BetterV: Controlled Verilog Generation with Discriminative Guidance

Zehua Pei [1] **Hui-Ling Zhen** [2] **Mingxuan Yuan** [2] **Yu Huang** [2] **Bei Yu** [1]

Abstract

Due to the growing complexity of modern Integrated Circuits (ICs), there is a need for automated circuit design methods. Recent years have seen increasing research in hardware design language generation to facilitate the design process. In this work, we propose a Verilog generation framework, BetterV, which fine-tunes large language models (LLMs) on processed domain-specific datasets and incorporates generative discriminators for guidance on particular design demands. Verilog modules are collected, filtered, and processed from the internet to form a clean and abundant dataset. Instruct-tuning methods are specially designed to fine-tune the LLMs to understand knowledge about Verilog. Furthermore, data are augmented to enrich the training set and are also used to train a generative discriminator on particular downstream tasks, providing guidance for the LLMs to optimize Verilog implementation. BetterV has the ability to generate syntactically and functionally correct Verilog, outperforming GPT-4 on the VerilogEval benchmark. With the help of task-specific generative discriminators, BetterV achieves remarkable improvements on various electronic design automation (EDA) downstream tasks, including netlist node reduction for synthesis and verification runtime reduction with Boolean Satisfiability (SAT) solving.

1. Introduction

Large language models (LLMs) are leading the world due to their strong capability in generating and understanding natural language at a massive scale, which makes their potential applications and benefits vast across various domains and tasks. In the field of coding, LLMs can act as seasoned assistants for developers by providing advice on programming, finding and fixing bugs in code, and even generating entire codebases from descriptions (Chen et al., 2021; Nijkamp et al., 2023).

Electronic design automation (EDA) is a set of software and services for designing integrated circuits (ICs), which work together in the design flow to conceive and analyze circuit designs. The slowing down of Moore's Law puts increasing pressure on EDA, creating an emergent need to further improve and automate the design flow. Therefore, it is expected that LLMs will be incorporated into the EDA flow. LLMs in EDA have already achieved remarkable success in various tasks (He et al., 2023; Liu et al., 2023a).

Hardware design languages (HDLs), such as Verilog and VHDL, describe the hardware design at the very beginning of the design flow, playing an important role in the EDA flow and strongly influencing the subsequent stages. However, writing HDL is time-consuming and prone to bugs, making it more expensive for today's complex ICs. Therefore, it is promising to utilize LLMs to automatically generate the desired HDL. Recently, several works have focused on Verilog generation (Dehaerne et al., 2023; Thakur et al., 2023). However, these works primarily refine LLMs with selected datasets, neglecting the direct integration of syntactic or functional correctness in the model's development. Moreover, they insufficiently address pivotal downstream tasks in the EDA process, which should be crucial evaluations for the generated Verilog.

Despite the expectation that LLMs can play an important role in Verilog generation, several challenges need to be addressed. Firstly, the complex and strict requirements of hardware designs restrain LLMs from learning and understanding the knowledge related to Verilog. Secondly, there are limited Verilog resources available globally, which often leads to problems of overfitting and data bias during LLM fine-tuning. Moreover, considering the complicated and varied downstream tasks further complicates the issue. In industrial production, optimizing downstream targets from Verilog design to physical implementation involves numerous iterations and cannot be accomplished in a single step. This makes it difficult for LLMs to anticipate future stages and impractical to fine-tune the LLMs for each downstream task.

Controllable text generation involves techniques that train extra discriminators to guide LLMs in desired directions (Scialom et al., 2020). However, these works primarily

[1]The Chinese University of Hong Kong, Hong Kong SAR [2]Noah's Ark Lab, Huawei, Hong Kong SAR. Correspondence to: Bei Yu <byu@cse.cuhk.edu.hk>.

Proceedings of the 41^{st} International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

focus on controlling natural language text representation during their development. In this work, we explore the potential of using this technique for optimization tasks, which is more challenging and fundamentally different from merely handling text representation.

In this work, we propose a framework, BetterV, for Verilog generation by instruct-tuning LLMs on our processed datasets and incorporating generative discriminators to optimize Verilog implementation for various downstream tasks. We utilize the alignment between Verilog and C programs to help LLMs effectively understand Verilog. Additionally, we propose data augmentation to address the issue of limited Verilog data. Furthermore, we recognize that various Verilog implementations can differ significantly in terms of Power, Performance, and Area (PPA) or verification runtime. Therefore, we transform the problems in downstream tasks into optimization challenges for Verilog implementation. The generative discriminator guides the LLMs to generate or modify Verilog implementations directly from natural language, making them as efficient as possible for downstream tasks and effectively reducing the number of iterations in industrial production. With these techniques, BetterV successfully imparts domain-specific knowledge to LLMs and can adapt to any Verilog-related downstream tasks.

The contributions of this paper are summarized as follows:

- BetterV represents a groundbreaking development as the first endeavor to apply controllable text generation to engineering optimization challenges, specifically in optimizing downstream tasks in Electronic Design Automation (EDA). This approach not only introduces an innovative and promising research trajectory in EDA but also holds potential for application in optimization issues across various other domains.

- BetterV marks a pioneering advancement as the first downstream task-driven method for Verilog generation. Our experiments employing various generative discriminators on specific Electronic Design Automation (EDA) tasks have demonstrated notable effectiveness. This innovative approach is characterized by its task-specific discriminator guidance, enhancing both its training efficiency and practical utility.

- Utilizing fine-tuned 6.7B/7B-parameter LLMs, without the application of prompt-engineering strategies, BetterV demonstrates the capacity to generate syntactically and functionally correct Verilog, which surpasses GPT-4 when evaluated on the VerilogEval benchmark.

- BetterV offers a versatile solution for data augmentation, tailored to meet diverse specifications in Verilog implementation. This approach addresses the issue posed by the scarcity of Verilog resources effectively.

2. Related Works

In this section, we briefly introduce some advancements and applications of LLMs for Verilog generation in Section 2.1. We also discuss the development of discriminator-guided controllable generation in Section 2.2.

2.1. LLMs for Verilog Generation

Large language models (LLMs) have shown remarkable performance on code generation with either training a model from the beginning or fine-tuning a pre-trained model (Nijkamp et al., 2023; Roziere et al., 2023). The success of LLMs on code generation arouses the interest of study of LLMs on hardware design. As a widely recognized hardware description language (HDL), the generation of Verilog using LLMs has garnered significant attention and undergone extensive exploration. Some studies pay attention to construct customized datasets for the fine-tuning of LLMs, such as (Thakur et al., 2023) and (Dehaerne et al., 2023), who collect Verilog from the internet and process the data prior to training. VerilogEval (Liu et al., 2023b) and RTLCoder (Liu et al., 2023c) further study the importance of problem descriptions and then generate various problem-answer pairs as dataset. RTLCoder also considers the quality feedback on different data by ranking them with scores during the training. Researchers also try utilizing the prompt-engineering techniques to enhance the generation ability, such as the self-planning used in RTLLM (Lu et al., 2023). VerilogEval (Liu et al., 2023b) and RTLLM (Lu et al., 2023) build benchmarks to evaluate the generated Verilog on their functional or syntactic correctness.

2.2. Discriminator-guided Controllable Generation

Controlling LLMs has been widely explored during recent years. Class-conditional language models (CCLMs), such as CTRL (Keskar et al., 2019), aim at controlling the generation by appending a control code in the beginning of training sequences. Discriminator-guided controllable generation is an important technique used for controllable generation, which combines a discriminator with the generative LLMs. The constraints are modeled by calculating the conditional probability on each class and related to the next-token probabilities by Bayes rule. (Holtzman et al., 2018) and (Scialom et al., 2020) predict the labels by feeding each candidate next token into a discriminator and hence guiding the generation to desired directions. PPLM (Dathathri et al., 2020) further employs a forward and backward process with the gradient from a discriminator to update the latent states of LLMs to guide the target generation. In order to reduce the cost of using discriminator for each possible next tokens, GeDi (Krause et al., 2020) contrastively trains the CCLMs as generative discriminators to guide the generation during the decoding. DEXPERTS (Liu et al., 2021) attempts to further improve the performance by incorporating expert and anti-expert during the decoding.

3. Algorithm

In this section, we describe the framework of BetterV, starting with an overview and then detailing the instruct-tuning process and the implementation of the generative discriminator.

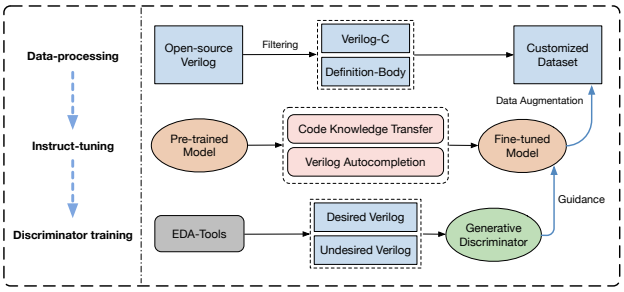

Figure 1. The overview of BetterV.

3.1. Framework Overview

BetterV conducts instruct-tuning for domain-specific understanding and combines generative discriminators to guide different EDA downstream tasks. The framework overview is demonstrated in Figure 1. Our approach first constructs a customized dataset from open-source Verilog, as described in Section 3.2, before the instruct-tuning stage. During instruct-tuning, various instruction problems are employed to teach the LLMs domain-specific knowledge, as detailed in Section 3.3. After that, we implement data augmentation to further enrich the dataset and prepare the labels for training the generative discriminator, as explained in Section 3.4. Finally, in Section 3.5, the generative discriminator is trained using a hybrid generative-discriminative loss and then guides the LLMs through Bayes' rule.

In the rest of this section, we present the details of BetterV in guiding the LLMs for downstream EDA tasks optimization.

3.2. Instruct-Tuning Data-Processing

Inspired by (Dehaerne et al., 2023), we collect open-source repositories on GitHub that contain Verilog or SystemVerilog. At the same time, we check the repository licenses to ensure they permit modification and distribution. We filter out auto-generated files, which are mostly repetitive, and non-permissive files that contradict the licenses. As in (Dehaerne et al., 2023), we filter files with too many or too few lines. These files are then processed by extracting the Verilog modules and functions using regular expressions. We further analyze the extracted contents by measuring the token number after encoding them with the tokenizer and removing contents that exceed a predefined maximum token number. This is because contents exceeding the maximum token number will be truncated by the tokenizer during training, preventing the provision of meaningful information with incomplete contents. Finally, we filter out Verilog modules that have syntactic errors.

To further support our instruct-tuning tasks, we implement two types of processing on the collected contents. First, we use a V2C tool to translate Verilog into C (Mukherjee et al., 2016), thereby transferring the hardware design information represented in Verilog into the C program. The dataset is then appended with the translated C programs that can be derived from the Verilog and do not exceed the maximum token number. Second, we split the Verilog modules and functions into their definitions (including the module header and input and output definitions) and their bodies.

Finally, our constructed dataset contains a set of Verilog-C pairs and a set of Verilog definition-body pairs.

3.3. Domain-specific Instruct-Tuning

The fine-tuning of LLMs is crucial because it determines how well the domain-specific knowledge is learned by the LLMs and how familiar the LLMs become with the customized tasks. This is particularly important for the task of Verilog generation since the existing corpus of Verilog is much smaller than that of general programming languages such as C and Python. The lack of Verilog corpus not only means that pre-trained LLMs learn less about it but also increases the importance of transferring more domain-specific knowledge through fine-tuning. Previous works implement fine-tuning by first generating problem descriptions with LLMs and then using the LLMs to generate the corresponding answers, i.e., the Verilog modules. However, such machine-generated descriptions are often verbose and fallible, failing to transfer useful and reliable knowledge during fine-tuning. Additionally, since this generation is done by LLM inference, it results in time consumption and resource occupancy, significantly increasing the implementation cost.

To solve this issue, we propose domain-specific instruct-tuning, with examples shown in Figure 2. Firstly, we introduce a novel alignment method that maps Verilog code to C, facilitating the LLM's understanding of Verilog. We leverage the abundant knowledge that LLMs already have about general programming languages, such as C. We then construct instructions that ask the LLMs to translate Verilog into C and also translate from C to Verilog, allowing the LLMs to learn to understand Verilog through the implementation of C. With the help of the V2C tool, the dataset is easy to obtain, as described in the last section. Additionally, to improve the performance of LLMs in following Verilog generation instructions, we construct instructions for Verilog autocompletion, where the module definition is provided in the instruction and the completed module is the answer. It should be noted that we do not add extra natural language descriptions in instruct-tuning since the implementation of the C program already acts as a functional description to teach the LLMs.

It should be noted that the LLMs follow an auto-regressive

System Prompt:

You are an experienced Verilog engineer.

Instruction:

You are supposed to translate the following Verilog into C program.

```
module d_latch(
    input d,
    input en,
    input rstn,
    output reg q
);

always @ (en or rstn or d)
    if (!rstn)
        q <= 0;
    else
        if (en)
            q <= d;
endmodule
```

Answer:

```
#include <stdio.h>
#include <stdbool.h>

void d_latch(
    int d,
    bool en,
    bool rstn,
    int *q) {

    if (!rstn) {
        *q = 0;
    } else if (en) {
        *q = d;
    }
}
```

Instruction:

Below is a definition of a Verilog module. And you are supposed to complete the Verilog program.

```
module d_latch(
    input d,
    input en,
    input rstn,
    output reg q
);
```

Answer:

```
module d_latch(
    input d,
    input en,
    input rstn,
    output reg q
);

always @ (en or rstn or d)
    if (!rstn)
        q <= 0;
    else
        if (en)
            q <= d;
endmodule
```

Figure 2. The examples for instruct-tuning. We take a simple `d_latch` Verilog module as example. The left and right part indicate the code knowledge transfer and Verilog autocompletion, respectively.

manner, where an LLM with parameters θ predicts the probability of a sequence $x = \{x_1, ..., x_n\}$, which includes both the instruction and the answer, and factorizes it using the chain rule of probability as follows (Keskar et al., 2019):

$$p_\theta(x) = \prod_{t=1}^{n} p_\theta(x_t|x_{<t}). \tag{1}$$

The generation of LLMs iteratively samples from $p_\theta(x_t \mid x_{<t})$ and then incorporates x_t back into the input for the next prediction. Therefore, the LLMs are trained to minimize the negative log-likelihood on a set of training sequences $D = \{x^1, ..., x^{|D|}\}$:

$$\mathcal{L} = -\frac{1}{|D|} \sum_{i=1}^{|D|} \frac{1}{n} \sum_{t=1}^{n} \log p_\theta(x_t^i|x_{<t}^i). \tag{2}$$

3.4. Data Augmentation

Verilog is scarce compared to other general programming languages, making LLMs prone to overfitting, which can decrease performance. Therefore, in this section, we consider using LLMs to create synthetic Verilog to increase the diversity and size of the training set. After instruct-tuning the LLM, it gains the ability to generate syntactically correct Verilog. Thus, it is convenient to directly use the fine-tuned LLMs for data augmentation. We first utilize the module headers collected during the data processing stage. Then, we construct instructions to prompt the LLMs to directly complete the modules. By setting a high temperature during generation, our LLMs can produce diverse augmented Verilog modules, forming a preliminary synthetic dataset. With the generated Verilog modules, we finally employ an EDA tool to check for syntactic correctness and filter out modules with syntax errors. By performing data augmentation, the robustness and generalization of the fine-tuned model are improved. Moreover, since all the remaining Verilog modules are syntactically correct, appending the synthetic dataset can further enhance the ability to generate syntactically correct Verilog.

Besides augmenting data for instruct-tuning, we need to prepare data for training the discriminator for downstream tasks. Although we can already classify the Verilog modules in our dataset using specific EDA downstream tools, we still need more data. To enable the discriminator to better distinguish between our desired and undesired Verilog, we further utilize our LLMs to generate Verilog modules by completing the module heads. These augmented modules are then labeled by the EDA tools based on special syntax or hardware attributes. Sometimes the attributes we consider for Verilog implementation are absolutely true or false, such as syntactic and functional correctness. In this case, the desired or undesired labels are directly assigned. Other times, the attributes have a relative quality, such as whether the netlist nodes are fewer after synthesis. In these cases, the desired or undesired labels are determined according to specific targets compared with references. For such cases, we always maintain a reference Verilog and then generate the corresponding data.

3.5. Generative Discriminator

After preparing the training data, we discuss how to train a generative discriminator and then use it to guide the LLMs to generate better Verilog. As classical class-conditional language models (CC-LMs), the LLMs are conditioned on an attribute variable, which is expressed as a control code c that is assigned at the beginning of the input sequences. Therefore, the probability that the LLMs predict becomes $p_\theta(x|c)$, which is the conditional probability of x given the attribute c, and its factorization is similar to Equation (1):

$$p_\theta(x|c) = \prod_{t=1}^{n} p_\theta(x_t|x_{<t}, c). \tag{3}$$

For the generative discriminator, we have binary cases to represent opposite attributes, i.e., a control code c and an anti-control code $\bar{c}$ (Krause et al., 2020). The inputs are respectively conditioned by them as $p_\theta(x|c)$ and $p_\theta(x|\bar{c})$ to guide the LLMs on $p_{LLM}(x)$. In our domain-specific scenarios, the preceding codes indicate which kind of Verilog attribute is desired or undesired. As described in Section 3.4,

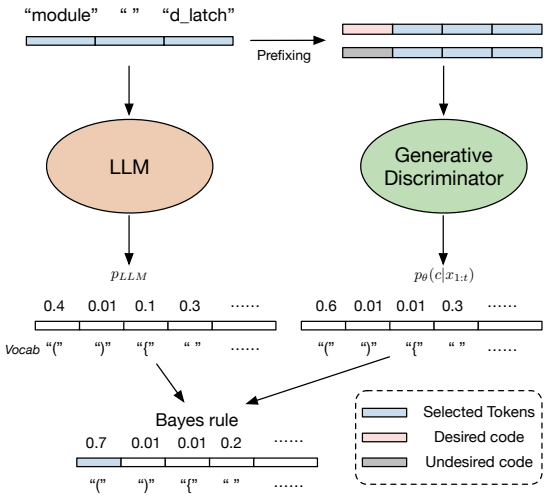

Figure 3. An example shows the guidance from generative discriminator.

we expect the generative discriminator to generate correct and better Verilog implementations, with their labels corresponding to the conditioned attributes.

The discriminator is first trained to predict the next token for each attribute during generation on a set of training sequences D $\{x^1, ..., x^{|D|}\}$ with the generative loss L_g:

$$\mathcal{L}_g = -\frac{1}{|D|}\sum_{i=1}^{|D|}\frac{1}{n}\sum_{t=1}^{n}\log p_\theta(x_t^i|x_{<t}^i, c'), \tag{4}$$

where $c' \in \{c, \bar{c}\}$. With the predicted $p_\theta(x_{1:t}|c)$ and $p_\theta(x_{1:t}|\bar{c})$, Bayes rule is used to compute the probability that each next token x_t belongs to the labeled class:

$$\begin{aligned} p_\theta(c_y|x_{1:t}) &= \frac{p(c_y)(\prod_{j=1}^{t} p_\theta(x_j|x_{<j}, c_y))^{\alpha/t}}{\sum_{c'\in\{c,\bar{c}\}} p(c')\prod_{j=1}^{t}(p_\theta(x_j|x_{<j}, c'))^{\alpha/t}} \\ &= \frac{p(c_y)p_\theta(x_{1:t}|c_y)^{\alpha/t}}{\sum_{c'\in\{c,\bar{c}\}} p(c')p_\theta(x_{1:t}|c')^{\alpha/t}}, \end{aligned} \tag{5}$$

where $c_y \in \{c, \bar{c}\}$ is the label for the sequence x, α is a learnable scale parameter, $p(c) = \frac{e^{b_c}}{\sum_{c'} e^{b_{c'}}}$ with b_c is also a learnable bias for control code c and the probabilities are normalized by the current sequence length t. Then a key point is raised that the discriminator need to be trained that can distinguish the class of each sequence, which means to discriminatively train the class-conditional generative models. Therefore, given the data with both the sequences D $\{x^1, ..., x^{|D|}\}$ and the corresponding labels $\{c_y^1, ..., c_y^{|D|}\}$, the discriminative loss $\mathcal{L}_d$ is defined as follows:

$$\mathcal{L}_d = -\frac{1}{|D|}\sum_{i=1}^{|D|}\log p_\theta(c_y^i|x_{1:n}^i). \tag{6}$$

Finally, the total loss function of generative discriminator training is defined as follows:

$$\mathcal{L}_{total} = \lambda\mathcal{L}_g + (1-\lambda)\mathcal{L}_d, \tag{7}$$

where λ is a hyper-parameter to balance the weight between generative loss and discriminative loss.

After training the generative discriminator, we can use it to guide the sampling of LLMs. In Figure 3, we simply show how this works with an example. The weighted decoding with Bayes rule is employed to guide the generation:

$$p_w(x_t|x_{<t}, c) \propto p_{LLM}(x_t|x_{<t})\, p_\theta(c|x_t, x_{<t})^w, \tag{8}$$

where w is a hyper-parameter to control the influence of the weighted conditional probability. With the weighted decoding, all the potential next tokens x_t in the vocabulary are updated. Moreover, some filtering methods on the tokens before sampling are utilized. The insight is to maintain the high probability tokens and filter out the low probability tokens. First, we define the complete vocabulary set as $\mathcal{V}$. Then we rank all the tokens x_t on the probability $p_\theta(c|x_t, x_{<t})$ to form a new set $\mathcal{V}_{rank}$, and we maintain the top tokens that have the minimum number m such that:

$$\sum_{x_t\in\mathcal{V}_{rank}[1:m]} p_w(x_t|x_{<t}, c) \geq \rho, \tag{9}$$

which is a minimum of at least ρ in cumulative probability mass on $p_w(x_t|x_{<t}, c)$ and the maintained set is defined as $\mathcal{V}_m = \mathcal{V}_{rank}[1:m]$. Furthermore, to avoid the case that the high $p_\theta(c|x_t, x_{<t})$ are filtered out, the tokens that $p_\theta(c|x_t, x_{<t}) > \tau$ are maintained and form the set $\mathcal{V}_\tau$. Therefore, the tokens that we keep before the sampling of generation are given by $\mathcal{V}_k = \mathcal{V}_\tau \cup \mathcal{V}_m$.

With this generative discriminator training pipeline and the equipped downstream task-specific augmented data, we can model any optimization scenario related to Verilog implementation and train a distinct discriminator to guide towards our desired directions. At the same time, the loops in industrial production and the manual cost can be effectively reduced by jointly considering the corresponding downstream constraints during the phase of Verilog generation.

4. Experiments

In this section, we first introduce the experimental setting as well as the evaluation metrics in Section 4.1. Then in Section 4.2 we present our results on functional correctness and compared with other methods, followed by further results with different downstream tasks incorporated with the discriminator (including Synthesis Nodes Reduction in Section 4.3.1 Verification Runtime Reduction in Section 4.3.2) and ablation study in Section 4.4.

4.1. Experimental Setting

We fine-tune the CodeLlama-7B-Instruct (Roziere et al., 2023) and DeepSeek-Coder-6.7b-Instruct (Guo et al., 2024)

as generative LLMs, and fine-tune TinyLlama (Zhang et al., 2024) and DeepSeek-Coder-1.3b-Instruct (Guo et al., 2024) as generative discriminator, respectively. For CodeQwen1.5-7B-Chat (Bai et al., 2023), we can only fine-tune it since there is no suitable smaller size version of it. BetterV is trained with the help of DeepSpeed ZeRO (Rajbhandari et al., 2020) and LoRA (Hu et al., 2021) on the DeepSpeed-Chat (Yao et al., 2023). The experiments are conducted on a machine with two NVIDIA Tesla V100S PCIe 32 GB graphics cards with CUDA driver 11.4.

We employ the VerilogEval (Liu et al., 2023b), which comprises various problems with either machine-generated or human-crafted, as our evaluation benchmark. Following VerilogEval, we measure the Verilog functional correctness with simulation through the pass@k metric with unbiased estimator:

$$\text{pass@}k := \mathop{\mathbb{E}}_{\text{problems}} \left[\frac{1 - \binom{n-c}{k}}{\binom{n}{k}} \right], \tag{10}$$

where $n \leq k$ samples are generated per problem and a problem is solved if any of the k samples passes the unit tests.

Due to the Plug-and-Play nature of the discriminator, BetterV can work on various downstream tasks. For other downstream tasks besides the functional correctness, we can still utilize the pass@k metric and the problems in VerilogEval but the correctness is measured by different tools in Yosys (Wolf) according to different downstream task. In our experiments, we sample n = 20 code completions per problem for each downstream task and measuring pass@k with k = 1,5,10.

For all the models, we employ the Adam optimizer (Kingma & Ba, 2014) with β_1 = 0.9 and β_2 = 0.95 and the cosine learning rate decay (Loshchilov & Hutter, 2016) to schedule our learning rate. For the generative LLMs fine-tuning process, we train it for 4 epochs using an initial learning rate of 9.65e-6 with a batch size of 4. For the generative discriminator, we train it for 3 epochs using an initial learning rate of 9.65e-6 with a batch size of 8. The LoRA dimension for both LLMs and discriminator is set as 128. For different downstream tasks they have different sensitivity for the incorporation of generative discriminators, hence the setting of λ, w, ρ and τ are different and task-sensitive. We initialize the value of learnable scale α as 1 and bias b as 0.

4.2. Functional Correctness

For functional correctness, the target is to correctly complete the Verilog module given the problem description and module definition. The performance of BetterV is compared with GPT-3.5, GPT-4, CodeLlama-7B-Instruct (Roziere et al., 2023), DeepSeek-Coder-6.7b-Instruct (Guo et al., 2024), CodeQwen1.5-7B-Chat (Bai et al., 2023), ChipNeMo (Liu et al., 2023a), Thakur et al. (Thakur et al., 2023), VerilogEval (Liu et al., 2023b) and RTLCoder (Liu et al., 2023c).

Table 1. Comparison of functional correctness on VerilogEval.

Model	VerilogEval-machine			VerilogEval-human		
	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10
GPT-3.5	46.7	69.1	74.1	26.7	45.8	51.7
GPT-4	60.0	70.6	73.5	43.5	**55.8**	**58.9**
CodeLlama	43.1	47.1	47.7	18.2	22.7	24.3
DeepSeek	52.2	55.4	56.8	30.2	33.9	34.9
CodeQwen	46.5	54.9	56.4	22.5	26.1	28.0
ChipNeMo	43.4	-	-	22.4	-	-
Thakur et al.	44.0	52.6	59.2	30.3	43.9	49.6
VerilogEval	46.2	67.3	73.7	28.8	45.9	52.3
RTLCoder-Mistral	62.5	72.2	76.6	36.7	45.5	49.2
RTLCoder-DeepSeek	61.2	76.5	81.8	41.6	50.1	53.4
BetterV-CodeLlama	64.2	75.4	79.1	40.9	50.0	53.3
BetterV-DeepSeek	67.8	79.1	84.0	45.9	53.3	57.6
BetterV-CodeQwen	**68.1**	**79.4**	**84.5**	**46.1**	53.7	58.2

As shown in Table 1, the results demonstrate that our BetterV-CodeQwen have achieved the state-of-the-art performance on VerilogEval, where the pass@1 on VerilogEval-machine and VerilogEval-human outperforms GPT-4 by 8.1 and 2.6, respectively. It should be noted that the pre-trained model CodeLlama-7B-Instruct has the lowest performance, but after instruct-tuning and the guidance of discriminator a huge performance is improved.

4.3. Customized Generation in BetterV

Besides the measurement on functional correctness, it is important to consider the performance in EDA downstream tasks, which is our customized generation. Note that in the rest of the paper, we conduct the experiments only based on the model CodeLlama-7B-Instruct (Roziere et al., 2023) to demonstrate our results. Therefore, "**BetterV**" is employed to replace "BetterV-Codellama", i.e. the model fine-tuned on CodeLlama-7B-Instruct (Roziere et al., 2023). And "**BetterV-base**" refers to the base models of "BetterV-CodeLlama", i.e. the model has undergone instruction tuning but have not yet enhanced by discriminator.

4.3.1. Synthesis Nodes Reduction

Firstly, we need to consider the hardware attributes. Therefore, we train the discriminator to improve the performance of Verilog related to later EDA stage, i.e. the synthesis. We make the labels for discriminator depending on whether the logic networks that synthesised from the generated Verilog have less nodes than the reference Verilog. The netlist nodes number are collected by using the "`proc; aigmap; stat`" commands from Yosys, which transform the Verilog into an And-Inverter-Graph (AIG). The instruction becomes rewriting the reference to have less nodes after synthesis as shown in Figure 4. We select several problems in VerilogEval-human to intuitively demonstrate the performance of BetterV, i.e. the reference nodes number and the generated nodes number are represented. The presented nodes number in the table are the average nodes number of the generated Verilog. As shown in Table 2, with the guidance from discriminator, BetterV can always generate Verilog that has less netlist nodes. The last two columns refer to the proportion of improvements in node reduction compared to

BetterV-Base ("Com Base") and Reference ("Com Ref") respectively. It can be found that, on average, the Verilog generated by BetterV have 46.52% fewer nodes than the reference model and 31.68% less nodes than BetterV-base.

The success of BetterV on the node number reduction after synthesis is significant, since it marks that LLMs start to participate in optimizing the PPA (Power, Performance, Area) of circuit design in EDA flow. Furthermore, optimizing the Verilog implementation at the beginning of the EDA flow has proved to have meaningful influence on later stages.

Table 2. Synthesis nodes reduction with discriminator.

Problem	Ref	BetterV-base	BetterV	Com Base	Com Ref
ece241_2013_q8	657	333.5	255.3	23.44%	61.14%
m2041_q6	1370	692.7	685.6	1.03%	49.95%
counter_2bc	673	666.2	518.9	22.11%	22.89%
review2015_count1k	487	493.4	402.6	18.44%	17.33%
timer	498	294.3	247.3	15.97%	50.34%
edgedetect2	58	189.9	47.4	75.03%	18.27%
counter1to10	325	266.3	240.3	9.76%	26.06%
2013_q2afsm	826	308.8	296.6	3.95%	64.09%
dff8p	50	42.3	37.8	10.63%	24.4%
fsm3comb	844	167.9	104.4	37.82%	87.63%
rule90	6651	12435.6	4536.9	63.52%	31.79%
mux256to1v	2376	2439.6	557.2	77.16%	76.54%
fsm2	389	186.53	121.9	34.65%	68.66%
fsm2s	396	163.7	144.1	11.97%	63.61%
ece241_2013_q4	2222	1789.5	897.4	49.85%	59.61%
conwaylife	43794	547400.3	27037.4	95.06%	38.26%
count_clock	3187	2497.5	2222.2	11.02%	30.27%
countbcd	1589	932.0	849.3	8.87%	46.55%

4.3.2. VERIFICATION RUNTIME REDUCTION

BetterV can not only deal with the problem in synthesis, but also participate in the optimization on formal verification. In this section, we discuss the improvement on Boolean Satisfiability (SAT) runtime reduction when doing formal verification on the Verilog by rewriting the Verilog implementation. If we consider the concrete stages during the EDA flow, such as synthesis in last section, as the key for the circuit (PPA) performance, then the verification process decides how safe to produce a circuit. Since the engineers always spend plenty of time to carefully verify the circuit designs, the runtime of formal verification is always a bottleneck to improve its scalability. Therefore, it is essential to consider the impact of Verilog implementation to formal verification runtime. Since using the SAT is one of the mainstream of formal verification, we conduct experiments to reduce the SAT solving time.

Table 3. Verification runtime reduction with discriminator.

Design	Ref (s)	BetterV-base (s)	BetterV (s)	Com Base	Com Ref
b03	1.233	1.252	0.857	31.54%	30.49%
b06	0.099	0.083	0.078	6.02%	21.21%
Spinner	1.577	1.343	1.064	20.77%	32.53%
traffic_light_example	0.583	0.497	0.480	3.42%	17.67%
Rotate	1.153	1.126	1.034	8.17%	10.32%

Instruction:

Below is a Verilog module. And you are supposed to rewrite this Verilog module with the same definition but less And-Inverter-Graph nodes after synthesis.

```
module d_latch(
    input d,
    input en,
    input rstn,
    output reg q
);

always @ (en or rstn or d)
    if (!rstn)
        q <= 0;
    else
        if (en)
            q <= d;
endmodule
```

Figure 4. An example to instruct the LLMs to rewrite the Verilog module to reduce the AIG nodes after synthesis.

We make the labels for discriminator depending on whether generated SystemVerilog has less SAT solving time than the reference. The SAT solving time is collected by using the "`hierarchy; proc; opt; sat -verify -seq 100 -tempinduct -prove-asserts`" commands from Yosys, which solve the SAT problem to prove all the asserts in a circuit with 100 time steps. The instruction becomes rewriting the reference to reduce verification runtime for solving SAT problems, which is the same as the example shown in Figure 4 after replacing part of the description to "but less Boolean Satisfiability (SAT) solving time". Since the VerilogEval benchmark doesn't contain SystemVerilog that includes the assertions inside the design, we choose another benchmark for our experiment in this section. ANSI-C benchmarks give Verilog with safety assertions and can be used for the evaluation of BetterV (Mukherjee et al., 2015). Some designs among them are selected to evaluate the performance of BetterV. Note that for the "Rotate" case, we uncomment all the assertions to enhance the difficulty. As shown in Table 3, after the discrimiative guidance, BetterV is able to further decrease the SAT solving time. We also show the improvement ratio between the verification time of Verilog generated by BetterV and BetterV-base ("Com Base") and reference Verilog ("Com Ref") in the last two columns respectively. It shows that the Verilog generated by BetterV can save 22.45% in verification time compared with the reference Verilog, and 13.99% in time compared with BetterV-base. The ability to reduce the verification runtime indicates that LLMs can understand how to rewrite the implementation to accelerate the provement. And it gives the hope that we will be able to solve more complex and complicated problems and hence facilitate the issue of scalability in formal verification.

Table 4. Impact of discriminator on functional correctness.

Model	pass@1	pass@5	pass@10
CodeLlama	18.2	22.7	24.3
CodeLlama + Dis	20.3	24.1	24.7
BetterV-CodeLlama-base	40.0	49.5	53.0
BetterV-CodeLlama	40.9	50.0	53.3

Table 5. Impact of discriminator on syntactic correctness.

Model	pass@1	pass@5	pass@10
CodeLlama	41.5	50.8	53.8
CodeLlama + Dis	49.1	58.4	61.1
BetterV-CodeLlama-base	82.6	97.6	99.2
BetterV-CodeLlama	87.1	98.2	99.3

4.4. Ablation Study

4.4.1. Impact of Discriminator

We also show that the generative discriminator has the ability to further enhance the performance after the guidance. In the task of functional correctness, the labels of discriminator are made by checking whether the generated modules are functional equivalence with the reference module. The equivalence checking is done by the eqy tool in Yosys. With the problems in VerilogEval-human, the results in Table 4 illustrate that the guidance from the discriminator can not only enhance the capability on our fine-tuned LLMs, i.e. these base models, but also on the original pre-trained LLMs, i.e. CodeLlama. This observation indicates that our trained discriminator can be employed to any LLM, only if they have the same vocabulary size.

For syntactic correctness, we only consider whether the syntax of the generated Verilog is corrected or not. We construct the labels of generative discriminator based on whether the generated module can be complied by the Yosys with the command "`prep`", which is a generic synthesis script. We also evaluate the performance with the problems in VerilogEval-human, which is demonstrated in Table 5 and the results show that in the task of generate syntactically correct Verilog, BetterV can also achieve remarkable performance, i.e. over 99 pass@10. It can also be observed that the discriminator can help largely enhance the syntactic correctness, i.e. 7.6 and 4.5 improvement on pass@1 for the original CodeLlama-7B-instruct model and the BetterV-base, respectively.

5. Discussion

5.1. Current limitations

We have shown that BetterV can achieve remarkable performance on Verilog generation and the generative discriminator enables improvements on Verilog-related EDA downstream tasks. While BetterV contributes valuable insights and a practical method for enhancing the performance of open-source LLMs, it still has limitations on closed-source models due to the need for access to token-level probabilities. The research about guidance generation on closed-source models remains an unsolved issue.

In addition, while the generative discriminator uses only a small model, the additional computational overhead it brings cannot be ignored. Research to reduce this overhead is needed to accelerate the guidance in an accuracy-optimal manner. For example, it is possible to use model inference acceleration methods such as quantization and pruning on the discriminator.

5.2. Applicability to other domains

The potential applicability of our methods to other domains with similar challenges of data scarcity and precision requirements is promising. For example, many domains utilize structured languages (e.g., SQL for databases, LaTeX for typesetting, and XML for web services). The principles behind BetterV's instruct-tuning and generative discriminators can be adapted to improve the generation of such languages, particularly where precision and correctness are non-negotiable. Fields such as genomics, where data is highly structured and errors can have significant consequences, could benefit from our approach. The instruct-tuning methodology could be adapted to understand the syntax and semantics of genetic sequences, while generative discriminators could be used to guide the generation of sequences to complex biological constraints.

6. Conclusion

In summary, this paper introduces a novel framework, BetterV, for Verilog generation, aimed at controlling Verilog implementation and optimizing its performance across various aspects. We present a complete and easy-to-follow method for collecting and processing Verilog data. Our domain-specific instruct-tuning successfully teaches large language models (LLMs) the knowledge of Verilog using our processed data from the internet. A data augmentation process is proposed to further enhance the diversity of the dataset. Specific generative discriminators are then trained to meet the requirements of different downstream tasks in the electronic design automation (EDA) flow. The experimental results demonstrate the state-of-the-art capabilities of BetterV on various tasks. BetterV marks a pioneering advancement in optimizing the PPA (Power, Performance, Area) of circuit design and introduces a promising direction to accelerate the verification process. Future work could explore additional domain-specific adaptations and incorporate other powerful techniques in LLMs to further enhance BetterV's performance and applicability.

Impact Statement

This paper aims to contribute to the advancement of the Machine Learning field. While our work may have various societal implications, we do not find it necessary to emphasize any particular consequences here.

References

Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Dathathri, S., Madotto, A., Lan, J., Hung, J., Frank, E., Molino, P., Yosinski, J., and Liu, R. Plug and Play Language Models: A Simple Approach to Controlled Text Generation. In *International Conference on Learning Representations (ICLR)*, 2020.

Dehaerne, E., Dey, B., Halder, S., and De Gendt, S. A Deep Learning Framework for Verilog Autocompletion Towards Design and Verification Automation. *arXiv preprint arXiv:2304.13840*, 2023.

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y., et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

He, Z., Wu, H., Zhang, X., Yao, X., Zheng, S., Zheng, H., and Yu, B. ChatEDA: A large language model powered autonomous agent for EDA. In *ACM/IEEE Workshop on Machine Learning CAD (MLCAD)*, pp. 1–6. IEEE, 2023.

Holtzman, A., Buys, J., Forbes, M., Bosselut, A., Golub, D., and Choi, Y. Learning to write with cooperative discriminators. *arXiv preprint arXiv:1805.06087*, 2018.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Keskar, N. S., McCann, B., Varshney, L. R., Xiong, C., and Socher, R. Ctrl: A conditional transformer language model for controllable generation. *arXiv preprint arXiv:1909.05858*, 2019.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Krause, B., Gotmare, A. D., McCann, B., Keskar, N. S., Joty, S., Socher, R., and Rajani, N. F. Gedi: Generative discriminator guided sequence generation. *arXiv preprint arXiv:2009.06367*, 2020.

Liu, A., Sap, M., Lu, X., Swayamdipta, S., Bhagavatula, C., Smith, N. A., and Choi, Y. DExperts: Decoding-time controlled text generation with experts and anti-experts. *arXiv preprint arXiv:2105.03023*, 2021.

Liu, M., Ene, T.-D., Kirby, R., Cheng, C., Pinckney, N., Liang, R., Alben, J., Anand, H., Banerjee, S., Bayraktaroglu, I., et al. ChipNeMo: Domain-Adapted LLMs for Chip Design. *arXiv preprint arXiv:2311.00176*, 2023a.

Liu, M., Pinckney, N., Khailany, B., and Ren, H. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8. IEEE, 2023b.

Liu, S., Fang, W., Lu, Y., Zhang, Q., Zhang, H., and Xie, Z. RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution. *arXiv preprint arXiv:2312.08617*, 2023c.

Loshchilov, I. and Hutter, F. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Lu, Y., Liu, S., Zhang, Q., and Xie, Z. RTLLM: An open-source benchmark for design rtl generation with large language model. *arXiv preprint arXiv:2308.05345*, 2023.

Mukherjee, R., Kroening, D., and Melham, T. Hardware verification using software analyzers. In *IEEE Computer Society Annual Symposium on VLSI*, pp. 7–12. IEEE, 2015. ISBN 978-1-4799-8719-1.

Mukherjee, R., Tautschnig, M., and Kroening, D. v2c – a Verilog to C translator tool. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9636 of *LNCS*, pp. 580–586. Springer, 2016. ISBN 978-3-662-49673-2.

Nijkamp, E., Hayashi, H., Xiong, C., Savarese, S., and Zhou, Y. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*, 2023.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *ACM/IEEE Supercomputing Conference (SC)*, pp. 1–16. IEEE, 2020.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Scialom, T., Dray, P.-A., Lamprier, S., Piwowarski, B., and Staiano, J. Discriminative adversarial search for abstractive summarization. In *International Conference on Machine Learning (ICML)*, pp. 8555–8564. PMLR, 2020.

Thakur, S., Ahmad, B., Fan, Z., Pearce, H., Tan, B., Karri, R., Dolan-Gavitt, B., and Garg, S. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. In *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, pp. 1–6. IEEE, 2023.

Wolf, C. Yosys open synthesis suite. https://yosyshq.net/yosys/.

Yao, Z., Aminabadi, R. Y., Ruwase, O., Rajbhandari, S., Wu, X., Awan, A. A., Rasley, J., Zhang, M., Li, C., Holmes, C., Zhou, Z., Wyatt, M., Smith, M., Kurilenko, L., Qin, H., Tanaka, M., Che, S., Song, S. L., and He, Y. DeepSpeed-Chat: Easy, Fast and Affordable RLHF Training of ChatGPT-like Models at All Scales. *arXiv preprint arXiv:2308.01320*, 2023.

Zhang, P., Zeng, G., Wang, T., and Lu, W. TinyLlama: An Open-Source Small Language Model, 2024.