HAMM-LIPPSTADT
UNIVERSITY OF APPLIED SCIENCES

AI for Automatic HDL Code Generation from High-Level Descriptions

Department Lippstadt 2

Advance Hardware Engineering

submitted by

Md Ruhul Kuddus Saleh

Electronic Engineering

Mat.Nr.: 2219956

md-ruhul-kuddus.saleh@stud.hshl.de

18 December 2025

Supervisor: Prof. Dr.–Ing. Ali Hayek

Abstract

Development of a Hardware Description Language (HDL) is among the most time-consuming and expert-resource-intensive steps of FPGA and ASIC designs. With the growing complexity of digital systems and pressing demands of time-to-market, conventional manual HDL designs using the Register Transfer-Level (RTL) approach are no longer adequately scalable. However, with recent AI technology and advancements achieved through the use of large language models (LLMs) in the processing and translation of natural language descriptions, there is a new potential channel of automatic generation of HDL from a natural language description or from an algorithmic description.

The HDL design synthesis comparison between professionally developed HDL designs and designs generated with AI on a target Xilinx Artix-7 FPGA evaluates synthesis feasibility, resource usage, power performance, and possible implementation on silicon. Although the AI designs have correct functional intent and proper HDL syntax, important deficiencies are found in its support for design tools, power performance, overall system architecture understanding, and constraints on implementation. There are instances where synthesis fails or goes beyond device capabilities, making it unimplementable in hardware.

The outcome of the study clearly states that the current application of AI is more optimal in the role of an HDL code assistant, rather than being an independent hardware designer. Also, there is an imperative need for major breakthroughs in the field of device-aware training, power-efficient designs, and verification.

Contents

Introduction

Moore's Law and Dennard scaling have traditionally been the pillars sustaining the semiconductor industry. However, lately, the rate of progress has gradually declined [1]. This has increased the focus on the concept of heterogeneous computing architecture, which lays the foundation using combined CPU architecture specific hardware such as GPUs, FPGAs, or ASIC [2]. This redistribution of workload on platforms aligned for the specific function helps focally maximize performance along with power savings [3].

FPGAs are considered the brightest component within the industry. This assists in the processing associated with specialized calculations [4]. However, FPGAs are complex in nature. They require knowhow in the realm of HDL programming, specifically Verilog or VHDL programming [10]. Manual processing within the handling associated with high-level algorithms in terms of generating the RTL model has always proven time-consuming and error-prone [6, 7]. Moreover, the High-Level Synthesis (HLS) tools available are often limited in handling specialized software-centric processing efficiently [8, 9].

This has sparked the idea of using AI, specifically Large Language Models (LLMs), in the processing [5]. This paper aims at exploring the feasibility associated with AI-synthesized HDL. They compare the AI-synthesized designs with the designs embedded by professionals. This paper aims at exploring the implementation of AI in the hardware description within the industry [10].

High-Level Descriptions for FPGA Design

High-level descriptions form the bridge between the algorithmic intention and the hardware implementation in both conventional high-level synthesis and AI-based design flows. The different descriptions vary based on the level of abstraction, expressiveness, and ability to model hardware constructs like parallelism, time, and data transport. The description approach used has a major impact on design productivity and the quality of the produced RTL.

A. Software-Like Languages

The traditional HLS flow relies on software-oriented programming languages like C, C++, OpenCL, and SystemC to program hardware functionality [1]. The mentioned programming languages are widely understandable and supported, although not optimal in their implementation on FPGA architectures. They are natural and suited for software design, which hides hardware parallelizations, making developers entirely dependent on implementation-specific directives to manage parallelization and pipelining [6]. Moreover, these software-oriented design languages lack sufficient control over memory, and there are no timing-related constructs that make cycle-accurate implementation and optimization difficult [8, 10].

B. Domain Specific Languages

Domain-specific languages (DSLs) such as Chisel, Bluespec, MyHDL, and PyMTL have overcome these challenges through the use of hardware-oriented abstraction constructs [2]. The mentioned languages facilitate hardwired structural description and offer parallelism as a primitive, hence offering a simpler description of a hardware design [4]. The modeling tools, such as MATLAB and Simulink, facilitate fast functional modeling, while efficient implementation, in most cases, demands optimization [9].

C. Behavioral vs. Functional Descriptions

High-level descriptions can thus be categorized into Behavioral or Functional. Behavioral styles are good at explaining the behavior of algorithms, while having little architectural relevance. This results in data paths and control logic being abstracted by the synthesizer or the AI tools, which proves inefficient at times [5]. Functional styles, which include dataflow-based models, address the dependency architecture comprehensively, thus being relatively free from ambiguities in the synthesis process [3]. Hence, description style selection assumes importance in synthesizing hardware at the desired level, either using AI or the conventional approach at times [7].

Traditional High-Level Synthesis Techniques

High-Level Synthesis (HLS) has also been recognized as an innovative approach for Electronic Design Automation (EDA) in bridging the increasing productivity gap in hardware design. Since HLS enables designers to model their hardware designs in high-level languages (HLLs), it automatically translates the description of an algorithm into an RTL description, thus offering faster design development time as it does not require RTL coding manually. However, the broader design space associated with high-level synthesis also brings in its own design difficulties.

A. Core HLS Workflow

The translation of high-level code to hardware involves a streamlined set of steps for synthesis and optimization. Scheduling involves identifying the clock cycle during which an operation should take place, trading off between clock cycles, throughput, and timing constraints, keeping data dependencies in mind.

Resource allocation involves specifications related to hardware resource types and quantities, which could range from arithmetic units, memory blocks, to specifying which operations and variables bind to resource instances, also taking into account resource constraints for implementation. For exploiting parallelism, optimization techniques involving unrolling and pipelining are carried out, which enable concurrent execution, hence speeding up execution by executing a number of iterations concurrently for loops. High-level data storage is then assigned to physical memory resources, which range from register memory to Block RAM (BRAM), also requiring manual partitioning to satisfy bandwidth constraints. Finally, specifying control logic through finite state machines (FSMs) for controlling datapath execution and data transfer is required for designing digital systems by controlling data transfer between different stages through FSMs that implement control flow during system execution.

B. Existing HLS Tools

The HLS environment encompasses a range of tools, including proprietary tools and academic tools. The proprietary tools include Xilinx Vivado HLS (Vitis HLS) and the Intel HLS Compiler, which are designed for specific FPGA architectures. Cadence Stratus HLS and Mentor Catapult HLS enable a wider design space exploration of FPGAs and ASICs. At the same time, open-source academic tools like LegUp and ROCCC enable a flexible environment for customizing the tools. While the environment provides more insight into the design process, the tools lack the timing closure and design-specific optimization of the proprietary tools.

C. Limitations of Traditional HLS

However, the existing traditional HLS tools have various drawbacks that impede their autonomy. The HLS tools tend to parse software-centric code literally and lack the capability to infer the optimal hardware architecture without the involvement of the designer. Obtaining optimal performance requires the tedious insertion of directives by the designer, which makes the exploration of the design space longer and repetitive.

The existing HLS tools tend to behave inefficiently while processing irregular control flow or dynamically varying memory access patterns, which lead to suboptimal hardware consumption and resource usage. The longer synthesis and analysis time introduces hurdles in the optimization process by leading to delayed feedback on power consumption, performance, and area. The above hindrances open up avenues for exploring the usage of AI-based approaches that facilitate the efficient inference and optimization of architecture.

V. AI Techniques for HDL Code Generation

The shortcomings of conventional High-Level Synthesis (HLS), specifically the need for human optimization and the expensiveness of design space exploration, have necessitated the involvement of Artificial Intelligence (AI) in the design flow process. More advanced AI methods, based on predictive machine learning and generative Large Language Models (LLM), seek to automate the optimization process and minimize human involvement during hardware design [5, 10].

A. Machine Learning for Performance Prediction

These models use machine learning algorithms to approximate the Quality of Results (QoR) attributes such as area, latency, and power consumption without the need for complete synthesis [6]. These models are trained using data from past designs to provide feedback efficiently and aid the selection of pragmas for loop pipelining and memory partitioning [2, 8].

B. Deep Learning

Deep learning techniques make it possible to better model complex hardware graphs. Graph Neural Networks (GNNs) analyze structural graphs like Abstract Syntax Trees

(ASTs) and Dataflow Graphs (DFGs), focusing on dependencies that are useful in understanding resource utilization and timing information in code [8].

Models based on the transformer neural network further enhance this analysis by finding long-term dependencies in code that are significant in determining hardware efficiency [5].

C. Large Language Models

Large Language Models (LLMs), namely ChatGPT, CodeLlama, and StarCoder, bring generative capabilities to hardware EDA. They have the ability to transform natural-language descriptions into synthesizable HDL and assist with code completion and refactoring [5, 9, 11]. The limitations of these approaches lie in hallucination, lack of awareness regarding timing aspects, and ignorance of specific device constraints [10, 11].

VI. Experimental Analysis and Results

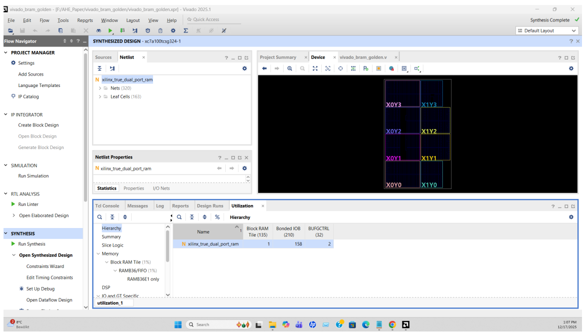
For a practical estimate of the viability of using AI-based hardware descriptions, a comparative experimental test was done on professionally designed "golden reference" designs and HDL generated using state-of-the-art large language models. The test was done on a Xilinx Artix-7 FPGA technology using the Vivado 2025.1 design tools. The memory inference HDL was tested using ChatGPT-5.2, while the more complex control HDL was generated using Claude Sonnet 4.5.

A. Experiment 1: Memory Inference Using ChatGPT-5.2

The initial experiment tested if it was possible for an AI model to deduce a correct dual-port BRAM and to implement conventional power management techniques, such as clock enables. The reference design used was that of a professional reference design taken from a Xilinx memory template.

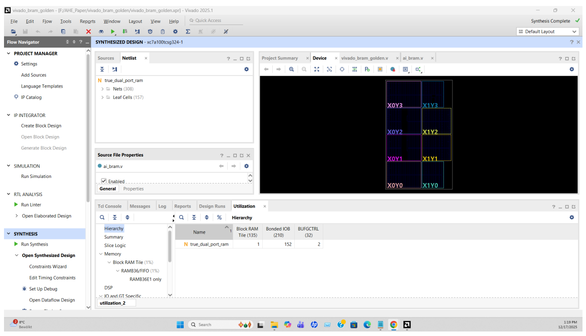
Table 1: BRAM Resource Utilization and Workflow Outcome

Metric	**Professional (Xilinx Template)**	**AI-Generated (ChatGPT-5.2)**	**Difference**
Block RAM Tiles	1 (1.00%)	1 (1.00%)	0%
Bonded IOB	158	152	−3.8%
Synthesis Status	**Pass**	**Fail**(Syntax Error)	**Workflow Failure**



Name	Block RAM Tile (135)	Bonded IOB (210)	BUFGCTRL (32)
xilinx_true_dual_port_ram	1	158	2

(a) Professional BRAM Utilization

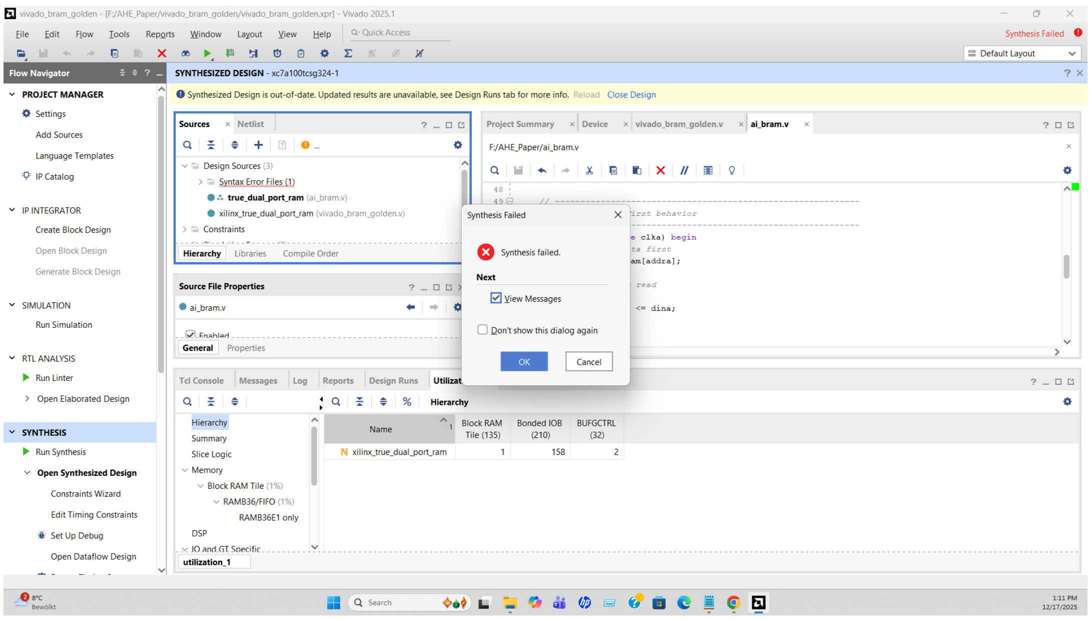


Name	Block RAM Tile (135)	Bonded IOB (210)	BUFGCTRL (32)
true_dual_port_ram	1	152	2

(b) AI BRAM Utilization

Figure 1: Utilization reports for BRAM inference showing identical memory resource usage.

ChatGPT-5.2 was able to correctly infer the RAMB36E1 primitive. Unfortunately, there was an issue with the tool flow compatibility during the synthesis step. ChatGPT-5.2 was able to produce SystemVerilog language constructs while assigning the Verilog-2001 file extension. This resulted in the design being rejected by the Vivado compiler because of the improper syntax.



vivado_bram_golden - [F:/AHE_Paper/vivado_bram_golden/vivado_bram_golden.xpr] - Vivado 2025.1

Synthesis Failed

SYNTHESIZED DESIGN - xc7a100tcsg324-1

Synthesized Design is out-of-date. Updated results are unavailable, see Design Runs tab for more info. Reload Close Design

Design Sources (3)
Syntax Error Files (1)
true_dual_port_ram (ai_bram.v)
xilinx_true_dual_port_ram (vivado_bram_golden.v)
Constraints

F:/AHE_Paper/ai_bram.v

Synthesis Failed

Synthesis failed.

Next

[✓] View Messages

[ ] Don't show this dialog again

OK Cancel

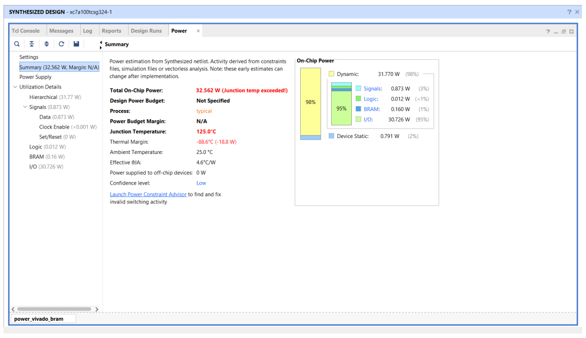
Name	Block RAM Tile (135)	Bonded IOB (210)	BUFGCTRL (32)
xilinx_true_dual_port_ram	1	158	2

Figure 2: Vivado synthesis failure screenshot showing the syntax errors in the AI-generated HDL.

There was a more critical difference between the power analysis. Even though both designs used the same memory resources, the AI-designed architecture used nearly twice the on-chip power.

Table 2: Power Consumption Comparison for BRAM Design

Power Component	Professional	AI-Generated	Increase
BRAM Power	0.160 W	0.321 W	+100% (2x)
I/O Power	30.726 W	61.414 W	+100% (2x)
Total On-Chip Power	**32.562 W**	**64.294 W**	+97%



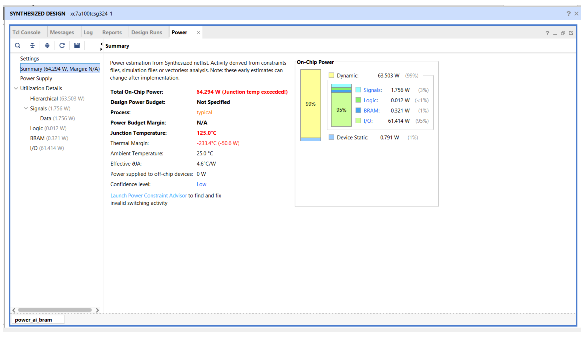
Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**32.562 W (Junction temp exceeded!)**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**125.0°C**
Thermal Margin:	-88.6°C (-18.8 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

Dynamic:	31.770 W	(98%)
Signals:	0.873 W	(3%)
Logic:	0.012 W	(<1%)
BRAM:	0.160 W	(1%)
I/O:	30.726 W	(95%)
Device Static:	0.791 W	(2%)

(a) Professional Power Report



Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**64.294 W (Junction temp exceeded!)**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**125.0°C**
Thermal Margin:	-233.4°C (-50.6 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

Dynamic:	63.503 W	(99%)
Signals:	1.756 W	(3%)
Logic:	0.012 W	(<1%)
BRAM:	0.321 W	(1%)
I/O:	61.414 W	(95%)
Device Static:	0.791 W	(1%)

(b) AI-Generated Power Report

Figure 3: Comparison of on-chip power consumption; the AI design (b) exceeds junction temperature limits.

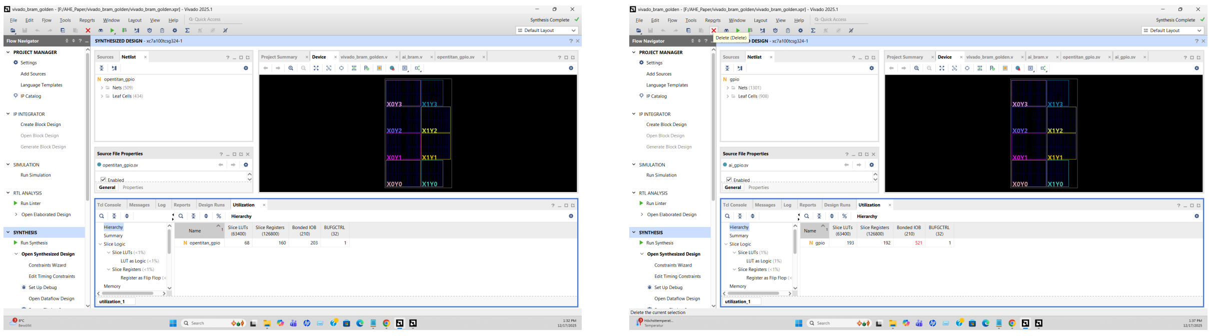
The professionally designed code incorporated clock-enabling signals (ena/enb) that disabled the BRAM during idling cycles. The generated code lacked the enabling signals, which caused the memory and I/O to run continuously. The effect caused the dynamic power consumption to increase by almost twice the original amount, thus lacking power consumption design reasoning.

B. Experiment 2: Generation of Control Logic Using Claude Sonnet 4.5

The second experiment involved the generation of a GPIO controller with a masked write capability. This was an exercise in understanding system-level interfaces as well as realizing efficient hardware-level parallelism. The original design was based on an OpenTitan project.

Table 3: Logic Resource and Physical Constraint Utilization

Metric	**Professional (OpenTitan)**	**AI-Generated (Claude 4.5)**	**Status**
Slice LUTs	68	193	2.8× Increase
Slice Registers	160	192	1.2× Increase
Bonded IOB	203	521	**FAILED (148%)**
Max IOB Available	210	210	—
Total Leaf Cells	434	908	**2.1x Area**


Sources Netlist
opentitan_gpio
Nets (509)
Leaf Cells (434)

Source File Properties
opentitan_gpio.sv
☑ Enabled
General Properties

Tcl Console Messages Log Reports Design Runs Utilization

Hierarchy
Summary
Slice Logic
Slice LUTs (<1%)
LUT as Logic (<1%)
Slice Registers (<1%)
Register as Flip Flop (<
Memory

Name	Slice LUTs (63400)	Slice Registers (126800)	Bonded IOB (210)	BUFGCTRL (32)
opentitan_gpio	68	160	203	1

utilization_1

(a) Professional GPIO Utilization

Sources Netlist
gpio
Nets (1301)
Leaf Cells (908)

Source File Properties
ai_gpio.sv
☑ Enabled
General Properties

Tcl Console Messages Log Reports Design Runs Utilization

Hierarchy
Summary
Slice Logic
Slice LUTs (1%)
LUT as Logic (1%)
Slice Registers (<1%)
Register as Flip Flop (<
Memory

Name	Slice LUTs (63400)	Slice Registers (126800)	Bonded IOB (210)	BUFGCTRL (32)
gpio	193	192	521	1

utilization_1

(b) AI GPIO Utilization

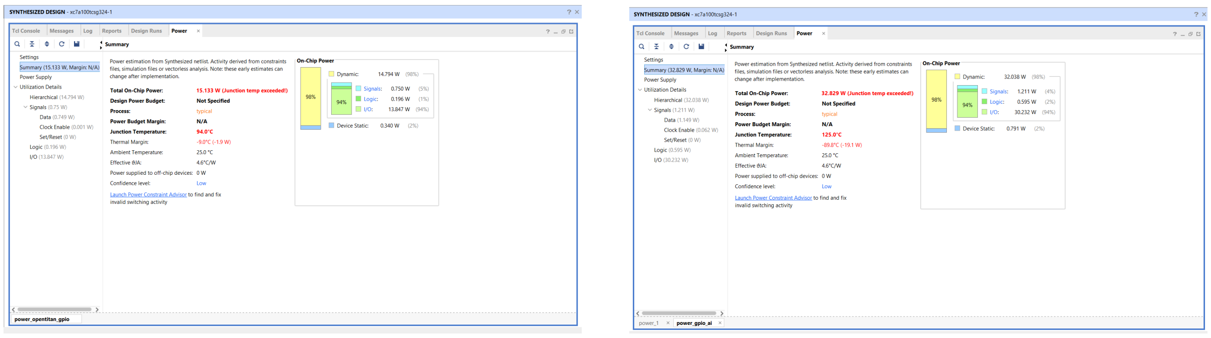
Figure 4: Utilization reports highlighting the massive increase in Bonded IOB and LUTs in the AI-generated design.

The hardware unimplementability of the design created by the AI involved many I/O pins. In contrast to the professional design approach, the professional design connected the configuration registers using an internal bus. The design created by the AI made many internal signals available through the top-level port. This led to a hardware need for 521 I/O pins, which was above the hardware capability by 148%.

Moreover, the AI implementation utilized software-style ”for loops” for the masked writes. Although correct in functionality, the loops were unrolled during the synthesis process, leading to an abundance of combinational logic. In contrast, the professional implementation utilized a single-cycle approach with bitwise masking.

Table 4: Power and Thermal Analysis Results

Metric	**Professional (OpenTitan)**	**AI-Generated Code**	**Difference / Status**
Logic Power	0.196 W	0.595 W	+203% (3×)
I/O Power	13.847 W	30.232 W	+118%
Total On-Chip Power	15.133 W	32.829 W	+117%
Junction Temp	**94.0 °C (Safe)**	**125.0 °C (Failed)**	**DESTRUCTIVE**


SYNTHESIZED DESIGN - xc7a100tcsg324-1
Tcl Console Messages Log Reports Design Runs Power
Summary

Settings
Summary (15.133 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (14.794 W)
Signals (0.75 W)
Data (0.749 W)
Clock Enable (0.001 W)
Set/Reset (0 W)
Logic (0.196 W)
I/O (13.847 W)

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	15.133 W (Junction temp exceeded!)
Design Power Budget:	Not Specified
Process:	typical
Power Budget Margin:	N/A
Junction Temperature:	94.0°C
Thermal Margin:	-9.0°C (-1.9 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power
Dynamic:	14.794 W	(98%)
Signals:	0.750 W	(5%)
Logic:	0.196 W	(1%)
I/O:	13.847 W	(94%)
Device Static:	0.340 W	(2%)

power_opentitan_gpio

(a) Professional Logic Power

SYNTHESIZED DESIGN - xc7a100tcsg324-1
Tcl Console Messages Log Reports Design Runs Power
Summary

Settings
Summary (32.829 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (32.038 W)
Signals (1.211 W)
Data (1.149 W)
Clock Enable (0.062 W)
Set/Reset (0 W)
Logic (0.595 W)
I/O (30.232 W)

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	32.829 W (Junction temp exceeded!)
Design Power Budget:	Not Specified
Process:	typical
Power Budget Margin:	N/A
Junction Temperature:	125.0°C
Thermal Margin:	-89.8°C (-19.1 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power
Dynamic:	32.038 W	(98%)
Signals:	1.211 W	(4%)
Logic:	0.595 W	(2%)
I/O:	30.232 W	(94%)
Device Static:	0.791 W	(2%)

power_1 power_gpio_ai

(b) AI Logic Power

Figure 5: Thermal analysis showing logic power and junction temperature deltas.

The generated design was capable of attaining the highest permissible junction temperature of 125°C, which would activate the thermal shutdown in the actual design, thus making it incapable of being manufactured.

C. Summary of Key Observations

The results obtained from the experiments have shown that there are three basic challenges associated with the use of the AI-HDL. First, the AI model has a lack of system architecture awareness, causing unmanageable I/O. Secondly, the use of other power-aware design methodologies, such as clock gating, contributes to an increase in the net dynamic power. Finally, the lack of software design methodologies, which includes clock gating, causes a major imbalance in logic usage and resultant thermal characteristics.

VII. Tools and Prototypes Available

AI-driven hardware design is beginning to move from research into early commercial deployment, promising to reduce manual effort in traditional EDA design flows and increase efficiency [2, 5].

Graph Neural Networks used by academic prototypes, such as HARP, have already predicted optimization impacts and accelerated design space exploration [6]. Chip-Chat and Autochip provide iterative improvements to HDL code quality using LLM-based generators [5, 10]. Hybrid flows use neural cost models to enable near-real-time timing and area estimation without full synthesis [8].

Commercial Tools: AI is being introduced by vendors for managing difficult SoC designs. Synopsys DSO.ai uses reinforcement learning to automatically perform synthesis and floorplanning. Cadence Cerebrus is an AI-driven parameter tuner that seeks the optimal PPA; Siemens EDA uses AI-assisted verification for speed in bug detection. These commercial tools enhance, instead of replacing, human designers [5].

Open-source Initiatives: These include datasets like VHDL-Eval and VHDL-Xform, providing support for HDL-specific LLMs and neural cost models, especially for under-represented languages [9, 4]. Community-driven tools focus on automated refactoring of AI-generated HDL with the goal of improving readability and synthesizability [10].

VIII. Benefits and Challenges

Benefits

- **Development Speed:** AI-driven flows speed up development with automatic pragma insertion and parameter tuning [8].
- **Accessibility:** Rapid prototyping is thus possible even for non-experts [1, 10].
- **Consistency:** AI-assisted refactoring and test generation may be used under expert supervision to boost consistency and overall Quality of Results (QoR) [11].

Challenges

- **Synthesizability:** AI-generated HDL may include non-synthesizable constructs and thus require human review [10].
- **Data Scarcity:** The training data of limited size, especially for VHDL, restricts the generalization capability of the model [9].
- **Physical Constraints:** Most AI models do not consider timing, power, and physical constraints, resulting in syntactically correct but physically infeasible designs [11].
- **Reliability:** The integration of probabilistic AI into deterministic EDA flows raises reliability and verification issues, highlighting the need for human oversight [5, 7].

Conclusion

The project demonstrates the beginnings of the useful role of AI models, such as ChatGPT or Claude, as a "code assistant" in the design of hardware, inferring primitives effectively and accelerating the prototyping stage. In their present form, they demonstrate the lack of system-level architectural insight, leading to the generation of software-ish logic that suffers from inefficiency and, thus, critical physical hazards, including overly heavy I/O and doubled power.

The neglect of important power management strategies, including clock gating, reiterates the incompleteness of these tools vis-à-vis manufacturability and thermal robustness. Nonetheless, the use of AI-flows, backed by academic efforts, industrial tools, and open-source projects, allows the amalgamation of AI training data that includes a device perspective, along with online EDA feedback, to fill the gap between the syntax of the model and the hardware that needs to be built.

Appendix: Source Code Listings

This appendix contains the Verilog and SystemVerilog source code used during the experimental analysis. These listings include both the professional reference designs and the AI-generated implementations for the BRAM and GPIO benchmarks.

A. Professional BRAM Reference (Xilinx Template)

```verilog
// Professional Source: Xilinx Vivado Language Template (Fixed)
// Module: True Dual Port RAM with Dual Clocks
// Standardized values: Width=32, Depth=1024 for benchmarking

module xilinx_true_dual_port_ram #(
    parameter RAM_WIDTH = 32,                       // Fixed: 32-bit data
    parameter RAM_DEPTH = 1024,                     // Fixed: 1024 entries
    parameter RAM_PERFORMANCE = "HIGH_PERFORMANCE", // Select "HIGH_PERFORMANCE" or "LOW_LATENCY"
    parameter INIT_FILE = ""                        // Leave blank
)(
    input [clogb2(RAM_DEPTH-1)-1:0] addra,  // Port A address bus
    input [clogb2(RAM_DEPTH-1)-1:0] addrb,  // Port B address bus
    input [RAM_WIDTH-1:0] dina,             // Port A RAM input data
    input [RAM_WIDTH-1:0] dinb,             // Port B RAM input data
    input clka,                             // Port A clock
    input clkb,                             // Port B clock
    input wea,                              // Port A write enable
    input web,                              // Port B write enable
    input ena,                              // Port A RAM Enable
    input enb,                              // Port B RAM Enable
    input rsta,                             // Port A output reset
    input rstb,                             // Port B output reset
    input regcea,                           // Port A output register enable
    input regceb,                           // Port B output register enable
    output [RAM_WIDTH-1:0] douta,           // Port A RAM output data
    output [RAM_WIDTH-1:0] doutb            // Port B RAM output data
);

    // Scaling function for address width
    function integer clogb2;
    input integer depth;
    for (clogb2=0; depth>0; clogb2=clogb2+1)
        depth = depth >> 1;
    endfunction

    // 2D Array for RAM storage
    reg [RAM_WIDTH-1:0] BRAM [RAM_DEPTH-1:0];
    reg [RAM_WIDTH-1:0] ram_data_a = {RAM_WIDTH{1'b0}};
    reg [RAM_WIDTH-1:0] ram_data_b = {RAM_WIDTH{1'b0}};

    // Initialize memory to zero
    generate
        if (INIT_FILE != "") begin: use_init_file
            initial
```

```verilog
                $readmemh(INIT_FILE, BRAM, 0, RAM_DEPTH-1);
        end else begin: init_bram_to_zero
            integer ram_index;
            initial
                for (ram_index = 0; ram_index < RAM_DEPTH; ram_index = ram_index + 1)
                    BRAM[ram_index] = {RAM_WIDTH{1'b0}};
        end
    endgenerate

    // PORT A OPERATION
    always @(posedge clka) begin
        if (ena) begin
            if (wea)
                BRAM[addra] <= dina;
            ram_data_a <= BRAM[addra];
        end
    end

    // PORT B OPERATION
    always @(posedge clkb) begin
        if (enb) begin
            if (web)
                BRAM[addrb] <= dinb;
            ram_data_b <= BRAM[addrb];
        end
    end

    // Output Register Stage
    generate
        if (RAM_PERFORMANCE == "LOW_LATENCY") begin: no_output_register
            assign douta = ram_data_a;
            assign doutb = ram_data_b;
        end else begin: output_register
            reg [RAM_WIDTH-1:0] douta_reg = {RAM_WIDTH{1'b0}};
            reg [RAM_WIDTH-1:0] doutb_reg = {RAM_WIDTH{1'b0}};

            always @(posedge clka) begin
                if (rsta)
                    douta_reg <= {RAM_WIDTH{1'b0}};
                else if (regcea)
                    douta_reg <= ram_data_a;
            end

            always @(posedge clkb) begin
                if (rstb)
                    doutb_reg <= {RAM_WIDTH{1'b0}};
                else if (regceb)
                    doutb_reg <= ram_data_b;
            end

            assign douta = douta_reg;
            assign doutb = doutb_reg;
        end
    endgenerate

endmodule
```
Listing 1: Xilinx True Dual Port RAM Template

B. AI-Generated BRAM (ChatGPT-5.2)

```systemverilog
module true_dual_port_ram #(
    parameter int RAM_WIDTH = 32,
    parameter int RAM_DEPTH = 1024,
    parameter string RAM_PERFORMANCE = "LOW_LATENCY", // or "HIGH_PERFORMANCE"
    parameter string INIT_FILE = ""                    // optional memory init file
)(
    // Port A
    input  logic                         clka,
    input  logic                         wea,
    input  logic [$clog2(RAM_DEPTH)-1:0] addra,
    input  logic [RAM_WIDTH-1:0]          dina,
    output logic [RAM_WIDTH-1:0]          douta,

    // Port B
    input  logic                         clkb,
    input  logic                         web,
    input  logic [$clog2(RAM_DEPTH)-1:0] addrb,
    input  logic [RAM_WIDTH-1:0]          dinb,
    output logic [RAM_WIDTH-1:0]          doutb,

    // Reset (used only for output registers)
    input  logic                         rst
);

    localparam int ADDR_WIDTH = $clog2(RAM_DEPTH);

    // Memory array
    logic [RAM_WIDTH-1:0] ram [0:RAM_DEPTH-1];

    // Internal read data (pre-output register)
    logic [RAM_WIDTH-1:0] douta_int;
    logic [RAM_WIDTH-1:0] doutb_int;

    // ------------------------------------------------------------
    // Optional memory initialization
    // ------------------------------------------------------------
    generate
        if (INIT_FILE != "") begin : gen_init_file
            initial $readmemh(INIT_FILE, ram);
        end else begin : gen_init_zero
            integer i;
            initial begin
                for (i = 0; i < RAM_DEPTH; i = i + 1)
                    ram[i] = '0;
            end
        end
    endgenerate

    // ------------------------------------------------------------
    // Port A - Read-First behavior
    // ------------------------------------------------------------
    always_ff @(posedge clka) begin
        // Read old data first
        douta_int <= ram[addra];

        // Write after read
        if (wea) begin
            ram[addra] <= dina;
        end
    end

    // ------------------------------------------------------------
    // Port B - Read-First behavior
    // ------------------------------------------------------------
    always_ff @(posedge clkb) begin
        // Read old data first
        doutb_int <= ram[addrb];

        // Write after read
        if (web) begin
            ram[addrb] <= dinb;
        end
    end

    // ------------------------------------------------------------
    // Output register stage (performance selection)
    // ------------------------------------------------------------
    generate
        if (RAM_PERFORMANCE == "HIGH_PERFORMANCE") begin : gen_high_perf
            // Extra registered output stage (2-cycle latency)
            always_ff @(posedge clka) begin
                if (rst)
                    douta <= '0;
                else
                    douta <= douta_int;
            end

            always_ff @(posedge clkb) begin
                if (rst)
                    doutb <= '0;
                else
                    doutb <= doutb_int;
            end
        end else begin : gen_low_latency
            // Direct output (1-cycle latency)
            always_comb begin
                douta = douta_int;
                doutb = doutb_int;
            end
        end
    endgenerate

endmodule
```

Listing 2: ChatGPT-Generated Dual Port RAM

C. Professional GPIO Reference (OpenTitan)

```systemverilog
// PROFESSIONAL BENCHMARK: OpenTitan GPIO (Cleaned for Synthesis)
// Source: Derived from lowRISC OpenTitan 'gpio.sv'

// 1. PACKAGE DEFINITION
package gpio_pkg;
  parameter int NumIOs = 32;
endpackage

// 2. MAIN MODULE
module opentitan_gpio
  import gpio_pkg::*;
(
  input  logic                clk_i,
  input  logic                rst_ni,

  // Register Interface (Simplified for Benchmarking)
  input  logic                reg_we,
  input  logic [31:0]         reg_addr,
  input  logic [31:0]         reg_wdata,
  output logic [31:0]         reg_rdata,

  // GPIO Ports
  input  logic [NumIOs-1:0] cio_gpio_i,
  output logic [NumIOs-1:0] cio_gpio_o,
  output logic [NumIOs-1:0] cio_gpio_en_o,

  // Interrupt Output
  output logic [NumIOs-1:0] intr_gpio_o
);

  // Registers
  logic [31:0] data_in_q;
  logic [31:0] direct_out_q;

  // Input Synchronization (Standard Double Flop for Safety)
  logic [31:0] cio_gpio_sync_1;
  logic [31:0] cio_gpio_sync_2;

  always_ff @(posedge clk_i or negedge rst_ni) begin
    if (!rst_ni) begin
        cio_gpio_sync_1 <= '0;
        cio_gpio_sync_2 <= '0;
    end else begin
        cio_gpio_sync_1 <= cio_gpio_i;
        cio_gpio_sync_2 <= cio_gpio_sync_1;
    end
  end

  // Edge Detection Logic
  logic [31:0] event_rise;
  logic [31:0] event_fall;

  always_ff @(posedge clk_i) begin
    data_in_q <= cio_gpio_sync_2;
  end

  assign event_rise = cio_gpio_sync_2 & ~data_in_q;
  assign event_fall = ~cio_gpio_sync_2 & data_in_q;

  // ------------------------------------------------------
  // THE CRITICAL TEST: MASKED WRITES
  // Professional code uses efficient bitwise logic in one cycle.
  // ------------------------------------------------------
  always_ff @(posedge clk_i or negedge rst_ni) begin
    if (!rst_ni) begin
      direct_out_q <= '0;
    end else if (reg_we) begin
      case (reg_addr[7:0])
        8'h00: direct_out_q <= reg_wdata; // Direct Write

        // Masked Lower: Update lower 16 bits based on upper 16 mask
        8'h04: direct_out_q[15:0] <= (direct_out_q[15:0] & ~reg_wdata
   [31:16]) | (reg_wdata[15:0] & reg_wdata[31:16]);

        // Masked Upper: Update upper 16 bits based on upper 16 mask
        8'h08: direct_out_q[31:16] <= (direct_out_q[31:16] & ~reg_wdata
   [31:16]) | (reg_wdata[15:0] & reg_wdata[31:16]);
      endcase
    end
  end

  assign cio_gpio_o = direct_out_q;
  assign cio_gpio_en_o = {32{1'b1}}; // All outputs enabled for this
   test

  // Interrupt Logic (Simplified for synthesis comparison)
  // Hardcoded enables to force synthesis to keep the logic
  logic [31:0] intr_ctrl_en_rising  = 32'hFFFFFFFF;
  logic [31:0] intr_ctrl_en_falling = 32'hFFFFFFFF;

  always_ff @(posedge clk_i or negedge rst_ni) begin
    if (!rst_ni)
      intr_gpio_o <= '0;
    else
      intr_gpio_o <= (event_rise & intr_ctrl_en_rising) |
                     (event_fall & intr_ctrl_en_falling);
  end

endmodule
```

Listing 3: OpenTitan GPIO Benchmark

D. AI-Generated GPIO (Claude Sonnet 4.5)

```
module gpio #( parameter int unsigned GpioWidth = 32 ) (
  input logic clk_i, rst_ni,
  input logic [GpioWidth-1:0] cio_gpio_i,
  output log// Professional GPIO Module with Interrupt Support
// Copyright (c) 2024. All rights reserved.
// Designed for 32-bit General Purpose Input/Output control

module gpio #(
  parameter int unsigned GpioWidth = 32
) (
  input  logic                    clk_i,
  input  logic                    rst_ni,

  // Pad interface
  input  logic [GpioWidth-1:0]    cio_gpio_i,
  output logic [GpioWidth-1:0]    cio_gpio_o,
  output logic [GpioWidth-1:0]    cio_gpio_en_o,

  // Register interface - Direct Write
  input  logic                    direct_out_we_i,
  input  logic [GpioWidth-1:0]    direct_out_data_i,
  input  logic                    direct_oe_we_i,
  input  logic [GpioWidth-1:0]    direct_oe_data_i,

  // Register interface - Masked Write Upper [31:16]
  input  logic                    masked_out_upper_we_i,
  input  logic [15:0]             masked_out_upper_data_i,
  input  logic [15:0]             masked_out_upper_mask_i,
  input  logic                    masked_oe_upper_we_i,
  input  logic [15:0]             masked_oe_upper_data_i,
  input  logic [15:0]             masked_oe_upper_mask_i,

  // Register interface - Masked Write Lower [15:0]
  input  logic                    masked_out_lower_we_i,
  input  logic [15:0]             masked_out_lower_data_i,
  input  logic [15:0]             masked_out_lower_mask_i,
  input  logic                    masked_oe_lower_we_i,
  input  logic [15:0]             masked_oe_lower_data_i,
  input  logic [15:0]             masked_oe_lower_mask_i,

  // Interrupt configuration
  input  logic [GpioWidth-1:0]    intr_ctrl_en_rising_i,
  input  logic [GpioWidth-1:0]    intr_ctrl_en_falling_i,
  input  logic [GpioWidth-1:0]    intr_ctrl_en_lvlhigh_i,
  input  logic [GpioWidth-1:0]    intr_ctrl_en_lvllow_i,

  // Interrupt status (write-1-to-clear)
  input  logic                    intr_state_we_i,
  input  logic [GpioWidth-1:0]    intr_state_data_i,
  output logic [GpioWidth-1:0]    intr_state_o,

  // Interrupt output
  output logic [GpioWidth-1:0]    intr_o
);

  //
    =============================================================================
```

```systemverilog
// Internal Signals
//
 ============================================================================

logic [GpioWidth-1:0] gpio_out_q, gpio_out_d;
logic [GpioWidth-1:0] gpio_oe_q, gpio_oe_d;
logic [GpioWidth-1:0] gpio_in_sync1_q, gpio_in_sync2_q;
logic [GpioWidth-1:0] gpio_in_prev_q;
logic [GpioWidth-1:0] intr_state_q, intr_state_d;

// Edge detection signals
logic [GpioWidth-1:0] rising_edge, falling_edge;

//
 ============================================================================

// Input Synchronization (Double-Flop for Metastability Protection)
//
 ============================================================================

always_ff @(posedge clk_i or negedge rst_ni) begin
  if (!rst_ni) begin
    gpio_in_sync1_q <= '0;
    gpio_in_sync2_q <= '0;
    gpio_in_prev_q  <= '0;
  end else begin
    gpio_in_sync1_q <= cio_gpio_i;
    gpio_in_sync2_q <= gpio_in_sync1_q;
    gpio_in_prev_q  <= gpio_in_sync2_q;
  end
end

//
 ============================================================================

// Edge Detection Logic
//
 ============================================================================

assign rising_edge  = gpio_in_sync2_q & ~gpio_in_prev_q;
assign falling_edge = ~gpio_in_sync2_q & gpio_in_prev_q;

//
 ============================================================================

// GPIO Output Data Register Logic
//
 ============================================================================

always_comb begin
  gpio_out_d = gpio_out_q;

  // Direct write has priority
  if (direct_out_we_i) begin
    gpio_out_d = direct_out_data_i;
  end else begin
    // Masked write upper [31:16]
    if (masked_out_upper_we_i) begin
      for (int i = 0; i < 16; i++) begin
        if (masked_out_upper_mask_i[i]) begin
          gpio_out_d[16+i] = masked_out_upper_data_i[i];
        end
      end
    end

    // Masked write lower [15:0]
    if (masked_out_lower_we_i) begin
      for (int i = 0; i < 16; i++) begin
        if (masked_out_lower_mask_i[i]) begin
          gpio_out_d[i] = masked_out_lower_data_i[i];
        end
      end
    end
  end
end

always_ff @(posedge clk_i or negedge rst_ni) begin
  if (!rst_ni) begin
    gpio_out_q <= '0;
  end else begin
    gpio_out_q <= gpio_out_d;
  end
end

assign cio_gpio_o = gpio_out_q;

// ============================================================================
// GPIO Output Enable Register Logic
// ============================================================================
always_comb begin
  gpio_oe_d = gpio_oe_q;

  // Direct write has priority
  if (direct_oe_we_i) begin
    gpio_oe_d = direct_oe_data_i;
  end else begin
    // Masked write upper [31:16]
    if (masked_oe_upper_we_i) begin
      for (int i = 0; i < 16; i++) begin
        if (masked_oe_upper_mask_i[i]) begin
          gpio_oe_d[16+i] = masked_oe_upper_data_i[i];
        end
      end
    end

    // Masked write lower [15:0]
    if (masked_oe_lower_we_i) begin
      for (int i = 0; i < 16; i++) begin
        if (masked_oe_lower_mask_i[i]) begin
          gpio_oe_d[i] = masked_oe_lower_data_i[i];
        end
      end
    end
  end
end

always_ff @(posedge clk_i or negedge rst_ni) begin
  if (!rst_ni) begin
    gpio_oe_q <= '0;
  end else begin
    gpio_oe_q <= gpio_oe_d;
  end
end

assign cio_gpio_en_o = gpio_oe_q;

// =============================================================================
// Interrupt Detection Logic
// =============================================================================
logic [GpioWidth-1:0] intr_detect;

always_comb begin
  for (int i = 0; i < GpioWidth; i++) begin
    intr_detect[i] = (intr_ctrl_en_rising_i[i]  & rising_edge[i]) |
                     (intr_ctrl_en_falling_i[i] & falling_edge[i]) |
                     (intr_ctrl_en_lvlhigh_i[i] & gpio_in_sync2_q[i]) |
                     (intr_ctrl_en_lvllow_i[i]  & ~gpio_in_sync2_q[i]);
  end
end

// =============================================================================
// Interrupt State Register (Write-1-to-Clear)
// =============================================================================
always_comb begin
  intr_state_d = intr_state_q;

  // Set interrupt bits when detected
  intr_state_d = intr_state_q | intr_detect;

  // Clear interrupt bits on write-1-to-clear
  if (intr_state_we_i) begin
    intr_state_d = intr_state_d & ~intr_state_data_i;
  end
end

always_ff @(posedge clk_i or negedge rst_ni) begin
  if (!rst_ni) begin
    intr_state_q <= '0;
  end else begin
```

```
      intr_state_q <= intr_state_d;
    end
  end

  assign intr_state_o = intr_state_q;
  assign intr_o = intr_state_q;

  //
   ============================================================================

  // Assertions for Verification
  //
   ============================================================================

`ifndef SYNTHESIS
  // Check that direct and masked writes don't occur simultaneously
  always_ff @(posedge clk_i) begin
    if (rst_ni) begin
      assert (!(direct_out_we_i && (masked_out_upper_we_i ||
   masked_out_lower_we_i)))
        else $error("Direct and masked GPIO output writes should not
   occur simultaneously");
      assert (!(direct_oe_we_i && (masked_oe_upper_we_i ||
   masked_oe_lower_we_i)))
        else $error("Direct and masked GPIO OE writes should not occur
   simultaneously");
    end
  end
`endif

endmoduleic [GpioWidth-1:0] cio_gpio_o, cio_gpio_en_o,
  // ... [Additional Interface Ports] ...
);
  // Software-style loop-based masking (Causes Logic Inefficiency)
  always_comb begin
    gpio_out_d = gpio_out_q;
    if (masked_out_lower_we_i) begin
      for (int i = 0; i < 16; i++) begin
        if (masked_out_lower_mask_i[i]) begin
          gpio_out_d[i] = masked_out_lower_data_i[i];
        end
      end
    end
  end
endmodule
```

Listing 4: Claude-Generated GPIO with Loop-Based Masking

Bibliography

[1] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, Aug. 2012.

[2] M. W. Numan, B. J. Phillips, G. S. PudDY, and K. Falkner, "Towards Automatic High-Level Code Deployment on Reconfigurable Platforms: A Survey of High-Level Synthesis Tools and Toolchains," *IEEE Access*, vol. 8, pp. 174692-174722, 2020.

[3] R. Nane et al., "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591-1604, Oct. 2016.

[4] J. Marjanovic, "LOW VS HIGH LEVEL PROGRAMMING FOR FPGA," in *Proc. IBIC2018*, Shanghai, China, 2018, pp. 1-7.

[5] S. Alsaqer, S. Alajmi, I. Ahmad, and M. Alfailakawi, "The potential of LLMs in hardware design," *Journal of Engineering Research*, vol. 13, no. 1, pp. 2392-2404, 2025.

[6] Z. Liu, Y. Dou, J. Jiang, and J. Xu, "Automatic Code Generation of Convolutional Neural Networks in FPGA Implementation," in *Proc. 2016 IEEE International Conference on Parallel and Distributed Processing with Applications*, Changsha, China, 2016, pp. 1-8.

[7] M. Dossis and G. Dimitriou, "Are HLS Tools Healthy? The C-Cubed Project," *Engineering, Technology & Applied Science Research*, vol. 5, no. 2, pp. 790-794, 2015.

[8] Y. Bai et al., "Learning to Compare Hardware Designs for High-Level Synthesis," in *Proc. 2024 ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD '24)*, Salt Lake City, UT, USA, Sep. 2024, pp. 1-7.

[9] P. Vijayaraghavan et al., "Chain-of-Descriptions: Improving Code LLMs for VHDL Code Generation and Summarization," in *Proc. 2024 ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD '24)*, Salt Lake City, UT, USA, Sep. 2024, pp. 1-10.

[10] R. Al Amin, M. G. R. Lincoln, and R. Obermaisser, "Towards LLM-Assisted HDL Generation and Verification," in *Proc. 2025 14th Mediterranean Conference on Embedded Computing (MECO)*, Budva, Montenegro, Jun. 2025, pp. 1-4.

[11] S. Thakur et al., "Autochip: Automating hdl generation using llm feedback," *arXiv preprint arXiv:2311.04887*, 2023.