FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Extending an Open-Source ASIC Toolchain to Support Near-Data-Processing Accelerators

Rodrigo Ribeiro Novais Alves

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. João Paulo de Castro Canas Ferreira

July 28, 2025

Abstract

In modern data-intensive applications, such as artificial intelligence, bioinformatics, and financial modeling, there is a demand for high-performance hardware to overcome the inefficiencies of traditional computing architectures. These inefficiencies stem from the significant disparity between Central Processing Unit (CPU) processing speeds and memory access times, resulting in substantial delays and increased energy costs due to data movement. This dissertation addresses these challenges by extending the open-source ASIC design toolchain, OpenLane 2, to support the development and integration of Near-Data-Processing Accelerators (NDPAccs). These accelerators minimize data movement by performing computations near memory, utilizing Static Random Access Memory (SRAM) for its low latency and high throughput, and enabling integration with other computational units.

The proposed approach automates critical design steps preceding and subsequent to the execution of OpenLane 2 while enabling the generation of parametrized accelerators tailored to specific workloads. A partial evaluation of the implemented designs, which includes a thorough assessment of key metrics such as power efficiency, timing delays, and silicon area utilization, provides a robust validation of the research. Results demonstrate that the developed flow can fully characterize the designs while having a robust infrastructure to support their development.

The base flow of the implementation, OpenLane 2, is a comprehensive stack of tools that take a design from RTL to GDSII. However, it lacks summarized power estimation and maximum operating frequencies. For that reason, this work introduces new capabilities to hardware accelerator design, making advanced methodologies accessible and fostering scalable, energy-efficient solutions for data-driven industries. It lays a solid foundation for future advancements in near-data-processing, addressing the computational demands of modern applications.

Resumo

As aplicações modernas com grande volume de dados, como a inteligência artificial, a bioinformática e a modelação financeira, exigem hardware de elevado desempenho para ultrapassar as ineficiências das arquitecturas de computação tradicionais. Essas ineficiências resultam da disparidade significativa entre as velocidades de processamento da Unidade Central de Processamento (CPU) e os tempos de acesso à memória, resultando em atrasos substanciais e aumento dos custos de energia devido à movimentação de dados. Esta dissertação aborda estes desafios alargando a cadeia de ferramentas de conceção de ASIC de código aberto, OpenLane 2, para apoiar o desenvolvimento e a integração de Near-Data-Processing Accelerators (NDPAccs). Estes aceleradores minimizam a movimentação de dados, efectuando cálculos perto da memória, utilizando a memória estática de acesso aleatório (SRAM) devido à sua baixa latência e elevado débito, e permitem a integração com outras unidades computacionais.

A abordagem proposta automatiza etapas críticas de projeto que precedem e sucedem a execução do OpenLane 2, permitindo gerar aceleradores parametrizados adaptados a cargas de trabalho específicas. Uma avaliação parcial dos designs implementados, que inclui uma avaliação minuciosa dos principais parâmetros, como a eficiência energética, os atrasos de tempo e a utilização da área de silício, fornece uma validação sólida da investigação. Os resultados demonstram que o fluxo desenvolvido pode caraterizar completamente os projetos, ao mesmo tempo que dispõe de uma infraestrutura robusta para apoiar o seu desenvolvimento.

O fluxo base da implementação, OpenLane 2, é um conjunto abrangente de ferramentas que transforma um design de RTL em GDSII. Contudo, carece de relatórios de consumo de energia e de frequências máximas de operação. Por essa razão, este trabalho introduz novas capacidades no design de aceleradores em hardware, tornando metodologias avançadas acessíveis e fomentando soluções escaláveis e eficientes para indústrias orientadas para dados. Assim, fornece uma base sólida para futuros avanços em near-data-processing, respondendo às exigências computacionais das aplicações modernas.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

When we evaluate this dissertation through the framework of the United Nations Sustainable Development Goals (SDGs), its ability to inspire significant change stands out. This impact is not limited to pushing the boundaries of technological progress; it also involves reinforcing durable infrastructure, improving safety in urban environments, and ensuring broader access to digital resources. This evaluation demonstrates how the flow for NDPAcc development empowers a more sustainable and inclusive society.

The specific Sustainable Development Goals mentioned have the following names:

SDG 7 Ensure access to affordable, reliable, sustainable and modern energy for all

- **Target 7.3** By 2030, double the global rate of improvement in energy efficiency

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

- **Target 9.5** Enhance scientific research, upgrade the technological capabilities of industrial sectors in all countries, in particular developing countries, including, by 2030, encouraging innovation and substantially increasing the number of research and development workers per 1 million people and public and private research and development spending

SDG 12 Ensure sustainable consumption and production patterns

- **Target 12.2** By 2030, achieve the sustainable management and efficient use of natural resources

SGD	Target	Contribution	Performance Indicators and Metrics
7	7.3	Near-Data-Processing Accelerators reduce energy consumption by minimizing data movement. More than that, can perform the same operation with less clock cycles when compared to a traditional CPU architecture.	Energy efficiency ratio, power consumption, carbon footprint reduction, benchmark performance.
9	9.5	Open-source ASIC toolchain democratizes access to advanced computing design, enabling global innovation in energy-efficient computing.	Adoption rate, research output, accessibility index, design time saved.
12	12.2	Energy-efficient accelerators reduce computational waste, lowering resource demand.	Resource efficiency, energy resource savings, material use reduction, lifecycle impact.

Agradecimentos

Em primeiro lugar, gostaria de expressar a minha profunda gratidão ao Professor João Canas Ferreira, pela generosidade do seu tempo, pela orientação inestimável e pelo apoio incondicional que me proporcionou ao longo de todo este processo.

Agradeço de coração à minha família — aos meus pais, à minha irmã, aos meus avós, tios e primos — que sempre me encorajam e acreditam que consigo atingir todos os objetivos a que me proponho. Como vos digo muitas vezes: a melhor prenda é poder estar convosco.

Aos amigos da FEUP, do NEEEC e de Barcelos, cujo companheirismo e diversidade de personalidades tornaram esta etapa muito mais divertida. Em especial, àqueles que me acompanhavam diariamente para tomar um "cafezinho" ou ir almoçar ao "Bar de Minas".

Por fim, um agradecimento muito especial à Bia: pela paciência, pelo colo, pela motivação e pelo apoio incansável não só durante a realização desta dissertação, mas em todos os momentos da minha vida. Sem ti, não teria sido possível superar tantas adversidades.

Rodrigo Ribeiro Novais Alves

"There is no failure except in no longer trying."

- Elbert Hubbard

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
ASIC	Application-specific Integrated Circuit
BL	Bit line
CGRA	Coarse-Grained Reconfigurable Arrays
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
DUT	Device Under Test
EDA	Electronic Design Automation
FIFO	First-In, First-Out
FST	Fast Signal Trace
GDSII	Graphic Data System II
IC	Integrated Circuit
IMC	In-memory Computing
ISA	Instruction Set Architecture
LSU	Load-Store Unit
NDP	Near-Data-Processing
NDPAcc	Near-Data-Processing Accelerator
NMOS	N-channel Metal-Oxide-Semiconductor
NMP	Near-Memory Processing
OS	Operating System
PC	Personal Computer
PDK	Process Design Kit
PIM	Processing in Memory
PMOS	P-channel Metal-Oxide-Semiconductor
PVT	Process, Voltage, and Temperature
RISC-V	Reduced Instruction Set Computing V
RTL	Register-transfer level
SDC	Synopsys Design Constraints
SDF	Standard Delay Format
SoC	System on Chip
SPEF	Standard Parasitic Exchange Format
SPICE	Simulation Program with Integrated Circuit Emphasis
SRAM	Static Random Access Memory
STA	Static Timing Analysis
TCL	Tool Command Language
VCD	Value Change Dump
WL	Word Line
WNS	Worst Negative Slack

Chapter 1

Introduction

1.1 Context

The demand for more efficient processing of large-scale data is increasing at an unprecedented rate. This surge is driven by the adoption of data-intensive algorithms across various domains, including image processing, bioinformatics, physics, finance, and artificial intelligence [1]. These fields rely heavily on massive datasets that require extensive computational effort to process effectively.

Data-intensive applications typically involve repetitive and data-heavy computations. A typical example is matrix multiplication, a fundamental operation in many scientific and machine learning applications. In this operation, a vector is multiplied by each row of a matrix, and the resulting values are summed. While the computation itself is straightforward, the efficiency hinges on the ability to access data swiftly and in parallel.

Traditional computer architectures, while versatile in handling a wide range of tasks, are not optimized for such repetitive and data-centric operations. They are limited by their sequential memory access capabilities and insufficient parallelism. In the case of matrix multiplication, for instance, these architectures process each element sequentially, with memory access restricted to one per clock cycle. This limitation results in significant performance bottlenecks. In contrast, specialized hardware accelerators can process multiple data points simultaneously, leveraging memory bitlines for parallelism and optimizing performance for specific workflows.

The inefficiency of conventional memory access is not a new problem. First highlighted in 1994, the so-called "Memory Wall" arose from the growing disparity between the speed of Central Processing Units (CPUs) and the slower memory access times. This disparity has only worsened over time. Today, data transfers can take up to 1,000 times longer and consume 40 times more energy than computations on a CPU [2, 3]. This energy cost has become the dominant challenge, as the volume of memory accesses now far outweighs the computational load.

To address these challenges, the concept of Processing in Memory (PIM) has emerged. PIM refers to specialized accelerators designed to minimize the inefficiencies of traditional architectures by situating computation closer to the memory. PIM solutions can be broadly classified into two categories: In-Memory Computing (IMC) and Near-Memory Processing (NMP). The

former involves modifying the internal structure and behavior of memory to incorporate computation capabilities directly. The latter, in contrast, places a specialized accelerator close to the memory, optimizing data processing without altering the memory's fundamental operation. Both approaches aim to reduce energy consumption and improve performance for data-intensive tasks.

1.2 Problem and Motivation

The growing demand for hardware accelerators optimized for data-intensive applications underscores another pressing issue: the complexity and cost of designing Integrated Circuits (ICs). For decades, companies have heavily invested in developing specialized tools to meet the increasing demand for ICs. These tools, known as Electronic Design Automation (EDA) tools, automate various stages of the IC development process, from Register Transfer Level (RTL) design to verification. While these tools have significantly accelerated technological advancements, they have also introduced substantial complexity and high costs.

The financial barriers associated with these tools are steep. Leading companies charge tens of thousands of euros annually for access, making them inaccessible to individuals and small organizations. Furthermore, the expertise required to develop alternatives to these tools is rare, as the intricacies of the workflow demand deep technical knowledge. Consequently, access to the IC design process is effectively restricted to those who can afford the high subscription fees or are employed by the few companies that dominate the market.

In response to this exclusivity, some experts have developed open-source alternatives for parts of the IC design workflow. However, these solutions are far from ideal. They often suffer from lower performance compared to proprietary tools and lack seamless integration across the entire development process. This fragmented ecosystem poses a significant challenge for individuals or organizations attempting to create custom ICs without access to expensive EDA tools.

Summarizing the problem, the dual pressures of rising demand for specialized hardware to accelerate data-intensive applications and the lack of accessible, high-performance development tools present a substantial obstacle. Addressing these challenges is not only crucial for advancing IC technology but also for democratizing access to its development, enabling broader innovation in the field.

1.3 Objectives

The primary objective of this dissertation is to enhance a toolchain for ASIC design, as much as possible, Open-Source, with integration for Near-Data-Processing Accelerators (NDPAccs). It is intended to achieve a flow that can generate NDPAccs and integrate them with the chosen memory and CPU. Additionally, it is intended to have a detailed characterization of the implemented solution.

With that, the project can be detailed in the following objectives:

1. **Toolchain Extension and Integration** - Extend an open-source Application-Specific Integrated Circuit (ASIC) design toolchain, such as OpenFlow 2, to support workflows tailored for NDPAccs. This involves creating new modules and enhancing existing ones to automate critical steps in the design process, including synthesis, floorplanning, and routing, but especially to automate the interconnections between the different components of the design. The goal is to streamline the development of accelerators, making the toolchain more accessible and efficient for designing NDP systems.

2. **Integration with Cache Memory** - Design and implement methods and interfaces for integrating NDPAccs into the cache memory hierarchy. This involves ensuring integration with Static Random Access Memory (SRAM) to minimize data transfer overheads and improve throughput. By tightly coupling accelerators with cache memory, the integration aims to reduce latency, improve data locality, and lower energy consumption, enhancing the potential of this type of system.

3. **Detailed Characterization** - Perform an analysis of the developed NDPAccs, focusing on their interaction with memory systems and overall performance. This includes evaluating key metrics such as power consumption, timing delays, and chip area to validate their efficiency and practicality. The characterization process ensures that the accelerators meet the requirements of specific applications while maintaining compatibility with the broader system architecture, providing insights for further optimization.

In summary, this dissertation aims to bridge the gap in automated workflows for designing and integrating NDPAccs with memory and CPU systems by enhancing open-source toolchains. Through the extension and refinement of these toolchains, the development of automated accelerator generation systems, and the seamless integration of NDPAccs into cache memory hierarchies, this work seeks to enable efficient, customizable, and scalable solutions for modern computing challenges. Additionally, the detailed characterization of the implemented designs will ensure their performance, energy efficiency, and practicality, providing a solid foundation for further research and development in the field of Near-Data-Processing.

1.4 Contributions

This document presents the solution developed for this dissertation, which addresses the objectives stated in the previous section. After all the developments here presented, the contributions to the enticed problem were the following:

1. **OpenLane 2 Capabilities Extension** - All the scripts and the detailed exposition of the flow added new possibilities to design with these tools. In addition to having a more robust verification, both before and after running the full OpenLane 2 flow, a more direct and concise evaluation of the solution was achieved. The aggregation of statistics for each design run is key to helping a developer reach the intended goals for their projects.

2. **Modular RTL Infrastructure for NDPAccs** - Implemented configurable infrastructure to simulate NDPAccs according to predefined specifications, with margin to configure most of the properties of the signals in one place. It is easy to replace the accelerator to test others with the same top-level behavior.

3. **Compilation of Statistics** - Provided scripts can extract more details from the implemented system when compared to the previous flow and unite them in one place. Easy to compare with other designs in terms of performance.

4. **Comparison between different Flows** - A complementary exploration of the current open-source flows and their capabilities was added to provide a study of these implementations and the broader landscape of this space.

These developments can be found in the GitHub repository [4] of the dissertation's creator.

1.5 Document Structure

The remainder of this document is structured as follows: Chapter 2 delves into the state-of-the-art and related work, discussing fundamental concepts that support the motivation and problem addressed, while also reviewing the latest solutions presented in the literature. Chapter 3 outlines the proposed solution, providing an overview and detailing the structured research plan, including a timeline and objectives for the investigation. Chapter 4 presents the results and discussion, analyzing the outcomes of the research. Chapter 5 offers a characterization of the developed accelerators, evaluating key metrics such as power efficiency, timing delays, and silicon area utilization. Finally, Chapter 6 concludes the document, summarizing the key insights and highlighting their importance, while also suggesting directions for future work.

Chapter 2

Background and State of the Art

This Chapter focuses on examining the core concepts that are integral to the theme of this dissertation. It provides a detailed exploration of the foundational ideas that underpin the research, ensuring a clear understanding of the subject matter. Additionally, it reviews the partial solutions and underlying principles that serve as the building blocks for the proposed implementation. By analyzing these elements, this Chapter aims to establish a comprehensive context, linking existing knowledge with the goals of this dissertation to highlight how the desired implementation will build upon and contribute to the current state of the art.

2.1 Near-Data-Processing

Processing-In-Memory (PIM) represents a significant evolution in computer architecture, addressing one of the most critical challenges in modern computing — the inefficiencies caused by frequent data movement between processors and memory. Traditional architectures, based on the von Neumann model [2], rely on a clear separation between computation and memory. This separation has created what is known as the "memory wall", where the time and energy required for data transfers dominate system performance. PIM mitigates these issues by bringing computation closer to the data, either within the memory itself or in close proximity to it, thereby reducing latency and energy consumption.

The importance of PIM has grown with the rise of data-intensive applications, such as artificial intelligence, machine learning, graph analytics, bioinformatics, and large-scale simulations. These workloads often require the processing of massive datasets, leading to frequent memory accesses and significant bottlenecks in traditional systems. PIM offers a solution by enabling computation to occur where the data is stored, drastically reducing the need for data movement. This not only improves performance but also reduces the overall power consumption of the system.

PIM can be broadly categorized into two approaches: IMC and NDP. IMC integrates computational capabilities directly into the memory structure, allowing basic operations such as logical or arithmetic tasks to be performed within the memory cells. This approach modifies the internal design of the memory and is often used for highly specialized tasks. On the other hand, NMP

places dedicated computation units near the memory, such as accelerators adjacent to SRAM or Dynamic Random Access Memory (DRAM). These accelerators are optimized for specific types of workloads, such as deep learning or cryptographic operations, and leverage their proximity to memory for faster and more efficient data access.

NDP involves placing computational accelerators close to memory without altering the internal structure of the memory. These accelerators are designed to handle specific workloads efficiently by minimizing the data transfer distance. For instance, NDP can accelerate tasks like matrix multiplications, neural network computations, or other repetitive data-heavy operations, making it particularly suitable for modern applications requiring high data throughput and low latency [2, 5].

The significance of PIM, and particularly NDP, lies in its ability to overcome the fundamental inefficiencies of traditional memory hierarchies. By co-locating computation and memory, PIM architectures reduce data movement, improve energy efficiency, and enable greater parallelism. As the scale and complexity of data continue to grow, PIM technologies are poised to play a pivotal role in advancing the performance and scalability of next-generation computing systems.

In these type of solutions, there are different approaches in terms of how the accelerators are organized. It is common for them to have communication with the CPU, as seen in Figure 2.1, in order to synchronize with each other, which is key to make the solution efficient and viable.

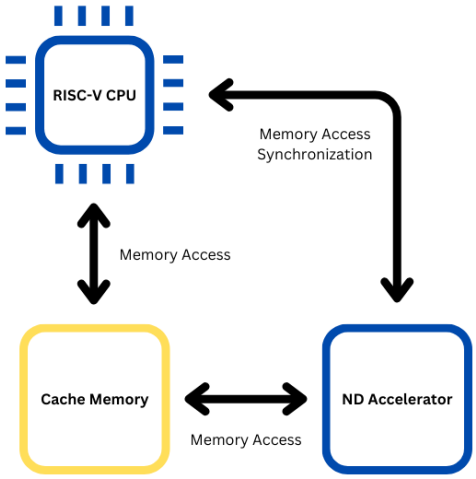

Figure 2.1: Infrastructure concept of a Near-Data-Processing Accelerator.

It is essential that the two computational units in the system (represented by the blue blocks) maintain proper communication. This is essential to avoid writing to the same memory space simultaneously, as they share a common memory (represented by the yellow block). The decision to use shared memory is fundamental to the system's design, as it enables certain operations, which would typically be handled by the CPU, to be processed more efficiently by the specialized unit, NDPAcc. However, if communication between the units is not effectively managed, the intended performance improvement could result in a significant bottleneck instead.

2.2 Near-Data-Processing Accelerators

When implementing NDPAccs, various architectural approaches have been explored in the literature, each presenting unique advantages and trade-offs. Two noteworthy approaches are the Gemmini [6] and Coarse-Grained Reconfigurable Arrays (CGRAs), both of which highlight effective strategies for achieving efficient computation near memory while addressing distinct application requirements. Adding to these, the paradigm of Map-reduce [7] was also explored, since the decomposition of data, parallelism and then aggregation of results, can also be applied to the architecture of NDPAccs.

The Gemmini architecture [8] takes a modular and system-level approach to Near-Data-Processing. It emphasizes the seamless integration of accelerators with multi-level memory hierarchies, such as caches and DRAM. By prioritizing modularity and configurability, Gemmini supports a wide range of computational tasks and adapts to the specific requirements of different applications. This architecture is designed to operate in close proximity to memory, enabling accelerators to access data directly from caches or DRAM, thereby reducing latency and energy overhead associated with data movement. One of its defining characteristics is the ability to model interactions between accelerators and the memory hierarchy, addressing critical aspects such as memory contention, cache coherence, and system-level bandwidth utilization. For example, simulation frameworks like NDPmulator provide capabilities for assessing these interactions under realistic system conditions, incorporating the effects of operating system overheads and memory contention on NDPAcc performance [1]. This makes Gemmini particularly suitable for workloads requiring efficient memory access and minimal delays.

In addition to its operational efficiencies, Gemmini facilitates simulation and evaluation of NDP systems. It provides insights into how accelerators interact with memory and other system components, allowing exploration of diverse workloads such as matrix multiplications, graph analytics, and stencil computations. Its versatility and adaptability ensure compatibility with varying computational demands. Recent studies have demonstrated the importance of configurable parameters like memory bandwidth and interconnection protocols, which significantly influence the efficiency of accelerators when integrated into multi-level memory hierarchies.

2.2.1 CGRA

CGRAs, on the other hand, are reconfigurable hardware architectures tailored to execute computationally intensive tasks efficiently. They consist of a grid of processing elements connected through a programmable interconnection network. These processing elements perform word-level operations, making CGRAs particularly suitable for data-intensive applications such as convolutional neural networks (CNNs) and signal processing.

A key feature of CGRAs is their distributed scratchpad memory, which stores intermediate data locally within the array. This reduces reliance on external memory and minimizes memory bandwidth requirements, enabling energy-efficient computation. For instance, studies have shown that CGRA-based CNN accelerators achieve a 1.93× higher performance per memory bandwidth

and 2.92× greater area performance compared to conventional GPUs [9]. The reconfigurable interconnection network allows CGRAs to dynamically adapt to different workloads by allocating resources and optimizing data flow for specific tasks.

CGRAs excel in scenarios where parallelism and data reuse are critical. For instance, in CNN acceleration, CGRAs leverage local memory and parallel processing elements to efficiently handle convolutional operations, significantly reducing the need for external memory access. Their flexibility, combined with the ability to support diverse workloads, makes CGRAs a compelling option for implementing NDPAccs. By employing temporal blocking and on-chip memory reuse techniques, CGRAs further enhance energy efficiency and reduce external memory pressure, providing a scalable solution for low-power embedded systems and other memory-constrained environments.

2.2.2 Map-reduce

Map-Reduce is a programming model designed to process large datasets efficiently by breaking the task into two key phases[7]. In the Map phase, data is transformed into intermediate key-value pairs, allowing parallel processing of smaller chunks. In the Reduce phase, these intermediate results are aggregated to produce a final output. This approach is widely used in data-intensive applications like big data analytics and machine learning, where scalability and parallelism are critical.

When applied to NDPAccs, Map-Reduce takes advantage of hardware designed to perform computations close to where the data resides, such as in memory units like SRAM or DRAM. NDPAccs aim to minimize the costly movement of data between memory and the CPU, a common bottleneck in traditional systems. By integrating Map-Reduce into NDPAccs, the Map phase can be executed in parallel across multiple processing elements located near the data, generating intermediate results quickly due to reduced memory access latency. The Reduce phase then combines these results efficiently, often using hardware-optimized mechanisms like reduction networks.

This integration offers significant benefits. It enhances performance by leveraging parallelism and proximity to data, reducing the time spent waiting for data transfers. It also improves energy efficiency by cutting down on power-hungry data movement across the system. For example, in a task like counting occurrences of a specific value in a large dataset, an NDPAcc could split the data into segments, process each segment locally in the Map phase to identify matches, and then sum the counts in the Reduce phase—all without sending the full dataset to a distant CPU.

2.3 EDA Tools and Hardware Components

Here will be analysed the EDA tools and hardware components that could be used in order to implement as system that meets the needs of this dissertation. In order to have an overall figure of which were explored, Table 2.1 shows each tool and component that was explored, as well as its possible contribution to the project.

Table 2.1: EDA tools and hardware components searched for the project.[1]

EDA Tool / Hardware Component	Functionality
OpenLane 2 [10]	ASIC development flow
mflowgen [11]	ASIC development flow
Icarus Verilog [12]	RTL simulator
Verilator [13]	RTL simulator
OpenSTA [14]	STA tool
OpenTimer [15]	STA tool
OpenRAM [16]	Implemented RAM generator tool
IOb-Cache [17]	RTL RAM generator tool
Hazard3 [18]	RISC-V CPU implementation

2.3.1 OpenLane 2

When the research for open-source tools started, the one that seemed to be the most plausible flow was mflowgen [11, 19]. This flow implemented the full stack of development using tools like Yosys [20] and Graywolf [21]. It allows the implementation of diverse modules for each part of the design flow of an ASIC. This type of modularity is important since, in some cases, the open-source tools probably will not be enough or might fail in a certain part, so having the possibility to choose another one or using proprietary tools should solve the problem. Other than that, it gives more freedom to the implementation and has already an advanced starting point in terms of the final desired Toolchain.

Even though mflowgen seemed to be fit, it did not provide a solid documentation, is starting to be deprecated and in spite of providing more modularity, it aggregated them in bigger steps, provided by the tools it used, as can be seen in Figure 2.2, compared to the detailed ones on OpenLane 2, seen in Figure 2.3. Additionally, it was recently updated to use almost solely the Open-ROAD project tools, so it seemed to be better to explore more robust and up-to-date flows.

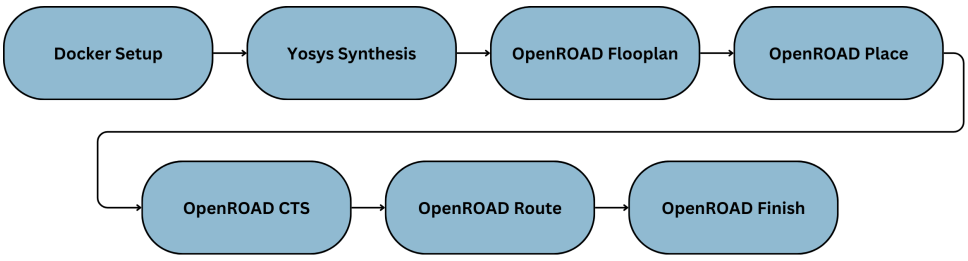

Figure 2.2: mflowgen default steps (according to [22]).

With the exploration of these type of tools, the OpenLane 2 project is a very plausible option and was chose to be used in this project. It is developed and used by the Efabless Corporation[23], so it has more support and updates than mflowgen, that is only developed by the community.

[1] Tools and components in **bold** represent the ones that were used during the dissertation.

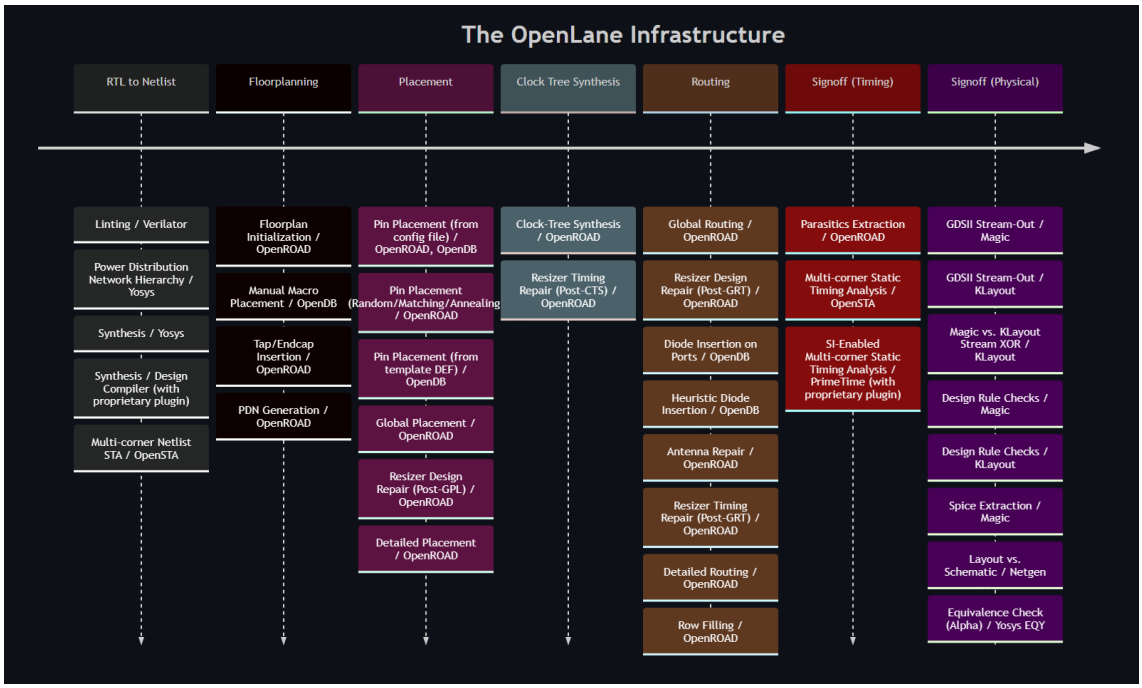

Figure 2.3: OpenLane 2 default steps (from [10]).

OpenLane 2 [10, 24] is a modular, open-source Register-transfer level to Graphic Data System II (RTL-to-GDSII) flow designed to address the challenges of reproducibility, customization, and flexibility in ASIC design. Unlike its predecessor, OpenLane 1 [25, 26], which primarily relied on Tool Command Language (TCL) based workflows, OpenLane 2 introduces a structured and modular framework where each task in the flow, such as synthesis, floorplanning, and routing, is encapsulated in independent "steps". These steps function with defined inputs and outputs, ensuring consistent behavior across different environments. This modularity allows to easily customize the flow by adding, replacing, or reordering steps to meet specific design requirements. Here the steps are much more explicit and easier to understand, compared to OpenLane 1 and mflowgen, where the usage of Docker implied more abstraction.

A key improvement in OpenLane 2 is its focus on reproducibility and modularity. Each step operates in isolation, without modifying external files or relying on undefined external resources. This ensures that the same input configuration consistently produces the same output, making the flow highly reliable for iterative design and debugging. Additionally, the platform's use of the Nix build system ensures that the environment is fully contained, with all dependencies explicitly defined [24]. This approach eliminates issues caused by system variations and improves cross-platform compatibility, particularly between Linux and macOS.

OpenLane 2 also introduces a Python-based Application Programming Interface (API) for defining and managing flows programmatically, as shown in Figure 2.4 . This API simplifies the creation of custom workflows and supports integration with other tools. For users less familiar with programming, configurations can be managed through JSON or YAML files, allowing for broad accessibility. The flow can also incorporate plugins to support additional steps or proprietary EDA

tools, making it highly adaptable for both open-source and hybrid workflows.

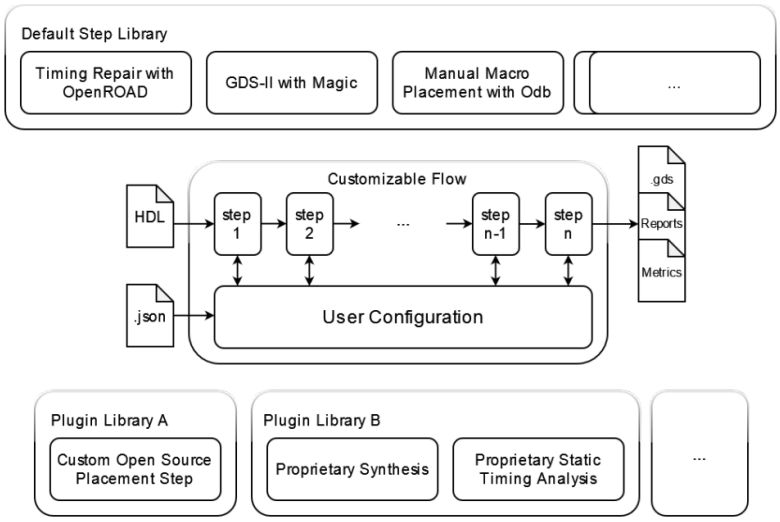

Figure 2.4: Graphic on how to add steps to OpenLane 2 (from [24]).

Adding to its configurability, OpenLane 2 is able to use different Process Design Kits (PDKs), being the default the SkyWater 130nm [27]. This design kit is open-source and widely used, being developed by SkyWater Technology Foundry in partnership with Google, which shows that it is well supported.

2.3.2 OpenSTA

OpenSTA is a robust open-source tool crafted for static timing analysis (STA), an essential process in designing digital integrated circuits to ensure reliable performance at specified clock frequencies. Static timing analysis is a cornerstone of chip design, verifying the timing behavior of circuits without requiring functional simulation. It meticulously examines all signal paths to ensure compliance with constraints like setup and hold times, preventing issues such as data corruption or system failures due to timing violations.

OpenSTA is equipped with a comprehensive set of features tailored to the needs of digital circuit designers. It verifies setup and hold times to ensure data signals align correctly with clock edges, analyzes clock domain crossings to mitigate risks like metastability in multi-clock designs, and computes path delays while reporting slack—the margin between required and actual timing—to pinpoint critical paths requiring optimization [14]. Its incremental analysis capability allows efficient re-evaluation of timing after minor design changes, accelerating the optimization process. Additionally, OpenSTA supports multi-scenario analysis, evaluating circuit performance under diverse conditions such as temperature or voltage variations, ensuring robustness in real-world applications.

The tool integrates seamlessly with other open-source tools, such as Yosys [20] for logic synthesis and OpenROAD [28] for physical design, forming a cohesive RTL-to-GDSII pipeline. It

supports industry-standard file formats, including Liberty (.lib) files for cell timing data and Synopsys Design Constraints (.sdc) files for timing specifications, ensuring compatibility with both open-source and commercial workflows OpenLane Documentation. OpenSTA's TCL command interface provides configurability, enabling users to script and automate timing analysis tasks, which is particularly valuable for tailoring the tool to specific project requirements or experimental setups. As an open-source platform, OpenSTA permits modifications to its source code, fostering innovation in timing analysis methodologies OpenSTA GitHub.

On the other hand, OpenTimer [15] is also a powerful open-source tool that performs STA. OpenTimer's compatibility with industry-standard formats, such as Liberty files for cell timing data and SDC files for timing specifications, ensures seamless integration into both open-source and commercial design workflows, thereby enhancing its versatility across diverse projects. It is a capable tool used in projects like Qflow [29], but it lacks the ability to produce accurate power estimation, particularly dynamic power estimation. In this aspect, OpenSTA is more complete, making it a better choice for this dissertation.

Beyond its technical prowess, OpenSTA plays a transformative role in democratizing ASIC design. By offering a high-quality, freely available STA solution, it lowers financial barriers, empowering a diverse range of stakeholders to perform rigorous timing verification. This accessibility is crucial in an era where custom hardware — such as energy-efficient processors and domain-specific accelerators — is increasingly vital for applications like machine learning and big data analytics [14]. OpenSTA contributes to a vibrant open-source ecosystem that drives the development of cutting-edge hardware, aligning with the OpenROAD project's vision of inclusive innovation.

In summary, OpenSTA's extensive feature set, seamless integration, and adaptability make it a cornerstone of modern ASIC design flows. Its role in ensuring timing integrity while supporting a more equitable design landscape underscores its significance in advancing semiconductor technology, paving the way for a future where innovative chip designs can emerge from diverse sources.

2.3.3 OpenRAM

As previously stated, the project relies on the usage of SRAM to be the data support for the NDPAcc. The usage of a L1 cache is essential due to its high throuput and low latency when compared to lower levels and DRAM. The usual structure of a 6T SRAM module is shown in Figure 2.5.

This figure shows the simplified architecture of an SRAM array and a detailed view of a 6T SRAM cell. The SRAM array is organized as a grid of rows and columns, where each cell stores one bit of data. The row decoder activates the word line (WL) for the row corresponding to the given address, while the column decoder selects the bit lines (BL and $\overline{BL}$) for reading or writing data to a specific cell.

The 6T SRAM cell, shown on the right, consists of six transistors. Four transistors (M1, M2, M3, and M4) form two cross-coupled inverters, creating a latch that stores a single bit (0 or 1).

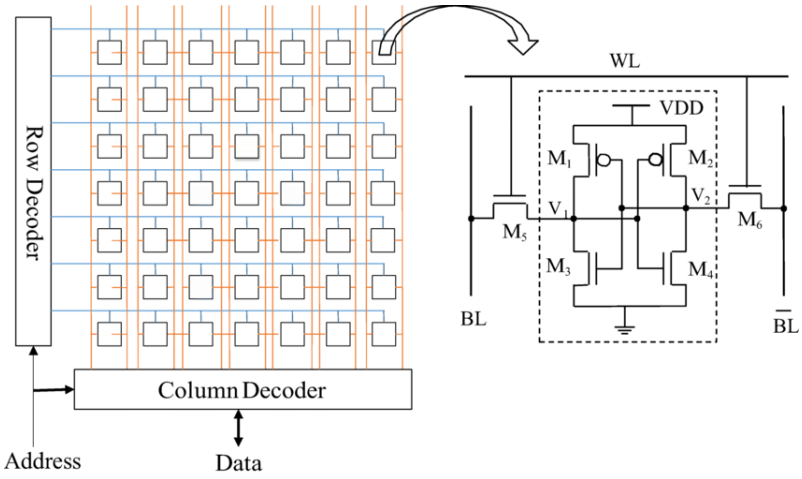

Figure 2.5: 6T SRAM structure and transistor level schematic (from [30]).

The other two transistors (M5 and M6) are access transistors controlled by the WL, which connect the latch to the bit lines (BL and $\overline{BL}$) during read or write operations.

A key advantage of this design is that the 6T SRAM cell retains its stored value as long as power is supplied, unlike DRAM, which requires periodic refreshing. This, allied with their simple use, makes SRAM faster and simpler for high-performance applications.

With the nature of the project being full open-source and the usage of proprietary memories being non-compliant and with cost for its usages, the OpenRAM is a good solution for generating memories that fit the needs of the project.

OpenRAM is an open-source memory compiler designed to streamline the design and characterization of SRAMs for use in SoC and microprocessor projects [16]. It simplifies what is traditionally a complex and time-intensive process by providing tools to generate, verify, and characterize SRAM layouts for various sizes, configurations, and technologies. Unlike commercial memory compilers, OpenRAM is fully open-source, making it entirely customizable and accessible to both academic and industry users. Its primary goal is to make SRAM design more flexible and efficient, while addressing the challenges posed by traditional proprietary solutions, such as restricted customization and high costs.

The tool operates using a modular framework that divides the design process into two main parts: the front-end and the back-end. In the front-end, the user specifies parameters such as memory size, word size, and aspect ratio. Based on this input, OpenRAM automatically generates Simulation Program with Integrated Circuit Emphasis (SPICE) models, GDSII layouts, Verilog models, and timing/power libraries. The back-end is responsible for characterizing the generated memory, leveraging SPICE simulations to produce detailed timing and power results that can be integrated into larger system designs. This separation ensures that the design and verification workflows remain clear and organized, even for complex memory configurations.

OpenRAM's architecture reflects its flexibility and adaptability. Its SRAM designs include key components such as a bit-cell array (typically based on 6T cells but supporting alternatives like 7T

and 8T), peripheral circuits (e.g., address decoders, word-line drivers, column multiplexers, precharge circuits, sense amplifiers, and write drivers), and control logic for synchronous operation. These components are dynamically generated based on user-defined specifications, ensuring that the final design is both optimized and tailored to the application. OpenRAM also supports multi-bank SRAM designs to balance area, power, and performance requirements, making it suitable for memory-intensive applications. Since the project is not focused on the memory, the simpler and easier to use the module, the better, like a 6T cell [31, 32], since the intent is to show how the implementation should work, so a simpler cell is easier to implement and to use in order to demonstrate what is intended.

One of the standout features of OpenRAM is its technology independence. It uses technology directories containing specific rules, layer maps, and custom cell libraries, enabling users to adapt it to a wide range of process technologies. OpenRAM includes reference implementations for FreePDK45 (non-fabricable) and SCMOS (fabricable) design kits, but it can also be extended to other nodes with custom technology files, like the one that OpenLane 2 already supports, SkyWater 130nm [33]. This adaptability is further enhanced by its Python-based implementation, which allows to easily modify or extend the tool to meet specific design requirements.

OpenRAM also excels in its characterization capabilities. It uses SPICE simulations to analyze the timing, power, and area of the generated SRAM designs. The results are output in standard formats like Liberty (.lib) files, which can be directly used in system-level simulations and ASIC workflows. This detailed characterization is critical for verifying the memory's performance and its compatibility with the rest of the design, before actually testing the whole system together.

2.3.4 IOb-Cache

IOb-Cache is a high-performance, configurable open-source cache implemented in Verilog, designed to optimize data access in modern digital systems, particularly system-on-chip (SoC) architectures. As part of the IObundle project, IOb-Cache addresses the critical challenge of memory latency by providing a flexible and efficient solution for managing data in hardware, making advanced cache design accessible to a wide range of users, from academic researchers to small-scale innovators [17].

In digital systems, a cache serves as a fast, small memory component that stores frequently accessed data, significantly reducing the time required to retrieve information from slower main memory. By bridging the speed gap between processors and memory, caches are essential for enhancing system performance. However, designing an effective cache requires careful consideration of parameters such as size, associativity, replacement policies, and write strategies, which vary depending on the application.

IOb-Cache distinguishes itself through its extensive configurability, enabling designers to tailor the cache to specific requirements. Its key features include support for pipeline architectures, allowing one request per clock cycle for both reads and writes, which ensures high throughput. It offers multiple back-end interfaces, including native pipelined and AXI4, providing flexibility

for integration with various memory systems. Designers can choose between write-through/not-allocate and write-back/allocate policies to balance performance and data consistency. Additionally, IOb-Cache allows customization of parameters such as the number of ways, address widths, front-end and back-end data widths, number of lines, words per line, and replacement policies, including Least Recently Used (LRU), Pseudo-LRU Modified (PLRUm), and Pseudo-LRU Tree (PLRUt). An optional cache-control module facilitates performance measurement, cache invalidation, and monitoring of the write-through buffer status, enhancing design optimization [17].

The cache's modular architecture, comprising front-end, cache core, and back-end modules, simplifies integration into diverse designs. This modularity allows designers to swap or customize interfaces without altering the core functionality. Configuration is managed through Py2hwsw, a tool that generates hardware and software components based on a Python description file (`iob_cache.py`), which users can modify to meet specific needs [17]. This approach streamlines the design process, making IOb-Cache adaptable to applications ranging from embedded systems to high-performance computing.

Performance evaluations highlight IOb-Cache's efficiency. When integrated into the IOb-SoC, it supports multi-level cache systems and achieves a clocks-per-instruction (CPI) as low as 1.055, as demonstrated by the Dhrystone benchmark, indicating near-optimal performance. This is driven by its ability to process one request per cycle and its efficient write-through buffer, which enables fast, often single-cycle writes—a significant improvement over other caches that may require multiple cycles for write operations. Compared to other open-source caches like PoC.cache, IOb-Cache offers advantages such as configurable back-end data widths, AXI4 support, a write-through buffer, a broader range of replacement policies, and a modular design. While PoC.cache supports features like individual line invalidation and full-associative caches, IOb-Cache's performance edge, particularly in write operations, makes it a compelling choice for many applications [17].

IOb-Cache seamlessly integrates with open-source tools like Yosys for synthesis and OpenROAD for physical design, forming a cohesive RTL-to-GDSII pipeline. Its compatibility with standard interfaces, such as AXI4, ensures it can be incorporated into both open-source and industry-standard workflows. The cache's open-source nature allows users to modify its source code, fostering innovation in cache design and enabling tailored solutions for specific use cases.

2.3.5 RISC-V CPU Hazard3

For the integration of the NDPAcc with a working system, as shown in Figure 2.1, there is a need for a CPU, to show the integration between them and have implementation results.

The Hazard3 Reduced Instruction Set Computing V (RISC-V) CPU [18] will be utilized as the central processing unit in this work to manage the integration and coordination of NDPAccs. Its lightweight and modular design make it particularly suitable for embedded systems and resource-constrained environments, aligning with the goals of developing energy-efficient and high-performance computing architectures.

As an open-source implementation of the RISC-V instruction set architecture (ISA), Hazard3 provides flexibility for customization, allowing for the integration of additional extensions such as the multiplication/division extension or the compressed instruction extension, depending on the system requirements. Its small silicon footprint and low power consumption enable seamless integration with the memory subsystem and accelerators, ensuring an optimized balance between performance and energy efficiency.

The Hazard3 CPU will serve as a key component for managing tasks such as instruction scheduling, data transfer coordination, and interaction with cache memory in the overall system. Its simple pipeline architecture ensures sufficient computational capability while maintaining low design complexity and minimizing overhead. This makes it an ideal choice for controlling and coordinating the operation of SRAM-based NDPAccs, where efficiency and compatibility with memory-intensive workflows are critical. By leveraging the capabilities of the Hazard3 processor, this project will demonstrate how lightweight RISC-V CPUs can be effectively integrated into advanced accelerator-memory systems.

Additionally, the reliability and usability of the Hazard3 CPU are demonstrated by its integration into the Raspberry Pi Pico 2 [34], a compact and efficient microcontroller platform designed for embedded and IoT applications. The Pico 2 utilizes Hazard3 for its lightweight, energy-efficient design, supporting that it is capabile to handle real-world tasks in resource-constrained environments. This integration further highlights the practicality and flexibility of the Hazard3 architecture, reinforcing its suitability for managing the SRAM-based NDPAccs developed in this work.

2.4 Other Related Work

The concept of NDPAccs is not new, as previously referenced, and thus, studies on their usage and implementation are already available. One of them is the NDPmulator framework [1], which simulates a type of NDPAcc using the gem5 [35] simulation environment, thereby helping to understand how these accelerators behave and are tested. With that in mind, the following text explores further this project.

The NDPmulator framework presents a comprehensive simulation environment for evaluating the integration and performance of NDPAccs within modern memory hierarchies, addressing a range of challenges critical to the design of NDP architectures. Its technical focus on accurately simulating the interaction of NDPAccs with multi-level memory hierarchies, including caches (L1, L2, and LLC) and DRAM, directly relates to the work that will be developed in this dissertation. Specifically, NDPmulator models key architectural features, such as coherence protocols, memory contention, and queuing delays, which are central to understanding and optimizing the integration of accelerators into memory subsystems.

A significant contribution of NDPmulator is its detailed modeling of memory traffic patterns and cache utilization, which are vital for analyzing how NDPAccs interact with the memory hierarchy. The simulation accounts for the impact of NDPAcc memory requests on shared resources,

modeling both read and write contention at various levels of the hierarchy. This level of granularity is particularly relevant to the dissertation's focus on designing SRAM-based NDPAccs, where efficient memory access and reduced contention are primary design goals. For example, the framework models contention in memory controllers and cache queues, which directly impact latency and throughput, insights that are crucial for optimizing cache-integrated NDPAccs.

Something that is here implemented is the previously stated Gemmini architecture. It is a key feature of the framework, since it is a fairly modular architecture, which leverages its usage for different scenarios where the user wants to apply the accelerator.

NDPmulator also evaluates the effect of NDPAccs on memory bandwidth saturation and latency variation under high-intensity workloads. The framework provides insights into the trade-offs between increased NDPAcc activity and its impact on overall system performance, particularly when accelerators generate frequent, high-bandwidth memory requests. These insights directly translate to the SRAM-based designs in this dissertation, where achieving low-latency access and efficient bandwidth utilization is a priority for tightly coupled accelerator-cache integration.

In addition to memory-level interactions, NDPmulator incorporates hardware/software co-design considerations, such as Operating System (OS) overheads, device driver interactions, and scheduling of NDPAcc tasks. For instance, the simulation accounts for the latencies introduced by OS-level thread scheduling and the synchronization of NDPAcc memory requests, both of which can significantly affect real-world performance. This level of analysis is particularly relevant when designing accelerators for integration into shared memory systems, as the dissertation aims to do. Understanding these interactions allows for better characterization and optimization of SRAM-based accelerators to ensure compatibility with existing system architectures.

Another feature of NDPmulator is its ability to simulate heterogeneous workloads, such as graph traversal, stencil computation, and matrix multiplications, which are representative of NDP workloads. These workloads allow for an in-depth evaluation of the performance benefits of NDPAccs, particularly in terms of reduced memory access latency, increased data locality, and parallel execution. The methodologies used in NDPmulator to evaluate these workloads provide a framework for benchmarking and validating the designs developed in this dissertation, ensuring their applicability to real-world data-intensive applications.

While NDPmulator focuses on simulation, its findings directly inform the physical design and integration workflows of this dissertation. By leveraging the technical insights from NDPmulator, such as memory traffic modeling, contention management, and workload analysis, the dissertation extends these principles to the development of an automated workflow for designing and characterizing SRAM-based NDPAccs. These accelerators are optimized for integration into cache memory hierarchies, aligning with the architectural principles and performance metrics explored in NDPmulator.

Chapter 3

Development Overview

3.1 Workflow

To develop a solid flow for characterizing NDPAccs and building them, a solid base was essential, as many steps of the IC design are tedious and challenging to automate. After the search carried out and referred to in Chapter 2, OpenLane 2 was chosen.

The default flow was used since it satisfied the needs for this implementation. It could be extended by utilizing its Python API, but that would require running the entire flow at once. For the iterative nature of key steps in IC development (Functional Verification, Timing, and Power Analysis), the use of scripts and tools for these steps was employed separately. Although the previously stated steps, which are critical to achieving a reliable design, are already implemented in the OpenLane 2 steps, as shown in Figure 2.3, they do not provide the repetitive nature needed.

The extended flow, which integrates the default from OpenLane 2, is shown in Figure 3.1.

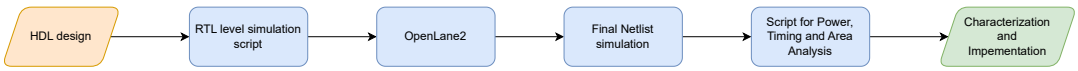

Figure 3.1: Diagram that shows the chain of steps of the Implemented Flow.

Firstly, the HDL design should be developed, typically in RTL, using Verilog and/or SystemVerilog. To verify that this is running as expected, a testbench that instantiates the desired Device Under Test (DUT) and inputs the necessary signals and data for a proper simulation of the device's expected behavior is required.

This first step is crucial before running the whole flow, as it can detect Synthesis errors and verify if the system is functioning as expected, which saves a significant amount of time. This is particularly important since OpenLane 2 takes significantly more time to run. Additionally, this first debugging script helps to save unnecessary resource usage and file creation that the default path generates if the Synthesis errors are not critical. For this, the use of Icarus Verilog [12] or Verilator [13] was implemented, as each has advantages compared to the other.

Once the design's functionality is verified, it undergoes the various steps of OpenLane 2, from RTL to GDSII, with multiple error and warning verifications for each step. Some significant steps to check when running OpenLane 2 are Yosys' synthesis and OpenSTA's STA. These indicate whether the design can be synthesized and if it complies with the timing restrictions, respectively. Additionally, it provides all the necessary files to conduct the next steps in the flow, including the design netlist and files that characterize its delays and behavior.

With the netlist gathered, using the previously developed testbench for the design, the DUT can now be instantiated with the actual implementation and verified that the RTL-to-netlist behavior is as expected. This can be achieved by using a simulation tool like the previously mentioned Icarus Verilog and Verilator. In this step, the Value Change Dump (VCD) file should be obtained to perform a more accurate analysis of the system. The VCD file provides the activity of the nodes in the IC to OpenSTA, which helps it estimate the system's power usage more accurately.

Typically, at this point, the design was only run at a low starting frequency, which is recommended to prevent early errors in the design and may not be sufficient to meet the project's goals of either timing or power. Since OpenLane 2 does not provide reliable power estimation and gathering timing can be difficult, a step was taken to produce these statistics. It has the objective of estimating the Maximum Frequency, Power Usage, and area of the design, combined with all the desired timing corners – combinations of process, voltage, and temperature (PVT) variations. With that, it can be determined what constraints should be changed when running OpenLane 2 if it does not already meet the requirements for achieving timing closure, power consumption, and area optimization.

The analysis carried out in the last step could need to be run several times for each design iteration. When it finally meets all the requirements, this analysis provides a good characterization of the solution achieved.

3.2 Hardware Modules

The proposed ASIC design flow was developed concurrently with a system that serves as a base platform for NDPAccs. All of the modules were produced at an RTL, using Verilog as the HDL. How these modules were interconnected and encapsulated is presented in Figure 3.2.

Firstly, the Cache module was developed using the already implemented IOb-cache project, which was primarily configured according to some standards of a RISC-V CPU, allowing the controller to utilize the same interface length for all its interfaces.

The cache requires RAM as its backbone to perform operations as intended. For that purpose, a simplified vector of memory was created as the RAM for the system. The purpose was not the performance it could provide but to behave as a normal would, providing the capabilities needed for the cache to operate as expected.

These first modules are encapsulated in the Memory Wrapper, which provides the necessary connections between them and facilitates easier testing and better integration with the rest of the

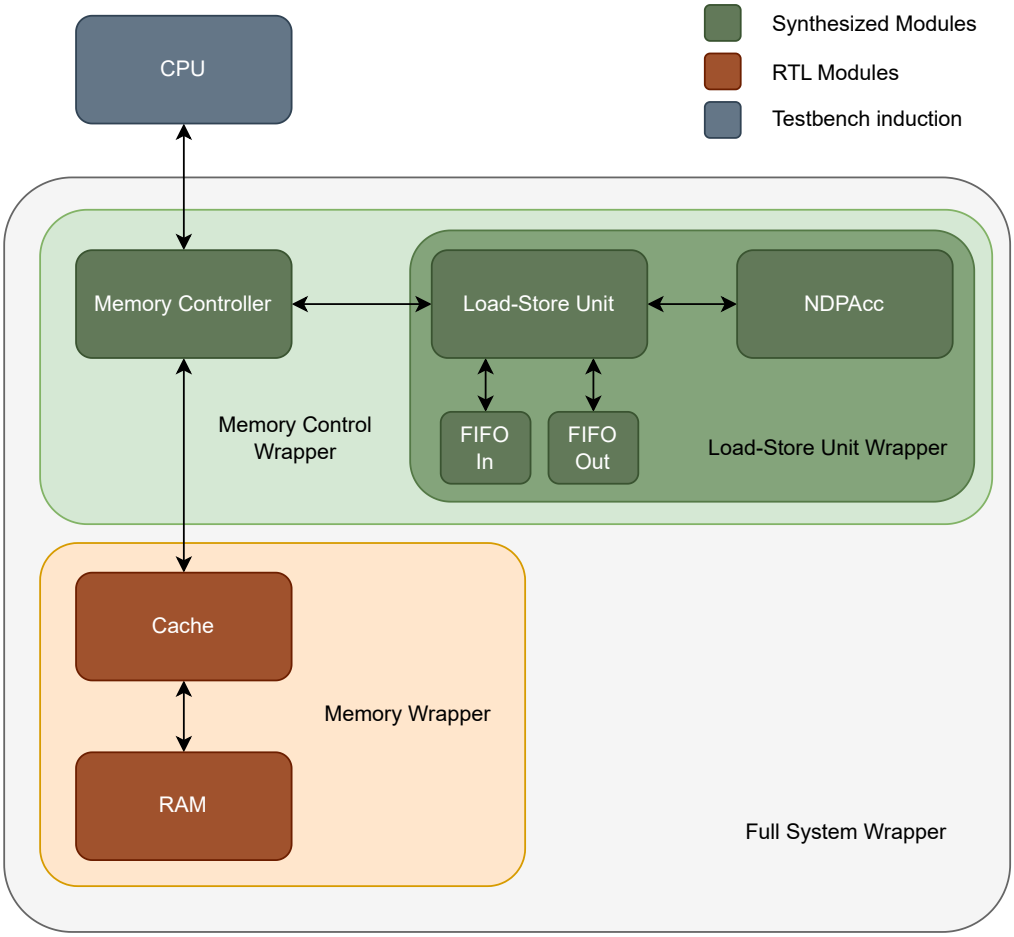

Figure 3.2: A high-level view of the Implemented Hardware, with the representation of the interactions between each module.

infrastructure. These were implemented as Verilog modules, as the intended infrastructure worked well and did not introduce any potential problems with synthesizing the entire cache.

Since the main objective was to create a template and flow for easier testing of NDPAccs, some simple accelerators were developed to facilitate a base comparison between them and to validate the overall implementation. To handle requests to use the accelerator, as well as all the traffic necessary between the accelerator and other modules, and to schedule the accelerator, a Load-Store Unit (LSU) was created. Additionally, to achieve greater modularity, two First-In, First-Out (FIFO) queues were attached to the LSU. Also, for validation and implementation purposes, a wrapper for this system was created.

To manage the entire interaction and grant permission to access the Memory modules, a Memory Controller handles requests from both the CPU and the LSU to access memory. Furthermore, it has a queue for access requests to the memory, which are processed using the interface provided by the Memory Wrapper.

The Memory Controller, together with the LSU Wrapper, forms the Memory Control Wrapper, which is the part of the system that was synthesized and run through the entire flow.

To validate the system's overall function, the Full System Wrapper integrates both the Memory Control Wrapper and the Memory Wrapper. It leaves interfaces to the outside of the module for a testbench to excite the system and mimic the behavior of a CPU attempting to access memory.

Chapter 4

Design and Evaluation Methodology of the Flow and Hardware

This chapter focuses on how the already implemented toolchain was extended with custom scripts to validate the desired solution better. Additionally, methodologies for evaluating the achieved results are also present, demonstrating their efficiency and providing a basis for comparison with similar ones.

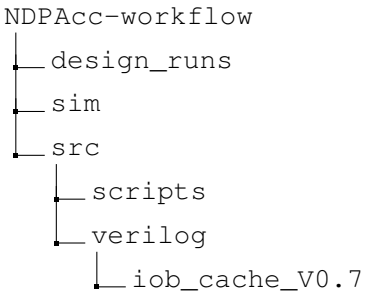

Figure 4.1: Project's directories structure.

4.1 Flow Implementation and Evaluation Methodology

As previously mentioned in Section 3.1, were created add-ons to enrich the OpenLane 2 toolchain. These additions were mainly through scripts. This section depicts each of the following that were critical for each of the stages mentioned after, being these steps according to Figure 3.1:

- `run_simulation_v2.py` & `config_run.yaml` - These combined conduct the RTL level simulation of the design.
- `analysis.py` & `config_analysis.yaml` - Together generate the STA, Power and Area reports.

For typical uses of this flow, there should be no need to rerun without checking results for multiple frequencies. Before rerunning, with different constraints at the input of OpenLane 2, any design in the development stages should be reviewed to determine if a rerun is necessary. In the particular case of analyzing the performance of the flow, especially for the OpenLane 2 step, which is the most time and resource-consuming, auxiliary scripts were used:

- `power_time_flow.py` - Runs OpenLane 2 for the proposed design for the desired frequencies, *N* times for each one.
- `average_rpt.py` - Aggregates results from `power_time_flow.py`, calculating average and standard deviation for data gathered.

4.1.1 RTL simulation

The script `run_simulation_v2.py` is entirely dependent on the `config_run.yaml`, which provides several settings for the simulation, from files to simulation flags. These files are available in the project' `/src/scripts` folder for storing purposes, but they are also hard linked at `/sim`, where it is recommended to make the simulation. The entries that should be defined in `config_run.yaml` are described in Table 4.1.

Table 4.1: Configurations to run RTL simulation.[1]

Entry	Type	Description
`flags_include`	list of str/dict	Flags to include in simulation, and possible pairs of entries
`location_source_verilog`	string	Path to folder with the source files of the design
`wave_out_{name}_module`[2]	string	Wave name for GTKWave with the produced wave
`sim_out_{name}_module`[2]	string	Name of the output from Icarus Verilog
`tb_files_{name}_module`[2]	list of str	Name of the testbench
`{name}_files`[2]	list of str	Name of source files needed for the design

Naming files according to established standards is also crucial for large designs with multiple interconnected modules. For that, Table 4.2 shows some examples of what patterns were used for the naming of the various files present in the project.

In addition to these initial entries, if a design is new, it should also be added to the script `run_simulation_v2.py`, as it requires the entries that it accepts to be defined. The code snippet presented in Listing 4.2 shows how the argument `--source-set` is defined, and thus, a choice should created for the new design to be simulated.

[1] All *Entries* referred to in this table are mandatory, at least for one module.
[2] In this variable, `{name}` refers to be the name of the design.

Table 4.2: Files naming patterns

Name	**Pattern**	**Example**
Design name[3]	None, used as reference for others	`lsu_wrapper.v`
Testbench[3]	{design_name}_tb.v	`lsu_wrapper_tb.v`
Icarus Output	tb_{design_name}	`tb_lsu_wrapper`
Wave file	wave_{design_name}.vcd	`wave_lsu_wrapper.vcd`

Adding to that, the Listing 4.3 demonstrates what each choice implies in the program. That said, should also be added a new condition to these, configuring correctly which files are needed for that design, taking into account what is in the `config_run.yaml`. An example of how to run the simulation is shown in Listing 4.1.

Listing 4.1: Example running RTL simulation

```
$ python3 run_simulation_v2.py --source-set acc_tb --compiler
  iverilog --wave
```

Listing 4.2: Design selection argument

```
parser.add_argument(
    "--source-set",
    nargs="+",
    choices=["fs_tb", "mem_tb", "loaded_mem_tb", "acc_tb", "ram_tb", "
      ↪ fifo_tb", "lsu_tb", "lsu_wrapper_tb", "acc_max_min_tb", "
      ↪ acc_property_tb"],
    default=["fs_tb"],
    help="Choose which simulation to run (default: fs_tb)."
)
```

Listing 4.3: Argument `source-set` action example

```
elif "lsu_tb" in args.source_set:
    print("\n------------------------\nRunning LSU testbench.\n
      ↪ ------------------------\n")
    src_files = (config['tb_files_lsu_module'] + config['lsu_files'] +
      ↪ config['acc_files'] + config['fifo_files'])
    wave_out = config['wave_out_lsu_module']
    sim_out = config['sim_out_lsu_module']
    sdf_file = config.get('sdf_file_lsu_module', None)
```

This script has more arguments than just `-source-set`, but it is the one that should always be set, due to the need to test different parts of the design.

[3]Module names should have the same name as the file.

Table 4.3: Command-line arguments for running the RTL simulation.[4]

Argument	Type	Description
`-source-set`	string	Choose which simulation to run. Options: `fs_tb`, `mem_tb`, `loaded_mem_tb`, `acc_tb`, `ram_tb`, `fifo_tb`, `lsu_tb`, `lsu_wrapper_tb`, `acc_max_min_tb`, `acc_property_tb`. Default: `fs_tb`.
`-compiler`	string	Verilog compiler to use. Options: `iverilog`, `verilator`. Default: `iverilog`.
`-use-sdf`	boolean	Enable SDF backannotation during simulation. Requires an SDF file specified in `config.yaml`.
`-wave`	boolean	Open GTKWave to view waveform output after simulation.

With all the previously stated settings, the script can be run to obtain simulation results for the design. Using Icarus Verilog as the preferred tool for simulation, the implementation attempts to compile the design using `iverilog` with all inputs set to the defined values. It then uses the output of this compilation to run `vvp`, which executes the compiled file. If `iverilog` reports any issues when compiling, such as syntax errors, it stops the run and displays the error to be fixed.

4.1.2 OpenLane 2

After confirming that the system works as expected at RTL, the OpenLane flow should be run. For that, using its `nix-shell` environment, a folder where the runs of the designs would be configured and stored is important. In the project, this folder is `/design_runs`, where here subfolders were created for each design that is pretended to run. Inside the subfolder for each design, a `config.json` file needs to be created, similar to the one in Listing 4.4. It shows which configurations were used throughout all the simulations, but OpenLane 2 supports many other configuration variables that can be used, which can be found in their manual [10].

Listing 4.4: Argument `source-set` action example

```
{
  "DESIGN_NAME": "lsu_wrapper",
  "VERILOG_FILES": [
    "../../src/verilog/accelerator_sum.v",
    "../../src/verilog/fifo.v",
    "../../src/verilog/lsu_wrapper.v",
    "../../src/verilog/load_store_unit.v"
  ],
  "CLOCK_PERIOD": 10,
  "CLOCK_PORT": "clk",
  "PDK": "sky130A"
}
```

[4] All *Arguments* are optional since default values are defined, but it is recommended to at least use `-source-set`.

Once the project is set, OpenLane 2 can be run using the command shown in the Listing 4.5. This will instantiate the default flow of OpenLane 2, with the steps shown in Figure 2.3.

Listing 4.5: Example running RTL simulation

```
[nix-shell:~/design_runs/lsu_acc/]$ openlane config.json
```

When the flow completes without error, a `/runs` directory should have been created inside the design folder. In the example shown before in Listing 4.5, this directory is `/lsu_acc`, where the run files were saved. The files resulting from the run are inside the `/final` folder. If the folder is not there, it probably means that an error occurred, and the `.log` files for each step should be checked to find where the problem lies.

In that generated files folder, a `.nl.v` should be found, which is crucial for the next step of simulating with the generated netlist.

4.1.3 STA, Power, and Area

Having the first implementation of the design helps to refine it and achieve the required constraints. For that, a simulation using the previously stated netlist is crucial. The usage of an open-source tool for this simulation was not possible. This is possibly due to the fact that the open-source library used, SkyWater 130nm, was not compatible with the standards required by Icarus Verilog and Verilator. To validate the entire flow, a proprietary simulator was used - Verification Compiler Simulator (VCS) from Synopsys) - which, in conjunction with the initial testbench, successfully simulated the netlist.

This is particularly important, as the simulation with the netlist generates the VCD file necessary for a more accurate estimation of power. Furthermore, it ensures that the netlist accurately represents the initial function of the system design.

To evaluate the performance of an IC, a critical analysis that must be conducted is Static Timing Analysis (STA). STA is essential to ensure that signals propagate through the circuit within the required time constraints, typically arriving at their destinations before the next clock cycle or before they are needed for further operations. This analysis confirms that the IC will operate correctly at its intended frequency.

For this purpose, various tools are available; however, to maintain the use of open-source solutions, OpenSTA was selected. OpenSTA generates the necessary reports to perform STA after the design has been processed through the OpenLane 2 flow. The post-run analysis with OpenSTA allows designers to assess the circuit's timing performance across various timing corners, ensuring reliability under different conditions.

One of the key advantages that led to the choice of OpenSTA over similar tools, such as OpenTimer, was its superior set of capabilities for accurately estimating the power consumption of the IC. This is crucial since it was one of the major missing points of OpenLane 2 that was intended to be tackled.

Additionally, OpenSTA's reports reveal whether there is a sufficient timing margin to increase the IC's operating frequency. If the analysis indicates positive timing headroom, the design can likely operate faster. This insight makes the process iterative, enabling fine-tuning by adjusting constraints and rerunning the OpenLane 2 flow.

In STA, slack is a key metric, defined as the difference between the required time for a signal to arrive and its actual arrival time. The setup slack for a timing path is calculated as follows:

$$\text{Slack} = T_c - (\text{T}_{\text{ckq}} + \text{T}_{\text{combo}} + \text{T}_{\text{setup}})$$

Where:

- T_c: Clock period (time between consecutive clock edges).
- T_{ckq}: Clock-to-Q delay of the launch flip-flop.
- T_{combo}: Combinational logic delay along the path.
- T_{setup}: Setup time of the capture flip-flop.

A positive slack means the signal arrives early enough, while a negative slack indicates it arrives too late, risking circuit failure.

To estimate the maximum operating frequency securely, designers rely on the Worst Negative Slack (WNS), the smallest (most negative) slack across all timing paths:

$$\text{WNS} = \min(\text{Slack}_i) \text{ for all paths } i$$

In OpenSTA, the command `report_worst_slack -max` retrieves the WNS. A negative WNS signals a timing violation, suggesting the circuit may fail at the specified frequency. The maximum frequency (f_{max}) is calculated as:

$$f_{\text{max}} = \frac{1}{T_c + |\text{WNS}|}$$

With that in mind, the TCL script to run OpenSTA needs to go through the frequencies until achieving $WNS = 0$, meaning that T_c is the maximum frequency that should be used. For this, the script `analysis.py` generates the TCL script using this approach to achieve the maximum frequency of operation for each implementation in the desired corners. The generation by the Python script enables customization and user input for the execution of OpenSTA.

Although it is not its primary purpose, OpenSTA can also estimate the system's power consumption. By using the command `report_power`, a report is generated with the dynamic (switching and internal) and static (leakage) power consumption of the system.

Power estimation in digital circuits involves calculating the total power consumption, which is the sum of two main components: static power and dynamic power. Static power arises from leakage currents in the circuit's components when they are not in use, and it is estimated by summing the leakage power of each component, typically derived from their respective specifications.

Meanwhile, dynamic power is consumed by signal transitions and depends on factors like switching activity, load capacitance, supply voltage, and operating frequency. It is calculated using the formula:

$$P_{\text{dynamic}} = \alpha \cdot C \cdot V^2 \cdot f$$

Where α is the switching activity factor (obtained from simulation data or approximated with default values), C is the load capacitance, V is the supply voltage, and f is the operating frequency.

Thus, the total power consumption is expressed as:

$$P_{\text{total}} = P_{\text{static}} + P_{\text{dynamic}}$$

This process provides a detailed breakdown of power usage, enabling designers to identify and optimize power-intensive areas, although more advanced analysis may be required for complex optimizations.

Another important factor for a design is its area. In this department, the `analysis.py` script only scavenges the already generated reports from OpenLane 2. It relies on its information to present the die and core areas of the system.

To obtain all the metrics previously discussed, the entries presented in Table 4.4 should mostly be configured in the `config_analysis.yaml`.

Table 4.4: Configurations to analyze design.

Entry	**Type**	**Description**
`design_name`	string	Name of the top module
`runs_dir`	string	Path to the folder where the runs for the design are stored
`run_tag`	string	Name of the folder of the run to be analyzed
`netlist_rel_path`	string	Relative path to the netlist file
`sdc_rel_path`	string	Relative path to the SDC file
`spef_rel_path`	string	Relative path to the SPEF file
`area_report_rel_path`	string	Relative path to the are report file
`vcd_path`	string	Path to the VCD file generated by the netlist simulation
`lib_files`	list of str	Path to the liberty files for each corner

From these entries, the script cannot be provided with `vcd_path` and/or `run_tag`. If the first one is not provided, the power estimation provided will not account for the actual switching activity of the implemented system. If not presented, the later version will be replaced by the most recent run of OpenLane 2 for that design.

Since this script was not designed to accept arguments from the command line, it is run by using a Python interpreter followed by the script's name.

After the run is completed, a summary report will be printed to the terminal and also available in the `/reports` folder inside the preferred run. In the same folder, a detailed power report is presented for each liberty file provided to the script. If the file is run twice for the same implementation run, it will replace the previous data with the most recent one.

Given the information provided by the developed script, the user can now decide whether to proceed with the system's performance or rerun OpenLane 2. In the test case presented, there were no goals to achieve in terms of power, timing, or area. Therefore, for testing purposes, the main objective was to achieve the maximum frequency possible for each design.

To do this, the restriction would be applied back to the `config.json` file in the base directory of the design, and then the analysis would be repeated.

4.1.4 Flow Performance Methodology

To evaluate the efficiency and energy demands of the implemented workflow, a systematic approach was taken to gather performance metrics during the IC design process. The process was initiated by employing the `/usr/bin/time` utility, which has been proven to be a reliable source for power estimation of similar systems [36]. By prefixing the command, it captured the following time statistics: real-time, showing the total duration from start to finish; user time, reflecting CPU effort on user-level tasks; and system time, tracking kernel-level operations. This trio illustrated the workflow's duration and whether it relied heavily on CPU power or I/O operations.

While the time data was being collected, power consumption was being captured by `powerstat`. Launched just before the workflow began and stopped right after it finished, `powerstat` recorded the system's energy use, logging average power consumption, peak power spikes, and the total energy consumed.

The data collection was a routine process, not a one-time effort. Each time the OpenLane 2 workflow ran, `/usr/bin/time` measured the timing, and `powerstat` tracked the power metrics. After each run, the logs from both tools were parsed to extract the relevant metrics, ensuring every run was added to a growing dataset.

To make this process robust and adaptable, `power_time_flow.py`, a Python script, automates the workflow for clock frequencies suitable for each design, as it relies heavily on each implementation to determine which frequencies are relevant. The script updated the clock period in the OpenLane 2 configuration file for each frequency and ran the workflow three times at each setting. For every run, it collected timing and power data, outputting them to separate reports for each run.

Then, to summarize the gathered information, the results were averaged using `average_rpt.py` to reduce variability. Repeating and averaging the runs provided reliable data for comparing different configurations.

The inputs and outputs of these two scripts are depicted in Table 4.5. An example of the summary created after running these two scripts is presented in Section A.2 of the Appendix.

Integrating these methods into the evaluation of the flow helps to better understand the feasibility of open-source tools compared to proprietary ones. Additionally, it provides more insight

Table 4.5: Inputs and outputs of `power_time_flow.py` .

Entry	**Type**	**Description**
`frequencies`	list of num	Input for frequencies that were to be run
`top_dir`	string	Relative path to top-level directory of design runs
`stats_dir`	string	Directory were stats of the design would be saved
`output_file`	string	Name and location of the average report compilation
`Frequency`	number	Entry in `output_file` that shows the start of the mean stats for a given frequency
`Average time`	number	Mean time to run OpenLane 2 for that frequency and design
`Average power`	number	Mean power to run OpenLane 2 for that frequency and design

into how the tools function when approaching tighter constraints to the design, highlighting the variability in time consumed when performing them.

4.2 Hardware Modules Implementation and Evaluation

Section 4.1 depicted how the solution for the flow can be used and how it is built. As previously stated, to validate this flow, a hardware implementation was necessary, which can be said to demonstrate that the flow is capable of supporting this kind of development. For this reason, the system presented in Figure 3.2 was developed and serves as a modular infrastructure for future testing and system building.

The design main types of infrastructure are the following:

1. **RTL modules** - Base verilog modules that depict the fundamental behavior of the pretended portion of the design.

2. **Testbenches** - Needed to start the developed RTL modules, generating inputs to trigger their functionality and then verifying the output correctness.

3. **Wrappers** - These mainly focus on aggregating RTL modules and interconnecting them to make more complex structures. Usually, testebenches are also made to verify their correct behavior.

4. **Headers** - Define common values to be used across multiple files.

Additionally, the files used to configure IOb-Cache are also important, as it is a key part of the design structure.

4.2.1 IOb-Cache configuration

Starting with the later referred, the IOb-Cache project [17] should be cloned to the device where it would be configured.

To configure it, two distinct files can be modified: its `Makefile` and `iob_cache.py`. In Table 4.6, the configurations that are more noticeable for this implementation and the location of these files can be inferred.

Table 4.6: Main parameters to run IOb-Cache.

Variable	File	Value	Description
`BE_IF`	`Makefile`	IOb	Determines if uses native or AXI4 interface.
`BE_DATA_W`	`Makefile`	64	Word width to RAM interface.
`NLINES_W`	`iob_cache.py`	10	Configures the number of cache lines, given by 2^{NLINES_W}.
`WORD_OFFSET_W`	`iob_cache.py`	3	Sets the number of words per line, which is $2^{WORD\_OFFSET\_W}$.
`WRITE_POL`	`iob_cache.py`	0	Set to 0 for write-through or set to 1 for write-back.
`FE_DATA_W`	`iob_cache.py`	32	Word width to CPU/NDPAccs interface.

It is important to note that the width of the Cache Line can be inferred by multiplying the number of words per line of cache, $2^{WORD\_OFFSET\_W}$, by the width of the cache's word, FE_DATA_W. In this implementation, the width of the Cache Line is given by $2^3 * 32 = 256$ bits.

Setting these values allows to shape the cache to behave as the user would like. This is a suitable choice for this work, as it contributes to the modularity intended for the entire project.

4.2.2 RTL modules

As previously mentioned, a solid structure for developing and testing NDPAccs was a key point in this dissertation. The file with Verilog Headers (`.vh`), which is called `constants.vh`, is a file used throughout the whole project, meaning that any intended change on the significant parameters of the design just needs to be changed in this file for it to be propagated in the whole design.

Additional constants can be added; however, the ones that need to be defined and are currently in use throughout the project are presented in Table 4.7.

The main component to test and first developed was a simple NDPAcc. The complexity of the operation was not the primary goal but rather an example of an operation that could benefit from

[5] Computing and memory sides refer to the interfaces when seen from the cache. RAM is on the memory side, and the connection to CPU/NDPAccs is on the computing side.

Table 4.7: Constants for the implemented system.[5]

Variable	Value	Description
FE_DATA_W	32	Data width for computing side modules.
FE_ADDR_W	22	Address width for computing side modules.
FE_STRB_W	4	Bit selection for computing side modules.
BE_DATA_W	64	Data width for memory side modules.
BE_ADDR_W	24	Address width for memory side modules.
BE_STRB_W	8	Bit selection for memory side modules.
FIFO_DEPTH	16	Depth for FIFO units.

being calculated at an accelerator rather than on a CPU. With that purpose, three were created following the interface presented in Figure 4.2 and using similar state machines as the one presented at Listing 4.6.

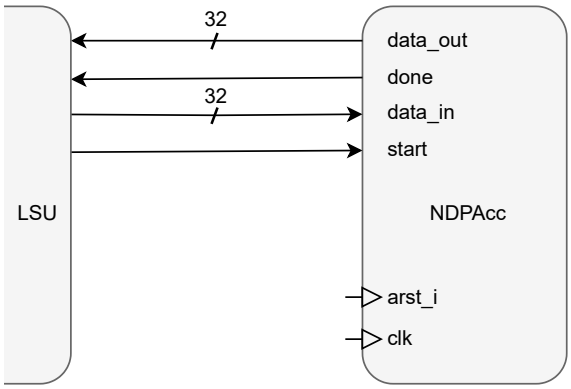

Figure 4.2: NDPAcc interface.

Listing 4.6: Design selection argument

```
always @(posedge clk or posedge arst_i) begin
      if (arst_i) begin
         state <= IDLE;
         ...
      end else begin
         case (state)
            IDLE: begin
               done <= 0;
               if (start) begin
                  state <= DONE;
                  ...
               end
            end
            DONE: begin
               done <= 1;
```

```
16                if (!start) begin
17                   state <= IDLE;
18                end
19             end
20             default: begin
21                state <= IDLE;
22                done <= 0;
23             end
24          endcase
25       end
26    end
```

The NDPAcc is triggered by setting `start` to 1 for a clock cycle, and the operation should be completed in the next clock cycle. The operation can take more clock cycles since the system is projected to wait for the `done` signal to rise to 1.

The three NDPAccs developed are contained in the files `accelerator_sum.v`, `accelerator_max_min.v` and `accelerator_property.v`. The latter gets its name from its implementation, which was designed with the concept of map-reduce in mind. It analyzes a specific characteristic of the numbers collected and outputs the number of them that possess it. In the current implementation, it indicates how many of the numbers are even.

All of these implementations are simplified since they all divide the complete word given into smaller 8-bit numbers. This was implemented to demonstrate the advantages of having multiple numbers in the input to the accelerator, which further minimizes the number of accesses needed by them. One improvement for the future is to work with complete lines of cache to perform operations closer to real-world scenarios.

To handle requests to the NDPAccs more effectively and ensure that inputs and outputs are not lost due to requests not being attended to in a timely manner, the Load-Store Unit module was created, which is present in `load_store_unit.v`.

Its interface accepts requests to use the accelerator and plans its use. Can accept multiple sequential requests if the data to be processed by the NDPAcc is sequentially organized and starting at a given base address.

The inputs and outputs of this module are shown in Figure 4.3, with the interconnection that should be done to other modules.

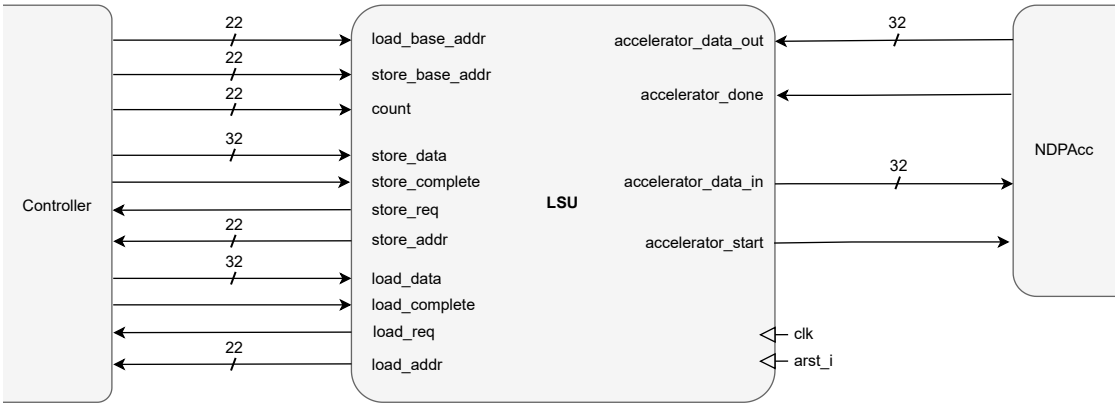

Figure 4.3: LSU interface.

Loading and storing signals are handled by the LSU after it starts operating. It also signals when to request more data and when it is received, as well as for storing the computed data.

The memory controller created utilizes a Round-Robin decision-making algorithm to determine whether the CPU or the LSU will have access to the cache. It also stores the requests in a FIFO, ensuring that no requests are left unattended or lost.

Since its interconnection is too complex, a diagram of its signals was not developed, but its header of the RTL implementation is presented in Section B.1 in the Appendix.

4.2.3 Wrappers and Testbenches

To test the developed modules, testbenches for each part needed to be created in order to validate their functionality. More than that, since these modules needed to interact with each other, being part of a bigger system, wrappers were required to ensure the correct coupling between them.

Additionally, the need to synthesize only parts of the whole design also increased the need for creating system wrappers, allowing those wrapped parts to be synthesized in one go.

These aggregations of modules can be easily depicted in Figure 3.2 and are detailed in the following enumeration:

1. A testbench for each NDPAcc was created, since each one had different needs as input, as well as on how to verify its output - `accelerator_sum_tb.v`, `accelerator_max_min_tb.v`, `accelerator_count_tb.v`;

2. To wrap the LSU with a NDPAcc, the `lsu_wrapper.v` was created as well as a testbench for it `lsu_wrapper_tb.v`;

3. The memory controller needed to be able to synthesize and tested together with the LSU wrapper, so the `ctrl_lsu_wrapper.v` and `ctrl_lsu_wrapper_tb.v` was created;

4. The memory controller needed to be able to synthesize and tested together with the LSU wrapper, so the `ctrl_lsu_wrapper.v` and `ctrl_lsu_wrapper_tb.v` was created;

5. Since the IOb-Cache always needed to have the RAM attached to work properly, `memory_wrapper.v` was developed, as well as two testbenches: `memory_wrapper_tb.v` and `memory_wrapper_loaded_tb.v`. Their difference is that the loaded one uses the system task `$readmemh` to initialize the RAM with full data before starting the operation;

6. To conjugate and test the full system, `full_system_wrapper.v` couples the previously stated wrappers into a full system and is tested by `full_system_wrapper_tb.v`.

For each of these, multiple checks were necessary since the interaction between the units was not always as expected, and their implementation was somewhat basic. However, given the nature of the project, they were sufficient to provide a base for testing the developed workflow.

Chapter 5

Results and Discussion

In this Chapter, the results of the implementation will be discussed. The process of how these values were obtained for each part of the system is presented in Chapter 4.

5.1 Toolchain

Starting with the toolchain, the tests to evaluate it primarily focused on its performance when running on a Personal Computer (PC). For that purpose, the following system was used to conduct all the performance testing:

- **Product name** - Lenovo Legion 5 15IMH05H;
- **CPU** - Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz;
- **RAM** - 16GiB of SODIMM DDR4 Synchronous 2933 MHz;
- **GPU** - GeForce RTX 2060 Mobile;
- **OS** - Ubuntu 22.04.5 LTS.

The OS was initially intended to be a version of Ubuntu 24.04 LTS; however, it was not compatible with all the necessary tools to run the flow, so an older and more stable version was chosen.

As previously stated, the results were obtained by running the flow multiple times for the same design and different frequencies. The main objective was to test how the flow behaves when the limits of the PDK used are being reached, and it cannot be further optimized to meet the proposed constraints.

Timing was the constraint chosen for testing, and with the help of the developed scripts, the maximum frequency that a design could reach was easily determined through a few iterations. After realizing that for three accelerator implementations and a combination of one of them with the LSU, the `power_time_flow.py` was set to run three times for each of the given frequencies.

These frequencies were chosen arbitrarily by analyzing the evolution of the design as its boundaries were pushed. When the maximum frequency was too high, a significant increase was made to test the evolution in that domain. However, upon reaching it, the intervals were refined, and the flow was run slightly above and below the maximum frequency.

Here, the main difference is that when it exceeds the maximum frequency for typical PVT conditions (typical transistors, nominal voltage, and 25 °C), OpenLane 2 will not produce any output, as this imposes a significant violation of the flow. Moreover, when the flow was slightly below, it would need to work particularly hard to reach the desired frequency, which was indeed the case in reality. Initially, it was assumed that as the maximum frequency was approached, the flow would theoretically consume more power and take longer to run. This was partially verified when the flow was run on battery power only. However, since it consumes a significant amount of resources to run all of this, and the battery does not hold charge very well without being connected to the power supply, the results were uneven and probably misleading, as the same environmental conditions were not guaranteed to be consistent across all the tests.

So, with that in mind, all the tests were run using the PC connected to the power source. The results from those runs are presented in Figure 5.1, which shows the power consumption across different designs, and Figure 5.2 shows the time it took to run OpenLane 2 for each design.

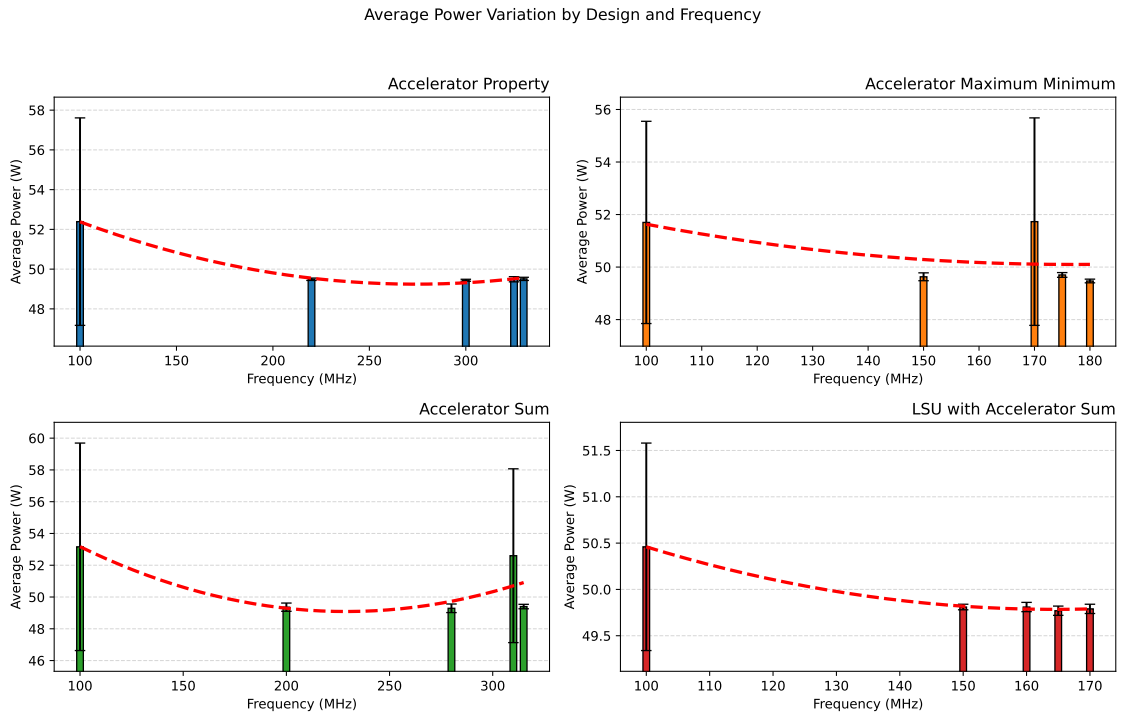

Figure 5.1: Power consumption for running OpenLane 2, where each graphic represents a different design.

Unlikely what was anticipated and partially verified when running the flow solely on battery power, the results show that, within the same design, the time and power required to complete

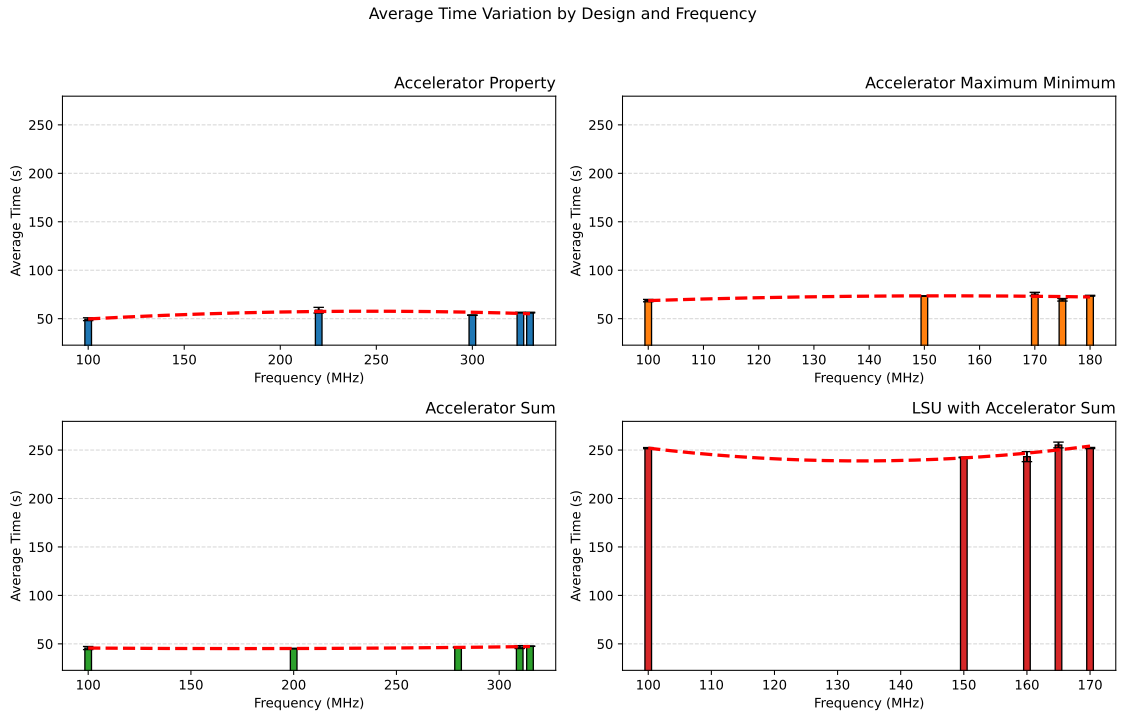

Figure 5.2: Time elapsed from start to finish of OpenLane 2, where each graphic represents a different design.

the flow are mostly constant. This does not mean that the flow did not need to make further optimizations to the system. When observing the log files and the characterization of the system at frequencies closer to the maximum frequency, it was noted that the implementation differed and that further optimizations were needed to achieve the desired results.

To estimate the power consumption, the power usage of the PC was compared to its regular usage. This included calculating the power usage when performing light tasks, such as browsing the internet or writing, as well as when the PC was entirely idle. The average consumption can be compared in Table 5.1. Outside of the average power consumption when running the flow, which was calculated as depicted in Section 4.1.4, the means were calculated by performing three 10-minute runs of a similar workload across each scenario.

Table 5.1: Power consumption through different workloads.

Workload	**Average Power (W)**	**Standard Deviation**
Idle	17.30	1.12
Light work	23.71	2.88
Running OpenLane 2	50.09	1.21

With these results, the extra steps needed for optimization are not significantly relevant to impact the performance of the OpenLane 2 flow for designs with the size and complexity of the

ones developed. On the other hand, it can be noted that for larger projects, with the need to synthesize more extensive parts of a design, the time it takes for the flow to run can be a hindrance during development.

For the `analysis.py` script, a brief performance test was conducted to infer if it would take much time during the development of an ASIC. In terms of time to run, it always took less than a minute to run sixteen different timing corners for the same design. With that, it can be concluded that its impact would be minimal on the overall development overhead.

5.2 Hardware Implementations

The flow, as implemented, already had a goal to extract key information from the design being implemented. As previously stated, power consumption, maximum frequency, and area of the design are reported using OpenLane 2 and OpenSTA, primarily through the `analysis.py` Python script; these values are saved for each run of the flow in their respective directories.

To evaluate the implemented NDPAccs and the impact of the infrastructure enabled to implement them, this analysis was run for the `accelerator_sum.v` and `lsu_wrapper.v`, where the NDPAcc used with the latter was the one mentioned earlier so that some conclusions could be taken from the obtained results. These results are shown in Tables 5.2 and 5.3.

To further test these implementations and demonstrate that variations in functionality can impact their behavior, the analysis was also conducted for the other two implemented NDPAccs. To make the text more readable, and since the first two sections can already display the main results of this type, the results of `accelerator_max_min.v` and `accelerator_property.v` are presented in Section B.2 of the Appendix.

All of these results were obtained using the Liberty files provided by the PDK (SkyWater 130nm[27]), as well as the nominal conditions for the Standard Parasitic Exchange Format (SPEF) file, which were obtained after each run of the design. For this purpose, nominal values were used, as they serve as a good benchmark for the designs. To more easily analyze the results obtained, only six representative PVT timing corners were presented, as showing all sixteen would be too extensive to present as is. Can also be noted that the "Type" column can have three values: "fast", "typical" and "slow", which represent the type of transistors used, when considering the fabrication process output. As these names suggest, depending on their quality of fabrication, the NMOS and PMOS can be faster, slower or have a typical speed, considering the usual behavior of these for the fabrication process used.

It is highlighted in each Table the maximum frequency for the most common PVT condition (typical transistors, nominal voltage, and 25 °C), as well as the maximum and minimum value of power and maximum frequency across each design. These key values provide a global perspective on each design and its overall limitations.

These results demonstrate that the implemented tools help the development of this type of system, which can be said since, when focusing on the maximum frequency estimation for the most typical values of PVT conditions (typical transistors, nominal voltage, and 25 °C), it can

be seen the gain of performance that can be made, since it goes up in the order of the tenths of megahertz. This conclusion supports the need for the implemented extension to reliably and quickly achieve the intended maximum frequency and/or power usage targets.

Table 5.2: Maximum Frequency and Power Estimation for different runs and PVTs of the Addition Accelerator.

Freq (MHz)	Type	Temp (°C)	Voltage (V)	Max Freq (MHz)	Power (mW)
100	fast	-40	1.56	213.472	0.76
		100	1.65	275.941	1.65
	typical	25	1.80	**217.367**	1.06
		100	1.80	220.660	1.12
	slow	-40	1.28	**41.983**	0.09
		100	1.60	139.290	0.54
150	fast	-40	1.56	249.186	0.75
		100	1.65	337.814	1.72
	typical	25	1.80	**254.016**	1.06
		100	1.80	259.036	1.12
	slow	-40	1.28	43.187	0.08
		100	1.60	153.480	0.51
200	fast	-40	1.56	268.590	0.76
		100	1.65	381.023	1.83
	typical	25	1.80	**280.668**	1.10
		100	1.80	288.705	1.17
	slow	-40	1.28	46.446	0.08
		100	1.60	173.605	0.54
280	fast	-40	1.56	295.207	0.75
		100	1.65	436.907	1.89
	typical	25	1.80	**309.863**	1.09
		100	1.80	319.688	1.16
	slow	-40	1.28	47.250	**0.07**
		100	1.60	184.608	0.52
310	fast	-40	1.56	297.215	0.76
		100	1.65	**435.455**	**1.89**
	typical	25	1.80	**312.076**	1.11
		100	1.80	319.688	1.17
	slow	-40	1.28	47.233	0.07
		100	1.60	185.130	0.52
Die area = 5614.28 μm^2 | **Core Area** = 3336.95 μm^2.					

Table 5.3: Maximum Frequency and Power Estimation for different runs and PVTs of the Load-Store Unit Wrapper.

Freq (MHz)	Type	Temp (°C)	V (V)	Max Freq (MHz)	Power (mW)
100	fast	-40	1.56	140.938	12.4
		100	1.65	177.124	18.4
	typical	25	1.80	**148.608**	18.0
		100	1.80	153.660	19.1
	slow	-40	1.28	**21.069**	1.01
		100	1.60	88.983	8.49
150	fast	-40	1.56	154.931	13.7
		100	1.65	200.110	20.8
	typical	25	1.80	**164.663**	20.1
		100	1.80	170.667	**21.3**
	slow	-40	1.28	21.261	1.03
		100	1.60	94.296	9.06
160	fast	-40	1.56	156.411	13.7
		100	1.65	202.898	20.9
	typical	25	1.80	**165.914**	20.0
		100	1.80	170.223	21.1
	slow	-40	1.28	21.216	1.01
		100	1.60	93.157	8.85
165	fast	-40	1.56	159.261	12.1
		100	1.65	**208.051**	18.7
	typical	25	1.80	**168.907**	17.8
		100	1.80	175.700	19.0
	slow	-40	1.28	21.274	**0.846**
		100	1.60	96.306	7.97
Die area = 94 696.7 μm^2 | **Core Area** = 84 745 μm^2.					

5.3 Discussion

The overall flow provides a solid base for the characterization and development of NDPAccs. From the previous sections, this enhances the study of these systems while also providing a stable enough infrastructure to test them without introducing too much overhead to the entire development. Additionally, the performance of the developed simple accelerators is acceptable, considering the

PDK being used.

These results do not mean that there are no limitations on the current implementation. In terms of the flow, it could be more exhaustive, as it only verifies the maximum frequency of the design by using reports from OpenSTA[14]. Additionally, the infrastructure could have been better implemented by conducting more exhaustive testing of its modules and incorporating greater modularity between them.

Other limitations noted when using these open-source tools were that some functionalities were underdeveloped or not present at all. These limitations felt were the following:

1. **Compilation and waveform viewer integration** - The simulators used, Icarus Verilog or Verilator[13], do not have a Graphical User Interface (GUI) implemented, which can be overwhelming when trying to activate all the command-line flags required. Furthermore, each time after using them, GTKWave[37] or a similar tool must be used to open the VCD output of the simulator. It would be beneficial to have an integrated system that combines simulation and waveform visualization, allowing users to select a signal in the RTL file and view it in the wave visualizer.

2. **Reuse of the compilation folder on OpenLane 2** - When using OpenLane 2, the problem of having to reinstall the OS was encountered, as the disk space ran out due to the quickly accumulating files produced by multiple runs of the same device. To tackle this, using the same project files when the same entries for OpenLane 2 are given can reduce the amount of storage needed and also provide a cleaner working space.

3. **Low compatibility with some features** - To deeply verify each implementation and to extend the flow further, the usage of the concept of back annotation into the netlist of the design, using a Standard Delay Format (SDF) file generated by OpenLane 2, was desired. However, problems arose from the start. Initially, neither the netlist could be run on either Icarus Verilog or Verilator, and neither provides excellent support for SDF back annotation. The simulators will issue warnings for not supporting certain types of delays introduced by the SDF file. Icarus Verilog has several limitations compared to VCS, and Verilator does not support it. OSS CVC, a Verilog compiler and simulator, is an open-source version of Tachyon that looks promising, as it fully supports back-annotation.

Additionally, a faster file type for waveforms, Fast Signal Trace (FST), is available, which helps facilitate the opening of extensive simulations with GTKWave. However, unfortunately, it was not possible to get it to work either. In comparison, VCS worked out of the gate, requiring only a minor adjustment to the standard cell library file of the PDK, which VCS identified as a problem. In that sense, it would be an improvement to add this type of support.

Also, a newer waveform viewer is available as an open-source tool, known as Surfer [38]. When compared to GTKWave, loading a considerably sized VCD (2.9 GiB), it loads it much faster. It uses less memory to open it, while also allowing the use of multiple threads that further accelerate the process. More than that, it still supports the previously stated FST file format,

which can further improve load times. It is also said to integrate more high-level information from modern hardware-type systems, analysis, and verification tools to simplify debugging [39]. It seems a promising tool to take a step forward in viewing waveforms and to better integrate newer features into open-source.

5.4 Evaluation of Current Open-Source Toolchains

During the development of this work, several other toolchains were identified, apart from the one chosen for the dissertation and due to the nature of this work, where the main objective was to extend one of these toolchains to better support NDPAccs, an overview of these flows seemed appropriate to include.

1. **OpenLane 2**[10] - the "next-generation" Efabless[23] flow has become a flagship open-source RTL-to-GDSII toolchain. It provides a fully automated end-to-end ASIC flow built entirely on open tools. In its default "Classic" configuration, OpenLane 2 runs RTL-to-GDSII using components such as Yosys[20] (synthesis), OpenROAD[28] (placement and routing (P&R), Magic[40] (layout editing), Netgen[41] (LVS), KLayout[42] (DRC), and others. A single Python-based configuration orchestrates the pipeline, allowing easy tuning of area, power, or timing goals. By default, it targets the SkyWater 130nm PDK (with partial support for other PDKs like GF foundries). OpenLane 2's rewrite in Python and Nix-based reproducible builds greatly improve robustness and extensibility. Its main strengths are full automation and a vibrant community: thousands of designers use OpenLane, and extensive documentation and support are available. In practice, OpenLane 2 lowers barriers by eliminating proprietary tools and guiding users through synthesis, floorplanning, P&R, and final layout generation.

2. **mflowgen**[11] - a modular flow generator aimed at flexible design exploration. Rather than a fixed script, it allows users to programmatically assemble a directed graph of "nodes" (each a sandboxed step, such as synthesis, P&R, etc.). For example, mflowgen ships with templates for both open and commercial tools (e.g., Yosys or Synopsys DC for synthesis; RePlAce[28], GrayWolf[21], or Innovus for P&R) and even provides an example 45nm ASIC design kit (FreePDK45[43]). Designers can "instantiate subgraphs" and apply parameter sweeps, enabling parallel design-space runs (e.g., sweeping clock constraints or exploring area–power tradeoffs). In short, mflowgen abstracts away tool invocation behind reusable nodes and parameters. Its advantages include extreme flexibility and scalability: the same flow definition can target different PDKs (from older nodes to advanced nodes, such as TSMC 16/28 or SkyWater 130nm) by swapping in different technology libraries. mflowgen's comprehensive Python API and documentation enable researchers to rapidly prototype or modify flows without needing to rewrite scripts. This makes mflowgen ideal for sophisticated projects that need custom flows or systematic exploration.

3. **Hammer**[44] - originated from UC Berkeley's RISC-V/VLSI projects, is a plugin-based physical-design framework that unifies many EDA tools under one API. Rather than running a fixed flow, Hammer generates the low-level scripts itself. Designers specify high-level intents (e.g., floorplans, timing constraints) and Hammer invokes the chosen tool plugins (e.g., Synopsys Design Compiler, Cadence Innovus, or open-source versions like Yosys/OpenROAD) using those parameters. This abstraction means Hammer is largely technology-agnostic; it has been applied to real chips, ranging from 28nm to advanced open PDKs (e.g., the 7nm ASAP7 library[28]). In essence, Hammer provides reusability and portability: the same flow description can be retargeted by loading different plugins or technology JSONs. Its design philosophy ("tame the complexity of EDA tools and PDKs") emphasizes speed and flexibility. The framework is therefore best suited for advanced use cases that require quick iteration across tools or processes—e.g., research prototypes or multi-project chip generators—where manually rewriting scripts for each tool or technology would be impractical.

4. **LibreLane**[45] - is essentially an OpenLane flow hosted in the cloud. Built by the American University in Cairo's Open Hardware Lab under FOSSi[46], it inherits the same core toolset as OpenLane, including OpenROAD, Yosys, Magic, Netgen, and KLayout. Its defining feature is browser-based access: the entire RTL-to-GDSII flow runs in a Google Colaboratory notebook. This zero-install approach removes hardware barriers—anyone with a web browser can compile a design on SkyWater 130nm (the target PDK) without local compute power or software setup. LibreLane thus democratizes ASIC design by enabling instant experimentation and collaboration. While it uses the same robust flow internally, LibreLane's convenience is its main advantage: designers can learn and iterate on real chip layouts with no installation hassle. In short, LibreLane broadens participation in open ASIC projects by combining the proven OpenLane flow with cloud flexibility.

5. **VSDflow**[47] - an educational flow developed alongside VLSI System Design courses. Its goal is to teach students complete RTL-to-GDSII implementation using open tools. VSDflow offers a plug-and-play C-to-GDSII flow: users can start from C (or RTL), and VSDflow takes over the rest. In practice, a design in C is first converted to Verilog (via an Algorithm to Hardware Intermediate Representation (AHIR) compiler), then synthesized by Yosys, P&R by the Qflow tools, and finally timing-verified with OpenTimer[15]. The output is a GDSII file, along with performance and area reports. Importantly, VSDflow is structured so that learners can intervene or add "hooks" at any stage (e.g., supplying custom constraints or switching tools). Initially demonstrated on an OSU 180nm kit (and now extending to SkyWater's 130nm process), VSDflow emphasizes simplicity and hands-on learning. Its documentation includes tutorials and example projects. Compared to other flows, VSDflow's advantage lies in its pedagogical focus: it prioritizes clarity and ease of setup over aggressive optimization, making it an ideal stepping stone for newcomers to ASIC design.

6. **Qflow**[29] - one of the oldest fully open-source ASIC flows, renowned for its simplicity and low barriers. It strings together a minimal toolchain: Yosys for logic synthesis, GrayWolf for placement, QRouter[48] for routing, and Magic (with Netgen) for layout and LVS. (OpenSTA/OpenTimer can be used for timing sign-off in recent versions.) By default, Qflow utilizes the OSU open-source standard-cell libraries; however, community projects have adapted it to support newer processes, such as SkyWater 130nm. Unlike more automated flows, Qflow typically requires some user configuration, but it offers extensive documentation and a GUI that walks designers through each step. Its design philosophy is straightforward: every needed external tool is invoked in a standard sequence, and errors are easier to isolate. This makes Qflow popular for small to medium designs (generally up to $\tilde{3}$0k gates at 65nm and above), hobby projects, and quick proofs-of-concept. In short, Qflow's chief strength is its accessibility: it has no proprietary dependencies, a lightweight footprint, and transparent operation, allowing designers to learn ASIC flows without an EDA license.

7. **LibrEDA**[49] - a cutting-edge Rust-based library focusing exclusively on the back-end of ASIC design (placement, legalization, routing). It does not include RTL synthesis; instead, it operates on gate-level netlists and leverages open PDK data (e.g., FreePDK45). The framework provides standardized I/O for formats such as OASIS, LEF/DEF, and example algorithms, including an electron-placer for initial cell placement, a Tetris-legalizer to adjust spacing, and a Mycelium-router for global routing. The goal is to create an open ecosystem where researchers can collaborate to develop and test new physical design algorithms. LibrEDA's creators emphasize democratization and transparency of EDA tools. Being written in Rust, it offers high performance and reliability for large datasets. In essence, LibrEDA serves niche use cases: it is a research and testbed platform for physical design experts and academics, rather than a turnkey flow. Its advantage lies in its openness and modularity, enabling experimentation with novel P&R strategies that would be difficult to integrate into a monolithic flow.

In Table 5.4, there is a summary of key benefits and differences for the presented flows.

Based on the provided summary, there are distinct approaches to the same problem of designing an overall flow that encompasses the work of RTL-GDSII implementations. It is beneficial to see the various uses of tools to generate broader choices, as well as observe some convergences, such as the widespread adoption of OpenROAD tools and the SkyWater 130nm PDK, which can accelerate the advancement of open-source tools and flows. Some of these have additional approaches, such as a GUI, using Rust for configurations, and utilizing nodes that attempt to mimic the newer foundry available nodes, like the ASAP7.

One clear advantage of these tools when compared to proprietary ones is their availability to everyone in any station, since the high prices of proprietary tools limit the companies and institutions to have limited number of licenses and thus just providing them in specific machines, which can cause problems for an easy development (computer maintenance, need to have that machine

working, being at the same local as the PC, among others). It is a true relief that, regardless of the machine, the necessary tool for developing the design is available for free and just works.

Table 5.4: Key benefits and differences of different open-source ASIC development flows.

Flow	**Key Tools/Components**	**Supported PDKs**	**Main Advantages**
OpenLane 2[10]	Yosys[20], OpenROAD[28], Magic[40], Netgen[41], KLayout[42]	SkyWater 130nm[27] (default), GlobalFoundries GF180MCU[50] (implemented), Guides how to incorporate own PDK	Fully automated RTL-GDSII flow; Python-configured; mature community support
mflowgen[11]	Yosys[20], RePlAce[28], GrayWolf[21], QRouter[48]	FreePDK45[43] (default), Supports other nodes (SkyWater 130nm[27], TSMC/GlobalFoundries nodes)	Highly modular "node-based" flow; supports open and commercial tools; built-in Assembly Design Kits (ADKs)
Hammer[44]	Plugin API for Synopsys DC, Innovus, OpenROAD[28], among others	ASAP7 (7nm) (default), SkyWater 130nm[27] (implemented), Explains how to use NanGate45 (45nm), others can be configured via plugins	Unified API for vendor tools; plugin architecture (tools/PDKs abstracted); enables reusable flows across techs; ideal for rapid prototyping and research
LibreLane[45]	Yosys[20], OpenROAD[28], Magic[40], Netgen[41], KLayout[42] (same as OpenLane)	SkyWater 130nm[27] (default), GlobalFoundries GF180MCU[50] (implemented), Guides how to incorporate own PDK	Browser/Colab-based OpenLane; zero-install Cloud flow; accessible anytime from any device; great for learning and collaboration
VSDflow[47]	Yosys[20], GrayWolf[21], QRouter[48], OpenTimer[15] (for STA)	Originally OSU 180nm[50], also SkyWater 130nm[27]	Educational RTL-to-GDSII flow; highly automated; integrates tutorials and examples; "hooks" allow swapping steps; prioritizes teachability over extreme optimization
Qflow[29]	Yosys[20], GrayWolf[21], QRouter[48], Magic[40], Netgen[41]	OSU standard-cell libraries, SkyWater 130nm[27]	Simple, tried-and-true flow; having a GUI (unique); minimal setup; ideal for small/medium designs ($\leq$30k gates); fully open source with extensive docs
LibrEDA[49]	Rust libraries: electron-placer, tetris-legalizer, mycelium-router	FreePDK45[43] (demonstrated), support for other nodes	Research-focused P&R framework; having its code based on Rust

Chapter 6

Conclusions and Future Work

The development of ASICs is not a recent phenomenon, and it is certainly a field that has considerable potential for future growth. Although it has been an important aspect of the daily lives of almost everyone, its development is still far from democratized due to a low offer of tools to develop it. This low proliferation is caused by the high costs of having proprietary tools to develop them and that open-source tools do not have the right capabilities or take too many resources when compared to the previous ones. Fortunately, this is where the proposed solution in this dissertation comes into play, contributing to more reliable and capable open-source flows.

In addition to ASICs, NDPAccs are becoming increasingly common due to the recent trend of developing more specialized ICs, such as those for machine-learning applications, biomedics, and others. With that, this project demonstrates how infrastructure can be created to support the testing and development of NDPAccs, as well as extend the functionalities of the already comprehensive OpenLane 2.

This implementation can now extract more characteristics of the developed IC by primarily leveraging Icarus Verilog and OpenSTA, which contribute to the power, timing, and area analysis. Furthermore, the flow implemented is efficient for this type of project, as it enables the use of development tools without introducing excessive overhead to the overall development of the NDPAcc.

The infrastructure created to test and characterize NDPAccs, as well as the extension of OpenLane 2 with new capabilities, represents a significant step towards broader availability of tools for ASIC design.

6.1 Future Work

The following improvements to the presented solution could be explored:

1. **Add more related NDPAccs capabilities to the flow** - The flow could be able to have more statistics towards NDPAccs, for example, measuring the benefit of having this system when compared to a traditional CPU. The actual implementation of a CPU in this environment can help the development of real-world scenarios to test the designs.

2. **Extend modularity** - Modularity could be extended in both the flow and infrastructure. A configuration file that can be used across the fully implemented flow, similar to how OpenLane 2 operates, would be a key benefit, as the extensions implemented need to be configured individually.

3. **Refine scripts** - The developed scripts could be improved for a more user-friendly experience by not having to be directly altered when not too profound changes are to be made. For example, `power_time_flow.py` needs to be directly modified to run for different frequencies.

4. **Add more exhaustive testing** - The provided testbenches can be improved by being more unpredictable to the system, with testing variations across each run, and also try to mimic more real use scenarios.

5. **Test more tools to add** - Testing more open-source tools to add to the flow, such as the ones presented in Chapters 2 and 5, which can enrich it and make a full open-source flow provide industry-like ASIC design experience.

6. **Explore new application domains** - Adapting the flow for specific application areas such as machine learning accelerators, IoT devices, or biomedical implants, where custom silicon can offer significant advantages, could increase its relevance. Developing application-specific libraries or modules that can be easily integrated into designs for these domains would add versatility.

These remarks show that there is still a lot to be explored in this growing field of open-source ASIC design flow, and works such as this can help the dissemination of the development of ICs.

References

[1] João Vieira et al. "NDPmulator: Enabling Full-System Simulation for Near-Data Accelerators From Caches to DRAM". In: *IEEE Access* 12 (2024). Conference Name: IEEE Access, pp. 10349–10365. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2024.3352924.

[2] Daichi Fujiki et al. *In-/Near-Memory Computing*. Synthesis Lectures on Computer Architecture. Cham: Springer International Publishing, 2021. ISBN: 978-3-031-00644-9. DOI: 10.1007/978-3-031-01772-8.

[3] Win. A. Wulf et al. *Hitting the memory wall*. 1994. DOI: 10.1145/216585.216588.

[4] *GitHub - Rodrialves/NDPAcc-workflow*. URL: https://github.com/Rodrialves/NDPAcc-workflow (accessed on 06/30/2025).

[5] Onur Mutlu et al. "A modern primer on processing in memory". In: *Emerging computing: from devices to systems: looking beyond Moore and Von Neumann*. Springer, 2022. URL: https://link.springer.com/chapter/10.1007/978-981-16-7487-7_7.

[6] UC Berkeley Architecture Research. *Gemmini*. en. URL: https://github.com/ucb-bar/gemmini.

[7] Jeffrey Dean et al. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[8] Hasan Genc et al. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 769–774. DOI: 10.1109/DAC18074.2021.9586216.

[9] Masakazu Tanomoto et al. "A CGRA-based approach for accelerating convolutional neural networks". In: *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE. 2015, pp. 73–80.

[10] *efabless/openlane2*. original-date: 2023-01-16T00:29:32Z. Dec. 30, 2024. URL: https://github.com/efabless/openlane2 (accessed on 01/02/2025).

[11] *mflowgen/mflowgen*. original-date: 2019-11-10T01:48:23Z. Dec. 30, 2024. URL: https://github.com/mflowgen/mflowgen (accessed on 01/02/2025).

[12] *Icarus Verilog*. en. URL: https://steveicarus.github.io/iverilog/.

[13] Wilson Snyder. *Verilator*. en. URL: https://www.veripool.org/verilator/.

[14] Tutu Ajayi et al. "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC '19. Las Vegas, NV, USA: Association for Computing Machinery, 2019. DOI: 10.1145/3316781.3326334.

[15] Tsung-Wei Huang et al. “OpenTimer: A high-performance timing analysis tool”. In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2015, pp. 895–902. DOI: 10.1109/ICCAD.2015.7372666.

[16] *OpenRAM*. URL: https://openram.org/ (accessed on 01/02/2025).

[17] João V. Roque et al. “IOb-Cache: A High-Performance Configurable Open-Source Cache”. In: *Algorithms* 14.8 (2021). ISSN: 1999-4893. DOI: 10.3390/a14080218.

[18] Luke Wren. *Wren6991/Hazard3*. original-date: 2021-05-22T09:20:12Z. Jan. 7, 2025. URL: https://github.com/Wren6991/Hazard3 (accessed on 01/07/2025).

[19] Alex Carsello et al. “mflowgen: a modular flow generator and ecosystem for community-driven physical design: invited”. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. DAC ’22: 59th ACM/IEEE Design Automation Conference. San Francisco California: ACM, July 10, 2022, pp. 1339–1342. ISBN: 978-1-4503-9142-9. DOI: 10.1145/3489517.3530633.

[20] *YosysHQ/oss-cad-suite-build*. original-date: 2021-02-01T08:34:29Z. Jan. 2, 2025. URL: https://github.com/YosysHQ/oss-cad-suite-build (accessed on 01/02/2025).

[21] rubund. *rubund/graywolf*. original-date: 2014-10-11T09:35:44Z. Dec. 30, 2024. URL: https://github.com/rubund/graywolf (accessed on 01/02/2025).

[22] *Integration with OpenROAD — mflowgen documentation*. URL: https://mflowgen.readthedocs.io/en/latest/stdlib-openroad.html#openroad (accessed on 01/07/2025).

[23] *Efabless*. en. URL: https://efabless.com/.

[24] Mohamed Gaber et al. *OpenLane 2: Making the Most Popular Open Source ASIC Flow Modular*. URL: https://woset-workshop.github.io/PDFs/2024/17_OpenLane_2_Making_the_Most_.pdf.

[25] *GitHub - The-OpenROAD-Project/OpenLane: OpenLane is an automated RTL to GDSII flow based on several components including OpenROAD, Yosys, Magic, Netgen and custom methodology scripts for design exploration and optimization.* GitHub. URL: https://github.com/The-OpenROAD-Project/OpenLane (accessed on 01/02/2025).

[26] Sarah Hesham et al. “Digital ASIC Implementation of RISC-V: OpenLane and Commercial Approaches in Comparison”. In: *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE. 2021, pp. 498–502.

[27] *Welcome to SkyWater SKY130 PDK’s documentation! — SkyWater SKY130 PDK 0.0.0-369-g7198cf6 documentation*. URL: https://skywater-pdk.readthedocs.io/en/main/index.html (accessed on 01/02/2025).

[28] *OpenROAD*. URL: https://theopenroadproject.org/ (accessed on 01/05/2025).

[29] *Qflow 1.3: An Open-Source Digital Synthesis Flow*. URL: http://opencircuitdesign.com/qflow/ (accessed on 06/12/2025).

[30] Wendong Wang et al. “Aging-Resilient SRAM-based True Random Number Generator for Lightweight Devices”. In: *Journal of Electronic Testing* 36 (June 2020). DOI: 10.1007/s10836-020-05881-6.

[31] Ajay Sudhir Bale et al. “Comparative Evaluation of 6T, 8T and 10T Ternary SRAM Cells Schemes for Future VLSI and Power Electronics”. In: *2024 Second International Conference on Smart Technologies for Power and Renewable Energy (SPECon)*. Apr. 2024, pp. 1–7. DOI: 10.1109/SPECon61254.2024.10537347.

[32] Sparsh Mittal et al. “A survey of SRAM-based in-memory computing techniques and applications”. In: *Journal of Systems Architecture* 119 (Oct. 1, 2021), p. 102276. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2021.102276.

[33] Jesse Cirimelli-Low et al. “SRAM Design with OpenRAM in SkyWater 130nm”. In: *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2023 IEEE International Symposium on Circuits and Systems (ISCAS). ISSN: 2158-1525. May 2023, pp. 1–5. DOI: 10.1109/ISCAS46773.2023.10181379.

[34] *Pico-series Microcontrollers - Raspberry Pi Documentation*. URL: https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html (accessed on 01/07/2025).

[35] Nathan Binkert et al. “The gem5 simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.

[36] Yannick Becker et al. “Software based estimation of software induced energy dissipation with powerstat”. In: *From Science to Society: The Bridge provided by Environmental Informatics* (2017), pp. 69–73.

[37] *GTKWave*. en. URL: https://gtkwave.sourceforge.net/.

[38] *Surfer*. en. URL: https://surfer-project.org/.

[39] Frans Skarman et al. “Surfer—An Extensible Waveform Viewer”. In: ().

[40] Timothy Edwards. *Magic VLSI Layout Tool*. en. URL: https://github.com/RTimothyEdwards/magic.

[41] Timothy Edwards. *Netgen complete LVS tool for comparing SPICE or verilog netlists*. en. URL: https://github.com/RTimothyEdwards/netgen.

[42] Matthias Koefferlein. *KLayout layout viewer and editor*. URL: https://www.klayout.de/.

[43] *FreePDK45 | NC State EDA*. en-US. URL: https://eda.ncsu.edu/freepdk/freepdk45/.

[44] Ucb-Bar. *Hammer*. en. URL: https://github.com/ucb-bar/hammer/tree/7dfbb0b581048dc61523d35aebda924a78eb0428.

[45] Librelane. *LibreLane*. en. URL: https://github.com/librelane/librelane.

[46] *FOSSi Foundation: the international not for profit organisation which promotes and protects the open source silicon chip movement*. en. URL: https://fossi-foundation.org/.

[47] Kunalg. *VSDFLOW*. en. URL: https://github.com/kunalg123/vsdflow.

[48] Timothy Edwards. *Qrouter detail router for digital ASIC designs*. en. URL: https://github.com/RTimothyEdwards/qrouter.

[49] *LIBReda - Libre Chip Design Framework*. URL: https://libreda.org/.

[50] Google. *PDK for GlobalFoundries' 180nm MCU bulk process technology (GF180MCU)*. en. URL: https://github.com/google/gf180mcu-pdk.

Appendix A

Flow

A.1 Simulation Script

Here is presented the full Python script `run_simulation_v2.py`, which is responsible for making the preliminary simulation of a design easier with the available simulators.

```
1  import os
2  import subprocess
3  import shutil
4  import yaml
5  import argparse
6
7
8  # Configuration
9  YAML_FILE = "config_run.yaml"  # Name of the YAML configuration file
10 IVERILOG_CMD = "iverilog"  # Icarus Verilog compiler
11 VERILATOR_CMD = "verilator"  # Verilator compiler
12 VVP_CMD = "vvp"            # Icarus Verilog runtime
13 GTKWAVE_CMD = "gtkwave"    # GTKWave viewer
14
15
16 def check_tools(compiler):
17     """Check if required tools are installed based on the compiler choice.
           """
18     if compiler == "iverilog":
19         tools = [IVERILOG_CMD, VVP_CMD, GTKWAVE_CMD]
20     else:  # verilator
21         tools = [VERILATOR_CMD, GTKWAVE_CMD]
22
23     for tool in tools:
24         if shutil.which(tool) is None:
25             print(f"Error: {tool} is not installed or not in PATH.")
26             exit(1)
```

```
27
28
29 def load_yaml_config(yaml_file):
30     """Load the YAML configuration file."""
31     with open(yaml_file, 'r') as file:
32         config = yaml.safe_load(file)
33     return config
34
35
36 def compile_verilog(compiler, src_files, sim_out, sim_location,
       src_location, pdk_files=None):
37     """Compile the Verilog files into an executable using the specified
           compiler."""
38     if compiler == "iverilog":
39         if pdk_files is not None:
40             cmd = ["iverilog", "-o", sim_out] + src_files.split()
41         else:
42             cmd = ["iverilog", "-o", sim_out] + src_files.split()
43         print(f"Compiling with command: {' '.join(cmd)}")
44         try:
45             subprocess.run(cmd, check=True)
46             print(f"Compilation successful. Generated: {sim_out}")
47         except subprocess.CalledProcessError as e:
48             print(f"Compilation failed: {e}")
49             exit(1)
50     else:  # verilator
51         # Ensure -I options are formatted as a single string (e.g., -I../
               path)
52         cmd = ["verilator"] + src_files.split() + ["-o", f"{sim_out}.v"]
53         try:
54             subprocess.run(cmd, check=True)
55             print(f"Verilator compilation successful. Generated C++ files
                   in obj_dir.")
56             # Build the executable
57             build_cmd = ["make", "-C", "obj_dir", "-f", f"V{sim_out}.mk",
                   f"V{sim_out}"]
58             print(f"Building with command: {' '.join(build_cmd)}")
59             subprocess.run(build_cmd, check=True)
60             print(f"Build successful. Generated executable: obj_dir/V{
                   sim_out}")
61         except subprocess.CalledProcessError as e:
62             print(f"Verilator compilation or build failed: {e}")
63             exit(1)
64
65
```

```
66  def run_simulation(compiler, sim_out, use_sdf=False, sdf_file=None):
67      """Run the simulation using the specified compiler's runtime,
            optionally with SDF."""
68      if compiler == "iverilog":
69          cmd = [VVP_CMD, sim_out]
70          if use_sdf and sdf_file:
71              cmd += [f"+sdf_file={sdf_file}"]
72          print(f"Running simulation with command: {' '.join(cmd)}")
73          try:
74              subprocess.run(cmd, check=True)
75              print("Simulation completed.")
76          except subprocess.CalledProcessError as e:
77              print(f"Simulation failed: {e}")
78              exit(1)
79      else:  # verilator
80          cmd = [f"./obj_dir/V{sim_out}"]
81          if use_sdf and sdf_file:
82              cmd += [f"+sdf_file={sdf_file}"]
83          print(f"Running simulation with command: {' '.join(cmd)}")
84          try:
85              subprocess.run(cmd, check=True)
86              print("Simulation completed.")
87          except subprocess.CalledProcessError as e:
88              print(f"Simulation failed: {e}")
89              exit(1)
90
91
92  def open_gtkwave(wave_out):
93      """Open GTKWave to view the waveform file."""
94      waveform_file = f"{wave_out}.vcd"
95      if os.path.exists(waveform_file):
96          cmd = [GTKWAVE_CMD, waveform_file]
97          try:
98              subprocess.Popen(cmd)
99              print(f"Opened GTKWave with {waveform_file}")
100         except Exception as e:
101             print(f"Failed to open GTKWave: {e}")
102     else:
103         print(f"Error: Waveform file {waveform_file} not found.")
104
105
106 def main():
107     """Main function to orchestrate the simulation process."""
108     parser = argparse.ArgumentParser(description="Run Verilog simulation
            with configurable source files.")
```

```
109     parser.add_argument(
110         "--source-set",
111         nargs="+",
112         choices=["fs_tb", "ctrl_wrapper_tb", "mem_tb", "loaded_mem_tb", "
                acc_tb", "ram_tb", "fifo_tb", "lsu_tb", "lsu_wrapper_tb", "
                acc_max_min_tb", "acc_property_tb"],
113         default=["fs_tb"],
114         help="Choose which simulation to run (default: fs_tb)."
115     )
116     parser.add_argument(
117         "--compiler",
118         choices=["iverilog", "verilator"],
119         default="iverilog",
120         help="Choose which Verilog compiler to use (default: iverilog)."
121     )
122     parser.add_argument(
123         "--use-sdf",
124         action="store_true",
125         help="Enable SDF backannotation during simulation (SDF file must
                be specified in config.yaml)."
126     )
127     parser.add_argument(
128         "--wave",
129         action="store_true",
130         help="Open GTKWave to view the waveform output after simulation."
131     )
132     args = parser.parse_args()
133
134     check_tools(args.compiler)
135     config = load_yaml_config(YAML_FILE)
136     sim_location = config['sim_location']
137     src_location = config['src_location']
138     base_dir_verilog = config['location_source_verilog']
139     flags_include = config['flags_include']
140
141
142     # Variables to hold configuration for the selected testbench
143     src_files = []
144     wave_out = ""
145     sim_out = ""
146     sdf_file = None
147
148     # Combine source files and fetch SDF file from the selected sets
149     if "fs_tb" in args.source_set:
```

```
150         print("\n------------------------\nRunning full system testbench.\
                n------------------------\n")
151         src_files = config['tb_files_top_module'] + config['IoB_files'] +
                config['ram_files'] + config['acc_files'] + config['ctrl_files'
                ] + config['lsu_files'] + config['fifo_files'] + config['
                lsu_wrapper_files']
152         wave_out = config['wave_out_top_module']
153         sim_out = config['sim_out_top_module']
154         sdf_file = config.get('sdf_file_top_module', None)
155     elif "ctrl_wrapper_tb" in args.source_set:
156         print("\n------------------------\nRunning control testbench.\n
                ------------------------\n")
157         src_files = config['tb_files_ctrl_wrapper_module'] + config['
                ctrl_lsu_wrapper_files'] + config['ctrl_files'] + config['
                acc_files'] + config['lsu_files'] + config['fifo_files'] +
                config['lsu_wrapper_files']
158         wave_out = config['wave_out_ctrl_wrapper_module']
159         sim_out = config['sim_out_ctrl_wrapper_module']
160         sdf_file = config.get('sdf_file_ctrl_module', None)
161     elif "mem_tb" in args.source_set:
162         print("\n------------------------\nRunning memory testbench.\n
                ------------------------\n")
163         src_files = config['tb_files_mem_module'] + config['IoB_files'] +
                config['ram_files'] + config['acc_files']
164         wave_out = config['wave_out_mem_module']
165         sim_out = config['sim_out_mem_module']
166         sdf_file = config.get('sdf_file_mem_module', None)
167     elif "loaded_mem_tb" in args.source_set:
168         print("\n------------------------\nRunning loaded memory testbench
                .\n------------------------\n")
169         src_files = config['tb_files_mem_loaded_module'] + config['
                ram_files'] + config['acc_files'] + config['IoB_files']
170         wave_out = config['wave_out_mem_loaded_module']
171         sim_out = config['sim_out_mem_loaded_module']
172         sdf_file = config.get('sdf_file_mem_loaded_module', None)
173     elif "acc_tb" in args.source_set:
174         print("\n------------------------\nRunning accelerator testbench.\
                n------------------------\n")
175         src_files = config['tb_files_acc_module'] + config['acc_files']
176         wave_out = config['wave_out_acc_module']
177         sim_out = config['sim_out_acc_module']
178         sdf_file = config.get('sdf_file_acc_module', None)
179     elif "ram_tb" in args.source_set:
180         print("\n------------------------\nRunning RAM testbench.\n
                ------------------------\n")
```

```
181         src_files = config['tb_files_ram_module'] + config['ram_files']
182         wave_out = config['wave_out_ram_module']
183         sim_out = config['sim_out_ram_module']
184         sdf_file = config.get('sdf_file_ram_module', None)
185     elif "fifo_tb" in args.source_set:
186         print("\n------------------------\nRunning FIFO testbench.\n
                ------------------------\n")
187         src_files = config['tb_files_fifo_module'] + config['fifo_files']
188         wave_out = config['wave_out_fifo_module']
189         sim_out = config['sim_out_fifo_module']
190         sdf_file = config.get('sdf_file_fifo_module', None)
191     elif "lsu_tb" in args.source_set:
192         print("\n------------------------\nRunning LSU testbench.\n
                ------------------------\n")
193         src_files = config['tb_files_lsu_module'] + config['lsu_files'] +
                config['acc_files'] + config['fifo_files']
194         wave_out = config['wave_out_lsu_module']
195         sim_out = config['sim_out_lsu_module']
196         sdf_file = config.get('sdf_file_lsu_module', None)
197     elif "lsu_wrapper_tb" in args.source_set:
198         print("\n------------------------\nRunning system wrapper
                testbench.\n------------------------\n")
199         src_files = config['tb_files_lsu_wrapper_module'] + config['
                lsu_wrapper_files'] + config['lsu_files'] + config['acc_files']
                 + config['fifo_files']
200         wave_out = config['wave_out_lsu_wrapper_module']
201         sim_out = config['sim_out_lsu_wrapper_module']
202         sdf_file = config.get('sdf_file_lsu_wrapper_module', None)
203     elif "acc_max_min_tb" in args.source_set:
204         print("\n------------------------\nRunning accelerator max/min
                testbench.\n------------------------\n")
205         src_files = config['tb_files_acc_max_min_module'] + config['
                acc_max_min_files']
206         wave_out = config['wave_out_acc_max_min_module']
207         sim_out = config['sim_out_acc_max_min_module']
208         sdf_file = config.get('sdf_file_acc_max_min_module', None)
209     elif "acc_property_tb" in args.source_set:
210         print("\n------------------------\nRunning accelerator property
                testbench.\n------------------------\n")
211         src_files = config['tb_files_acc_property_module'] + config['
                acc_property_files']
212         wave_out = config['wave_out_acc_property_module']
213         sim_out = config['sim_out_acc_property_module']
214         sdf_file = config.get('sdf_file_acc_property_module', None)
215
```

```
216     # Validate SDF usage
217     if args.use_sdf and not sdf_file:
218         print(f"Error: --use-sdf is enabled, but no SDF file is specified
                for {args.source_set} in config.yaml.")
219         exit(1)
220 
221     pdk_files = None
222     if args.use_sdf and sdf_file:
223         pdk_files = " ".join(f"-l {f}, " for f in config["pdk_files"])
224         print(f"\nPDK files for SDF: {pdk_files}\n")
225 
226     # Prepend base_dir_verilog to each file in src_files
227     src_files = [os.path.join(base_dir_verilog, file) for file in
            src_files]
228 
229     # Join all source files into a single string
230     src_files_str = ""
231 
232     for flag in flags_include:
233         if isinstance(flag, str):
234             src_files_str += f"{flag} "
235         elif isinstance(flag, dict):
236             for key, value in flag.items():
237                 src_files_str += f"{key} {value} "
238 
239     src_files_str += " ".join(src_files)
240 
241 
242     # Compile and run simulation based on the chosen compiler
243     compile_verilog(args.compiler, src_files_str, sim_out, sim_location,
            src_location, pdk_files)
244     run_simulation(args.compiler, sim_out, use_sdf=args.use_sdf, sdf_file=
            sdf_file)
245     if args.wave:
246         open_gtkwave(wave_out)
247 
248 
249 if __name__ == "__main__":
250     main()
```

A.2 Reports

Example of the report output of `average_rpt.py`.

Listing A.1: Flow Stats Dataset

```
Frequency: 100 MHz
Average Time: 251.97 seconds
Stddev Time: 0.60 seconds
Average Power: 50.46 Watts
Stddev Power: 1.12 Watts

Frequency: 150 MHz
Average Time: 242.37 seconds
Stddev Time: 0.34 seconds
Average Power: 49.81 Watts
Stddev Power: 0.03 Watts

Frequency: 160 MHz
Average Time: 243.19 seconds
Stddev Time: 5.16 seconds
Average Power: 49.81 Watts
Stddev Power: 0.05 Watts

Frequency: 165 MHz
Average Time: 255.23 seconds
Stddev Time: 2.96 seconds
Average Power: 49.77 Watts
Stddev Power: 0.05 Watts

Frequency: 170 MHz
Average Time: 252.07 seconds
Stddev Time: 0.62 seconds
Average Power: 49.79 Watts
Stddev Power: 0.05 Watts
```

Appendix B

Hardware

B.1 RTL Modules

Header of the `memory_controller.v` file, showing which connections it takes.

Listing B.1: Memory Controller inputs and outputs

```
1
2  `include "constants.vh"
3
4  module memory_controller (
5     // CPU interface
6     input wire           cpu_valid,
7     input wire [`FE_ADDR_W-1 :0] cpu_addr,
8     input wire [`FE_DATA_W-1 :0] cpu_wdata,
9     input wire [`FE_STRB_W-1 :0] cpu_wstrb,
10    output reg [`FE_DATA_W-1 :0] cpu_rdata,
11    output reg           cpu_rvalid,
12    output wire          cpu_ready,
13
14    // Accelerator interface
15    input wire           acc_valid,
16    input wire [`FE_ADDR_W-1 :0] acc_addr,
17    input wire [`FE_DATA_W-1 :0] acc_wdata,
18    input wire [`FE_STRB_W-1 :0] acc_wstrb,
19    output reg [`FE_DATA_W-1 :0] acc_rdata,
20    output reg           acc_rvalid,
21    output wire          acc_ready,
22
23    // Cache interface
24    output wire          cache_valid,
25    output wire [`FE_ADDR_W-1 :0] cache_addr,
26    output wire [`FE_DATA_W-1 :0] cache_wdata,
27    output wire [`FE_STRB_W-1 :0] cache_wstrb,
```

```
28      input wire [`FE_DATA_W-1 :0] cache_rdata,
29      input wire           cache_rvalid,
30      input wire           cache_ready,
31
32      // Clock and reset
33      input wire     clk,
34      input wire     arst_i
35  );
```

B.2 Results

Here the remainder of the results obtained to the other implemented accelerators, `accelerator_max_min.v` and `accelerator_property.v`, are presented.

Table B.1: Maximum Frequency and Power Estimation for different runs and PVTs of the Accelerator for maximum and minimum calculation.

Freq (MHz)	Type	Temp (°C)	V (V)	Max Freq (MHz)	Power (mW)
100	fast	-40	1.56	156.411	1.53
		100	1.95	212.779	**3.61**
	typical	25	1.80	**153.660**	2.13
		100	1.80	156.411	2.23
	slow	-40	1.28	**27.375**	0.168
		100	1.60	95.394	1.06
150	fast	-40	1.56	172.010	1.47
		100	1.95	241.830	3.60
	typical	25	1.80	**169.563**	2.05
		100	1.80	173.147	2.16
	slow	-40	1.28	28.013	0.150
		100	1.60	101.685	0.984
170	fast	-40	1.56	179.305	1.40
		100	1.95	256.000	3.49
	typical	25	1.80	**177.604**	1.97
		100	1.80	180.292	2.06
	slow	-40	1.28	28.668	**0.141**
		100	1.60	104.858	0.931
175	fast	-40	1.56	180.540	1.44
		100	1.95	**256.501**	3.57
	typical	25	1.80	**178.329**	2.02
		100	1.80	181.039	2.12
	slow	-40	1.28	28.531	0.143
		100	1.60	104.858	0.950
Die area = 9222.82 μm^2 | **Core Area** = 6277.27 μm^2.					

Table B.2: Maximum Frequency and Power Estimation for different runs and PVTs of the Accelerator for even numbers count.

Freq (MHz)	Type	Temp (°C)	V (V)	Max Freq (MHz)	Power (mW)
100	fast	-40	1.56	213.472	0.757
		100	1.65	247.306	1.03
	typical	25	1.80	**217.367**	1.06
		100	1.80	220.660	1.12
	slow	-40	1.28	**41.983**	0.092
		100	1.60	139.290	0.543
150	fast	-40	1.56	249.186	0.751
		100	1.65	295.874	1.04
	typical	25	1.80	**254.016**	1.06
		100	1.80	259.036	1.12
	slow	-40	1.28	43.187	**0.080**
		100	1.60	153.480	0.509
200	fast	-40	1.56	268.590	0.760
		100	1.65	329.327	1.10
	typical	25	1.80	**280.668**	1.10
		100	1.80	288.705	1.17
	slow	-40	1.28	46.446	0.081
		100	1.60	173.605	0.540
240	fast	-40	1.56	281.270	0.755
		100	1.65	347.671	1.09
	typical	25	1.80	**293.883**	1.09
		100	1.80	302.707	1.16
	slow	-40	1.28	46.811	0.077
		100	1.60	178.816	0.528
280	fast	-40	1.56	316.599	1.21
		100	1.65	365.103	1.64
	typical	25	1.80	**317.366**	1.67
		100	1.80	315.077	**1.70**
	slow	-40	1.28	52.513	0.112
		100	1.60	172.010	0.719
320	fast	-40	1.56	322.837	1.19
		100	1.65	374.491	1.63
	typical	25	1.80	**323.635**	1.65
		100	1.80	323.635	1.69
	slow	-40	1.28	50.529	0.104
		100	1.60	155.853	0.605
Die area = 6006.232 μm^2 | **Core Area** = 3661.01 μm^2.					