Leo Moser, BSc

Greyhound: A RISC-V SoC with tightly coupled eFPGA on IHP SG13G2

MASTER'S THESIS
to achieve the university degree of
Diplom-Ingenieur

submitted to
Graz University of Technology

Supervisor
Dipl.-Ing. Dr. techn. Tobias Scheipel, BSc
Advisor
Dipl.-Ing. Meinhard Kissich, BSc

Institute of Technical Informatics
Embedded Architectures & Systems Group

Graz, July 2025

OK. 3 2 1 Let's Jam!

Acknowledgements

First and foremost, I want to thank my supervisors and advisors Tobias and Meinhard for showing as much enthusiasm for this project as I do, for encouraging me all the way, and for helping me get this across the finish line.

This thesis would not have been possible without the broader Free and Open Source Silicon Community, who believe in a future where open-source EDA tools and open-source PDKs are the norm, not the the exception.

Special thanks to Tim Edwards for his support during my time at Efabless and thereafter.

I would also like to thank IHP for providing and continuously improving the IHP Open Source PDK, and the German Federal Ministry of Research, Technology and Space for funding the public German project FMD-QNC (16ME083), without which the tapeout of Greyhound would not have been possible.

But most importantly, I thank my partner Katharina, who put up with *and* supported me during this intense time, even when the computer was running for yet another night to try different place & route solutions.

Thank you.

Graz, July 2025
Leo Moser

Abstract

Europe is facing a challenge that is expected to become even more pronounced over the next decade: a skills shortage in the very large scale integration domain. In order to attract more students to this domain, chip design education needs to become more accessible. Information should not be hidden behind non-disclosure agreements and expensive proprietary tooling.

The Free and Open Source Silicon (FOSSi) community has published proposals and roadmaps on how to achieve silicon democratization. Three key elements are needed: open-source Electronic Design Automation (EDA) tools, open-source manufacturable Process Design Kits (PDKs) and open-source Intellectual Property (IP) blocks. Already today, open-source EDA tools, PDKs, and IPs are used in education all over the world.

In this thesis, we present Greyhound, the first open-source RISC-V System on Chip (SoC) with tightly coupled embedded Field-Programmable Gate Array (eFPGA) in the IHP SG13G2 130 nm BiCMOS process. Greyhound utilizes the full FOSSi ecosystem: from open-source EDA tools, to an open-source PDK, to open-source IP blocks and frameworks. For the RISC-V core, the CV32E40X from the OpenHW Group is used. The eFPGA fabric is generated using FABulous, an embedded FPGA framework. It features 32×I/O, 784×LUT4+FF, 98×MUX, 7×SRAM (4KiB), 7×MAC (8bit·8bit+20bit), 14×Register file, 1×WARMBOOT, 1×CPU_IRQ and 4×CPU_IF. The bitstream for the fabric can be generated with a fully open-source toolchain: Yosys as synthesis tool and nextpnr for place & route. The eFPGA can be used either to implement a custom instruction extension for the CPU, a custom peripheral for the SoC, or as a standalone FPGA.

Under typical operating conditions, the maximum clock frequency of the SoC is 55 MHz. Greyhound's die is 3.6mm×5mm in size. To ensure a correct and manufacturable design, we simulate Greyhound, perform an abstract Layout Versus Schematic (LVS) check, and run the minimal Design Rule Check (DRC). Furthermore, we taped out Greyhound through the IHP-Open-DesignLib funded by a public German project FMD-QNC (16ME083) in April 2025.

Keywords: Open source hardware, Reconfigurable logic, Field programmable gate arrays, RISC-V, Application specific integrated circuits, Electronic design automation

Kurzfassung

Europa steht vor einer Herausforderung, die sich im kommenden Jahrzehnt noch verschärfen dürfte: ein Fachkräftemangel im Bereich Very Large Scale Integration. Um mehr Studenten für diesen Bereich zu gewinnen, muss die Chipdesign-Ausbildung zugänglicher gemacht werden. Informationen sollten nicht hinter Geheimhaltungsvereinbarungen und teuren proprietären Werkzeugen versteckt werden.

Die Free and Open Source Silicon (FOSSi) Gemeinschaft hat Vorschläge und Strategien veröffentlicht, wie die Demokratisierung des Siliziums erreicht werden kann. Es werden drei Schlüsselelemente benötigt: quelloffene EDA-Tools (Electronic Design Automation), quelloffene fertigungsfähige PDKs (Process Design Kits) und quelloffene IP-Blöcke (Intellectual Property). Bereits heute werden quelloffene EDA-Tools, PDKs und IPs in der Ausbildung auf der ganzen Welt eingesetzt.

In dieser Arbeit stellen wir Greyhound vor, das erste quelloffene RISC-V System on Chip (SoC) mit eng gekoppeltem embedded Field-Programmable Gate Array (eFPGA) im IHP SG13G2 130 nm BiCMOS Prozess. Greyhound nutzt das gesamte FOSSi-Ökosystem: von Open-Source EDA Tools über ein Open-Source PDK bis hin zu Open-Source IP Blöcken und Frameworks. Für den RISC-V Kern wird der CV32E40X von der OpenHW Group verwendet. Der eFPGA-Fabric wird mit FABulous, einem Embedded FPGA Framework, erzeugt. Er verfügt über 32×I/O, 784×LUT4+FF, 98×MUX, 7×SRAM (4KiB), 7×MAC (8bit·8bit+20bit), 14×Register file, 1×WARMBOOT, 1×CPU_IRQ und 4×CPU_IF. Der Bitstream für den Fabric kann mit einer vollständig quelloffenen Toolchain erzeugt werden: Yosys als Synthesewerkzeug und nextpnr für place & route. Das eFPGA kann entweder als benutzerdefinierte Befehlserweiterung für die CPU, als benutzerdefinierte Peripherie für das SoC oder als eigenständiges FPGA verwendet werden.

Unter typischen Einsatzbedingungen beträgt die maximale Taktfrequenz des SoCs 55 MHz. Der Chip von Greyhound ist 3,6 mm mal 5 mm groß. Um ein korrektes und herstellbares Design zu gewährleisten, haben wir Greyhound simuliert, eine abstrakte Layout Versus Schematic (LVS) Prüfung durchgeführt und den minimalen Design Rule Check (DRC) ausgeführt. Darüber hinaus haben wir Greyhound bei der IHP-Open-DesignLib, welche im Rahmen eines öffentlichen deutschen Projekts FMD-QNC (16ME083) finanziert wurde, für ein Tapeout im April 2025 eingereicht.

Contents

CHAPTER 1

Introduction

The **Free and Open Source Silicon (FOSSi)** movement has gained a large momentum over the last few years. This is thanks to openly accessible **Process Design Kits (PDKs)**, as well as several **free chip shuttles**[1] such as the Open MPW (Multi Project Wafer) shuttles and the IHP-Open-DesignLib. These shuttles have shown that Application-Specific Integrated Circuit (ASIC) design with open-source Electronic Design Automation (EDA) tools is not only possible theoretically, but is a path that leads to actual functional chips.

The FOSSi ecosystem enables the realization of projects and platform that would not have been possible otherwise: **Tiny Tapeout** [1], a shared silicon tapeout platform, further lowers the cost barrier to get to your own silicon while making it as easy as possible to tapeout your chip through web-based tools. This, in turn, introduced an even larger demographic to chip design that would otherwise never have come into contact with it, further driving its spread. This positive feedback loop of openness and innovation is what started in the software world decades ago and is now an integral part of it. We believe that this development is bound to happen in the Very Large Scale Integration (VLSI) domain as well.

1.1 Motivation

The FOSSi movement arrives at a time when access to silicon for everyone is more important than ever before: The **skill shortage** in the VLSI sector in Europe is at an all time high. While the need for workforce in the VLSI sector in Europe is increasing, the number of graduates from VLSI related field does not increase at the same rate [2]. The FOSSi community believes that this problem can be solved by **fostering the open-source silicon ecosystem** and by lowering the entrance barrier to newcomers.

Several letters and recommendations [3, 4, 5] have been published by the community, outlining how to achieve this. Chip design education needs to become more accessible in order to attract more students to this domain. Information should not be hidden behind Non-Disclosure Agreements (NDAs) and expensive proprietary tooling.

Three key elements are needed to achieve **silicon democratization**: open-source Electronic Design Automation (EDA) tools, open-source manufacturable Process Design Kits

[1] A "chip shuttle" in this context refers to a multi-project wafer service.

(PDKs) and open-source Intellectual Property (IP) blocks. Only when **all three** of these assets are available, a FOSSi ecosystem is possible. This openness also makes it possible to inspect the silicon in a way that would otherwise not be possible: from the Register Transfer Level (RTL) code down to the transistors. This is the place where innovation can take place: in a completely open environment that enables reproducible designs as it should be in academia.

1.2 Research Questions and Contributions

Out of the problems described above, we derive one Research Question (RQ) for this thesis.

Research Question RQ 1

How can a System on Chip with a tightly coupled embedded Field Programmable Gate Array be developed for educational purposes in mind?

We propose a platform that can be used in education to address the skills shortage in VLSI related fields, in order to deal with RQ 1. This platform can be applied to many educational use cases such as teaching how to program a RISC-V microcontroller, writing a custom instruction extension, implementing a custom peripheral, or programming the eFPGA as a standalone device.

However, for this platform to be fully effective in an educational context, it needs to be open: everyone needs to have access to the source code of the System on Chip (SoC) in order to reproduce the final layout, make changes to the RTL code, and learn from the design. This implies that the chip must be implemented with open-source EDA tools and an open-source PDK. Yet, it does not stop here: while the RISC-V ecosystem is already very much open, FPGA ecosystems often are not. Therefore, the chip needs to integrate an eFPGA that can be used together with open source tools in order to synthesize and implement user designs.

The main contribution toward answering RQ 1 presented in this thesis is **Greyhound** [6], the first open-source RISC-V SoC with tightly coupled eFPGA in the IHP SG13G2 130 nm BiCMOS process. Greyhound utilizes the full FOSSi ecosystem: from open-source EDA tools, to an open-source PDK, to open-source IP blocks and frameworks.

The **CV32E40X** [7] from the OpenHW Group is selected as Greyhound's RISC-V core, enabling a custom instruction interface. The SoC has a basic set of peripherals such as a Quad Serial Peripheral Interface (QSPI) Flash controller with builtin cache, a QSPI Pseudostatic RAM (PSRAM) controller and a Universal Asynchronous Receiver Transmitter (UART). The functionality of the SoC can be extended using the on-chip eFPGA which can be used to implement a custom instruction or a custom peripheral. The eFPGA of Greyhound is generated by FABulous [8], an embedded FPGA framework. The eFPGA features 32×I/O, 784×LUT4+FF, 98×MUX, 7×SRAM (4KiB), 7×MAC (8bit·8bit+20bit), 14×Register file, 1×WARMBOOT, 1×CPU_IRQ and 4×CPU_IF.

The CPU interface enables the eFPGA to implement a custom instruction for the RISC-V core, or a custom peripheral for the SoC. The eFPGA can trigger up to 4 Interrupt Requests (IRQs) of the CPU. There are many ways to upload a bitstream: The eFPGA can act as Serial

Peripheral Interface (SPI) controller or SPI peripheral, the CPU can stream the bitstream to a memory mapped address or simply trigger a reconfiguration from one of the 16 slots. Each slot in the SPI Flash is `0x4000` bytes in size. The eFPGA may also be used standalone from the SoC.

By choosing FABulous, the bitstream for the eFPGA can be generated with open-source tools: **Yosys** for synthesis and **nextpnr** for Place & Route (PnR) [9]. This makes Greyhound not only a great reference design for FOSSi chip design, but also enables Greyhound itself to be used in an open educational context: students can learn to program the RISC-V CPU, write their own design for the eFPGA, develop custom instruction extensions and peripherals, analyze the PnR of the Field-Programmable Gate Array (FPGA) design, and dive into the Application-Specific Integrated Circuit (ASIC) design of Greyhound itself.

Greyhound is designed with **open-source EDA tools** and the **IHP Open Source PDK** [10]. The die is 3.6mm×5mm in size using the **IHP SG13G2** 130 nm BiCMOS process. Under typical operating conditions, the SoC reaches a maximum clock frequency of 55 MHz. Greyhound is **taped out** through the **IHP-Open-DesignLib** [11] funded by a public German project FMD-QNC (16ME083) [12] in April 2025.

We summarize our contributions as follows:

- We present Greyhound, an open-source, education-focused, reprogrammable and extensible, RISC-V SoC with tightly coupled eFPGA.
- We perform the physical implementation of Greyhound using open-source EDA tools and the IHP Open Source PDK.
- We tape out Greyhound in IHP SG13G2 130 nm BiCMOS process in April of 2025.

1.3 Thesis Structure and Organisation

The remaining thesis is structured as follows: Chapter 2 explains important concepts and provides information about the tools and frameworks used for Greyhound. Chapter 3 compares Greyhound to existing work and compares how they differ. Chapter 4 covers the design of the SoC, lists the IP blocks used and concludes with the verification of the SoC. Chapter 5 goes into detail how the eFPGA is generated using FABulous, which additional tiles are developed for Greyhound, how the physical implementation of the eFPGA using LibreLane works, and how the eFPGA is verified in simulation. The final chip design is presented in Chapter 6, which explains and evaluates the challenges that need to be overcome to arrive at the chip layout, ensure it is correct and manufacturable, and submit the design. Finally, the work is summarized in Chapter 7 and future work is proposed.

CHAPTER 2

Background and Terminology

This chapter gives an overview of the technologies and frameworks used for the work in this thesis. First, RISC-V, an open-source Instruction Set Architecture (ISA) is introduced together with the RISC-V core that is chosen for Greyhound. FABulous, an embedded FPGA framework is introduced and the term PDK is explained with a concrete example of the ihp-sg13g2 PDK. We explain what the purpose of a digital design flow is and which open-source EDA tools were used in this thesis. Finally, cocotb, the testbench environment for this work, is introduced.

2.1 RISC-V

RISC-V [13] is an open-source standard ISA originally developed at the University of California in Berkeley. RISC-V has a selection of base ISAs and a range of optional extensions, therefore it can be easily scaled to cover different use cases. Greyhound uses a RISC-V core as the CPU of the chip, namely the CV32E40X.

The RISC-V standard approves the usage of custom instruction extensions by reserving an opcode range just for this purpose. In this work, we make use of one of these reserved opcodes to enable the implementation of custom instruction via Greyhound's eFPGA.

2.2 CV32E40X

The CV32E40X from the OpenHW Group [7] serves as the CPU for the SoC of Greyhound. Its history started as a fork of the OpenHW CV32E40P [14] which in turn evolved from the RI5CY core of the PULP platform [15]. CV32E40X is a small and efficient 32-bit RISC-V core with a 4-stage pipeline. It has support for a range of instruction set extensions. In the configuration for this work, the core implements the `RV32I` base instruction set with the `M`, `A`, `C`, `Zicntr`, `Zihpm`, `Zicsr`, `Zifencei`, `Zba_Zbb_Zbc_Zbs` and the `Zca_Zcb_Zcmp_Zcmt` instruction extensions.

In addition, the CV32E40X implements the `Xif` (eXtension Interface) as a custom instruction set extension, which we enabled. By default this interface does not implement new instructions, but provides the means to extend the CV32E40X with custom or standardized instructions, without the need to change the RTL of the core. It allows low latency read and write access to the register file, custom Control and Status Registers (CSRs), and read and write

access to the memory bus. All opcodes that are not used by the CV32E40X are offloaded to the `Xif`, enabling to implement both custom instruction extensions and standardized instructions, i.e., `F` (single-precision floating-point), `P` (Packed SIMD) or `V` (Vector) extensions.

The main reason why this core is chosen for this work is its `Xif` extension: it enable us to connect the eFPGA fabric to the CV32E40X without modifying the RTL code of the core. For Greyhound we only make use of the read and write access to the register file for custom Arithmetic Logic Units (ALUs) type instructions.

2.3 FABulous

FABulous [16] is an embedded FPGA framework used to generate custom FPGA fabrics. It comes with a basic tile library consisting of a Configurable Logic Block (CLB), a dedicated register file, a Multiply-Accumulate (MAC) unit, I/O blocks and more. Designers can create their own custom Basic Elements (BELs) that can be included inside custom tiles of the fabric.

FABulous provides a full ecosystem with integration into the Yosys and nextpnr EDA tools. After generation of the fabric, FABulous creates a nextpnr model, as in a description of the Programmable Interconnect Points (PIPs) and BELs. This model includes the BELs and PIPs of the FPGA, ready for nextpnr to perform Place & Route (PnR). FABulous also provides mappings to the primitives of the FPGA for Yosys in order perform synthesis.

The currently supported CLB architecture is similar to the one of the iCE40 [17] series from Lattice. The reason for that is their carry-chains are already supported by Yosys [18]. But there is a significant difference: each Logic Cell (LC) also contains a multiplexer that is fed by the outputs of the LCs and can be configured as 1×MUX8, 2×MUX4 or 4×MUX2 as shown in Figure 5.3 in Chapter 5.

FABulous supports a frame-based configuration of the FPGA fabric. Instead of a large shift register, the configuration bits are addressed using `FrameData` and `FrameStrobe` signals. This is similar to how a memory is implemented: `FrameData` acts like the bit lines in a memory and `FrameStrobe` like the word lines respectively. Using this configuration scheme instead of a shift register-based implementation allows partial reconfiguration of a single frame. A frame spans the whole column and tiles can contain up to 20 frames in the current implementation.

2.4 Process Design Kit

In order to design an Integrated Circuit (IC), a PDK is needed for the targeted process. There are open-source manufacturable PDKs [19] [20] [10] and non-manufacturable PDKs [21] [22], which are used for research or teaching. Most PDKs are tailored to a specific process in order to get the most out of the process by providing rules that are tight but still reliable. Some PDKs trade this last bit of performance for some laxer rules which allow the PDK to be mapped to several processes. One famous example is the SCMOS rules from MOSIS [23], which are lambda (λ) based. The minimum gate length of a process is said to be 2 times λ. From there on all geometry is specified using multiples of λ, which enables designs to be portable between different processes.

PDKs can be considered to consist of several levels: At the lowest level there is information about the process, design rules, and models for primitive devices such as NMOS and PMOS transistors. At higher levels are Parametrizable Cells (PCells), standard cell libraries and complete IPs such as Static Random-Access Memory (SRAM). The quality of a PDK determines reliability, area cost, power consumption and performance of the manufactured IC.

In this work, the open-source ihp-sg13g2 PDK is selected, which targets the SG13G2 130nm BiCMOS process from IHP. This PDK is under active development at the time of writing and is, therefore, currently in preview status. The PDK is primarily designed for use with open-source EDA tools. By selecting this PDK, there are no restrictions on sharing Greyhound's design files, including the final layout. It also means that it is possible for anyone to reproduce this work. While LibreLane, the digital design flow infrastructure library, strives for reproducibility, it has not yet been tested whether the final output of Greyhound is a binary match between different runs and different systems.

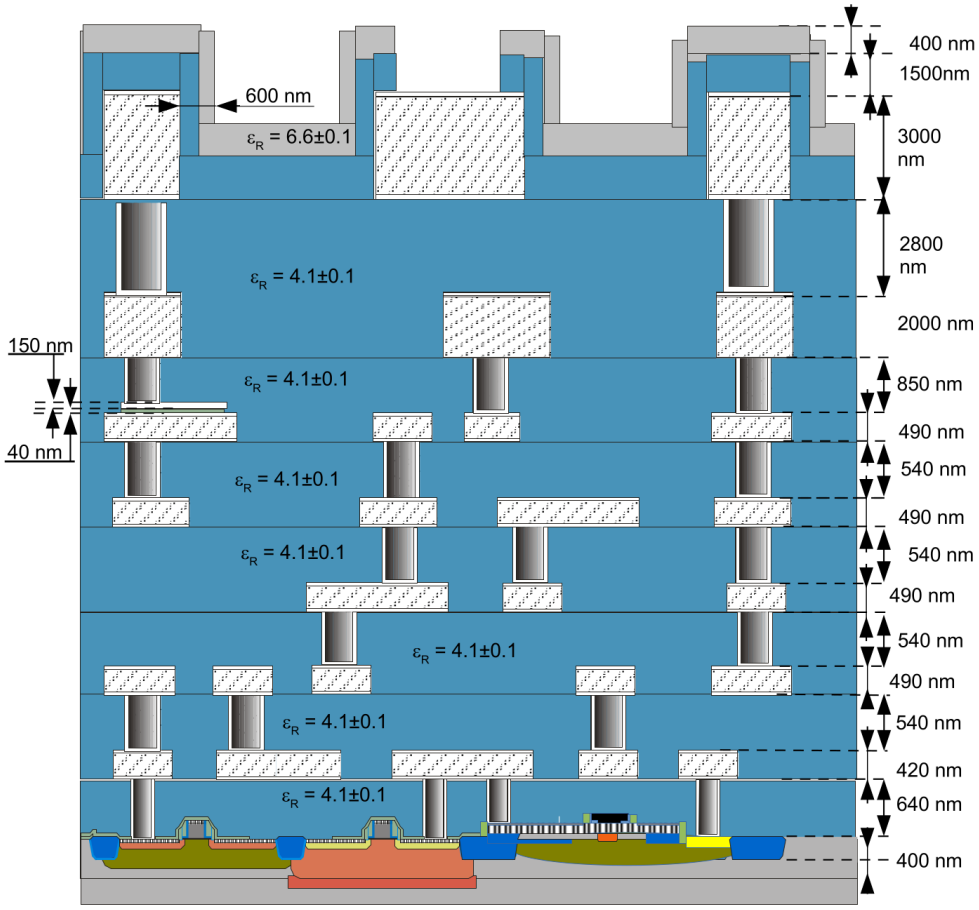


Figure 2.1: SG13G2 process cross-section schematic [24].

In terms of capabilites, the the SG13G2 process [24] can be compared to the SKY130 process [25]. There exists an open-source PDK for both, and they are very similar in terms of feature size and metal stack-up.

Compared to SKY130 where the channel length is 150nm, SG13G2 is a "real" 130nm process, with a 130nm channel length. SKY130 has 5 metal layers and one local interconnect layer. SG13G2 has 5 metal layers and two top-metal layers. These two top-metal layers, `TopMetal2` and `TopMetal1` are much thicker compared to the other metal layers as seen in Figure 2.1. They are mainly intended for the Power Distribution Network (PDN) in order to supply power to the macros and thus the transistors with a low voltage drop.

Both processes use aluminum metal layers, only the local interconnect layer of SKY130 is made of Titanium Nitride (TiN), which has a higher resistivity than aluminum. This layer should therefore only be used for routing short distances, e.g., in digital standard cells.

SG13G2 has a lower logic density due to coarser interconnects (larger minimum width and spacing) and less optimized standard cells in the open-source PDK (as of now). Nevertheless, SG13G2 is capable enough for digital designs, like Basilisk [26], an open-source Linux capable SoC on SG13G2, has demonstrated.

2.5 Digital Design Flow

A digital design flow, also called ASIC design flow, performs the physical implementation of a digital design. Usually, a PDK comes with a Standard Cell Library (SCL) or sometimes even with several different ones that are optimized, e.g., for area or low latency. These SCLs contain the layout and abstract views of characterized standard cells. The cells implement digital functions such as AND, OR, XOR or even more complex logic function such as AND-OR-Invert (AOI) or OR-AND-Invert (OAI). AOI cells are logic cells constructed of one or more AND gates connected to a NOR gate. OAI is the inverse, one or more OR gates connected to a NAND gate.

An SCL also contains storage elements such as flip-flops and latches. The more standard cells a library consists of, the better the synthesis tool can map the Hardware Description Language (HDL) code to the available gates.

The goal of the digital design flow is to get from the HDL code to the final layout of the circuit while ensuring that the generated layout equals the original circuit and does not violate any design rules. This is achieved by applying methods such as Layout Versus Schematic (LVS) and Design Rule Check (DRC) before sending the files to be manufactured.

OpenROAD

The OpenROAD [27] project was launched in 2018. It's goal is to bring down the barriers of cost, expertise and unpredictability that designers are currently challenged by. OpenROAD implements an important part of the digital design flow, namely PnR. The inputs to OpenROAD are the Verilog netlist and associated libraries and constraints. The aim is to arrive at the final layout while optimizing area, power and speed. This is usually done in several steps, each of which fulfills a specific task. According to OpenROADs documentation [28] these steps include:

- Floorplanning
- Placement
- Clock Tree Synthesis
- Routing
- Finishing

However, this comprises just a high-level overview. Usually, these high-level steps are split into smaller steps, such as global and detailed placement or global and detailed routing.

OpenROAD by itself is not a complete digital design flow. First, the RTL code of the design needs to be synthesized, and finally the layout needs to be prepared for manufacturing (finishing). Finishing includes fill insertion, in which dummy geometry is inserted to ensure a consistent density across layers. Fill insertion is done in a separate step outside of OpenROAD for Greyhound.

Therefore, there are several digital design flows that integrate OpenROAD. One of them is OpenROAD-flow-scripts [29] which is the official flow from OpenROAD. Another one is LibreLane [30] (formerly OpenLane 2) which is used for the work in this thesis.

LibreLane

LibreLane [30] is an infrastructure library that allows for construction of digital ASIC design flows. It enables translation of RTL code all the way to Graphic Design System II (GDSII). LibreLane is a fork of OpenLane 2 which in turn evolved from OpenLane (1) [31]. It is a community-driven project.

The default `Classic` flow, for example, is a digital design flow based on open-source tools such as Yosys, OpenROAD, magic [32], KLayout [33] and netgen [34] to translate RTL code all the way to GDSII and perform DRC and LVS checks.

LibreLane is described as an infrastructure library, meaning the focus is on integrating other tools as steps into the flow. Its architecture is well thought: LibreLane offers so called **Flows**, such as the `"Classic"` flow which includes OpenROAD, to perform the ASIC implementation steps. These flows consist of **Steps** which receive a configuration and a state, operate on these states and finally return them for the next step. The states are immutable, i.e., you can only every create a copy of a state before modifying it. This ensures that LibreLane is reproducible and prevents unexpected side effects when rerunning whole flows or just parts of a flow.

The flows and steps can be configured using a variety of configuration formats such as JSON, TCL or YAML. It is even possible to modify the flow just using the configuration by inserting, deleting or replacing steps. LibreLane also offers a Python API to programmatically control flows, define new flows and create new steps. Another feature that we use in this work is to create a plugin that adds new flows and steps to LibreLane.

2.6 cocotb

cocotb [35] is an open-source coroutine-based co-simulation testbench environment.

Since cocotb is mainly written in Python, the testbenches are also written in Python, providing a large ecosystem of Python modules and example code. cocotb is simulator-agnostic, meaning that the underlying simulator can easily be exchanged. It is possible to choose from a wide range of simulators: open-source simulators such as Icarus Verilog [36], Verilator [37], and all common proprietary simulators. This is possible thanks to cocotb's Generic Procedural Interface (GPI), which is cocotb's abstraction of Verilog Procedural Interface (VPI), VHDL Procedural Interface (VHPI), and Foreign Language Interface (FLI). All of these are interfaces to the simulators which allow to query signals, set values, set up callbacks etc.

In this work, we make use of cocotb for all of our testbenches: verifying the functionality of the SoC and the eFPGA as well as for the chip top-level simulation.

CHAPTER 3

Related Work

This chapter discusses contributions related to the work in this thesis. It is split into three sections, one for systems with reconfigurable hardware, another for SoCs that use the IHP Open Source PDK with open-source EDA tools, and the last section for custom instruction extensions. Furthermore, we compare these contributions to Greyhound in each section and explain how our work in this thesis differs from those contributions.

3.1 Systems with Reconfigurable Hardware

One common characteristic of embedded systems is that the hardware in the field cannot simply be replaced or updated. Their whole lifetime, the hardware stays the same and outdated cryptographic implementations or hardware bugs could lead to exploitation of the system.

One solution to this problem is presented in moreMCU [38]: A Runtime-reconfigurable RISC-V Platform for Sustainable Embedded Systems. The FPGA-based implementation of the design allows fine-grained partial reconfiguration. This makes it possible to swap out custom instruction implementations and dynamic peripherals during operation. Scheipel et al. further present the concept of an Operating System (OS) in moreMCU, which enables the emulation of functionality that is not yet available in hardware, e.g., during reconfiguration. Of course, reconfigurablity also has its drawbacks, such as larger area requirements and power usage as well as a lower maximum frequency.

This work is similar to Greyhound, although largely different in one key aspect: Greyhound implements its SoC and eFPGA in an ASIC and does not use an off-the-shelf commercial FPGA. Greyhound also allows reconfiguring the eFPGA to provide custom instruction extensions or custom peripherals, although not both at the same time. The interface of the eFPGA can only ever be connected to the `Xif` of the CV32E40X to implement a custom instruction extension, or to the OpenBus Interface (OBI) of the SoC to implement a custom peripheral. Which of the two options is active can be selected using the fabric configuration peripheral. This decision was made to keep the interface between eFPGA and SoC/RISC-V core slim.

While, for small quantities, an FPGA-only solution might be cost effective, embedded systems are often deployed in large quantities were an ASIC implementation makes financially

sense. For that reason Greyhound implements the static parts of the system, i.e., the SoC with RISC-V core as an ASIC. The eFPGA is naturally also part of the ASIC, but can be reconfigured. Implementing the static parts of the system in an ASIC also has other advantages such as higher performance and less energy consumption compared to an FPGA implementation on the same process node.

A more comparable design is FlexBex from Nguyen et al [18]. Their work presents an eFPGA directly coupled to an Ibex RISC-V core taped out in a 180nm TSMC process. Similar to Greyhound, the eFPGA is also generated using the FABulous framework. The eFPGA contains three slots where each slot can hold one custom instruction implementation. These slots do not refer to temporal slots like it is the case on Greyhound where several bitstreams can be enabled one after another, but instead to spatial slots in the fabric.

FlexBex allows the eFPGA to be configured either serial external or via streaming the bitstream directly from the RISC-V core using a custom instruction. Greyhound is more flexible in the configuration of its eFPGA: it has an SPI controller or peripheral interface and allows the CPU to trigger reconfiguration or stream bitstream data via a memory-mapped peripheral. Greyhound does not have distinct spatial slots in its eFPGA. This could be emulated in a future revision of Greyhound by passing parts or the whole opcode to the eFPGA to distinguish different custom instructions. Greyhound also supports partial reconfiguration triggered from the RISC-V core and, compared to FlexBex, it allows for true parallel configuration without congesting the system bus by utilizing the separate SPI controller.

FlexBex does not allow custom peripherals in the eFPGA to be connected to the system bus of the SoC, unlike Greyhound. It is however possible to implement custom peripherals using custom instructions similar to how AVR microcontrollers access their peripherals. This is possible with both Greyhound and FlexBex.

While FlexBex allows to delay instructions by 0 to 15 cycles, Greyhound uses the `funct7` field in the R-type instruction format, with a total of 0 to 127 delay cycles as shown in Table 3.1.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode

Table 3.1: RISC-V R-type instruction format.

Another major distinction is that Greyhound is developed using open-source EDA tools and the IHP Open Source PDK, whereas the physical implementation of FlexBex was performed using proprietary tools and PDK. The size of the actual chip is also different: 2mm×4.5mm (FlexBex) compared to 3.6mm×5mm (Greyhound). The eFPGA of FlexBex is also smaller in terms of resources, only 348×LUT4+FF compared to Greyhounds 784×LUT4+FF.

There have been several Open MPW submissions with FABulous FPGAs using sky130: FABulous FPGA [39] on MPW2, FuseRISC2 [40] on MPW3, and the fully open FABulous eFPGA [41] on MPW5.

The first two of these submissions were implemented using proprietary EDA tools and are therefore not compared here. The fully open FABulous eFPGA [41] on MPW5 has been generated with open-source EDA tools. Its fabric has slightly less resources than Greyhounds eFPGA: 576×LUT4+FF compared to 784×LUT4+FF. It also contains DSP blocks, register files, and 6×1KiB BRAMs. Greyhound only contains single-port SRAMs, although with more capacity at 7×4KiB. The main difference between the fully open FABulous eFPGA and Greyhound is that the former is not a full-chip design. For the Open MPW submissions, only the user project of the Caravel harness was implemented. The Caravel harness itself with the RISC-V SoC is a fixed system and reused for all Open Multi-Project Wafer (MPW) submissions. This also means that the eFPGA could not be integrated as tightly as in Greyhound, since the Caravel harness does not provide a custom instruction interface. Greyhound in contrast is a full-chip implementation using ihp-sg13g2. The complete chip including the padring was generated for this design.

3.2 System on Chip on SG13G2

Greyhound is not the first chip with SoC designed using the IHP Open Source PDK. Basilisk [26] in 2024 and MLEM [42] in 2025 are two examples of open-source SoCs designed with open-source EDA tools and an open-source PDK.

Basilisk is a fully Linux-capable 64-bit RISC-V SoC with a die are of 34 mm². It includes a wide range of peripherals such as HyperRAM, USB 1.1, and VGA. The SoC of Greyhound cannot be directly compared since it is intended for micro-controller applications, whereas Basilisk can run a general-purpose OS such as Linux. One difference in how those two chips are designed is the choice of the EDA tooling. Basilisk uses OpenROAD [27] directly and Greyhound uses LibreLane which builds on top of OpenROAD. Another difference is that Basilisk does not contain any reconfigurable logic such as an eFPGA.

MLEM [42] is an implementation of the Croc [43] SoC platform. It is an educational platform with a dedicated user domain similar to the Caravel harness of the Open MPW runs. Greyhound is also an educational platform in the sense that the chip could be used for teaching: writing FPGA designs with Verilog, exploring how PnR on an FPGA works, microcontroller programming, extending the functionality of the SoC with custom designs, and more.

Greyhound's focus is therefore about what someone can do with the actual chip, rather than being a platform for chip design, as is the case with MLEM. Though the chip design of Greyhound itself can also be studied by students, as it is open source.

3.3 Custom Instruction Extensions

There are a number of efforts to provide a common custom instruction interface with various functionality. One of them is SCAIE-V [44], it offers a lot of flexibility and features: from custom multi-cycle instructions to control flow customization.

SCAIE-V supports a number of operations such as reads from the register file, writes to the register file, access to the program counter and more. The custom instruction can then be integrated into the selected core using a SCAIE-V configuration: the configuration specifies at which cycle counts the register values need to be provided and when the output value is ready. SCAIE-V is compatible with several cores such as the Piccolo, VexRiscv, ORCA and the PicoRV.

For Greyhound, we settled on the CV32E40X RISC-V core from the OpenHW Group with support for the CORE-V-XIF [45] custom instruction interface as explained in Chapter 2.2 and 4.1. This Custom Instruction Interface (XIF) interface has support for an additional feature compared to SCAIE-V, namely custom CSRs. However, in our implementation, we limit custom instructions to multi-cycle R-type [13] with two source registers (`rs1` and `rs2`) and one destination register (`rd`) in order to limit the number of signals between the eFPGA fabric and the SoC.

This is similar to the constraints in FRANCIS-V [46], an integration framework for generating single-cycle R-type custom instruction extensions with two source registers (`rs1` and `rs2`) and one destination register (`rd`). This suggests that it may be possible to generate custom instruction for Greyhound using FRANCIS-V, although this is not investigated further in this thesis.

CHAPTER 4

Design of the RISC-V SoC

This chapter focuses on the SoC, which IPs were selected and how they were integrated into the system as a whole. It covers the configuration of the eFPGA fabric via a fabric configuration peripheral and explains how custom instruction extensions or custom peripherals can be implemented. Finally, the compilation of firmware is covered and how the verifiifaction environment for the SoC is created using cocotb.

They key features of the SoC are its CV32E40X RISC-V core, 8 KiB of builtin SRAM, a QSPI eXecute in Place (XiP) Flash controller with direct mapped cache, a QSPI PSRAM controller, and a UART. Two additional peripherals are used to configure the eFPGA fabric and to access the custom peripheral of the eFPGA.

The rationale for this feature selection is to provide a platform with a small area footprint in the chip that still is powerful enough for most microcontroller applications. One use case for the RISC-V SoC is the processing of sensor data, where compute-intensive operations can be offloaded to a custom instruction extension. Another use case would be to extend the platform with custom peripherals implemented as a user design in the eFPGA fabric: A certain application might require multiple SPI controllers or perhaps a specialized peripheral. Any of these can be implemented in the eFPGA as a custom peripheral as long as the resources of it are not exceeded.

4.1 Choosing the RISC-V CPU

The CPU core of Greyhound is selected to be a RISC-V core due to the open nature and extensibility of the RISC-V ISA. There exists a wide range of open source RISC-V cores with various features and capabilities to choose from. However, the main selection feature for Greyhound is that the RISC-V core should offer a custom instruction interface.

In theory, any core with the source code available can be extended with custom instructions. However, in reality this involves a full understanding of the implementation, e.g., the pipeline, editing of the source files to integrate the custom instruction, and verifying that no other functionality has been impacted. This is no trivial matter, as this procedure has to be repeated every time the core is updated and has changed significantly.

In contrast, a custom instruction interface can be used to extend the core with a custom

instruction without modifying its source code. For this reason we choose the CV32E40X RISC-V core. One of its distinct features is the `Xif` custom instruction set extensions, which allows to extend the CV32E40X with custom or standardized instructions. In the case of Greyhound the custom instruction will be implemented by the eFPGA fabric and is therefore reconfigurable.

There are also further considerations to be taken, for example, which standard extensions are enabled by default and which ones should be enabled in addition. For the CV32E40X core, the following standard instruction set extensions are always enabled:

- `C`: Standard Extension for Compressed Instructions
- `Zicntr`: Standard Extension for Base Counters and Timers
- `Zihpm`: Standard Extension for Hardware Performance Counters
- `Zicsr`: Control and Status Register Instructions
- `Zifencei`: Instruction-Fetch Fence
- `Zca_Zcb_Zcmp_Zcmt`: Code-Size Reduction Extensions

In Addition, we chose to enable the following optional standard instruction set extensions for the CV32E40X core:

- `M`: Standard Extension for Integer Multiplication and Division
- `A`: Atomic Instructions
- `Zba_Zbb_Zbc_Zbs`: Bit Manipulation Extensions

`RV32IMAC` gives us the basic 32-bit integer RISC-V ISA with the standard extension for integer multiplication and division as well as support for atomic instructions and compressed instructions. The remaining extensions provide further performance enhancements and other features. The optional standard instruction set extensions were enabled because the physical implementation still fit in the available area and a more powerful core enables more and more diverse applications.

As the instructions and data are stored in an external non-volatile SPI Flash, the interface to it can likely turn out to be the main bottleneck. Some measures have been taken to alleviate this bottleneck to the SPI flash, such as supporting a QSPI interface which can transmit 4 bits in one cycle and by using a cache connected to the SPI flash (Section 4.2). Support for the standard extension for compressed instructions (`C`) also helps in mitigating this bottleneck when fetching instructions from the QSPI Flash, as two compressed instructions fit into 32 bit.

4.2 Choosing Third-Party IPs

The CV32E40X on its own does not make a SoC. The core needs to be able to fetch instructions from an external memory, and in order to execute more complex programs, it needs an external RAM to complement the internal SRAM. This would be sufficient in order for the core to perform computations, but in this work we also add a UART for simple I/O functionality.

QSPI Flash Controller

Since there is no on-chip non-volatile memory available, the core needs to fetch the instructions from an external non-volatile memory. For Greyhound this external memory is a QSPI Flash as explained in Section 4.1. For the controller we decide on the `EF_QSPI_XIP_CTRL` [47] from Efabless. It is a QSPI XiP Flash controller with direct mapped cache.

While an SPI Flash only allows for serial data access, a QSPI Flash allows to send 4 bits at once. Support for XiP enables the flash to automatically increment the address for further reads, without the need to send the whole address again. This is useful if several words of data are read consecutively, as statistically the subsequent data may be needed soon.

Accessing the Flash has a high latency. For that reason the QSPI Flash controller integrates a direct mapped cache, which makes it practical to read multiple words using XiP. With each access to the Flash, one line of the cache is filled. Greyhound configures the cache with 8 lines of 32 bytes.

With this configuration, the CPU does not need to access the SPI Flash during tight loops. This has shown to speed up `crt0` initialization considerably, which has a positive impact on simulation runtime and on the startup of the chip. It has also shown to be helpful in other computational tasks that make use of tight loops or to reduce the latency when waiting in a loop.

QSPI PSRAM Controller

Greyhound includes 8 KiB of integrated SRAM for fast memory access. This should be sufficient for most microcontroller applications. In addition, the eFPGA can implement a custom peripheral that makes the 7×SRAM (4KiB) of the fabric available to the SoC, for a total of 36 KiB of SRAM. More advanced applications, however, may require access to more RAM, even at the cost of higher latency. Thus, a QSPI PSRAM controller was integrated into the SoC.

Viewed from the outside, the PSRAM appears as slower SRAM connected via a QSPI interface. In reality, PSRAM contains no SRAM at all, but only acts as one. Internally PSRAMs are actually Dynamic RAMs (DRAMs) with a self-refresh circuitry. Therefore, PSRAM is a cheap method to add external RAM without the need for a DRAM interface and periodic refresh cycles.

The `EF_PSRAM_CTRL` [48] third-party IP from Efabless is selected as QSPI PSRAM controller. This peripheral has no cache at all. It is recommended to keep data that is accessed often in the internal 8 KiB SRAM. This must be done by the linker at build-time.

UART

For communication and debugging purposes, we have integrated a UART in the SoC of Greyhound. It would be possible to implement a UART in the eFPGA as a custom peripheral, however, providing a dedicated peripheral will free up more resources for the user. We choose `EF_UART` [49], again from Efabless' open-source IP library

This IP core is a highly configurable UART with variable baudrate and frame format, line-break detection, configurable receiver timeout, loopback capability, glitch filter and a variety of interrupts sources.

Of course, it is possible to implement a UART in the eFPGA and make it available as a peripheral via the memory-mapped interface. However, there are other reasons to include a standalone UART in the SoC: for debugging in case the eFPGA does not work as expected, or simply to reserve the eFPGA resources for other use cases.

OpenBus Interface IP

To build the complete SoC some additional IP is necessary. Since the CV32E40X core uses the OBI [50] standard for its address and data bus, the SoC also needs additional OBI compatible IP.

The PULP platform has a collection of OBI compatible IP [51] ready of which the following is used in this work:

- `obi_mux`: A multiplexer for the OBI protocol.
- `obi_demux`: A demultiplexer for the OBI protocol.
- `obi_err_sbr`: A error subordinate, responding with the error bit set.
- `obi_sram_shim`: An adapter for a standard SRAM.

Out of simplicity and to save on resources we decided to use a multiplexer to share a bus for data and instructions as seen in Figure 4.1. The IP collection also contains a crossbar (`obi_xbar`) that could be used to improve the performance of the SoC if enough area is available.

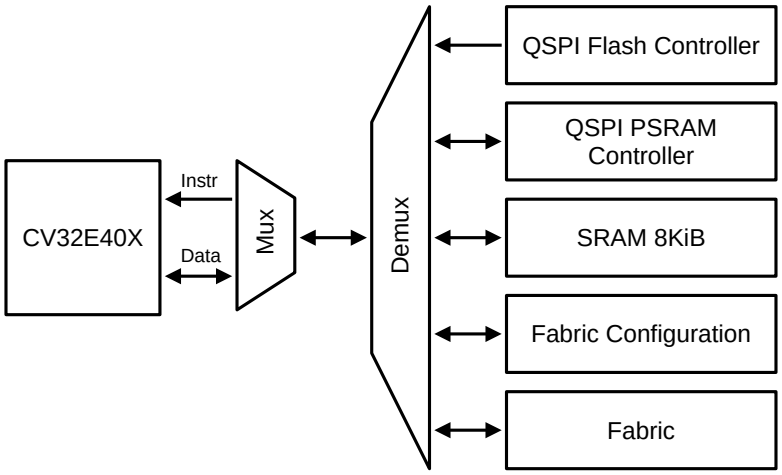


Figure 4.1: Block diagram of Greyhounds SoC.

Additionally, a bridge from OBI to Advanced High-performance Bus Lite (AHBL) is needed. Such a bridge was developed for a project from the OpenHW Group, the `obi2ahbm_adapter` [52]. This IP is used to connect the IPs from Efabless such as the QSPI Flash and PSRAM controller as well as the UART to the SoC.

4.3 CPU Core Integration

In order to build the complete SoC, the CPU core needs to be configured correctly and integrated together with the other IP.

The first issue that arises is that Yosys only has support for certain SystemVerilog features and not the entire feature set. This is sufficient for most of the IP available as open source, but not for all, as is the case with the CV32E40X core.

One solution is to pay for the Tabby CAD Suite [53] offered by YosysHQ, which adds a proprietary front-end to Yosys in order to support SystemVerilog according to the latest specification. Since we want Greyhound to be completely open source (this includes the build process) we resort to a tool called sv2v [54] to convert the SystemVerilog code of the CV32E40X core to Verilog. The Verilog source code can further be synthesized and simulated.

A different tool that could be considered for SystemVerilog support is yosys-slang [55], which integrates Yosys with the slang SystemVerilog parser. It has also been used for tapeouts such as for MLEM.

The CV32E40X core includes a Physical Memory Attribution (PMA) unit. The PMA unit is used to add the following attributions to the memory map:

1. `main` - Code execution is allowed, memory is considered idempotent.
2. `bufferable` - Writes will utilize the write buffer.
3. `cacheable` - Memory region is cached.
4. `atomic` - Region supports atomic operations.

Therefore the PMA had to be configured to match the memory map in Table 4.1.

Another consideration to make during core integration is the clock gating of the CV32E40X core. During sleep, the clock to the CPU is halted using a clock gate to prevent this part of the clock tree to charge and discharge, which reduces dynamic power consumption. Since clock gates are technology-specific and usually not automatically inferred, the CV32E40X has a dedicated `cv32e40x_clock_gate` module. For PnR the generic clock gate is replaced with the `sg13g2_lgcp_1` standard cell from the ihp-sg13g2 PDK.

4.4 Memory Map

Table 4.1 shows an overview of all available peripheral devices.

Since the core starts executing from `0x0000_0080`, the QSPI Flash controller is mapped to offset `0x00000000`. The interrupt vector table is located at `0x00000000` to `0x0000_007F`.

The QSPI Flash controller is followed by the builtin SRAM and the QSPI PSRAM controller at `0x10000000` and `0x20000000`, respectively. At `0x30000000` the UART is located. The fabric configuration and fabric peripheral are used to configure and communicate with the fabric. More information about these peripherals is given in Chapter 5.

Address	Name	Description
`0x00000000`	`FLASH_BASE`	QSPI XIP Flash
`0x10000000`	`SRAM_BASE`	8kB of SRAM
`0x20000000`	`PSRAM_BASE`	QSPI PSRAM
`0x30000000`	`UART0_BASE`	Highly configurable UART
`0x40000000`	`FABRIC_CONFIG_BASE`	Fabric configuration peripheral
`0x50000000`	`FABRIC_BASE`	Interface to the fabric

Table 4.1: Memory map of the SoC.

Since the CV32E40X as a RISC-V core has a 32-bit address space, we are relatively unconstrained in mapping the peripherals to the memory map. The QSPI Flash protocol allows the transfer of 24-bit addresses. Hence the lower 24-bit of address room are reserved for the QSPI Flash peripheral. To reduce complexity, we reserve a 24-bit address room for all peripherals, incrementing only the uppermost nibble of the address for each peripheral. For a peripheral that uses fewer address bits, e.g., the built-in SRAM, this means that it is repeated several times in the 24-bit address room.

4.5 Fabric Config Peripheral

The RISC-V core in Greyhound needs a way to configure the eFPGA fabric. This could be done with one or several custom instructions, however, we decide to implement a peripheral that manages the eFPGA. This has the benefit that it could be reused in other SoCs that make use of the OBI, a custom instruction extension, however, always depends on the interface of the core.

The fabric config peripheral is responsible for configuring the eFPGA fabric, the registers are shown in Table 4.2. The `REG_XIF_OR_PERIPH` at offset `0x0` is used to set the interface to the eFPGA fabric as either custom instruction or as peripheral. This must match the actual bitstream loaded in the fabric, otherwise the RISC-V core may get to a state where it executes a custom instruction through the interface, but the fabric contains the implementation of a peripheral or vice versa, leading to undefined behavior.

The current state of the fabric config peripheral can be read from offset `0x4` with the register `REG_FABRIC_CONFIG_BUSY`. If the peripheral is currently performing configuration, no further reconfiguration should be triggered via `REG_TRIGGER_SLOT` (`0xC`) nor data should be written to the `REG_BITSTREAM` (`0x8`) register. Once configuration has completed, the `REG_FABRIC_CONFIG_BUSY` register will return 0. This makes it possible for the RISC-V core to trigger a reconfiguration of the FPGA fabric and fetch or process data in the meantime.

Offset	Register	Description
0x0	REG_XIF_OR_PERIPH	[W] Write a 0 or 1. Sets the interface to the fabric as custom instruction interface (0) or as peripheral (1).
0x4	REG_FABRIC_CONFIG_BUSY	[R] Readonly. Returns 1 if the fabric is under configuration, 0 if not.
0x8	REG_BITSTREAM	[W] Write bitstream data to this register.
0xC	REG_TRIGGER_SLOT	[W] Trigger a reconfiguration by writing the slot to this register (0-15).

Table 4.2: Registers of Fabric Config Peripheral.

Another way of implementing the REG_XIF_OR_PERIPH functionality would be to add a configuration bit to the bitstream that determines whether the bitstream is intended for a custom instruction extension or for a custom peripheral. This approach was only discovered after the physical implementation of Greyhound was already well advanced.

A solution that would make both approaches obsolete is to add a custom instruction interface **and** a custom peripheral interface to the eFPGA instead of multiplexing them. However, this requires a larger eFPGA fabric in order to provide a good connectivity of both interfaces to the routing fabric.

4.6 Fabric Peripheral

The fabric peripheral can be accessed when REG_XIF_OR_PERIPH is set to 1. In this case the fabric will be interfaced as a peripheral. It is the responsibility of the user to upload a bitstream that correctly handles bus requests.

In order to implement a custom peripheral, the peripheral_wrapper must be instantiated in the user design, its module header is shown in Listing 4.1.

```
module peripheral_wrapper (
    output          REQ,
    output          WE,
    output [3: 0]   BE,
    output [23:0]   ADDR,
    output [31:0]   WDATA,

    input           GNT,
    input           RVALID,
    input  [31:0]   RDATA
);
```

Listing 4.1: Module header of peripheral wrapper.

The bus interface of the custom peripheral follows the OBI [50] from the OpenHW Group

and is mapped to address `0x50000000` in the address space of the SoC. The address supplied to the peripheral wrapper is a byte address. The test designs in Chapter 6 include two custom peripherals: One that uses the `RegFile` primitives to make a 32×32 bit memory available via OBI and another one that makes all 7 `IHP_SRAM_1024x32` primitives available for a total memory of 7168×32 bit.

4.7 Custom Instruction Extension

The fabric can implement a custom instruction which can be accessed by the CPU when `REG_XIF_OR_PERIPH` is set to `0`. The interface to the fabric is kept simple: the fabric receives two operands and returns the result. To process instructions in more than one cycle, it is possible to specify after how many cycles the result should be read.

In order to implement a custom instruction, the `xif_wrapper` must be instantiated in the user design, its module header is shown in Listing 4.2.

```
module xif_wrapper (
    output [31:0] RS1,
    output [31:0] RS2,
    input  [31:0] RESULT,
);
```

Listing 4.2: Module header of xif wrapper.

The custom instruction extension implements the `0x5B` opcode, which is free for custom use as specified in the RISC-V specification [13]. The instruction encoding is R-type: `.insn r opcode7, funct3, funct7, rd, rs1, rs2`. `rs1` and `rs2` are the two source registers, and `rd` is the destination register. `funct7` specifies the number of cycles after which the result is read. A custom instruction can easily be executed directly from C code using the `.insn` pseudo directive in GCC as shown in Listing 4.3.

```
int a=42; int b=3; int c;

// Read the result after 13 cycles
__asm__ volatile (
    ".insn␣r␣0x5b,␣0,␣13,␣%0,␣%1,␣%2"
    : "=r" (c)
    :  "r" (a),
       "r" (b)
);
```

Listing 4.3: Executing a custom instruction in C code.

Where `a` and `b` are the variables for the operands and `c` is the variable for the result.

4.8 Compiling Firmware

In order to make writing firmware for Greyhound as straightforward as possible, the repository includes a Board Support Package (BSP). A BSP usually contains code that is needed to get the system to boot and provides basic functionality such as drivers for peripherals. It contains the `crt0` which clears and initializes memory, initializes the stack pointer, configures the vector table for interrupts and finally invokes `main`. With the external SPI Flash, Greyhound can access large programs. Therefore the BSP library of Greyhound includes a libc implementation, namely newlib [56]. Consequently, the BSP contains implementations for the libc system calls. For example, the `write` syscalls calls the platform specific `EF_UART_writeChar` function, in order to write a characters via the UART of the SoC. Therefore, it is possible to write a string using `printf` and the string will be sent via UART. The UART must be configured in advance with the correct baudrate for the system frequency as seen in Listing 4.4.

Listing 4.4: Writing a string via printf to the UART.

```
int main()
{
  EF_UART_setGclkEnable(UART0_BASE, 1);
  EF_UART_enable(UART0_BASE);
  EF_UART_enableRx(UART0_BASE);
  EF_UART_enableTx(UART0_BASE);
  EF_UART_disableLoopBack(UART0_BASE);
  EF_UART_disableGlitchFilter(UART0_BASE);

  EF_UART_setDataSize(UART0_BASE, 8);
  EF_UART_setTwoStopBitsSelect(UART0_BASE, false);
  EF_UART_setParityType(UART0_BASE, NONE);
  EF_UART_setTimeoutBits(UART0_BASE, 0);
  // baudrate = clock_f / ((PR+1)*8)
  EF_UART_setPrescaler(UART0_BASE, F_CPU/(BAUDRATE*8)-1);

  // Write using UART
  printf("Hello World!\n");

  return 0;
}
```

4.9 Verification of the SoC

For verification we chose cocotb [35] as the testbench environment. cocotb enables us to write the testbenches and verification code in Python. In Addition, cocotb is simulator-agnostic because of its GPI, thus making it easier to switch the underlying simulator.

Two simulators were considered for Greyhound's simulations: Icarus Verilog [36] and

Verilator [37]. Since the simulation model of the SPI Flash requires `z` (high-impedance) values in simulation, and Verilator is a 2-state simulator, Icarus Verilog is chosen as the underlying simulator. Due to Greyhound's BSP it is straightforward to compile C programs for the SoC. A Makefile is set up for each of the programs to compile it and link the BSP. Additionally, the binary file is converted to a text file with hex content that can be used to initialize the memory of the SPI Flash inside the Verilog simulation.

For the SoC two tests were implemented: `hello_world` and `custom_instruction`.

The first test, `hello_world`, ensures that the RISC-V core can successfully fetch instructions from the SPI Flash and access the built-in memory and UART. The cocotb test case inside the testbench sets up a UART source and sink at a baudrate of 115200 bps. The testbench sends a single character, and after receiving this character sends back `"Hello World!\n"`. Once the last character is received, the testbench asserts the message to ensure it is correct. As shown in Figure 4.2, the test case is successful.

```
test_hello_world passed
**************************************************************************************
** TEST                                STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
**************************************************************************************
** greyhound_soc_tb.test_hello_world   PASS     2876800.00         134.07      21456.78  **
**************************************************************************************
** TESTS=1 PASS=1 FAIL=0 SKIP=0                2876800.00         134.55      21381.58  **
**************************************************************************************
```

Figure 4.2: Simulation summary of the "Hello World" test case.

The second test, `custom_instruction`, is used to verify the custom instruction interface. The program outputs the return value of the custom instruction in hexadecimal via UART which is then asserted against the expected value. Since at this level only the SoC is tested without the connected eFPGA fabric, the custom instruction simply returns a dummy value: `0xDEADBEEF`. Figure 4.3 shows the digital waveforms of this test case: the first two signals are clock and reset, the next section shows QSPI Flash activity, and the last section shows the UART receive and transmit signals. It can be seen that approximately 2 ms after startup, that there is activity on the UART transmit line: the RISC-V core writes the message `0xDEADBEEF`.

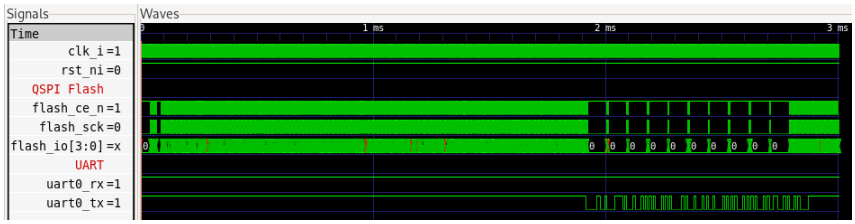


Figure 4.3: Simulation summary of the "Custom Instruction" test case.

These two tests are sufficient as a first step to determine that the SoC is correctly implemented and the CPU can access the peripherals, except for the eFPGA fabric. Chapters 5.8 and 6.3 explain how the eFPGA fabric is simulated and how the final chip is verified.

CHAPTER 5

Generation of the FPGA Fabric

This chapter presents the FPGA fabric part of Greyhound and discusses the design decisions. It also gives an overview of the BELs of the fabric, both the default ones from the FABulous standard library and the custom made ones.

5.1 Generating the Fabric with FABulous

This section gives a high-level overview of the FPGA fabric. Figure 5.1 shows the tile map of the fabric that is part of Greyhound.

The FPGA fabric configuration was adapted from the default FABulous fabric with changes to its tile map as well to the geometry of the individual tiles to fit the given core area and changes to its functionality by adding custom BELs.

The design decision behind this architecture is as follows: Each CLB contains a fast carry chain which connects to all 8 LCs of a CLB. The input to the carry chain is at the bottom of the tile and the output at the top. Naturally this means that CLBs can be stacked vertically in order to build up a longer carry chain for wide additions. Therefore, tiles of a specific type are ordered in columns. This, in turn, means that the data flow is preferred horizontally making it a good choice to place the I/O tiles on the left and the SRAM tiles on the very right.

As of now, FABulous has support for only one global clock net. In theory the tiles and BELs could be easily changed to accommodate more than one global clock net, but the setup for the EDA tools has currently only support for one. The clock net is build up as a clock ladder, meaning that clock edge does not arrive at all flip-flops at the same time. The global clock enters the tiles at the bottom and is propagated upwards row by row. Each row adds a delay to the clock which leads to a "clock wave". Operations that go with the clock wave have naturally a higher timing slack, and operations that go the opposite direction have lower timing slack. Normally, hold-violations wouldn't be of concern in this fabric, as the multiplexers in the switch matrices delay the signals enough to prevent any hold-violations. But due to the clock wave the PnR tool needs to consider even the hold time of the flip flops for a stable circuit. Note that the preferred data flow goes horizontally and the clock wave propagates vertically. It might be of benefit in a future fabric to propagate the clock wave horizontally too in order for the PnR tool to make better use of it.

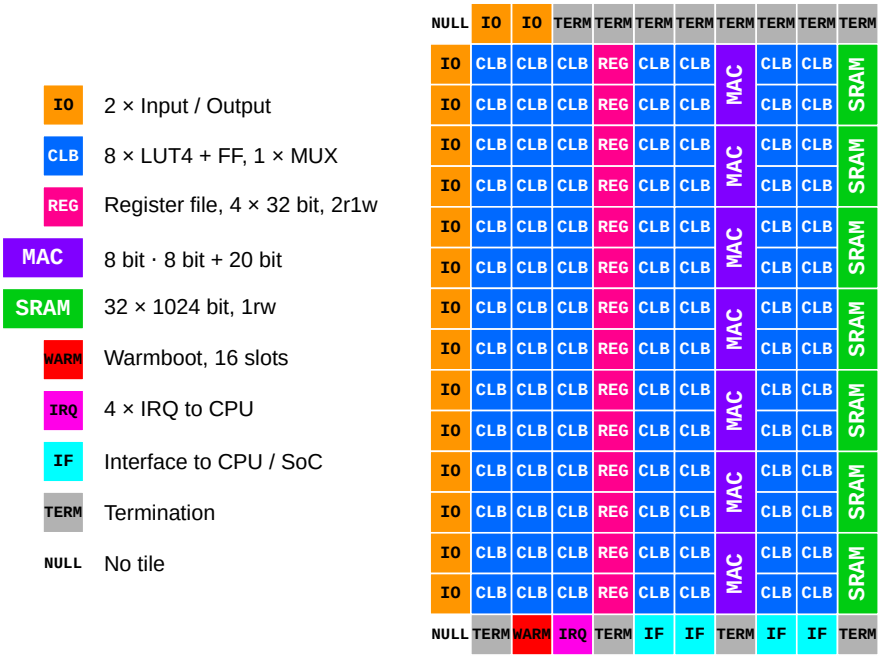


Figure 5.1: Overview of the FPGA tiles in Greyhound and its tile map of the fabric.

5.2 Basic Elements

This section describes the primitives found in Greyhounds FPGA fabric. Some of the BELs are coming from the default FABulous tiles, and some are custom-made for Greyhound. From the point of view of the synthesis tool, BELs are called primitives.

Logic Cell

The LUT4 of the LC can implement any combinatorial logic function with up to 4 inputs; it has a carry logic for a fast carry chain at its inputs. The D-FF is the memory element of the LC and is fed by the output of the Look-Up Table (LUT). It shares its clock and control signals with the other LCs in the CLB. A multiplexer at the output selects either the output of the LUT4 or the D-FF.

We chose a 4 input LUT because on the one hand the default fabric of FABulous currently only supports LUT4, and on the other hand another because a 4 input LUT fits the technology we use for Greyhound at 130 nm well. A comparable example is the Virtex-II series manufactured on a 150 nm process which contains 4 input LUTs [57]. Only in more advanced nodes, where logic becomes denser and routing delays in the fabric dominate, larger LUTs are chosen.

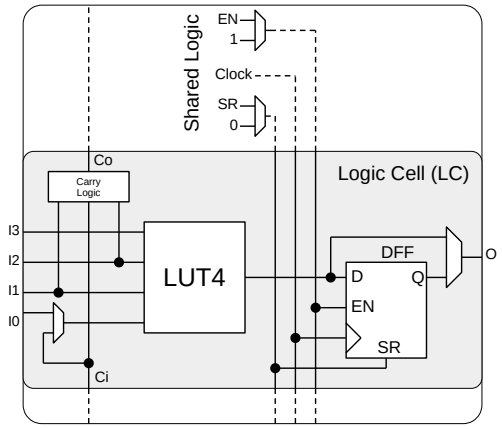


Figure 5.2: The Logic Cell with LUT4, D-FF and carry chain.

Configurable Logic Block

The CLB as seen in Figure 5.3 is the heart of the fabric, but it's not a primitive by itself. Each CLB contains 8 LCs, a fast carry chain connected to all of the LCs and one Multiplexer (MUX). The MUX is fed by the outputs of the LCs and can be configured as 1×MUX8, 2×MUX4 or 4×MUX2.

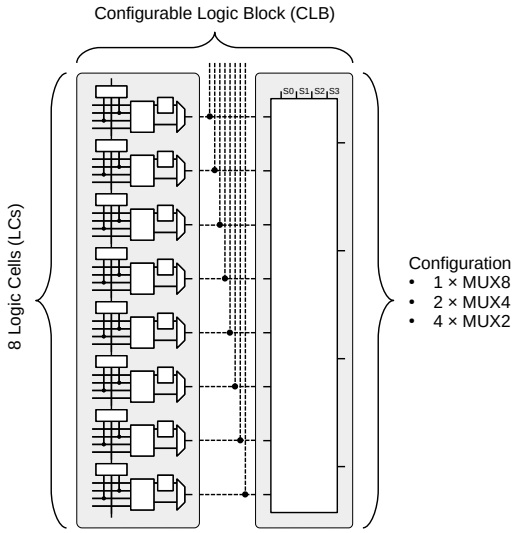


Figure 5.3: The Configurable Logic Block with 8 LCs and one MUX

With the exception of the MUX, the structure of the CLB is very similar to the iCE40 series

from Lattice [17]. The reason for this is that this specific type of carry chain was already implemented in Yosys, the synthesis tool, for the iCE40 series. The FABulous team reused this the technology mapping for their own architecture, which became the default architecture in FABulous. For more information about synthesis, see Chapter 5.6.

Multiply Accumulate

The `MULADD` primitive performs a MAC operation. It has two 8-bit wide inputs A and B, which are multiplied together to form a 16-bit value. This 16-bit value is in turn extended to a 20-bit value by either zero-extending or sign-extending. Next, this 20-bit value is fed into the 20-bit adder with another 20-bit value. This other 20-bit value is either selected as the third input C, or the output of the 20-bit `ACC` register. The 20-bit output of the adder then goes to the output multiplexer `ACCout` which produces the final 20-bit output value Q. The other input of `ACCout` is the output of the `ACC` register. The `ACC` register is fed by the output of the adder and can be synchronously cleared with the `clr` signal. All three inputs, A, B, and C, can be registered if desired.

This setup is quite flexible: it allows to combinatorially multiply two 8-bit numbers and and add a 20-bit number. It also allows to the same operation, but registering all the inputs. It is also possible to continuously accumulate the product of the two 8-bit numbers using the `ACC` register.

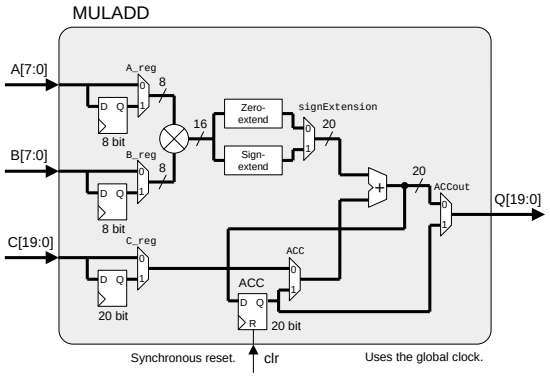


Figure 5.4: The Multiply Accumulate unit.

Register File

The `RegFile` primitive offers 32×4 bit memory with 2 read ports and 1 write port. The read operations can be synchronous or asynchronous.

This structure is perfect as a register file for example of a RISC-V core. With only 8 `RegFile` primitives it is possible to create a 32×32 bit register file as needed to implement the RV32I base ISA.

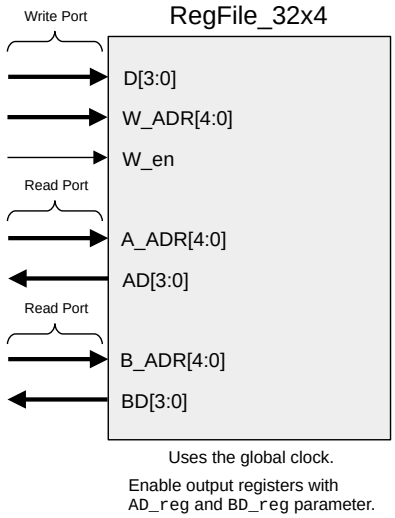


Figure 5.5: The Register File, 4 bit wide and 32 bit deep with one write and two read ports.

5.3 Additional Tiles

IHP SRAM

The `IHP_SRAM_1024x32` primitive has 1024×32 bit of memory with one read/write port and individual bit enable for writes.

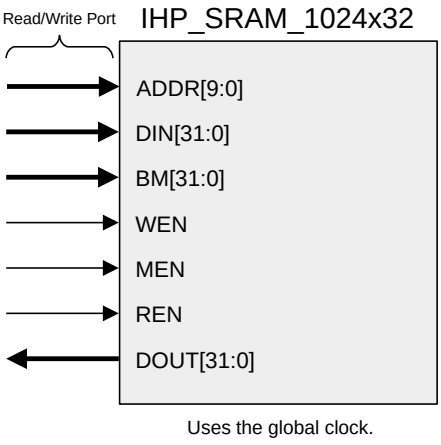


Figure 5.6: The IHP SRAM, 32 bit wide and 1024 bit deep.

Compared to other memories on FPGAs, which are usually dual-ported, this memory has

only one read or write port. Normally, a BRAM with at least one read and one separate write port is preferred as this makes it possible to implement First In First Outs (FIFOs) for, e.g., clock domain crossing. The IHP Open Source PDK currently only contains single-ported SRAMs, hence the single-ported SRAMs on Greyhound. If dual port SRAMs are available in the PDK in the future, Greyhound could be updated to use dual-ported SRAMs.

Warmboot

The `WARMBOOT` enables reconfiguring the fabric from one of 16 slots.

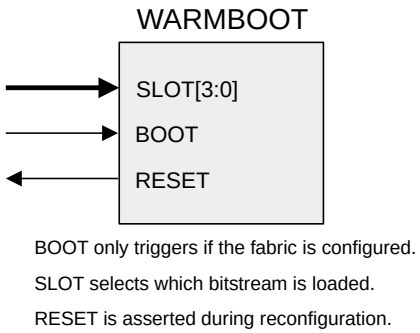


Figure 5.7: The WARMBOOT primitive with 16 different slots to choose from, a trigger and a reset signal.

With only 784×LUT4, Greyhound's eFPGA fabric is on the small side. However, by making use of the `WARMBOOT` primitve, its capacity can be increased significantly simply by time-multiplexing different bitstreams. First, a bitstream is loaded that performs one functionality. Next, either by itself or by user input, the design selects another slot on the `WARMBOOT` peripheral and triggers the `BOOT`. As soon as the fabric config module receives the trigger, it asserts the reset to the fabric and fetches the new bitstream from the SPI Flash with an offset corresponding to the selected slot. Once the fabric has been configured with the new bitstream, the reset is deasserted and the new design starts to operate.

One scenario where this is very useful is to have a "bootloader" bitstream in slot 0 that is able to write data into the other slots of the SPI Flash and enable those bitstreams. This could be done using the Nitro µACM core[1], which implements a USB CDC ACM device in the FPGA fabric making it possible to upload a new bitstream via USB into the SPI Flash.

CPU Interrupt Request

The `CPU_IRQ` primitive is used to send IRQs to the CPU domain. There are 4 IRQ lines available and all of them can be triggered separately.

[1] `https://github.com/no2fpga/no2muacm`

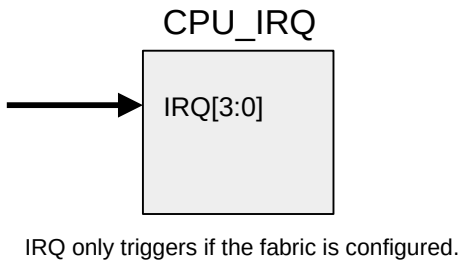


Figure 5.8: The CPU_IRQ primitive with 4 separate IRQ lines.

CPU Interface

`CPU_IF` is a primitive with 16 inputs and 16 outputs to the CPU domain. Its purpose is implementation dependent. On Greyhound 4×`CPU_IF` are used to implement the custom instruction and peripheral interface.

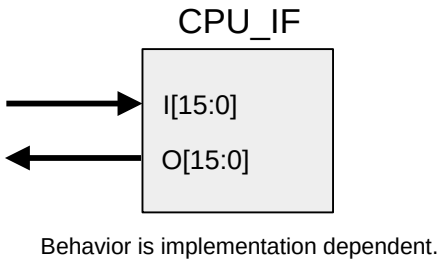


Figure 5.9: The CPU_IF primitive with 16-bit input and 16-bit output.

5.4 Physical Implementation of the Fabric

The most straightforward way of implementing the FPGA is as a sea-of-gate approach. In this approach all of the standard cells of the fabric are placed and routed at once. The problem with this approach is that it doesn't scale well; the time it takes to implement the fabric increases exponentially with the size of the fabric. However, there is a different approach: tile-based. With this approach, the FPGA fabric is split up into regular tiles, which are implemented once and finally stitched together. This approach scales well with the size of the fabric as the pre-implemented tiles simply need to be placed. Another advantage is that timing information for a single tile can be stored in a de-duplicated manner. FABulous does not yet make use of a deduplicated database, but might in the near future.

LibreLane Plugin for FABulous

The LibreLane plugin [58] builds on the author's previous work [59] and has been extended to support heterogeneous fabrics with support for FABulous SuperTiles.

Implementing the Tiles

In the first step, all of the tiles are generated by FABulous and implemented using LibreLane. In this step it is important to consider the physical dimensions of the tiles and where the pins and PDN straps are placed. Figure 5.10 shows some of these considerations: The pin order for one edge needs to match the opposite side, so that when two tiles are abuted the correct pins touch. There should be a pin clearance at the corner or otherwise it might become hard to route. The PDN straps need to fully extend to the boundary, so that the tiles below and above simply continue the PDN.

Using the OpenLane 2 plugin, the pins are placed on-grid, meaning the pins are placed according to the routing grid of the metal layers. This aids the detailed router when connecting traces to the pins. What else needs to be considered is the geometry of the tile: The width should be a multiple of the routing grid pitch for the vertical pins and the height should be a multiple of the routing grid pitch for the horizontal pins.

This is necessary because, although the pins of a single tile are on-grid, as soon as two tiles are connected and the routing grid was not considered for the tile size, the second tile might have off-grid pins. Therefore, the height should be a multiple of 0.68 µm (`met3` pitch) and the width should be a multiple of 0.46 µm (`met2` pitch) for ihp-sg13g2. An example of a YAML configuration file of a tile is shown in Listing 5.1. As can be seen, the width and height are a multiple of the corresponding routing grid pitch.

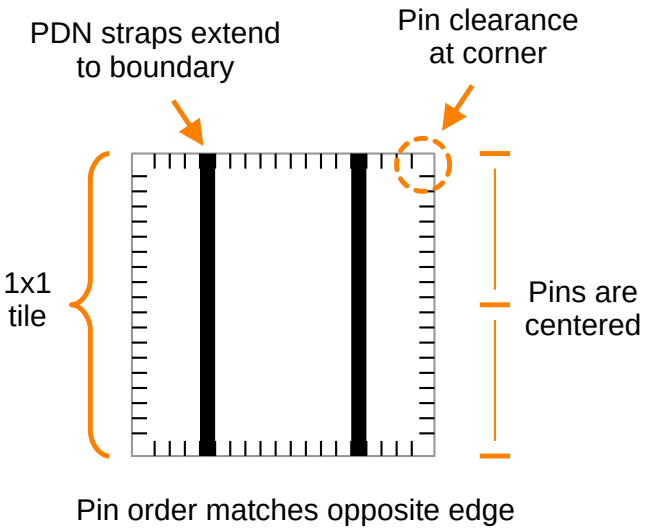


Figure 5.10: Pin placement and PDN strap position considerations for a single tile.

```
meta:
  version: 2
  flow: FABulousTile

FABULOUS_TILE_DIR: dir::.
SYNTH_STRATEGY: "AREA 2" # "AREA 0" "AREA 1" "AREA 2" "AREA 3"

# Basics
DESIGN_NAME: SimpleCLB
VERILOG_FILES:
  - dir::../../models_pack.v
  - dir::../../custom.v

# Clock
CLOCK_PERIOD: 20
CLOCK_PORT: UserCLK

# Area
FP_SIZING: absolute
pdk::ihp-sg13g2:
    # Height should be multiple of 0.42 (Metal2 pitch)
    # Width should be multiple of 0.48 (Metal3 pitch)
    DIE_AREA: [0, 0, 231.84, 241.92]
    PL_TARGET_DENSITY_PCT: 80
```

Listing 5.1: Tile YAML configuration: SimpleCLB.

Stitching the Fabric

The remaining work consists of placing the tiles according to the tile map of the fabric. The tile map is given using the FABulous fabric CSV file and contains the same mapping as seen in Figure 5.1. The LibreLane FABulous Plugin receives the CSV file using the fabric YAML configuration file, specifically the `FABULOUS_FABRIC_CONFIG` key, as can be seen in Listing 5.2.

Another thing to notice in this configuration file is that there is a small gap between the boundary of the top-level fabric macro and the tiles inside of it. This is so that the top-level pins can be placed within this gap. On the top, left and right sides, this gap equals 0.42 um, while at the bottom it is slightly larger with 4.2 um. On the first three sides only pins need to be placed, which are 0.48 um wide. But at the bottom side, the clock net needs to be routed to connect to each column, requiring more space.

However, for the physical implementation of the fabric the OpenLane 2 plugin needs to know the physical dimensions of each tile. Tiles can have varying sizes as long as each row of the fabric has the same height and each column has the same width. Normally, termination tiles do not contain much logic and can therefore be smaller than the tiles in the core of the fabric.

The physical size for the tiles of the fabric are specified in the fabric YAML config for LibreLane using the `FABULOUS_TILE_SIZES` key. The value is a dictionary of tile name and tile size pairs. The tile name is a regular expression and can either match a single tile, e.g., `LUT4AB`,

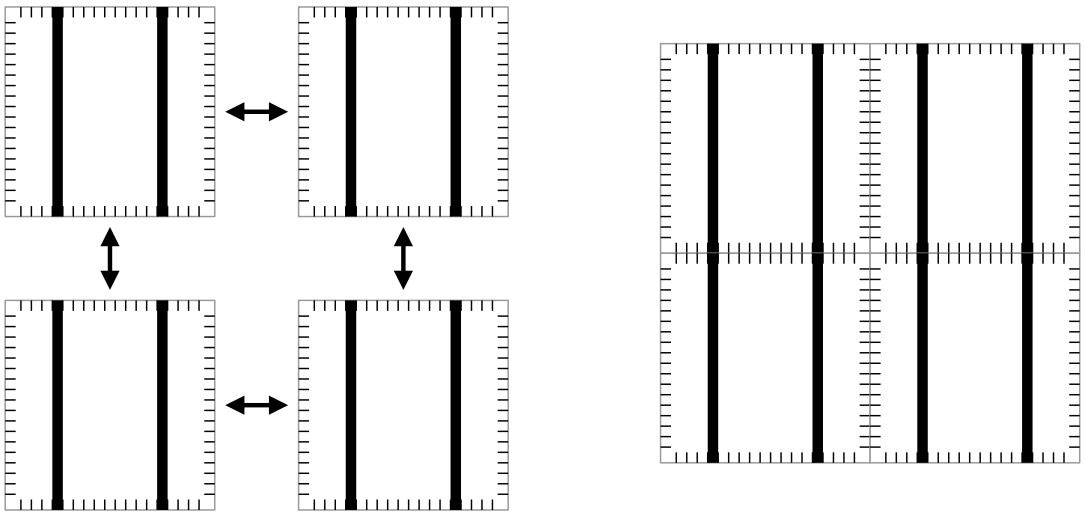
or multiple tiles, e.g., `N_*` matches both `N_term_single` and `N_term_single2`. Supertiles are specified using their sub-tiles, e.g., for `DSP` this would be `DSP_bot` and `DSP_top`. Currently only simple geometries are supported for super tiles such as 1x2 tiles. This approach proofed to be relatively simple but effective. Another automatic approach would be to get the tile sizes by parsing their respective Library Exchange Format (LEF) view.

```
meta:
  version: 2
  flow: FABulousFabric

# Basics
DESIGN_NAME: eFPGA
VERILOG_FILES: []

# FABulous config
FABULOUS_FABRIC_CONFIG: dir::fabric.csv
FABULOUS_TILE_LIBRARY: dir::../fabulous_tiles/tiles/

FABULOUS_TILE_SPACING: 0
# Vertical spacing should be multiple of 0.42 (Metal2 pitch)
# Horizontal spacing should be multiple of 0.48 (Metal3 pitch)
# The halo was chosen so that there's just enough space for the pins
# except for the bottom: here the clock net needs to be routed
FABULOUS_HALO_SPACING: [ 0.48, 4.2, 0.48, 0.48 ]

# Match spacing if possible
FP_IO_HLENGTH: 0.48
FP_IO_VLENGTH: 0.48
```

Listing 5.2: Fabric YAML configuration: Basic configuration.

```
FABULOUS_TILE_SIZES:
  LUT4AB:           [ 231.84, 241.92 ]
  RegFile:          [ 266.40, 241.92 ]
  N_term_single2:   [ 266.40,  60.90 ]
  S_term_single2:   [ 266.40,  59.22 ]
  DSP_*:            [ 196.32, 241.92 ]
  N_term_DSP:       [ 196.32,  60.90 ]
  S_term_DSP:       [ 196.32,  59.22 ]
  IHP_SRAM_*:       [ 108.00, 241.92 ]
  N_term_IHP_SRAM:  [ 108.00,  60.90 ]
  S_term_IHP_SRAM:  [ 108.00,  59.22 ]
  W_*:              [  68.64, 241.92 ]
  S_*:              [ 231.84,  59.22 ]
  N_*:              [ 231.84,  60.90 ]
```

Listing 5.3: Fabric YAML configuration: Tile sizes.



(a) Tiles apart (b) Stitched together

Figure 5.11: Connecting tiles by abutment. (a) shows the tiles apart and (b) shows the tiles together. The pins correctly align with the pins of the neighboring tiles.

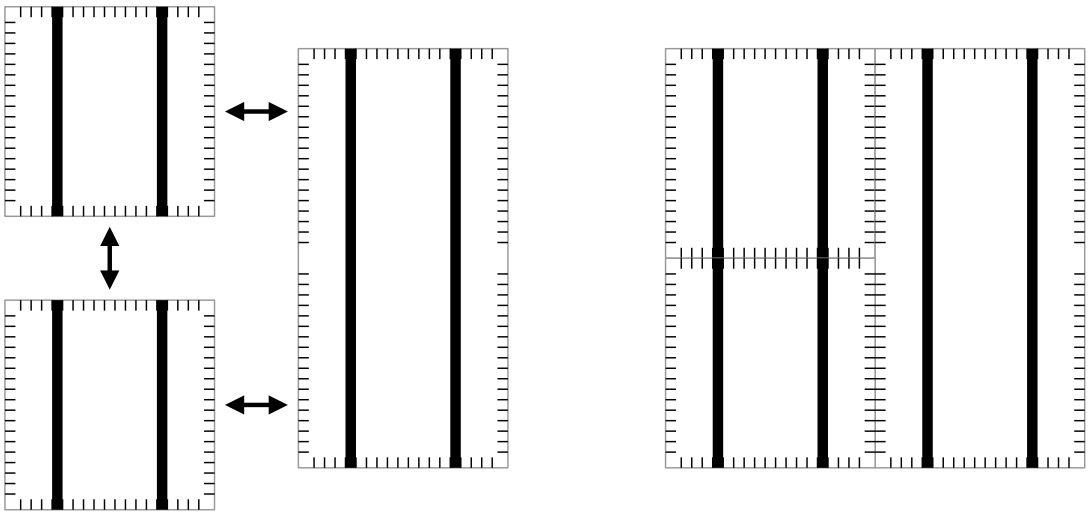
Using the tile map and the tile sizes, the FABulous LibreLane plugin can place the macros accordingly. The tiles are connected by abutment, i.e., there is no space between the tiles. Stitching a fabric with simple 1×1 tiles is as simple as placing the tiles next to each other, as seen in Figure 5.11

There is enough space between the tiles at the boundary and the boundary of the macro in order to insert the top-level pins. There have been issues with pins near the corners of the macro: Sometimes those pins were not aligned with the pins of the respective tiles. This is due the way how pins are placed: Pins are first assigned to a `Section` of 200 possible positions. Next the pin positions are optimized using the Hungarian matching solver. Finally the pins are assigned their real positions. At the edges it could happen that some of the pins are assigned to a non optimal `Section` and therefore do not align with the pins of the tile.

To circumvent this issue a custom step was added to the LibreLane FABulous plugin: `OpenROAD.ManualIOPlacement`. This step allows to manually specify the pin placement for those pins with non-optimal placement near the edges.

Another problem to solve are the top-level pins for the PDN straps. Since we want to allow to place the horizontal PDN straps during the top-level integration of the eFPGA macro, we do not yet place horizontal PDN straps when stitching the fabric.

Unfortunately the vertical PDN straps in the tiles are not considered as top-level pins of the eFPGA macro, but rather as pins of the individual tile macros. To solve this issue another step was added to the LibreLane FABulous plugin: `Odb.FABulousPower`. This step places power and ground pins on the exact locations of the vertical PDN straps. Therefore, the straps are recognized as top-level pins. Since those pins are only connected during top-level integration, the following variable had to be set in the LibreLance config as seen in Listing 5.4.



(a) Tiles apart (b) Stitched together

Figure 5.12: Connecting tiles by abutment. (a) shows the tiles apart and (b) shows the tiles together. Supertiles need to have pins centered at each edge segment.

```
# No extract-unique, because the power straps
# are connected in the top-level integration
MAGIC_NO_EXT_UNIQUE: true
```

Listing 5.4: Do not extract pins as unique.

This option ensures that the top-level pins are always connected during extraction, even if they are not yet connected in the layout. This is needed for LVS of the fabric to succeed.

Note that all vertical straps of the eFPGA macro must be connected to the top-level PDN, otherwise there will be tiles without a power connection!

Stitching a fabric with supertiles works similarly to stitching a fabric with single tiles, as shown in Figure 5.12.

The plugin will consider each edge separately, i.e., the pins that belong to sub-tile of a supertile are placed centered in relation to that edge segment. This is necessary so that supertiles can abut normal 1×1 tiles and the pins are still aligned.

After stitching together the fabric using the LibreLane FABulous plugin we obtain a macro that can be integrated into the top-level design of the chip. For Greyhound the eFPGA macro is shown using the OpenROAD GUI under Figure 5.13.

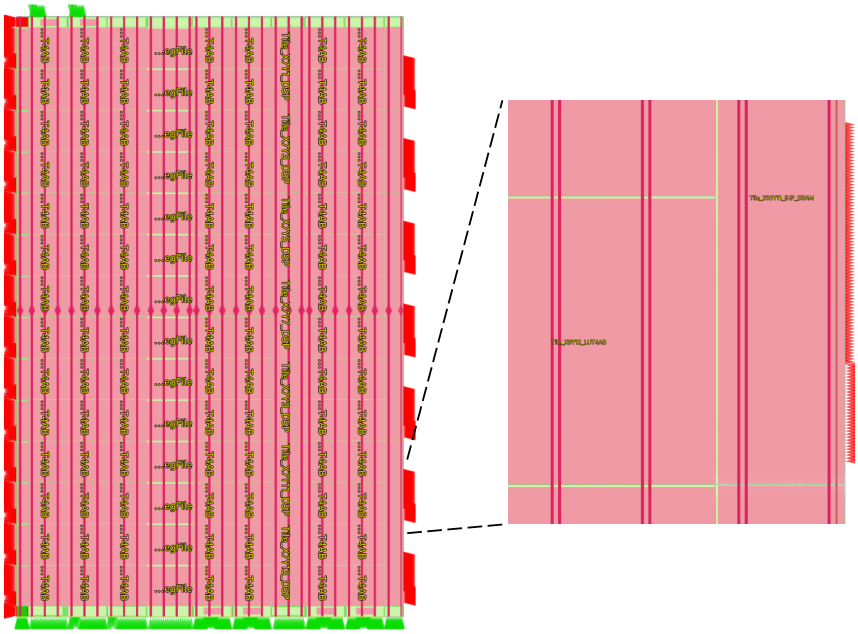


Figure 5.13: The eFPGA fabric viewed in the OpenROAD GUI. On the right side is a zoom in on one of the `IHP_SRAM` tiles in the bottom right.

5.5 Configuration of the eFPGA

There are several paths for uploading a bitstream to the eFPGA fabric in Greyhound, these paths can be viewed in Figure 5.14.

The reason for providing these configuration paths is to enable different applications: We wanted to make it possible to use the eFPGA stand-alone. This implies configuring the eFPGA fabric without the use of the RISC-V SoC. We achieve this by providing an SPI controller and an SPI peripheral that allow the eFPGA to be actively and passively configured from externally. Since the SPI controller is already present, we have not only created the possibility to trigger at startup, but also to enable the user design on the eFPGA fabric to trigger a reconfiguration using the `WARMBOOT` tile.

Another application is to make it possible to not only configure the eFPGA using the RISC-V core by writing the bitstream to a peripheral, but by making it possible for the core to trigger a reconfiguration which uses the SPI controller. This way the core can perform other computations while the eFPGA fabric is reconfigured.

Another application is the ability to configure the eFPGA with the RISC-V core by writing the bitstream to a peripheral as explained in Chapter 4.5. Additionally, the RISC-V core can trigger a reconfiguration that uses the SPI controller. In this way, the core can perform other computations while the eFPGA fabric is being reconfigured.

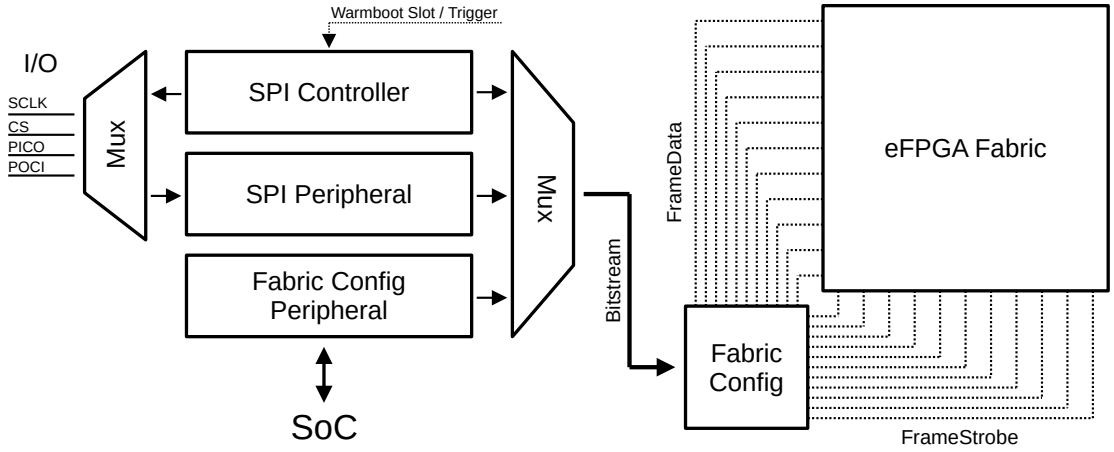


Figure 5.14: Block diagram of the configuration path of the eFPGA Fabric.

1. SPI as Controller

To enable this mode, the `fpga_mode` pin of the chip needs to be tied low. In this mode the eFPGA acts as SPI controller and loads a bitstream from an external SPI flash upon startup.

Additionally, the `WARMBOOT` tile can trigger a reconfiguration from a different slot of the external SPI flash. In total there are 16 different slots to choose from. One slot is `0x4000` bytes large, with `0x3A88` bytes being actual bitstream data and the remaining bytes being padding.

Finally, by writing the slot number to the `REG_TRIGGER_SLOT` register of the `FABRIC_CONFIG` peripheral, the CPU can also trigger a reconfiguration of the FPGA from an external SPI flash. Again, `fpga_mode` must be 0.

2. SPI as Peripheral

To enable this mode, the `fpga_mode` pin of the chip needs to be tied high.

In this mode the eFPGA as a SPI peripheral. The bitstream needs to be supplied in big-endian order via an external SPI controller.

3. CPU Writes Bitstream

The third way is to upload bitstream data directly via the CPU. The CPU writes the raw bitstream data to the `REG_BITSTREAM` register of the `FABRIC_CONFIG_BASE` peripheral. The advantage of this mode is that only one SPI flash needs to be populated on the Printed Circuit Board (PCB).

Approximate times for configuration based on functional simulation:

- SPI as Controller @50MHz: 5ms
- SPI as Peripheral @10Mhz: 22.5ms
- CPU Writes Bitstream @50MHz: 13.3ms

5.6 Bitstream Generation with Yosys & nextpnr

FABulous offers support for open-source bitstream generation for their generated FPGA fabrics. FABulous targets Yosys as synthesis tool and nextpnr for PnR. Previously, FABulous also had support for VPR of the VTR [60] flow as PnR tool, but this support has been removed in favor of supporting a single tool with the available development resources.

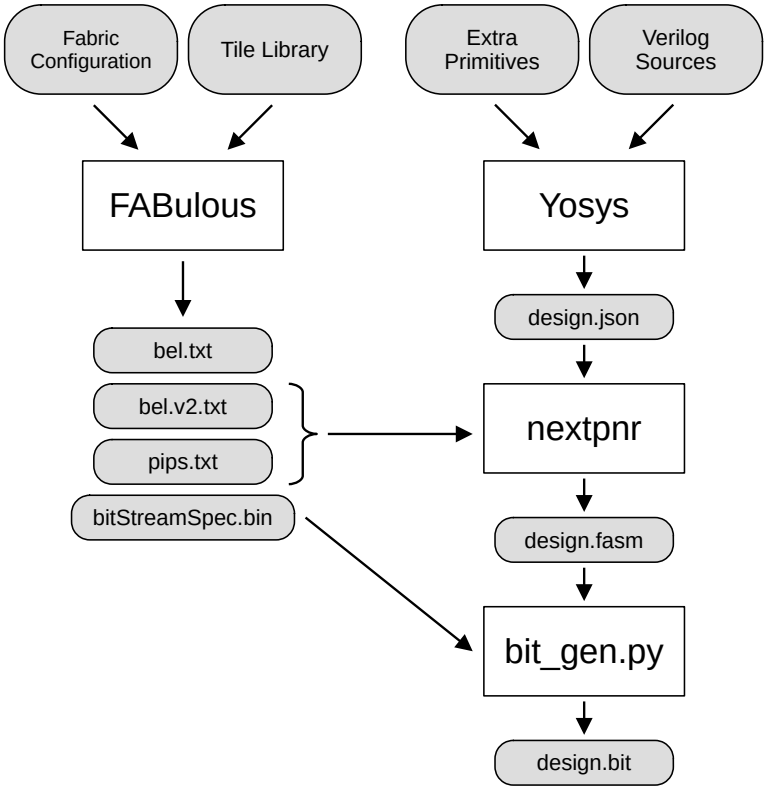


Figure 5.15: The bitstream generation flow.

As part of the tile and fabric generation, FABulous also generates a model of the fabric for nextpnr. The `bel.v2.txt` and `pips.txt` and the deprecated `bel.txt`. As the name suggests, `bel.v2.txt` holds the information about all BELs of the fabric. This includes their

tile position, their "slot" (an alphanumerical character) as well as all inputs and outputs and the configuration bits for parameters. The PIPs of the fabric are described in `pips.txt`. The `bitStreamSpec.bin` file contains the information in order to map the features to their position in the bitstream.

The `synth_fabulous` synthesis scripts already contains the primitives and mappings of the default fabric. Therefore, we only need to supply the Verilog blackboxes for the extra primitives such as `WARMBOOT` or `CPU_IF`. Yosys generates the `design.json` file which contains the synthesized netlist mapped to the primitves available in the target FPGA. PnR is performed using nextpnr; it takes the `bel.v2.txt`, `pips.txt` and `design.json` in order to generate the `design.fasm`. In order to get to the final binary bitstream, `bit_gen.py` takes the `bitStreamSpec.bin` and `design.fasm` in order to map the features to their correct offset in the bitstream. The whole flow can be seen in Figure 5.15.

To prevent manually instantiated primitives from being optimized away, we use the attribute `(*keep*)` on the primitive blackbox modules. This is necessary for the `CPU_IRQ` primitive, for example, as Yosys would otherwise optimize the primitive away, as it has no side effects known to Yosys.

5.7 Evaluation of the eFPGA Fabric

The designs shown in Table 5.1 were implemented for Greyhound. They are also called "user designs" to differentiate them from any HDL code of the ASIC.

Name	Description
`all_zeros`	All outputs are set to zero.
`all_ones`	all outputs are set to one.
`counter`	Implements a 32-bit counter.
`passthrough`	Inputs are connected to outputs
`sram`	All SRAMs muxed together for testing.
`peripheral`	Simple peripheral (32×32 r/w registers).
`peripheral_sram`	All SRAMs available as peripheral.
`xif`	Custom instruction extension (32-bit addition).
`trigger_irq`	Trigger an IRQ after a few cycles.
`trigger_slot0`	Trigger warmboot reconfiguration to slot 0 after a few cycles.
`trigger_slot1`	Trigger warmboot reconfiguration to slot 1.
`serv`	SERV in 4-bit configuration with CSRs enabled on servant, 4 kByte memory.
`fazyrv`	FazyRV in 1-bit configuration and default SoC, 4 kByte memory.

Table 5.1: User designs for Greyhound.

	LC usage QERV	LC usage FazyRV
Greyhound	720	754
iCE40 UP5K	709	634

Table 5.2: Comparison of the QERV and FazyRV core in terms of LC usage between Greyhound's eFPGA and the iCE40 UP5K.

Some of the more demanding designs in terms of required resources are `serv` and `fazyrv`.

The former implements the servant SoC with the SERV [61] RISC-V CPU in 4-bit configuration, called QERV, with CSRs enabled. This configuration uses 720/784 of the available LCs, that is 91%. In addition to the LCs, 10/14 `RegFile_32x4` are automatically inferred and one `IHP_SRAM_1024x32` is used for memory.

The latter, FazyRV [62] in 1-bit configuration uses 754/784 or 96% of the available LCs with 8/14 `RegFile_32x4` and one `IHP_SRAM_1024x32`. This design implements the FazyRV SoC.

As can be seen, Greyhound's eFPGA fabric is capable enough to implement a whole SoC with a soft RISC-V core. Normally this should not be necessary, as Greyound provides a standalone RISC-V core itself and the full resources of the fabric can therefore be used for custom instruction extensions or peripherals.

Since the architecture of Greyhound's LCs is very similar to those in the iCE40 series of Lattice, we show a comparison of the resource usage for QERV and FazyRV SoC in Table 5.2.

The number of LCs used for QERV in Greyhound and iCE40 UP5K is similar, although slightly lower for the latter. The number of LCs for FazyRV is drastically larger for Greyhound than it is for iCE40 UP5K, a difference of +19%.

The large difference in LC usage is made up for by the use of memory. Greyhound uses 8 `RegFile_32x4` to implement the register file for the FazyRV core, and one `IHP_SRAM_1024x32` for the memory of the SoC. There are no dedicated register file primitives on iCE40, so an `ICESTORM_RAM` has to be used, as is the case for the memory of the SoC. However, it is striking that the FazyRV SoC on the iCE40 uses 10 `ICESTORM_RAM`. It seems that even small memories are mapped onto one of these large RAMs, whereas on Greyhound they are implemented in LCs, hence the difference.

Finally, it should be noted that on the iCE40 UP5K both designs use 5 global buffers `SB_GB`, whereas the eFPGA of Greyhound only has one global clock buffer.

5.8 Verification of the FPGA Fabric

For verification of the eFPGA fabric we again use cocotb as the verification framework.

In order to load the bitstream into the fabric we write the `upload_bitstream` coroutine, which takes a path to the bitstream file (`.bin`) and supplies the data bitstream data to the `fabric_config` peripheral, which in turn drives the `FrameData` and `FrameStrobe` signals of the eFPGA fabric to write the data into the configuration bits.

The test cases shown in Table 5.3 use the respective user designs from Table 5.1. First the eFPGA fabric is initialized with the bitstream of the user design, next the design is tested

Test Case	Description
`all_zeros`	Apply random input values, assert the output to all zeros.
`all_ones`	Apply random input values, assert the output to all ones.
`counter`	Deassert the reset of the 32-bit counter, after a certain number of cycles assert the counter value.
`passthrough`	Apply random values to the input and assert the same values on the output.
`sram`	Write random values to the SRAM, assert that the values were correctly written.
`peripheral`	Write to the peripheral by emulating bus accesses, assert that the values were correctly written.
`xif`	Test the custom instruction interface by applying random values to the operands and assert that the output is the addition of those.

Table 5.3: Simulation test cases for the fabric.

by applying stimuli and observing the outputs. For static test cases such as `all_zeros` or `all_ones` this is as simple as setting the inputs to random values and ensuring the outputs are still all zeros or all ones respectively. For a test case like the `sram` one, that means actually writing data into the `IHP_SRAM_1024x32` primitive, reading it back and ensuring the data is still intact.

One problem we encountered is that after loading a new configuration into the fabric while another was already loaded, it could happen that logic loops would form in the routing fabric. If no delays are added to the routing fabric, the whole simulation could get stuck. We tried to prevent this issue by clearing the configuration bits of the fabric in reverse order, however, some logic loops still formed. Therefore, we simply rearranged the order of the test cases that would lead to logic loops, or we would only enable some of the test cases in a single simulation run. To avoid this problem altogether, one can enable delays in the routing fabric, although this has shown to slow down the simulation down considerably.

The simulations of the eFPGA fabric are performed with the RTL code only. We tried to perform a Gate Level (GL) simulation of the eFPGA fabric as well, but `x` values propagate throughout the routing fabric until most of the signals just display an unknown state. After debugging with the FABulous team, this issue is most likely because of too pessimistic x-propagation in the simulator. Unfortunately, Icarus Verilog does not seem to have an option to tweak this behavior, therefore we decide to use RTL simulation only.

The FABulous eFPGA fabrics in some of the Open MPW shuttles have already proven that FABulous is able to generate correct RTL code - as have our simulations - and Yosys synthesizes it correctly. Therefore, we decide it is acceptable to go forward with Greyhound with only a RTL simulation of the fabric.

CHAPTER 6

Implementation, Verification, and Evaluation

In this chapter we discuss the physical implementation of Greyhound using open-source EDA tools. First we compare the SG13G2 process used for Greyhound to the similar SKY130 process to see what the capabilities for digital designs are. Next, we use LibreLane to perform the majority of the digital design flow, only a few steps are done using postprocessing scripts. Finally we cover verification of the final chip design by simulating the chip, performing LVS and running DRC.

6.1 Physical Implementation

For the physical implementation of the digital part of the chip, namely PnR we use LibreLane (cf. Chapter 2.5). It is used to generate the padring, implement the design, and insert the sealring. Afterwards, some further postprocessing steps are carried out outside of LibreLane in order to prepare the chip for submission.

I/O Padring Generation

The padring of Greyhound is part of the design and is generated by the chip-level connections module in OpenROAD called `pad` [63]. In order to integrate this into the LibreLane flow for Greyhound we created a custom step called `OpenROAD.Padring` for padring generation from simple configuration variables. This step in turn calls the `pad` module in OpenROAD to generate the padring.

This is different in contrast to the way how padring generation worked in the Open MPW shuttles for sky130. Here the padring and the core design are two separate macros. The padring was created manually and a Design Exchange Format (DEF) with the pin placements for the core design was created. After the core is finished both macros are merged together.

The advantages of the approach used in Greyhound is:

1. It is more flexible as the padring is defined via configuration variables in LibreLane.

Cell	Function	Comment
`sg13g2_IOPadIn`	Dedicated input.	
`sg13g2_IOPadOut*`	Dedicated output.	Available in 4mA, 16mA and 30mA.
`sg13g2_IOPadTriOut*`	Output with tri-state enable, no input.	Available in 4mA, 16mA and 30mA.
`sg13g2_IOPadInOut*`	Output with tri-state enable, with input.	Available in 4mA, 16mA and 30mA.
`sg13g2_IOPadAnalog`	Analog connection.	
`sg13g2_IOPadIOVss`	IO ground supply.	
`sg13g2_IOPadIOVdd`	IO power supply.	
`sg13g2_IOPadVss`	Core ground supply.	
`sg13g2_IOPadVdd`	Core power supply.	

Table 6.1: IO cells available in the open-source ihp-sg13g2 PDK.

2. It allows to perform full-chip LVS during the LibreLane flow.

Table 6.1 shows the available cells in the open-source ihp-sg13g2 PDK. Some of the cells are available in different drive-strengths. For the `sg13g2_IOPadOut*` and `sg13g2_IOPadInOut*` cells, we choose the drive-strength of 30mA (short-circuit current). This gives us the highest drive-strength, but does not cost more area, as all I/O cells have the same area footprint (same length and width). In total, Greyhound's padring consists of 64 I/O pads as this easily matches with common packages such as QFN-64. The generated padring can be seen in Figure 6.3.

Power Distribution Network

The PDN is responsible for delivering power from the pads to the standard cells and therefore to the transistors. The padring made out of the IO cells shipped with the PDK contains a power ring for both the IO power domain and the core power domain. To make integration with the PDN network easier, we create another power ring for the core power domain in the die area of the chip. This power ring connects to the power and ground pads of the core domain. The core power is distributed to the standard cells and macros using vertical and horizontal PDN straps as seen in Figure 6.3. Due to the current setup and the development state of the PDK, the PDN network uses more layers than necessary: The PDN network is built up of `TopMetal2`, `TopMetal1`, and `Metal5`. This is due to the routing directions defined in the PDK at the time Greyhound was implemented. Meanwhile, the routing directions of all layers have been swapped in the IHP Open PDK, enabling a future implementation of Greyhound to only use `TopMetal2` and `TopMetal1` and keep `Metal5` fully free for routing.

Due to issues in the Clock Tree Synthesis (CTS) the core area was restricted to the bottom area of the chip. Still, the PDN has to cover the full die area in order to connect to the `eFPGA` macro and the SRAMs of the fabric.

The PDN generator module in OpenROAD called `pdn` has support for using regions with a certain voltage domain. Unfortunately we couldn't get this feature to work in Greyhound and resorted to hacks in OpenROAD to set the core area to the desired size, while keeping the die area for PDN generation. We plan to pursue this further in order to build Greyhound without patches to OpenROAD.

Chip Art

The chip art or logo for the chip (Figure 6.1) is drawn using the `TopMetal2.drawing` layer and uses non-45 degree angles. An SVG file was converted to the GDSII layout using a Python script. Any off-grid coordinates are snapped to the manufacturing grid of 5nm. To prevent fill being placed during filler generation a `TopMetal2.nofill` outline is created around the logo.

The logo insertion is not integrated into the LibreLane flow, since when placing the logo as a macro no standard cell sites would be generated underneath. We have therefore opted for a simple solution: place a TopMetal2 blockage for routing and PDN generation over the area where the logo should be inserted, and insert the logo as a postprocessing script with a KLayout Python script.

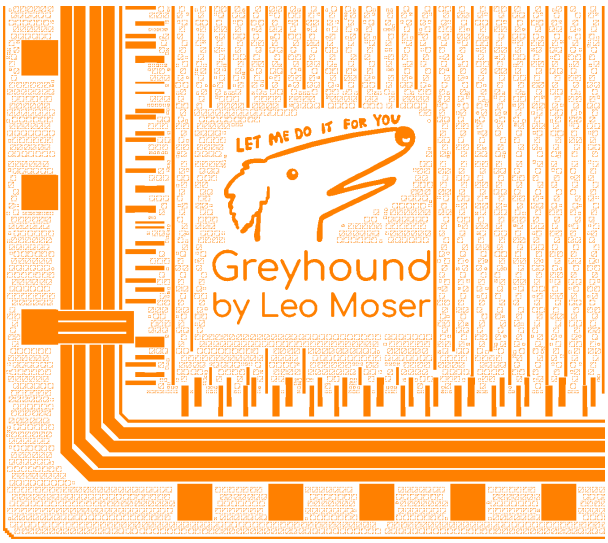
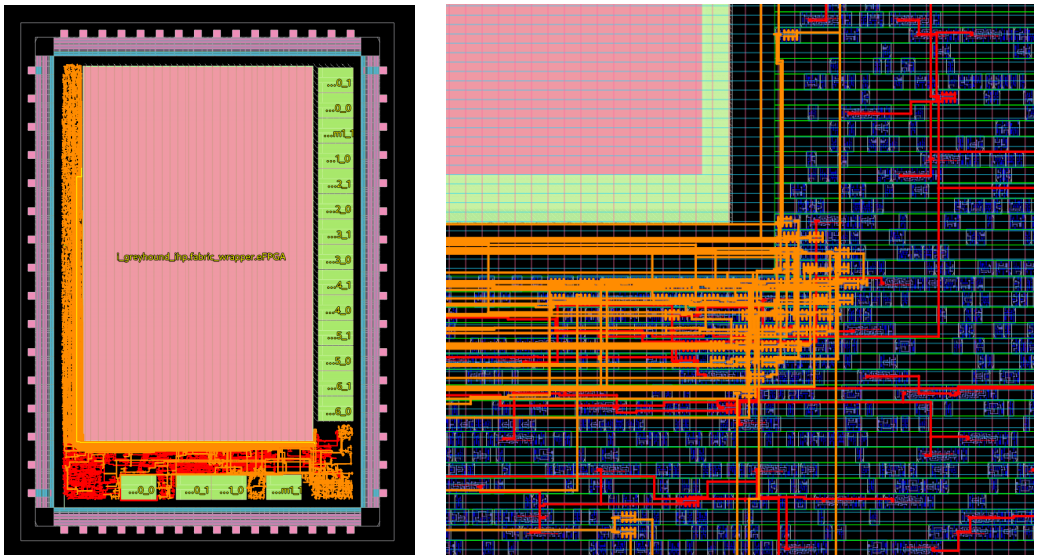


Figure 6.1: The logo on TopMetal2 of the chip. Fill is created around, but not inside the logo.

Clock Tree Synthesis

The goal of CTS is to create a balanced clock tree to ensure the clock signal arrives at all sequential elements such as latches or flip flops at approximately the same time. During CTS OpenROAD estimates net delays using RC-constants. Only after detailed routing more accurate delays are obtained by performing a Standard Parasitic Exchange Format (SPEF) extraction.

Under normal circumstances, the net delays obtained using the RC-constants and the delays obtained using SPEF extraction should agree sufficiently, i.e., the setup and hold times of sequential elements are not violated after SPEF extraction. Unfortunately, this was not the case for Greyhound using the ihp-sg13g2 PDK. After SPEF extraction Static Timing Analysis (STA) uncovered hold-violations that were previously not seen using the RC-constants due to an unbalanced/skewed clock tree.



(a) L-shape (b) Clock buffers

Figure 6.2: Problems during CTS. (a) shows the L-shaped core area with a long clock trace highlighted and (b) shows a large number of clock buffers clustered in the bottom right corner of the eFPGA macro.

The root cause for this issue was uncovered by examining the STA reports after detailed routing. By taking a look at the worst case entry, the reports showed that the path from the clock root buffer to the clock input of the starting point and the the ending point was skewed. Taking a look at the STA reports before SPEF extraction showed no violations. Therefore, the clock tree skew must have been added during SPEF extraction, which is confirmed by large net delays after extraction. Visualizing the clock tree in the OpenROAD GUI (cf. Figure 6.2) showed that it contained very long metal traces, also due to non-optimal clock buffer placement. Three main reasons for this problem were uncovered:

1. A large number of clock buffers were clustered around the bottom right corner of the eFPGA macro leading to long traces in the clock tree.

2. The core shape of Greyhound was initially L-shaped in order to place the `FrameData` registers directly next to their input ports of the FPGA fabric. Due to this shape the clock tree contained long vertical metal traces that amplified the discrepancy between RC-constants and SPEF extraction, as the net delays increased even more compared to the cell delays. After tapeout, it was discovered that this discrepancy is most likely due

to a bug in OpenROAD[1]: only the horizontal component of the clock tree resistance was considered during RC-estimation.

3. The RC-constants set by LibreLane seemed suspect. It seemed that the calculated R and C values for the clock net were swapped, although the values were entered correctly.

To address the first point and second point, the core area is restricted to the bottom part of the chip, as can be seen in Figure 6.3. This reduces long metal traces in the clock tree which leads to the clock tree being better balanced after SPEF extraction. Additionally, some of the clock buffers are no longer clustered in the bottom right corner.

For the third point, instead of manually setting the RC-constants we opted not to set them at all. If the RC-constant are not set manually, OpenROAD will read them from the technology LEF file and therefore uses the correct values. With these two workarounds in place, all hold violations that previously occurred after SPEF extraction could be resolved.

[1] `https://github.com/The-OpenROAD-Project/OpenROAD/discussions/6651`

6.2 Finished Chip Layout

The final implementation of the chip can be seen in Figure 6.3 through the OpenROAD GUI.

The large macro in the center is the eFPGA. On the left of it there is some space for the routing and on the right are the SRAMs for the fabric. Underneath is the sea of gates of the SoC with four more SRAM macros. Each SRAM macro is 16-bit wide, therefore two of each are combined for a data width of 32-bit. What can not be seen in the OpenROAD GUI is the chip-art, as it is added in a later step. However, the area for the logo is already reserved using a TopMetal2 blockage in the lower left of the chip.

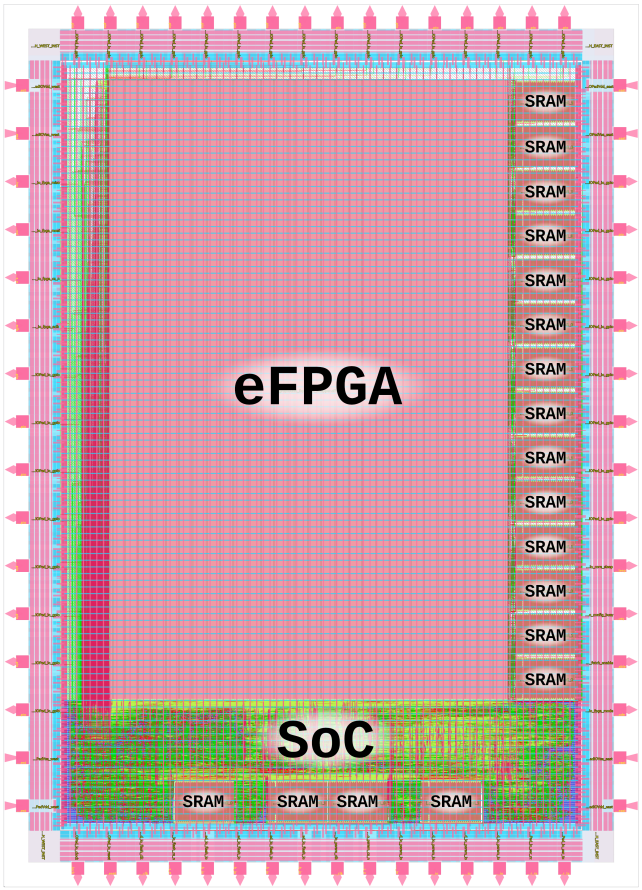


Figure 6.3: The layout of Greyhound as shown in the OpenROAD GUI overlayed with an annotation for the individual macros.

In the next step, the GDSII of Greyhound is streamed out from the LEF/DEF using KLayout and magic. The process is carried out with both tools in order to be able to detect errors during streamout that affect only one of the tools. An XOR check is then run against both layouts to detect any differences. Afterwards, DRC is run to ensure no design rules are violated. Both the streamout process and the final DRC are explained in more detail in Chapter 6.3.

Table 6.2 shows the functional cells of the SoC of Greyhound. This excludes buffers such as `sg13g2_buf_1`, delay gates such as `sg13g2_dlygate4sd1_1`, antenna diodes such as `sg13g2_antennanp`, tie cells, I/O cells, and any macros, i.e., the eFPGA and the SRAM macros. Cells with the same drive-strength are merged together, e.g., all `sg13g2_nand2_1` and `sg13g2_nand2_2` cells are merged into a single category.

The purpose of this table is to get an overview of the gates required just for the SoC of Greyhound. One gate stands out: a single `sg13g2_lgcp`. The `sg13g2_lgcp` is a clock-gating cell, which is manually instantiated to gate the clock tree of the CV32E40X core in order to enable it to sleep when the `wfi` instruction is called.

Cell	Count
sg13g2_a21o	666
sg13g2_a21oi	4646
sg13g2_a221oi	774
sg13g2_a22oi	2968
sg13g2_and2	813
sg13g2_and3	166
sg13g2_and4	50
sg13g2_dfrbp	6648
sg13g2_dlhq	81
sg13g2_inv	1283
sg13g2_lgcp	1
sg13g2_mux2	3725
sg13g2_mux4	67
sg13g2_nand2	2809
sg13g2_nand2b	674
sg13g2_nand3	665
sg13g2_nand3b	55
sg13g2_nand4	528
sg13g2_nor2	5249
sg13g2_nor2b	404
sg13g2_nor3	1069
sg13g2_nor4	366
sg13g2_o21ai	3309
sg13g2_or2	314
sg13g2_or3	126
sg13g2_or4	67
sg13g2_xnor2	1515
sg13g2_xor2	883

Table 6.2: Functional cell count of the SoC, excluding buffers, delay gates, antenna diodes, tie cells, I/O cells, and macros such as the eFPGA and the SRAM.

6.3 Verification of the Chip

Before we send the chip to production, we want to have a high degree of confidence that it will work as designed. In order to gain this level of confidence, we run the final chip top-level simulations in Chapter 6.3 to ensure all of the blocks are connected correctly. The eFPGA fabric is solely simulated in RTL code due to the issue mentioned in Chapter 5.8. The whole SoC and the configuration logic of the fabric, however, is simulated with the RTL code and with the synthesized GL code. Afterwards, we run STA in Chapter 6.3 to check all timing paths for violations and acquire the maximum clock frequencies for all available corners. In Chapter 6.3 we perform LVS to prove the layout matches the GL netlist. And finally, in Chapter 6.3 we run DRC upon the layout to ensure it is manufacturable.

Chip Top-Level Simulation

We perform a chip top-level simulation, meaning only the actual IO pads are probed and written to in the cocotb testbench. Again, this is only possible if the simulator supports `z` (high-impedance) values, hence we use Icarus Verilog also for this simulation. The first things that are done in the testbench are applying a clock to `io_clock_PAD` and asserting the reset on `io_reset_PAD`. After the reset is deasserted depending on the test case the eFPGA bitstream is loaded either via the SPI peripheral interface (`io_fetch_enable_PAD` = 0) or as a SPI controller while the testbench writes the bitstream data via an SPI controller. Another way to upload the bitstream is the RISC-V core triggering a reconfiguration or directly writing the bitstream data to the fabric configuration peripheral at address `0x40000000`.

The test cases in Table 6.3 verify that the eFPGA fabric can be configured via all intended configuration modes and that all features are accessible: custom instruction execution, access to the custom peripheral, triggering of an IRQ. These tests are successfully run using the RTL code of Greyhound, and using the GL code for the SoC and the configuration logic. Several issues were caught this way, including inverted output-enable polarity on the FPGA I/Os.

Shown in Figure 6.4 is the simulation summary for the `custom_instruction` test case. As can be seen, only one test case is executed at a time. This limitation is due to the need for the SPI Flash model to load a different file. To run the other test cases, they need to be individually enabled and the simulation rerun, however, this may be improved in the future.

```
**************************************************************************************************
** TEST                                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
**************************************************************************************************
** greyhound_ihp_top_tb.test_hello_world           SKIP          0.00          0.00          -.--  **
** greyhound_ihp_top_tb.test_custom_instruction    PASS   15474380.00        748.36      20677.60  **
** greyhound_ihp_top_tb.test_fpga_all_zeros        SKIP          0.00          0.00          -.--  **
** greyhound_ihp_top_tb.test_fpga_all_ones         SKIP          0.00          0.00          -.--  **
** greyhound_ihp_top_tb.test_cpu_trigger_fpga      SKIP          0.00          0.00          -.--  **
** greyhound_ihp_top_tb.test_fpga_peripheral       SKIP          0.00          0.00          -.--  **
** greyhound_ihp_top_tb.test_fpga_peripheral_sram  SKIP          0.00          0.00          -.--  **
** greyhound_ihp_top_tb.test_fpga_irq              SKIP          0.00          0.00          -.--  **
** greyhound_ihp_top_tb.test_fpga_blinky           SKIP          0.00          0.00          -.--  **
**************************************************************************************************
** TESTS=9 PASS=1 FAIL=0 SKIP=8                  15474380.00        748.91      20662.45  **
**************************************************************************************************
```

Figure 6.4: Summary of the chip top-level simulation for the `test_custom_instruction` test case.

Test Case	Description
`hello_world`	RISC-V core executes "Hello World!" program, testbench asserts the UART output.
`fpga_all_ones`	eFPGA receives `all_ones` bitstream as SPI peripheral, testbench asserts all ones.
`fpga_all_zeros`	eFPGA loads `all_zeros` bitstream as SPI controller, testbench asserts all zeros.
`cpu_trigger_fpga`	RISC-V core executes a program to trigger the eFPGA in reconfiguring itself in SPI controller mode. Testbench asserts the output.
`custom_instruction`	RISC-V core executes a program to read the bitstream data from its SPI Flash and writes it to the fabric configuration peripheral. The bitstream contains a custom instruction implementation, of which the testbench asserts the output.
`fpga_peripheral`	eFPGA loads `peripheral` bitstream as SPI controller, RISC-V core accesses the peripheral by writing and reading from it. The testbench asserts that the memory transactions are correct.
`fpga_peripheral_sram`	eFPGA loads `peripheral_sram` bitstream as SPI controller, RISC-V core accesses the peripheral by writing and reading from it. The testbench asserts that the memory transactions are correct.
`fpga_irq`	eFPGA loads `trigger_irq` bitstream as SPI controller, RISC-V core waits for an IRQ. The testbench asserts that the core has woken up.
`fpga_blinky`	eFPGA loads `trigger_slot1` bitstream as SPI controller. This bitstream triggers reconfiguration from slot 1 using the `WARMBOOT` primitive. The bitstream in slot 1 (`trigger_slot0`) triggers reconfiguration from slot 0. The testbench asserts that the output toggle according to the active bitstream.

Table 6.3: Test cases for chip top-level simulation.

Static Timing Analysis

Static Timing Analysis (STA) is performed during the LibreLane flow both before and after detailed routing. The static in STA means that the circuit is not simulated but rather all timing paths are statically checked for setup and hold violations by calculating the delays. Table 6.4 shows the STA results for Greyhound over all three corners. No hold violations were detected.

The critical path for the maximum clock frequency starts at `_77900_/Q`, also called:

```
i_greyhound_ihp.i_greyhound_soc.cv32e40x_core.controller_i.
    bypass_i.if_id_pipe_i[176]
```

and goes through:

```
i_greyhound_ihp.i_greyhound_soc.i_obi_mux.i_rr_arb.gen_arbiter.
    gen_int_rr.gen_fair_arb.i_lzc_lower.gen_lzc.in_tmp[0]
```

and finally ends in `_77994_/D`, a D-Flip-Flop (FF) of type `sg13g2_dfrbp_1`.

This suggests that the critical path is in the hazard/bypass/stall control instance of the CV32E40X core and the arbitration logic of the SoC.

corner	frequency
`nom_fast_1p32V_m40C`	85 MHz
`nom_typ_1p20V_25C`	55 MHz
`nom_slow_1p08V_125C`	34 MHz

Table 6.4: STA results for the SoC after PnR and SPEF extraction.

Layout Versus Schematic

LVS is used to ensure that the layout and the schematic match and provide the same functionality. They layout of Greyhound is generated as follows:

After PnR OpenROAD produces the LEF and DEF file of Greyhound. LibreLane then streams out the GDSII from the LEF/DEF using KLayout and magic separately. The two resulting GDSII files must contain the same geometry. This is checked via an XOR step in LibreLane. After the XOR check the sealring is inserted. The spice netlist of the layout is extracted using magic. This netlist will be compared against the powered GL Verilog netlist. In a postprocessing step the logo of Greyhound is inserted. Finally, filler insertion is performed to get to the final layout.

The extraction of the layout is done using abstract cells. This means that cells such as the standard cells, the SRAMs, the eFPGA and the I/O cells are not extracted down to the transistor level, but rather are handled as abstract cells. This has a few benefits: extraction is much faster and the resulting netlist is smaller in size. The reason we chose this method is because the magic setup for the IHP Open Source PDK does not yet correctly extract the IO cells. As long as the macros themselves are LVS clean and the obstructions in the LEF view are correct - as it should be - this is not an issue.

For LVS we use netgen [34]. To get to a full match the netlists had to be slightly edited since the Verilog models of the power and ground pads do not contain any power connections. These edits only change how the power pads are connected to the power nets. The edits have been thoroughly verified and are documented in the Makefile in the repository of Greyhound [6].

Running LVS, netgen gives us the the much sought-after message:

```
Final result: Circuits match uniquely.
```

Design Rule Check

To ensure the GDSII layout does not violate any rules, we perform a DRC. The IHP Open Source PDK has a DRC set up for both magic and KLayout. There exist two different runsets for KLayout:

- Minimum Rule Set
- Maximum Rule Set

To be eligible for submission of the design to the IHP-Open-DesignLib [11] shuttle program, the minimum rule set must be passed. This is to ensure the design is manufacturable, whereas the maximum rule set is to ensure proper operation of the devices. Since Greyhound does not include any custom analog designs and we make use of the digital standard cells provided by the PDK, we decide to run only the minimum rule set.

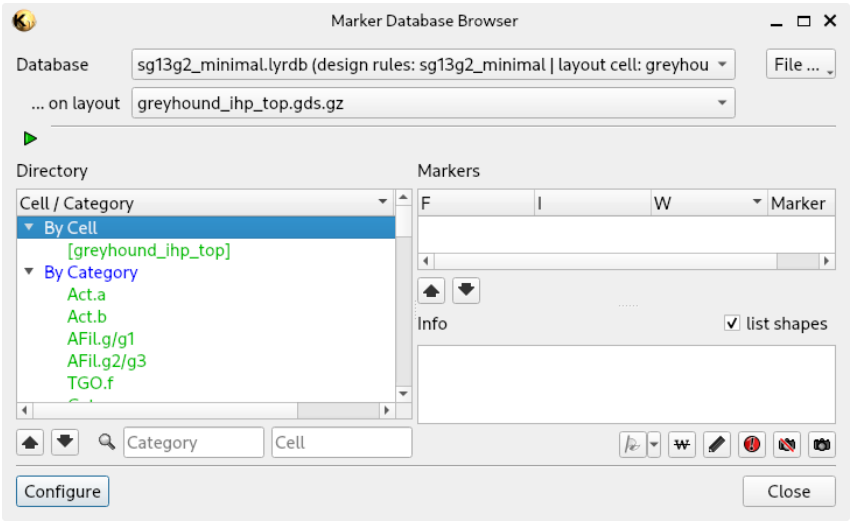


Figure 6.5: KLayout "Marker Database Browser" showing a clean top-level cell for **greyhound_ihp_top** with the **sg13g2_minimal.lydrb** ruleset.

As shown in Figure 6.5, Greyhound passes the minimum rule set without any violations and is therefore eligible for submission.

6.4 Submission for Manufacturing

With STA, LVS and DRC passing, Greyhound can be submitted to the IHP-Open-DesignLib [11] shuttle program. The IHP Free MPW runs are funded by a public German project FMD-QNC (16ME083). Without this funding, the tapeout of Greyhound would not have been possible.

The IHP Free MPW runs include 7 tapeouts from 2024 to 2025. The available area ranges from 10 to 220 mm^2 and the available process technologies are SG13G2 and SG13CMOS. SG13G2 is the BiCMOS process and SG13CMOS is a subset of the same process for CMOS only. Greyhound has been submitted to the shuttle on May 9, 2025, therefore the technology is SG13G2. However, Greyhound would also be manufacturable on SG13CMOS.

The submission process is as follows: Greyhound is submitted via a Pull Request (PR) to the `TO_Apr2025`[2] repository. Upon submission, a rejection test is triggered via the Continuous Integration (CI), which runs the KLayout minimum rule set. Once passed, Greyhound is eligible for the selection process, where the priority of designs is weighted according to some criteria such as completeness of data and documentation, area utilization, uniqueness and first-time submission. Fortunately, Greyhound was selected and received the `FMD_QNC_09_` prefix as seen in Listing 6.4.

```
File name: FMD_QNC_09_greyhound_ihp_top.gds.gz
Top cell name: greyhound_ihp_top
Testfield: T586
Check Summe: 09eddbdd4b2b3ea8835b736985d4c77d
Sent: 11-04-2025, 11:56:51
registered size: (3.6 mm X 5.0 mm )
calculated size: (3.536 mm X 4.936 mm))
ID (use this as reference for questions): 54557
```

Listing 6.1: Information on the submitted design.

As of writing, Greyhound is being processed at IHP's pilot line facility which will take around 6 months. After the chip is manufactured, we will be able to loan the chip for a duration of up two years in order to perform verification and characterization. The results should be made public through the IHP-Open-DesignLib repository. After these two years, the chip will go back to the IHP Open Chip Depot to be rented by someone else.

[2] `https://github.com/IHP-GmbH/TO_Apr2025/pull/9`

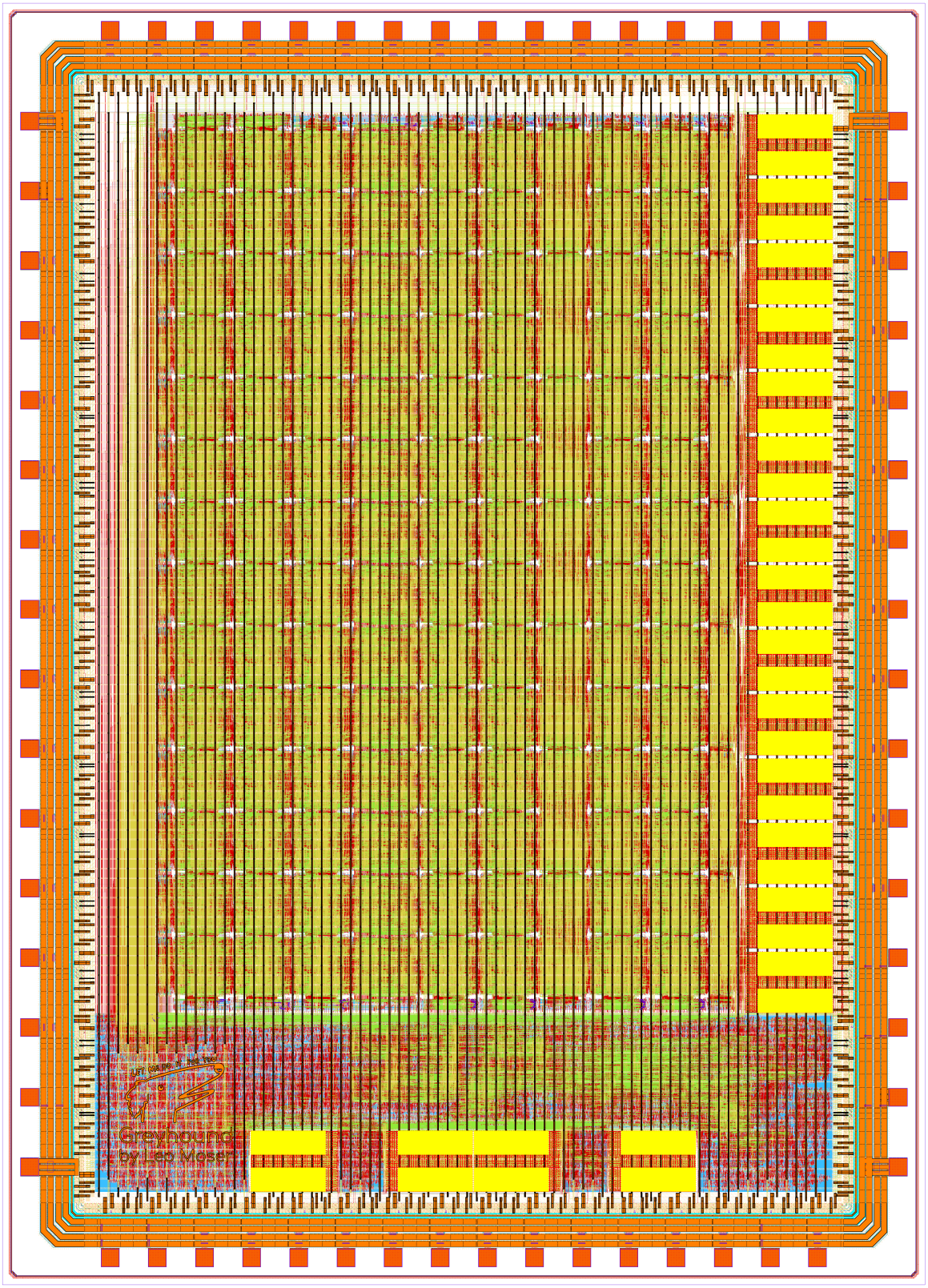


Figure 6.6: Final layout of Greyhound without fill inserted.

CHAPTER 7

Conclusion and Future Work

In this thesis, we presented Greyhound, a RISC-V SoC with tightly coupled eFPGA, designed with open-source EDA tools and the IHP Open Source PDK for the SG13G2 130 nm BiCMOS process.

A RISC-V SoC with QSPI Flash and PSRAM controller, builtin SRAM and UART was implemented using the CV32E40X as core. Firmware for the RISC-V core and SoC can be easily compiled thanks to the BSP of Greyhound including newlib, a libc implementation. Greyhound's eFPGA fabric was generated using FABulous and contains 784×LCs (LUT4+FF), 7×SRAM, 7×MAC, 14×Register files and 32×I/Os. Custom tiles were implemented to provide connectivity to the RISC-V CPU and the SoC, enable warmboot functionality, and allow the fabric to trigger up to 4 IRQs. The eFPGA can act as a custom instruction extension for the CPU, a custom peripheral at address `0x50000000` of the SoC, or as a standalone FPGA.

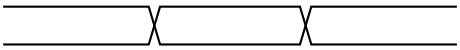
In this work, we developed custom configuration logic to configure the eFPGA fabric as SPI controller or SPI peripheral. The RISC-V core and the eFPGA itself are able to trigger a reconfiguration from one of 16 slots in the SPI Flash. Alternatively the RISC-V core can directly write bitstream data to the configuration logic, enabling partial reconfiguration.

We generated bitstreams for the eFPGA using a fully open-source toolchain: Yosys as synthesis tool and nextpnr for PnR. We compared two soft-core implementations on Greyhound's eFPGA, namely SERV and FazyRV, with the iCE40 UP5k from Lattice. The results show that Greyhound's eFPGA fabric has similar efficiency in resource usage, although much less total resources are available compared to the iCE40 UP5k.

We managed to gain a high degree of confidence in the chip design by performing extensive simulation of individual components such as the SoC and the eFPGA fabric but also of the chip top-level using the RTL netlist and the GL netlist of the SoC and the configuration logic. We simulated all the different configuration paths, including configuring the eFPGA fabric at startup using the SPI controller and SPI peripheral, writing a bitstream using the RISC-V core, triggering a reconfiguration from the eFPGA using the `WARMBOOT` tile, and triggering a reconfiguration from the RISC-V core.

STA results showed that the SoC runs at 55MHz for the `nom_typ_1p20V_25C` corner. We performed a successful LVS check on the design with abstract macros. Greyhound passes the KLayout minimal DRC without any violations.

The final die size of Greyhound is 3.6mm×5mm. The chip was submitted to the IHP-Open-DesignLib, a publicly funded German project FMD-QNC (16ME083), and Greyhound was accepted for manufacturing in April 2025. In about 6 months after submission, we will receive chips and be able to verify and characterize Greyhound.



Future work is possible mainly in three broad topics: physical implementation, custom instruction interface, and improving the eFPGA architecture. In this work, we have already made some efforts into shrinking the tiles, but there is still room for improvement:

1. Remapping of configuration bits.
2. Swapping of pins.
3. Usage of transmission gate with integrated SRAM cell.

The first two improvements are outlined in Chung et al. [64]. Since the order of the configuration bits does not matter for functionality, the individual latches can be swapped. A different order can lead to less congestion during routing and, therefore, to smaller tiles. The second improvement is an extension of the same idea: if a different order in configuration bits leads to less congestion, a different pin order can also lead to less congestion. Thirdly, multiplexers in the switch matrix are simple to implement but require a large area. Using transmission gates instead of multiplexers and SRAM cells instead of latches for configuration memory could result in smaller tiles.

The custom instruction interface between the CV32E40X and the eFPGA was kept simple in this work (two operands, one result) in order to keep the number of signals low. This simplified the physical implementation. However, if the full XIF of the CV32E40X with handshaking is used instead, this would enable multi-cycle instructions without having to specify the delay cycles in the instruction. This would also make it possible to pipeline the custom instruction execution. In addition, support for custom CSRs and direct memory access can also be added.

Another improvement to the custom instruction interface is, instead of splitting it up into 4×`CPU_IF` tiles, the creation of a single supertile that spans multiple columns. In this way, no special primitives wrapper is required in the Verilog code.

The architecture of the eFPGA can be improved as well. It is currently very similar to Lattice's iCE40 series. The FABulous team has reused support for the carry chain in Yosys. In this architecture, the carry chain is prior the LUT, but could be more effective after it. Greyhound's eFPGA uses SRAM blocks instead of Block Random-Access Memorys (BRAMs). This is not optimal since BRAMs are used to implement FIFOs, which are used in many designs. To implement a proper BRAM, we need a dual-port SRAM for the ihp-sg13g2 PDK. Finally, Greyhound's eFPGA currently has no timing information in nextpnr. The FABulous team is working on timing annotation support, however, it is not yet complete.

Bibliography

[1] M. Venn, "Tiny tapeout: A shared silicon tape out platform accessible to everyone," vol. 16, no. 2, pp. 20–29. [Online]. Available: https://ieeexplore.ieee.org/document/10584359 → [p1]

[2] European Chip Skills Academy. (2025) Skills Strategy 2024. [Online]. Available: https://chipsacademy.eu/wp-content/uploads/2024/11/ECSA-Skills-Strategy-2024.pdf → [p1]

[3] FOSSi Foundation. (2025) Open Source Chips for Europe. [Online]. Available: https://open-source-chips.eu/ → [p1]

[4] ——. (2025) Importance of Open-Source EDA Tools for Academia. [Online]. Available: https://open-source-eda-letter.eu/ → [p1]

[5] Free Silicon Foundation. (2025) Recommendations and roadmap for the development of open-source silicon in the EU. [Online]. Available: https://wiki.f-si.org/images/1/19/Recommendations_and_roadmap_open_silicon_2023_11_03.pdf → [p1]

[6] Leo Moser. (2025) Greyhound: A RISC-V SoC with tightly coupled eFPGA on IHP SG13G2. [Online]. Available: https://github.com/mole99/greyhound-ihp → [p2], [p53]

[7] OpenHW Group. (2025) CV32E40X Repository. [Online]. Available: https://github.com/openhwgroup/cv32e40x → [p2], [p5]

[8] FPGA-Research. (2025) FABulous Repository. [Online]. Available: https://github.com/FPGA-Research/FABulous → [p2]

[9] D. Shah *et al.*, "Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs," in *Proc. of the 27th IEEE Int. FCCM Symp.* IEEE, 2019. → [p3]

[10] IHP GmbH. (2025) IHP Open Source PDK. [Online]. Available: https://github.com/IHP-GmbH/IHP-Open-PDK → [p3], [p6]

[11] ——. (2025) IHP-Open-DesignLib. [Online]. Available: https://ihp-open-ip.readthedocs.io → [p3], [p53], [p54]

[12] BMBF. (2025) FMD-QNC. [Online]. Available: https://www.elektronikforschung.de/projekte/fmd-qnc → [p3]

[13] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*, RISC-V Foundation, 5 2017. → [p5], [p14], [p22]

[14] OpenHW Group. (2025) CV32E40P Repository. [Online]. Available: https://github.com/openhwgroup/cv32e40p → [p5]

[15] A. Traber et al. (2025) RI5CY Core: Datasheet. [Online]. Available: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf → [p5]

[16] D. Koch, N. Dao, B. Healy, J. Yu, and A. Attwood, "FABulous: An embedded FPGA framework," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. Association for Computing Machinery, pp. 45–56. [Online]. Available: https://doi.org/10.1145/3431920.3439302 → [p6]

[17] Lattice Semiconductor. (2025) iCE40 UltraPlus: Datasheet. [Online]. Available: https://www.latticesemi.com/view_document?document_id=51968 → [p6], [p28]

[18] N. Dao, A. Attwood, B. Healy, and D. Koch, "FlexBex: A RISC-v with a reconfigurable instruction extension," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, pp. 190–195. → [p6], [p12]

[19] SkyWater Technology. (2025) SKY130 Open Source PDK. [Online]. Available: https://github.com/google/skywater-pdk → [p6]

[20] Globalfoundries. (2025) GF180MCU Open Source PDK. [Online]. Available: https://github.com/google/gf180mcu-pdk → [p6]

[21] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm finFET predictive process design kit," vol. 53, pp. 105–115. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002626921630026X → [p6]

[22] NCSU. (2025) FreePDK45. [Online]. Available: https://eda.ncsu.edu/freepdk/freepdk45/ → [p6]

[23] MOSIS. (2025) MOSIS SCMOS Rules. [Online]. Available: https://www.egr.msu.edu/classes/ece410/demlow/files/DRC_rule_scmos.pdf → [p6]

[24] IHP PDK Authors. (2025) SG13G2 Process Specification Rev. 1.2. [Online]. Available: https://github.com/IHP-GmbH/IHP-Open-PDK/blob/main/ihp-sg13g2/libs.doc/doc/SG13G2_os_process_spec.pdf → [p7], [p8], [p65]

[25] SkyWater PDK Authors. (2025) SkyWater SKY130 Process Design Rules. [Online]. Available: https://skywater-pdk.readthedocs.io/en/main/rules.html → [p8]

[26] P. Scheffler, P. Sauter, T. Benz, F. K. Gürkaynak, and L. Benini, "Basilisk: An end-to-end open-source linux-capable RISC-v SoC in 130nm CMOS." [Online]. Available: http://arxiv.org/abs/2406.15107 → [p8], [p13]

[27] T. Ajayi and D. Blaauw, "Openroad: Toward a self-driving, open-source digital layout implementation tool chain," in *Proceedings of Government Microcircuit Applications and Critical Technology Conference*, 2019. → [p8], [p13]

[28] OpenROAD Contributors. (2025) OpenROAD Documentation. [Online]. Available: https://openroad.readthedocs.io → [p8]

[29] ——. (2025) OpenROAD-flow-scripts Repository. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts → [p9]

[30] LibreLane Contributors. (2025) LibreLane Repository. [Online]. Available: https://github.com/librelane/librelane → [p9]

[31] M. Shalan and T. Edwards, "Building OpenLANE: A 130nm OpenROAD-based tapeout-proven flow," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020. → [p9]

[32] Tim Edwards. (2025) Magic VLSI Layout Tool . [Online]. Available: http://opencircuitdesign.com/magic → [p9]

[33] Matthias Köfferlein. (2025) KLayout project. [Online]. Available: https://www.klayout.de → [p9]

[34] NETGEN Authors. (2025) NETGEN - a general-purpose netlist management system. [Online]. Available: http://opencircuitdesign.com/netgen → [p9], [p53]

[35] cocotb Authors. (2025) cocotb - open source coroutine-based cosimulation testbench. [Online]. Available: https://www.cocotb.org → [p10], [p23]

[36] Icarus Verilog Authors. (2025) Icarus Verilog. [Online]. Available: https://github.com/steveicarus/iverilog → [p10], [p23]

[37] Verilator Authors. (2025) Verilator, the fastest Verilog/SystemVerilog simulator. [Online]. Available: https://www.veripool.org/verilator → [p10], [p24]

[38] T. Scheipel, F. Angermair, and M. Baunach, "moreMCU: A runtime-reconfigurable RISC-v platform for sustainable embedded systems," in *2022 25th Euromicro Conference on Digital System Design (DSD)*, pp. 24–31, ISSN: 2771-2508. [Online]. Available: https://ieeexplore.ieee.org/document/9996937/ → [p11]

[39] Nguyen Cong Dao. (2025) FABulous FPGA on MPW-2. [Online]. Available: https://web.archive.org/web/20241106121050/https://platform.efabless.com/projects/202 → [p12]

[40] ——. (2025) FuseRISC2 on MPW-3. [Online]. Available: https://web.archive.org/web/20241205045913/https://platform.efabless.com/projects/524 → [p12]

[41] ——. (2025) open FABulous eFPGA on MPW-5. [Online]. Available: https://web.archive.org/web/20250210121233/https://platform.efabless.com/projects/769 → [p12], [p13]

[42] A. Traber et al. (2025) IIS Chip Gallery MLEM. [Online]. Available: http://asic.ethz.ch/2024/MLEM.html → [p13]

[43] P. Sauter, T. Benz, P. Scheffler, H. Pochert, L. Wüthrich, M. Povišer, B. Muheim, F. K. Gürkaynak, and L. Benini, "Croc: An end-to-end open-source extensible RISC-v MCU platform to democratize silicon." [Online]. Available: http://arxiv.org/abs/2502.05090 → [p13]

[44] M. Damian, J. Oppermann, C. Spang, and A. Koch, "SCAIE-v: an open-source SCAlable interface for ISA extensions for RISC-v processors," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. Association for Computing Machinery, pp. 169–174. [Online]. Available: https://dl.acm.org/doi/10.1145/3489517.3530432 → [p13]

[45] OpenHW Group. (2025) CORE-V eXtension Interface. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/x_ext.html → [p14]

[46] F. Egert, "FRANCIS-v: FRAmework for iNtegrating custom instructionS into RISC-v systems," accepted: 2023-10-10T10:53:25Z. [Online]. Available: https://repositum.tuwien.at/handle/20.500.12708/188868 → [p14]

[47] Efabless. (2025) EF_QSPI_XIP_CTRL. [Online]. Available: https://github.com/efabless/EF_QSPI_XIP_CTRL → [p17]

[48] ——. (2025) EF_PSRAM_CTRL. [Online]. Available: https://github.com/efabless/EF_PSRAM_CTRL → [p17]

[49] ——. (2025) EF_UART. [Online]. Available: https://github.com/efabless/EF_UART → [p17]

[50] Open HW Group. (2025) OpenBus Interface Specification. [Online]. Available: https://github.com/openhwgroup/obi → [p18], [p21]

[51] PULP Platform. (2025) OBI SystemVerilog IPs. [Online]. Available: https://github.com/pulp-platform/obi → [p18]

[52] OpenHW Group. (2025) OBI2AHB. [Online]. Available: https://github.com/openhwgroup/cve2/tree/main/examples/obi2ahb → [p19]

[53] YosysHQ. (2025) Tabby CAD Datasheet. [Online]. Available: https://www.yosyshq.com/tabby-cad-datasheet → [p19]

[54] sv2v Authors. (2025) sv2v: SystemVerilog to Verilog. [Online]. Available: https://github.com/zachjs/sv2v → [p19]

[55] yosys-slang Contributors. (2025) yosys-slang: SystemVerilog frontend for Yosys. [Online]. Available: https://github.com/povik/yosys-slang → [p19]

[56] Corinna Vinschenm, Jeff Johnston. (2025) Newlib. [Online]. Available: https://sourceware.org/newlib/ → [p23]

[57] xilinx. (2025) Virtex matrix. [Online]. Available: https://www.xilinx.com/publications/matrix/virtexmatrix.pdf → [p26]

[58] Leo Moser. (2025) OpenLane 2 Plugin for FABulous. [Online]. Available: https://github.com/mole99/openlane_plugin_fabulous → [p32]

[59] L. Moser, M. Kissich, T. Scheipel, and M. Baunach, "Stitching FPGA fabrics with FABulous and OpenLane 2," in *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions*, ser. CF '24 Companion. Association for Computing Machinery, pp. 71–74. [Online]. Available: https://dl.acm.org/doi/10.1145/3637543.3652874 → [p32]

[60] M. A. Elgammal, A. Mohaghegh, S. G. Shahrouz, F. Mahmoudi, F. Koşar, K. Talaei, J. Fife, D. Khadivi, K. Murray, A. Boutros, K. B. Kent, J. Goeders, and V. Betz, "VTR 9: Open-source CAD for fabric and beyond FPGA architecture exploration," just Accepted. [Online]. Available: https://dl.acm.org/doi/10.1145/3734798 → [p39]

[61] SERV Authors. (2025) SERV - The SErial RISC-V CPU. [Online]. Available: https://github.com/olofk/serv → [p41]

[62] M. Kissich and M. Baunach, "FazyRV: Closing the gap between 32-bit and bit-serial RISC-v cores with a scalable implementation," in *Proceedings of the 21st ACM International Conference on Computing Frontiers*, ser. CF '24. Association for Computing Machinery, pp. 240–248. [Online]. Available: https://dl.acm.org/doi/10.1145/3649153.3649195 → [p41]

[63] OpenROAD Authors. (2025) Chip-level Connections in OpenROAD - pad. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenROAD/tree/master/src/pad → [p43]

[64] K. L. Chung, N. Dao, J. Yu, and D. Koch, "How to shrink my FPGAs — optimizing tile interfaces and the configuration logic in FABulous FPGA fabrics," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. Association for Computing Machinery, pp. 13–23. [Online]. Available: https://dl.acm.org/doi/10.1145/3490422.3502371 → [p58]

List of Figures

List of Tables

List of Listings

List of Abbreviations

AHBL	Advanced High-performance Bus Lite
ALU	Arithmetic Logic Unit
AOI	AND-OR-Invert
ASIC	Application-Specific Integrated Circuit
BEL	Basic Element
BRAM	Block Random-Access Memory
BSP	Board Support Package
CI	Continuous Integration
CLB	Configurable Logic Block
CSR	Control and Status Register
CTS	Clock Tree Synthesis
DEF	Design Exchange Format
DRAM	Dynamic RAM
DRC	Design Rule Check
EDA	Electronic Design Automation
eFPGA	embedded Field-Programmable Gate Array
FF	Flip-Flop
FIFO	First In First Out
FLI	Foreign Language Interface
FOSSi	Free and Open Source Silicon
FPGA	Field-Programmable Gate Array
GDSII	Graphic Design System II
GL	Gate Level
GPI	Generic Procedural Interface
HDL	Hardware Description Language
IC	Integrated Circuit
IP	Intellectual Property
IRQ	Interrupt Request
ISA	Instruction Set Architecture

LC Logic Cell
LEF Library Exchange Format
LUT Look-Up Table
LVS Layout Versus Schematic

MAC Multiply-Accumulate
MPW Multi-Project Wafer
MUX Multiplexer

NDA Non-Disclosure Agreement

OAI OR-AND-Invert
OBI OpenBus Interface
OS Operating System

PCB Printed Circuit Board
PCell Parametrizable Cell
PDK Process Design Kit
PDN Power Distribution Network
PIP Programmable Interconnect Point
PMA Physical Memory Attribution
PnR Place & Route
PR Pull Request
PSRAM Pseudostatic RAM

QSPI Quad Serial Peripheral Interface

RQ Research Question
RTL Register Transfer Level

SCL Standard Cell Library
SoC System on Chip
SPEF Standard Parasitic Exchange Format
SPI Serial Peripheral Interface
SRAM Static Random-Access Memory
STA Static Timing Analysis

TiN Titanium Nitride

UART Universal Asynchronous Receiver Transmitter

VHPI VHDL Procedural Interface
VLSI Very Large Scale Integration
VPI Verilog Procedural Interface

XIF Custom Instruction Interface
XiP eXecute in Place

SEE YOU SPACE COWBOY...