# Anatomy of a standard cell

Before you draw a cell of your own, it's worth reading one from the library. The SKY130 high-density library ships 446 cells in `$PDK_ROOT/sky130A/libs.ref/sky130_fd_sc_hd/mag/`, and the smallest of them, `sky130_fd_sc_hd__inv_1`, is the same inverter you are about to draw yourself. In Lab 01 you open it and get four different checks to agree about it. This page tells you what to look for while you do.

## The constraints every cell obeys

A standard cell is a transistor-level layout that follows a few constraints, so that a machine can place thousands of them without having to think about how they fit together:

| Constraint | `sky130_fd_sc_hd` |
|---|---|
| Every cell is the same height | 2.72 µm |
| Every cell's width is a multiple of one site | 0.46 µm |
| Power rails sit at the same y in every cell, in metal1 | VGND straddles y = 0, VPWR straddles y = 2.72 |
| Rails span the full width, so abutting cells connect them | yes |
| Pins are on li1, so they can be reached from above | yes |

`inv_1` is 1.38 µm × 2.72 µm = 3.7536 µm², which Lab 01 prints. The `FIXED_BBOX` property in the `.mag` file says `0 0 276 544`, in units of 0.005 µm, so multiplying by 0.005 gives you those two dimensions.

Because the rails straddle the cell boundary, two cells placed side by side each contribute half of a shared rail, and the supply runs continuously along the row without anyone drawing a wire. If you flip a cell vertically and put it above the first, the two share a rail again. Sharing rails this way is the reason chips built from standard cells are laid out in rows.

## What is inside `inv_1`

When you open it in Lab 01, Part 2, you will find:

- **an n-well** across the top half, with p-diffusion inside it, which forms the PMOS
- **n-diffusion** in the bottom half, which forms the NMOS
- **one poly stripe** crossing both, which is the gate of both transistors. Lab 01 reads `w=0.65 l=0.15` for the NMOS and `w=1 l=0.15` for the PMOS back out of the geometry.
- **six ports**: `A`, `Y`, `VPWR`, `VGND`, and two you might not expect, `VPB` and `VNB`, the bodies of the PMOS and the NMOS. The cell exposes its own well and substrate as terminals.

The cell needs those last two ports because it leaves out the taps that would tie its well and substrate to the supplies, as the next section explains.

## Why the cell has no well tap

If you run the full design-rule check on the foundry's own inverter, it fails. Lab 01 has you do exactly that, and prints:

```
ok  DRC violations, default style                0   (reference 0)
ok  DRC violations, full style                   3   (reference 3)
...
THE THREE THINGS THE FULL RULE SET COMPLAINS ABOUT
    All nwells must contain metal-connected N+ taps (nwell.4)
```

```
      P-diff distance to N-tap must be < 15.0um (LU.3)
      N-diff distance to P-tap must be < 15.0um (LU.2)
```

Those are three violations in a cell the foundry ships, from a library that has been taped out hundreds of times. All three come from the same omission, which is that the cell contains no tap. The n-well is not tied to VPWR and the substrate is not tied to VGND, and that is why `VPB` and `VNB` are ports instead of being wired up inside the cell.

The library leaves the taps out on purpose, because taps cost area. A tap in every inverter would mean one every 1.38 µm, far more than any process needs, when the latch-up rule asks only for one within 15 µm. So the library puts the taps in a separate cell, `sky130_fd_sc_hd__tapvpwrvgnd_1`, which is 0.46 µm wide, and the flow drops a few of them into each row. You will place them by hand in the capstone.

The same idea applies to more than taps:

> Whether a cell is DRC clean depends on what is placed around it. Some rules can only be satisfied by a cell's neighbours, and a checker run on one cell by itself will always report them.

It is much cheaper to learn this on the foundry's cell in the first lab than on your own cell in the third, where you would see the same errors and assume you had broken something.

## Drive strength

```
sky130_fd_sc_hd__inv_1   inv_2   inv_4   inv_6   inv_8   inv_12   inv_16
sky130_fd_sc_hd__and2_0  and2_1  and2_2  and2_4
```

The number after the last underscore is the drive strength. Each step up is the same logic function built with wider transistors, which gives it more current, but also more area and more input capacitance. `and2_0` is the weakest AND in the library, and `and2_4` has roughly four times its drive. Choosing between sizes like these is most of what a synthesis tool spends its time on. In the capstone we hand you `and2_0` on purpose, so that you can see what the cheapest choice costs in delay.

## Who drew these cells

Each cell's `.mag` file records where the cell came from. Near the end of `sky130_fd_sc_hd__inv_1.mag`, among its properties, you'll find these three lines:

```
string GDS_END 2272374
string GDS_FILE $PDKPATH/libs.ref/sky130_fd_sc_hd/gds/sky130_fd_sc_hd.gds
string GDS_START 2268880
```

Instead of geometry, those lines hold a byte range in the foundry's GDS file. The `.mag` file was produced by `open_pdks`, which read SkyWater's released GDSII back into Magic so that the cells could be used in an open flow, and the file keeps a pointer to the GDS it came from. When you write GDS from a design that uses these cells, Magic doesn't regenerate their geometry; it copies those bytes straight through. When it writes the capstone's reference block, for example, Magic logs:

```
   Copying output for cell sky130_fd_sc_hd__and2_0 from $PDKPATH/.../sky130_fd_sc_hd.gds
   Generating output for cell datapath_ref
```

So the foundry drew these cells in a commercial tool and shipped them as GDSII, and Magic reads them without having created them.

## Why this course uses Magic

That raises a practical question: is Magic the right tool for you to draw a minimum-size cell in? Magic is a rough editor. It thinks in tiles, in a "corner-stitched" plane model that makes some edits behave in ways that surprise people used to a polygon editor, and packing a cell down to the last 0.005 µm in it is a fight. KLayout is a nicer polygon editor.

This course uses Magic anyway, for reasons that have to do with the flow rather than the editor:

- Magic has a live design-rule checker. Errors appear as white hatching while you draw, with a count in the toolbar. KLayout has no such checker, and seeing your errors as you make them is what teaches you the rules.
- Magic extracts: it turns geometry back into a SPICE netlist, with parasitics, in one command. Every lab from Lab 01 to the capstone relies on that.
- The open SKY130 ecosystem (netgen's setup file, `open_pdks`, LibreLane's sign-off) is wired to Magic's output formats.

The course also never asks you to draw a minimum-size cell, which is where Magic is hardest to use. Your inverter in Lab 03 is deliberately roomy, and Lab 05 measures what the extra room cost, which teaches you more than an afternoon of fighting the editor would.

KLayout is the tool for looking at layouts. It renders GDS well and has a proper layer browser, and you'll open your finished capstone block in it.