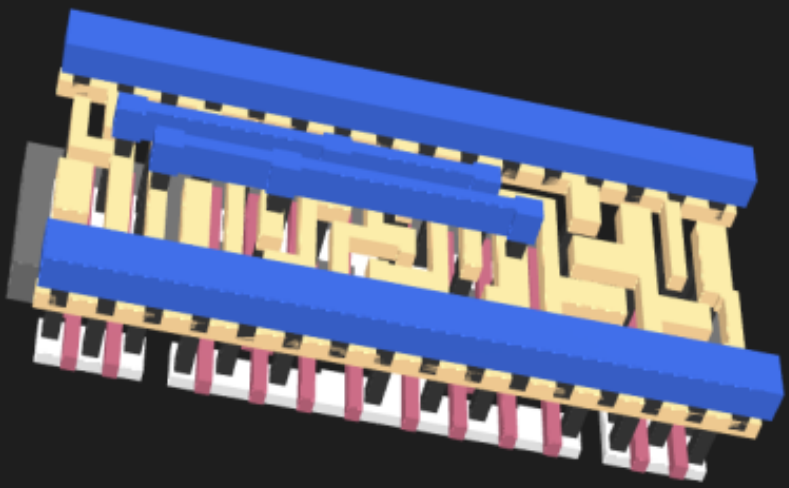
Introduction to Chip Design

Using Open-Source Tools



Martin Schoeberl

Introduction to Chip Design

Using Open-Source Tools

First Edition

Introduction to Chip Design

Using Open-Source Tools

First Edition

Martin et al.

Email: martin@jopdesign.com
Visit the source at https://github.com/os-chip-design/chip-design-book

First edition published 2026 by Kindle Direct Publishing,
https://kdp.amazon.com/

Library of Congress Cataloging-in-Publication Data

TBD... Schoeberl, Martin

xxxl
Martin ...
Includes bibliographical references and an index.
ISBN xxx

Manufactured in the United States of America.
Typeset by Martin Schoeberl.

Contents

List of Figures

List of Tables

Listings

Foreword

It is an exciting time to be in the world of chip design....

Preface

This book is an introduction to chip design with a focus on using open-source tools and open-source PDKs.

I have not used any large language model (LLM) to write even a single sentence. All mistakes are mine, not a halizunisation by an LLM. I use Grammarly for grammar checking and Copilot for writing code.

Acknowledgements

1 Introduction

This book is an introduction to chip design using open-source tools. It covers the steps needed to produce a chip for a design described in a hardware description language (HDL), down to the files that are sent to the fab. Those steps are often called the backend design.

This book does not cover digital design, such as basic Boolean equations and sequential circuits. For this topic, we refer to other textbooks, such as [4] and [10]. When we need to describe some circuits, we will describe them in Chisel [2] and in Verilog.

This book is optimized for reading on a tablet (e.g., an iPad) or a laptop. We include links to further reading in the running text, primarily to Wikipedia articles.

1.1 History of Design Tools

Very early chips were designed by hand, and the photo masks were produced by *drawing* them with tape. As this process does not scale, computer-aided design (CAD) tools have been developed. Another term used is: electronic design automation (EDA).

Alberto Sangiovanni-Vincentelli was invited to give a keynote speech at the 40th Design Automation Conference (DAC). That speech resulted in a paper on the history (and future) of EDA [8], mostly work as presented at DAC.

Alberto co-founded Cadence Design Systems and Synopsys, the two major EDA tool vendors.

Read https://arxiv.org/pdf/2311.02055

1.1.1 Simulation

IBM's Astap (advanced statistical analysis program) and UC Berkeley's Spice (simulation program for integrated circuits emphasis)

Early tools, still in use today: Spice, Espresso, Magic,

1.2 Tool Installation

The magic of open-source tools is that we can install them on a personal computer without any concerns of licensing and license servers. The downside of open-source tools is the variety of installation options. There is no single best way for the installation. Furthermore, open-source tools are best supported on a Unix operating system, and preferably Linux. Support under macOS comes second. MS Windows is best served by using the Windows Subsystem for Linux.

1.2.1 Nix Based

From OpenLane2 (now LibreLane) on, the preferred installation is Nix-based. Nix is a cross-platform package manager that also provides native binaries for x86 and ARM-based systems (e.g., Mac laptops). The LibreLane installation documentation explains how to install Nix, and also how to set up the Nix cache for the LibreLane tools.

Another option to install Nix is via Determinate. Note that you need to install the LibreLane cache to avoid building all the tools yourself.

1.2.2 Docker Images

Another option is to use Docker containers with ready-installed tools. Harald Prettl, from the Johannes Kepler University of Linz, provides IIC-OSIC-TOOLS (Integrated Infrastructure for Collaborative Open Source IC Tools), a Docker container based on Ubuntu 24.04 LTS for the following CPU architectures: x86_64/amd64 and aarch64/arm64.

The folder `/foss/designs` is the place to access user data on your local machine. It points to the directory pointed by the environment variable `DESIGNS`, the default is `$HOME/eda/designs`. To change this to start with your home folder, invoke the Docker image with:

```
DESIGNS=$HOME ./start_x.sh
```

Note that, as of the time of writing, the default PDK is the IHP PDK, which is not yet supported in the LibreLane flow from within the container.[1] Switch to the Sky130 PDK with the following command:

```
sak-pdk sky130A
```

[1] See: https://github.com/iic-jku/IIC-OSIC-TOOLS/issues/147

1.2.3 Compiling from Source

As the tools are open-source, it is always a (theoretical) option to compile them from source. However, often there are library dependencies that are not easy to resolve. Therefore, we recommend installing the tools with some packaging software that ensures compatibility between the various tools.

1.2.4 Installing LibreLane

When you use the `nix`-based setup, you need to install LibreLane. Install LibreLane by cloning it:

```
git clone git@github.com:librelane/librelane.git
```

You set up your environment by entering `nix` as follows:

```
cd librelane
nix-shell
```

The frontend command is `librelane`. Check the version you have installed with:

```
librelane --version
```

LibreLane also installs the Sky130 PDK using the `ciel` tool. The PDKs are stored under `$HOME:/.ciel`.

1.2.5 Further Packages

There are several other distributions of the open-source chip design tools available:

OSS CAD Suite is a collection of tools with a focus on open-source design flow for FPGAs.

YoWASP (Yosys WebAssembly Synthesis & PnR) is a collection of tools targeting FPGA design flow compiled to WebAssembly. Therefore, they can be easily run on different platforms, even as a Visual Studio Code plugin.

```
import librelane
print(librelane.__version__)
```

Listing 1.1: Python script to show the version of the installed LibreLane (version.py).

1.3 Hello World

After installing the tools, we want to explore them in action. The *Hello World* version for the backend chip design is to synthesize a minimal circuit from the HDL description to generate a GDSII file and visualize it in an editor. Note that all code examples in this book are part of the book's GitHub repository.

Execute 'nix-shell' in your LibreLane installation. Try to check your LibreLane version by invoking Python on the code shown in Listing 1.1.

Then, switch to a working directory and create the following small Verilog design (`adder.v`), as shown in Listing 1.3. This example is an adder, including registers at the input and output ports, so we can use static timing analysis (STA) to explore the maximum clocking frequency.

Furthermore, you need a configuration for your flow. You can find all possible configuration parameters at the Universal Flow Configuration Variables section of the LibreLane documentation. However, most variables can be left at their default values.

Config files can support more than one PDK, see the LibreLane example. Configuration can be in YAML or JSON format. We will use the `adder.yaml` file with a minimal configuration, as shown in Listing 1.5.

For a fully automatic run of the full flow, you can execute:

```
librelane adder.yaml
```

This single command runs the complete synthesis flow from RTL to GDSII. As the design is very simple, the flow will run in about a minute. Figure 1.1 shows the final GDSII file for the adder in KLayout.

A flow takes, at the time of this writing, with LibreLane v2.4.8, 74 steps, each reporting in its folder. During the flow, a lot of information is printed out. Scrolling back in the terminal, you can see reports of design checks and resource usage. Every time you run the flow, it will create a new folder under folder `runs`. The name of the folder will be `RUN_year_month_day_hh-mm-ss`.

```
module adder (
    input clock,
    input  [7:0] a,
    input  [7:0] b,
    output [7:0] sum
);

    reg [7:0] reg_a, reg_b, reg_sum;

    always @(posedge clock) begin
        reg_a <= a;
        reg_b <= b;
        reg_sum <= reg_a + reg_b;
    end;

    assign sum = reg_sum;

endmodule
```

Listing 1.2: A pipelined adder as a Hello World example for LibreLane (adder.v).

```
DESIGN_NAME: adder
VERILOG_FILES: dir::adder.v
CLOCK_PERIOD: 20
CLOCK_PORT: clock
```

Listing 1.3: The YAML configuration file (adder.yaml).

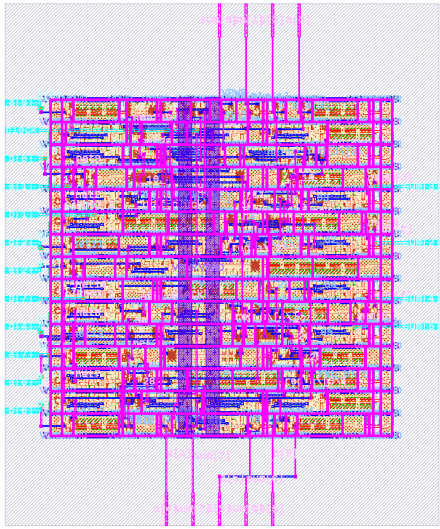


Figure 1.1: The synthesized adder as visualized in KLayout.

1.3.1 Exploring the Design

A first step to explore the design is to view the GDS in KLayout with:

```
librelane --last-run --flow openinklayout adder.yaml
```

In Klayout, you can explore the design. Use the rulers to measure your design. How large is it? Would it fit into a Tiny Tapeout Tile? How large is a Tiny Tapeout tile? When you expand the adder in Cells, you can see which cells have been used. By double-clicking on a cell type, it is removed from the display. This is an indirect way to show the cells in the GDS. The flip-flop standard cells in Sky130 have names that include “dfxtp” in their name. Double-click that cell to see how many flip-flops are used. How many should there be?

Another option is to open the design with the OpenROAD viewer:

```
librelane --last-run --flow openinopenroad adder.yaml
```

Matt Venn has a good selection of layers and color assignments at klayout_gds.xml. You can use it with klayout as follows:

```
klayout -l klayout_gds
```

That configuration file is part of the summary tools, which can be used for a quick exploration of the design. Clone the repository, add it to your path, and set the `PDK_ROOT` variable (easiest into a file that you source, or add those lines to your `$HOME/.bashrc`):

```
export PATH=$PATH:$HOME/path/to/librelane_summary
export PDK_ROOT=$HOME/.ciel
```

Explore the tool with:

```
summary.py --help
```

When in a folder that contains the LibreLane runs, explore the GDS of the latest run with:

```
summary.py --gds
```

At the time of this writing, the flow generates 74 folders containing the results and reports for each individual step. You can find those subfolders in the folder `runs/RUN_DATE_TIME`. Not all reports are equally important. We will explore some of them.

Linting and a First Area Estimate

The first four steps perform linting of the design, and any Verilog syntax errors are identified during this process. An initial estimate of the used cells can be found in the report `06-yosys-synthesis/reports/stat.rpt`. For our pipelined adder, we can identify the 24 flip-flops as `dfxtp_2` cells in that report:

```
...
   Number of cells:                 60
     sky130_fd_sc_hd__a31oi_2        1
     sky130_fd_sc_hd__and2_2         4
     sky130_fd_sc_hd__dfxtp_2       24
...
```

We use the Sky130 PDK and can find the documentation of each cell online. E.g., the DFF is described at dfxtp. The file `adder.nl.v` contains the design synthesized to standard cells from the used PDK. As our example is very simple, we can manually inspect the generated Verilog.

Static Timing Analysis

In folder `nn-openroad-stapostpnr`, the static timing analysis (STA) reports are found.

Size

When not constraining the chip's size, LibreLane will choose the size.

Design Rule Checks

Summary

The subfolder `final` contains a summary of the project in files `metrics.csv` and `metrics.json`. It includes timing information, size of the design, number of standard cells used, and other detailed metrics.

1.3.2 Change the Design

The current example has non-initialized registers after power-up. Although those random values will be cleared with valid values after on and two clock cycles, it is good practise to reset registers. This expcially useful for verification.

There are two ways flip-flops can be reset: (1) with an asynchronous reset or (2) with a synchronous reset. Explain which one is to be preferred and why. Which one needs a larger area? This question is not easy to explain by a back-of-the-envelope calculation when we do (yet) not know the available standard cells and their sizes.

However, we can easily decide this question by exploring both versions with SkyWater130. Change the design to reset all registers. Run both versions through the flow. Then explore the size and which flip-flop types have been used.

1.4 Manual Flow with Python

However, we can also run the individual steps from Python. We will reuse the pipelined adder for this exploration from Section 1.3. The following example is inspired by running LibreLane on Google Colab, a nice way to explore LibreLane just from your browser.

All code from this book can be found in the GitHub repo of the book: code.

Start Python from within your Nix shell,

```
import ciel
from ciel.source import StaticWebDataSource
from librelane.common import get_opdks_rev, ScopedFile

ciel.enable(
    ciel.get_ciel_home(),
    "sky130",
    get_opdks_rev(),
    data_source =
        StaticWebDataSource("https://fossi-foundation.github.io/ciel-rel
)
```

Listing 1.4: Setting up the PDK (pdk.py).

```
python
```

and run the following commands. Import LibreLane and print its version.

```
import librelane
print(librelane.__version__)
```

1.4.1 Configuration

To run our steps we need to configure the PDK via `ciel`[2] and configure our project.

Execute the code from Listing 1.4 to setup the PDK. If you have downloaded the code from the book, you can execute the script from within Python with the following command:

```
exec(open("pdk.py").read())
```

A LibreLane `Flow` needs to be configured. Execute Listing 1.5 to configure your project. The configuration is similar to the YAML file we used in the initial example.

The results of the following steps will be placed into the folder `librelane_run`.

1.4.2 Synthesis with Yosys

The design is split into so-called *steps*. Therefore, we need to import `Step` and

```
from librelane.config import Config

Config.interactive(
    "adder",
    PDK = "sky130A",
    CLOCK_PORT = "clock",
    CLOCK_NET = "clock",
    CLOCK_PERIOD = 20,
    PRIMARY_GDSII_STREAMOUT_TOOL = "klayout",
)
```

Listing 1.5: Configure the project (config.py).

```
from librelane.steps import Step
from librelane.state import State

initial_state = State()

Synthesis = Step.factory.get("Yosys.Synthesis")
Synthesis.display_help()
```

Listing 1.6: Get started with the steps (steps.py).

```
synthesis = Synthesis(state_in=initial_state,
    VERILOG_FILES=["adder.v"])
synthesis.start()
```

Listing 1.7: Running the synthesis (synth.py).

```
Floorplan = Step.factory.get("OpenROAD.Floorplan")

floorplan = Floorplan(state_in=synthesis.state_out)
floorplan.start()
```

Listing 1.8: Running the floorplanning (`floor.py`).

`State`, as shown in Listing 1.6). The last command shows us help with the steps.

We start with the Yosys synthesis. Yosys converts high-level Verilog to a low-level Verilog that includes instances of standard cells. We pass the Verilog files to `Yosys.Synthesis`. We can run the synthesis with the code from Listing 1.7.

Explore the reports and the generated low-level Verilog in folder `1-yosys-synthesis`.

1.4.3 Floorplanning

The next step is floorplanning. It determines the size of the chip and generates a cell placement grid. Each cell in the grid is called a *site*. A standard cell usually occupies multiple sites by width (e.g., a D-FF has a larger width than a NAND gate). We run the floorplanning with the code from Listing 1.8. Explore the reports in folder `2-openroad-floorplan`.

1.4.4 Tap/Endcap Cell Insertion

This step places endcap cells and tap cells. Endcap cells are placed at the beginning and end of each row. They terminate each power line.

The tap cells connect VDD to the n-well and GND to the p-substrate. To save area, the standard cells do not connect to the tap themself. The maximum distance between tap cells is dependent on the foundry process. Run the tap placements with the code from Listing 1.9.

To view the result, start the OpenRoad GUI with:

```
openroad -gui
```

and then open `adder.odb` from folder `3-openroad-tapendcapinsertion`.

[2] `ciel` is a PDK managemant tool, part of LibreLane.

```
TapEndcapInsertion =
    Step.factory.get("OpenROAD.TapEndcapInsertion")

tdi = TapEndcapInsertion(state_in=floorplan.state_out)
tdi.start()
```

Listing 1.9: Running the tap cell insertation (tap.py).

```
IOPlacement = Step.factory.get("OpenROAD.IOPlacement")

ioplace = IOPlacement(state_in=tdi.state_out)
ioplace.start()
```

Listing 1.10: Running the I/O placement (io.py).

1.4.5 I/O Placement

Metal pins are placed at the edges of the design for the top-level inputs and outputs. The default placement can be overridden by a placement configuration.

Run the tap placements with the code from Listing 1.10 and explore the result with the OpenRoad GUI.

1.4.6 Power Distribution Network (PDN)

The next step is to generate the power distribution network (PDN). It is a pattern of vertical and horizontal straps. The vertical straps are thicker and distribute power to the horizontal straps. The horizontal straps are delivering power to the individual standard cells. More details of the configuration of the PDN can be found in the LibreLane documentation.

Run the PDN generation with the code from Listing 1.11 and explore the result with the OpenRoad GUI. Figure 1.2 shows the PDN of our small adder example.

1.4.7 Global Placement

Global placement places the standard cells, aiming to minimize the distance between connected cells. The cells are not exactly placed, as we can see in Figure 1.3.

```
GeneratePDN = Step.factory.get("OpenROAD.GeneratePDN")

pdn = GeneratePDN(
    state_in=ioplace.state_out
)
pdn.start()
```

Listing 1.11: Running PDN generation (pdn.py).

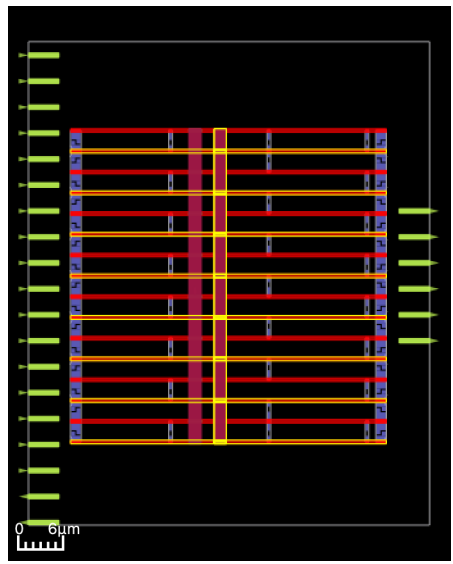


Figure 1.2: Chip layout with pins and power distribution network.

```
GlobalPlacement =
    Step.factory.get("OpenROAD.GlobalPlacement")

gpl = GlobalPlacement(state_in=pdn.state_out)
gpl.start()
```

Listing 1.12: Global placement (glp.py).

```
DetailedPlacement =
    Step.factory.get("OpenROAD.DetailedPlacement")

dpl = DetailedPlacement(state_in=gpl.state_out)
dpl.start()
```

Listing 1.13: Local placement (dlp.py).

The placement is even *illegal* as it is not properly aligned with the grid. This is fixed in the next step of detailed placement, also called placement legalization.

Run the global placement with the code from Listing 1.12 and explore the result with the OpenRoad GUI.

1.4.8 Detailed Placement

Detailed placement aligns the cells with the grid; we call this also *legalizing* it. Run the detailed placement with the code from Listing 1.13 and explore the result with the OpenRoad GUI. Figure 1.4 shows the now well-aligned cells.

1.4.9 Clock Tree Synthesis (CTS)

After placing all cells, we create the clock tree, including buffers for the clock. Run the detailed placement with the code from Listing 1.14.

1.4.10 Global and Detailed Routing

As the result of global routing is stored in an internal data structure, we see no effect on the actual design. Therefore, we combine global routing and local routing

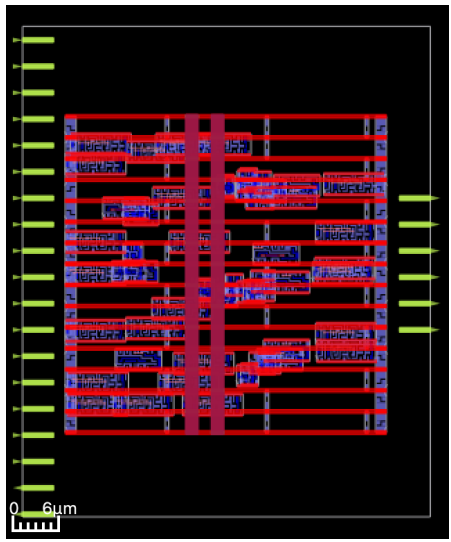


Figure 1.3: Global placement.

```
CTS = Step.factory.get("OpenROAD.CTS")

cts = CTS(state_in=dpl.state_out)
cts.start()
```

Listing 1.14: Local placement (cts.py).

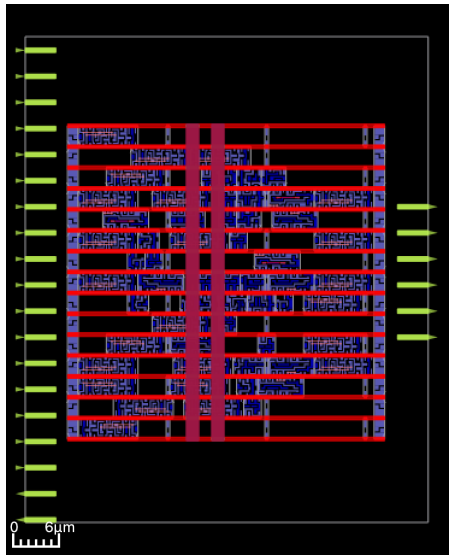


Figure 1.4: Legalized placement

```
GlobalRouting = Step.factory.get("OpenROAD.GlobalRouting")

grt = GlobalRouting(state_in=cts.state_out)
grt.start()

DetailedRouting =
    Step.factory.get("OpenROAD.DetailedRouting")

drt = DetailedRouting(state_in=grt.state_out)
drt.start()
```

Listing 1.15: Global and local routing (route.py).

```
FillInsertion = Step.factory.get("OpenROAD.FillInsertion")

fill = FillInsertion(state_in=drt.state_out)
fill.start()
```

Listing 1.16: Fill insertion (fill.py).

in a single Python script to be executed (Listing 1.15). Figure 1.4 shows the routed design.

1.4.11 Fill Insertion

Execute Listing 1.16 and explore the output with the OpenRoad GUI.

1.4.12 Resistance/Capacitance Extraction (RCX)

Extracting resistance and capacitance as preparation for timing analysis.
Execute Listing 1.17.

1.4.13 Static Timing Analysis (STA)

Execute Listing 1.18.

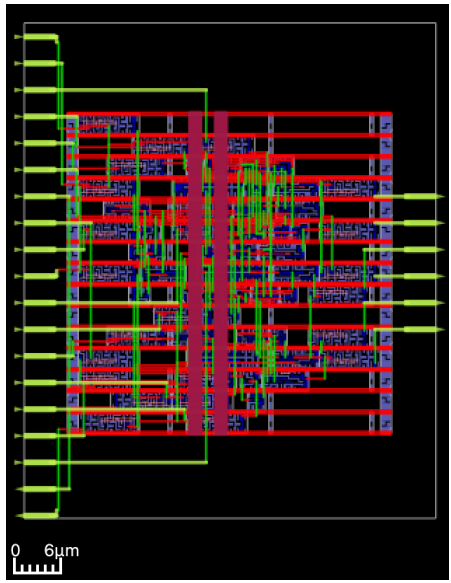


Figure 1.5: The routed design.

```
RCX = Step.factory.get("OpenROAD.RCX")

rcx = RCX(state_in=fill.state_out)
rcx.start()
```

Listing 1.17: Fill insertion (rcx.py).

```
STAPostPNR = Step.factory.get("OpenROAD.STAPostPNR")

sta_post_pnr = STAPostPNR(state_in=rcx.state_out)
sta_post_pnr.start()
```

Listing 1.18: Static timing analysis (sta.py).

```
StreamOut = Step.factory.get("KLayout.StreamOut")

gds = StreamOut(state_in=sta_post_pnr.state_out)
gds.start()
```

Listing 1.19: Stream out the GDS file (gds.py).

```
DRC = Step.factory.get("Magic.DRC")

drc = DRC(state_in=gds.state_out)
drc.start()
```

Listing 1.20: Design rule checks (drc.py).

1.4.14 Generating the GDSII

Finally, we can stream out the GDSII of our design to ship for manufacturing.

Execute Listing 1.19. It is now again time to explore the final result with KLayout

1.4.15 Design Rule Checks (DRC)

Execute Listing 1.20 for the design rules check.

1.4.16 SPICE Extraction

Execute Listing 1.21 for the spice extraction.

```
SpiceExtraction = Step.factory.get("Magic.SpiceExtraction")

spx = SpiceExtraction(state_in=drc.state_out)
spx.start()
```

Listing 1.21: Spice extraction (spice.py).

```
LVS = Step.factory.get("Netgen.LVS")

lvs = LVS(state_in=spx.state_out)
lvs.start()
```

Listing 1.22: Spice extraction (lvs.py).

1.4.17 Layout vs. Schematic (LVS)

Execute Listing 1.22 for the layout vs. schematic check.

1.5 Notes

Collect notes here for the flow.

1.5.1 Tuning the Synthesis Flow

LibreLane has several variables that can be set to tweak the flow. .

```
"//": "Hold slack margin - Increase them in case you are getting hold violations.
"PL_RESIZER_HOLD_SLACK_MARGIN": 0.1,
"GRT_RESIZER_HOLD_SLACK_MARGIN": 0.05,
```

2 Open-Source Production Frameworks

Chip design consists of several steps from synthesis down to GDSII files. Scripts usually orchestrate those steps. In the following, we describe different frameworks that contain the needed open-source tool, but also the scripts to run the flow.

2.1 OpenROAD

OpenROAD [1] started as a DARPA-sponsored project to enable an end-to-end design flow of chips from RTL to the final chips without human intervention, within a maximum of 24 hours. The documentation is published on Read the Docs. OpenROAD includes the following open-source tools:

- ifp defines the core area, the rows, and the tracks.
- pdn is a power distribution network (PDN) generator.
- Tapcell inserts tapcells or endcaps.
- gpl performs global placement. The tool is an extension of the RePlAce tool.
- Gate Resizer to optimize the design until the maximum utilization is reached.
- OpenDP performs detailed placements.
- TritonCTS 2.0 is performs clock tree synthesis.
- FastRoute performs global routing.
- TritonRoute performs detailed routing.
- OpenRCX performs parasitics extraction.
- And some more.

Use the links for documentation of the individual tools.

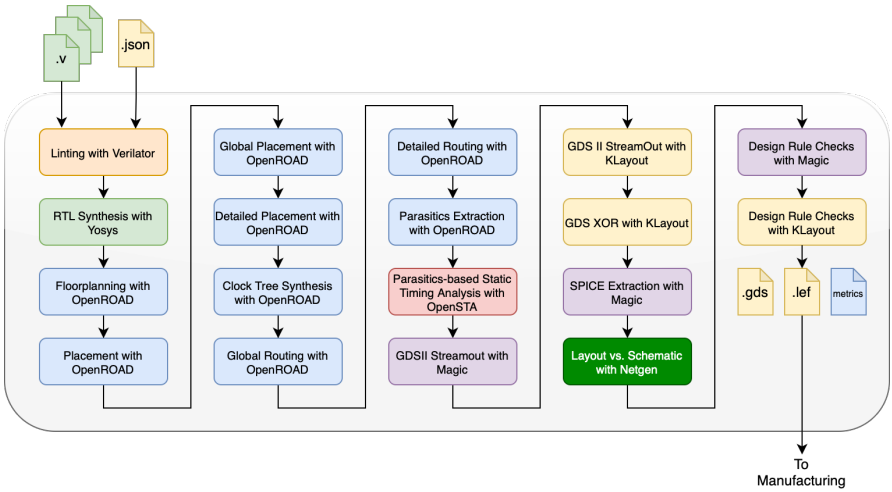


Figure 2.1: OpenLane design flow, including the OpenROAD flow in blue. Copyright 2020-2022 Efabless Corporation and contributors, License: Apache 2.0.

2.2 OpenLane

OpenLane [5, 13] is a collection of EDA tools, such as Yosys, Magic, Netgen, and KLayout. It also includes OpenROAD. OpenLane consists of TCL scripts to automate the flow from RTL, described in Verilog, to the chip production data as a GDSII file.

OpenLane was developed by eFabless, and the documentation is published on Read the Docs.

2.2.1 The Design Flow

Figure 2.1 shows the design flow of OpenLane.

2.2.2 OpenLane2 and LibreLane

OpenLane2 is a new, partly rewritten,[1] framework for chip design with a focus on substituting TCL scripts with Python scripts. The documentation is published on Read the Docs. However, the original OpenLane documentation is richer than the OpenLane2 documentation. Therefore, consider reading that one as a basis.

Efabless, the original developer of OpenLane and OpenLane2, exited the business in spring 2024. To continue the development of OpenLane, the repository was cloned/forked in April 2025, and the project was renamed to LibreLane.

2.2.3 Running the Flow *Manually*

2.3 Caravel

Caravel is a SoC framework developed by Efabless for the Google/Skywater 130nm Open PDK. It is the basis for the chipIgnite MPW shuttles. It contains a padframe, a housekeeping block, a management area including a RISC-V core, and a user project wrapper. The user area is 3000 µm x 3600 µm (10 mm^2). To put this into perspective, a simple 3-stage RISC-V pipeline, such as Wildcat [12], fits into a 0.2 mm^2 area. A 4 KiB memory from eFabless has a 0.18 mm^2 area.

ChipFoundry now maintains Caravel. Note that the transition to ChipFoundry is still in progress; therefore, some documentation still contains the eFabless name. Additionally, some datasheets are outdated (e.g., referring to the PicoRV as the management processor, which is now the VexRiscv).

Caravel is, in fact, two artifacts that are combined during the MPW chip production. The Caravel harness, which combines the RISC-V CPU and IO management with the user project, is used to build the final Caravel tile. Figure 2.2 shows this combination of the two components to the final Caravel. The so-called `user_project_wrapper` is the design that is submitted to ChipFoundry. As users, we do not need to directly deal with the Caravel harness.

2.3.1 Caravel Harness

Figure 2.3 shows a simplified block diagram of the Caravel harness. The Caravel harness contains the management core, the user project, the padframe, and GPIOs configured by an SPI controller.

[1] OpenLane2 started out of OpenLane and OpenROAD scripts

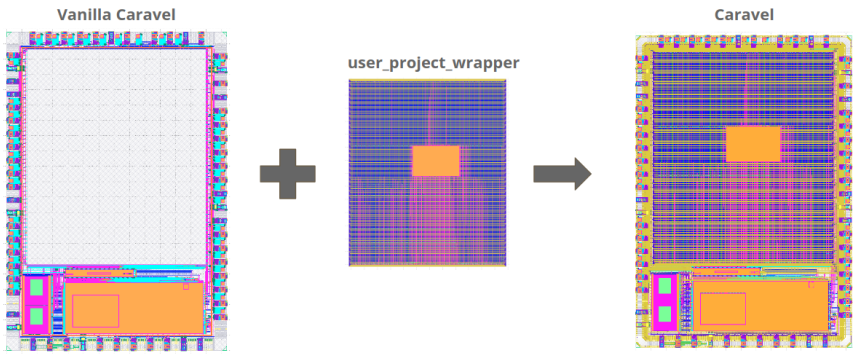


Figure 2.2: The combination of the Caravel harness with the user project results in the final Caravel tile for the MPW run. Copyright 2020-2022 Efabless Corporation and contributors, License: Apache 2.0.

The management core contains a VexRiscv RISC-V core, 256-word memory (latch-based), shared for instruction and data, and peripherals, connected via a Wishbone bus. The peripherals include a flash controller, a UART, an SPI master, and a logic analyzer. The Wishbone bus and the logic analyzer are connected to the user area. The management core is built with the LiteX framework, which itself uses Migen, a Python project to build digital hardware.

The RISC-V core in the management runs firmware out of the Flash to configure the user project GPIOs, interact with the user area via the Wishbone bus, and interact with the logic analyzer.

The user project is connected to the GPIO pins, the Wishbone bus of the RISC-V CPU, the logic analyzer, and interrupt pins to the RISC-V CPU. The example components are split into a wrapper and the *real* component. For example, the user project is split into a `user_project_wrapper` and a `user_proj_example`.

The chipIgnite project includes a board containing the Caravel chip, a power supply, a serial Flash for the firmware, and an FTDI FT232 for communication between a PC via USB.

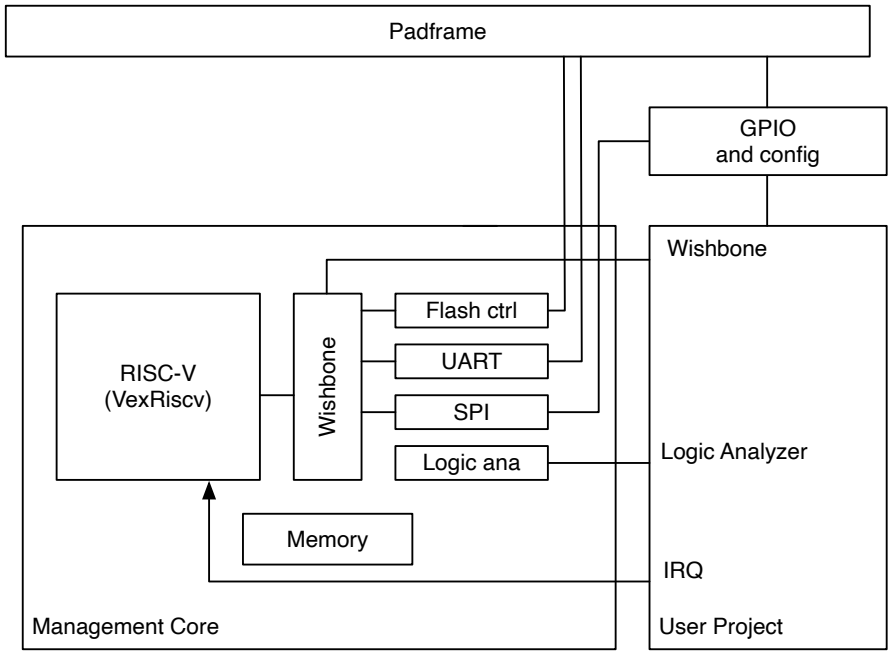


Figure 2.3: The Caravel Harness

2.3.2 Caravel User Project

The Caravel framework itself is split into two repositories: the user project (often called `user_project_wrapper`, as this is the top-level of the user project) and Caravel itself. The example user project contains a tiny example (a counter) to get started. Within that project, all the needed tools are installed, including a lite version of Caravel, the management core for simulation, LibreLane, and the Sky130 PDK.

The starting point is the user project. The flow is run within Docker. Follow the instructions in the Caravel User Project. The dependencies on Docker and Python are listed there. Missing is the dependency on two Python libraries: `python-tk` and `Click`.

There are two ways to integrate your design into Caravel: (1) Harden your user project first and then instantiate it in the wrapper as a hard macro, and then run the hardening of the wrapper; (2) Harden the whole project at once (include your user project as Verilog files in the wrapper configuration). The example project performs the individual hardening.

Following video from ChipFoundry gives an overview of the flow of the example project. Then follow the instructions in the Caravel User Project.

2.3.3 Setup

Start by cloning the user project by selecting *Use this template*. Install the prerequisites and clone your repository. Note that the project requires several GB of space on your hard disc. The initial clone is 1/4 GB. The needed tools are installed with:

```
make setup
```

During this setup phase, a lightweight version of Caravel (`caravel_lite`), the management core for simulation, the Sky130 PDK, a Docker with LibreLane, and timing scripts are downloaded.

The download will take several minutes and needs around 7.5 GB of hard disk space.

2.3.4 Hardening the User Project

The example project consists of a simple 16-bit counter that is connected to the Wishbone bus, the logic analyzer, and 16 output pins. With the Wishbone connection, the counter can be controlled (i.e., started and stopped) from the management core. The source can be found in `verilog/rtl/user_proj_example.v`.

To get started, we recommend hardening and testing the user project as it is. After you have successfully hardened and tested the example, add your modules to the project. The default configuration is set up to harden the user project first and include the GDS as a hard macro in the wrapper project.

The user project contains two projects in folder `openlane`:

1. `user_proj_example` and
2. `user_project_wrapper`

To simplify the hardening, it is recommended to keep those names.[2] Both folders contain a `config.json` for the LibreLane run. Explore the two configuration files. You will see that the die area for the user project is set to 2800 µm x 1760 µm, which is smaller than the available 10 mm2 on the Caravel project. Change this setting later to fit your design. In the configuration file of the wrapper, we can see that the user project is instantiated as a hard macro at position `[60, 15]`.

The user project is hardened with:

```
make user_proj_example
```

That command starts the hardening of the user project with LibreLane, generating the GDS file and a gate-level netlist with instances of the Sky130 standard cells for gate-level simulation. That GDS can be found in folder `gds`. Use e.g., KLayout to explore that hard macro. The gate-level netlist in folder `verilog/gl`. The LibreLane reports and generated files can be found in the folder `openlane/user_proj_example/runs`.

The project now needs 10 GB of disk space. Note that many of the generated files have already been committed to the original git repository, which creates annoying false source changes. Avoid committing those changes after every run.

The default configuration uses Docker to run the LibreLane tools. Depending on your setup (processor and operating system), running with Docker can be significantly slower than running the tools natively. For example, the user project required 2 hours to harden on my Mac with an M1. Most of the time was spent on magic spice extraction (1h30). When running it natively using Nix, the whole run executes in 15'. You can select Nix with:

```
make user_proj_example LIBRELANE_USE_NIX=1
```

The next step is to harden the project wrapper with:

[2] Although a name that includes a word such as *example* sounds a bit silly, most projects keep the names.

```
make user_project_wrapper
```

This wrapper includes the user project as a hard macro. Similar to the user project, the run generates the GDS file and the Verilog file for gate-level simulation. The GDS file is the one that is submitted to the fab for production. The project has expanded to 11 GB.

GPIO Configuration

Finally, before we can run a precheck, we need to update the default configuration for the GPIO pins in `verilog/rtl/user_defines.v`. GPIO[0] to GPIO[4] are used by the management core and cannot be changed. The example design uses 16 output pins. Therefore, we set them to OUTPUT and all other pins, which are not used as an INPUT, as follows:

```
...
`define USER_CONFIG_GPIO_5_INIT  `GPIO_MODE_USER_STD_OUTPUT
`define USER_CONFIG_GPIO_6_INIT  `GPIO_MODE_USER_STD_OUTPUT
`define USER_CONFIG_GPIO_7_INIT  `GPIO_MODE_USER_STD_OUTPUT
`define USER_CONFIG_GPIO_8_INIT  `GPIO_MODE_USER_STD_INPUT_NOPULL
`define USER_CONFIG_GPIO_9_INIT  `GPIO_MODE_USER_STD_INPUT_NOPULL
`define USER_CONFIG_GPIO_10_INIT `GPIO_MODE_USER_STD_INPUT_NOPULL
...
```

Tim Edwards wrote on the FOSSI chat:

> The user project example is certainly not using best practices. The way I designed the caravel harness (I did not design the user project example), pins 1 to 4 are used by the housekeeping SPI to communicate with the host computer through the FTDI chip and USB. So it is generally "occupied" and becomes a bit difficult to make use of for a user project. It can be done (contrary to what the instructions might say): The purpose of the harness chip design was to give the user project the maximum number of pins to use, if needed. Best practice (again, not the user example) is to leave the lower five pins (or seven, if you count the UART) alone unless you run out of pins and absolutely need them. It is always possible to run a program off of the flash that reconfigures all of the pins, including the lower ones, to whatever you want. Reconfiguring the pins that are connected to the FTDI will result in the inability to communicate with the chip without doing a complicated power cycle

and simultaneous reset. So that's why it's not recommended: It will make working with the chip very annoying. But it is neither prohibited, nor impossible.

For the pin assignment, see also the schematic of the board that comes with the ChipFoundry offer.

2.3.5 Testing

The Caravel project uses cocotb, a Python-based testing framework for testing the RTL of the design and also the gate-level netlist. The tests with cocotb run in their own Docker image. Therefore, on the first run, the Docker image will be downloaded. Note that this Docker image is an x86 image, which will run slowly on a Mac ARM.

```
make cocotb-verify-all-rtl
```

You can run a single test, e.g., the `counter_wb` test with:

```
make cocotb-verify-counter_wb-rtl
```

Note that pulling the Docker image might fail. In that case, do it manually:

```
docker pull docker.io/efabless/dv:cocotb
```

The tests can be found on `verilog/dv/cocotb`. The above `make` target runs four tests for the counter user project. Running the tests generates a `sim` folder and timestamped folders for each run. Within this folder, you find `runs.log` summarizing the test runs.

```
pip install caravel-cocotb
```

https://pypi.org/project/caravel-cocotb/

2.3.6 Timing Analysis

Extract parasitics for `user_project_wrapper` and its macros, create the spef file, and finally run the timing analysis with OpenSTA.

```
make extract-parasitics
make create-spef-mapping
make caravel-sta
```

2.3.7 Precheck

Before submitting your design to the fab (i.e., to ChipFoundry), it is highly recommended to run the precheck.

```
make precheck
```

Clones the `mpw_precheck` repository in the home folder, which is the default location. The installation location can be overwritten by setting `PRECHECK_ROOT`.

```
make run-precheck
```

As the LVS may take a long time, we can skip it with:

```
DISABLE_LVS=1 make run-precheck
```

Precheck generates a new folder `precheck_results` for all the logs. The main logfile is `precheck.log`. The results folder also includes individual reports, e.g., the LVS report. Precheck also checks for documentation issues, which gives errors from the cocotb documentation. Change the `Makrfile` for the `run-prechcheck` target and add `--private` to the call `mpw_precheck.py`.

The Docker for the precheck is built for an x86 and therefore runs very slowly on a Mac M1 with the Rosetta x86 emulation. The precheck of the simple example ran for almost 4 hours.

2.3.8 Submission to ChipFoundry

Submission to chipfactory needs only two files: the `gds/user_project_wrapper.gds` and `verilog/rtl/user_defines.v`, where the default for the GPIO pins directions are defined. The files can also be compressed with `gzip`.

ChipFoundry uses a command-line tool (`cf`) to manage uploading of your design via SFTP. Before uploading your design for manufacturing, you need to register for an SFTP Account. However, that account will only be enabled after paying the initial $ 500 non-refundable reservation deposit.

2.3.9 A Wishbone Peripheral

Caravel uses the Wishbone [7] bus to interface the processor core with the user design. Wishbone is an open-source bus definition intended to be used on-chip. The Wishbone specification defines a point-to-point communication. Wishbone is a public domain standard used by several open-source IP cores.

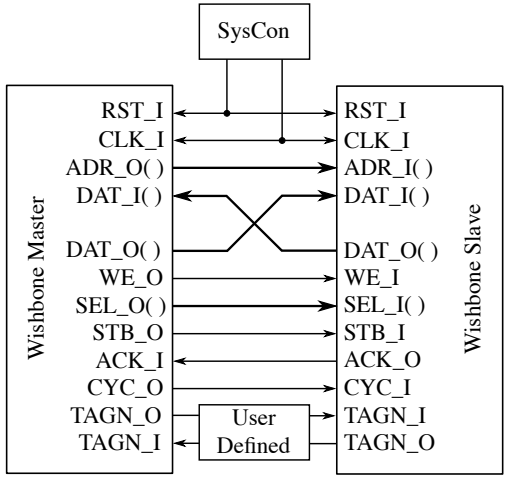


Figure 2.4: Wishbone interface

The Wishbone Bus

Figure 2.4 shows the connection between a Wishbone master and a Wishbone servant. Wishbone is a simple request/acknowledgment interface. The master signals a request by asserting `SEL_O` and `STB_O`. An asynchronous slave can, in the same cycle, reply with `ACK_O`. A synchronous slave can delay the acknowledgment. The master needs to hold the address, data, and control signals valid through the whole read or write cycle.

Although the bus specification is relatively simple, the Wishbone documentation is a bit lengthy, with 128 pages. However, we can follow RULE 3.40 and PERMISSION 3.10 from the specification to build simple Wishbone devices.

Rule 3.40:

> As a minimum, the master interface must include the following signals: `ACK_I`, `CLK_I`, `CYC_O`, `RST_I`, and `STB_O`. As a minimum, the slave interface must include the following signals: `ACK_O`, `CLK_I`, `CYC_I`, `STB_I`, and `RST_I`. All other signals are optional.

Permission 3.10:

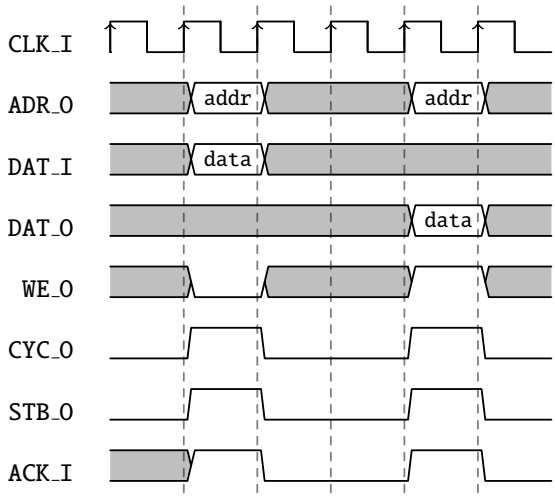


Figure 2.5: Wishbone asynchronous read followed by an asynchronous write

> If in standard mode, the slave guarantees it can keep pace with all master interfaces, and if the `ERR_I` and `RTY_I` signals are not used, then the slave's `ACK_O` signal may be tied to the logical AND of the slave's `STB_I` and `CYC_I` inputs. The interface will function normally under these circumstances.

Of course, to build a useful peripheral device, we also need to include the address, data in and out, and write enable signals.

Figure 2.5 shows a read and write transaction with an so-called *asynchronous* device. The terminology of asynchronous devices is from the Wishbone spec. It just means that for a single clock cycle bus transaction, the `ACK` signals need to be generated combinationally from `CYC` and `STB`.

Having this combinational loop from the master to the device and back to the master can lead to large combinational circuits when several devices are connected to the Wishbone bus. To break this loop, the `ACK` signal can be registered, leading to a two-clock-cycle-long transaction. This configuration is called a synchronous slave in the Wishbone specification. Figure 2.6 shows the timing diagram for a read and write on a synchronous device.

The latest Wishbone specification (B4) adds a pipelined definition. Note that the

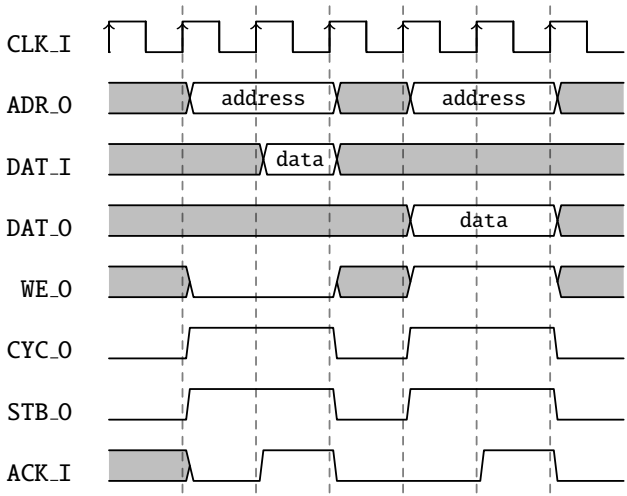


Figure 2.6: Wishbone synchronous read followed by a synchronous write

specification now includes two different, not necessarily compatible, specifications. The non-pipelined access is then called Wishbone *classic*. The pipelined mode has the following Issue: the slave does not know if the master will issue a pipelined request. It would need to observe changing addresses, which is not practical. To be on the safe side, a synchronous slave would need to assert `STALL_O`. However, as the Wishbone interface of the Caravel user project does not include a `STALL_I` input, we assume that there will be no pipelined Wishbone transaction.

A Simple Wishbone Device

The Wishbone interface for the user device is mapped to address `0x3000_0000` till `0x3fff_ffff`. We will design a very simple user device example with an 8-bit output port and an 8-bit input port. Both ports can be mapped to the same address, and we do not need any additional address decoding.

Listing 2.1 shows our very simple Wishbone device, coded in Chisel. It contains a `Bundle` definition for the interface and the main module `WishboneExample`. That module has two IO interfaces: (1) `wb` to the Wishbone bus from Caravel, and (2) `io` to 8 input and output pins. Note that we do not use the original, a bit dated, names for the Wishbine bus.

```
import chisel3._

// Wishbone interface definition (classic, minimal)
class WishboneIO(addrWidth: Int, dataWidth: Int) extends
    Bundle {
    val cyc  = Input(Bool())
    val stb  = Input(Bool())
    val we   = Input(Bool())
    val addr = Input(UInt(addrWidth.W))
    val din  = Input(UInt(dataWidth.W))
    val dout = Output(UInt(dataWidth.W))
    val ack  = Output(Bool())
}

// Simple Wishbone device: 8-bit in, 8-bit out
class WishboneExample extends Module {
    val wb = IO(new WishboneIO(addrWidth = 1, dataWidth =
        32))
    val io = IO(new Bundle {
        val in  = Input(UInt(8.W))
        val out = Output(UInt(8.W))
    })

    val outReg = RegInit(0.U(8.W))

    // wishbone combinational ack generation
    wb.ack := wb.cyc && wb.stb

    io.out := outReg
    // input with two FFs to contain meta stability
    wb.dout := RegNext(RegNext(io.in))

    // Wishbone write
    when(wb.cyc && wb.stb && wb.we) {
        outReg := wb.din
    }
}

object WishboneExample extends App {
    emitVerilog(new WishboneExample, Array("--target-dir",
        "generated"))
}
```

Listing 2.1: A simple Wishbone device in Chisel (`WishboneExample.scala`).

To keep the device simple, we use a combination acknowledgement for Wishbone. On a write, the output register `outReg` is written. It is directly connected to the 8-bit output pins. The input is connected to two levels of FFs to contain metastability. Otherwise, that input is directly connected to the Wishbone data input ports.

If you prefer to read Verilog, the same example in Verilog is shown in Listing 2.2. The Verilog code is a cleaned-up version of the Verilog that is generated by the Chisel example.

2.3.10 Notes

chipfoundry upload: https://github.com/chipfoundry/cf-cli?tab=readme-ov-file#installation

register an account at: https://chipfoundry.io/sftp-registration

2.4 Tiny Tapeout

Tiny Tapeout is a project that takes the idea of a multi-project wafer to the next level. Tiny Tapeout uses one tile of an MPW shuttle on Skywater130 or IHP and divides that tile into 512 smaller tiles. Therefore, the production cost can be further split, allowing for the sale of one tile for less than 100 USD.

Tiny Tapeout uses GitHub actions to harden user designs. Therefore, this is the easiest way to produce a chip. No tools need to be installed locally. The project is configured within two files: `config.json` and `info.yaml`.

2.4.1 Local Hardening

It is also possible to run the hardening locally, avoiding the long latency of GitHub actions. Follow the instructions on the TT website.

However, we can also do the local hardening with our installation of LibreLane instead of using the Docker image. After installing the Python dependencies into a Python virtual environment, execute the Python script to generate the configuration:

```
./tt/tt_tool.py --create-user-config
```

This command generates the file `usr_config.json` in folder `src` with information extracted from the `info.yaml` file. This file is merged with the `config.json` into `config_merged.json`. We can now harden the design by running:

```
librelane config_merged.json
```

```
module WishboneExample(
  input         clock,
  input         reset,
  input         wb_cyc,
  input         wb_stb,
  input         wb_we,
  input         wb_addr,
  input  [31:0] wb_din,
  output [31:0] wb_dout,
  output        wb_ack,
  input  [7:0] io_in,
  output [7:0] io_out
);
  reg [7:0] outReg;
  reg [7:0] wb_dout_REG;
  reg [7:0] wb_dout_REG_1;
  assign wb_dout = {{24'd0}, wb_dout_REG_1};
  assign wb_ack = wb_cyc & wb_stb;
  assign io_out = outReg;
  always @(posedge clock) begin
    if (reset) begin
      outReg <= 8'h0;
    end else if (wb_cyc & wb_stb & wb_we) begin
      outReg <= wb_din[7:0];
    end
    wb_dout_REG <= io_in;
    wb_dout_REG_1 <= wb_dout_REG;
  end
endmodule
```

Listing 2.2: A simple Wishbone device in Verilog (WishboneExample.v).

2.5 wafer.space

wafer.space is a new MPW service started by Tim Ansell and Leo Moser. It uses GF130 and offers 1000 dies with a user space of 20 mm^2. The project template is just the pad ring and a counter as an example design.

2.6 Maybe something from Edu4Chip?

3 Memories

When describing memories in an HDL, the synthesis tool will not generate on-chip memories, as we are used to in an FPGA. It will use generated storage out of DFF and multiplexers, which is very expensive. As an example, a 1024-bit memory is needed to implement the register file for an RISC-V processor. Implementing an RV-32I version in a 3-stage pipeline, the register file needs 320 µm x 320 µm or about 55% of the processor area [11].

An efficient memory cell is built out of six-transistor (6T) SRAM cells, as shown in Figure 3.1. The middle four transistors implement two cross-coupled inverters. The other two transistors are used for reading and writing the bit.

However, standard synthesis tools do not generate memories based on 6T SRAM cells. The tools to generate on-chip memories are called memory compilers. OpenRAM is an open-source memory compiler [6]. Furthermore, the Sky130 PDK includes a few memories generated with OpenRAM.

3.1 Flip-Flop and Latch-based Memories

When describing memories in an HDL, the synthesis tool will usually infer FFs for the storage and large multiplexers for reading. The register file example is probably a bit extreme, as it describes a memory with two read ports, meaning we need two read multiplexers.

Listing 3.1 shows the Chisel code of a flip-flop-based memory. The memory is 128 bytes, which is 1024 bits, the same size as the register file example discussed at the start of the chapter. The area for that memory in Sky130 is 290 µm x 290 µm (0.08 mm^2), which is similar to that of the register file mentioned at the beginning of the chapter. The 1 KiB register-based memory, organized as 256 x 32-bit, uses 810 µm x 810 µm (0.66 mm^2).

128 x 8: Chisel 3.6: 314 x 310, but DRC errors (setup violations). Chisel 6.x: 290 x 290, DRC OK

CF RAM 1024x32: 310 x 390 OpenRAM: 256x32: 500x400

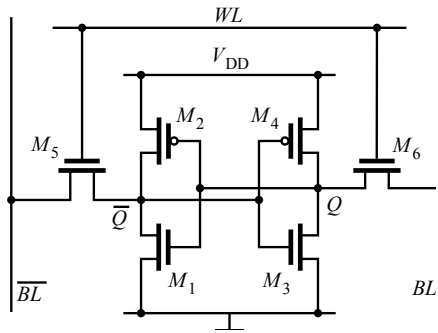


Figure 3.1: A six-transistor (6T) CMOS SRAM cell.

```
import chisel3._

class FlipFlopMemory extends Module {
    val io = IO(new Bundle {
        val addr = Input(UInt(7.W))
        val din  = Input(UInt(8.W))
        val we   = Input(Bool())
        val dout = Output(UInt(8.W))
    })


   val mem = Reg(Vec(128, UInt(8.W)))
   when(io.we) {
       mem(io.addr) := io.din
   }
   io.dout := mem(io.addr)
}

object FlipFlopMemory extends App {
    emitVerilog(new FlipFlopMemory, Array("--target-dir",
        "generated"))
}
```

Listing 3.1: 128 Bytes of Flip-Flop based Memory (`FlipFlopMemory.scala`).

3.2 Exploring OpenRAM Memories

The process of including those macros is not straightforward. First errors occur in nn-magic-writelef. To ignore those errors, we can add the following line to our .yaml file:

```
MAGIC_CAPTURE_ERRORS: false
```

The 1 KiB OpenRAM memory, organized as 256 x 32-bit, uses 490 µm x 420 µm (0.2 mm^2).

3.3 DFFRAM

DFFRAM is a memory generator using FFs or latches. The GitHub releases contain ready-to-use GDS macros. A 1 KiB memory, latch-based, uses 430 µm x 440 µm (0.19 mm^2).

3.4 CF RAM

You can install the commercial RAM from ChipFoundry and the DFFRAM with their IP management tool `ipm`, using the available Python virtual environment `venv`:

```
source venv/bin/activate
pip install cf-ipm
ipm install CF_SRAM_1024x32
ipm install DFFRAM256x32
deactivate
```

Note that on a Linux machine `ipm` is installed at `.loca/bin`, which might not be in your PATH.

4 KiB (1024 x 32) 0.17 mm2

3.4.1 SRAM links

SRAM on TT09: https://github.com/FriedrichWu/tt09-sram/tree/main

Uri's IHP SRAM test on TT:mhttps://tinytapeout.com/chips/ttihp0p2/tt_um_urish_sram_test

Another SRAM compiler: https://github.com/rahulk29/sram22 including precompiled memories: https://github.com/rahulk29/sram22_sky130_macros

Matt design: https://github.com/mattvenn/zero_to_asic_mpw7/tree/mpw7/openlane/user_project_wrapper

OpenRAM playground: https://gist.github.com/proppy/1054e1618f5f90ce3af529a

DFFRAM: https://github.com/AUCOHL/DFFRAM

https://tinytapeout.com/specs/memory/

Tutorial: https://armleo-openlane.readthedocs.io/en/merge-window-4/tutorials/openram.html

and at https://openlane.readthedocs.io/en/latest/tutorials/openram.html

4 List of Chapters

4.1 Notes and Pointers (Reading List)

Intro to OpenLane: https://openlane2.readthedocs.io/en/latest/getting_started/newcomers/index.html

https://vlsi.ethz.ch/wiki/VLSI_Lectures ETHZ notes

https://github.com/OS-EDA/Course IHP course

https://github.com/open-source-eda-birds-of-a-feather/open-source-eda-bi github.io/blob/main/doc/slides_2025/BOF25_PULP_mbertuletti.pdf ETH presenation

https://github.com/open-source-eda-birds-of-a-feather/open-source-eda-bi github.io/blob/main/doc/slides_2025/DAC25%20Recent%20Experiences%20Markarian-v3a.pdf UCSD course

On STA: https://www.zerotoasiccourse.com/terminology/sta/

Chip design book: https://link.springer.com/book/10.1007/978-90-481-9591-6

Local TT hardening: https://tinytapeout.com/guides/local-hardening/

https://github.com/iic-jku/SKY130-RTL-with-Custom-Standardcell-to-GDSII/blob/main/README.md

Use the correct layer map with KLayout:

```
klayout -l ../../dependencies/pdks/sky130A/libs.tech/klayout/tech/sky130A.lyp runs
```

4.1.1 Matt Venn Links

List of efabless projects: https://github.com/mattvenn/efabless_project_tool clone and efabless_tool.py

```
./efabless_tool.py --fields id,sumary,giturl --list | grep -i sram
```

New summary tool from Matt: https://github.com/mattvenn/librelane_summary

An article from Matt https://www.zerotoasiccourse.com/post/excited_by_silicon/?mc_cid=5a18c03517&mc_eid=ba890f57df

TT config.json has hints what to do on hold time violation (also change cycle time for setup violations)

4.2 The MOSFET and CMOS Technology

4.3 Standard Cells

We can explore the standard cells contained in the Sky130 PDK with KLayout:

```
klayout $HOME/.ciel/sky130A/libs.ref/sky130_fd_sc_hd/gds/sky130_fd_sc_hd.gds
```

However, with Matt's summary tool, the invocation is easier and the layers have names and better colors:

```
summary.py --show-sky130
```

Right-click on any cell in the *Cells* window to select a new one with *Show as New Top*.

4.3.1 FABs

4.3.2 PDK

We have three PDKs available in open source. Therefore, we can compare them as an exercise.

4.4 The Design Flow

4.5 Hardware Description Languages

4.5.1 Verilog

4.5.2 VHDL

4.5.3 SystemVerilog

4.5.4 Chisel

[2] [10]

4.5.5 Other Languages

4.5.6 Amaranth

4.5.7 SpinalHDL

4.5.8 MyHDL

4.5.9 Clash

4.5.10 Spade

4.5.11 Generator Scripting Languages

4.6 Open-Source Tools

4.6.1 Magic

John Ousterhout wrote Magic at UCB. Now he is at Stanford and has written a book on agile SW development. https://web.stanford.edu/~ouster/cgi-bin/home.php

4.6.2 ABC

4.6.3 Yosys

Yosys [14] started as a Bachelor's project by Clifford Wolf at the Technical University of Vienna [15]. Yosys is a free and open-source software for Verilog HDL synthesis. Yosys synthesizes Verilog HDL to logically equivalent netlists. Yosys uses external tools, such as Berkeley's ABC [3] for combinational logic minimization.

4.7 Use Cases

In this section, we describe some uses of open-source tools with open-access PDKs.

With the initial Google-sponsored tapeout, a lot of projects have been taped out on SkyWater130. All projects have been submitted through eFabless, and the open-source projects have been listed on the eFabless website. However, with the closure

of eFabless, the list of projects got lost. We tried to recover as many projects as possible as a reference for future open-source tapeouts.

Basilisk is a RISC-V core developed at the ETH Zurich and the University of Bologna [9]. Basilisk is one of the largest designs today implemented with open-source tools. It is a 64-bit Linux-capable RISC-V code. Besides the core itself, it includes IO devices, such as a DRMA controller, USB host, and video output. The core was implemented in the IHP's 130 nm BiCMOS technology with the open-source PDK in 34 mm2 and can be clocked at a nominal 1.2 V voltage at 77 MHz.

A Resources

- LibreLane is the current collection of tools for open-source chip design
- The Caravel User Project is the starting point for a chip designed for the Caravel platform produced by
- ChipFoundry can produce you prototyping chip
- Caravel the harness for the ChipFoundry MPW run
- The Caravel documentation is a bit outdated
- Caravel Simulation
- Management core documentation
- The PCB containing the Caravel chip

B Acronyms

Hardware designers and computer engineers like to use acronyms. However, it takes time to get used to them. Here is a list of common terms related to digital design and computer architecture.

ADC analog-to-digital converter

ALU arithmetic and logic unit

ASIC application-specific integrated circuit

CAD computer-aided design

Chisel constructing hardware in a Scala embedded language

CISC complex instruction set computer

CPI clock cycles per instruction

CPU central processing unit

CRC cyclic redundancy check

DAC digital-to-analog converter

DFF D flip-flop, data flip-flop

DMA direct memory access

DRAM dynamic random access memory

EDA electronic design automation

EMC electromagnetic compatibility

ESD electrostatic discharge

FF flip-flop

FIFO first-in, first-out

FPGA field-programmable gate array

GDS graphic design system

HDL hardware description language

HLS high-level synthesis

IC integrated circuit (also instruction count in computer architecture)

IDE integrated development environment

ILP instruction-level parallelism

IO input/output

ISA instruction set architecture

JDK Java development kit

JIT just-in-time

JVM Java virtual machine

LC logic cell

LRU least-recently used

LSB least significant bit

MMIO memory-mapped IO

MSB most significant bit

MUX multiplexer

OO object oriented

OOO out-of-order

OS operating system

PDK Process Development Kit

RAM random access memory

RISC reduced instruction set computer

SDRAM synchronous DRAM

SRAM static random access memory

UART universal asynchronous receiver/transmitter

VHDL VHSIC hardware description language

VHSIC very high speed integrated circuit

Bibliography

[1] Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, Lutong Wang, Zhehong Wang, Mingyu Woo, and Bangqi Xu. Invited: Toward an open-source digital flow: First learnings from the openroad project. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, page 76. ACM, 2019.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In *The 49th Annual Design Automation Conference (DAC 2012)*, pages 1216–1225, San Francisco, CA, USA, June 2012. ACM.

[3] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[4] William J. Dally, R. Curtis Harting, and Tor M. Aamodt. *Digital design using VHDL: A systems approach*. Cambridge University Press, 2016.

[5] Ahmed Ghazy and Mohamed Shalan. Openlane: The open-source digital asic implementation flow. In *Workshop on Open-Source EDA Technology (WOSET)*, 2020.

[6] Matthew R. Guthaus, James E. Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. Openram: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6.

[7] Wade D. Peterson. WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores, revision: B.4. Available at http://www.opencores.org, September 2010.

[8] A. Sangiovanni-Vincentelli. The tides of EDA. *IEEE Design & Test of Computers*, 20(6):59–75, 2003.

[9] Paul Scheffler, Philippe Sauter, Thomas Benz, Frank K. Gürkaynak, and Luca Benini. Basilisk: An end-to-end open-source linux-capable risc-v soc in 130nm cmos, 2024.

[10] Martin Schoeberl. *Digital Design with Chisel*. Kindle Direct Publishing, 2019. available at https://github.com/schoeberl/chisel-book.

[11] Martin Schoeberl. The educational risc-v microprocessor wildcat. In *Proceedings of the Sixth Workshop on Open-Source EDA Technology (WOSET)*, 2024.

[12] Martin Schoeberl. Wildcat: Educational risc-v microprocessors. In *Architecture of Computing Systems – ARCS 2025*, 2025.

[13] Mohamed Shalan and Tim Edwards. Building openlane: A 130nm openroad-based tapeout- proven flow : Invited paper. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–6, 2020.

[14] Claire Wolf and Johann Glaser. Yosys-a free verilog synthesis suite. In *In: Proceedingsof the 21st Austrian Workshop on Microelectronics (Austrochip)*.

[15] Clifford Wolf. Design and implementation of the yosys open synthesis suite. Bachelor thesis, Vienna University of Technology, 2013.