SynthAI: A ReAct-CoT-RAG Framework for Automated Modular HDL/HLS Design Generation

Anonymized Authors
Affiliation
address
Email: email

***Abstract*—In this work, we present an innovative method for the automated generation of Hardware Description Language (HDL) and High-Level Synthesis (HLS) designs. Our approach integrates ReAct agents, Chain-of-Thought (CoT) prompting, web search, and the Retrieval-Augmented Generation (RAG) framework. Moving beyond traditional LLM training, we utilize a dynamic, internet-enhanced RAG knowledge base. ReAct agents, through CoT prompting, break down complex hardware designs into smaller modular tasks. Compared to results from prior methods, SynthAI yields synthesizable designs that accurately follow the description in a larger number of instances; i.e., our initial results show significant improvements in functional correctness and synthesizability in simple, single-module, logic design. We also show that a multi ReAct agent system (such as SynthAI) can be used for complex multi-module logic design based on a single prompt. The SynthAI code is provided via the following anonymized repo: https://anonymous.4open.science/r/FPGA_AGI-82C8/**

I. Introduction

Multiple studies have benchmarked and evaluated the capabilities and limitations of Large Language Models (LLMs) in the emergent field of automated hardware code generation. We begin by providing a general overview of the field.

LLMs have shown promising capabilities in generating basic logic hardware designs. The concept of deriving hardware description code from natural language was explored by *Pearce et al.* [1]. Further advancements were discussed in the field of conversational hardware design, emphasizing the challenges and opportunities, as highlighted by *Blocklove et al.* [2] and *Chang et al.* [3]. All three have shown LLMs' ability to generate basic logic code.

In the realm of specialized LLMs for code generation, *Thakur et al.* [4] investigated LLMs' capacity to produce Verilog code after fine-tuning. They demonstrated that fine-tuning LLMs with Verilog datasets enhances their ability to generate syntactically correct code. Their study was groundbreaking in developing an evaluation framework for both functional analysis and syntax testing of the generated code. The main contribution of their work lies in providing open-source training, evaluation scripts, and LLM checkpoints. However, a notable limitation is their method's difficulty in managing advanced Verilog code generation tasks.

The LLM4SecHW framework, developed by *Fu et al.* [5], focuses on employing LLMs for hardware design debugging, particularly addressing the scarcity of domain-specific data. Their significant contribution was that of creating a dataset for hardware design defects, paving the way for novel applications in hardware security.

Liu et al. [6] introduced VerilogEval, a framework aimed at evaluating LLMs in Verilog code generation. Their work is distinguished by establishing the first benchmarking framework capable of conducting comprehensive evaluations across various Verilog code generation tasks.

Du et al. [7] explored LLMs' application in generating HDL code for wireless signal processing, incorporating in-context learning and Chain-of-Thought (CoT) prompting. Their work stands out for its potential in generating complex HDL code for advanced applications, offering insightful contributions to wireless communication hardware.

In our work, we present a novel framework that, in effect, wraps LLMs, enabling them to utilize tools like web searches and database queries. Alongside the latter, we implement diverse prompting techniques, such as CoT, to achieve high-quality code akin to that produced by retrained or fine-tuned LLMs, without the need for actual retraining or fine-tuning. Our approach's uniqueness lies in its access to real-time internet data and a rich database comprising code samples, digital design guidelines, and FPGA datasheets. From our experiments, this enables higher quality code generation capability compared to prior approaches. Furthermore, our method transcends hardware description languages beyond Verilog, encompassing VHDL, System Verilog, and C++ for High-Level Synthesis (HLS). Notably, our approach is capable of planning and executing modular designs based on single prompts, showcasing its versatility and adaptability to complex hardware design tasks. The SynthAI code is provided via the following anonymized repo: https://anonymous.4open.science/r/FPGA_AGI-82C8/

II. Foundational LLM-based Technologies used in SynthAI

Here we highlight the different existing technologies we use as foundations for our proposed framework.

A. MRKL

Modular Reasoning, Knowledge, and Language (MRKL) is a specialized system in the field of artificial intelligence that focuses on enhancing the capabilities of large language models (LLMs) [8]. MRKL works by augmenting the natural language processing abilities of LLMs, enabling them to not

only generate responses based on their training data but also to access and incorporate additional information from outside databases or knowledge bases. This integration allows for more informed and accurate responses, especially in situations where the LLM might otherwise lack sufficient data. In the context of our work, we augment the generic (non-fine-tuned) LLMs with digital design resources such as Verilog/VHDL code and/or digital design guides and textbooks. We also empower the LLM to autonomously execute Python code through an integrated Python shell.

B. ReAct agents

ReAct agents [9], within the context of the MRKL framework, are a class of intelligent systems designed to enhance the dynamic interaction and decision-making capabilities of LLMs. In practice, ReAct agents facilitate a more integrated approach to handling tasks, where they interact with the MRKL system to provide a continuous loop of **reasoning, action,** and **feedback**. To be more precise, the ReACt/MRKL agent as implemented in LangChain [10] and cited below consists of the following steps (as commands to the LLM):

```
"""
Thought: Think about what to do next
Action: The action to take, is one of [{
    tool_names}]
Action Input: The input to the action
Observation: The result of the action
... (this Thought/Action/Action Input/
    Observation can repeat N times)
Thought: The final answer is reached
Final Answer: The final answer to the
    original input question
"""
```

In essence, the agent is granted access to a suite of tools (namely, *tool_names*), within which it iteratively engages in "Thought" steps, thereby making informed decisions about whether or not to utilize a specific tool in the next Action step. After using a tool, it assesses the result and executes "Thought" steps followed by more Action steps, until it reaches a "Final Answer".

C. Retrieval Augmented Generation (RAG)

Another technique we use in SynthAI is Retrieval Augmented Generation (RAG) [11], [12], [13], a sophisticated method in AI that boosts LLMs by incorporating external knowledge retrieval into their generative process. In practice, RAG involves constructing a vector database from semantic data. This database acts as a search tool for the LLM, enabling it to supplement its inherent knowledge (embedded in its parameters) with external, task-specific information. This approach is pivotal in expanding the knowledge base of our ReAct MRKL agents, ensuring they stay updated and relevant to specific tasks. For instance, in developing a RISC-V CPU, RAG allows one to integrate diverse resources like textbooks and coding best practices specific to RISC-V design. This integration not only broadens the information horizon for the agents but also tailors their responses and actions to be more aligned with current, domain-specific knowledge.

D. Chain of Thought prompting

The final AI concept used in our SynthAI framework is *Chain of thought*(CoT) prompting [14], [15] which is a method that enhances the reasoning capabilities of LLMs. COT prompting involves guiding the LLMs through a series of logical steps or thought processes to arrive at a conclusion, rather than directly generating an answer. CoT prompting has shown significant improvements in the performance of LLMs across various tasks, including arithmetic, commonsense, and symbolic reasoning. The key to its effectiveness lies in its ability to make the LLM's reasoning process more structured, thereby improving accuracy and reducing errors that might arise from model hallucinations.

III. SynthAI Framework

Our work described here aimed at developing a comprehensive proof of concept that seamlessly translates a user-defined design objective into a fully-realized hardware solution. The core of our approach involves a suite of specialized agents that collaboratively interpret the provided objective, meticulously evolve it into a detailed design, and subsequently generate synthesizable HDL/HLS code (along with corresponding test benches if asked). Here, we predominantly concentrate on elucidating the agent responsible for the generation of the synthesizable code. Alongside this primary focus, we also provide a concise overview of the other integral components (agents) of our SynthAI system.

A. Core Agents

SynthAI consists of three agents: a *Requirement Agent*, a *Module Design Agent* and a *HDL Generation Agent*[1]. We next describe each of these agents.

1) Requirement Agent: The Requirement Agent **ReA** in our SynthAI system plays a pivotal role in shaping the initial phase of the design process. Upon receiving a user-defined objective, **ReA** is tasked with generating a comprehensive set of goals, requirements, and a module tree diagram. In our current implementation, this functionality is achieved through a streamlined **ReA** agent programmed with specific instructions to articulate the necessary text that embodies these design parameters. Notably, **ReA** is not only equipped with internet search capabilities but also utilizes a Retrieval Augmented Generation (RAG) system. This RAG is curated with an extensive array of digital design resources, including textbooks on digital design, FPGA datasheets, various technical documentation (e.g., reference designs), and board specifications. This rich repository of information empowers **ReA** to produce well-informed and contextually relevant design prerequisites, setting a solid foundation for the subsequent stages of the design process.

[1]There is also an optional test-bench agent.

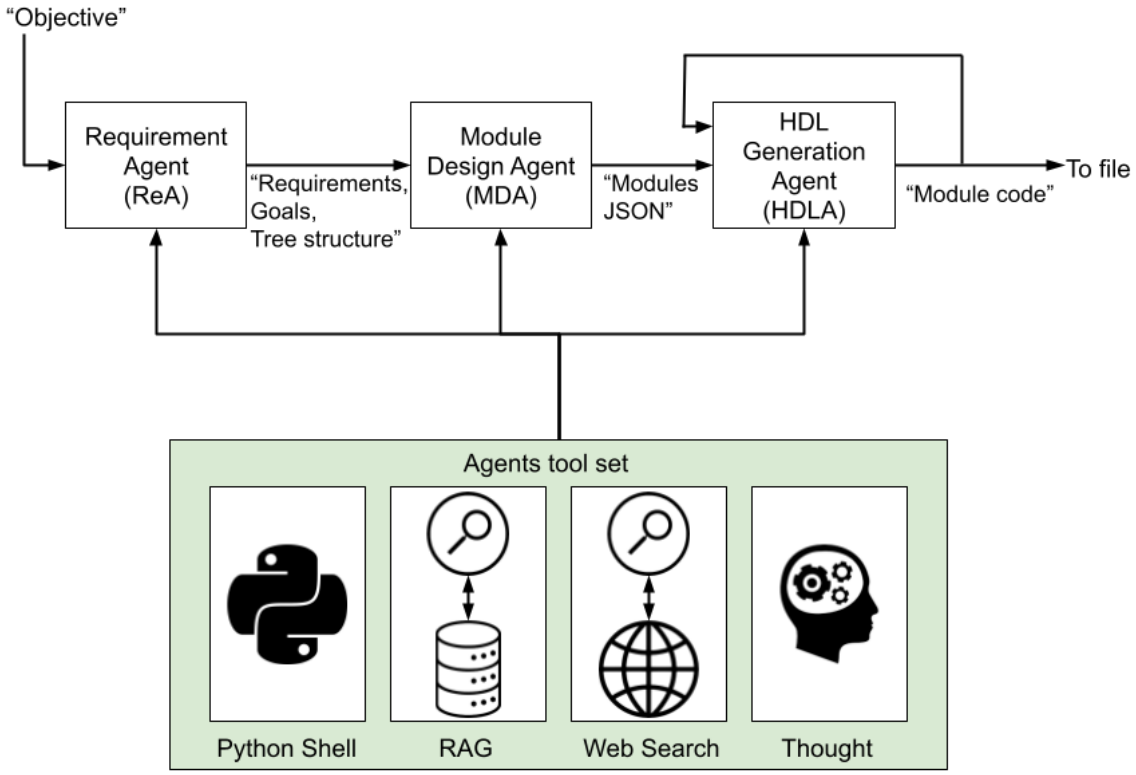

Fig. 1. Flow diagram of SynthAI and agent/tool interactions.

2) Module Design Agent: For the purpose of module generation, we innovatively adapted a ReAct agent. This adaptation involves specific modifications to the base instructions of the agent to align with the **ReA** design goals, requirements and tree diagram. A key alteration in our approach is the enhancement of the "thought" process. Contrarily to the conventional sequence of one 'thought' followed by one 'action', our Module Design Agent **MDA** incorporate 'thought' as a tool. This innovative modification essentially blends Chain of Thought (CoT) prompting with the ReAct framework, thereby enriching the ovarall decision-making process.

The following is a structured outline of the operational flow of **MDA**:

```
"""
Generate a list of necessary modules and
    their descriptions for the FPGA based
    hardware design project.
Include a top module which contains all of
     the other modules in it. Return the
    results in an itemized markdown format.
You have access to the following tools:
{tools}
Use the following format:
Goals: the main goal(s) of the design
Requirements: step by step plan
Tree: tree structure
Break: you Must always break down the
    necessary sub-tasks in markdown format
Thought: you should think about what to do
Action: the action to take, should be one
    of [{tool_names}]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Thought/Thought/Action/
    Action Input/Observation can repeat N
    times)
Thought: I now know the final answer
Final Answer: the final answer to the
    original input question must be a list
    of JSON dicts of modules and
    descriptions with the following format
    for each module
{{  "Module_Name": "name of the module",
"ports": ["specific inputs and outputs,
    including bit width"],
"description": "detailed description of
    the module function including any
    computed values or details needed for
```

```
    generating the module code",
"connections": ["specific other modules it
     must connect to"],
"hdl_language": "hdl language to be used"
}}
"""
```

This structured approach enables **MDA** to systematically process and refine its understanding of the design requirements before committing to a specific action. The final output is expected to be a detailed JSON dictionary of modules, encompassing all necessary attributes, from module names to HDL languages, thereby providing a comprehensive blueprint for the design.

Note that **MDA** is also buttressed by a customized RAG, internet search tool, Python shell tool and the "thought" tool which was mentioned earlier.

3) HDL Generation Agent: The HDL generation agent **HDLA** operates by receiving a meticulously compiled list of modules from the previous stage, which it uses as a foundation for generating HDL/HLS code. It iterates over the given JSON list of modules previously generated by the **MDA** and attempts to write codes for each module.

HDLA explicitly receives both the primary aspects of the design, including the goals, requirements, and the tree diagram generated by **ReA**, as well as a detailed 'Module list', which is a comprehensive enumeration of the modules generated by the **MDA**. Additionally, throughout its iterative process, **HDLA** revisits 'Module Codes' for modules it has previously coded and completed, ensuring consistency and integration of its own work into the evolving design.

The prompt template for the HDLA is provided in the following:

```
"""
You will code the module given after "
    Module". You will write fully
    functioning code not code examples or
    templates.
The final solution you prepare must
    compile into a synthesizable FPGA
    solution. It is of utmost important
    that you fully implement the module and
     do not leave anything to be coded
    later.
Some guidelines:
- DO NOT LEAVE ANYTHING TO THE USER AND
    FULLY DESIGN A SYNTHESIZABLE CODE USING
     THE TOOLS UNLESS explicitly told to do
     so.
- DO NOT leave any placeholders for the
    user. fill everything out and take
    advantage of the tools you have access
    to.
- IF you need to perform any analysis or
    computations, check the tools for
    guidelines.
- When using document search, you might
    have to use the tool multiple times and
     with various search terms in order to
    get better resutls.
- Leave sufficient amount of comments in
    the code to enable further development
    and debugging.
- Make sure that you go always through the
     "Break" step in the following
You have access to the following tools:
{tools}
Use the following format:
Goals: the main goal(s) of the design
Requirements: step by step plan
Tree: Tree Structure
Module list: list of modules you will
    build
Module Codes: HDL/HLS code for the modules
     that you have already built
Module: The module that you are currently
    building
Break: you Must always break down the
    necessary sub-tasks in markdown format
Thought: you MUST always think of an
    Action either to learn about doing
    something or to find examples
Action: the action to take, should be one
    of [{tool_names}]
 Action Input: the input to the action
 Observation: the result of the action
 ... (this Thought/Action/Action Input/
    Observation can repeat N times)
Thought: I now know the final answer
Final Answer: You write the HDL/HLS code.
    Synthesizable HLS/HDL code in line with
     the Goals, Requirements and tree. This
     code must be fully implemented and no
    aspect of it should be left to the user
    . Do not return any comments or extra
    formatting.
"""
```

Troughout this module design process, **MDA** engages in a 'Thought and Action' cycle, where it contemplates the best course of action, executes it, and observes the outcome. This loop continues until the agent is confident in its final output. The culmination of this process is the generation of synthesizable HLS/HDL codet. Note that this agent also has access to our "RAG" tool as well as our "thought" tool which is our innovative way of enabling our ReAct agent to engage in a CoT process.

B. Customized RAG and other tools

1) The LLM models: Here we employed the OpenAI GPT-4-1106-preview model, notably for its remarkable context length capability. This model can process a context of up to 128,000 tokens. This extensive context length is invaluable for accommodating the complex nature of our targeted tasks, allowing for comprehensive processing and analysis of large amounts of data.

2) RAG tool: For embedding generation and vector search, as part of the RAG, SynthAI utilizes the text-embedding-ada-002 model. Note that for creating accurate semantic embeddings essential for effective vector-based information

retrieval, there are text-embedding models that are better than text-embedding-ada-002. However, this model is cheap to deploy, faster and compatible with the other models we are using. We deployed RAG with Maximal Marginal Relevance (MMR) [16] for vector search. MMR is a technique used to select documents that balance relevance to a given query with diversity among the results. This approach is designed to overcome the challenge of returning highly similar or redundant documents in response to a query.

Our RAG version returns the top 3 results based on the MMR criteria such that the search operation by the agent can provide a set of results that are not only relevant to the search query but also diverse, offering a broader range of information on the topic.

3) Web search tool: SynthAI also uses a Google search API and allow the agents to perform internet searches. In particular, SynthAI allows **ReA** and **MDA** to use the web search tool but **HDLA** cannot use internet search (it only needs to search the available documents, datasheets and code samples).

4) Python shell tool: The core agents **ReA, MDA** and **HDLA** are provided with a python shell tool where they can write their own python code and execute it. The primary purpose of this tool is to act as a scientific calculator and enable the core agents to perform complex calculations and analysis beyond LLMs' superficial ability to perform only simple arithmetic.

IV. Results and comparisons

SynthAI framework differs significantly from previous LLM-based works because, in effect, we incorporate the LLM into an MRKL agent. This integration allows the agent to selectively utilize various tools to formulate queries, aiding in the generation of functional, synthesizable designs. Furthermore, SynthAI facilitates a comprehensive modular design process, encompassing everything from the initial intake of an 'objective' to the final delivery of a modular design, complete with code and test benches.

To assess the performance of SynthAI and how it compares to that of other works, we report on two sets of comparisons here: a first is a) between our **HDLA** with *Thakur et al.* [4] and a second set is b) between our end to end solution with that from *Du et al.* [7].

Starting with *Thakur et al.* we compare the performance of **HDLA** with the hardest problem set mentioned in the Table 2 of *Thakur et al.'s* work. We reproduced parts of this Table here for reader convenience. In their work, *Thakur et al.* compare a series of fine-tuned models against various problem sets (*Basic, Intermediate, Advanced*). Here we only compare our work with the advanced set because we get a consistently (i.e., 100%) correct code which is synthesizable and passes all of the behavioral tests that we crafted for the Basic and Intermediate cases (as reported in *Thakur et al.*).

For the *Advanced* set presented in Table I, we generated 10 samples for each problem (a total of 50 samples) and performed behavioral simulations against our test-benches. We then calculated a score based on the fraction of sample codes that pass the tests. Note that *Thakur et al.* present two scores: one for the fraction of samples that compile (which, in our case, approaches the value of 1) and another for the fraction of samples that pass the tests.

TABLE I
Summary of the "Advanced" Problems from *Thakur et. al*[4] and their difficulty levels.

Difficulty	Description
Advanced	Signed 8-bit adder with overflow
Advanced	Counter with enable signal
Advanced	FSM to recognize 101
Advanced	64-bit arithmetic shift register
Advanced	ABRO FSM

HDLA achieves a score of 0.78 in our tests, approximately twice as high as the highest score reported by *Thakur et al.* in their work.

Important note: While our method outperforms the simple prompt completions discussed in *Thakur et al.* [4], it is crucial to acknowledge that the model we use in SynthAI, i.e., OpenAI GPT-4-1106-preview, differs significantly. This model is closed-source and has a larger context length (128,000 tokens), making it more costly to operate. In contrast, most models used by *Thakur et al.* (except for code-davinci-002) are smaller, open-source, and can be run on a workstation.

Our second more interesting comparison is with the results of *Du et al.* [7]. In their work, they moved away from simple HDL codes generated via simple prompt completion tasks and tried to generate sophisticated code (in particular FFT) via CoT prompting and in-context learning (ICL), both of which we do as well with SynthAI. Their approach towards ICL as well as dealing with the issues of the multi-step thinking (inherent to LLMs generally [**?**]) is to take a step by step multi-prompt approach. In particular, for FFT code generation, they start by asking the LLM (in their case ChatGPT) to design a butterfly computation and a complex multiplication module. Then they ask the model (given its history or previously designed modules) to design a 4-point FFT module out of the previously designed modules and the 2-point FFT that they pass as an input. They also explicitly ask the model to leave the twiddle factors to be computed later. They argue that it is too complex for the AI LLM. They follow up by asking ChatGPT to generate the twiddle factors after giving it detailed instructions on how to generate them. They claim that both of these approaches are in fact successful. In other words, the step by step in-context learning approach leads to proper FFT code generation and computation of twiddle factors

Our approach in SynthAI is different in that we do not "prime" the LLM through a manual step by step design. We give a detailed prompt at the beginning and let SynthAI figure out the hierarchy and do the design without explicit human intervention. In the case of FFT design, We also allow the SynthAI agents to compute the twiddle factors on their own given the Python shell tool that we made available to them. Our goal in SynthAI is to delegate more of the intellectual work to the MRKL agents. Our input prompt, partly borrowed

from *Du et al.*, is the following:

```
objective = """
Write an eight-point DIF-FFT verilog code.
     You will require the following IP
    cores to build the target four-point FT
     IP core.

Here is the template of butterfly
    computation IP Core,
butterfly_computation butterfly_inst(.clk
    (),.reset(),.enable(),.in1(), .in2(),.
    done(),.out1(), .out2());
And here is the template of the four-point
     FFT IP Core,
fft_4_point fft_4_point_inst(.clk(),.reset
    (),.enable(),.x1(),..., .x4(),.done(),.
    y1(),..., .y4());
And here is the template of complex
    multiplication IP Core,
complex_multiplication
    complex_multiplication_inst(.clk(),.
    reset(),.enable(),.in1(), .in2(),.done
    (),.out());

Here's the template of the target eight-
    point FFT IP Core,
fft_8_point fft_8_point_stage_2_1(.clk(),.
    reset(),.enable(),.x1(), ..., .x8(),.y1
    (), ..., .y8(),.done());

Do not forget clock, reset and done!
For the the fft_4_point module. Implement
    it like the following:

module fft_4_point(inputs);
// To be implemented by the user
endmodule

The input and output signals (x's and y's)
     are 32 bits each with 16 bits in the
    real part (MSB) and 16 bits in the
    imaginary part (LSB).
You have to compute the twiddle factors
    for an eight-point fft and scale them
    to 16 bits using the python_repl tool.
    The range for 16 bit integer is
    [-32768, 32767].
Define twiddle factors as local params
    within the top module.
"""
```

We ran the end to end SynthAI 15 times and collected 15 different designs. Our first observation was that in most of the cases SynthAI correctly understands what the objective is asking and properly defines the modular architecture of the design. We conducted 15 distinct trials using the SynthAI system, resulting in 15 unique designs. Our preliminary analysis revealed that SynthAI generally excels in interpreting the design objectives and structuring the modular architecture appropriately, as detailed in Appendix A. In the majority of trials, it effectively engineered both the butterfly and the complex multiplier modules. However, there were instances where it diverged from our specified prompt, such as combining x and y ports into buses or omitting the reset signal. A consistent shortfall was observed in its inability to accurately connect inputs to the butterfly modules and to correctly apply the twiddle factors in the multiplication process. In other words, while it could generate the single modules it failed at generating a working design. Note also that 10 out of these 15 designs were in fact synthesizable (but not functional).

SynthAI does not effectively interpret the Cooley-Tukey algorithm for FFT, nor does it autonomously search for such complex algorithms. This limitation aligns with challenges faced by AI in executing hierarchical, recursive, or dynamic programming-based designs, which require a deterministic, predefined strategy. While a ReAct agent can perform tasks like planning, searching, and Chain-of-Thought reasoning with appropriate tools, it still struggles with complex procedures without direct instruction. We attempted a second test by introducing in-context learning within the prompt by augmenting the above prompt with the following description of the FFT algorithm:

```
objective = """
Write an eight-point DIF-FFT verilog code.
     You are creating this design based on
    the Radix-2 Cooley-Tukey method of
    using two N/2 point FFT modules to
    create a N point FFT module.
The N-point FFT algorithm processes input
    data through a series of "butterfly"
    modules, which pairwise combine
    elements to produce outputs that are
    sums and differences of the inputs.
Following the butterfly stage, half of the
     outputs are multiplied by complex
    exponential twiddle factors to
    incorporate the necessary frequency
    domain scaling.
Finally, these processed elements are fed
    into two smaller N/2-point FFT blocks,
    which recursively apply the same
    process until the base case is reached,
     resulting in the final transformed
    sequence.
...The rest of the prompt as previously
    mentioned.
"""
```

Utilizing the updated prompt led to markedly improved outcomes, with 14 of the 15 designs achieving structural accuracy and correctly representing the algorithm's staging. Additionally, 11 designs accurately calculated the twiddle factors. However, there was a notable divergence in implementation methods: 7 out of the 15 designs adopted a combinational approach for the multiplier and butterfly modules, using input sensitivity rather than a clock, whereas the remaining 8 designs were fully sequential. Also, enable and done signals were ignored in some cases (An example is given in Appendix E). Overall we consider this second attempt successful.

Utilizing the 'Harris and Harris' digital design textbook [17]

included in the RAG knowledge base, SynthAI successfully completed the design of a simple RISC-V CPU. Due to space constraints, the detailed discussion and outcomes of the CPU design are omitted here. However, the initial prompt used for this design is available in a Jupyter notebook on our GitHub repository.

V. Conclusion

The SynthAI framework we developed and described here is a multi-agent system with document search and python execution ability. Compared to earlier works, SynthAI achieves a notable improvement in automated HDL generation, particularly for complex electronic design tasks. While SynthAI outperforms traditional LLM-based methods in accuracy and modular structuring, its limitations are apparent in handling complex algorithms without detailed instructions, emphasizing the need for continuous human collaboration in AI-driven processes. Despite these latter challenges, SynthAI's ability to adapt to augmented prompts suggests potential for further advancements in electronic design automation. Its success in integrating multiple tools and generating synthesizable code points towards evolving roles of AI in design methodologies, balancing automation with human expertise.

References

[1] H. Pearce, B. Tan, and R. Karri, "Dave: Deriving automatically verilog from english," in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, 2020, pp. 27–32.

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

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

[4] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog rtl code generation," *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. [Online]. Available: https://par.nsf.gov/biblio/10419705

[5] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, "Llm4sechw: Leavering domain-specific large language model for hardware debugging," *Asian Hardware Oriented Security and Trust (AsianHOST)*, 2023.

[6] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[7] Y. Du, S. C. Liew, K. Chen, and Y. Shao, "The power of large language models for wireless communication system development: A case study on fpga platforms," *arXiv preprint arXiv:2307.07319*, 2023.

[8] E. Karpas, O. Abend, Y. Belinkov, B. Lenz, O. Lieber, N. Ratner, Y. Shoham, H. Bata, Y. Levine, K. Leyton-Brown *et al.*, "Mrkl systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning," *arXiv preprint arXiv:2205.00445*, 2022.

[9] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[10] H. Chase, "Langchain," https://github.com/langchain-ai/langchain, 2022, accessed: [your date of access].

[11] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, 2023.

[12] Z. Levonian, C. Li, W. Zhu, A. Gade, O. Henkel, M.-E. Postle, and W. Xing, "Retrieval-augmented generation to improve math question-answering: Trade-offs between groundedness and human preference," *arXiv preprint arXiv:2310.03184*, 2023.

[13] W. Ding, Y. Cao, D. Zhao, C. Xiao, and M. Pavone, "Realgen: Retrieval augmented generation for controllable traffic scenarios," *arXiv preprint arXiv:2312.13303*, 2023.

[14] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[15] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," *arXiv preprint arXiv:2210.03493*, 2022.

[16] J. Carbonell and J. Goldstein, "The use of mmr, diversity-based reranking for reordering documents and producing summaries," in *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, 1998, pp. 335–336.

[17] S. Harris and D. Harris, *Digital Design and Computer Architecture, RISC-V Edition*. Morgan Kaufmann, 2021.

Appendix

To further exemplify SynthAI, we provide examples of the agents' thought, action and observation process along with our prompts.

A. Modular Design of the 8-point FFT from 4-FFT

The following JSON formatted design contains the typical modular design generated by SynthAI for the 8-point FFT task:


```
[
    {
        "Module_Name": "
    Two_Point_FFT_Module",
        "ports": ["input [63:0]
    complex_input1", "input [63:0]
    complex_input2", "output [63:0]
    complex_output1", "output [63:0]
    complex_output2"],
        "description": "Performs a 2-point
     FFT on two 32-bit complex numbers and
    produces two 32-bit complex numbers as
    output.",
        "connections": ["Butterfly_Module"
    , "Complex_Multiplication_Module"],
        "hdl_language": "Verilog"
    },
    {
        "Module_Name": "Butterfly_Module",
        "ports": ["inputs omitted by
    authors to conserve space"],
        "description": "Performs the
    butterfly operations required for the
    FFT algorithm on four 32-bit complex
    numbers.",
        "connections": ["
    Two_Point_FFT_Module", "
    Complex_Multiplication_Module", "
    Top_Module"],
        "hdl_language": "Verilog"
    },
    {
        "Module_Name": "
    Complex_Multiplication_Module",
        "ports": ["input [63:0]
    complex_input1", "input [63:0]
    complex_input2", "output [63:0]
    complex_output"],
```

```
        "description": "Performs
    multiplication of two 32-bit complex
    numbers, considering fixed-point
    representation.",
        "connections": ["Butterfly_Module"
    , "Top_Module"],
        "hdl_language": "Verilog"
    },
    {
        "Module_Name": "Top_Module",
        "ports": ["input [255:0]
    complex_input", "output [255:0]
    complex_output"],
        "description": "Integrates all
    other modules, manages data flow, and
    provides the final 4-point FFT output."
    ,
        "connections": ["
    Two_Point_FFT_Module", "
    Butterfly_Module", "
    Complex_Multiplication_Module"],
        "hdl_language": "Verilog"
    }
]
```

Note that is some cases it also included a twiddle factor module where it decided to hard-code the twiddle factors in a lookup table.

B. Agent queries the vector database via the RAG tool to learn more about the FFT Algorithm

Thought: Before coding, I should verify the algorithm for a four-point FFT to ensure correct implementation.
Action: Doc_Search
Action Input: Radix-2 four-point FFT algorithm
Observation: 'The Radix-2 four-point FFT algorithm involves dividing the FFT computation into two stages. In the first stage, two 2-point FFTs are performed separately on the even and odd input values...

C. Agent fails to properly compute the twiddle factors

The following is an example where the agent incorrectly computes the twiddle factors using a python shell that we allowed it to use for computation. It does so for a 16-point FFT instead of an 8-point FFT and the results are incorrect.

... Action: Think ... Observation: your thought is: I need to compute the twiddle factors for an eight-point FFT and scale them to 16-bit precision. The twiddle factors for an N-point FFT are given by W_N^k = e^(-j*2*pi*k/N) for k=0,1,...,N-1. For an eight-point FFT, this means I need to compute W_8^k for k=0,1,...,7. I will use Python to calculate these values and scale them to 16-bit precision.

Now that I have a clear understanding of the twiddle factors, I will proceed to compute them using Python.
Action: python_repl
Action Input:

```
import numpy as np

# Define the number of points in the FFT
    and the precision
N = 8
precision = 2**15  # 16-bit precision with
     one bit for sign

# Compute the twiddle factors
twiddle_factors = np.exp(-2j * np.pi * np.
    arange(N) / N)

# Scale the twiddle factors to 16-bit
    precision
scaled_twiddle_factors = np.round(
    precision * twiddle_factors).astype(int
    )

# Separate the real and imaginary parts
real_parts = scaled_twiddle_factors.real
imag_parts = scaled_twiddle_factors.imag

# Print the twiddle factors in 16-bit
    fixed-point format
for k in range(N):
    print(f"Twiddle factor W_8^{k}: ({
    real_parts[k]}, {imag_parts[k]})")
```

Observation: Twiddle factor W_8^0: (32768, 0)
Twiddle factor W_8^1: (23170, 0)
Twiddle factor W_8^2: (0, 0)
Twiddle factor W_8^3: (-23170, 0)
Twiddle factor W_8^4: (-32768, 0)
Twiddle factor W_8^5: (-23170, 0)
Twiddle factor W_8^6: (0, 0)
Twiddle factor W_8^7: (23170, 0)

Thought: There seems to be an error in the computation. I need to correct the Python code to properly compute the real and imaginary parts of the twiddle factors. ...

D. Agent properly computes the twiddle factors

The following is an example of how an agent uses the python tool to properly compute the twiddle factors for an 8-point FFT.

... > Entering new RequirementAgentExecutor chain...
Thought: To begin with, we need to compute the twiddle factors for an eight-point FFT and scale them to 16 bits. The twiddle factors for an N-point FFT are given by \(e^{-j\frac{2\pi k}{N}} \), where j is the imaginary unit, k is the index of the twiddle factor, and N is the number of points in the FFT.

For an eight-point FFT, we have $N = 8$, and we need to compute the twiddle factors for $k = 0, 1, 2, 3$ (since the twiddle factors are periodic with period $N/2$).

The real and imaginary parts of the twiddle factors are given by: - Real part: $\cos\left(\frac{2\pi k}{N}\right)$ - Imaginary part: $-\sin\left(\frac{2\pi k}{N}\right)$

We will compute these values and scale them to fit within the 16-bit signed integer range.
Action: python_repl

Action Input:

```
import numpy as np

# Define the number of points in the FFT
    and the twiddle factor index range
N = 8
k_values = range(N//2)

# Compute the twiddle factors and scale
    them to 16-bit signed integer range
twiddle_factors = [(int(32767 * np.cos(2 *
     np.pi * k / N)), int(-32767 * np.sin(2
     * np.pi * k / N))) for k in k_values]
print(twiddle_factors)
```

Observation: [(32767, 0), (23169, -23169), (0, -32767), (-23169, -23169)] ...

E. 8-point FFT module design sample created by SynthAI

In what follows we present an example 8-point FFT design generated by SynthAI.

```
module Butterfly_Computation_Module(
    input clk,
    input reset,
    input [31:0] x1,
    //... the rest of the ports were removed by
    the authors to conserve space...
    output reg [31:0] intermediate_8
);

// Split the 32-bit inputs into 16-bit real and
    imaginary parts
wire [15:0] x1_real = x1[31:16];
wire [15:0] x1_imag = x1[15:0];
// ... the authors removed some wire
    declarations to conserve space

// Intermediate variables for butterfly
    computation
reg [15:0] temp_real;
reg [15:0] temp_imag;

always @(posedge clk or posedge reset) begin
    if (reset) begin
        intermediate_1 <= 32'd0;
// ... the authors removed some initializations
    to conserve space
        intermediate_8 <= 32'd0;
    end else begin
        // Butterfly computations
        // Stage 1: Pairwise addition and
    subtraction
        // Output 1 and 5
        temp_real = x1_real + x5_real;
        temp_imag = x1_imag + x5_imag;
        intermediate_1 <= {temp_real, temp_imag
    };

        temp_real = x1_real - x5_real;
        temp_imag = x1_imag - x5_imag;
        intermediate_5 <= {temp_real, temp_imag
    };

        // Output 2 and 6
        temp_real = x2_real + x6_real;
        temp_imag = x2_imag + x6_imag;
        intermediate_2 <= {temp_real, temp_imag
    };

        temp_real = x2_real - x6_real;
        temp_imag = x2_imag - x6_imag;
        intermediate_6 <= {temp_real, temp_imag
    };

        // Output 3 and 7
        temp_real = x3_real + x7_real;
        temp_imag = x3_imag + x7_imag;
        intermediate_3 <= {temp_real, temp_imag
    };

        temp_real = x3_real - x7_real;
        temp_imag = x3_imag - x7_imag;
        intermediate_7 <= {temp_real, temp_imag
    };

        // Output 4 and 8
        temp_real = x4_real + x8_real;
        temp_imag = x4_imag + x8_imag;
        intermediate_4 <= {temp_real, temp_imag
    };

        temp_real = x4_real - x8_real;
        temp_imag = x4_imag - x8_imag;
        intermediate_8 <= {temp_real, temp_imag
    };
    end
end

endmodule
```

Listing 1. Butterfly Computation Module

```
module Twiddle_Factor_Multiplication_Module(
    input clk,
    input reset,
    input [31:0] intermediate_5,
    input [31:0] intermediate_6,
    input [31:0] intermediate_7,
    input [31:0] intermediate_8,
    output reg [31:0] twiddled_5,
    output reg [31:0] twiddled_6,
    output reg [31:0] twiddled_7,
    output reg [31:0] twiddled_8
);

// Local parameters for the twiddle factors
localparam [31:0] W0 = {16'd32767, 16'd0}; //
    (32767, 0)
localparam [31:0] W1 = {16'd23169, 16'hA57E}; //
     (23169, -23169) with 2's complement for
    negative
localparam [31:0] W2 = {16'd0, 16'h8001}; // (0,
     -32767) with 2's complement for negative
localparam [31:0] W3 = {16'hA57E, 16'hA57E}; //
    (-23169, -23169) with 2's complement for
    negative

// Split the 32-bit inputs into 16-bit real and
    imaginary parts
wire [15:0] intermediate_5_real = intermediate_5
    [31:16];
wire [15:0] intermediate_5_imag = intermediate_5
    [15:0];
wire [15:0] intermediate_6_real = intermediate_6
    [31:16];
wire [15:0] intermediate_6_imag = intermediate_6
    [15:0];
```

```
wire [15:0] intermediate_7_real = intermediate_7[31:16];
wire [15:0] intermediate_7_imag = intermediate_7[15:0];
wire [15:0] intermediate_8_real = intermediate_8[31:16];
wire [15:0] intermediate_8_imag = intermediate_8[15:0];

// Intermediate variables for complex multiplication
reg [31:0] product_5, product_6, product_7, product_8;

// Complex multiplication function
function [31:0] complex_multiply;
    input [31:0] a, b; // a = a_real + j*a_imag, b = b_real + j*b_imag
    reg [15:0] a_real, a_imag, b_real, b_imag;
    reg [31:0] result;
    begin
        a_real = a[31:16];
        a_imag = a[15:0];
        b_real = b[31:16];
        b_imag = b[15:0];
        result[31:16] = a_real * b_real - a_imag * b_imag;
        result[15:0] = a_real * b_imag + a_imag * b_real;
        complex_multiply = result;
    end
endfunction

always @(posedge clk or posedge reset) begin
    if (reset) begin
        twiddled_5 <= 32'd0;
        twiddled_6 <= 32'd0;
        twiddled_7 <= 32'd0;
        twiddled_8 <= 32'd0;
    end else begin
        // Perform complex multiplication with twiddle factors
        product_5 = complex_multiply(intermediate_5, W1);
        product_6 = complex_multiply(intermediate_6, W2);
        product_7 = complex_multiply(intermediate_7, W3);
        product_8 = complex_multiply(intermediate_8, W0); // W0 is (1,0) so it's just a pass-through

        // Assign the twiddled outputs
        twiddled_5 <= product_5;
        twiddled_6 <= product_6;
        twiddled_7 <= product_7;
        twiddled_8 <= product_8;
    end
end

endmodule
```

Listing 2. Twiddle Factor Multiplication Module

```
module FFT_8_Module(
    input clk,
    input reset,
    input [31:0] x1,
    //... the rest of x and y ports were removed by the authors to conserve space...
    output [31:0] y8
);

// Internal wires to connect the modules
wire [31:0] intermediate_1, intermediate_2, intermediate_3, intermediate_4;
wire [31:0] intermediate_5, intermediate_6, intermediate_7, intermediate_8;
wire [31:0] twiddled_5, twiddled_6, twiddled_7, twiddled_8;

// Instantiate the Butterfly_Computation_Module
Butterfly_Computation_Module butterfly_computation(
    .clk(clk),
    .reset(reset),
    .x1(x1),
    //... the rest of the ports were removed by the authors to conserve space...
    .intermediate_8(intermediate_8)
);

// Instantiate the Twiddle_Factor_Multiplication_Module
Twiddle_Factor_Multiplication_Module twiddle_factor_multiplication(
    .clk(clk),
    .reset(reset),
    .intermediate_5(intermediate_5),
    .intermediate_6(intermediate_6),
    .intermediate_7(intermediate_7),
    .intermediate_8(intermediate_8),
    .twiddled_5(twiddled_5),
    .twiddled_6(twiddled_6),
    .twiddled_7(twiddled_7),
    .twiddled_8(twiddled_8)
);

fft_4_point fft_4_point_inst_1(
    .clk(clk),
    .reset(reset),
    .intermediate_1(intermediate_1),
    .intermediate_2(intermediate_2),
    .intermediate_3(intermediate_3),
    .intermediate_4(intermediate_4),
    .y1(y1),
    .y2(y2),
    .y3(y3),
    .y4(y4)
);

fft_4_point fft_4_point_inst_2(
    .clk(clk),
    .reset(reset),
    .intermediate_1(intermediate_1),
    .intermediate_2(intermediate_2),
    .intermediate_3(intermediate_3),
    .intermediate_4(intermediate_4),
    .y5(y5),
    .y6(y6),
    .y7(y7),
    .y8(y8)
);

endmodule
```

Listing 3. Twiddle Factor Multiplication Module