4: Performance comparison of PrivacyGen, Llama-3-70B-4bit, GPT-3.5, GPT-4, VeriGen, StarCoder, RTLcoder on problems of different difficulty levels in the RTLLM benchmark. The first and second highest scores are marked in blue and green.

Difficulty	Design	Llama-3-70B-4bit		PrivacyGen		GPT-3.5		GPT4		VeriGen[33]		StarCoder[19]		RTLCoder[22]	
		Syntax	Func	Syntax	Func	Syntax	Func	Syntax	Func	Syntax	Func	Syntax	Func	Syntax	Func
Rookie	adder_8bit	5	P	5	P	3	P	4	P	3	P	2	NP	5	P
	calendar, counter_12, edge_detect, pe														
	right_shifter	5	P	5	P	4	P	5	P	0	N/A	3	P	5	P
	Average	100%	100%	100%	100%	100%	83.3%	100%	100%	83.3%	50.0%	100%	83.3%	100%	83.3%
Crafter	adder_16bit	5	P	5	P	1	NP	3	P	3	NP	2	NP	3	NP
	alu, freq_div, JC_counter, multi_16bit, multi_booth_8bit, parallel2serial, RAM, synchronizer, serial2parallel, signal_generator														
	width_8to16	5	P	5	P	4	P	5	P	4	P	3	NP	5	P
	Average	83.3%	66.7%	91.7%	91.7%	100%	50.0%	100%	66.7%	100%	33.3%	100%	25.0%	100%	66.7%
Disciple	accu	5	NP	1	P	2	P	5	P	4	NP	3	NP	4	NP
	adder_32bit(CLA), adder_pipe_64bit, asyn_fifo, multi_pipe_4bit, multi_pipe_8bit, div_8bit, div_16bit, fsm, pulse_detect														
	traffic_light	0	NP	1	NP	4	NP	4	P	5	N/A	5	N/A	4	P
	Average	54.5%	0%	100%	36.3%	72.7%	9.09%	100%	45.4%	72.7%	0%	81.8%	0%	90.9%	18.2%
Total		75.9%	51.7%	96.6%	72.4%	89.7%	37.9%	100%	65.5%	86.2%	24.1%	93.1%	27.6%	96.6%	48.3%

state-of-the-art performance in functionality accuracy and is slightly lower than GPT-4 in syntax accuracy. Table 4 shows detailed results on problems of different difficulty levels in RTLLM. PrivacyGen's average accuracy on Disciple problems is much higher than the foundational model and is very close to GPT-4. On Crafter problems, PrivacyGen achieves state-of-the-art performance. These results show that the Verilog generation capability of LLM is significantly enhanced through fine-tuning on our dataset.

5.2 Data leakage Evaluation

As described in Section 1, our approach does not threaten data privacy since it does not transmit any data externally. Table 5 intuitively estimates the upload and download traffic generated by the existing LLM-based automatic hardware generation solutions during their entire process, where only the traffic of the best model is counted. Constructing PrivacyGen generates no upload traffic.

6 Conclusion

This work proposes a localized, transparent, LLM-based solution PrivacyGen for automatic hardware design generation. The entire workflow does not rely on any commercial LLM and is executed in a local and completely controllable environment. It also contributes a new approach to generating high-quality datasets through the code explanation method. Through low-cost fine-tuning experiments, we construct a model with performance close to GPT-4 on complex designs of the RTLLM benchmark and better than GPT-4 on the

Table 5: Upload and download traffic statistics

Works	Download		Upload
	Dataset	Model	Sensitive Data
[4],[24],[35]	N/A	N/A	100%
VerilogEval	300MB	32GB	10MB
VeriGen	300MB	32GB	0
RTLCoder	0	14GB	55.1 MB
PrivacyGen	1.6GB	141GB	0

VerilogEval benchmark. PrivacyGen will be open-sourced and our approach allows for generating your own dataset to construct an LLM solution under complete firewall protection and transparency.

Acknowledgments

Portions of this work were supported by the National Science Foundation (2340949 and 2419880).

References

[1] Bo Adler, Niket Agarwal, et al. 2024. Nemotron-4 340B Technical Report. *CoRR* abs/2406.11704 (2024), 34. https://doi.org/10.48550/ARXIV.2406.11704 arXiv:2406.11704

[2] AI@Meta. 2024. Llama 3 Model Card. https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[3] Dan Biderman, Jose Javier Gonzalez Ortiz, et al. 2024. LoRA Learns Less and Forgets Less. *CoRR* abs/2405.09673 (2024), 39. https://doi.org/10.48550/ARXIV.2405.09673 arXiv:2405.09673

[4] Jason Blocklove, Siddharth Garg, et al. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. In *5th ACM/IEEE Workshop on Machine Learning for CAD, MLCAD 2023, Snowbird, UT, USA, September 10-13, 2023*. IEEE, Snowbird, UT, USA, 1–6. https://doi.org/10.1109/MLCAD58807.2023.10299874

[5] Mark Chen, Jerry Tworek, et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021), 35. arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[6] Tianyu Chen, Hangbo Bao, et al. 2022. The-x: Privacy-preserving transformer inference with homomorphic encryption. *arXiv preprint arXiv:2206.00216* (2022).

[7] Leo de Castro, Antigoni Polychroniadou, et al. 2024. Privacy-Preserving Large Language Model Inference via GPU-Accelerated Fully Homomorphic Encryption. In *Neurips Safe Generative AI Workshop 2024*. NeurIPS Workshop, Vancouver, Canada, 19. https://openreview.net/forum?id=Rs7h1od6ov

[8] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:cs.CL/2501.12948 https://arxiv.org/abs/2501.12948

[9] Matthew DeLorenzo, Animesh Basak Chowdhury, et al. 2024. Make Every Move Count: LLM-based High-Quality RTL Code Generation Using MCTS. *CoRR* abs/2402.03289 (2024), 7. https://doi.org/10.48550/ARXIV.2402.03289 arXiv:2402.03289

[10] Tim Dettmers, Artidoro Pagnoni, et al. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, et al. (Eds.). Curran Associates, Inc., New Orleans, LA, USA, 28. http://papers.nips.cc/paper_files/paper/2023/hash/1feb87871436031bdc0f2beaa62a049b-Abstract-Conference.html

[11] Xiaoyu Fan, Kun Chen, et al. 2022. Nfgen: Automatic non-linear function evaluation code generator for general-purpose mpc platforms. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 995–1008.

[12] Weimin Fu, Shijie Li, et al. 2024. Hardware Phi-1.5B: A Large Language Model Encodes Hardware Domain Specific Knowledge. In *Proceedings of the 29th Asia and South Pacific Design Automation Conference (ASPDAC '24)*. IEEE Press, Incheon, Republic of Korea, 349–354. https://doi.org/10.1109/ASP-DAC58780.2024.10473927

[13] Weimin Fu, Shijie Li, et al. 2025. A Generalize Hardware Debugging Approach for Large Language Models Semi-Synthetic, Datasets. *IEEE Transactions on Circuits and Systems I: Regular Papers* 72, 2 (2025), 623–636. https://doi.org/10.1109/TCSI.2024.3487486

[14] Weimin Fu, Kaichen Yang, et al. 2023. LLM4SecHW: Leveraging Domain-Specific Large Language Model for Hardware Debugging. In *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. 1–6. https://doi.org/10.1109/AsianHOST59942.2023.10409307

[15] Ran Gilad-Bachrach, Nathan Dowlin, et al. 2016. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*. PMLR, 201–210.

[16] Husheng Han, Xinyao Zheng, et al. 2024. TensorTEE: Unifying Heterogeneous TEE Granularity for Efficient Secure Collaborative Tensor Computing. *arXiv preprint arXiv:2407.08903* (2024).

[17] Edward J. Hu, Yelong Shen, et al. 2021. LoRA: Low-Rank Adaptation of Large Language Models. *CoRR* abs/2106.09685 (2021). arXiv:2106.09685 https://arxiv.org/abs/2106.09685

[18] Haoran Li, Yulin Chen, et al. 2023. Privacy in large language models: Attacks, defenses and future directions. *arXiv preprint arXiv:2310.10383* (2023).

[19] Raymond Li, Loubna Ben Allal, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[20] Mingjie Liu, Teodor-Dumitru Ene, et al. 2023. Chipnemo: Domain-adapted llms for chip design. *arXiv preprint arXiv:2311.00176* (2023).

[21] Mingjie Liu, Nathaniel Pinckney, et al. 2023. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–8.

[22] Shang Liu, Wenji Fang, et al. 2023. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. *arXiv preprint arXiv:2312.08617* (2023).

[23] Xuanqi Liu and Zhuotao Liu. 2023. Llms can understand encrypted prompt: Towards privacy-computing friendly transformers. *arXiv preprint arXiv:2305.18396* (2023).

[24] Yao Lu, Shang Liu, et al. 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)*. IEEE, 722–727.

[25] Mahmoud Nazzal, Deepak Vungarala, et al. 2024. A Dataset for Large Language Model-Driven AI Accelerator Generation. *arXiv preprint arXiv:2404.10875* (2024).

[26] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/ARXIV.2303.08774 arXiv:2303.08774

[27] OpenAI. 2024. OpenAI Business Terms. https://openai.com/policies/business-terms/

[28] Hammond Pearce, Baleegh Ahmad, et al. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[29] Nazneen Rajani, Lewis Tunstall, et al. 2023. No Robots. https://huggingface.co/datasets/HuggingFaceH4/no_robots.

[30] Samyam Rajbhandari, Olatunji Ruwase, et al. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–14.

[31] Amit Sharma, Teodor-Dumitru Ene, et al. 2024. Assessing Economic Viability: A Comparative Analysis of Total Cost of Ownership for Domain-Adapted Large Language Models versus State-of-the-art Counterparts in Chip Design Coding Assistance. *arXiv preprint arXiv:2404.08850* (2024).

[32] Synopsys. 2021. VCS® functional verification solution. https://www.synopsys.com/verification/simulation/vcs.html

[33] Shailja Thakur, Baleegh Ahmad, et al. 2023. Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[34] Shailja Thakur, Baleegh Ahmad, et al. 2024. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems* 29, 3 (2024), 1–31.

[35] Shailja Thakur, Jason Blocklove, et al. 2023. Autochip: Automating hdl generation using llm feedback. *arXiv preprint arXiv:2311.04887* (2023).

[36] Clifford Wolf, Johann Glaser, et al. 2013. Yosys-A Free Verilog Synthesis Suite. https://api.semanticscholar.org/CorpusID:202611483