Rome was Not Built in a Single Step: Hierarchical Prompting for LLM-based Chip Design

Andre Nakkab
New York University
andre.nakkab@nyu.edu

Sai Qian Zhang
New York University
sai.zhang@nyu.edu

Ramesh Karri
New York University
rkarri@nyu.edu

Siddharth Garg
New York University
siddharth.garg@nyu.edu

Abstract

Large Language Models (LLMs) are effective in computer hardware synthesis via hardware description language (HDL) generation. However, LLM-assisted approaches for HDL generation struggle when handling complex tasks. We introduce a suite of hierarchical prompting techniques which facilitate efficient stepwise design methods, and develop a generalizable automation pipeline for the process. To evaluate these techniques, we present a benchmark set of hardware designs which have solutions with or without architectural hierarchy. Using these benchmarks, we compare various open-source and proprietary LLMs, including our own fine-tuned Code Llama-Verilog model. Our hierarchical methods automatically produce successful designs for complex hardware modules that standard flat prompting methods cannot achieve, allowing smaller open-source LLMs to compete with large proprietary models. Hierarchical prompting reduces HDL generation time and yields savings on LLM costs. Our experiments detail which LLMs are capable of which applications, and how to apply hierarchical methods in various modes. We explore case studies of generating complex cores using automatic scripted hierarchical prompts, including the first-ever LLM-designed processor with no human feedback.

Keywords

LLM, Hardware design, Hierarchy, Automation

ACM Reference Format:
Andre Nakkab, Sai Qian Zhang, Ramesh Karri, and Siddharth Garg. 2024. Rome was Not Built in a Single Step: Hierarchical Prompting for LLM-based Chip Design. In *Proceedings of MLCAD (MLCAD '24).* ACM, New York, NY, USA, 11 pages.

1 Introduction

Hierarchical design is a key concept for creating complex computer hardware in an organized fashion. The goal of hierarchy is to break complex modules into manageable submodules, the way one might define a function in high-level code. However, recent efforts into LLM-based generation of hardware description language (HDL) code [11, 18, 19] generate modules non-hierarchically, i.e., as single blocks of straight-line code. Although these methods succeed on simple designs like bit-parallel adders and shift registers [17], they struggle on complex designs in recent benchmarks, such as finite-state machines (FSM), large-scale many-to-1 multiplexers, and larger arithmetic blocks [10]. Since straight-line code blocks for complex designs are longer than the hierarchical alternatives, they may hallucinate [9]; the LLM generates incorrect or unrelated text. Additionally, long outputs increase response latency and sometimes fail due to output length limits.

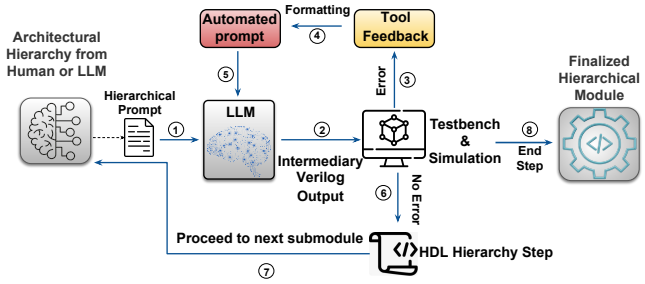

Figure 1: Automatic Hierarchical Prompting Pipeline.

In this paper, we develop and evaluate *hierarchical prompting* techniques to facilitate automated generation of *modular* HDL code. We explore hierarchical Verilog generation in two major modalities, each occurring in the real-world. In the *human-driven* mode, the prompt contains a human-proposed hierarchy that the LLM must extract and implement, as well as iterative compiler feedback from unit tests for each submodule. In the more challenging *purely generative* mode, the LLM gets only a basic (non-hierarchical) prompt and therefore must make its own design decisions to implement the target module. We implement an 8-stage pipeline to automate these techniques in a generalizable fashion, which we refer to as Recurrent Optimization via Machine Editing (ROME). This allows an LLM to closely emulate human HDL development practices.

Existing benchmark suites like VerilogEval [10] and RTLLM [12] do not address hierarchy. We introduce *a new benchmark suite of complex modules with explicit hierarchical solutions, including associated prompts and testbenches for both the top-level modules, and unit tests for submodules.* The target modules in our benchmark pose unique challenges to LLMs. These include a 32-bit data-dependent

MLCAD '24, Sept. 9-11, 2024, Snowbird, UT

ACM ISBN 979-8-4007-0699-8/24/09
DOI:10.1145/3670474.3685964

left rotation (also known as a barrel shifter) that requires rarely-seen syntax which is difficult to generate from scratch; an Advanced Encryption Standard (AES) block cipher which has multiple complex submodules that are difficult to organize within a single large Verilog script; and, a universal asynchronous receiver and transmitter (UART) interface which requires a multi-part FSM to function. These designs could not be implemented at all by any of the open-source or commercial LLMs we tried in non-hierarchical mode, motivating the need for advanced hierarchical prompting methods. The full list of benchmarks are in Table 5 in the Appendix.

Evaluations on 8 state-of-art LLMs demonstrate that hierarchical prompt structuring dramatically improves LLM performance on hardware design tasks, enabling successful generation of modules that would otherwise be impossible. Finally, we report case studies on the generation of complex hardware modules outside of the context of the hierarchical prompting benchmarks. We target a 16-bit MIPS processor and a 32-bit RISC-V processor, and present the first-ever purely LLM-designed processor with no human feedback.

2 Background and Related Work

2.1 An Introduction of LLM Operation for Text Generation

Transformer-based deep neural networks (DNN) have enabled advances across a wide range of domains, excelling in language-related tasks. LLMs operate by processing text inputs structured as *tokens*, When presented with a sequence of input tokens, LLMs output a probability distribution spanning the complete *vocabulary* to predict next token in the sequence. This process repeats until a full sequence of tokens, referred to as a *completion*, is produced. Consider an LLM $P_\phi(\vec{y}|\vec{x})$, where ϕ denotes the set of model parameters, $\vec{x}$ and $\vec{y}$ represent the vector of input and output tokens. The LLM will generate the probability distribution for the next token y_n, and output $\vec{y} = \{y_n | 1 \leq n \leq N\}$ is produced autoregressively:

$$y_n = \arg\max_{v \in V} P_\phi(v|\vec{x}, y_{<n}) \tag{1}$$

$1 \leq n \leq N$, N is the length of the output, and V is the set of vocabulary. Equation 1 continues iterating until a designated end-of-sequence token is encountered.

2.2 Related Work

In recent years, LLMs have demonstrated their proficiency in code generation for software programming languages like C and Python [4, 6, 7, 13, 14, 16, 21]. This is possible because LLMs are trained using extensive datasets of code that encompass either one programming language or a combination of multiple languages. Training datasets used can be substantial in size, reaching hundreds of gigabytes of text. Inputs supplied to these LLMs come in various formats, including instructions, comments, code excerpts, or some combination thereof. LLMs can further be tasked to generate hierarchical high-level models [8]. These approaches use Chain-of-Thought prompting to improve LLM reasoning by granularizing problems into sub-problems. This technique improves performance on mathematical word problems [22]. Thus, LLMs are capable of understanding the functional intent of the design task at hand when the task is hierarchically structured.

It is possible to use an LLM to conversationally generate synthesizable HDL at the processor-scale using feedback from a human hardware designer [2]. These methods are relatively expedient, but struggle with consistency and are difficult to evaluate due to the subjective and non-reproducible nature of human feedback. Human intervention also precludes automation. It has alternatively been shown once in the past that one can automate the generation of entire CPUs using traditional deep learning methods [5]. However, these methods take on the order of multiple hours to successfully train and produce a given RTL design for tape-out. Finally, some success has been found by fine-tuning open-source LLMs specifically to produce HDL [11, 18]. Benchmarks have recently been developed to evaluate these LLMs on their Verilog generation performance [10, 12]. Based on these benchmarks, even bespoke fine-tuned models struggle to compete with powerful proprietary LLMs like GPT-3.5 and GPT-4.

3 Hierarchical Prompting

Standard flat prompting involves straightforwardly asking the LLM to generate your desired module. This is effective for small, simple modules, but will often lead to messy, incorrect outputs when applied to more complex hardware structures. The goal of introducing architectural hierarchy via prompting is to allow an LLM to mimic the design process employed by a human engineer. Rather than writing every step of a given hardware element sequentially, we break it down into functional components and pick out reusable blocks that are simpler to produce, and slot them into the greater design.

3.1 Sources of Hierarchy

Hierarchical prompting can take a few forms depending on the resources available to the user. As a baseline, we can utilize a human-defined hierarchy as an input to the pipeline, which results in solely LLM-generated Verilog and no human involvement beyond the planning stage. This is effective when there are specific design constraints for the final module that the LLM might not recognize on its own. We refer to this method as *human-driven hierarchical prompting* (HDHP).

At its most automated, hierarchical prompting can be used to generate our final module purely from the outputs of the LLM. We can describe some desired module and ask the LLM to give us a breakdown of the necessary blocks. From there, we can ask the LLM to generate the next block in the sequence. We refer to this method as *purely generative hierarchical prompting* (PGHP). This is effective for the more common hierarchical modules, as information about them likely appears in standard LLM training datasets, but is a very difficult task when applied to rarely implemented modules.

Our benchmark presumes HDHP-based design methods by default, as knowledge of the hierarchy allows us to create effective unit tests for each submodule in order to fully implement our feedback loop. Conversely, when utilizing PGHP the submodules which the LLM selects are consistent across runs. We can work around this by including a thorough, human-written testbench for the top module whose inputs & outputs we know, and then optionally allowing the LLM to generate its own unit tests as we go.

3.2 ROME Hierarchical Generation Pipeline

As seen in Figure 1, we formulate an automatable design pipeline which implements hierarchical HDL generation to create a complete hierarchical hardware module from end to end. Our proposed method broadly works in three phases:

Hierarchy Extraction. In Step 1, we extract a list of submodules necessary to implement the design from a natural language description provided by the user. Extraction of submodules could be from a list provided by the user in HDHP mode, or extracted from the LLM in the more challenging PGHP mode.

Submodule Implementation. Next, we iterate through the extracted list of submodules, and in each iteration, ask the LLM to produce HDL for one submodule (*Step 2* in Figure 1). If unit tests for submodules are provided, the generated HDL is simulated via an HDL simulator, in our case Icarus Verilog (iVerilog) as it is open-source and easy to automate. Errors from the simulation output are then extracted and fed back to the LLM with an automated request to fix the design, forming a feedback loop that iteratively corrects errors in a given submodule. (*Steps 3, 4, 5* in Figure 1). Once a submodule passes tests, the generated code is logged, and we proceed to generate the next submodule (Steps 6,7). If no unit tests are provided as in the case of PGHP mode, the first generated submodule instance is picked.

Top-Level Module Integration. Finally, we request the LLM to integrate all generated submodules into a top-level module, which in turn has its own testbench. It is then put through the same tool-based feedback loop as the submodules, before finally being output as a completed hierarchical design. This is even possible in PGHP mode, as we know the expected behavior of the top-level module. We consider the run a success if the top module passes its testbenches.

3.3 Prompting Structure and Techniques

Within the pipeline, we employ several methods to ensure that our prompts are informative at each step. We begin with a system prompt which precedes all prompting with every model regardless of what module is being generated. This seeks to reduce output randomness by setting constraints for the LLMs to abide by. Our system prompt was structured as:

> Our goal is to provide complete Verilog modules based on user-provided specifications. Only the specified module is necessary, no testbenches or supplementary modules are needed. Examples of compiler and simulation errors for a given module may be provided, in which case we will proceed to correct the module. All modules will be provided in their complete and correct form.

We then use a benchmark-specific global prompt describing the overall design objective, akin to non-hierarchical approaches and, in the HDHP mode, we iteratively append a submodule prompt, one for each submodule in the list provided by the designer. Therefore, the first prompt provided to the LLM for a 64-to-1 multiplexer designer looks as below, where the top-level prompt is in blue and the first submodule prompt is in green. Note that for each submodule, we only provide its name and the submodule interface to the LLM.

```
include prev_submods.v   #contains previously generated Verilog
      submodules
top_module = "64-to-1 multiplexer"
prev_modules = ["2-to-1 multiplexer","4-to-1 multiplexer"]
next_module = "8-to-1 multiplexer"
next_io = "mux8_1(input [2:0] sel, input [7:0] in, output reg out
      )"
prompt = "We will be designing a <top_module> in Verilog using
      hierarchical submodules. We have generated the following
      submodules: <prev_modules> implemented as:
<prev_submods.v>
Please use the previous submodules to hierarchically generate a <
      next_module> defined as:
module <next_io>;
//Insert code here
endmodule"

next_output = Feedback_Loop(prompt)
append next_output to prev_submods.v
Move to next step in hierarchy
```

(a) Pseudocode for automatic creation of a hierarchical prompt from a template which references previously generated hierarchy steps in the form of Verilog modules, as well as a call to the generative loop which creates the output module.

```
module mux2_1(input in1, input in2, input select, output out);
    assign out = select ? in2 : in1;
endmodule
module mux4_1(input [1:0] sel, input [3:0] in, output out);
    wire out1, out2;
    // First level of multiplexer
    mux2_1 m1(.in1(in[0]), .in2(in[1]), .select(sel[0]), .out(
        out1));
    mux2_1 m2(.in1(in[2]), .in2(in[3]), .select(sel[0]), .out(
        out2));
    // Second level of multiplexer
    mux2_1 m3(.in1(out1), .in2(out2), .select(sel[1]), .out(out))
        ;
endmodule
```

(b) The LLM-generated hierarchical Verilog module, *prev_submods.v*, which the above pseudocode includes as part of the prompt. Note that it contains all prior hierarchical modules, i.e., *mux2_1* and *mux4_1*.

Figure 2: Structure of a hierarchical step, which uses automated prompting and existing hierarchy to generate a new module.

> We will be designing a 64-to-1 multiplexer in Verilog using hierarchical submodules. We begin by generating a 2-to-1 multiplexer with the following structure: module mux2_1(in1, in2, select, out)

An example of automatic hierarchical prompt creation can be seen in Figure 2, with Figure 2a showing the pseudocode instantiating the prompt template and Figure 2b representing the file containing the modules generated so far.

A subtlety in prompting is the distinction between conversational and non-conversational (or text completion) LLMs. In conversational LLMs, prior prompts and responses are automatically added to the LLM's context. For conversational LLMs, our hierarchical prompting approach starts with the global prompt that describes the top-level module and the first submodule. In subsequent iterations, we can simply request the next submodule since previously generated submodules are implicitly remembered in the LLM's context.

For text completion LLMs, however, we have to strategically insert responses from prior steps. We begin with the same global prompt as above that describes the top-level design and asks for the

first submodule. For models with large context windows, we could simply include the code of all the previous submodules within the prompt as we progress. For many open-source LLMs with shorter context windows, however, we must implement what we call the *relay prompt* technique as we progress deeper into the hierarchy, as shown in Figure 3.

Specifically, for a given step, we include the full code of the submodule generated immediately prior to the current step, but only provide the module instantiation line for earlier steps. This becomes a list of all prior submodules that the LLM has access to, always including the complete elaboration of the most recently generated submodule. This allows "leap-frogging" from less complex modules to more complex ones without running into length limits. Along with this prior context, we ask for the next submodule in the design hierarchy. For example, the final text-completion prompt for the decoder hierarchy can be seen in Figure 3b. The full code for the 3-to-8 decoder is included, in addition to the module instantiation for the 2-to-4 decoder. This is sufficient information to improve performance. Though we give the example of the decoder for simplicity, relay prompting is an efficiency tool which becomes much more important for large hierarchies, like the 128-bit AES cipher which consists of hundreds of lines of code. Most open-source LLMs are text-completion models, and this prompting method allows them to leverage hierarchy the same way a conversational LLM like GPT-4 would. All of our benchmarks use relay prompts by default for standardization and efficiency, though it is possible to include full prior submodule information at each step for smaller-scale hierarchies.

```
We will be designing a 5-to-32 decoder in Verilog using
     hierarchical submodules. We begin by generating a 2-to-4
     decoder:
module decoder2to4(<human-sourced I/O>)
<LLM-generated module>
endmodule
We can then use that module hierarchically to generate a 3-to-8
      decoder:
module decoder3to8(
```

(a) Initial prompt for 2-to-4 decoder and follow-up prompt for 3-to-8 decoder.

```
The following Verilog implements a 5-to-35 decoder utilizing
     hierarchical submodules. We have the following module(s)
     already available for use:
module decoder2to4(<human-sourced I/O>)
We can then use that module hierarchically to generate a 3-to-8
     decoder:
module decoder3to8(<LLM-generated I/O>)
<LLM-generated module>
endmodule
We can then use these modules to hierarchically generate a 5-to
     -32 decoder:
module decoder5to32(
```

(b) Final prompt for 5-to-32 decoder with compressed version of 2-to-4 decoder.

Figure 3: Example of Hierarchical Verilog Decoder Implementation using text-completion LLM. Model outputs are in blue.

4 Methods and Model Selection

We selected eight relevant LLMs that are at the cutting edge of the field. Table 1 shows the full list of models selected and their results on each of the module benchmarks. Of those eight, six are open-source. For those open-source models, all benchmarking inference was run on a single NVIDIA A100 80GB GPU. Inference for the GPT models was run using the OpenAI API. We lock two important model parameters during inference: temperature, which is commonly thought of as a "creativity" value and determines how random the LLMs token generation is, and top-p, which sets a probability threshold for generated tokens, allowing only those tokens above the threshold to be selected from the probability distribution. Higher temperature and lower top-p lead to more random outputs, and vice-versa. Temperature values for each model were locked at 0.5, and top-p for each model was set at 0.9.

We chose to test the unspecialized Llama 2 [20], and the more recent Llama 3 [1] models to see how generalist open-source LLMs compare to specialized models. We hypothesized that even these models would see considerable performance improvements via hierarchical prompting. As a more specialized option, we include the Code Llama [16] model which is fine-tuned on code. It is effective at HDL generation despite not being its intended purpose. We test a pair of LLMs fine-tuned for Verilog generation, namely VeriGen 16b [18] and RTL-Coder [11].We elected not to use the baseline models for each of these, as generalist Llama models of similar size are included in our list. Finally, we fine-tuned our own Code Llama-Verilog model to make the already competitive baseline Code Llama more effective. Our open-source contenders were compared against GPT-3.5 Turbo, GPT-4 black-box LLMs [15].

5 Results and Evaluation

We ran each benchmark using the selected LLMs for 10 iterations per model-method-module combination to get a more statistical sense of how each model performs. The NH experiments still utilized the tool feedback loop for error fixing, but had no hierarchy applied to prompting. Ten iterations per module were allowed. We tracked both the *pass@k* values and the wall-clock time it took to generate the outputs. The *pass@k* metric is defined as the likelihood that one or more of the top-k LLM-generated modules will pass the testbench [4]. Mathematically:

$$pass@k = 1 - \left[\frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{2}$$

n is the number of generation attempts, c is the number of correct attempts that pass testing, and k is success threshold.

Table 1 shows the *pass@k* values for each model-method-module permutation on the benchmark. Hierarchical prompting boosts performance on complex designs, and especially so on on weaker models that fail completely with standard NH prompting. Consider the performance of Llama-2, which is not intended for code generation, much less HDL. Without hierarchical prompting, it fails at every task on the benchmark as it often cannot generate Verilog syntax. It fails on simpler submodules like 8-to-1 multiplexers. However, when guided hierarchically, it has the potential to succeed at even some complicated tasks.

Comp. Results	LLM Used	Llama 2		Code Llama		VeriGen		CL-Verilog		RTL-Coder		Llama 3		GPT-3.5		GPT-4	
	Parameters	13b		13b		16b		13b		7b		8b		~ 20b*		~ 1.8t*	
	Open/Closed	Open		Open		Open		Open		Open		Open		Closed		Closed	
Module benchmark	Prompt method Results	NH	H	NH	H	NH	H	NH	H	NH	H	NH	H	NH	H	NH	H
64-to-1 Multiplexer	pass@1:	0.0	0.4	0.0	0.7	0.0	0.8	0.3	0.8	0.0	0.4	0.1	0.8	0.0	0.7	0.2	0.9
	pass@5:	0.0	0.976	0.0	1.0	0.0	1.0	0.916	1.0	0.0	0.976	0.5	1.0	0.0	1.0	0.78	1.0
	Avg. Time (s):	621.54	342.63	634.01	302.65	642.11	310.64	573.21	315.28	407.88	345.72	521.04	306.27	606.43	327.34	1325.69	507.54
5-to-32 Decoder	pass@1:	0.0	0.5	0.0	0.8	0.1	0.7	0.1	0.9	0.2	0.7	0.0	0.7	0.0	0.8	0.4	1.0
	pass@5:	0.0	0.996	0.0	1.0	0.5	1.0	0.5	1.0	0.78	1.0	0.0	1.0	0.0	1.0	0.976	1.0
	Avg. Time (s):	567.85	274.33	543.24	310.41	577.98	333.83	566.48	329.65	475.2	361.73	488.92	301.97	532.41	215.33	829.26	379.65
32-bit Barrel Shifter	pass@1:	0.0	0.0	0.0	0.2	0.0	0.3	0.0	0.4	0.0	0.0	0.0	0.1	0.0	0.1	0.3	0.7
	pass@5:	0.0	0.0	0.0	0.78	0.0	0.916	0.0	0.976	0.0	0.0	0.0	0.5	0.0	0.5	0.917	1.0
	Avg. Time (s):	322.47	79.51	301.68	51.23	401.65	43.7	296.54	35.21	256.32	61.23	309.22	29.64	312.08	18.27	450.66	42.44
4x4 Systolic Array	pass@1:	0.0	0.0	0.0	0.2	0.0	0.5	0.0	0.5	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.5
	pass@5:	0.0	0.0	0.0	0.78	0.0	0.996	0.0	0.996	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.996
	Avg. Time (s):	1358.99	456.22	1312.05	422.45	1472.91	467.38	1342.15	481.27	1021.44	503.24	1101.77	325.18	1276.39	297.63	2452.41	402.26
UART 8-bit	pass@1:	0.0	0.2	0.0	0.4	0.0	0.4	0.0	0.6	0.0	0.2	0.0	0.4	0.0	0.7	0.0	0.8
	pass@5:	0.0	0.78	0.0	0.976	0.0	0.976	0.0	1.0	0.0	0.78	0.0	0.976	0.0	1.0	0.0	1.0
	Avg. Time (s):	2482.17	672.9	2100.58	614.07	2603.12	682	2529.74	673.21	1763.06	699.53	1754.22	573.64	1800.45	564.23	3144.22	752.76
AES Block Cipher	pass@1:	0.0	0.0	0.0	0.2	0.0	0.3	0.0	0.3	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.5
	pass@5:	0.0	0.0	0.0	0.78	0.0	0.916	0.0	0.916	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.996
	Avg. Time (s):	3212	783.54	3201.78	795.4	3456.11	809.54	3203.55	732.14	2387.92	960.29	2603.51	706.59	2652.42	694.71	3822.64	806.89

Table 1: Results on hierarchical design benchmarks for open-source and closed-source proprietary LLMs. The results for each model are separated by prompting method — (i) flat, non-hierarchical (NH) or hierarchical (H) . We report *pass@k* for $n = 10$ attempts, and the average generation time per benchmark. Top open-source hierarchical performers are in green, and top non-hierarchical performers are in blue; generation time is the tie-breaker where needed.

When applied to specialized models, the results are even more impressive. Hierarchical prompting enables open-source LLMs to outperform standard flat prompting outputs from GPT-3.5 and GPT-4. Furthermore, these techniques on the GPT models yield further performance improvement, allowing GPT-3.5 and GPT-4 them to succeed consistently on difficult modules. Overall, every LLM sees improvement with hierarchical prompting.

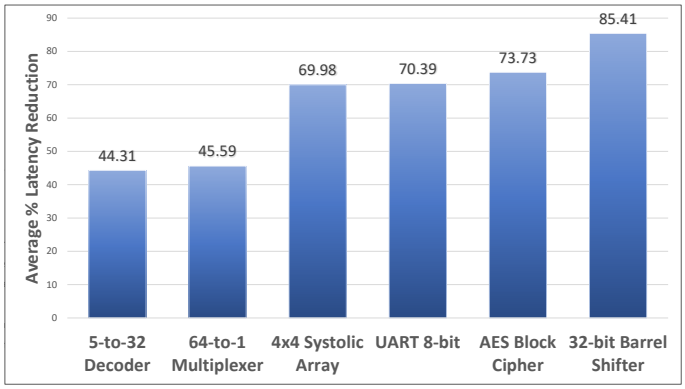

Figure 4: Hierarchical prompting yields consistent time savings vs. flat prompting, as seen by average % latency reduction. More time savings are seen on modules which are difficult or impossible to generate non-hierarchically, or on those for which flat outputs are longer than hierarchical alternatives.

Hiearchical Prompting also reduces code generation time. Figure 4 reports the average percent time reduction due to hierarchical prompting across all LLMs for each of our benchmark. We see that the more difficult modules tend to have a much greater reduction in time, as successful hierarchical generation allows us to skip the lengthy error-handling process which sees the models re-generating past outputs and accounts for the majority of generation time.

PGHP Accuracy by LLM	Code Llama	VeriGen	CL-Verilog	RTL-Coder	GPT-3.5	GPT-4
Multiplexer	0.15	0.15	0.25	0.05	0.2	0.85
Decoder	0.05	0.1	0.1	0.15	0.15	1.0
Barrel Shifter	0.05	0.0	0.1	0.0	0.0	0.35

Table 2: Accuracy of LLM-decided architectural hierarchy out of 20 iterations. The inconsistency of most models when generating their own hierarchical plan is a major contributing factor to the failure of PGHP outside of GPT-4.

5.1 Purely Generative Results

We implemented PGHP techniques for our benchmarks, but saw consistent failure for models *except for* GPT-4. To diagnose the source, we selected 3 of the simplest benchmarks, and used a subset of our LLMs to generate 20 hierarchies each and evaluated them against the golden hierarchy plan from the HDHP version. Most LLMs performed inconsistently on this task, missing key submodules or inserting extraneous submodules (Table 2).

On the other hand, we find that GPT-4 significantly improves over flat prompting with PGHP, as shown in Table 3. PGHP is able to generate valid implementations for Systolic Array and UART on which GPT-4 fails completely in flat NH mode. Further, we see substantial gains in accuracy for the three simpler benchmarks. As we will see next, PGHP with GPT-4 is also successful in automatically designing a single-cycle MIPS processor.

GPT-4 PGHP	Multiplexer	Decoder	Barrel Shift.	Sys. Array	UART	AES
pass@1	0.5	1.0	0.3	0.1	0.3	0.0
pass@5	0.996	1.0	0.917	0.5	0.916	0.0

Table 3: *pass@k* when applying purely generative hierarchical prompting (PGHP) to GPT-4 on each benchmark. Note improvement over flat prompting for most modules.

5.2 Identifying Common Failure Modes

We often see errors when LLMs generate text for too long and lose the original context of their goal. This occurs both for conversational and text-completion LLMs. We circumvented this by requesting no additional elements be generated via our system prompt, and re-inputting earlier context as a global prompt. Once a task is completed, text-completion LLMs tend to hallucinate. A common example is the unnecessary generation of testbenches or a random additional module. This is avoided by truncating outputs at a useful end-token, usually the "endmodule" in Verilog.

Conversational LLMs can fall into "perseverative" loops, a termed borrowed from neurology [3], continuing to repeat actions or words when the stimulus that brought on those behaviors has stopped, or when a competing stimulus has occurred that would normally trigger new behavioral routes. One example is the continued use of an unnecessary always block when writing barrel shifter with non-hierarchical prompting, which can occur even when the LLM receives direct/detailed human feedback. As seen in Figure 5 in the Appendix, the LLM will confirm it has done as asked, while continuing to output the same syntax as before. Avoiding such behavioral loops is another benefit of hierarchical prompting.

6 Case Studies and Processor Generation

To stress test our techniques, we hierarchically generated a full MIPS 16-bit single-cycle processor using GPT-3.5 and our Code Llama-Verilog model based on the PGHP paradigm. Flat prompting is unable to approach a functional processor design without considerable human oversight [2], but we hypothesized that hierarchy would bridge this gap and allow for automation. We tasked each model to first define a hierarchical structure for the processor as a list of submodules, then generate the processor stepwise.

The models generated most necessary submodules, but missed key elements and struggled with assigning wire and signal names uniformly across modules, as seen in prior experiments. Tool feedback was helpful, but insufficient to bridge these issues. Ensuring all input and output wires/signals were named appropriately required human intervention and certain submodules like the control unit had to be directly requested, but all functional components were LLM-generated. After these interventions, we were able to synthesize the processors in Vivado, and successfully simulate processor instructions. We repeat this process by generating a RISC-V 32-bit processor utilizing GPT-4. Many of the issues present in GPT-3.5 are less problematic in GPT-4, and required much less human intervention. The newer model is better at wiring up interconnected modules and produces detailed descriptions of hierarchical architectures.

To fully test this capability, we implemented the PGHP technique once more to generate another MIPS core via GPT-4 with no human intervention. After iterative tool feedback, GPT-4 converged on a synthesizeable processor that covered a version of the full MIPS ISA. Design and simulation results are shown in the Appendix. Figure 6 shows the RTL and Figure 7 shows a waveform for this PGHP-sourced processor.

	Hierarchical			Non-Hierarchical			Savings
	I/P (tokens)	O/P (tokens)	Cost $	I/P (tokens)	O/P (tokens)	Cost $	%
Multiplexer	92	2376	0.00484	91	3283	0.00666	27.23
32-b Barrel	262	1977	0.00422	191	4268	0.00873	51.69
16-b MIPS	434	14226	0.02868	1243	31033	0.06314	54.58
32-b RISC-V	795	17310	0.03542	1593	42338	0.08593	58.8

Table 4: Cost of input (I/P) tokens processed and output (O/P) tokens generated of the modules using hierarchical and non-hierarchical prompting. Values based on GPT-3.5 tokenizer pricing.

We posit that this is the first-ever purely LLM-designed processor. That is, the design decisions were made entirely by the LLM with no human input, and all error handling was done automatically with tool feedback. Beyond the initial prompt of *"Please define the necessary submodules in a 16-bit single cycle MIPS processor,"* no human design intervention was required. We also see that the time taken to generate our processors is on the order of minutes, rather than multiple hours as seen in past methods. The time taken to complete the PGHP-based processor was 23 minutes, 37.85 seconds.

In order to get a sense of financial cost savings, we calculate the price-per-token and number of tokens generated when applying our hierarchical pipeline to GPT-3.5. We compare results for our full multiplexer hierarchy, our 32-bit barrel shifter, our MIPS processor, and our RISC-V processor. Table 4 contains the full cost analysis. As one might expect, complex modules lead to higher costs and achieve more financial savings when generated hierarchically.

7 Conclusion

In this paper, we have proposed and evaluated Hierachical Prompting as a key tool for automated HDL code generation for complex modules. We show that with hierarchical prompting, even smaller fine-tuned LLMs can correctly generate HDL for complex modules, when traditional flat prompting fails. On powerful models like GPT-4, hierarchical prompting is even more impressive, enabling the automatic generation of a single-cycle MIPS core. Overall, these methods give considerable insight into the potential of LLMs with either manually specified or automatically extracted design hierarchy.

There is considerable potential in this line of inquiry. We hope to include additional hardware design methods as part of a larger pipeline in the future. Considering the successes of methods like high-level synthesis (HLS), it stands to reason that leveraging different tools for different tasks could further improve results. We plan to fine-tune additional models with hierarchy in mind. Careful training dataset formulation could potentially lead to models which excel at hierarchical tasks, and may bridge the gap on PGHP performance for smaller models. We hope to expand evaluation resources for future benchmarking efforts to increase the strength of our *pass@k* metric, ideally $n = 200$ samples per test.

References

[1] AI@Meta. 2024. Llama 3 Model Card. (2024). https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[2] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. *arXiv preprint arXiv:2305.13243* (2023).

[3] Hugh W. Buckingham and Sarah S. Christman. 2008. Chapter 12 - Disorders of Phonetics and Phonology. In *Handbook of the Neuroscience of Language*, Brigitte Stemmer and Harry A. Whitaker (Eds.). Elsevier, San Diego, 127–136. https://doi.org/10.1016/B978-0-08-045352-1.00012-4

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[5] S. Cheng, P. Jin, Q. Guo, Z. Du, R. Zhang, Y. Tian, and Y. Chen. 2023. Pushing the Limits of Machine Design: Automated CPU Design with AI. *arXiv preprint arXiv:2306.12456* (2023).

[6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[7] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[8] X. Jiang, Y. Dong, L. Wang, Q. Shang, and G. Li. 2023. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689* (2023).

[9] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. arXiv:2404.00971 [cs.SE]

[10] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. arXiv:2309.07544 [cs.LG]

[11] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024. RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution. arXiv:2312.08617 [cs.PL]

[12] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2023. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model. arXiv:2308.05345 [cs.LG]

[13] James Manyika. 2023. *An overview of Bard: an early experiment with generative AI.* Technical Report. Technical report, Google AI.

[14] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[15] OpenAI. 2023. GPT-4 Technical Report. http://arxiv.org/abs/2303.08774. https://doi.org/10.48550/arXiv.2303.08774 arXiv:2303.08774 [cs].

[16] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023).

[17] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[18] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2023. VeriGen: A Large Language Model for Verilog Code Generation. *arXiv preprint arXiv:2308.00708* (2023).

[19] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. 2023. AutoChip: Automating HDL Generation Using LLM Feedback. arXiv:2311.04887 [cs.PL]

[20] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288* (2023).

[21] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[22] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]

A Additional Figures

Top Modules	Submodules
64-to-1 Multiplexer	2-to-1 mux 4-to-1 mux 8-to-1 mux 16-to-1 mux 32-to-1 mux 64-to-1 mux
5-to-32 Decoder	2-to-4 decoder 3-to-8 decoder 5-to-32 decoer
32-bit Barrel Shifter	8-bit Barrel Shifter Rotation Control 32-bit Barrel Shifter
4x4 Systolic Array	Processor Element Control Logic Top-level 4x4 Array
8-bit UART	Baud Rate Generator Receiver Transmitter State definitions Top-level UART
128-bit AES Block Cipher	S-box Key Memory Encipher Block Decipher Block Control Logic Helper Functions(e.g.: inverse shift rows, mix columns, etc.)

Table 5: Top-level modules and corresponding submodules of the hierarchical benchmarks.

User Prompt:

"This output utilizes an always block. Please use purely combinational logic. That is, directly assign output values for each shift amount."

LLM Output:

"My apologies for the confusion. Here's the modified code without the always block, using direct assignments for each possible shift amount:"

```
module data_dependent_rotate(input [7:0] data_in, input [2:0]
shift_amount, output reg [7:0] rotated_data);

    // Manual assignment based on shift_amount

    always @* begin

        case(shift_amount)

            3'b000: rotated_data = data_in;

                        Code continues...
```

Figure 5: Perseveration-like behavior in GPT-3.5 when asked for rare syntax.

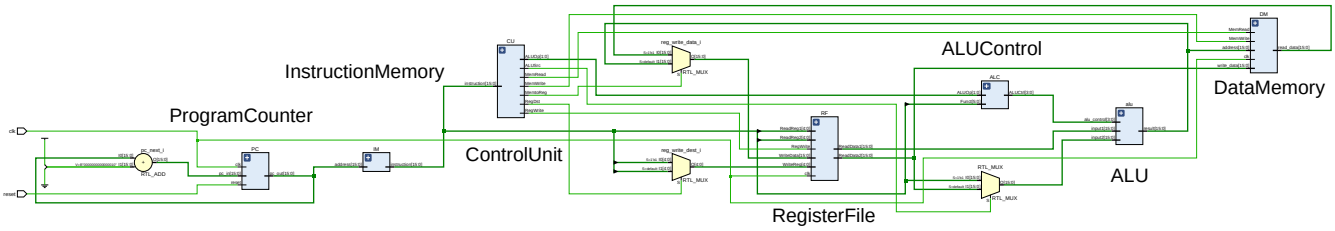

Figure 6: Elaborated Design Schematic of LLM-Generated MIPS Processor. Zoom in for submodule information.

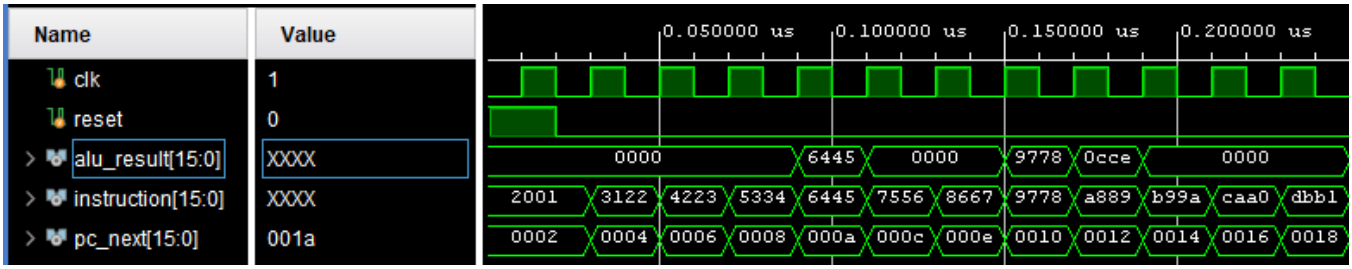

Figure 7: Sample waveform of running a series of instructions. Our first two instructions test the load and store functions, moving values between the registers and data memory. We then load two values from data memory, 12,006 and 13,663, into the registers R4 and R5 respectively. We then test the ADD function to sum them to 25669, or 6445 in hex. This intentionally echoes the value of the instruction for easy confirmation on the waveform. We then load two more values, 38,776 and 1,104 to test the OR instruction, to once again echo the instruction value of 9778 hex. We then test the SUBI instruction by subtracting 35,498 from 38,776 to get 3,278, or 0CCE hex. We then store all of our outputs thus far into data memory at various addresses to confirm successful saving. More thorough testing omitted for brevity. Though the LLM implements opcodes that are notably different from the standard MIPS ISA, all instructions are present and functional. Full instruction series: LW R7, 8(R7); SW R8, 9(R8); LW R4, 1(R4); LW R5, 3(R5); ADD R6, R4, R5; LW R7, 5(R7); LW R8, 6(R8); OR R9, R7, R8; SUBI R10, R9, 35498; SW R6, 9(R0); SW R9, 10(R0); SW R10, 11(R0);

B Artifact Evaluation

All relevant artifacts, including scripts, are publicly available at:

https://github.com/ajn313/ROME-LLM

They are additionally publicly archived at:

https://zenodo.org/records/13323449

https://figshare.com/projects/ROME-LLM/214771

We provide an example Python notebook to be used in Google Colab that includes all necessary elements for convenience. All that is required is an OpenAI API key with GPT-4 usage enabled. We present a novel algorithm here in the form of our automated design pipeline. It utilizes both an error handling feedback loop in addition to a hierarchical feedback loop in order to build on simpler module to make more complex circuits & structures. The benchmark we utilized is the novel Hierarchical ROME LLM benchmark described within the paper. This includes a number of modules to be tested. We include unit tests to validate the performance of our tool.

All simulations are run using the open-source iVerilog tool which will be downloaded by the Colab notebook and will run via script calls in the Colab environment. No other compiler/simulator is required. All software dependencies are handled by the Colab notebook, but a list of dependencies will additionally be available on the GitHub. The notebook will default to using GPT-4 as the model, though this can easily be changed to another OpenAI model. No download is necessary as it will run remotely through the OpenAI API. Instructions will be included on how to instead use an open-source model, including our own CL-Verilog model, though it will be easiest to test using GPT. Additionally, CL-Verilog will be released on Hugging Face.

Running all benchmarks may take numerous hours, but we include a subset of examples that should be less time consuming while still displaying performance. Measurements can include pass@k, time to completion, or binary pass/fail. These values are affected by the stochasticity of LLMs and may not match up perfectly to our own observations, for better or worse. Results should be reproducible via our Colab notebook, though there is likely to be notable variance between runs, as our pass@k measurements show. The final output should be a functional version of the target module which passes testbench simulation.

The testbenches can be downloaded from GitHub and uploaded to Colab, or cloned directly from the repo. No more than a few megabytes of local disk space should be required if downloading is preferred, as the testbenches are on average around 2kb. One option for quickly opening our notebook is githubtocolab.com, which will open the notebook directly in Google Colab from GitHub. Instructions are on the tool's site.

MIT license is used for code where applicable. All artifacts will be archived on Zenodo and FigShare and publicly available on GitHub at the above links.

Zenodo DOI:

https://doi.org/10.5281/zenodo.13323449