OpenPCells

PCell Design Guide and API

Patrick Kurth

August 10, 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 now 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 fea-

tures.

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 expects to receive one object as return value of that function.

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 that we have to return (the whole cell is a lua module. If you are not familiar with this, just keep in mind that you have to start your file with 'return function()'). Then we will define some parameters of the cell and afterwards create the rectangle. Finally the created cell is returned (now from within the function, so it is the return value of the function. Don't confuse it with returning the function from the module).

The simple rectangle looks like this:

```
-- put the cell inside a function
return function(args)
    -- process cell arguments (parameters)
    pcell.setup(args)
    local width  = pcell.process_args("width",  1.0)
    local height = pcell.process_args("height", 1.0)
    pcell.check_args()

    -- create the shape
    local obj = layout.rectangle(generics.metal(1), width, height)

    -- return the object
    return obj
end
```

Let's walk through this line-by-line (sort of). First, the function for the cell-generation is started. We also pass some arguments to the cell, which allows us to use parametrization.

```
return function(args)
```

In general, every cell will include this line at some point. Next we start the argument parsing:

```
    pcell.setup(args)
```

This will store the argument table, so that we don't have to refer to it for every process call. Furthermore it resets internal pcell states and is needed if more than one cell is used.

Now that we have set up the arguments, we can actually parse them:

```
    local width  = pcell.process_args("width",  1.0)
    local height = pcell.process_args("height", 1.0)
```

Arguments are passed in key-value-pairs, we have to provide the key and the default value. The processing will parse and evaluate it's arguments depending on the type, which we can state explicitly (as third argument to `pcell.process_args`), but usually the type can be inferred automatically (this is not the case if you only allow integers for numbers, for example). The default value for both parameters is 1.0, which is a *size*, meaning it has a unit. What does 1.0 mean? It could be (and mostly will be) 1 µm, but we don't know that yet. It depends on the technology settings, but we don't care about that here.

After we have parsed all arguments, we can check if everything went right. This will catch any wrongly spelled arguments:

```
    pcell.check_args()
```

This is optional, but good practice.

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

```
    local obj = layout.rectangle(generics.metal(1), width, height)
```

There is a lot going on here: We use the `layout.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 cell-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 `layout.rectangle` in a local variable `obj`, which is a hint to the type: All layout 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 can return what we have created:

```
    return obj
```

And to write correct lua we finally close the function body:

```
end
```

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

```
    pcell.setup(args)
    local width  = pcell.process_args("width",  1.0)
    local height = pcell.process_args("height", 1.0)
    local pitch  = pcell.process_args("pitch",  2.0)
    local rep    = pcell.process_args("rep",    10)
    pcell.check_args()
```

The default arguments are 2.0 for the pitch and 10 for the number of repetitions, which creates a ten-by-ten array of rectangles with a spacing of 1.0 and a width and height of 1.0. Again, remember that we don't care about real dimensions and units here.

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

```
    for i = 1, rep do
        for j = 1, rep do
            local o = layout.rectangle(
                generics.metal(1), width, 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 layout 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 an array of points (which are currently mostly interpreted as 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. We silently dropped the line about the creation of the main cell object `obj`, which simply looks like this (and of course comes before the loops):

```
    local obj = object.create()
```

This whole way of creating is a bit cumbersome, so there is a function that does exactly that: `layout.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:

```
    local obj = layout.multiple(
        layout.rectangle(generics.metal(1), width, height),
        rep, rep, pitch, pitch
    )
```

`layout.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 return as before. The whole cell looks like this:

```
-- put the cell inside a function
return function(args)
    -- process cell arguments (parameters)
    pcell.setup(args)
    local width  = pcell.process_args("width",  1.0)
    local height = pcell.process_args("height", 1.0)
```

```
    local pitch  = pcell.process_args("pitch",  2.0)
    local rep    = pcell.process_args("rep",    10)
    pcell.check_args()

    -- create the main object
    local obj = object.create()

    -- first naive attempt
    for i = 1, rep do
        for j = 1, rep do
            local o = layout.rectangle(
                generics.metal(1), width, height
            )
            obj:merge_into(o)
        end
    end

    -- better approach
    local obj = layout.multiple(
        layout.rectangle(generics.metal(1), width, height),
        rep, rep, pitch, pitch
    )

    -- return the object
    return obj
end
```

Now you already now how to create rectangles, the existence of generic layers, `layout.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:

```
return function(args)
    pcell.setup(args)
    local fingers    = pcell.process_args("fingers",    4)
    local fwidth     = pcell.process_args("fwidth",     0.1)
    local fspace     = pcell.process_args("fspace",     0.1)
    local fheight    = pcell.process_args("fheight",    1)
    local foffset    = pcell.process_args("foffset",    0.1)
```

```
    local rwidth     = pcell.process_args("rwidth",     0.1)
    local firstmetal = pcell.process_args("firstmetal", 1)
    local lastmetal  = pcell.process_args("lastmetal",  2)
    pcell.check_args()
```

As before, we create the main object:

```
    local momcap = object.create()
```

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

```
    for i = firstmetal, lastmetal do
        momcap:merge_into(layout.multiple(
            layout.rectangle(generics.metal(i), fwidth, fheight),
            fingers + 1, 1, 2 * pitch, 0
        ):translate(0, foffset))
        momcap:merge_into(layout.multiple(
            layout.rectangle(generics.metal(i), fwidth, fheight),
            fingers, 1, 2 * pitch, 0
        ):translate(0, -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 `layout.multiple` automatically arranges them centered, so that this "just works".

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

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

The `xpitch` for `layout.multiple` is 0, which is ok since we only have a `xrep` of 1. 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.

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

With this the pcell is finished, we only have to remember to return the created object and close the function:

```
    return momcap
end
```

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, main object and return value.

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 `pointarray`. Internally, most of the layout functions and shapes work with pointarrays, but for simpler cells we don't need to deal with them. Here is how this looks for the octagonal inductor:

```
        local pathpts = pointarray.create()

        pathpts:append(point.create(-r + 0.5 * tanpi8 * width,  sign * radius))
        pathpts:append(point.create(-r,  sign * radius))
        pathpts:append(point.create(-radius,  sign * r))
        pathpts:append(point.create(-radius, -sign * r))
        pathpts:append(point.create(-r, -sign * radius))
        pathpts:append(point.create(-r + 0.5 * tanpi8 * width, -sign * radius))
```

2 Available PCells

3 API Documentation

3.1 Layout Module

`layout.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.

`layout.multiple(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.

3.2 Object Module

3.3 Shape Module

3.4 Pointarray Module

3.5 Point Module