A Tutorial on Open-Source Analog-Mixed Signal Chip Design with the ihp-sg13g2 Open-PDK

Simon Dorrer Harald Pretl

2026-05-13

Table of contents

> **Warning**
>
> This documentation is a Work in Progress.

> **Important**
>
> This tutorial and the underlying template repository require the IIC-OSIC-TOOLS container with tag `2026.05` or later. All commands below assume that the container has been entered and that the working directory is the cloned repository root.

1 Overview

This tutorial is a step-by-step walk-through of the open-source analog-mixed signal (AMS) chip flow based on the `ihp-sg13g2-ams-chip-template` repository for the IHP SG13G2 130-nm Open-PDK. It covers the tool installation, the RTL and analog macro design, and the chip-level assembly with padframe, logos, and fill structures.

The example design used throughout the tutorial contains:

- Two digital `counter_top` macros (8-bit synchronous up-counters),
- Two `inverter_top` macros, one used as a 4-channel CMOS digital inverter, one as a 2-channel analog inverter,
- One `RM_IHPSG13_1P_1024x32_c2_bm_bist` single-port SRAM macro,
- A 32-pad padframe (8 pads per side) with I/O, power, analog, and bidirectional pads,
- The top-level chip assembly with PDN, JKU logos, and KLayout fill.

The tutorial ends with a set of exercises that apply the same flow to a modified counter, a modified inverter, and a modified top-level floorplan.

> **Tip**
>
> The tutorial can be used both for self-study and as the basis for a workshop. Each section can be followed independently.

2 Tools, the IIC-OSIC-TOOLS Container

The open-source AMS flow used here relies on several tools, namely Xschem, Magic, KLayout, ngspice, CACE, LibreLane, Yosys, OpenROAD, Verilator, cocotb, GTKWave, and Surfer. To make installation and version handling easier, they are all bundled into the **IIC-OSIC-TOOLS** Docker container maintained by the IIC at JKU Linz. The container also ships the IHP SG13G2 Open-PDK used by this template.

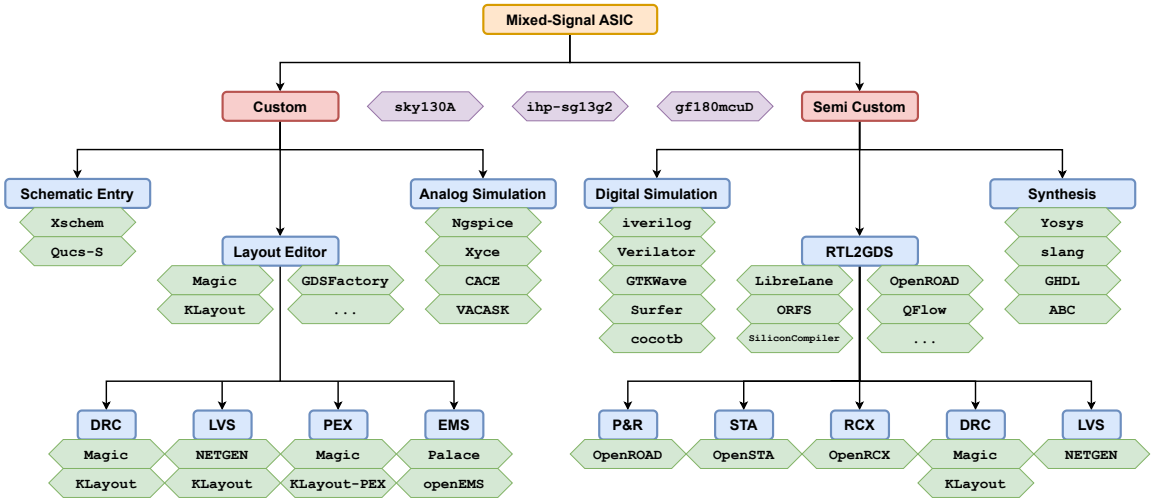

Figure 1: Overview of the open-source tools shipped with the IIC-OSIC-TOOLS container.

2.1 Installation

The container runs on Linux, macOS, and Windows. The installation instructions are kept in the `iic-jku/IIC-OSIC-TOOLS` GitHub repository. Video walk-throughs are available for Linux and Windows.

> **Ubuntu / Linux installation**
>
> - IIC-OSIC-TOOLS installation on Ubuntu (Part 1)
> - IIC-OSIC-TOOLS installation on Ubuntu (Part 2)

> **Windows installation**
>
> - IIC-OSIC-TOOLS installation on Windows (from 37:20 to 45:15)

On macOS, the same Docker-based instructions apply. See the container's `README.md` for OS-specific notes.

2.2 Selecting the IHP SG13G2 PDK

The container can host several Open-PDKs (SkyWater 130, GlobalFoundries 180, IHP SG13G2). The active PDK is selected per `designs/` folder via a marker file called `.designinit`. Two options are available.

1. **Per-design marker file.** Copy a `.designinit` file referencing the IHP SG13G2 PDK into your `designs/` directory and restart the container. The PDK is then picked automatically every time the container is launched in that directory.
2. **On-the-fly switch.** Run

   ```
   sak-pdk ihp-sg13g2
   ```

   inside the container. This switches the active PDK for the current session without modifying any file.

> **Tip**
>
> By default the container is already configured for `ihp-sg13g2`, so both options above are only needed when switching between PDKs.

3 Repository Overview

3.1 Purpose

This Makefile-driven repository simulates, builds, and verifies (LVS, DRC, PEX) an analog mixed-signal chip for the ihp-sg13g2 130-nm Open-PDK, including padframe generation and top-level assembly. It uses:

- **LibreLane** for digital macro hardening, padframe generation, and top-level assembly,
- **Xschem** for schematic entry,
- **Ngspice**, **VACASK**, and **CACE** for analog simulation,
- **KLayout** for viewing and routing of the layout,
- **Magic** + **Netgen** and **KLayout** for LVS, DRC, and PEX,
- **SystemVerilog**, **cocotb**, **GTKWave**, and **Surfer** for digital simulation.

The repository is the right starting point for custom silicon. Just clone it, enter the IIC-OSIC-TOOLS container, run `make all`, and the result is a tapeout-ready analog-mixed signal chip. It also acts as a regression test for the open-source tools on the IHP SG13G2 Open-PDK.

3.2 Directory Structure

The repository is organised into the following top-level folders. The full listing is in the main README.md.

Folder	Purpose
doc/	Designer documentation (specifications, pinout, floorplan) and PDK / tool cheatsheets.
flow/	LibreLane top-level configuration (config.yaml, pdn_cfg.tcl, chip_top.sdc) and the ArtistIC logo flow.
ip/	External IP cells (bondpads, custom IO cells, JKU logos).
layout/	Compressed GDS streams of the top-level chip (chip_top.gds.gz and chip_top_logo_fill.gds.gz).
macros/	In-tree macros (counter/ and inverter/), each with its own Makefile and README.
netlist/	Synthesis, layout, and PEX netlists used for LVS and simulation.
release/	Tapeout submission artifacts, versioned (v.1.0.0/, …).
render/	Chip and macro renders for documentation.
rtl/	Top-level RTL (chip_top.sv, chip_core.sv).
schematic/	Xschem schematics and symbols for the chip top.
scripts/	Helper Python and shell scripts (rendering, logo placement, plotting).
testbenches/	Cocotb and Xschem testbenches for the top-level chip.
tutorial/	This Quarto tutorial.
verification/	DRC / LVS / signoff reports.

3.3 Workflow in a Nutshell

The flow is driven by Makefiles. After cloning the repository and entering the IIC-OSIC-TOOLS container, every step is invoked via a `make` target. The default goal of every `Makefile` is `help`, so `make` (without arguments) prints the list of available targets.

```
git clone https://github.com/iic-jku/ihp-sg13g2-ams-chip-template.git
cd ihp-sg13g2-ams-chip-template
make help              # show available targets at the chip top level
```

Each macro under macros/ has its own `Makefile` and `README.md` with macro-local targets. They are called by the top-level build targets, but can also be invoked directly from inside the macro folder.

3.3.1 Most important top-level targets

The full list is in the top-level Makefile. The relevant subset is summarised below.

Target	What it does
`make help`	Print the help banner and list every target with its description.
`make init-submodules`	Initialise and update git submodules (for example the ArtistIC logo flow).
`make sim-rtl-cocotb` / `sim-gl-cocotb`	Run the RTL / gate-level cocotb testbench for the chip top.
`make sim-gl-xschem`	Run the gate-level Xschem testbench. Converges, but it may take a long time depending on the hardware used — not included in `sim-all`.
`make sim-view-cocotb`	Open the latest cocotb waveform in GTKWave (or Surfer with `WAVEFORM_VIEWER=surfer`).
`make librelane` / `librelane-nodrc`	Run the LibreLane flow for the chip top (with / without DRC steps).
`make copy-reports` / `copy-gds` / `copy-netlist` / `copy-render`	Copy the latest LibreLane artifacts back into the source tree.
`make build-bondpad` / `build-logos`	Build the bondpad and the JKU logos under `ip/`.
`make build-counter` / `build-inverter` / `make build-macros`	Build the digital and analog macros individually or together.
`make build-top`	Run LibreLane on the chip top, copy back all artifacts, add logo and fill, render the final GDS.
`make build-all`	Init submodules, build bondpad, build logos, build macros, and `build-top`.
`make klayout-verify` / `magic-verify`	Run LVS, DRC, and PEX for a given cell with KLayout or Magic.
`make all`	Full simulation, build, and DRC pass.

Target	What it does
`make release VERSION=2.1.0`	Copy GDS, netlists, and renders into `release/v.<VERSION>/` for tapeout submission.

3.3.2 Most important counter Makefile targets

The counter Makefile provides:

Target	What it does
`make lint-verilog` / `make lint-verilog-all`	Lint a single cell or the full counter design with Verilator.
`make sim-rtl-verilog`	Run the SystemVerilog testbench with Icarus Verilog.
`make sim-rtl-cocotb` / `sim-gl-cocotb`	Run the cocotb RTL / gate-level simulation.
`make sim-view-verilog` / `sim-view-cocotb`	Open the latest waveform in GTKWave or Surfer.
`make sim-gl-xschem` / `sim-view-xschem`	Run the mixed-signal XSPICE simulation in Xschem and plot the results.
`make librelane` (and `-nodrc`, `-magicdrc`, …)	Run the LibreLane flow for the counter macro.
`make generate-xspice`	Convert the post-synthesis netlist into an XSPICE model for Xschem.
`make build-fpga` / `make build-top` / `make all`	FPGA bitstream, full ASIC build, or both plus simulation in one call.

See the counter README for details.

3.3.3 Most important inverter Makefile targets

The inverter Makefile provides:

Target	What it does
`make sim-xschem TB=<tb>` / `make sim-view-xschem`	Run a specific Xschem testbench and plot its results with Python.
`make sim-cace`	Run CACE Monte-Carlo / mismatch characterisations.
`make klayout-lvs` / `magic-lvs`	Run LVS using KLayout or Magic + Netgen.
`make klayout-drc` / `magic-drc`	Run DRC using KLayout or Magic.

Target	What it does
`make klayout-pex` / `magic-pex` (`EXT_MODE=<1|2|3>`)	Run parasitic extraction in C, CC, or full-RC mode.
`make klayout-verify` / `magic-verify`	Run LVS + DRC + PEX for a given cell.
`make lef` / `make lib` / `make verilog`	Generate the LEF, the Liberty stub, and the Verilog stub for integration.
`make build-top` / `make all`	Build deliverables, or run simulation, verification, and build together.

See the inverter README for details.

4 Counter Macro Implementation

The `counter` macro is the digital reference macro of the template. It is a parameterisable synchronous up-counter with synchronous active-high reset; the top-level wrapper converts the chip's active-low reset to an active-high signal. The default width is 8 bits, so the counter wraps from 255 back to 0.

4.1 Folder Structure of the Counter Macro

In `macros/counter/`, the relevant folders are:

Folder	Content
`rtl/`	SystemVerilog sources: `constants.sv`, `counter.sv`, `counter_top.sv`.
`testbenches/`	`verilog/`, `cocotb/`, and `xschem/` testbenches with pre-configured GTKWave and Surfer view files.
`flow/`	LibreLane configuration (`config.yaml`, `pin_order.cfg`, `impl.sdc`, `signoff.sdc`).
`final/`	Final views copied from the latest LibreLane run (GDS, LEF, LIB per corner, NL, PNL, SPEF, VH).
`fpga/`	Optional FPGA flow for the pico-ice board.
`netlist/`	Gate-level netlists (`nl/`, `pnl/`, `spice/`, `xspice/`).
`schematic/`	Xschem symbol of the macro for mixed-signal usage.
`scripts/`	Helper scripts (XSPICE conversion, plotting, layout rendering).
`verification/`	Yosys, antenna, STA, IR-drop, DRC, LVS, and manufacturability reports.

Folder	Content
render/	Macro renders for documentation.

> **Tip**
>
> Read the counter README first to get familiar with the folder layout and the available Makefile targets. Most of the commands below are documented there as well.

4.2 Reading the RTL

The macro is split into a parameterised counter and a thin top-level wrapper that converts the chip's active-low reset to the internal active-high reset of the counter.

4.2.1 `counter.sv`

The arithmetic is implemented in `counter.sv`. It is an N-bit synchronous up-counter with reset, enable, and wrap at `CTR_MAX`.

4.2.2 `counter_top.sv`

`counter_top.sv` wraps the counter and converts the active-low reset (`reset_n_i`) to an active-high signal.

Shared defaults like the bit-width and the clock frequency are defined in `constants.sv`. They are exposed as `` `define `` macros instead of a SystemVerilog `package`, because Yosys 0.64 cannot parse `import pkg::*` in a module header.

> **Note**
>
> Open the three RTL files (using `gedit` or `vim`) and follow the counter through reset, hold, increment, and wrap-around. Check why the constants are `` `define ``-style macros and not `package` parameters.

4.3 Linting

Linting is done with Verilator.

```
cd macros/counter
make lint-verilog                # lint the full counter_top design
make lint-verilog CELL=counter   # lint only the standalone counter
make lint-verilog-all            # lint counter and counter_top in sequence
```

`make lint-verilog-all` is also the lint step called by `make all`.

4.4 Simulation

4.4.1 RTL with SystemVerilog testbench (Icarus Verilog)

A SystemVerilog testbench (`counter_top_tb.sv`) checks reset, hold, increment, and wrap-around. Run it with the following commands.

```
make sim-rtl-verilog
make sim-view-verilog                      # open in GTKWave (default)
make sim-view-verilog WAVEFORM_VIEWER=surfer
```

> **Important**
>
> Surfer is currently not working on macOS, and might not work on Windows/WSL. If you want to use Surfer, please run the container on Linux.

> **Figure placeholder. RTL Verilog waveform of `counter_top` in GTKWave or Surfer.**

4.4.2 RTL and gate-level with cocotb

The cocotb testbench (`counter_top_tb.py`) covers the same four behaviours from Python, and is the one called by `make all`.

```
make sim-rtl-cocotb                # RTL simulation
make sim-gl-cocotb                 # post-synthesis gate-level simulation
make sim-view-cocotb               # view in GTKWave
make sim-view-cocotb WAVEFORM_VIEWER=surfer
```

The gate-level cocotb run uses the post-synthesis netlist `netlist/nl/counter_top.nl.v`, which is copied from the latest LibreLane run by `make copy-netlist`.

Figure placeholder. Cocotb gate-level waveform of `counter_top`.

4.5 LibreLane Hardening

The LibreLane flow synthesises, places, and routes the macro, runs the sign-off steps, and writes its final views to `flow/final/`. The wrapper target `make build-top` runs the flow, copies the reports, final folders, netlists, and the render back into the source tree, and renders the final GDS.

```
make build-top
```

> **Tip**
>
> The macro can also be built as an FPGA bitstream (fpga/) for a pico-ice board with `make build-fpga`. The FPGA path is not used in this tutorial and is listed here for reference.

4.6 Viewing the Macro

After the flow finishes, the macro layout can be inspected with KLayout. The container provides the `ke` alias for `klayout -e` (edit mode).

```
ke macros/counter/final/gds/counter_top.gds
```

The LibreLane render is copied to macros/counter/render/img/counter_top_librelane.png, and the high-resolution render from `lay2img.py` is saved as `counter_top.png` in the same folder. The sign-off reports are written to macros/counter/verification/.

Figure placeholder. `counter_top` layout in KLayout.

4.7 Gate-Level Mixed-Signal Simulation with Xschem

A more accurate sign-off step is the gate-level mixed-signal simulation in Xschem with the ngspice XSPICE event-driven engine. The post-synthesis Verilog netlist (`yosys-synthesis/<TOP>.nl.v`) is converted into an XSPICE model with the following command.

```
make generate-xspice
```

The conversion pipeline is `.nl.v → .vp → .spice → .xspice`, followed by a pin-reorder pass to match the Xschem symbol (schematic/xschem/counter_top.sym).

The Xschem testbench (counter_top_tb_tran.sch) can be opened and simulated interactively.

```
cd macros/counter/testbenches/xschem
xschem counter_top_tb_tran.sch
```

In Xschem, hold **Shift** and click the **Simulate** button in the schematic to launch the simulator. You can also hold **Ctrl** and click the **Simulate** arrow, followed by a **Ctrl**-click on the **Load waves** arrow to load the simulation results into the integrated waveform viewer.

> **Note**
>
> These arrows in Xschem are called "launchers" and are essentially a series of Tcl commands. Since Xschem is fully scriptable via Tcl, repetitive tasks can be easily automated. Select a launcher and press `q` (Properties) to inspect or modify its underlying Tcl commands.

The same simulation can also be run non-interactively from the Makefile, and the result can be plotted from Python.

```
make sim-gl-xschem
make sim-view-xschem
```

> **Figure placeholder. Xschem mixed-signal testbench `counter_top_tb_tran`.**

> **Figure placeholder. Plotted Xschem simulation results from `scripts/plot_simulations/plot_counter_top.py`.**

4.8 Build and Verify Everything

The full lint, simulation, and build chain is run with the following command.

```
make all
```

This calls `lint-verilog-all`, `sim-all`, `build-fpga`, and `build-top` in sequence. LVS and DRC are already covered by the LibreLane sign-off steps, so no extra verification target is needed for the counter.

5 Inverter Macro Implementation

The `inverter` macro is the analog reference macro of the template. It is a four-channel CMOS inverter, drawn at transistor level, and verified with the comprehensive AMS sign-off chain (LVS, DRC, PEX, Monte-Carlo simulation for process variations and mismatch with CACE). Two instances are used in the chip core. One is used as a 4-channel digital inverter (`inverter1`), the other as a 2-channel analog inverter (`inverter2`). See the chip specifications for the full picture.

5.1 Folder Structure of the Inverter Macro

In `macros/inverter/`, the relevant folders are:

Folder	Content
`schematic/xschem/`	Xschem schematics and symbols (`inverter.sch`, `inverter.sym`, `inverter_top.sch`, `inverter_top.sym`, `inverter_top_pex.sym`).
`layout/`	KLayout-edited GDS streams (`*.klay.gds`) and the exported `inverter_top.gds`.
`testbenches/xschem/`	Transient, AC open-loop, output-DC, and top-level transient testbenches.
`netlist/`	`schematic/` (CDL / SPICE for LVS), `layout/` (extracted), `pex/` (PEX SPICE).
`scripts/`	Python plotters, sizing notebooks, layout rendering, PEX pin reordering.
`verification/`	`drc/`, `lvs/`, and `cace/` (process, mismatch, supply sweeps).
`final/`	Build deliverables for the top-level integration (LEF, LIB, Verilog stub, GDS).
`render/`	Macro renders for documentation.

> **Tip**
>
> Read the inverter README first to get familiar with the folder layout and the available Makefile targets. Most of the commands below are documented there as well.

5.2 Schematic-Level Simulation

Go to the Xschem testbench folder and start Xschem.

```
cd macros/inverter/testbenches/xschem
xschem
```

Four testbenches are available in testbenches/xschem/.

- inverter_tb_tran.sch, single-channel transient response,
- inverter_tb_ac_ol.sch, open-loop AC characterisation,
- inverter_tb_Vout.sch, DC input sweep,
- inverter_top_tb_tran.sch, top-level transient on `inverter_top`.

Load a testbench in Xschem, then hold **Ctrl** and press the **Simulate** arrow in the schematic to launch the simulator manually.

Figure placeholder. Xschem `inverter_tb_tran` simulation result.

Useful Xschem keybindings

- Mark a symbol and press **e** to descend into the underlying circuit, then **Ctrl** + **e** to go back up.
- Mark a symbol and press **i** to descend into the symbol drawing, then **Ctrl** + **e** to go back up.

This is the recommended way to inspect or modify the underlying circuit and the graphical symbol of any device.

Non-interactive runs and post-processing are also available through the Makefile.

```
cd macros/inverter
make sim-xschem TB=inverter_tb_tran
make sim-view-xschem CELL=inverter        # plot results with Python
```

The full simulation suite (all four testbenches plus the CACE flow) is triggered with `make sim-all`.

5.3 Process Variation and Mismatch (CACE)

CACE drives ngspice via a characterisation YAML file (verification/cace/inverter.yaml) and produces Monte-Carlo, mismatch, and supply-sweep plots.

```
make sim-cace
```

Plots are written to verification/cace/results/inverter/. This step is mainly relevant for advanced designers that want to characterise the macro across PVT corners.

5.4 Inspect / Edit the Layout

The layout is created and edited in KLayout. The container alias `ke` opens KLayout in edit mode.

```
cd macros/inverter/layout
ke inverter_top.klay.gds
```

After modifying the layout, export a clean `.gds` file (for example `inverter_top.gds`). The Magic-based flows always work on `.gds`, while the KLayout-based flows accept either `.gds` or `.klay.gds`.

> **Figure placeholder. `inverter_top` layout in KLayout.**

5.5 DRC, LVS, and PEX

The inverter macro can be verified with two sign-off paths, KLayout and Magic + Netgen. Each path covers DRC, LVS, and PEX.

5.5.1 Design Rule Check

```
make klayout-drc          # KLayout DRC (FEOL, density, extra rules disabled)
make magic-drc            # Magic DRC via sak-drc.sh
```

Reports are saved to verification/drc/.

5.5.2 Layout-Versus-Schematic

```
make klayout-lvs          # uses CDL netlist exported from Xschem
make magic-lvs            # uses SPICE netlist exported from Xschem
```

The schematic netlist is exported automatically by the matching `*-lvs-netlist` sub-target. Reports are saved to verification/lvs/, and the extracted layout netlist is moved to netlist/layout/.

5.5.3 Parasitic Extraction

```
make klayout-pex                    # default EXT_MODE=3 (full-RC)
make magic-pex EXT_MODE=2           # CC mode
```

PEX writes a SPICE netlist into `netlist/pex/`. The subcircuit name is renamed to `<CELL>_pex`, and the pin order is rearranged to match the Xschem PEX symbol (`inverter_top_pex.sym`).

> **Post-layout simulation**
>
> Open one of the Xschem testbenches and swap the original symbol with the PEX symbol to use the extracted netlist in the same testbench. This is the recommended way to compare schematic-only and post-layout simulation results.

5.5.4 Run Everything at Once

The verify targets bundle LVS, DRC, and PEX in one command.

```
make klayout-verify         # CELL defaults to inverter_top
make magic-verify
```

5.6 Build Deliverables for Top-Level Integration

For the inverter to be picked up by the chip-level LibreLane flow, it must expose a **LEF** (abstract pin / blockage view), a **Liberty timing library** stub, and a **Verilog stub**. They are built individually or via `build-top`.

```
make lef          # final/lef/inverter_top.lef
make lib          # final/lib/inverter_top.lib (Liberty stub)
make verilog      # final/vh/inverter_top.v (pin classification from PEX)
make build-top    # the three steps above plus copy-gds and render-gds
```

`make build-top` also copies the GDS to `final/gds/` and renders the macro to `render/img/inverter_top.png`.

5.7 Full Macro Pass

The full simulation, verification, and build chain is run with the following command.

```
make all
```

This calls `sim-all` (Xschem + CACE), `klayout-verify-all`, `magic-verify-all`, and `build-top` in sequence. Depending on the host hardware, a full pass can take a few minutes.

6 Top-Level Implementation

This section covers the chip-level assembly. The top-level files that need to be understood (and, if needed, modified for a new chip) are:

- `rtl/chip_top.sv`, the padframe wrapper. It instantiates one I/O cell per pad and exposes the global `clk_PAD`, `rst_n_PAD`, `input_PAD`, `output_PAD`, `bidir_PAD`, and `analog_PAD` buses.
- `rtl/chip_core.sv`, the core logic. It instantiates the digital and analog macros (`counter1`, `counter2`, `inverter1`, `inverter2`, `sram_0`) and wires them to the I/O bus.
- `flow/librelane/config.yaml`, the top-level LibreLane configuration. It defines the padframe order (`PAD_WEST`, `PAD_NORTH`, `PAD_SOUTH`, `PAD_EAST`), `DIE_AREA`, `CORE_AREA`, `CLOCK_PERIOD`, the macro list (`MACROS:`), non-default routing rules, PDN parameters, and STA corners.
- `flow/librelane/pdn_cfg.tcl`, the Tcl callback that builds the OpenROAD PDN. The bottom of the file contains the custom PDN connects for `inverter1`, `inverter2`, and `sram_0`. This is the part that has to be touched when adding new macros.
- `flow/librelane/chip_top.sdc`, timing constraints (clock period, input/output delays).
- Reference documentation, specifications.md, pinout.md, and floorplan.md.

6.1 Understanding `chip_top.sv`

The padframe wrapper `chip_top.sv` declares the chip's parameters and ports. The parameter block at the top of the module controls the number of pads of each type, and is the place to change when the pinout changes.

The body of the module is mostly mechanical. One `generate` block per pad type (`vdd_pads`, `vss_pads`, `iovdd_pads`, `iovss_pads`, `inputs`, `outputs`, `bidirs`, `analogs`) instantiates the matching IHP I/O cell, plus single instances for the clock and reset pads. Normally only the parameter counts need to be edited here.

6.2 Understanding `chip_core.sv`

The core `chip_core.sv` is where the macros are instantiated and wired to the chip's I/O. The `enable` signal is mapped to `input_PAD[0]`, the two `counter_top` instances share the clock and reset, the two `inverter_top` instances are wired up (one digital, one analog), and the SRAM output is XOR-reduced to a single bit on `output_PAD[16]`. The output assignments at the bottom of the file are naturally the bit-to-role mapping documented in pinout.md.

6.3 Understanding `config.yaml`

The chip-level LibreLane configuration `config.yaml` covers four sections.

1. **Padframe layout.** The `PAD_WEST`, `PAD_NORTH`, `PAD_SOUTH`, and `PAD_EAST` lists, in the order documented in `doc/pinout.md`.
2. **Floorplan and timing.** `DIE_AREA`, `CORE_AREA`, `CLOCK_PERIOD`, `FP_SIZING`, `PL_TARGET_DENSITY_PCT`.
3. **`MACROS:` block.** For every macro, the GDS, LEF, Liberty libraries, power-pin connections, and placement coordinates (`location`, `orientation`).
4. **PDN parameters.** `PDN_CORE_RING`, ring widths and spacings, `PDN_MACRO_CONNECTIONS`, and the path to `pdn_cfg.tcl`.

For floorplan changes (moving macros, changing the die size, adding new macros), the `MACROS:` section is the one to edit. The rectangular constraints documented in `doc/floorplan.md` apply. Stay inside the core, respect the M3 routing grid for inverters, and never overlap with another macro rectangle.

6.4 Understanding `pdn_cfg.tcl`

`pdn_cfg.tcl` is the OpenROAD callback that builds the power-distribution network. Most of the file is generic boilerplate from LibreLane and OpenLane.

For macro changes, the bottom of the file is the relevant part. It defines per-instance PDN grids and macro connects. The current chip provides the following.

- A default macro grid (TM2 to TM1) used by the two counters.
- A custom grid for `inverter1` and `inverter2` (TM2 to TM1 and TM2 to Metal5, because each inverter macro has its own internal M5 / TM1 ring).
- A custom grid for `sram_0` (Metal5 stripes bridged to Metal4 and TopMetal1, because the SRAM exposes its supplies on M4 and M5).

When new macros are added, the matching `define_pdn_grid` plus `add_pdn_connect` block at the bottom has to be extended (or the default grid is used if the macro layers line up with the chip-level grid).

6.5 Top-Level Simulation

The same RTL and gate-level cocotb flow as in Section 4 is available at chip level.

```
make sim-rtl-cocotb
make sim-gl-cocotb
make sim-view-cocotb                              # GTKWave
make sim-view-cocotb WAVEFORM_VIEWER=surfer       # Surfer
```

The cocotb testbench at chip level is `testbenches/cocotb/chip_top_tb.py`, and it covers the global enable, reset, counters, and bidir behaviour documented in specifications.md.

> **Figure placeholder. `chip_top` cocotb waveform.**

For mixed-signal sign-off, the `testbenches/xschem/chip_top_tb_tran.sch` testbench exists and can be opened in Xschem the same way as for the counter or the inverter.

```
cd testbenches/xschem
xschem
```

> Note
>
> The top-level Xschem testbench converges, but it may take a long time depending on the hardware used. `make sim-gl-xschem` is therefore not part of `sim-all` and must be called manually when needed.

> Tip
>
> Inspect the Xschem symbol of `chip_top` to see the connections defined in `chip_core.sv` visually. Use **`E`** / **`Shift + E`** and **`I`** / **`Shift + I`** to descend into and back out of subcircuits and symbols.
> Note that the SRAM is not in the schematic. IHP does not ship a SPICE model or an Xschem symbol for the `RM_IHPSG13_1P_1024x32_c2_bm_bist` macro, so it is excluded from the schematic-level simulation.

6.6 Building the Chip

When the floorplan and pinout are settled, the chip can be built from scratch with the following command.

```
make build-all
```

This runs the steps in order, `init-submodules`, `build-bondpad`, `build-logos`, `build-macros`, and `build-top`. It is the right starting point on a fresh machine after cloning the repository and entering the IIC-OSIC-TOOLS container.

To re-run only the top-level flow (for example after a change in `config.yaml` or `chip_core.sv`), without rebuilding the bondpad, logos, or macros, run the following command.

```
make build-top
```

Internally, `build-top` chains `librelane-nodrc`, `copy-reports`, `copy-gds`, `copy-netlist`, `copy-render`, `add-logo-fill`, and `render-gds`. The final GDS is written to `layout/chip_top_logo_fill.gds.gz`.

6.7 Viewing the Chip

After a successful build, the chip can be opened with KLayout.

```
ke layout/chip_top_logo_fill.gds.gz   # KLayout edit mode
```

The render generated by `lay2img.py` is in `render/img/`. The sign-off reports (Yosys, antenna, STA, IR-drop, LVS, manufacturability) are in `verification/reports/`.

> **Figure placeholder. `chip_top_logo_fill` layout in KLayout.**

6.8 Chip-Level Verification

DRC, LVS, and PEX are also available at chip level.

6.8.1 DRC

```
make magic-drc CELL=chip_top              # Magic DRC on the bare chip
make magic-drc CELL=chip_top_logo_fill    # Magic DRC including logo + filler
make klayout-drc-minimum                  # fast pre-check KLayout DRC
make klayout-drc-regular                  # full KLayout DRC (slower)
```

`klayout-drc-minimum` is a quick pre-check on the final top-level layout including the logo and the fill structures. `klayout-drc-regular` does the full check, but takes longer.

6.8.2 PEX

```
make magic-pex CELL=chip_top                # only chip_top, default EXT_MODE=1
```

> **Tip**
>
> At chip level, PEX is only run on `chip_top` (without logo and filler), and the default `EXT_MODE=1` (C-decoupled) keeps the runtime manageable.

6.8.3 LVS

```
make magic-lvs                              # top-level Magic + Netgen LVS
```

> **Warning**
>
> The top-level LVS is currently under construction. The infrastructure is in place, but a fully clean run cannot be expected yet.

6.9 Full Pass

The comprehensive simulation, build, and DRC pass for the complete chip is run with the following command.

```
make all
```

This calls `sim-all` (RTL/GL cocotb, GL Xschem when available), then `build-all`, then Magic DRC on both `chip_top` and `chip_top_logo_fill`. Depending on the host hardware, this can take a while.

6.10 Releasing for Tapeout

When the chip is clean, the `release` target copies the GDS, the netlists, and the renders into a versioned folder for tapeout submission.

```
make release VERSION=2.1.0
```

The artifacts are written to release/v.2.1.0/ (`gds/`, `netlist/{layout,pnl,spice}/`, `img/`).

> **Important**
>
> Run `make all` before `make release`. The release target only copies files, it does not rebuild or re-verify the chip.

7 Additional Exercises

The following exercises repeat the corresponding sections of this tutorial with a modification of the design.

7.1 Exercise 1: From 8-bit Up-Counter to 16-bit Down-Counter

Repeat Section 4 for a 16-bit synchronous down-counter that wraps from 0 back to $2^{16} - 1$. The recommended steps are:

1. Update the RTL (`counter.sv`, `counter_top.sv`, `constants.sv`). Change the bit-width and turn the increment block into a decrement, with wrap-around from 0 to `CTR_MAX`.
2. Update the testbenches in `testbenches/verilog/`, `testbenches/cocotb/`, and `testbenches/xschem/` to cover the new behaviour. Adjust the bus width in the waveform view files (`*.gtkw`, `*.surf.ron`) as well.
3. Run `make sim-all` to validate the new RTL.
4. Re-run the LibreLane flow with `make build-top`. Check the timing reports, a 16-bit counter at 50 MHz must still close timing.
5. Regenerate the XSPICE model with `make generate-xspice`, and re-run the gate-level Xschem simulation.

> **Bonus for advanced designers**
>
> Replace the binary counter with a 16-bit Gray counter (only one bit toggles per transition). The same testbench infrastructure can be reused with small changes to the self-check expectations.

7.2 Exercise 2: Inverter to Three-Stage Ring Oscillator

Repeat Section 5 for a three-stage ring oscillator with an output buffer, derived from the existing quad inverter. The recommended steps are:

1. Copy the inverter macro folder to a new location (for example `macros/ringosc/`), and adapt the Makefile names.

2. Update the schematic in Xschem. Connect three inverters in a ring, and tap the output of the last stage through a buffer. The fourth inverter of the quad inverter can naturally be used as the buffer.
3. Update the symbol of the macro (`schematic/xschem/inverter_top.sym`) and the PEX symbol (`schematic/xschem/inverter_top_pex.sym`) to expose a single output net, instead of `vin1..vin4` and `vout1..vout4`.
4. Update the transient testbench (`testbenches/xschem/inverter_top_tb_tran.sch`) to measure the oscillation frequency. Delete the AC, DC, and standalone transient testbenches that are no longer relevant.
5. Optionally (advanced), adapt the CACE `inverter.yaml` to characterise the oscillation frequency across PVT.
6. Re-run the verification chain, `make klayout-verify`, `make magic-verify`, and `make build-top`.

7.3 Exercise 3: Update Pinout and Floorplan of the Top-Level

Building on Exercises 1 and 2, swap `inverter2` for the ring oscillator, delete `counter2`, and connect the now 16-bit `counter1` to the freed-up south-side pads. The recommended steps are:

1. Replace `inverter2` with the ring oscillator macro in `chip_core.sv`.
2. Replace three of the four analog pads with VSS pads, and connect the ring oscillator's output to the fourth analog pad.
3. Delete `counter2` and rewire all eight south-side pads to the new 16-bit `counter1` value.
4. Update the parameters of `chip_top.sv` so that `NUM_OUTPUT_PADS`, `NUM_BIDIR_PADS`, `NUM_ANALOG_PADS`, and `NUM_VSS_PADS` match the new pin count (see Section 6).
5. Update the macro instantiation and wiring in `chip_core.sv`.
6. Update the placement coordinates in the `MACROS:` block of `config.yaml`. Keep the `doc/floorplan.md` constraints in mind, namely stay in the core, no overlaps, and respect the M3 grid for inverter-like macros.
7. Update the PAD lists (`PAD_WEST`, `PAD_NORTH`, `PAD_SOUTH`, `PAD_EAST`) in `config.yaml` to match the new pinout.
8. Update `pdn_cfg.tcl` so that the new macros are covered by the correct PDN grid (default for digital, `inverter_top`-style for analog).
9. Rebuild the chip from scratch with `make build-all`, then verify with `make magic-drc CELL=chip_top` and `make klayout-drc-minimum`.

8 Acknowledgements

This project is funded by the JKU/SAL IWS Lab, a collaboration of Johannes Kepler University and Silicon Austria Labs.

Listing 1 `macros/counter/rtl/counter.sv`

```
// SPDX-FileCopyrightText: 2026 Simon Dorrer and Harald Pretl
// SPDX-License-Identifier: Apache-2.0
// Description: This file implements an N-bit up counter with
// synchronous reset & enable in SystemVerilog.

`default_nettype none
`ifndef __COUNTER__
`define __COUNTER__

module counter #(
  parameter int unsigned       CTR_BW  = 8,
  parameter logic [CTR_BW-1:0] CTR_MAX = 2**CTR_BW - 1
)(
  input logic                clock_i,
  input logic                reset_i,
  input logic                enable_i,

  output logic [CTR_BW-1:0] counter_value_o
);

  // Counter implementation
  always_ff @(posedge clock_i) begin
    if (reset_i) begin
      // Synchronous reset clears the counter value
      counter_value_o <= '0;
    end else if (enable_i) begin
      // Increment the counter value by 1, wrap around at CTR_MAX
      if (counter_value_o == CTR_MAX) begin
        counter_value_o <= '0;
      end else begin
        counter_value_o <= counter_value_o + 1;
      end
    end
  end

endmodule // counter

`endif
`default_nettype wire
```

Listing 2 `macros/counter/rtl/counter_top.sv`

```
// SPDX-FileCopyrightText: 2026 Simon Dorrer and Harald Pretl
// SPDX-License-Identifier: Apache-2.0
// Description: This file implements the top-level wrapper module
// of the counter macro in SystemVerilog.

`default_nettype none
`ifndef __COUNTER_TOP__
`define __COUNTER_TOP__

module counter_top #(
  parameter  int unsigned CTR_MAX = `COUNTER_MAX_DEFAULT,
  localparam int unsigned CTR_BW  = $clog2(CTR_MAX + 1)
)(
  input logic               clock_i,
  input logic               reset_n_i,
  input logic               enable_i,

  output logic [CTR_BW-1:0] counter_value_o
);

  // Internal active-high reset (wrapper handles polarity conversion)
  logic reset;
  assign reset = ~reset_n_i;

  // Embed Counter
  counter #(
    .CTR_BW(CTR_BW),
    .CTR_MAX(CTR_MAX)
  ) counter_0 (
    .clock_i(clock_i),
    .reset_i(reset),
    .enable_i(enable_i),
    .counter_value_o(counter_value_o)
  );
endmodule // counter_top

`endif
`default_nettype wire
```

Listing 3 `rtl/chip_top.sv (excerpt)`

```
module chip_top #(
    // Power / ground pads for digital core / analog front-end
    parameter int unsigned NUM_VDD_PADS    = 1,
    parameter int unsigned NUM_VSS_PADS    = 1,

    // Power / ground pads for I/O
    parameter int unsigned NUM_IOVDD_PADS  = 1,
    parameter int unsigned NUM_IOVSS_PADS  = 1,

    // Signal pads
    parameter int unsigned NUM_INPUT_PADS  = 1,  // excluding clock and reset pads
    parameter int unsigned NUM_OUTPUT_PADS = 17,
    parameter int unsigned NUM_BIDIR_PADS  = 4,
    parameter int unsigned NUM_ANALOG_PADS = 4
) ( ... );
```

Listing 4 `rtl/chip_core.sv (excerpt)`

```
// Counter 1
assign output_out[3:0]  = counter1_value[3:0];

// Counter 2
assign output_out[11:4] = counter2_value;

// Inverter 1 (digital)
assign output_out[12]   = inv1_dout1;
// ...

// SRAM parity
assign output_out[16]   = ^sram_0_out;
```