A Tutorial on Open-Source Analog Mixed-Signal Chip Design with the ihp-sg13cmos5l Open-PDK

Simon Dorrer Harald Pretl

2026-11-18

Table of contents

1 Tutorial Overview **1**
1.1 Guidelines & Cheatsheets . . . 3

2 Tools **3**
2.1 IIC-OSIC-TOOLS Overview . . . 3
2.2 IIC-OSIC-TOOLS Installation . . . 4
2.3 PDK Selection . . . 4

3 Repository Overview **6**
3.1 Purpose . . . 6
3.2 Examples . . . 6
3.3 Directory Structure . . . 6
3.4 Workflow . . . 7
3.4.1 Top-level targets . . . 9
3.4.2 Digital macro targets . . . 10
3.4.3 Analog macro targets . . . 12

4 Digital Macro Implementation **13**
4.1 Directory Structure . . . 13
4.2 Reading the RTL . . . 14
4.2.1 `counter.sv` . . . 14
4.2.2 `counter_top.sv` . . . 14
4.3 Linting . . . 15
4.4 Simulation . . . 15
4.4.1 RTL with SystemVerilog testbench (Icarus Verilog) . . . 15

4.4.2 RTL and gate-level with cocotb . . . 16
4.5 LibreLane Hardening . . . 16
4.5.1 Sizing the Macro . . . 17
4.6 Viewing the Macro . . . 19
4.7 RTL Mixed-Signal Simulation with Xschem . . . 19
4.8 Gate-Level Mixed-Signal Simulation with Xschem . . . 19
4.8.1 The symbol is a source file . . . 20
4.8.2 Running the gate-level testbench . . . 21
4.9 Parasitic Extraction (PEX) . . . 22
4.10 Lint, Build, Verify and Simulate Everything . . . 24

5 Analog Macro Implementation **24**
5.1 Directory Structure . . . 25
5.2 Getting Started with Xschem . . . 26
5.3 Schematic-Level Simulation . . . 27
5.3.1 Anatomy of the AC Testbench . . . 29
5.3.2 Batch Simulation with the Makefile . . . 36
5.4 Schematic-Level Drawing . . . 36
5.4.1 Drawing the Inverter Schematic . . . 36
5.4.2 Drawing the Inverter Symbol . . . 40
5.5 Process Variation and Mismatch (CACE) . . . 41
5.6 Transistor Sizing . . . 43
5.6.1 Design Trade-offs: Choosing g_m/I_D and L . . . 44
5.6.2 Sizing the Inverter . . . 46
5.7 Getting Started with KLayout . . . 48
5.8 Inspect / Edit the Layout . . . 50
5.8.1 Drawing the `inverter` Unit Cell . . . 54
5.9 DRC, LVS, and PEX . . . 54
5.9.1 Design Rule Check . . . 55
5.9.2 Layout-Versus-Schematic . . . 56
5.9.3 Parasitic Extraction . . . 58
5.9.4 Post-Layout Simulation . . . 59
5.9.5 Run Full Verification Flow at Once . . . 60
5.10 Build Deliverables for Top-Level Integration . . . 60
5.11 Full Analog Macro Flow . . . 61

6 Bondpad and Logo Implementation **61**
6.1 Directory Structure . . . 62
6.2 Bondpad Generation . . . 62
6.3 Logo Generation . . . 64
6.4 Top-Level Assembly . . . 67

7 Top-Level Implementation **67**
7.1 Understanding `chip_top.sv` . . . 68
7.2 Understanding `chip_core.sv` . . . 69
7.3 Understanding `config.yaml` . . . 69
7.4 Understanding `pdn_cfg.tcl` . . . 70
7.5 Top-Level Simulation . . . 71
7.6 Building the Chip . . . 76
7.7 Viewing the Chip . . . 79
7.8 Top-Level Verification . . . 79
7.8.1 DRC . . . 79
7.8.2 LVS . . . 82
7.8.3 PEX . . . 82
7.9 Packaging . . . 82
7.10 Full Flow at Once . . . 83
7.11 Releasing for Tapeout . . . 85

8 Start Your Own Chip **85**
8.1 What `make project` Does . . . 86
8.2 After `make project` . . . 87

9 Additional Exercises **88**
9.1 Exercise 1: From 8-bit Up-Counter to 16-bit Down-Counter . . . 88
9.2 Exercise 2: Self-Biased Single-Ended Inverter Amplifier . . . 88
9.3 Exercise 3: Three-Stage Ring Oscillator with Output Buffer . . . 89
9.4 Exercise 4: Update Pinout and Floorplan of the Top-Level . . . 91
9.5 Exercise 5: Update the Bondplan Configuration . . . 91

10 RFIC Flow **92**

11 Acknowledgements **92**

! Important

This tutorial and the underlying template repository require the IIC-OSIC-TOOLS container with tag `2026.09` or later. All commands below must be executed from within the container, and the working directory must be a folder containing the corresponding `Makefile`.

i Note

We happily accept pull requests to fix typos or add content! If you want to discuss something that is not clear, please open an issue!

1 Tutorial Overview

This tutorial is a step-by-step walk-through of the open-source analog mixed-signal (AMS) chip flow based on the `ihp-sg13cmos5l-ams-chip-template` repository for the ihp-sg13cmos5l 130nm Open-PDK. ihp-sg13cmos5l is the CMOS-only 130 nm Open-PDK of IHP's SG13 platform with a five-layer metal stack (`Metal1` to `Metal4` plus the thick `TopMetal1`), so compared to ihp-sg13g2 it has no SiGe HBTs, no `Metal5` and no `TopMetal2`. The folder structure, the Makefile targets and the flow are identical to the SG13G2 sibling of this template, only the PDK-specific parts differ. It covers:

- Overview and Installation of IIC-OSIC-TOOLS
- Digital macro design
  - Design files written in SystemVerilog
  - Linting with Verilator
  - Simulation with Icarus Verilog and cocotb
  - Viewing waveforms with GTKWave and Surfer
  - Synthesis, placement, routing, and verification with Yosys, OpenROAD, and LibreLane
- Analog macro design
  - Schematic entry in Xschem
  - Simulation with Ngspice and VACASK
  - Monte Carlo / mismatch characterisation with CACE
  - Layout in KLayout
  - Verification with DRC, LVS, PEX using KLayout, Magic + Netgen
- Top-level assembly and padframe generation with logos and fill structures using SystemVerilog and the LibreLane flow.
- Starting your own chip from the template with a single `make project` target.

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 power distribution network (PDN), logos, and fill structures

The tutorial ends with a set of exercises that apply the same flow to a modified counter, a ring oscillator using the inverters, and a modified top-level floorplan and pinout. This way, all important steps of the flow are covered in a hands-on manner.

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

> **Note**
>
> After the tutorial, readers should be able to use the template repository for custom silicon projects and understand how to apply the open-source tools in the IIC-OSIC-TOOLS container to design, simulate, build, and verify AMS chips with the ihp-sg13cmos5l Open-PDK.

1.1 Guidelines & Cheatsheets

The following guidelines and cheatsheets should be helpful for the tutorial, for the use of the template repository in general, and for implementing your own custom silicon projects.

- Designer's Guidelines
- Linux Cheatsheet
- SystemVerilog Cheatsheet
- Verilog Cheatsheet
- Xschem Cheatsheet
- Ngspice Cheatsheet
- KLayout Cheatsheet
- KLayout Productivity Suite / Plugins
- LibreLane Cheatsheet
- ihp-sg13cmos5l Layout Calculation Cheatsheet
- ihp-sg13cmos5l LV NMOS / PMOS and HV NMOS / PMOS Techsweeps

2 Tools

Numerous open-source chip design tools cover the entire development workflow for purely digital, purely analog, or mixed analog-digital chips. Managing these many tools regarding installation, updates, and maintenance is complex, especially given the rapid pace of change. Therefore, the Institute for Integrated Circuits and Quantum Computing at Johannes Kepler University (JKU) Linz, Austria, decided to build an all-in-one Docker container (IIC-OSIC-TOOLS) that is updated monthly and supports all essential tools and PDKs. This eliminates the need for the time-consuming individual installation of many tools, and a reproducible development environment (important for reproducing past tapeouts) is available within a GNU/Linux-based virtual machine.

2.1 IIC-OSIC-TOOLS Overview

Figure 1 shows a rough overview of the most important tools in the IIC-OSIC-TOOLS container. Additional tools are also included, which are documented on the associated GitHub page. The open-source AMS flow used in this tutorial relies on several tools, namely Xschem, Magic, Netgen, KLayout, Ngspice, VACASK, CACE, LibreLane, Yosys, OpenROAD, Verilator, cocotb, GTKWave, and Surfer. The container also ships with four PDKs: SkyWater sky130A, GlobalFoundries gf180mcuD, IHP ihp-sg13g2 and IHP ihp-sg13cmos5l used by this template.

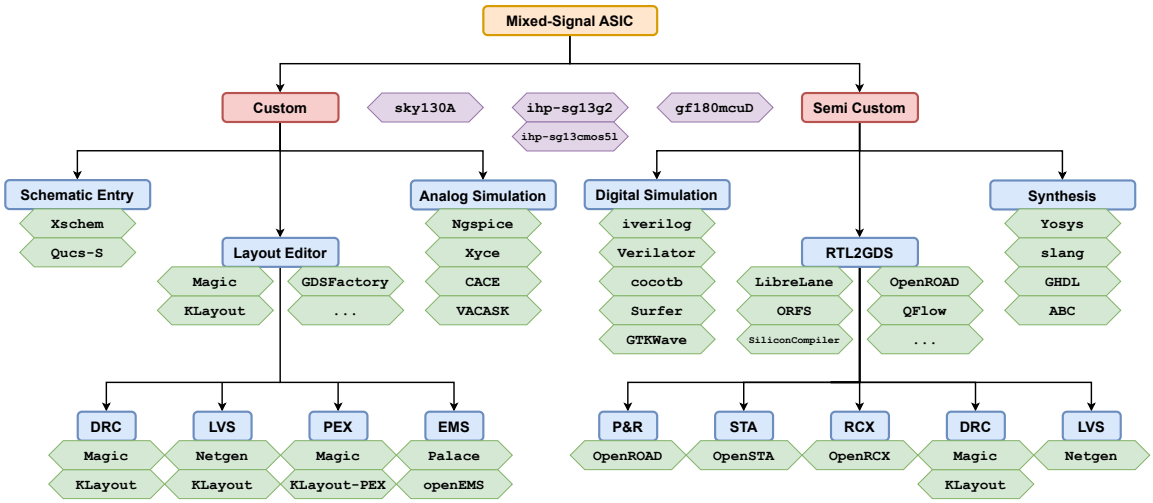


Figure 1: Overview of the most important open-source tools and PDKs shipped with the IIC-OSIC-TOOLS container.

2.2 IIC-OSIC-TOOLS Installation

The container runs on Linux, macOS, and Windows. The installation instructions are documented in the container's `README.md`. For this tutorial, we recommend the quick one-liner installation.

Additional video walk-throughs are available for Linux and Windows.

> **Ubuntu / Linux installation**
>
> - IIC-OSIC-TOOLS installation on Ubuntu

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

2.3 PDK Selection

As mentioned above, the container includes four Open-PDKs. To activate or switch the active PDK, the following two options are available. More detailed instructions can be found in the container's README.md.

1. **`.designinit` file:** Copy the `.designinit` file referencing the ihp-sg13cmos5l 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-sg13cmos5l
   ```

   inside the container. This switches the active PDK for the current session without modifying any file. When called without arguments (`sak-pdk`), a list of installed PDKs is shown.

> **Tip**
>
> By default, the container is configured for `ihp-sg13g2`, so for this template the active PDK has to be switched to `ihp-sg13cmos5l` with one of the two options above before running any Makefile target. The targets take the PDK from the `PDK` and `PDK_ROOT` environment variables, and every `Makefile` checks the selection before it runs anything.

All targets except `help` and `clean` (plus `clean-all` at the top level) compare `$PDK` against `REQUIRED_PDK`, which defaults to `ihp-sg13cmos5l`, and abort before any tool starts:

```
make librelane
```

```
Makefile:27: *** PDK is "ihp-sg13g2", but this chip needs "ihp-sg13cmos5l". Run `sak-pdk ihp
```

The check is a parse-time conditional at the top of each `Makefile`, not a prerequisite of each target, so it costs nothing and cannot be forgotten when a target is added. Without it a wrong PDK mixes technologies silently: LibreLane hardens against the wrong tech LEF and standard cells, DRC and LVS run against the wrong rule deck, Magic reads the wrong `magicrc`, and the results still look valid.

Since this tutorial can also be followed with another PDK, the expected PDK is a variable. Override it, or switch the check off entirely:

```
make <target> REQUIRED_PDK=<pdk>      # require a different PDK
make <target> REQUIRED_PDK=           # skip the check
```

3 Repository Overview

3.1 Purpose

This Makefile-driven repository simulates, builds, and fully verifies (DRC, LVS, PEX) a complete analog mixed-signal chip for the ihp-sg13cmos5l 130nm 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 DRC, LVS and PEX verification
- **SystemVerilog**, **cocotb**, **GTKWave** and **Surfer** for digital simulation

The repository is the starting point for your own custom silicon and provides a universal design flow solution: Just clone the repo, enter the IIC-OSIC-TOOLS container, and run `make all` to get a tapeout-ready analog mixed-signal chip. Focus on your design and do not worry about the tools and the design flow!

Furthermore, it serves as a regression test for the above-mentioned open-source tools and their dependencies using the ihp-sg13cmos5l Open-PDK.

3.2 Examples

Examples based on the SG13G2 sibling of this template are:

- TinyWhisper: An Open-Source Fully-Integrated Multi-Mode Short-Wave Transmitter for Amateur Radio Applications in 130-nm CMOS
- SPARX: An Open-Source, Automated, Programmatically Generated, Frequency-Scalable Six-Port Receiver in 130-nm CMOS
- Multi-Project Chip for the wafer.space gf180mcuD MPW run

3.3 Directory Structure

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

Folder	Purpose
`doc/`	Designer documentation, including specifications, pinout, floorplan, and the PDK and tool cheatsheets.

Folder	Purpose
flow/	LibreLane top-level configuration with config.yaml, pdn_cfg.tcl, chip_top.sdc, plus the ArtistIC logo flow.
ip/	External IP cells including bondpads, custom IO cells, and logos.
layout/	Compressed GDS streams of the top-level chip, including chip_top.gds.gz and chip_top_logo_fill.gds.gz.
LICENSES/	License texts referenced by the SPDX headers and REUSE.toml annotations.
macros/	Recursive-style macros such as counter/ and inverter/, each with its own Makefile and README.
netlist/	Synthesis, schematic, layout, and PEX netlists used for top-level LVS and simulation.
packaging/	Automated bondplan generation: config, flow script, package and bondplan GDS, bonding diagrams, and bond report (see Section 7.9).
release/	Tapeout submission artifacts grouped by version.
render/	Chip and macro renders used in the documentation.
rtl/	Top-level RTL sources, including chip_top.sv and chip_core.sv.
schematic/	Xschem schematics and symbols for the top-level chip.
scripts/	Helper Python and shell scripts for the logo and fill step, the PEX symbol, the port checks, and `make project` (see Section 8).
testbenches/	Xschem and cocotb testbenches for the top-level chip.
tutorial/	This Quarto tutorial and its supporting materials.
verification/	DRC, LVS, and signoff reports.

3.4 Workflow

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 target of every `Makefile` is `help`, so `make` (without arguments) prints the list of available targets. Every `Makefile` checks the active PDK first and stops on a mismatch, see Section 2.3. After entering the container, the starting folder is `/foss/designs/`.

```
git clone https://github.com/iic-jku/ihp-sg13cmos5l-ams-chip-template.git
cd ihp-sg13cmos5l-ams-chip-template
make help              # show available targets at the top level
```

> **Tip**
>
> Instead of `git clone`, you can start by clicking the green “Use this template” button on the GitHub page to create your own copy of the repository. This way, you can push your

changes to your own GitHub repository and even make it public if you want to share your custom silicon design with the community. When the copy is to become your own chip, `make project NAME=<name>` renames it and removes the template-only content, see Section 8.

To find and open a design file, run `make open` in the container's VNC/noVNC desktop. It shows one button per design file of the folder, grouped by directory, and a click opens the file in its tool: Xschem for `.sch` and `.sym`, KLayout for `.gds`, GTKWave for waveforms and gvim for RTL, scripts and configs. The checkboxes select the file types, and the filter field keeps only the paths that contain the typed text. Figure 2 shows the top level with only the schematics and symbols selected. The window is rescanned every 15 s, so files a running flow writes appear on their own. The top-level README lists every file type and option.

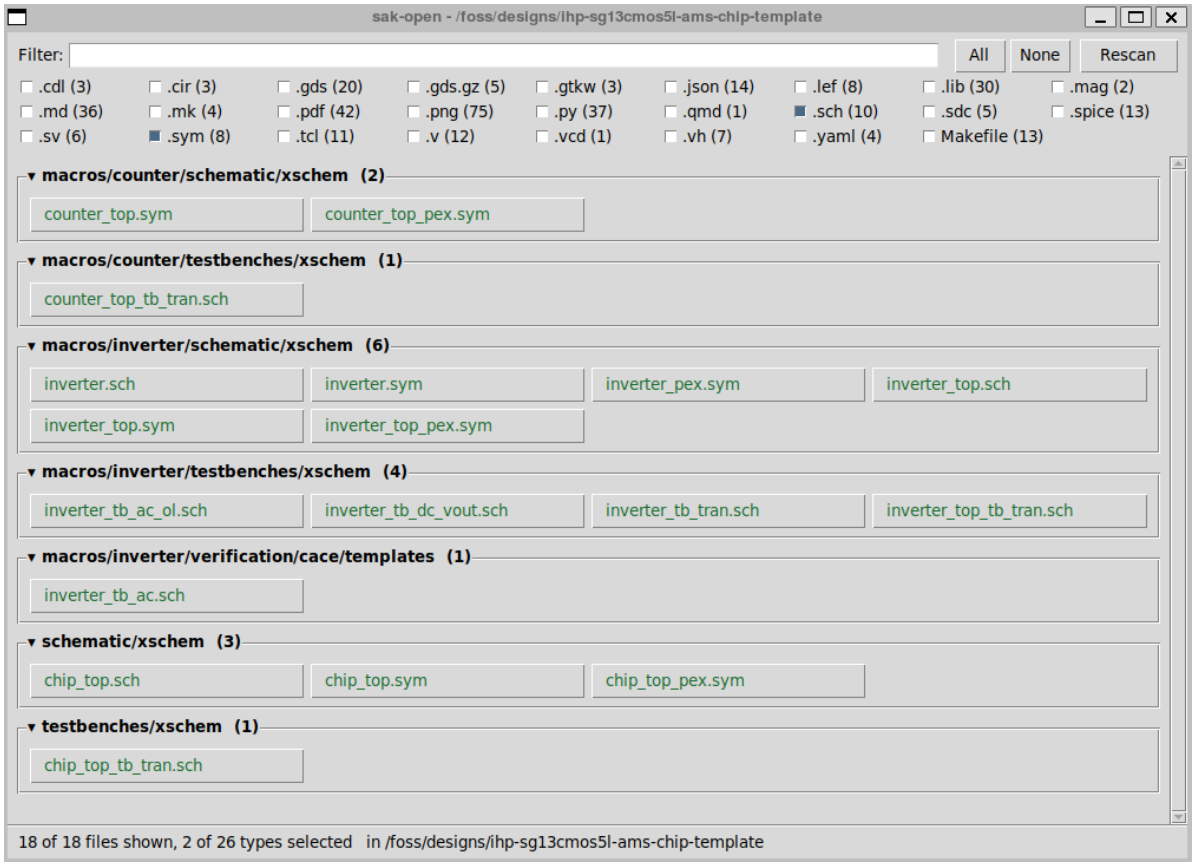


Figure 2: `make open` at the top level after `None` and ticking `.sch` and `.sym`: all 18 schematics and symbols of the chip and both macros, each one click away from Xschem.

Each macro under `macros/` has its own `Makefile` and `README.md`, including macro-specific targets. These targets are invoked by the top-level build flow, but you can also run them

directly from inside each macro folder. Figure 3 shows how the Makefile targets across the repository are connected.

The colour scheme groups targets by the final deliverable they produce, so beginners can follow one colour from `make all` to the generated outputs:

- **Orange (top chip / main)**: This branch includes `make all` and all targets that contribute to the top-level chip GDS. In the `build-top` chain of the top-level Makefile, these are `librelane-nodrc`, `copy-*`, `add-logo-fill`, and `render-gds`.
- **Yellow (bondpad IP)**: This branch includes `build-bondpad` and all targets inside `ip/sg13cmos5l_ip__bondpad_70x70/`.
- **Purple (logos IP)**: This branch includes `build-logos` and the two sub-Makefiles under `ip/sg13cmos5l_ip__jku/` and `ip/sg13cmos5l_ip__jku_names/`.
- **Blue (digital macro)**: This branch includes `build-counter` and the `macros/counter/` Makefile.
- **Green (analog macro)**: This branch includes `build-inverter` and the `macros/inverter/` Makefile.
- **Red (packaging)**: This branch includes `bondplan` and the automated bondplan flow in `packaging/` (see Section 7.9).
- **Grey (simulation and verification)**: This branch includes intermediate targets that feed the coloured deliverables (`sim-all`, `build-all`, `build-macros`, `init-submodules`, `sim-*-cocotb`).

Solid arrows represent direct `$(MAKE) <target>` calls within a single Makefile. Dashed arrows represent calls that descend into a subdirectory: recursive `$(MAKE) -C <dir> all` calls into a sub-Makefile (one per coloured deliverable), or the Python bondplan flow in `packaging/`. The numbers on the second level indicate the execution order of `make all` (build, verify, simulate, package), and the vertical order inside the coloured boxes reflects the execution order of each sub-Makefile's `all` target.

3.4.1 Top-level targets

The top-level Makefile provides:

Target	What it does
`make help`	Print the help banner and list every target with its description.
`make open`	Open the design files of this folder in the `sak-open.py` file browser (needs the VNC/X11 desktop).
`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 (GL) cocotb testbench for the top-level.

Target	What it does
`make sim-gl-xschem`	Run the gate-level Xschem testbench in batch mode (`TB` defaults to `<CELL>_tb_tran`). Part of `sim-all`, but it may take a long time depending on the hardware used.
`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 top-level (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 logos under `ip/`.
`make build-counter` / `build-inverter` / `build-macros`	Build the digital and analog macros individually or together.
`make build-top`	Run LibreLane on the top-level, copy back all artifacts, add logos and fill structures, and render the final GDS.
`make build-all`	Init submodules, build bondpad, build logos, build macros, and build top-level.
`make klayout-verify` / `magic-verify`	Run DRC and LVS with KLayout, or DRC, LVS, and PEX with Magic, for a given cell.
`make bondplan`	Generate the bondplan (die in package, bondwires, pin table) in `packaging/` (see Section 7.9).
`make all`	Full build, DRC verification, simulation, and bondplan generation.
`make release VERSION=2.1.0`	Copy GDS, netlists, and renders into `release/v.<VERSION>/` for tapeout submission.
`make clean` / `make clean-all`	Delete the generated files of the chip top-level, or of the chip top-level plus the IPs and the macros.
`make project NAME=<name>`	Start a new chip from the template: remove the template-only content and rename `chip_top`, `chip_core` and `chip_logo` (see Section 8).

See the top-level README for more details and how to use the Makefile targets.

3.4.2 Digital macro targets

The counter Makefile provides:

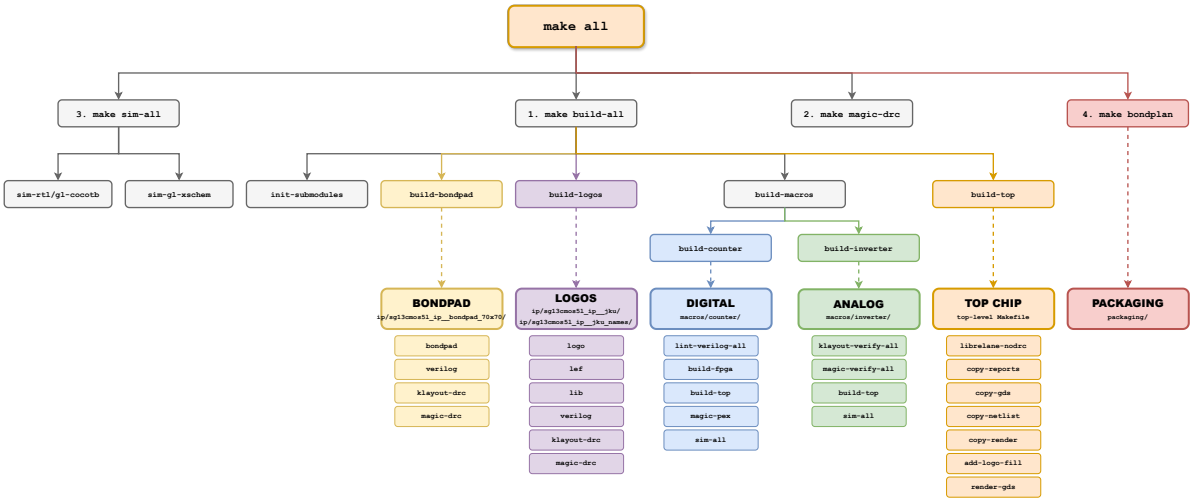


Figure 3: Overview of the Makefile targets that `make all` resolves to. Every coloured branch corresponds to one deliverable (top chip, bondpad, logos, digital macro, analog macro, packaging). The grey targets connect `make all` to those branches.

Target	What it does
`make open`	Open the design files of this folder in the `sak-open.py` file browser.
`make lint-verilog` / `make lint-verilog-all`	Lint a single cell or the full counter design with Verilator.
`make sim-rtl-verilog`	Run the testbench with Icarus Verilog (`<CELL>_tb.sv`, falling back to `<CELL>_tb.v`).
`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 librelane` (and `-nodrc`, `-magicdrc`, …)	Run the LibreLane flow for the counter macro.
`make symbol-gl` (`FORCE=1`)	Scaffold the Xschem symbol `<CELL>.sym` from the ports of `netlist/pnl/<CELL>.pnl.v`. For a new macro, refuses to overwrite an existing symbol.
`make symbol-check`	Check that `<CELL>.sym` still describes those ports. Runs automatically before every XSPICE conversion.
`make generate-xspice`	Convert the LibreLane-extracted SPICE netlist into an XSPICE model for Xschem.
`make sim-gl-xschem` / `sim-view-xschem`	Run the mixed-signal XSPICE simulation in Xschem (batch mode) and plot the results. `TB` defaults to `<CELL>_tb_tran` and `SCRIPT` to `plot_<CELL>`.

Target	What it does
`make build-fpga`	Run the full FPGA flow: lint, synthesis, P&R, and bitstream generation. The board is selected with `BOARD=` and defaults to the pico-ice.
`make build-top`	Run LibreLane, copy reports, netlists, and final views, generate the XSPICE model, and render the final GDS.
`make symbol-pex`	Build the Xschem PEX symbol `<CELL>_pex.sym` from `<CELL>.sym`. Runs automatically before every extraction.
`make klayout-pex` / `magic-pex` (`EXT_MODE=<1|2|3>`)	Extract the parasitics of the hardened macro from `final/gds/` into `netlist/pex/`. `klayout-pex` (KPEX) does not support the PDK yet and currently fails.
`make all`	Run lint, the FPGA build, the full macro build flow, the parasitic extraction, and all simulations.
`make clean`	Delete all generated files and folders of the macro, including the FPGA outputs.

See the counter README for more details and how to use the Makefile targets.

3.4.3 Analog macro targets

The inverter Makefile provides:

Target	What it does
`make open`	Open the design files of this folder in the `sak-open.py` file browser.
`make sim-xschem` / `make sim-view-xschem`	Run an Xschem testbench (batch mode) and plot its results with Python. `TB` defaults to `<CELL>_tb_tran` and `SCRIPT` to `plot_<CELL>`, override with `TB=<tb>` / `SCRIPT=<script>`.
`make sim-cace`	Run CACE Monte Carlo / mismatch characterisations.
`make klayout-drc` / `magic-drc` (`DRC_LEVEL=<precheck|macro|regular>`)	Run DRC using KLayout or Magic.
`make klayout-lvs` / `magic-lvs`	Run LVS using KLayout or Magic + Netgen.
`make symbol-pex`	Build the Xschem PEX symbol `<CELL>_pex.sym` from `<CELL>.sym`. Runs automatically before every extraction.
`make klayout-pex` / `magic-pex` (`EXT_MODE=<1|2|3>`)	Run parasitic extraction in C, CC, or full-RC mode. `klayout-pex` (KPEX) does not support the PDK yet and currently fails.

Target	What it does
`make klayout-verify` / `magic-verify`	Run DRC + LVS (KLayout) or DRC + LVS + PEX (Magic) for a given cell.
`make lef` / `make lib` / `make verilog`	Generate the LEF, the Liberty stub, and the Verilog stub for top-level integration.
`make check-boundary`	Check that the top cell layout carries the PR boundary box on layer 189 that the chip flow needs.
`make build-top`	Build the analog macro deliverables: PR boundary check, LEF, LIB, Verilog stub, GDS and rendered layout image.
`make all`	Run KLayout / Magic verification, the analog macro build flow, and all simulations.
`make clean`	Delete all generated files and folders of the macro, including the CACE outputs.

See the inverter README for more details and how to use the Makefile targets.

4 Digital 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 implemented with SystemVerilog in `counter.sv`. The top-level wrapper `counter_top.sv` 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 Directory Structure

The counter design files are located in `macros/counter/`, and 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 deliverables (GDS, LEF, LIB per corner, NL, PNL, SPEF, VH) copied from the latest LibreLane run for top-level integration.
`fpga/`	Optional FPGA emulation flow for three boards (pico-ice, iCEBreaker, ULX3S).

Folder	Content
netlist/	nl/, pnl/, spice/ and xspice/ netlists (see LibreLane cheatsheet for more details).
schematic/	Xschem symbol of the macro for use in mixed-signal Xschem testbenches.
scripts/	Helper scripts (XSPICE conversion, plotting, layout rendering).
verification/	Yosys, antenna, STA, IR-drop, DRC, LVS, and manufacturability reports.
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 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 logic is implemented in `counter.sv`. It is an N-bit synchronous up-counter with reset and enable, and it wraps at `CTR_MAX`. N defaults to 8, so the counter counts from 0 to 255 and then wraps back to 0.

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 `reset`.

Shared defaults are defined in `constants.sv`. For example, for the final implementation, the maximum counter value is defined in `constants.sv` with the variable `COUNTER_MAX_DEFAULT`. For the testbenches, `CLK_FREQ_DEFAULT` is defined there to set the default clock frequency (e.g. 50 MHz) for the simulations. 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`), follow and understand the counter through reset, hold, increment, and wrap-around.

4.3 Linting

Linting is done with Verilator and can be executed with the following commands.

```
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
```

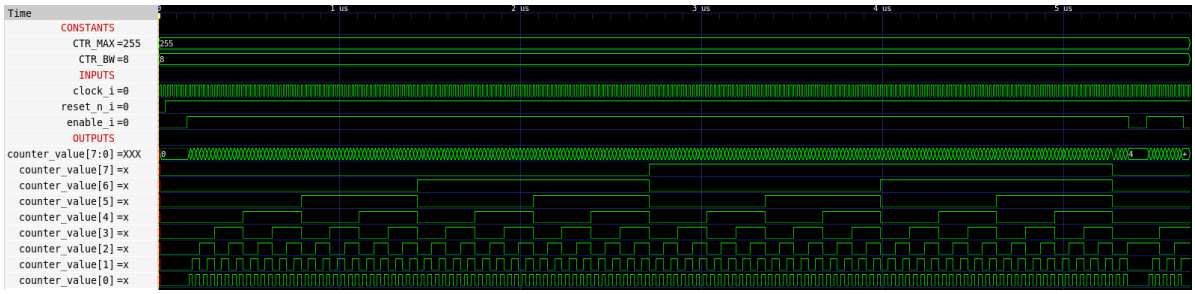


Figure 4: RTL Verilog waveform of `counter_top` in GTKWave.

4.4.2 RTL and gate-level with cocotb

The cocotb testbench (`counter_top_tb.py`) checks the same four cases (reset, hold, increment, and wrap-around) as the SystemVerilog testbench, but implements them in Python.

```
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
```

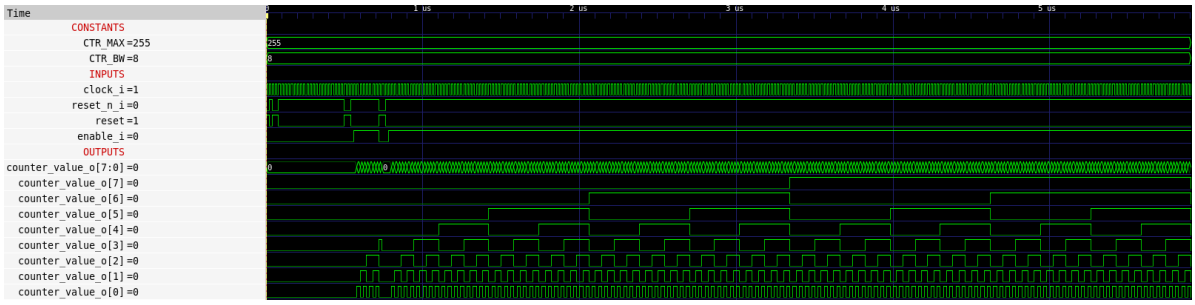


Figure 5: Gate-level cocotb waveform of `counter_top` in GTKWave.

The gate-level cocotb run uses the gate-level Verilog netlist produced after synthesis: `netlist/nl/counter_top.nl.v`. In this tutorial, this netlist for the counter already exists. For new designs, run `make build-top` first to execute the LibreLane flow. It also copies the netlists from the latest LibreLane run back into the source tree with `make copy-netlist`.

4.5 LibreLane Hardening

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

```
make build-top
```

> **Tip**
>
> The macro can also be built as an FPGA bitstream (`fpga/`) with `make build-fpga`. `counter_top` is synthesised straight onto board pins, without a board-top wrapper, and three boards are supported: pico-ice and iCEBreaker (Lattice iCE40) and ULX3S (Lattice ECP5). Pick one with `make build-fpga` for the default pico-ice, or `make -C`

`fpga BOARD=ulx3s all` for another. IIC-OSIC-TOOLS carries the build chain for all three boards, so the bitstreams build inside the container, and only flashing needs the host, since the container has no USB access. Details are in `fpga/README.md`. The FPGA flow is not used in this tutorial and is only listed here for reference.

4.5.1 Sizing the Macro

Three parameters in `config.yaml` decide how compact and how fast the hardened macro is.

- **`DIE_AREA`** is the macro outline `[x0, y0, x1, y1]` in µm (`FP_SIZING: absolute`). The standard-cell rows fill a core area inside it, which is 188 µm × 68 µm for the counter. The aspect ratio matters as much as the area: the outline has to fit the slot of the macro in the top-level floorplan and leave room for the pins listed in `pin_order.cfg`.
- **`PL_TARGET_DENSITY_PCT`** is the cell density that global placement aims for. At 40 %, the placer spreads the cells so that most of the core stays free for the clock tree, the buffers added during timing repair, and the routing. The target should stay above the utilisation (cell area divided by free core area), which global placement prints as `Utilization` in its log. A lower target triggers the warning `[GPL-0302] Target density 0.4000 is too low for the available free area`, and global placement then raises the target to the utilisation on its own.
- **`SYNTH_STRATEGY`** selects the ABC script that Yosys uses for logic optimisation and technology mapping. The `AREA 0` to `AREA 3` scripts favour a small cell area, the `DELAY 0` to `DELAY 4` scripts favour short critical paths. `AREA 0` is the LibreLane default. `AREA 3` and `DELAY 4` always add buffering and gate sizing, so in the test runs below, the counter synthesised to 675 µm² with `AREA 0`, 704 µm² with `DELAY 0`, 720 µm² with `AREA 3`, and 742 µm² with `DELAY 4`. The best strategy for a design is found by trying.

No formula gives the right values up front. They are found iteratively. A run of a small macro takes only a few minutes (`make librelane-nodrc` skips the DRC steps), and the following procedure converges within a handful of runs.

1. **Start large and balanced.** Set a generous `DIE_AREA`, several times the cell area that Yosys reports in `verification/stat.rpt`, and a balanced strategy such as `AREA 0` or `DELAY 0`. This run finishes quickly and gives the reference values for cell area, utilisation, and timing slack.
2. **Shrink the die until placement complains.** Reduce `DIE_AREA` step by step and leave `PL_TARGET_DENSITY_PCT` unchanged. The first complaint is the `[GPL-0302]` warning, which means the die is now too small for the target density. The run can still finish, but shrinking further ends in a hard error, typically `[DPL-0036] Detailed placement failed` once the buffers added by design repair, clock tree synthesis, or timing repair no longer fit.

3. **Step back.** Enlarge the die again until the warning is gone and run the full flow. The result places, routes, and meets timing at the configured density, and it is the starting point for the next step. A denser macro is possible by raising `PL_TARGET_DENSITY_PCT` on purpose, at the cost of less room for routing and timing repair.
4. **Trade timing slack for area.** Read the worst setup slack in the slowest corner (`nom_slow_1p08V_125C`). If there is margin left, switch to a strategy with a smaller cell area (the `Chip area` line in `*-yosys-synthesis/reports/stat.rpt` of the run) and repeat steps 2 and 3, since a smaller cell area allows a smaller die. If setup fails, move towards `DELAY`, up to `DELAY 4`, or give the placer more room.

Each iteration changes only two values and reads back a few numbers, so the loop can also be scripted or handed to a coding agent. After every run, check the following in the run folder `flow/librelane/runs/<run>/`:

- `warning.log`: the `[GPL-0302]` warning, among others.
- `final/metrics.csv`: `design__instance__utilization`, `timing__setup__ws`, `timing__hold__ws`, and `route__drc_errors`.
- `*-openroad-stapostpnr/summary.rpt`: worst setup and hold slack and the violation counts in all six STA corners. `make copy-reports` copies it to `verification/stapostpnr_summary.rpt`.

The counter sits far from these limits. In test runs at the configured 200 µm × 100 µm, global placement reported a utilisation of 6.3 % against the target of 40 %, and the worst setup slack was 13.4 ns of the 20 ns clock period. At the same target density, the counter placed without the warning at 100 µm × 60 µm (34 % utilisation), warned but still routed and met timing at 60 µm × 60 µm (62 %), and failed detailed placement at 55 µm × 60 µm (70 %). The runs that finished kept 13.4 to 13.5 ns of setup slack. Switching to `AREA 0` was not enough to rescue the 55 µm × 60 µm die. Repeating these runs is a good first exercise.

> **Note**
>
> The configuration sets `GRT_ALLOW_CONGESTION: true`, so global routing continues despite overflow. A die that is too small to route then shows up as detailed routing DRC errors, and the `Checker.TrDRC` step stops the flow.

> **Important**
>
> A new die size changes the LEF of the macro. Update its placement in the `MACROS` block of the top-level `config.yaml` and in `doc/floorplan.md` (see Section 7). The copy targets take the newest run in `flow/librelane/runs/`, so finish with `make build-top` on the final values.

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). With the following command, the final GDS from the latest LibreLane run is opened in KLayout. With the keys `0`, `1`, `2`, …, `9` you can view different layers as explained in the KLayout cheatsheet or in the klayout-layer-shortcuts plugin.

```
ke final/gds/counter_top.gds
```

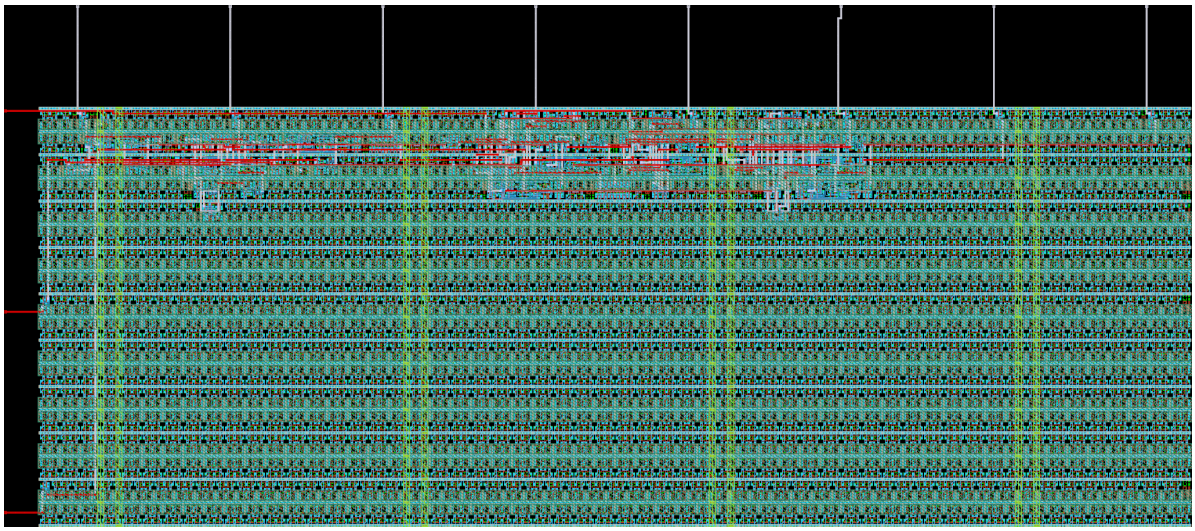


Figure 6: `counter_top` layout in KLayout (200 µm × 100 µm).

The LibreLane render is copied to `render/img/counter_top_librelane.png`, and the high-resolution renders from `sak-render.py` are saved as `counter_top_white.png` and `counter_top_black.png` in the same folder. The verification reports are written to `verification/`.

4.7 RTL Mixed-Signal Simulation with Xschem

For an introduction to Ngspice and Verilog RTL co-simulation with Xschem, see Stefan Schippers' video: Ngspice + Verilog Co-Simulation in Xschem. This RTL mixed-signal simulation is currently not included in this repository, but it will be added in a future update.

4.8 Gate-Level Mixed-Signal Simulation with Xschem

A more accurate validation step is the gate-level mixed-signal simulation in Xschem with the Ngspice XSPICE event-driven engine. The LibreLane-extracted SPICE netlist

(`netlist/spice/<CELL>.spice`, copied from the latest run by `copy-netlist`) is converted into an XSPICE model with the following command, which is also executed automatically as part of `make build-top`.

```
make generate-xspice
```

The conversion pipeline is `.spice → .xspice` (`spi2xspice.py`), followed by a pin-reorder pass with `sak-pin-reorder.py` (a SAK tool installed in the IIC-OSIC-TOOLS container) to match the Xschem symbol (`schematic/xschem/counter_top.sym`).

> **Note**
>
> The pins are matched **by name**: every pin in the Xschem symbol carries a `sim_pinname=<netlist_name>` property, so the alphabetically sorted ports of the extracted netlist are mapped correctly regardless of their order. Since `make build-top` regenerates the XSPICE model right after copying the netlists, the model always matches the current LibreLane run.

4.8.1 The symbol is a source file

The macro is hardened, so it has no schematic, and Xschem's "make symbol from a schematic" (key `a`) has nothing to work from. `schematic/xschem/counter_top.sym` is drawn by hand, and everything downstream bends to it: the pin order of the XSPICE model and of the extracted PEX netlists is sorted to match it. It is therefore never regenerated by a build. The testbench wires to pin **coordinates**, and a pin one grid step off its wire floats without any error, so a build that moved pins would quietly break the simulation instead of failing it.

Two targets support that arrangement without taking ownership of the file away from you.

```
make symbol-gl      # scaffold schematic/xschem/<CELL>.sym, for a new macro
make symbol-check   # verify the committed symbol, runs inside generate-xspice
```

`symbol-gl` reads the ports of the powered netlist `netlist/pnl/<CELL>.pnl.v` and writes a first-cut symbol in which every port is a pin with its direction, its bus range and its `sim_pinname`. The geometry follows Xschem's own symbol generator, so loading the result and saving it again returns it unchanged. It is a scaffold: rename the pins to house style (`clock_i` becomes `di_clock`, `counter_value_o[0..7]` becomes `do_b[0..7]`), arrange them, draw the body, then commit it. Renaming is safe because `sim_pinname` holds the binding to the netlist. The target refuses to overwrite an existing symbol, because that is where the hand work lives.

`symbol-check` is the half that runs in the build. It compares the committed symbol against the same netlist and fails if a port was added, removed, renamed, given a different width, or drawn

with the wrong direction. Its most useful check is that **every pin declares `sim_pinname`**: a single pin missing the property drops `sak-pin-reorder.py` out of name matching and into matching by position, which is wrong for an alphabetically sorted netlist and can mis-wire the macro without any error at all.

> **Tip**
>
> For a new digital macro the order is: harden it once, `make copy-netlist`, `make symbol-gl`, arrange the symbol in Xschem, and only then wire the testbench to it and run `make all`.

4.8.2 Running the gate-level testbench

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

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

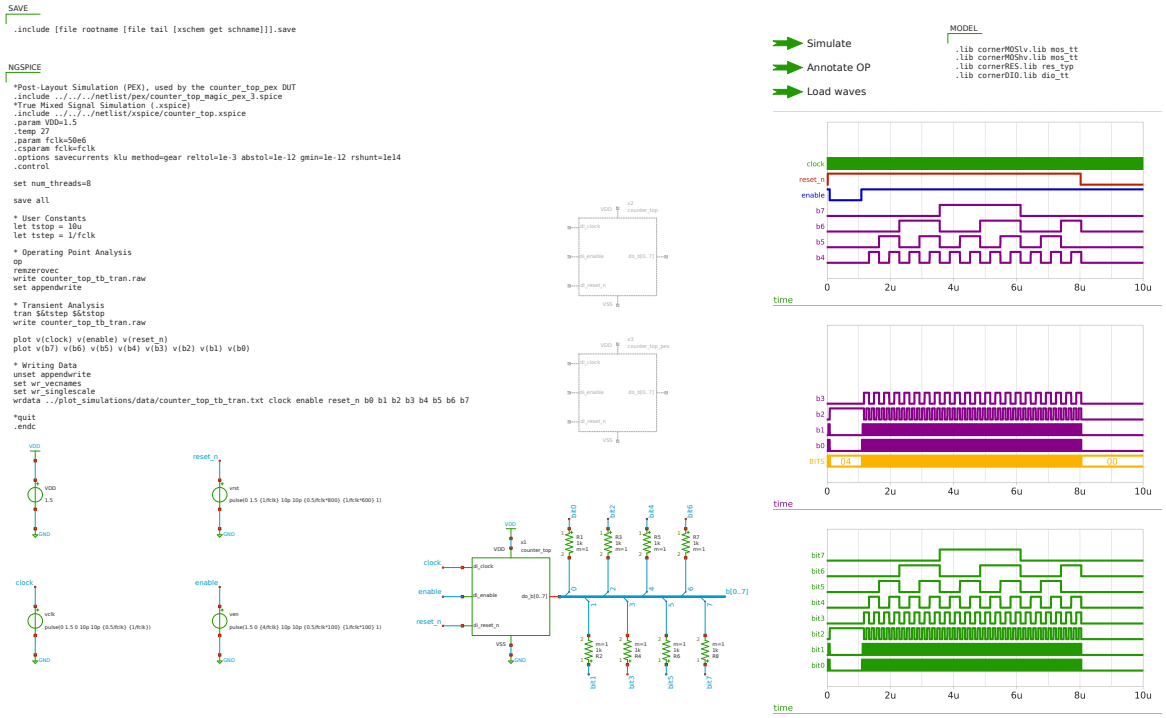


Figure 7: Xschem mixed-signal gate-level testbench `counter_top_tb_tran`.

In Xschem, hold **Ctrl** and click the **Simulate** button in the schematic to launch the simulator. Once the simulation has finished, you can hold **Ctrl** and click the **Load waves** arrow to load the simulation results into the integrated waveform viewer.

> **Tip**
>
> This testbench also shows multiple ways to connect and display bus signals in Xschem. For more information, refer to the Xschem documentation: Use Bus/Vector notation for signal bundles / arrays of instances.

> **Note**
>
> The green 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. This is also the beauty of Xschem and the reason why it is possible to also run simulations non-interactively, generate netlists, and run LVS with only Makefile targets, as you will see below.

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 SCRIPT=plot_counter_top
```

sim-gl-xschem runs in batch mode: it netlists the testbench with **xschem netlist** and then calls **ngspice -b** directly, so **make** blocks until the run finishes and sees its exit status. The testbench is selected with **TB=<testbench_name>** (default: **<CELL>_tb_tran**). Since the run is headless, the **plot** commands in the testbench's **.control** block are a no-op. The testbench exports its results with **wrdata** to **testbenches/xschem/plot_simulations/data/**, from where **sim-view-xschem** plots them with Matplotlib.

4.9 Parasitic Extraction (PEX)

The XSPICE model above is a *gate-level* view of the macro: every standard cell becomes an XSPICE primitive driven by its Liberty delay, and the wiring is ideal. Parasitic extraction gives the *transistor-level* view instead. It reads the hardened layout **final/gds/counter_top.gds** and writes a SPICE netlist into **netlist/pex/** in which every standard cell is flattened to transistors and the wiring carries the resistances and capacitances measured from the layout.

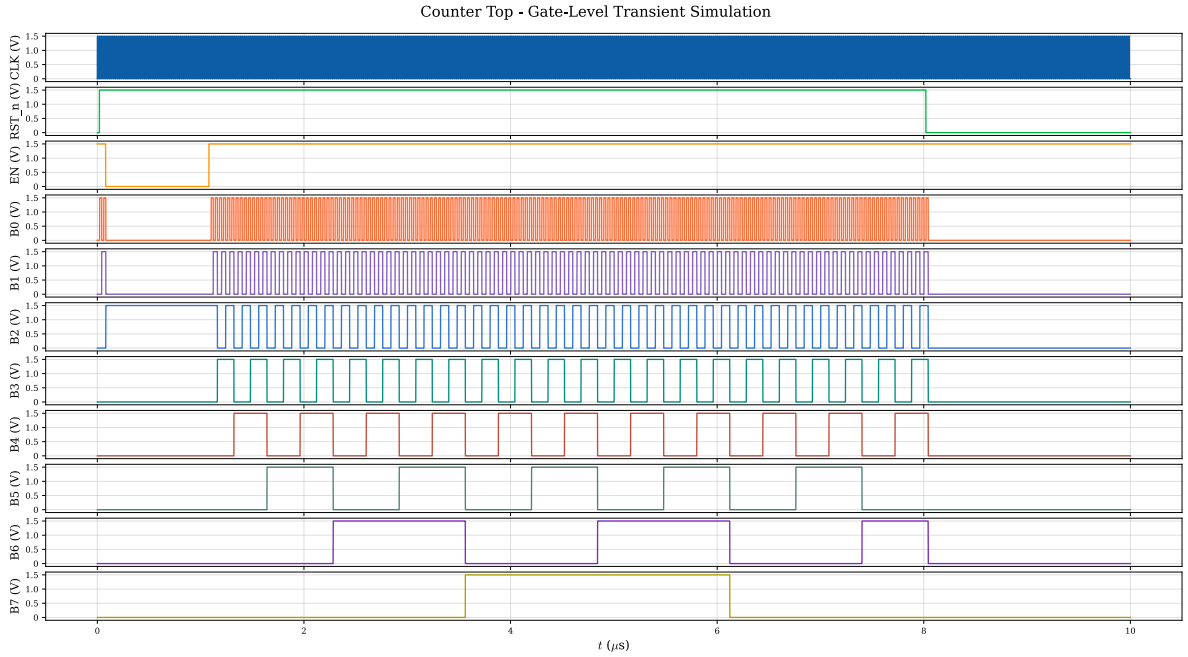


Figure 8: Plotted Xschem simulation results of `counter_top_tb_tran` from `testbenches/xschem/plot_simulations/plot_counter_top.py`.

```
make magic-pex                                  # full-RC, the default EXT_MODE=3
make magic-pex EXT_MODE=1                       # C-decoupled, fastest
make magic-pex EXT_MODE=3 THRESHOLD=5000 MINRES=500 MINDELAY=2   # full-RC with custom extres
```

> **Warning**
>
> `make klayout-pex` runs KLayout-PEX (KPEX), which does not support the `ihp-sg13cmos5l` PDK yet, so the target currently fails. Use `magic-pex` until KPEX gains SG13CMOS5L support.

Both targets read `final/gds/<CELL>.gds`, so `make build-top` (or at least `make copy-final`) has to have run first, and both abort with a clear message if it is missing. They start by running `make symbol-pex`, which derives `schematic/xschem/counter_top_pex.sym` from `counter_top.sym` so the PEX symbol can never go stale, and they finish by reordering the extracted `.subckt` ports to that symbol's pin order with `sak-pin-reorder.py` and verifying them with `scripts/check_pex_ports.py`. For `counter_top`, the full-RC default extracts 4351 transistors and roughly 2100 capacitors and 4700 resistors. Magic's `extresist` step is not deterministic, so the exact R and C counts move slightly from run to run while the transistor count stays the same.

The transient testbench already `.includes` the extracted netlist next to the XSPICE model, and it carries two spare DUT instances above the schematic, `x2` of `counter_top.sym` and `x3` of `counter_top_pex.sym`, both parked with `spice_ignore=true`. Swapping the wired-up `x1` for `x3` turns the run into a post-layout simulation.

> **Warning**
>
> A post-layout run simulates every transistor of the macro. In the IIC-OSIC-TOOLS container, 60 ns of transient take about 80 s, while the full 10 µs gate-level XSPICE run of the same testbench finishes in under 2 s. The XSPICE model stays the right tool for functional runs. Use the extracted netlist on short, targeted stimuli to check timing and signal integrity.

> **Note**
>
> The testbench uses looser Ngspice tolerances than the analog macro (`reltol=1e-3 abstol=1e-12 gmin=1e-12 rshunt=1e14`). Full-RC extraction splits the nets into fragments, and some of the resulting nodes have no DC path to ground, so with the analog macro's settings the post-layout run aborts at the initial timepoint with `Timestep too small`. The gate-level XSPICE run gives the same result either way.

4.10 Lint, Build, Verify and Simulate Everything

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

```
make all
```

This calls `lint-verilog-all`, `build-fpga`, `build-top`, `magic-pex`, and `sim-all` in sequence. The simulations run after the build, so the gate-level simulations run on the netlists, the XSPICE model and the extracted netlist produced by this build. `klayout-pex` is commented out in the recipe, since it is a second full extraction that nothing in the flow consumes, and KPEX does not support the `ihp-sg13cmos5l` PDK yet anyway. DRC and LVS are already covered by the LibreLane verification steps, so no extra verification target is needed for the digital macro.

5 Analog Macro Implementation

The `inverter` macro is the analog reference macro of the template. It is a 4-channel CMOS inverter, drawn at transistor level, and verified with a comprehensive verification chain:

- Layout vs. Schematic (LVS) with KLayout or Magic + Netgen,
- Design Rule Checking (DRC) with KLayout or Magic,
- Parasitic Extraction (PEX) with Magic (KPEX, the KLayout extractor, does not support the PDK yet),
- Monte Carlo simulation of process and mismatch variations with CACE.

Two instances are used in the chip core. One is used as a 4-channel digital inverter (`inverter1`), and the other as a 2-channel analog inverter (`inverter2`). See the chip specifications for the full picture.

> **Note**
>
> An inverter can act as a digital circuit (when the input is driven by a digital signal and the output is switching between the supply rails). Here, we characterise the inverter as an analog circuit, meaning that we are interested in its continuous voltage transfer characteristic and its small-signal voltage gain.

Figure 9 shows the circuit of one channel. The NMOS M_1 and the PMOS M_2 form the complementary inverter. The transistors M_{dummy1} and M_{dummy2} are dummy devices. They do not contribute to the circuit function but improve matching in the layout, because the active transistors then see identical neighbouring structures on both sides.

5.1 Directory Structure

The inverter design files are located in `macros/inverter/`, and the relevant folders are:

Folder	Content
`schematic/xschem/`	Xschem schematics and symbols (`inverter.sch`, `inverter.sym`, `inverter_top.sch`, `inverter_top.sym`, and the generated `inverter_top_pex.sym`).
`layout/`	The unit cell `inverter.gds`, the editing source `inverter_top.klay.gds` with its library map `inverter_top.klay.klib`, and the exported `inverter_top.gds`.
`testbenches/xschem/`	Transient, ac open-loop, dc input sweep, and top-level transient testbenches.
`netlist/`	`schematic/` (CDL for KLayout LVS / SPICE for Magic + Netgen LVS), `layout/` (extracted for LVS), `pex/` (SPICE).
`scripts/`	Sizing notebooks (`sizing/`), the PR boundary check (`check_boundary.py`), and the PEX port check (`check_pex_ports.py`).
`verification/`	`drc/`, `lvs/`, and `cace/` (process, mismatch, supply sweeps).

Folder	Content
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 Getting Started with Xschem

Xschem is a schematic capture tool where circuits are built from symbols, wires, and net labels. Everything is stored in plain text files (`.sch` for schematics, `.sym` for symbols), which makes them friendly for version control and search-and-replace in a text editor.

Always start Xschem from the schematic or testbench folder of the macro, so that the local `xschemrc` is picked up. It sources the PDK configuration, adds the macro's schematic and testbench folders to the symbol search path, and pins the netlist directory to `testbenches/xschem/simulations/`, where Ngspice runs and writes its results.

```
cd macros/inverter/schematic/xschem
xschem                  # empty session
xschem inverter.sch     # or load a schematic directly at startup
```

The most important shortcuts are listed in Table 1. A complete list is available in Xschem under `Help > Keys` and in the Xschem Cheatsheet.

Table 1: Essential Xschem shortcuts.

Shortcut	Action
`Ins`	Insert a symbol from the component library
`w`	Draw a wire
`m`	Move the selected objects
`c`	Copy the selected objects
`Del`	Delete the selected objects
`q`	Edit the properties of the selected object
`Shift + R`	Rotate the selection
`Shift + F` / `Shift + V`	Mirror the selection horizontally / vertically
`u` / `Shift + U`	Undo / redo

Shortcut	Action
`f`	Zoom full
`Ctrl + s`	Save
`Ctrl + o`	Open a schematic or symbol
`e`	Descend into the schematic of the selected symbol
`i`	Descend into the symbol view of the selected symbol
`Ctrl + e`	Return to the parent schematic
`a`	Make a symbol from the current schematic
`Shift + T`	Toggle the `spice_ignore` flag on the selected instances (excluded from the netlist)
`n`	Generate a netlist

> **Tip**
>
> Since Xschem is a text-based schematic editor, you can also open `*.sch` and `*.sym` files in a text editor and modify them there. This is especially useful for search-and-replace operations.

5.3 Schematic-Level Simulation

Go to the Xschem testbench folder and start Xschem.

```
cd macros/inverter/testbenches/xschem
xschem
xschem <testbench_name>.sch # load a testbench schematic directly at startup
```

Four testbenches are available in `testbenches/xschem/`.

- `inverter_tb_tran.sch`, single-channel transient response (see Figure 10),
- `inverter_tb_ac_ol.sch`, open-loop ac characterisation (see Figure 11),
- `inverter_tb_dc_vout.sch`, dc input sweep (see Figure 12),
- `inverter_top_tb_tran.sch`, top-level transient on `inverter_top` (see Figure 13).

Load a testbench in Xschem by pressing **`Ctrl + o`**. Each testbench is operated with the three launcher buttons in the schematic. Hold **`Ctrl`** and click **Simulate** to netlist the schematic and start Ngspice. Once the simulation has finished, hold **`Ctrl`** and click **Load waves** to load the results into the integrated waveform viewer (the embedded graphs), or **Annotate OP** to back-annotate the operating point into the schematic. **Simulate** also regenerates the FET

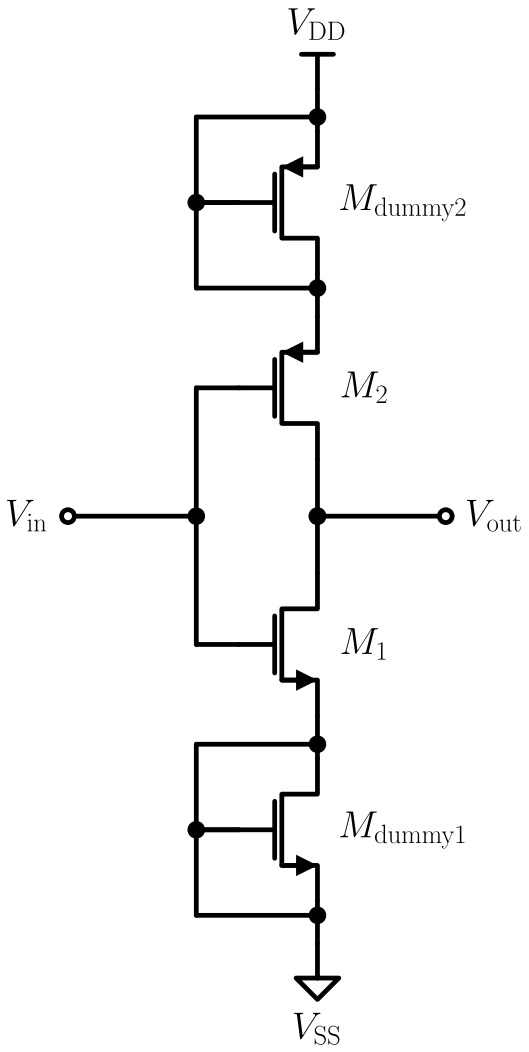


Figure 9: The CMOS inverter circuit with dummy transistors.

`.save` file described in Section 5.3.1, so the operating-point annotation always matches the devices currently in the schematic. The following figures show the testbench schematics, the plotted results follow in Section 5.3.2.

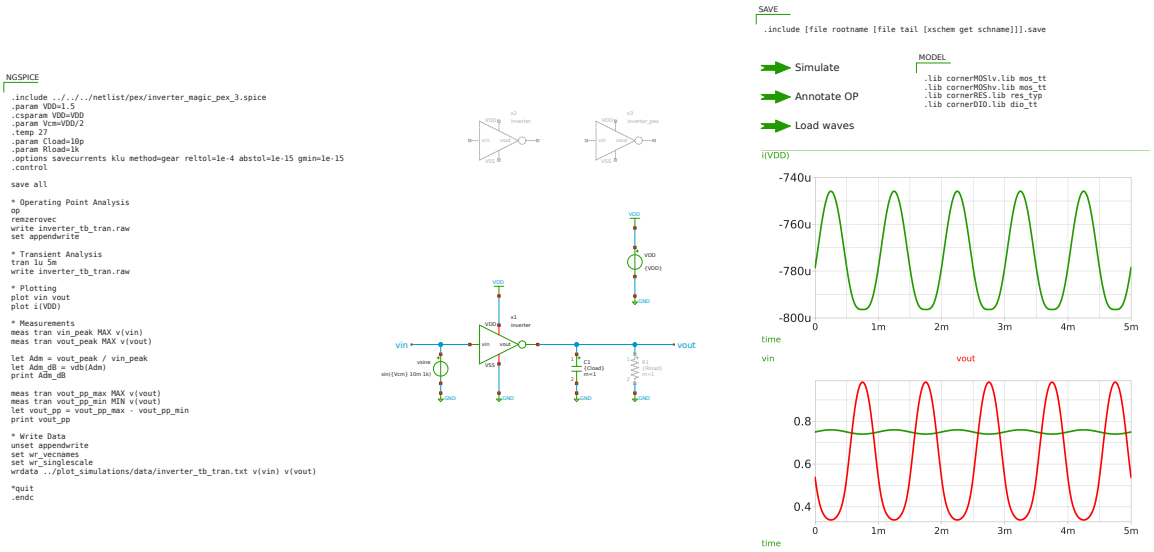


Figure 10: Xschem testbench `inverter_tb_tran` (single-channel transient response on `inverter`).

5.3.1 Anatomy of the AC Testbench

We look at the ac testbench in detail, because it contains every ingredient a complete analog testbench needs. Open `inverter_tb_ac_ol.sch` (see Figure 11) and identify the following parts:

1. The device under test. An instance `x1` of `inverter.sym`. Two spare instances `x2` (`inverter.sym`) and `x3` (`inverter_pex.sym`) are also placed, with the attribute `spice_ignore=true` so they are excluded from the netlist. They are used later for the post-layout comparison in Section 5.9.4.

2. Supplies and bias. A voltage source `VDD` with `value={VDD}` powers the circuit between the `VDD` rail and ground. The input source `vin` has the value

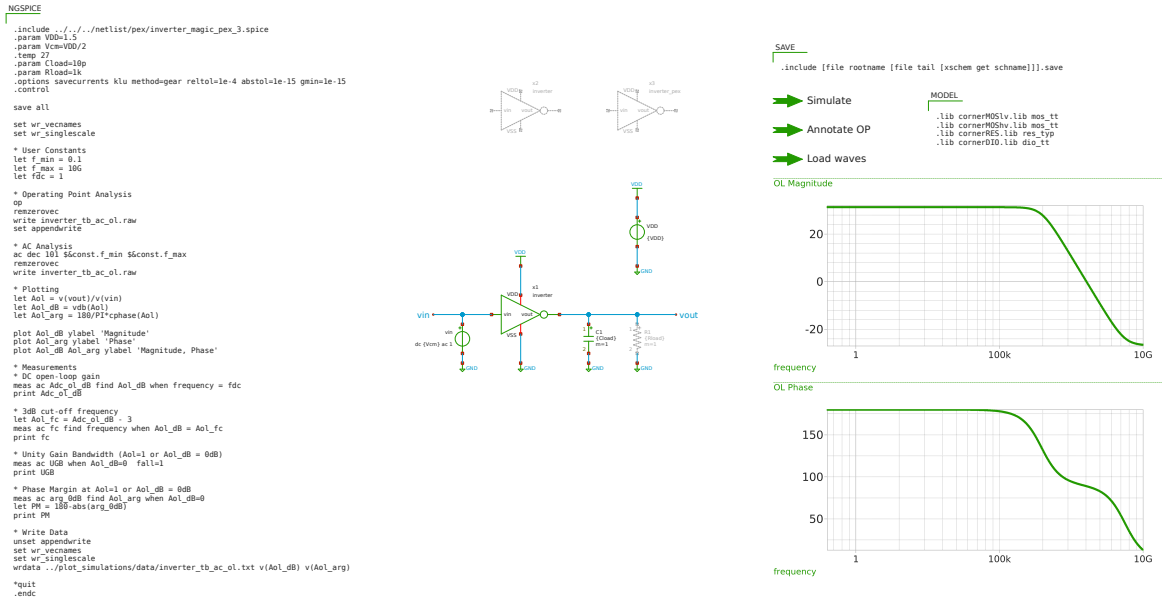


Figure 11: Xschem testbench `inverter_tb_ac_ol` (open-loop ac characterisation on `inverter`).

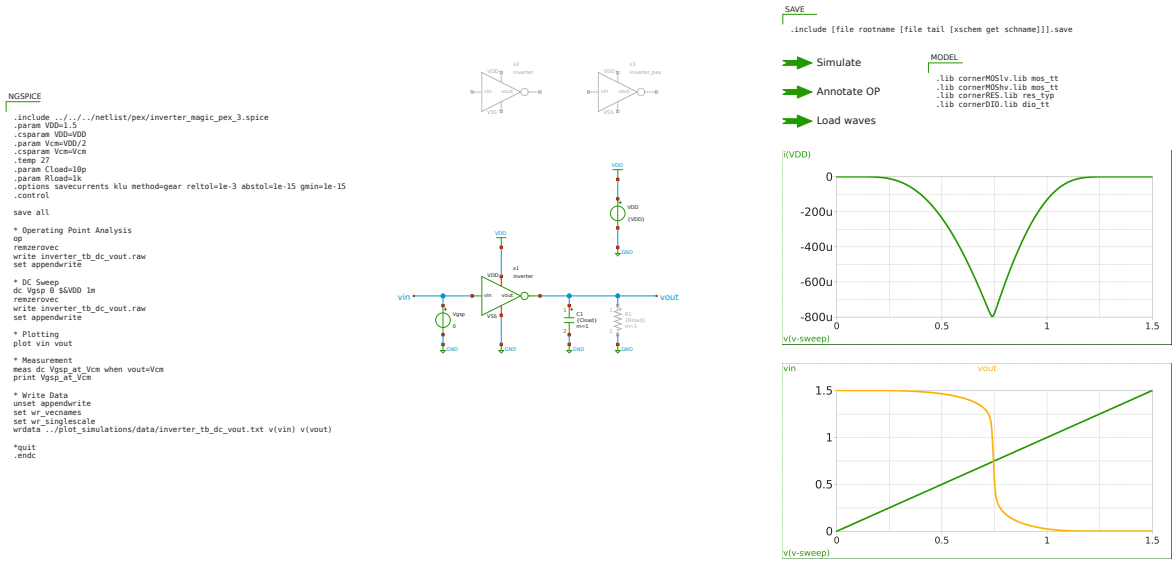


Figure 12: Xschem testbench `inverter_tb_dc_vout` (dc input sweep on `inverter`).

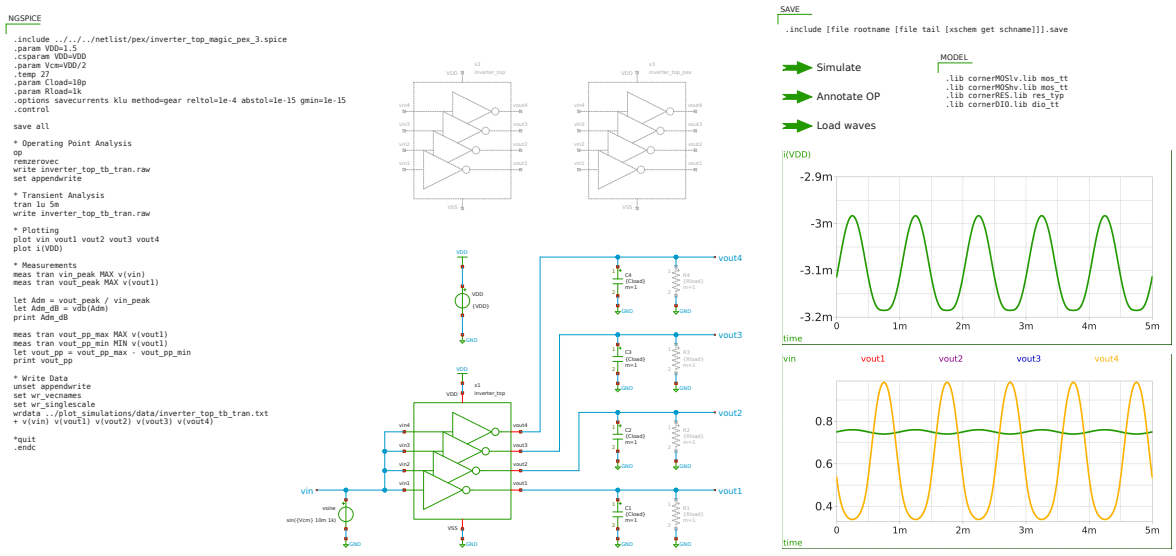


Figure 13: Xschem top-level testbench `inverter_top_tb_tran` (transient on `inverter_top`).

```
dc {Vcm} ac 1
```

The `dc {Vcm}` part biases the input at the switching threshold $V_{\mathrm{cm}} = V_{\mathrm{DD}}/2$, where the inverter has its highest gain. The `ac 1` part sets the small-signal stimulus amplitude to 1 V. This does not overdrive the circuit, because the ac analysis linearises the circuit around the operating point. With an input of exactly 1, the output magnitude directly equals the voltage gain, so `v(vout)/v(vin)` is a normalised transfer function.

3. The load. A capacitor `C1` with `value={Cload}` (10 pF) loads the output. A load resistor `R1` with `value={Rload}` is placed as well but disabled with `spice_ignore=true`. Toggle it with `Shift + T` if you want to study resistive loading.

4. The model includes. A `code_shown.sym` block named `MODEL` pulls in the PDK device models with the typical corner:

```
.lib cornerMOSlv.lib mos_tt
.lib cornerMOShv.lib mos_tt
.lib cornerRES.lib res_typ
.lib cornerDIO.lib dio_tt
```

The bare file names work because the container points Ngspice at the `.spiceinit` of the active PDK (`$PDK_ROOT/$PDK/libs.tech/ngspice/.spiceinit`), which adds the PDK model directory to the Ngspice source path and loads the compiled PSP OSDI models. To simulate a different process corner, replace `mos_tt` with, for example, `mos_ss` or `mos_ff`.

5. The control block. A second `code_shown.sym` block named `NGSPICE` contains the parameters, simulator options, and the `.control` section:

```
.include ../../../netlist/pex/inverter_magic_pex_3.spice
.param VDD=1.5
.param Vcm=VDD/2
.temp 27
.param Cload=10p
.param Rload=1k
.options savecurrents klu method=gear reltol=1e-4 abstol=1e-15 gmin=1e-15
.control
save all

* User Constants
let f_min = 0.1
let f_max = 10G
let fdc = 1
```

```
* Operating Point Analysis
op
remzerovec
write @schname.raw
set appendwrite

* AC Analysis
ac dec 101 $&const.f_min $&const.f_max
remzerovec
write @schname.raw
...
.endc
```

Read it top to bottom:

- The `.include` line loads the extracted PEX netlist from `netlist/pex/`, which defines the subcircuit `inverter_pex`. It is not used while `x1` is active, but it must be present for the post-layout swap in Section 5.9.4. The extracted netlists are tracked in the repository, so a fresh clone simulates right away, and `make magic-pex CELL=inverter` refreshes them (see Section 5.9.3).
- The `.param` lines define the supply, the input bias, and the load as parameters, so one edit changes the whole testbench. The `.temp` line sets the simulation temperature in °C. It is an Ngspice directive of its own, not a parameter.
- The `.options` line selects the KLU sparse solver and tightens the tolerances for a clean ac result.
- Inside `.control`, first an operating point analysis (`op`) runs and its result is written to the `.raw` file. This is what **Annotate OP** reads.
- Then the ac analysis sweeps 101 points per decade from 0.1 Hz to 10 GHz and appends the result to the same `.raw` file.

6. The measurements. After the analysis, the control block computes the open-loop gain and extracts the key numbers:

```
let Aol = v(vout)/v(vin)
let Aol_dB = vdb(Aol)
let Aol_arg = 180/PI*cphase(Aol)

* DC open-loop gain
meas ac Adc_ol_dB find Aol_dB when frequency = fdc

* 3dB cut-off frequency
let Aol_fc = Adc_ol_dB - 3
```

```
meas ac fc find frequency when Aol_dB = Aol_fc

* Unity Gain Bandwidth (Aol=1 or Aol_dB = 0dB)
meas ac UGB when Aol_dB=0 fall=1

* Phase Margin at Aol=1 or Aol_dB = 0dB
meas ac arg_0dB find Aol_arg when Aol_dB=0
let PM = 180-abs(arg_0dB)
```

The phase starts at 180° because the inverter is an inverting amplifier. With the sizing from Section 5.6 you should measure approximately $A_{\mathrm{ol}} = 31.3\,\mathrm{dB}$, $f_{\mathrm{c}} = 2.9\,\mathrm{MHz}$, and UGB = 107 MHz (see Figure 14).

7. The data export. The final `wrdata` command writes the traces to `plot_simulations/data/`, so the Python plotting script and the batch flow from Section 5.3.2 can use them.

8. Graphs and the save file. The embedded graph boxes display magnitude and phase after **Load waves**. A small `SAVE` code block pulls in the FET save file:

```
.include [file rootname [file tail [xschem get schname]]].save
```

The Tcl expression evaluates to the bare file name of the current schematic, for example `inverter_tb_ac_ol.save`, so the include resolves inside `testbenches/xschem/simulations/`, where Ngspice runs. That file holds a `.save` line for every transistor parameter (`ids`, `gm`, `gds`, `vth`, …), which is what the `annotate_fet_params` symbols inside the inverter and the **Annotate OP** launcher read back from the `.raw` file. Without it, Ngspice would not store these internal device quantities and the annotation would stay empty.

You never have to create the file by hand: both the **Simulate** launcher and `make sim-xschem` (see Section 5.3.2) write it before every run with `write_data [save_params]`. Since `testbenches/xschem/simulations/` is generated and git-ignored, a fresh clone works right away. Xschem's **IHP > Create FET .save file** menu entry writes the same file manually.

> **Tip**
>
> The dc and transient testbenches follow exactly the same pattern. Only the analysis command changes: `dc Vgsp 0 $&VDD 1m` sweeps the input source in 1 mV steps, and `tran 1u 5m` simulates 5 ms with a sine input `sin({Vcm} 10m 1k)`, a 10 mV amplitude at 1 kHz. Both testbenches add a `.csparam VDD=VDD` line next to the `.param` line, because a parameter is visible inside the `.control` block (as `$&VDD`) only through `.csparam`. In the dc sweep, at which input voltage does `vout` cross $V_{\mathrm{cm}} = 0.75\,\mathrm{V}$? The simulation reports approximately 0.745 V, slightly below $V_{\mathrm{DD}}/2$.

5.3.2 Batch Simulation with the Makefile

Every testbench can also be run non-interactively through the Makefile, which is useful for reproducible results and scripting.

```
cd macros/inverter
make sim-xschem                          # run the default testbench (inverter_top_tb_tran)
make sim-xschem TB=<testbench_name>      # run another testbench non-interactively
make sim-view-xschem SCRIPT=plot_inverter      # plot results of inverter with Python
make sim-view-xschem SCRIPT=plot_inverter_top  # plot results of inverter_top with Python
```

`sim-xschem` runs in batch mode: it netlists the testbench with `xschem netlist` and then calls `ngspice -b` directly, so `make` blocks until the run finishes and sees its exit status. The testbench is selected with `TB=<testbench_name>` and defaults to `<CELL>_tb_tran`. Since the run is headless, the `plot` commands in the testbench's `.control` block are a no-op. Every testbench exports its results with `wrdata` to testbenches/xschem/plot_simulations/data/ instead.

The Makefile target `sim-view-xschem` calls the Python plotters in testbenches/xschem/plot_simulations/, selected with the `SCRIPT` variable (default: `plot_<CELL>`), to parse the raw simulation output and plot the results with Matplotlib. The following figures show some of these plotted results.

> **Tip**
>
> The full simulation suite, including all four testbenches plus the CACE flow (see Section 5.5), is triggered with `make sim-all`.

5.4 Schematic-Level Drawing

To inspect or modify the underlying circuit of a symbol, mark the symbol and press `e` to descend into the underlying circuit, then `Ctrl + e` to go back up. To inspect or modify the graphical symbol of a device, mark the symbol and press `i` to descend into the symbol drawing, then `Ctrl + e` to go back up again.

5.4.1 Drawing the Inverter Schematic

Figure 17 displays the schematic of the `inverter` cell with additional small-signal parameters for the NMOS and PMOS transistors at the chosen operating point. To view these small-signal parameters, hold `Ctrl` and click **Annotate OP** after the simulation has finished. Afterwards, descend into the inverter schematic.

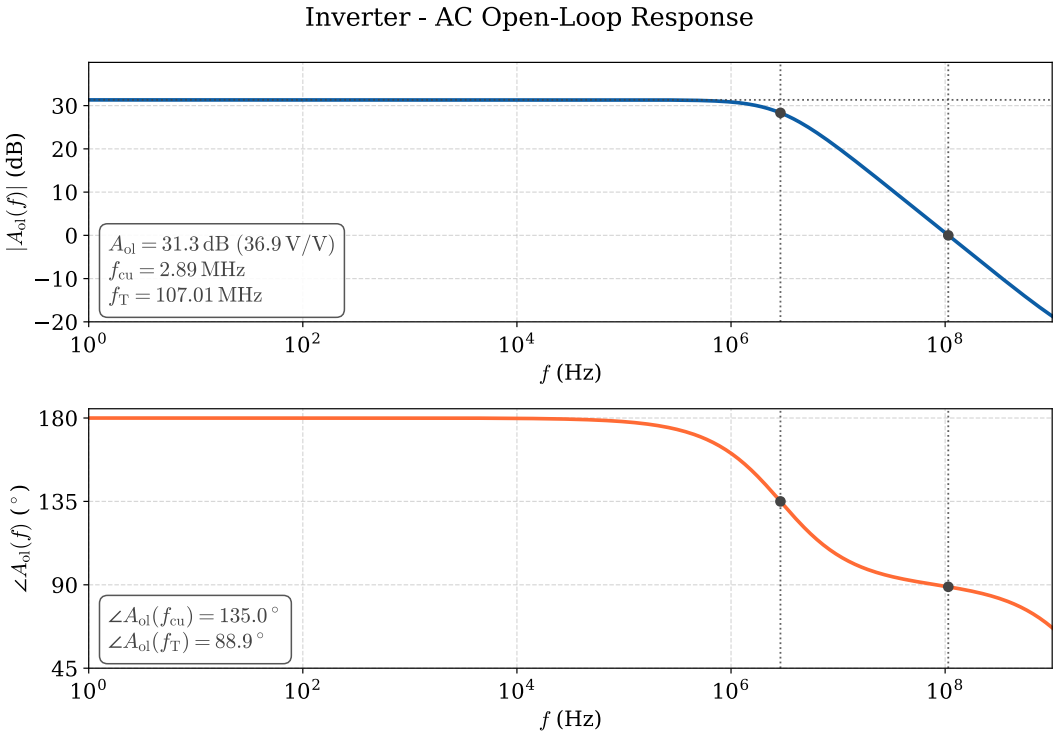


Figure 14: Plotted simulation result of `inverter_tb_ac_ol`.

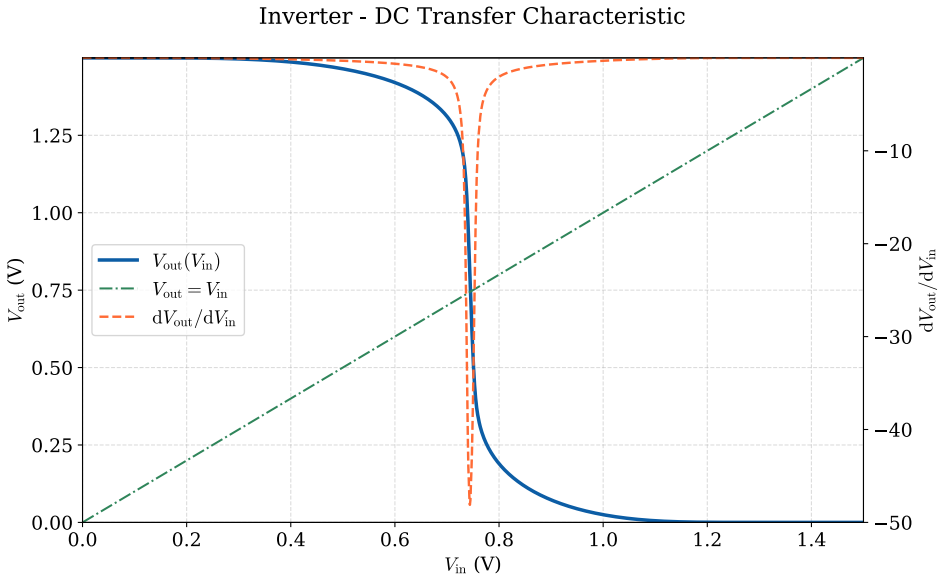


Figure 15: Plotted simulation result of `inverter_tb_dc_vout`.

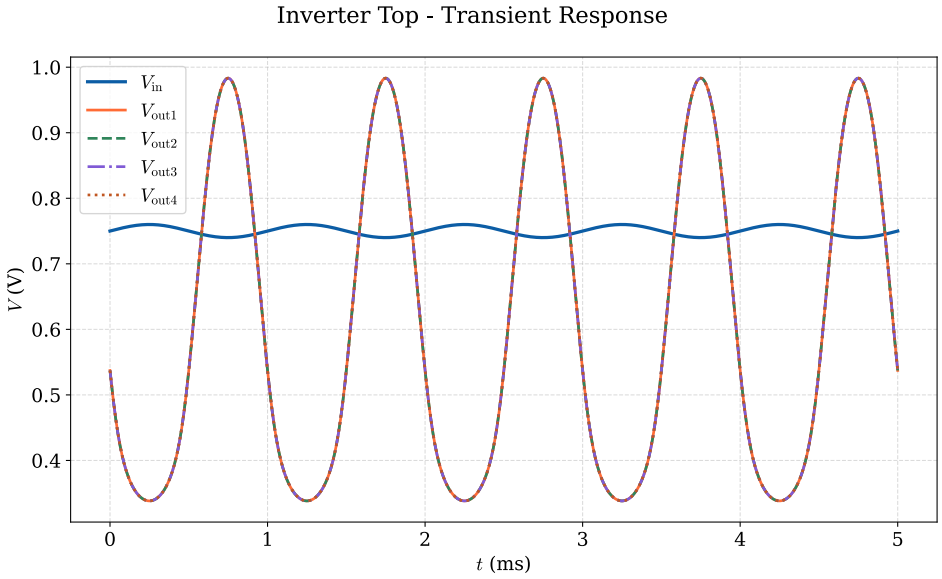


Figure 16: Plotted simulation result of `inverter_top_tb_tran`.

To draw this schematic from scratch, start Xschem in `macros/inverter/schematic/xschem/` and go through the following steps. The reference is `inverter.sch`, and the device geometry comes from the sizing in Section 5.6 (see Table 2).

1. **Place the transistors.** Press `Ins`, and pick `sg13_lv_nmos.sym` and `sg13_lv_pmos.sym` from the `sg13cmos5l_pr` PDK library. Place the NMOS M_1 at the bottom and the PMOS M_2 at the top.
2. **Set the device parameters.** Select a transistor and press `q`. Enter the values from Table 2, for example for M_1: `l=1.0u`, `w=20.0u`, `ng=20`, `m=1`. Note that `w` is the total width and `ng` is the number of fingers. Keep `spiceprefix=X`, because the PSP transistor models of the PDK are SPICE subcircuits and must be instantiated with an `X` prefix. Keep `mm_ok=1` as well, it is the per-instance switch of the PDK's mismatch models, which the CACE mismatch runs in Section 5.5 rely on.
3. **Place the dummy transistors.** Copy each device with `c` and change the properties to the dummy sizing (`w=2.0u`, `ng=2` for the NMOS and `w=12.0u`, `ng=2` for the PMOS). Wire all four terminals of each dummy to its supply rail (gate, source, drain, and bulk of M_{dummy1} to `VSS`, those of M_{dummy2} to `VDD`), so the dummies are off and only their geometry matters.
4. **Wire the circuit.** Press `w` and connect the gates of M_1 and M_2 to the input net, the drains to the output net, the NMOS source and bulk to `VSS`, and the PMOS source and bulk to `VDD`.
5. **Add the pins.** Press `Ins` and place `ipin.sym` for the input `vin`, `iopin.sym` for the output `vout`, and `iopin.sym` for `VDD` and `VSS`. The pin names define the port names of

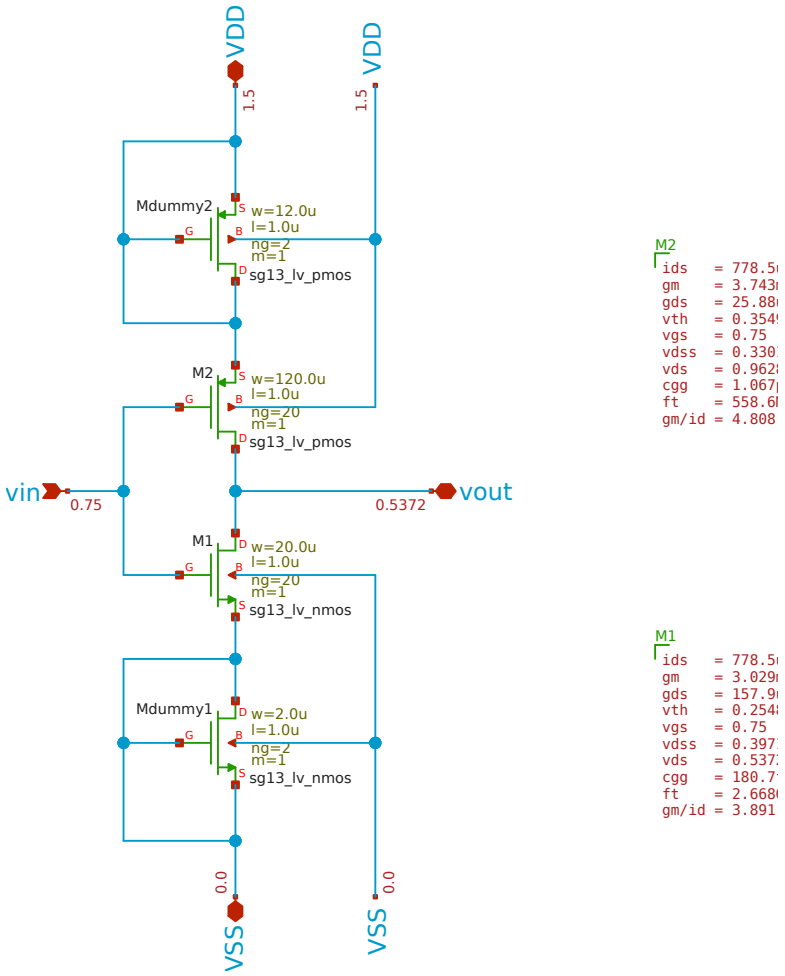


Figure 17: Xschem `inverter` schematic with dummy transistors and OP annotations.

the subcircuit later used for LVS, so name them exactly `vin`, `vout`, `VDD`, and `VSS`.

6. **Annotate (optional).** Place `annotate_fet_params.sym` next to M_1 and M_2 and set `ref=M1` and `ref=M2`. After a simulation, these display the operating-point parameters of the devices directly in the schematic, as visible in Figure 17.
7. **Save** with `Ctrl + s` as `inverter.sch`.

5.4.2 Drawing the Inverter Symbol

To use the inverter in a testbench or in the 4-channel `inverter_top.sch`, it needs a symbol. Figure 18 shows the finished symbol, the reference is `inverter.sym`.

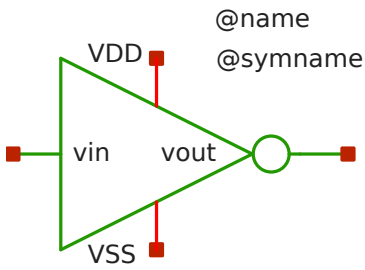


Figure 18: Xschem `inverter` symbol.

1. **Auto-generate the symbol.** With `inverter.sch` open, press `a`. Xschem creates `inverter.sym` with one pin per schematic pin, placed on a rectangular outline.
2. **Open the symbol.** Load it with `Ctrl + o` (or select a placed instance and press `i`).
3. **Redraw the shape.** Delete the auto-generated rectangle and draw the triangle with the output bubble using the line (`l`) and arc tools. Keep the pin boxes, they carry the electrical information (`name` and `dir` attributes).
4. **Arrange the pins.** Place `vin` on the left, `vout` on the right, `VDD` on top, and `VSS` at the bottom. Add the small text labels with `t`.
5. **Check the global properties.** Press `q` with nothing selected. The symbol must have `type=subcircuit` and `format="@name @pinlist @symname"`. With `type=subcircuit`, Xschem descends into `inverter.sch` during netlisting and emits the full transistor-level subcircuit. Keep the `spectre_format` line that Xschem wrote next to it, see the note in Section 5.9.3.
6. **Save** the symbol.

You can now place the symbol in any schematic with `Ins`. The 4-channel `inverter_top.sch` places four instances of it, and its symbol `inverter_top.sym` is drawn the same way. The PEX symbols `inverter_pex.sym` and `inverter_top_pex.sym` are not drawn by hand, `make symbol-pex` derives them (see Section 5.9.3).

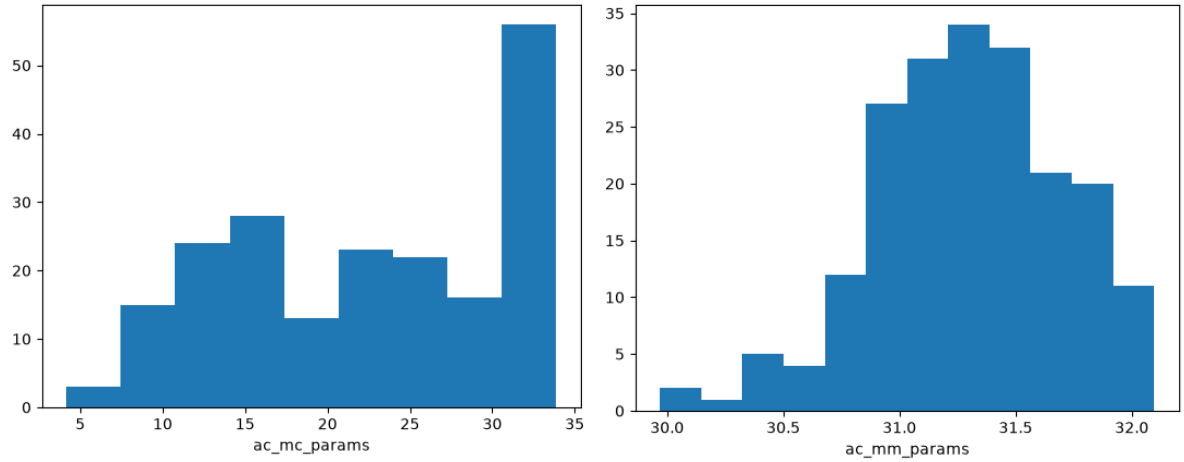
5.5 Process Variation and Mismatch (CACE)

CACE drives Ngspice via a characterisation YAML file (`verification/cace/inverter.yaml`) to run Monte Carlo simulations and produce process and mismatch variation plots.

```
make sim-cace
```

Plots are written to `verification/cace/results/inverter/`. This step is mainly relevant for advanced designers who want to characterise the analog macro across PVT corners. The following figures show the results of the CACE simulations for 200 iterations.

Figure 19 shows the process (left) and mismatch (right) variations of the dc open-loop gain A_{ol}, while Figure 20 plots its variation across several process corners at one fixed supply voltage.



(a) Process variation of A_{ol}.

(b) Mismatch variation of A_{ol}.

Figure 19: CACE process (left) and mismatch (right) variations of A_{ol}.

Figure 21 shows the process (left) and mismatch (right) variations of the open-loop lower cut-off frequency f_{cu}, while Figure 22 plots its variation across several process corners at one fixed supply voltage.

> **Note**
>
> Note that the above process and mismatch variation results are poor for an analog circuit. Also, the open-loop gain varies significantly across corners. This is because the example is only a simple inverter without a compensation technique. For a practical single-ended inverter amplifier, one would at least add a feedback resistor from output to input to stabilise the operating point and therefore the gain across corners. If you want to see improved variation results, you can try adding a feedback resistor in the `inverter.sch`

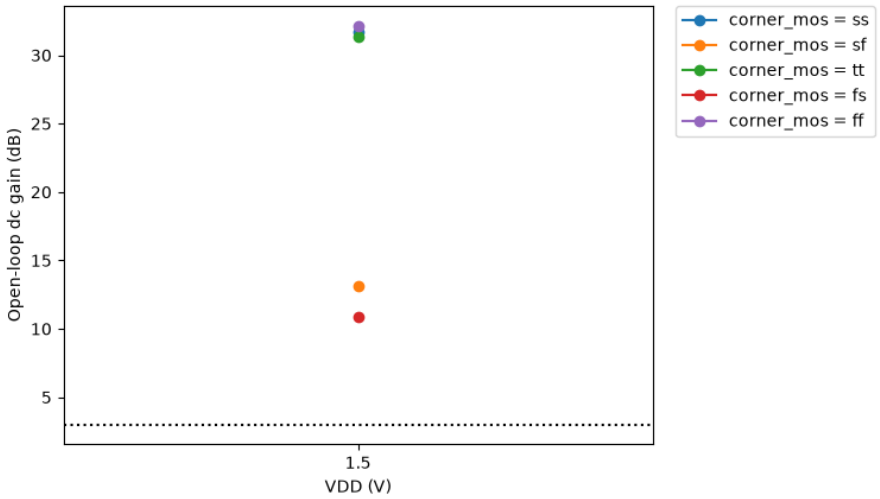
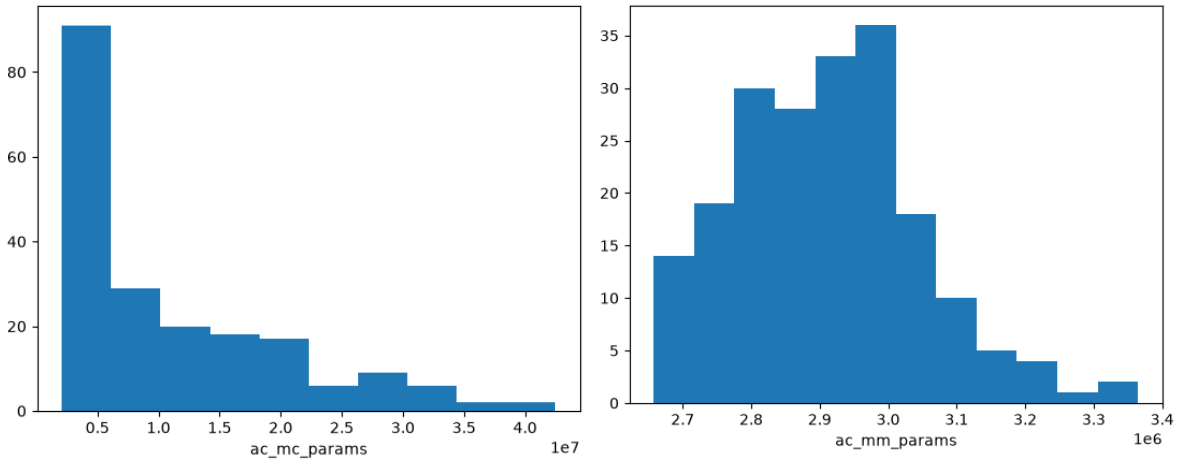


Figure 20: CACE variation of A_{ol} across several process corners.



(a) Process variation of f_{cu}.

(b) Mismatch variation of f_{cu}.

Figure 21: CACE process (left) and mismatch (right) variations of f_{cu}.

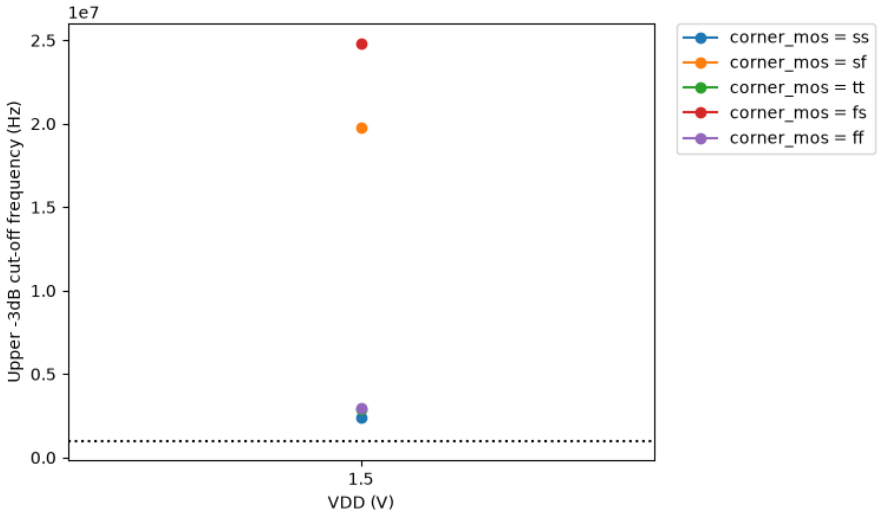


Figure 22: CACE variation of f_{cu} across several process corners.

schematic and running the CACE flow again.

Tip

In the `inverter.yaml` characterisation file, more parameter sweeps can be added, for example, across temperature, supply voltages or load capacitance.

Important

Do not `print()` from a CACE postprocess hook. The hooks in `verification/cace/scripts/` post-process the raw Ngspice results before CACE evaluates them against the specification, and CACE 2.11.0 redirects their stdout into its own rich logger, which writes back to stdout. The first `print()` therefore recurses without end, and `make sim-cace` hangs instead of failing. That is why the debug prints in `inverter_tb_ac.py` are commented out. Write intermediate values to a file if you need to inspect them.

5.6 Transistor Sizing

During simulation, it could happen that some specifications are not fulfilled and that the inverter must be resized. In this tutorial, the inverter is sized with the $g_{\mathrm{m}}/I_{\mathrm{D}}$ method (see Transistor Sizing Using $g_{\mathrm{m}}/I_{\mathrm{D}}$ Methodology for theoretical background). Instead of relying on textbook square-law equations, the method looks up the actual device behaviour from

pre-characterised SPICE simulations, parameterised by g_m/I_D. An overview of these techsweeps can be found in the folder `doc/sizing/`. The chosen g_m/I_D point sets the inversion level of the transistor and therefore the trade-off between intrinsic gain (high g_m/I_D, weak inversion) and bandwidth (low g_m/I_D, strong inversion). These trade-offs are discussed in Section 5.6.1. For each chosen operating point the lookup tables return all relevant figures of merit (I_D/W, g_m/g_{ds}, g_m/C_{gg}, S_{TH}, etc.) at the actual bias condition, which is then used to compute W for a given drain current.

5.6.1 Design Trade-offs: Choosing g_m/I_D and L

Before sizing anything, two numbers have to be picked: the transconductance efficiency g_m/I_D and the channel length L. Everything else follows from them, so it is worth understanding what each one buys and what it costs.

g_m/I_D says how much transconductance you get per ampere of bias current, in S/A or V^{-1}. It is a normalised, technology-independent measure of the **inversion level** of the device.

Figure 23 shows the trade-offs in one picture. The arrows point from the centre towards the corner where each design parameter reaches its **best** value, so the same parameter is at its worst in the opposite direction.

Read it corner by corner:

- **Strong inversion, short** L (bottom left): f_T and input impedance Z_{in} are maximal, gate capacitance C_{gg} and area are minimal. This is the high-speed corner.
- **Strong inversion, medium** L (left): minimal thermal noise PSD S_{TH} and maximal current density I_D/W.
- **Moderate inversion, short** L (bottom): the speed-efficiency product $f_T \cdot g_m/I_D$ is maximal, the best overall compromise between bandwidth and current.
- **Moderate inversion, long** L (top): the output resistance $r_{ds} = 1/g_{ds}$ is maximal. This is where current sources belong.
- **Weak inversion, long** L (top right): intrinsic gain g_m/g_{ds} is maximal, while flicker corner frequency f_{co}, noise factor γ, flicker noise S_{FL}, offset V_{OS}, and power P are minimal. This is the low-power, low-noise, high-gain corner.
- **Weak inversion, medium** L (right): overdrive V_{od} and $V_{ds,sat}$ are minimal, so this corner gives the most voltage headroom.

Two corners stay empty on purpose. Strong inversion with a long channel and weak inversion with a short channel optimise nothing, they combine the drawbacks of both settings, so in practice they make little sense.

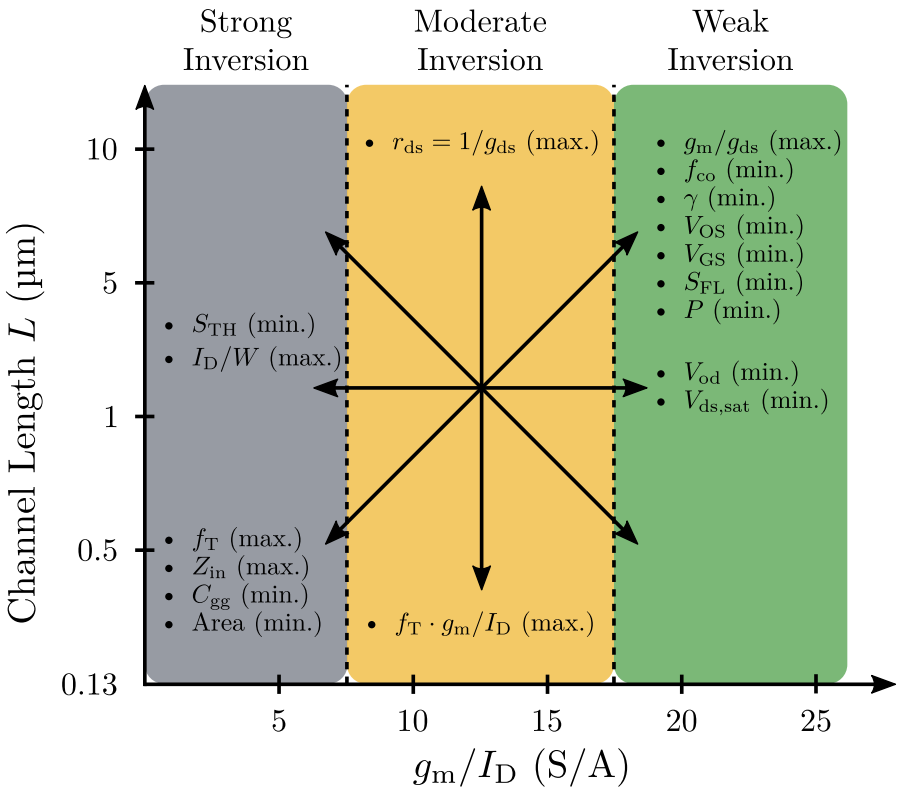


Figure 23: Overview of key design trade-offs depending on the channel length L and the transconductance efficiency g_m/I_D. Taken from Tradeoffs in Saturation in the Analog Circuit Design course at JKU.

Rules of thumb

Fast circuits use **strong inversion with short channels**. Low-power and low-noise circuits use **weak inversion with long channels**. Current sources use **moderate inversion with longer channels**, to maximise their output resistance. Everything else starts in moderate inversion and is moved from there.

Where does our inverter sit?

The sizing below lands at $g_m/I_D = 3.76\,\mathrm{V}^{-1}$ with $L = 1\,\mu\mathrm{m}$, which is deep strong inversion with a comparatively long channel, that is the corner the map leaves empty. This is a deliberate compromise for this example, not an oversight: the stage has to drive a 10 pF load, which demands a lot of g_m and therefore a lot of current, while the long channel keeps the intrinsic gain and the matching of this very simple circuit acceptable. It also explains the modest simulated gain of about 31 dB. Keep the map in mind when you size your own blocks, where you are free to choose the operating point rather than inheriting it.

5.6.2 Sizing the Inverter

The full sizing flow for the inverter is implemented in the Jupyter notebook `scripts/sizing/sizing_inverter.ip` The notebook uses `pygmid` to query the pre-characterised ihp-sg13cmos5l NMOS/PMOS lookup tables (`sg13cmos5l_lv_nmos.mat` and `sg13cmos5l_lv_pmos.mat`). An overview of the g_m/I_D MOSFET characterisation procedure can be found here. The MOS models of ihp-sg13cmos5l are shared with ihp-sg13g2, so the lookup tables and the resulting sizing are the same as in the SG13G2 sibling of this template.

The Jupyter notebook `sizing_inverter.ipynb` runs through the following steps:

1. Define the design targets: $V_{DD} = 1.5\,\mathrm{V}$, $V_{cm,in} = V_{cm,out} = V_{DD}/2 = 0.75\,\mathrm{V}$, $I_{out} = 0.75\,\mathrm{mA}$, $C_{load} = 10\,\mathrm{pF}$ and $L = 1\,\mu\mathrm{m}$. L is chosen longer than the minimum to increase intrinsic gain and reduce mismatch, but at the cost of increased area and reduced bandwidth.
2. Look up g_m/I_D, g_m/g_{ds}, g_m/C_{gg} and I_D/W based on L and the operating-point voltages.
3. Compute the required widths $W = I_D/(I_D/W)$ for NMOS and PMOS, and derive the resulting W_{PMOS}/W_{NMOS} ratio.
4. Round the continuous widths to a practical finger configuration with equal finger counts for NMOS and PMOS (for better layout matching) and recompute the small-signal parameters with the final widths.
5. Estimate the open-loop gain $A_{ol} = -g_{m,tot}/g_{ds,tot}$ and the output resistance $R_{out} = 1/g_{ds,tot}$ of the complementary inverter stage.

The resulting device geometry is summarised in Table 2.

Table 2: Sized device geometry of the complementary inverter and its dummy devices.

Device	L (m)	Fingers (NF)	W / Finger (m)	Total W (m)
NMOS M_1	1.0	20	1.0	20.0
PMOS M_2	1.0	20	6.0	120.0
NMOS M_{dummy1}	1.0	2	1.0	2.0
PMOS M_{dummy2}	1.0	2	6.0	12.0

With these final widths the notebook computes the small-signal parameters listed in Table 3. As a reference, compare these values with the OP annotations in Figure 17.

Table 3: Calculated small-signal parameters with final widths.

Parameter	NMOS M_1	PMOS M_2	Inverter (NMOS ‖ PMOS)
Drain current I_{D} (mA)	0.89	0.78	–
Transconductance efficiency $g_{\text{m}}/I_{\text{D}}$ (V^{-1})	3.76	3.76	–
Inversion level	strong	strong	–
Transconductance g_{m} (mS)	3.35	2.92	–
Output conductance g_{ds} (S)	116	29	–
Gate capacitance C_{gg} (fF)	154.32	824.86	–
Transit frequency $f_{\text{T}} = g_{\text{m}}/(2\pi\, C_{\text{gg}})$ (GHz)	2.91	0.54	–
Thermal noise PSD S_{TH} at 1 Hz (pV^2/Hz)	38.95	73.98	–
Flicker corner frequency f_{co} (MHz)	3.87	2.28	–
$g_{\text{m,tot}} = g_{\text{m,NMOS}} + g_{\text{m,PMOS}}$ (mS)	–	–	6.28
$g_{\text{ds,tot}} = g_{\text{ds,NMOS}} + g_{\text{ds,PMOS}}$ (S)	–	–	145

From these values, the open-loop dc gain A_{ol}, the output resistance R_{out}, the open-loop cut-off frequency f_{cu}, and the unity-gain frequency f_{T} with a dominant capacitive load $C_{\text{load}} = 10\,\text{pF}$ can be estimated as

$$|A_{\mathrm{ol}}| = \frac{g_{\mathrm{m,tot}}}{g_{\mathrm{ds,tot}}} = \frac{6.28\,\mathrm{mS}}{145\,\mathrm{S}} = 43.2 = 32.72\,\mathrm{dB},$$

$$R_{\mathrm{out}} = \frac{1}{g_{\mathrm{ds,tot}}} = \frac{1}{145\,\mathrm{S}} = 6.89\,\mathrm{k},$$

$$f_{\mathrm{cu}} = \frac{1}{2\pi\, R_{\mathrm{out}}\, C_{\mathrm{load}}} = \frac{1}{2\pi \cdot 6.89\,\mathrm{k} \cdot 10\,\mathrm{pF}} = 2.31\,\mathrm{MHz},$$

$$f_{\mathrm{T}} = \frac{g_{\mathrm{m,tot}}}{2\pi\, C_{\mathrm{load}}} = \frac{6.28\,\mathrm{mS}}{2\pi \cdot 10\,\mathrm{pF}} = 100\,\mathrm{MHz}.$$

Table 4 compares these hand-calculated values against the open-loop ac simulation result from `inverter_tb_ac_ol` (see Figure 14).

Table 4: Calculated vs. simulated open-loop characteristics of the inverter.

Quantity	$g_{\mathrm{m}}/I_{\mathrm{D}}$ calculation	ac open-loop simulation
Open-loop dc gain A_{ol}	32.72 dB	31.33 dB
Open-loop cut-off frequency f_{cu}	2.31 MHz	2.89 MHz
Unity-gain frequency f_{T}	100 MHz	107 MHz

> **Tip**
>
> The notebook is fully parametric. Changing L, the target $g_{\mathrm{m}}/I_{\mathrm{D}}$ or I_{out} and re-running the cells immediately produces a new sizing point, so it can be used as a starting template for sizing other simple analog blocks with the ihp-sg13cmos5l lookup tables. Try it out yourself by decreasing the transistor length to $L = 0.5\,\mathrm{m}$ and see how the small-signal parameters and the resulting open-loop gain and cut-off frequency change. What would you expect based on the trade-offs above?

5.7 Getting Started with KLayout

The layout is created and edited in KLayout. Open it in edit mode with `ke`, an alias for `klayout -e` provided by the container. With the ihp-sg13cmos5l PDK active, KLayout loads the `sg13cmos5l` technology, which brings the layer properties, the `SG13_dev` PCell library, and the `SG13CMOS5L PDK` menu with the DRC and LVS runners.

```
ke                                # empty session
ke layout/inverter_top.klay.gds # or open a layout directly at startup
```

Some basics before drawing:

- **Database unit.** When creating a new layout (`File > New Layout`), set the database unit (DBU) to 0.001 m. This is important for the IHP Open-PDK. The KLayout cheatsheet explains how to rescale a layout that was started with the wrong DBU.
- **Layer roles.** Each drawing layer has three roles: `MetalX.drawing` carries the actual geometry, `MetalX.pin` marks the pin shapes, and `MetalX.text` carries the pin labels used by LVS.
- **PCells.** The PDK provides parametric cells (PCells) in the `SG13_dev` library, for example `nmos`, `pmos`, resistors (`rppd`, `rsil`, `rhigh`), capacitors (`cap_cmomi`, `cap_cmomf`, `moscap_n`, `moscap_p`), taps (`ntap1`, `ptap1`), and `via_stack`. Select a placed PCell and press `Q` to edit its parameters. The library keeps the name it has on SG13G2, but its device list is the SG13CMOS5L one, so it has no MIM capacitor and no HBTs, see the KLayout cheatsheet.

The most important shortcuts are listed in Table 5, taken from the KLayout cheatsheet in this repository.

Table 5: Essential KLayout shortcuts with the productivity plugins installed.

Shortcut	Action
`Q`	Properties of the selected object / PCell
`M`	Move quickly
`A`	Align tool
`C`	Copy object
`P`	Path / drawing tool (double-click to finish)
`R`	Draw rectangle
`T`	Place text
`Shift + P`	Set pin (input, output, …) for LVS
`O`	Place vias during routing (path mode)
`E` / `Ctrl + E`	Descend into / ascend from an instance
`Shift + R`	Rotate instance by 90°
`Shift + H` / `Shift + V`	Flip instance horizontally / vertically
`F`	Zoom full
`U` / `Shift + U`	Undo / redo
`K` / `Shift + K`	Ruler+ / clear all rulers
`0` … `9`	Layer focus (see Table 6)

Most of these shortcuts come from the KLayout Productivity Suite, a collection of plugins developed at JKU that make manual layout much faster. The suite is installed in the IIC-OSIC-TOOLS container, so nothing has to be set up. The full documentation with animations is here. A short summary:

Table 6: The KLayout productivity plugins in a nutshell.

Plugin	What it does
Move Quickly Tool (`M`)	Rapid repositioning of shapes and instances with mouse or keyboard, including diagonal moves and multi-object selection
Align Tool (`A`)	Align objects to each other by clicking reference features (middle, corner, or edge markers)
Pin Tool (`Shift + P`)	Place LVS-ready pins (pin shape plus text label) with automatic layer detection from the current layer selection
Layer Shortcuts (`0` … `9`)	Fast layer visibility switching: `1` … `7` focus on one metal with its vias, `8` gate poly, `9` diffusion, `0` shows the default layers, `,` hides them, `Shift + 1` … `Shift + 9` extend the current focus by one more layer group
Netlist Import (`File > Import > Netlist`)	Import a SPICE netlist and place the cell instances with technology-specific mapping. Great for starting a layout from the schematic netlist
Library Manager (`File > Cell Library Manager`)	Manage hierarchical layouts and cell libraries, and export a clean layout for tapeout
Auto Backup (`File > Automatic Backups`)	Periodic layout backups with configurable interval and rotation
Vector File Export (`File > Export Vector File`)	Export the layout view as PDF or SVG

5.8 Inspect / Edit the Layout

After the schematic is finalised and the design targets are met, the layout can be created. The layout of the `inverter` cell is hierarchical, and the top-level cell `inverter_top` contains four instances of the `inverter` cell. Around the 4-channel inverter layout, a power ring is drawn, which is connected to the `VDD` and `VSS` pins of the four inverters. Its horizontal segments are on `Metal3` and its vertical segments on `Metal4`, 2.2 µm wide, and the two layers meet in a 4 by 5 `Via3` array at each of the eight ring corners. This power ring is required for top-level integration, connecting the macro power to the chip's PDN. It stays clear of `TopMetal1`, so the `TopMetal1` stripes of the chip's PDN can cross the macro and feed the ring (see Section 7).

With the following command, the GDS of the 4-channel inverter is opened in KLayout as shown in Figure 24. Use the layer shortcuts `0`, `1`, `2`, …, `9` from Table 6 to view the individual layers.

```
ke layout/inverter_top.klay.gds
```

`inverter_top.klay.gds` is the source of truth. After modifying the layout, export a clean `.gds` file (for example `inverter_top.gds`) by pressing `File > Export Layout For Tapeout`, which resolves every PCell and library reference into a static cell so the result opens without the PDK loaded. Never hand-edit the exported file: keeping the two in step by editing both is how they drift apart, and the drift is invisible because the sign-off targets only look at the exported one. All verification flows (DRC, LVS, and PEX with both KLayout and Magic) accept either `.gds` or `.klay.gds`, while the build targets (LEF export, GDS copy, and rendering) use the exported `.gds`.

> **Warning**
>
> The export re-evaluates the PCells against the **installed** PDK, so device geometry can change without anyone touching the layout. That happened here. For the identical stored parameters the installed `SG13_dev pmos` draws its `NWell` (`31/0`) 0.11 µm tighter on every side than the geometry frozen into the exported GDS, which pulled the two mirrored pmos wells away from the hand-drawn `NWell` straps that bridge them and left a 0.11 µm gap, reported as four `NW.b1` markers (min. PWell width between NWell regions). The straps were re-derived from the wells they have to join, so they abut either revision. Today's SG13G2 PDK still draws the wider well, so a layout carried between the two templates has to be re-exported under the target PDK and re-checked, not copied. Re-run DRC, LVS and PEX after every export.

> **Important**
>
> Never delete the box on the `prBoundary` layer (`189/0`) that frames the macro, and make sure a re-drawn layout still carries it. The chip flow derives the bounding box of every macro from this layer: Magic maps all datatypes of layer 189 to its boundary layer, and LibreLane's `Magic.StreamOut` step reads that boundary back for every macro it places. A layout without the box fails the whole chip build with `Failed to extract PR boundary from GDSII view of macro 'inverter_top'`, two flows away from the layout that caused it. Treat the box as part of the layout: keep it in the KLayout editing source `inverter_top.klay.gds` so every re-export carries it, and hide it at render time with `sak-render.py -x` if it spoils a chip shot. `make check-boundary` catches a missing box in the macro build, see Section 5.10.

KLayout also supports a 2.5D viewer, which can be used to inspect the layout. To open the 2.5D viewer, press `SG13CMOS5L PDK > BEOL 2.5D Viewer`. Figure 25 shows the `inverter_top` layout in the 2.5D viewer.

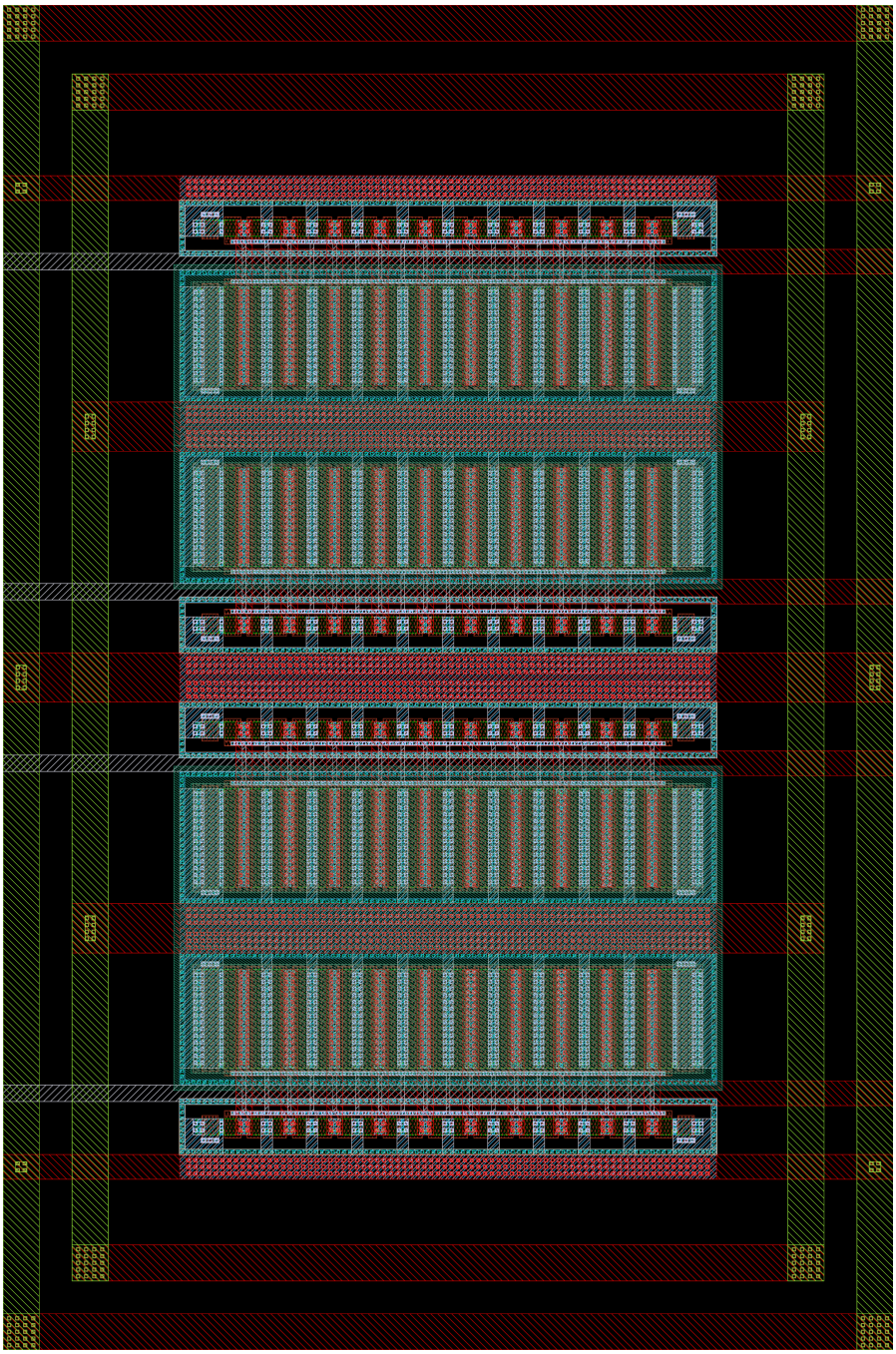


Figure 24: `inverter_top` layout in KLayout.

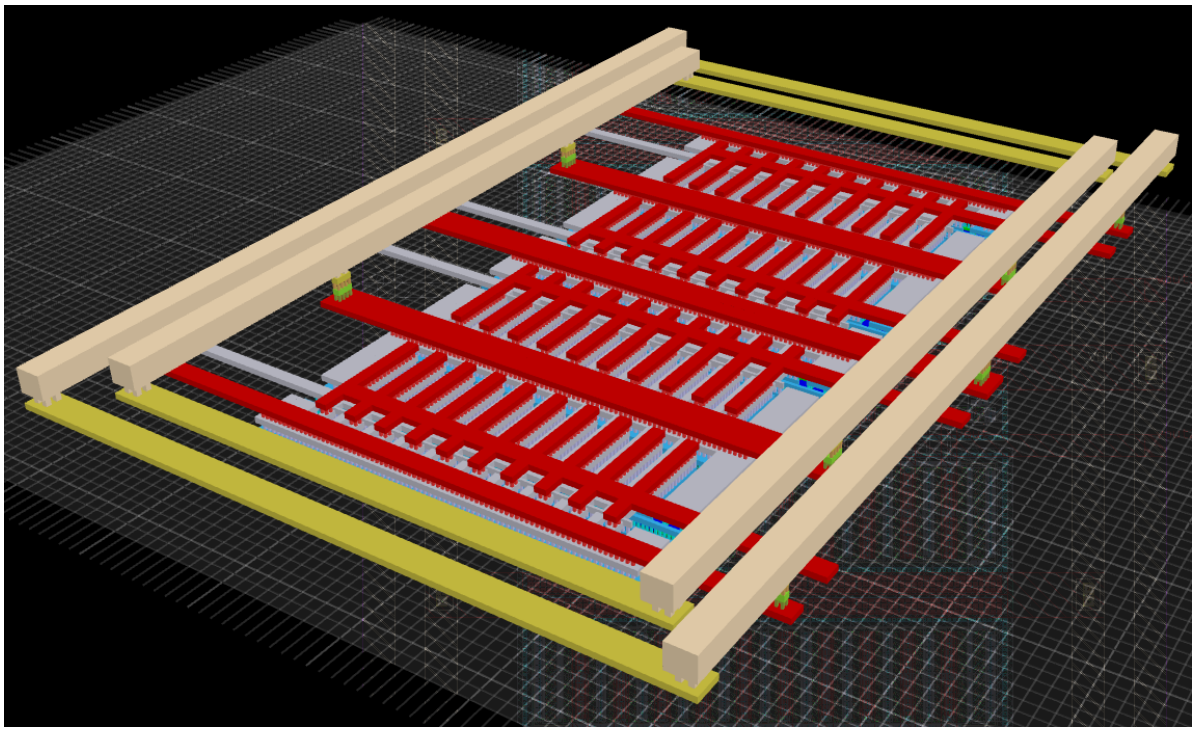


Figure 25: `inverter_top` layout in KLayout 2.5D viewer.

5.8.1 Drawing the `inverter` Unit Cell

The hierarchy starts with the unit cell. `layout/inverter.gds` is its editing source, and `inverter_top.klay.gds` pulls it in as a library through `inverter_top.klay.klib`, so the four instances follow every edit of the unit cell. To draw the unit cell from scratch, open KLayout with `ke` and go through the following steps. The finished result is the `inverter` cell inside Figure 24.

1. **Create the cell.** `File > New Layout`, top cell name `inverter`, DBU 0.001 m.
2. **Place the transistor PCells.** Drag `nmos` and `pmos` from the `SG13_dev` library into the layout. Place the PMOS row above the NMOS row.
3. **Match the schematic.** Select each PCell and press `Q`. Set the same `w`, `l`, and `ng` values as in the schematic (Table 2): the NMOS with $20 \times 1.0\,\mathrm{m}$ fingers and the PMOS with $20 \times 6.0\,\mathrm{m}$ fingers, both with $L = 1.0\,\mathrm{m}$. As in the schematic, `w` is the total width that the PCell splits into `ng` fingers. Add the two dummy devices next to the active devices in the same way.
4. **Substrate and well contacts.** Tie the substrate to `VSS` and the n-well of the PMOS to `VDD` with `ptap1` / `ntap1` contacts.
5. **Route.** Connect the gates (input), the drains (output), and the source rails with `Metal1` and `Metal2`. Use the path tool `P`, place vias with `O` or with `via_stack` PCells. Keep signal routing on the lower metals within a macro. Use the layer shortcuts `1`, `2`, `8`, `9` to focus on the layer you are working on.
6. **Add the pins.** Select the target metal layer and press `Shift + P` to place the pins `vin`, `vout`, `VDD`, and `VSS`. The pin tool places the pin shape on `MetalX.pin` and the label on `MetalX.text`, which is exactly what LVS expects.
7. **Save** as `layout/inverter.gds`. Then open `inverter_top.klay.gds`, check the four instances and the power ring, and export `inverter_top.gds` as described above.

> **Tip**
>
> The layout drawing order follows the matching rules of analog design: active devices in the centre with common orientation, dummies at the edges, and taps close to the devices. Also check the layout tips in the KLayout cheatsheet.

5.9 DRC, LVS, and PEX

The inverter macro can be verified with two verification flows, either with KLayout or with Magic + Netgen. The Magic + Netgen flow covers DRC, LVS, and PEX, while the KLayout flow covers DRC and LVS, since KPEX does not support the `ihp-sg13cmos5l` PDK yet. Magic + Netgen provides only non-interactive command-line flows, while KLayout also has interactive GUI-based DRC and LVS.

5.9.1 Design Rule Check

The non-interactive DRC can be run with the following Makefile targets (both use `sak-drc.sh`). The `CELL` variable can be set to run DRC on a specific cell, or it defaults to `inverter_top` if not set. The `DRC_LEVEL` variable selects the KLayout DRC level: `precheck`, `macro` (default), or `regular`. It is ignored by `magic-drc`, since Magic has no selectable rule decks and always runs the full rule set compiled into the PDK's Magic tech file.

Check	`precheck`	`macro` *(default)*	`regular`
Foundry precheck rule set			
Pin / label			
Full FEOL + BEOL rules	–		
Off-grid / angle	–		
Recommended rules	–		
Extra rules	–	–	
Density (full-chip fill)	–	–	
Antenna	–	–	

```
make klayout-drc [CELL=<cellname>] [DRC_LEVEL=<precheck|macro|regular>]   # default CELL=inve
make magic-drc [CELL=<cellname>]    # default CELL=inverter_top
```

Reports are written into per-cell run folders (`<CELL>.magic.drc/` for Magic, `<CELL>.klayout.drc/` for KLayout) inside `verification/drc/`. The run folders are wiped at the start of each run, so they always reflect the latest run only.

A successful KLayout run ends with the line

```
INFO    | KLayout DRC Check Passed: No DRC violations detected in the layout.
```

and the Magic report `<CELL>.magic.drc.rpt` lists `COUNT: 0` for the cell.

> **Tip**
>
> In KLayout, a non-interactive DRC run can be launched from the GUI via the menu **`SG13CMOS5L PDK > Run Klayout DRC`**, and the DRC options can be modified via **`SG13CMOS5L PDK > SG13CMOS5L DRC Options`**. The **`F9`** / **`F10`** shortcuts of the IIC-OSIC-TOOLS container are bound to the `ihp-sg13g2` menu entries only.

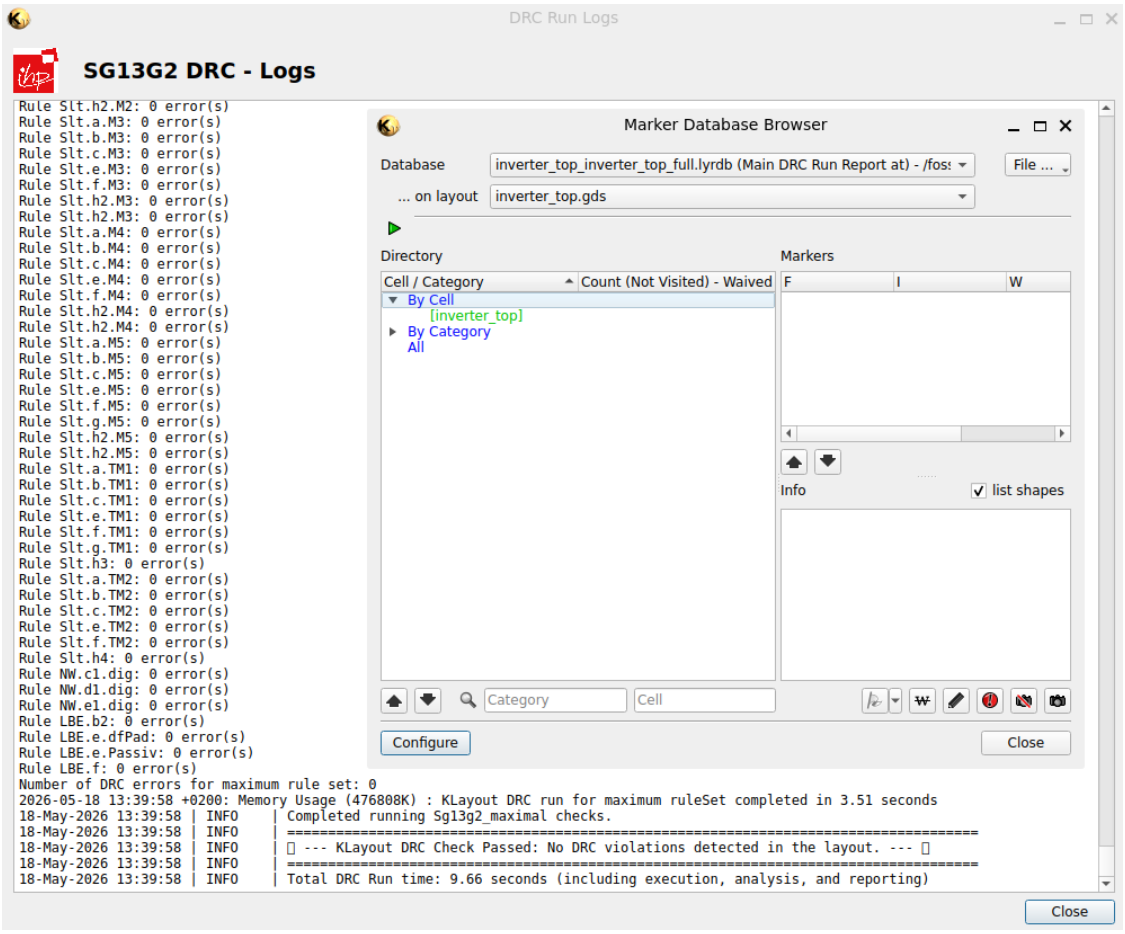


Figure 26: KLayout DRC output windows.

5.9.2 Layout-Versus-Schematic

The non-interactive LVS can be run with the following Makefile targets. The `CELL` variable can be set to run LVS on a specific cell, or it defaults to `inverter_top` if not set.

```
make klayout-lvs [CELL=<cellname>]  # default CELL=inverter_top, uses CDL netlist exported fi
make magic-lvs [CELL=<cellname>]    # default CELL=inverter_top, uses SPICE netlist exported
```

The schematic netlist is exported automatically by the matching `*-lvs-netlist` sub-target and moved to `netlist/schematic/`. The extracted layout netlist is moved to `netlist/layout/` and the reports are written into per-cell run folders (`<CELL>.magic.lvs/` for Magic + Netgen,

`<CELL>.klayout.lvs/` for KLayout) inside `verification/lvs/`. The run folders are wiped at the start of each run, so they always reflect the latest run only.

Substrate and well contacts are not devices for LVS in either flow (KLayout skips the `ntap1` / `ptap1` device extraction, Magic extracts them as substrate connections), so the taps in the layout need no matching device in the schematic. A successful KLayout run prints:

```
| Status           | PASS                                      |
| Mode             | COMPARE                                   |
| Outcome          | Comparison mode: PASS (netlists match). |
| Warnings         | 0                                         |
| Errors           | 0                                         |
```

The Magic + Netgen flow reports:

```
Cell pin lists are equivalent.
Device classes inverter_top and inverter_top are equivalent.

Final result: Circuits match uniquely.
```

> **Note**
>
> The Netgen report additionally states `Netlists match uniquely with property errors` and lists one warning per transistor about a `mm_ok` property that exists only on the schematic side. It is the mismatch switch of the Xschem symbol (see Section 5.4.1), which the layout PCell does not carry, so the two netlists cannot agree on it. This is expected with the current PDK and can be ignored as long as the final result reads `Circuits match uniquely.`

> **Tip**
>
> In KLayout, a non-interactive LVS run can be launched from the GUI via the menu **`SG13CMOS5L PDK > Run Klayout LVS`**. The LVS options can be modified via **`SG13CMOS5L PDK > SG13CMOS5L LVS Options`**, where the schematic netlist `inverter_top_klayout.cdl` has to be selected manually. Set `deep` as the run mode. The **`F11`** / **`F12`** shortcuts of the IIC-OSIC-TOOLS container are bound to the `ihp-sg13g2` menu entries only.

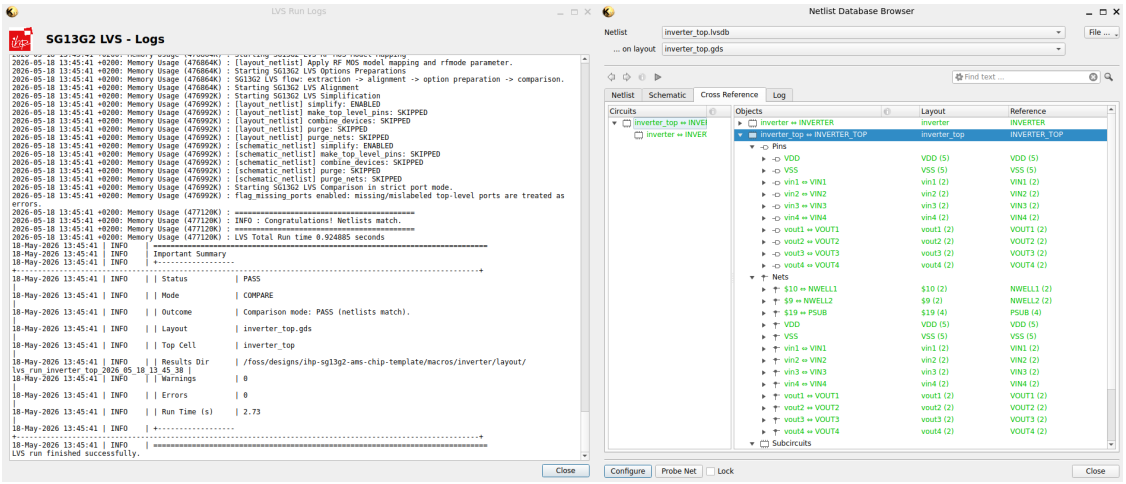


Figure 27: KLayout LVS output windows.

5.9.3 Parasitic Extraction

The non-interactive PEX can be run with the following Makefile target. The `CELL` variable can be set to run PEX on a specific cell, or it defaults to `inverter_top` if not set. The `EXT_MODE` variable can be set to choose between different extraction modes,

- `EXT_MODE=1` for C-decoupled mode,
- `EXT_MODE=2` for C-coupled mode,
- `EXT_MODE=3` for full-RC extraction.

If not set, it defaults to `EXT_MODE=3`.

For full-RC extraction (`EXT_MODE=3`), `magic-pex` also exposes the `sak-pex.sh extresist` tuning parameters (ignored in `EXT_MODE=1/2`):

- `THRESHOLD` - extresist threshold in mOhm (`-t`, default `10000` = 10 Ohm),
- `MINRES` - extresist minimum resistance in mOhm (`-r`, default `1000` = 1 Ohm),
- `MINDELAY` - extresist minimum delay in ps (`-y`, default `1`; `0` = gate by resistance).

```
make magic-pex [CELL=<cellname>] [EXT_MODE=<1|2|3>] [THRESHOLD=<mOhm>] [MINRES=<mOhm>] [MINDE
```

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

That PEX symbol is generated, not maintained by hand. Both PEX targets start by running `make symbol-pex`, which copies `schematic/xschem/<CELL>.sym` to `schematic/xschem/<CELL>_pex.sym`

and changes `type=subcircuit` into `type=primitive`. Everything else is inherited, above all the pin order that `sak-pin-reorder.py` sorts the extracted netlist to. The `primitive` type keeps Xschem from descending into a schematic of the same name, so the instance is netlisted as it stands and its subcircuit comes from the included PEX netlist instead. Because the symbol is rebuilt before every extraction, it cannot go stale when a pin is added, removed or renamed, and it can be built on its own with `make symbol-pex [CELL=<cellname>]`. Edit `<CELL>.sym`, never the generated `<CELL>_pex.sym`.

After the reorder, the PEX target runs `scripts/check_pex_ports.py` on the extracted netlist and fails if a port of the subcircuit is not connected to any element inside it. A floating port usually means a missing or misspelled pin label in the layout, and it is better caught here than as a silently open input in the post-layout simulation.

> **Note**
>
> Every symbol in this template carries `spectre_format="@name ( @pinlist ) @symname"` next to its `format` line. Xschem writes that line itself whenever a symbol is generated from a schematic's pin list (key `a`), and only the Spectre netlister reads it, which is also the one that drives VACASK. The SPICE netlister used for Ngspice ignores it. Do not delete it: without it, instances of that symbol are silently dropped from a Spectre/VACASK netlist and its `subckt` line comes out with an empty port list, with no warning at all.

> **Warning**
>
> `make klayout-pex [CELL=<cellname>] [EXT_MODE=<1|2|3>]` runs KLayout-PEX (KPEX), which does not support the `ihp-sg13cmos5l` PDK yet, so the target currently fails. Use `magic-pex` until KPEX gains SG13CMOS5L support. For the same reason, `klayout-pex` is commented out in the `klayout-verify` target.

5.9.4 Post-Layout Simulation

Now we close the loop and simulate the extracted layout in the same testbenches. Every testbench already carries a spare instance of the PEX symbol (`x3`, see Section 5.3.1), and the PEX symbol has the same pin positions as the cell symbol, so the swap takes a few key presses:

1. Open a testbench, for example `inverter_tb_ac_ol.sch`.
2. Select the active instance `x1` (`inverter.sym`) and press `Shift + T` to set its `spice_ignore` flag. It is now excluded from the netlist.
3. Select the spare instance `x3` (`inverter_pex.sym`), move it with `m` onto the position of `x1`, and press `Shift + T` to activate it. Its pins land on the same wires, so nothing else

changes.

4. The `.include ../../../netlist/pex/inverter_magic_pex_3.spice` line at the top of the control block now supplies the subcircuit definition. If you extracted with a different `EXT_MODE`, adapt the file name.
5. Re-run the simulation with `Ctrl` + click on **Simulate**.

The same swap works in `inverter_top_tb_tran.sch` with `inverter_top_pex.sym` and `inverter_top_magic_pex_3.spice`, which is the post-layout view that `make all` refreshes before it simulates (see Section 5.11).

Compare the post-layout results against the schematic-level results. What differences do you see, and why?

Think about it before simulating. The extracted netlist adds the wiring resistance and the capacitance of every drawn wire to the circuit. The dc transfer characteristic should barely change, because almost no current flows into the parasitics at dc. In the ac response, the added output and internal capacitances and the interconnect resistance shift the poles, so expect a slightly reduced bandwidth and small changes in the measured gain. If your post-layout results deviate strongly, that is usually a sign of a layout problem, for example a too narrow or too long wire on a critical net.

5.9.5 Run Full Verification Flow at Once

If you want to run the full verification flow at once, the verify targets bundle DRC, LVS, and PEX (`magic-verify`) or DRC and LVS (`klayout-verify`) in one command.

```
make klayout-verify [CELL=<cellname>]   # default CELL=inverter_top
make magic-verify [CELL=<cellname>]     # default CELL=inverter_top
```

5.10 Build Deliverables for Top-Level Integration

For the inverter to be picked up by the top-level LibreLane flow, it must expose a **LEF** (abstract pin / blockage view), a **Liberty timing library** stub, and a **Verilog stub**. The Verilog stub is referenced by `rtl/chip_top.sv` for top-level integration. These files can be built with individual Makefile targets or via `build-top`.

```
make check-boundary   # check the PR boundary box on layer 189 in layout/inverter_top.gds
make lef              # final/lef/inverter_top.lef
make lib              # final/lib/inverter_top.lib
make verilog          # final/vh/inverter_top.vh
make build-top        # the four steps above plus copy-gds and render-gds
```

`make build-top` runs `check-boundary` first, so a top cell without the PR boundary box fails the macro build in a second with a message that names the layer, instead of failing the chip flow much later (see the callout in Section 5.8). The check runs `scripts/check_boundary.py`, which also reports the size of the box and warns when the drawn geometry extends beyond it, because the top-level flow places the macro by the boundary and anything outside it may overlap a neighbour.

`make build-top` also copies the GDS to `final/gds/` and renders the macro to `render/img/inverter_top_white.png` and `render/img/inverter_top_black.png`, as displayed in Figure 24.

5.11 Full Analog Macro Flow

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

```
make all
```

This calls `klayout-verify-all`, `magic-verify-all`, `build-top`, and `sim-all` (Xschem + CACE) in sequence. The simulations run last, so the `inverter_top` testbench includes the PEX netlist produced by this run. Depending on the hardware used, a full pass can take a few minutes, especially the CACE simulation step.

6 Bondpad and Logo Implementation

The `ip/` folder holds the three custom IP cells that the top-level chip relies on: a bondpad and two logos. These IPs are generated and require zero hand drawing. Each IP ships with a Python script and Makefile targets that produce the final GDS and LEF, and run KLayout and Magic DRC. This makes it easy to re-spin a different bondpad size or shape, swap the logo image, or move the artwork to another metal layer without touching layout tools.

The three IP cells covered in this section are:

- `sg13cmos5l_ip__bondpad_70x70`, a 70 µm × 70 µm square bondpad built from IHP's `SG13_dev` KLayout PCell library,
- `sg13cmos5l_ip__jku`, the Johannes Kepler University Linz logo, generated from a PNG image,
- `sg13cmos5l_ip__jku_names`, a companion text block with the names of the institute, generated the same way as the JKU logo.

All three IPs follow the same folder structure (`script/`, `final/`, `verification/`) and the same Makefile conventions as the digital and analog macros (`make help`, `make all`, `make clean`, `make klayout-drc`, `make magic-drc`). The logo folders `sg13cmos5l_ip__jku` and `sg13cmos5l_ip__jku_names` have an additional `logo/` folder that contains the source PNG image.

6.1 Directory Structure

The IP design files are located in `ip/`, and the relevant folders are:

Folder	Content
`sg13cmos5l_ip__bondpad_70x70/script/`	KLayout Python script (`bondpad.py`) that instantiates the `SG13_dev::bondpad` PCell and writes the GDS and LEF.
`sg13cmos5l_ip__jku/logo/` and `sg13cmos5l_ip__jku_names/logo/`	Source PNG images (`jku_logo.png`, `jku_names.png`) for the two logos.
`sg13cmos5l_ip__jku/script/` and `sg13cmos5l_ip__jku_names/script/`	PNG-to-GDS converter (`img2lay.py`) shared by both logos.
`final/gds/`, `final/lef/`, `final/lib/`, `final/vh/`	Build deliverables for top-level integration: GDS, LEF, Liberty, and Verilog blackbox stubs.
`verification/drc/`	DRC reports produced by `make klayout-drc` and `make magic-drc`.

> **Tip**
>
> Read the bondpad README, the JKU logo README, and the JKU names README first to get familiar with the Makefile targets and parameters. Most of the commands below are also documented there.

6.2 Bondpad Generation

The bondpad is generated by the Python script `script/bondpad.py`, which runs in KLayout batch mode (`klayout -zz`). It

1. instantiates the `bondpad` PCell from IHP's `SG13_dev` library,
2. flattens the PCell into a static top cell,
3. writes the layout as a GDSII file, and
4. generates a matching LEF macro description for OpenROAD / LibreLane.

Geometry and metal stack are parameterised through three Makefile variables. The defaults match the chip's padframe and the SG13CMOS5L PDK defaults.

Variable	Default	Description
`DIAMETER`	`70.0`	Bondpad diameter / side length in µm.
`SHAPE`	`square`	Pad geometry. Allowed: `square`, `octagon`, `circle`.
`BOTTOM_METAL`	`3`	Lowest metal in the pad stack: `1`=`Metal1` .. `4`=`Metal4`. Top is fixed to `TopMetal1`, the only top metal the SG13CMOS5L PCell offers. Default `3` matches `bondpad_bottomMetal` in `sg13cmos5l_tech.json`.

The available Makefile targets for the bondpad are summarised below.

Target	What it does
`bondpad`	Generate GDS and LEF via `bondpad.py` (`make bondpad [DIAMETER=<um>] [SHAPE=<square|octagon|circle>] [BOTTOM_METAL=<1-4>]`).
`verilog`	Generate a Verilog blackbox stub.
`klayout-drc`	Run KLayout DRC (`make klayout-drc [CELL=<cellname>] [DRC_LEVEL=<precheck|macro|regular>]`).
`magic-drc`	Run Magic DRC (`make magic-drc [CELL=<cellname>]`).
`open`	Browse this folder with `sak-open.py` and open each file in its tool (needs the VNC/X11 desktop).
`clean`	Remove the `final/` and `verification/drc/` output directories.

To execute the `clean`, `bondpad`, `verilog`, `klayout-drc`, and `magic-drc` targets in sequence, run:

```
cd ip/sg13cmos5l_ip__bondpad_70x70
make all
```

View the bondpad layout in KLayout with:

```
ke final/gds/sg13cmos5l_ip__bondpad_70x70.gds
```

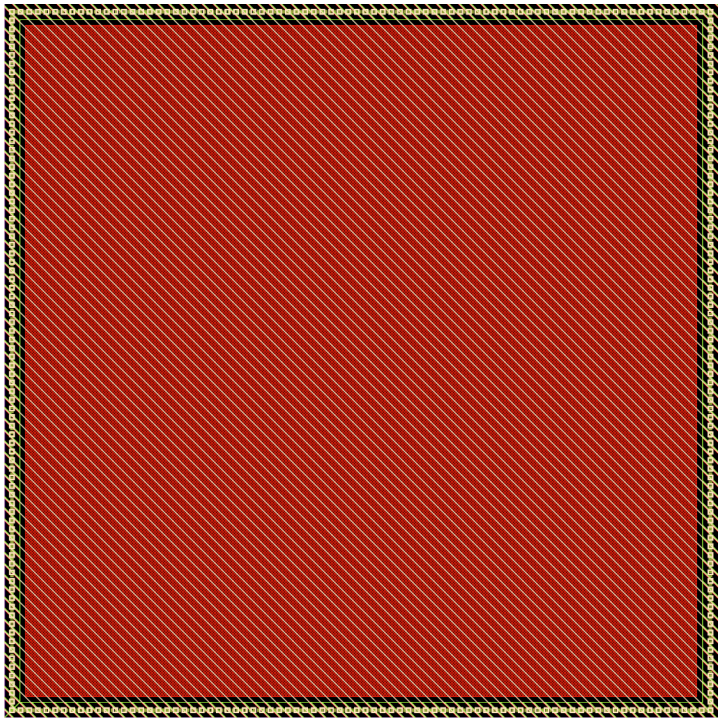


Figure 28: `sg13cmos5l_ip__bondpad_70x70` layout in KLayout.

The bondpad goes from `Metal3` up to `TopMetal1` by default, but the `BOTTOM_METAL` Makefile variable can be set to change the lowest metal in the stack. The pad is surrounded by the pad recognition layer `dfpad` (`41/0`).

The `bondpad` target accepts the geometry and metal-stack variables as command-line overrides. For example, to build an octagonal 80 µm bondpad from `Metal2` up to `TopMetal1`, run:

```
make bondpad DIAMETER=80 SHAPE=octagon BOTTOM_METAL=2
```

6.3 Logo Generation

Both logo IPs share the same Makefile targets and PNG-to-GDS converter `script/img2lay.py`, built on the `klayout.db` Python module. The conversion proceeds in five steps:

1. Open the source PNG (`logo/jku_logo.png` or `logo/jku_names.png`) and convert it to a binary mask using a brightness threshold (default `128`).

2. Optionally invert the mask (`--invert`) so that the dark logo regions become the foreground.
3. Draw every horizontal run of foreground pixels as one rectangle of `--pixel-size` µm height on the selected metal layer.
4. Merge adjacent rectangles into larger polygons (`--merge`) to keep the GDS small and DRC-friendly.
5. Add the block-bounding rectangle on the boundary layers `prBoundary` (`189/0`) and `NoMetFiller` (`160/0`).

The `logo` target hands the `LAYER` name to `img2lay.py`, which resolves it through the PDK's KLayout layer properties file (`Metal4` is `50/0`), and the `lef` target writes the same name into the LEF obstruction, so the artwork in the GDS and the blockage exposed to the router always stay consistent.

Both logos share the same set of Makefile variables as shown below. The `jku_names` logo uses a finer standard pixel size to ensure the readability of the smaller text elements.

Variable	Default (`jku` / `jku_names`)	Description
`BLOCK_SIZE`	`100` / `100`	Physical side length of the rendered logo in µm. The image is scaled to it.
`PIXEL_SIZE`	`0.50` / `0.25`	Edge of one rendered pixel in µm. Must stay `0.21` µm, the `Metal4` minimum space, because single pixels and single-pixel gaps are drawn as is.
`LAYER`	`Metal4` / `Metal4`	Foreground metal layer. One of `Metal1`..`Metal4`, `TopMetal1`.

The image is scaled to `BLOCK_SIZE` by `img2lay.py` (`--block-size`), so the GDS boundary and the LEF `SIZE` always agree, and `PIXEL_SIZE` only sets the resolution of the artwork.

The available Makefile targets for the logos are summarised below.

Target	What it does
`logo`	Convert the PNG into a GDSII layout using `img2lay.py`.
`lef`	Emit a minimal LEF macro (`CLASS BLOCK`, full-area `OBS` on `LAYER_NAME`).
`lib`	Emit a Liberty timing stub (empty cell) so the LibreLane flow can pick the cell up.
`verilog`	Emit a Verilog blackbox stub (`module sg13cmos5l_ip__jku;`) with no ports.
`klayout-drc`	Run KLayout DRC (same `DRC_LEVEL` selection as the bondpad).
`magic-drc`	Run Magic DRC.

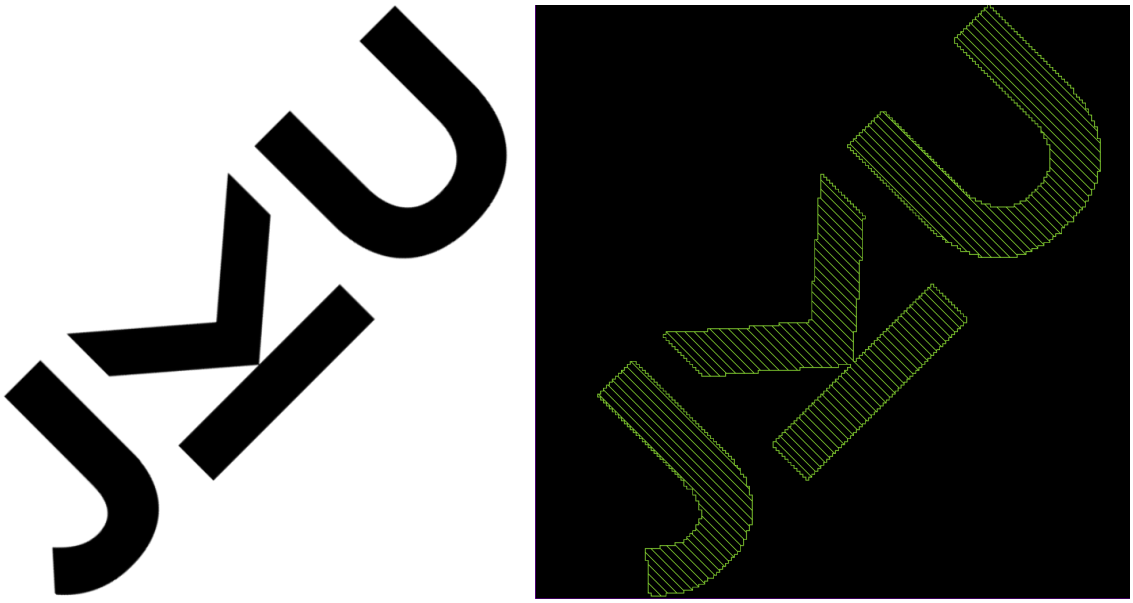
Target	What it does
`open`	Browse this folder with `sak-open.py` and open each file in its tool (needs the VNC/X11 desktop).
`clean`	Remove `final/` and `verification/drc/`.

To execute the `clean`, `logo`, `lef`, `lib`, `verilog`, `klayout-drc`, and `magic-drc` targets in sequence, run:

```
cd ip/sg13cmos5l_ip__jku                # or ip/sg13cmos5l_ip__jku_names
make all
```

View the JKU logo or JKU names layout in KLayout with:

```
ke final/gds/sg13cmos5l_ip__jku.gds     # or sg13cmos5l_ip__jku_names.gds
```



(a) `sg13cmos5l_ip__jku` logo. (b) `sg13cmos5l_ip__jku` layout in KLayout.

Figure 29: `sg13cmos5l_ip__jku` logo (left) and `sg13cmos5l_ip__jku` layout in KLayout (right).

> **Tip**
>
> To replace the logo with your own artwork, drop a square PNG image into `logo/` and either rename it to `jku_logo.png` / `jku_names.png`, or override `IMG_FILE` in the Makefile. A clean black-on-white image with a few hundred pixels per side and high contrast works best.

6.4 Top-Level Assembly

For top-level assembly, the top-level Makefile exposes two wrapper targets that descend into the IP folders and call `make all`:

```
make build-bondpad      # build ip/sg13cmos5l_ip__bondpad_70x70
make build-logos        # build ip/sg13cmos5l_ip__jku and ip/sg13cmos5l_ip__jku_names
```

Both targets are part of `make build-all` (see Section 7), so the IP cells are refreshed automatically before the chip is hardened with LibreLane. The generated GDS, LEF, Verilog stub and Liberty files are referenced by `flow/librelane/config.yaml` when assembling the chip top-level. Furthermore, the Verilog stubs are instantiated in `rtl/chip_top.sv`.

7 Top-Level Implementation

This section covers the top-level assembly and padframe generation. The top-level files that need to be understood and 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` signals.
- `rtl/chip_core.sv`, the core logic. It instantiates the digital and analog macros (`counter1`, `counter2`, `inverter1`, `inverter2`, `sram_0`) and wires them together and to the I/O pads.
- `flow/librelane/config.yaml`, the top-level LibreLane configuration. It defines the padframe order (`PAD_WEST`, `PAD_NORTH`, `PAD_SOUTH`, `PAD_EAST`), the chip area (`DIE_AREA`, `CORE_AREA`), the PDN core ring (`PDN_CORE_RING_VWIDTH`, `PDN_CORE_RING_HWIDTH`, `PDN_CORE_RING_VSPACING` and `PDN_CORE_RING_HSPACING`), the clock settings (`CLOCK_PORT`, `CLOCK_NET` and `CLOCK_PERIOD`), the macro list (`MACROS`), the bondpads, the logos, non-default routing rules (`NON_DEFAULT_RULES`, `DRT_ASSIGN_NDR`, and `RSZ_DONT_TOUCH_LIST`), STA corners, and references to `pdn_cfg.tcl` and `chip_top.sdc`.

- `flow/librelane/pdn_cfg.tcl`, the Tcl script 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 for the chip, specifications.md, pinout.md, and floorplan.md.

For your own chip, `make project` renames these files and every reference to them in one step (Section 8), and their content is what you then adapt.

7.1 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 update when the pinout changes.

The remaining code does not need to be modified for a pinout change, since the `generate` blocks automatically instantiate the correct number of I/O cells based on the parameter values. One `generate` block per pad type (`g_vdd_pads`, `g_vss_pads`, `g_iovdd_pads`, `g_iovss_pads`, `g_inputs`, `g_outputs`, `g_bidirs`, `g_analogs`) instantiates the matching IHP I/O cell, plus single instances for the clock and reset pads. An example code snippet of the input pad generation with a `generate` block is shown below. The `input_PAD` signals are exposed as ports of the `chip_top` module and are connected to the pads of the I/O cell, while the `input_PAD2CORE` signals are connected to the internal core logic in `chip_core.sv`. The same structure, or a very similar one, applies to the other pad types.

> **Note**
>
> Normally, only the parameter counts need to be edited in `chip_top.sv`. Only multi-clock and multi-reset chips would require modifications to the clock and reset pad generation, since the current template supports only one clock and one reset pad.

> **Note**
>
> The `sg13cmos5l_IOPadAnalog` cell exposes two internal core-side nodes per analog pad:
>
> - `padres`: path through the pad's **secondary** ESD resistor
> - `padbare`: **direct** path to the pad metal (only primary ESD diodes)
>
> In this template, `inverter2` uses both paths:
>
> - `vin1`/`vin2` are connected to `analog_PADRES[0:1]` (resistor-protected input path)
> - `vout1`/`vout2` are driven onto `analog_PADBARE[2:3]` (direct output path to avoid extra RC)

Inside `chip_core.sv`, these nets appear as the corresponding lowercase core signals `analog_padres[*]` and `analog_padbare[*]`.

Listing 6 `rtl/chip_top.sv (code snippet)`

```
generate
    for (genvar i = 0; i < NUM_ANALOG_PADS; i++) begin : g_analogs
        (* keep *)
        sg13cmos5l_IOPadAnalog analog_pad (
            `ifdef USE_POWER_PINS
            .iovdd   (IOVDD),
            .iovss   (IOVSS),
            .vdd     (VDD),
            .vss     (VSS),
            `endif
            .pad     (analog_PAD[i]),
            .padres  (analog_PADRES[i]),
            .padbare (analog_PADBARE[i])
        );
    end
endgenerate
```

7.2 Understanding `chip_core.sv`

The core `chip_core.sv` is where the macros are instantiated and wired to the chip's I/O pads. 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 input and output assignments at the top and bottom of the file correspond to the bit-to-role mapping documented in pinout.md. For improved clarity, every assignment is written on a separate line. Advanced designers can also use bus notation for more compact code.

7.3 Understanding `config.yaml`

The top-level LibreLane configuration `config.yaml` covers the following sections and references `pdn_cfg.tcl` and `chip_top.sdc`.

1. **Padframe generation.** `PAD_WEST`, `PAD_NORTH`, `PAD_SOUTH`, and `PAD_EAST`, in the order documented in `doc/pinout.md`.

2. **PDN core ring.** PDN_CORE_RING, PDN_CORE_RING_VWIDTH, PDN_CORE_RING_HWIDTH, PDN_CORE_RING_VSPACING, and PDN_CORE_RING_HSPACING.
3. **Clock settings.** CLOCK_PORT, CLOCK_NET, and CLOCK_PERIOD.
4. **STA corners.**
   - nom_fast_1p32V_m40C
   - nom_fast_1p65V_m40C
   - nom_slow_1p08V_125C
   - nom_slow_1p35V_125C
   - nom_typ_1p20V_25C
   - nom_typ_1p50V_25C
5. **Non-default routing rules.** NON_DEFAULT_RULES, DRT_ASSIGN_NDR, and RSZ_DONT_TOUCH_LIST.
6. **Macro, bondpad, and logo placement.** MACROS with per-macro files and placement (location, orientation), plus top-level bondpad and logo placement settings, together with chip area and placement constraints (DIE_AREA, CORE_AREA, FP_SIZING, PL_TARGET_DENSITY_PCT).

> **Note**
>
> The NON_DEFAULT_RULES section defines a dedicated routing rule for the analog traces of inverter2 (PADRES and PADBARE). In this template, NDR_analog sets a wider wire width and larger spacing for the selected metals and vias, which helps reduce coupling and preserve analog signal quality.
> The DRT_ASSIGN_NDR section applies this rule during detailed routing by matching net names with regular expressions (^.*PADRES.*$ and ^.*PADBARE.*$). The RSZ_DONT_TOUCH_LIST then protects the same analog nets from resizing, buffering, or other optimization steps, so the routed analog paths remain unchanged.

7.4 Understanding pdn_cfg.tcl

pdn_cfg.tcl is the OpenROAD callback that builds the power-distribution network (PDN). Most of the file follows the standard structure provided by LibreLane and OpenLane.

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

- A default macro grid (Metal4 to TopMetal1, the vertical and the horizontal top-level PDN layer) used by the two counters.
- A custom grid for inverter1 and inverter2 with the same Metal4 to TopMetal1 connect. Each inverter macro has its own internal power ring whose horizontal segments are on Metal3 and whose vertical segments are on Metal4, so TopMetal1 stays free: the top-level

TopMetal1 stripes cross the ring and feed it, while no top-level stripes are routed inside the ring. A ring that also occupied TopMetal1 would make pdngen cut every top-level stripe at the macro and leave the ring unfed.

- A custom grid for sram_0 (a plain Metal4 to TopMetal1 connect, because the SRAM exposes its supplies as vertical Metal4 stripes, which is already the top-level vertical PDN layer, so the top-level TopMetal1 stripes drop via stacks onto every pin and no bridging stripes are needed).

When new macros are added, the matching define_pdn_grid plus add_pdn_connect block at the bottom must be extended (or the default grid is used if the macro layers line up with the top-level grid). The custom grid for the inverters is shown below as an example. The comments explain the reasoning behind the layer choices and the connection structure.

7.5 Top-Level Simulation

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

```
make sim-rtl-cocotb
make sim-gl-cocotb
make sim-view-cocotb                              # use GTKWave as waveform viewer
make sim-view-cocotb WAVEFORM_VIEWER=surfer       # use Surfer as waveform viewer
```

The cocotb testbench at top level is testbenches/cocotb/chip_top_tb.py, and it covers the global enable, reset, counters, and bidir behaviour documented in doc/specifications.md. The result is shown in Figure 30.

For mixed-signal simulation, the testbenches/xschem/chip_top_tb_tran.sch testbench can be opened in Xschem for interactive simulation, the same way as for the counter or the inverter, and is shown in Figure 31.

```
cd testbenches/xschem
xschem chip_top_tb_tran.sch
```

Again, the simulation and plotting can be run non-interactively from the command line with the following targets.

```
make sim-gl-xschem
make sim-view-xschem SCRIPT=plot_chip_top
```

sim-gl-xschem runs in batch mode: it netlists the testbench with xschem netlist and then calls ngspice -b directly, so make blocks until the run finishes and sees its exit status.

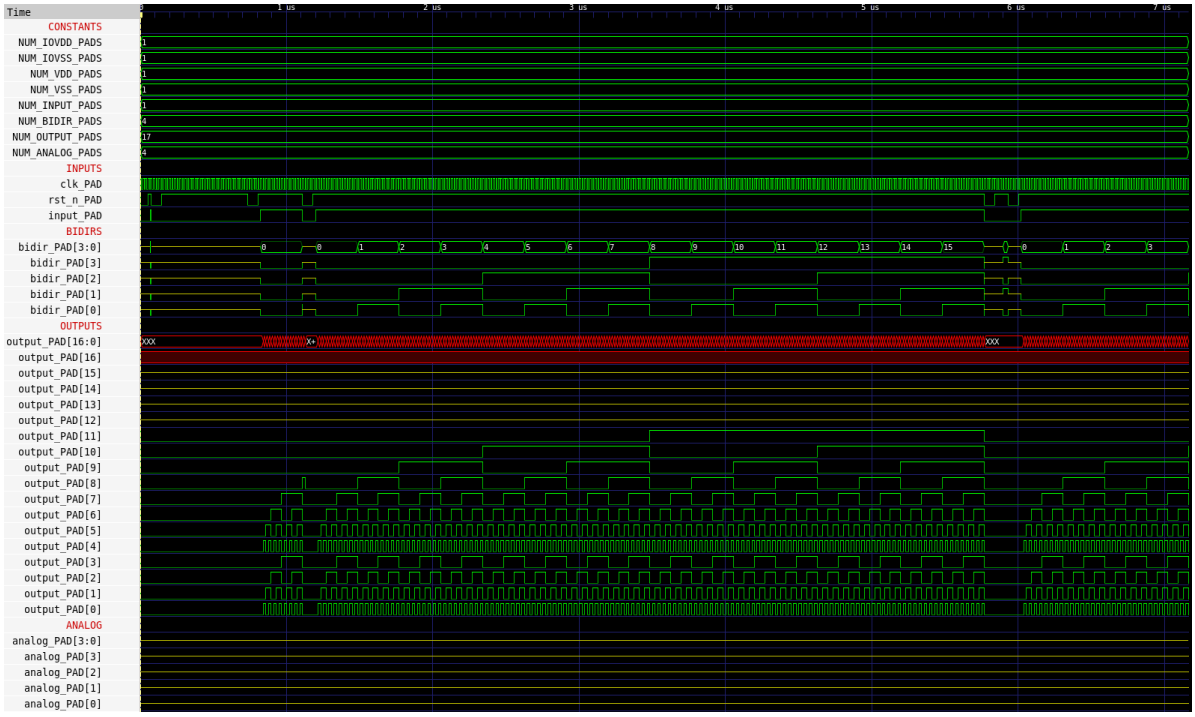


Figure 30: `chip_top` cocotb waveform in GTKWave.

The testbench is selected with `TB=<testbench_name>` (default: `<CELL>_tb_tran`). Since the run is headless, the `plot` commands in the testbench's `.control` block are a no-op. The testbench exports its results with `wrdata` to `testbenches/xschem/plot_simulations/data/`, from where `sim-view-xschem` plots them with Matplotlib.

> **Note**
>
> The top-level Xschem testbench converges, but it may take a long time depending on the hardware used. `make sim-gl-xschem` is part of `sim-all`, so a full `make sim-all` includes it. For this tutorial, the simulation data is already available in the repository in `testbenches/xschem/plot_simulations/data/chip_top_tb_tran.txt`, so `make sim-view-xschem SCRIPT=plot_chip_top` can be run without running the simulation first.

The following plots show the results of the top-level transient simulation. The first plot shows the analog inverter response, while the other three plots show the digital outputs of the two counters, the digital inverter, and the bidirs.

When the testbench is opened in Xschem, the schematic of `chip_top` can be inspected with the shortcuts **`e`** / **`Ctrl + e`**, and the symbol can be shown with **`i`** / **`Ctrl + e`** as explained in Section 5. The symbol and schematic are shown in Figure 36 and Figure 37, respectively. The schematic shows the connections between the padframe and the core macros, while the symbol

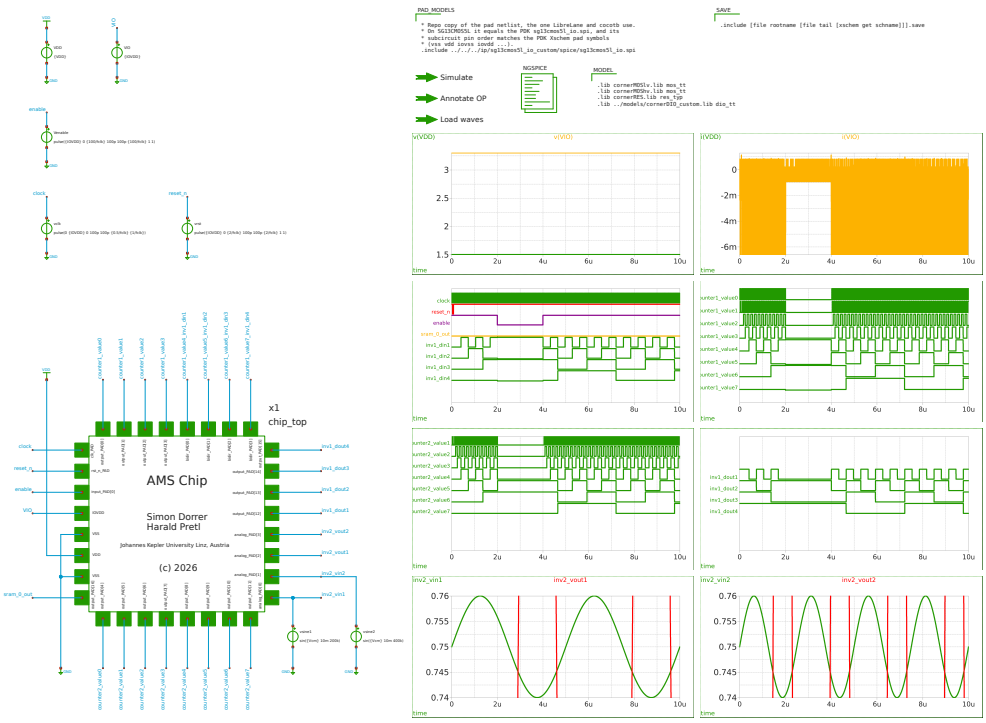


Figure 31: Xschem top-level mixed-signal testbench `chip_top_tb_tran`.

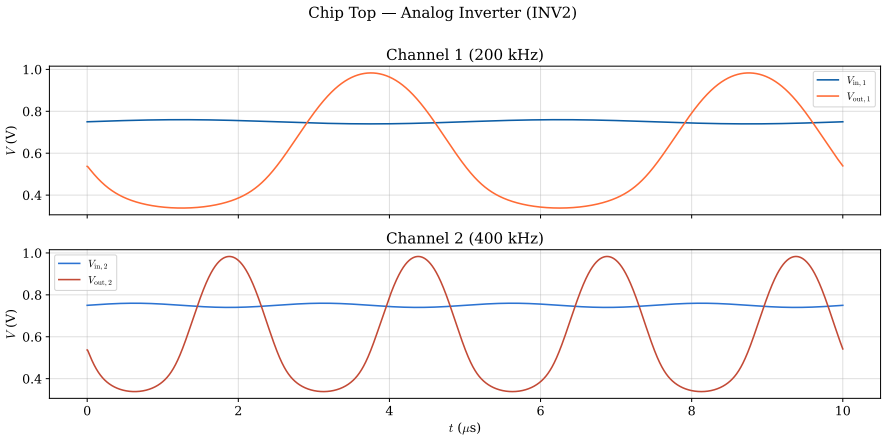


Figure 32: Plotted simulation result of `chip_top_tb_tran`: analog inverter 2 response.

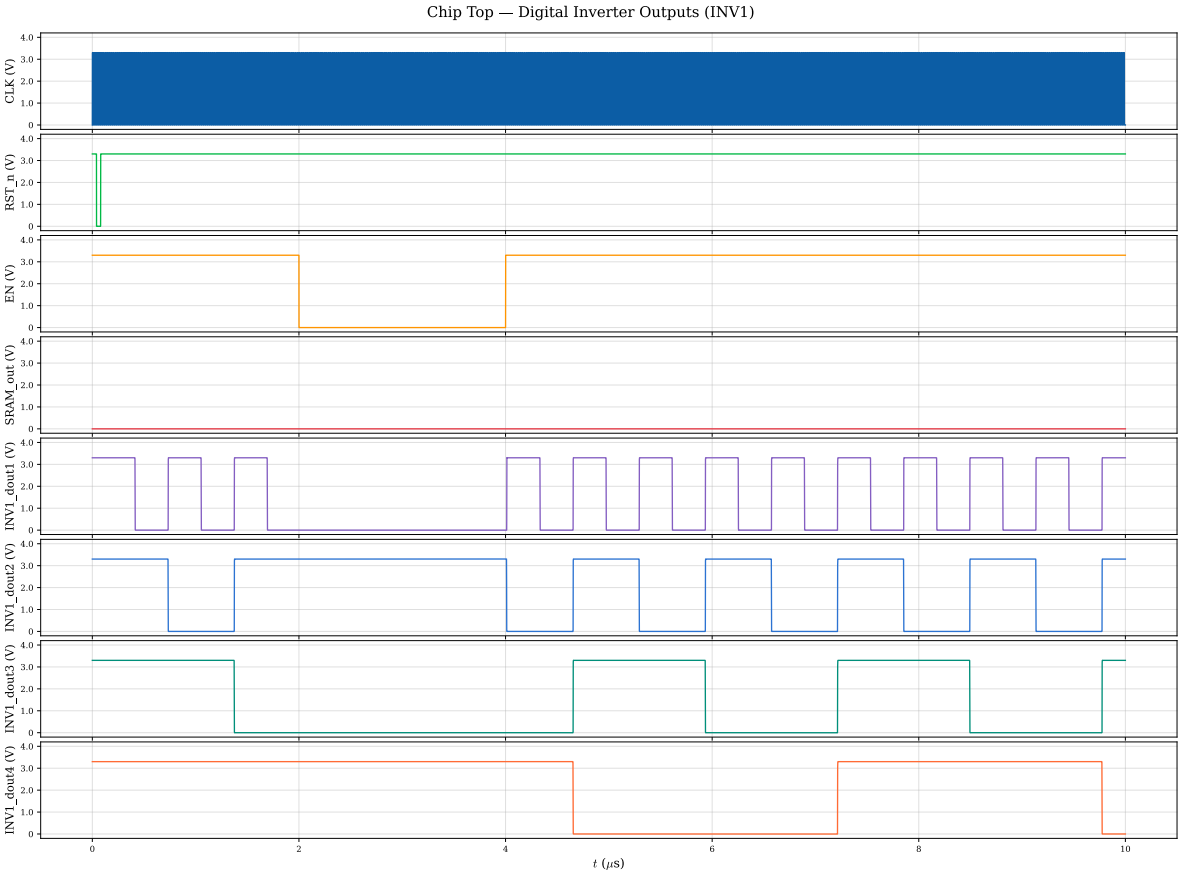


Figure 33: Plotted simulation result of `chip_top_tb_tran`: digital inverter 1 outputs.

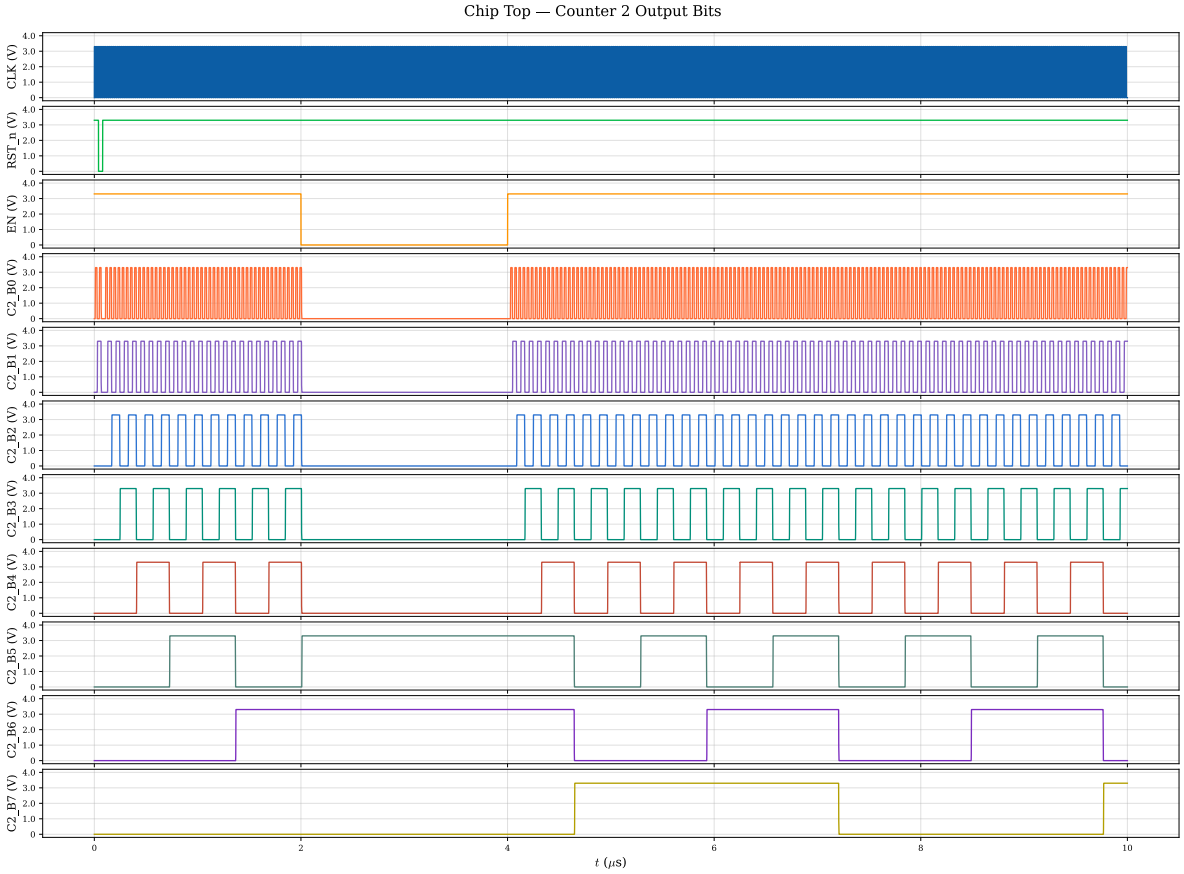


Figure 34: Plotted simulation result of `chip_top_tb_tran`: digital counter 2 outputs.

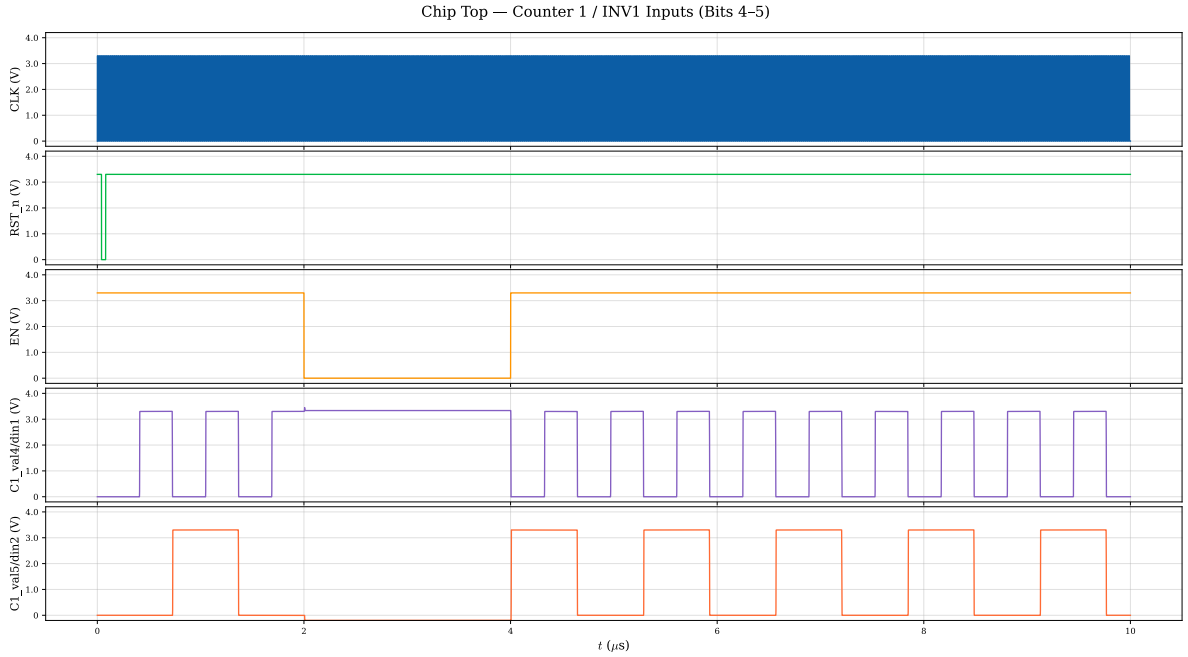


Figure 35: Plotted simulation result of `chip_top_tb_tran`: digital counter 1 / inverter 1 bidirs.

shows the pinout of the chip. The SRAM is not included in the schematic, since IHP does not ship a SPICE model or an Xschem symbol for the `RM_IHPSG13_1P_1024x32_c2_bm_bist` macro.

7.6 Building the Chip

When the floorplan and pinout are planned and configured, 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
```

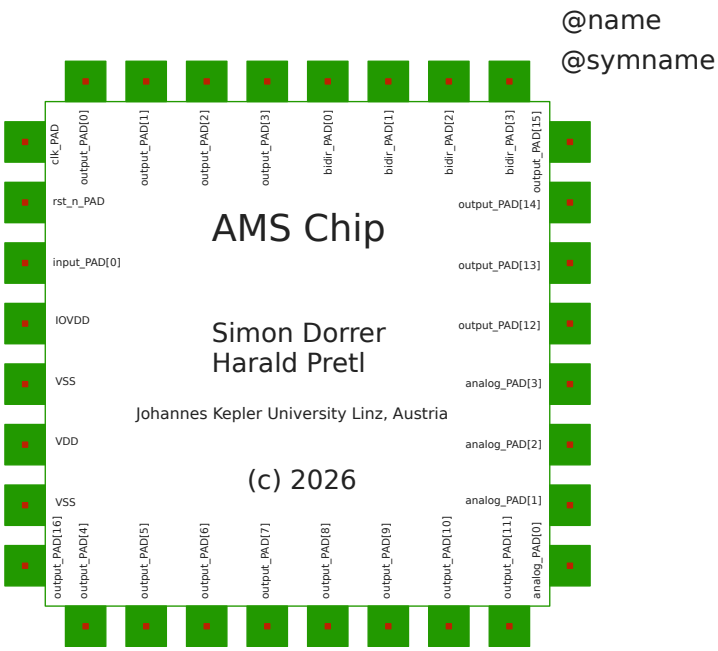


Figure 36: Xschem symbol of `chip_top`, showing the pinout of the chip.

Internally, `build-top` runs `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`.

The renders generated by `sak-render.py`, which runs during the `render-gds` step, are located in `render/img/`. The verification reports (Yosys, antenna, STA, IR-drop, LVS, manufacturability) from the top-level LibreLane flow can be found in `verification/reports/`. The netlists are located in `netlist/`.

> **Note**
>
> Note that currently the LibreLane flow streams out the top-level GDS `layout/chip_top.gds.gz` without the logos and the fill structures. To add the chip logo (PNG → GDS) and the fill structures on top of the LibreLane output, run:
>
> ```
> make add-logo-fill
> ```
>
> This calls `scripts/add_logo_fill.sh` with `TOP` and writes `layout/chip_top_logo_fill.gds.gz`. This step is already called by `make build-top`. In the future, it is planned to replace this script and Makefile target with a custom LibreLane step.

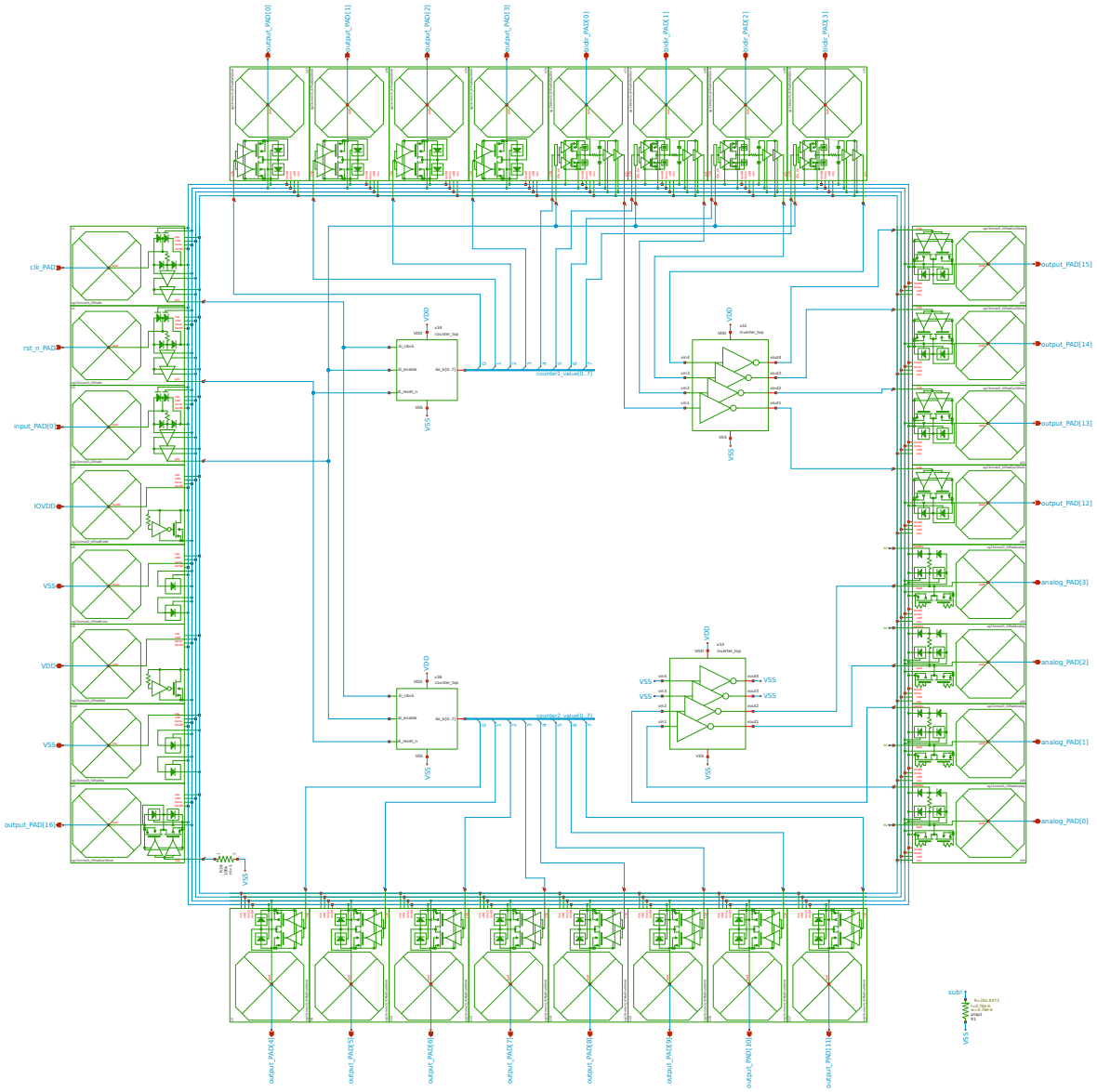


Figure 37: Xschem schematic of `chip_top`, showing the connections between the padframe and the core macros.

> **Tip**
>
> The difference between `chip_top.gds.gz` and `chip_top_logo_fill.gds.gz` is that the former is the raw top-level layout without any logos or filler cells, while the latter includes the logos and filler cells added for manufacturability and cosmetic purposes. For visual debugging in KLayout, it is recommended to use `chip_top.gds.gz` to avoid the visual clutter of the logos and filler, while for DRC and LVS, `chip_top_logo_fill.gds.gz` should be used since it is the actual layout that will be taped out.

7.7 Viewing the Chip

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

```
ke layout/chip_top.gds.gz              # Chip layout without logos and filler
ke layout/chip_top_logo_fill.gds.gz    # Chip layout with logos and filler
```

7.8 Top-Level Verification

DRC, LVS, and PEX are also available for the full chip.

7.8.1 DRC

The following DRC targets are available for the top-level.

```
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 on chip_top_logo_fill
make klayout-drc-regular                   # full KLayout DRC on chip_top_logo_fill.gds.gz (sl
```

`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. `magic-drc` is faster than `klayout-drc`, but it does not support the full set of DRC rules and is less accurate on the complex top-level layout, so it is recommended to use both.

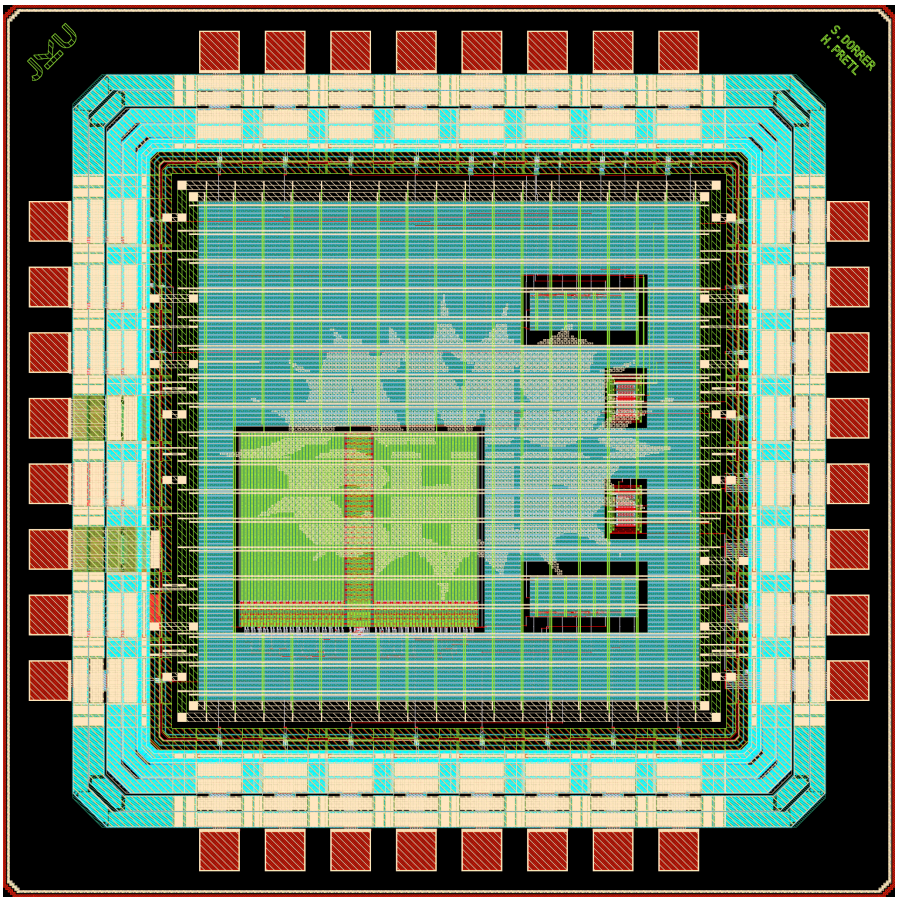


Figure 38: `chip_top_logo_fill` layout in KLayout (“all” layers).

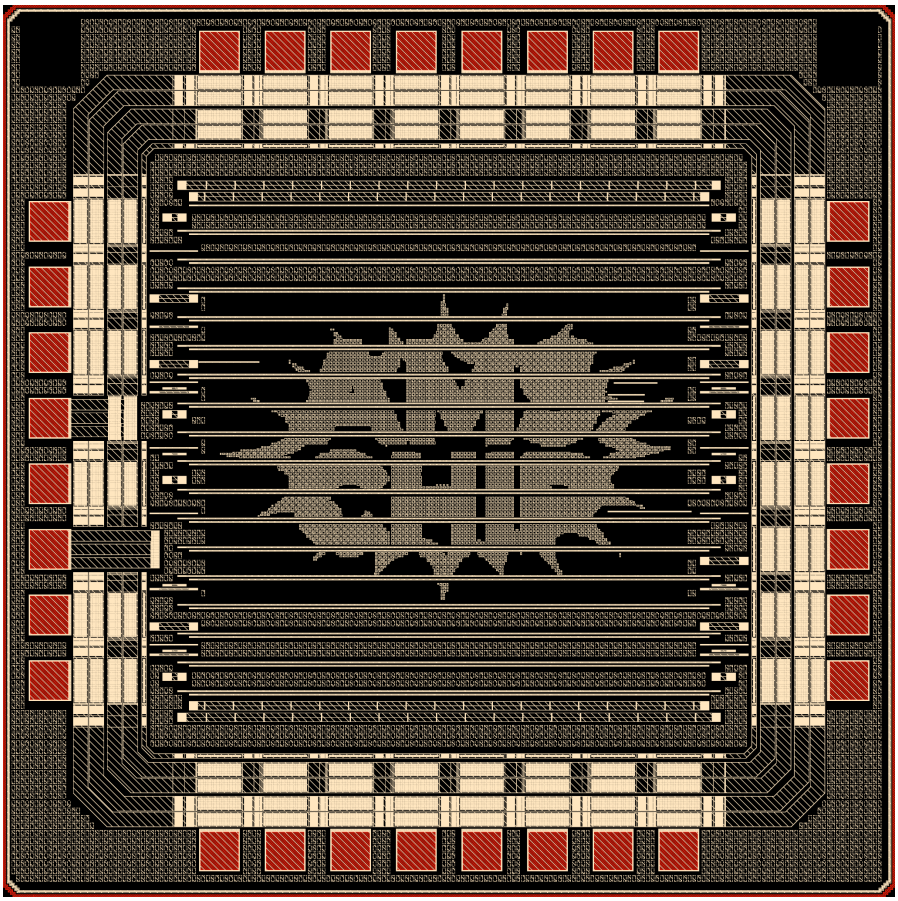


Figure 39: `chip_top_logo_fill` layout in KLayout with only `TopMetal1` selected.

7.8.2 LVS

The following LVS targets are available for the top-level.

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

> **Warning**
>
> The top-level LVS is currently under construction. The flow is in place, but some issues and limitations in the ihp-sg13cmos5l PDK must be fixed before it can run successfully.

7.8.3 PEX

The following PEX target is available for the top-level.

```
make magic-pex                                # only chip_top (without logo and filler), default
make magic-pex EXT_MODE=3 THRESHOLD=5000 MINRES=500 MINDELAY=2   # full-RC with custom extres
```

For full-RC extraction (`EXT_MODE=3`), `magic-pex` also exposes the `sak-pex.sh extresist` tuning parameters `THRESHOLD` (`-t`, default `10000` mOhm), `MINRES` (`-r`, default `1000` mOhm) and `MINDELAY` (`-y`, default `1` ps; `0` = gate by resistance). They are ignored in `EXT_MODE=1/2`.

As in the macros, `magic-pex` starts by running `make symbol-pex`, which rebuilds `schematic/xschem/chip_top_pex.sym` from `chip_top.sym` with `type=subcircuit` changed into `type=primitive`, so the PEX symbol always matches the current chip symbol.

> **Warning**
>
> At the top level, PEX is only run on `chip_top` (without logo and filler), and the default `EXT_MODE=1` (C-decoupled) is used to keep the runtime manageable. However, the extracted netlist `chip_top_magic_pex_1.spice` is still huge (~75 MB, ~566k lines), and the post-layout simulation must be run on a powerful server.

7.9 Packaging

The die is packaged in a QFN-32 (5 × 5 mm) open-molded plastic package (OmPP) from the EUROPRACTICE ASIC packaging offering. The package matches the padframe exactly: all 32 bondpads (8 per side, see Section 7) are wired one-to-one to the 32 package leads (8 per

side). Bonding is done with 25 µm gold wire, the longest wire measures 1.70 mm, and the cover is glued on after bonding.

The bonding diagram below is generated fully automatically with `make bondplan` from the final chip GDS (`layout/chip_top_logo_fill.gds.gz`) and the EUROPRACTICE package library `EP_PACKAGES_08022018.gds`. The flow extracts the die bondpads together with their names, places the die in the package cavity, draws the bondwires, fills in the title block, and checks wire lengths, crossings, spacing, lead skew, and the clearance around the sensitive analog wires. The pinout and all flow settings are kept in a single configuration file, so the diagram never goes out of sync with the chip. Details are documented in the packaging README.

```
make bondplan                          # uses the default VERSION (1.0.0)
make bondplan VERSION=2.1.0            # stamp another version on the sheet
```

The `VERSION` variable is passed to the flow and printed in the title block (`DIE: CHIP_TOP - V.1.0.0`), so the version number is maintained in the Makefile only. The outputs are the bondplan GDS (`packaging/layout/chip_top_bondplan.gds`), the bond report with summary and bond table (`packaging/result.md`), and the bonding diagram images (`packaging/render/chip_top_bondplan_{white,black}.{png,svg}`).

> **Tip**
>
> Check the flow log and `packaging/result.md` after every run. Unbonded named pads and NC package pins are reported explicitly, so a forgotten connection in the `PINOUT` table is immediately visible. Warnings are raised for bondwires that are too long (`BONDWIRE_MAX_LENGTH`), cross each other, come too close to each other, land skewed on their lead (`BONDWIRE_MAX_SKEW`), or violate the guard clearance of sensitive wires (`GUARDED_PINS` / `GUARD_SPACING`).

7.10 Full Flow at Once

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

```
make all
```

This calls `build-all` first, then Magic DRC on both `chip_top` and `chip_top_logo_fill`, then `sim-all` (RTL and gate-level cocotb plus the gate-level Xschem run), and finally `bondplan`. The simulations therefore run on the netlists that this build has just produced. To shorten the runtime, the KLayout DRC is currently not included in `make all`. However, depending on the hardware used, `make all` can still take a while.

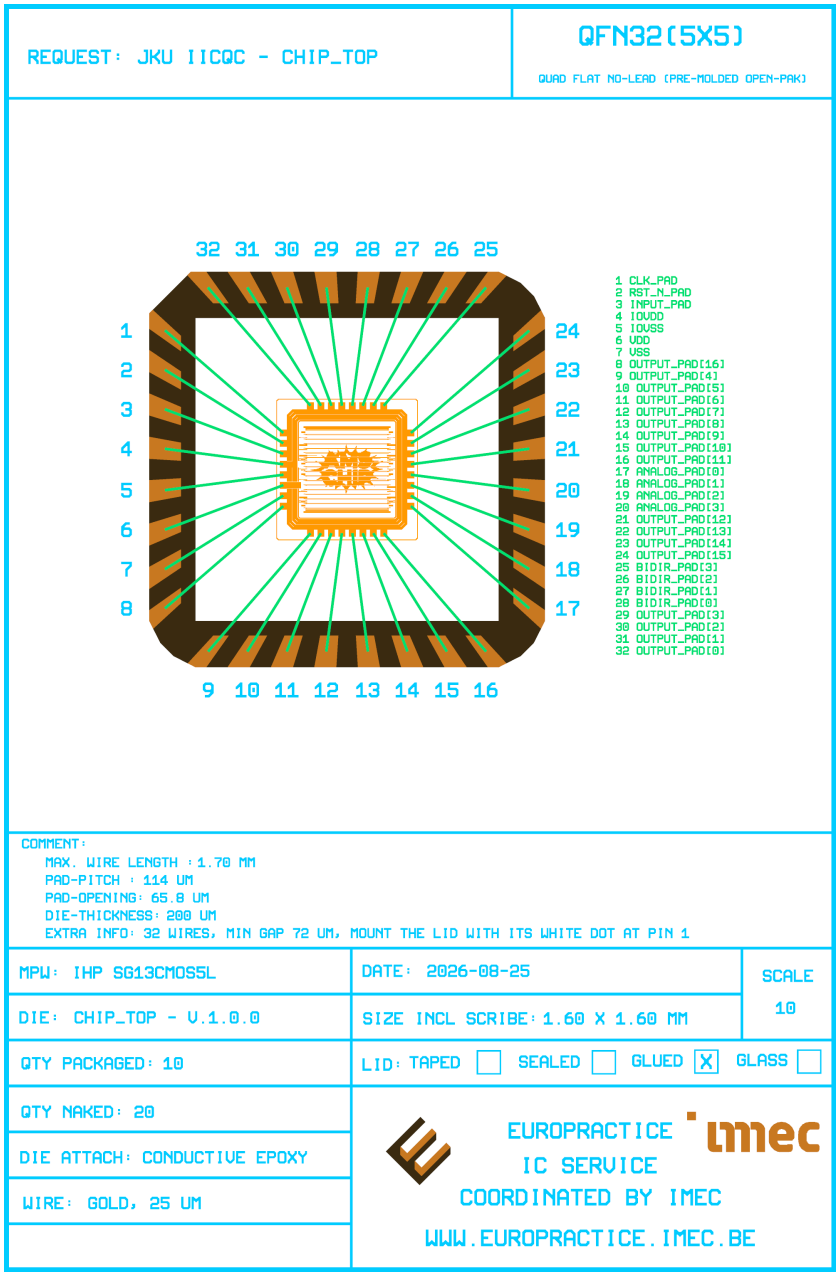


Figure 40: Automatically generated bonding diagram of the ihp-sg13cmos5l AMS template chip in a QFN-32 package.

7.11 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/{schematic,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.

8 Start Your Own Chip

The template is meant to become your own chip. Work through this tutorial on the template as it is, then start your chip from a copy of it: create your own repository with the green "Use this template" button on GitHub (or fork or copy the repository) and name it `ihp-sg13cmos5l-<name>` after the chip, clone it, and enter the IIC-OSIC-TOOLS container. In the top-level folder of the clone, one target turns the copy into a new chip with its own name:

```
make project NAME=<name> DRY_RUN=1   # list the changes, touch nothing
make project NAME=<name>             # start the chip <name>
```

`NAME` is a lowercase letter followed by lowercase letters, digits or underscores, for example `mychip`. It must not be the name of a macro or an IP: with `NAME=counter`, the chip top `counter_top` would be the same cell as the top cell of the counter macro. The target needs no PDK.

> **Tip**
>
> With the repository named `ihp-sg13cmos5l-<name>`, for example `ihp-sg13cmos5l-mychip`, `git clone` gives its folder the same name, and `make project` writes that name into the directory tree of the README. If your clone has another name, rename it before `make project`: rename the repository on GitHub (Settings > General, the old URLs redirect), point `origin` at the new URL with `git remote set-url origin <URL>`, close VS Code and every terminal in the folder, and rename the folder from its parent folder. `make project` lists the same steps when the folder name differs.

8.1 What `make project` Does

1. **It checks before it changes anything.** It refuses to run in a git tree with uncommitted changes, so that `git reset --hard && git clean -fd` can undo the run completely, and in a checkout whose `origin` is still the template repository. `FORCE=1` skips both checks. Every change is planned first, so a refused run, for example because of a name that is already in use, leaves the tree untouched.
2. **It deletes the outputs of the template chip** with `make clean`: the GDS in `layout/`, the netlists, the renders, the reports, the simulation results and the bondplan. They carry the old name and come from a build of the template chip. The macros and IPs keep their committed views, so the new chip builds without rebuilding them.
3. **It removes what only makes sense in the template**: this tutorial (`tutorial/`) and the workflow that publishes it, `doc/ihp-structure-proposals/`, `CITATION.cff`, the content of `release/`, and every block between `template-only` markers in the README, `AGENTS.md`, the `Makefile` and the configuration files. Figures that a remaining file still shows move to `doc/fig/`, and `ToDo.md` starts empty.
4. **It renames the chip.** `chip_top`, `chip_core` and `chip_logo` become `<name>_top`, `<name>_core` and `<name>_logo`, in file names and in file contents (see Table 7). `TOP` in the `Makefile` follows, and with it every target that takes the name from `TOP`, including the logo and fill step and the bondplan.
5. **It gives the README a new header** that records the template version it started from, and deletes itself: `scripts/new_project.py` and the `project` target are gone afterwards.
6. **It checks the result.** No old name, no link into a deleted path and no `template-only` marker may be left, and `reuse lint` has to pass when it is installed. The report at the end lists what is left for you.

Table 7: Names in the template and after `make project NAME=mychip`.

Template	After `make project NAME=mychip`
`TOP = chip_top` in the `Makefile`	`TOP = mychip_top`
`DESIGN_NAME: chip_top` in `config.yaml`	`DESIGN_NAME: mychip_top`
`rtl/chip_top.sv`, `rtl/chip_core.sv`	`rtl/mychip_top.sv`, `rtl/mychip_core.sv`
`flow/librelane/chip_top.sdc`	`flow/librelane/mychip_top.sdc`
`schematic/xschem/chip_top.sch`, `.sym`	`schematic/xschem/mychip_top.sch`, `.sym`
cocotb `chip_top_tb.py`	`mychip_top_tb.py`
Xschem `chip_top_tb_tran.sch`	`mychip_top_tb_tran.sch`
the instance `i_chip_core`	`i_mychip_core`
`flow/logo/chip_logo_mono.png`	`flow/logo/mychip_logo_mono.png`
`layout/chip_top_logo_fill.gds.gz`	`layout/mychip_top_logo_fill.gds.gz`, written by the first build

8.2 After `make project`

Review the result, commit it, and build the new chip:

```
git status              # undo: git reset --hard && git clean -fd
git add -A
git commit -m "started <name> from ihp-sg13cmos5l-ams-chip-template"
make init-submodules    # the logo and fill step needs flow/artistic
make build-top          # or make all, which also fetches it
```

The first build writes the layout, the netlists, the renders and the reports under the new name (see Section 7). Then make the chip your own:

- **Your own blocks.** In `macros/`, start a digital or an analog macro as a renamed copy of an example (see Section 4, Section 5 and Section 9.3):

  ```
  make macro FROM=counter NAME=<macro>    # digital macro
  make macro FROM=inverter NAME=<macro>   # analog macro
  ```

  The macros README lists every place where a new macro is registered at the chip top-level. Remove the example macros once your own ones replace them.

- **Floorplan, pinout and package.** Adapt the die area, the pad ring and the macro placements in `flow/librelane/config.yaml` (see Section 7), the `PINOUT` map in `packaging/config.yaml` (see Section 7.9), and the chip documentation in `doc/`. Section 9.4 and Section 9.5 walk through exactly these changes.

- **Regression.** The `regression` target, which CI runs on every push to `main`, tests the counter and inverter macros. Point it at your own macros once the examples are gone.

> **Important**
>
> The logos stay, and they put the template's identity on the die. Replace them before a tapeout, Section 6 shows how the logo IPs are generated from PNG images:
>
> - `ip/sg13cmos5l_ip__jku_names/` draws the names of the template authors.
> - `flow/logo/<name>_logo_mono.png` draws the "AMS CHIP" artwork on TopMetal1 (see Figure 39).
> - `ip/sg13cmos5l_ip__jku/` draws the JKU logo, replace it where it does not apply.

> **Note**
>
> `make project` deletes this tutorial from the new chip, but it stays available online. The template's top-level README describes the target as well.

9 Additional Exercises

The following exercises repeat the corresponding sections of this tutorial with modifications to the design.

9.1 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 $2^{16} - 1 = 65535$. The recommended steps are:

1. Update the RTL (`counter.sv`, `counter_top.sv`, `constants.sv`). Change the bitwidth 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 pass timing constraints.
5. Update the bus range of the output pin in the Xschem symbol (`counter_top.sym`), and re-run the gate-level Xschem simulation. The XSPICE model itself is regenerated automatically by `make build-top` (or manually with `make generate-xspice`), and the PEX symbol `counter_top_pex.sym` together with the extracted netlist by `make magic-pex`. Forgetting this step no longer costs you a debugging session: `make symbol-check` runs inside `generate-xspice` and names the eight ports the symbol is now missing. Run `make symbol-gl FORCE=1` in a scratch copy of the macro if you would rather read the widened pin off a fresh scaffold than edit the range by hand.

> Bonus for advanced designers
>
> Replace the binary counter with a 16-bit Gray counter (only one bit toggles per transition).

9.2 Exercise 2: Self-Biased Single-Ended Inverter Amplifier

Repeat Section 5 with a self-biased single-ended inverter amplifier by adding a feedback resistor R_f between the output and the input of the inverter. As discussed in Section 5.5, the bare CMOS inverter has poor PVT robustness because nothing pins the gate bias to $V_\mathrm{DD}/2$. A feedback resistor closes a DC loop that forces the operating point to the high-gain crossover ($V_\mathrm{in} \approx V_\mathrm{out} \approx V_\mathrm{DD}/2$) regardless of process corner, which also linearises the small-signal transfer characteristic and turns the inverter into a usable single-ended amplifier. The disadvantage of this technique is the reduced input resistance, which equals $R_\mathrm{in} \approx 1/(g_\mathrm{m,NMOS} + g_\mathrm{m,PMOS})$. The recommended steps are:

1. Open the inverter schematic (`inverter.sch`) in Xschem and add a resistor between the `vin` and `vout` nets of the inverter. Use the `rhigh.sym` device from the `sg13cmos5l_pr` PDK library, the high-sheet poly resistor. Keep `spiceprefix=X` and connect its `body` terminal to `VSS`. The symbol computes the value from the geometry: with a sheet resistance of 1360 /sq this is roughly 2.7 k per m of length at $W = 0.5$ m, so keep W at 0.5 m and set the value with L (the `b` parameter folds a long resistor into segments). Start with a value of $R_f = 1$ M and adjust as needed. What can you observe?
2. Re-run the schematic-level testbenches from Section 5 and observe the change in behaviour. First, always carefully think about what the expected behaviour would be before running the simulations, and then check if the results match your expectations. Note that the ac open-loop testbench drives `vin` directly with a voltage source, which overrides the self-biasing. To observe it, also try removing the dc bias from the input (for example by coupling the source through a large capacitor) and let the feedback set the operating point.
3. Re-run the full schematic-level simulation suite with `make sim-all` to refresh the plotted results in `fig/inverter_top/`.
4. Re-run the CACE flow with `make sim-cace` and compare the new process and mismatch plots against Figure 19 and Figure 21. The spread of A_{ol} across corners should be visibly narrower than the open-loop reference.
5. [Optional] Sweep R_f in the CACE characterisation YAML (`inverter.yaml`) to study the circuit behaviour across a range of feedback resistor values.
6. [Optional] Once the schematic results are convincing, propagate the resistor to the layout in `layout/` with the `rhigh` PCell, using the same W and L as in the schematic and dummy resistors of minimum length above and below it, and re-run the verification chain (`make klayout-verify`, `make magic-verify`) plus `make build-top`. Compare the post-layout gain and bandwidth against the schematic level as in Section 5.9.4.

> **Tip**
>
> Choose R_f carefully. Too small, and the loop loads the amplifier output and degrades the closed-loop gain. Too large, and the time constant with the input parasitic capacitance dominates, so the amplifier loses gain at the low end of the signal band. A value in the 1–10 M range is a sensible starting point for this inverter sizing.

9.3 Exercise 3: Three-Stage Ring Oscillator with Output Buffer

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

1. Start the new macro as a renamed copy of the inverter with `make macro FROM=inverter NAME=ringosc` in `macros/`. The target renames the files, the cells inside the GDS files and every reference, see the macros README for what it does and what is left.

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`) to expose a single output net, instead of `vin1..vin4` and `vout1..vout4`. The PEX symbol `inverter_top_pex.sym` is rebuilt from it by `make symbol-pex`, which the PEX targets run automatically.

4. Update the transient testbench (`testbenches/xschem/inverter_top_tb_tran.sch`) to measure the oscillation frequency. Remove the input source. Since the delay per stage is small, simulate a few hundred nanoseconds with a fine step, and measure the period between two consecutive rising crossings of V_{cm} on the output net:

```
meas tran t_rise1 when v(vout)=Vcm rise=1
meas tran t_rise2 when v(vout)=Vcm rise=2
let T_osc = t_rise2 - t_rise1
let f_osc = 1/T_osc
print f_osc
```

   `Vcm` is visible inside the `.control` block only through a `.csparam Vcm=Vcm` line next to `.param Vcm=VDD/2`, as the dc testbench shows (see Section 5.3.1). The expected relation is $f_{\mathrm{osc}} = 1/(2 \cdot N \cdot t_{\mathrm{d}})$ with $N = 3$ stages and t_{d} the effective stage delay. An ideal simulation can rest on the metastable point where all stages sit at V_{cm}, so if the ring does not start, kick it with an initial condition on one ring node, for example `.ic v(osc)=0`. Delete testbenches that are no longer relevant.

5. Resize the unit inverter and watch the frequency. Vary `w`, `l`, and the finger count `ng` of M_1 and M_2 in the unit cell schematic (`ringosc.sch`), and re-run the transient simulation for each variant. Predict the outcome before you simulate, and use Section 5.6.1 as your guide.

6. [Optional] Adapt the CACE `inverter.yaml` to characterise the oscillation frequency across PVT.

7. Update the layout in KLayout. The existing `inverter_top.klay.gds` can be used as a starting point, but the internal routing has to be modified to form the ring and the buffer stage. After editing, export a clean `ringosc_top.gds`. Keep the box on the `prBoundary` layer (`189/0`) and resize it with the macro, otherwise the top-level flow cannot place it. `make check-boundary` verifies this.

8. Re-run the verification chain, `make klayout-verify`, `make magic-verify`, and `make build-top`. Then compare the post-layout oscillation frequency against the schematic level as in Section 5.9.4. The parasitics add delay per stage, so the frequency drops.

9.4 Exercise 4: Update Pinout and Floorplan of the Top-Level

Building on Section 9.1 and Section 9.3, 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` values.
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 7).
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 routing-grid alignment of inverter-like macros (`X` a multiple of 0.48 µm, `Y` a multiple of 0.42 µm).
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`.

9.5 Exercise 5: Update the Bondplan Configuration

Building on Section 9.4, update the bondplan configuration in `packaging/config.yaml` so that the packaged chip matches the new pinout (ring oscillator instead of `inverter2`, 16-bit `counter1`, no `counter2`, three additional VSS pads). The recommended steps are:

1. Rebuild the final GDS of the modified chip (`make build-all` from Section 9.4), since the bondplan flow reads the die pads and their names directly from `layout/chip_top_logo_fill.gds.gz`.
2. Update the `PINOUT` table in `packaging/config.yaml`: the south-side pads now carry `counter1_value[15:8]`, three of the four east-side analog pads became VSS pads, and the fourth carries the ring-oscillator output. Keep the geometric order in mind (pins 1-8 left top→bottom, 9-16 bottom left→right, 17-24 right bottom→top, 25-32 top right→left) so that the bondwires do not cross.
3. Connect the freed-up VSS pads as downbonds to the exposed pad by adding an `EPAD` list (e.g. `EPAD: ["VSS", "VSS", "VSS"]`) and marking their former leads as NC (`~`). The exposed pad of the QFN-32 is the package ground, and the die is attached with conductive epoxy, so ground downbonds are shorter and lower-inductance than wires to leads.

4. Re-run `make bondplan` and read the flow log and `packaging/result.md` carefully: only the deliberately unbonded pads may appear as *unbonded named pads* (plus the seal-ring artifact), and there must be no wire-length, crossing, spacing, lead-skew, or guard-clearance warnings.
5. Inspect the generated bonding diagram in `packaging/render/` and the bondplan GDS in KLayout (`ke packaging/layout/chip_top_bondplan.gds`). The title block of the A4 drawing sheet (die size, pad pitch, wire count, maximum wire length) is refilled automatically on every run.
6. [Optional] Experiment with the die placement: rotate the die with `DIE_PLACEMENT.orientation` (e.g. `E`) or move it off-center with `DIE_PLACEMENT.location`, re-run the flow, and observe how the wire-length, spacing, and skew checks react. Restore a placement where all checks pass.

10 RFIC Flow

Are you interested in an open-source RFIC flow? Check it out here.

11 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 WITH SHL-2.1
// 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 WITH SHL-2.1
// 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 `macros/counter/flow/librelane/config.yaml (code snippet)`

```
# Layout / Floorplan Rules
DIE_AREA: [0, 0, 200, 100]
PL_TARGET_DENSITY_PCT: 40

# Valid values are AREA 0-3 and DELAY 0-4, LibreLane defaults to AREA 0
# DELAY favours speed, AREA favours cell area
# AREA 3 and DELAY 4 always add buffering and gate sizing, so AREA 3 can be larger than AREA
SYNTH_STRATEGY: DELAY 4
```

Listing 4 `rtl/chip_top.sv (code snippet)`

```
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 5 `rtl/chip_top.sv (code snippet)`

```
generate
    for (genvar i = 0; i < NUM_INPUT_PADS; i++) begin : g_inputs
        sg13cmos5l_IOPadIn input_pad (
            `ifdef USE_POWER_PINS
            .iovdd (IOVDD),
            .iovss (IOVSS),
            .vdd   (VDD),
            .vss   (VSS),
            `endif
            .p2c   (input_PAD2CORE[i]),
            .pad   (input_PAD[i])
        );
    end
endgenerate
```

Listing 7 `rtl/chip_core.sv (code snippet)`

```
// ======================================================
// Input Assignments
// ======================================================
logic enable;
assign enable = input_in[0];

// Inverter 1
wire inv1_din1 = bidir_in[0];
wire inv1_din2 = bidir_in[1];
wire inv1_din3 = bidir_in[2];
wire inv1_din4 = bidir_in[3];

// Inverter 2
wire inv2_vin1 = analog_padres[0];
wire inv2_vin2 = analog_padres[1];
// ======================================================
```

Listing 8 `rtl/chip_core.sv (code snippet)`

```
// ====================================================
// Output Assignments
// ====================================================
// Digital outputs
// Counter 1
assign output_out[0]  = counter1_value[0];
assign output_out[1]  = counter1_value[1];
assign output_out[2]  = counter1_value[2];
assign output_out[3]  = counter1_value[3];

// Counter 2
assign output_out[4]  = counter2_value[0];
assign output_out[5]  = counter2_value[1];
assign output_out[6]  = counter2_value[2];
assign output_out[7]  = counter2_value[3];
assign output_out[8]  = counter2_value[4];
assign output_out[9]  = counter2_value[5];
assign output_out[10] = counter2_value[6];
assign output_out[11] = counter2_value[7];

// Inverter 1
assign output_out[12] = inv1_dout1;
assign output_out[13] = inv1_dout2;
assign output_out[14] = inv1_dout3;
assign output_out[15] = inv1_dout4;

// SRAM
// Reduce the 32-bit SRAM output to a single bit by computing the parity.
// output_out[16] = 1 when an odd number of bits in sram_0_out are 1.
// output_out[16] = 0 when an even number of bits in sram_0_out are 1.
assign output_out[16] = ^sram_0_out;

// Digital bidirectionals (output side)
// Bidir output enable: drive when `enable` is high (counter visible),
// float as input when low (so external stimuli can drive inverter1).
assign bidir_out[0] = counter1_value[4];
assign bidir_oe[0]  = enable;

assign bidir_out[1] = counter1_value[5];
assign bidir_oe[1]  = enable;

assign bidir_out[2] = counter1_value[6];
assign bidir_oe[2]  = enable;

assign bidir_out[3] = counter1_value[7];
assign bidir_oe[3]  = enable;

// Analog outputs
assign analog_padbare[2] = inv2_vout1;
assign analog_padbare[3] = inv2_vout2;
// ====================================================
```

Listing 9 `flow/librelane/pdn_cfg.tcl (code snippet)`

```
# Custom connect for the inverter macros (inverter1, inverter2).
# The top-level PDN is on Metal4 (vertical) and TopMetal1 (horizontal).
# Each inverter has its own power ring with the horizontal segments on Metal3 and the vertica
# The top-level TopMetal1 stripes therefore cross the macro and one Metal4 to TopMetal1 conne
# A ring that occupied both PDN layers would make pdngen cut every top-level stripe at the ma
# Note: within an inverter's power ring no top-level Metal4 or TopMetal1 power rails are rout
define_pdn_grid \
    -macro \
    -instances "i_chip_core.inverter1 i_chip_core.inverter2" \
    -name inverter_top \
    -starts_with POWER \
    -halo "$::env(PDN_HORIZONTAL_HALO) $::env(PDN_VERTICAL_HALO)"

add_pdn_connect \
    -grid inverter_top \
    -layers "$::env(PDN_VERTICAL_LAYER) $::env(PDN_HORIZONTAL_LAYER)"
```