



# Domain-adaptive LLMs for Chip Design

*Tutorial 1: AI Assisted Hardware Design - Will AI Elevate or Replace Hardware Engineers?*

Hanxian Huang

Advisor: Prof. Jishen Zhao

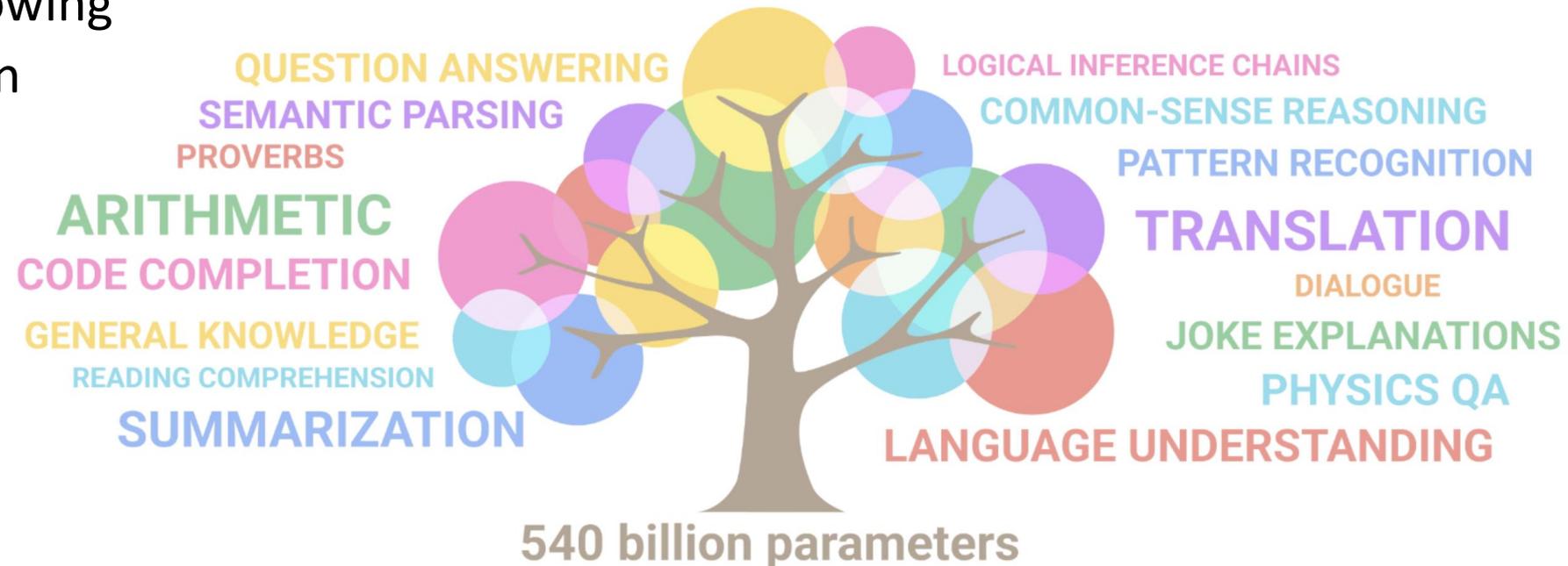
University of California San Diego

# Outline

- 1. Why domain-adaptive LLMs
- 2. Domain-adaptive LLM techniques
- 3. Domain-adaptive LLMs for Chip design tasks

# 1. Why domain-adaptive LLMs

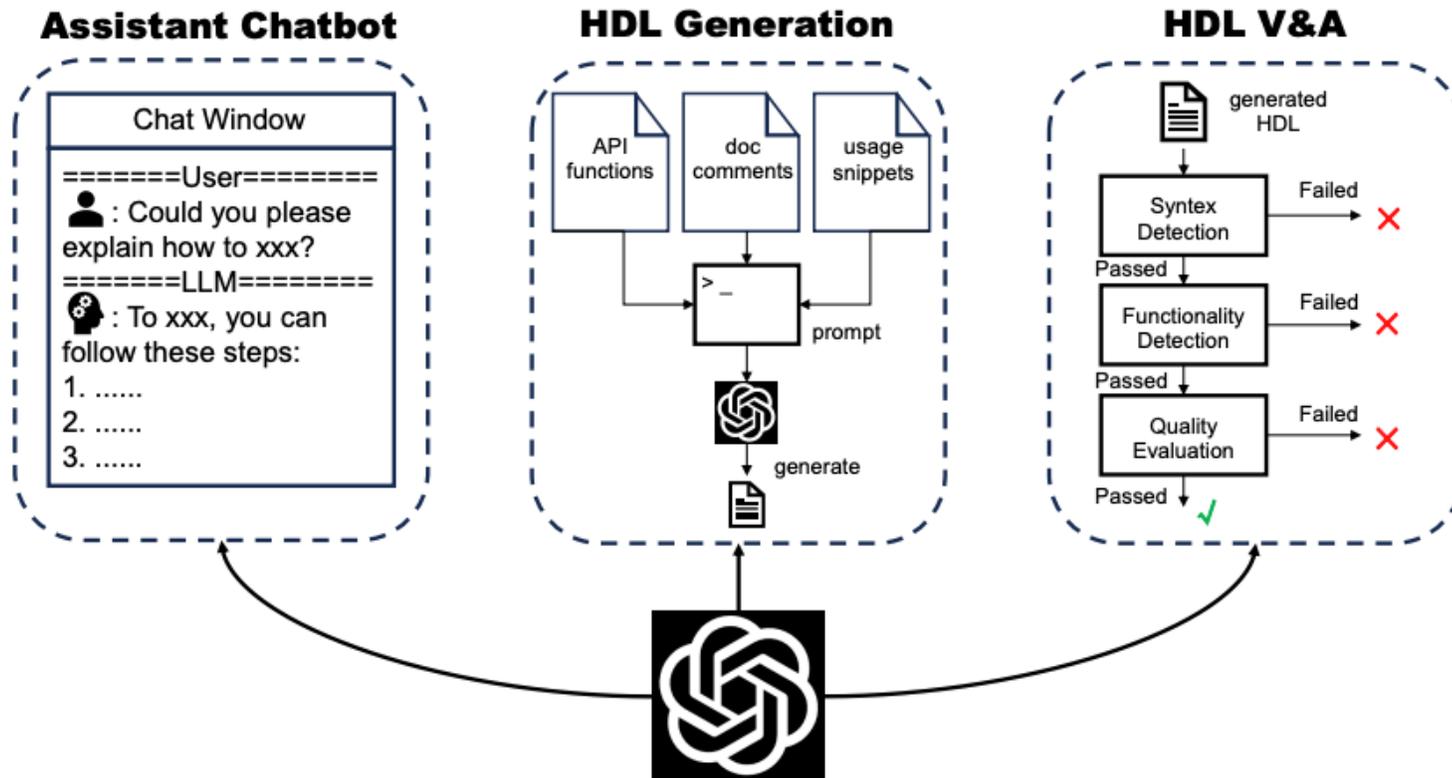
- (1). Leverage Extensive Knowledge and Capabilities of Pretrained Models
  - Natural language understanding and generation
  - Reasoning and problem solving
  - Instruction following
  - Code generation
  - Multi-modality



*As the scale of the model increases, the performance improves across tasks while also unlocking new capabilities.*

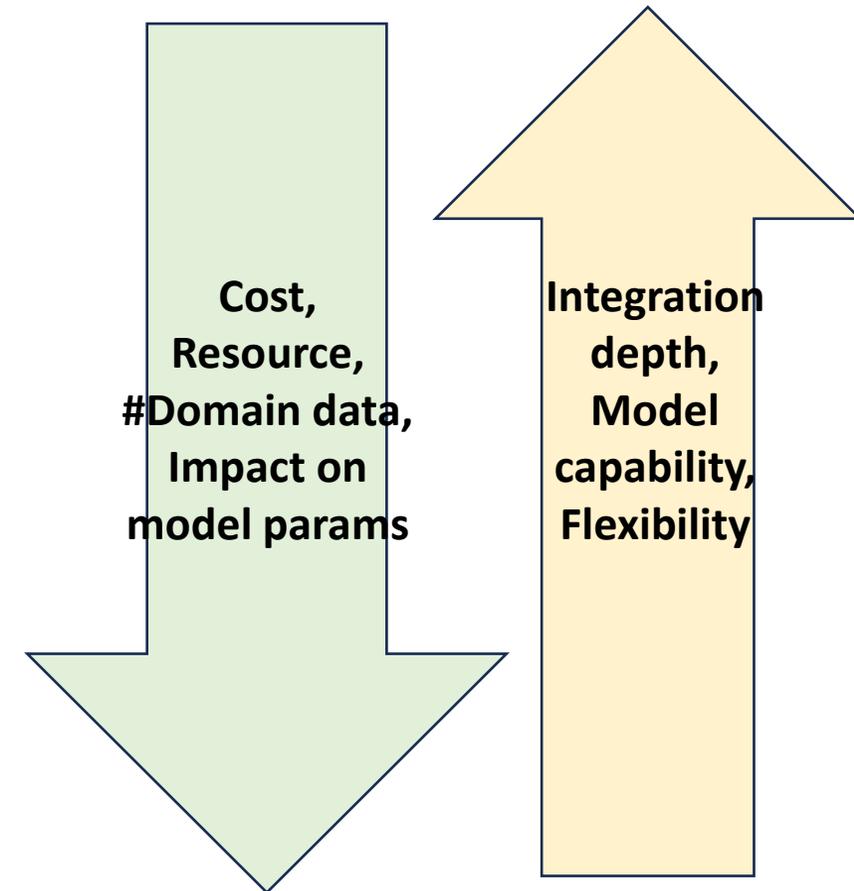
# 1. Why domain-adaptive LLMs

- (2). Incorporate Specialized Knowledge



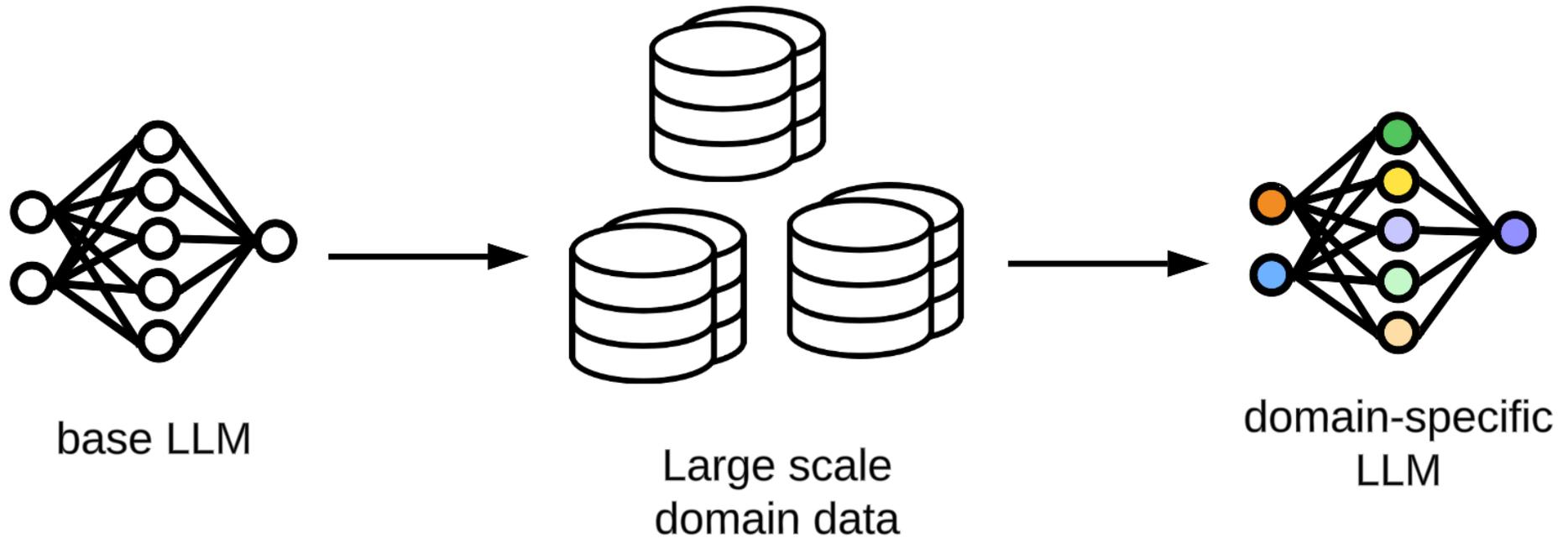
## 2. Domain-adaptive LLM techniques

- 1) In Model Training Stage
  - Deep integration of domain-specific knowledge
- 2) In Model Fine-tuning Stage
  - Enhancing model performance for specific tasks
- 3) In Model Inference Stage
  - Adapts the model dynamically in real-time, without altering the model's underlying parameters
  - Relies on external guidance or data retrieval to produce desired (domain-specific) outputs



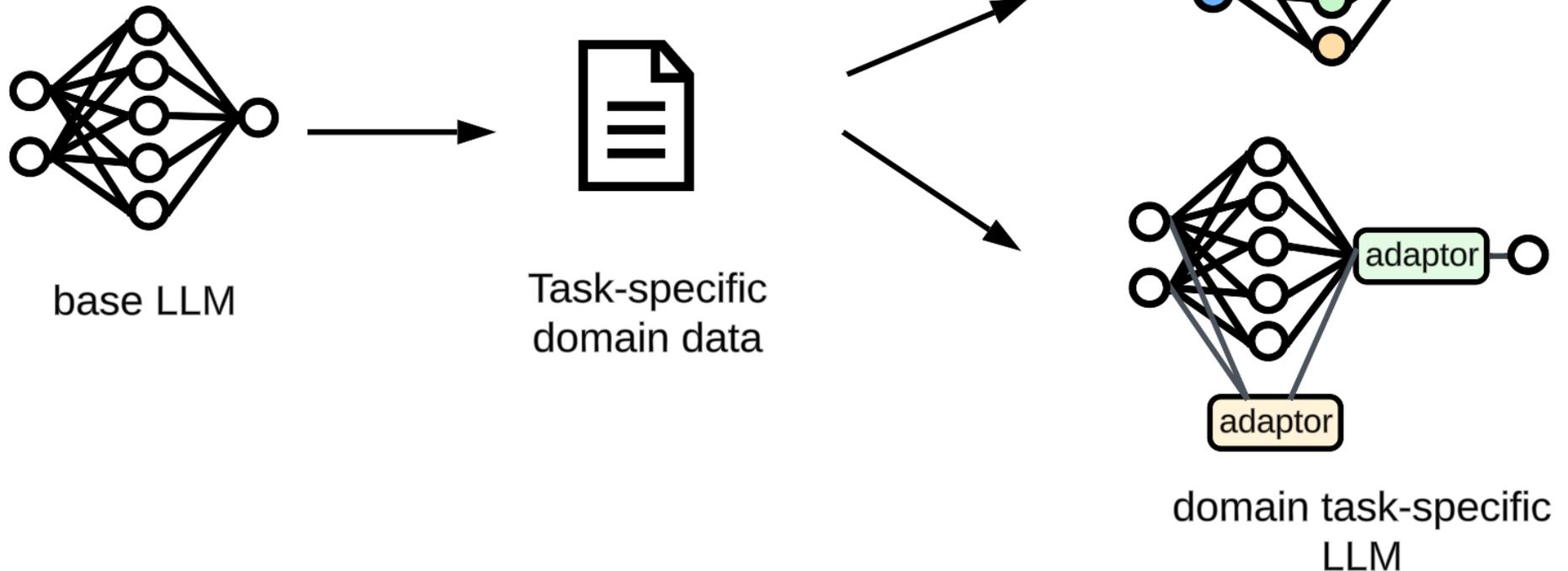
## 2. Domain-adaptive LLM techniques

- 1) Model Training Stage
  - Domain-Adaptive Pre-Training



## 2. Domain-adaptive LLM techniques

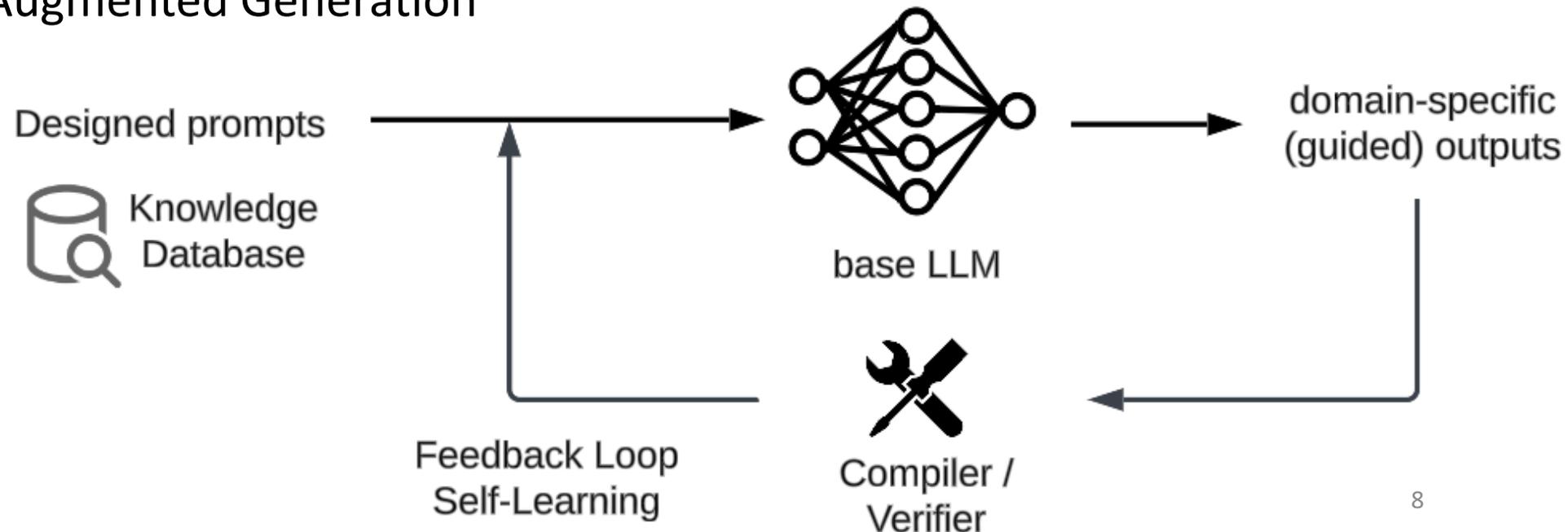
- 2) Model Fine-tuning Stage
  - Task-specific Fine-tuning
  - Adapter Modules



## 2. Domain-adaptive LLM techniques

- 3) Model Inference Stage

- Prompt engineering
  - In-context learning, Chain of thought
- Agent: Multi-turn feedback loop, Self-correction
- Retrieval-Augmented Generation



# 3. Domain-adaptive LLMs for Chip design tasks

- How to choose the domain-adaptive techniques? Consider:

- (1) The models you have access to

- The extent of access to foundation model gradients

- (2) Compute budget

- (3) Tasks and the amount of task-specific data available



```
graph LR; A[Pre-training] --> B[Fine-tuning]; B --> C[Inference]; style A stroke:#008000,stroke-width:2px; style B fill:#ffff00; style C fill:#ffcc99;
```

Pre-training

Fine-tuning

Inference

### 3. Domain-adaptive LLMs for Chip design tasks

- 1) Domain-adaptive pre-training (DAPT)

Pre-training

Fine-tuning

Inference

### 3. Domain-adaptive LLMs for Chip design tasks

- 1) Domain-adaptive pre-training (DAPT)
- E.g., ChipNemo
- To prepare for DAPT:
  - Available foundation model
    - e.g., Llama
  - **LARGE** domain dataset
    - hardware-related code
      - software, RTL, verification testbenches
    - natural language datasets
      - hardware specifications, documentation
  - Training resources

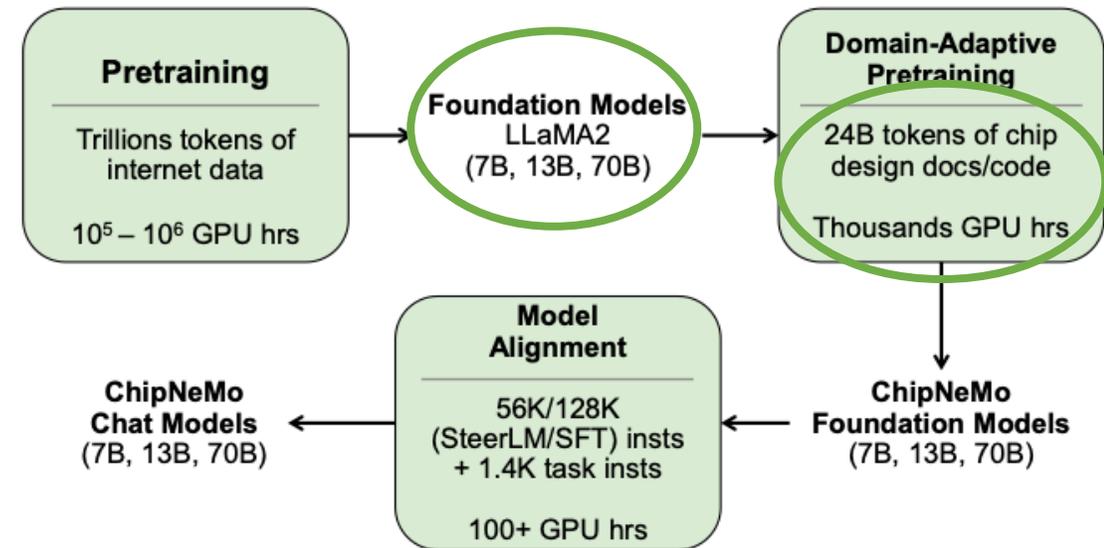


Figure 1: ChipNeMo Training Flow

### 3. Domain-adaptive LLMs for Chip design tasks

- 1) Domain-adaptive pre-training (DAPT)
- How to perform DAPT:
  - Tune a domain-adaptive tokenizer
    - General purpose corpus + in-domain corpus
  - Autoregressive language modeling objective
    - Can be similar to general foundation model pre-training
      - E.g., next-token prediction

### 3. Domain-adaptive LLMs for Chip design tasks

- 1) Domain-adaptive pre-training (DAPT)
- Outcome of DAPT:
  - Domain-adaptive foundation model (e.g., ChipNemo 7B, 13B, 70B)
  - Which can be tuned for various chip design tasks

Pre-training

Fine-tuning

Inference

### 3. Domain-adaptive LLMs for Chip design tasks

- 2) Task-specific fine-tuning (TSFT)

Pre-training

Fine-tuning

Inference

## 3. Domain-adaptive LLMs for Chip design tasks

- 2) Task-specific fine-tuning (TSFT)
  - Foundation model
  - Specific tasks
  - (Smaller) Domain task datasets
  - Benchmarks and Metrics

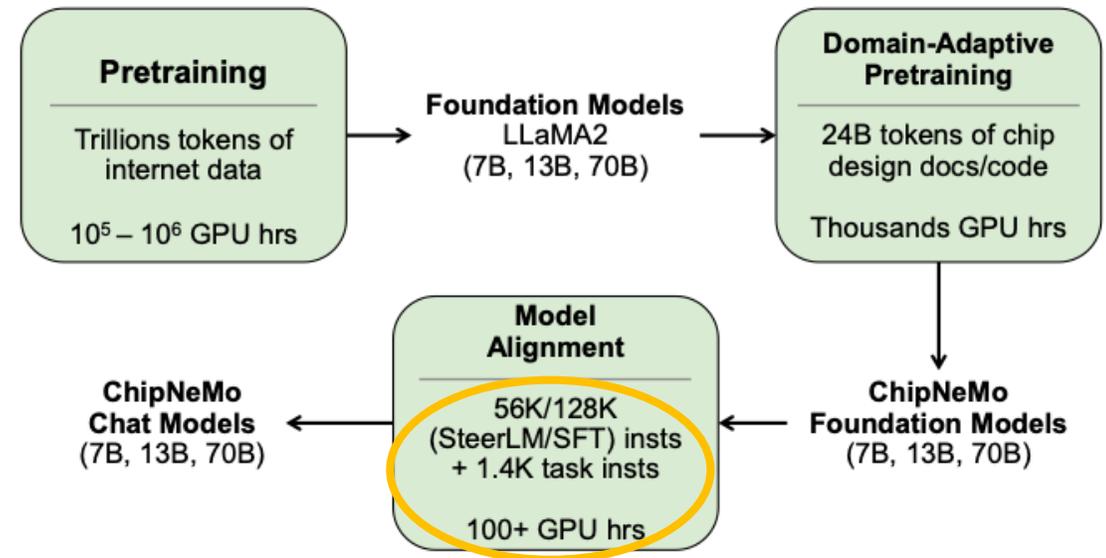


Figure 1: ChipNeMo Training Flow

### 3. Domain-adaptive LLMs for Chip design tasks

- 2) Task-specific fine-tuning (TSFT)
- Foundation models
  - Open-sourced general-purpose models
    - CodeGen, CodeGen2
    - CodeGeeX
    - Mistral
    - StarCoder, StarCoder2
    - Llama, Code-Llama
    - WizardCoder
    - ...
  - Your own DAPT models, e.g., ChipNemo

### 3. Domain-adaptive LLMs for Chip design tasks

- 2) Task-specific fine-tuning (TSFT)
- Downstream tasks, e.g.,
  - Verilog/EDA script generation
  - Verification
  - Bug detection and analysis
  - Layout optimization
  - Assistant Chatbot
  - ...

### 3. Domain-adaptive LLMs for Chip design tasks

- 2) Task-specific fine-tuning (TSFT)
- Domain task datasets
  - Sources:
    - Textbook
    - GitHub open-sourced code
    - Synthetic data (by human, by machine)
    - Proprietary data
    - Contest

# 3. Domain-adaptive LLMs for Chip design tasks

- 2) Task-specific fine-tuning (TSFT)
- Domain task datasets
  - Sources:
    - Textbook
    - GitHub open-sourced code
    - Synthetic data (by human, by machine)
    - Proprietary data
    - Contest
  - Labeling based on the target tasks:
    - RTL script generation: design description/domain instruction + reference code/testbench
    - HW security: Common Vulnerability Enumerations (CVEs), Common Weakness Enumerations (CWEs)
    - Debug/repair: correct vs wrong script pairs

**Problem Description:**

Given an 8-bit input vector [7:0], reverse its bit ordering.

```
module top_module (  
    input [7:0] in,  
    output [7:0] out  
);
```

**Canonical Solution:**

```
assign {out[0], out[1], out[2], out[3], out[4],  
    ↪ out[5], out[6], out[7]} = in;  
endmodule
```

An example of problem description + reference code

### 3. Domain-adaptive LLMs for Chip design tasks

- 2) Task-specific fine-tuning (TSFT)
- Benchmarks
  - RTLLM
  - VerilogEval-Human
  - VerilogEval-Machine

# 3. Domain-adaptive LLMs for Chip design tasks

- 2) Task-specific fine-tuning

- Benchmarks

- RTLLM**

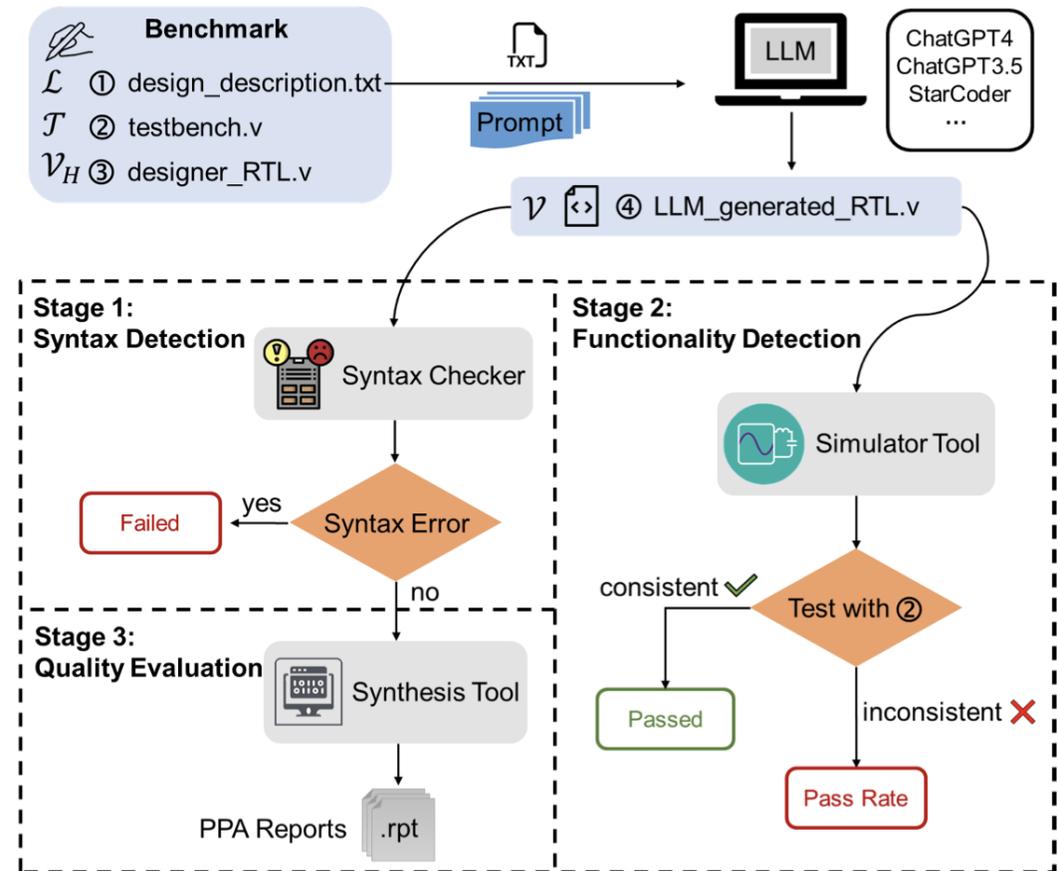
- 30 HDL design tasks
- Testbenchs
- Makefile

- Metrics**

- Syntax, Semantic
- Pass@k

$$pass@k := \mathbb{E}_{Problems} \left[ 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]$$

- PPA



# 3. Domain-adaptive LLMs for Chip design tasks

TABLE II: Benchmark Descriptions and Scales

	Design	Description	Lines of Code	Circuit Scale (Cells)
Arithmetic	accu	Accumulates 8-bit data and output after 4 inputs	64	195
	adder_8bit	An 8-bit adder	26	58
	adder_16bit	A 16-bit adder implemented with full adders	137	130
	adder_32bit	A 32-bit carry-lookahead adder	181	312
	adder_64bit	A 64-bit ripple carry adder based on 4-stage pipeline	197	1340
	multi_8bit	An 8-bit booth-4 multiplier	84	34
	multi_16bit	An 16-bit multiplier based on shifting and adding operation	65	817
	multi_pipe_4bit	A 4-bit unsigned number pipeline multiplier	43	120
	multi_pipe_8bit	An 8-bit unsigned number pipeline multiplier	92	578
	div_8bit	An 8-bit radix-2 divider	72	94
	div_16bit	A 16-bit divider based on subtraction operation	45	1855
	Logic	JC_counter	4-bit Johnson counter with specific cyclic state sequence	22
right_shifter		Right shifter with 8-bit delay	17	466
mux		Multi-bit mux synchronizer	46	19
counter_12		Counter module counts from 0 to 12	37	38
freq_div		Frequency divider for 100M input clock, outputs 50MHz, 10MHz, 1MHz	51	64
signal_generator		Signal generator produces square, sawtooth, and triangular waveforms	52	135
serial2parallel		1-bit serial input and output data after receiving 6 inputs	62	66
parallel2serial		Convert 4 input bits to 1 output bit	41	24
pulse_detect		Extract pulse signal from the fast clock and create a new one in the slow clock	38	6
edge_detect		Detect rising and falling edges of changing 1-bit signal	39	7
FSM		FSM detection circuit for specific input	77	24
width_8to16		First 8-bit data placed in higher 8-bits of the 16-bit output	50	117
traffic_light		Traffic light system with three colors and pedestrian button	106	117
calendar		Perpetual calendar with seconds, minutes, and hours	37	121
RAM		8x4 bits true dual-port RAM	50	1834
asyn_fifo		An asynchronous FIFO 16x8 bits	149	686
ALU		An ALU for 32bit MIPS-ISA CPU	111	2435
PE	A Multiplying Accumulator for 32bit integer	27	1439	
risc_cpu	Simplified RISC_CPU with clock generator, instruction register, accumulator, arithmetic logic unit, data controller, state controller, etc.	518	407	

- 2) Task-specific fine-tuning
- Benchmarks
  - **RTLLM**

Pre-training

Fine-tuning

Inference

# LLMs for Chip design tasks

TABLE II: Benchmark Descriptions and Scales

Design	Description	Lines of Code	Circuit Scale (Cells)	
arithmetic	accu	Accumulates 8-bit data and output after 4 inputs	64	195
	adder_8bit	An 8-bit adder	26	58
	adder_16bit	A 16-bit adder implemented with full adders	137	130
	adder_32bit	A 32-bit carry-lookahead adder	181	312
	adder_64bit	A 64-bit ripple carry adder based on 4-stage pipeline	197	1340
	multi_8bit	An 8-bit booth-4 multiplier	84	34
	multi_16bit	An 16-bit multiplier based on shifting and adding operation	65	817
	multi_pipe_4bit	A 4-bit unsigned number pipeline multiplier	43	120
	multi_pipe_8bit	An 8-bit unsigned number pipeline multiplier	92	578
	div_8bit	An 8-bit radix-2 divider	72	94
	div_16bit	A 16-bit divider based on subtraction operation	45	1855
	logic	JC_counter	4-bit Johnson counter with specific cyclic state sequence	22
right_shifter		Right shifter with 8-bit delay	17	466
mux		Multi-bit mux synchronizer	46	19
counter_12		Counter module counts from 0 to 12	37	38
freq_div		Frequency divider for 100M input clock, outputs 50MHz, 10MHz, 1MHz	51	64
signal_generator		Signal generator produces square, sawtooth, and triangular waveforms	52	135
serial2parallel		1-bit serial input and output data after receiving 6 inputs	62	66
parallel2serial		Convert 4 input bits to 1 output bit	41	24
pulse_detect		Extract pulse signal from the fast clock and create a new one in the slow clock	38	6
edge_detect		Detect rising and falling edges of changing 1-bit signal	39	7
FSM		FSM detection circuit for specific input	77	24
width_8to16		First 8-bit data placed in higher 8-bits of the 16-bit output	50	117
traffic_light		Traffic light system with three colors and pedestrian button	106	117
calendar		Perpetual calendar with seconds, minutes, and hours	37	121
RAM		8x4 bits true dual-port RAM	50	1834
asyn_fifo		An asynchronous FIFO 16x8 bits	149	686
ALU		An ALU for 32bit MIPS-ISA CPU	111	2435
PE	A Multiplying Accumulator for 32bit integer	27	1439	
risc_cpu	Simplified RISC_CPU with clock generator, instruction register, accumulator, arithmetic logic unit, data controller, state controller, etc.	518	407	

23

Implement the design of 4bit unsigned number pipeline multiplier. It consists of two levels of registers to store intermediate values and control the multiplication process.

## Module name:

multi\_pipe\_4bit

## Input ports:

clk: Clock signal used for synchronous operation.

rst\_n: Active-low reset signal. Defined as 0 for chip reset and 1 for reset signal inactive.

mul\_a: Input signal representing the multiplicand with a data width of "size" bits.

mul\_b: Input signal representing the multiplier with a data width of "size" bits.

## Output ports:

mul\_out: Product output signal with a data width of 2\*size bits.

## Parameter:

size = 4

### 3. Domain-adaptive LLMs for Chip design tasks

- 2) Task-specific fine-tuning
- Benchmarks
  - **VerilogEval**
    - Verilog-Machine:
      - problem description generated by GPT-3.5
    - Verilog-Human:
      - manual conversion from HDLBit website to problem description

Benchmark	Language	Train	Test
HumanEval [21]	Python	-	164
MBPP [22]	Python	374	500
APPS [23]	Python	5,000	5,000
<b>VerilogEval-machine</b>	Verilog	8,502	143
<b>VerilogEval-human</b>	Verilog	-	156

### 3. Domain-adaptive LLMs for Chip design tasks

- 2) Task-specific fine-tuning
- Benchmarks
  - **VerilogEval**
    - Verilog-Machine:
      - problem description generated
    - Verilog-Human:
      - manual conversion from HDLBit problem description

```
{"task_id": "count10",  
"simple_description": " This module increments the 4-bit output q  
on each rising edge of the clock, resetting it to 0 when reset is  
asserted or when q reaches 9.",  
"detail_description": " This Verilog module is a simple counter. It  
has one input, clk, one output, q, and one reset input. The clk input  
is used to trigger the counter. The reset input is used to reset the  
counter to 0. The output q is a 4-bit register that stores the current  
count value. The module has an always block that is triggered on  
the positive edge of the clk signal. Inside the always block, an if-else  
statement is used to determine the next value of q. If the reset  
input is asserted or the current value of q is equal to 9, then q is  
reset to 0. Otherwise, q is incremented by 1."}
```

```
{"task_id": "count1to10", "detail_description": "Make a decade  
counter that counts 1 through 10, inclusive. The reset input is active  
high synchronous, and should reset the counter to 1."}
```

### 3. Domain-adaptive LLMs for Chip design tasks

- 2) Task-specific fine-tuning
- An example: RTLCoder
  - Task: RTL code generation
  - Foundation model: Mistral-7B-v0.1 and DeepSeek-Coder-6.7b
  - Data collection: 27K instruction-code pairs

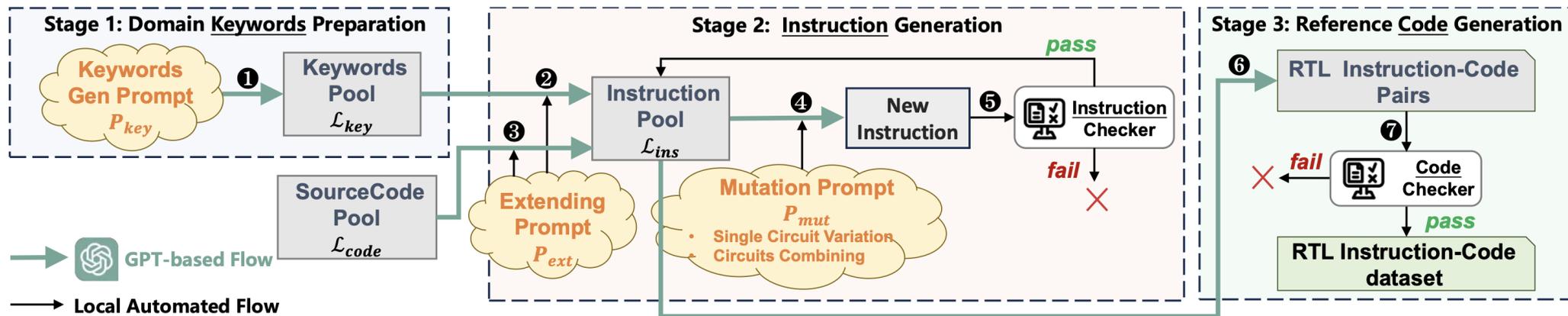
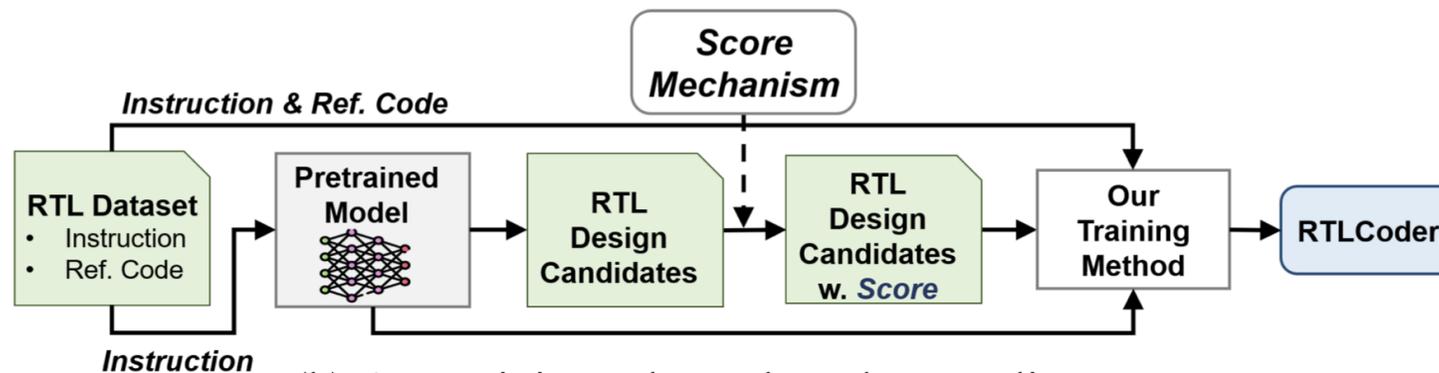


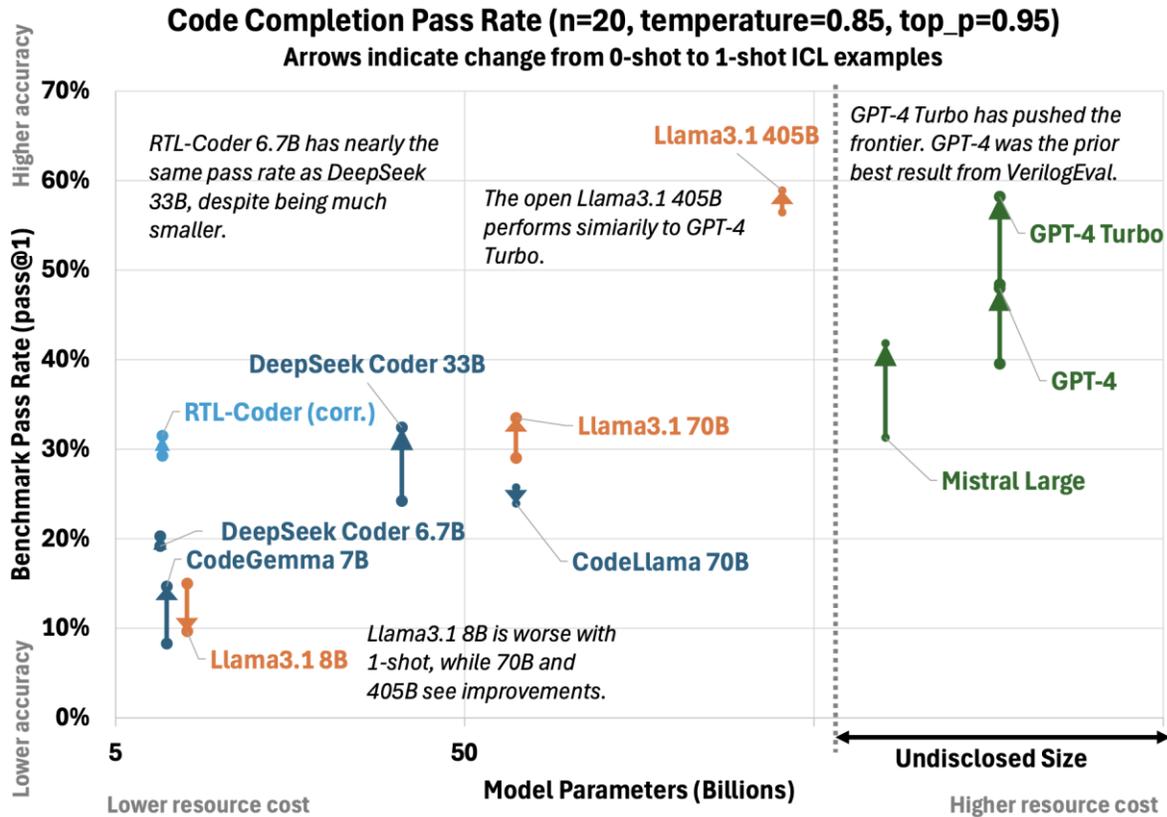
Figure 1: Our proposed automated training dataset generation flow.

# 3. Domain-adaptive LLMs for Chip design tasks

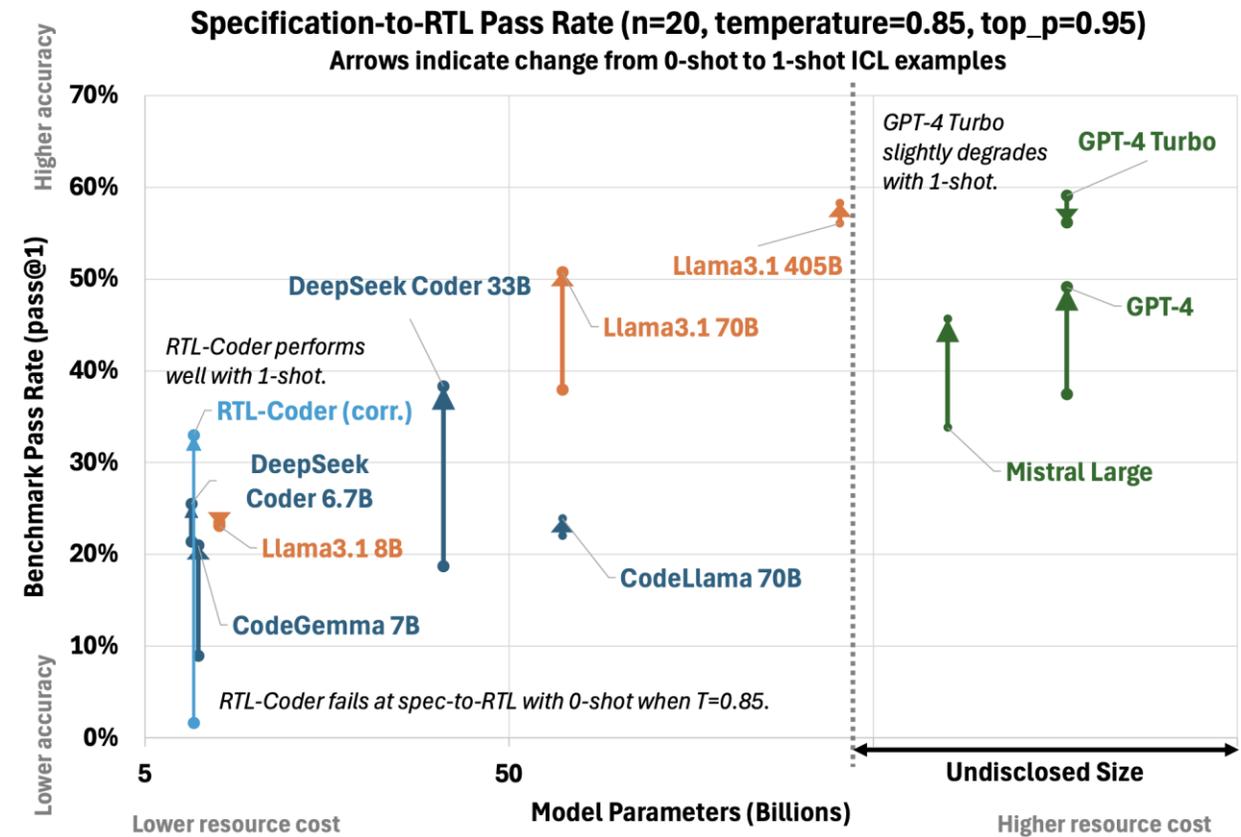
- 2) Task-specific fine-tuning
- An example: RTLCoder
  - Task: RTL code generation
  - Foundation model: Mistral-7B-v0.1 and DeepSeek-Coder-6.7b
  - Data collection: 27K instruction-code pairs
  - Fine-tuning flow:



# 3. Domain-adaptive LLMs for Chip design tasks



(a) Code Completion Task



(b) Specification-to-RTL Task

### 3. Domain-adaptive LLMs for Chip design tasks

- 2) Task-specific fine-tuning (TSFT)

- More examples:

- VerilogEval: Evaluating Large Language Models for Verilog Code Generation (Liu et al.)
- Benchmarking Large Language Models for Automated Verilog RTL Code Generation (Thakur et al.)
- RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution (Liu et al.)
- Data is all you need: Finetuning LLMs for chip design via an automated design-data augmentation framework (Chang et al.)
- LLM4SecHW: Leveraging Domain-Specific Large Language Model for Hardware Debugging (Fu et al.)
- Unlocking Hardware Security Assurance: The Potential of LLMs (Meng et al.)

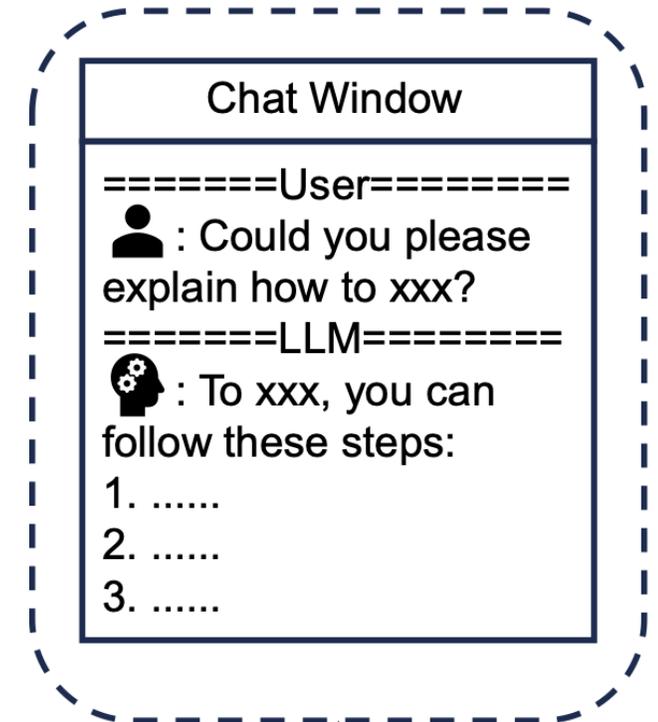
RTL code generation

Debugging  
Security

# 3. Domain-adaptive LLMs for Chip design tasks

- 3) Prompt engineering
  - Assistant ChatBot
    - Natural language understanding
    - Problem solving
    - Interaction

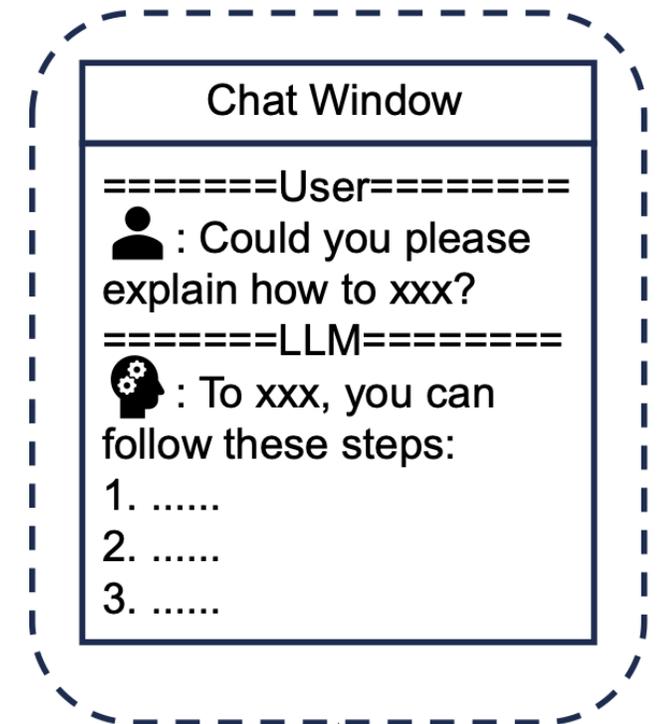
## Assistant Chatbot



# 3. Domain-adaptive LLMs for Chip design tasks

- 3) Prompt engineering
  - Assistant ChatBot
  - How to use:
    - ChatBot models:
      - *Instruction-tuned 'conversational' models*
      - E.g., ChatGPT, Llama-chat, ChipNemo-chat

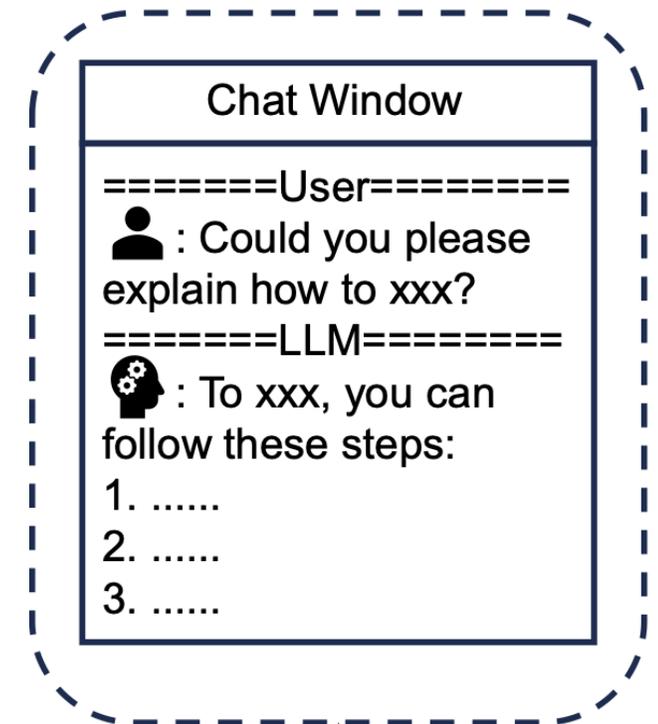
## Assistant Chatbot



# 3. Domain-adaptive LLMs for Chip design tasks

- 3) Prompt engineering
  - Assistant ChatBot
  - How to use:
    - ChatBot models:
      - *Instruction-tuned 'conversational' models*
      - E.g., ChatGPT, Llama-chat, ChipNemo-chat
    - Design prompts/queries: prompt engineering
      - Role play: “assume you are a hardware expert”
      - Provide (pos/neg) examples (few-shot learning)
      - Breakdown complicated problem: plan before execute
      - Specify constraints and output format: use [BEGIN], [END] tags
      - Chain-of-thought: “...solve this step by step”
      - Multi-turn interactions

## Assistant Chatbot



# 3. Domain-adaptive LLMs for Chip design tasks

- 3) Prompt engineering
  - Assistant ChatBot

## An initial prompt:

Please act as a professional verilog designer, try to understand the requirements below and reason how to solve the problem step by step. Based on your reasoning, complete the module with syntax correct Verilog code.

Implement the Verilog design of 4bit unsigned number pipeline multiplier. It consists of two levels of registers to store intermediate values and control the multiplication process. Assume that signals are positive clock/clock edge triggered unless otherwise stated.

```
module multi_pipe_4bit#(parameter size = 4)(
  input          clk      ,
  input          rst_n    ,
  input[size-1:0] mul_a   ,
  input[size-1:0] mul_b   ,
  output reg [size*2-1:0] mul_out
);
```

# 3. Domain-adaptive LLMs for Chip design tasks

- 3) Prompt engineering
  - Assistant ChatBot

## An initial prompt:

Please act as a professional verilog designer, try to understand the requirements below and reason how to solve the problem step by step. Based on your reasoning, complete the module with syntax correct Verilog code.

Implement the Verilog design of 4bit unsigned number pipeline multiplier. It consists of two levels of registers to store intermediate values and control the multiplication process. Assume that signals are positive clock/clock edge triggered unless otherwise stated.

```
module multi_pipe_4bit#(parameter size = 4)(
  input          clk      ,
  input          rst_n    ,
  input[size-1:0] mul_a   ,
  input[size-1:0] mul_b   ,
  output reg [size*2-1:0] mul_out
);
```

## Reponse:

Designing a 4-bit unsigned number pipeline multiplier involves creating a module that performs multiplication in stages, utilizing registers to hold intermediate results and control the multiplication process efficiently. Here's a step-by-step approach:

### 1. Understanding the Module Interface

Inputs: clk (clock), rst\_n (active-low reset), mul\_a, and mul\_b (4-bit multiplicands). Output: mul\_out (8-bit product). Parameters: size = 4 indicates the bit-width of the inputs.

### 2. Designing the Pipeline Stages Step by Step

Step1: Extension of input signals:

The input signals are extended by adding "size" number of zero...

Step2: Multiplication operation:

For each bit position of the multiplier, the partial product is calculated as ...

Step3: Add of partial products:

The module uses registers to store the intermediate sum values.

If the reset signal (rst\_n) is low ... If rst\_n is high, ...

Step4: Final product calculation:

On the positive edge of the clock signal (clk) or the falling edge of the reset signal (rst\_n), the module calculates the final product...

### 3. Implementation (not shown here)

**[With correct syntax but wrong functionality]**

# 3. Domain-adaptive LLMs for Chip design tasks

- 3) Prompt engineering
  - Assistant ChatBot more examples:
    - ChipNemo (Nvidia)
    - RapidGPT (PrimisAI)
    - ChatEDA: A Large Language Model Powered Autonomous Agent for EDA (He et al.)
    - ChipGPT: How far are we from natural language hardware design (Chang et al.)
    - Chip-Chat: Challenges and Opportunities in Conversational Hardware Design (Blocklove et al.)

# 3. Domain-adaptive LLMs for Chip design tasks

- 3) Agent: prompt engineering + simulator
  - Feedback loop + self-correction:
    - 1. Initial prompt creation
    - 2. Receive output from agent
    - 3. Feedback output to simulator/compiler
    - 4. Extract feedback
    - 5. Prompt based on feedback
    - 6. Repeat interaction for multi-turn prompting (1-5)

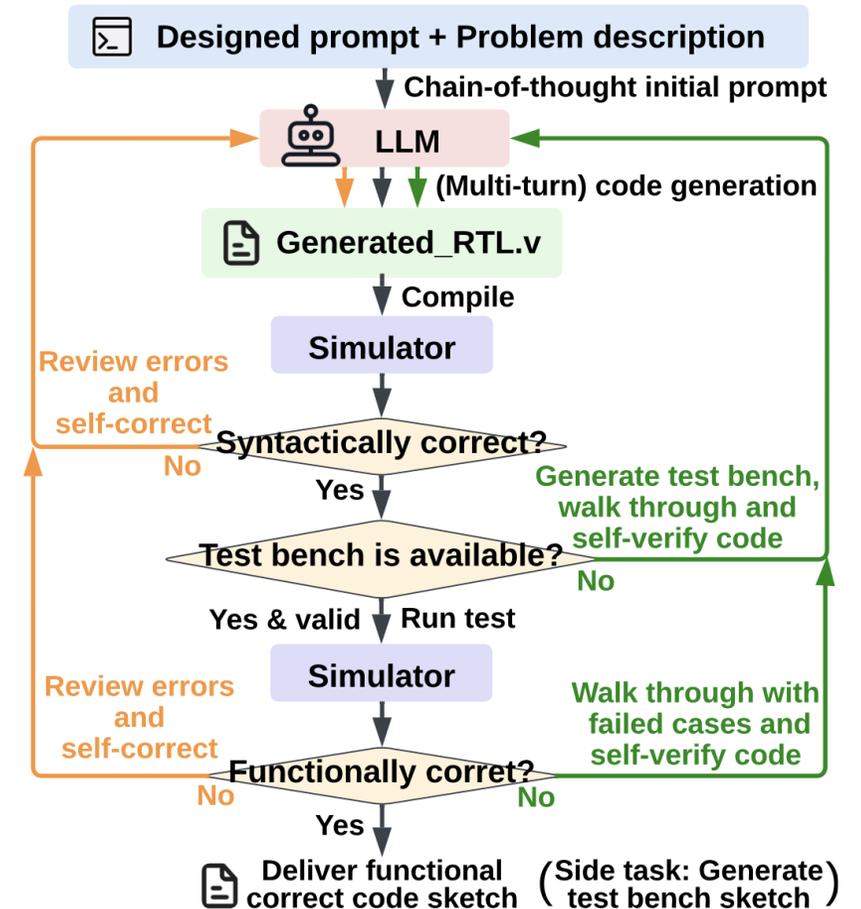


Figure 3: VeriAssist workflow. 36

# 3. Domain-adaptive LLMs for Chip design tasks

 **A follow-up prompt with self-verification:**

Please generate a testbench with test cases based on the generated RTL code.  
Walk through the RTL code step by step across time steps using a test case as input to reason the output. Based on this analysis, revise the code if necessary.

# 3. Domain-adaptive LLMs for Chip design tasks

## A follow-up prompt with self-verification:

Please generate a testbench with test cases based on the generated RTL code. Walk through the RTL code step by step across time steps using a test case as input to reason the output. Based on this analysis, revise the code if necessary.

## Response

Let's create a testbench and walk through the RTL code ...

### 1. Generated Testbench:

```
//Reset
#100;
rst_n = 1;
#20;
// Test Case 1: 3 * 5 = 15
mul_a = 4'b0011; // 3
mul_b = 4'b0101; // 5
#10; // Wait a stage
// Test Case 2: 15 * 15 = 225
mul_a = 4'b1111; // 15
mul_b = 4'b1111; // 15
#10; // Wait a stage
verify_output(4'b0011, 4'b0010, 4'b0110);
...
```

# 3. Domain-adaptive LLMs for Chip design tasks

## A follow-up prompt with self-verification:

Please generate a testbench with test cases based on the generated RTL code. Walk through the RTL code step by step across time steps using a test case as input to reason the output. Based on this analysis, revise the code if necessary.

## Response

Let's create a testbench and walk through the RTL code ...

### 1. Generated Testbench:

```
//Reset
#100;
rst_n = 1;
#20;
// Test Case 1: 3 * 5 = 15
mul_a = 4'b0011; // 3
mul_b = 4'b0101; // 5
#10; // Wait a stage
// Test Case 2: 15 * 15 = 225
mul_a = 4'b1111; // 15
mul_b = 4'b1111; // 15
#10; // Wait a stage
verify_output(4'b0011, 4'b0010, 4'b0110);
...
```

### 2. Step by Step Code Walking Through with a Test Case:

#### (1) Initial Conditions:

clk is toggling every 10 ns due to always #5 clk = !clk;

rst\_n is initially 0 (reset active).

#### (2) At t = 100 ns:

Reset Release: rst\_n transitions from 0 to 1. However, the inputs mul\_a and mul\_b haven't been set yet, so there are no operations performed related to multiplication.

#### (3) At t = 110 ns:

Inputs Set: mul\_a is set to 3 (0011) and mul\_b to 5 (0101).

#### (4) First Clock Cycle After Input Set (t = 110 ns to 120 ns):

The multi\_pipe\_4bit module starts processing the inputs.

Partial Product Generation based on each bit of mul\_b:

For mul\_b[0] (1): mul\_a << 0 = 0011

For mul\_b[1] (0): 0

For mul\_b[2] (1): mul\_a << 2 = 1100

For mul\_b[3] (0): 0

...

3. Implementation (not shown here)

**[Improved with more test cases passed]**

# 3. Domain-adaptive LLMs for Chip design tasks



## A follow-up prompt with self-correction:

[Simulator logs] + Please review the simulator feedback for the previously generated code. Please analyze the errors and examine the code with an input in the failed cases and deductively reason out the output. Based on this analysis, fix the errors in the previous code.



## Response:

Based on the testing results you've shared, the issues likely stem from how the partial products are combined.

1. Revised Approach:

...

2. Step by Step Code Walking Through with a Failed Test Case:

# 3. Domain-adaptive LLMs for Chip design tasks



## A follow-up prompt with self-correction:

[Simulator logs] + Please review the simulator feedback for the previously generated code. Please analyze the errors and examine the code with an input in the failed cases and deductively reason out the output. Based on this analysis, fix the errors in the previous code.



## Response:

Based on the testing results you've shared, the issues likely stem from how the partial products are combined.

1. Revised Approach:

...

2. Step by Step Code Walking Through with a Failed Test Case:

## 3. Revised Verilog Code: [with fully correct functionality]

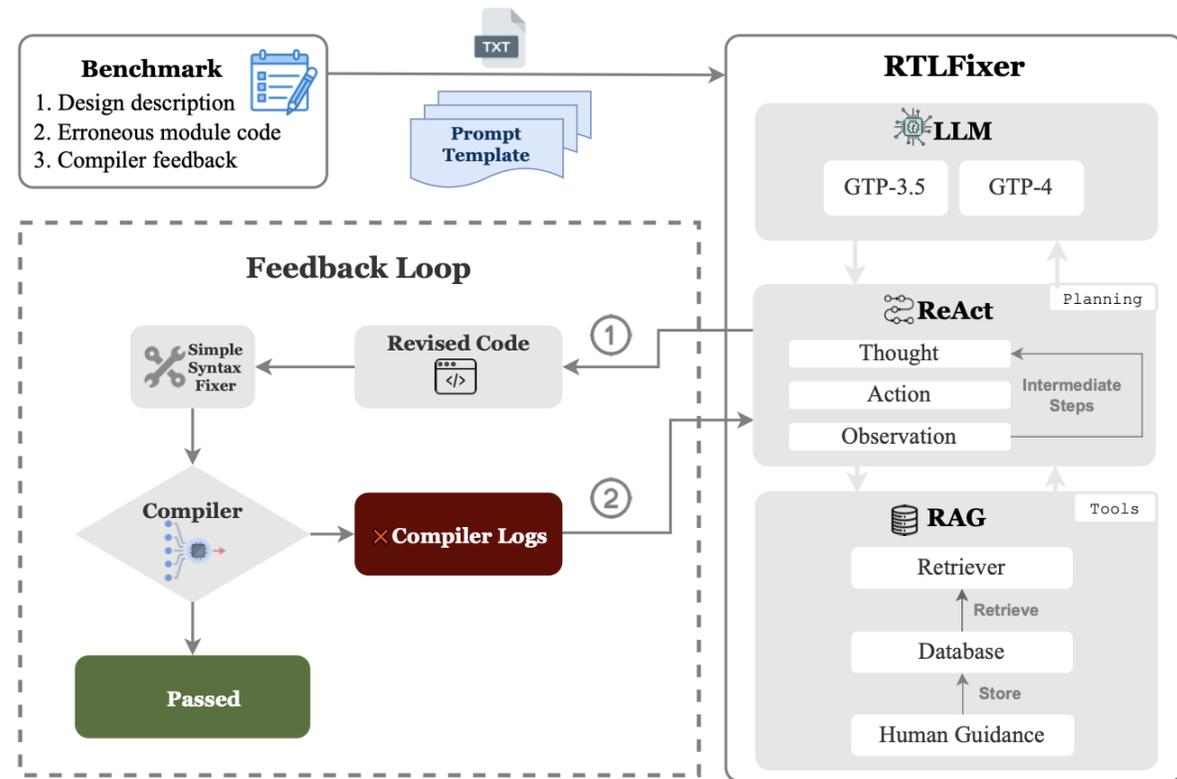
```
...
genvar i;
// Generate partial products using combinational logic
generate
  for (i = 0; i < size; i = i + 1) begin : gen_partial_products
    assign partial_products[i] = mul_b[i] ? (mul_a << i) : 0;
  end
endgenerate
// First addition stage: Accumulate first half of the partial products
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    intermediate_sum[0] <= 0;
    intermediate_sum[1] <= 0;
  end else begin
    intermediate_sum[0] <= partial_products[0] + partial_products[1];
    intermediate_sum[1] <= partial_products[2] + partial_products[3];
  end
end
// Second addition stage: Accumulate the remaining partial products
and calculate the final output
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    final_sum <= 0;
    mul_out <= 0;
  end else begin
    final_sum <= intermediate_sum[0] + intermediate_sum[1];
    mul_out <= final_sum;
  end
end
end
```

# 3. Domain-adaptive LLMs for Chip design tasks

## • 3) Prompt engineering

### • Retrieval-Augmented Generation

- Categorize various syntax errors
- Collect human expert guidance: detailed explanations for compiler logs
- Build a database for compiler logs, error code segments and human guidance



# 3. Domain-adaptive LLMs for Chip design tasks

- 3) Prompt engineering
  - **Retrieval-Augmented Generation**
    - Categorize various syntax errors
    - Collect human expert guidance: detailed explanations for compiler logs
    - Build a database for compiler logs, error code segments and human guidance

**Compiler Logs:**

Object 'clk' is not declared. Verify the object name is correct. If the name is correct, declare the object.

**Human Expert Guidance:**

Check if 'clk' is an input. If not, and if 'clk' is used within the module, make sure the name is correct. If it's meant to trigger an 'always' block, replace 'posedge clk' with '\*'.

**Compiler Logs:**

Index cannot fall outside the declared range for vector

**Human Expert Guidance:**

Carefully examine the index values to prevent encountering 'index out of bound' errors in your code. When utilizing parameters for indexing, try to use binary strings for performing the indexing operation instead.

# 3. Domain-adaptive LLMs for Chip design tasks

- 3) Prompt engineering
  - **Retrieval-Augmented Generation**
    - Categorize various syntax errors
    - Collect human expert guidance: detailed explanations for compiler logs
    - Build a database for compiler logs, error code segments and human guidance

**Compiler Logs:**

Object 'clk' is not declared. Verify the object name is correct. If the name is correct, declare the object.

**Human Expert Guidance:**

Check if 'clk' is an input. If not, and if 'clk' is used within the module, make sure the name is correct. If it's meant to trigger an 'always' block, replace 'posedge clk' with '\*'.

**Compiler Logs:**

Index cannot fall outside the declared range for vector

**Human Expert Guidance:**

Carefully examine the index values to prevent encountering 'index out of bound' errors in your code. When utilizing parameters for indexing, try to use binary strings for performing the indexing operation instead.

Prompt	RAG	Simple	iverilog	Quartus	GPT-4
One-shot	w/o	0.414	0.536	0.587	0.91
	w/	-	0.800	0.899	0.98
ReAct	w/o	0.671	0.731	0.799	0.92
	w/	-	0.820	0.985	0.99

fix rate

# Future Directions

- Borrow ideas and techniques from adjacent domains:
  - LLMs for software development, formal verification
- More (open-sourced) datasets and benchmarks are needed
  - Human-in-the-loop approaches
  - Incorporating PPA metrics
  - Make the best of multi-modal data
    - Text, code, diagrams, specifications
- Collaboration between academia and industry
  - Academia: theoretical and fundamental topics, advancing novel algorithms
  - Industry: solving practical real-world problems, more data and resource

Thank you for listening!



# Domain-adaptive LLMs for Chip Design

Hanxian Huang

[hah008@ucsd.edu](mailto:hah008@ucsd.edu)

hanxian97.github.io

Advisor: Prof. Jishen Zhao

University of California San Diego