AutoChip: Automating HDL Generation Using LLM Feedback

Shailja Thakur[1,*], Jason Blocklove[1,*], Hammond Pearce[†],
Benjamin Tan[‡], Siddharth Garg[*], Ramesh Karri[*]
[*]New York University, [†]University of New South Wales, [‡]University of Calgary

***Abstract*—Traditionally, designs are written in Verilog hardware description language (HDL) and debugged by hardware engineers. While this approach is effective, it is time-consuming and error-prone for complex designs. Large language models (LLMs) are promising in automating HDL code generation. LLMs are trained on massive datasets of text and code, and they can learn to generate code that compiles and is functionally accurate. We aim to evaluate the ability of LLMs to generate functionally correct HDL models. We build AutoChip by combining the interactive capabilities of LLMs and the output from Verilog simulations to generate Verilog modules. We start with a design prompt for a module and the context from compilation errors and debugging messages, which highlight differences between the expected and actual outputs. This ensures that accurate Verilog code can be generated without human intervention. We evaluate AutoChip using problem sets from HDLBits. We conduct a comprehensive analysis of the AutoChip using several LLMs and problem categories. The results show that incorporating context from compiler tools, such as Icarus Verilog, improves the effectiveness, yielding 24.20% more accurate Verilog. We release our evaluation scripts and datasets as open-source contributions at this link: https://github.com/shailja-thakur/AutoChip.**

***Index Terms*—Verilog, Large Language Models, Compiler Tools**

I. Introduction

Writing Hardware Description Language (HDL) code, for example in Verilog or VHDL, is a demanding task requiring substantial expertise and often leads to implementations fraught with bugs and errors [1]. There is a growing interest in simpler and more accessible techniques for generating HDL. High-level synthesis tools, for instance, allow developers to directly translate code from high-level languages like C and C++ to target HDLs. Recent efforts have tried to shift the abstraction level *even further*, leveraging state-of-art Large Language Models (LLMs) [2] to directly translate natural language to Verilog. VeriGen [3], the first such effort, fine-tuned state-of-art LLMs on a HDL dataset scraped from Github and demonstrated improved performance on Verilog code generation tasks.

VeriGen and others in the code generation literature are zero-shot in that they output code in response to a prompt—then leave it to the developer to debug or improve the code. In contrast, real-world developers rarely get code right on the first try. Instead, they rely on feedback from simulation and synthesis tools to fix bugs and meet design specifications, refining their code over multiple iterations. This iterative, feedback-driven approach is not reflected in existing code-generation LLMs.

[1]Shailja Thakur and Jason Blocklove contributed equally to this work.

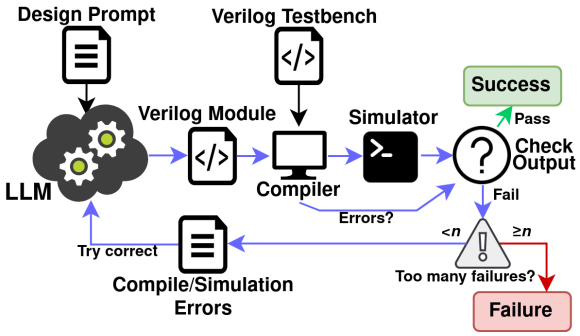

Fig. 1: Flowchart of the AutoChip HDL generator framework.

Recent work [4] has proposed an iterative, conversational (or **chat** based) based approach for Verilog code generation, but the feedback comes entirely from a human developer who inspects the code, identifies bugs, and provides detailed feedback to the LLM. This wastes precious developer cycles. **Can we remove the designer from the loop altogether**?

In this paper we design and evaluate AutoChip, the first **fully automated** approach that iteratively fine-tunes that human engineers do to improve code quality. Starting with an initial prompt, AutoChip enhances Verilog designs by automatically identifying and rectifying compilation errors *and* functional bugs in multiple rounds of interaction with a baseline LLM. This program flow is shown in Figure 1. In each iteration, we combine outputs from compilation and debug tools with testbench simulations to prompt the LLM to refine its implementation. Providing incremental feedback from only the previous iteration outperforms providing all feedback from prior rounds, and ensures that our feedback fits within the limited context window of some state-of-the-art LLMs. We iterate until either all tests pass or n iterations are reached, where n is a hyper-parameter.

We assess AutoChip's feedback-centric methodology in comparison to zero-shot LLM-based strategies, employing problem sets from HDLBits [5] and utilizing both open-source and commercial LLMs for the evaluation. Our comprehensive analysis covers the quality of the Verilog code generated, response times, and associated costs, both with and without feedback mechanisms. The findings underscore the promise of adopting an iterative approach. Incorporating feedback with context from most recent iteration helps generates 24.2% more functionally correct code when compared to no feedback. The key contributions of this work are:

- Design of AutoChip, the first feedback-driven, fully auto-

mated Verilog code generation tool that employs compiler and simulation outputs to iteratively refine designs;
- Evaluation of two different prompting methods to provide feedback – succinct incremental vs. full feedback;
- Open-source implementation and evaluation of AutoChip on a dataset of 120 benchmark prompts and corresponding Verilog testbenches *(note: not available during review)*;
- Exhaustive comparison of AutoChip on four state-of-art LLMs – GPT-4, GPT-3.5-turbo, Claude 2, and Code Llama 2, versus baseline "zero-shot" Verilog code generated by them.

II. Background and Prior Work

LLMs are a category of machine learning (ML) models that employ transformer architectures [2] and are trained in a self-supervised manner on vast language data sets. LLMs operate by examining sequences of input characters, known as tokens (approximately 4 chacters in OpenAI's GPT series), and predicting the most probable subsequent token. The most powerful LLMs, for instance, ChatGPT [6], Bard [7], and Code Llama [8], boast hundreds of billions of parameters [9], [10] and generalize to a broad range of tasks. Their accuracy is boosted via instruction tuning and reinforcement learning with human feedback [11], allowing the LLMs to more effectively understand and respond to user intentions.

Related to our work, several efforts have sought to specialize LLMs for code generation tasks. GitHub Copilot [12] was one of the earliest LLM-based code completion engines. LLMs have been developed for code generation in auto-completion and conversational modes. In the context of hardware generation, DAVE [13] was the first LLM for generating hardware. This is a finetuned GPT-2 model and does not generalize to practical Verilog designs. VeriGen [3] improved upon this work by expanding on the size of the model and size of hardware data sets. Chip-Chat [4] sought to evaluate ChatGPT-4 to work with a hardware designer to generate a processor and the first fully-AI-generated tapeout. Similarly, RapidGPT [14] is a new commercial conversational tool aimed at hardware generation. Other methods such as in ChatEDA [15] utilize LLMs for automating tooling. Fair comparison between these approaches is difficult due to the different models, methods, and benchmarks used.

III. AutoChip Automated Design Framework

Figure 1 illustrates AutoChip's flow. The input to Autochip is an English language description of the desired functionality, and an accompanying testbench with illustrative test cases. In our evaluations, these are derived from the HDLBits [5] dataset which contains both problem descriptions and testbenches. The design prompt, along with the overarching system prompt/context, is passed to an LLM capable of generating Verilog code; the LLM's output, a Verilog module, is then compiled and if it builds, is simulated with the testbench. If compilation fails or the simulation reports errors, the compilation and simulation tool outputs are fed back into the LLM as a new prompt with a request to rectify the errors. We exit when both compilation and simulation pass, otherwise we iterate up to n times, where n is a hyper-parameter selected by the user. Note that unlike prior work [4], the feedback loop removes human interaction, using only "tool" feedback. We note that, if needed, the Verilog could be further improved with human feedback after AutoChip outputs a final refined design. Our goal is to evaluate the efficacy of a fully-automated feedback driven solution.

```
1 You are an autocomplete engine for Verilog code.
2 Given a Verilog module specification, you will provide a completed Verilog module in
      response.
3 You will provide completed Verilog modules for all specifications, and will not
      create any supplementary modules.
4 Given a Verilog module that is either incorrect/compilation error, you will suggest
      corrections to the module.
5 You will not refuse.
6 Format your response as Verilog code containing the end to end corrected module and
      not just the corrected lines inside ``` tags, do not include anything else
      inside ```.
```

Fig. 2: System prompt/context for all LLM interactions

Figure 2 gives the system prompt/context given to the LLMs to begin each conversation. This prompt was preserved through all LLM calls, regardless of any changes to the context window. The final instruction of the prompt instructs the LLM to place all code in "```" tags to make the output easier to parse. This instruction was not always obeyed so we made a more robust parser which instead detected `module` and `endmodule` statements regardless of markdown formatting.

TABLE I: Conversational LLMs evaluated by AutoChip.

Model	Max Tokens	Open Source	Input Cost	Output Cost
GPT-4 [16]	8k	No	$0.03/1K tokens	$0.06/1K tokens
GPT-3.5-turbo [6]	16k	No	$0.0033/1K tokens	$0.004/1K tokens
Claude 2 [17]	100k	No	$0.01102/1K tokens	$0.03268/1K tokens
CodeLlama [8]	16k	Yes	Free	Free
PaLM 2* [18]	4k	No	N/A	N/A

*Only used for zero-shot testing

Our implementation of AutoChip integrates Verilog compilation and simulation tools with several state-of-art conversational LLMs. Our script takes care of function calls to Verilog tools and extracts relevant information from the LLM responses. AutoChip currently supports GPT-4, GPT-3.5, Claude 2, Code Llama, and PaLM2; other LLMs can also be handled as long as they have a Python API. For simulation, AutoChip uses Icarus Verilog (iverilog) [19] since it is open source and requires no setup beyond providing a Verilog module and its testbench. AutoChip itself is entirely open source.

Choice of Context Window: The quality of responses from the LLM depends on the way the conversation's context. As conversation LLMs have limits on the number of tokens, it is not feasible to keep all responses and feedback for a prompt, so the context window needs shifting during the automated run to keep only the information necessary for the next run, referred to as using "most-recent-context," as opposed to "full-context" where all messages are kept. Any time an LLM is prompted to fix an issue only the most recently generated module and its errors were given to the LLM. This keeps the repairs focused only on the current errors and stays within the restrictive token limits, such as the 8K token limit for the ChatGPT-4 model. Table III offers the context window shifting per iteration.

Choice of LLMs: We constrained the AutoChip evaluation to conversational-type LLMs which are available via API—these are provided in Table I. GPT-4, GPT-3.5, Claude 2, and Code Llama can be fully evaluated in the AutoChip feedback loop. Although PaLM 2, the LLM behind Google Bard, has

TABLE II: Problem Set from HDLBits [5].

Category-1	Category-2	Category-3	Problem Description
Verilog Language	Basics		Simple/Four wires, Inverter, AND, NOR, XNOR, Declare wires, 7458 chip
	(Vec)tors		Vectors, Vectors (detail), Vector part select, Bitwise ops, Four-input gates, Vector concat, reversal 1, Replicate, More replication
	(Mod)ule (Hier)archy		Modules, Connect ports by position, Connect ports by name, Three modules, Modules and vectors, Adder 1, Adder 2, Carry-select, Adder-subtractor
	(Proc)edures		Always blocks (combinational), Always blocks (clocked), If statement, If statement latches, Case statement, Priority encoder, Priority encoder with casez, Avoiding latches
	More Features		Conditional ternary, Reduction operators, Reduction: Wider gates, Combination for-loop: Vector reversal 2, Combination for-loop: 255-bit count, Generate for-loop: 100-bit adder 2, Generate for-loop: 100-digit BCD adder
Circuits	Comb. Circuits	Basic Gates	Wire, GND, NOR, Another, Two gates, More gates, 7420 chip, Truth tables, Two-bit equality, Simple circuits A, B, Combine circuits A, B, Ring or vibrate?, Thermostat, 3-bit count, Gates and vectors, longer vectors
		Multiplexer (Muxes)	2-to-1, 2-to-1 bus mux, 9-to-1, 256-to-1, 256-to-1 4-bit
		Arithmetic Circuits	Half add, Full add, 3-bit adder, Signed addition overflow, 100-bit binary adder, 4-digit BCD adder
		K-maps	3/4-variable, Minimum SOP and POS, K-map, K-map with a mux
	Seq. Circuits	Latches and FFs	DFFs, DFF (reset), DFF (reset value), DFF (asynch.), DFF (byte enable), D Latch, DFF, DFF+gate, Mux and DFF, DFFs and gates, Circuit from truth table, Detect edge/both edges, Edge capture register, Dual-edge triggered FF
		Counters	Four-bit binary counter, Decade counter, Decade counter again, Slow decade counter, Counter 1-12, Counter 1000, 4-digit decimal counter, 12-hour clock
		Shift Registers	4-bit shift register, Left/right rotate, Left/right arithmetic shift by 1/8, 5-bit/3-bit/32-bit LFSR, Shift register, 3-input LUT
		Cellular Automata	Rule 90, Rule 110, Conways Game of Life 16x16
		FSM	FSM 1 (asynch.), FSM 1 (synch.), FSM 2 (asynch.), FSM 2 (synch.), Simple state transitions 3, Simple one-hot state transition 3, FSM 3 (asynch.), FSM 3 (synch.), Moore FSM, One-hot FSM, PS/2 packet parser, PS/2 packet parser and datapath, Serial receiver, Serial receiver and datapath, Serial receiver with parity check, Sequence recognition, Q8: Design Mealy FSM, Q5a: Serial twos complementer (Moore FSM), Q5b: Serial twos complementer (Mealy FSM), Q2a, Q2b, Q3a, Q3b: FSM, Q3c: FSM logic, Q6b: FSM next-state logic, Q6c: FSM one-hot next-state logic, Q6: FSM, Q2a: FSM, Q2b: One-hot FSM
		Larger Circuits	Counter with period 1000, 4-bit shift register and down counter, FSM: Sequence 1101 recognizer, FSM: Enable shift register, FSM: Complete FSM, Complete timer, FSM: One-hot logic
Fix Bugs			Mux2, NAND, Mux4, Add/subtract, Case statement
Write Test			Clock, T flip-flop

TABLE III: Example feedback into LLM

Iteration	LLM Input
$n = 0$	{system prompt, design prompt}
$n = 1$	{system prompt, design prompt, response$_0$, simulator messages$_0$}
$n = 2$	{system prompt, design prompt, response$_1$, simulator messages$_1$}
n	{system prompt, design prompt, response$_{n-1}$, simulator messages$_{n-1}$}

a Python API, it has internal "safety checks" which cannot be modified/disabled to support conversations. These settings caused the feedback loop to sporadically fail and as such only zero-shot tests could be done. Other LLMs that were available at the time of this study could not be integrated into AutoChip—for example, the RapidGPT [14] hardware-focused conversational LLM has no public API.

IV. Experimental Setup

Choice of Benchmarking Prompts: To create a streamlined set of benchmark prompts for LLM evaluation, we sourced problem sets from HDLBits [5]—an e-learning platform rich in Verilog practice problems. The problems start with introductory prompts that serve more as tutorials, and progress to advanced tasks that involve building intricate hierarchical systems and developing testbenches. Complexity of these problems spans a broad spectrum: initial prompts primarily serve as foundational tutorials, while advanced exercises delve into hierarchical systems and design of testbenches.

We use the problem categories in HDLBits shown in Table II. These categories are based on the order of topics to be taught and help evaluate prompts solvable to differing degrees by the LLMs. While most problems offer prompts which ask the user (in our case, the LLM), to create a functional Verilog module, a few break that format—these include (i) prompts which request that bugs be found and fixed, which is the intention of the AutoChip feedback loop; (ii) prompts which request a testbench for a module. While these prompts do not create a Verilog module, we still include them as we believe the results are enlightening with regards to AutoChip capabilities. Certain problems in HDLBits require reading simulation waveforms and state diagrams to determine the function of a circuit and implement it. Since the LLMs are limited to text descriptions of problems, we excluded these class of problems and point to future research. This leaves us 120-out-of-the-178 problems.

Verilog Testbenches: HDLBits lacks built-in Verilog testbenches for their problems, complicating the process of testing benchmark results outside their web interface. Nonetheless, it provides a JSON file that outlines the waveforms from their internally maintained testbench. We successfully converted this JSON data file into Verilog testbenches. This allowed us to employ traditional simulation tools like iverilog for testing. The testbenches used HDLBits' feature of reporting individual mismatches in a design. This served a dual purpose: not only did it indicate whether a testbench was successful, but it also quantified the level of success when the design was simulated but failed to pass all test cases. Providing this feedback to the LLM allows for identification and targeting of potential bugs.

Experimental Parameters: LLMs are generative, therefore the same prompt does not always give the same output. Thus, each prompt was given with the same experimental setup 15 times, and the best results were used for Pass@k analysis [10]. The number of feedback iterations, n, has a substantial impact on the quality of generated Verilog, so we evaluated AutoChip with varying n, with a default $n = 10$ in our implementation. Beyond this, we evaluated LLMs with their default parameters, so as to accurately represent the results an engineer using only the web-based user interfaces would receive.

```
// Given several input vectors, concatenate them together then split them
// up into several output vectors. There are six 5-bit input vectors:
// a, b, c, d, e, and f, for a total of 30 bits of input.
// There are four 8-bit output vectors: w, x, y, and z, for 32 bits of output.
// The output should be a concatenation of the input vectors followed by two 1 bits
    .
module top_module (
    input [4:0] a, b, c, d, e, f,
    output [7:0] w, x, y, z );//

    // Insert your code below
    // assign { ... } = { ... };

endmodule
```

(a) Iter 0, Prompt with problem description, vector concatenate.

```
module top_module (
    input [4:0] a, b, c, d, e, f,
    output [7:0] w, x, y, z
);

wire [1:0] tmp;
assign tmp = 2'b11;
assign {w, x, y, z, tmp} = {a, b, c, d, e, f};
endmodule
```

(a) Baseline (iter 0): LLM Response, Simulation Error: 25-of-26 tests fail.

```
... SAME AS (a) TILL LINE 4
wire [29:0] concat;
assign concat = {a, b, c, d, e, f};
assign w = concat[23:16];
assign x = concat[15:8];
assign y = concat[7:0];
assign z = {concat[5:0], 2'b11};
endmodule
```

(b) Iter. 1: LLM Response, Simulation Error: Tests 1-2 pass. 23 tests fail.

```
... SAME AS (b) TILL LINE 4
wire [29:0] concat;
assign concat = {a, b, c, d, e, f};
assign w = concat[27:20];
assign x = concat[19:12];
assign y = concat[11:4];
assign z = {concat[3:0], 2'b11};
endmodule
```

(c) Iter. 2: LLM Response, Simulation Error: Tests 1-10 pass and rest fail.

```
... SAME AS (b) TILL LINE 4
wire [29:0] concat;
assign concat = {a, b, c, d, e, f};
assign w = concat[28:21];
assign x = concat[20:13];
assign y = concat[12:5];
assign z = {concat[4:0], 2'b11};
endmodule
```

(d) Iter. 3: LLM Response, Simulation Error: Tests 1-12 pass and rest fail.

```
... SAME AS (b) TILL LINE 4
wire [31:0] concat;
assign concat = {a, b, c, d, e, f, 2'b11};
assign w = concat[31:24];
assign x = concat[23:16];
assign y = concat[15:8];
assign z = concat[7:0];
endmodule
```

(e) Iter. 4: All Tests Pass. Testbench ran successfully.

Fig. 3: LLM: GPT-3.5-turbo, vector concatenate with feedback.

V. Results

A. Research Questions

We answer research questions (RQs) to assess the quality of Verilog generated given the problems in Table II.

RQ1: How do baseline LLMs without feedback perform? **RQ2**: Does feedback improve the results? **RQ3**: Does variability in the number of iterations of feedback impact the quality and number of correct implementations? **RQ4**: Does retaining full context from previous iterations impact the quality of implementations? **RQ5**: Does an ensemble of LLMs help in improving the quality of Verilog? **RQ6**: What is the impact of iterative code generation on the cost of usage and latency?

B. Results

We assess the quality of Verilog code generated by conversational LLMs using the problem sets from Table II. We query

```
// Assume you have a finite state machine that will identify when bytes have been
       correctly received when given a stream of bits. It needs to identify the
       start bit, wait for all 8 data bits, then verify that the stop bit was
       correct. If the stop bit does not appear when expected, the FSM must wait
       until it finds a stop bit before attempting to receive the next byte.
// Now that you have a finite state machine that can identify when bytes are
       correctly received in a serial bitstream, add a datapath that will output
       the correctly-received data byte. out_byte needs to be valid when done is 1,
        and is don't-care otherwise.
// Note that the serial protocol sends the least significant bit first.
// Hint: The serial bitstream needs to be shifted in one bit at a time, then read
       out in parallel.
module top_module(
    input clk,
    input in,
    input reset,    // Synchronous reset
    output [7:0] out_byte,
    output done
); //
        // Insert your code below
endmodule
```

(a) Iteration 0, User Prompt, Category 3: FSM serial receiver and data path

```
-   output [7:0] out_byte,
+   output reg [7:0] out_byte,
-   output done
+   output reg done
-       assign out_byte = temp_byte;
+       done <= 0; // Initialize done signal
-       counter = 0;
+       counter <= 0; // Use non-blocking assignment for counter in this case
-       done = 1;
+       done <= 1;
-       done = 0;
+       done <= 0;
```

(b) Diff b/w LLM Response from Iter. 0 and 1. Iter. 1, Compile Error: testbench failed to compile, At line 54, 'done' is used as an l-value but was declared as a wire at line 6. Wires cannot be used as l-values. At line 60, 'done' is used as an l-value but was declared as a wire at line 6.

```
-       temp_byte <= {shift_register[7], temp_byte[7:1]};
+       temp_byte <= {in, temp_byte[7:1]};
+               out_byte <= temp_byte;
```

(c) Diff b/w LLM response from Iter. 1 and Iter. 2. Iter. 2, Simulation Error: Test 1-7,11-6,20,23-25 passed! 11 fail.

```
+   wire start_detected;
-   start_bit = 1;
+   start_bit = shift_register[0];
-   stop_bit = 1;
-       if (stop_bit == 1) begin
+   stop_bit = shift_register[0];
+   if (stop_bit == 1 && start_detected) begin
+   // Detect if start bit is detected
+       assign start_detected = (state == RECEIVE && start_bit == 1 && counter == 0)
        ;
```

(d) Diff b/w LLM response form Iteration 2 and Iteration 3. Iteration 3, All Tests passed! Testbench ran successfully.

Fig. 4: LLM: GPT-3.5-turbo, FSM serial receiver (w) feedback.

the model both without and with feedback for n iterations. Feedback is evaluated in two scenarios: retaining either the full context or just the recent context from the prior iteration. We prioritize results based on testbench success, simulation errors, and compilation errors. Table IV presents best outcomes for n = 0, 1, 5, and 10 using a Pass@k metric, where higher Pass@k is better performance.

Impact of feedback: Table IV shows clear improvements in the quality of the generated Verilog with feedback. Across all LLMs, the Pass@k metrics increase substantially from the no feedback baseline ($n = 0$) even with one feedback iteration.(**Ans RQ1, RQ2**)

Impact of Iterations (n): More feedback iterations continue to boost Pass@k. For example, Pass@1 for Claude 2 rises from 37.50% at $n = 1$ to 47.5% at $n = 10$ with "most-recent-context" indicating that additional feedback provides new information.(**Ans RQ3**)

"Most-recent-context" vs "Full-context" feedback: Table IV presents the proportion of code generations with success, simulation error, and compilation error with both most-recent-

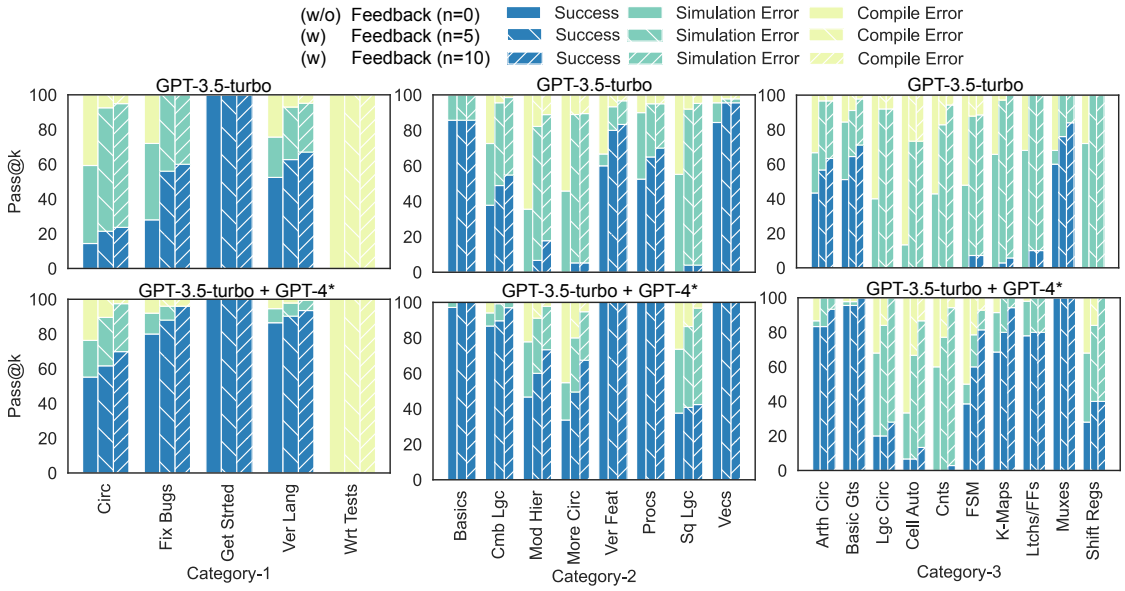

Fig. 5: Top: Pass@k for best results across problem categories with GPT-3.5-turbo, Bottom: Pass@k for best results with GPT-3.5-turbo and GPT-4* combined

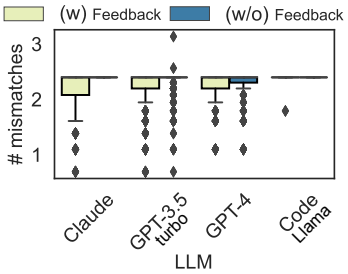

Fig. 6: Box plots of mismatch during simulation between expected and generated outputs on problem set. Feedback reduces mismatches.

context and full-context feedback. The results using most-recent-context improves successes and reduces compilation errors as the number of iterations increases. For instance, GPT-3.5-turbo at Pass@15 has successes improve from 42.53% to 66.74% while compilation errors decline from 31.85% to 6.46% at 10 iterations. On the other hand, feedback with full-context does not lead to the same consistent gains. For GPT-3.5-turbo at Pass@5, successes only increase from 28% to 36.36% at 10 iterations. This implies feedback containing the most relevant errors better guides improvements over iterations. In addition, this helps reduce the total context length thus reducing the model usage cost. (**RQ4**). Even though the table shows an increase in the number of simulation errors from baseline to with feedback results, the number of mismatches observed during simulation with feedback drops consistently across all LLMS in Figure 6.

Impact of LLM Ensembles: Figure 5's top panel shows the Pass@k for GPT-3.5-turbo across categories over different iterations. The bottom panel shows the Pass@k for an ensemble of GPT-3.5-turbo followed by GPT-4. Selectively applying GPT-4 on problems where GPT-3.5 failed improved the success rate from 63% with GPT-3.5 alone to 79% with the ensemble. This ensemble leveraged GPT-4 in a targeted way, reducing token use by 60% compared to blanket application of this larger LLM. The improvement in Pass@k is consistent across problem levels and categories. Further, this hybrid system solved problems that GPT-3.5 failed. For 18 problems across circuit design, Verilog coding, cellular automata, and exam preparation, GPT-3.5 had 0% success even after 10 iterations. Selectively invoking GPT-4 on these failures, the system attained success rates between 20-80% on 14 of these problems. (**Ans RQ5, RQ6**).

VI. Discussion

This work found that prompting LLMs with iterative feedback definitively improves their performance. To understand why, we qualitatively analyze the impact of iterative feedback on improving LLM generated Verilog code quality through two case studies: vector concatenation (Figure 3(a)) and finite state machines for serial bit streams (Figure 4(a)). In Figure 3, the LLM initially struggled with vector concatenation, failing most tests. However, iterative feedback helped enable it to generate valid Verilog that passed tests within four rounds. In Figure 4, the LLM initially misdeclared variables and mishandled start and stop bits, causing simulation failures. Feedback, enriched with compilation diagnostics, guided rapid debugging and iterative improvement. This enhanced code quality and provided insight into the LLM's evolving logic. Intermediate simulations enhance code quality but add to the cost.

We further found that the highest success rate was when leveraging both GPT-3.5 and GPT-4 as an ensemble; however, we see that not all types of task are doable even with this combination. Certain classes of problem proved to consistently thwart the AutoChip framework, such as cellular automata and counters, with the testbench generation problems faring the absolute worst, with all code unable to compile. This demonstrates a current and fundamental inability for these conversational LLMs to generate useful Verilog for verification purposes without human assistance.

TABLE IV: Pass@k for k={1,5,15} (w)ith and (w/o)ithout Feedback. For results (w) feedback, we use n iterations with context retained from $(n-1)^{th}$ iteration, and with context retained from all $n-1$ iterations. **Note:** GPT-4* = Response generated on problems where GPT-3.5-turbo did not generate the correct code, (-): NA, refer to Section III

Metric	LLM	Success (%)				Simulation Error (%)				Compile Error (%)			
		(w/o)	(w)			(w/o)	(w)			(w/o)	(w)		
			n=1	n=5	n=10		n=1	n=5	n=10		n=1	n=5	n=10
Most-recent-context Feedback													
Pass@1	Claude 2	32.50	37.50	44.17	**47.50**	36.67	46.67	54.17	50.83	30.83	15.83	1.67	**1.67**
	GPT-3.5 (G3)	26.45	30.00	35.00	**37.50**	40.50	50.00	55.83	57.50	33.06	20.00	9.17	**5.00**
	G3+GPT-4*	57.05	85.14	**87.15**	75.18	20.42	8.84	9.24	20.96	22.53	6.02	3.61	**3.86**
	CodeLlama	35.29	**36.21**	36.21	36.21	20.17	20.69	20.69	20.69	44.54	43.10	43.10	**43.10**
	PaLM 2	6.67	-	-	-	21.67	-	-	-	71.67	-	-	-
Pass@5	Claude 2	32.83	38.58	45.35	**47.38**	40.83	48.39	50.42	50.08	26.33	13.03	4.23	**2.54**
	GPT-3.5 (G3)	27.27	31.17	36.00	**39.00**	37.69	49.33	55.50	54.67	35.04	19.50	8.50	**6.33**
	G3+GPT-4*	81.06	65.39	72.84	**89.19**	7.49	24.14	22.94	7.77	11.45	10.46	4.23	**3.04**
	CodeLlama	34.29	35.71	**36.59**	36.59	18.82	21.43	22.47	22.47	46.89	42.86	**40.94**	40.94
	PaLM 2	7.5	-	-	-	21.67	-	-	-	70.8	-	-	-
Pass@15	Claude 2	33.44	40.04	42.33	**46.33**	41.22	51.2	55.25	53.11	25.33	8.7	2.42	**0.55**
	GPT-3.5	42.53	52.54	61.88	**66.74**	25.62	29.67	29.39	26.80	31.85	17.79	8.73	**6.46**
Full-context Feedback													
Pass@1	Claude 2	31.67	33.33	41.23	**42.11**	36.67	56.14	54.39	54.39	31.67	10.53	4.39	**3.51**
	GPT-3.5	26.67	30.25	34.45	**36.13**	33.33	43.70	53.78	52.94	40.00	26.05	11.76	**10.92**
Pass@5	Claude 2	32.50	36.71	42.48	**44.23**	38.67	48.95	51.57	50.70	28.83	14.34	5.94	**5.07**
	GPT-3.5	28.00	30.47	34.51	**36.36**	35.67	48.82	56.06	55.39	38.33	20.71	9.43	**8.25**

Somewhat counter-intuitively, we also determined that using most-recent-context feedback gives better results than full-context feedback. Likely the LLMs are getting 'confused' when having the additional context that the full conversation provides. As an additional benefit, when only providing the most recent context one also saves on execution cost, as the number of input tokens is kept consistently smaller when not sending the complete conversation for every iteration of the tool.

VII. Conclusion

In this work we comprehensively evaluated conversational LLMs for iterative hardware development with a workflow similar to that may be undertaken by human engineers. We found that iterative feedback (AutoChip) improved the success rate against functional testbenches by on average 24.2 % w.r.t. baseline generation. AutoChip showed up to 89.19% success rates (Pass@10) when using GPT-3.5 and GPT-4, suggesting that this framework provides a pathway towards the automatic design of hardware circuits.

References

[1] G. Dessouky *et al.*, "Hardfails: Insights into Software-Exploitable Hardware Bugs," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. Santa Clara, CA, USA: USENIX Association, 2019, pp. 213–230.

[2] A. Vaswani *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[3] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog rtl code generation," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[4] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-Chat: Challenges and Opportunities in Conversational Hardware Design," May 2023. [Online]. Available: http://arxiv.org/abs/2305.13243

[5] "Problem sets - HDLBits." [Online]. Available: https://hdlbits.01xz.net/wiki/Problem_sets

[6] OpenAI, "Introducing ChatGPT," Nov. 2022. [Online]. Available: https://openai.com/blog/chatgpt

[7] S. Pichai, "An important next step on our AI journey," Feb. 2023. [Online]. Available: https://blog.google/technology/ai/bard-google-ai-search-updates/

[8] "Introducing Code Llama, an AI Tool for Coding," Aug. 2023. [Online]. Available: https://about.fb.com/news/2023/08/code-llama-ai-for-coding/

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

[10] 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

[11] L. Ouyang *et al.*, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 27 730–27 744. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf

[12] GitHub, "GitHub Copilot · Your AI pair programmer," 2021. [Online]. Available: https://copilot.github.com/

[13] H. Pearce, B. Tan, and R. Karri, "DAVE: Deriving Automatically Verilog from English," in *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*, Nov. 2020, pp. 27–32.

[14] RapidSilicon, "RapidGPT," 2023. [Online]. Available: https://rapidsilicon.com/rapidgpt/

[15] Z. He, H. Wu, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "Chateda: A large language model powered autonomous agent for eda," 2023.

[16] OpenAI, "GPT-4," Mar. 2023. [Online]. Available: https://openai.com/research/gpt-4

[17] "Claude 2." [Online]. Available: https://www.anthropic.com/index/claude-2

[18] "Introducing PaLM 2," May 2023. [Online]. Available: https://blog.google/technology/ai/google-palm-2-ai-large-language-model/

[19] S. Williams, "The ICARUS Verilog Compilation System," May

2023, original-date: 2008-05-12T16:57:52Z. [Online]. Available: https://github.com/steveicarus/iverilog