AD104 — Layout

Draw a CMOS inverter as a chip layout in Magic and get it through DRC and LVS, then place and route four standard cells by hand.

UofT ASIC Team

Contents

1 AD104 — Layout 5

I Start here

2 Getting started 13

3 What a layout is 17

4 SiliWiz 21

5 Anatomy of a standard cell 23

6 Lab 01: Open a cell you did not draw 27

II Design rules

7 Drawing in Magic 35

8 Reading DRC errors 39

9 The full DRC rule set 43

10 Lab 02: Break it on purpose 45

III Your first cell

11 Stick diagrams 53

12 Extraction 57

13 Layout versus schematic (LVS) 61

14 Lab 03: Draw an inverter 63

IV Building with cells

15 Instances and hierarchy 73

16 Lab 04: A buffer and an AND gate 77

V What layout costs

17 Area, parasitics and delay 85

18 Lab 05: What it costs 89

VI Place and route

19 Routing by hand 95

20 What sets the clock frequency 99

21 Capstone: Place and route 101

Appendices 109

A Magic cheat sheet 111

B DRC error decoder 115

C SKY130 layer map 119

1. AD104 — Layout

AD103 ended with a CMOS inverter you could simulate: two symbols and a wire. In AD104 you draw that inverter as a layout, the coloured rectangles a foundry turns into masks. You check it the way a real tapeout does, with a design rule check (DRC, can it be manufactured?) and a layout-versus-schematic check (LVS, is it the circuit you meant?). Then you measure what your drawing costs compared with the foundry's own cell, and you finish by placing and routing a small block of standard cells by hand.

This course is part of the **UofT ASIC Team** education materials, and it's published at **https://uoftasic.com/ad104/**. The labs run inside the shared **workspace** you set up in IC101.

At a glance

Who it's for	Analog-track students who have finished AD103
Track	Analog
Prerequisites	IC101 → AD101 → AD102 → AD103, in that order
Main tool	**Magic** 8.3.681, the layout editor. We assume you haven't used it before, and Getting started starts from there.
Also	netgen for LVS, KLayout 0.30.11 for looking at GDS, ngspice 47 for timing what the geometry costs you
Math level	Arithmetic and a ruler, because every dimension in this course is a number of microns.
Time	About 17 hours, self-paced

Course path

Work down this table in order. A **Read** row is one or more guide pages: concepts, figures and worked examples, with nothing to run. A **Do** row is one lab, where you run every command yourself. Each lab lists the pages to read before you start it.

#		What	Time
1	Read	Getting started · What a layout is · SiliWiz · Anatomy of a standard cell	1 h
2	Do	Lab 01: Open a cell you did not draw. Measure the foundry's inverter from its layout file, then get DRC, extraction and LVS to agree about it.	1.25 h

#		What	Time
3	Read	Drawing in Magic · Reading DRC errors · The full DRC rule set	1 h
4	Do	Lab 02: Break it on purpose. Fix five inverter layouts, each with one mistake that breaks a real SKY130 rule, until DRC and LVS both pass.	2 h
5	Read	Stick diagrams · Extraction · Layout versus schematic (LVS)	1 h
6	Do	Lab 03: Draw an inverter. Draw a CMOS inverter in Magic from an empty frame until it is DRC clean and LVS clean.	3 h
7	Read	Instances and hierarchy	30 min
8	Do	Lab 04: A buffer and an AND gate. Wire an inverter and a NAND into a buffer and an AND gate without drawing a new transistor.	2 h
9	Read	Area, parasitics and delay	30 min
10	Do	Lab 05: What it costs. Work out what the Lab 03 inverter's drawing costs in area, then measure what it costs in delay.	1 h
11	Read	Routing by hand · What sets the clock frequency	45 min
12	Do	Capstone: Place and route. Place four standard cells, route them by hand, and measure how fast the block can be clocked.	3 h

The times are estimates, and they leave out the optional parts at the end of some labs. Lab 03 and the capstone take longest, because Lab 03 is the first cell you draw yourself and the capstone is the first block you route.

Keep these open while you work: Magic cheat sheet · DRC error decoder · SKY130 layer map

If you get stuck, ask in the team Discord; nobody expects you to work it out alone at 2 a.m. Getting help says what to include in your question.

The workbench

This course is pinned to the workbench image **`hpretl/iic-osic-tools:2026.08`**, which ships **Magic 8.3.681**, **netgen 1.5.323** and **KLayout 0.30.11**. You start it the way IC101 showed you, and Lab 01 has you load this course's environment in it and check the Magic, netgen and ngspice versions.

If the environment setup in Lab 01 fails, you can still run `make` in a lab folder, because every lab pins `PDK=sky130A` itself and ships its own `.magicrc`. That means the labs run to completion in a bare container with no setup at all, and each one ends in `PASS` or `FAIL`, with a reason.

What this course is about

A schematic says *what is connected to what*, and a layout says *what is where*. It's the layout, rather than the schematic, that decides what a chip costs in area, speed, power and yield.

You will draw a transistor as coloured rectangles, find out from the tools that it cannot be manufactured, fix it, prove it is still the circuit you meant, and then measure what your arrangement of rectangles costs compared with the foundry's.

The team lead's summary of layout is that ninety per cent of it is messing things up, and we built the course around that, which is why Lab 02 gives you real broken layouts to fix before you've drawn anything of your own.

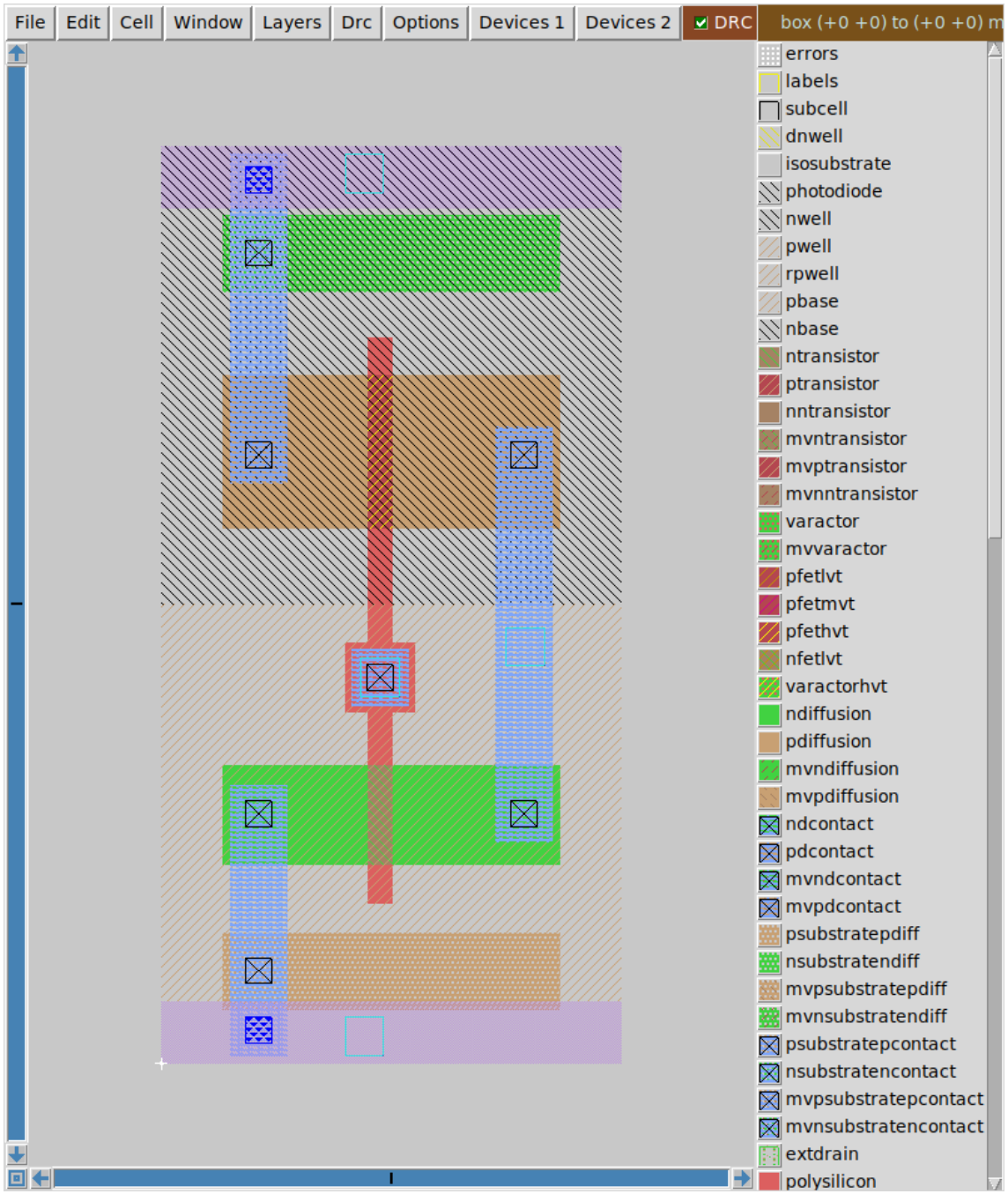


Figure 1.1: The inverter you are going to draw

Your inverter at the end of Lab 03: nwell hatched at the top, PMOS in orange p-diffusion, NMOS in green n-diffusion, and one red poly stripe crossing both. That stripe is the gate, and its width is L. The contacts are the boxed X's, the local interconnect is pale blue, and the two supply rails are purple metal1.

What you'll do

1. Open a cell you did not draw, the foundry's own inverter, and get four checks to agree about it before you draw anything.
2. Fix five broken layouts, each with one mistake that breaks a real SKY130 rule, even though the default rule set calls one of them clean.
3. Draw a CMOS inverter from an empty frame until it is DRC clean and LVS clean.

4. Turn that inverter into a buffer and an AND gate without drawing a single new transistor.
5. Compare your Lab 03 cell against the foundry's, with parasitics extracted. You'll find that the same transistors, drawn differently, take 4.8× the area and run 30 % slower in a ring oscillator.
6. Place and route four standard cells by hand, get the row clean, and measure the highest clock frequency your layout can run at.

What you'll have at the end

You'll finish with a standard cell you drew yourself, a GDSII file of a small block you placed and routed by hand, and a number for how fast that block can be clocked. You'll also have the habit that makes layout manageable: **run the checker, read the rule name, change one thing, and run it again.**

Next courses

- **AD201** — Analog Signal Processing · **AD202** — Mixed Signal
- **DD103** — RTL on FPGAs & ASICs, where a tool does the placing and routing you did by hand, for a whole chip, in about a minute

I Start here

2 **Getting started** 13
2 Prerequisites 13
2 The workbench 13
2 Magic 13
2 What the labs give you 14
2 Getting help 14
2 Problems opening Magic 14

3 **What a layout is** 17
3 How a chip is printed 17
3 The layers, bottom to top 17
3 Where the transistor is 18
3 Diffusion as a wire, and tying the wells 19

4 **SiliWiz** 21
4 The interface 21
4 Where SiliWiz differs from a real process 21
4 Why it is in this course 22

5 **Anatomy of a standard cell** 23
5 The constraints every cell obeys 23
5 What is inside `inv_1` 23
5 Why the cell has no well tap 23
5 Drive strength 24
5 Who drew these cells 24
5 Why this course uses Magic 24

6 **Lab 01: Open a cell you did not draw** . 27
6 Before you start 27
6 Part 1: Load the course 27
6 Part 2: Look at the cell in Magic 28
6 Part 3: Measure the transistors 28
6 Part 4: Run the check 29
6 Part 5: Read the extracted netlist 30
6 Part 6: Run the flow by hand 30
6 Part 7: Browse the library 31
6 Part 8: Explore SiliWiz (optional) 32
6 Finishing 32
6 Answers 32

2. Getting started

Before you start Lab 01 you need three things: the workbench, this course's files inside it, and Magic opening with the SKY130 technology loaded. The third is the one that usually goes wrong, and when it does, the message you get doesn't tell you why. This page explains how the pieces fit together. You won't set anything up here: IC101 sets up the workbench, and Lab 01, Part 1 loads this course into it.

Prerequisites

IC101 → AD101 → AD102 → AD103.

Of the four, AD103 matters most. You should have drawn a CMOS inverter in XSchem and swept its transfer curve, and you should be comfortable with the idea that W and L are numbers you choose. AD104 takes that same inverter and asks where the silicon goes.

The workbench

Everything in this course runs in the noVNC desktop from IC101, which is the image `hpretl/iic-osic-tools:2026.08`. You start it the way IC101 showed you, open `http://localhost` in a browser, and type commands in the terminal on that desktop.

The image can work with more than one manufacturing process, and the environment variable `$PDK` chooses which one. This course needs **`sky130A`**, but the image ships with `PDK=ihp-sg13g2`, a real 130 nm process from a different foundry, with its own layer names and its own rules. Magic opens in IHP's process without complaint, so the first sign that anything is wrong is that none of the layers this course names are there.

The course environment script, `/foss/designs/common/.designinit`, switches the whole shell to SKY130, and it also defines `mod`, the command that copies a course's files into your workspace. The change lasts only as long as that shell, so you load the script once in every new terminal you open.

Magic

Magic is older than most of the people taking this course: it started at Berkeley in 1983, and it looks it. It is also the only open-source layout editor with a live design-rule checker, a built-in extractor and a scripting console, which is why the open SKY130 ecosystem is built around it.

Whenever a lab has you open Magic, you do it with `make edit`, run from inside the lab package. From Lab 02 on, that starts Magic in the folder that holds the `.mag` file and passes it the cell's name without the `.mag` extension; in `labs/lab-03-draw-an-inverter/mag`, for example, it runs `magic -d X11 my_inv`. Lab 01's cell is the foundry's, so there `make edit` hands Magic the full path of the file in the PDK instead.

Two windows appear: **`layout1`**, which shows the drawing, and **`tkcon`**, a Tcl console. Most of your work in this course happens in the console. Every Magic command in the guide pages and labs can be typed there, and the lab scripts drive Magic by feeding it the same commands.

The `-d X11` option

Magic has three graphics drivers in this image, and `X11` is the one that works. The Cairo/OpenGL driver (`-d XR`) can fail with a `BadAlloc` inside the container, and when it does you get a black window rather than no window at all, so it takes a while to work out what went wrong. The team's `.designinit` defines a `magic130` alias that uses `-d X11` for the same reason.

Where Magic gets the technology

When Magic starts, it decides which technology to load from two startup files:

- The image's own `~/.magicrc` loads whichever PDK `$PDK` names. With the image's default `PDK=ihp-sg13g2` you get IHP's layers and none of SKY130's.
- A file called `.magicrc` in **the directory you start Magic in**. Every lab package has one at its top level, and it loads SKY130 only if nothing has loaded it yet; if `~/.magicrc` already did, it does nothing. The check is there because a `.magicrc` that loads SKY130 a second time makes Magic print `Error parsing ".magicrc": can't rename to "closewrapperonly": command already exists` and `Bad local startup file ".magicrc", continuing without.` in the first two lines of the console, which makes a toolchain that is working fine look broken.

Every lab's Makefile also sets `PDK=sky130A` for the Magic it starts, so `make edit` opens in SKY130 even if `.designinit` failed for you.

If no technology file gets loaded at all, Magic falls back on its own default technology, `scmos`, which has about a dozen layers and none of SKY130's. You'll see either `Failed to load technology`, or a Magic that opens normally but has no `nwell` in its layer list.

What the labs give you

Every lab is a folder with a `Makefile`, and typing `make` on its own runs every check and prints the result:

`make`	run every check and print `PASS` or `FAIL` with a reason
`make edit`	open the cell you are working on in Magic
`make clean`	delete `results/`

Because the Makefiles set `PDK=sky130A` themselves, **the labs work even if `.designinit` did not**. If a setup step in Lab 01 doesn't work for you, run `make` in the lab anyway: it checks everything itself and tells you in plain words what it found.

Getting help

Ask in the team Discord. Layout is a craft, and it's quicker to have someone show you than to read about it. A screenshot of your Magic window, with the DRC count in the toolbar visible, is usually enough for someone to spot the problem.

When a tool reports a problem, paste its exact words. `drc why` names a rule and netgen names a net, and those names are specific enough that someone helping you can search for them, which they can't do with "it doesn't work".

Problems opening Magic

Setup problems with `.designinit` and `mod` are covered at the step where they happen, in Lab 01, Part 1.

What you see	What it means
`Failed to load technology "sky130A"`	No SKY130 startup file was loaded. Open the cell with `make edit` from its lab folder, or copy a lab's `.magicrc` into the folder you start Magic in.
Magic opens, and there is no `nwell` in the layer list	Same cause: the technology did not load.
`Bad local startup file ".magicrc", continuing without.`	A `.magicrc` loaded SKY130 a second time. Use a copy of a lab's `.magicrc`, which checks whether SKY130 is already loaded first.
`Cell ... couldn't be read`	You named a cell that does not exist in this folder, or you are in the wrong folder. Magic then creates an empty cell with that name, which is why an empty window is a common symptom.
The layout is one empty rectangle with a name in it	That's an unexpanded subcell. Select it and press `x`, or type `expand`.

What you see	What it means
Everything looks dim and the title says `[NOT BEING EDITED]`	You are looking at a cell that is not the edit cell. Typing `load <name>` makes it the edit cell.

3. What a layout is

A schematic tells you which terminals connect to which, while a layout is a **set of instructions for a machine that prints patterns of light** onto a wafer. This page explains what that means in practice: how a wafer is made, which layers you draw, and where the transistor is in a drawing that never mentions one.

How a chip is printed

Fabricating a wafer is a loop that repeats about seventy times. First the wafer is coated in something, and light is shone through a mask so that the mask's pattern lands on the coating. Then either the parts that were lit or the parts that weren't are washed away, and something is done to whatever is now exposed: ions are implanted into it, an oxide is grown on it, it's etched away, or metal is plated into it. Finally the coating is stripped off, and the loop starts again with the next mask.

A **layout is the drawing on those masks**, so every rectangle you paint in Magic ends up as a hole or a blob on a piece of quartz that the light goes through.

That has two consequences, and both are easier to say than to get used to:

- **There is no wire layer and no transistor layer.** There is a layer where the silicon was doped n-type and a layer where polysilicon was deposited, and wherever the two cross you get a transistor. So the transistor in a layout is just two overlapping rectangles that somebody drew.
- **Everything is a rectangle, and every rectangle has a minimum size.** The minimum comes from the printing itself: the light has a wavelength, the etch spreads, and the wafer isn't perfectly flat. The design rules are the foundry telling you what it can print with acceptable yield.

The layers, bottom to top

SKY130 has about 40 mask layers, and this course uses ten of them. Physically they stack in this order, and the order explains most of the rules:

Magic layer	On the wafer	What it is for
`nwell`	a deep n-type tub in the p-substrate	somewhere to put PMOS
`ndiff` / `pdiff`	heavily doped silicon at the surface	the source and drain of a transistor, and the wire between them
`ptap` / `ntap`	doped silicon of the *same* type as the well underneath	a handle to hold the well at a known voltage
`poly`	polysilicon deposited on a very thin oxide	the **gate**: wherever it crosses diffusion, you get a transistor
`licon1` (`ndc`/`pdc`/`pc` in Magic)	a tungsten plug	the only way up from silicon or poly
`li1` (`li`)	a thin local-interconnect layer	short wiring inside a cell
`mcon`	a plug from li1 to metal1	the only way up from li
`met1` … `met5`	aluminium/copper wiring	wires
`via1` … `via4`	plugs between metal layers	the only way between metals

The pattern is rigid and worth memorising: **conductor, plug, conductor, plug, conductor.** You can't connect metal1 to poly directly; you have to go poly → `licon1` → li1 → `mcon` → metal1, and half of a beginner's DRC errors come from trying to skip one of those steps.

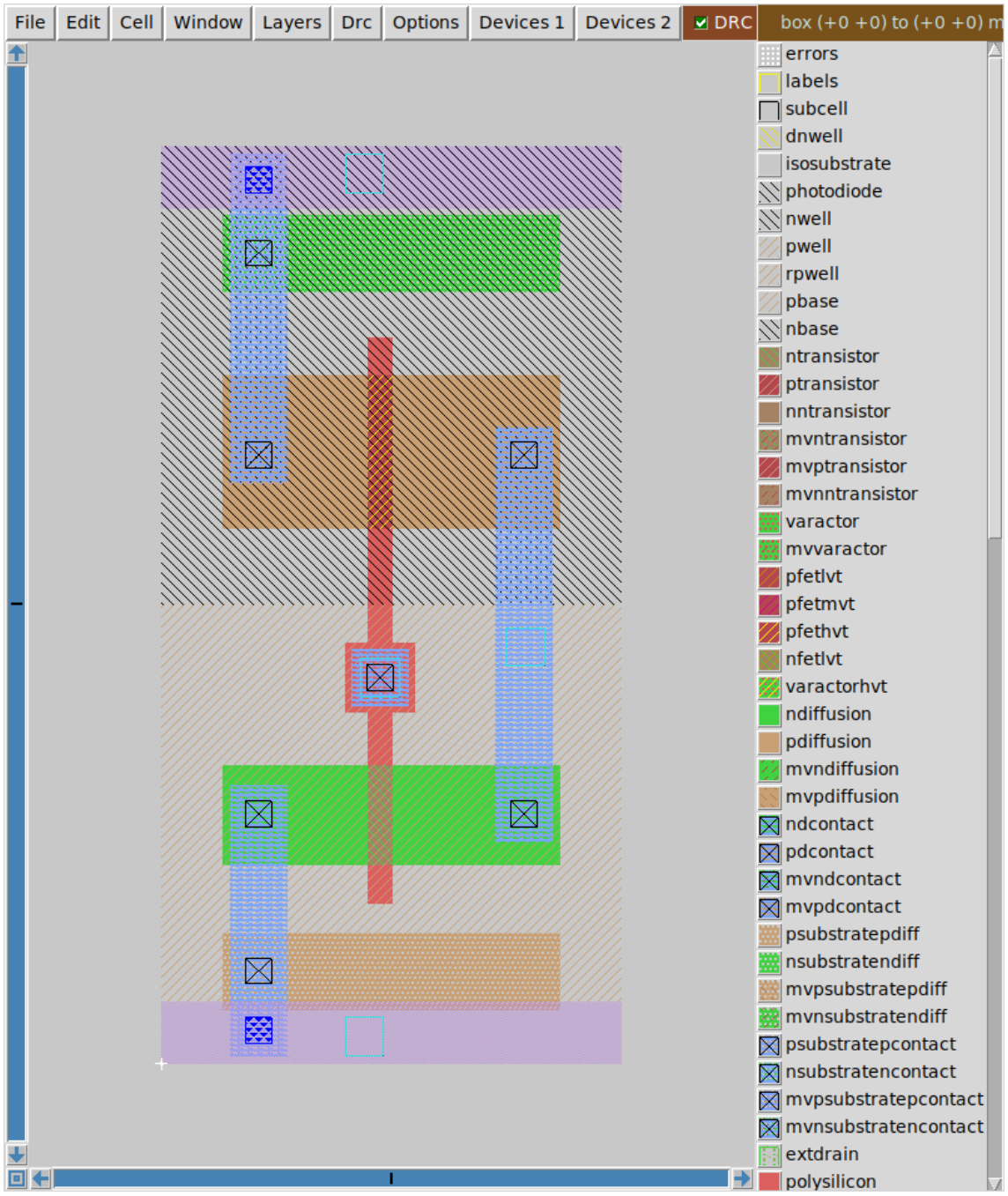


Figure 3.1: The inverter, layer by layer

Read this picture from the bottom up. The purple bands at the very bottom and very top are the metal1 supply rails. The brown strip just above the lower rail is the p-tap, and the green block above that is the NMOS n-diffusion. The red stripe running from below the n-diffusion to above the p-diffusion is a single strip of polysilicon, which forms the gate of both transistors. The pale-blue vertical runs are local interconnect, and the boxed X's are contacts. Every contact is 0.17 µm square, because that's the only size the process makes.

Where the transistor is

You won't find an object called "transistor" anywhere in that picture. What's there instead is:

- a strip of green n-diffusion, 0.65 µm tall,
- a red poly stripe, 0.15 µm wide, crossing it,

- and a contact on either side.

The poly sits on a few nanometres of oxide over the silicon. A voltage on the poly pulls electrons to the surface of the silicon underneath, which joins the two diffusion regions into one conducting channel. That means **the channel is the part of the diffusion that the poly covers**, and it tells you where to read the transistor's size off the drawing:

- ***W* is the height of the diffusion where the poly crosses it.** A wider channel carries more current.
- ***L* is the width of the poly stripe where it crosses.** A longer channel carries less current.

In this course you never type `W=0.65`. Instead, in Lab 03 you draw a rectangle 0.65 µm tall, and Magic reads `w=0.65` back out of it. That round trip, in which you draw a shape and a tool tells you the device parameter, is the point where layout stops being abstract.

Diffusion as a wire, and tying the wells

Diffusion is also a wire. The strip of n-diffusion between two poly gates conducts, which is why two transistors in series can share it and need no contact between them; you'll see this in the NAND gate in Lab 04. It's a poor wire, though, because it's resistive and carries a lot of junction capacitance, so nobody uses it to route any further than the next transistor.

The wells need tying to a supply. The p-substrate under your NMOS and the n-well under your PMOS are the *bodies* of your transistors, and the body is a real terminal that affects how the transistor behaves. If nothing ties the wells to a supply, they float and the threshold voltages wander. In the worst case, the parasitic bipolar structures that are present in every CMOS process latch up and short the supply to ground until you remove power. The `ptap` and `ntap` strips in the picture are there to prevent that. They're also the thing people most often forget in a first layout, and the default design-rule check doesn't mention them; see The full DRC rule set.

4. SiliWiz

SiliWiz runs in a browser tab at **https://app.siliwiz.com/**. You don't need an account, and there's nothing to install.

You paint layout layers in SiliWiz, and it shows you the **cross-section** of the silicon you've described and simulates it. It doesn't model the SKY130 process and it isn't a production tool, but it's the quickest way to turn "poly over diffusion makes a transistor" from a sentence you've read into something you've seen. You can spend twenty minutes with it in Part 8, the optional last part of Lab 01.

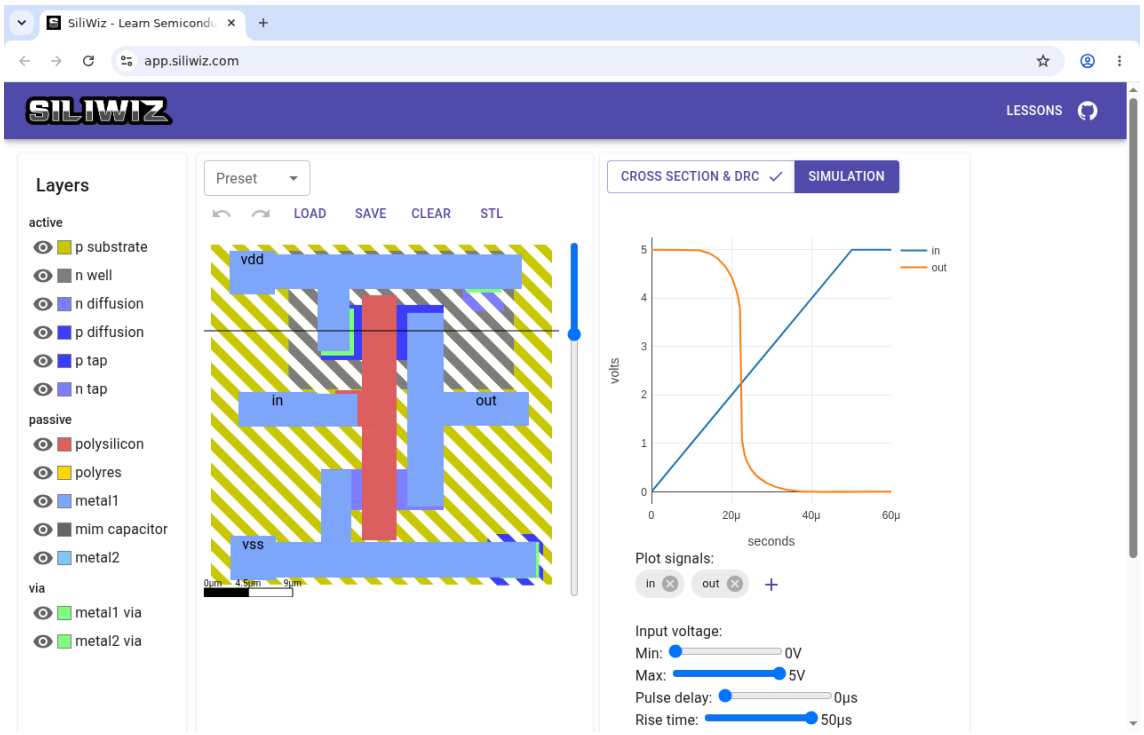


Figure 4.1: SiliWiz with the inverter preset loaded

The layer palette runs down the left, the layout you're painting is in the middle, and the right side shows either the cross-section or, as here, a simulation of what you drew. The black horizontal line across the layout is the cross-section cut, and when you drag the blue slider beside it, the picture on the right changes.

The interface

- **LESSONS**, at the top right, links to short written walkthroughs at https://tinytapeout.com/siliwiz/: *introduction*, *resistors*, *capacitors*, *nmos*, *pmos*, *inverter*, *cmosinverter*, *divider* and *parasitics*.
- The **Preset** dropdown above the drawing area loads a finished example.
- The right-hand panel has two tabs: **CROSS SECTION & DRC** and **SIMULATION**.
- The **vertical blue slider** to the right of the layout moves the cross-section cut line. It is the control that matters most, and the rest of the interface is there to support it.

Where SiliWiz differs from a real process

SiliWiz is a teaching model, and it helps to know where it simplifies things:

- **The process is invented.** The layer names line up with SKY130's (`n well`, `p diffusion`, `polysilicon`, `metal1`, `metal2`, `p tap`, `n tap`), but the dimensions, the design rules and the device models don't, so don't carry any number from SiliWiz over into Magic.
- **It has no LVS.** Nothing in SiliWiz checks that what you drew is the circuit you meant, and that turns out to be the harder half of checking a layout; Layout versus schematic (LVS) covers it.
- **The cross-section is schematic.** It draws each region as a flat-topped block, whereas real doping profiles are gradients.

None of these simplifications gets in the way of what you'll use SiliWiz for, which is looking at the inside of the wafer. Magic never shows you that view, and SiliWiz is the only tool in this course that does.

Why it is in this course

SiliWiz comes from the TinyTapeout project. The team has put the TinyTapeout **shuttle** out of scope for these courses (no submissions, no tiles, no deadlines), but that decision was about the shuttle rather than TinyTapeout's teaching material, and we think this tool is a good one. Nothing in AD104 depends on SiliWiz, so if you skip it, all you lose is the intuition it gives you.

5. 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.

6. Lab 01: Open a cell you did not draw

You'll load this course into the workbench, open the foundry's own inverter, `sky130_fd_sc_hd__inv_1`, in Magic, and measure its two transistors from the layout file. Then you'll run four checks on the same cell (DRC under two rule sets, extraction, and LVS) and get them to agree about it. If they do, you know your toolchain works, so any problem you run into after this lab is a problem in your drawing.

Package: `labs/lab-01-open-the-cell/`

Before you start

- Read Getting started, What a layout is, SiliWiz and Anatomy of a standard cell.
- Finish IC101: you need the `workspace` clone and the noVNC desktop.
- Keep the Magic cheat sheet open in another tab.
- Have paper or a text file ready for the numbers you work out in Part 3.

Time: about 1 hour 15 minutes, plus 20 minutes for the optional SiliWiz part. `make` itself takes about a second and a half.

Part 1: Load the course

If you haven't set up the workbench yet, do IC101 Lab 03 first.

1. Start the noVNC desktop the way IC101 showed you, and open a terminal on it. Type every command in this lab in that terminal, not on your laptop.
2. Load the course environment and check the PDK. The setting lasts only as long as the terminal you load it in, so you'll need to load it again in every new terminal you open.

   ```
   . /foss/designs/common/.designinit
   echo $PDK
   ```

   `echo $PDK` must print `sky130A`. If it prints `ihp-sg13g2`, which is the image's default, the first line didn't run in this terminal.
3. Get the course files with `mod`, a command that `.designinit` defines.

   ```
   mod add ad104     # first time only
   mod ad104         # puts you in the course folder
   ```
4. Check the versions of the three tools whose output this course prints:

   ```
   magic --version
   netgen -batch quit | grep '^Netgen'
   ngspice --version | grep ngspice-
   ```

   ```
   8.3.681
   Netgen 1.5.323 compiled on Tue Aug 25 11:33:45 AM CEST 2026
   ** ngspice-47 : Circuit level simulation program
   ```

   If you have a different version, your outputs may not match the ones printed in these labs.

What you see	What it means
`.designinit: No such file or directory`	The workspace isn't set up, so do IC101 Lab 03. Until then you can still run `make` in any lab, because every lab package pins its own PDK.
`couldn't read Username for 'https://github.com'` on `mod add ad104`	The repo didn't answer an anonymous request. That's a problem on our side rather than anything you did, so please tell us in Discord.

For anything else, see IC101's Troubleshooting page.

You should see `sky130A` from step 2 and the three versions from step 4.

Part 2: Look at the cell in Magic

What is in the foundry's inverter, and what is missing from it?

1. From the course folder, go into the lab package and open the cell:

   ```
   cd labs/lab-01-open-the-cell
   make gui
   ```

   (`make edit` does the same thing, and it's the name every other lab uses.) Magic opens two windows with the cell loaded: `layout1`, which shows the drawing, and `tkcon`, a Tcl console where you type Magic commands. You get the terminal back while Magic stays open, so you can keep using both.

2. Using Anatomy of a standard cell as a map, find the items below. The figure in What a layout is shows which colour is which layer. To check a position or a size, put the cursor box on it (left-click its lower-left corner, right-click its upper-right) and type `box` in the console, which prints the box's width, height and corners in microns.

   - the n-well across the top, and the p-diffusion inside it
   - the n-diffusion in the bottom half
   - the one poly stripe that crosses both diffusions, which is why the two gates are connected without a wire
   - the two metal1 rails, straddling y = 0 and y = 2.72
   - the contacts, each of which is 0.17 µm square

3. Now look for something the cell doesn't have, a well tap. A tap is a contacted strip of `ptap` or `ntap`, which is silicon doped the *same* type as the well under it and tied to a rail. Source and drain contacts don't count as taps, because their diffusion is the opposite type to the well. (What a layout is explains what a tap is for.)

You should see every item in step 2, and no tap anywhere in the cell. Leave Magic open, because you'll use it again in Part 6.

Part 3: Measure the transistors

How wide and how long are the two transistors? You can work this out yourself from the layout file, before any tool extracts it for you.

1. In the terminal, print the file's header and its two transistor layers:

   ```
   F=$PDK_ROOT/sky130A/libs.ref/sky130_fd_sc_hd/mag/sky130_fd_sc_hd__inv_1.mag
   head -4 $F
   grep -A1 -e '<< scnmos >>' -e '<< scpmoshvt >>' $F
   ```

   ```
   magic
   tech sky130A
   magscale 1 2
   timestamp 1785844050
   ```

```
<< scnmos >>
rect 120 47 150 177
<< scpmoshvt >>
rect 120 297 150 497
```

`scnmos` is the NMOS and `scpmoshvt` the high-threshold PMOS. Each `rect` line is one rectangle, given as `x1 y1 x2 y2`, and it covers the patch of diffusion under the poly stripe, which is the transistor's channel (What a layout is).

2. The header says `magscale 1 2`, so one unit in the file is 0.005 µm (Drawing in Magic). For each rectangle, work out its height and its width in µm, and write them down. The height is the transistor's W and the width is its L.

You should see two `rect` lines, and four numbers in your notes: W and L for each transistor. Part 4 checks them against what the tools measure.

Part 4: Run the check

Do the tools measure the cell the way you did?

1. Run the lab:

```
make
```

The `Makefile` sets `PDK=sky130A` itself, so this works even if `.designinit` failed for you. It takes about a second and a half (1.35, 1.46 and 1.60 s on three timed clean runs) and prints:

```
== magic: load, DRC, extract   (sky130_fd_sc_hd__inv_1)
== netgen: layout vs the foundry's own netlist

  THE FOUNDRY'S INVERTER, MEASURED BY YOUR TOOLS

  ok  cell width                              1.38 um   (reference 1.38)
  ok  cell height                             2.72 um   (reference 2.72)
  ok  DRC violations, default style                 0   (reference 0)
  ok  DRC violations, full style                    3   (reference 3)
  ok  devices extracted                             2   (reference 2)

      cell area  1.38 x 2.72 = 3.7536 um^2

  THE TWO TRANSISTORS IT FOUND
      X0  sky130_fd_pr__nfet_01v8          W=0.65  L=0.15   drain area 0.169 um^2
      X1  sky130_fd_pr__pfet_01v8_hvt      W=1  L=0.15   drain area 0.26 um^2

  LVS: Circuits match uniquely

  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)
      ...

PASS  Magic read the foundry's own layout, found zero violations under the
      default rule set and the three expected context rules under the full
      one, pulled two transistors out of the geometry, and netgen agreed
      they are the circuit the foundry says the cell is.
```

If it stops or a number differs:

What you see	What it means
`DRC violations, default style` is not 0	The technology did not load. Check that `$PDK_ROOT` points at the PDKs (`/foss/pdks` in the workbench image) and that `.magicrc` is in the folder you ran `make` from.
`results/report.txt is missing`	Magic didn't finish, so read `results/magic.log`.
`LVS: netgen wrote no report`	Read `results/netgen.log`. The cause is usually `$PDK_ROOT` again.
`magic: command not found`	You're not in the workbench container.

2. Compare the two transistor lines with the W and L you wrote down in Part 3.
3. Look at the count under the full rule set. The foundry's own cell fails the full rule set, and that's the answer we expect. (Magic calls a rule set a DRC *style*, which is why the output says `default style` and `full style`.) Why does the full rule set find three violations in a cell that has been taped out hundreds of times?

   Answer: A1, in Answers at the end.

You should see `PASS`, `0` under the default rule set and `3` under the full one, and `W=0.65 L=0.15` and `W=1 L=0.15` matching the numbers you wrote down.

Part 5: Read the extracted netlist

What circuit did Magic pull out of the geometry?

1. Print the netlist Magic extracted:

```
make netlist
```

```
* NGSPICE file created from sky130_fd_sc_hd__inv_1.ext - technology: sky130A

.subckt sky130_fd_sc_hd__inv_1 A VGND VNB VPB VPWR Y
X0 Y A VGND VNB sky130_fd_pr__nfet_01v8 ad=0.169 pd=1.82 as=0.169 ps=1.82 w=0.65
l=0.15
X1 Y A VPWR VPB sky130_fd_pr__pfet_01v8_hvt ad=0.26 pd=2.52 as=0.26 ps=2.52 w=1 l=0.15
.ends
```

2. Count the ports; there are six. `A` and `Y` are the input and output, and `VPWR` and `VGND` are the supplies. What are `VNB` and `VPB`, and why are they pins?

   Answer: A2, in Answers at the end.

3. Find `w=0.65` and `w=1`. Nobody typed those numbers in; Magic got them by measuring the same two rectangles you measured in Part 3.
4. Find `pfet_01v8_hvt`. It tells you the PMOS is the high-threshold flavour, which you'll need to remember in Lab 03.

You should see two devices, six ports, and `w=0.65 l=0.15` on the NMOS.

Part 6: Run the flow by hand

Which commands does Magic run for `make`, and which of them are easy to get wrong?

Everything Magic did for `make` in Part 4 comes down to a short list of commands that you can type yourself. Every remaining lab in the course uses this same flow, so it's worth running it once with your own fingers.

1. In the `tkcon` console (the Tcl console window that opened with Magic in Part 2), type these one at a time:

```
select top cell
drc euclidean on
drc check
drc catchup
drc count total
drc style drc(full)
drc check
drc catchup
drc count total
drc why
extract do local
extract all
ext2spice lvs
ext2spice -o /tmp/inv.spice
```

2. Watch the count change when you switch rule sets. The first `drc count total` prints `Total DRC errors found: 0`, the second prints `Total DRC errors found: 3`, and `drc why` then names the three rules.
3. The last line wrote the extracted netlist to `/tmp/inv.spice`. In the terminal, compare it with the one `make` wrote:

   ```
   diff /tmp/inv.spice results/sky130_fd_sc_hd__inv_1.lvs.spice
   ```

   `diff` prints nothing when the files are the same.

Three of those lines are easy to leave out or get wrong:

- **`select top cell`**: `drc why` reports only on what's inside the cursor box, and a freshly opened cell has no box worth speaking of. If you skip this line, `drc why` tells you `No errors found.` on a cell that has three violations.
- **`drc catchup`**: `drc check` only queues the re-check, and `drc catchup` waits for it to finish. If you count before it finishes, the check hasn't found anything yet, so there's nothing to count.
- **`drc count total`**, not `drc count`. Plain `drc count` prints two lines, `Cell sky130_fd_sc_hd__inv_1 has 3 error tiles.` and then `Total DRC errors found: 0`. That second line has the same label as the output of `drc count total`, but it counts something else: the *hierarchical* total across subcells, which is always 0 on a flat cell. So if you read the last line of `drc count`, the number never seems to change.

You should see `0`, then `3`, then the same three rule names as in Part 4, and no output from `diff`, which means the flow you typed by hand gave the same netlist as `make`.

Part 7: Browse the library

How does the library offer more drive strength for the same logic?

1. Count the cells in the library and list the inverters:

   ```
   ls $PDK_ROOT/sky130A/libs.ref/sky130_fd_sc_hd/mag/ | wc -l
   ls $PDK_ROOT/sky130A/libs.ref/sky130_fd_sc_hd/mag/ | grep __inv_
   ```

   The first prints `446`. The second lists seven inverters: `inv_1`, `inv_2`, `inv_4`, `inv_6`, `inv_8`, `inv_12` and `inv_16`. They all have the same logic, and the drive strength goes up from each one to the next.
2. Open the whole library GDS in KLayout, which is a much better viewer than Magic. `make klayout` starts it in viewer mode, where it can't edit anything, so you can browse safely:

   ```
   make klayout
   ```

3. Before you look, write down a prediction: is `inv_8` taller than `inv_1`, wider, or both? Then open `inv_8` next to `inv_1` and see where the factor of eight went.

   Answer: A3, in Answers at the end.

You should see 446 cells, seven inverters, and an `inv_8` that is visibly wider than `inv_1`.

Part 8: Explore SiliWiz (optional)

This part runs in your browser and needs no setup. It takes about twenty minutes.

1. Open **https://app.siliwiz.com/**. SiliWiz describes the interface. Load the inverter from the **Preset** dropdown above the drawing area to see a finished example.
2. Open **LESSONS** (top right) and do the *nmos* lesson. Drag the cross-section cut line with the blue slider so that it passes through the middle of the poly stripe, and look at the cross-section: substrate, a thin oxide, poly on top, and doped regions on either side.
3. Now drag the cut so that it misses the poly. The cross-section becomes two doped regions with nothing joining them, because the transistor exists only where the poly crosses.
4. Move a contact until it hangs off the edge of the diffusion it sits on, and open the **CROSS SECTION & DRC** tab. SiliWiz's rules aren't SKY130's, but its complaint, "this thing does not surround that thing by enough", has the same shape as half the errors you'll fix in Lab 02.
5. Do the *parasitics* lesson. Stretch a metal wire longer and watch what happens to the simulated waveform. That's the argument of Lab 05, made in two minutes.

You should see the transistor appear and disappear in the cross-section as the cut moves on and off the poly, and a DRC complaint as soon as the contact leaves its diffusion.

Finishing

You're done when `make` ends in `PASS` and the W and L you worked out from the `.mag` file in Part 3 match the `W=` and `L=` it prints for both transistors.

Questions:

- Why does the full rule set find three violations in a cell that has been taped out hundreds of times? Part 4, step 3 answers it.
- Why does the last line of `drc count` never seem to change when you switch rule sets? Part 6 answers it.

Submit: nothing for this lab.

Next on the Course path: read Drawing in Magic, Reading DRC errors and The full DRC rule set, then do Lab 02: Break it on purpose.

Answers

A1. Part 4: Run the check

All three come from the same omission. The cell has no well tap, because the library puts its taps in a separate cell, and Anatomy of a standard cell explains why it's done that way. `make` prints the same explanation under the three rule names, where the output above shows `....`

A2. Part 5: Read the extracted netlist

They're the bulk terminals, the fourth terminal on each device line. `VNB` is the NMOS body (the p-substrate) and `VPB` the PMOS body (the n-well). The cell exposes them as pins because it doesn't tie them to anything itself, since it has no well tap, as you found in Part 2.

A3. Part 7: Browse the library

It's only wider. Every cell in the library is 2.72 µm tall, and each step up in drive strength is the same logic with wider transistors (Anatomy of a standard cell), so the extra area has to go sideways.

II Design rules

7 Drawing in Magic 35
7 The mouse 35
7 The console 35
7 Layer names 35
7 Contact layers 36
7 Labels and ports 36
7 The `.mag` file 37
7 The commands the labs run 37

8 Reading DRC errors 39
8 What a violation looks like 39
8 The four DRC commands 40
8 “In one direction” 40
8 The four kinds of rule 40
8 Iterating to a clean layout 41
8 Fixing by rule 41

9 The full DRC rule set 43
9 What `full` adds 43
9 Why a missing tap is dangerous 43
9 The rule of thumb 44
9 The same fault in the netlist 44
9 Checkers in general 44

10 Lab 02: Break it on purpose 45
10 Before you start 45
10 Part 1: Run the checker on all five cells 45
10 Part 2: Read one violation in Magic 46
10 Part 3: Fix the first four cells 47
10 Part 4: Fix drc_05 48
10 Part 5: Erase instead of fix 48
10 Finishing 49
10 Going further (optional) 49
10 Answers 49

7. Drawing in Magic

Magic is built around one idea, and once you have it the rest is mostly vocabulary: **there is a box, you move the box, and you tell a layer to fill the box.** Magic calls that painting, and everything else you do, including contacts, labels, ports and zooming, is done either to the box or with it.

This page explains the commands you'll use from Lab 02 onwards, and you'll draw your first cell with them in Lab 03.

The mouse

Left	move the **lower-left** corner of the box to the cursor
Right	move the **upper-right** corner of the box to the cursor
Middle	**paint** the box with the layer under the cursor, or **erase** if the cursor is over empty space

So to draw a rectangle you left-click one corner, right-click the opposite corner, and then middle-click a swatch of the layer you want in the palette down the right side of the window. There is no click-and-drag. The first hour of working this way is annoying, but after that it's faster than dragging would be.

If you have no middle button, or you want exact coordinates, you can type the command instead. In this course you always want exact coordinates, so typing is how you'll draw most of the time.

The console

Magic's second window, `tkcon`, is the console where you type commands, and every command on the rest of this page is typed there.

```
box 0.4um 1.3um 2.6um 1.95um     ;# put the box at these micron coordinates
paint ndiff                      ;# fill it with n-diffusion
erase ndiff                      ;# or take it away again
```

Use the `um` suffix. Without it, Magic reads a number in *lambda*, which is 0.01 µm in this technology, so a box you meant in microns comes out a hundred times too small, and Magic gives no error message about it. Every command in every lab writes `um`, and yours should too.

Two more commands you'll use constantly:

```
box values                      ;# what is the box now? (in internal units, 0.005 um)
findbox zoom                     ;# zoom the view to the box
```

`box values` **reads the box back in internal units of 0.005 µm**, even though a plain number you give to `box` *sets* it in lambda. That inconsistency has cost people whole afternoons. The safe habit is to always set the box with `um`, and to read the numbers `box values` returns as 200 to the micron.

Layer names

The palette on the right shows each layer's display name, but the console takes short aliases instead. These are the ones this course uses:

Console name	Palette	What it is
`nwell`, `pwell`	nwell, pwell	the wells
`ndiff`, `pdiff`	ndiffusion, pdiffusion	source/drain silicon

Console name	Palette	What it is
`ntap`, `ptap`	nsubstratendiff, psubstratepdiff	well and substrate taps
`ndc`, `pdc`	ndcontact, pdcontact	contact from diffusion up to li
`ntapc`, `ptapc`	nsubstratencontact, psubstratepcontact	contact from a tap up to li
`poly`	polysilicon	the gate layer
`pc`	pcontact	contact from poly up to li
`li`	locali	local interconnect
`mcon`	viali	contact from li up to metal1
`m1`, `m2`	metal1, metal2	wiring
`via1`	m2contact	metal1 to metal2
`pfethvt`	pfethvt	makes a PMOS the high-threshold flavour

`tech layers` prints the full list with every alias.

Contact layers

Contacts surprise everyone who is new to Magic, because you don't place a contact as an object. Instead you **paint the contact layer over an area**, and Magic treats that region as the whole stack at once: the diffusion, the cut, and the local interconnect on top. So `paint ndc` over a 0.17 × 0.17 µm box gives you one properly formed n-diffusion contact.

That has a few consequences, and it's worth knowing them before they confuse you:

- For a single cut, the contact region must be **exactly 0.17 µm** in each direction (rule `licon.1`). If you make it bigger, Magic generates an array of cuts instead.
- The plain layers still have to surround the contact. The diffusion has to extend past the contact, and so does the li. The rules for those are `licon.5c` and `li.5`, and `li.5` is one of the five rules you fix in Lab 02.
- Erasing `li` from a contact doesn't remove the contact; Magic demotes it instead.

Labels and ports

As far as LVS is concerned, a cell has no terminals until you name them, and there are two commands that give things names:

```
box 1.30um 2.40um 1.55um 2.65um
label A c li          ;# put the label "A", centred, on the li layer here

select top cell
port makeall          ;# promote every label in the cell to a port
```

`label` names a piece of geometry, and `port makeall` then declares those names to be the cell's terminals. That second step is what makes the extracted netlist come out as `.subckt my_inv A Y VPWR VGND` rather than as a subcircuit with no pins, so every lab step that adds labels ends with `port makeall`. If netgen ever says a cell's pin list is empty, a missing `port makeall` is the first thing to check.

A label always goes into the cell you're editing, on that cell's own geometry under the box. If the only shape under the box belongs to a subcell, the label still lands in your cell, but on empty space: Magic prints `Moving label "A" from locali to space`, and a label on space isn't attached to any net, so it never becomes a pin on the extracted `.subckt` line. That's why the blocks in Lab 04 and the capstone paint their own small patch of `li` or `m1` before labelling it.

The `.mag` file

```
save my_inv           ;# writes my_inv.mag in the current directory
```

A `.mag` file is plain text, made up of one section per layer, each listing that layer's rectangles:

```
<< ndiff >>
rect 80 260 520 390
```

Check the header before you trust those numbers. The unit a `.mag` file uses is set by its `magscale` line, and the two kinds of cell in this course use different units:

Cell	Header	One unit is	`rect 80 260 520 390` means
A foundry cell from the PDK	`magscale 1 2`	0.005 µm	0.40 ⟶ 2.60 µm
Anything this course draws	*no `magscale` line*	0.01 µm	0.80 ⟶ 5.20 µm

So the rectangle above spans 0.40 ⟶ 2.60 µm horizontally if it's **in a foundry cell**. In your own `mag/my_inv.mag`, the same 0.40 ⟶ 2.60 µm diffusion is stored as tiles spanning `40 130` to `260 195`. Magic splits a painted shape into several `rect` lines, so read the extent of the whole shape rather than any one line. If you apply the foundry cell's unit to a cell you drew, you'll read every dimension as half its real size, and nothing will warn you; it's the same kind of factor-of-two trap as `box values`. Magic does tell you which kind of cell you have every time it loads one (`Scaled magic input cell my_inv geometry by factor of 2`), and `box values` always reports in 0.005 µm units, whichever kind of file you're in.

When a lab says that a shipped cell has a particular dimension, the number comes from the cell's `.mag` file, so you can check it yourself with `grep`. In Lab 03, Part 9 you open your own `.mag` file in a text editor.

The commands the labs run

Later guide pages cover each of these commands, and together they make up the whole flow:

```
drc check                 ;# find rule violations
drc why                   ;# name the rule broken under the box
drc style drc(full)       ;# switch to the rule set that catches taps

extract do local          ;# geometry -> circuit
extract all
ext2spice lvs             ;# write it in a form netgen understands
ext2spice -o my_inv.spice

ext2spice cthresh 0       ;# ...and keep every parasitic capacitance
gds write my_inv.gds      ;# the file a foundry would take
```

The Magic cheat sheet lists these with their keyboard equivalents, plus about a dozen more.

8. Reading DRC errors

> **Ninety per cent of layout is messing stuff up.** — the team lead, and the reason this course is shaped the way it is

A design rule check (DRC) reports each problem in your layout as one sentence, and each sentence says something specific about what the foundry's equipment can and can't make. Every rule exists for a physical reason and every violation names the rule it broke, so the rule name tells you what to fix.

The skill this course teaches is a loop: **read the rule name, understand the constraint, change one thing, and run the check again.** In Lab 02 one pass of that loop takes about four seconds, and you'll spend a lot of your layout time going round it.

What a violation looks like

Magic runs the check continuously as you draw. It covers each violation with white hatching and keeps a count in the toolbar, and to find out what a particular violation is, you put the box over it and ask:

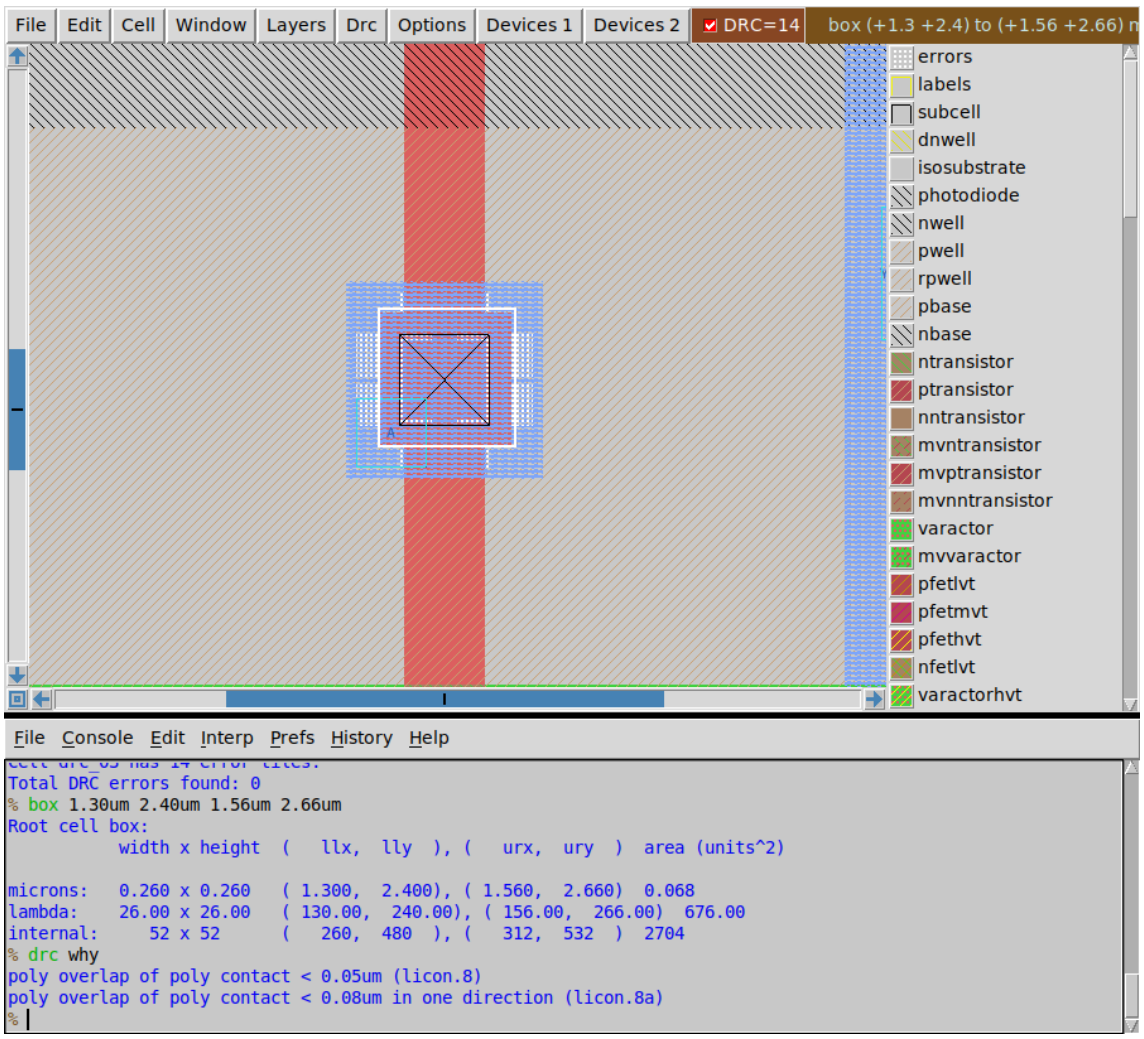


Figure 8.1: A real violation, and the rule it broke

The toolbar says `DRC=14`, and the white hatching sits around the poly contact. With the box over the hatching, `drc why` names the rule twice: once as the plain enclosure rule and once as the directional version of it.

```
% drc why
poly overlap of poly contact < 0.05um (licon.8)
poly overlap of poly contact < 0.08um in one direction (licon.8a)
```

That's cell `drc_03` from Lab 02. Its poly landing pad has been shrunk to 0.25 × 0.25 µm around a 0.17 µm cut, which leaves 0.04 µm of poly on each side, and the rule asks for 0.05.

The four DRC commands

```
drc check          ;# re-run the checker over the whole cell
drc catchup        ;# wait for it to finish (batch scripts need this)
drc count total    ;# how many violations
drc why            ;# name the rules broken under the current box
```

You'll use `drc why` the most, because `drc count total` only tells you how many violations there are, while `drc why` tells you what they are. When you do want the count, use `drc count total` rather than plain `drc count`. The last line of plain `drc count` is a hierarchical total, and on a cell with no subcells it reads 0.

"In one direction"

You'll see this phrasing often:

```
Local interconnect overlap of poly contact < 0.08um in one direction (li.5)
Metal1 overlap of local interconnect contact < 0.06um in one direction (met1.5)
N-diffusion overlap of N-diffusion contact < 0.06um in one direction (licon.5c)
```

It's easy to misread this as "0.08 µm somewhere". What the phrase actually means is: **pick an axis, x or y, and on that axis the surrounding layer must extend past the contact by at least that much on *both* sides.**

So a contact centred in a shape that's generous vertically but tight horizontally passes, because the y axis satisfies the rule on its own. A contact tucked into a corner fails, because neither axis has clearance on both sides.

That's why the li (local interconnect) strips in Lab 03's reference inverter are 0.37 µm wide around a 0.17 µm contact. The extra width gives 0.10 µm of overhang on the left and on the right, so the x axis satisfies the rule by itself.

It's also why the fix for one of these violations is rarely to make everything bigger. Usually you only need to make the shape bigger *on one axis*, and that often costs no area at all.

The four kinds of rule

Almost every SKY130 rule you'll meet is one of four kinds:

Kind	Reads like	Physically
width	`Local interconnect width < 0.17um (li.1)`	the printer can't resolve a line thinner than this
spacing	`Metal1 spacing < 0.14um (met1.2)`	two features closer than this merge, or short together
enclosure / overlap	`N-well overlap of P-Diffusion < 0.18um (diff/tap.8)`	the two masks don't align perfectly, so you leave a margin
area	`Local interconnect minimum area < 0.0561um^2 (li.6)`	a shape smaller than this lifts off or doesn't fill

Once you can sort a message into one of those four kinds, you know which kind of edit fixes it: make the shape wider, move it away, make the shape *around* it bigger, or make it bigger overall.

Iterating to a clean layout

The reference inverter in Lab 03 wasn't drawn correctly the first time. Its first version had 62 violations across seven distinct rules: contacts too close to the edges of their diffusion, li exactly the size of the contacts under it, a poly gate stripe narrower than the contact meant to land on it, and taps with no overhang. The second version had zero.

It took two iterations, and that wasn't because the first attempt was careless. Nobody can hold thirty enclosure rules in their head, and we don't expect you to. You draw what's obviously right, the checker names the places where you were wrong, and each rule name tells you which number to change.

That's why Lab 02 comes before Lab 03: you meet five real violations on somebody else's layout, where being wrong costs you nothing, before you meet them on your own.

Fixing by rule

The worst habit in layout is nudging shapes a little at a time until the white hatching goes away. It does work eventually, but it takes ten times as long and teaches you nothing, and the layout it produces is usually right only by accident, so a single edit can make it wrong again.

Instead, take the rule name and number from `drc why`, look the number up, change the geometry so that it satisfies the rule with a comfortable margin, and run the check again.

The DRC error decoder lists every rule this course can make you hit, with its number and the fix.

9. The full DRC rule set

Magic's SKY130 technology file defines its design rules in several **variants**, and it lists them on this line:

```
style drc variants (fast),(full),(routing)
```

The first variant listed is the default, so unless you choose another one, Magic checks your layout with **`fast`**, which leaves out a whole category of rules.

What `full` adds

The `full` variant adds four rules that matter to you, and none of them is in the default set:

```
variants (full)
 cifmaxwidth nwell_missing_tap 0 bend_illegal
        "All nwells must contain metal-connected N+ taps (nwell.4)"
 cifmaxwidth tap_missing_licon 0 bend_illegal
        "All taps must be contacted (licon.16)"
 cifmaxwidth ptap_missing 0 bend_illegal
        "N-diff distance to P-tap must be < 15.0um (LU.2)"
 cifmaxwidth ntap_missing 0 bend_illegal
        "P-diff distance to N-tap must be < 15.0um (LU.3)"
```

All four are about **well taps**, the connections that hold the body of your transistors at a known voltage. They're left out of the default set because they are expensive to compute, and because on a single cell they usually flag a problem that isn't really there: standard cells leave their taps out on purpose. The foundry's own inverter fails three of them (see Anatomy of a standard cell).

The fourth, `licon.16`, never fires on the pinned image, not even on a tap with no contact. The layer it checks, `tap_missing_licon`, is every tap minus a layer called `tap_with_licon`, and the tech file builds `tap_with_licon` by growing outwards from every tap, contacted or not, so the subtraction always leaves nothing. A tap with no contact still gets caught, but by the rules it leaves unsatisfied: Lab 02's `drc_05` has an n-tap with no contact, and the full rule set reports `nwell.4` and `LU.3` for it.

Why a missing tap is dangerous

A missing tap is hard to catch because nothing you normally look at shows it:

- The layout looks complete.
- `drc check`, with the default rule set, reports **0**.
- The cell extracts.
- It might even simulate, depending on what the floating body does.

Here is the block you build in the capstone, a row of four standard cells called `datapath`, checked both ways before its tap cells go in:

```
      DRC, default rule set  0
      DRC, full rule set     32
```

The default check reported none of the thirty-two violations in this block. That matters because on real silicon, missing taps cause **latch-up**. Every CMOS process contains parasitic bipolar transistors between the well, the substrate and your diffusions, and the taps are what hold the well and substrate firmly at their supplies. If nothing holds them there, a transient can turn those transistors on, and once they're on they form a self-sustaining short from supply to ground. The chip stops working until you cut the power, and sometimes it still doesn't work after that.

The rule of thumb

> **Before you believe a DRC count of zero, run it again with `drc style drc(full)`.**

Every lab in this course that runs DRC runs both rule sets and prints both counts side by side, so inside a lab you never have to remember to do this. When you work in Magic by hand, you have to switch to the full set yourself, with this sequence:

```
drc style drc(full)
drc check
drc catchup
drc count total
```

To switch back to the default rule set, run `drc style drc(fast)`.

The same fault in the netlist

A missing tap also shows up outside DRC. Two independent tools notice it, and neither of them uses the word “tap”. If you extract a block with no taps and look at the netlist, you see this:

```
.subckt datapath CLK D A B Y VGND VPWR
Xbuf0 buf0/A VGND VSUBS dff0/VPB VPWR Y sky130_fd_sc_hd__buf_1
```

The bulk terminals came out as **`VSUBS`** and **`dff0/VPB`**. Those are invented node names, which appear because nothing connects those wells to a supply. Netgen then reports a mismatch, because the schematic you compare against has the same terminals on VGND and VPWR.

With the taps in place, the same line becomes:

```
Xbuf0 buf0/A VGND VGND VPWR VPWR Y sky130_fd_sc_hd__buf_1
```

The invented names and the Netgen mismatch are two symptoms of the same missing taps, and neither of them is a DRC error in the rule set you were running. Lab 02′s fifth cell has this fault at the scale of a single inverter, so you meet it first in a design small enough to see all of it.

Checkers in general

What this page shows about taps holds for any checker:

> **When a checker passes, it has told you that none of the rules it ran fired, which is not the same as telling you the design is right.**

The digital track meets the same problem from a different direction. In DD103 an entire chip hardens successfully in the *wrong PDK*, behind three green ticks and a real `.gds`, because nothing in the flow checks that you meant that PDK. In both tracks the habit that protects you is the same: know which checks you’re running, and ask what they aren’t looking at.

10. Lab 02: Break it on purpose

In this lab you'll fix five copies of the same inverter. We put one mistake in each copy, and each mistake breaks a real SKY130 design rule (two of them break two rules at once, as Part 1 shows). We've put this lab before you draw anything of your own on purpose, because the best time to read your first DRC errors is on a layout you have no attachment to.

Package: `labs/lab-02-break-it-on-purpose/`

Before you start

- Finish Lab 01, so the course is loaded and your tools check out.
- Read Drawing in Magic, Reading DRC errors and The full DRC rule set.
- Keep the DRC error decoder open in another tab.

Time: about 2 hours. Each `make` takes about 4 seconds (we timed two clean runs at 3.97 s and 4.40 s).

Part 1: Run the checker on all five cells

What does each cell break, as the checker sees it?

1. Go into the lab package and run it:

```
cd labs/lab-02-break-it-on-purpose
make
```

   As shipped, it prints:

```
  FIVE CELLS, FIVE MISTAKES

  cell      DRC default  DRC full   LVS
  ------------------------------   ------------------------
     drc_01           3         3   match
     drc_02           2         2   match
     drc_03          14        14   match
     drc_04           2         2   match
     drc_05           0         1   Netlists do not match

  drc_01 still breaks:
      N-well overlap of P-Diffusion < 0.18um (diff/tap.8)
  drc_02 still breaks:
      Local interconnect overlap of poly contact < 0.08um in one direction (li.5)
  drc_03 still breaks:
      poly overlap of poly contact < 0.08um in one direction (licon.8a)
      poly overlap of poly contact < 0.05um (licon.8)
  drc_04 still breaks:
      Local interconnect spacing < 0.17um (li.3)
  drc_05 still breaks:
      All nwells must contain metal-connected N+ taps (nwell.4)
      P-diff distance to N-tap must be < 15.0um (LU.3)

FAIL  Still to fix: drc_01, drc_02, drc_03, drc_04, drc_05
```

2. Before you touch anything, look at the `drc_03` row. It reports 14 violations, even though we changed only one number in that cell. How can one mistake produce fourteen?

Answer: A1, in Answers at the end.

3. Now look at the `drc_05` row. It has zero violations under the default rule set, yet netgen says the netlists do not match, so the cell passes the default check and is still the wrong circuit. You'll come back to it in Part 4.

You should see `FAIL` and all five cells listed under "Still to fix".

Part 2: Read one violation in Magic

How do you get Magic to show you a violation and name its rule? Work through `drc_03` once with these steps; Parts 3 and 4 repeat them for the other cells.

1. Print every violation in cell 3 with its rule name:

```
make why C=3
```

```
drc_03 FAST 14
drc_03 FULL 14
drc_03 WHY poly overlap of poly contact < 0.08um in one direction (licon.8a)
drc_03 WHY poly overlap of poly contact < 0.05um (licon.8)
```

2. Open the cell in Magic:

```
make edit C=3
```

Magic opens with a green ☑ `DRC` in the toolbar and no white hatching anywhere, even though this cell has fourteen violations. Why does it look clean?

Answer: A2, in Answers at the end.

3. To make Magic check the whole cell, type this in the `tkcon` console:

```
drc check
```

(`Drc → DRC update` on the menu bar does the same with the mouse.) The toolbar turns red and shows a count, and white hatching appears where the violations are.

4. Put the cursor box over one of the white hatched areas (left-click one corner, right-click the other) and type:

```
drc why
```

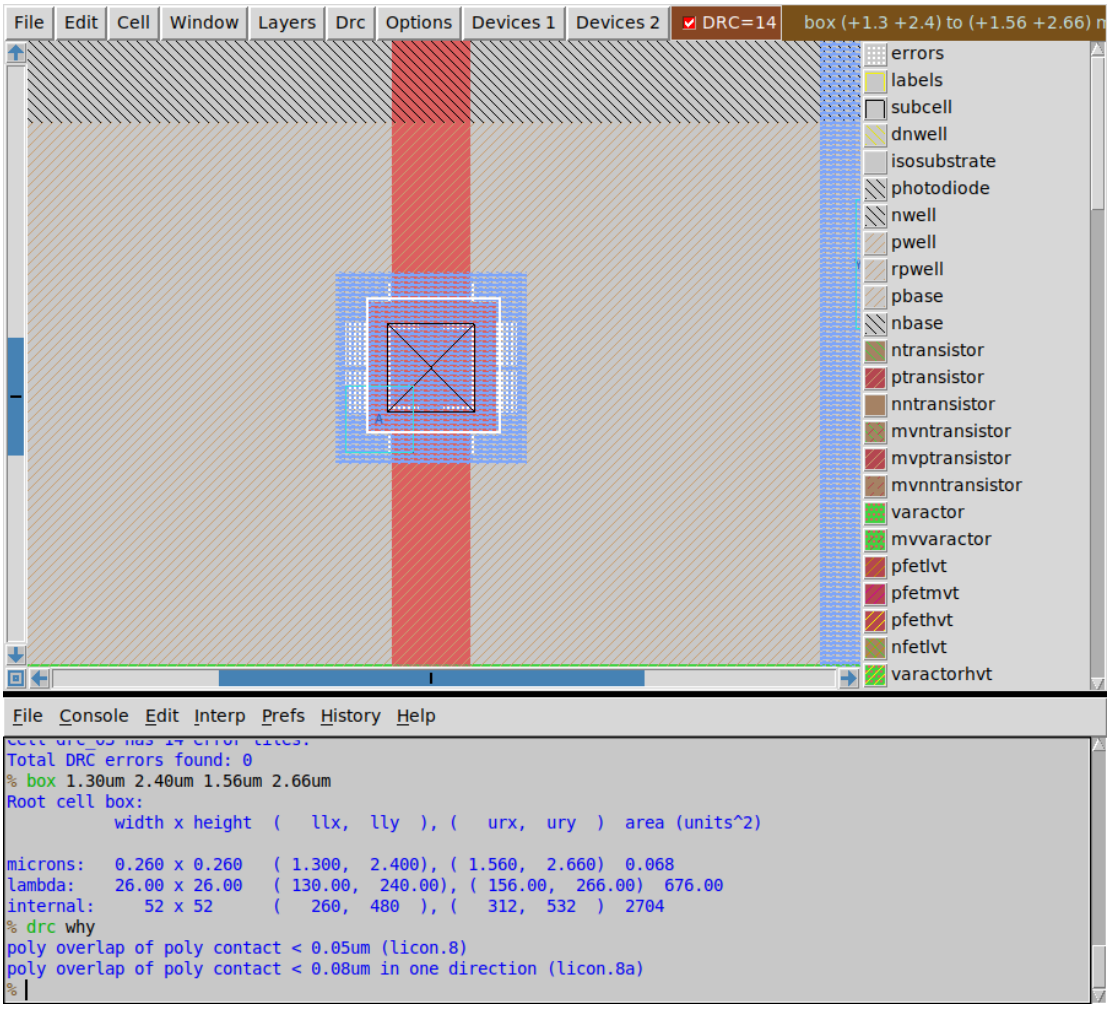


Figure 10.1: What that looks like

Cell `drc_03`, with `DRC=14` in the toolbar and white hatching around the poly contact. `drc why` names the rule twice, once in its plain form and once in its "in one direction" form.

5. Look up `licon.8` in the DRC error decoder and work out what it protects.

You should see `DRC=14` in the toolbar after `drc check`, and `drc why` naming `licon.8` and `licon.8a`.

Part 3: Fix the first four cells

What does each rule protect, and what change satisfies it? For each of `C=1` to `C=4`:

1. Run `make why C=<n>` to get the rule names.
2. Run `make edit C=<n>`, then `drc check` in the console.
3. Put the box over a hatched area and run `drc why`.
4. Look the rule up in the DRC error decoder, work out what it protects, and change the geometry to satisfy it with a comfortable margin. Work from what the rule asks for, rather than nudging shapes until the hatching disappears.
5. Type `save` in the console, then run `make` in the terminal again.

The rules that two of the five cells break use the phrase "in one direction". As Reading DRC errors explains with examples, it means that you pick an axis, x or y, and on that axis the outer layer must extend past the inner one by at least the rule's amount **on both sides**. Only one axis needs to be generous, which is why the fix often costs you nothing.

If you get stuck, open the hint for that cell only, and only after you have tried.

Hint for drc_01: A3, in Answers at the end.

Hint for drc_02: A4, in Answers at the end.
Hint for drc_03: A5, in Answers at the end.
Hint for drc_04: A6, in Answers at the end.
`solutions/README.md` in the package explains how to get unstuck without reading the answer.
You should see `drc_01` to `drc_04` at `0` in both DRC columns, still `match` in LVS.

Part 4: Fix drc_05

The default checker has reported `drc_05` clean since your first run, yet the cell is wrong. Which tools notice, and what are they telling you? Look at how each one reports the problem before you fix it.

1. Two independent tools notice the problem. The first is the full rule set, which names these two rules:

   ```
   All nwells must contain metal-connected N+ taps (nwell.4)
   P-diff distance to N-tap must be < 15.0um (LU.3)
   ```

   Both are context rules, which a cell can satisfy only with help from what is placed around it (Anatomy of a standard cell explains why).

2. The second is netgen, which reports a mismatch without ever using the word "tap". Open `results/drc_05.lvs.spice` and find the PMOS line:

   ```
   X1 Y A VPWR w_0_600# sky130_fd_pr__pfet_01v8_hvt w=1 l=0.15
   ```

   What is `w_0_600#`, and why is the PMOS body on it?

   Answer: A7, in Answers at the end.

3. Fix it with the same loop as Part 3 (`make why C=5`, `make edit C=5`, `drc check`, `drc why`), then `save` and run `make`. This time, type `drc style drc(full)` in Magic before `drc check`. Under the default rule set this cell has no violations, so `drc why` would have nothing to name.

4. If you had only run the default rule set, what would have told you about this cell?

   Answer: A8, in Answers at the end.

The habit to take from this part is the rule of thumb from The full DRC rule set: **before you believe a DRC count of zero, run `drc style drc(full)`.** The labs that run DRC show you both counts, but tools outside this course won't do that for you.
You should see `drc_05` at `0` under both rule sets and `match` in LVS, and `make` ending in `PASS`.

Part 5: Erase instead of fix

What happens if you make a violation go away by deleting the shape that broke the rule? You could take every one of these cells to zero violations that way, so try it once, on `drc_04`.

1. Run `make edit C=4`, erase the `A` pad, and `save`. Before you run `make`, predict what it will show in the DRC and LVS columns for `drc_04`.

2. Run `make`. The li spacing error is gone, and `make` says:

   ```
   Note: drc_04 is clean and no longer an inverter.
   Erasing the shape that broke the rule is not a fix. Move it, resize it,
   or add the thing it needed - the circuit has to survive.
   ```

   Why does this lab run LVS on all five cells, and not only on `drc_05`?

   Answer: A9, in Answers at the end.

3. Put the pad back the way you had fixed it (`u` in the layout window undoes the erase, or repaint it), `save`, and run `make` again.

You should see the note in step 2, and `drc_04` back to `match` after step 3.

Finishing

You're done when `make` ends in `PASS`, with all five of your fixed cells at 0 violations under both rule sets and `match` in LVS.

Questions:

- What does "in one direction" mean in a rule name? Part 3 explains it.
- Which cell would you have shipped broken if you had only run the default rule set, and what would have warned you? Part 4, step 4 answers it.

Submit: nothing for this lab.

Going further (optional)

To start again or check your answers:

```
make reset        # throw your fixes away and rebuild the five broken cells
make solutions    # write the answer key into solutions/
```

`tcl/make_broken.tcl` generates the broken cells and has one `BREAK` comment for each change, so it doubles as the answer key. Read it after you finish, not before.

Next on the Course path: read Stick diagrams, Extraction and Layout versus schematic (LVS), then do Lab 03: Draw an inverter.

Answers

A1. Part 1: Run the checker on all five cells

A single contact that is 0.01 µm short of its enclosure produces fourteen error tiles, because Magic reports each edge of the offending region separately. So the violation count tells you how many tiles Magic flagged rather than how many mistakes there are, and fourteen errors can mean a single number to change.

A2. Part 2: Read one violation in Magic

It looks clean because Magic hasn't checked it yet. Its background checker only re-checks tiles you have *edited*, and loading a file from disk doesn't edit anything, so nothing in the cell has been checked since you opened it.

A3. Part 3: Fix the first four cells (Hint for drc_01)

`diff/tap.8` is an enclosure rule. An implant mask has to extend past what it is implanting, because two masks never align perfectly.

A4. Part 3: Fix the first four cells (Hint for drc_02)

`li.5` is an enclosure rule: a wire has to overhang the contact under it. Paint the wire first, bigger than the contact, and then put the contact inside it.

A5. Part 3: Fix the first four cells (Hint for drc_03)

`licon.8` and `licon.8a` are enclosure rules. A 0.15 µm poly stripe is too narrow to hold a 0.17 µm contact, which is why gates get a landing pad.

A6. Part 3: Fix the first four cells (Hint for drc_04)

`li.3` is a spacing rule, so this violation means that two wires on different nets are too close together.

A7. Part 4: Fix drc_05

`w_0_600#` is the PMOS body. It's an invented node name, and the body is on it instead of on VPWR because the n-well tap has no contact on it, so nothing ties the well to VPWR.

A8. Part 4: Fix drc_05

Netgen would have. LVS reported `Netlists do not match` on a cell that the default rule set called clean.

A9. Part 5: Erase instead of fix

Because a change that satisfies DRC can break the circuit, as erasing the pad just did. A layout has two jobs, passing the rules and being the right circuit, and a change that does one at the expense of the other hasn't moved you forward.

III Your first cell

11 Stick diagrams 53
11 The convention 53
11 The inverter 53
11 The NAND 54
11 Choosing the order 55

12 Extraction 57
12 The commands 57
12 Reading the netlist 57
12 Node names you did not choose 58
12 PMOS flavours 58
12 Two kinds of extraction 58
12 What extraction does not know 59

13 Layout versus schematic (LVS) 61
13 netgen 61
13 A match 61
13 Two failures to recognise 61
13 Where the schematic netlist comes from 62
13 Why the labs run LVS 62

14 Lab 03: Draw an inverter 63
14 Before you start 63
14 Part 1: Sketch it on paper 63
14 Part 2: Start from the frame 63
14 Part 3: Draw the two diffusions 65
14 Part 4: Draw the gate 65
14 Part 5: Add the contacts 66
14 Part 6: Wire it in li 66
14 Part 7: Name the pins 67
14 Part 8: Fix the PMOS flavour 67
14 Part 9: Read what you drew 68
14 Finishing 69
14 Going further (optional) 69
14 Answers 69

11. Stick diagrams

A stick diagram is a layout with all the dimensions thrown away: coloured lines for the layers, crosses for the contacts, and no measurements. What it does record is **what order things go in**, which is the one thing that's expensive to get wrong, and you can work that out in ninety seconds on paper instead of an hour in Magic. You'll sketch one at the start of Lab 03, before you draw anything.

The convention

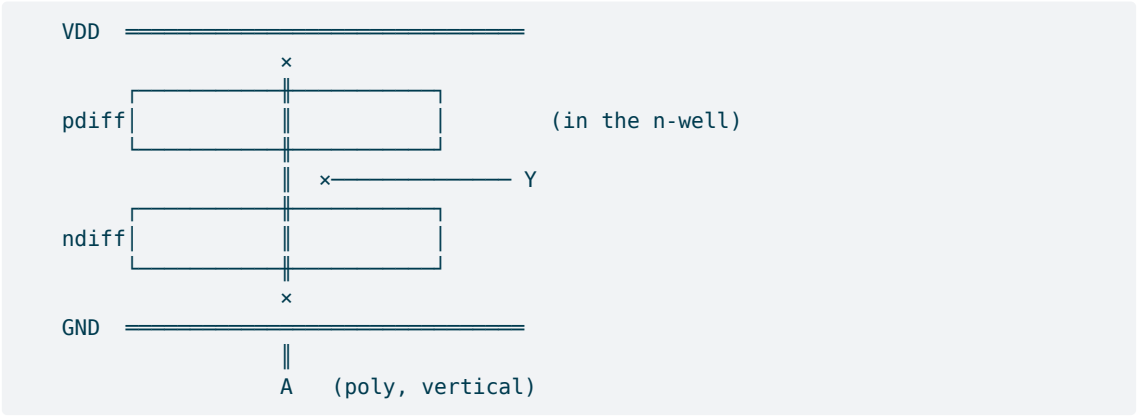
The convention uses four colours and one symbol, and each has a single rule:

Green	n-diffusion, drawn horizontally at the level of the NMOS transistors.
Yellow / brown	p-diffusion, also drawn horizontally, but higher up, in the n-well.
Red	polysilicon, drawn **vertically**. Every red line that crosses a diffusion makes one transistor.
Blue	metal (or li), which can run in any direction, because it sits above everything else.
×	a contact, which connects two of the layers listed above it.

Two more conventions make every cell in a library look alike:

- **Power at the top, ground at the bottom**, in horizontal metal across the full width of the cell.
- **PMOS above, NMOS below.** The PMOS transistors sit in the n-well, and the n-well is tied to the positive rail at the top, so this order is more than a matter of taste.

The inverter



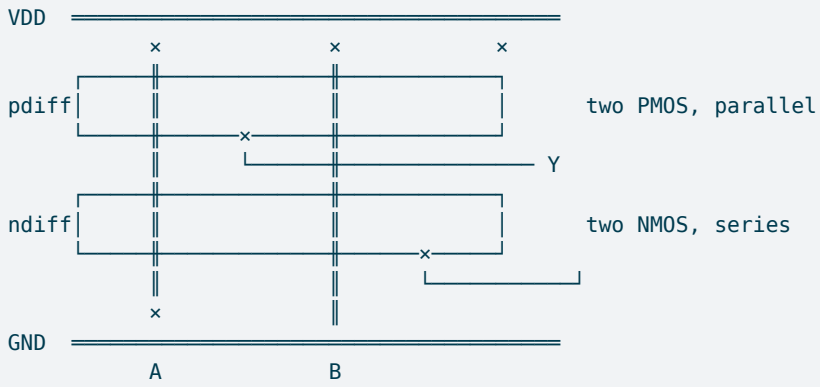
One red stripe crosses the p-diffusion once and the n-diffusion once, so it makes two transistors, and because it is one stripe, their gates are already connected. That shared gate is why a CMOS inverter is compact: you get the gate connection for free, by drawing one rectangle instead of two rectangles and a wire.

In your AD103 schematic you needed an explicit wire from one gate to the other, so for the inverter the layout is actually *simpler* than the schematic. That won't be true of any cell after this one.

The NAND

A 2-input NAND is made of two NMOS transistors **in series** and two PMOS transistors **in parallel**.

When two transistors are in series, the drain of one is the source of the other. On a schematic that connection is a wire, but on silicon it means **the two terminals are the same piece of diffusion**. You draw one strip of diffusion and put two poly stripes across it, and the region between the stripes *is* the shared node, so it needs no contact, no wire and no extra space.



If you count the contacts on the n-diffusion, there are **two**, one at each end. The middle of that strip carries current but has no contact, because a contact there would cost area and add junction capacitance to an internal node, and it would gain you nothing in return.

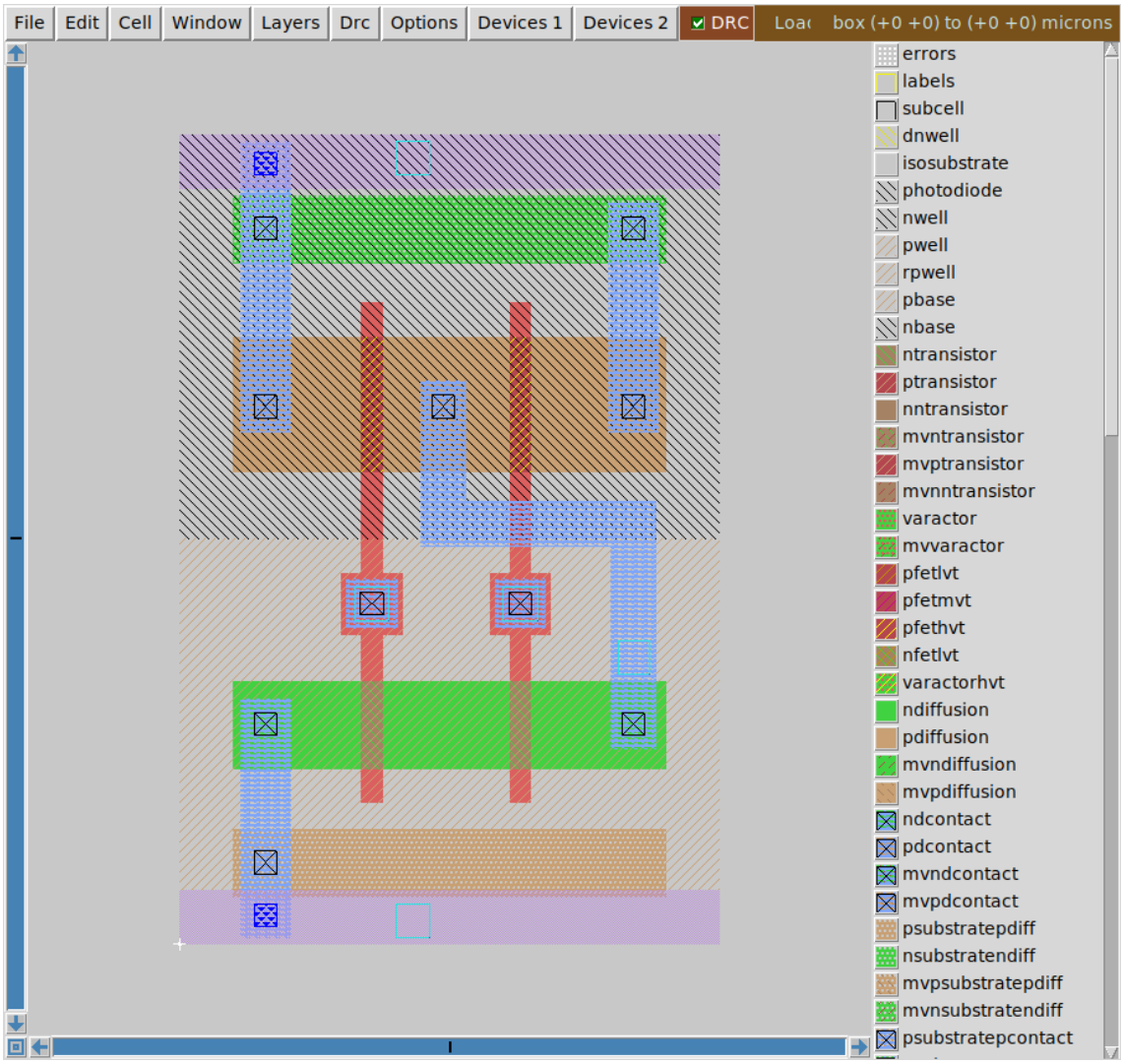


Figure 11.1: The NAND2 you get from that sketch

The same sketch drawn in Magic, with its two red poly stripes. The green n-diffusion runs under both of them, with contacts only at the far left and far right and none between the gates. The orange p-diffusion above it has three contacts, because parallel devices alternate source, drain, source.

Choosing the order

For a 2-input gate the order of the transistors barely matters, but for anything bigger it decides whether the cell is one strip of diffusion or three. Each break in the strip costs you a contact, a gap and roughly a quarter of a micron of width.

At this level a rule of thumb is all you need: **arrange the inputs so that consecutive transistors in the series chain share a terminal.** If you write the gate as a chain of source–drain pairs and you can walk through every transistor without lifting your pen, you can draw it as one unbroken strip. (Formally, you are asking whether the transistor graph has an Euler path.)

You won't need to choose an order yourself in AD104. In the capstone, though, you'll place the foundry's `and2_0`, which is a single unbroken diffusion strip with three gates on it, and when you do you should recognise that someone chose that order on purpose.

12. Extraction

A layout in Magic is a set of painted rectangles. Extraction is Magic reading those rectangles back and working out **what circuit they actually form**, which is not necessarily the circuit you meant to draw. That's the reason to do it at all, because an extractor that reported your intentions would tell you nothing you didn't already know.

The commands

```
extract do local          ;# extract this cell's own geometry
extract all               ;# ...and every subcell in it
ext2spice lvs             ;# format the result for netgen
ext2spice -o my_inv.spice
```

`extract` writes a `.ext` file in Magic's own format, which isn't very readable, and `ext2spice` turns that file into a SPICE netlist. The `lvs` argument is a preset that formats the netlist the way netgen wants it: no parasitic resistors, the subcircuits kept hierarchical, and no `.include` of the models.

Reading the netlist

Here is the extracted netlist of the reference inverter for Lab 03. Yours will have the same devices, though its ports may be listed in a different order, depending on the order in which you created the labels:

```
.subckt my_inv A Y VPWR VGND
X0 Y A VGND VGND sky130_fd_pr__nfet_01v8     ad=0.715 pd=3.5 as=0.6175 ps=3.2 w=0.65
l=0.15
X1 Y A VPWR VPWR sky130_fd_pr__pfet_01v8_hvt ad=1.1   pd=4.2 as=0.95   ps=3.9 w=1
l=0.15
.ends
```

Every field in it comes from something you drew:

Field	Where it came from
`.subckt my_inv A Y VPWR VGND`	the four labels you placed, promoted by `port makeall`
`X0 Y A VGND VGND`	drain, gate, source and bulk, worked out from which shapes touch which
`w=0.65`	the **height** of the n-diffusion under the poly
`l=0.15`	the **width** of the poly stripe where it crosses
`ad=0.715`	the area of the drain diffusion, in µm²
`pd=3.5`	the perimeter of the drain diffusion, in µm
`sky130_fd_pr__pfet_01v8_hvt`	the device flavour, decided by which implant layer you painted

`ad` and `pd` are easy to skim past, and they matter a great deal. Every square micron of drain diffusion is a reverse-biased junction to the body (the p-substrate under the NMOS, the n-well under the PMOS), and a reverse-biased junction is a capacitor, so the more drain area you draw, the more capacitance sits on the drain. Your inverter's drain area is **4.2 times** the foundry's, and Lab 05 measures what that costs.

Node names you did not choose

Extracting a NAND gate gives you a node with a name like `a_300_260#`:

```
X0 a_300_260# A VGND VGND sky130_fd_pr__nfet_01v8 w=0.65 l=0.15
X3 Y B a_300_260# VGND     sky130_fd_pr__nfet_01v8 w=0.65 l=0.15
```

That node is the diffusion shared between the two series NMOS transistors, the piece of silicon between them with no contact on it. You never labelled it, so Magic named it after its coordinates and added a `#` to mark it as internal. Seeing that name confirms that the two devices share one strip of diffusion, which is what you drew them to do.

So a `#` in a node name is nothing to worry about in itself. It is a problem when a node *you* labelled shows up with a `#`, because that means the label did not land on the geometry you thought it did.

PMOS flavours

The two PMOS models below are not interchangeable:

```
sky130_fd_pr__pfet_01v8        # regular threshold
sky130_fd_pr__pfet_01v8_hvt    # high threshold
```

Which one you get depends on whether the `pfethvt` implant layer covers the gate region. The foundry's standard cells all use the **hvt** PMOS. If you draw a PMOS and forget the implant, you get the regular one instead, and:

- **DRC never mentions it**, because both are legal devices.
- The layout looks identical to a correct one apart from a single hatched rectangle.
- LVS fails with a device-class mismatch. That's the outcome you want, because it means the mistake has been caught.
- Without LVS you would ship a chip with the wrong threshold voltages everywhere.

Lab 03's checker looks for this mistake and names it when it finds it.

Two kinds of extraction

You can extract the same geometry in two ways and get two different netlists:

```
ext2spice lvs             ;# devices only. This is what LVS wants.
ext2spice cthresh 0       ;# ...and every parasitic capacitance as well
```

`cthresh` is a threshold in femtofarads, and Magic throws away any capacitance smaller than it. The default drops the small ones, and since most of them are small, most of them go. If you set it to `0`, Magic keeps every capacitance it found:

```
.subckt my_inv A Y VPWR VGND
X0 Y A VGND VGND sky130_fd_pr__nfet_01v8     ...
X1 Y A VPWR VPWR sky130_fd_pr__pfet_01v8_hvt ...
C0 A Y      0.02936f
C1 Y VPWR   0.06727f
C2 A VPWR   0.10808f
C3 Y VGND   0.20685f
C4 A VGND   0.27253f
C5 VPWR VGND 1.33952f
.ends
```

That netlist has six capacitors that nobody drew. They're there because any two conductors near each other form a capacitor, and a layout is full of conductors near each other.

Use the `lvs` netlist for LVS and the `cthresh 0` one for simulation; Labs 03 and 04 and the capstone write both. If you compare a parasitic netlist against a schematic in netgen, the capacitors have nothing in the schematic to match, so the comparison produces only noise.

What extraction does not know

- **It does not know what you meant.** Checking the layout against what you meant is LVS's job, and that's the subject of the next page.
- **It does not model resistance by default.** `ext2spice` can extract resistors too, but that makes the flow much slower, and this course doesn't need it. Do keep in mind that li is about 12.8 Ω per square while metal1 is 0.125, so a long li wire is a real resistor, and the numbers in this course leave it out.
- **It does not model the third dimension beyond the layer stack**, which is why Magic is called a 2.5-D extractor. Its answers are good enough to be useful, but not good enough for sign-off.

13. Layout versus schematic (LVS)

DRC asks whether a layout *can be manufactured*. LVS asks a different question: whether the layout *is the circuit you meant*.

The two questions are independent. A layout can be perfectly manufacturable and still be a short circuit, and a layout can be exactly the right circuit and still be impossible to build. You need both answers, and they come from different tools.

netgen

The LVS tool you'll use is **netgen**, which compares two SPICE netlists as graphs. It ignores coordinates, port order, and the names of devices and internal nets. Instead it walks through the two circuits and asks whether there is a one-to-one correspondence between their devices and between their nets. The names of the cell's pins are the exception, and the first failure below is what you get when they disagree.

```
netgen -batch lvs \
  "results/my_inv.lvs.spice my_inv" \
  "spice/my_inv_sch.spice my_inv" \
  "$PDK_ROOT/sky130A/libs.tech/netgen/sky130A_setup.tcl" \
  results/lvs.report
```

The command takes four arguments: your layout netlist and the cell in it, the schematic netlist and the cell in it, the PDK's setup file, and the file to write the report to.

The setup file is easy to treat as a formality, but netgen depends on it. It tells netgen that `sky130_fd_pr__nfet_01v8` has interchangeable source and drain, which W and L to compare and with what tolerance, and which properties to ignore. Without it, netgen reports every symmetric device as a mismatch.

A match

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

The word "uniquely" means that netgen found exactly one correspondence between the two circuits. A symmetric circuit can have several, and when it does, netgen says so.

Two failures to recognise

"Netlists match uniquely" with "failed pin matching"

This is the more confusing of the two failures, so in Lab 04 we have you produce it on purpose with `make swapped`:

```
Netlists match uniquely.

Subcircuit pins:
Circuit 1: my_nand2         |Circuit 2: my_nand2
----------------------------|----------------------------
VPWR                        |VPWR
Y                           |Y
VGND                        |VGND
B                           |A **Mismatch**
A                           |B **Mismatch**

Final result: Top level cell failed pin matching.
```

The report is making two separate statements, and it helps to read them apart:

- **The topology is right.** The layout has four devices, wired exactly as the schematic says.
- **Two names are on the wrong shapes.** To make the match work, netgen had to swap `A` and `B`.

A NAND is symmetric in its inputs, so the circuit still works, and nothing else in the flow will ever notice the swap: simulation passes and DRC passes. The timing model is wrong all the same, because in a series NMOS stack the input nearest the output is measurably faster than the one nearest ground.

To fix it, swap the two labels in the layout, not in the schematic.

Different device or net counts

```
Circuit 1 contains 4 devices, Circuit 2 contains 5 devices.
```

When the counts differ, something is missing from one of the circuits. In these labs the cause is almost always one of these:

- **A wire that doesn't land.** Your li (local interconnect) or metal wire stops 0.02 µm short of the pin it was meant to touch, so the layout has one more net than the schematic, and one input floats.
- **A missing label.** Without a label there is no port, and so nothing for netgen to compare.
- **A missing tap**, which turns the bulk terminals into invented nodes. The full DRC rule set explains this.
- **The wrong device flavour**, such as `pfet_01v8` in the layout where the schematic says `pfet_01v8_hvt`. Netgen reports this as a device class with no partner.

Whatever the cause, read the report from the bottom up. The last table before `Final result:` names the device or net that couldn't be matched, and that name points you at the problem.

Where the schematic netlist comes from

In this course the schematic netlist comes from a file. Each lab ships a small `.spice` file with the circuit written out by hand, because a hand-written netlist is unambiguous and runs in a bare container:

```
.subckt my_inv A Y VPWR VGND
XM1 Y A VGND VGND sky130_fd_pr__nfet_01v8     w=0.65 l=0.15
XM2 Y A VPWR VPWR sky130_fd_pr__pfet_01v8_hvt w=1    l=0.15
.ends
```

In a real project it comes from your schematic editor. With XSchem, which you used in AD103, the command is `xschem -n -s -q -x file.sch`. There is one trap here that's worth knowing about now, so that it doesn't cost you an afternoon later:

> If you netlist a schematic on its own, XSchem writes its devices at the top level with no `.subckt` around them, and netgen then says `Cannot find cell my_inv`. Instead, netlist a *wrapper* schematic that instantiates your symbol, so that the netlist has a subcircuit in it for netgen to compare.

Why the labs run LVS

DRC failures are loud and local: Magic hatches the spot and `drc why` names the rule, so they're cheap to fix. LVS failures are quiet, and the cause can be anywhere in the cell.

Take a wire that stops 0.02 µm short of its pin. At any zoom you would normally use, it looks perfect. It passes DRC, because a gap doesn't violate any rule, and it extracts without complaint. Netgen is the only tool in the flow that notices, because it counts the nets and finds one too many.

That's why every lab in this course except Lab 05 runs LVS, including Lab 02. In Lab 02 the point is that **erasing the shape that broke a DRC rule doesn't fix it**: the DRC count drops to zero, but LVS immediately reports that the cell is no longer an inverter.

14. Lab 03: Draw an inverter

AD103's capstone was a CMOS inverter that you could simulate. In this lab you draw the same circuit as geometry: two transistors sharing one gate, clean under both DRC rule sets, and checked by netgen to be the circuit you meant.

Package: `labs/lab-03-draw-an-inverter/`

Before you start

- Finish Lab 02.
- Read Stick diagrams, Extraction and Layout versus schematic (LVS). You'll use Drawing in Magic constantly.
- Have paper and a pen for Part 1.

Time: about 3 hours. Each `make` takes about 1.1 s.

The sizes: Wn = 0.65 µm, Wp = 1.00 µm, L = 0.15 µm. These are exactly the numbers in `sky130_fd_sc_hd__inv_1`, which you read in Lab 01. In Lab 05 you put your cell next to the foundry's and measure the difference, and because the transistors are identical, the only thing left to compare is the way each cell is drawn. L = 0.15 µm is the SKY130 minimum (`poly.1a`), but your cell will still be nearly five times the foundry's area, because area depends on how tightly the shapes are packed rather than on L.

Part 1: Sketch it on paper

What goes where, before any dimensions?

1. Draw the inverter stick diagram from Stick diagrams **from memory**, not copied: PMOS above, NMOS below, one vertical poly stripe crossing both.
2. Mark **where the two well taps go** and which rail each one connects to. The guide page's sketch leaves them out, and leaving them out is also the most common bug in a first layout, so drawing them by hand once is cheap insurance.
3. Decide which side of the cell the input is on and which side the output is on, and why. Part 6 shows where this lab's reference cell puts them.

The whole sketch takes about ninety seconds.

You should see a sketch with both taps marked, the output on one side and the input on the gate.

Part 2: Start from the frame

What does the frame already give you, and what is missing?

1. Go into the lab package and lay down the frame:

```
cd labs/lab-03-draw-an-inverter
make scaffold
```

`make scaffold` **throws away whatever is in `mag/my_inv.mag`** and starts again from the frame, so use it now, and after this only when you want to start over.

2. Open the cell in Magic:

```
make edit
```

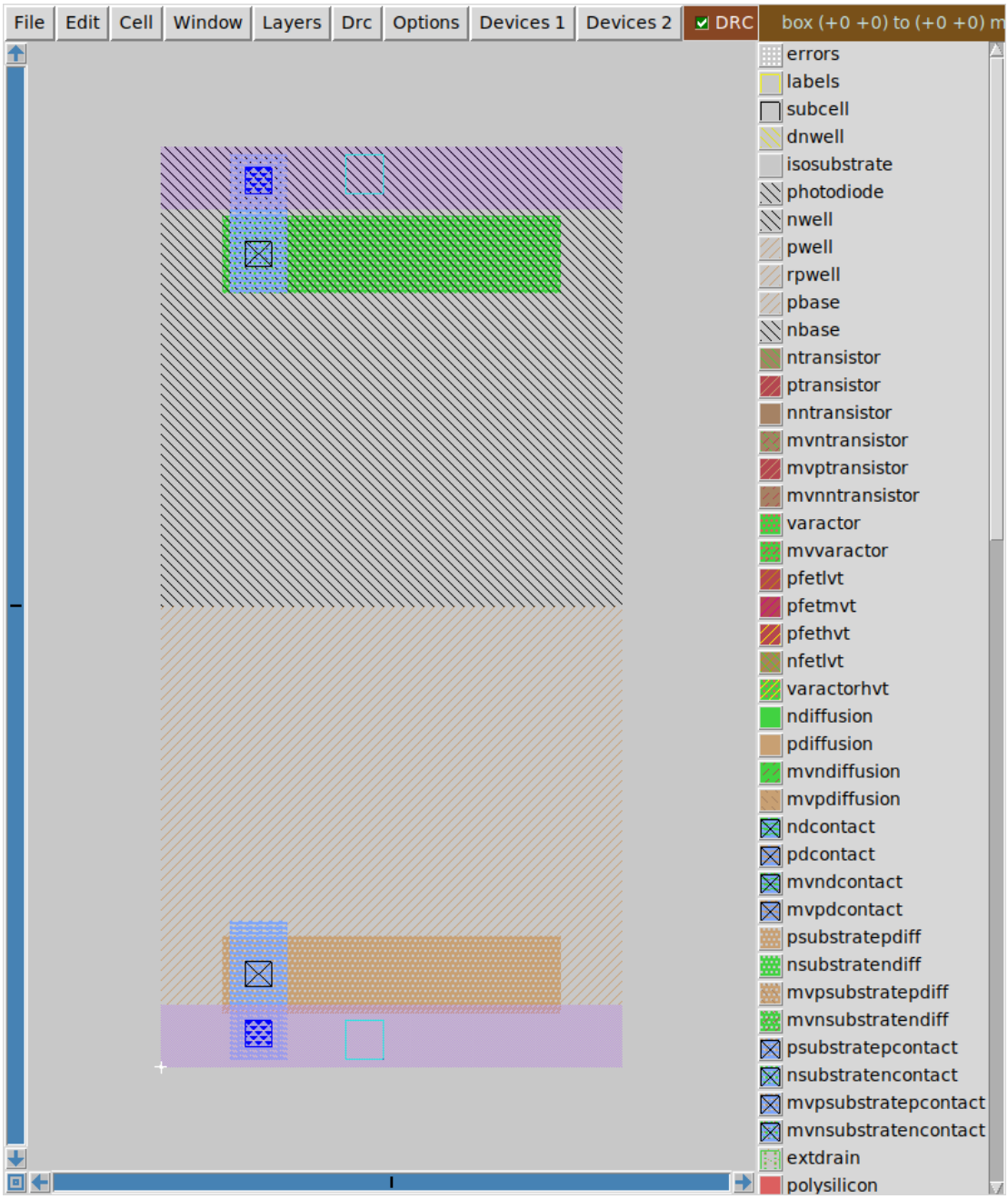


Figure 14.1: The frame you start from

The frame has everything except the transistors. The n-well is the grey hatch across the top half, and the p-substrate is the orange hatch below it. The green strip is the n-well tap and the brown one is the substrate tap, each with a contact and a short li run up to a purple metal1 rail. The middle of the cell is left for you to draw.

The frame already contains:

- the n-well (y = 3.00 to 6.00) and the p-well below it
- the two metal1 supply rails, with `VGND` and `VPWR` already labelled
- both **well taps**, contacted, with li up to the rails through `mcon`

Compare that with your sketch. The taps you marked are already drawn, so the frame takes care of the tap problem from Lab 02, and what's missing is the middle of the cell.

3. Run the checker:

```
make
```

```
  WHAT MAGIC FOUND IN my_inv.mag

      transistors           0   (0 n, 0 p)

      ports                 VGND VPWR

      DRC, default rule set  0
      DRC, full rule set     0

      LVS                   Top level cell failed pin matching.

FAIL  No transistors at all. A transistor is the place where a poly stripe
      crosses a diffusion. If Magic sees no crossing, either the poly does
      not overlap the diffusion, or one of them is not the layer you think
      it is - check the layer name in the title bar as you paint.
      ...
```

The checker reports every failure in this lab like that, in words, naming the layer or the rule involved. Fix one thing at a time, and run `make` again after each fix.

You should see `FAIL` with "No transistors at all".

Part 3: Draw the two diffusions

How big is each diffusion, and is a diffusion on its own a transistor?

1. In the `tkcon` console (or by painting with the mouse):

```
box 0.40um 1.30um 2.60um 1.95um
paint ndiff                       ;# 0.65 um tall. The height IS W.

box 0.40um 3.50um 2.60um 4.50um
paint pdiff                       ;# 1.00 um tall, inside the nwell
```

The n-well starts at y = 3.00, so the PMOS diffusion clears its edge by 0.50 µm, which is comfortably more than the 0.18 µm that `diff/tap.8` needs.

2. Save:

```
save my_inv
```

`make` reads `mag/my_inv.mag` from disk. Painting in Magic changes only what's on your screen, and the file stays as it was until you `save`. Until then, every `make` reports on the last version you saved, and neither Magic nor `make` warns you that this is why the output hasn't changed. From here on, **save before every `make`**.

3. Before you run `make`, predict how many transistors it will find.

   Answer: A1, in Answers at the end.

4. Run `make` in the terminal.

You should see still `transistors 0`.

Part 4: Draw the gate

Where does the stripe go, and what sets W and L?

1. Paint the poly stripe:

```
box 1.35um 1.05um 1.50um 4.75um
paint poly
```

The stripe is 0.15 µm wide, and that width is the transistors' L. It runs from below the NMOS diffusion to above the PMOS one, overhanging each by 0.25 µm, because a gate that stops inside the diffusion leaves a leakage path round its end (`poly.8`).

2. `save my_inv`, then run `make`.

You should see two transistors with the right W and L, and DRC still at zero under both rule sets:

```
      transistors            2   (1 n, 1 p)
        X0   sky130_fd_pr__nfet_01v8          W=0.65  L=0.15
        X1   sky130_fd_pr__pfet_01v8          W=1  L=0.15
```

The checker also complains that `X1` is a `pfet_01v8` where it expected a `pfet_01v8_hvt`. Leave that for now; you'll fix it in Part 8.

Part 5: Add the contacts

What do the contacts need around them?

1. Paint four contacts, each **exactly 0.17 × 0.17 µm**:

```
box 0.55um 1.55um 0.72um 1.72um ; paint ndc     ;# NMOS source
box 2.28um 1.55um 2.45um 1.72um ; paint ndc     ;# NMOS drain
box 0.55um 3.90um 0.72um 4.07um ; paint pdc     ;# PMOS source
box 2.28um 3.90um 2.45um 4.07um ; paint pdc     ;# PMOS drain
```

2. The gate needs a contact too, but it can't sit on the 0.15 µm stripe. `licon.8` asks for 0.05 µm of poly all round a 0.17 µm cut, and that takes a stripe 0.27 µm wide. Instead, draw a landing pad of poly in the gap between the diffusions and put the contact on that:

```
box 1.20um 2.30um 1.65um 2.75um ; paint poly
box 1.34um 2.44um 1.51um 2.61um ; paint pc
```

3. `save my_inv`. Before you run `make`, predict whether the DRC count goes up or stays at zero.
4. Run `make`.

You should see DRC jump to 35 under both rule sets, with three entries under `rules broken`:

```
      rules broken:
        Local interconnect overlap of poly contact < 0.08um in one direction (li.5)
       Local interconnect overlap of diffusion contact < 0.08um in one direction (li.5)
        Local interconnect minimum area < 0.0561um^2 (li.6)
```

Why did five correctly sized contacts break two li rules?
Answer: A2, in Answers at the end.
In Part 6 you fix all 35 at once.

Part 6: Wire it in li

Which contacts join which, and how wide does the li have to be?

1. Paint four li runs. Each one has to overhang its contacts by 0.08 µm on both sides along one axis (`li.5`), and the 0.37 µm width is there to leave room for that:

```
box 0.45um 0.05um 0.82um 1.82um ; paint li    ;# VGND: tap -> NMOS source
box 2.18um 1.45um 2.55um 4.17um ; paint li    ;# Y: NMOS drain -> PMOS drain
box 0.45um 3.80um 0.82um 5.95um ; paint li    ;# VPWR: PMOS source -> tap
box 1.24um 2.34um 1.61um 2.71um ; paint li    ;# A: the gate
```

2. `save my_inv`, then run `make`.

You should see DRC back to 0 under both rule sets, and LVS reporting `Top level cell failed pin matching`, because the circuit is there but `A` and `Y` don't have names yet.

Part 7: Name the pins

How does netgen know which shape is `A` and which is `Y`?

1. Label the input and output, turn the labels into ports, and save:

```
box 1.30um 2.40um 1.55um 2.65um ; label A c li
box 2.25um 2.60um 2.50um 2.85um ; label Y c li
select top cell
port makeall
save my_inv
```

2. Run `make`.

You should see four ports, `VGND VPWR A Y`, and two transistors. LVS still says `Netlists do not match`; Part 8 explains why.

Part 8: Fix the PMOS flavour

Why does LVS still fail when DRC is clean and every pin has a name?

1. Read the device lines in the `make` output again:

```
        X1   sky130_fd_pr__pfet_01v8          W=1  L=0.15
```

   The device is a `pfet_01v8` rather than a `pfet_01v8_hvt`, which means you have built the wrong transistor. The foundry's PMOS is the high-threshold flavour, and the flavour you get is decided by an implant mask that nothing you've drawn so far mentions. Why did DRC never complain?

   Answer: A3, in Answers at the end.

2. Paint the implant over the PMOS gate region and save:

```
box 1.35um 3.50um 1.50um 4.50um
paint pfethvt
save my_inv
```

3. Run `make`.

You should see `PASS`:

```
  WHAT MAGIC FOUND IN my_inv.mag

      transistors            2   (1 n, 1 p)
        X0   sky130_fd_pr__nfet_01v8          W=0.65  L=0.15
        X1   sky130_fd_pr__pfet_01v8_hvt      W=1  L=0.15

      ports                  VGND VPWR A Y

      DRC, default rule set  0
      DRC, full rule set     0

      LVS                    Circuits match uniquely.

PASS  Two transistors, the right flavours, the right W and L, four ports,
      zero DRC violations under both rule sets, and netgen agrees your
      geometry is the schematic.
```

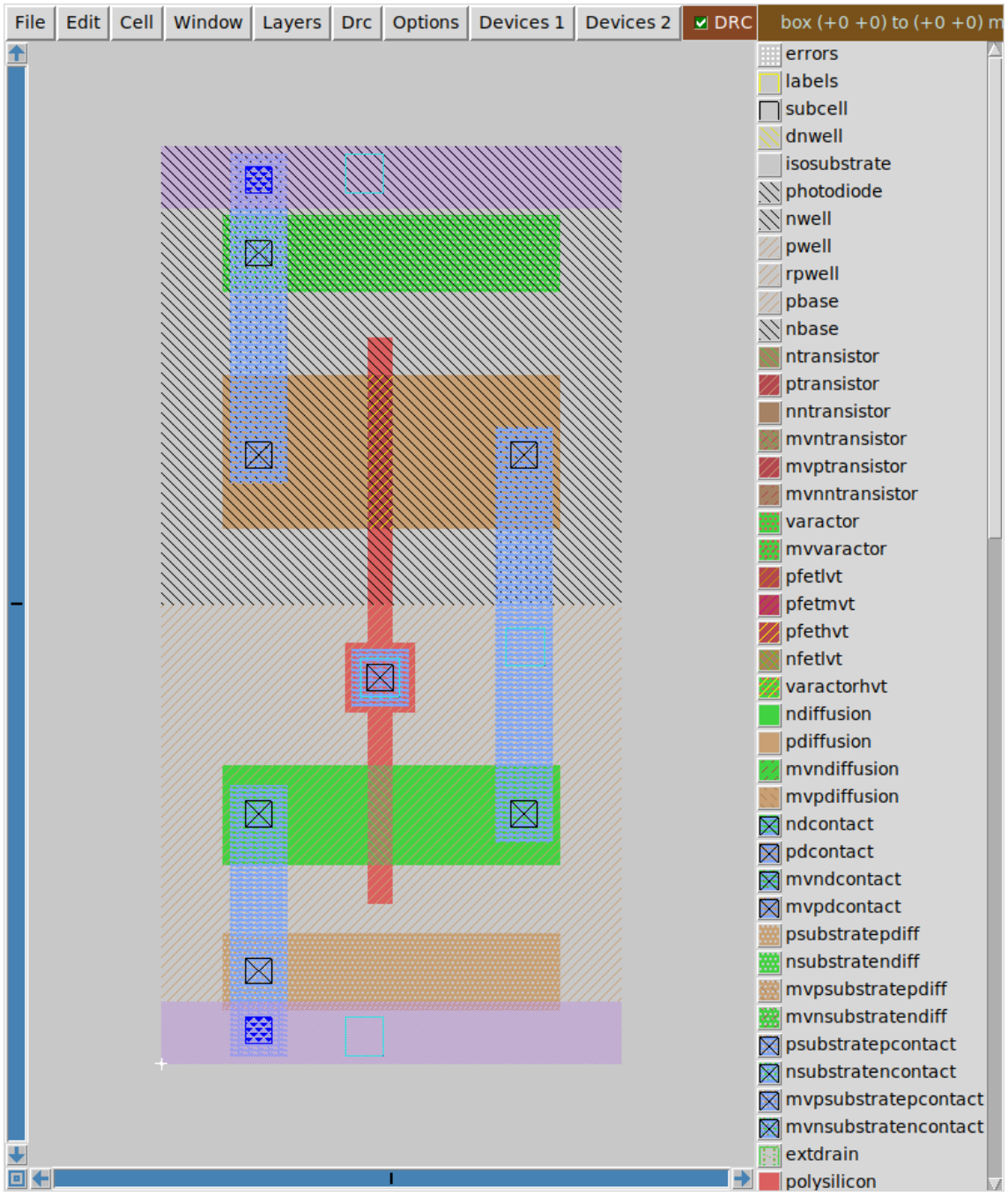


Figure 14.2: The finished cell

Part 9: Read what you drew

Where in your geometry do `w=0.65` and `w=1` come from?

1. Print the netlist Magic extracted from your geometry:

```
make netlist
```

```
.subckt my_inv VGND VPWR A Y
X0 Y A VGND VGND sky130_fd_pr__nfet_01v8 ad=0.715 pd=3.5 as=0.6175 ps=3.2 w=0.65
l=0.15
```

```
X1 Y A VPWR VPWR sky130_fd_pr__pfet_01v8_hvt ad=1.1 pd=4.2 as=0.95 ps=3.9 w=1 l=0.15
.ends
```

Extraction explains every field. Your port order depends on the order in which you created the labels, so it may differ from the reference's.

2. Open `mag/my_inv.mag` in a text editor and find the `<< ndiff >>` section. The file has no `magscale` line, so one unit is 0.01 µm, and the section's `rect` lines together span `40 130` to `260 195`. That's your n-diffusion, 0.40 ⟶ 2.60 µm by 1.30 ⟶ 1.95 µm, which Magic has split into tiles. Drawing in Magic explains the units.

You should see `w=0.65` and `w=1`, both at `l=0.15`, and you should be able to point at the rectangle whose height is `w=0.65`.

Finishing

You're done when `make` ends in `PASS` on your own `mag/my_inv.mag`, and `make netlist` shows two devices, `w=0.65` and `w=1`, both at `l=0.15`.

Questions:

- Which rectangle's height is `w=0.65`? Part 9, step 2 answers it.
- Why did your cell need LVS, as well as DRC, to show that it was the wrong circuit? Part 8 answers it.

Submit: nothing for this lab.

Going further (optional)

After yours passes, compare it with the reference:

- `make refcheck` runs every check on the reference cell instead of yours.
- `tcl/inverter.tcl` draws the reference cell command by command, with a comment on every line saying which rule the number comes from. It didn't come out right the first time either: the package's `solutions/README.md` shows the 62 violations its first version had.

Next on the Course path: read Instances and hierarchy, then do Lab 04: A buffer and an AND gate.

Answers

A1. Part 3: Draw the two diffusions

None. A transistor is the place where a poly stripe crosses a diffusion, and there's no poly yet, so a diffusion on its own isn't a transistor.

A2. Part 5: Add the contacts

Each contact came with its own patch of li exactly the size of the cut, but li has to overhang a contact (`li.5`). That 0.17 µm square of li is also smaller than li's minimum area of 0.0561 µm² (`li.6`).

A3. Part 8: Fix the PMOS flavour

Both are legal devices, so DRC has nothing to object to. Only LVS catches the mistake, because it compares your devices with the schematic's, and that is the case for always running LVS.

IV Building with cells

15 Instances and hierarchy 73
15 Placing an instance 73
15 Abutment 73
15 Hierarchy 74
15 Three common mistakes 74
15 What a place-and-route tool does 74

16 Lab 04: A buffer and an AND gate . . 77
16 Before you start 77
16 Part 1: Look at the NAND 77
16 Part 2: Wire the buffer 79
16 Part 3: Name the buffer's pins 79
16 Part 4: Build the AND gate 79
16 Part 5: Simulate the AND gate 80
16 Part 6: Meet the pin-swap message 80
16 Part 7: Take away the block's rails (optional) . . 81
16 Finishing 82
16 Going further (optional) 82
16 Answers 82

15. Instances and hierarchy

By the end of Lab 03 you have drawn an inverter by hand, and you aren't going to draw a million more of them that way.

From here on the work is **instancing**: you place copies of finished cells next to each other and draw the wires between them. The rest of a chip is built the same way, and that's possible because the hard part, the transistors, is drawn only once. You'll compose your first two blocks this way in Lab 04.

Placing an instance

In Magic, you place an instance by typing these commands in the console:

```
addpath ../mag                  ;# where to look for cells
box 3um 0um 3um 0um             ;# a zero-size box at the spot you want
getcell my_inv                  ;# drop a copy with its origin there
identify inv2                   ;# give this copy a name
```

`getcell` places an **instance**, which refers to `my_inv` rather than copying its geometry. If you run those four commands in a new cell, from a folder next to `mag/`, and save it, the `.mag` file records the whole instance in four lines:

```
use my_inv  inv2 ../mag
timestamp 1787733743
transform 1 0 300 0 1 0
box 0 0 300 600
```

The `use` line names the cell, the instance and the folder to look for the cell in. The `timestamp` is a copy of the one at the top of `my_inv.mag`, `transform` puts the cell's origin at x = 300, which is 3 µm in this file's 0.01 µm units, and `box` is the outline of `my_inv` in its own coordinates. Magic writes the folder only on the first `use` of each cell in a file, so in Lab 04's `solutions/buf_ref.mag` the line for `inv1` ends in `../mag` and the line for `inv2` doesn't. When the cell is in the same folder as the file that uses it, as in Lab 04's `mag/my_buf.mag`, none of the `use` lines has a path.

Because every instance refers back to `my_inv.mag`, editing that one file changes all of them. That's the reason to use instances, and it's also why Lab 04's blocks are only a few lines long.

Abutment

Both of Lab 04's cells, `my_inv` and `my_nand2`, are **6.00 µm tall**, with their VGND rail at y = 0.00–0.40 and their VPWR rail at y = 5.60–6.00. When you put the second cell at an x equal to the width of the first, the two cells touch edge to edge, and:

- the two VGND rails become one continuous strip,
- the two VPWR rails become one continuous strip,
- the wells touch and merge,
- and the second cell has power without you drawing anything.

That is what makes a cell a **standard cell**: it keeps to a fixed height and fixed rail positions, so that placing cells is just arithmetic.

The foundry's library keeps to a height of 2.72 µm and a site width of 0.46 µm, so every cell in it is a multiple of 0.46 µm wide. In the capstone you place four of its cells at x = 0, 7.36, 9.66 and 11.96, each one the running total of the widths of the cells before it, and the rails line up without any further work from you.

Hierarchy

In outline, Lab 04's buffer is:

```
load my_buf
place my_inv inv1 0.0
place my_inv inv2 3.0
paint li from inv1/Y across to inv2/A
label A, label X, label VGND, label VPWR
```

When you extract it, you get this netlist:

```
.subckt my_buf A X VGND VPWR
Xinv1 A inv2/A VPWR VGND my_inv
Xinv2 inv2/A X VPWR VGND my_inv
.ends
```

At this level there are two instances and the net between them, and no devices at all. The devices are inside `my_inv`, which is written out separately in the same file. netgen compares hierarchically, so in one run it checks that `my_buf` is two `my_inv` in series *and* that `my_inv` is the inverter it's supposed to be.

The AND gate uses the same idea with a different pair of cells, `my_nand2` feeding `my_inv`. An AND gate has six transistors, and you don't draw any of them.

Three common mistakes

1. A label over a subcell's geometry names nothing.

Say you place two cells and abut them, and the rails connect. Then you put the box over a rail and type `label VGND`. A label always goes into the cell you're editing, so it lands in your cell, but the rail under it is geometry inside `inv1` and your cell has nothing of its own there. So Magic puts the label on empty space (if you type `label VGND c m1`, it says so: `Moving label "VGND" from metal1 to space`), and a label on space isn't attached to any net. *Your* cell then comes out of extraction with no supply ports at all, and netgen reports a pin-matching failure on a block that is otherwise perfect.

The fix is to give the top cell some geometry of its own to put the name on. Paint a metal1 rectangle over the row (it's the same net, so it changes nothing electrically) and put the label on that. Every composed block in this course does this, and the solutions READMEs for Lab 04 and the capstone explain it as a decision you can break on purpose to see what happens.

2. The layers above a cell are not always empty.

`and2_0` and `buf_1` contain metal1 only in their rails, but the flip-flop `dfxtp_1` also has internal metal1 straps at y = 1.415–1.645 and 1.755–1.985, running most of its width. If you route metal1 across the flip-flop without knowing that, you get `met1.2` spacing violations from geometry you didn't draw.

That's why real routers carry a blockage map for every cell, and it's also why the capstone's `D` pin takes its via at an unexpected place. Before you route over a cell, read the `<< metal1 >>` section of its `.mag` file; the capstone has you do this.

3. The path recorded in the `.mag` file is relative to where you saved it.

`use my_inv  inv2 ../mag` means "look in `../mag`, starting from the folder this `.mag` file is in", rather than from the folder you started Magic in. That's why you can load Lab 04's `solutions/buf_ref` from the lab folder and Magic still finds `my_inv` in the lab's `mag/`. If you move the file to a folder with no `../mag` next to it, though, Magic can't find the cell: it prints a `couldn't be read` error for `my_inv.mag`, and the instance shows up as an empty box with a name in it. The labs avoid this by always changing to a known directory first and setting `addpath` after that, and the order is worth copying in your own work: **`cd` first, `addpath` second.**

What a place-and-route tool does

Placement and routing were the first parts of layout to be automated, and they are the most thoroughly automated, because the job is enormous and it's arithmetic. In a real flow,

1. synthesis picks a cell for each gate in the netlist, choosing drive strengths,
2. the place-and-route tool finds legal positions in rows for every one of them,
3. inserts tap cells and filler,
4. routes hundreds of thousands of nets on five metal layers,

5. and then checks its own work with the same DRC and LVS you are running.

In DD103 that whole flow runs in **about a minute** on a small design, and DD103's own pages time it at 57 s for the vending machine and 59 s for the elevator. Placing four cells and routing three wires by hand in the capstone is no substitute for that tool, but it does show you what the tool's output is a picture of.

16. Lab 04: A buffer and an AND gate

In this lab you'll build two more cells without drawing a single new transistor, because both are made from cells that are already drawn: a buffer is two copies of the Lab 03 inverter, and an AND gate is a NAND gate followed by the same inverter. The work is placing the cells, abutting them, wiring them together and naming their pins, which is also what the rest of a chip is made of.

Package: `labs/lab-04-buffer-and-and/`

Before you start

- Get Lab 03 to `PASS`.
- Read Instances and hierarchy. This lab also uses Stick diagrams and Layout versus schematic (LVS).

Time: about 2 hours, not counting the optional Part 7. The full `make` takes 10 to 15 seconds (10.3 and 13.4 s on two timed clean runs), and nearly all of that is ngspice reading the SKY130 model library for the truth-table simulation. `make layout` runs only the DRC and LVS part, which takes about 1.5 s, so use it while you are wiring.

Part 1: Look at the NAND

How is the NAND you were given wired, in silicon?

1. Go into the lab package and build the starting cells:

   ```
   cd labs/lab-04-buffer-and-and
   make cells        # (re)build mag/my_inv.mag and mag/my_nand2.mag
   make scaffold     # two blocks, each with its two cells already placed
   ```

   `mag/my_inv.mag` is the Lab 03 reference inverter rather than the one you drew, so this lab works on its own. Be careful with `make scaffold`: it **throws away your work in `my_buf` and `my_and2`**, so run it again only if you want to start over.

2. The NAND comes already drawn, and `tcl/nand2.tcl` is the script that drew it, commented line by line. Open the cell:

   ```
   make edit CELL=my_nand2
   ```

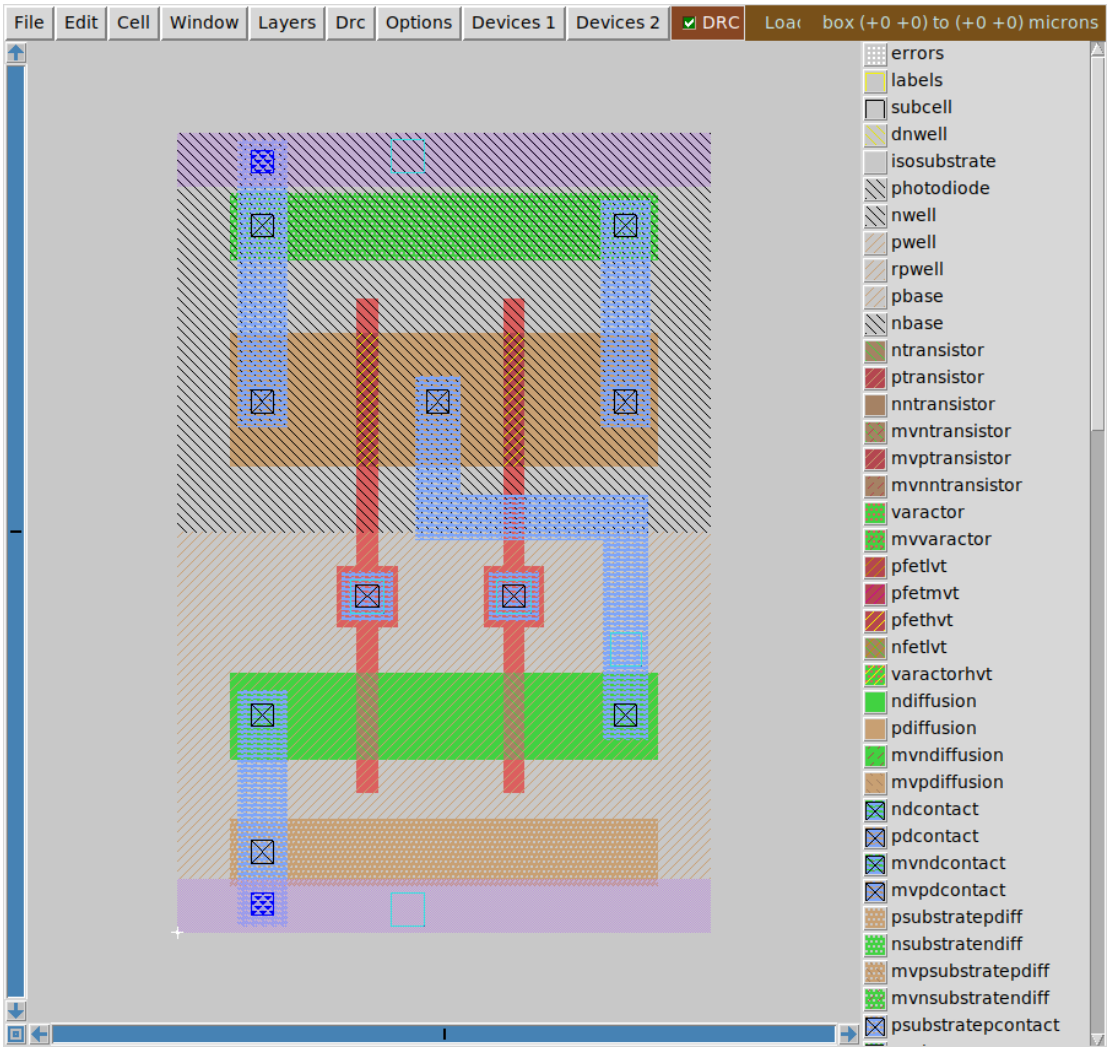


Figure 16.1: The NAND2

*Two red poly stripes cross one strip of green n-diffusion, with contacts only at the far left and far right and nothing between the gates. The two NMOS are in **series**, and on silicon that means the drain of one and the source of the other are the same piece of doped silicon.*

The PMOS above them are in **parallel**, with three contacts that alternate source, drain, source, and the middle one goes to `Y`.

3. Run the layout checks once, so that Magic extracts the blocks:

```
make layout
```

It fails, because nothing is wired or named yet, but it extracts both blocks on the way. Open `results/my_and2.lvs.spice` and find the `my_nand2` subcircuit, where the two NMOS lines share a node you never named:

```
X0 a_300_260# A VGND VGND sky130_fd_pr__nfet_01v8 ... w=0.65 l=0.15
X3 Y B a_300_260# VGND sky130_fd_pr__nfet_01v8 ... w=0.65 l=0.15
```

`a_300_260#` is the shared diffusion between the two gates. You never gave it a name, so Magic made one up from its coordinates.

Why is there no contact on the shared node between the two NMOS?
Answer: A1, in Answers at the end.

You should see the NAND's single n-diffusion strip under both gates, and `a_300_260#` in the netlist.

Part 2: Wire the buffer

What do the two inverters already share, and what is missing?

1. Open the buffer:

   ```
   make edit CELL=my_buf
   ```

   `make scaffold` gave you `my_buf` containing two `my_inv` instances, at x = 0 and x = 3.0, with the block's own metal1 rails painted over the row and already labelled `VGND` and `VPWR`. Both cells are 6.00 µm tall with their rails at the same heights, so the rails already abut into two continuous strips and the wells have merged. That means power is already connected, even though you haven't drawn anything yet.

2. What's missing is the signal from the first inverter to the second. `inv1`'s output `Y` is an li shape at x = 2.18–2.55, and `inv2`'s input `A` is at x = 3 + 1.24 = 4.24–4.61, both at around y = 2.4, so a single li rectangle can join them:

   ```
   box 2.18um 2.40um 4.61um 2.65um
   paint li
   ```

You should see a single li rectangle running from `inv1`'s output to `inv2`'s input, which is all the wiring the buffer needs.

Part 3: Name the buffer's pins

How do you give the block, rather than the cells inside it, its own pins?

1. Label the input and output, turn the labels into ports, and save:

   ```
   box 1.30um 2.40um 1.55um 2.65um ; paint li ; label A c li
   box 5.20um 2.42um 5.45um 2.67um ; paint li ; label X c li
   select top cell
   port makeall
   save my_buf
   ```

   Each of the two label lines paints a small patch of `li` before it places the label. Why not put the label straight on the inverter's own pin?

   Answer: A2, in Answers at the end.

2. Run the layout checks:

   ```
   make layout
   ```

You should see the `my_buf` row read `ok`, `0`, `0`, `match`. The `my_and2` row will keep failing until you wire it in Part 4.

Part 4: Build the AND gate

Can you find the AND gate's pins yourself?

1. Open the AND gate:

   ```
   make edit CELL=my_and2
   ```

   It contains `my_nand2` (instance `nd0`) at x = 0 and `my_inv` (instance `in0`) at x = 4.0, with the block's rails painted and labelled the same way as in `my_buf`.

2. `nd0`'s output is at x = 3.20–3.53, and `in0`'s input is at 4 + 1.24 = 5.24–5.61, so join them with one li rectangle, as you did in the buffer:

```
box 3.20um 2.40um 5.61um 2.65um
paint li
```

3. Paint li patches and label them: `A` and `B` over the NAND's two inputs, and `X` over the inverter's output. This time you work out the coordinates yourself. You'll find the NAND's input li runs on the `A` and `B` lines in `tcl/nand2.tcl`. For the inverter's output, start from the `X` patch you painted in Part 3, at x = 5.20–5.45 on `inv2`'s `Y`. That inverter sat 3.0 µm from the block's origin and `in0` sits 4.0 µm from it, so the patch moves 1.0 µm to the right.
4. Run `select top cell`, `port makeall` and `save my_and2`, and then run `make layout`.

You should see both rows read `ok`, with `0`, `0` and `match`, and `PASS  Both blocks are clean under both rule sets`.

Part 5: Simulate the AND gate

Does the AND gate really AND, on the extracted layout?

1. Before you run it, write down the output `X` you expect for each of the four input rows.
2. Run the full check:

   ```
   make
   ```

   This extracts both blocks, runs LVS on them against `spice/cells_sch.spice`, and then simulates the AND gate's truth table on the **extracted layout, with parasitics**:

```
  TWO BLOCKS, BUILT OUT OF CELLS

  block      DRC default  DRC full   LVS                      contains
          ---------------------------------               ------------------------
----------------------------
  ok my_buf            0         0   match                    my_inv, my_inv
  ok my_and2           0         0   match                    my_inv, my_nand2

  THE AND GATE, SIMULATED

      ok  A=0  B=0   X = 0.000000 V   (expected 0.0)
      ok  A=0  B=1   X = 0.000000 V   (expected 0.0)
      ok  A=1  B=0   X = 0.000001 V   (expected 0.0)
      ok  A=1  B=1   X = 1.800000 V   (expected 1.8)

PASS  A buffer and an AND gate, both clean under both rule sets, both
      matching their schematics, and the AND gate really does AND.
```

The AND gate you just simulated has six transistors, and you didn't draw any of them in this lab.

`src/mktb.py` generates the testbench **around your netlist's own port order**, instead of assuming one. It does this because Magic numbers a cell's ports in the order you created the labels, so if you labelled `X` before `A`, your `.subckt` line lists the ports in a different order from the reference's. A testbench that connected the terminals in the wrong order would give a wrong answer without raising any error.

You should see all four truth-table rows marked `ok`, and `PASS`.

Part 6: Meet the pin-swap message

What does netgen say when the circuit is right but two names are on the wrong shapes?

1. LVS your NAND against a netlist with `A` and `B` exchanged:

   ```
   make swapped
   ```

```
Netlists match uniquely.

Subcircuit pins:
Circuit 1: my_nand2         |Circuit 2: my_nand2
----------------------------|----------------------------
VPWR                        |VPWR
Y                           |Y
VGND                        |VGND
B                           |A **Mismatch**
A                           |B **Mismatch**

Final result: Top level cell failed pin matching.
```

2. Read the report as two separate statements, which don't contradict each other:
   - **the topology is right**: all four devices are wired exactly as the schematic says
   - **two names are on the wrong shapes**: netgen had to swap `A` and `B` to make the match work

Why will nothing else in the flow notice this?
Answer: A3, in Answers at the end.
To fix it, swap the two labels in the layout rather than the pin names in the schematic.
You should see `Netlists match uniquely` and `failed pin matching` in the same report.

Part 7: Take away the block's rails (optional)

What happens to the block's supply ports if it has no rails of its own?

1. Open the buffer with `make edit CELL=my_buf`, and erase the block's two metal1 rail rectangles in the console:

   ```
   box 0um 0um 6um 0.40um ; erase m1
   box 0um 5.60um 6um 6um ; erase m1
   save my_buf
   ```

   The `VGND` and `VPWR` labels go too, because they sat on those rectangles, while the inverters' own rails are left untouched.

2. Run `make layout`, and the `my_buf` row fails:

   ```
       my_buf          0        0   Top level cell failed pin matching  my_inv, my_inv
   ```

   and the `FAIL` line names the missing ports:

   ```
   FAIL  my_buf has no port called VGND, VPWR. Its ports are: A X.
   ```

   The two inverters' rails still abut into continuous strips. Why does the block lose its supply ports anyway?

   Answer: A4, in Answers at the end.

3. Put the rails and their labels back with the scaffold's own commands (from `tcl/scaffold.tcl`), then run `make layout` again:

   ```
   box 0um 0um 6um 0.40um ; paint m1
   box 0um 5.60um 6um 6um ; paint m1
   box 2.60um 0.05um 2.85um 0.30um ; label VGND c m1
   box 2.60um 5.70um 2.85um 5.95um ; label VPWR c m1
   select top cell
   port makeall
   save my_buf
   ```

You should see `my_buf has no port called VGND, VPWR` after step 2, and the `my_buf` row back to `ok`, `0`, `0`, `match` after step 3.

Finishing

You're done when `make` ends in `PASS` on the two blocks you wired and named: both clean under both rule sets, both `match` in LVS, and all four truth-table rows `ok`.

Questions:

- Why is there no contact on the shared node between the two NMOS in the NAND? Part 1 answers it.
- `make swapped` reports `Netlists match uniquely` and `failed pin matching` together. What is right, and what is wrong? Part 6 answers it.

Submit: nothing for this lab.

Going further (optional)

`make ref` and `make refcheck` build and check the reference blocks. The package's `solutions/README.md` discusses three decisions in them.

Next on the Course path: read Area, parasitics and delay, then do Lab 05: What it costs.

Answers

A1. Part 1: Look at the NAND

In series, the drain of one transistor is the source of the other, so on silicon they are the same piece of diffusion, and the region between the two gates is itself the shared node. A contact there would cost area and add junction capacitance to an internal node without gaining you anything (Stick diagrams).

A2. Part 3: Name the buffer's pins

A label names a net only if it sits on a shape in the cell you're editing. The input pin you want to name is a shape inside `inv1`, so a label placed straight on it would stay in `my_buf` with nothing of `my_buf`'s under it. Magic moves a label like that onto empty space (`Moving label "A" from locali to space`), where it isn't attached to any net, and `my_buf` would come out with no `A` port. Painting your own 0.25 µm patch on top of the pin (it's the same net, and electrically it adds nothing) gives the top cell a shape of its own to name. The scaffold's metal1 rails are there for the same reason: they give the `VGND` and `VPWR` labels top-cell geometry to sit on.

A3. Part 6: Meet the pin-swap message

A NAND is logically symmetric in its inputs, so with `A` and `B` swapped both simulation and DRC still pass. What the swap does get wrong is the timing model, because in a series stack the input nearer the output is faster than the one nearer ground.

A4. Part 7: Take away the block's rails (optional)

The labels went with the rectangles, and the only rail metal left is inside `inv1` and `inv2`. A label over that metal would stay in `my_buf` but land on empty space, as in Part 3, so the block has no shape of its own to carry `VGND` and `VPWR`, and it comes out of extraction with no supply ports at all.

V

What layout costs

17 Area, parasitics and delay 85
17 The numbers 85
17 FO4 delay and ring-oscillator delay 86
17 Where the femtofarads come from 86
17 Area 87
17 Summary 87

18 Lab 05: What it costs 89
18 Before you start 89
18 Part 1: Predict the price 89
18 Part 2: Run the comparison 89
18 Part 3: Compare the two delay rows 90
18 Part 4: Look at the capacitors 90
18 Part 5: Read the ring oscillator 91
18 Part 6: Measure your own cell (optional) 91
18 Finishing 92
18 Going further (optional) 92
18 Answers 92

17. Area, parasitics and delay

This page explains the experiment that Lab 03's transistor sizes were chosen for.

Take **the same two transistors**, with the same width, the same length and the same device flavours, right down to the model name, and draw them twice: once the way the foundry did, packed to the last 0.005 µm, and once the way you do in Lab 03, with room to move. Because the devices are identical, any difference between the two cells has to come from the drawing. You run the comparison in Lab 05.

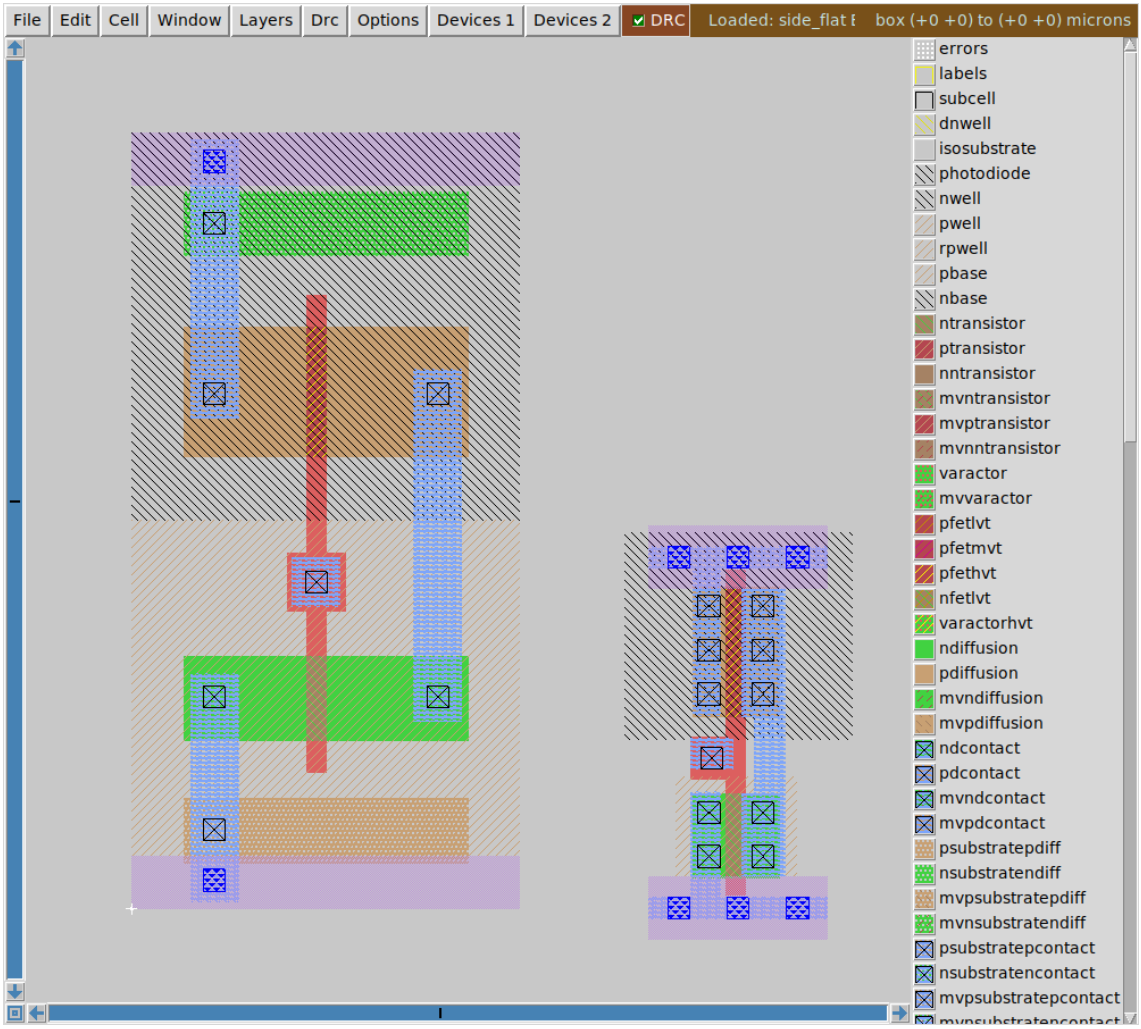


Figure 17.1: Yours on the left, the foundry's on the right, at the same scale

Both cells are one NMOS at W = 0.65 µm and one high-Vt PMOS at W = 1 µm, with L = 0.15 µm for both. The devices are the same, and yours takes 4.8 times the area.

The numbers

Lab 05 runs the extraction and both simulations and prints this table:

	yours	the foundry's	ratio
cell width	3.00 µm	1.38 µm	2.17 ×
cell height	6.00 µm	2.72 µm	2.21 ×
cell area	**18.00 µm²**	**3.7536 µm²**	**4.80 ×**
NMOS W / L	0.65 / 0.15	0.65 / 0.15	1.00 ×
PMOS W / L	1 / 0.15	1 / 0.15	1.00 ×
NMOS drain junction area	0.715 µm²	0.169 µm²	4.23 ×
PMOS drain junction area	1.1 µm²	0.26 µm²	4.23 ×
extracted capacitance, total	1.86 fF	1.424 fF	1.31 ×
FO4 delay, average	**62.88 ps**	**57.73 ps**	**1.09 ×**
ring-oscillator stage delay	**42.58 ps**	**32.80 ps**	**1.30 ×**

Look at the two delay rows. Your cell is 9 % slower than the foundry's in one test and 30 % slower in the other, and most of this page is about why the two tests disagree by that much.

FO4 delay and ring-oscillator delay

A gate's delay is, roughly, how long its transistors take to charge whatever is attached to the output. That load has two parts:

- **the load you are driving**: the gate capacitance of the cells downstream
- **the load you brought with you**: your own drain junctions and your own wiring, which have to be charged whether anything is connected or not

Your layout controls only the second part, and for a given layout it stays close to a fixed number of femtofarads.

In a **fan-out of 4 (FO4)** test, the output drives four copies of the same cell. The gate capacitance of those four copies dominates the load, and it's the same for both cells because their W and L are the same. Your extra junction area is then only a small share of the total, so you come out 9 % behind.

A ring oscillator is a loop in which each stage drives exactly one copy of itself and nothing else. With only one gate downstream, the self-load is a much bigger fraction of the total, so your 4.23 × junction area shows up in full and you come out 30 % behind.

> **Layout parasitics act roughly like a fixed extra load, so they're lost in the noise when a gate drives something big, and they dominate when it drives something small.**

Most gates in a real chip drive something small, which is why the foundry packs its cells to the last nanometre even though "it is only a few femtofarads".

Where the femtofarads come from

Here is part of what `ext2spice cthresh 0` pulls out of each cell:

```
yours, all of it              the foundry's, only the lines on Y
C0 Y A       0.02272f         C1 Y VPWR    0.07413f
C1 VPWR A    0.10808f         C5 Y VPB     0.01774f
C2 Y VPWR    0.06413f         C6 VGND Y    0.05975f
C3 Y VGND    0.33997f         C7 Y A       0.03773f
```

```
C4 VPWR VGND 1.32518f         C11 Y VNB    0.1626f
                              ... 14 lines in total
```

(Magic doesn't always write the `C` lines in the same order, so yours may be numbered differently, but the values don't change from run to run.)

Your total is higher, 1.86 fF against 1.424 fF, and so is the part that Lab 05 prints as *"of it on the output"*: 0.4268 fF for yours against 0.3519 fF for theirs. Neither ratio, 1.31 × and 1.21 ×, comes anywhere near the 4.23 × of the drain junction areas.

That's because the extracted capacitors don't include the drain junctions. They are the capacitance of the **wiring** and the well: between one wire and another, from the wires down to the substrate and the well (the lines ending in `VNB` and `VPB` in the foundry's list), and between the well and the substrate. The **junction capacitance** of the source and drain diffusions lives inside the device model instead, which computes it from the `ad`, `as`, `pd` and `ps` parameters, and your values for those are 4.23 × theirs.

Adding it up:

- The extracted capacitance is against you, but only by 1.2 to 1.3 times.
- Junction capacitance is strongly against you, by a factor of four.
- Junction capacitance is also the bigger of the two numbers.

You can't read this off the parasitic netlist, because the junction capacitance isn't in it. You have to simulate both cells to see the total, which is why Lab 05 runs the simulations.

Area

18 µm² against 3.75 µm² can sound like an academic difference, but it stops being one when you multiply it by the number of cells on a chip.

A modest open-source SoC has a few hundred thousand standard cells. At the foundry's density that comes to about 1.5 mm² of logic, and at yours it's 7 mm². Die cost goes roughly with area, and yield goes *down* as area goes up, so each finished chip costs several times more and you get fewer good ones per wafer. That is the whole commercial argument for a standard-cell library, and it's the reason a library is drawn once, by someone very patient, and then instanced a million times.

Summary

Your layout is deliberately roomy so that you can draw it by hand in an afternoon, and Lab 05 puts a price on that choice. These are the points to take away:

1. **Identical transistors don't give identical circuits**, because everything that isn't in the schematic is decided by the layout.
2. **The cost of a loose layout is small per gate and very large per chip.**
3. **You can measure it.** Measuring the gap between your cell and the foundry's takes one `make` and about half a minute, so you don't have to argue about it.

18. Lab 05: What it costs

In this lab you compare two layouts of the same two transistors, your Lab 03 inverter and the foundry's `inv_1`, to see what the drawing costs in area and in delay. You start by working out the area price from numbers you already have. Then you extract both cells with every parasitic capacitance kept, simulate them, and check your arithmetic against the measurement.

Package: `labs/lab-05-what-it-costs/`

Before you start

- Get Lab 03 to `PASS`.
- Read Area, parasitics and delay. This lab also uses Extraction.
- Have paper or a text file ready for the numbers you work out in Parts 1 and 5.

Time: about 1 hour, not counting the optional Part 6. Each full `make` takes about half a minute (27 to 39 s on seven timed clean runs, depending on what else the machine is doing). Nearly all of that time goes on the two ngspice runs, about 13 s for the fan-out deck and 15 s for the two ring oscillators, which between them have 62 extracted subcircuits to simulate for 20 ns of transient. Each run starts by reading the whole SKY130 model library.

Part 1: Predict the price

How much does the drawing cost in area, judging only from numbers you have already seen?

1. The Lab 03 cell is 3.00 × 6.00 µm, and Lab 01 measured `inv_1` at 1.38 × 2.72 µm. Work out how many times bigger the Lab 03 cell is, and write it down.
2. The extracted netlists give each transistor's drain junction area as `ad=`: 0.715 µm² for the Lab 03 NMOS and 1.1 µm² for its PMOS (Lab 03, Part 9), against 0.169 µm² and 0.26 µm² for `inv_1` (Lab 01, Part 5). Work out both ratios and write them down.
3. Guess how much slower the Lab 03 cell will be. Is it closer to the area ratio, the junction ratio, or much less than either? Write down a number, because Part 3 compares it with the measurement.

You should see three ratios and a guess in your notes. Part 2 checks the ratios against the table.

Part 2: Run the comparison

Does the measurement agree with your arithmetic?

1. Go into the lab package and run it:

   ```
   cd labs/lab-05-what-it-costs
   make
   ```

   By default it compares the **reference** Lab 03 inverter, in `mag/my_inv.mag`, with the foundry's. The reference uses the same transistor and wiring coordinates as Lab 03's steps, so if you followed those steps, the `yours` column describes the cell you drew. Part 6 (optional) swaps in your own file instead. The run prints:

   ```
   THE SAME TWO TRANSISTORS, DRAWN TWICE

                                     yours       foundry           ratio
   ------------------------------------------------------------------------
   cell width                            3          1.38 um       2.17 x
   ```

```
  cell height                              6          2.72 um       2.21 x
  cell area                               18         3.754 um2      4.80 x

  NMOS: sky130_fd_pr__nfet_01v8
     W                                  0.65          0.65 um       1.00 x
     L                                  0.15          0.15 um       1.00 x
     drain junction area               0.715         0.169 um2      4.23 x
     drain junction perim                3.5          1.82 um       1.92 x
  PMOS: sky130_fd_pr__pfet_01v8_hvt
     W                                     1             1 um       1.00 x
     L                                  0.15          0.15 um       1.00 x
     drain junction area                 1.1          0.26 um2      4.23 x
     drain junction perim                4.2          2.52 um       1.67 x

  extracted capacitance, total          1.86         1.424 fF       1.31 x
       ... of it on the output        0.4268        0.3519 fF       1.21 x

  DELAY, fan-out of 4
  t_pHL                                37.32         33.95 ps       1.10 x
  t_pLH                                88.43         81.51 ps       1.08 x
  average                              62.88         57.73 ps       1.09 x

  RING OSCILLATOR, 31 stages, nothing but the cell driving itself
  period                                2640          2034 ps       1.30 x
  frequency                            378.8         491.7 MHz      0.77 x
  one stage                            42.58          32.8 ps       1.30 x
  ...

PASS  Both cells extracted with parasitics, both simulated, and the
      comparison is above. There is no wrong answer here - it is a
      measurement, and yours is yours.
```

2. Find the four `1.00 x` rows for W and L. They're the control in this experiment: the transistors in the two cells are identical, down to the model name, so every other difference in the table comes from the way they were drawn.
3. Compare the `cell area` row and the two `drain junction area` rows with the ratios you worked out in Part 1.

You should see `1.00 x` on every W and L row, `4.80 x` for the cell area and `4.23 x` on both drain junction area rows, matching your Part 1 ratios to rounding, and `make` ending in `PASS`.

Part 3: Compare the two delay rows

Why does the same drawing cost more delay in one test than in the other?

1. Find the two delay ratios in your table: the fan-out 4 `average` row and the ring `one stage` row. On this page they are 1.09 x at fan-out 4 and 1.30 x in the ring, so the cell is 9 % slower in one test and 30 % slower in the other. Compare yours with the guess you wrote down in Part 1.
2. Work out why the two ratios differ before you read the `WHAT TO TAKE FROM THIS` block at the end of your `make` output, because that block gives the answer. Hint: a gate's delay is how long it takes to charge whatever hangs off its output, and that load has two parts: what you are driving, and what you brought with you.

   Answer: A1, in Answers at the end.

You should see that the self-loaded test (the ring) shows the bigger penalty.

Part 4: Look at the capacitors

Where does the extra capacitance live?

1. Print both parasitic netlists side by side:

```
make caps
```

2. Compare the two capacitance rows with the drain junction areas from Part 1:

```
  extracted capacitance, total          1.86         1.424 fF       1.31 x
       ... of it on the output        0.4268        0.3519 fF       1.21 x
```

   Your cell has more extracted capacitance than the foundry's, but only 1.2 to 1.3 times as much, even though its drain junctions are 4.23 times the size. As Area, parasitics and delay explains, the extracted `C` elements are the capacitance of the wiring and the well, and the drain junctions aren't among them.

3. Where does the junction capacitance of the source and drain diffusions live, and why isn't it in the `C` lines?

   Answer: A2, in Answers at the end.

You should see the `C` lines for both cells, with no junction capacitance among them.

Part 5: Read the ring oscillator

How long does one gate take when it drives nothing but a copy of itself?

A loop of an odd number of inverters can't settle, because whatever value a node holds comes back inverted one lap later. So the loop oscillates, with period $2Nt_p$ for N stages of delay t_p each. That makes it the cleanest way to measure a single gate delay, because there is no source and no external load, only the cell driving one copy of itself.

1. Open `src/mkring.py`, which writes both 31-stage rings. It is about fifty lines long, and `N = 31` near the top sets the number of stages. Near the end it starts each ring's nodes alternately low and high, so that a single edge runs round each ring.

2. From the `period` row of your table, work out the delay of one stage for each cell, $t_p = T/(2 \times 31)$, and write both down. Then compare them with the `one stage` row.

   Answer: A3, in Answers at the end.

The foundry's 33 picoseconds is the floor for this process at 1.8 V: it's the delay of one gate driving nothing but a copy of itself. The capstone's timing numbers are multiples of it, and What sets the clock frequency builds on it.

You should see your two stage delays agree with the `one stage` row, to rounding.

Part 6: Measure your own cell (optional)

1. Copy your Lab 03 cell in and run the comparison again:

```
make mine MAG=../lab-03-draw-an-inverter/mag/my_inv.mag
make
```

   A cell drawn by following Lab 03 lists its ports as `VGND VPWR A Y`, because the frame you started from already had `VGND` and `VPWR` labelled, while the reference lists them as `A Y VPWR VGND`. So `make` reads the order out of your extracted netlist, wires both decks around it, and prints it on its `wrote results/fo4.spice` line. If your cell is missing a port, `make` stops there instead and names the ports it found.

2. Check the W and L rows. If they don't read `1.00 x`, your transistors aren't the ones Lab 03 asked for, so the comparison no longer isolates the effect of the drawing.

To go back to the reference cell afterwards, run `git checkout mag/my_inv.mag` and then `make`.

You should see `1.00 x` on the W and L rows. If you used Lab 03's coordinates exactly, the whole table is the same as the reference's.

Finishing

You're done when the area and drain-area ratios you worked out in Part 1 match the table's `cell area` and `drain junction area` rows, and the stage delays you worked out in Part 5 match its `one stage` row.

Questions:

- Why does the ring show a bigger penalty than fan-out 4? Part 3 answers it.
- Where does junction capacitance live, and why is it not in the `C` lines? Part 4, step 3 answers it.

Submit: nothing for this lab.

Going further (optional)

`make table` prints the comparison again from the results already on disk, so it returns instantly. `make pex` runs only the extraction, which takes well under a second.

Next on the Course path: read Routing by hand and What sets the clock frequency, then do the Capstone: Place and route.

Answers

A1. Part 3: Compare the two delay rows

At fan-out 4 most of the load is the four downstream gates, and their input capacitance is the same whichever cell is driving them. In the ring each stage drives one gate and nothing else, so your 4.23 × junction area makes up a large share of the total load. Area, parasitics and delay has the full explanation.

A2. Part 4: Look at the capacitors

It's inside the device model, which computes it from the `ad`, `as`, `pd` and `ps` parameters on each device line, and your drain areas are 4.23 × theirs. Because it lives in the model, you can't read it off the parasitic netlist; the only way to see its effect is to simulate, which is why this lab does.

A3. Part 5: Read the ring oscillator

For the table on this page: 2640.2 ps / 62 = 42.58 ps for your cell, and 2033.6 ps / 62 = 32.80 ps for the foundry's, which the table prints as `32.8`. The table rounds the periods to 2640 and 2034, which is why dividing the rounded numbers gives 42.58 and 32.81.

VI Place and route

19 Routing by hand 95
19 The layer stack as a routing problem 95
19 Which layer to route on 96
19 The routing band 97
19 Why routers alternate directions 97
19 Metal already inside a cell 97
19 Pin shapes 97
19 The routing loop 97

20 What sets the clock frequency 99
20 The timing rule 99
20 One gate delay 99
20 A real path 99
20 Why a real chip runs at 200 MHz 100
20 The trade-offs 100

21 Capstone: Place and route 101
21 Before you start 101
21 Part 1: Look at the placement 101
21 Part 2: Check what is above each cell 102
21 Part 3: Route the three nets 102
21 Part 4: Bring out the pins 103
21 Part 5: Add the tap cells 104
21 Part 6: Time it 104
21 Part 7: Write the GDS 105
21 Finishing 106
21 Going further (optional) 106
21 Answers 106

19. Routing by hand

The capstone asks you to route only three nets, but three are enough to run into every basic idea in interconnect. This page explains those ideas, and you route the nets yourself in the capstone.

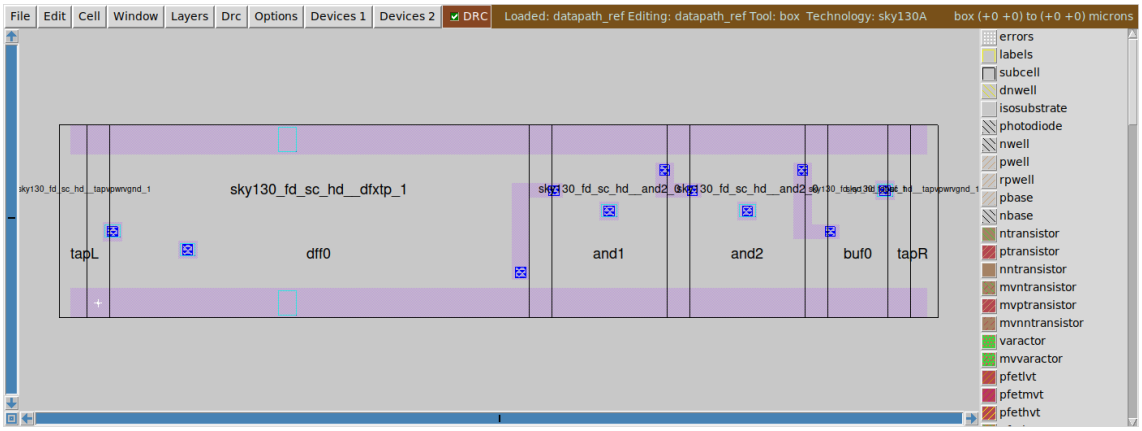


Figure 19.1: Four cells and three wires

The capstone block with its subcells collapsed, so the cells are outlines and your routing is the only real geometry on screen. Left to right: a tap cell, the flip-flop, two AND gates, the buffer, and another tap. The pink metal1 shapes and the boxed contacts are all of the routing you do by hand.

The layer stack as a routing problem

You can't connect just any layer to any other. The layers form a single ladder, and a connection can only step from one rung to the next through the contact or via between them:

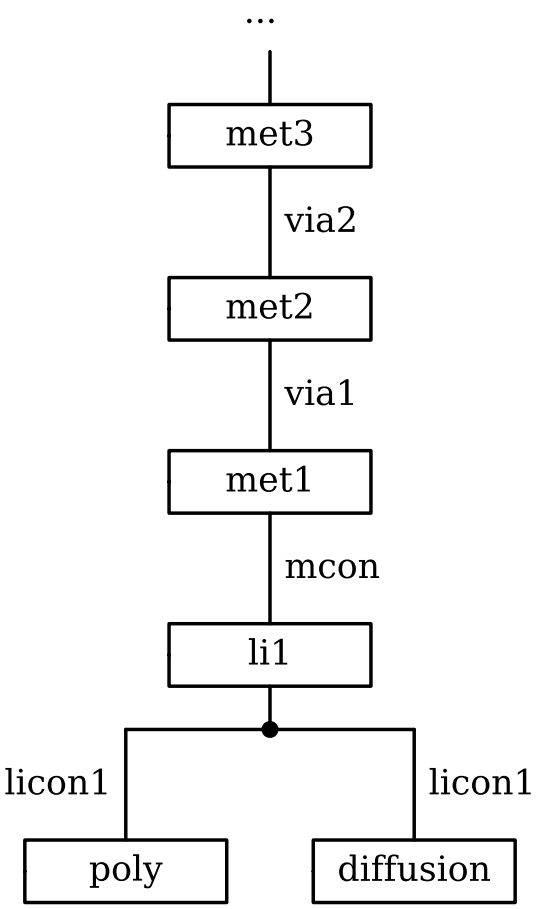


Figure 19.2: The ladder, with the contact or via that joins each pair of rungs.

Each of those contacts and vias is a plug of a fixed size, and each plug has enclosure rules for the layers on both sides of it. So a connection from a cell pin to a wire two microns away is built in four steps:

1. an `mcon` on the pin (li1 → metal1), exactly 0.17 µm square
2. a metal1 pad around it, surrounding the cut by 0.03 µm everywhere and by 0.06 µm on both sides of one axis (`met1.4`, `met1.5`)
3. metal1 from that pad to the next one
4. another pad and another `mcon` at the far end

The capstone's pads are 0.29 µm wide around a 0.17 µm cut, which leaves 0.06 µm on each side. That satisfies `met1.5` along the x axis, and it's the smallest width that does.

Which layer to route on

li1 is the layer the cells' own pins are drawn on, so a wire on li needs no via at either end. The price is resistance: li1 is about **12.8 Ω per square**, against **0.125** for metal1, which makes it about a hundred times worse. Use it only for a wire short enough that you can see both ends. Lab 04′s blocks route on li for that reason: the wire there is 2.4 µm long, so saving the vias is worth more than the extra resistance costs.

metal1 is where ordinary local routing goes. Of the cells in the capstone, all but one use metal1 only for their rails, so the band between the rails is free space above every cell **except the flip-flop**, which has internal metal1 straps of its own.

metal2 and up aren't used by `sky130_fd_sc_hd` at all, so if you need to cross the row, go up to them.

The routing band

The rails occupy y = −0.24 to 0.24 and y = 2.48 to 2.96, and two metal1 shapes have to be at least 0.14 µm apart. So any metal1 wire that isn't part of the supply has to stay between:

```
y = 0.38   and   y = 2.34
```

That leaves 1.96 µm of vertical space, shared by every signal in the row. It's enough for the capstone, but not for a real block, which is why routers alternate directions.

Why routers alternate directions

The convention is that each metal layer runs in one direction only, and the layers take turns. In SKY130 the standard-cell library's technology LEF, `sky130_fd_sc_hd__nom.tlef`, sets metal1 horizontal, metal2 vertical, metal3 horizontal, and so on up to metal5. Metal1 runs the same way as the supply rails in your row.

That can look like bureaucracy, but it's the only thing that makes routing tractable. Two wires on the same layer going the same way compete for the same track, while two wires on adjacent layers at right angles can cross without interacting. Once every layer has a direction, the question "is there room?" stops being about arbitrary geometry and becomes a question about a grid of tracks, which a computer can answer.

All three of your nets go on metal1, which breaks the rule, but with only three nets you can afford to. While you route them, notice that you can keep the whole picture in your head, and that somewhere around ten nets you no longer could.

Metal already inside a cell

Before you run metal1 across a cell, check what metal1 the cell already has by reading the `<< metal1 >>` section of its `.mag` file. On the workbench image, `and2_0` has 16 metal1 rectangles and `buf_1` has 12, all of them in the rails. The flip-flop, `dfxtp_1`, has 74, including two long horizontal straps across the middle of the cell. Those straps are why the capstone's `D` connection can't put its via at the D label: a pad there would sit 0.07 µm from a strap, which breaks `met1.2`.

The D pin's li shape extends down to y = 0.715, clear of the straps, so the via goes down there instead. **Finding another place on the same pin to land** is the routing move you'll use most often, and once you've done it by hand you'll know what the tool is doing when it does the same thing.

Pin shapes

A cell's pin is a rectangle of li, and often it's several rectangles in different places, all on the same net. `buf_1`, for example, publishes its output `X` at three different heights. Any of them is a legal place to land, so pick the one your wire can reach.

You'll find the coordinates in the cell's `.mag` file, on its `flabel` lines. Here is the one for `and2_0`'s output `X`:

```
flabel locali s 397 425 431 459 0 FreeSans 340 0 0 0 X
port 7 nsew signal output
```

The units are 0.005 µm, as in every foundry cell's `.mag` file (Drawing in Magic shows how your own cells differ), so that pin covers x = 1.985–2.155 and y = 2.125–2.295, measured from the cell's origin. Add the x position you placed the cell at, and you know where to put the `mcon`.

The routing loop

The capstone turns this page into a step-by-step procedure: find both pins in absolute coordinates, put an `mcon` and a pad on each, join them inside the band, and then check your work. You need two checks, because they catch different mistakes. DRC, run under both rule sets (see The full DRC rule set), catches spacing and enclosure errors and missing tap cells, and LVS catches a wire that doesn't quite land on its pin.

20. What sets the clock frequency

Everyone knows that a chip has a clock speed, but few people are told what sets it. The answer is in the kind of layout you have been drawing, and this page works it out from numbers you measure in Lab 05 and the capstone.

The timing rule

Between two flip-flops there is some logic. In one clock period, three things have to happen in sequence:

1. the first flip-flop has to produce its output after the clock edge: the **clock-to-Q** delay
2. that value has to get through the logic and the wires: **the path delay**
3. it has to be stable at the second flip-flop's input a little *before* the next clock edge: the **setup time**

$$T_{\text{clk}} \geq t_{\text{clk}\to Q} + t_{\text{logic}} + t_{\text{setup}} \tag{20.1}$$

$$f_{\max} = \frac{1}{T_{\text{clk}}} \tag{20.2}$$

Static timing analysis is built on this one rule, and the rest of it is bookkeeping around those three terms.

One gate delay

The fastest thing this process can do is one inverter driving one inverter and nothing else, and you measured exactly that in Lab 05 with a **ring oscillator**, an odd number of inverters connected in a loop. Lab 05, Part 5 explains why such a loop can't settle and oscillates instead, with a period of $2Nt_p$ for N stages that each take t_p. The lab builds two 31-stage rings out of extracted layouts, one from the foundry's inverter and one from yours, and measures them:

	period	frequency	one stage
the foundry's `inv_1`	2033.6 ps	491.7 MHz	**32.80 ps**
your Lab 03 inverter	2640.2 ps	378.8 MHz	**42.58 ps**

So **33 picoseconds** is the floor: the delay of one gate in SKY130 at 1.8 V driving nothing but a copy of itself. Every other delay on this page can be counted in multiples of it.

A real path

The capstone block is a flip-flop, two AND gates and a buffer, placed in a row 14.6 µm long. If you extract it with parasitics, apply a clock edge and watch Y, you get these numbers:

```
CLK rise -> Y rise        536.17 ps    (measured on your extracted layout)
D setup at the flop       103.32 ps    (sky130_fd_sc_hd__tt_025C_1v80.lib)
shortest legal period     639.49 ps
f_max                    1563.75 MHz
```

That works out to **1.56 GHz**, for four levels of logic and almost no wire.

The two delays in it come from two different places:

- **The 536 ps is simulated** on your own geometry, with every parasitic capacitance Magic could find.
- **The 103 ps is looked up** in `sky130_fd_sc_hd__tt_025C_1v80.lib`, the same Liberty file the digital track synthesises against in DD103. Setup time depends on the flip-flop's internals, so you can't get it from your own layout. This is the point where the analog and digital tracks read the same file.

536 ps is **16 inverter delays** for only four gates, so it isn't the number of gates that makes the path slow. Most of it is the flip-flop's clock-to-Q, and the two `and2_0` cells you are given are the weakest AND in the library.

Why a real chip runs at 200 MHz

The gap between your 1.56 GHz and a real chip's 200 MHz comes out of the same arithmetic, and you can follow it one term at a time.

Logic depth. Four levels of logic is a toy. A real critical path, such as a 32-bit adder carry chain, a cache tag comparison, or a decoder feeding a mux tree, is routinely twenty to forty levels deep. At the same cost per level, about 134 ps (536 ps over four levels), that comes to 2.7–5.4 ns before any of the terms below.

Wires. Your capstone's longest net is about 2 µm. A net crossing a modest block is hundreds of microns long, and a bus crossing a die is millimetres. Wire delay grows roughly with the *square* of length, because both resistance and capacitance grow with it. Past a few hundred microns, almost all of the delay is in the wire, and the gates are a rounding error.

DD103's ripple-adder page measures the same effect in the digital track: a 4-bit ripple carry takes 1.481 ns and a 64-bit one takes 24.996 ns, so the delay comes from the length of the *chain* rather than from any one gate.

Clock skew and jitter. The clock edge doesn't arrive at every flip-flop at the same moment, and every picosecond of disagreement comes straight off the time budget.

Margin. All of the numbers above are for the typical corner at 25 °C. Sign-off is done at the slow-slow corner, at 125 °C and low supply, where everything is 30–50 % worse, and the chip still has to work there.

If you stack all of those on the 1.56 GHz you measured, a few hundred megahertz is what is left. Nobody has to be careless for that to happen; it's what the arithmetic gives.

The trade-offs

You can attack every one of those terms, and every attack costs something you also care about:

You want	You do	You pay
less path delay	use stronger cells (`and2_2` not `and2_0`)	area, and input capacitance upstream
less path delay	fewer levels of logic	more gates in parallel, more area
less wire delay	put connected things next to each other	the work of placement, which is a global problem
less self-load	pack the layout tighter	an afternoon of your time (Lab 05 measures what it buys)
more headroom	pipeline it: cut the path with another flip-flop	latency, and a flip-flop's worth of area and power

Lab 05 measures the fourth row directly: in a ring, your loose inverter is 30 % slower than the foundry's, even though the transistors are identical.

Because none of these moves is free, a synthesis and place-and-route tool has to choose among them, a few million times over, guided by an objective function. This course has been about understanding what it is choosing between.

21. Capstone: Place and route

You'll place four standard cells in a row, route three wires between them by hand, and then measure how fast you can clock what you built. This is the job a place-and-route tool does, at a scale you can hold in your head, so that when you watch a tool do it for a hundred thousand cells you know what you are looking at.

Package: `labs/capstone-place-and-route/`

Before you start

- Finish Labs 01–05.
- Read Routing by hand and What sets the clock frequency. Instances and hierarchy covers placing cells, and why a label on a subcell's shape does not name your block.
- Keep the DRC error decoder open in another tab.

Time: about 3 hours, and about half of that goes on routing. `make layout` runs only DRC and LVS and takes about 1.4 s, so it's the one to use while you route. The full `make` adds the timing simulation and takes about 15 seconds (12.6 to 15.6 s on four timed clean runs).

The circuit:

```
Q  = DFF(CLK, D)          sky130_fd_sc_hd__dfxtp_1
n1 = Q  AND A             sky130_fd_sc_hd__and2_0
n2 = n1 AND B             sky130_fd_sc_hd__and2_0
Y  = BUF(n2)              sky130_fd_sc_hd__buf_1
```

That's a flip-flop, two AND gates and a buffer. Together they make a register bit with some logic after it, which is the most common four-cell cluster in any synchronous chip. Because it has a clock, the timing question at the end of the lab is a real one.

Part 1: Look at the placement

What does a row of four standard cells look like before you wire anything?

1. Go into the lab package, lay down the placement, and open it:

   ```
   cd labs/capstone-place-and-route
   make scaffold     # four cells placed, nothing wired
   make edit
   ```

   Be careful with `make scaffold`, because it **throws away your routing**. Use it now, and after that only when you want to start over.

   It places the four cells at x = 0, 7.36, 9.66 and 11.96. Those numbers are a running total of the cells' widths, which come from each cell's `FIXED_BBOX` property. Every cell in the library is 2.72 µm tall with its rails at the same heights, so the rails abut into two continuous strips and the wells merge. At this scale placement is just arithmetic, and `tcl/place.tcl` is nothing more than four `getcell` calls.

2. Before you run the layout checks on the bare placement, predict the DRC count under each rule set. The foundry's inverter in Lab 01 is a clue. Then run them:

   ```
   make layout
   ```

   ```
     YOUR BLOCK
   ```

```
outline                13.72 x 3.20 um   =  43.90 um2
cells placed           2 x and2_0, 1 x buf_1, 1 x dfxtp_1
cells in the row       and2_0, buf_1, dfxtp_1
ports                  (none)
DRC, default rule set  0
DRC, full rule set     32
LVS                    Top level cell failed pin matching

rules broken:
  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)
```

The same geometry gives zero violations under one rule set and thirty-two under the other, before you have drawn a single wire. You'll fix the thirty-two in Part 5.

You should see `DRC, default rule set 0` and `DRC, full rule set 32`.

Part 2: Check what is above each cell

Where can your metal1 go without hitting metal1 that is already inside a cell?

`and2_0` and `buf_1` contain metal1 only in their rails, but **`dfxtp_1` has metal1 inside the cell as well**, and your routing has to keep clear of it.

1. List the flip-flop's metal1:

```
awk '/<< metal1 >>/,/<< labels >>/' \
  $PDK_ROOT/sky130A/libs.ref/sky130_fd_sc_hd/mag/sky130_fd_sc_hd__dfxtp_1.mag
```

   If you add `| grep -c '^rect'` to the command, it counts the rectangles instead, and you get 74 for `dfxtp_1`. Run the same count on `sky130_fd_sc_hd__and2_0.mag` and `sky130_fd_sc_hd__buf_1.mag` and you get 16 and 12, all of them in the rails.

2. Find the flip-flop's two internal metal1 straps, at y = 1.415–1.645 and y = 1.755–1.985, which run most of the cell's width. The file is in units of 0.005 µm, so look for rectangles between y = 283 and 397.

3. The obvious place for the `D` input's via is at the D label, around y ≈ 1.19. Work out what a metal1 pad there would do next to the straps (`met1.2` is metal1 spacing, 0.14 µm), and where else on the D pin you could land instead.

   Answer: A1, in Answers at the end.

You should see the two straps in the listing, and a plan for where `D`'s via goes.

Part 3: Route the three nets

How do you get from each pin to the next without touching anything else?

net	from	to
`Q`	`dff0/Q` at x 6.950–7.120, y 0.425–0.595	`and1/A` at x 7.505–7.675, y 1.785–1.955
`n1`	`and1/X` at x 9.345–9.515, y 2.125–2.295	`and2/A` at x 9.805–9.975, y 1.785–1.955
`n2`	`and2/X` at x 11.645–11.815, y 2.125–2.295	`buf0/A` at x 12.105–12.275, y 1.105–1.275

Those coordinates come from the `flabel` lines in each cell's `.mag` file, with the x the cell was placed at added on (Routing by hand shows how).

For each net:

1. Paint an `mcon` on each pin, **exactly 0.17 µm square**, inside the li shape. For example, the `Q` net starts with:

```
box 6.950um 0.425um 7.120um 0.595um ; paint mcon
```

2. Paint a metal1 pad around each `mcon`. A pad 0.29 µm wide leaves 0.06 µm of metal1 on each side of the `mcon`, which satisfies `met1.5` on the x axis.
3. Join the two pads with metal1. Everything has to stay between **y = 0.38 and y = 2.34**, because the rails occupy y ≤ 0.24 and y ≥ 2.48, and metal1 spacing is 0.14 µm.
4. `save datapath`, then run `make layout`. If DRC reports a violation, use its rule name to find out what's wrong and fix it.

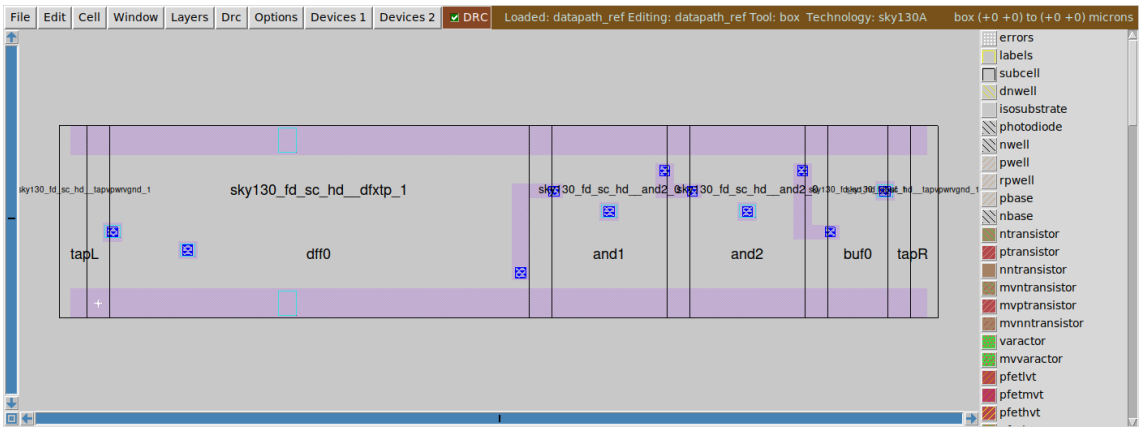


Figure 21.1: The finished row

With the subcells collapsed, each cell is drawn as an outline and your routing is the only real geometry on the screen. From left to right: `tapL`, `dff0`, `and1`, `and2`, `buf0`, `tapR`. You add the two taps in Part 5.

You should see `DRC, default rule set 0` after all three nets are routed. The full rule set will still report 32, and LVS will still fail, until you've done Parts 4 and 5.

Part 4: Bring out the pins

How does your block get pins of its own?

1. Bring out the block's own pins, `CLK`, `D`, `A`, `B` and `Y`. Each one is an `mcon` on a cell's li pin, with a metal1 pad over it and a label on the pad. The circuit at the top tells you which cell each pin belongs to: `CLK` and `D` go to the flip-flop, `A` and `B` to the free input of `and1` and `and2`, and `Y` comes from the buffer's output. Their coordinates are in each cell's `flabel` lines. Use the spot you chose in Part 2 for `D`.
2. Paint metal1 rails across the whole row and label them `VGND` and `VPWR`, so the supply labels sit on geometry that belongs to *this* cell. If you put the label on one of the standard cells' own rails instead, it would stay in your block but land on empty space, because the rail under it is the standard cell's, and a label on space doesn't name any net.
3. Turn the labels into ports. This is a separate step, and it's the one everybody skips:

```
select top cell
port makeall
save datapath
```

   A label is a name written on geometry, while a **port** is a name that the extractor puts on the `.subckt` line. `port makeall` promotes every top-cell label to a port. Without it, `make layout` reports `ports (none)` and LVS fails with `Top level cell failed pin matching`, and the printed advice about putting labels on the top cell's own geometry won't help, because yours already are.
4. Run `make layout`:

```
ports                  CLK D A B Y VGND VPWR
DRC, default rule set  0
DRC, full rule set     32
LVS                    Netlists do not match

rules broken:
  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)
```

You should see `ports` listing `CLK D A B Y VGND VPWR`, default DRC at 0, and the full rule set still at 32. LVS has moved on from `failed pin matching` to `Netlists do not match`: the block's pins are all there now, and the taps you add in Part 5 fix the rest.

Part 5: Add the tap cells

What are the 32 violations, and what fixes them?

All 32 violations are about **well taps**. Nothing in your row ties the n-well to VPWR or the substrate to VGND, because `sky130_fd_sc_hd` cells are drawn without taps on purpose, as you saw in Lab 01 on the foundry's own inverter.

1. The extracted netlist shows the same fault, although it never uses the word "tap". Look at it:

```
grep Xbuf0 results/datapath.lvs.spice
```

```
Xbuf0 buf0/A VGND VSUBS dff0/VPB VPWR Y sky130_fd_sc_hd__buf_1
```

`VSUBS` and `dff0/VPB` are invented node names, given to bodies that aren't connected to anything.

2. You fix this by placing a cell rather than by drawing a shape. Place a `sky130_fd_sc_hd__tapvpwrvgnd_1`, 0.46 µm wide, at each end of the row. In the console:

```
box -0.46um 0um -0.46um 0um ; getcell sky130_fd_sc_hd__tapvpwrvgnd_1 ; identify tapL
box 13.34um 0um 13.34um 0um ; getcell sky130_fd_sc_hd__tapvpwrvgnd_1 ; identify tapR
```

Two taps 14.26 µm apart are enough to satisfy `LU.2` and `LU.3`, because those rules only ask for a tap within 15 µm.

3. `save datapath` and run `make layout`. Before you run the `grep` from step 1 again, predict what the body terminals on the `Xbuf0` line will be now. Then run it; the same line now reads:

```
Xbuf0 buf0/A VGND VGND VPWR VPWR Y sky130_fd_sc_hd__buf_1
```

You should see 0 under both rule sets and `match` in LVS.

Part 6: Time it

How fast can you clock what you built?

1. Run the full check:

```
make
```

```
  YOUR BLOCK

      outline               14.64 x 3.20 um   =  46.85 um2
      cells placed          2 x and2_0, 1 x buf_1, 1 x dfxtp_1
      cells in the row      and2_0, buf_1, dfxtp_1, tapvpwrvgnd_1
```

```
    ports                  CLK D A B Y VGND VPWR
    DRC, default rule set  0
    DRC, full rule set     0
    LVS                    match

HOW FAST CAN YOU CLOCK IT

    CLK rise -> Y rise       536.17 ps    (measured on your extracted layout)
    D setup at the flop      103.32 ps    (sky130_fd_sc_hd__tt_025C_1v80.lib)
    shortest legal period    639.49 ps
    f_max                   1563.75 MHz
```

2. Match the lines to the timing rule:

$$T_{\text{clk}} \geq t_{\text{clk}\to Q} + t_{\text{logic}} + t_{\text{setup}} \tag{21.1}$$

   The first two terms are simulated on your own geometry, with every parasitic capacitance Magic could find. The third is looked up in `sky130_fd_sc_hd__tt_025C_1v80.lib`, because setup time is a property of the flip-flop's internals and you can't get it from your layout. As What sets the clock frequency points out, that is the same Liberty file the digital track synthesises against in DD103.

3. Divide your `CLK rise -> Y rise` by the foundry cell's `one stage` delay from Lab 05 (32.8 ps on that page), and write down how many inverter delays it is. Why do four gates add up to that many?

   Answer: A2, in Answers at the end.

You should see an `f_max` printed. The reference block gives 1563.75 MHz.

Part 7: Write the GDS

What do you hand to a foundry, and what is in it?

1. Write the file a foundry would accept:

```
make gds
```

   The reference block's GDS is **25,778 bytes**; yours will be within a few hundred bytes of that. Everything before this step (the `.mag` files, DRC, LVS, the extracted netlist) was for convincing yourself that the layout is right, and GDSII is the file you would actually hand over.

2. Look at it in KLayout:

```
klayout    -nn    $PDK_ROOT/sky130A/libs.tech/klayout/tech/sky130A.lyt    results/
datapath.gds
```

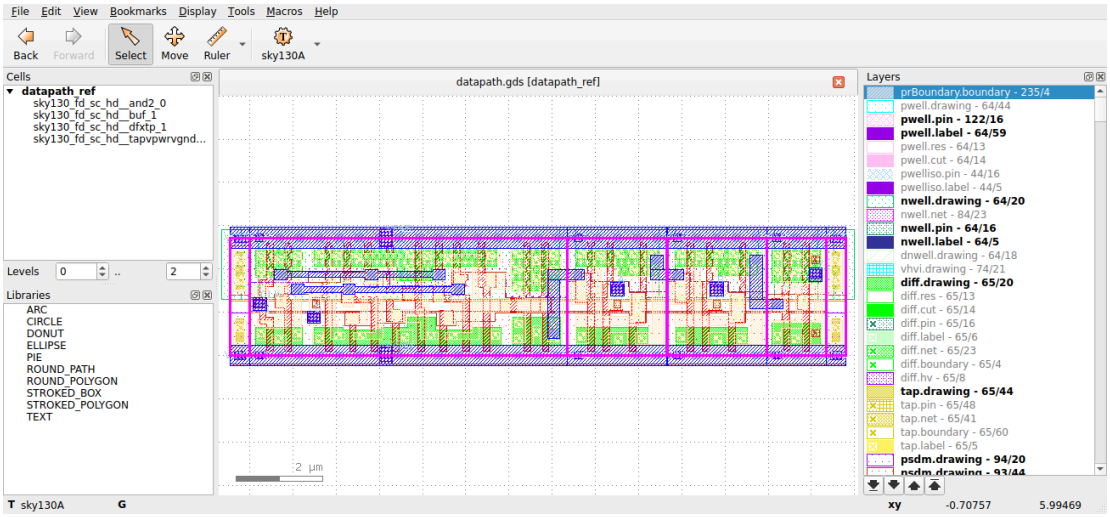


Figure 21.2: The GDS

3. In the cell tree, notice that the library cells are still separate cells, because the GDS is hierarchical. Magic didn't regenerate their geometry, either. It copied the bytes from the foundry's own GDS file, which is what the `GDS_START` and `GDS_END` properties in their `.mag` files are for.

You should see `results/datapath.gds`, and the library cells listed separately in KLayout's cell tree.

Finishing

You're done when `make` on your own routed block reports 0 under both rule sets, `match` in LVS and an `f_max`, and `make gds` has written `results/datapath.gds`.

Questions:

- Why did the default rule set report 0 while the full one found 32 violations?

  Answer: A3, in Answers at the end.

- Name three reasons a real chip runs at a fraction of your 1.56 GHz.

  Answer: A4, in Answers at the end.

Submit: nothing for this lab.

Going further (optional)

`make ref` and `make refcheck` build and check the reference block. `tcl/route_ref.tcl` routes it with a comment on every coordinate; the package's `solutions/README.md` discusses five of the decisions in it. They're worth reading once your own block passes.

You have now done by hand, for four cells, what DD103 does with LibreLane for a whole design in about a minute: place cells in rows, insert taps, route, check DRC, check LVS, and write GDS. The tool repeats the same steps a hundred thousand times, guided by a cost function, and it never forgets the taps.

Next on the Course path: this is the last lab in AD104. The home page lists the next courses.

Answers

A1. Part 2: Check what is above each cell

A pad at the label would sit 0.07 µm from a strap, which breaks `met1.2`. The D pin's li shape runs all the way down to y = 0.715, so you can take the via down there instead. Finding somewhere else on the same pin to land is the routing move you'll make most often.

A2. Part 6: Time it

With the numbers on these pages, 536 ps is 16 inverter delays. Most of it is the flip-flop's clock-to-Q, and the `and2_0` cells you were given are the weakest AND in the library. Swapping in `and2_2` would win back some of that time, at the cost of more area.

A3. Finishing

All 32 break the three well-tap rules (`nwell.4`, `LU.2`, `LU.3`), and the default rule set leaves those rules out; only `drc(full)` checks them (The full DRC rule set). The cells are drawn without taps on purpose, so the row needed tap cells, which you added in Part 5.

A4. Finishing

Any three of: logic depth (a real critical path is twenty to forty levels, not four), wires (delay grows roughly with the square of length, and real nets are hundreds of microns to millimetres long), clock skew and jitter, and margin (sign-off is at the slow-slow corner, 125 °C and low supply, where everything is 30–50 % worse). What sets the clock frequency works through each one.

Appendices

A. Magic cheat sheet

This page lists every Magic command the course uses. You can type any of them into the `tkcon` console, and where a table has a keyboard column, that column gives the key or mouse action that does the same thing in the layout window.

Starting

```
cd <a directory with a .magicrc in it>
magic -d X11 <cellname>          # opens layout1 + tkcon
magic -dnull -noconsole script.tcl   # batch, which is what the labs do
```

`-d X11` selects the X11 display driver, which is the one that works in this container. The `-d XR` driver can fail with a `BadAlloc` error and leave you looking at a black window, so stick with X11.

The image's `~/.magicrc` loads whichever PDK `$PDK` names. Magic also reads a `.magicrc` from the directory you start it in, which is why you `cd` first. Each lab package has one of these, and it loads SKY130 only if nothing has loaded it yet:

```
if {[tech name] eq "sky130A"} {
    # Already loaded by ~/.magicrc. Nothing to do.
} elseif {[info exists env(PDK_ROOT)]} {
    source $env(PDK_ROOT)/sky130A/libs.tech/magic/sky130A.magicrc
} else {
    puts stderr "PDK_ROOT is not set. Try:  export PDK_ROOT=/foss/pdks"
}
```

The box

Console	Keyboard / mouse	
`box <x1>um <y1>um <x2>um <y2>um`	left-click, then right-click	set the box
`box values`		read it back, **in units of 0.005 µm**
`box width`, `box height`		read the box's width or height
`findbox zoom`		zoom the view to the box
	`z` / `Z`	zoom in / out
	`v`	view everything

Always write `um`. Magic reads a number with no unit as lambda, which is 0.01 µm here, and it doesn't warn you when it does.

Painting

Console	Keyboard	
`paint <layer>`	middle-click on a layer swatch	fill the box
`erase <layer>`	middle-click on empty space	clear the box

Console	Keyboard	
`erase`		clear every layer in the box
`copy` / `move`		copy or move the current selection
`select`	`s`	select what is under the cursor
`select area`	`a`	select everything in the box
`select top cell`		select the whole cell
`undo` / `redo`	`u` / `U`	
`tech layers`		print every layer name and alias

Layers used in this course

```
nwell pwell             ndiff pdiff             ntap ptap
ndc pdc                 ntapc ptapc             pc
poly                    li                      mcon
m1 m2                   via1                    pfethvt
```

`ndc`, `pdc`, `pc`, `ntapc`, `ptapc` and `mcon` are **contact types**, which means that painting one fills the whole stack (the lower layer, the cut and the upper layer) in one step. For a single contact, the cut has to be exactly 0.17 µm square.

Labels and ports

```
box 1.30um 2.40um 1.55um 2.65um
label A c li            ;# name the geometry under the box, on layer li
select top cell
port makeall            ;# turn every label in the cell into a terminal
port <name> index <n>   ;# set a port's number, if order matters to you
```

A label always goes into the cell you're editing, and it names a net only if it sits on geometry **belonging to that cell**. If you place one over a shape that belongs to a subcell, Magic moves it onto empty space (`Moving label "A" from locali to space`), where it names nothing, so paint a small shape of your own over the subcell's and label that.

DRC

```
drc check               ;# re-run
drc catchup             ;# wait for it to finish (batch scripts need this)
drc count total         ;# how many violations: the number you want
drc count               ;# careful: prints "Cell X has 3 error tiles." and then
                        ;#   "Total DRC errors found: 0" - that second line is a
                        ;#   hierarchical total and reads 0 on a flat cell
drc why                 ;# name the rules broken under the box (set one first:
                        ;#   with no box, 'select top cell')
drc listall why         ;# every violation in the cell, with rule text
drc euclidean on        ;# measure corners properly rather than as squares
drc style drc(full)     ;# the rule set that catches missing well taps
drc style drc(fast)     ;# back to the default
```

The `Drc` menu on the layout window does the same things with the mouse:

Menu	Same as
Drc → DRC update	`drc check`. Use it when a cell you've just opened shows a green ☑ `DRC` and no hatching.
Drc → DRC report	`drc listall why`. It doesn't need a cursor box.
Drc → DRC Find next error	Puts the box on the next violating tile *and zooms to it*, which makes it the fastest way to find a hatched area you can't see at the current zoom. It has no console equivalent worth typing.
Drc → DRC Complete / DRC Fast	`drc style drc(full)` / `drc style drc(fast)`

Magic doesn't check a cell you have only opened. Its background checker re-checks the tiles you *edit*, and loading a `.mag` from disk doesn't edit anything, so a cell with real violations opens showing a green ☑ `DRC` and no white hatching. Run `drc check` (or `Drc → DRC update`) every time you open a cell, before you believe what the DRC indicator says.

Extraction

```
extract do local       ;# this cell
extract all            ;# and every subcell
ext2spice lvs          ;# preset: devices only, hierarchical, for netgen
ext2spice -o out.spice
ext2spice cthresh 0    ;# keep every parasitic capacitance
ext2spice cthresh infinite   ;# back to none
```

Hierarchy

```
addpath ../mag         ;# where to find cells (set it AFTER any cd)
box 3um 0um 3um 0um
getcell my_inv         ;# place an instance with its origin at the box
identify inv2          ;# name this instance
expand                 ;# draw subcells properly instead of as outlines
flatten -nolabels flat ;# copy the whole hierarchy into one cell
cellname list children ;# what is instanced in this cell
```

Files

```
load <cell>            ;# read cell.mag and make it the edit cell
load <cell> -force     ;# start a new empty cell with that name
save <cell>            ;# write cell.mag
cd <dir>               ;# Magic has its own working directory
gds write out.gds      ;# the file a foundry would take
property FIXED_BBOX    ;# a cell's placement outline, if it has one
```

Batch scripts

```
cd results
addpath ../mag
load my_inv
select top cell
drc euclidean on
drc check ; drc catchup
```

```
puts "DRC [drc list count total]"
extract do local ; extract all
ext2spice lvs
ext2spice -o my_inv.spice
quit -noprompt
```

The lab scripts follow this pattern, and you run one with `magic -dnull -noconsole thatfile.tcl`. Put `cd` **before** `addpath`, because `addpath` stores the path exactly as you typed it and doesn't re-resolve it, so a relative path like `../mag` given before a `cd` points at the wrong place afterwards.

netgen (LVS)

```
netgen -batch lvs \
  "layout.spice cellname" \
  "schematic.spice cellname" \
  "$PDK_ROOT/sky130A/libs.tech/netgen/sky130A_setup.tcl" \
  report.txt
```

Look for `Circuits match uniquely.`, which means the layout and the schematic agree. If you get anything else, read the report from the bottom up.

KLayout (viewing)

```
klayout -nn $PDK_ROOT/sky130A/libs.tech/klayout/tech/sky130A.lyt out.gds
```

`-nn` loads the SKY130 technology, so the layers are named and coloured properly.

B. DRC error decoder

This page lists every DRC rule you can run into in this course, with what each one protects and what to change in your layout when Magic reports it. The values come from `$PDK_ROOT/sky130A/libs.tech/magic/sky130A.tech` on the pinned workbench image, so you can `grep` for any rule name in that file and read the line yourself.

Several messages include the words **"in one direction"**. That means you pick an axis, x or y, and on that axis the surrounding layer has to extend past the inner one by at least the given amount on **both** sides. The other axis doesn't matter, so a shape that is generous vertically and flush horizontally still passes.

Contacts (`licon`)

Message	Value	What to do
`... contact width < 0.17um (licon.1)`	0.17	A single cut is **exactly** 0.17 × 0.17 µm, so make the painted contact region that size, or a multiple of it with gaps between the cuts.
`Diffusion contact spacing < 0.17um (licon.2)`	0.17	Two cuts are too close together. Move one of them.
`N-diffusion overlap of N-diffusion contact < 0.06um in one direction (licon.5c)`	0.06	The plain diffusion has to extend past the contact. Widen the diffusion, or move the contact inward.
`N-tap overlap of N-tap contact < 0.12um in one direction (licon.7)`	0.12	This is the same rule for taps, and the margin is twice as big.
`poly overlap of poly contact < 0.05um (licon.8)`	0.05	You've landed a contact on a bare gate stripe. A 0.15 µm poly stripe is too narrow to hold a 0.17 µm cut, so draw a wider landing pad for the contact.
`poly overlap of poly contact < 0.08um in one direction (licon.8a)`	0.08	This is the directional half of the same rule, and it usually fires together with `licon.8`.
`Diffusion contact to standard cell gate < 0.05um (licon.11)`	0.05	A source or drain contact is too close to the poly. Move it away from the gate.
`poly contact spacing to diffusion < 0.19um (licon.14)`	0.19	The contact on the poly landing pad is too near the diffusion. Move the pad out into the gap between the two diffusions.
`All taps must be contacted (licon.16)`	—	**Full rule set only**, and on the pinned image it never fires (see Full rule set only below). A tap with no contact on it isn't con-

Message	Value	What to do
		nected to anything, so it does nothing.

Local interconnect (`li`)

Message	Value	What to do
`Local interconnect width < 0.17um (li.1)`	0.17	The wire is thinner than the process can print. Widen it.
`Local interconnect spacing < 0.17um (li.3)`	0.17	Two li shapes on different nets are too close together. Move one of them, or make it narrower.
`Local interconnect overlap of ... contact < 0.08um in one direction (li.5)`	0.08	The li has to overhang the contact under it. Paint the li **first**, a bit bigger than you need, and then paint the contact inside it.
`Local interconnect minimum area < 0.0561um^2 (li.6)`	0.0561	The patch is too small to survive processing. The smallest legal rectangle is 0.17 × 0.33 µm.

Metal1 and its via (`met1`, `mcon`, `via`)

Message	Value	What to do
`mcon.width < 0.17um (mcon.1)`	0.17	An mcon is exactly 0.17 µm square, like every other cut.
`mcon.spacing < 0.19um (mcon.2)`	0.19	Two mcons are too close together.
`Metal1 width < 0.14um (met1.1)`	0.14	Widen the wire.
`Metal1 spacing < 0.14um (met1.2)`	0.14	**This is the most common routing error.** It's usually a signal wire too near a supply rail. In the capstone's row of foundry cells the legal band between the rails runs from y = 0.38 to 2.34 µm.
`Metal1 overlap of local interconnect contact < 0.03um (met1.4)`	0.03	The metal1 pad has to cover the mcon on every side.
`Metal1 overlap of local interconnect contact < 0.06um in one direction (met1.5)`	0.06	On top of `met1.4`, the pad has to extend 0.06 µm past the mcon on both sides of one axis. A 0.29 µm pad around a 0.17 µm cut does it.
`Metal1 minimum area < 0.083um^2 (met1.6)`	0.083	The pad meets the enclosure rules but is still too small overall. A 0.29 × 0.29 µm pad is 0.0841 µm², which is just enough.

Message	Value	What to do
`Via1 width < 0.26um (via.1a + 2 * via.4a)`	0.26	A drawn via1 tile has to be at least 0.26 µm wide, which makes it bigger than an mcon.
`Metal2 width < 0.14um (met2.1)` / `spacing (met2.2)`	0.14	Metal2 has the same width and spacing as metal1.

Diffusion and poly

Message	Value	What to do
`Diffusion width < 0.15um (diff/tap.1)`	0.15	Your W is below the minimum.
`Diffusion spacing < 0.27um (diff/tap.3)`	0.27	Two diffusion strips are too close together. This is the rule that decides how tall a cell has to be.
`poly width < 0.15um (poly.1a)`	0.15	Your L is below the minimum, which in SKY130 is 0.15 µm.
`poly spacing < 0.21um (poly.2)`	0.21	Two gates are too close together. This rule sets the pitch of a series stack.
`poly spacing to Diffusion < 0.075um (poly.4)`	0.075	A poly landing pad is crowding a diffusion that it isn't meant to be a gate for.
`N-Diffusion overhang of nFET < 0.25um (poly.7)`	0.25	The diffusion has to extend past the gate on both sides, because that is where the source and drain are.
`poly overhang of transistor < 0.13um (poly.8)`	0.13	The gate stripe has to run past the edge of the diffusion, or the channel leaks round the end of the gate.

Wells and taps

Message	Value	What to do
`N-well width < 0.84um (nwell.1)`	0.84	
`N-well spacing < 1.27um (nwell.2a)`	1.27	Two separate wells have to be far apart. Wells that touch are fine, because they merge into one.
`N-well overlap of P-Diffusion < 0.18um (diff/tap.8)`	0.18	The well has to extend past the PMOS diffusion all the way round.
`N-Diffusion spacing to N-well < 0.34um (diff/tap.9)`	0.34	Your NMOS is too close to the well above it.

Message	Value	What to do
`P-tap spacing to N-well < 0.13um (diff/tap.11)`	0.13	
`N-tap minimum area < 0.07011um^2 (nsd.10b)`	0.07011	The tap strip is too small to be worth making.

Full rule set only

The default DRC run doesn't check these four rules. The full DRC rule set explains why they matter and how to run them.

Message	What to do
`All nwells must contain metal-connected N+ taps (nwell.4)`	Your n-well is floating. Connect it to VPWR with an `ntap` + `ntapc` + li + mcon, or put a `tapvpwrvgnd` cell in the row.
`All taps must be contacted (licon.16)`	This rule is meant for a tap diffusion with no contact on it, but on the pinned image it never fires, even on exactly that mistake. Lab 02's `drc_05` has an n-tap with no contact, and the full rule set reports `nwell.4` and `LU.3` for it instead. If you see those two, check that every tap has its contact.
`N-diff distance to P-tap must be < 15.0um (LU.2)`	This rule guards against latch-up. Every NMOS needs a substrate tap somewhere within 15 µm of it.
`P-diff distance to N-tap must be < 15.0um (LU.3)`	This is the same latch-up rule for the other polarity.

The foundry's own standard cells fail three of these rules when you check a single cell on its own. That's on purpose, because in a real row the taps come from a separate cell.

Looking up a rule yourself

```
grep -n 'met1.5' $PDK_ROOT/sky130A/libs.tech/magic/sky130A.tech
```

The numbers in that file are in nanometres, so `140` means 0.14 µm. The final authority is SkyWater's published rule deck. Magic's tech file is one implementation of that deck, and occasionally it's more conservative than the deck itself.

C. SKY130 layer map

This page lists the layers this course uses, from the bottom of the stack to the top. For each one it gives the name you type in Magic, the number the GDS file uses, and what the layer physically is.

The GDS numbers are written as `layer/datatype`. They come from the `calma` lines in `$PDK_ROOT/sky130A/libs.tech/magic/sky130A.tech`, and they're the numbers you see in KLayout's layer panel. The sheet resistances are the `resist` lines in the same file, converted from milliohms to ohms per square.

Magic	GDS	Physically	Sheet R
`nwell`	64/20	an n-type tub in the p-substrate	950 Ω/□
`pwell`	64/44 (marker)	the p-substrate itself. SKY130 has no p-well mask, so Magic draws a p-well only as a marker that lets it reason about the body, and the marker doesn't become a real layer.	4400 Ω/□
`ndiff`	65/20 + 93/44	n+ silicon at the surface	120 Ω/□
`pdiff`	65/20 + 94/20	p+ silicon at the surface	197 Ω/□
`ntap`	65/44 + 93/44	n+ silicon **inside the n-well**: a handle on the well	120 Ω/□
`ptap`	65/44 + 94/20	p+ silicon in the substrate: a handle on the substrate	197 Ω/□
`poly`	66/20	polysilicon on the thin gate oxide	48.2 Ω/□
`pfethvt`	78/44	the HVTP implant, which makes a PMOS high-threshold	—
`pc` (poly contact)	66/44 + 95/20	a tungsten plug from poly to li, plus the NPC blocking mask	—
`ndc` / `pdc` / `ntapc` / `ptapc`	66/44	the same plug, from silicon to li	—
`li`	67/20	local interconnect	**12.8 Ω/□**
`mcon`	67/44	plug from li to metal1	—
`m1`	68/20	metal1	**0.125 Ω/□**
`via1`	68/44	plug from metal1 to metal2	—
`m2`	69/20	metal2	0.125 Ω/□

Magic	GDS	Physically	Sheet R
via2 / m3	69/44 / 70/20	and upward, to metal5	

Reading the table

Diffusion and tap are the same GDS layer. diff is 65/20 and tap is 65/44, so they share one mask and differ only in datatype. Whether a piece of either one is n-type or p-type depends on which implant mask (nsdm 93/44 or psdm 94/20) covers it. Physically, a ptap and a pdiff are identical, and they differ only in where they sit: the tap in the substrate, the diffusion in an n-well. That's why the tap rules and the diffusion rules look so similar.

li is a hundred times more resistive than metal1. Its sheet resistance is 12.8 Ω/□, against 0.125 Ω/□ for metal1. A 10 µm li wire 0.17 µm wide is 59 squares long, so it has about **750 Ω** of resistance, while the same wire in metal1 has 7 Ω. That's why li is called *local* interconnect: it gets you from a transistor to the wire, and it isn't meant to be used as the wire itself.

Poly is a poor conductor. Its sheet resistance is 48.2 Ω/□. Poly is the gate, and you can use it as a short wire inside a cell, but it is never used for routing.

The contact ladder

There is exactly one legal route from the silicon up to the metal. The ladder below uses the PDK's layer names (licon1, li1, met1) rather than Magic's, with the GDS numbers in parentheses:

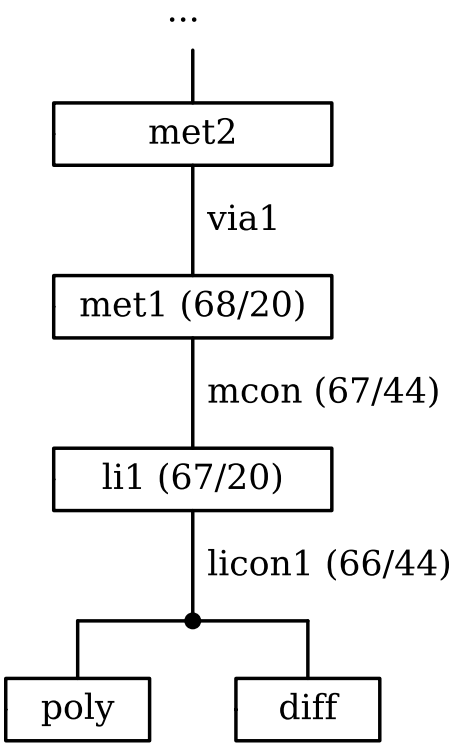


Figure C.1: The contact ladder, with each layer's GDS number in parentheses.

That means you can't connect poly straight to metal1, or diffusion straight to metal2; you go up one step at a time. Every step is a plug of a fixed size, with enclosure rules at both ends.

Implant masks

`nsdm` (93/44) and `psdm` (94/20) decide whether a piece of active silicon is n+ or p+. In Magic you never paint them directly. You paint `ndiff` or `ptap`, and Magic generates the right implant when it writes the GDS. You'll still see them named in DRC messages, though:

```
NSDM width < 0.40um (diff/tap.5)
```

They also appear in the `MASKHINTS_NSDM` properties of the foundry's cells, which is where the library records hand-tuned implant shapes that Magic's automatic generation would not have produced.

The one implant you *do* paint is `hvtp` (78/44), as `pfethvt`, because nothing else in the layout says which threshold flavour you want for a PMOS.

Viewing layers in KLayout

```
klayout -nn $PDK_ROOT/sky130A/libs.tech/klayout/tech/sky130A.lyt yourfile.gds
```

The layer panel on the right lists every layer in the file with its numbers, and clicking a layer's eye icon hides it. The fastest way to see the standard-cell rows underneath a routed block is to hide everything above `met1`.

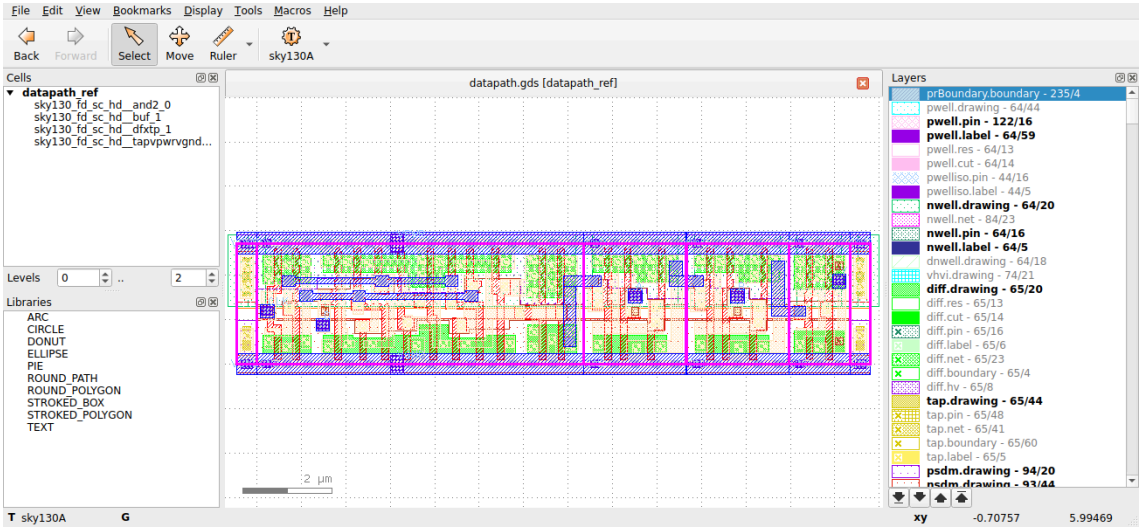


Figure C.2: The capstone block as GDS

The capstone's GDS in KLayout. The cell tree on the left has `datapath_ref` at the top, with `and2_0`, `buf_1`, `dfxtp_1` and the tap cell under it, and the layer list on the right shows the numbers from the table above.