arXiv:2305.13243v1 [cs.LG] 22 May 2023

Chip-Chat: Challenges and Opportunities in Conversational Hardware Design

Jason Blocklove
New York University
New York, NY USA
jason.blocklove@nyu.edu

Siddharth Garg
New York University
New York, NY USA
siddharth.garg@nyu.edu

Ramesh Karri
New York University
New York, NY USA
rkarri@nyu.edu

Hammond Pearce
University of New South Wales
Sydney, Australia
hammond.pearce@unsw.edu.au

***Abstract*—Modern hardware design starts with specifications provided in natural language. These are then translated by hardware engineers into appropriate Hardware Description Languages (HDLs) such as Verilog before synthesizing circuit elements. Automating this translation could reduce sources of human error from the engineering process. But, it is only recently that artificial intelligence (AI) has demonstrated capabilities for machine-based end-to-end design translations. Commercially-available instruction-tuned Large Language Models (LLMs) such as OpenAI's ChatGPT and Google's Bard claim to be able to produce code in a variety of programming languages; but studies examining them for hardware are still lacking. In this work, we thus explore the challenges faced and opportunities presented when leveraging these recent advances in LLMs for hardware design. Using a suite of 8 representative benchmarks, we examined the capabilities and limitations of the state of the art conversational LLMs when producing Verilog for functional and verification purposes. Given that the LLMs performed best when used interactively, we then performed a longer fully conversational case study where a hardware engineer co-designed a novel 8-bit accumulator-based microprocessor architecture. We sent the benchmarks and processor to tapeout in a Skywater 130nm shuttle, meaning that these 'Chip-Chats' resulted in what we believe to be the world's first wholly-AI-written HDL for tapeout.**

***Index Terms*—Hardware Design, CAD, LLM**

I. INTRODUCTION

A. Trends in hardware design

As digital designs continue to grow in capability and complexity, software components in Integrated Circuit (IC) Computer Aided Design (CAD) have adopted machine learning (ML) throughout the Electronic Design Automation flow (e.g. [1]–[3]). Where traditional approaches try to formally model each process, ML-based approaches focus on identifying and exploiting generalizable high-level features or patterns [3]—meaning ML can augment or even replace certain tools. Still, ML research in IC CAD tends to focus on the back-end processes such as logic synthesis, placement, routing, and property estimation. In this work, we instead explore the challenges and opportunities when applying an emerging type of ML model to the earliest stages of the hardware design processes: the writing of Hardware Description Language (HDL) itself.

B. Automating Hardware Description Languages (HDLs)

While hardware designs are expressed in formal languages (HDLs), they actually begin the design lifecycle as specifications provided in natural language (e.g. English-language requirements documents). The process of translating these into the appropriate HDL (e.g. Verilog) must be done by hardware engineers, which is both time-consuming and error-prone [4]. Alternative pathways such as using high-level synthesis tools [5] can enable developers to specify functionality in higher-level languages like C, but these methods come at the expense of hardware efficiency. This motivates the exploration of Artificial Intelligence (AI) or ML-based tools as an alternative pathway for translating specifications to HDL.

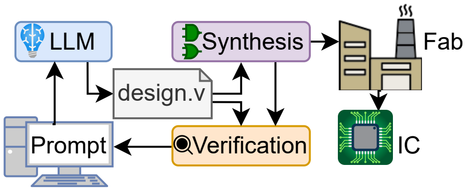


Fig. 1. Can conversational LLMs be used to iteratively design hardware?

The obvious candidate for this machine translation application comes from the Large Language Models (LLMs) [6] popularized by commercial offerings such as GitHub Copilot [7]. LLMs claim to produce code in a variety of languages and for a variety of purposes. Still, they focus on software, and benchmarks for these models evaluate them for languages such as Python, rather than on the needs present in the hardware domain. As such, adoption by the hardware design community continues to lag behind that in the software domain. Although steps for benchmarking the 'autocomplete' style models have begun to appear in the literature [8], the latest LLMs such as OpenAI's ChatGPT [9] and Google's Bard [10] instead provide a different 'conversational' chat-based interface to their capabilities.

Therefore, we pose the following question: What are the potential advantages and obstacles associated with integrating these tools into the HDL development process (Figure 1)? We perform two conversational experiments. The first experiment involves predefined conversation flows and a series of benchmark challenges (Section III), while the second one entails an open-ended "free chat" approach, where an LLM serves as a co-designer in a larger project (Section IV).

In order to comprehend the significance of this emerging technology, it is crucial to conduct observational studies like this one. Similar studies are being carried out for ChatGPT in various domains, including healthcare [11], software [12], and education [13]. Our investigation into the impact of conversational LLMs on hardware design is both relevant and timely.

C. Contributions

Our contributions include the following:

- Conducting the first investigation into the use of conversational LLMs in Hardware design.
- Developing benchmarks to evaluate the capabilities of LLMs for functional hardware development and verification.
- Conducting an observational study on the end-to-end co-design of a complex application in Hardware, utilizing ChatGPT-4.
- Achieving a significant milestone by using AI to write the complete HDL for a tapeout for the first time.
- Providing practical recommendations for the effective utilization of cutting-edge conversational LLMs in hardware-related tasks.

Open-source: All benchmarks, tapeout toolchain scripts, generated Verilog and LLM conversation logs are provided on Zenodo [14].

II. BACKGROUND AND RELATED WORK

A. Large Language Models (LLMs)

Large Language Models (LLMs) are built with Transformer [15] architectures. Early examples include BERT [16] and GPT-2 [17], but it wasn't until the GPT-3 [18] family of models that the relative capabilities of these models became apparent. These include Codex [6], which has billions of learned parameters and is trained on millions of open-source software repositories. In the state of the art there are dozens of LLMs, open-source, closed-source, and commercial, with options for general and task-specific applications.

Still, all LLMs share commonalities. They act as 'Scalable sequence prediction models' [6], meaning that given some 'input prompt' they will output the 'most likely' continuation of that prompt (think of them as a 'smart autocomplete'). For this I/O, they use *tokens*, which are common character sequences specified using byte pair encoding.This is efficient as LLMs have a fixed context size, meaning that they can ingest more text than they could by operating over characters. For OpenAI's models, each token represents about 4 characters, and their context windows range up to 8,000 tokens in size (meaning they can support about 16,000 characters of I/O).

B. Large Language Models for hardware design

The first work exploring LLMs for use in the hardware domain was by Pearce et al. [19]. They fine-tuned a GPT-2 model (that they termed DAVE) over synthetically generated Verilog snippets and evaluated the model outputs lexically for 'undergraduate-level' tasks. However, due to the limited training data, the model does not generalize to unfamiliar tasks. Thakur et al. [8] extended this idea, exploring both how model performance for generating Verilog could be evaluated rigorously and using different strategies for training Verilog-writing models. Other works have investigated the implications of such models: [20] examined the incidence rates of 6 types of hardware bugs in Verilog code by GitHub Copilot, and when [21] explored if automated bug repair could be achieved using the Codex models, they also included two hardware CWEs in Verilog.

In the industry, there is also increasing interest: Efabless has recently-announced the AI Generated Design Contest [22] with judges from companies such as Qualcomm and Synopsys. New companies like RapidSilicon are promoting upcoming (but not yet released) tools like RapidGPT [23] which will work in this space.

C. Instruction-tuned 'conversational' models

Recently, a new kind of training methodology,'Reinforcement Learning with Human Feedback (RLHF)' [24], has been applied to LLMs. By combining this with labelled data for specific intents, one can produce instruction-tuned models more capable of following a user's intent. Where previous LLMs focused on 'autocompletion', they can be instead trained to 'follow instructions'. Methodologies not requiring the (non-scalable) human feedback have followed [25]. These can then be fine-tuned to better focus on *conversational* style interactions. Models such as ChatGPT [9] (including ChatGPT-3.5 and ChatGPT-4 versions), Bard [10], and HuggingChat [26] were all trained using these techniques. They provide an exciting new potential interface for works in the hardware domain. However, to the best of the authors knowledge, no such application has yet been explored.

III. EXPLORING 'SCRIPT'ED BENCHMARKS

A. Overview

There is essentially an infinite number of ways to 'chat' with the conversational models. To explore the potential for a 'standardized' and 'automatable' flow using conversational LLMs, we define a rigid, 'scripted' conversation flow over a series of benchmarks. Using consistent metrics we then evaluate a series of LLMs, determining the relative success or failure of a conversation based on the level of instruction needed to pass accompanying testbenches. However, while the conversation flow was kept structurally identical, it inherently has some variation between test runs based on the evaluator needing to decide (a) what feedback was necessary in each step and (b) how that human feedback is to be formatted.

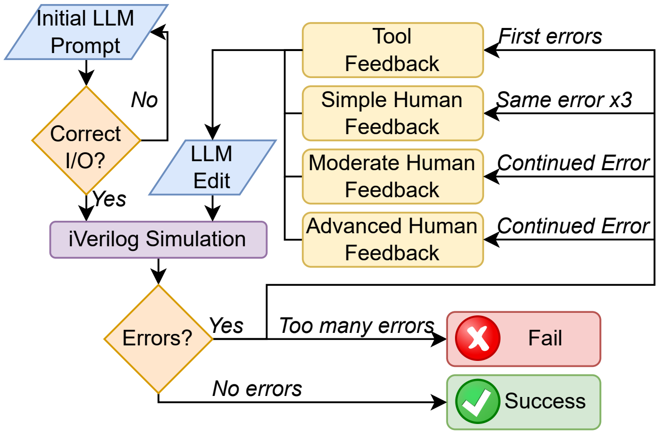


Fig. 2. Simplified LLM conversation flowchart

B. Methodology

Conversation flow: Figure 2 details the general flow of the conversations with the LLMs for creating the hardware benchmarks. The initial prompts detailed in Figure 3 and Figure 4 are first given to the tool. The output design is then visually evaluated to determine if it meets the basic design specification. If a design does not meet the specifications it is regenerated with the same prompt up to five times, after which it fails if it still does not meet the specifications.

Once the design and testbench have been written, they are compiled with Icarus Verilog (iverilog) [27] and, if the compilation succeeds, simulated. If no errors are reported then the design passes with no feedback necessary (NFN). If instead either of those actions report errors they are fed back into the model and it is asked to "Please provide fixes.", referred to as tool feedback (TF). If the same error or type of error appears three times then simple human feedback (SHF) is given by the user, usually by stating what type of problem in Verilog would cause this error (e.g. syntax error in declaring a signal). If the error continues, moderate human feedback (MHF) is given with slightly more directed information being given to the tool to identify the specific error, and if the error persists then advanced human feedback (AHF) is given which relies on pointing out precisely where the error is and the method of fixing it. Once the design compiles and simulates with no failing test cases, it is considered a success. If however advanced feedback does not fix the error or should the user need to write any Verilog to address the error, the test is considered a failure. The test is also considered a failure if the conversation exceeds 25 messages, matching the OpenAI rate limit on ChatGPT-4 messages per 3 hours.

Special circumstances needed to be taken into account within the conversations. Due to the limits placed on how much output a model could give in a single response, files or explanations would often be cut off from finishing; in those cases the model would be prompted with "Please continue". The code following a continue often started from before the final line of the earlier message, so when the code

```
I am trying to create a Verilog model for a shift register. It must meet the following specifications:

- Inputs:
        - Clock
        - Active-low reset
        - Data (1 bit)
        - Shift enable
- Outputs:
        - Data (8 bits)

How would I write a design that meets these specifications?
```

Fig. 3. 8-bit shift register: Design prompt

```
Can you create a Verilog testbench for this design? It should be self-checking and made to work with iverilog for simulation and validation. If test cases should fail, the testbench should provide enough information that the error can be found and resolved.
```

Fig. 4. Testbench prompt

was copied into the file for compilation and simulation it was edited to form a cohesive block. However, no additional HDL was added for this process. Similarly, there were occasions when responses included comments for the user to add their own code. If these comments would prevent functionality, such as leaving an incomplete array of values, the response would be regenerated, otherwise it was left as-is.

Prompting for functionality: This consistent and conversationally-styled prompt was built as follows: "I am trying to create a Verilog model for a [test name].". The specifications would then be provided, with input and output ports defined and any further specifics needed (such as the expected sequence to be produced by the sequence generator), then the remark "How would I write a design that meets these specifications?". Figure 3 shows the design prompt for the 8-bit shift register which was used as the initial evaluation for each of the LLMs.

Prompting for verification: The testbench prompt (Figure 4) was kept the same for all designs, as requesting a testbench did not need to contain any additional information about the design which was created. This is because the testbench prompt will follow the design produced by the LLM, meaning they can take all existing conversation information into account. It requested that all testbenches were compatible with iverilog for ease of simulation and testing and to help ensure that only Verilog-2001 standards were used.

C. Real-world design constraints

This work aims to investigate the application of conversational generative large language models for real-world hardware design, which has synthesis, budgetary, and tape-out constraints. Therefore, for this project, we targeted the real-world platform Tiny Tapeout 3 [28], which sells small areas (1000 standard cells) of a Skywater 130nm shuttle. This adds constraints to the design: Specifically, a limitation on IO – each design was only allowed 8 bits of input and 8 bits of output. As the goal for the standard challenge benchmarks was to implement several at once, 3 bits of the input were thus reserved for a multiplexer to select which benchmark's output is emitted. This meant that each benchmark could only include 5 bits of input, including a clock and reset.

The Tiny Tapeout toolflow relies on OpenLane [29] meaning that we were restricted to synthesizable Verilog-2001 HDL. The relatively small area, while not a major concern for the challenge benchmarks, did impact the processor components and interface (Section IV).

TABLE I
BENCHMARK DESCRIPTIONS

Benchmark	Description
8-bit Shift Register	Shift register with enable
Sequence Generator	Generates a specific sequence of eight 8-bit values
Sequence Detector	Detects if the correct 8 3-bit inputs were given consecutively
ABRO FSM	One-hot state machine for detecting inputs A and B to emit O
Binary to BCD	Converts a 5-bit binary input into an 8-bit BCD output
LFSR	8-bit Linear Feedback Shift Register
Traffic Light FSM	Cycle between 3 states based on a number of clock cycles
Dice Roller	Simulated rolling either a 4, 6, 8, or 20-sided die

TABLE II
EVALUATED CONVERSATIONAL LLMS

Model	Release Date	Company	Open Access	Open Source
ChatGPT-4 [30]	14 Mar. 2023	OpenAI	No	No
ChatGPT-3.5 [9]	30 Nov. 2022	OpenAI	Yes	No
Bard [10]	21 Mar. 2023	Google	Yes	No
HuggingChat [26]	April 2023	HuggingFace	Yes	Yes

D. Challenge benchmarks

The benchmarks for this challenge were designed to give some insight in to the level of hardware that the different LLMs could write. The targeted functionality is commonly implemented in hardware, and usually taught at the level of an undergraduate digital logic curriculum. The benchmarks are given in table I.

Some benchmarks had their own specific requirements beyond the initial design, to help examine how the LLMs handled different design constraints. The sequence generator and detector were both given their specific patterns to generate or detect respectively, the ABRO was to use one-hot state encoding, and the LFSR had a specific initial state and tap locations. Other benchmarks, like the shift register, were kept minimally descriptive to note if there were any patterns with the output of the models when less constrained.

E. Model evaluation: Metrics

Four different conversational LLMs were evaluated for their proficiency in creating Verilog for hardware design, shown in Table II. As an initial test, each of these models was prompted with the 8-bit shift register benchmark prompt with the intention of proceeding through the conversational flow in Section III-B. The responses to the design prompts for each LLM are shown in Figures 5, 6, 7, and 8.

Each of these tests was treated as the start of the full conversational flow, so while both ChatGPT models were able to meet the specifications and begin moving through the design flow, both Bard and HuggingChat failed to meet the initial criteria of the specifications. The responses to the initial prompt given to Bard and HuggingChat were regenerated five times, following the planned conversational flow, and both failed repeatedly. Bard consistently failed to meet the given design specifications and HuggingChat's Verilog output was not syntactically correct past the module definition. Figures 7 and 8 represent the final attempts of both models.

Given the poor performance of Bard and HuggingChat on the initial Challenge Benchmark prompt, we decided to move forward with the full set of tests only for ChatGPT-4 and ChatGPT-3.5, both of which were able to continue with the conversational flow consistently. For the complete set of benchmarks, we ran these conversations three times, as the LLMs are non-deterministic and capable of responding differently to the same input prompts. This repetition thus provides a basic measure of how consistently they are able to create the different benchmarks and testbenches, and how different runs given the same initial prompt can differ in their implementation.

```
module shift_register (
    input wire clk,
    input wire reset_n,
    input wire data_in,
    input wire shift_enable,
    output reg [7:0] data_out
);

always @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        data_out <= 8'b0; // Active-low reset: Clear the
            register when reset_n is low
    end else if (shift_enable) begin
        data_out <= {data_out[6:0], data_in}; // Shift the
            register and insert the new data bit
    end
end

endmodule
```

Fig. 5. 8-bit shift register attempt from ChatGPT-4.

```
module shift_register(
    input clk,
    input reset_n,
    input data,
    input shift_enable,
    output reg [7:0] q
);

always @(posedge clk, negedge reset_n) begin
    if (!reset_n) begin
        q <= 8'b0;
    end else if (shift_enable) begin
        q <= {q[6:0], data};
    end
end

endmodule
```

Fig. 6. 8-bit shift register attempt from ChatGPT-3.5.

```
1  module shift_register (
2    input clk,
3    input rst,
4    input [7:0] data,
5    input shift_enable,
6    output reg [7:0] data_out
7  );
8
9    always @(posedge clk or posedge rst) begin
10     if (rst) begin
11       data_out <= 8'b0;
12     end else begin
13       if (shift_enable) begin
14         data_out <= {data_out[6:0], data};
15       end
16     end
17   end
18
19 endmodule
```

Fig. 7. 8-bit shift register attempt by Bard. Input on line 4 is too wide.

```
module ShiftRegister(input clk, input rst_n, input data_in,
      input shft_en, output reg [7:0] q);

... //it generates a number of parameters here which are
     not referred to

assign data_out = ~data_in & shft_en & rst_n ? 1'b0 : (q
     [7:1] << 1) | {1'b0, data_in};
always @(*) begin
 # /* x'FF  */ CLK_RQSTDCLK *;
 # /reset_n R0 R0[(int*)&(_CTL--)] <= 32'h8000_0000;
 if (rst_n && !shft_en) begin
  $setuphold (posedge reset_n , negedge active ) ;
  $display("***␣RESET␣***\n");
  `#addr `addr:`addr[(int*) &POS]`pos ?(int) ( ((`addr`:
       addr[_ADDR]+ $_INTRTVALU)/2):1'b1;`addr:(addr<>`addr
       )`pos ,"Full␣scan␣done."));`cout!="Division␣by␣zero!
       " & '@( !_LUT) && (`grandpa')?(sprintf(`snip`,strlen
       (&_TI),($half*`cout>&1)'hash ()): snprintf(!reserve2
       ((const char*)(type(*``cout)strlen`buf++->_GBIT]))))
        oe=tmp2 && tmp3 ? (__flash)((void*)realloc($(int*)(
       idx=(*(u*`ptr`)|(~(address>
  removecomments Tr1ESran diff gred$\elleoformatfloat
       format-----------------------`charTmpLim} Len
       rollover Leng len Len Non Rollovr OptimizationCenter
       }\n resident AsymptoticAspect}\np ResetOpt \All %
  ...```
```

Fig. 8. 8-bit 'shift register' by HuggingChat (truncated, formatted).

Compliant vs. Non-compliant designs: Given that the language models created both the functional code and verification testbenches, when a design 'passed' the testbench it may still have been 'non-compliant' with the original specification. We thus mark each result as 'compliant' or 'non-compliant' overall.

F. Example conversation

Figure 9 provides the examples of the remainder of the conversation with ChatGPT-4 for the shift register T1—for succinctness, we remove the portions of the response that are irrelevant. This conversation flow follows the initial design prompt in Figure 3), the design returned in Figure 5, and the testbench prompt in Figure 4).

Unfortunately, the testbench it generates contains erroneous traces (relevant portion presented in Figure 9a). When simulated, this will print error messages. These are returned to ChatGPT-4 using the message in Figure 9b. This prompts ChatGPT-4 to repair the testbench, giving the code in Figure 9c. The errors are resolved, and the design and testbench now validate against one another, meaning that the conversational design flow criteria is met. In addition, human review shows that the shift register complies with the original specification.

G. Results

All chat logs are provided in the data repository [14]. Table III shows the results of the three test-sets of the scripted benchmarks run with both ChatGPT-4 and -3.5.

ChatGPT-4 performed well. The majority of benchmarks passed, most of which only required tool feedback. ChatGPT-4 most frequently needed human feedback in testbench design.

Several failure modes were consistent, with a common error being the addition of SystemVerilog-specific syntax in the design or testbench. For example, it would often try to use `typedef` to create states for the FSM models, or instantiate arrays of vectors, neither of which are supported in Verilog-2001.

In general, the testbenches produced by ChatGPT-4 were not particularly comprehensive. Still, the majority of the designs that passed their accompanying testbenches were also deemed to be compliant. The two non-compliant 'passes' were Dice Rollers which did not produce a pseudo-random output. The Dice Roller from test set T1 would output a 2 for one roll and then only 1 for all subsequent rolls, regardless of the die selected. Meanwhile, Dice Roller T3 would change values, but only between a small set (dependent on the chosen die) which rapidly repeated. To close the design loop, we synthesized test set T1 from the ChatGPT-4 conversations for Tiny Tapeout 3, adding in a wrapper module that was designed, but not tested, by ChatGPT-4. In all the design took 85 combinational logic units, 4 diodes, 44 flip flops, 39 buffers, and 300 taps to implement.

ChatGPT-3.5: ChatGPT-3.5 performed notably worse than ChatGPT-4, with the majority of the conversations resulting in a failed benchmark, and the majority of those that passed their own testbenches being non-compliant. The modes of failure were less consistent with ChatGPT-3.5 than they were for ChatGPT-4, with a wide variety of issues introduced between each conversation and benchmark. It required corrections to the design and the testbenches

```
// Test case 1: Simple shift
data_in = 1’b1;
shift_enable = 1’b1;
repeat (8) @(posedge clk);
if (data_out !== 8’b10000000) $display(”Error:␣Test␣case␣1␣
     failed.␣Expected:␣10000000,␣Received:␣%b”, data_out);

// Test case 2: Shift with alternating bits
// ... STIMULUS TRUNCATED FOR SPACE
if (data_out !== 8’b10101010) $display(”Error:␣Test␣case␣2␣
     failed.␣Expected:␣10101010,␣Received:␣%b”, data_out);

// Test case 3: Shift without shift_enable
data_in = 1’b0;
shift_enable = 1’b0;
@(posedge clk);
if (data_out !== 8’b10101010) $display(”Error:␣Test␣case␣3␣
     failed.␣Expected:␣10101010,␣Received:␣%b”, data_out);

// Test case 4: Reset during shift operation
data_in = 1’b1;
shift_enable = 1’b1;
@(posedge clk);
reset_n = 0;
@(posedge clk);
reset_n = 1;
if (data_out !== 8’b00000000) $display(”Error:␣Test␣case␣4␣
     failed.␣Expected:␣00000000,␣Received:␣%b”, data_out);

// Display success message and end simulation
$display(”All␣test␣cases␣passed!”);
$finish;
```

(a) Portion of 8-bit shift register testbench with errors

```
When running the simulation it gives the following output.
     Please provide fixed code.

Error: Test case 1 failed. Expected: 10000000, Received:
     01111111
Error: Test case 2 failed. Expected: 10101010, Received:
     01010101
Error: Test case 3 failed. Expected: 10101010, Received:
     01010101
All test cases passed!
```

(b) Tool feedback prompt for 8-bit shift register

```
// Test case 1: Simple shift
data_in = 1’b1;
shift_enable = 1’b1;
repeat (8) @(posedge clk);
if (data_out !== 8'b01111111) $display(”Error:␣Test␣case␣1␣
     failed.␣Expected:␣01111111, Received: %b”, data_out);

// Test case 2: Shift with alternating bits
// ... STIMULUS TRUNCATED FOR SPACE
if (data_out !== 8'b01010101) $display(”Error:␣Test␣case␣2␣
     failed.␣Expected:␣01010101, Received: %b”, data_out);

// Test case 3: Shift without shift_enable
data_in = 1’b0;
shift_enable = 1’b0;
@(posedge clk);
if (data_out !== 8'b01010101) $display(”Error:␣Test␣case␣3␣
     failed.␣Expected:␣01010101, Received: %b”, data_out);

// Test case 4: Reset during shift operation
data_in = 1’b1;
shift_enable = 1’b1;
@(posedge clk);
reset_n = 0;
@(posedge clk);
reset_n = 1;
if (data_out !== 8’b00000000) $display(”Error:␣Test␣case␣4␣
     failed.␣Expected:␣00000000,␣Received:␣%b”, data_out);

// Display success message and end simulation
$display(”All␣test␣cases␣passed!”);
$finish;
```

(c) Corrected portion of testbench code. Replaced values bold / highlighted.

Fig. 9. Remaining portions of the successful shift register T1 conversation with ChatGPT-4. The design is compliant.

TABLE III
BENCHMARK CHALLENGE RESULTS

Benchmark	Test Set	ChatGPT-4			ChatGPT-3.5		
		Outcome	Compliant	# Messages	Outcome	Compliant	# Messages
Shift Register	T1	TF	✓	3	SHF	✓	13
	T2	TF	✓	9	FAIL	-	25
	T3	AHF	✓	15	FAIL	-	11
Sequence Gen.	T1	AHF	✓	14	FAIL	-	25
	T2	TF	✓	4	FAIL	-	7
	T3	AHF	✓	20	FAIL	-	25
Sequence Det.	T1	FAIL	-	24	FAIL	-	21
	T2	SHF	✓	9	SHF	✗	8
	T3	TF	✓	13	SHF	✗	8
ABRO	T1	FAIL	-	16	FAIL	-	25
	T2	AHF	✓	20	MHF	✓	15
	T3	TF	✓	12	NFN	✗	3
LFSR	T1	TF	✓	12	FAIL	-	25
	T2	SHF	✓	7	TF	✓	4
	T3	SHF	✓	9	FAIL	-	11
Binary to BCD	T1	TF	✓	4	SHF	✗	8
	T2	NFN	✓	2	FAIL	-	12
	T3	SHF	✓	9	TF	✗	4
Traffic Light	T1	TF	✓	4	FAIL	-	25
	T2	SHF	✓	12	FAIL	-	13
	T3	TF	✓	5	FAIL	-	18
Dice Roller	T1	SHF	✗	8	MHF	✗	9
	T2	SHF	✓	9	FAIL	-	25
	T3	SHF	✗	18	NFN	✗	3

far more often than ChatGPT-4.

H. *Observations*

Of the four LLMs examined with the challenge benchmarks, only ChatGPT-4 performed adequately, though it still required human feedback for most conversations to be both successful and compliant with the given specifications. When fixing errors, ChatGPT-4 would often require several messages to fix minor errors, as it struggled to understand exactly what specific Verilog lines would cause the error messages from iverilog. The errors it would add also tended to repeat themselves between conversations quite often.

ChatGPT-4 also struggled much more to create functioning testbenches than functioning designs. The majority of benchmarks required little to no modification of the design itself, instead necessitating testbench repair. This is particularly true of FSMs, as the model seemed unable to create a testbench which would properly check the output without significant feedback regarding the state transitions and corresponding expected outputs. ChatGPT-3.5, on the other hand, struggled with both testbenches and functional designs.

IV. CO-DESIGN SPACE EXPLORATION: FREE CHAT

A. *Overview*

Real-world hardware design will have broader and more complex requirements than those we investigated in the previous Section III. This is a challenge when considering the previously used methodology, which scripted and constrained the way that a human could interact with the LLMs. However, given the relative success of the various levels of human feedback, we seek to investigate if unstructured conversations might allow for greater levels of performance and mutual creativity. Investigating this would in general be done with a large-scale user-study, with hardware engineers paired with the tool during development. Such studies have been done in the software domain for LLMs, e.g. this example from Google which paired their proprietary LLM with >10,000 software developers [31] and found measurable, positive impacts on developer productivity (reduced their coding iteration duration by 6 % and reduced number of context switches by 7 %). We aim to motivate such a study for the hardware domain by performing a proof of concept experiment, where we pair an LLM (the best-performing model, OpenAI’s ChatGPT-4) with an experienced hardware design engineer (one of the paper authors), and

```
Let us make a brand new microprocessor design together. We'
    re severely constrained on space and I/O. We have to
    fit in 1000 standard cells of an ASIC, so I think we
    will need to restrict ourselves to an accumulator
    based 8-bit architecture with no multi-byte
    instructions. Given this, how do you think we should
    begin?
```

Fig. 10. 8-bit accumulator-based processor: Starting co-design prompt

qualitatively examine the outcome when tasked with making a more complex design, as outlined next.

B. Design Task: An 8-bit accumulator-based microprocessor

Constraints: We again adhere to the requirements established in Section III-C. We wish for ChatGPT-4 to write all the processor's Verilog (excluding the top-level Tiny Tapeout wrapper). To ensure we can load and unload data from the processor, we require all registers to be connected in a 'scan chain' of shift registers.

Overall goal: Co-design of an 8-bit accumulator-based architecture. The initial prompt to ChatGPT-4 is provided in Figure 10. Given the space restriction, we aimed for a von Neumann type design with 32 bytes of memory (combined data and instruction) .

Task partitioning: Given the strengths and weaknesses of the LLMs explored, and to avoid producing 'non-compliant' designs (see Section III-E), for this design task the experienced human engineer was responsible (a) for shepherding ChatGPT-4, and (b) for verifying its output (e.g. syntax checks, authoring verification code / testbenches). Meanwhile, ChatGPT-4 was solely responsible for the Verilog code for the processor. It also produced the majority of the processor's specification.

C. Method: Conversation flow

General process: The microprocessor design process began by defining the Instruction Set Architecture (ISA), then implementing components that the ISA would require, before combining those components in a datapath with a control unit to manage them. Simulation and testing were used to find bugs which were then repaired.

Conversation threading: Given that ChatGPT-4, like other LLMs, has a fixed-size context window (see Section II-A), we assumed that the best way to prompt the model is by breaking up the larger design into subtasks which each had its own 'conversation thread' in the interface. This keeps the overall length below 16,000 characters. A proprietary back-end method performs some kind of text reduction when the length exceeds this, but details on its implementation are scarce. Since ChatGPT-4 does not share information between threads, the human engineer would copy the pertinent information from the previous thread into the new first message, growing a 'base specification' that slowly comes to define the processor. The base specification eventually included the ISA, a list of registers (Accumulator 'ACC', Program Counter 'PC', Instruction Register 'IR'), the definitions for the memory bank, ALU, and control unit, and a high-level overview of what the processor should do in each cycle. Most of the information in this specification was produced by ChatGPT-4 and copy/pasted and lightly edited by the human.

Topics: One topic per thread worked well for the early design stages of the processor (with one exception, where the ALU was designed in the same thread as the multi-cycle processor clock cycle timing plan). However, once the processor got to the simulation stage and we ran programs on it, we found mistakes and bugs in the specification and implementation. Rather than starting new conversation threads and rebuilding the previous context, the design engineer instead chose to continue previous conversation threads where appropriate. We illustrate this in our flow map in Table IV, where the 'Cont. T. ID' column indicates if they 'Continued' a previous thread (and if so, which thread).

TABLE IV

CONVERSATION FLOW MAP: THE PROCESSOR WAS BUILT THROUGH A LINEAR FLOW OF 125 USER MESSAGES ACROSS 18 TOPICS IN 11 'CONVERSATION THREADS'.

Cont. T. ID	T. ID	Topic	# User Msgs	# Restart	# User Lines	# User Chars	# LLM Lines	# LLM Chars
-	00	Specification	22	10	45	5025	498	44818
-	01	Register specification	6	2	59	4927	91	9961
-	02	Shift registers and memory	5	5	65	5444	269	9468
-	03	Multi-cycle planning and ALU	7	2	103	7284	243	10148
-	04	Control signal planning	13	21	216	9205	414	20364
-	05	Control Unit state logic	12	11	216	9898	742	21663
-	06	ISA to ALU opcode	4	0	72	4576	149	5624
-	07	Control unit output logic	11	6	266	8632	518	19180
-	08	Datapath components	12	0	144	5385	516	15646
-	09	Python assembler	3	4	127	4231	218	6270
00	10	Spec. branch update	1	1	14	1275	15	1635
07	11	Control Unit branch update	2	2	98	3743	101	3969
08	12	Datapath branch update	2	0	25	888	20	726
11	13	Control Unit bug fixing	6	1	190	5413	241	8001
-	14	Memory mapped components	7	0	79	3079	516	16237
-	15	Shift Register bug fix	2	0	38	985	85	2593
12	16	Datapath bug fixing & updates	6	0	116	2979	128	4613
14	17	Memory mapped constants	4	0	21	849	101	4655
03	18	ALU optimization	1	0	2	98	32	1368
TOTALS			125	65	1896	83916	4897	206939

```
This looks excellent. According to this list, please
    produce the module definition for a control unit in
    Verilog which could operate the processor datapath.
    Please comment the purpose of each I/O. If a signal is
     for controlling a multiplexer, please also comment
    what each possible value should correspond to in the
    datapath.
```

Fig. 11. The most difficult prompt (10 restarts), which was provided in Topic 04 after ChatGPT-4 produced a list of datapath control signals and definitions.

Restarts: Sometimes ChatGPT-4 outputs suboptimal responses. If so, the engineer has two options: (1) continue the conversation and nudge it to fix the response, or (2) use the interface to force ChatGPT-4 to 'restart' the response, i.e. regenerating the result by pretending the previous answer never occured. Choosing between these has trade-offs and requires professional judgement: continuing the conversation allows for the user to specify which parts of the previous response are good or bad, but regeneration will keep the overall conversation shorter and more succinct (valuable considering the finite context window size). Still, as can be seen from the '# Restart' column in Table IV, the number of restarts tended to decrease as the engineer grew more experienced with using ChatGPT-4, with Topics 00-07 having 57 restarts compared to Topics 08-18 having just 8. The highest individual number of restarts on a single message was 10, in Topic 04 (Control signal planning) which has the message in Figure 11. This was a difficult prompt because it asks for a specific kind of output with a significant amount of detail, but eventually yielded a satisfactory answer as listed in Figure 12.

D. Results: ISA

All chat logs are provided in the data repository [14]. The ISA co-generated with ChatGPT-4 in Conversation 00 (and updated in 10) is presented in Table V. It is a relatively straightforward accumulator-based design with some notable features: (1) given the size constraints, the memory-access 'Instructions with Variable-Data Operands' use just five bits to specify the memory address, meaning the processor would be limited to an absolute maximum of 32 bytes of memory. (2) There is just one instruction with an immediate-data encoding. (3) The instructions use the full 256 possible byte

```verilog
module control_unit (
    // Clock input
    input wire clk,
    // Reset input
    input wire rst,
    // Input from the Instruction Register (IR)
    input wire [7:0] instruction,
    // Enables writing to the PC
    output wire PC_write_enable,
    // Selects the input for the PC multiplexer
    // 00: PC + 1 (FETCH cycle, BEQ_FWD, BNE_FWD)
    // 01: ACC (JMP, JSR)
    // 10: PC - 3 (BEQ_BWD, BNE_BWD)
    output wire [1:0] PC_mux_select,
    // Enables writing to the ACC
    output wire ACC_write_enable,
    // Selects the input for the ACC multiplexer
    // 00: ALU output
    // 01: Memory contents (LDA, LDAR)
    // 10: PC (JSR)
    output wire [1:0] ACC_mux_select,
    // Enables loading new instruction into IR from memory
    output wire IR_load_enable,
    // Control signal specifying the ALU operation
    output wire [3:0] ALU_opcode,
    // Selects input B for the ALU multiplexer
    // 0: Memory contents (ADD, SUB, AND, OR, XOR)
    // 1: Immediate (ADDI)
    output wire ALU_inputB_mux_select,
    // Enables writing to memory (STA)
    output wire Memory_write_enable,
    // Selects input for memory address multiplexer
    // 00: IR[4:0] (LDA, STA, ADD, SUB, AND, OR, XOR)
    // 01: ACC (LDAR)
    // 10: PC (Instruction fetching)
    output wire [1:0] Memory_address_mux_select
);
```

Fig. 12. Code produced by ChatGPT-4 for difficult prompt (11th attempt). It is still missing some I/O, corrected by later messages.

encodings. (4) The JSR instruction makes it possible to implement subroutine calls, albeit a little awkwardly (there's no stack pointer). (5) The branch instructions are restrictive but useful. Skipping two instructions backwards allows for efficient polling (e.g. load an input, mask it for relevant bit, then check if 0 or not). Skipping 3 instructions forwards allows to skip over the instructions needed for a JMP or JSR. These were co-designed over a number of iterations, including a later modification (Conversations 10-12, the 'branch update') which increased the jump forwards from 2 instructions to 3 after during simulation we realized that we could not easily encode JMP/JSR in just 2 instructions. (5) The LDAR instruction allows for pointer-like dereferences for memory loads. This enabled us to efficiently use a table of constants in our memory map (added in Conversation 17) to convert binary values into LED patterns for a 7-segment display.

E. Results: Processor implementation

The processor datapath was assembled in Conversation 08 and is illustrated in Figure 13. The von Neumann design (shared memory for data and instructions) necessitated a 2-state multi-cycle control unit ('FETCH' and 'EXECUTE'). A third 'HALT' state is entered after reaching a HLT instruction (reset to exit). 'HALT' also sets a `processor_halted` output flag. Notably, because the 'FETCH' state also increments the PC register, the branch instructions in the ISA require '-3' and '+2' modifiers. The Memory Bank is parameterized globally, allowing the human engineer to change the memory size from inside the Tiny Tapeout wrapper (the only file they authored, which was used to perform non-processor-related wiring). The processor was eventually synthesized with 17 bytes of register memory, with the 17th byte used for I/O (7-segment LED outputs, one button input). A look-up constant memory table of 10 bytes used for segment patterns was concatenated. After synthesis, the processor results in the 'GDS' in Section IV-E.

TABLE V
ISA CO-CREATED WITH CHATGPT-4 (USES ALL 256 ENCODINGS).

Instruction	Description	Opcode
Instructions with Immediate Operands		
ADDI	Add immediate to Accumulator	1110XXXX
Instructions with Variable-Data Operands		
LDA	Load Accumulator with memory contents	000MMMMM
STA	Store Accumulator to memory	001MMMMM
ADD	Add memory contents to Accumulator	010MMMMM
SUB	Subtract memory contents from Accumulator	011MMMMM
AND	AND memory contents with Accumulator	100MMMMM
OR	OR memory contents with Accumulator	101MMMMM
XOR	XOR memory contents with Accumulator	110MMMMM
Control and Branching Instructions		
JMP	Jump to memory address	11110000
JSR	Jump to Subroutine (save address to ACC)	11110001
BEQ_FWD	Branch if ACC==0, forward (PC = PC + 3)	11110010
BEQ_BWD	Branch if ACC==0, backward (PC = PC - 2)	11110011
BNE_FWD	Branch if ACC!=0, forward (PC = PC + 3)	11110100
BNE_BWD	Branch if ACC!=0, backward (PC = PC - 2)	11110101
HLT	Halt the processor until reset	11111111
Data Manipulation Instructions		
SHL	Shift Accumulator left	11110110
SHR	Shift Accumulator right	11110111
SHL4	Shift Accumulator left by 4 bits	11111000
ROL	Rotate Accumulator left	11111001
ROR	Rotate Accumulator right	11111010
LDAR	Load Accumulator via indirect mem. access M[ACC]	11111011
DEC	Decrement Accumulator	11111100
CLR	Clear (Zero) Accumulator	11111101
INV	Invert (NOT) Accumulator	11111110

F. Observations

In general, ChatGPT-4 produced relatively high-quality code, as can be seen by the short verification turnaround. Once the Python assembler was written (Conversation 09), the bug-fixing Conversations (10-13, 15-16) used just 19 of the total 125 messages. Given the 25 messages per 3 hours rate limit by ChatGPT-4, the total time budget for this design was 22.8 hours of ChatGPT-4 (including the restarts). The actual generation averaged around 30 seconds per message: with no rate limit the whole design *could* have been completed in $<$100 minutes, subject to the human engineer. Although ChatGPT-4 produced the Python assembler with relative ease, it struggled to author programs written for our design, and no non-trivial test programs were written by ChatGPT. Overall, we evaluated all 24 instructions across a series of comprehensive human-authored assembly programs both in simulation and FPGA emulation.

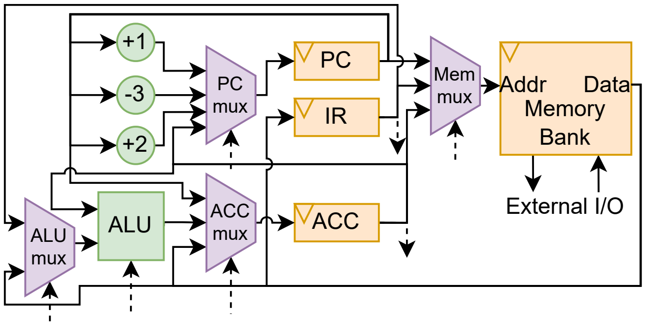
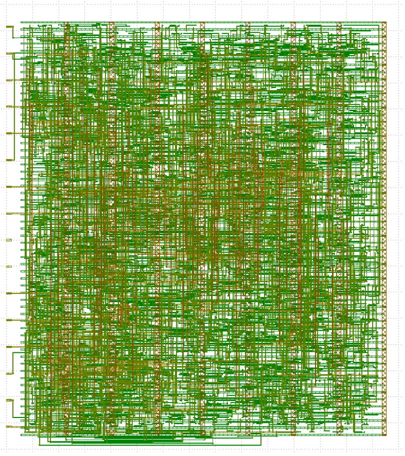


Fig. 13. Accumulator-based datapath designed by GPT-4 (illustration by human). Control signals indicated with dotted lines.



Component	Count
Comb. Logic	999
Diode	4
Flip Flops	168
Buffer	126
Tap	300

Above: (a) Components.

Left: (b) Final processor GDS render by 'klayout', I/O ports on left side, grid lines = 0.001 um.

Fig. 14. Processor synthesis information.

V. EVALUATION

A. Discussion

Steps for practical adoption: Ideally with the rise of conversational LLMs it would be possible to go from idea to functional design with minimal effort. Although much emphasis has been placed on their single-shot performance (i.e., making a design in a single step) we found for hardware applications that they function better as a *co-designer*. Where they work in lock-step with an experienced engineer, they may serve as an effort 'force multiplier', providing 'first-pass' designs which may then be tweaked and quickly iterated over.

A notable observation from the scripted benchmark tests is that the overall outcome depends heavily on the early interactions: the response to the initial prompt and first few instances of feedback. In many cases there were simple errors that took many iterations of feedback to resolve because the LLMs failed to understand the correlation between error and fix. As a result, we recommend evaluating the responses to the early prompts, and if they are unsatisfactory, consider 'restarting' the conversation from an earlier point.

State of the art performance: HuggingFace's HuggingChat was the obvious worst-performer, struggling at times to even write coherent Verilog. Google's Bard was better at this, but was still unable to follow instructions with enough detail that it could be evaluated. OpenAI's ChatGPT-3.5 and ChatGPT-4 could both follow specifications and write Verilog, but only ChatGPT-4 could do it reliably. Still, it managed to make a compliant, functional output without human input in fewer than half of the conversations. Once it had this input, though, ChatGPT-4 was able to produce compliant code for 20/24 benchmarks; and this performance was echoed with the free chat process co-design. Here, the model was able to both help create the specification and implement it into Verilog. The major limitation with the state of the art performance is in authorship of testbench and verification code. We believe that this reflects the (non-)availability of suitable open-source training data.

B. Threats to Validity

Reproducibility: As the conversational LLMs tested are non-deterministic and generative, the outputs are not consistently reproducible. This is shown in the variable successes of the benchmark conversations where some conversations for a single benchmark succeeded with simple human feedback and few messages and others failed outright. Both versions of ChatGPT are closed-source and run remotely, so we are unable to examine the parameters of the model and analyze the method for generating outputs. The conversational nature of these tests hamper the reproducibility, as each user response in the conversation depends on the previous model response, so slight variations can create substantial changes in the final design. Regardless, we do provide the full conversation logs for result reconstruction [14].

Statistical Validity: As the goal of this work was to design hardware conversationally, we did not automate any part of this process, and each conversation needed to be done manually. This limited the scale of the experiments that could be performed, which were also hampered by rate limits and model availability (both OpenAI's ChatGPT-4 and Google's Bard still have limited access at time of writing). As a result, the three test cases may not provide enough data to draw formal statistical conclusions.

VI. CONCLUSIONS

Challenges: While it is clear that using a conversational LLM to assist in designing and implementing a hardware device can be beneficial overall, the technology is not yet able to consistently design hardware with only feedback from verification tools. The current state-of-the-art models do not perform well enough at understanding and fixing the errors presented by these tools to create complete designs and testbenches with only an initial human interaction.

Opportunities: Still, when the human feedback is provided to the more capable ChatGPT-4 model, or it is used to co-design, the language model seems to be a 'force multiplier', allowing for rapid design space exploration and iteration. In general, ChatGPT-4 could produce functionally correct code, which could free up designer time when implementing common modules. Potential future work could involve a larger user study to investigate this potential, as well as the development of conversational LLMs specific to hardware design to improve upon the results.

REFERENCES

[1] A. B. Kahng, "Machine Learning Applications in Physical Design: Recent Results and Directions," in *Proceedings of the 2018 International Symposium on Physical Design*, ser. ISPD '18. New York, NY, USA: Association for Computing Machinery, Mar. 2018, pp. 68–73. [Online]. Available: https://doi.org/10.1145/3177540.3177554

[2] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 1–6. [Online]. Available: https://doi.org/10.1145/3195970.3196026

[3] G. Huang, J. Hu, Y. He, J. Liu, M. Ma, Z. Shen, J. Wu, Y. Xu, H. Zhang, K. Zhong, X. Ning, Y. Ma, H. Yang, B. Yu, H. Yang, and Y. Wang, "Machine Learning for Electronic Design Automation: A Survey," *ACM Transactions on Design Automation of Electronic Systems*, vol. 26, no. 5, pp. 40:1–40:46, Jun. 2021. [Online]. Available: https://doi.org/10.1145/3451179

[4] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "HardFails: Insights into Software-Exploitable Hardware Bugs," 2019, pp. 213–230. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky

[5] P. Coussy and A. Morawiec, *High-level synthesis*. Springer, 2010, vol. 1.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder,

B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," Jul. 2021. [Online]. Available: http://arxiv.org/abs/2107.03374

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

[8] 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," Dec. 2022. [Online]. Available: http://arxiv.org/abs/2212.11140

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

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

[11] T. H. Kung, M. Cheatham, A. Medenilla, C. Sillos, L. D. Leon, C. Elepaño, M. Madriaga, R. Aggabao, G. Diaz-Candido, J. Maningo, and V. Tseng, "Performance of ChatGPT on USMLE: Potential for AI-assisted medical education using large language models," *PLOS Digital Health*, vol. 2, no. 2, p. e0000198, Feb. 2023, publisher: Public Library of Science. [Online]. Available: https://journals.plos.org/digitalhealth/article?id=10.1371/journal.pdig.0000198

[12] A. Ahmad, M. Waseem, P. Liang, M. Fehmideh, M. S. Aktar, and T. Mikkonen, "Towards Human-Bot Collaborative Software Architecting with ChatGPT," Feb. 2023. [Online]. Available: http://arxiv.org/abs/2302.14600

[13] M. R. King and chatGPT, "A Conversation on Artificial Intelligence, Chatbots, and Plagiarism in Higher Education," *Cellular and Molecular Bioengineering*, vol. 16, no. 1, pp. 1–2, Feb. 2023. [Online]. Available: https://doi.org/10.1007/s12195-022-00754-8

[14] Anonymized for review, "Data Repository for Chip-Chat: Challenges and Opportunities in Conversational Hardware Design," May 2023. [Online]. Available: https://zenodo.org/record/7953724

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "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

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[17] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "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

[18] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "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

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

[20] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, pp. 754–768, iSSN: 2375-1207.

[21] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models." IEEE Computer Society, Oct. 2022, pp. 1–18. [Online]. Available: https://www.computer.org/csdl/proceedings-article/sp/2023/933600a001/1He7XI0Qlzi

[22] efabless, "AI Generated Design Contest," May 2023. [Online]. Available: https://efabless.com/ai-generated-design-contest

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

[24] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe, "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

[25] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi, "Self-Instruct: Aligning Language Model with Self Generated Instructions," Dec. 2022. [Online]. Available: http://arxiv.org/abs/2212.10560

[26] HuggingFace, "HuggingChat," May 2023. [Online]. Available: https://huggingface.co/chat

[27] "Icarus Verilog." [Online]. Available: http://iverilog.icarus.com/home

[28] "Tiny Tapeout," May 2023. [Online]. Available: https://tinytapeout.com/

[29] "OpenLane," May 2023, original-date: 2020-07-20T19:35:02Z. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenLane

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

[31] M. Tabachnyk and S. Nikolov, "ML-Enhanced Code Completion Improves Developer Productivity," Jul. 2022. [Online]. Available: http://ai.googleblog.com/2022/07/ml-enhanced-code-completion-improves.html