OpenPCells

PCell Design Guide and API

Patrick Kurth

November 3, 2020

This is the official documentation of the OpenPCells project. It is split in several different files for clarity. This document provides an overview of the creation of PCells in the OpenPCells environment as well as a detailed API documentation. If you are looking for a general overview of the project and how to use it, start with the user guide, which also contains a tutorial for getting started quickly. If you want to know more about the technical details and implementation notes, look into the technical documentation.

Contents

1 PCell Creation – Introductory Examples

We will start this documentation by a series of examples to show the main features and API functions. The to-be-created cells will get increasingly complex to demonstrate various features.

Every cell is defined by a function where all shapes making up the shape are described. This function gets called by the cell generation system, which passes the main object and a table with all defined parameters.

1.1 Simple Rectangle

The first example is a simple rectangle of variable width and height. As mentioned, all the code for the rectangle resides in a function `layout()`. The parameters of the cell are defined in a function `parameters()`, which is optional in theory, but since we're designing pcells, there is not much point of leaving it out. In `layout()`, we receive the main object and the defined parameters. Here we can modify the object based on the parameters.

The simple rectangle looks like this:

```
-- define parameters
function parameters()
    pcell.add_parameters(
        { "width",  100 },
        { "height", 100 }
    )
end

-- define layout
function layout(obj, _P)
    -- create the shape
    local rect = geometry.rectangle(generics.metal(1), width, height)
    -- merge into main cell
    obj:merge_into(rect)
end
```

Let's walk through this line-by-line (sort of). First, we declare the function for the parameter definition:

```
function parameters()
```

In the function, we add the parameters, here we use the width and the height of the rectangle:

```
pcell.add_parameters(
    { "width",  100 },
    { "height", 100 }
)
```

We can add as many parameters as we like (`pcell.add_parameters()` accepts any number of arguments). For every argument, the first entry in the table is the name of the parameter, the second entry is the default value. This is the simplest form, we can supply more information for finer control. We will see some examples of this later on.

The default value for both parameters is 100, which is a *size*, meaning it has a unit. Physical/geometrical parameters like width or height are specified in nanometers.[1]

This is all for the `parameters()` function, so let's move on to `layout()`. This functions takes two arguments: the main object that will be placed in the layout and the table with parameters for the cell (which already includes any parsed arguments given before the cell creation).

We can name them in any way that pleases us, the common name used in all standard cells distributed by this project is `_P` (as hommage to the global environment `_G` in lua). Of course it is possible to "unpack" the parameters, storing them in individual variables, but for cells with many parameters this rather is a bloat.

```
function layout(obj, _P)
```

Now that we have all the layout parameters, we can already create the rectangle:

```
local rect = geometry.rectangle(generics.metal(1), width, height)
```

There is a lot going on here: We use the `geometry.rectangle` function to create a rectangle with with and height (second and third argument). Since we are creating shapes of IC geometry, we have to specify a layer. But we also want to create technology-independent pcells, so there is a generics system for layers. Right now we are just using the `generics.metal` function, which takes a single number as argument. `generics.metal(1)` specifies the first metal (counted from silicon), you can also say something like `generics.metal(-2)`, where `-1` is the index of the highest metal. Lastly we save the return value of `geometry.rectangle` in a local variable `rect`, which is a hint to the type: All geometry functions return `objects`, which has some consequences for the use of these functions. We will get into that later.

That is all we have to do for the geometry of the cell, so we merge that into the main cell:

```
obj:merge_into(rect)
```

This cell can now be created by calling the main program with an appropriate interface and technology. Note that there's another manual about that, so we won't get into any details here.

1.2 Array of Rectangles

Now that we know how to create a rectangle, we want to create an entire array, that is a rectangular area made up of several individual rectangles. This could be used for example as filling. We will setup the cell exactly as before, we only have to add two new parameters:

[1] Of course you can do what you want in a cell, but the modules that process the cells after creation work in nanometers. It is best practice to do the same.

the repetition and the pitch (we will start with quadratic arrays with equal pitch in both directions):

```
function parameters()
    pcell.add_parameters(
        { "width",  100 },
        { "height", 100 },
        { "pitch",  200 },
        { "rep",     10 }
    )
end
```

The default arguments are 200 for the pitch and 10 for the number of repetitions, which creates a ten-by-ten array of rectangles with a spacing of 100 and a width and height of 100. Again, remember that we work in nanometers here.

For the repetition we could use a loop to create the objects:

```
for i = 1, P.rep do
    for j = 1, P.rep do
        local o = geometry.rectangle(
            generics.metal(1), P.width, P.height
        )
        obj:merge_into(o)
    end
end
```

Looks ok, but what's with the `obj:merge_into(o)`? This is a *method* of objects and needs a little explaining of the object system. As mentioned earlier, every geometry function creates what is called an object. An object is a collection of shapes, where each shape is made up of a layer-purpose-pair and points (which can currently be interpreted as rectangle or polygon). The cell generation systems expects to receive only one object from the main cell function, so how do we return more than one shape? We can merge several objects into one, which is exactly what we are doing here.

In order for this to work, we also have to move the rectangles to the correct position, something that we didn't learn yet. This comes later, as this also involves some math we don't want to talk about right now. Just keep in mind that the above loop is wrong and cumbersome. In any ways, there is a function that does exactly what we want to achieve: `geometry.multiple`. It takes an object as first argument and then the repetition in x and y and the pitch in x and y and returns an array of repeated objects with the center in the origin. With it, we can replace the whole loop construct with:

```
obj:merge_into(geometry.multiple(
    geometry.rectangle(generics.metal(1), P.width, P.height),
    P.rep, P.rep, P.pitch, P.pitch
))
```

`geometry.multiple` also already merges all objects so we don't have to take care of that. Therefor, we receive a single object which we simply can merge directly into the main cell. The whole cell looks like this:

```
function parameters()
    pcell.add_parameters(
        { "width",  100 },
        { "height", 100 },
        { "pitch",  200 },
        { "rep",     10 }
    )
end

function layout(obj, _P)
    -- first naive (and wrong) attempt (don't use!)
    for i = 1, P.rep do
        for j = 1, P.rep do
            local o = geometry.rectangle(
                generics.metal(1), P.width, P.height
            )
            obj:merge_into(o)
        end
    end

    -- better approach
    obj:merge_into(geometry.multiple(
        geometry.rectangle(generics.metal(1), P.width, P.height),
        P.rep, P.rep, P.pitch, P.pitch
    ))
end
```

Now you already now how to create rectangles, with generic layers, `geometry.multiple` and object merging. With this, one can already built a surprising amount of pcells. However, we have to discuss how we can create layers other than metals, vias and shapes with more complex outlines than rectangles. We will talk about that in the remaining cell tutorials.

1.3 Metal-Oxide-Metal Capacitors

Many technologies don't have so-called metal-insulator-metal capacitors (mimcaps), so the standard way to implement capacitors is be using interdigitated metals. Let's do that. As before, we set up the pcell. Useful parameters are the number of fingers, the width and height of the fingers and the spacing in between. Furthermore, we shift one collection of fingers (one plate) up and the other down to separate them and connect them together. Lastly, we also specify the used metals and the width of the connecting rails:

```
function parameters()
    pcell.add_parameters(
        { "fingers(Number of Fingers)", 4 },
        { "fwidth(Finger Width)",      100 },
        { "fspace(Finger Spacing)",    100 },
        { "fheight(Finger Height)",   1000 },
        { "foffset(Finger Offset)",    100 },
```

```
        { "rwidth(Rail Width)",        100 },
        { "firstmetal(Start Metal)",     1 },
        { "lastmetal(End Metal)",        2 }
    )
end
```

The parameter definition also shows how you can use better names for displaying: Simply write them in parantheses. When querying the defined parameters of a cell, the display names are used, but within the cell the regular names are significant. This enables easier syntax: `_P.fingers` as opposed to `_P["Number of Fingers"]`.

In `layout()` we loop over all metals to draw the fingers. We don't have to create every finger separately, with `geometry.multiple` this becomes very simple. Since the upper and lower fingers are one-off and `geometry.multiple` centeres all objects, we only have to move them a little bit up/down. This is done with `object.translate` (a method of an object), taking x- and y-offset as arguments:

```
local pitch = _P.fwidth + _P.fspace

for i = _P.firstmetal, _P.lastmetal do
    momcap:merge_into(geometry.multiple(
        geometry.rectangle(generics.metal(i), _P.fwidth, _P.fheight),
        _P.fingers + 1, 1, 2 * pitch, 0
    ):translate(0, _P.foffset))
    momcap:merge_into(geometry.multiple(
        geometry.rectangle(generics.metal(i), _P.fwidth, _P.fheight),
        _P.fingers, 1, 2 * pitch, 0
    ):translate(0, -_P.foffset))
```

We create two arrays of fingers, one for the "upper plate", one for the "lower plate". All fingers have the same width, height and pitch. For the upper plate, we use one more finger, the placement in `geometry.multiple` automatically arranges them centered, so that this "just works". The `ypitch` for `geometry.multiple` is 0, which is ok since we only have a `yrep` of 1.

The rails connecting the fingers are created in a similar manner:

```
    momcap:merge_into(geometry.multiple(
        geometry.rectangle(generics.metal(i),
            (2 * _P.fingers + 1) * (_P.fwidth + _P.fspace), _P.rwidth
        ),
        1, 2, 0, 2 * _P.foffset + _P.fheight + _P.rwidth
    ))
end
```

The `end` delimits the `for`-loop.

What remains is the drawing of the vias between the metals. For this we introduce a new `generics` function: `generics.via`. It takes two arguments for the start- and end-metal for the via stack. We don't have to specify the individual vias between each layer in the stack, this is resolved later by the technology translation. The vias are placed in the rails:

```
momcap:merge_into(geometry.multiple(
    geometry.rectangle(generics.via(_P.firstmetal, _P.lastmetal),
        (2 * _P.fingers + 1) * (_P.fwidth + _P.fspace), _P.rwidth
    ),
    1, 2, 0, 2 * _P.foffset + _P.fheight + _P.rwidth
))
```

With this the pcell is finished, the entire listing is in `cells/passive/capacitor/mom.lua`.

1.4 Octagonal Inductor

RF designs often require on-chip inductors, which usually are built in an octagonal shape due to angle restrictions in most technologies (no true circles or better approximations available). We will show how to built a differential (symmetric) octagonal inductor with a variable number of turns (integers). We will skip some basic techniques that we already discussed a few times such as setting up the cell body, cell parameters and main object. Look into `cells/passive/inductor/octagonal.lua` for the defined parameters.

An inductor is basically a wire routed in a special manner, therefor we will describe the inductor as a `path`. This is a series of points that gets converted into a polygon with a `width`. To create a path, we have to pass the points, which we will store in a `table`. Here is how this looks for the octagonal inductor:

```
local pathpts = {}
local prepend = util.make_insert_xy(pathpts, 1)
local append = util.make_insert_xy(pathpts)

append(-r + _scale_tanpi8(_P.width / 2),  sign * radius)
append(-r,  sign * radius)
append(-radius,  sign * r)
append(-radius, -sign * r)
append(-r, -sign * radius)
append(-r + _scale_tanpi8(_P.width / 2), -sign * radius)
```

`util.make_insert_xy` is a helper function, that returns a function that appends/prepends points to an array. It's purpose is to simplify code, one might as well just use `table.insert`.

This is just an excerpt from the cell, the entire code generating the path points is a bit complex and involves some mathematical thoughts. Since this tutorial is about how to build the code for cells, the actual points will not be discussed.

After the points are assembled, we can create the path. The cell only draws half of the inductor, so we draw the path twice, one time with mirrored points (notice `util.xmirror(pathpts)` in the second line):

```
inductor:merge_into(
    geometry.path(mainmetal, pathpts, _P.width, true)
)
inductor:merge_into(
```

```
    geometry.path(mainmetal, util.xmirror(pathpts), _P.width, true)
)
```

The `geometry.path` function takes four arguments: the layer, the points of the path, the width and whether to use a miter- or a bevel-join. Bevel-join is default, so `true` is specified for a miter-join.

2 Available PCells

In the following subsections, all available cells will be documented. The current status is rather a poor one, but work is ongoing.

2.1 Transistor

The transistor might be the most important cell and currently it's also definitely the most complex one. Therefor, this documentation starts with a description of the goal. Figure 1 shows an example with all geometrical parameters, a summary of all parameters can be found in table 1. The cell draws a number of gates on top of an active area (with some implant/well/etc. markers). Furthermore, it draws some metals and vias (not shown in figure 1) in the source/drain regions

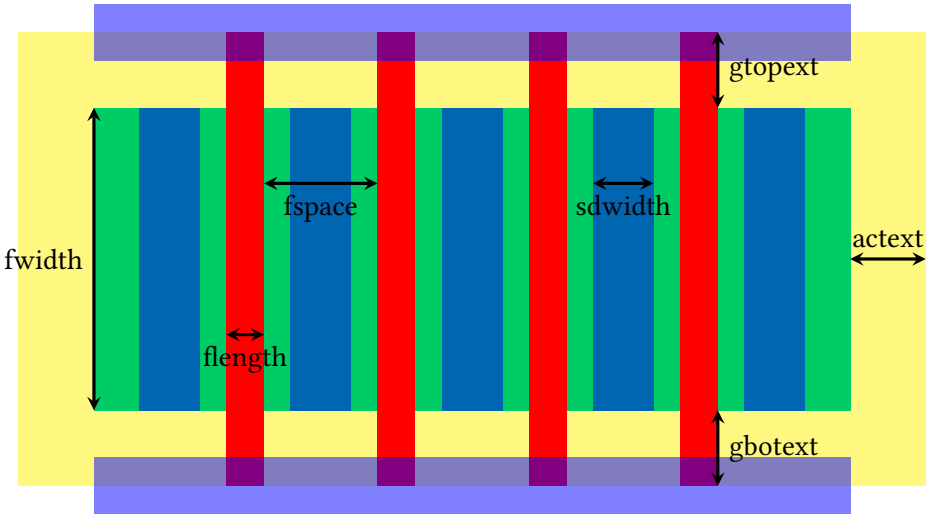

Figure 1: Overview of the transistor

and for gate contacts.

Parameter	Meaning	Default
channeltype	Type of Transistor	”nmos”
oxidetype	Oxide Thickness Index	1
vthtype	Threshold Voltage Index	1
fingers	Number of Fingers	4
fwidth	Finger Width	1.0
gatelength	Finger Length	0.15
fspace	Space between Fingers	0.27
actext	Left/Right Extension of Active Area	0.03
sdwidth	Width of Source/Drain Metals	0.2
sdconnwidth	Width of Source/Drain Connection Rails Metal	0.2
sdconnspace	Space of Source/Drain Connection Rails Metal	0.2
gtopext	Gate Top Extension	0.2
gbotext	Gate Bottom Extension	0.2
typext	Implant/Well Extension around Active	0.1
cliptop	Clip Top Marking Layers (Implant, Well, etc.)	false
clipbot	Clip Bottom Marking Layers (Implant, Well, etc.)	false
drawtopgate	Draw Top Gate Strap	false
drawbotgate	Draw Bottom Gate Strap	false
topgatestrwidth		0.12
topgatestrext		1
botgatestrwidth		0.12
botgatestrext		1
topgcut	Draw Top Poly Cut	false
botgcut	Draw Bottom Poly Cut	false
connectsource	Connect all Sources together	false
connectdrain	Connect all Drains together	false

Table 1: Summary of Transistor Parameters

3 API Documentation

3.1 geometry Module

`geometry.rectangle(layer, width, height)`

Create a rectangular shape with a width of `width` and a height of `height` in the layer-purpose-pair `layer` (usually a generic). The function returns an object.

`geometry.multiple(obj, xrep, yrep, xpitch, ypitch)`

Creates a rectangular array (mosaic) of an object with `xrep` repetitions in x and `yrep` repetitions in y. `xpitch` and `ypitch` are the center-to-center space in x and y direction. The

entire array gets centered. The function returns the merged objects.

3.2 Object Module

3.3 Shape Module

3.4 Pointarray Module

3.5 Point Module