Retrieval Augmented Verilog Generation

Stanford CS224N Custom Project

Joseph Rejive
Department of Computer Science
Stanford University
jrejive@stanford.edu

Abstract

While there are many large language models capable of generating code in popular programming languages such as Java and Python, they perform poorly on Hardware Description Languages (HDLs) due to the lack of large open-source datasets. This work aims to improve HDL generation by finetuning a large language model to generate Verilog given a design question as the input. Specifically, I explore whether using retrieval augmented generation can reduce hallucinations and improve syntactic and functional code correctness in the generated Verilog. I present raftcoder-7B, a model trained to be robust to inaccurate retrievals during retrieval augmented generation. The model outperforms all baselines for the $pass@1$ metric measuring functional correctness. However, while performing retrieval aware generation during inference improves functional correctness, the syntactic correctness of raftcoder-7B degrades when compared to a finetune-only approach.

1 Key Information

- Mentor: Anna Goldie

2 Introduction

Today, there are many open source, finetuned, large language models that are capable of generating code, such as DeepSeek Coder, Codestral, and Code Llama. While they perform very well on popular coding languages, they fall short on Hardware Description Languages as only a small portion of the training data contains HDLs. In the DeepSeek Coder paper, the authors mention that only 0.01GB of the 797GB dataset contain Verilog samples (Guo et al., 2024). The scarcity of open-source HDL data also meant research in LLM-based HDL generation was limited. In 2022, however, Thakur et al. compiled one of the first open-source datasets for Verilog, after which more papers emerged using their dataset to improve Verilog generation.

This work explores how capable large language models are in generating hardware designs, and the extent in which supervised finetuning on a small synthetic dataset can improve performance. In this paper, I aim to answer the following question: can retrieval augmented generation improve syntactic and functional correctness of Verilog generation? By using the data compiled by Thakur et al., I pretrained and finetuned a large language model using an approach proposed by Zhang et al. to make models robust to inaccurate RAG retrievals during inference. For my RAG pipeline, I utilize a two-level semantic search approach to retrieve relevant Verilog snippets given a design question, which are then used as context to generate a module which answers the design question. Results show that while this approach outperforms all baselines in the $pass@1$ functional metric, inaccurate retrievals result in worse syntactic correctness when compared to a finetune-only model.

3 Related Work

3.1 Verilog Generation and Evaluation

One of the earliest works in LLM-based Verilog generation was done by Thakur et al., who created one of the first open-source Verilog datasets. Their work features finetuning an LLM on this dataset and evaluating model performance on 17 Verilog design questions from the HDLBits website. Later, Liu et al. presented an open-source evaluation framework using all the design questions available from the HDLBits website and a set of testbenches to evaluate correctness. My work uses the dataset provided by Thakur et al., the evaluation framework of Liu et al., and the retrieval augmented generation training method of Zhang et al. to finetune and evaluate a model for Verilog generation.

3.2 Retrieval Aware Finetuning

Retrieval aware finetuning (RAFT) is a method of integrated RAG with supervised fine-tuning (Zhang et al., 2024). They formulate the problem as a question (Q), a set of documents (D_k), and a chain of thought (CoT) style answer (A^*). They differentiate between two types of documents: "oracle" documents (D^*) and "distractor" documents (D_i). Oracle documents contain information from which the answer can be obtained, while distractor documents are not relevant to the question. For a fraction of the samples, P, the set of documents D_k contain at least one oracle document. The remaining $1 - P$ samples contain only distractor documents.

- P% of samples: Q + D^* + D_2 + ... + D_k -> A^*
- (1-P)% of samples: Q + D_1 + D_2 + ... + D_k -> A^*

The authors' motivation in removing the oracle document for some samples is to make the model robust and learn domain-specific information instead of always referring to the context. Their results show that RAFT significantly outperforms various baselines on multiple datasets. In my work, I utilize the RAFT approach to improve performance of the model when irrelevant Verilog module snippets are retrieved and used as context.

4 Approach

4.1 Data Processing

Thakur et al. compiled a 1.4GB dataset consisting of Verilog from open-source Github repositories, as well as examples from 70 Verilog-based textbooks. Significant data processing was performed to generate the pretrain and finetune datasets. First, various free-use copyright/license comments were cleaned from each entry in the dataset. Then data deduplication was performed using the minhash algorithm with a Jaccard similarity of 0.95 (implemented by the open-source text-dedup library (Mou et al., 2024)). After deduplication and comment cleaning, I was left with a 678MB pretraining dataset.

For finetuning, the Pyverilog module was first used to determine whether a module was syntactically correct and fully standalone, meaning that it didn't instantiate other modules (T-Y et al., 2022). Modules were also pruned if they contained more than 400 lines or didn't contain a meaningful design (an example is given in Section A.3 of the appendix). Next, a design question (Q) was automatically generated by Llama3-70B by providing the entire module and asking the LM to generate a Verilog design question, as was done by Liu et al. Further processing was also done to ensure the dataset had the following RAFT format:

Instruction: Given the question and the different Verilog code snippets in the context, design a Verilog module that meets the specifications.
Design Question: Q
Context: D^*, D_2, ..., D_k
Module Header: $module(input..., output...)$;

Document selection for finetuning was done as follows: for $P\%$ of the samples, N consecutive lines from the reference module were randomly selected to be the oracle document D^*. I then selected N consecutive lines from other $K - 1$ randomly selected modules to generate the distractor documents

$D_2, ..., D_k$. For the remaining $(1-P)\%$ of samples, I selected N consecutive lines from K modules that were not the golden module. These samples only contained distractor documents in the context, as described by the RAFT paper. Here, $P = 0.5$, $N = 8$, and $K = 4$ are the hyperparameters that I selected.

In this work, the model is also provided with the module header to ensure that the input/output signals used in the generated design match the input/output names in the evaluation framework. To increase my dataset size and help the model become robust to inaccurate retrievals during inference, I duplicated the question/answer pairs by a factor of two and ensured each sample contained different Verilog snippets in the context. After all data processing was done, I was left with a 68MB finetuning dataset containing 13,300 question-answer pairs.

4.2 Training

For this work, I used the DeepSeek-Coder-7B large language model for Verilog generation. To improve the model's understanding of Verilog, I performed domain-specific continued pretraining of the model. The pretrained model was then finetuned using the RAFT question/answer dataset described in Section 4.1. For all training runs, QLORA was used to limit the number of trainable parameters and reduce GPU memory requirements.

4.3 Inference

During inference, the txtai library was used to create a pipeline for retrieval augmented generation (Mezzetti et al., 2024). To create a vector database, the question/answer finetuning dataset was used. The design questions were encoded to create embedding vectors, and a SQLite backend was used to store the corresponding Verilog module. For retrieval, I implemented a two-level semantic search pipeline. The first search takes the Verilog design question and retrieves the top 4 modules whose descriptions most closely match the original design question. These modules are then split into snippets containing 10 lines each, which was done to limit the context size fed into the model. The second semantic search helps narrow down which snippets might be useful to keep in the context. It starts with an initial inference run in which the original design question and module header are fed as the input to the model (without any Verilog retrievals in the context). The embedding of this intermediate output is then compared against all the retrieved snippet embeddings, and the top 4 snippets with the highest cosine similarity with the intermediate output are selected to be the context snippets for the actual inference run.

5 Experiments

5.1 Evaluation method

To check for functional correctness, the $pass@k$ metric was used. For this metric, k solutions to a design problem are generated, and the problem is considered solved if any of the k designs pass a set of unit tests. The evaluation framework developed by Liu et al. use a slightly different formula to produce low variance estimates for $pass@k$.

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

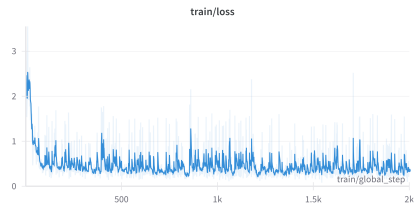
In their testbenches, Liu et al. create $n >= k$ input stimuli for each design question, and c is the number of stimuli where the generated module's outputs match the canonical solution's output. The evaluation dataset used in this framework was created from problems from the HDLBits website (Liu et al., 2023).

5.2 Experimental details

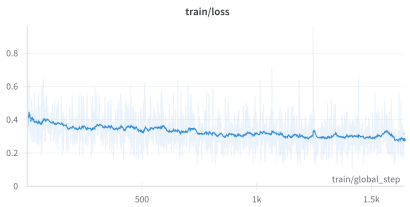
The initial domain-specific continued pretraining was done via QLORA with the following hyperparameters: $r = 16$, $alpha = 32$, and $learning\_rate = 1e-4$. Although one epoch consisted of roughly 5000 steps, pretraining was only done for 2000 steps on a single A100 due to time and

budget constraints. The finetuning run also had the same QLORA rank and alpha parameters, but the learning rate was increased to $3e-4$. The finetuning run lasted 1600 steps on a single A100. For inference the NoInstruct-small-Embedding-v0 model was used as the RAG embedding model. This model takes in text input and outputs an embedding vector of size 384.

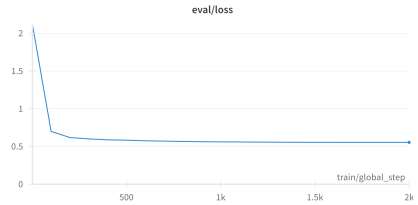
5.3 Results



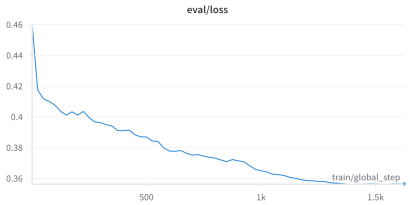
(a) Pretraining Training Loss



(b) Finetuning Training Loss



(c) Pretraining Eval Loss



(d) Finetuning Eval Loss

Figure 1: Training Losses

The loss curves for both the pretraining and finetuning runs can be found in Figure 1. While both train loss curves are very noisy, the eval loss curves are both decreasing, with the pretraining eval loss starting to plateau after 500 steps. Table 1 compares the results of my method with the following baselines:

- verilog-sft: The codegen-16B model that was further pretrained and finetuned by Liu et. al.
- GPT3.5 - Performance of ChatGPT obtained by Liu et al.
- deepseek-coder-7b-instruct-v1.5 : The open-sourced DeepSeek-Coder instruct-tuned model, based on the Llama architecture

All $pass@k$ scores in this paper were obtained using the evaluation framework of Liu et al.

	pass@1	pass@5	pass@10
deepseek-coder-7b-instruct-v1.5	40.6	53.8	57.3
verilog-sft	46.2	67.3	73.7
GPT3.5	46.7	**69.1**	**74.1**
raftcoder-7B (mine)	**52.4**	62.9	66.4

Table 1: Results of raftcoder-7B for Verilog generation.

From Table 1, we can see that raftcoder-7B surprisingly outperforms all baselines at the $pass@1$ metric, while the larger parameter verilog-sft and GPT3.5 models outperform raftcoder-7B in $pass@5$ and $pass@10$. This suggests that larger parameter models are capable of generating more diverse outputs. The results also show that supervised finetuning on a dedicated Verilog dataset improves generation quality, as raftcoder-7B outperforms the baseline deepseek-coder-7b-instruct-v1.5 model across all $pass@k$ metrics. Ablation tests were also performed and the following notation is used in Table 2:

- deepseek-coder-7b-sft : The DeepSeek-Coder model, both pretrained and finetuned on a Verilog dataset, without RAFT for training nor RAG during inference
- deepseek-coder-7b-sft-rag: The same model as the above, but with RAG during inference

	pass@1	pass@5	pass@10
deepseek-coder-7b-sft	49.7	**62.9**	**66.4**
deepseek-coder-7b-sft-rag	30.8	41.3	51.5
raftcoder-7B (mine)	**52.4**	**62.9**	**66.4**

Table 2: Results of ablation experiments.

Table 2 highlights the efficacy of the RAFT method proposed by Zhang et al. We can see that integrating retrieval augmented generation during finetuning helps make the model more robust to inaccurate retrievals, as raftcoder-7B outperforms deepseek-coder-7b-sft-rag. However, it seems like implementing RAFT only slightly improves the model's performance in generating Verilog designs. Similar to Table 1, raftcoder-7B outperforms all other models at the $pass@1$ metric, while performing identically to deepseek-coder-7b-sft at $pass@5$ and $pass@10$.

An analysis of the generated output was also performed to determine if retrieval augmented generation improves syntactic correctness. In Table 3, raftcoder-7B is compared to a finetune-only approach.

	Total fails	Compilation fails	Simulation fails
deepseek-coder-7b-sft	72	33	39
raftcoder-7B (mine)	68	36	32

Table 3: Analysis of fails for pass@1

Interestingly, raftcoder-7B has worse syntactic correctness than the finetune-only baseline, even though its functional correctness is superior. The degradation in syntactic correctness is likely due to attending to irrelevant retrievals in the context, which is discussed in Section 6.2.

6 Analysis

6.1 Finetune Data Quality

In this work, a design question was synthetically generated by Llama3-70B by providing the entire module and asking the LM to generate a Verilog design question. Section A.1 in the appendix shows an example of a finetune question/answer entry. We can see that Llama3-70B was able to create a good design question, explaining the module inputs and outputs correctly. It also provides sufficient information for each "case" statement so that a correct design can be created.

6.2 Retrieval Augmented Generation

The similarity in $pass@k$ scores of raftcoder-7B and the finetuned model without RAG suggests that the retrieved Verilog modules have a limited impact during inference. In Section A.4 of the Appendix, we can see the descriptions of 4 modules that were retrieved for the given question where raftcoder-7B produced an incorrect design. In this example none of the retrievals are similar to the design question being asked. Additionally, the generated design is the following:

```
module top_module (input clk, input a, output reg q);
    always @ (posedge clk) begin
        if (cyc!=0) begin
               cyc<=cyc+1;
        end
    end
```

```
    always @ (posedge clk) begin
        z_split_1 <= ~m_din;
    end
    always @ (posedge clk) begin
        z_split_2 <= 0;
        z_split_2 <= z_split_1;
    end
    always @ (posedge clk) begin
        f_split_2 = f_split_1;
        f_split_1 = m_din;
    end
    reg [15:0] l_split_1, l_split_2;
    always @ (posedge clk) begin
        q = ~a;
    end
endmodule
```

The cannonical solution is:

```
module top_module (input clk, input a, output reg q);
always @(posedge clk)
    q <= ~a;
endmodule
```

Interestingly, the end of the incorrect generation contains the solution (barring the difference in usage of "=" and "<="). Additionally, the statements before are actually obtained from snippets of the retrieved module from Context1:

```
...
reg [15:0] f_split_1, f_split_2;
always @ (posedge clk) begin
  f_split_2 = f_split_1;
  f_split_1 = m_din;
end
...
reg [15:0] z_split_1, z_split_2;
always @ (posedge clk) begin
  z_split_1 <= 0;
  z_split_1 <= ~m_din;
end
...
```

While using the RAFT approach improved raftcoder-7B's functional performance as compared to deepseek-coder-7b-sft-rag, it seems like the model is still attending to irrelevant information in the context. This offers an explanation as to why raftcoder-7B's syntactic correctness is worse than its finetune-only counterpart, deepseek-coder-7b-sft. When snippets are taken from a module at a line-by-line granularity and placed into the context, it is possible that attending to these snippets can cause raftcoder-7B to hallucinate and introduce undeclared variables in its generation.

7 Conclusion

This work highlights the effectiveness of using LLM-generated synthetic data in finetuning a model with a small dataset. Additionally, I was able to replicate the results from Zhang et al. and verify that RAG performance improves when finetuning is performed via the RAFT approach. However as syntactic correctness can be extremely sensitive to irrelevant code snippets in the context, this work underscores the need for effective retrievals during inference if RAG is employed. As the size of the dataset is the primary limitation in this work, it would be interesting to see if the RAFT approach further reduces the sensitivity to irrelevant context when the finetuning dataset is on the order of

gigabytes. Overall, the success of finetuning and the RAFT approach is a promising stepping stone towards using large language models to accelerate the hardware design process.

8 Ethics Statement

The main ethical considerations of this project are regarding how the Verilog dataset is obtained. It's important to make sure all examples are obtained from open-source repositories. This is especially crucial as the digital designs of various hardware companies are treated as closely guarded intellectual property (IP). Training a model on proprietary Verilog examples can potentially lead to IP theft if the end user develops a design where the model generated the intellectual property of another company. To mitigate this issue, it's important to always check the license in which data is released, and to avoid including data if the licence is missing or not permissive for the intended use case. Additionally, another point of concern is regarding the Verilog textbooks in which Thakur et al. extracted examples from. They mention in their paper that 70 textbooks were downloaded as PDFs from an online e-library (Thakur et al., 2024), but it is unclear whether the textbooks are legally hosted on the e-library. This poses the question of whether we should train a model on a dataset that might contain samples that weren't legally or ethically obtained. This issue can be mitigated by receiving explicit permission from the authors of the textbooks that are in the dataset. Of course, another approach would be to only source the dataset from open-source repositories and limit the size of the dataset.

References

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y.K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. In *arXiv*.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. Verilogeval: Evaluating large language models for verilog code generation. In *arXiv*.

David Mezzetti, Datta, Azam Din, Ian A Wilson, Tibor Lipusz, Grant Sherrick, and Nelson Chu. 2024. txtai.

Chenghao Mou, Chris Ha, Kenneth Enevoldsen, Louis Owen, Yongkeun Hwang, and Peiyuan Liu. 2024. text-dedup.

Shinya T-Y, Lenny Truong, Jaewon Hur, Teguh Hofstee, Makoto Kishimoto, Tao Chen, Deepak Yadav, Mohamed Gaber, and Yashas Nagavane Dattatreya. 2022. Pyverilog.

Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2024. Benchmarking large language models for automated verilog rtl code generation. In *Association for Computing Machinery (ACM)*.

Tianjun Zhang, Shishir G. Patil, Naman Jain, Sheng Shen, Matei Zaharia, Ion Stoica, and Joseph E. Gonzalez. 2024. Raft: Adapting language model to domain specific rag. In *arXiv*.

A Appendix (optional)

A.1 LLM Generated Design Question

```
module mux(opA,opB,sum,dsp_sel,out);
       input [3:0] opA,opB;
       input [4:0] sum;
       input [1:0] dsp_sel;
       output [3:0] out;

       reg cout;

       always @ (sum)
              begin
```

```
                        if (sum[4] == 1)
                                cout <= 4'b0001;
                        else
                                cout <= 4'b0000;
                end

        reg out;

        always @(dsp_sel,sum,cout,opB,opA)
                begin
                        if (dsp_sel == 2'b00)
                                out <= sum[3:0];
                        else if (dsp_sel == 2'b01)
                                out <= cout;
                        else if (dsp_sel == 2'b10)
                                out <= opB;
                        else if (dsp_sel == 2'b11)
                                out <= opA;
                end

endmodule
```

Design question: Design a Verilog module that implements a 4-bit output multiplexer with a display select feature. The module takes two 4-bit input operands "opA" and "opB", a 5-bit input sum, and a 2-bit display select input "dsp_sel". The module outputs a 4-bit value "out".

The module should perform the following operations based on the "dsp_sel" input:

- If "dsp_sel" is 00, the module should output the lower 4 bits of the input sum.
- If "dsp_sel" is 01, the module should output a 4-bit value indicating the carry-out of the sum (1 if the sum has a carry-out, 0 otherwise).
- If "dsp_sel" is 10, the module should output the value of "opB".
- If "dsp_sel" is 11, the module should output the value of "opA".

A.2 Llama3-70B Prompt Template

Given a Verilog module, create a design question to which the provided module is the reference solution. For example if the input module is:

```
module half_adder(
    input a,
    input b,
    output s,
    output c
);
    assign s = a^b;
    assign c = a&b;
endmodule
```

Your output should be: Create a Verilog module that implements a half adder given 1 bit inputs 'a' and 'b'. The module should output a 1 bit sum 's' and a 1 bit carry 'c'. The sum bit is the XOR of both inputs, while the carry bit is the AND of both inputs.
Now do the same for the following Verilog module: *MODULE*
When creating a design question for the above module, provide high-level details of what is happening inside it, and ensure that the question you generate can be answered with the details provided. Describe all inputs, outputs and parameters (if any are used). If case statements are needed, provide sufficient details on the behavior in each case statement. Your response should not include "Here is the design question for the provided Verilog module:".

A.3 Non-useful Module Example

```
    module sky130_fd_sc_ms__a311oi (
    //# {{data|Data Signals}}
    input A1 ,
    input A2 ,
    input A3 ,
    input B1 ,
    input C1 ,
    output Y ,

    //# {{power|Power}}
    input VPB ,
    input VPWR,
    input VGND,
    input VNB
);
endmodule
```

A.4 Retrieval Analysis

```
Question:
This Verilog module is a simple combinational logic circuit that has one
    input (a) and one output (q). It is triggered by a clock signal (clk)
    and has a register (q) that stores the output value. The module uses a
    positive edge triggered always block to implement the logic. When the
    clock signal goes from low to high, the output q is set to the inverse
    of the input a. This means that when the input a is high, the output q
    will be low and vice versa. The module will continuously update the
    output q whenever the clock signal changes from low to high.

Context0:
Design a Verilog module that implements a synchronous logic circuit with
    two enable inputs (ena and enb) and two write enable inputs (wea and
    web). The module has a single output (out) and a clock input (clk). The
     output (out) is set to 1 only when all four inputs (ena, enb, wea, and
     web) are high simultaneously, and this condition is detected on the
    rising edge of the clock. The module should also include a testbench to
     verify its functionality. The testbench should toggle the clock and
    set the enable and write enable inputs in a specific sequence to test
    the module's behavior.
module main;

Context1:
Design a Verilog module that demonstrates various aspects of digital
    circuit behavior, including signal assignment, conditional statements,
    and clock-based sequential logic. The module should have a single input,
     "clk", which is a clock signal.

The module should maintain an internal counter, "cyc", which increments
    every clock cycle. The counter is used to control the behavior of the
    module at different clock cycles.

The module should also have an internal 16-bit register, "m_din", which is
    initialized to a specific value at clock cycle 1 and then updated at
    subsequent clock cycles based on the counter value.

The module should include several blocks of combinational logic that
    perform different operations on "m_din", including assignment,
```

```
    inversion, and bitwise operations. These blocks should be sensitive to
    changes in "m_din" and update their outputs accordingly.

The module should also include several blocks of sequential logic that
    update their outputs on every clock edge. These blocks should perform
    operations such as assignment, inversion, and bitwise operations on "
    m_din", as well as maintain internal state.

The module should include several assertions that check the values of
    internal signals at specific clock cycles. If these assertions fail,
    the simulation should stop.

Finally, the module should print out the value of "m_din" at every clock
    cycle, and print a success message when the simulation is complete.

Context2:
Design a Verilog module that implements a sequential circuit with a single
    input "in", a clock input "clk", and an active-low reset input "rst_n".
     The module should output a single bit "out". The circuit's behavior is
     as follows: it keeps track of the previous two input values and
    outputs 1 if the input sequence is "01" or "111", and 0 otherwise. The
    module should have three states: S0 (previous input is 0), S1 (previous
     input is 1), and S2 (previous input is 1 and the input before that is
    also 1). The module should reset to state S0 and output 0 when the
    reset input is low.

Context3:
Design a synchronous, resettable Verilog module that implements a 2-input,
    1-output multiplexer. The module should have a parameterizable input
    width, specified by the WIDTH parameter, which defaults to 8 bits. The
    module takes in two input signals, "a" and "b", of the same width, and
    outputs a signal "c" of the same width. The module also takes in a
    clock signal "clk" and a reset signal "rst".
```