HLS TO ASIC LAYOUT FLOW DESIGN

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by

Wenbo Li (wl764)

Hao Wu (hw794)

MEng Field Advisor: Professor Zhiru Zhang

Degree Date: May 2025

Abstract

Master of Engineering Program
School of Electrical and Computer Engineering
Cornell University
Design Project Report

Project Title: HLS to ASIC Layout Flow Design

Authors: Wenbo Li (wl764), Hao Wu (hw794)

Abstract:

This project addresses a critical gap in the ASIC design workflow by developing an automated toolchain that bridges High-Level Synthesis (HLS) and physical layout implementation. While HLS tools like Catapult effectively translate C++ code into register-transfer level (RTL) designs, the subsequent conversion to physical layouts traditionally requires extensive manual intervention and specialized knowledge. Our solution provides an end-to-end automated flow from C++ code to GDSII layouts, enabling developers with limited hardware expertise to efficiently create and evaluate manufacturable chip designs. The tooling interface we developed takes user-specified design constraints and automatically generates a complete ASIC layout through integration of Catapult HLS for RTL generation and OpenROAD for physical implementation. This innovation significantly streamlines the hardware design process and makes ASIC design more accessible to software developers and researchers from various fields.

Authors' Contributions

Wenbo Li (wl764)

- Deployed docker image for OpenROAD and write the documents
- Developed the automated connection generation scripts for array architectures
- Developed the top-level integration system for generating Verilog connections
- Created the integration between HLS output and OpenROAD input
- Designed and executed testing procedures for the toolchain
- Wrote documentation for the toolchain usage and implementation

Hao Wu (hw794)

- Designed the overall architecture of the HLS-OpenROAD Driver
- Deployed docker image for OpenROAD and write the documents
- Implemented the constraint management system for translating high-level constraints to physical directives
- Created the configuration system for OpenROAD
- Conducted experiments with different placement strategies
- Wrote documentation for the constraint specification format

Executive Summary

This HLS to ASIC Layout Flow Design bridges the gap between High Level Synthesis (HLS) and physical ASIC implementation. Although HLS developers design hardware using some high level languages like C++ or Python and then convert them into RTL design, they still need to integrate these submodules and IPs into a top module design, and then send them into physical ASIC flow to generate the final GDS file. During these steps especially when they send the design into the physical ASIC flow, they should add clock, floorplan and etc constraints at some stages, which many of the HLS developers may not be quite familiar with.

Our key accomplishments include:

- Development of an HLS-OpenROAD Driver that automates the entire flow from C++ code to GDSII layout files ready for chip tape-out
- Creation of a flexible user interface that allows developers to specify designs in C++ and constraints and connection relationships through simple JSON files
- Implementation of an automated connection generation system for array architectures based on user-specified dimensions in YAML format
- Successful demonstration of the toolchain through implementation of a 2×2 systolic array design with various layout strategies

The primary challenges we encountered during development included:

- Difficulty deploying OpenROAD with docker and making it run ASIC flow successfully
- Learning what constraints and how to add them to the ASIC flow
- Complexity in translating high-level constraints to low-level physical directives
- Challenges in connecting HLS and ASIC flow

For future work, we plan to implement two significant enhancements to our toolchain. First, we will encapsulate the functionality into Python classes, allowing users to define module connections and physical constraints through direct method calls rather than JSON files.

This approach will enable users to programmatically declare submodule connection relationships and physical constraints by simply calling functions, eliminating the need to write complex configuration files. Second, we will develop scripts to support a wider range of circuit architectures beyond systolic arrays. These scripts will help users more easily generate connection relationships for various hardware designs, including tree-based structures, mesh networks, pipeline architectures, and custom topologies.

Contents

Authors' Contributions **2**

Executive Summary **3**

1 Introduction and Background **7**

1.1 Background 7

1.1.1 High Level Synthesis(HLS) 7

1.1.2 OpenROAD ASIC Flow 8

1.2 Problem Statement 8

1.3 Project Objectives 9

1.4 Approach Overview 9

2 Methodology and Implementation **11**

2.1 System Architecture Overview 11

2.2 Implementation Details 11

2.2.1 HLS to RTL Conversion 12

2.2.2 Configuration and Constraint Management 12

2.2.3 OpenROAD Integration 13

2.2.4 Demonstration Implementation 14

2.3 Testing Methodology 14

3 Results and Discussion **16**

3.1 Layout Generation Results 16

3.1.1 Hardened Macro Manual Placement 16

3.1.2 Hardened Macro Auto Placement with I/O Constraints 18

3.1.3 Unconstrained Array Implementation 19

3.1.4 Simplified User Interface 20
3.1.5 Automated Connection Generation 20
3.1.6 Flexible Constraint Injection 20
3.1.7 End-to-End Automation 20
3.2 Challenges and Solutions 21
3.2.1 Docker GUI Issues 21
3.2.2 Constraint Translation 21
3.2.3 Module Placement 21
3.2.4 HLS Integration 21

4 Conclusions and Future Work **22**
4.1 Project Achievements 22
4.2 Lessons Learned 22
4.3 Future Work 23

5 Acknowledgements **24**

6 References **25**

A Code Repository **26**
A.1 Github Repo 26

B User Guide **27**
B.1 Installation 27
B.2 Analyze Results 27

Chapter 1

Introduction and Background

1.1 Background

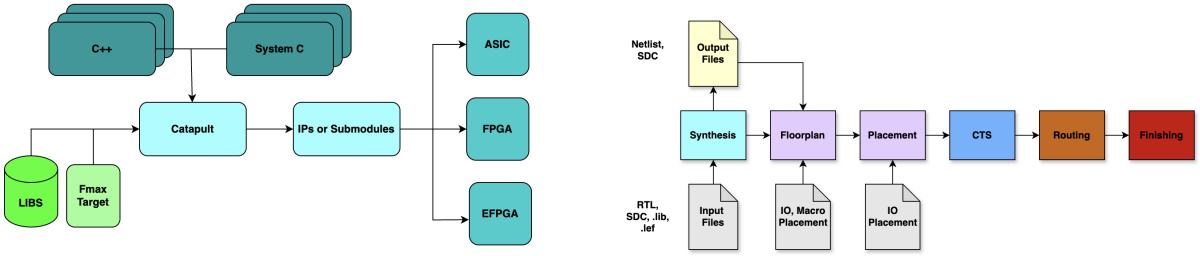
The field of ASIC design traditionally requires a deep understanding of hardware description languages (HDLs) such as Verilog or VHDL. However, high-level synthesis (HLS) tools allow software engineers to design hardware using high-level programming languages such as C++. While HLS tools like Catapult can generate RTL code from C++, the conversion of this RTL into a physical layout for manufacturing has not been fully automated. Currently, there is no unified toolchain that allows users to automatically convert an HLS-based design to a physical ASIC layout with minimal manual intervention. The absence of such a toolchain limits the potential for rapid hardware design and prototyping.

1.1.1 High Level Synthesis(HLS)

High-Level Synthesis(HLS) enables designers to automatically translate C or C++ code into synthesizable Verilog RTL. Catapult HLS, developed by Siemens EDA (formerly Mentor Graphics), provides a comprehensive environment for this translation process. It allows software engineers and algorithm developers to design hardware using familiar high-level programming languages rather than hardware description languages like Verilog or VHDL. The tool performs automatic analysis of the code, extracts parallelism, and generates optimized hardware implementations with consideration for area, timing, and power constraints. Catapult HLS supports various target technologies including ASICs, FPGAs, and eFPGAs, making it versatile for different application domains. This approach significantly accelerates the hardware design process and makes it accessible to developers with limited hardware design expertise.

1.1.2 OpenROAD ASIC Flow

OpenROAD provides an open-source RTL-to-GDSII flow, automating the physical design process including synthesis, floorplanning, placement, clock tree synthesis (CTS), routing, and finishing steps like DRC/LVS checking. As part of the DARPA-funded OpenROAD project, it aims to democratize chip design by reducing the barriers to physical implementation. The flow takes synthesizable RTL as input, along with physical constraints such as SDC timing files, I/O placements, and macro placements, and produces manufacturable layouts. Unlike commercial EDA tools that can cost millions in licensing fees, OpenROAD enables academic and industry users to complete physical design without expensive proprietary software. Its modular architecture allows for continuous improvement by the open-source community while maintaining compatibility with industry-standard file formats and design methodologies.



(a) Catapult HLS (b) OpenROAD

Figure 1.1: High-Level Synthesis and Physical Design Tools

1.2 Problem Statement

While HLS tools efficiently generate RTL from high-level languages, and physical design tools like OpenROAD can produce layouts from RTL, there exists a significant gap between these two stages. Designers must manually prepare RTL outputs from HLS and add different constraints for physical implementation, requiring specialized knowledge of both domains.

- **Physical Constraint Specification:** HLS focuses primarily on functional design and performance optimization, but does not address physical layout constraints essential for manufacturing. Identifying the appropriate stage to inject these physical constraints presents a major challenge.
- **Complexity of Tool Integration:** Setting up and managing the interaction between various tools presents significant technical challenges, especially for HLS developers without extensive hardware design experience. The differences in input formats, optimization objectives, and configuration parameters complicate the creation of a unified design flow.

- **Barrier to Entry:** The specialized hardware development knowledge required for physical implementation effectively creates a barrier that prevents software engineers and algorithm developers from fully leveraging HLS capabilities for end-to-end hardware design.

These challenges highlight the need for an automated, integrated toolchain that bridges the gap between HLS-generated RTL and physical layout implementation, enabling a more accessible and efficient hardware design process.

1.3 Project Objectives

In response to these challenges, our porject established the following objectives:

- **End-to-End Automation:** Develop a toolchain that seamlessly connects high-level C++ designs to physical GDSII layouts, eliminating manual intervention in intermediate stages.
- **User-Friendly Interface:** Create an intuitive interface that allows users with minimal hardware design expertise to specify their designs and constraints effectively.
- **Flexible Constraint Management:** Develop methods for users to specify physical design constraints like IO constraints or floorplan constraints at an appropriate level of abstraction, with automatic translation to tool-specific directives.
- **Tool Integration:** Implement robust integration between Catapult HLS and OpenROAD ASIC Flow to ensure smooth data flow and consistent optimization.

1.4 Approach Overview

Our approach to solving these challenges focused on creating an HLS-OpenROAD Driver orchestrating the entire flow from C++ to GDSII, managin the complexities of tool interaction and data transformation. And the key elements of our approach include:

- **User Input Processing:** Accepting C++ module designs and connection specifications and physical constraints in JSON format, with specialized support for array architectures through simplified YAML configuration.
- **RTL Generation:** Utilizing Catapult HLS to transform C++ designs into synthesizable Verilog RTL.
- **Top-Level Integration:** Automatically generating top-level Verilog to connect individual modules based on specified connection relationships written in JSON file.

- **Constraint Translation:** Converting user-specified high-level constraints into detailed directives for OpenROAD and send them into the flow at the right stage, including floorplanning, I/O assignment, and timing requirements.

Chapter 2

Methodology and Implementation

2.1 System Architecture Overview

We developed HLS-OpenROAD Driver, bridging high-level design and physical implementation. The system architecture consists of several key components:

- **Input File:** C++ module designs, specification of connection relationships and physical constraints in JSON file and for systolic array architecture, we developed a script that generates connections based on user-specified array dimensions in YAML file.
- **Constraint Management:** Processes user-defined physical constraints such as placement information, I/O pin locations and timing requirements.
- **Submodule RTL Generation:** Uses Catapult HLS to translate C++ design into submodule Verilog RTL.
- **Top-Level Integration:** Automatically generates top-level Verilog by connecting the submodules based on the specified relationships.
- **Physical Design:** Coordiates with OpenROAD to perform synthesis, floorplanning, placement, clock tree synthesis and routing to produce the final layout.

2.2 Implementation Details

Our HLS-OpenROAD Driver is designed to support and automate the complete design flow by handling three key phases: high-level synthesis, constraint configuration, and physical implementation. This framework streamlines the process of generating the final design layout in an efficient and repeatable manner. The following sections provide a detailed explanation of how each phase is managed within our driver, as well as how these phases are integrated to enable a fully automated flow from RTL to layout.

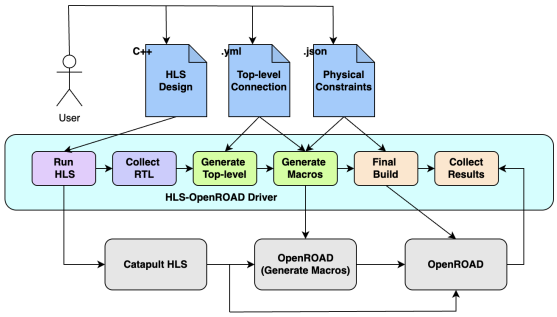


Figure 2.1: HLS-OpenROAD Driver System Architecturen

2.2.1 HLS to RTL Conversion

We first utilized Catapult HLS to transform C++ designs into synthesizable RTL code. This process involved configuring the tool with appropriate HLS directives, constraints, and architectural settings to guide the generation of optimized RTL code tailored to the target design requirements and performance goals. This involved:

- Setting up appropriate HLS directives and constraints
- Configuring the tool to generate optimized RTL for the target architecture

Once the RTL modules were generated, we developed Python scripts to automatically extract port information from the individual sub-modules produced by Catapult HLS. This port information, including signal names, directions, and widths, is essential for the subsequent integration steps.

Using the extracted port data, combined with the user-provided connection relationships between modules, our scripts automatically construct the top-level RTL wrapper. This wrapper consolidates the sub-modules into a unified design hierarchy, ensuring correct interconnections and interface definitions. The generated top-level RTL serves as the input for the OpenROAD flow, enabling a smooth transition from high-level synthesis to physical design implementation.

2.2.2 Configuration and Constraint Management

We developed a flexible configuration system to extract user's target from json file, includes:

- Extracts necessary information from JSON file. Parses essential configuration data such as clock period, library paths, macro placement constraints, and I/O pin assignments. Ensures that all design-specific constraints are systematically captured and prepared for downstream processes.

- Generates customized TCL scripts based on the extracted information to guide key OpenROAD tasks, including macro placement (macro_placement.tcl), floorplanning, and pin placement. Supports rapid iteration and easy modification by decoupling configuration from the core flow. Translates high-level physical constraints into detailed placement directives in SDC file
- Converts user-defined high-level physical constraints into detailed placement directives and timing constraints in SDC format, ensuring alignment between intended design specifications and tool-level implementations.
- Creates all necessary Makefile configuration files (config.mk, block.mk) to manage and automate the entire flow efficiently. Facilitates consistent and reproducible builds across different design iterations and configurations.
- Leverages YAML files containing array dimension information to automatically generate accurate connection relationships for systolic arrays. Simplifies the integration of complex, regular structures by automating the generation of interconnects and module instantiations.

2.2.3 OpenROAD Integration

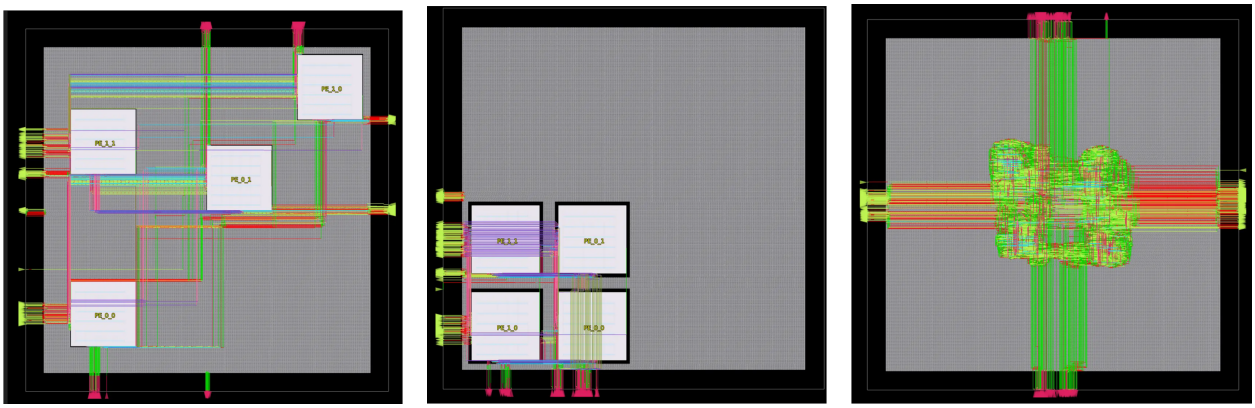
Our toolchain seamlessly integrates with the OpenROAD flow to perform key physical design tasks in a fully automated manner, while still allowing flexibility for user-defined constraints and design preferences. The integrated flow supports the following functionalities:

- Floorplanning with user-specified constraints. Automates the floorplan generation based on user-defined die size, core utilization, and aspect ratio settings. Applies placement blockages, halos, and keep-out zones as defined in the configuration files to enforce design rules and accommodate system-level integration requirements.
- Macro placement with support for both automated and manual options. Supports both automated macro placement using OpenROAD's built-in algorithms and manual placement through user-defined scripts. Ensures macro alignment, spacing, and orientation to meet design and manufacturability guidelines.
- I/O pin assignment with custom location constraints. Automates I/O pin placement according to user-specified edge constraints (e.g., restricting pins to specific sides of the die). Facilitates early exploration of I/O planning to evaluate feasibility and optimize routing congestion around critical regions.
- Clock tree synthesis and optimization. Integrates clock tree synthesis and performs clock optimization to meet user-defined timing constraints. Ensures balanced clock distribution while minimizing skew and insertion delay.
- Executes detailed routing and ensures compliance with technology-specific design rules. Performs automatic design rule checking (DRC) and generates reports to verify the correctness and manufacturability of the final layout.

2.2.4 Demonstration Implementation

For demonstration purposes, we implemented a Processing Element (PE) design in C++ and used our toolchain to automatically generate a 2×2 systolic array design with defined dimension in the YAML file and constraints information in the JSON file. And we explored different constraint settings to demonstrate the flexibility of our approach:

1. Manually placed hardened macros by specifying (x,y) coordinates in the die area
2. Automatically placed hardened macros based on optimization
3. Flatten design without hardening macros



(a) Manually Placed Hardened Macros (b) Automatically Placed Hardened Macros (c) Flatten Design without Hardening Macros

Figure 2.2: Different Implementations of 2×2 Systolic Array

2.3 Testing Methodology

Since the implementations we made for all the demos are only observable in the final physical layout—such as the placement of macros, overall chip area, I/O pin locations, and utilization metrics—it is necessary to re-run the entire flow after each change. We then verify the results by inspecting the generated layout diagrams to ensure that the updates are correctly applied. This visual inspection allows us to confirm whether the new floorplanning or placement constraints have been properly configured and integrated into the flow. Given that these changes are not easily reflected in earlier design stages or reports, this step is essential to validate the correctness of our setup.

When implementing designs with higher complexity, we can also verify the constraint settings by reviewing the generated log and report files. These files provide detailed information that allows us to confirm whether the number of macros, timing constraints, and physical layout constraints have been correctly applied and satisfied. This approach complements visual

inspection of the layout and provides an additional layer of validation for the correctness of the design flow.

Chapter 3

Results and Discussion

3.1 Layout Generation Results

Our toolchain successfully generated multiple layout architectures under varying constraint settings, demonstrating its flexibility and adaptability to different design scenarios. The clock constraint shown below was consistently applied across all three implementations to ensure a fair and consistent comparison of performance, area, and power characteristics.

```
# Auto-generated constraint.sdc
set clk_name       clk
set clk_port_name  clk
set clk_period     475
set clk_io_pct     0.3

set clk_port [get_ports $clk_port_name]

create_clock -name $clk_name -period $clk_period $clk_port

set non_clock_inputs [lsearch -inline -all -not -exact [all_inputs] $clk_port]

set_input_delay  [expr $clk_period * $clk_io_pct] -clock $clk_name $non_clock_inputs
set_output_delay [expr $clk_period * $clk_io_pct] -clock $clk_name [all_outputs]
```

Figure 3.1: clk_period

3.1.1 Hardened Macro Manual Placement

By packing each PE as a hardened macro with specified (x, y) coordinates, we achieved a highly controlled layout with predictable timing characteristics. This approach is particularly useful when hardware designers require full control over physical placement to meet specific design requirements. For example, when integrating the design into a larger system, designers may need to fit it within a specific architecture or place it in a designated area of the die to enable further analysis or optimization. The following figure illustrates the configuration of manually setting up macro locations by specifying their (x, y) coordinates.

```
place_macro -macro_name PE_0_0 -location {25 25}
place_macro -macro_name PE_0_1 -location {100 100}
place_macro -macro_name PE_1_0 -location {150 150}
place_macro -macro_name PE_1_1 -location {25 120}
```

Figure 3.2: manual_macro_placement

```
# Auto-generated config.mk
export PLATFORM              = asap7
export DESIGN_NAME           = SystolicArray
export DESIGN_NICKNAME       = pe_hls
export VERILOG_FILES         = $(sort $(wildcard $(DESIGN_HOME)/src/pe_hls/*.v))
export SDC_FILE              = $(DESIGN_HOME)/asap7/pe_hls/constraint.sdc

export PLACE_DENSITY         = 0.55
export CORE_ASPECT_RATIO     = 1.0
export RTLMP_BOUNDARY_WT = 0
export RTLMP_FLOW ?= 1
export BLOCKS ?= pe
export SYNTH_HIERARCHICAL = 1
export MACRO_PLACEMENT_TCL   = $(DESIGN_HOME)/asap7/pe_hls/macro-placement.tcl
export CORE_MARGIN           = 10
export DIE_AREA  = 0 0 200 200
export CORE_AREA = 10 10 190 190
```

Figure 3.3: manual_macro_placement

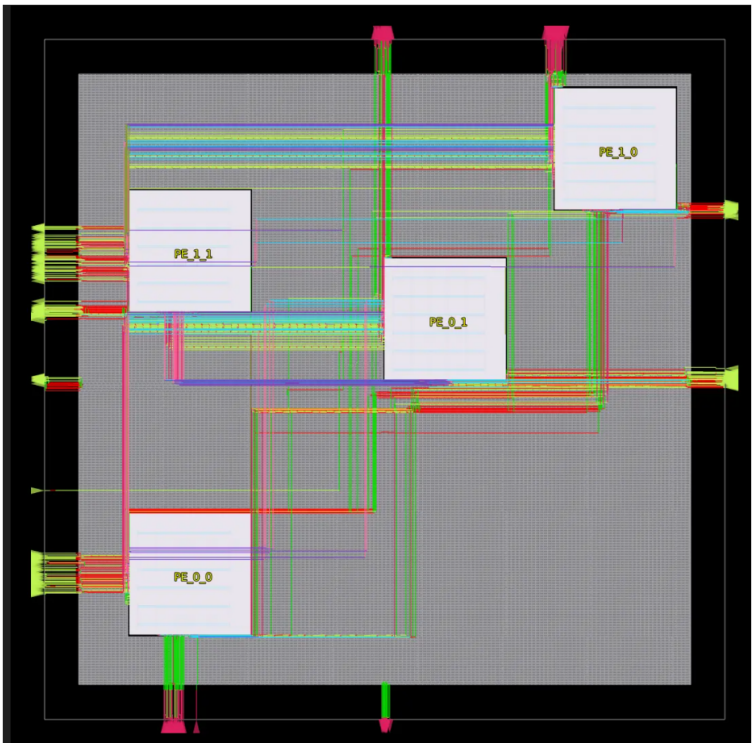


Figure 3.4: Manually Placed Hardened Macros

3.1.2 Hardened Macro Auto Placement with I/O Constraints

Limiting I/O pins to specific edges (left and bottom only) while allowing OpenROAD to optimize internal placement demonstrated the flexibility of our approach. This feature is particularly important for designs with placement constraints. For example, if an IP block only has access to I/O pins on certain edges, it is beneficial to perform synthesis and floorplanning under these constraints to better understand the design's actual performance and conduct more accurate evaluations. The following figure illustrates the configuration of setting up macro locations automatically.

```
# Auto-generated config.mk
export PLATFORM              = asap7
export DESIGN_NAME           = SystolicArray
export DESIGN_NICKNAME       = pe_hls
export CORE_UTILIZATION      = 55
export PLACE_DENSITY         = 0.75
export CORE_ASPECT_RATIO     = 1.0
export CORE_MARGIN           = 2
export VERILOG_FILES         = $(sort $(wildcard $(DESIGN_HOME)/src/pe_hls/*.v))
export SDC_FILE              = $(DESIGN_HOME)/asap7/pe_hls/constraint.sdc

export BLOCKS ?= PE
export SYNTH_HIERARCHICAL = 1
export PLACE_PINS_ARGS = -annealing
```

Figure 3.5: auto_macro_placement

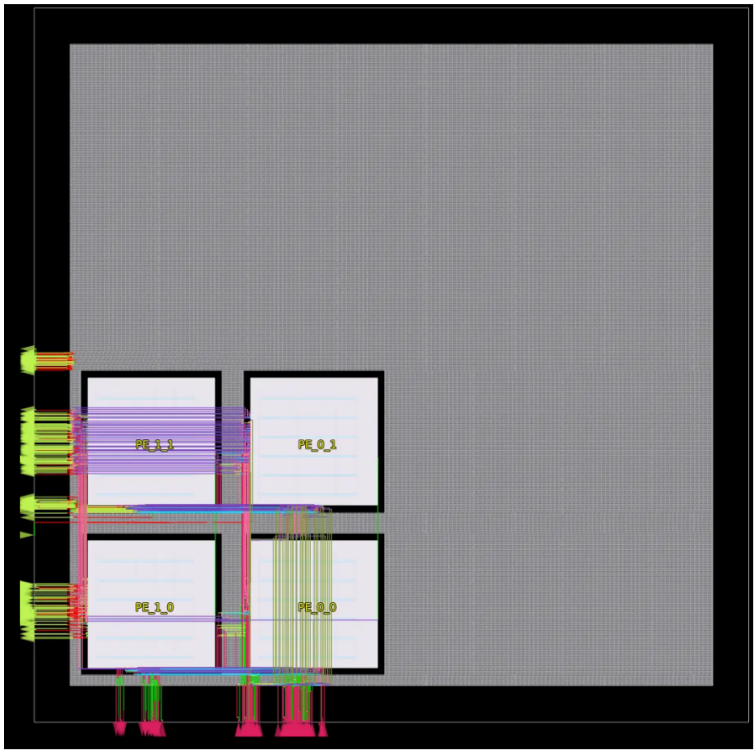


Figure 3.6: I/O Pin Constraints on Left and Bottom Edges

3.1.3 Unconstrained Array Implementation

Generating a 2×2 systolic array without specifying macros showcased the tool's capability to handle complex designs through automated optimization. This is particularly valuable when engineers want to quickly obtain an initial estimation of the design's area, power, and performance at an early stage of the design cycle. Such early-stage evaluations enable more informed architectural decisions, guide design space exploration, and help identify potential bottlenecks before committing to detailed physical design and floorplanning. The following figure illustrates the configuration of floorplaning the same design without using macro.

```
export PLATFORM               = asap7

export VERILOG_FILES = $(sort $(wildcard ./designs/src/pe_hls/*.v))
export SDC_FILE      = ./designs/$(PLATFORM)/pe_hls/constraint.sdc

export CORE_UTILIZATION       = 50
export CORE_ASPECT_RATIO      = 1
export CORE_MARGIN            = 2
export PLACE_DENSITY          = 0.80

export PLACE_PINS_ARGS = -annealing
```

Figure 3.7: no_macro

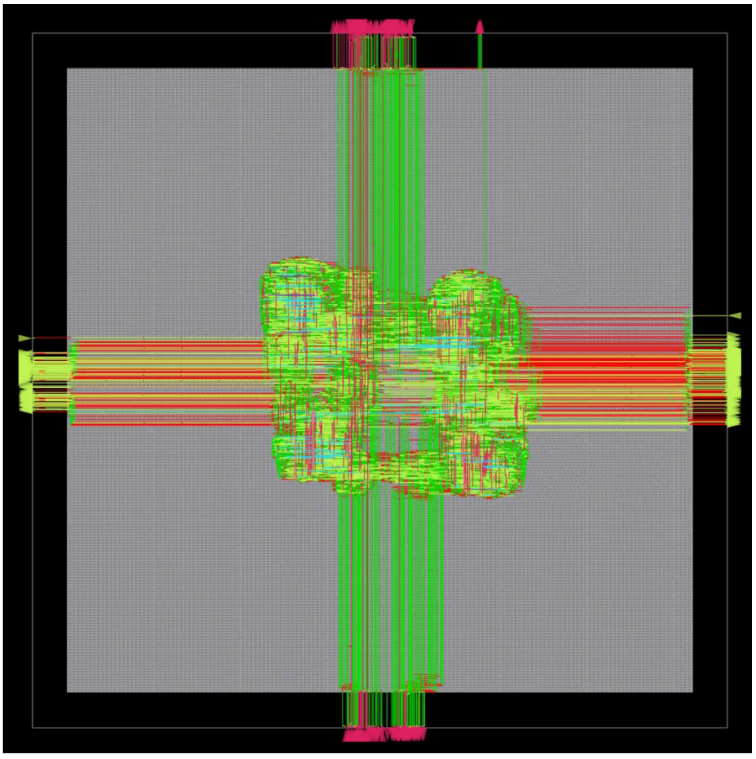


Figure 3.8: 2×2 Systolic Array without Macro Specification

The layouts demonstrated good utilization of die area and effective routing, validating our

approach to automated physical design from HLS-generated RTL.

3.1.4 Simplified User Interface

Developers can specify high-level design goals and constraints—such as performance targets, area budgets, and I/O requirements—without needing in-depth knowledge of the complex physical design processes. By abstracting away low-level backend tasks like floorplanning, placement, routing, and constraint setup, the tool significantly lowers the barrier to entry for ASIC design. This empowers software and algorithm developers, as well as system architects, to engage in ASIC prototyping and exploration without requiring extensive expertise in EDA tools or physical implementation methodologies.

3.1.5 Automated Connection Generation

For array-type designs, our toolchain automatically creates the necessary connections based on simple dimensional specifications. Users need only define the array size and the submodules they use, and the scripts will generate the connection JSON file for the next steps to generate the top module that can reduce the information user have to write for the driver.

3.1.6 Flexible Constraint Injection

Users can customize their designs by creating macros, manually placing modules, and defining I/O pin locations to better align with their design intent, with constraints applied at the appropriate design stage. Users only need to specify the key constraints they care about through an easy-to-use interface, while our tool automatically generates the corresponding constraint files and sets up other necessary defaults to meet the minimum requirements for running the back-end physical design flow. For example, users only need to specify the modules they want to implement as macros, along with optional inputs such as macro halo and I/O locations, and our tool will generate all the necessary files to help push the design into the flow.

3.1.7 End-to-End Automation

The complete flow from C++ to GDSII is fully automated, drastically reducing the manual effort traditionally required for ASIC implementation. By streamlining key stages such as high-level synthesis, physical design, and constraint generation, the tool allows designers to focus more on architectural decisions and optimization strategies. This automation not only accelerates the overall design cycle but also enables rapid and iterative design space exploration, helping users efficiently evaluate multiple configurations and achieve better performance, area, and power trade-offs.

3.2 Challenges and Solutions

During the project, we encountered several challenges:

3.2.1 Docker GUI Issues

We faced difficulties with using the GUI interface within Docker containers, but after some trials, we have solved this problem.

3.2.2 Constraint Translation

Translating high-level constraints to low-level physical directives required careful consideration of the design hierarchy. We developed a comprehensive constraint mapping system to address this challenge.

3.2.3 Module Placement

Placing multiple modules into different designated areas presented challenges in hacking the ASIC flow to send SDC file and write tcl scripts to meet the floorplan requirements. Our solution involved reading OpenROAD docs and learn the demos for macro placement, and try to integrate it into our driver.

3.2.4 HLS Integration

Integrating with Catapult HLS required careful configuration and management of tool-specific parameters and relative paths among amount of generated files. We streamlined this process using python scripts.

Chapter 4

Conclusions and Future Work

4.1 Project Achievements

The HLS to ASIC Layout Flow Design project successfully addresses a critical gap in the hardware design workflow by creating an automated toolchain from high-level C++ descriptions to physical chip layouts. Our implementation significantly simplifies the ASIC design and evaluation process while greatly improving efficiency.

The tooling interface we developed enables users with minimal RTL or floorplanning experience to verify and evaluate designs, accelerating development cycles. By automating the transition from high-level descriptions to manufacturable layouts, our project lowers the barriers to entry for ASIC design and enables more rapid prototyping and innovation.

The successful demonstration of generating 2×2 systolic array designs under various constraint scenarios validates our approach and showcases the flexibility of our solution.

4.2 Lessons Learned

Through this project, we learned a lot especially about ASIC flow and HLS. We developed a much more comprehensive understanding of the complete ASIC design flow and for each critial step:

- **Floorplanning:** We learned how to estimate the initial die size based on design requirements and how to select an appropriate aspect ratio to balance area efficiency and routability. Defining the core area properly, while considering utilization targets, proved to be fundamental in creating a solid physical design foundation.
- **Macro Placement:** We discovered the significant influence of macro placement on both performance and area. By experimenting with different placement strategies, we observed how the physical proximity of hardened macros to related logic blocks can reduce critical path delay, improve timing, and minimize routing congestion. We

also learned about the trade-offs between automated and manual macro placement techniques.

- **IO Constraint Specification:** We developed skills in defining precise I/O pin constraints and recognized the importance of considering the physical placement of pins in optimizing routing resources. By strategically excluding certain die edges from pin assignment, we were able to simplify routing complexity, reduce congestion, and enhance the overall layout quality.
- **Clock Tree Synthesis(CTS):** We gained hands-on experience in CTS, understanding its role as a pivotal stage for distributing clock signals uniformly while managing clock skew and insertion delay. We saw how improper CTS could lead to severe timing violations, and we explored how clock tree optimization techniques contribute to achieving timing closure in the final design.
- **Placement and Routing:** We learned about the trade-offs in placement density, congestion management and detail routing techniques

Also, we learned a lot about how different physical constraints significantly affect design outcomes.

4.3 Future Work

Our next steps include:

1. **HLS Familiarity and Integration:** Further explore HLS capabilities and improve the integration between HLS and physical design tools.
2. **Python Interface Development:** Encapsulate the scripts and functions into python class, allowing users to directly call methods to define connections and set up constraints without writing the information in YAML or JSON files.
3. **Connection Setup Optimization:** Further optimize the connection setup scripts to flexibly support not only systolic array designs but also more general hardware architectures to make user not have to write much information in the connection JSON file

Chapter 5

Acknowledgements

We thank our supervisors, Professor Zhiru Zhang and PhD Student Yixiao Du, for their support and guidance throughout this project. We also acknowledge the developers of OpenROAD for answering our questions, providing valuable feedback, and helping us better understand how OpenROAD works and how to integrate it with our project. Finally, we appreciate the M.Eng program coordinators for providing the resources that made this project possible.

Use of AI-Powered Tools for Language Enhancement and Document Polishing Throughout the preparation of this report, AI-powered tools (ChatGPT) were employed for grammar checking, sentence refinement, allowing us to focus on technical content while maintaining high-quality language standards.

Chapter 6

References

1. A. B. Kahng, L. He, M. Kim, S. Kang, R. Varma, "OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Toolchain," Proceedings of the 56th Annual Design Automation Conference (DAC), 2019.

2. B. Nikolic, A. Iyer, K. A. Shah, "Systematic Design and Automation of High-Level Synthesis Flows for FPGAs and ASICs," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 38, no. 11, pp. 2040-2052, 2019.

3. OpenROAD Project, "OpenROAD Flow: User Guide and Documentation," 2024. [Online]. Available: `https://openroad.readthedocs.io/`

4. Siemens EDA, Catapult High-Level Synthesis (HLS), Siemens Digital Industries Software, 2024. [Online]. Available: `https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis-hls/`

Appendix A

Code Repository

A.1 Github Repo

The project repository is available on GitHub at: https://github.com/hw794/HLS-OpenROAD-flow.git

Appendix B

User Guide

The full user guidelines and manual can be found in the README.md file and the documentation section of the GitHub repository.

B.1 Installation

To install and set up the HLS to ASIC Layout Flow toolchain:

1. **Prerequisites:**
    - Catapult HLS (version 10.5 or later)
    - OpenROAD (version 2.0 or later)
    - Python 3.6 or later
    - Docker (for containerized execution)

B.2 Analyze Results

The generated outputs will include:

- Synthesized RTL for each module
- Top-level Verilog with all connections
- OpenROAD scripts and constraint files
- GDSII layout file
- Timing, area, and power reports