Benchmarking Large Language Models for Automated Verilog RTL Code Generation

Shailja Thakur*, Baleegh Ahmad*, Zhenxing Fan*, Hammond Pearce*,
Benjamin Tan†, Ramesh Karri*, Brendan Dolan-Gavitt*, Siddharth Garg*
*New York University, †University of Calgary

***Abstract*—Automating hardware design could obviate a significant amount of human error from the engineering process and lead to fewer errors. Verilog is a popular hardware description language to model and design digital systems, thus generating Verilog code is a critical first step. Emerging large language models (LLMs) are able to write high-quality code in other programming languages. In this paper, we characterize the ability of LLMs to generate useful Verilog. For this, we fine-tune pre-trained LLMs on Verilog datasets collected from GitHub and Verilog textbooks. We construct an evaluation framework comprising test-benches for functional analysis and a flow to test the syntax of Verilog code generated in response to problems of varying difficulty. Our findings show that across our problem scenarios, the fine-tuning results in LLMs more capable of producing syntactically correct code (25.9% overall). Further, when analyzing functional correctness, a fine-tuned open-source CodeGen LLM can outperform the state-of-the-art commercial Codex LLM (6.5% overall). We release our training/evaluation scripts and LLM checkpoints as open source contributions.**

***Index Terms*—Transformers, Verilog, GPT, LLM**

I. Introduction

State-of-the-art hardware design flows use hardware description languages (HDLs) such as Verilog and VHDL to specify hardware architectures and behaviors. However, the process of writing HDL code is time-consuming and bug-prone [1]. As design complexity grows, there is a need to reduce design costs and developer effort during hardware specification. High-level synthesis tools enable developers to specify functionality in languages like C but come at the expense of hardware efficiency. A promising new approach is the use of large language models (LLMs) [2] to *automatically* generate code from natural language specifications. LLMs are successful in generating code in languages like C and Python. Their use in generating HDL code requires study.

LLMs are deep neural networks, typically based on transformer architectures, that aim to model the underlying distribution of a natural or structured language corpus. Given a sequence of words (or "tokens") LLMs predict a distribution over the next word/token. Used in a loop, LLMs can complete paragraphs in English starting with the first sentence, or code from comments or initial lines of code.

We undertake the first comprehensive evaluation of the syntactic and functional correctness of synthesizable Verilog code generated by both open-source and commercial LLMs. There are several challenges. First, baseline LLMs, including GitHub Copilot which ostensibly generates code in many programming languages including Verilog, frequently fail syntax, synthesis, and functional checks [3]. Fine-tuning LLMs on a Verilog corpus can help, but requires a large dataset of Verilog code which is lacking. Prior work trained models on template-generated hardware specifications and corresponding Verilog, but this is time-consuming and does not generalize to unseen problems [4]. Finally, test problems and methods to evaluate the syntactic and functional correctness of LLM-generated code on a large scale are lacking.

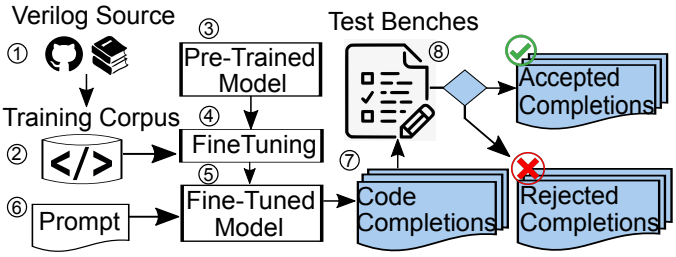

Fig. 1. Experimental Evaluation of LLM Verilog Completions

Our paper contributes the following. (1) By consolidating available open-source Verilog code as well as a selection of textbooks about Verilog, we create (to the best of our knowledge) the largest training corpus of Verilog code yet used for training LLMs. (2) Using this corpus, we examine fine-tuning five different pre-trained LLMs models with parameter counts ranging from 345M to 16B, producing five new fine-tuned models specialized for Verilog. (3) To evaluate the efficacy of the models and determine the effect of the parameter sizes, we design a set of Verilog coding problems with varying difficulty levels, and corresponding test benches to test the functional correctness of generated code. Open source: We provide the training/evaluation code[1] and LLM checkpoints[2].

Fig. 1 illustrates our experimental platform for studying the impact of parameters such as temperature, number of sequences generated per problem, and number of LLM parameters. Section III discusses creating the training data from GitHub and PDFs of Verilog textbooks ① with pre-processing ② and the five pre-trained LLMs ③ that we fine-tune ④ for completing Verilog code ⑤. Section IV explains the evaluation setup, including our hand-designed prompts ⑥. Section V presents our results from generating code suggestions ⑦ and evaluating with an analysis pipeline that compiles the Verilog and checks it against unit tests ⑧. Section VI discusses how

[1]https://github.com/shailja-thakur/VGen
[2]https://huggingface.co/shailja

978-3-9819263-7-8/DATE23/© 2023 EDAA

our evaluation shows that the largest code-based LLMs (i.e., CodeGen-16B) fine-tuned on our Verilog corpus outperforms all other evaluated LLMs. Qualitatively, our best-performing LLMs can generate functioning code for challenging problems.

II. Background and Related Work

A. Background

Transformer-based deep neural networks [5] have demonstrated impressive ability in myriad domains, including language-related tasks. Inputs to LLMs are in the form of tokens— a set of common character sequences where each has a unique numeric identifier using a byte pair encoding [6]. Given a sequence of tokens as an input prompt, the LLM outputs a probability distribution over the vocabulary for the next token given the prompt. A token is picked from this distribution, often the most likely token, appended to the prompt, and this sequence is fed back to the LLM, yielding a new token. This is repeated to generate a *completion*, a sequence of tokens that completes the input prompt.

LLMs for code are trained on a corpus of code in a target programming language or sometimes even on a mix of source code files in various languages. Dataset sizes can often be on the order of hundreds of gigabytes. Prompts for these LLMs can be in the form of comments, code snippets, or both. An LLM trained on a mix of programming languages will often (implicitly) infer the language from the prompt.

LLMs are expensive to train from scratch due to their large datasets and massive parameter counts. However, pre-trained LLMs can be specialized for a user task by fine-tuning them on a specialized dataset. Fine-tuning is significantly faster than training from scratch because it only requires a small number of training epochs. Several LLMs pre-trained for both natural language and code either make the weights available, like NVIDIA's MegatronLM [7] or Salesforce's CodeGen models [8], or provide fine-tuning through an API, like AI21studio's Jurassic-1 (J1) models.[3]

B. Prior Work

Programming is challenging, given the need for human designers to interpret and transform natural language specifications into programming structures. This motivates the use of natural language processing (NLP) to transform language to code [9]. Hardware design using Verilog HDL is similar to programming. Prior work explored NLP techniques for generating assertions [10], albeit on a small scale. Pearce et al. trained `DAVE`, a small LLM to produce Verilog snippets from template-based natural language descriptions for a limited set of functions [4]. GitHub's Copilot was evaluated for security bugs produced during out-of-the-box Verilog completions [3] and was found to be lacking. This study is a large-scale exploration of the capabilities of LLMs across more design tasks using an automated evaluation framework. There is no open dataset to train and evaluate LLMs on writing Verilog.

[3]https://studio.ai21.com/docs/jurassic1-language-models/#general-purpose-models

TABLE I
Baseline LLM architectures used in our study.

Model-Parameters / Pre-Training Data	Layers	Heads	Em-bed.	Context Length
MegatronLM-355M [7] / NL [12], [13]	24	16	64	1024
J1-Large-7B[1] / NL [14]	32	32	128	4096
CodeGen-2B [8] / NL [15], Code	32	32	80	2048
CodeGen-6B / NL [15], Code	33	16	256	2048
CodeGen-16B / NL [15], Code	34	24	256	2048
code-davinci-002 [2] / NL [14], Code	NA	NA	NA	8000

III. LLM Training

We describe our method for training (or fine-tuning) LLM models. We begin by describing our curated Verilog datasets, followed by the LLM architectures and fine-tuning method.

A. Verilog Training Corpus

Our primary Verilog training corpus comes from open-source Verilog code in public GitHub repositories. Additionally, we also created a dataset of text from Verilog textbooks to understand whether that further improved LLM performance.

a) GitHub Corpus: We use Google BigQuery to gather Verilog repositories from GitHub, where it has a snapshot of over 2.8 million repositories. We use a query that looks for keywords such as "Verilog" and files with '.v' extension. We de-duplicated files (using MinHash and Jaccard similarity metrics [11]) and filtered files by keeping '.v' files that contain at least one pair of `module` and `endmodule` statements. Finally, we filtered large files (number of characters $\geq$ 20K). The training corpus from GitHub yielded $\sim$50K files / $\sim$300 MB.

b) Verilog Books Corpus: We downloaded 70 Verilog-based textbooks from an online e-library in PDF format, then extracted text using the Python-based tool pymuPDF which uses optical character recognition to extract text. Depending on the quality of the PDF, the text quality varies. We cleaned the text by filtering irrelevant passages (e.g., index, preface, and acknowledgments) and used regular expressions such as: `module(.*\n*\s*\t*)(\()((?!module)(?!endmodule).*\W*)*endmodule` to identify blocks of prose and associated Verilog snippets. From these blocks, overlapping sliding windows were used to produce training examples. The final Verilog corpus of textbook-extracted and GitHub code had a size of 400 MB.

B. Baseline LLM Architectures

Table I shows the LLMs we used and summarizes design parameters, including the number of layers, heads, embedding size (head dimension), context length, and the data source (natural language (NL) and/or code). As code-davinci-002 is derived from GPT-3 [2], its architecture is the same as GPT-3. Its exact parameters are not known, so we leave these as *NA*.

TABLE II
PROBLEM SET

Prob. #	Difficulty	Description
1	Basic	A simple wire
2	Basic	A 2-input and gate
3	Basic	A 3-bit priority encoder
4	Basic	A 2-input multiplexer
5	Intermediate	A half adder
6	Intermediate	A 1-to-12 counter
7	Intermediate	LFSR with taps at 3 and 5
8	Intermediate	FSM with two states
9	Intermediate	Shift left and rotate
10	Intermediate	Random Access Memory
11	Intermediate	Permutation
12	Intermediate	Truth table
13	Advanced	Signed 8-bit adder with overflow
14	Advanced	Counter with enable signal
15	Advanced	FSM to recognize '101'
16	Advanced	64-bit arithmetic shift register
17	Advanced	ABRO FSM*

**from Potop-Butucaru, Edwards, and Berry's "Compiling Esterel"*

C. LLM fine-tuning

We fine-tune five LLMs from Table I on our Verilog training datasets. Training the CodeGen LLMs was challenging due to the large number of parameters. At 16-bit precision, CodeGen-16B's parameters alone occupy 30 GB of GPU memory; fine-tuning additionally requires sufficient GPU memory to store optimizer states and intermediate calculations, requiring around 250GB across multiple GPUs. We use model and data parallelism and strategies for sharding the optimizer states across GPUs by basing our implementation on DeepSpeed.[4] We set the training hyperparameters to their defaults. The CodeGen LLMs (2B, 6B, 16B) are fine-tuned for 1 epoch on an HPC cluster with two RTX8000s, four RTX8000s, and three A100s, and training completes in two, four, and six days, respectively. Megatron-LM is fine-tuned for 9 epochs using one RTX8000 for 15 hours using the default configuration [7]. We use the off-the-shelf AI21 studio for fine-tuning J1-Large.

IV. LLM Evaluation Setup

The LLM input is a prompt from the problem set in Section IV-A. We truncated generated code at keywords `end` and `endmodule` and checked for compilation and functional correctness. We compile Verilog with Icarus Verilog v11.0.

A. The Problem Set

We created 17 Verilog problems inspired by classroom exercises and examples from the HDLBits website.[5] We assign a difficulty level for each problem (see Table II). The problem set covers combinational and sequential logic designs. It has three finite state machines with different requirements, operations such as permutation, shift left, and rotate, and basic blocks like a MUX, RAM, LFSR, adders, and counters. Examples for basic, intermediate, and advanced problems are shown in Fig. 2-4 respectively. They are obtained using CodeGen-16B-FT and edited for visualization and brevity.

[4]https://huggingface.co/docs/transformers/main_classes/deepspeed
[5]https://hdlbits.01xz.net/wiki/Problem_sets

```
// This is a 3-bit priority encoder. It outputs the
    position of the first high bit.
module priority_encoder(input[2:0] in,output reg[1:0]
    pos);
// If none of the input bits are high (i.e., input is
    zero), output zero.
// assign the position of the highest bit of in to pos.
```

(a) Prompt

```
always @(in)
if (in == 0)  pos <= 2'h0;
else if (in[0])  pos <= 2'h0;
else if (in[1]) pos <= 2'h1;
else pos <= 2'h2;
endmodule
```

(b) Correct completion

```
always @(in) begin
case(in)
 3'b000: pos=2'b00;
 3'b001: pos=2'b01;
 3'b010: pos=2'b10;
 3'b011: pos=2'b11;
default: pos=2'b00;
. . .
```

(c) Incorrect completion. Positions are offset by 1.

Fig. 2. Basic example - Problem 3: A 3-bit priority encoder

```
// This is a counter that counts from 1 to 12
module counter(input clk,input reset,output reg[3:0] q);
```

(a) Prompt

```
always @(posedge clk) begin
if(reset) q <= 4'd1;
else begin
if(q == 4'd12) q <= 4'd1;
else q <= q + 4'd1;
```

(b) Correct completion

```
always @ (posedge clk) begin
if(reset) q <= 4'd1;
else begin
q <= q + 4'd1;
end end endmodule
```

(c) Incorrect completion. Counter does not stop at 12.

Fig. 3. Intermediate example - Problem 6: A 1 to 12 counter

B. Input parameters

Each LLM query has a prompt, a sampling temperature (t), and a number of completions/prompt (n).

Prompts: We provide three prompts with increasing detail, i.e., low (L), medium (M), and high (H). Prompt L has an initial comment describing the function of the module and the module header with name and inputs/outputs with types. We declare internal signals. M includes L plus comments that describe the function using signal names. H replaces and/or appends comments in M with more detail and resembles pseudo-code instead of a predominantly natural language specification. Fig. 5 is an example for Problem 15. L has no lines highlighted (the prompt is lines 1–4). M includes L and lines highlighted yellow (the prompt is lines 1–8). H includes M and lines in gray (the prompt is lines 1–15).

Sampling temperature (t)**:** A higher value means that the LLM takes more risks and yields more creative completions. We use $t \in \{0.1, 0.3.0.5, 0.7, 1\}$.

Completions per prompt (n)**:** For each prompt, LLM gen-

```
// This is an FSM
// It outputs 1 when 1 is received for signals a and b
     irrespetive of their order, either simultaneously or
     non-simultaneously.
module abro(input clk,input reset,input a,input b,output
     z);
parameter IDLE = 0, SA = 1, SB = 2, SAB = 3;
reg [1:0] cur_state,next_state;
//Update state or reset on every clock edge
//Output z depends only on the state SAB
//The output z is high when cur_state is SAB
//cur_state is reset to IDLE when reset is high.
     Otherwise, it takes value of next_state.
//Next state generation logic:
//If cur_state is IDLE and a and b are both high, state
     changes to SAB
//If cur_state is IDLE,and a is high,state changes to SA
//If cur_state is IDLE,and b is high,state changes to SB
//If cur_state is SA,and b is high,state changes to SAB
//If cur_state is SB,and a is high,state changes to SAB
//If cur_state is SAB, state changes to IDLE
```

(a) Prompt

```
// Implements an FSM in Verilog
always @(posedge clk or posedge reset) begin
if(reset) cur_state <= IDLE;
else cur_state <= next_state; end
always @(cur_state or a or b) begin
case(cur_state)
IDLE: begin
if(a && b) next_state = SAB;
else if (a) next_state = SA;
else if (b) next_state = SB; end
SA: begin
if(b) next_state = SAB;
else next_state = SA; end
SB: begin
if(a) next_state = SAB;
else next_state = SB; end
SAB: begin
next_state = IDLE; end
default:
next_state = IDLE; endcase end
// Output logic:
// Output z is high when cur_state is SAB
// Output z is high when cur_state is IDLE and a and b
     are both high
// Output z is low otherwise.
assign z = (cur_state == SAB || (cur_state == IDLE && a
     && b));
endmodule
```

(b) Correct completion

```
... SAME AS (b) TILL LINE 24 ...
assign z = (cur_state == IDLE && a && b) || (cur_state
 == IDLE && a);
endmodule
```

(c) Incorrect completion. Output is not assigned to state SAB.

Fig. 4. Advanced example - Problem 17

erates n completions where $n \in \{1, 10, 25\}$. For J1-Large, we skip $n = 25$ because they do not support this value.

max_tokens: The maximum number of tokens generated for each completion was set to 300 for all LLMs except J1-Large. For J1-Large the limit is 256. Nucleus sampling probability mass (`top_p`) was set to the default value of 1.

C. Test benches

We developed a test bench to check each solution's functional correctness. The test benches exercise the designs for corner cases and are exhaustive for basic and some intermediate cases. For the remaining cases, the test benches are analogous to unit tests. This keeps the evaluation time

```

// This is a finite state machine that recognizes the
     sequence 101 on the input signal x.
module adv_fsm(input clk, input reset, input x, output z);
reg [1:0] present_state, next_state;
parameter IDLE=0, S1=1, S10=2, S101=3;
// output signal z is asserted to 1 when present_state is
// S101
// present_state is reset to IDLE when reset is high,
// otherwise it is assigned next state
// if present_state is IDLE, next_state is assigned S1 if
// x is 1, otherwise next_state stays at IDLE
// if present_state is S1, next_state is assigned S10 if
// x is 0, otherwise next_state stays at IDLE
// if present_state is S10, next_state is assigned S101 if
// x is 1, otherwise next_state stays at IDLE
// if present_state is S101, next_state is assigned IDLE
```

Fig. 5. Varying the prompt details: Low, Medium and High. Problem 15.

reasonable, e.g., for the RAM module, the data width is 8 and the address width is 6 in the prompt; an exhaustive test bench requires 2^{14} test inputs. In some cases, specifications in the prompts are ambiguous and thus can yield several correct responses. For example, when one does not specify whether a reset should be synchronous or asynchronous.

V. LLM Evaluation and Results

A. Research Questions

We answer research questions (RQs) regarding the quality of Verilog generation given the scenarios and test benches from Section IV: **RQ1**. How well do 'base' LLMs perform on the Verilog generation set? **RQ2**. Does fine-tuning LLMs improve that performance? **RQ3**. Are larger LLMs with more parameters better? **RQ4**. Does variability in problem description impact quality and the number of correct completions?

B. Results

We measure generated code quality using problem sets described in Section IV. A scenario is a combination of problems across difficulties and description levels. We query the models with all prompt $\times$ t $\times$ n combinations. For fairness, we present each model's "*best results*" by focusing on the completions generated with the t for each model for which their completions were most successful at compiling and passing the functional tests (for each problem difficulty and description level). We present these *best results* for $n = 10$ in Table III and Table IV. Table III shows the proportion of completions that compile and Table IV shows the proportion of completions that pass functional tests, for the completions produced by a given temperature setting that resulted in the most successful completions for each scenario. As in prior work [8], we characterize the model performance with the Pass@k metric, where k is the number of problems in a *scenario* times n, the number of suggestions per problem. A higher Pass@k indicates a relatively 'better' result. For compilation (Table III), the Pass@k metric reflects the proportion of completions that compile. For functional tests, this metric is the fraction of the k code samples that pass.

For interest, Table IV reports the inference time for each LLMs query, including communication time with a remote server if required. Note that the results are after fine-tuning the

TABLE III
PASS@(SCENARIO*n) AT n=10 FOR COMPILED COMPLETIONS (PASS=COMPILING), PT = PRE-TRAINED, FT = FINE-TUNED. BOLD REFLECTS THE (BEST) HIGHEST PERFORMANCE FOR THAT DIFFICULTY.

Model	**Model Type**	Basic	Intermediate	Advanced
MegatronLM-345M	PT	0.000	0.000	0.000
	FT	0.730	0.391	0.165
CodeGen-2B	PT	0.080	0.065	0.176
	FT	0.902	0.612	0.592
CodeGen-6B	PT	0.052	0.152	0.187
	FT	**0.987**	0.689	**0.599**
J1-Large-7B	PT	0.182	0.176	0.108
	FT	0.882	0.635	0.588
CodeGen-16B	PT	0.132	0.203	0.240
	FT	0.942	**0.728**	0.596
code-davinci-002	PT	0.847	0.452	0.569

model using the training corpus from GitHub only. We discuss the case for fine-tuning on GitHub and PDFs combined as an ablation study in the discussion. Fine-tuned CodeGen-16B LLM outperforms all LLMs. All fine-tuned LLMs outperform their pre-trained counterparts. [**Ans. RQ1 and RQ2**].

Completions vs. Temperature (t): Fig. 6 summarizes the Pass@(*scenario**n) metric for our experiments sweeping temperature. Pass@(*scenario**10) has the highest value for $t = 0.1$ and degrades exponentially with temperature. The LLM generates accurate solutions at low temperatures and accurate synthesizable codes are expected from fewer candidates.

Completions vs. # Completions/Prompt (n): We study synthesis quality as a function of completions/prompt. The right-hand panel in Fig. 6 shows the Pass@(*scenario**n) for all LLMs. Pass@(*scenario**1) is better than Pass@(*scenario**10). This improves as the number of completions increases. This is the case because the number of candidate solutions at low temperatures increases, increasing the completions passing the test benches. $n = 10$ is good for all problem difficulty levels.

Completions vs. LLM Size: Fig. 6 and 7 show that LLMs with more parameters (CodeGen-16B, code-davinci-002) outperform LLMs with less parameters such as Megatron-355M and CodeGen-2B. These LLMs yield more completions that pass test benches and more correct completions. [**Ans. RQ3**].

Completions vs. Prompts: Prompt quality impacts the LLM generation quality. We study the effect of variations in the prompt description at two levels: How do the difficulty of the prompt and the description of the prompt impact code completions? We use Pass@(*scenario**10) as the metric. The right-hand side panel in Fig. 7 shows that the Pass@(*scenario**10) decreases with increasing prompt difficulty. Simple problems such as AND are easy to translate to Verilog, as opposed to advanced problems such as LFSR. The left-hand side panel in Fig. 7 shows that the number of correct solutions decreases with terse prompts. [**Ans. RQ4**].

VI. DISCUSSION AND LIMITATIONS

Fine-tuned LLMs generate code that compiles better when compared to the pre-trained LLMs (Table IV). Using the best

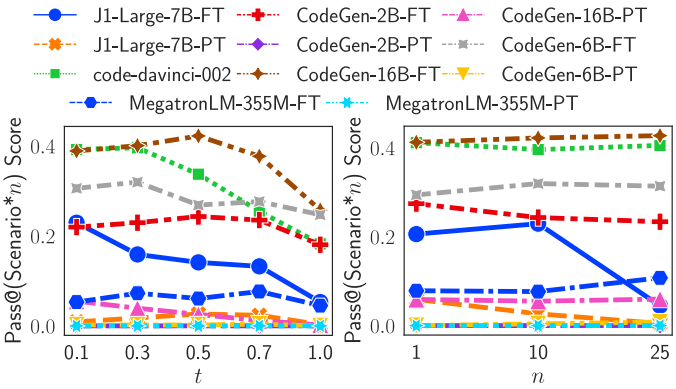

Fig. 6. Pass@(scenario*n) for scenarios passing test benches across temperature (t) and completions per prompt (n). Higher is better.

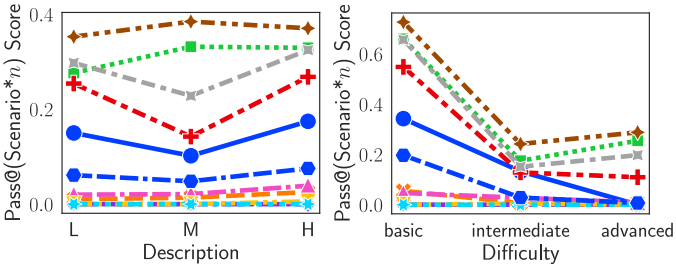

Fig. 7. Pass@(scenario*n) for scenarios passing test benches across problem difficulties and description levels. Higher is better.

Pass@(*scenario**10) values, only 11.9% of the completions generated by pre-trained LLMs compiled vs. 64.6% of those by fine-tuned LLMs. Thus, a designer may use these LLMs with text/pseudo-code to generate a syntactically-correct design "skeleton", and tweak it to meet functional requirements.

We used test benches to assess the generated Verilog. These test-benches are comprehensive for the Basic problems, but as the problems become more complex, the test-benches cover only those behaviors fully specified in the problem comments. As LLMs tend to provide similar responses when several completions per prompt are requested, the exact test-bench implementation can have a large impact on how many cases pass. We observe this in the LLMs' responses to FSM problems 8, 15, and 17. As the problem comments do not specify whether the reset is synchronous/asynchronous, the LLMs are free to produce any variation. For all problems, we verify whether an active-high reset results in the correct value at the output, but we do not test the asynchronous/synchronous corner case nor other similar edge conditions.

The best-performing LLM (CodeGen-16B (FT)) performed poorly for some problem sets. For any given problem, CodeGen-16B (FT) produced 540 completions, but for Problems 7 (LFSR) and 12 (Truth table), none passed, and for Problem 9 (Shift and Rotate), only one passed. We inspected the completions and observed that for Prob. # 7, the LLMs did not concatenate the most significant bits with the feedback value. This was the problem in most cases and a better prompt might yield a correct result. This indicates the importance of creating the best prompt, pointing to prompt engineering as future work. For Prob. #9, completions either do not

TABLE IV
PASS@(SCENARIO*n) AT n =10 FOR TEST BENCH PASSING COMPLETIONS (PASS=PASSED FUNCTIONAL TESTS), PT = PRE-TRAINED, FT = FINE-TUNED. BOLDED VALUE IN EACH TEST COLUMN REFLECTS THE (BEST) HIGHEST PERFORMANCE FOR THAT PROBLEM SET AND DIFFICULTY.

Model	Model Type	Inference Time (s)	Basic			Intermediate			Advanced		
			L	M	H	L	M	H	L	M	H
MegatronLM-355M	PT	3.628	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	FT	0.175	0.170	0.591	0.245	0.043	0.018	0.025	0.000	0.000	0.000
CodeGen-2B	PT	1.478	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.016	0.020
	FT	0.665	0.835	0.350	0.630	0.130	0.092	0.163	0.132	0.048	0.068
CodeGen-6B	PT	2.332	0.000	0.000	0.000	0.000	0.000	0.013	0.000	0.000	0.000
	FT	0.710	**1.000**	0.500	0.760	0.135	0.150	0.168	**0.284**	0.164	0.164
J1-Large-7B	PT	7.146	0.044	0.058	0.067	0.000	0.000	0.021	0.000	0.000	0.000
	FT	2.029	0.388	0.283	0.342	0.125	0.075	0.200	0.000	0.000	0.000
CodeGen-16B	PT	2.835	0.000	0.085	0.055	0.035	0.003	0.045	0.012	0.000	0.016
	FT	1.994	0.745	**0.720**	0.745	**0.213**	**0.270**	**0.255**	0.246	**0.290**	0.294
code-davinci-002	PT	3.885	0.520	0.685	**0.775**	0.175	0.200	0.150	0.156	0.184	**0.344**

cover all values of the shift or assign incorrect bit positions. For Prob. #12, completions are close to the actual solution by using all input values in `assign` statements but fail to form correct expressions between input bits. This suggests insufficient diversity in the training corpus.

Next, we study the impact of the training corpus on LLM fine-tuning. We conduct an ablation study using (a) CodeGen-16B fine-tuned with GitHub verilog repositories only and (b) CodeGen-16B fine-tuned with Verilog from Github and textbooks. The Pass@(*scenario**10) for (a) and (b) show that option (b) is marginally better (1.4%) than (a). This is the case because the Verilog corpus from PDFs adds more examples and this helps the LLM to generalize to Verilog.

VII. CONCLUSIONS

This paper describes a new paradigm for automatically generating and verifying Verilog from LLMs. Using the presented Pass@(*scenario**n) values from Tables III-IV, pre-tuned LLMs produced completions that are functionally correct only 1.09% of the time. This number increases to 27.0% after tuning, showing a clear benefit to fine-tuning LLMs over a specific language. The fine-tuned CodeGen-16B LLM was the most successful in completions with respect to functional correctness. Overall it produced functionally correct code 41.9% of time, whereas the commercially available state-of-the-art (non-fine-tuned) code-davinci-002 LLM produced functionally correct code 35.4% of time.

ACKNOWLEDGEMENTS

This research work was supported in part by NSF Award 1553419, NSF Award 1646671, NSF Award 2039607, and ARO Award 77191NC. The opinions, findings, and conclusions, or recommendations expressed are those of the author(s) and do not necessarily reflect the views of any sponsors.

REFERENCES

[1] G. Dessouky *et al.*, "Hardfails: Insights into Software-Exploitable Hardware Bugs," in *Proc. 28th USENIX Conf. Security Symp.* USENIX Association, 2019, pp. 213–230.

[2] M. Chen *et al.*, "Evaluating Large Language Models Trained on Code," Jul. 2021, arXiv:2107.03374 [cs]. [Online]. Available: http://arxiv.org/abs/2107.03374

[3] H. Pearce *et al.*, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in *2022 IEEE Symp. on Security and Privacy (SP)*, May 2022, pp. 754–768, iSSN: 2375-1207.

[4] H. Pearce, B. Tan, and R. Karri, "DAVE: Deriving Automatically Verilog from English," in *Proc. of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. ACM, Nov. 2020, pp. 27–32. [Online]. Available: https://dl.acm.org/doi/10.1145/3380446.3430634

[5] A. Vaswani *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems 30*, I. Guyon *et al.*, Eds. Curran Associates, Inc., 2017, pp. 5998–6008.

[6] P. Gage, "A New Algorithm for Data Compression," *C Users Journal*, vol. 12, no. 2, pp. 23–38, Feb. 1994.

[7] M. Shoeybi *et al.*, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," Mar. 2020, arXiv:1909.08053 [cs]. [Online]. Available: http://arxiv.org/abs/1909.08053

[8] E. Nijkamp *et al.*, "A Conversational Paradigm for Program Synthesis," Mar. 2022, arXiv:2203.13474 [cs]. [Online]. Available: http://arxiv.org/abs/2203.13474

[9] R. Mihalcea, H. Liu, and H. Lieberman, "NLP (Natural Language Processing) for NLP (Natural Language Programming)," in *Computational Linguistics and Intelligent Text Processing*, A. Gelbukh, Ed. Springer Berlin Heidelberg, 2006, pp. 319–330.

[10] C. B. Harris and I. G. Harris, "GLAsT: Learning formal grammars to translate natural language specifications into hardware assertions," in *Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2016, pp. 966–971.

[11] Z. Yan *et al.*, "PrivMin: Differentially Private MinHash for Jaccard Similarity Computation," May 2017, arXiv:1705.07258 [cs]. [Online]. Available: http://arxiv.org/abs/1705.07258

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," May 2019, arXiv:1810.04805 [cs]. [Online]. Available: http://arxiv.org/abs/1810.04805

[13] A. Radford *et al.*, "Language Models are Unsupervised Multitask Learners," p. 24, 2019. [Online]. Available: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[14] T. Brown *et al.*, "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, H. Larochelle *et al.*, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[15] L. Gao *et al.*, "The Pile: An 800GB Dataset of Diverse Text for Language Modeling," Dec. 2020, arXiv:2101.00027 [cs]. [Online]. Available: http://arxiv.org/abs/2101.00027