OpenPCells

User Guide

Patrick Kurth

December 2, 2022

This is the official documentation of the OpenPCells project. It is split in several different files for clarity. This document is the general userguide for everyday usage, installation and related topics. It also includes a quick start for the impatient, which is situated at the beginning of the document, *before* the introduction. If you are looking for a guide and documentation on the creation of parametrized cells, consult the celldesign manual. If you want to now more about the technical details and implementation notes, look into the technical documentation.

Contents

1 Quick Start

If you did not do this yet, clone the repository:

```
> git clone https://github.com/patrickschulz/openPCells
```

Go to the repository and create the main program:

```
> make
```

The project supplies a lua interpreter (the official code with some modifications) that simplify compatibility problems and path setup issues.

Now run the program:

```
> ./opc
```

You should see the following message:

```
This is the openPCell layout generator.
To generate a layout, you need to pass the technology, the export
   type and a cellname.
Example:
         opc --technology opc --export gds --cell stdcells/not_gate

You can find out more about the available command line options by
   running 'opc -h'.
```

This indicates that everything is working fine. Due to the non-standard interpreter, the paths for the toolchain are already set. You should be able to call `opc` from any path (of course by using the full name including path or modifying your shell PATH variable accordingly).

Now call the program from a different location with all the needed data: technology, export and cell. Since technology files are usually under non-dislosure agreements, openPCells comes with a generic technology (opc) which can be used to test the setup. However, this is a non-physical technology, so for real work you will have to write technology files yourself, which is not too hard. See the guide about writing technology files. Let's call the program:

```
opc --technology opc --export gds --cell basic/mosfet
```

or

```
opc --technology opc --export SKILL --cell basic/mosfet
```

The gds export creates a `openPCells.gds` file, the SKILL export a `openPCells.il`, which can be read with a text editor. Further current available export are SVG, magic, tikz and a debugging export.

This already concludes the quick start. For next steps, have a look in `cells/` to see how the pcells are implemented. Furthermore, `tech/` contains the technology files: opc, skywater130, freePDK45 and a template (which is a little bit outdated, take opc as template). The documentation is split in several files, this document is a general user guide containing broad information about "everything". For a detailed guide on how to write cell generators, see celldesign.pdf. The procedure of writing technology files is described in techfiles.pdf. For developers, techdoc.pdf gives an insight on the implementation, especially on the technology translation.

2 Introduction

Semiconductor technologies from the perspective of a designer of integrated circuits is a set of layers, in which shapes are created to make up the physical implementation of integrated circuits, the so-called layout. It is a long and complex way to get to a final layout that can be sent to the foundry: The designer has to start with an abstract schematic, check for all sorts of mistakes, processing mismatches, outer influences and so on and finally create the layout from the schematic. The last activity for itself is quite complex and can take a lot of time. Designers have to place individual cells such as transistors or passive components and connect them

with interconnection metals. Then the layout has to be checked against design rules and for correctness (layout versus schematic). The foundry supports designers by offering a process design kit (PDK), which includes parameterized cells (pcells). Sometimes the pcells have not the needed flexibility or certain types are not available (often an issue for RF designs with lots of passive components), so that designers have to built them themselves. Beside a static solution, pcells can of course also be built by designers, however, this can get quite complicated with a lot of edge cases. Furthermore, there are basically two ways of implementing pcells: in cadence virtuoso by using SKILL (a proprietary language based on lisp) or in python with pycells, developed by ciranova (now synopsys). Cadence pcells are quite closed, since you can only really access them with virtuoso and they are bound to one technolgy. Pycells are more open as they are developed in python, however, you still have to register with synopsys to access the tools and the cells are still bound to one technology. Furthermore, both techniques are linked to open access, the cell system IC designs are developed in nowadays. Despite the name, open access is not really open. You have to be part of the consortium to access the standard.

In general, pcells and pycells function the same way: They are implementations of the open access abstract pcell interface, which is a good idea in theory, but has some problems in practice. In my opinion, designers don't need pcells, that is, cells that can be parameterized even after they have been placed but "pgens" (parameterized cell generators, although I won't name it like that): programmble generators that can create certain cells. If you need to change anything later on, you have to re-generate it. To understand why this is beneficial you have to now something about IC layout and creating tape-outs. In short: Pcells (as opposed to "pgens", so pcells and pycells) save all the code, parameters and stuff that is needed to generate that. If you loose any referenced functions (or more likely: your colleague doesn't have it) or you change anything afterwards, it changes your design. Of course you can stream out the layout to GDS (which you have to do anyways for the foundry) but this removes the parameterized part, also removing the "advantage" of pcells.

Besides problems with persistence and sharing of designs, pcells also are inherently technology-bound. Every PDK comes with its own set of pcells, which can be annyoing.

This project aims to solve all these problems. We want to built technology-independent cell generators that can be easily accessed from any tool (open or closed source), but are tool-independent. Therefor the entire project is implemented in an easy-to-learn-install-and-embed language (lua) without any dependencies at all. A framework for writing tool interfaces is provided as well as templates for technology files. The project comes with a set of predefined cells which are actively maintained and developed.

3 Installation

The OpenPCells project aims to have zero dependencies (besides a C compiler) by writing all modules from scratch and including a lua interpreter. Therefore a simple `make` should be enough to build the project. The main Makefile is partly based on the Makefile from the lua project and

tries to figure out the platform it is running on. There is no reason why this project should not also run on systems where lua runs on, but it is only tested for linux.

3.1 Virtuoso

The virtuoso interface consists of two parts: an export module to create .il files to be loaded in virtuoso as well as a menu that helps creating a menu and callling the program with the right options. You can install this by loading `interface/virtuoso/init.il` BEFORE you open a layout view (this installs a trigger). The best option is to put the following line in your `.cdsinit`:

```
load("/path/to/opc/interface/virtuoso.il")
```

Of course you need to adapt the path.

Now open a layout view, there should be a new menu at the end of the available menus (right) called OpenPCells. If not, check your log window and write me an email.