The Power of Large Language Models for Wireless Communication System Development: A Case Study on FPGA Platforms

Yuyang Du[a], Soung Chang Liew[a*], Kexin Chen[a], Yulin Shao[b]
[a] The Chinese University of Hong Kong, Hong Kong SAR, China
[b] University of Exeter, Exeter, U.K.

***Abstract*—Large language models (LLMs) have garnered significant attention across various research disciplines, including the wireless communication community. There have been several heated discussions on the intersection of LLMs and wireless technologies. While recent studies have demonstrated the ability of LLMs to generate hardware description language (HDL) code for simple computation tasks, developing wireless prototypes/products via HDL poses far greater challenges because of the more complex computation tasks involved. In this paper, we aim to address this challenge by investigating the role of LLMs in FPGA-based hardware development for advanced wireless signal processing. We begin by exploring LLM-assisted code refactoring, reuse, and validation, using an open-source software-defined radio (SDR) project as a case study. Through the case study, we find that an LLM assistant can potentially yield substantial productivity gains for researchers and developers. We then examine the feasibility of using LLMs to generate HDL code for advanced wireless signal processing, using the Fast Fourier Transform (FFT) algorithm as an example. This task presents two unique challenges: the scheduling of subtasks within the overall task and the multi-step thinking required to solve certain arithmetic problem within the task. To address these challenges, we employ in-context learning (ICL) and Chain-of-Thought (CoT) prompting techniques, culminating in the successful generation of a 64-point Verilog FFT module. Our results demonstrate the potential of LLMs for generalization and imitation, affirming their usefulness in writing HDL code for wireless communication systems. Overall, this work contributes to understanding the role of LLMs in wireless communication and motivates further exploration of their capabilities.**

***Index Terms*—FPGA, Verilog, large language models, wireless communication, prototype**

I. Introduction

The emergence of large language models (LLMs) has garnered significant attention within the research community. Scholars from diverse scientific disciplines are intrigued by LLMs due to their potential to transcend the realm of natural language processing (NLP). Researchers in the wireless communication field have also taken notice of this trend, leading to a series of insightful discussions in the community [1]–[3].

Recent studies indicated that LLMs possess the ability to generate hardware description language (HDL), such as Verilog, thereby painting a promising picture of LLMs aiding researchers and hardware engineers in the foreseeable future. The advancement of LLM-written HDL is particularly encouraging for the wireless communication community, given that many wireless products and software-defined radio (SDR) prototypes rely on field programmable gate arrays (FPGA) platforms that are programmed using HDL. However, carrying out a complex FPGA project is challenging. Prior works on LLM-assisted Verilog programming only reported simple hardware logics, such as those for shift register and multi-function calculators with addition/subtraction/multiplication/division (see Section II-C for a detailed analysis). These operations are only small atomic computational tasks, whereas the algorithms for signal processing blocks in wireless communication are far more complex. Knowing only the Verilog code for these atomic operations does not guarantee successful Verilog code for the higher-level signal processing blocks. This raises a critical question: Can LLMs assist in the development of FPGA projects involving intricate wireless signal-processing algorithms? This question remains open and calls for further investigation. This paper presents our endeavors in addressing this compelling issue, contributing to the growing discourse on the potential intersection between LLMs and wireless communication technologies.

This paper contributes to two pertinent topics. *In our first contribution*, we investigate the potential of LLMs as a valuable tool for wireless researchers engaged in FPGA-based prototype/product development. To this end, we examine an open-source, FPGA-based SDR project [4] as a case study. We thoroughly analyze the project's code and conduct comprehensive experiments to explore the extent to which an LLM can assist in implementing a wireless system. Our examination identifies three pivotal uses of LLMs: code refactoring, code reuse, and code validation. These uses of LLM, while seemingly mundane, are indispensable in hardware development and point towards the capacity of LLMs to substantially amplify researchers' productivity and expedite their research and development process.

In our second contribution, we delve into the possibility of employing LLMs to generate sophisticated HDL code for advanced signal processing algorithms required in wireless communication. To illustrate this, we focus on the Fast Fourier Transform (FFT) algorithm as an example. We emphasize that utilizing LLMs to implement FFT in HDL presents signif-

**Corresponding author: S. C. Liew (soung@ie.cuhk.edu.hk).*

icantly more formidable challenges compared to generating code in commonly used languages such as C, Python, or MATLAB (see Section IV-A for a detailed discussion).

The first challenge we encountered in using LLMs to generate HDL code for FFT is the scheduling of subtasks inside the FFT module. In general, a complex computation task like FFT can be broken down into a connection of smaller subtasks [5]. In high-level languages like C, compilers and operating systems (OSs) can schedule the processing of subtasks to hardware processors in a computer so that programmers do not need to worry about the scheduling issue. However, HDL programming requires designers to interface directly with the hardware without the guidance provided by compilers or OSs. This necessitates a meticulous consideration of parallelism and the precedence relationship between subtasks in HDL programming, a nuance that we have observed to be lacking in current LLMs like ChatGPT. Consequently, using similar prompts as in previous works [6]–[13] cannot yield a workable FFT for more than four points. To address the challenge, our prompt design applies in-context learning (ICL). This few-shot learning technique enables an LLM to rapidly learn from the additional examples we give it about the parallelism/precedence inside a small-scale FFT (say four-point or eight-point), on which it has not been previously trained.

The second challenge is the limited multi-step thinking ability of LLM. Some recent works [14]–[17] have reported that LLMs do not perform well when given a complex multi-step task, as they cannot decompose the problem as a human engineer can. In an FFT module, however, there are some complex processings that need to be decomposed into multiple steps, say calculating the twiddle factors and expressing them in the form of signed binary numbers. To augment the multi-step problem-solving ability of LLMs with that of human engineers, we exploit Chain-of-Thought (CoT) prompting. This method teaches LLMs how to approach complex multi-step problems in a way that mimics human thinking, enabling them to handle more complex tasks with greater accuracy and efficiency.

By incorporating the latest research outcomes in NLP (i.e., ICL and CoT prompting) into the FPGA implementation of complex wireless communication algorithms, we achieved a remarkable milestone in this paper: the successful generation of a 64-point Verilog FFT module using LLM. To the best of our knowledge, this is the first LLM-written complex HDL module ever reported in the field. More importantly, our explorations provide valuable insights into the understanding of LLMs:

1) LLMs demonstrate remarkable generalization abilities. They can realize sophisticated iterative wireless communication algorithms in HDL, provided that all ambiguities are effectively addressed during the instructional phase.
2) LLMs exhibit a strong ability to imitate. Once taught the problem-solving approach of a human, they can comprehend complex calculations.

These insights highlight the potential of LLMs and pave the way for leveraging LLMs to write HDL code for wireless communication building blocks.

II. Background and Related Works

A. Large Language Models (LLMs)

LLMs leverage the transformer architecture [18]. Early research like BERT [19] and GPT-2 [20] paved the way for today's boom. But it was the advent of GPT-3 [21] and its successors that brought the public's attention to the potential of these models. Today, the landscape is diverse, featuring numerous LLMs, with an array of options for both general and task-specific applications.

Despite their variations, all LLMs share core characteristics. First, they all serve as "scalable sequence prediction models" [22], meaning that they generate the "most probable" continuation of an input prompt. Second, LLMs operate on "tokens", which are commonplace character sequences specified through byte pair encoding. This method allows efficient data handling within the constraint of LLMs' fixed context size. By operating over tokens instead of characters, LLMs can process more text. For instance, in OpenAI's models, each token corresponds to roughly 4 characters, and the context windows can accommodate up to 8,000 tokens.

B. FPGA-based SDR Development in Wireless Communication

Software-defined radios (SDRs) provide a flexible and cost-effective solution to adapt equipment to the fast-evolving wireless communication standards and serve various research purposes. Instead of hardware-centric traditional radios, where different hardware is required to process different signals, SDRs allow the functionalities of a radio system to be defined and altered through software, making it possible to support multiple standards and applications with a single platform.

Thanks to its parallel processing capabilities, reprogrammability, and high performance, FPGA plays a pivotal role in SDR development. It provides an efficient platform for implementing complex, computation-intensive signal processing algorithms essential in SDRs. The application of FPGA-based SDRs applications span across various domains, from cellular networks, WiFi, and satellite communications to specialized underwater communication systems.

FPGA programming involves using an HDL to describe digital circuitry, ranging from simple combinational circuits to intricate sequential circuits and more complex systems-on-chip (SoCs). There are two standard HDLs for FPGA: VHDL and Verilog, with this paper using Verilog as an example.

One of the most widely recognized open-source FPGA-based SDR projects in recent years is OpenWiFi [4], which aims to provide a fully software-defined, reprogrammable WiFi networking solution via Verilog programming, and it has attracted much research interest. OpenWiFi runs on a high-performance Xilinx FPGA board, which provides ample hardware resources to implement the IEEE 802.11 standards.

C. LLM for Hardware Design

Recent research has shown a growing interest in harnessing the capabilities of LLMs to assist researchers and engineers in hardware design. In [6], the authors employed GitHub Copilot to scrutinize the incidence rates of six types of Verilog bugs. Following this, [7] and [8] investigated the potential for automated bug repairs with LLMs' assistance. This trend is not limited to academia, as industry players such as RapidSilicon are promoting their upcoming LLM-assisted tool for hardware design, called RapidGPT [9].

Beyond assisting researchers and engineers, recent studies also show interest in replacing human HDL programmers with LLMs. Initial efforts in this direction were documented in [10], where a fine-tuned GPT-2 model was trained with synthetically generated Verilog snippets. However, the limited generalization ability to unfamiliar tasks was a notable shortcoming of [10]. Subsequent research [11] expanded on this concept by investigating various strategies for fine-tuning Verilog-writing models. More recently, two studies delved into LLMs' applications in chip design. The former, Chip-Chat [12], employed the latest LLM to design an 8-bit shift register chip; while the latter, ChipGPT [13], focused on the power-performance-area (PPA) optimization of an LLM-composed chip design.

In contrast to the aforementioned works, this paper makes significant contributions from two key aspects.

In comparison to the first category of works [6]–[9], the first contribution of this paper involves using LLMs to assist FPGA development and offers a comprehensive study of the role of LLMs in facilitating the entire FPGA development process, beyond merely bug fixing. We investigate previously unexplored areas, including validation and maintenance issues, and our research includes the first LLM study that further refines a real-time communication system, OpenWiFi, that has already undergone rigorous validation and practical demonstrations previously.

In comparison to the second category of works [10]–[13], our second contribution focuses on utilizing LLMs to write HDL code specifically for wireless communication hardware, involving significantly more complex signal-processing algorithms than those addressed in prior research. Earlier LLM-written HDL codes, as presented in [10] and [11], were confined to simple signal processing no harder than undergraduate-level assignments, as acknowledged by the authors in their subsequent work [12]. Although [12] and [13] advanced the complexity by tackling an 8-bit shift register and a multi-functional calculator, these tasks still fall short in complexity compared to communication algorithms such as FFT. As a result of tackling more demanding coding tasks, we encountered two challenges that had not been previously reported in prior works: the subtask scheduling challenge and the multi-step thinking challenge. Addressing these challenges necessitates the use of advanced methods, namely in-context learning (ICL) and chain-of-thought (CoT) prompting, which have never been considered in prior works [10]–[13].

III. Using LLMs to Assist FPGA Development

This section delves into the various tasks frequently encountered when prototyping wireless systems on FPGAs. Our contribution lies in the proposition of utilizing LLMs to amplify implementation efficiency and productivity in the realm of wireless communication research. By refining OpenWiFi [4], a well-known open-source FPGA-based SDR project, we not only provide valuable insights and practical experiences but also pave the way for AI-assisted SDR development on FPGA. The advantages of harnessing LLMs in hardware development are prominently demonstrated across three pivotal dimensions: 1) Code Refactoring, 2) Code Reuse, and 3) Code Validation.

A. Code Refactoring

Improving code quality is crucial in FPGA design, as even functioning code may still benefit from further enhancements [23]. Code refactoring is a routine task for FPGA engineers, involving manual review and editing. Recent research works suggest that artificial intelligence (AI) can assist in scanning and revising code [24]. This subsection demonstrates the competence of LLMs in code refactoring and shows how LLMs can offer valuable assistance to engineers in this kind of work. To illustrate this, we choose a simple signal delay module from OpenWiFi and showcase how an LLM can improve the code in terms of readability, efficiency, and reliability. Fig. 1 presents the original code, while Fig. 2 presents our prompt for ChatGPT.

```verilog
module delayT
#(
    parameter DATA_WIDTH = 32,
    parameter DELAY = 1
)
(
    input clock,
    input reset,

    input [DATA_WIDTH-1:0] data_in,
    output [DATA_WIDTH-1:0] data_out
);

reg [DATA_WIDTH-1:0] ram[DELAY-1:0];
integer i;

assign data_out = ram[DELAY-1];

always @(posedge clock) begin
    if (reset) begin
        for (i = 0; i < DELAY; i = i+1) begin
            ram[i] <= 0;
        end
    end else begin
        ram[0] <= data_in;
        for (i = 1; i < DELAY; i= i+1) begin
            ram[i] <= ram[i-1];
        end
    end
end

endmodule
```

Figure 1: The original code of a signal delay module in OpenWiFi.

```
// I am writing a signal delay module for my FPGA project.
// The module is written in Verilog. Could you please help me to
// revise the code to make it easier to understand and error-free.
// Besides providing me the revised code, I also want you
// to highlight your changes and explain the reason why you
// made such modifications. Here is the code:
```

Figure 2: The prompt we used to improve the signal delay module.

The first objective of code refactoring is to improve its *readability*. A Verilog project that is easy to read and understand facilitates easier future maintenance. Key characteristics of well-written code include consistent programming style, meaningful module/variable names, and sufficient comments that clearly explain code functionality. However, in practice, code contributors collaborating on a project may have their own programming styles. Despite widely accepted programming standards and additional coding requirements within a development team, poorly written code can still present challenges in terms of readability and maintainability.

LLMs offer a consistent programming style, thus aiding in unifying the code within the same FPGA project written by diverse coders. Additionally, LLMs possess the intelligence to address naming and commenting issues effectively. In our experiments, ChatGPT provides readability suggestions from three distinct perspectives. First, ChatGPT recommends using meaningful names for modules and variables. For instance, in Line 1, the original module name "DelayT" can be replaced with "DelayBuffer" for clarity. Similarly, in Line 15, variable "i" can be renamed to "index" for improved understanding. Second, ChatGPT identifies redundant code within the "always" block and suggests shortening it to enhance readability. The revised code, presented in Lines 17 to 21 of Fig. 3, demonstrates the reduction in code length while maintaining readability. Lastly, ChatGPT automatically adds detailed comments to assist readers in better comprehending the code. These comments are also displayed in Fig. 3.

```
1  module DelayBuffer
2  #(
3      // Module parameters, can be overridden when instantiating the module.
4      parameter DATA_WIDTH = 32,  // Width of data to be delayed
5      parameter DELAY = 1         // Delay in clock cycles
6  )
7  (
8      input wire clock,   // The clock signal
9      input wire reset_n, // Reset signal; when low, clears the delay line
10     input wire [DATA_WIDTH-1:0] data_in,      // Input data to be delayed
11     output wire [DATA_WIDTH-1:0] data_out // Output of the delayed data
12 );
13
14   reg [DATA_WIDTH-1:0] ram [0:DELAY-1];
15
16   always @(posedge clock or negedge reset_n) begin
17         if (reset) begin
18             ram <= 0;  // reset all at once
19         end else begin
20             ram <= {ram[DELAY-2:0], data_in};  // shift in new data
21         end
22     end
23
24   assign data_out = ram[DELAY-1];  // output the last in the line
25
26 endmodule
```

Figure 3: The revised code generated by ChatGPT.

The second objective of code refactoring is to enhance *efficiency*. Efficiency in hardware language differs significantly from that in software languages like C or Python. In hardware projects, designers must consider the physical implementation of the hardware within the chip after synthesis. For example, as highlighted by ChatGPT, the code within the "always" block is realized as a shift register in the FPGA. However, this implementation would be wasteful of resources if the function of the IP core is configured to delay the signal for a relatively long time, say more than 10 clock periods. A more efficient approach to implement delay function is using a read/write counter and a block RAM (mature RAM IP cores are readily available). Fig. 3 does not revise the code based on this suggestion, as it would involve a fundamental redesign of the module. However, we consider this comment to be of significant importance for achieving highly efficient hardware processing in OpenWiFi, especially considering that the delay module is frequently reused in their designs.

The third objective of code refactoring is to enhance *reliability*. Some code may appear to work well in a design simply because the bugs within it are not triggered. For instance, a flawed design may cause problems when the operating voltage or clock frequency is high. Additionally, advanced HDL design tools may automatically correct bugs during the synthesis stage or the place-and-route stage. However, these design tools could not revise the code itself, leaving underlying problems unresolved (although they may not manifest in the final output). Identifying and addressing such problems can be challenging as they produce the correct output at the moment but may cause trouble if triggered in the future, particularly when reusing the code on a different hardware platform with a higher operating voltage or if the HDL design tool changes. Although no one can guarantee bug-free code, and such issues are common in practice (sometimes referred to as "features" rather than "bugs"), avoiding such mistakes during the coding stage is crucial for reliable hardware design.

In light of the Verilog code and the prompt provided, ChatGPT highlights two severe problems that could lead to potential system instability. First, ChatGPT suggests adding "wire" data type specifications to the "input" and "output" ports. We consider this comment valuable. Although a Verilog synthesizer can infer the data type of these ports and apply default settings, it is good practice to explicitly state them to be sure. Second, ChatGPT recommends including the negative edge of active-low reset signals (i.e., reset_n) in the sensitivity list of the "always" block. This modification offers two advantages: 1) a digital circuitry utilizing an active-low reset signal is less likely to being erroneously triggered by noise compared to those employing an active-high reset signal [25]; 2) asynchronous reset is more reliable because the system can respond immediately upon detecting an error, without waiting for the rising edge of the clock signal [26]. Based on the authors' experience in the IC industry, "asynchronous active-low reset" is a widely adopted programming standard. And we believe this is a crucial issue overlooked by the developers of OpenWiFi.

In conclusion, through this and similar exercises, we have gained substantial confidence in asserting that LLMs can serve as valuable assistants in improving and refactoring Verilog codes.

B. Code Reuse

Reusing mature designs is a common approach for efficient FPGA development. Highly configurable code allows for easy reuse by simple parameter adjustments. For example, if we apply the default settings for the Verilog module presented in Fig. 3, it can delay a 32-bit signal by one clock period. However, if we intend to use the same code for a 64-bit signal and require a delay of four clock periods, we simply need to set the parameters DATA_WIDTH and DELAY to 64 and 4, respectively.

In realistic engineering scenarios, it is true that not all codes are written in a parameterized manner. This can lead to a large workload when attempting to customize the code for specific requirements. For instance, Fig. 4 shows the complex multiplier module used in OpenWiFi, where the code is specifically designed for input signals with a 16-bit data width. If there is a need to enhance the calculation precision to 32 bits for better signal processing accuracy, it is necessary to invest time to rewrite the code. This process can be tedious and time-consuming, requiring careful modifications and adjustments.

```verilog
module complex_mult
(
    input clock,
    input enable,
    input reset,
    input [15:0] a_i,
    input [15:0] a_q,
    input [15:0] b_i,
    input [15:0] b_q,
    input input_strobe,
    output reg [31:0] p_i,
    output reg [31:0] p_q,
    output output_strobe
);
  reg [15:0] ar, ai, br, bi;
  wire [31:0] prod_i, prod_q;
  wire [31:0] s_axis_a_tdata = {ai,ar} ;
  wire [31:0] s_axis_b_tdata = {bi, br} ;
  wire [63:0] m_axis_dout_tdata;
  assign prod_q = m_axis_dout_tdata[63:32];
  assign prod_i = m_axis_dout_tdata[31:0];
  complex_multiplier mult_inst (
    .aclk(clock),
    .s_axis_a_tvalid(input_strobe),
    .s_axis_a_tdata(s_axis_a_tdata),
    .s_axis_b_tvalid(input_strobe),
    .s_axis_b_tdata(s_axis_b_tdata),
    .m_axis_dout_tdata(m_axis_dout_tdata)
  );
  delayT #(.DATA_WIDTH(1), .DELAY(5)) stb_delay_inst (
      .clock(clock),
      .reset(reset),
      .data_in(input_strobe),
      .data_out(output_strobe)
  );
  always @(posedge clock) begin
      if (reset) begin
          ar <= 0; ai <= 0; br <= 0; bi <= 0; p_i <= 0; p_q <= 0;
      end else if (enable) begin
          ar <= a_i; ai <= a_q; br <= b_i; bi <= b_q;
          p_i <= prod_i; p_q <= prod_q;
      end
  end
endmodule
```

Figure 4: The original Verilog code for the complex multiplier in OpenWiFi.

We found that LLMs can provide significant assistance in such tasks. In Fig. 5 below, we present the prompt used for the parameterization job, and in Fig. 6, we show the new code generated by ChatGPT. As seen, ChatGPT successfully parameterizes the code while maintaining its correctness. The revised code is versatile and capable of accommodating diverse data widths and latency requirements by adjusting the module's parameter settings.

```
// I am writting a Verilog module for my FPGA system. I have a complex
// multiplier that deals with 16-bit input signals. However, I want to
// re-write the code in a parameterized manner. I want you to use the
// parameter funtion in Verilog language to make the data width of a_i,
// a_q, b_i, b_q, p_i, p_q become configurable. Here is the code: ...
```

Figure 5: The prompt we used to parameterize the complex multiplier.

```verilog
module complex_mult #(parameter DATA_WIDTH = 16, parameter PRODUCT_WIDTH = 32)
(
    input clock,
    input enable,
    input reset,
    input [DATA_WIDTH-1:0] a_i,
    input [DATA_WIDTH-1:0] a_q,
    input [DATA_WIDTH-1:0] b_i,
    input [DATA_WIDTH-1:0] b_q,
    input input_strobe,
    output reg [PRODUCT_WIDTH-1:0] p_i,
    output reg [PRODUCT_WIDTH-1:0] p_q,
    output output_strobe
);
    reg [DATA_WIDTH-1:0] ar, ai, br, bi;
    wire [PRODUCT_WIDTH-1:0] prod_i, prod_q;
    wire [2*DATA_WIDTH-1:0] s_axis_a_tdata = {ai,ar} ;
    wire [2*DATA_WIDTH-1:0] s_axis_b_tdata = {bi, br} ;
    wire [2*PRODUCT_WIDTH-1:0] m_axis_dout_tdata;
    assign prod_q = m_axis_dout_tdata[2*PRODUCT_WIDTH-1:PRODUCT_WIDTH];
    assign prod_i = m_axis_dout_tdata[PRODUCT_WIDTH-1:0];
    complex_multiplier #(2*DATA_WIDTH, 2*PRODUCT_WIDTH) mult_inst (
      .aclk(clock),
      .s_axis_a_tvalid(input_strobe),
      .s_axis_a_tdata(s_axis_a_tdata),
      .s_axis_b_tvalid(input_strobe),
      .s_axis_b_tdata(s_axis_b_tdata),
      .m_axis_dout_tdata(m_axis_dout_tdata)
    );
    delayT #(.DATA_WIDTH(1), .DELAY(5)) stb_delay_inst (
        .clock(clock),
        .reset(reset),
        .data_in(input_strobe),
        .data_out(output_strobe)
    );
    always @(posedge clock) begin
        if (reset) begin
            ar <= 0; ai <= 0; br <= 0; bi <= 0; p_i <= 0; p_q <= 0;
        end else if (enable) begin
            ar <= a_i; ai <= a_q; br <= b_i; bi <= b_q;
            p_i <= prod_i; p_q <= prod_q;
        end
    end
endmodule
```

Figure 6: The revised Verilog code for the complex multiplier.

By applying this method to modify the codes in OpenWiFi, we can enhance its user-friendliness for reuse and extensions. This approach facilitates easier customization and adaptation of the project to different specifications and requirements.

C. Code Validation

In FPGA development, code validation is a routine task that engineers undertake to ensure the correctness of the code. This typically involves writing a testbench and attempting to cover a wide range of input possibilities. This subsection points out a potential shift in the future: we may not need to invest extensive time in testbench development, as LLMs can assist in generating rigorous testbenches with comprehensive input coverage.

Fig. 7 presents the prompt we used to generate the testbench for the complex multiplier given in Fig. 6. The testbench code generated by ChatGPT is shown in Fig. 8. It is evident from the code that ChatGPT produces a well-structured testbench, incorporating all the necessary elements. We test the revised complex multiplier with the generated testbench, and the result indicates that both the revised module and the testbench are error-free.

```
// Could you please write a testbench for the code you wrote.
// I want you to use a 32 bit data width.
// I  want you to test at least ten possible input in your testbench.
```

Figure 7: The prompt we used to generate a testbench for the revised complex multiplier.

```verilog
`timescale 1ns / 1ps

module tb_complex_mult;
    reg clock;
    reg enable;
    reg reset;
    reg [31:0] a_i, a_q, b_i, b_q;
    reg input_strobe;
    wire [63:0] p_i, p_q;
    wire output_strobe;
// Instantiate the complex_mult module
    complex_mult #(32, 64) u1 (
        .clock(clock),
        .enable(enable),
        .reset(reset),
        .a_i(a_i),
        .a_q(a_q),
        .b_i(b_i),
        .b_q(b_q),
        .input_strobe(input_strobe),
        .p_i(p_i),
        .p_q(p_q),
        .output_strobe(output_strobe)
    );
// Generate a clock signal
    always #5 clock = ~clock;
// Test sequence
    initial begin
        // Reset and initialize the module
        clock = 0;
        reset = 1; enable = 0; input_strobe = 0;
        a_i = 0; a_q = 0; b_i = 0; b_q = 0;
        #10;
        reset = 0;
        enable = 1;
        // Apply test inputs
        #10; a_i = 10; a_q = 20; b_i = 30; b_q = 40; input_strobe = 1;
        #10; a_i = 50; a_q = 60; b_i = 70; b_q = 80; input_strobe = 1;
        #10; a_i = 90; a_q = 100; b_i = 110; b_q = 120; input_strobe = 1;
        #10; a_i = 130; a_q = 140; b_i = 150; b_q = 160; input_strobe = 1;
        #10; a_i = 170; a_q = 180; b_i = 190; b_q = 200; input_strobe = 1;
        #10; a_i = 210; a_q = 220; b_i = 230; b_q = 240; input_strobe = 1;
        #10; a_i = 250; a_q = 260; b_i = 270; b_q = 280; input_strobe = 1;
        #10; a_i = 290; a_q = 300; b_i = 310; b_q = 320; input_strobe = 1;
        #10; a_i = 330; a_q = 340; b_i = 350; b_q = 360; input_strobe = 1;
        #10; a_i = 370; a_q = 380; b_i = 390; b_q = 400; input_strobe = 1;
        // Finish the test
        $finish;
    end

endmodule
```

Figure 8: The testbench generated by ChatGPT.

Additionally, ChatGPT can also provide the expected outputs for each input it generated, which can further assist in the code validation process. For example, with the prompt presented in Fig. 9, we obtain more potential inputs for the testbench, and ChatGPT also outputs the corresponding calculation result for each input. This greatly simplifies the validation process.

```
// Please give me more possible input for the testbench. You don't need to
// re-write the testbench, just give me the input. Furthermore, I want
// you to tell me the expected output for each possible input.
```

Figure 9: The prompt we used to generate more possible input for the testbench.

IV. Challenges in Implementing Complex Signal-processing Algorithm in Verilog

Although previous research [10]–[13] has demonstrated LLMs' ability in generating basic hardware modules, such as shift registers or dice rollers (as discussed in Section II), employing LLMs to generate HDL code for advanced signal-processing algorithms remains unexplored. To bridge this gap, the following two sections present our efforts in pushing the knowledge boundary of LLMs. Specifically, we delve into utilizing LLMs to generate complex HDL code for advanced wireless communication algorithms, going beyond the simple code refinement task discussed in Section III. As a case study, we focus on FFT, a complex yet vital component in wireless communication hardware.

In this section, we highlight two challenges when employing LLMs to generate Verilog code for the FFT module, namely the "subtask scheduling" problem and the "multi-step thinking" problem. In the next section, we present our approaches to address these challenges through the utilization of in-context learning (ICL) and Chain-of-Thought (CoT) prompting techniques. The analysis in these two sections provides valuable insights to effectively leverage LLMs for generating complex HDL code specifically tailored for *iterative* signal-processing algorithms like FFT.

A. Verilog Code Generated by ChatGPT

Let us begin with a conversation with ChatGPT. In our experiments, we used the prompt given in Fig. 10 for multiple tries and found that ChatGPT was unable to generate the code for a 64-point FFT module. Specifically, the AI either provided a general framework for the FFT module, requiring additional manual input to complete the specific code, or offered an implementation limited to a trivial two-point FFT. We invite readers to personally engage with this exercise, as it offers a firsthand understanding of the current capabilities of ChatGPT in this particular application.

```
//Could you please help me to write an FFT module for my FPGA system.
// I want the module written in Verilog language. And here are
// details of my specification:
// 1. The FFT unit should have a length of 64-point.
// 2. Input signal should be a complex number with 16 bits
//    real part and 16 bits imaginary part
// 3. The output of the FFT should be natural order
// 4. The target input clock frequency is 200MHz
// I also provide you with the instantiation templet as follows.
//   fft_GPT_Generate your_instance_name (
//    .clk(aclk),
//    .resetn(resetn),
//    .enable(enable),
//    .FFI_input(FFI_input),
//    .input_valid(input_valid),
//    .FFT_output(FFT_output),
//    .output_valid(output_valid)
//  );
```

Figure 10: The prompt we used to generate a 64-point FFT.

Upon the failure, we scaled down the complexity of the task to generating an eight-point FFT module instead of a 64-point FFT module. ChatGPT managed to create code that seemed correct and professional (as depicted in Fig. 11). However, despite its polished appearance, the generated code failed to function as expected and did not pass the FFT testbench.

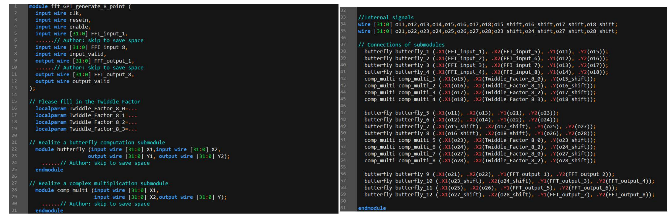


Figure 11: The eight-point FFT code produced by ChatGPT. To conserve space, we omit portions of the code that are highly similar or repetitive (refer to our comments on Line 6, Line 10, Line 24, and Line 30 for more details).

B. Challenges One: The Subtask Scheduling Problem

The first issue we identify in the generated code is that ChatGPT lacks an essential understanding of task scheduling and sequential control.

In general, complex computational tasks can be broken down into a series of simpler subtasks, among which parallel and precedence relationships often exist. For example, in the code provided in Fig. 11, we observe that the FFT computation can be decomposed into numerous butterfly computations and complex multiplications (as depicted from Line 38 to 59). Some subtasks, such as butterfly computations from the same FFT stage, can be executed concurrently. However, other tasks, like butterfly computations of successive stages, have a precedence relationship and cannot be parallelized.

To better elucidate this precedence relationship, we present the flow graph of an eight-point FFT in Fig. 12. Here, it is clear that the output of the orange/blue butterfly computation (in the first FFT stage) serves as the input for the red butterfly computation (in the second FFT stage). We cannot execute the red task until the blue and orange tasks have been completed, indicating a precedence relationship between the red task and the blue/orange task. For a more detailed explanation and formal definition of the precedence relationships among butterfly computations, we refer readers to Section II of a related paper [5].

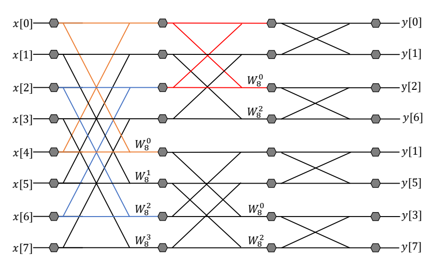


Figure 12: The flow graph of an eight-point FFT.

Bearing the above discussion in mind, we note that the precedence relationships among butterfly computations necessitate careful subtask scheduling when implementing FFT in Verilog. Unlike high-level programming languages, which have the assistance of compilers and OSs to manage subtask's scheduling[1], Verilog faces hardware directly. Therefore, designers need to consider the scheduling issue themselves in the HDL code so that these subtasks can be executed in a sequential manner.

Over the past few decades, FFT implementation has been extensively studied and several classic scheduling schemes have emerged [28]–[30]. The simplest approach for precise subtask execution control involves the use of enable signals and output-state-indicating signals. Specifically, if the execution of subtask A depends on the completion of subtask B, we can connect the output-state-indicating signal of B with the enable signal of A to manage their execution. Upon the completion of B, its output indicating signal becomes valid, which subsequently triggers the execution of A. The output-state-indicating signal is sometimes referred to as the "done" signal, as it becomes valid only when the associated subtask is fully executed.

We now look back to the code generated by ChatGPT. It is apparent that there is no task execution control in the implementation. Neither the basic method of using enable/done signals nor more advanced methods like state machines [31] are observed in the code. To confirm our observation, we validated the code using our testbench and found that many subsequent subtasks were prematurely executed before the outputs of their preceding tasks became valid. This resulted in erroneous outputs at the final stage. The experiment results corroborate our initial assertion: ChatGPT, in its current state, lacks awareness of subtask scheduling and sequential execution control. Therefore, it is incapable of generating a viable FFT module autonomously.

In Section V-A, we detail our approach to enabling ChatGPT to comprehend the concept of precedence relationships among subtasks and subsequently implement execution control using enable/done signals.

C. Challenges Two: The Multi-step Thinking Problem

The second issue we identified with the LLM-generated code is the inability of ChatGPT to correctly generate the twiddle factors, a crucial component in FFT calculations (refer to lines 16 to 19 in Fig. 11). This issue persisted regardless of the number of attempts or variations in the prompts we used.

Before diving into why ChatGPT is unable to generate these factors, it is necessary to provide a detailed understanding for the concept of twiddle factors. As observed in Fig. 12), the data in the course of the algorithm is multiplied by trigonometric constant coefficients, denoted as $W_N^k = e^{-j(2\pi k/N)}$, where $N = 8$ is the size of FFT, and index $k \in \{0, 1, ...N/2\}$. These coefficients are referred to as the twiddle factors.

[1]The scheduling issue can be less complex for high-level programming languages like C or python, as the software compilers and the OS can help to handle the scheduling problem. These tools can distinguish the parallel/precedence relationship and assign the butterfly computations to hardware processers accordingly, and users can just describe their algorithm without too much worry about the scheduling issue. For more details about how to implement FFT in C and what a compiler/OS can help in the implementation, we refer the reader to the documentation of FFTW, a high-performance FFT library [27].

In theory, the real and imaginary parts of W_N^k are numbers no larger than one. In hardware processing, however, things are different because digital circuitry is designed to handle integers expressed in binary form. Here, we illustrate how a human engineer would transform the complex number W_N^k into a 32-bit binary sequence, with a 16-bit imaginary part and a 16-bit real part, using W_8^1 as an example:

1) Step One (calculation): we have $W_8^1 = e^{-j(\pi/4\ )} = 0.7071 - 0.7071i$ form trigonometric calculations.
2) Step Two (scaling): we scale the real and imaginary parts of W_8^1 by multiplying them by a scaling factor, typically chosen as the maximum value that can be represented by the number of bits allocated for each part (in this case, 16 bits). Hence, we amplify $\mathrm{Re}\left(W_8^1\right)$ and $\mathrm{Im}\left(W_8^1\right)$ by $2^{15} - 1$. Now, we have $\mathrm{Re}\left(W_8^1\right) = 23169.5457$ and $\mathrm{Im}\left(W_8^1\right) = -23169.5457$.
3) Step Three (rounding): we do rounding operation on $\mathrm{Re}\left(W_8^1\right)$ and $\mathrm{Im}\left(W_8^1\right)$, and now we have $\mathrm{Re}\left(W_8^1\right) \approx 23170$ and $\mathrm{Im}\left(W_8^1\right) \approx -23170$.
4) Step Four (Conversion to binary): we convert $\mathrm{Re}\left(W_8^1\right)$ and $\mathrm{Im}\left(W_8^1\right)$ to their binary representations, which is "0101,1010,1000,0010" and "1010,0101,0111,1110", respectively.
5) Step Five (Concatenation): we concatenate the binary representations of the scaled real and imaginary parts to form a 32-bit binary sequence, with the higher 16 bits being the imaginary part and the lower 16 bits being the real part. We can now represent W_8^1 by "1010,0101,0111,1110, 0101,1010,1000,0010".

By following the above five steps, a human engineer can transform W_N^k into a 32-bit binary sequence suitable for hardware processing in digital circuitry. One more thing we note is that, when employing the 32-bit sequence for complex multiplication, we need to shrink the multiplication output appropriately to maintain accuracy, as we have amplified W_N^k in Step Two.

From the previous discussion, it becomes evident that generating the twiddle factors is not a straightforward process. It involves five different steps. Although the logical reasoning required for each individual step might not pose a significant challenge for ChatGPT, the entire problem becomes very difficult for the AI model, as it lacks the ability to decompose the problem into intermediate steps as a human engineer would do. This limitation, known as the lack of multi-step thinking ability, has also been observed in recent research within the NLP community [15], [16]. A number of studies have been carried out to enhance the capabilities of large language models like ChatGPT by aiding them in emulating human-like multi-step reasoning processes [14], [17]. This line of research aims to help AI overcome complex problems that require intermediate steps for solution.

Given the analysis above, we have identified the underlying reason why ChatGPT could not generate the twiddle factors in our initial trials. In Section V-B, we will further discuss our approach to addressing this "multi-step thinking" problem and making ChatGPT able to perform our task.

V. Solving implementation challenges via ICL and CoT Prompting

A. *In-context Learning (ICL) for Challenge One*

A brief introduction about ICL

Let us first briefly introduce the concept of ICL. The concept of ICL was popularized in [21], which introduced how to enable GPT-3 to learn from a few examples. In ICL, we give an LLM a prompt containing several question-answer pairs as examples to demonstrate how to complete a task. Following these pairs, a new, unaddressed question is appended to the prompt. The aim is for the LLM to analyze the previously given examples, extrapolate the underlying task, and provide an answer to this new question based on that learning context.

In Fig. 13, we give an example prompt for using LLMs in a news classification task. As input examples, we provide several news titles and their corresponding topic classifications, creating a series of question-answer pairs. We then present the LLM with a news title for which it must generate the relevant topic classification. To correctly answer this question, the model must analyze the provided examples to understand several aspects of the problem: the structure of the input (news titles), the range of possible outputs (possible news topics), the mapping from input to output (topic classification), and the formatting of the output (a single word with the first letter capitalized). With this understanding, ChatGPT generates the correct answer, i.e., “Technology”.

```
// I want you to help me with topic classification task for newspaper.
// I will give you some examples and you should learn from them.
// In each example, I will give you the title of a piece of news,
// and I will  tell you the correct topic classification for the news.

// Here are four examples.
// Title 1: "CPI in HK averaged 65.72 points from 1980 until 2023."
// Answer 1: Finance
// Title 2: "Jonas Vingegaard won le tour de france in 2022."
// Answer 2: Sport
// Title 3: "Apple developed a new technilogy called Apple Vision Pro."
// Answer 3: Technology
// Title 4: "Biden formally announces bid for second term as US president."
// Answer 4: Politics

// Now, I give you another piece of news, tell me its classification
// New question: Qualcomm Announces Multi-Year Collaboration with
//               Sony to Deliver Next Generation Smartphones.
```

Figure 13: An “new classification” example for ICL.

ICL distinguishes itself from conventional machine learning algorithms in several key ways [32], [33]. Most notably, it does not require any parameter optimization or the addition of new parameters to the model. ICL works effectively with only a handful of training examples to get an LLM operational on a new topic, and its natural language interface is intuitive, even for beginners.

There have been recent research efforts aiming to decipher why ICL performs so remarkably well. The prevailing theory is that an LLM can more effectively “locate” a previously learned concept with the assistance of ICL. Specifically, since an LLM is trained on a vast amount of text encompassing a wide range of topics and formats, it can model a diverse array of learned concepts with knowledge from various domains. An LLM can deliver better results if we assist it in selecting the most suitable domain knowledge with the hints provided in our ICL examples. For instance, in this paper, our task necessitates greater domain knowledge in HDL, as opposed to languages like C or Python. For a more comprehensive understanding of the underlying mechanisms that make ICL effective, we refer the reader to [34], [35].

ICL for our Verilog-writing task

We now demonstrate how we use ICL to build the 64-point FFT module in Verilog. We start by re-shaping the FFT flow graph in an iterative manner for ChatGPT’s easier understanding and imitation. In a typical FFT flow graph, such as the one presented in Fig. 12, an N-point FFT has $\log_2 N$ stages. In essence, the signal processing in the subsequent $\log_2 N - 1$ stages can be perceived as two parallel $N/2$-point FFT processes. Therefore, as illustrated in Fig. 14, using an eight-point FFT example, we can simplify the flow graph into two stages: the first stage consists of $N/2$ butterfly computations and $N/2$ complex multiplications, we term these two substages as stage 1-A and stage 1-B, respectively. The second stage encompasses two parallel $N/2$-point FFT processes. With the new iterative flow graph, we simplify the understanding of FFT and aid in the better comprehension of LLMs.

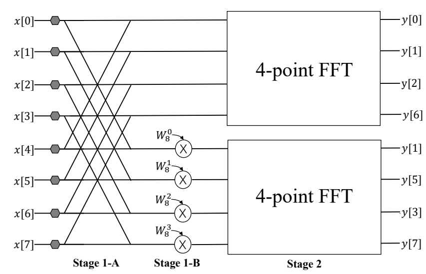


Figure 14: The iterative flow graph of an eight-point FFT.

We then analyze the precedence/parallel relationship within the iterative FFT flow graph. It is important to note that, beyond the structure of the flow graph, the available hardware resources can also influence these relationships. For instance, if an FPGA has limited hardware resources, to the point that it can only execute one butterfly computation at a time, the butterfly computations within the same stage (such as the four butterfly computations in Stage 1-A of Fig. 14) would have a precedence relationship. This is because they must be executed sequentially, i.e., one after another. On the other hand, if the FPGA holds abundant hardware resources, the butterfly computations within the same stage can be executed in a fully parallel manner. In this paper, we consider a scenario where the FPGA has ample hardware resources so that the precedence/parallelism relationships are solely determined by the flow graph itself. With this assumption, we characterize the precedence/parallelism relationship of an N-point FFT as follows:

1) **Stage 1-A**: The $N/2$ butterfly computations with this stage can be executed in parallel. These computations can be processed simultaneously when triggered by the external enable signal.
2) **Stage 1-B**: The $N/2$ complex multiplications within this stage can be executed in parallel, but their executions are

triggered by the completeness of butterfly computations in Stage 1-A.

3) **Stage 2** The two $N/2$-point FFT in Stage 2 within this stage can be executed in parallel, but their executions are triggered by the completeness of complex multiplications in Stage 1-B.

With the simplified precedence/parallel relationships and the iterative FFT flow graph discussed above, we now demonstrate how we generate our question-answer pairs and conduct ICL with the goal of creating a 64-point FFT module using ChatGPT.

In the first step, we use ChatGPT to generate two simple IP cores that will be frequently used in the subsequent FFT implementation: the butterfly computation IP core and the complex multiplication IP core. Fig. 15 elucidates the prompt specifically devised for this task, while Fig. 16 showcases the code generated by ChatGPT.

```
// I am working on an FPGA project written in Verilog language.
// I want you to write two important IP cores for me.

// The first IP core is for butterfly computation in FFT processing.
// You need to implement the basic butterfly processing in this IP core.
// Here is a template of the IP core, you need to understand the
// input/output from the template and realize the IP core accordingly.

// butterfly_computation your_inst_name(
//     .clk(clk),          // input, 1 bit, clock signal
//     .resetn(resetn),    // input, 1 bit, low-active reset signal
//     .enable(enable),   // input, 1 bit, enable signal
//     .in1(in1),         // input, 16 bits imag part plus 16 bits real part,
//     .in2(in2),         // input, 16 bits imag part plus 16 bits real part
//     .done(done),     // output, set high when the computation is complete
//     .out1(out1)        // output, 16 bits imag part plus 16 bits real part
//     .out2(out2)        // output, 16 bits imag part plus 16 bits real part
// );

// The second IP core is for complex multiplication in FFT processing.
// I will use it to multiply the output of a butterfly computation with
// the twiddle factor provided. Please note that the twiddle factor
// is amplified by approximately 2^15, you need to shrink the out
// accordingly
// Here is a template of the IP core, you need to understand the
// input/output from the template and realize the IP core accordingly.

// butterfly_computation your_inst_name(
//     .clk(clk),          // input, 1 bit, clock signal
//     .resetn(resetn),    // input, 1 bit, low-active reset signal
//     .enable(enable),   // input, 1 bit, enable signal
//     .in1(in1),         // input, 16 bits imag part plus 16 bits real part,
//     .in2(in2),         // input, 16 bits imag part plus 16 bits real part
//     .done(done),     // output, set high when the computation is complete
//     .out(out)        // output, a 32 bit production output
// );
```

Figure 15: The prompt we used to generate the butterfly computation IP core and the complex multiplication IP core.

```
module butterfly_computation (
  input wire clk,
  input wire resetn,
  input wire enable,
  input wire [31:0] in1,
  input wire [31:0] in2,
  output reg done,
  output wire [31:0] out1,
  output wire [31:0] out2
);
// Separate the real and imag parts for further computation
wire [15:0] in1_real = in1[15:0];
wire [15:0] in1_imag = in1[31:16];
wire [15:0] in2_real = in2[15:0];
wire [15:0] in2_imag = in2[31:16];
// Assign the output
reg [15:0] temp1_real, temp1_imag;
reg [15:0] temp2_real, temp2_imag;
// Always block for assignment
always @(posedge clk or negedge resetn) begin
  if (~resetn) begin
    done <= 1'b0;
    temp1_real <= 16'd0; temp1_imag <= 16'd0;
    temp2_real <= 16'd0; temp2_imag <= 16'd0;
  end
  else if (enable) begin
    // Butterfly computation
    temp_real1 <= in1_real + in2_real;
    temp_imag1 <= in1_imag + in2_imag;
    temp_real2 <= in1_real - in2_real;
    temp_imag2 <= in1_imag - in2_imag;
    done <= 1'b1;
  end
  else
    done <= 1'b0;
end
assign out1 = {temp1_imag, temp1_real};
assign out2 = {temp2_imag, temp2_real};
endmodule
```

```
module complex_multiply (
  input wire clk,
  input wire resetn,
  input wire enable,
  input wire [31:0] in1,
  input wire [31:0] in2,
  output reg done,
  output wire [31:0] out
);
// Separate the real and imag parts for further computation
wire [15:0] in1_real = in1[15:0];
wire [15:0] in1_imag = in1[31:16];
wire [15:0] in2_real = in2[15:0];
wire [15:0] in2_imag = in2[31:16];
// Registers for storing intermediate products
reg [31:0] real_product;
reg [31:0] imag_product;
// Always block to assign intermediate products
always @(posedge clk or negedge resetn) begin
  if (~resetn) begin
    done <= 1'b0;
    real_product <= 32'd0;
    imag_product <= 32'd0;
  end
  else if (enable) begin
    // Shrink real_product and imag_product by 2^15
    real_product <= ((in1_real * in2_real) - (in1_imag * in2_imag)) >>> 15;
    imag_product <= ((in1_real * in2_imag) + (in1_imag * in2_real)) >>> 15;
    done <= 1'b1;
  end
  else
    done <= 1'b0;
end
assign out[15:0] = real_product;
assign out[31:16] = imag_product;
endmodule
```

Figure 16: The Verilog code generated by GPT upon the prompt.

In the second step, we give the first question-example pair. The example question, as depicted in Fig. 17a, asks the LLM to generate a four-point FFT IP core, building upon the provided two-point FFT (which is identical to the butterfly computation IP core). Our example answer, as illustrated in Fig. 17b, employs two butterfly computations, two complex multiplications, and two two-point FFTs to construct a four-point FFT, adhering to the iterative structure delineated in Fig. 14. Furthermore, this example answer also demonstrates the methodology of connecting "enable" and "done" signals of sub-modules to effectuate the precedence/parallel relationships outlined above.

```
// I am writting a four-point DIF-FFT on FPGA. You can use the following IP
// cores to build the target four-point FFT IP core.

// Here is the templete of butterfly computation IP Core:
//     butterfly_computation butterfly_inst(
//       .clk(),
//       .reset(),
//        .enable(),
//        .in1(), .in2(),
//        .done(),
//        .out1(), .out2()
//     );

// And here is the templete of two-point FFT IP Core:
//    fft_2_point fft_2_point_inst(
//      .clk(),
//      .reset(),
//      .enable(),
//      .x1(), .x2(),
//      .done(),
//      .y1(), .y2()
//    );

// And here is the templete of complex multiplication IP Core:
//      complex_multiplication complex_multiplication_inst(
//        .clk(),
//        .reset(),
//        .enable(),
//        .in1(), .in2(),
//        .done(),
//        .out()
//      );

// I also give you the templete of the target four-point FFT IP Core:
//     fft_4_point fft_4_point_stage_2_1(
//        .clk(),
//        .reset(),
//        .enable(),
//        .x1(), .x2(), .x3(), .x4(),
//        .y1(), .y2(), .y3(), .y4(),
//        .done()
//     );

// Further, I also have some suggestions for you
// First, you don't need to calcualte the expressions of twiddle factors
// I will fill in the value myself later

// Second, let me tell you the major spirit in developing the 4-point FFT:
// In the first stage of FFT, you have two substage called stage 1a and 1b.
// In stage 1a, you use two butterfly_computation mudules;
// and then in stage 1b, you use two twiddle moduels
// In the second stage, you are facing two two-point FFT problem,
// so you just use two fft_2_point and slove the problem.
```

(a)

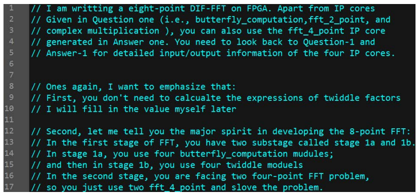
```
module fft_4_point (
  input wire clk,
  input wire reset,
  input wire enable,
  input wire [31:0] x1, input wire [31:0] x2,
  input wire [31:0] x3, input wire [31:0] x4,
  output wire [31:0] y1, output wire [31:0] y2,
  output wire [31:0] y3, output wire [31:0] y4,
  output wire done
);

// Please fill in the expression of twiddle factors
  localparam [31:0] Twiddle_Factor_W_4_0 = ...
  localparam [31:0] Twiddle_Factor_W_4_1 = ...;

//Stage 1a
  wire stage_1a_done;
  wire stage_1a_enable = enable;
  wire [31:0] stage_1a_out1, stage_1a_out2, stage_1a_out3, stage_1a_out4;

  butterfly_computation butterfly_stage_1a_1(
        .clk(clk),
        .reset(reset),
        .enable(stage_1a_enable),
        .in1(x1), .in2(x3),
        .out1(stage_1a_out1), .out2(stage_1a_out3),
        .done() );

  butterfly_computation butterfly_stage_1a_2(
        .clk(clk),
        .reset(reset),
        .enable(stage_1a_enable),
        .in1(x2), .in2(x4),
        .out1(stage_1a_out2), .out2(stage_1a_out4),
        .done(stage_1a_done) );

//Stage 1b
  wire stage_1b_done;
  wire stage_1b_enable = stage_1a_done;
  wire [31:0] stage_1b_out1, stage_1b_out2, stage_1b_out3, stage_1b_out4;

  assign stage_1b_out1 = stage_1a_out1;
  assign stage_1b_out2 = stage_1a_out2;

  complex_multiplication twiddle_stage_1b_1(
        .clk(clk),
        .reset(reset),
        .enable(stage_1b_enable),
        .in1(stage_1a_out3),.in2(Twiddle_Factor_W_4_0),
        .out(stage_1b_out3),
        .done() );

  complex_multiplication twiddle_stage_1b_2(
        .clk(clk),
        .reset(reset),
        .enable(stage_1b_enable),
        .in1(stage_1a_out4),.in2(Twiddle_Factor_W_4_1),
        .out(stage_1b_out4),
        .done(stage_1b_done) );

//Stage 2
  wire stage_2_done;
  assign stage_2_enable = stage_1b_done;
  wire [31:0] stage_2_out1, stage_2_out2, stage_2_out3, stage_2_out4;

  fft_2_point fft_2_point_stage_2_1(
        .clk(clk),
        .reset(reset),
        .enable(stage_2_enable),
        .x1(stage_1b_out1), .x2(stage_1b_out2),
        .y1(stage_2_out1), .y2(stage_2_out2),
        .done() );

  fft_2_point fft_2_point_stage_2_2(
        .clk(clk),
        .reset(reset),
        .enable(stage_2_enable),
        .x1(stage_1b_out3), .x2(stage_1b_out4),
        .y1(stage_2_out3), .y2(stage_2_out4),
        .done(stage_2_done) );

//Final output
  assign y1 = stage_2_out1;
  assign y2 = stage_2_out2;
  assign y3 = stage_2_out3;
  assign y4 = stage_2_out4;
  assign done = stage_2_done;

endmodule
```

(b)

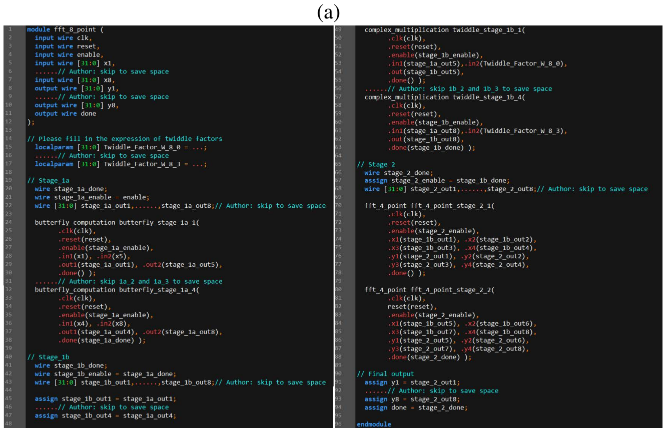
Figure 17: The first question-example pair that focuses on a four-point FFT.

In the third step, we proceed with the second question-example pair. The example question, presented in Fig. 18a, asks ChatGPT to develop an eight-point FFT module base on the provided four-point FFT, which is obtained in the first question-answer pair. Our example answer, showcased in Fig. 18b, outlines how the numerous sub-modules (consisting of four butterfly computations, four complex multiplications, and two four-point FFTs) are interconnected in accordance with the iterative FFT flow graph. Furthermore, we present the method of connecting "enable" and "done" signals once again, reinforcing this knowledge for ChatGPT.

In the fourth step, we cease providing examples. Instead, we pose a new question to ChatGPT akin to the previous example question: generate a 16-point FFT predicated on the eight-point FFT provided (i.e., the one we give as the example answer in step three). This time, ChatGPT produces an implementation code that is synthesizable and capable of generating outputs identical to those of a Xilinx 16-point FFT



(a)



(b)

Figure 18: The second question-example pair. In (b), to conserve space, we omit portions of the code that are highly similar or repetitive (refer to our comments on Line 6, Line 9, Line 16, Line 22, Line 31, Line 43, Line 46, Line 56, Line 68, and Line 92 for more details).

IP core, thereby ensuring functional correctness. The only persisting issue pertains to the absence of twiddle factors, a problem that we intend to address in the succeeding subsection (in the above benchmarking with the Xilinx IP core, we filled in the twiddle factors generated in Subsection B to the code).

Finally, we repeat step four in an iterative manner. This is, we ask ChatGPT to generate an N-point FFT with the provided $N/2$-point FFT (which was generated by ChatGPT in the preceding iteration). We do not stop the iteration until we acquire the desired FFT module. In this paper, as a proof of concept, we terminate at the 64-point FFT and present the generated code as in Fig. 19 below. We test the code with our 64-point FFT testbench and compare the output of the LLM-written module with the output of a 64-point FFT IP core provided by Xilinx. Experimental results reveal that the Verilog module, written by ChatGPT, is functionally accurate after the above iterative generation process.

B. Chain-of-Thought (CoT) Prompting for Challenge Two

A brief introduction about CoT prompting

A typical class of tasks that present challenges to language models is solving mathematical problems, particularly those requiring multi-step reasoning [14]–[17]. This challenge persisted as a tough problem in the NLP community until the advent of LLMs. In [36], the authors surprisingly discovered that their language model's arithmetic reasoning capability can be dramatically enhanced when the model size scales beyond 100 billion parameters. Furthermore, [37] found that guiding an LLM through a human's chain-of-thought in breaking down a multi-step problem into intermediary steps can enable the model in solving complex reasoning problems, which are unattainable with conventional prompting methods. These two inspiring discoveries have inspired a recent surge of research interest in chain-of-thought (CoT) prompting for LLMs.

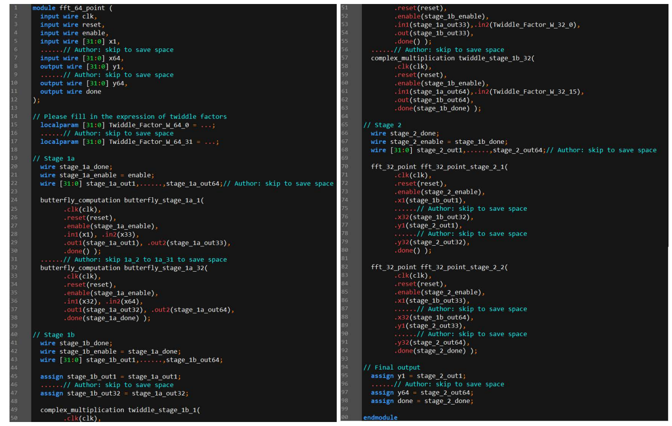


Figure 19: The 64-point FFT IP core generated by ChatGPT. To conserve space, we omit portions of the code that are highly similar or repetitive (refer to our comments on Line 6, Line 9, Line 16, Line 22, Line 31, Line 46, Line 56, Line 68, Line 75, Line 78, Line 87, Line 90, and Line 96 for more details).

We now give an example to illustrate the concept of CoT prompting. In this example, as shown in Fig. 20, the baseline prompt comprises ICL with a single example question-answer pair. In contrast, the CoT prompt extends the example answer to incorporate a chain of thought detailing how the problem should be dissected and tackled. For more examples illustrating the efficacy of CoT prompting, we refer interested readers to [14].

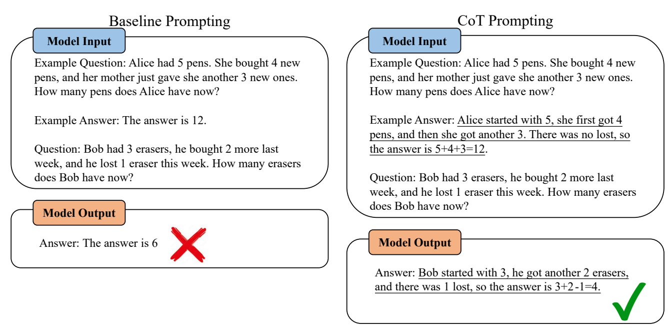


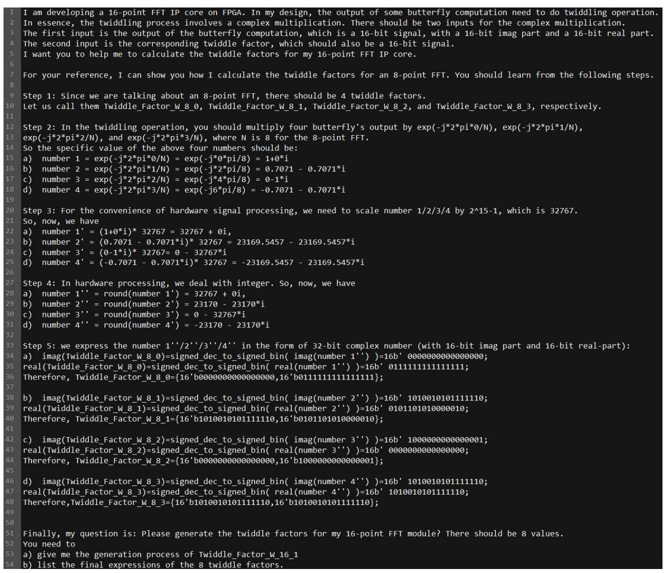
Figure 20: An example to illustrate the CoT prompting technique. The CoT reasoning processes are underlined.

From the above example, it is evident that ICL coupled with CoT prompting outperforms the baseline approach. However, it is important to note that we do not mean that a contemporary LLM cannot generate the correct answer using the baseline prompt. Our intention is to use this example to demonstrate the concept of CoT prompting and how it should be employed. In fact, LLMs nowadays have advanced beyond those reported in early studies and can produce correct results for the simple question depicted in Fig. 20, even without the assistance of ICL or CoT prompting. However, for more complex tasks, such as the twiddle factor generation tasks we describe in

Section IV-C, we observe that the challenge of multi-step reasoning persists. That motivates us to integrate CoT prompting within the ICL framework in this study.

CoT prompting for our twiddle factors generation task

In Section IV-C, we elucidated the process of converting twiddle factors into 32-bit sequences for digital circuitry. Here we describe the multi-step transformation process in detail and design the CoT prompt. As an illustration, our prompt employs the generation process of the twiddle factors for an eight-point FFT (i.e., W_8^0, W_8^1, W_8^2, and W_8^3) as examples. And then we ask ChatGPT to generate the twiddle factors for a 16-point FFT. We depict our prompt and the resulting twiddle factor sequences in Fig. 21 and Table I, respectively.



```
I am developing a 16-point FFT IP core on FPGA. In my design, the output of some butterfly computation need to do twiddling operation.
In essence, the twiddling process involves a complex multiplication. There should be two inputs for the complex multiplication.
The first input is the output of the butterfly computation, which is a 16-bit signal, with a 16-bit imag part and a 16-bit real part.
The second input is the corresponding twiddle factor, which should also be a 16-bit signal.
I want you to help me to calculate the twiddle factors for my 16-point FFT IP core.

For your reference, I can show you how I calculate the twiddle factors for an 8-point FFT. You should learn from the following steps.

Step 1: Since we are talking about an 8-point FFT, there should be 4 twiddle factors.
Let us call them Twiddle_Factor_W_8_0, Twiddle_Factor_W_8_1, Twiddle_Factor_W_8_2, and Twiddle_Factor_W_8_3, respectively.

Step 2: In the twiddling operation, you should multiply four butterfly's output by exp(-j*2*pi*0/N), exp(-j*2*pi*1/N),
exp(-j*2*pi*2/N), and exp(-j*2*pi*3/N), where N is 8 for the 8-point FFT.
So the specific value of the above four numbers should be:
a)  number 1 = exp(-j*2*pi*0/N) = exp(-j*0*pi/8) = 1+0*i
b)  number 2 = exp(-j*2*pi*1/N) = exp(-j*2*pi/8) = 0.7071 - 0.7071*i
c)  number 3 = exp(-j*2*pi*2/N) = exp(-j*4*pi/8) = 0-1*i
d)  number 4 = exp(-j*2*pi*3/N) = exp(-j6*pi/8) = -0.7071 - 0.7071*i

Step 3: For the convenience of hardware signal processing, we need to scale number 1/2/3/4 by 2^15-1, which is 32767.
So, now, we have
a)  number 1' = (1+0*i)* 32767 = 32767 + 0i,
b)  number 2' = (0.7071 - 0.7071*i)* 32767 = 23169.5457 - 23169.5457*i
c)  number 3' = (0-1*i)* 32767= 0 - 32767*i
d)  number 4' = (-0.7071 - 0.7071*i)* 32767 = -23169.5457 - 23169.5457*i

Step 4: In hardware processing, we deal with integer. So, now, we have
a)  number 1'' = round(number 1') = 32767 + 0i,
b)  number 2'' = round(number 2') = 23170 - 23170*i
c)  number 3'' = round(number 3') = 0 - 32767*i
d)  number 4'' = round(number 4') = -23170 - 23170*i

Step 5: we express the number 1''/2''/3''/4'' in the form of 32-bit complex number (with 16-bit imag part and 16-bit real-part):
a)  imag(Twiddle_Factor_W_8_0)=signed_dec_to_signed_bin( imag(number 1'') )=16b' 0000000000000000;
real(Twiddle_Factor_W_8_0)=signed_dec_to_signed_bin( real(number 1'') )=16b' 0111111111111111;
Therefore, Twiddle_Factor_W_8_0={16'b0000000000000000,16'b0111111111111111};

b)  imag(Twiddle_Factor_W_8_1)=signed_dec_to_signed_bin( imag(number 2'') )=16b' 1010010101111110;
real(Twiddle_Factor_W_8_1)=signed_dec_to_signed_bin( real(number 2'') )=16b' 0101101010000010;
Therefore, Twiddle_Factor_W_8_1={16'b1010010101111110,16'b0101101010000010};

c)  imag(Twiddle_Factor_W_8_2)=signed_dec_to_signed_bin( imag(number 3'') )=16b' 1000000000000001;
real(Twiddle_Factor_W_8_2)=signed_dec_to_signed_bin( real(number 3'') )=16b' 0000000000000000;
Therefore, Twiddle_Factor_W_8_2={16'b0000000000000000,16'b1000000000000001};

d)  imag(Twiddle_Factor_W_8_3)=signed_dec_to_signed_bin( imag(number 4'') )=16b' 1010010101111110;
real(Twiddle_Factor_W_8_3)=signed_dec_to_signed_bin( real(number 4'') )=16b' 1010010101111110;
Therefore,Twiddle_Factor_W_8_3={16'b1010010101111110,16'b1010010101111110};


Finally, my question is: Please generate the twiddle factors for my 16-point FFT module? There should be 8 values.
You need to
a) give me the generation process of Twiddle_Factor_W_16_1
b) list the final expressions of the 8 twiddle factors.
```

Figure 21: The CoT prompt we used to generate the 32-bit sequences for twiddle factors in a 16-point FFT.

Table I: LLM generated 32-bit sequences for twiddle factors in 16-point FFT

	Imag Part	Real Part	32-bit Sequence
W_{16}^0	1.0000	0.0000i	0000,0000,0000,0000, 0111,1111,1111,1111
W_{16}^1	0.9239	-0.3827i	1100,1111,0000,0101, 0111,0110,0100,0001
W_{16}^2	0.7071	-0.7071i	1010,0101,0111,1110, 0101,1010,1000,0010
W_{16}^3	0.3827	-0.9239i	1000,1001,1011,1111, 0011,0000,1111,1011
W_{16}^4	0.0000	-1.0000i	1000,0000,0000,0001, 0000,0000,0000,0000
W_{16}^5	-0.3827	-0.9239i	1000,1001,1011,1111, 1100,1111,0000,0101
W_{16}^6	-0.7071	-0.7071i	1010,0101,0111,1110, 1010,0101,0111,1110
W_{16}^7	-0.9239	-0.3827i	1100,1111,0000,0101, 1000,1001,1011,1111

Moreover, we verify that the same CoT prompt (employing W_8^0, W_8^1, W_8^2, and W_8^3 as examples) is applicable for generating twiddle factors for larger-scale FFTs, such as 32-point or 64-point. In other words, we can skip teaching ChatGPT about the factors generation of a 32-point FFT and directly jump to the factor generation of a 64-point FFT, which affirms that the LLM does internalize the crucial knowledge imparted through the CoT prompt (rather than simply parroting the input). Here we do not present the twiddle factor generation process for larger-scale FFTs due to page limitation. We encourage interested readers to give it a try themselves.

VI. Conclusion

This paper delves into the intersection of Large Language Models (LLMs) and wireless communication technologies, yielding inspiring results in utilizing LLMs to prototype wireless systems. Our research highlights the potential of LLMs in facilitating complex FPGA development within wireless systems.

We begin by demonstrating how an LLM can serve as a crucial assistant for FPGA development, providing examples in code refactoring, code reuse, and system validation. Moreover, we showcase LLMs' ability to generate sophisticated Hardware Description Language (HDL) codes for advanced signal-processing algorithms in wireless communication, with a focus on the fundamental Fast Fourier Transform (FFT) processing.

By addressing the subtask scheduling problem and multi-step thinking problem through In-context Learning (ICL) and the Chain of Thoughts (CoT) prompting techniques, we successfully generated a 64-point Verilog FFT module using LLMs for the first time. This exploration of LLMs' generalization and imitation capabilities expands their potential applications and underscores their value in the wireless communication domain.

References

[1] B. Brik, H. Chergui, L. Zanzi, F. Devoti, A. Ksentini, M. S. Siddiqui, X. Costa-Pérez, and C. Verikoukis, "A survey on explainable AI for 6G O-RAN: Architecture, use cases, challenges and research directions," *arXiv preprint arXiv:2307.00319*, 2023.

[2] I. F. Akyildiz, H. Guo, R. Dai, and W. Gerstacker, "Mulsemedia communication research challenges for metaverse in 6G wireless systems," *arXiv preprint arXiv:2306.16359*, 2023.

[3] L. Bariah, Q. Zhao, H. Zou, Y. Tian, F. Bader, and M. Debbah, "Large language models for telecom: The next big thing?" *arXiv preprint arXiv:2306.10249*, 2023.

[4] X. Jiao, W. Liu, M. Mehari, M. Aslam, and I. Moerman, "Openwifi: a free and open-source IEEE802. 11 SDR implementation on SoC," in *VTC2020-Spring*. IEEE, 2020, pp. 1–2.

[5] Y. Du, S. C. Liew, and Y. Shao, "Efficient FFT computation in IFDMA transceivers," *IEEE Trans. Wirel. Commun.*, 2023.

[6] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of Github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[7] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," *arXiv preprint arXiv:2112.02125*, 2021.

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

[9] RapidSilicon, "Rapidgpt," 2023, [Online]. Available: https://rapidsilicon.com/rapidgpt/.

[10] 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.

[11] 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," in *2023 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2023, pp. 1–6.

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

[13] 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.

[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] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," *arXiv preprint arXiv:2203.11171*, 2022.

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

[17] F. Shi, M. Suzgun, M. Freitag, X. Wang, S. Srivats, S. Vosoughi, H. W. Chung, Y. Tay, S. Ruder, D. Zhou *et al.*, "Language models are multilingual chain-of-thought reasoners," *arXiv preprint arXiv:2210.03057*, 2022.

[18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[20] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, 2019.

[21] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[22] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[23] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS today: Successes, challenges, and opportunities," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, pp. 1–42, 2022.

[24] C. Zarkos, "Verilog implementation of a low-cost vector AI accelerator and integration in a RISC-V processor," B.S. thesis, Universitat Politècnica de Catalunya, 2023.

[25] C. Kwok, P. Viswanathan, and P. Yeung, "Addressing the challenges of reset verification in SoC designs," in *Design and Verification Conference and Exhibition United States*, 2015.

[26] C. E. Cummings, D. Mills, and S. Golson, "Asynchronous & synchronous reset design techniques-part deux," *SNUG Boston*, vol. 9, 2003.

[27] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *IEEE ICASSP*, vol. 3. IEEE, 1998, pp. 1381–1384.

[28] C. F. Hsiao, Y. Chen, and C. Y. Lee, "A generalized mixed-radix algorithm for memory-based FFT processors," *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 57, no. 1, pp. 26–30, 2010.

[29] M. Garrido, "A survey on pipelined FFT hardware architectures," *J. Signal Process. Syst.*, pp. 1–20, 2021.

[30] Y. T. Ma, "A VLSI-oriented parallel FFT algorithm," *IEEE Trans. Signal Process.*, vol. 44, no. 2, pp. 445–448, 1996.

[31] S. Golson *et al.*, "State machine design techniques for Verilog and VHDL," *Synopsys Journal of High-Level Design*, vol. 9, no. 1-48, p. 12, 1994.

[32] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, "Rethinking the role of demonstrations: What makes in-context learning work?" *arXiv preprint arXiv:2202.12837*, 2022.

[33] S. Chan, A. Santoro, A. Lampinen, J. Wang, A. Singh, P. Richemond, J. McClelland, and F. Hill, "Data distributional properties drive emergent in-context learning in transformers," *Advances in Neural Information Processing Systems*, vol. 35, pp. 18 878–18 891, 2022.

[34] S. M. Xie, A. Raghunathan, P. Liang, and T. Ma, "An explanation of in-context learning as implicit bayesian inference," *arXiv preprint arXiv:2111.02080*, 2021.

[35] S. Garg, D. Tsipras, P. S. Liang, and G. Valiant, "What can transformers learn in-context? A case study of simple function classes," *Advances in Neural Information Processing Systems*, vol. 35, pp. 30 583–30 598, 2022.

[36] S. Narang and A. Chowdhery, "Pathways language model (PaLM): Scaling to 540 billion parameters for breakthrough performance," *Google AI Blog*, 2022.

[37] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano *et al.*, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.