Natural language is not enough: Benchmarking multi-modal generative AI for Verilog generation

Anonymous Author(s)

ABSTRACT

Natural language interfaces have demonstrated their potential in automating Verilog generation from high-level specifications using large language models, which receives much attention. However, this paper reveals that for spatially complex hardware structures, visual representations provide additional context critical for design intent, which may outperform only natural language input. Building upon this insight, our paper presents a benchmark of multi-modal generative models for Verilog synthesis from visual-linguistic inputs, encompassing both single modules and complex modules. Additionally, we introduce a visual and natural language Verilog query language to facilitate efficient and user-friendly multi-modal queries. To evaluate the performance of the proposed multi-modal hardware generative AI in Verilog generation tasks, we compare it with a popular method that relies solely on natural language. Our results demonstrate a significant accuracy improvement in the multi-modal generated Verilog compared to queries based solely on natural language. We hope to reveal a new field in the large hardware design model era, thereby fostering a more diversified and efficacious approach to hardware design.

1 INTRODUCTION

Recent advancements in the utilization of large language models for the generation of Verilog have elicited considerable interest within the domain of electronic design automation (EDA) [5, 6, 14, 16]. These models are emerging as a seminal technique for the automated generation of both Verilog code and EDA scripts [4, 14, 17], presenting a paradigm shift in the field. The overarching aim of this research stream is to assist hardware developers in expeditiously designing complex hardware systems without much knowledge of the specific hardware[8].

While natural language interfaces have shown promise for basic code generation tasks, hardware design poses unique challenges that cannot be addressed through language alone. Major hurdles remain in generating complex designs involving state machines and interactions between multiple modules as shown in Fig. 3 and Fig. 5. A key drawback is the difficulty in accurately conveying the spatial connections and complex nested structures of hardware, due to the linear nature of language. Our experiments indicate that a multi-modal generative model, which combines language with visual block diagrams and structural information, has the potential to overcome these limitations.

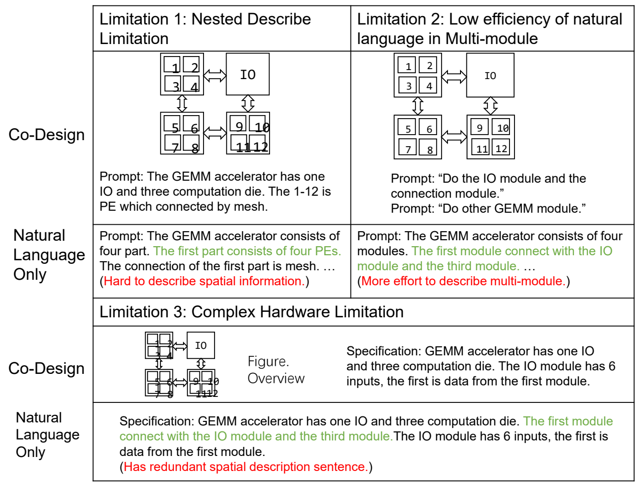
While our experiments demonstrate the promise of multi-modal generative models for hardware design tasks, there are three challenges to address. A key open issue is the **lack of standardized benchmarks** for evaluating and comparing different multi-modal architectures. Without common benchmarks and metrics, it is difficult to systematically analyze model performances or drive the field forward. Additionally, benchmarking multi-modal models currently requires large labeled datasets and lengthy training procedures. This prohibits fast iteration and exploration of new model innovations. There is a need for **benchmarking schemes that can quickly evaluate models** using smaller datasets or limited supervision. Finally, the concept of multi-modality itself needs to be rigorously defined for hardware tasks. Important questions remain around what modalities are most effective, how they should be jointly represented and processed, and how performance is quantitatively assessed on multi-faceted generation problems. To address these challenges, we propose a Verilog multi-modal benchmark, Verilog multi-modal model query language and a specification of the benchmark.

Table 1: Related Work Comparasion. Natural Language and Image Co-generation is a zero-shot method with a higher level.

Works	Task	Input	Output	Method
Vivado	Block Design	Diagram Template	Verilog	Rule-based
Qaw[2]	Notation Programming	Quantum Notation	QASM	Object Detecting
ChipNeMo[14]	Verilog Generation	Natural Language	Verilog	LLM
Thakur et al.[16]	Verilog Generation	Natural Language	Verilog	LLM
Ours	Verilog Generation	Img. + Natural Lang.	Verilog	multi-modal

In this paper, we elucidate the limitations inherent in relying exclusively on natural language-based Verilog generation methodologies and solve the challenges above. We introduce a novel co-generation query language to facilitate benchmarking, specifically designed to formalize the automatic co-generation process inherent in our proposed method. Furthermore, to facilitate the comparison and evaluation of potential works in this new field, we propose a benchmark with benchmark specification for future investigations in the field of vision-language Verilog models. The evaluation is conducted on GPT4V, GPT4, LLaVA and LLaMA, which are the typical fundamental models of natural language Verilog generation[4]. The results show that the multi-modal large language model for Verilog generation has a significant improvement over typical. In summary, we hope to reveal a new field in the large hardware design model era, thereby fostering a more diversified and efficacious approach to hardware design. The contributions are listed below:

- We reveal that for spatially complex hardware structures, visual representations introduced in this paper provide additional context critical for design intent, which may outperform only natural language input in automated hardware generation based on generative AI.
- We propose an innovative new multi-modal model query language to formalize vision-language descriptions, which lowers the token cost and enhances the generated code quality.


Limitation 1: Nested Describe Limitation
Limitation 2: Low efficiency of natural language in Multi-module
Co-Design
IO
1 2 3 4 5 6 7 8 9 10 11 12
Natural Language Only
Limitation 3: Complex Hardware Limitation
Co-Design
Figure. Overview
Natural Language Only

Prompt: The GEMM accelerator has one IO and three computation die. The 1-12 is PE which connected by mesh.

Prompt: "Do the IO module and the connection module."
Prompt: "Do other GEMM module."

Prompt: The GEMM accelerator consists of four part. The first part consists of four PEs. The connection of the first part is mesh. …
(Hard to describe spatial information.)

Prompt: The GEMM accelerator consists of four modules. The first module connect with the IO module and the third module. …
(More effort to describe multi-module.)

Specification: GEMM accelerator has one IO and three computation die. The IO module has 6 inputs, the first is data from the first module.

Specification: GEMM accelerator has one IO and three computation die. The first module connect with the IO module and the third module.The IO module has 6 inputs, the first is data from the first module.
(Has redundant spatial description sentence.)

Figure 1: Limitation of Natural Language Hardware Design in three aspects. The co-design row presents solutions using the multi-modal method. The "natural language only" row is the output of using the text-only language model. The green sentence means the redundant sentence, which can be represented by vision.

- We propose a hierarchy benchmark from simple to complex for evaluating multi-modal large model performance in Verilog generation, which will be open-source[1].
- We systematically evaluate the multi-modal large models with our benchmark from syntax, functionality and next-token success rate. With the vision representation, the testbench passing rate improves from 46.88% to 71.81% for GPT4 series, and from 13.41% to 25.88% for LLaMA series.

2 BACKGROUND & MOTIVATION

2.1 The limitations in current LLM-based RTL generation roadmap.

Researchers have explored LLMs in Verilog generation in Tab. 1. Benchmarking in [12, 13, 16] demonstrates these models' capacity to alleviate the burden on hardware designers. Progress has been made on fine-tuning for code completion [17], general RTL generation[4] and EDA tool script generation [14]. Beyond single-sentence models like GPT-3, conversational LLMs have also been proven capable of RTL-level repair[4, 7, 18, 20], quantum[11], in-memory computing[19], testing[1, 10, 15], and AI domain specific processors' design[8] fields. However, LLM-based RTL generation still has the following limitations:

Limitation 1: Natural language alone cannot properly describe the nested spatial structure of hardware, due to the inherent limitations of linguistic representations. The complex spatial relationships between components in computer hardware cannot be fully captured using only natural language descriptions. For example, consider the spatial accelerator as shown in Fig. 1. It contains the IO module, PE and controllers. The spatial interconnections and multi-layered component structures quickly overwhelm natural language. Terms like "below", "embedded", "adjacent", and "surrounding" cannot adequately express the nested 2D spatial structure. While natural language can convey some basic hardware organization, its linear, imprecise linguistic representations lack the relational expressiveness needed to model spatial information at multiple levels. More formal diagrams are required to depict the structural complexity fully.

Limitation 2: Inefficiency of Language-Based Descriptions in Multi-Module Hardware Design. In the field of complex hardware design, particularly when considering systems comprising up to n submodules, traditional language-based methodologies for describing interconnections reveal significant inefficiencies, which $n = 12$ is shown in Fig. 1. Such hardware systems can be conceptualized as a multigraph $G = (V, E)$, where each submodule is represented as a node, and the interconnections between these submodules are denoted as edges. The potential complexity of this system escalates to an order of $O(n^2)$ in terms of interconnections, consequently necessitating a description complexity of $O(n^2)$ tokens. This scenario underscores a fundamental limitation of single-modal language models, which become remarkably less efficient compared to multi-modal models in handling the intricate details of hardware design interconnectivity.

Limitation 3: Risk of Misalignment in Complex Hardware Designs Using Language Models. Misalignment constitutes a critical challenge in the application of large language models for complex hardware design. This issue is predominantly observed when descriptions are ambiguous or incomplete. A notable instance of this issue is the port connection, as shown in Fig. 1. When relying exclusively on textual input to delineate the functional aspects of a design, there is a pronounced tendency for the model to erroneously route signals to incorrect ports or even to unintended submodules, diverging from the anticipated configuration. This misalignment risk is substantially mitigated through the integration of visual inputs. By employing a multi-modal approach that includes vision, the model receives explicit, visual cues regarding the correct alignment of signals to their respective ports. This enhancement not only clarifies the intended design but significantly reduces the likelihood of alignment errors, thereby improving the accuracy and reliability of the model's output in hardware design applications.

2.2 Visual and natural language Hardware Co-Design Case Study

This section uses multi-module hardware and state machine as two cases to show how multi-modal generation excels over methods relying solely on natural language in the context of structural hardware.

Multi-Module Hardware Generation. Multi-module hardware is common in hardware design. We conduct a case study to explore whether multi-modal large models can generate multi-module structures better than text-only large language models. For example, as shown in Fig. 2, a chain of PEs has a spatial structure. We choose OpenAI GPT4 Vision as the multi-modal large model to synthesize the Verilog code. The result in Fig. 3 reveals that in a multi-module environment, the multi-modal large model can capture more hardware information compared with the method

[1]https://anonymous.4open.science/r/iccad2024685/

using only text input, which results in wrong output in only large language model mode.

State Machine Generation. State machines are often represented using a diagram. As shown in Fig. 4, we use the image to show the state machine that detects "10011". If the input match "10011", then the circuit output 1. We choose OpenAI GPT4 Vision as the multi-modal large model. The Verilog code produced was evaluated using the criterion of pass@5. Our results indicated that while the multi-modal model (GPT4V) successfully generated a version of the code that passed testbench evaluations, the text-only model version fell short in some test cases. In Fig. 5, the red text shows the wrong state transition. This example underscores the enhanced capability of vision information in extracting structural details from images, thereby significantly improving code generation accuracy.

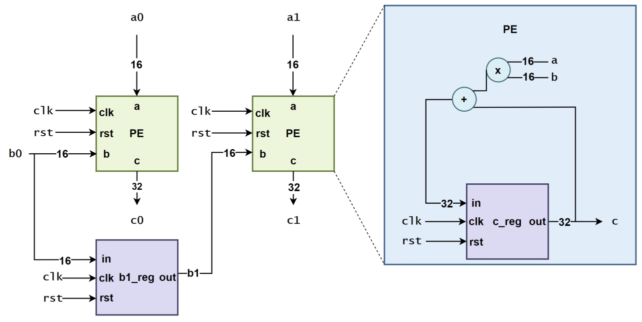
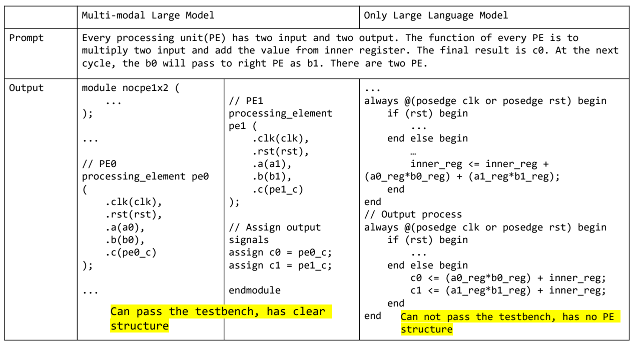


Figure 2: A case study to show a multiply and accumulate PE.

2.3 Motivation: Do we really need a new benchmark?

Previous benchmarks such as RTLLLM and Verigen still has the following challenges in multi-modal generation environments.

Challenge I: Incomplete modal Input without normalized form. Translating chip specification to RTL code needs to understand the hardware diagram, which further normalized the form of the diagram representation. Instead of directly drawing a chip diagram, we propose a normalized diagram definition and query language to match the specification requirement.



	Multi-modal Large Model		Only Large Language Model
Prompt	Every processing unit(PE) has two input and two output. The function of every PE is to multiply two input and add the value from inner register. The final result is c0. At the next cycle, the b0 will pass to right PE as b1. There are two PE.		
Output			

```
module nocpe1x2 (
    ...
);

...

// PE0
processing_element pe0
(
    .clk(clk),
    .rst(rst),
    .a(a0),
    .b(b0),
    .c(pe0_c)
);

...
    Can pass the testbench, has clear
    structure
```

```
// PE1
processing_element
pe1 (
    .clk(clk),
    .rst(rst),
    .a(a1),
    .b(b1),
    .c(pe1_c)
);

// Assign output
signals
assign c0 = pe0_c;
assign c1 = pe1_c;

endmodule
```

```
...
always @(posedge clk or posedge rst) begin
    if (rst) begin
        ...
    end else begin
        …
        inner_reg <= inner_reg +
(a0_reg*b0_reg) + (a1_reg*b1_reg);
    end
end
// Output process
always @(posedge clk or posedge rst) begin
    if (rst) begin
        ...
    end else begin
        c0 <= (a0_reg*b0_reg) + inner_reg;
        c1 <= (a1_reg*b1_reg) + inner_reg;
    end
end   Can not pass the testbench, has no PE
      structure
```

Figure 3: A case study to show that for multi-module hardware, the multi-modal model has a significant improvement over the language model.

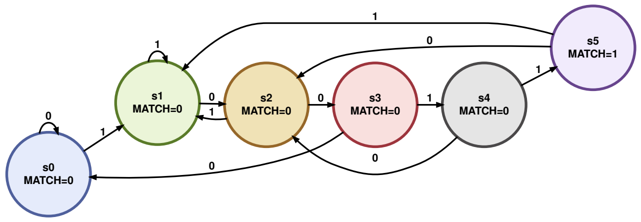
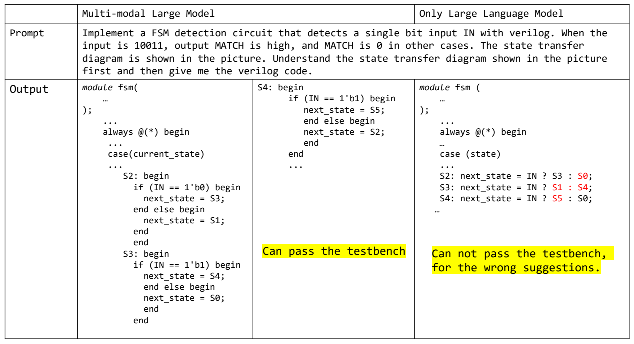


Figure 4: A case study of State machine to show visual and natural language co-design outperforms language model. This image shows a state machine to detect whether the input is 10011.



	Multi-modal Large Model		Only Large Language Model
Prompt	Implement a FSM detection circuit that detects a single bit input IN with verilog. When the input is 10011, output MATCH is high, and MATCH is 0 in other cases. The state transfer diagram is shown in the picture. Understand the state transfer diagram shown in the picture first and then give me the verilog code.		
Output			

```
module fsm(
    …
);
    ...
    always @(*) begin
     ...
     case(current_state)
     ...
        S2: begin
          if (IN == 1'b0) begin
            next_state = S3;
          end else begin
            next_state = S1;
          end
          end
        S3: begin
          if (IN == 1'b1) begin
            next_state = S4;
            end else begin
            next_state = S0;
            end
          end
```

```
S4: begin
      if (IN == 1'b1) begin
         next_state = S5;
         end else begin
         next_state = S2;
         end
      end
      ...

 Can pass the testbench
```

```
module fsm (
    …
);
    ...
    always @(*) begin
    …
    case (state)
    ...
    S2: next_state = IN ? S3 : S0;
    S3: next_state = IN ? S1 : S4;
    S4: next_state = IN ? S5 : S0;
   …

  Can not pass the testbench,
  for the wrong suggestions.
```

Figure 5: Compare the capability of generating Verilog between multi-modal and text-only language model. To follow a fair standard, we generate a Verilog description and test the generated code using pass@5.

Challenge II: No Complexity Classification for Multi-Modal Input Prompt. Human programmers often provide multi-modal model prompts with different complexity in different description levels from simple to complex. However, current benchmarks only explore the different prompts in multi-modal settings, which can not provide comprehensive benchmark results.

Challenge III: Coarse-grain output metrics. Program completion plays an important role in EDA editor scenario. Most LLM uses the predict-the-next-token paradigm, which computes the loss of the next token. However, current benchmarks(*e.g.* RTLLLM[13], Verigen[17]) directly provide holistic program output to compute their pass rate such as RTLLLM. Our benchmark splits the coarse-grain outputs into several fine-grain outputs, which provide a loss metric with the next N tokens.

3 VISUAL AND NATURAL LANGUAGE HARDWARE CO-DESIGN WORKFLOW

3.1 Visual and Natural Language Co-Generation Workflow

For an end-to-end Verilog generation flow, we split them as two parts as shown in Fig. 6. The frontend accepts natural language

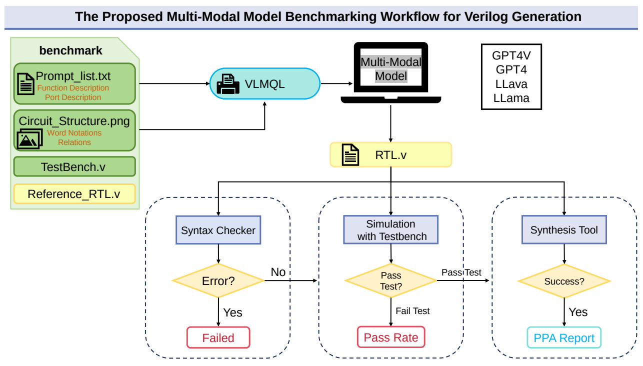


Figure 6: The overall end-to-end visual and natural language hardware co-generation workflow. The grey block is the proposed benchmark.

and image(*i.e.* hardware structure) for Verilog generation, which outputs verilog file. The backend accepts the Verilog file and outputs the PPA reports, GDSII layout and function analysis report. To do a democratic hardware design, we chose siliconcompiler and openlane as the backend, which are open-source EDA tools for ASIC synthesis.

3.2 The Formulation of the Circuit Structure in the Benchmark

To overcome the limitations of only natural language hardware generation and challenge I in Sec. 2, we illustrate the visual form and natural language form co-design knowledge as below. The visual form is used to generate top level connections and relations between each module. However, the detailed function can not be presented properly in the image/diagram only with visual information. The detailed information is collected in a natural language format. Several core concepts must be remembered in natural language designs below.

Word Notations in the visual hardware graph: As shown in Fig. 2, Word Notations are the connections between visual and natural language hardware representation, which are the words on the image. Users declare the name on each block to illustrate it clearly in a natural language format. For example, for a five stage pipeline, the execution stage is annotated on the image and the natural language part can use "In the execution stage, the processor [DO SOMETHING]". Without word notations, large models only see the block, which can not detect the right description. Therefore, word notations are the interface between image design and natural language design.

- Module name word notations: The diagram name provides a high-level name and description of the overall module or component represented visually. This allows connecting the diagram to natural language that references this module by name.
- Wire width word notations: Lines with variable width can represent connections of differing bandwidth or bitwidth. Annotating the width numerically clarifies the intent.
- Wire Function word notations: Text labels on wires indicate whether they carry data signals, control signals, clocks, etc. This clarifies the role of connections in the natural language.
- Block Function word notations: Major functional blocks are annotated with descriptive labels indicating their roles (e.g. ALU, multiplexer, register file). This allows precise natural language references.
- Ports word notations: Port labels designate the names and types of external interfaces. Natural language can then refer to interacting with specific ports.

Definition: ***Relations*** *in the visual hardware graph.* As shown in Fig. 3, relations are the arrows and connections on the image, which is the core part of a hardware structure image. The relations in hardware structure have the source and sink node, which represents input and output. According to relation theory in computer science, there are three possible relations on the image, one-to-one relations, one-to-many relations and many-to-many relations.

- Single arrow relations: These represent singular connections between two components, like an output from one gate going to the input of another gate. They show a direct 1-to-1 relationship(*e.g.* wire).
- 1-to Many and Many-to-many arrow relations: These represent bus connections where multiple wires are bundled together. They model relationships where a group of outputs connects to a group of inputs, showing 1-to-many (*e.g.* bus) or many-to-many (*e.g.* crossbar) connectivity.

Definition: ***Module Function Description*** *in natural language.* Module function description is the function in natural language format. For example, the sentence "a 3-8 decoder accepts a 3-bit number and output a 8-bit number where only one bit is one." is a typical module function description in natural language. The "3-8 decoder" is the word notations in the corresponding image.

Definition: ***Module Port Description*** *in natural language.* Module port description is the port width and function description in natural language format. For example, the sentence "the input of 3-8 decoder is a 3-bit width wire named `innum`." is a typical port description in the natural language description.

3.3 The proposed Multi-modal Hardware Design Benchmark

To select a benchmark scientifically, We form the benchmark selection as an optimization problem as Equ. 1, where B denotes the benchmark and $Data$ denotes world wide data. The target of the benchmark is to be consistent with the worldwide data, which means the distribution of the two datasets should be minimized.

$$\min D(B||Data) \tag{1}$$

To implement the Sec. 3.2, we prevent hardware image outliers without annotations by template coding, which converts raw data to a normalized form as shown in Sec. 3.2. Specifically, template coding accepts a hardware module image without annotations and then we add the annotations according to text description. In addition, we use four-element pairs as the basic form of the benchmark, which are text modal(*i.e.* `Prompt_list.txt`), diagram modal(*i.e.* `Circuit_Structure.png`), testbench for pass rate evaluation(*i.e.* `TestBench.v`), reference correct RTL Verilog program(*i.e.* `reference.v`).

Benchmarking Output Complexity via Hierarchy difficulty Workload. We classify the workloads into three levels from low complexity to high complexity to better benchmark the LLM performance on different design complexities. Concretely, the arithmetic level represents basic operators for numbers such as add, multiply and division. The logic level represents common controllers in hardware design such as `edge_detect` and `pulse_detect`. The advanced level represents such as the basic unit in CPU(*i.e.* 3 stage pipeline) and core unit in matrix multiplication(*i.e.* 4×4 GEMM).

Benchmarking Input Complexity via Multi-level Prompting. To solve the challenge II in Sec. 2, We add the multi-level prompt to the proposed multi-modal benchmark for the pre-trained multi-modal model, which can benchmark the design understanding ability. For example, a well-trained LLM can generate the correct RTL program from both a very simple prompt and a complex prompt. Specifically, the prompt complexity varies from low to high. The low level prompt provides basically only diagram without detailed natural language illustration. The middle-level prompt provides a diagram with simple natural language to describe the core function. The high-level prompt provides detailed information of the hardware such as the register and clock edge information.

Fine-grain Output Measurement. Motivated by challenge III, we propose a fine-grain output splitting method to provide more comprehensive evaluations, which uses token-by-token metrics. Assuming the output of the inference result is $\{tok'_0, tok'_1, \ldots, tok'_N\}$ and the reference benchmark is $\{tok_0, tok_1, \ldots, tok_N\}$, then the fine-grain loss on m is defined in Equ. 2, where the success is one when the two tokens are the same.

$$success^m = \Sigma_{i=0}^{m} 1_{tok'_i tok_i} \tag{2}$$

3.4 Facilitate Flexible Benchmarking using Query Language

To implement the above visual and natural language co-design formulation, we propose a large model query language. Using query language inference in a large language model can control LLM output and input, which can reduce inference time and increase accuracy via compact network request and template prompt[3]. We propose a query language for visual and natural language Verilog generation to automatically separate them which is called VLMQL(Verilog Large Model Query Language).

VLMQL Structure. VLMQL is designed to represent visual and natural language hardware co-design formulation in Sec. 3.2. It is a kind of controllable prompt engineering. The core of VLMQL is a function in Python as shown in Fig. 7. The return value is used as the input of large vision-language model. VLMQL consists of three parts: declarations for the visual and natural language input, agent flow description for manipulating EDA tools and constraint statements for prompt scheduling.

Mode Declaration: Three-level hardware representation. The hardware visual description is splited into three level, which are gate-level, algorithm level, function block level from concrete to abstract.

```
@vlmql.function
def pipeline_5stage():
    # mode declaration
    vlmql.set_mode("func_block")

    # large model declaration
    vlmql.lvm("gpt4v")
    vlmql.llm("gpt4v")
    vlmql.image_path("5stage.png")
    # function declaration
    vlmql.function(
        "The image show a 5 stage pipeline to add two number",
        "the execution stage add two number from register r1 and
register r0",
        "the write back stage write the result to register r2",
        ...
    )
    # EDA agent flow description
    vlmql.eda_flow("the area should large than (1000,1000)") # choose
siliconcompiler
    vlmql.eda_tool("siliconcompiler")
    # constraint statement
    vlmql.module_constraint("execute stage")
    return vlmql.run()
```

Figure 7: A case for VLMQL. The image shows a 5 stage pipeline, where we need to synthesis one of the stage in it.

Gate-level represents the image is logic gate. Algorithm level represents the basic blocks in the image that are the large model familiar algorithm, such as add, multiply block. For the large and customized design, VLMQL uses function block level to represent them, which suggests these blocks need to be illustrated. These levels are declared as the parameter as `vlmql.set_mode("func_block")`, which is the prompt of **Module function description**. This declaration set up a soft constraint for the users to draw the diagram, which then compiled to natural language prompt, such as "The basic block of input image is logic gate.".

Large Model Declaration: The parameter of Model and their Input. Large model declarations take the following task as input and then output prompt or choose the function inputs.

- Model Selection: To choose the target vision model and large language model, VLMQL uses `vlmql.lvm("[vision model]")` and `vlmql.llm("[language model]")` to choose the target model.
- Image Selection: To choose the input hardware structure image, VLMQL uses `vlmql.image_path("[image path]")`, which can obtain images and feed them into the large language model.

Function Declaration: The separation description of the module. The function declarations are a sequence of prompts. Every line in the function represents a prompt following specification in Sec. 3.2. The first part is the function description of every verilog module. The second part is the port description of every verilog module.

EDA Agent Flow Description. Besides using large model to generate verilog file, generating EDA script is also an important field[9]. VLMQL facilitates describing an end-to-end visual and natural language co-design workflow.

- Tool selection: This parts select the EDA tool as `vlmql.eda_tool ( "[EDATool]" )`. However, Large model may lack of EDA tool script knowledge. Therefore, VLMQL compiler provides the EDA tool script example as the prompt to the LLM, which is also called in-context learning.

Table 2: **Benchmark for multi-modal model Verilog generation.**

Type	Name	Description	Code Line
Advance	1x4 systolic	A 1x4 systolic array with 1 row and 4 columns of processing elements for high throughput parallel processing.	18
	1x2 systolic	A smaller 1x2 systolic array with 1 row and 2 columns of processing elements.	22
	2x2 systolic	A 2x2 systolic array with 2 rows and 2 columns of processing elements.	11
	4x4 gemm	A 4x4 matrix multiplication unit arranged as a 2D systolic array for high throughput parallel matrix multiplication	23
	5 stage pipeline	A 5 stage instruction pipeline to break down instruction processing into fetch, decode, execute, memory access, and writeback stages.	66
	3 stage pipeline	A simpler 3 stage instruction pipeline with fetch, execute and writeback stages.	39
	MAC PE	A MAC (multiply-accumulate) processing element for performing vector matrix multiplications.	12
	2state_fsm	A state machine module to transition between 2 states based on inputs.	23
	3state_fsm	A state machine module to transition between 3 states based on inputs.	36
	4state_fsm	A state machine module to transition between 4 states based on inputs.	38
	5state_fsm	A state machine module to transition between 5 states based on inputs.	38
	6state_fsm	A state machine module that detect the string 10011	77
Logic	Johnson_Counter	A 64-bit Johnson counter (torsional ring counter)	12
	alu	An ALU for 32bit MIPS-ISA CPU	99
	edge_detect	A module for edge detection, there is a slowly changing 1 bit signal a.	35
	freq_div	A frequency divider that the input clock frequency of 100MHz signal,	46
	mux	A multi-bit MUX synchronizer	40
	parallel2serial	A module for parallel-to-serial conversion	33
	pulse_detect	A module for edge detection, there is a slowly changing 1 bit signal data_in.	29
	right_shifter	A right shifter	12
Arithmetic	accu	A module to achieve serial input data accumulation output.	51
	adder_16bit	A module of a 16-bit full adder.	102
	add_16bit_csa	A 16-bit carry select adder.	114
	adder_32bit	A module of a carry lookahead 32 bit adder based on CLAs.	152
	adder_64_bit	A module of a ripple 64 bit adder.	171
	adder_8bit	A module of an 8 bit adder in gate level.	20
	div_16bit	A 16-bit divider module, dividend is 16-bit and divider is 16-bit.	33
	multi_booth	An 8bit booth-4 multiplier	71
	multi_pipe_4bit	The design of 4bit unsigned number pipeline multiplier.	33
	multipipe_8bit	The design of unsigned 8bit multiplier based on pipelining processing.	76

- EDA Flow elaboration: This parts uses natural language to tell LLM the detailed information. For example, silicon-compiler requires the input file name and the floorplan area. This serves as prompt in LLM, which uses primitive `vlmql.eda_flow("[natural language description]")`. The "natural language description" can be "change the total area to 210*210".

Constraint Statement: Constrain the input and output for low cost generation. For some scenes, users do not need to obtain all of the verilog of the module. For example, users may input a 5-stage cpu. But users only want to get the verilog code of execution stage. However, if following the typical workflow, the additional module except execution may cost many tokens. Therefore, constraints are necessary to release this additional tokens. These constraints are finally compiled to prompts. For example, For verilog generation task, users can use `vlmql.module_constraint("[modulename]")` to declare the module they want to generate, while other modules won't be generated. This will serves as a prompt.

4 EVALUATION

4.1 Evaluation Setup

Our study systematically explores the efficacy of multi-modal language models in generating Verilog code by establishing a comprehensive benchmark that spans a spectrum of complexity across three categories: arithmetic, digital circuit, and advanced hardware designs. The benchmark is meticulously structured to assess the incremental benefits of multi-modal inputs as we progress from simple to more complex cases. To evaluate the performance, each model within the GPT-4 and LLaMA series is tasked with generating Verilog code for each category five times, facilitating a robust and iterative testing methodology.

The evaluation focuses on three critical aspects: syntax correctness, functional accuracy, and PPA (Performance, Power, and Area) optimization, ensuring a holistic assessment of the Verilog code generated. The benchmark leverages a "pass@5" metric, which examines whether the correct Verilog code is produced within five attempts, providing insight into both the precision and reliability of the models.

4.2 Evaluation Result

The evaluation of multi-modal language models in the generation of Verilog code has yielded illuminating results, particularly in terms of syntax and functionality correctness.

Syntax Correctness. The introduction of vision representations has led to a notable enhancement in syntax correctness for both the GPT-4 and LLaMA series of models. The GPT-4 series shows a decrease in syntax error rates from 20.63% to 8.13%. This improvement highlights the models' increase proficiency in understanding and applying the syntactic rules of Verilog with the aid of visual context. The LLaMA series also demonstrated a reduction in syntax errors, though less pronounced, from 72.50% to 66.8%. These results suggest that while all models benefit from multi-modal inputs, those with more advanced architectures, like the GPT-4 series, exhibit a greater propensity for minimizing syntax errors.

Table 3: Syntax represents the number of Verilog code generated by LLM with syntax errors under pass@5. Function represents the testbench pass rate of the best-performing Verilog code under pass@5. V represents the Vision modal and T represents the natural language text modal.

benchmark		GPT4-V (V+T)		GPT4 (T)		GPT4-V (V)		LLaVa (V+T)		Llama (T)		LLaVa(V)	
		syntax	function	syntax	function	syntax	function	syntax	function	syntax	function	syntax	function
Logic	Johnson_Counter	0	98%	0	100%	0	98%	5	0%	0	97%	1	0%
	alu	0	100%	0	0%	0	0%	5	0%	5	0%	0	0%
	edge_detect	0	100%	0	100%	0	100%	0	98%	0	98%	0	0%
	freq_div	0	100%	0	0%	0	0%	4	96%	5	0%	0	0%
	mux	0	100%	1	100%	0	10%	1	0%	5	0%	0	100%
	parallel2serial	0	0%	0	0%	0	0%	0	100%	5	0%	0	0%
	pulse_detect	0	100%	0	0%	5	0%	0	0%	5	0%	3	0%
	right_shifter	0	100%	0	100%	0	100%	0	100%	0	100%	0	0%
	serial2parallel	0	100%	0	100%	2	0%	5	0%	0	0%	0	0%
	width_8to16	0	100%	0	100%	0	100%	1	0%	0	34%	0	0%
Arithmetic	accu	0	100%	0	100%	3	0%	5	0%	5	0%	3	0%
	adder_16bit	0	100%	0	100%	3	100%	5	0%	5	0%	8	0%
	add_16bit_csa	0	100%	0	100%	5	0%	2	100%	1	100%	2	0%
	adder_32bit	3	0%	0	0%	0	0%	5	0%	5	0%	0	0%
	adder_64_bit	0	0%	2	0%	0	0%	5	0%	5	0%	4	0%
	adder_8bit	0	100%	0	100%	2	5%	0	0%	3	0%	0	0%
	div_16bit	1	0%	5	0%	0	0%	5	0%	5	0%	2	0%
	multi_booth	0	50%	1	0%	0	0%	2	0%	5	0%	0	0%
	multi_pipe_4bit	0	50%	3	100%	2	50%	4	100%	1	0%	5	0%
	multipipe_8bit	0	100%	1	0%	0	0%	5	0%	5	0%	0	0%
Advanced	1x2nocpe	0	100%	3	0%	3	0%	0	34%	4	0%	0	0%
	1x4systolic	0	100%	0	100%	1	30%	5	0%	5	0%	0	0%
	2x2systolic	0	100%	2	0%	5	0%	5	0%	5	0%	2	0%
	3stagepipe	1	0%	5	0%	5	0%	5	0%	5	0%	0	0%
	4x4spatialacc	3	0%	5	0%	5	0%	5	0%	5	0%	0	0%
	5stagepipe	5	0%	5	0%	5	0%	5	0%	5	0%	0	0%
	fsm	0	100%	0	0%	1	0%	5	0%	3	0%	10	0%
	macpe	0	100%	0	100%	3	20%	0	100%	2	0%	2	0%
	statemachine	0	100%	0	0%	5	0%	5	0%	3	0%	0	0%
	5state_fsm	0	100%	0	0%	3	0%	5	0%	5	0%	3	0%
	3state_fsm	0	100%	0	100%	1	100%	5	0%	4	0%	0	0%
	4state_fsm	0	0%	0	100%	1	100%	3	100%	5	0%	0	0%
Success Rate			71.81%		46.88%		33.90%		25.88%		13.41%		3.13%

Functionality Correctness. Moving on to functionality, the testbench passing rate serves as a key indicator of functional correctness. For the GPT-4 series, the rate of successfully generating functionally accurate Verilog code improved remarkably from 46.88% to 71.81% with the integration of visual inputs. This leap underscores the models' enhanced capability to not only generate syntactically correct code but also functionally viable Verilog that aligns with the intended hardware design. The LLaMA series, conversely, experienced a modest improvement, with pass rates increasing from 13.41% to 25.88%. While the starting point for functionality correctness was lower for the LLaMA series, the addition of multi-modal data still contributed to a meaningful improvement in performance.

The results collectively indicate that multi-modality significantly uplifts the language models' ability to generate Verilog code that is both syntactically and functionally more accurate. These advancements attest to the substantial potential of incorporating visual data to augment the efficacy of language models in complex tasks like hardware design, ensuring that the code generated not only looks correct but also works correctly in practical applications.

4.3 Sensitivity Study

Multi-Level multi-modal Prompt. We benchmark the success rate on multi-level prompts from simple to complex as shown in Tab. 5. The results show that with the increase of the prompt information, the success rate has a further increase in most cases. Specifically, the prompts change from simple to detailed, while the success rate ranges from 40.6% to 71.8% in GPT4-V, from 9.38% to 25.88% in LLaVa. Therefore, the proposed multi-level prompt can distinguish the LLM-generating difference.

Fine-grain output Prompt. In addition, we measure the output verilog program with the output metric as shown in Tab. 6, which compares the next token predicting success rate, reflecting the LLM's program complete ability. The results show that the natural-only mode is weaker than the natural language and image co-design mode. Specifically, compared to the natural-language-only mode, the average success rate of the co-design mode improves from 63.64% to 71.72% in GPT series, from 20.20% to 28.28% in LLaMa series. These results support our speculation that the co-design mode is better than the natural-language-only mode in the token prediction task.

State Number Changes in FSM. To further explore the LLM sensitivity to design complexity, we analyze several control modules with state number changes (*i.e.* push button LED) in Tab. 4 with GPT4-V as the base model, where the transition in the table denotes the state transition success rate, the state denotes the state register declaration correctness, the output denotes the signal output correctness on every state. The results show that with the state number increase from 2 to 9, the success rate decreases from 100% to 0%. This shows that current LLM-generated hardware can not well capture the long distance information.

Table 4: The state number change success rate on push button case. This shows the generating success rate in pass@5.

State Number	2	3	4	5	6	7	8	9
Transition	100%	100%	80%	60%	60%	20%	0%	0%
State	100%	100%	100%	100%	100%	100%	100%	100%
Output	100%	100%	100%	100%	100%	100%	80%	80%

Table 5: The success rate changes from simple to complex. S,M,C represent simple, medium and complex prompts respectively.

Benchmark (function)		LLaVa			GPT4-V		
		S	M	C	S	M	C
Logic	Johnson_Counter	0%	0%	0%	0%	0%	98%
	alu	0%	0%	0%	20%	0%	100%
	edge_detect	0%	40%	98%	100%	100%	100%
	freq_div	0%	0%	96%	100%	80%	100%
	mux	0%	0%	0%	80%	100%	100%
	parallel2serial	0%	0%	100%	0%	0%	0%
	pulse_detect	0%	0%	0%	100%	100%	100%
	right_shifter	0%	100%	100%	100%	100%	100%
	serial2parallel	0%	0%	0%	0%	100%	100%
	width_8to16	0%	0%	0%	100%	100%	100%
Arithmetic	accu	0%	0%	0%	0%	0%	100%
	adder_16bit	0%	40%	0%	100%	80%	100%
	add_16bit_csa	0%	0%	100%	0%	100%	100%
	adder_32bit	0%	0%	0%	0%	0%	0%
	adder_64_bit	0%	0%	0%	0%	0%	0%
	adder_8bit	100%	100%	0%	100%	100%	100%
	div_16bit	0%	0%	0%	0%	0%	0%
	multi_booth	0%	0%	0%	0%	0%	50%
	multi_pipe_4bit	0%	0%	100%	0%	100%	50%
	multipipe_8bit	0%	0%	0%	0%	0%	100%
Advanced	1x2nocpe	60%	40%	34%	100%	100%	100%
	1x4systolic	40%	60%	0%	60%	20%	100%
	2x2systolic	0%	0%	0%	20%	0%	100%
	3stagepipe	0%	0%	0%	0%	0%	0%
	4x4spatialacc	0%	0%	0%	0%	0%	0%
	5stagepipe	0%	0%	0%	0%	0%	0%
	fsm	20%	0%	0%	0%	100%	60%
	macpe	0%	80%	100%	100%	100%	100%
	statemachine	0%	20%	0%	60%	100%	100%
	5state_fsm	20%	0%	0%	100%	100%	60%
	3state_fsm	0%	40%	0%	20%	100%	100%
	4state_fsm	60%	0%	100%	40%	100%	80%
Success Rate		9.38%	16.25%	25.88%	40.63%	55.63%	71.81%

4.4 Ablation Study

To further reveal the difference between the natural-language model-based hardware generation and multi-modal model-based hardware generation, we compared the image-only mode, text-only mode, and mix input mode. As shown in Tab. 3, the results show that from the success rate perspective, the mix input mode is the best and the image-only mode is the worst. Specifically, the average success rate of image-only mode is 33.90%, and the average success rate of mix input mode is 71.91% in GPT4-V. Therefore, we recommend LLM for RTL generation with multi-modal model rather than only use vision mode.

Table 6: The next token predicts success rate.

Benchmark	GPT4-V Series		Llava Series	
	Co-Design	NL-Only	Co-Design	NL-Only
1x2nocpe	100%	100%	67%	100%
1x4systolic	100%	100%	67%	33%
2state_fsm	100%	67%	67%	33%
2x2systolic	100%	100%	67%	67%
3stagepipe	67%	67%	0%	33%
3state_fsm	0%	100%	33%	0%
4state_fsm	67%	33%	0%	33%
4x4spatialacc	67%	67%	33%	0%
5stagepipe	33%	33%	0%	0%
5state_fsm	100%	100%	0%	0%
fsm	0%	0%	0%	0%
macpe	100%	67%	67%	33%
accu	100%	100%	33%	33%
adder_16bit	100%	33%	0%	33%
adder_16bit_csa	100%	100%	0%	0%
adder_32bit	67%	67%	0%	0%
adder_64bit	100%	33%	0%	0%
adder_8bit	100%	100%	67%	0%
div_16bit	33%	33%	0%	33%
multi_16bit	0%	33%	0%	0%
multi_booth	100%	67%	67%	33%
multi_pipe_4bit	0%	0%	0%	33%
multi_pipe_8bit	67%	67%	33%	0%
alu	100%	0%	0%	0%
edge_detect	67%	67%	33%	0%
freq_div	100%	100%	33%	33%
Johnson_Counter	67%	0%	33%	33%
mux	100%	100%	33%	33%
parallel2serial	100%	100%	100%	33%
pulse_detect	100%	100%	0%	0%
right_shifter	67%	67%	33%	0%
serial2parallel	33%	67%	33%	0%
width_8to16	33%	33%	33%	33%
Average	71.72%	63.64%	28.28%	20.20%

5 CONCLUSION

Our research underscores the significant potential of multi-modal large language models in Verilog generation. By integrating visual representations with natural language processing, we have achieved notable improvements in the generation of complex hardware designs. Our novel query language framework enhances code quality and efficiency, and the comprehensive benchmarks we established demonstrate a substantial increase in model performance. This approach not only advances hardware design methodologies but also provides a possible way for future research in generative AI applications within this field, marking a significant step towards more intuitive and efficient hardware design processes.

REFERENCES

[1] Ahmad, B., Thakur, S., Tan, B., Karri, R., and Pearce, H. Fixing hardware security bugs with large language models. *arXiv preprint arXiv:2302.01215* (2023).

[2] Arawjo, I., DeArmas, A., Roberts, M., Basu, S., and Parikh, T. Notational programming for notebook environments: A case study with quantum circuits. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2022), UIST '22, Association for Computing Machinery.

[3] Beurer-Kellner, L., Fischer, M., and Vechev, M. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages 7*, PLDI (2023), 1946–1969.

[4] Blocklove, J., Garg, S., Karri, R., and Pearce, H. Chip-chat: Challenges and opportunities in conversational hardware design. *arXiv preprint arXiv:2305.13243* (2023).

[5] Chang, K., Wang, Y., Ren, H., Wang, M., Liang, S., Han, Y., Li, H., and Li, X. Chipgpt: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019* (2023).

[6] Chen, L., Chen, Y., Chu, Z., Fang, W., Ho, T.-Y., Huang, Y., Khan, S., Li, M., Li, X., Liang, Y., et al. The dawn of ai-native eda: Promises and challenges of large circuit models. *arXiv preprint arXiv:2403.07257* (2024).

[7] Fang, W., Li, M., Li, M., Yan, Z., Liu, S., Zhang, H., and Xie, Z. Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms. *arXiv preprint arXiv:2402.00386* (2024).

[8] Fu, Y., Zhang, Y., Yu, Z., Li, S., Ye, Z., Li, C., Wan, C., and Lin, Y. Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models. In *ICCAD* (2023).

[9] He, Z., Wu, H., Zhang, X., Yao, X., Zheng, S., Zheng, H., and Yu, B. Chateda: A large language model powered autonomous agent for eda. *arXiv preprint arXiv:2308.10204* (2023).

[10] Kande, R., Pearce, H., Tan, B., Dolan-Gavitt, B., Thakur, S., Karri, R., and Rajendran, J. Llm-assisted generation of hardware assertions. *arXiv preprint arXiv:2306.14027* (2023).

[11] Liang, Z., Cheng, J., Yang, R., Ren, H., Song, Z., Wu, D., Qian, X., Li, T., and Shi, Y. Unleashing the potential of llms for quantum computing: A study in quantum architecture design. *arXiv preprint arXiv:2307.08191* (2023).

[12] Liu, M., Pinckney, N., Khailany, B., and Ren, H. VerilogEval: evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2023).

[13] Lu, Y., Liu, S., Zhang, Q., and Xie, Z. Rtllm: An open-source benchmark for design rtl generation with large language model, 2023.

[14] Mingjie Liu§, T. E. Chipnemo: Domain-adapted llms for chip design. *arXiv preprint arXiv:2307.09288* (2023).

[15] Orenes-Vera, M., Martonosi, M., and Wentzlaff, D. From rtl to sva: Llm-assisted generation of formal verification testbenches, 2023.

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

[17] Thakur, S., Ahmad, B., Pearce, H., Tan, B., Dolan-Gavitt, B., Karri, R., and Garg, S. Verigen: A large language model for verilog code generation. *arXiv preprint arXiv:2308.00708* (2023).

[18] Tsai, Y., Liu, M., and Ren, H. Rtlfixer: Automatically fixing rtl syntax errors with large language models. *arXiv preprint arXiv:2311.16543* (2023).

[19] Yan, Z., Qin, Y., Hu, X. S., and Shi, Y. On the viability of using llms for sw/hw co-design: An example in designing cim dnn accelerators. *arXiv preprint arXiv:2306.06923* (2023).

[20] Yao, X., Li, H., Chan, T. H., Xiao, W., Yuan, M., Huang, Y., Chen, L., and Yu, B. Hdldebugger: Streamlining hdl debugging with large language models. *arXiv preprint arXiv:2403.11671* (2024).