Improving Large Language Model Hardware Generating Quality through Post-LLM Search

Kaiyan Chang[1,3] **Haimeng Ren**[2] **Mengdi Wang**[1,3] **Shengwen Liang**[1] **Yinhe Han**[1]
Huawei Li[1] **Xiaowei Li**[1] **Ying Wang**[1]
[1]Institute of Computing Technology, Chinese Academy of Science
[2]ShanghaiTech University
[3]University of Chinese Academy of Science
Beijing, China 100190
changkaiyan@live.com,renhm2022@shanghaitech.edu.cn
{wangmengdi17s,liangshengwen,yinhes,lihuawei,lxw,wangying2009}@ict.ac.cn

Abstract

As large language models (LLMs) like ChatGPT exhibited unprecedented machine intelligence. However, existing LLM-based hardware generating frameworks generate verilog register transfer language(RTL) without considering its performance, power, area(PPA). To overcome this challenge, we design a post LLM search approach to **merge design space exploration(DSE) process into current LLM hardware generation workflow**, which enables the PPA optimization. At first, our framework begins by generating prompts for the LLM, which then produces initial Verilog programs. Second, an output manager corrects and optimizes these programs before collecting them into the final design space, which is constructed as a HDL search tree. Eventually, our work will do search through this space to select the optimal design under the target metrics. The evaluation shows that our approach improves generating Verilog quality, and broader design optimization space compared to prior work and native LLMs alone.

1 Intorduction

Recent advancements in large language models (LLMs) demonstrate their potential for automatic Hardware Description Language (HDL) code generation from high-level prompts[1, 2]. Envisioning a future where natural language expresses hardware logic design requirements holds the promise of revolutionizing chip design by maximizing creativity and complexity. While higher-level programming interfaces are emerging, they signify progress towards this vision. Fundamental generative models take natural language input and produce the target Verilog program as output. For example, ChatGPT, a state-of- the-art model, facilitates natural language programming. However, LLMs face limitations in adapting to chip design tasks, generating raw programs without guarantees of hardware-level correctness or enabling exploration of PPA in the design space. To address these challenges, we propose a natural language chip logic design method based on a post-LLM search tree, without modifying the large model. The GPT output manager generates potential programs, and an enumeration search selects the best design with the desired PPA. Experimental results show improved effectiveness. Our framework, ChipGPT, integrates the design exploration process into LLM-based hardware generation (Fig.1). This represents **a simple yet significant step towards formally integrating design space exploration into LLM-based hardware design flows**. Contributions include:

- Proposing a hardware description language search tree to generate Verilog programs without modifying LLM weights, seamlessly integrating with the latest LLM APIs.

Machine Learning for Systems Workshop at 37th NeurIPS Conference, 2023, New Orleans, LA, USA.

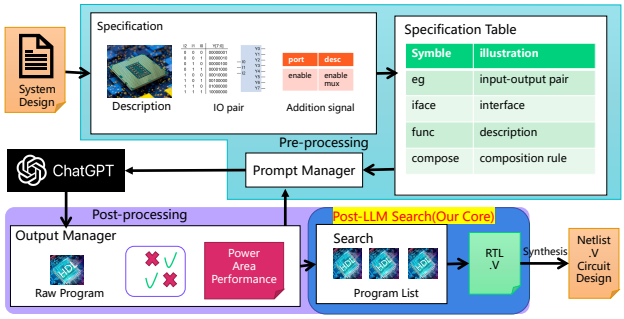

Figure 1: Our Framework uses a Hardware Generation Workflow with Core Part at the right bottom.

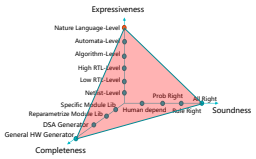

Figure 2: Productivity

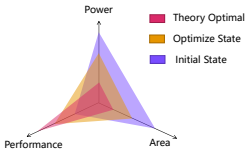

Figure 3: Design Space

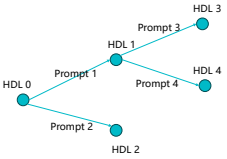

Figure 4: HDL Search Tree

- Overcoming challenges of power/area/performance agnostic LLMs in chip design through a post-LLM search method with an efficient output manager.
- ChipGPT outperforms previous agile chip logic design methods and native ChatGPT, showing improvement in programmability and scalability, with potential for larger-scale chip design extension.

2 Background and Motivation

Emerging hardware-driven large language models enable greater programmability to support HDL designers. These powerful models allow for more flexibility to specify desired behaviors directly in natural language. Instead of hand-crafting traditional HDL code, designers can leverage large language models to translate high-level descriptions into optimized hardware implementations. These large language models a natural language-level expressivity, as demonstrated in Fig. 2.

2.1 Generating hardware with Large Language Model

Researchers have explored the use of LLMs, where the model input is a natural language description of the hardware module, and the output is corresponding Verilog code. Benchmarks in [1, 3] demonstrate that these models' capacity to alleviate the burden on hardware designers. While the existing LLMs do not fully cover the entire space of Verilog generation, progress has been made on fine-tuning for completer code [4], generic RTL generation[5] and enhancements to the existing open source LLMs[6]. In hardware security, LLMs can help fix bugs [7] and generate assertions[8, 9]. Beyond single-sentence models like GPT-3, conversational LLMs have also proven capable for hardware generation [5]. LLMs are being applied to automate design for novel technologies like in-memory computing [10] and quantum [11]. However, these approaches do not directly optimize the quality of the generated HDL code itself. Although [12] applies template-based generation for AI chips, the general LLM generated hardware still needs to be improved. This work aims to bridge this gap by guiding LLM generation to achieve optimal implementations whithin physical design constraints.

2.2 Motivation

Challenge 1: Limited PPA Awareness in Hardware LLMs. Hardware LLMs lack awareness of PPA metrics, hindering optimal program generation. In Fig. 3, ideal program PPA results are in the red area, while LLM-generated programs remain PPA-agnostic due to their training process. Models like InstructGPT and ChatGPT, trained with general reinforcement learning, produce intuitively good

Table 1: ChatGPT Program Ranking Model on button count description(RP: Raw Programe)

Program	Power (mW)	Area (μm^2)	Rank Model
RP 0	4.2900e-02	139.199999	5
RP 1	1.3593e-02	265.200006	4
RP 2	1.0704e-02	193.600004	3
RP 3	9.7253e-03	187.200004	2
RP 4	1.0283e-02	196.000003	1

Table 2: Search Result of Button Count Problem

Select by Power	Select by Area	Direct Select
P3	P0	P0
P4	P3	P1
P2	P2	P2
P1	P4	P3
P0	P1	P4

programs but are not optimized for PPA. To address this, an output manager is introduced in Sec. 3.2 to enhance PPA optimization, aiming for the yellow area in Fig. 3.

Challenge 2: Complexities in Design Space Exploration. Navigating the design space for optimized implementations is challenging. Traditional exploration relies on clear parameters and well-defined design points, but the language model generation process lacks straightforward adjustment knobs. The proposed approach transforms the LLM's creative process into a state-transformed HDL search tree for enumerated searches, addressing the complexities in design space exploration.

3 Exploring Design Space by Post-LLM Search

3.1 Formallized Hardware Description Language Search Tree

To optimize the generated hardware for key metrics such as PPA, we introduce a search space that extends beyond the large language model, as depicted in Fig. 4. Each node represents a potential HDL program generated by the language model. In a conversational model, the complete sequence of prompts and model outputs cyclically returns to the model to elicit the subsequent response. This dynamic forms a state transition graph, where the generated programs act as states, and prompts drive the transitions between them. Conceptualizing prompt crafting and HDL code generation as a search tree facilitates structured exploration of the extensive program space the language model can produce. Traversing the tree allows for the identification of Pareto optimal points that balance trade-offs in PPA, guiding code generation towards high-quality hardware implementations.

3.2 Cost Manager

As discussed in Section 2.2, PPA are crucial optimization targets in chip logic design. However, large language models lack inherent support to directly optimize for these physical constraints. The model's internal reward system provides only a general quality ranking of generated programs. To address this limitation, our framework integrates a cost manager to refine PPA following the initial language model search. In this stage, EDA tools are employed to evaluate the PPA of each HDL program enumerated in the search tree. The results are appended to the program list (Equ. 1). A guided search is then conducted to select the optimal target version that aligns with design goals. For example, as shown in Tab. 1, the language model generates multiple code variants with different PPA trade-offs. The cost manager facilitates exploring these options to identify the one that best aligns with specific PPA priorities. By combining the creative potential of large language models with precisely targeted PPA optimization, the proposed approach aims to deliver high-quality hardware implementations.

3.3 Enumerative Search

A key question is whether an additional search over model outputs is necessary. Table 3.3 reveals the language model's internal ranking does not consistently match the order of optimal PPA. For the button counter module (see Table 3.3), the selected program heavily depends on the target optimization goal, with different solutions highlighted in yellow. This mismatch indicates the necessity to enhance the output manager with a guided search algorithm.By exploring the limited design space of plausible programs (typically less than 10), the approach extracts improved solutions. With a modest output size, an enumeration search suffices to select the optimal result based on the cost manager's PPA

evaluations. This search offers targeted refinement, ensuring the final hardware aligns with specified PPA constraints rather than solely following the model's default preferences.

$$program_list = \{\{p_1, PPA_1\}, \ldots, \{p_n, PPA_n\}\} \tag{1}$$

where p denotes a program. Enumerate search selects the best program in the program list of performance, area and power, which relies on the programmers' target.

4 Evaluation

We utilize the ChatGPT model (GPT-3.5) from OpenAI's website. Design Compiler and 65nm technology are employed for power and area evaluations, while simulation with handwritten testbeds provides performance cycle numbers. Line of code measurements use the `cloc` tool. **Due to page constraints, high-level evaluation results are presented here, with detailed analyses in Appendix B for interested readers.** To validate our workflow with ChatGPT, we compare it to the baseline ChatGPT model, which generates naïve code using only the module description as the prompt. Comparisons with traditional agile workflows involve Chisel[13] and high-level synthesis (Xilinx Vivado HLS). For fairness: 1) unroll directives are added to pipeline the HLS design, and Synopsys Design Compiler is used for power and area measurements, mirroring our approach. 2) HLS and Chisel implementations share the same specification, developed by the same 2-year graduate student, and are evaluated with an identical testbed suite. In Fig. 5, simple workloads show no significant PPA optimization relative to the baseline, given the limited raw program candidates. However, for complex workloads, our framework's targeted optimization at the search stage reduces average area by 47% (0.53x) and overall average area by 35% (0.65x) compared to the original ChatGPT model. This joint optimization of PPA objectives and program coherence for complex designs with ample raw candidates achieves substantial optimization.

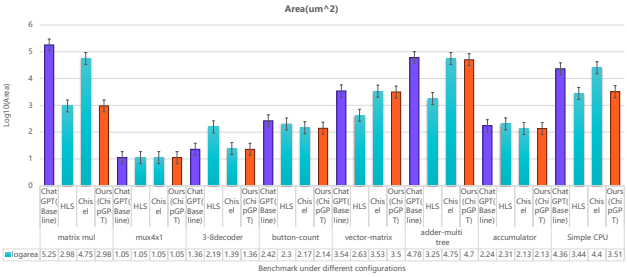

Figure 5: Area Comparison under Area Optimization Setting. y-axis is normalized with `log10()`.

5 Conclusions and Future Work

This paper explores natural language hardware design and proposes a post-LLM search method to improve large language model hardware generating quality. This is a simple yet significant step towards formally integrating design space exploration into LLM-based hardware design flow. By harnessing language, ChipGPT significantly accelerates chip development. Our method is an interface for GPT to address natural language hardware design and PPA optimization, which has an area reduction of 47% compared with the original ChatGPT in area target optimization mode.

References

[1] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[2] Hammond Pearce, Benjamin Tan, and Ramesh Karri. Dave: Deriving automatically verilog from english. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, pages 27–32, 2020.

[3] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl generation with large language model, 2023.

[4] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation. *arXiv preprint arXiv:2308.00708*, 2023.

[5] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. Chip-chat: Challenges and opportunities in conversational hardware design. *arXiv preprint arXiv:2305.13243*, 2023.

[6] Zhuolun He, Haoyuan Wu, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. Chateda: A large language model powered autonomous agent for eda. *arXiv preprint arXiv:2308.10204*, 2023.

[7] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. Fixing hardware security bugs with large language models. *arXiv preprint arXiv:2302.01215*, 2023.

[8] Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. Llm-assisted generation of hardware assertions. *arXiv preprint arXiv:2306.14027*, 2023.

[9] Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. From rtl to sva: Llm-assisted generation of formal verification testbenches, 2023.

[10] Zheyu Yan, Yifan Qin, Xiaobo Sharon Hu, and Yiyu Shi. On the viability of using llms for sw/hw co-design: An example in designing cim dnn accelerators. *arXiv preprint arXiv:2306.06923*, 2023.

[11] Zhiding Liang, Jinglei Cheng, Rui Yang, Hang Ren, Zhixin Song, Di Wu, Xuehai Qian, Tongyang Li, and Yiyu Shi. Unleashing the potential of llms for quantum computing: A study in quantum architecture design. *arXiv preprint arXiv:2307.08191*, 2023.

[12] Yonggan Fu, Yongan Zhang, Zhongzhi Yu, Sixu Li, Zhifan Ye, Chaojian Li, Cheng Wan, and Yingyan Lin. Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models. In *ICCAD*, 2023.

[13] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. *Design Automation Conference*, pages 1216–1225, 2012.

A Workflow Detail

A.1 Prompt Manager PM

Due to the context-aware feature of GPT, the prompt manager directly determines the quality of the generated code.

The goal of the prompt manager is to serialize natural language software specifications into a structured prompt format. By formatting the specifications in this way, prompts can be used as input for GPT. With well-designed prompts, GPT can generate code that successfully captures the details and purpose of the original specifications.

$$arch =< setup, submodule^N, compose > \tag{2}$$

$$submodule =< \{funcdesc, iface\}, addition > \tag{3}$$

To improve prompt quality, we design a template-based prompt manager where prompts are interconnected within templates. We first provide a formal representation for the prompt manager. Let

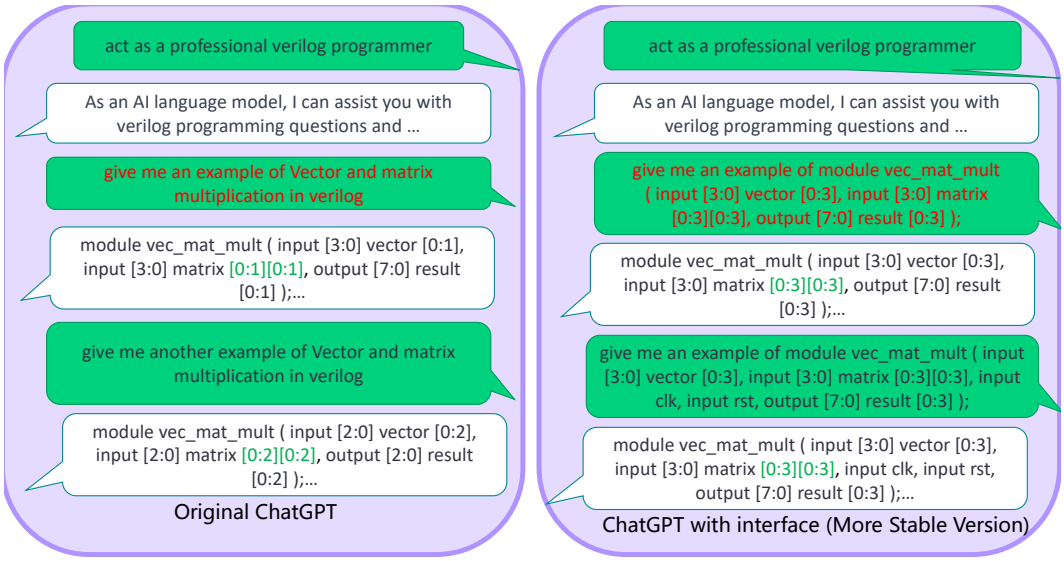

Figure 6: An observation of interface principle in prompt manager

$S = \{(Q_1, A_1), (Q_2, A_2), \ldots, (Q_N, A_N)\}$ denote a dialogue sequence with N question-answer pairs. GPT takes queries as input and generates the final answer. We define $arch$ in Equ. 2 as a query sequence, or prompt sequence. The query sequence comprises three partitions. Elements enclosed in $<>$ represent a sequence of prompts, while elements in $\{\}$ represent groups combined into a single prompt. The first setup prompt $<$ acts as a proficient Verilog programmer$>$ establishes an execution environment enabling the following queries to operate in a Verilog context. Next, for the basic modules in the specification, GPT generates N modules following the interface model principle in A.1.1 and post-addition principle in A.1.2. Finally, there is a top module following the $compose$ rule that connects these submodules from the second step. This composition adheres to the module composition principle in A.1.3.

A.1.1 Principle for Interface Model

An intuitive approach to generate Verilog program is to specify the module function as a prompt only. However, omitting the module interface declaration from the prompt makes the GPT ignore chip port specifications, where the generated module cannot be integrated seamlessly into the testbench environment. Moreover, including the interface declaration within the prompt defines attributes like bit widths and array sizes that are essential for generating high-quality internal module code. This enhanced context enables GPT to produce consistent versions with the same interface, as depicted in Fig. 9.

To introduce the interface-based model principle into our template-based prompt manager PM, we examine current port definitions in specifications and hardware description languages (HDLs), such as Verilog. Port definitions comprise five elements: 1. Direction (D): Whether the port is input, output, or inout 2. Data width (W): The bit width of the port (e.g. 32 bits) 3. Variable name (VN): The name given to the port (e.g. instruction) 4. Array size (S): The number of elements in an array port 5. Port description (PD): A text description of the port's purpose.

$$iface = \{D, W, VN, S, PD\}$$

Among the port definition elements, the variable name (VN) is particularly important. Because GPT is a variable name-aware model, the names given to ports help the model infer the appropriate program structure.

No Post-Addition Principle	Post-Addition Principle
Prompt: Give me an example of on chip network with ready and valid signal	Prompt 1: Give me an example of on chip network Prompt 2: add ready and valid signal
module network(input clk, input [7:0] data_in, input valid_in, output ready_in, output [7:0] data_out, output valid_out, input ready_out); // valid is not a output signal ...(Ignore the implementation)	*module* network(input clk, input rst, input [4:0] src_addr, input [4:0] dst_addr, input [7:0] data, input ready, output [7:0] data_out, output valid); ... (Ignore the implementation)

Correct

Figure 7: An example of post-addition Principle in LLM.

A.1.2 Post Addition Principle

In a production environment, it is common to add multiple cross-module handshake signals to existing modules(*e.g.* ready-valid mode). However, combining handshake signals directly into the interface prompt reduces the accuracy of the generated original program. GPT struggles to well understand programs created this way. The reason is that LLM's reward model always selects the highest-ranked raw program version based on both the handshake and function. It considers the weights of the handshake and function together. However, programmers prioritize the correctness of the function.

Based on the above observations, we place additional information such as cross-module handshake signal declarations after generating the first raw program in Equ. 3. Because this supplemental information is added after the primary module details, we refer to it as the "post-addition principle". For example, in Fig. 10, we show the effectiveness of the post-addition principle. In the left column, the handshake signals are added to the same prompt. As a result, the language model generates incorrect implementation where valid cannot be an output port. The right column implements the principle of post-addition, where the implementation is correct.

A.1.3 Module Composition Principle

An architecture-level chip specification includes not only component modules but also a top module to connect them. To gain a high-level overview, chip designers typically take a top-down design approach. However, if we prompt GPT to generate the top-level module directly, it lacks information about the submodules it should reference.

Based on this observation, we propose the "bottom-up composition principle" to improve the quality of the generated raw program. As shown in Equation 4, this principle specifies that **submodule interfaces must be declared before the top module**. For example, $<$ the submodule has interface $\ldots >$ followed by $<$ the top module consists of $\ldots >$. By following this sequence, when GPT generates the top module, the ports of the submodules are clearly defined. As a result, GPT can connect them appropriately. Because this principle is recursive, it can be applied to a wide range of architectural design scenarios. For example, in Fig. 11, we want to design a complex module. On the left, the add module and minus module are defined in the same prompt, where the LLM cannot output the correct implementation of the submodules. However, when applying the composition principle, the LLM can generate correct implementations of the submodules.

$$compose : submodule^{N} \longrightarrow topmodule \tag{4}$$

No Composition Principle	Composition Principle	
Prompt: Define top module and the two number add module output through FIFO to a two number minus module, where the input of minus module are two FIFO connect to two add modules.	Prompt 1: Define minus module as a module to substract two numbers. Prompt 2: Define add module as a module to add two numbers. Prompt 3: Define top module and the three numbers add module output through FIFO to a two number minus_module, where the input of minus_module are two FIFO connect to two add_modules.	

`module add_module(`
`  input clk,`
`  input reset,`
`  input [31:0] a,`
`  input [31:0] b,`
`  output [31:0] sum`
`);`
`// Note: This is unimplement`
`endmodule`
 	
`module minus_module(`
`  input clk,`
`  input reset,`
`  input [31:0] a,`
`  input [31:0] b,`
`  output [31:0] diff`
`);`
`// Note: This is unimplement`
`endmodule`
 	
`// The top module`
	Better
	
	`module minus_module(`
	`  input clk,`
	`  input reset,`
	`  input [31:0] a,`
	`  input [31:0] b,`
	`  output [31:0] diff`
	`);`
	
	`reg [31:0] sub;`
	
	`always @(posedge clk) begin`
	`  if (reset) begin`
	`  sub <= 0;`
	`  end else begin`
	`  sub <= a - b;`
	`  end`
	`end`
	
	`assign diff = sub;`
	
	`// The top module is the same as left`
		`module add_module(`
		`  input clk,`
		`  input reset,`
		`  input [31:0] a,`
		`  input [31:0] b,`
		`  output [31:0] sum`
		`);`
		
		`reg [31:0] add;`
		
		`always @(posedge clk) begin`
		`  if (reset) begin`
		`  add <= 0;`
		`  end else begin`
		`  add <= a + b;`
		`  end`
		`end`
		
		`assign sum = add;`
		
		`endmodule`

Figure 8: An example of composition principle in LLM.

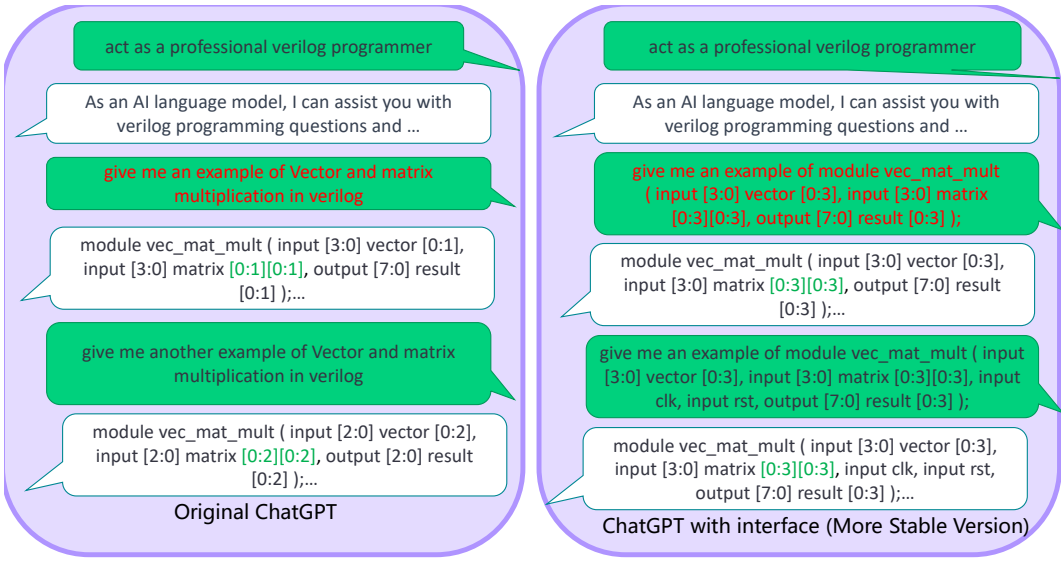

Figure 9: An observation of interface principle in prompt manager

No Post-Addition Principle	Post-Addition Principle
Prompt: Give me an example of on chip network with ready and valid signal	Prompt 1: Give me an example of on chip network Prompt 2: add ready and valid signal
module network(input clk, input [7:0] data_in, input valid_in, output ready_in, output [7:0] data_out, output valid_out, input ready_out); // valid is not a output signal ...(Ignore the implementation)	*module* network(input clk, input rst, input [4:0] src_addr, input [4:0] dst_addr, input [7:0] data, input ready, output [7:0] data_out, output valid); ... (Ignore the implementation)

Correct

Figure 10: An example of post-addition Principle in LLM.

No Composition Principle	Composition Principle	
Prompt: Define top module and the two number add module output through FIFO to a two number minus module, where the input of minus module are two FIFO connect to two add modules.	Prompt 1: Define minus module as a module to substract two numbers. Prompt 2: Define add module as a module to add two numbers. Prompt 3: Define top module and the three numbers add module output through FIFO to a two number minus_module, where the input of minus_module are two FIFO connect to two add_modules.	

module add_module(
input clk,
input reset,
input [31:0] a,
input [31:0] b,
output [31:0] sum
);
// Note: This is unimplement
endmodule
 	
module minus_module(
input clk,
input reset,
input [31:0] a,
input [31:0] b,
output [31:0] diff
);
// Note: This is unimplement
endmodule
 	
// The top module
	module minus_module(
	input clk,
	input reset,
	input [31:0] a,
	input [31:0] b,
	output [31:0] diff
	);
	
	reg [31:0] sub;
	
	always @(posedge clk) begin
	if (reset) begin
	sub <= 0;
	end else begin
	sub <= a - b;
	end
	end
	
	assign diff = sub;
	
	// The top module is the same as left
		module add_module(
		input clk,
		input reset,
		input [31:0] a,
		input [31:0] b,
		output [31:0] sum
		);
		
		reg [31:0] add;
		
		always @(posedge clk) begin
		if (reset) begin
		add <= 0;
		end else begin
		add <= a + b;
		end
		end
		
		assign sum = add;
		
		endmodule

Better

Figure 11: An example of composition principle in LLM.

```
module matrix_multiplier (
input signed [15:0] A [0:3][0:3],
input signed [15:0] B [0:3][0:3],
output reg signed [31:0] C [0:3][0:3]);
integer i, j, k;
always @ (*) begin
for (i = 0; i < 4; i = i + 1) begin
    for (j = 0; j < 4; j = j + 1) begin
        C[i][j] = 0;
        for (k = 0; k < 4; k = k + 1) begin
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        end
    end
end  end
endmodule
```

Power: **105.2380** **Area** **179679.995125**

```
module matrix_multiplier (
input signed [15:0] A [0:3][0:3],
input signed [15:0] B [0:3][0:3],
input clk,
output reg signed [31:0] C [0:3][0:3]);
reg signed [31:0] M [0:3][0:3];
always@(posedge clk) begin
for (i = 0; i < 4; i = i + 1) begin
    for (j = 0; j < 4; j = j + 1) begin
    M[i][j] <= A[i][0] * B[0][j] + A[i][1] * B[1][j] + A[i][2] * B[2][j] + A[i][3]
* B[3][j];
……
end  end
endmodule
```

Power: **122.3770** **Area** **198511.994667**

```
module matrix_multiplier (
input [15:0] A [0:3][0:3],
input [15:0] B [0:3][0:3],
output [31:0] C [0:3][0:3]);
wire [31:0] P [0:15];
assign P[0] = {A[0][0], A[0][1]} * {B[0][0], B[1][0]};
……
assign C[0][0] = P[0][15:8] + P[8][15:8] + P[4][15:8] + P[12][15:8];
endmodule
```

Power: **13.1363** **Area** **21743.999433**

Figure 12: Matrix Multiply Verilog Program list, where power is measured in milliwatts (mW), area is measured in micrometers squared (μm^2)

A.2 A Motivation Case study on PPA-aware output manager

As discussed in the method part, PPA are crucial metrics in chip logic design. However, GPT lacks comparable PPA configurations. In GPT model, the reward model only provides general ranking. Therefore, ChipGPT proposes a cost manager to optimize PPA following the GPT model.

This stage uses design tools to test the PPA for each program, appending the results to the list of program until the enumeration search is complete to select the target version. For example, in Fig. 12, GPT generates different versions, they have different PPAs. Therefore, the cost manager tries to use EDA tools to output their PPAs and search for the optimal one.

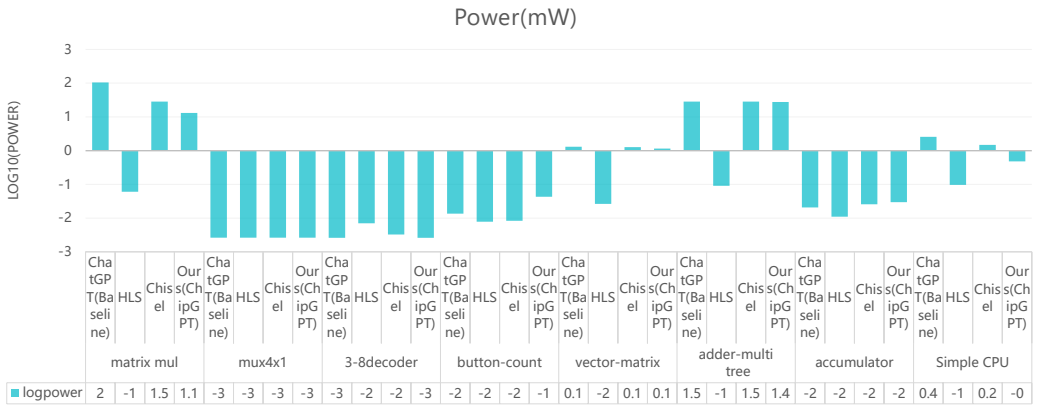

Figure 13: Power Comparison under Area Optimization Setting. To facilitate presentation, y-axis is normalized with `log10()`.

B Evaluation Detail

B.1 Evaluation Results Detail

The benchmarks include several typical hardware structures and algorithm accelerator implementations, as shown in Table 3. GPT exhibits different performance on different workloads. Therefore, we classify them into three parts based on complexity: 1) Composition (CM): The most complex architecture, requiring module composition. 2) Complex single module (CSM): More complex individual modules. 3) Simple single module (SSM): The simplest modules, with identical program lists and PPA.

Table 3: workload detail

Workload	Type	Brief Illustration
matmul	CSM	Multiplying two matrices 4x4 with 16 bits
mux	SSM	A 4x1 Multiplexer
3-8decoder	SSM	Select one of the eight lines in a 3-to-8 decoder
button	CSM	Counts the number of button presses
vecmat	CSM	Vector matrix multiply with 4-bits element
addmulti tree	CM	Add multiply tree with 8-bit operand
accumulator	CSM	Sum an array of 8-bits elements
simple CPU	CM	A simple CPU implementation

Fig. 14 compares the number of code lines generated for the same designs by our approach, high-level synthesis (HLS) tools, and the Chisel hardware design framework. On average, ChipGPT decreased the code volume by 9.25 times compared to HLS and 5.32 times compared to Chisel. This substantial reduction clearly demonstrates the enhanced programmability enabled by natural language techniques.

Fig. 15 shows less than 10 lines of code needing correction for all workloads. Simple modules like the 4x1 multiplexer required no feedback, demonstrating autonomous generation. For complex modules and integrated accelerators up to 100 lines of code, only minor corrections were needed.

Figure 13 presents a comparison of power consumption with area optimization enabled. The y-axis values are normalized with a log10() transformation to improve readability. This experiment utilizes the same settings as Figure 5 for consistent comparison.

B.2 Case Study

To demonstrate our method using open source EDA tools, we present an experimental case study of a matrix multiplier design targeting the Skywater130 130nm process design kit (PDK). The experiments

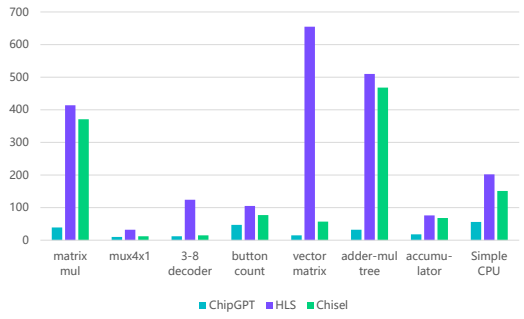

Figure 14: Generated code line number

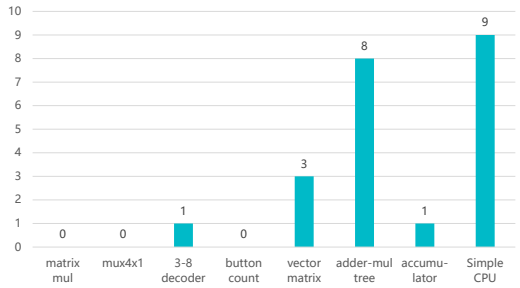

Figure 15: Human correction effort measured in line number of code

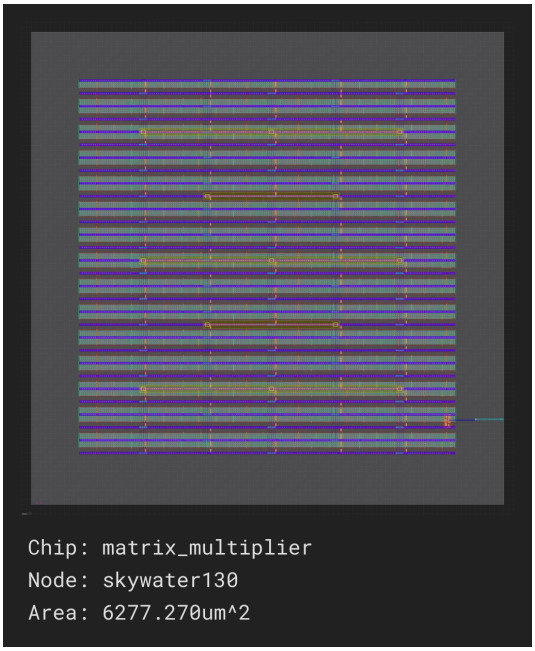

Figure 16: A HDL synthesis layout(Matrix Multiplier1).

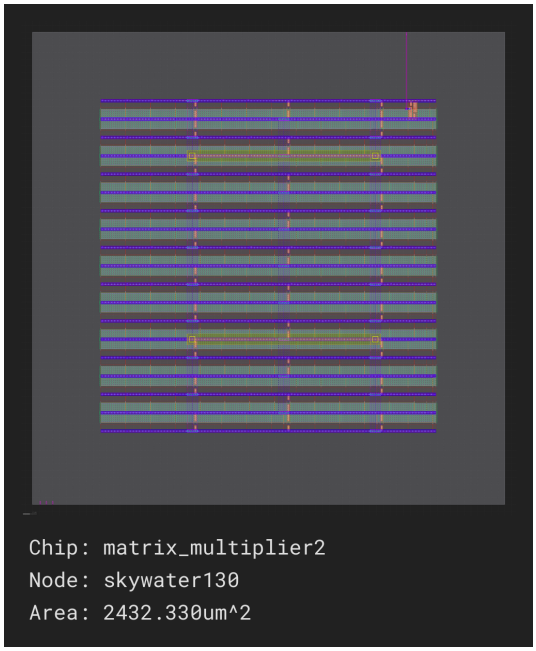

Figure 17: A HDL synthesis layout(Matrix Multiplier2).

utilize SiliconCompiler, an open source EDA flow comprising OpenLane and KLayout. Fig. 16 and Fig. 17 show layouts for two matrix multiplier versions. Tab. 4 and Tab. 5 detail synthesis results comparing the two versions. These multiplier designs showcase how different hardware versions generated by GPT exhibit tradeoffs in area and other key parameters.

Table 4: A HDL synthesis result(Matrix Multiplier1) in the GPT generating program list.

-	units	import0	syn0	floorplan0	physyn0	place0	cts0	route0	dfm0	export0	export1
errors		0	0	0	0	0	0	0	0	0	0
warnings		1	236	21	4	5	4	4	4	1	4
drvs		—	—	0	0	0	0	0	0	—	0
unconstrained		—	—	1	1	1	1	1	1	—	1
cellarea	um^2	—	11.261	100.096	100.096	100.096	100.096	100.096	100.096	—	100.096
totalarea	um^2	—	—	6277.27	6277.27	6277.27	6277.27	6277.27	6277.27	—	6277.27
utilization	%	—	—	1.595	1.595	1.595	1.595	1.595	1.595	—	1.595
peakpower	mw	—	—	0	0	0	0	0	0	—	0
leakagepower	mw	—	—	0	0	0	0	0	0	—	0
holdpaths		—	—	0	0	0	0	0	0	—	0
setuppaths		—	—	0	0	0	0	0	0	—	0
macros		—	—	0	0	0	0	0	0	—	0
cells		—	2	78	78	78	78	78	78	—	78
registers		—	—	0	0	0	0	0	0	—	0
buffers		—	—	0	0	0	0	0	0	—	0
pins		—	—	3	3	3	3	3	3	—	3
nets		—	—	5	5	5	5	5	5	—	5
vias		—	—	—	—	—	—	3	—	—	—
wirelength	um	—	—	—	—	—	—	11	—	—	—
memory	B	25.109M	39.027M	132.910M	128.781M	254.195M	131.031M	561.984M	140.871M	520.039M	139.230M
exetime		0.58	7.75	2.509	2.52	2.14	2.66	2.56	1.669	6.69	1.72
tasktime		0.971	11.638	4.933	4.386	6.809	7.762	6.458	6.503	8.574	4.802

Table 5: A HDL synthesis result(Matrix Multiplier2) in the GPT generating program list.

-	units	import0	syn0	floorplan0	physyn0	place0	cts0	route0	dfm0	export0	export1
errors		0	0	0	0	0	0	0	0	0	0
warnings		1	237	21	4	5	4	4	4	1	4
drvs		—	—	0	0	0	0	0	0	—	0
unconstrained		—	—	1	1	1	1	1	1	—	1
cellarea	um^2	—	11.261	41.29	41.29	41.29	41.29	41.29	41.29	—	41.29
totalarea	um^2	—	—	2432.33	2432.33	2432.33	2432.33	2432.33	2432.33	—	2432.33
utilization	%	—	—	1.698	1.698	1.698	1.698	1.698	1.698	—	1.698
peakpower	mw	—	—	0	0	0	0	0	0	—	0
leakagepower	mw	—	—	0	0	0	0	0	0	—	0
holdpaths		—	—	0	0	0	0	0	0	—	0
setuppaths		—	—	0	0	0	0	0	0	—	0
macros		—	—	0	0	0	0	0	0	—	0
cells		—	2	31	31	31	31	31	31	—	31
registers		—	—	0	0	0	0	0	0	—	0
buffers		—	—	0	0	0	0	0	0	—	0
pins		—	—	4	4	4	4	4	4	—	4
nets		—	—	6	6	6	6	6	6	—	6
vias		—	—	—	—	—	—	2	—	—	—
wirelength	um	—	—	—	—	—	—	11	—	—	—
memory	B	26.363M	39.355M	133.680M	129.047M	251.859M	131.805M	553.316M	138.086M	517.164M	138.348M
exetime		0.32	8.539	1.77	2	1.389	1.149	1.94	2.859	4.419	1.389
tasktime		0.713	12.111	2.872	4.528	4.111	3.031	5.962	8.839	5.959	4.492