Getting started

AD102 uses two tools from the workbench you set up in IC101. **ngspice**, the open-source descendant of Berkeley SPICE, simulates a circuit and tells you what it would do, and **XSchem** is the schematic editor you use to draw circuits for it. Both are already in the image, so you won't install anything on this page. Instead it explains how the pieces fit together, goes through the warnings you'll see every day, and describes the one mistake that costs students the most time. You'll do the setup itself in Lab 01, Part 1.

What this course assumes

Requirement	Notes
IC101 completed	The workbench comes up in your browser and the smoke test passes
AD101 completed	You can read a waveform, a spectrum, and a Bode plot
ECE110-level circuit analysis	Helpful, not required: $V = IR$, series and parallel, and what an RC corner is. AD101 covers the last one.

AD102 doesn't re-teach linear circuit analysis; node equations, Thévenin equivalents and phasors belong to ECE110. Instead this course asks two questions that ECE110 doesn't: where did the component come from, and what did it cost?

The PDK setting

A **Process Design Kit (PDK)** is the foundry's description of what a process can build: the device models, the schematic symbols, and the drawing rules. Think of it as the parts catalogue for a chip.

The workbench image can simulate more than one process, and the environment variable `PDK` chooses which one. The image's default is `ihp-sg13g2`, a real 130 nm process from IHP in Germany. It's a perfectly good process, but we measure SKY130 (`sky130A`) in AD102, so that's the one you need.

The workspace repo from IC101 has a setup script, `/foss/designs/common/.designinit`, that switches you to SKY130. You load it with a leading `.`:

```
. /foss/designs/common/.designinit
```

The `.` is the shell's `source` command. It runs the file in your current shell rather than in a child process, and that's the only way a script can change your shell's environment. If you type the path without the dot and the space, the file doesn't run in your current shell, so the command does nothing useful. The catch with sourcing is that the change lasts only as long as the shell it was made in, so every new terminal needs it again. On the IC101 workbench the image does this for you whenever a terminal opens (the `workspace` repo's own `.designinit` chains to this file), so you'll usually find `PDK` already set to `sky130A`. Loading it a second time does no harm.

When XSchem starts, it reads `$PDK` to decide which symbol library to load. Before you load `.designinit`, XSchem takes its settings from `~/.xschem/xschemrc`, which does this:

```
# Source the PDK xschemrc file
source $env(PDK_ROOT)/$env(PDK)/libs.tech/xschem/xschemrc
```

After `.designinit`, XSchem uses the workspace's own `common/xschemrc` instead, and that file reads `$PDK` too. Either way, it's the `PDK` variable that picks the parts catalogue. On the default you get a library called `sg13g2_pr`,

and after `.designinit` you get `sky130_fd_pr`. (`_pr` is the foundry's shorthand for **primitive devices**: the transistors, resistors and capacitors themselves, as opposed to logic gates built from them.) If you launch XSchem on the default, you get a schematic editor full of `sg13g2_*` devices with no SKY130 resistor anywhere, and no error message either, because nothing is broken; you've simply opened a different foundry's catalogue.

So make a habit of running `echo $PDK` before you launch XSchem. It should print `sky130A`.

The lab decks don't depend on the PDK setting, though XSchem does (see How the labs work below). You still need `.designinit` to get the course files, because it also defines `mod`.

Where your files live

`mod`, the shell function that `.designinit` defines, has two forms: `mod add ad102` clones this course into `/foss/designs/modules/ad102`, and `mod ad102` takes you there afterwards. The folder `/foss/designs` inside the container is really your `workspace` clone on your own computer, shared into the container (a bind mount). That means the course files are in `workspace/modules/ad102` on your disk, and deleting the container doesn't delete your work.

The `u` suffix trap

This mistake produces no error message, so we describe it here, before you meet it in Lab 01.

In an ngspice deck, most numbers take SI suffixes: `1k` is a thousand ohms, `0.7p` is 0.7 picofarads and `3n` is three nanoseconds. It therefore seems reasonable to write a device one micron wide as `W=1u`.

SKY130 device widths and lengths don't work that way, because in this PDK `W` and `L` are already plain numbers in microns. `W=1` means one micron, and `W=1u` means one micron × 10^{-6}, which is smaller than an atom.

Here are the two ways of writing a SKY130 high-sheet poly resistor that is one micron wide and ten long. This is an excerpt, not a file to save; the complete deck ships as `labs/lab-01-a-resistor-you-designed/spice/u_trap.spice`.

```
Xa a 0 0 sky130_fd_pr__res_high_po W=1  L=10
Xb b 0 0 sky130_fd_pr__res_high_po W=1u L=10u
```

The deck pushes exactly 1 µA through each one and reads the resistance as V/I. The right one gives 3550.443 Ω and the wrong one gives 3193.812 Ω. The wrong one doesn't blow up, and it doesn't return zero, infinity or a negative number that you'd notice. Instead it returns an ordinary resistance 10 % away from the right answer, which is the size of error you'd blame on a model, a corner, or yourself.

The only sign of the mistake is one line in the log, buried in the middle of the run:

```
Warning: r.xb.rbody: resistance too low or not given, set to 1 mOhm
```

Even so, ngspice exits with status 0. You'll run this deck yourself in Lab 01, Part 8.

Transistors fail more loudly. The same mistake on a MOSFET, which you'll meet again in AD103, stops the simulator:

```
could not find a valid modelname
    Simulation interrupted due to error!

Error: incomplete or empty netlist
```

```
       or no ".plot", ".print", or ".fourier" lines in batch mode;
no simulations run!
```

A foundry's model table is split into **bins**: each row is valid over a stated range of widths and lengths. A device whose geometry nobody characterised matches no bin, so ngspice rejects it. That error is the better of the two outcomes, because you can't miss it, whereas the resistor's 3193.812 Ω arrives quietly.

In any SKY130 deck, `W`, `L`, `w` and `l` carry no unit suffix. If a resistance is close but not right, look for a stray `u` before you look anywhere else.

Expected warnings

Whenever you run a deck that contains a SKY130 poly resistor, ngspice prints this before it has done a single calculation:

```
Warning: Model issue on line 4842 :
  .model xa:rhead_model r sw_et=0 isnoisy=0 rsh=    3.458312000000000e+02  ...
unrecognized parameter (sw_et) - ignored
unrecognized parameter (isnoisy) - ignored
unrecognized parameter (p2) - ignored
unrecognized parameter (q2) - ignored

Warning: Model issue on line 4843 :
  .model xa:rbody_model r sw_et=0 isnoisy=0 rsh=    3.173885000000000e+02  ...
unrecognized parameter (sw_et) - ignored
unrecognized parameter (isnoisy) - ignored
unrecognized parameter (p2) - ignored
unrecognized parameter (q2) - ignored
unrecognized parameter (p3) - ignored
unrecognized parameter (q3) - ignored
```

That's ten `unrecognized parameter` lines and two `Warning:` headers for one resistor. A deck with many resistors repeats the block (Lab 01′s 22-strip deck prints 170 such lines), and none of it means anything is wrong. SkyWater's resistor model cards carry parameters for noise and self-heating that this build of ngspice doesn't implement, so ngspice names each one, skips it and carries on, and your resistance is unaffected.

The block does tell you something useful, though: each resistor has two models. There's a `rhead_model` at `rsh = 345.8312` Ω per square and a `rbody_model` at `rsh = 317.3885` Ω per square. ("Ohms per square" is the unit chip designers use for a material's resistance; Ohms per square explains it.) The ends of a drawn resistor aren't made of the same material as the middle, which is why one drawn square of this device measures 695.4646 Ω rather than the 317.3885 Ω on the card. Explaining that gap is the first job in Lab 01.

A genuine ngspice failure starts a line with `Error:`, and the run ends with `no simulations run!`. That's how you tell a real failure from `Warning:` lines like the ones above, which you can ignore.

How the labs work

Every AD102 lab is a package with a `Makefile`. You go to the lab folder and type `make`, and it ends by printing a verdict: a line that says whether what you built matches what the lab expected, so you don't have to pick through a wall of numbers yourself. `make` runs everything, and each lab also has smaller targets, such as `make mine` (simulate only the file you edit) and `make check` (print the verdict again from the logs already written). Each package's README lists them.

All four lab packages run in a bare container with no environment setup at all: no `.designinit`, no `mod`, and `$PDK` still set to `ihp-sg13g2`. That's because every deck names its SKY130 model file by absolute path, so the PDK setting described above can't affect them. We built them that way on purpose, so if your setup went wrong, run `make` anyway and it will still work.

The one exception is an optional GUI step. `make edit` in Lab 03 opens the RC circuit in XSchem, and XSchem does read `$PDK` from your environment, so that step needs `.designinit` to have worked. The `make` that produces Lab 03's verdict doesn't need it.

Why the first `make` fails

Every AD102 lab package ships with the design a first-year student would draw, usually the naive one sized straight off the model card. The checker rejects it on purpose, and says so. This is Lab 01's message:

```
^ On a fresh clone this FAIL is the lab, not a broken
  package. spice/my_resistor.spice ships with the naive
  sizing on purpose. Edit the two L= numbers and run
  `make mine check` again.
```

Your job is to read your own wrong number, work out where the missing ohms went, and fix the two lengths that were wrong. `make` exits non-zero on a `FAIL`, so at this point a red exit code is what you should expect, and it doesn't mean your clone is broken.

SiliWiz

Part of this course is looking at a **cross-section**, which shows what a resistor or capacitor looks like sliced through, layer by layer. SiliWiz draws cross-sections in a browser, with nothing to install. You draw shapes on a simplified SKY130 layer stack, and it shows you the silicon underneath and simulates it. You'll use it in optional parts of Labs 01 and 02 and in the capstone.

Launch GUI tools from a terminal

These tools print their error messages to the terminal you launched them from, rather than in their own window, so launch GUI tools by typing their name in a terminal, never from a desktop icon. If you start XSchem from the applications menu there is no such terminal, so when something goes wrong the explanation is lost, and the program appears to do nothing at all.

Getting help

The course is self-paced, but that doesn't mean you have to work through it alone.

- Ask in the team Discord. No question is too basic, and most of what looks like your own mistake here is really a tool being unhelpful.
- Quote the exact error and the exit code. ngspice's first `Error:` line names the real problem, and everything after it follows from it. Paste that line, along with the command you ran and the output of `ngspice -v | head -2`.
- Say what the number was. Analog debugging is mostly comparing a number you got against a number you expected. "The filter is wrong" is hard to help with, while "I expected 3550.443 Ω and got 3193.812 Ω" can be answered in one reply.
- If a command on these pages doesn't do what the page says, that's a bug in the course, and we'd like you to report it. We ran every command here before writing it down, so a mismatch means something has changed, and we want to know.

Next: Resistors on a chip.